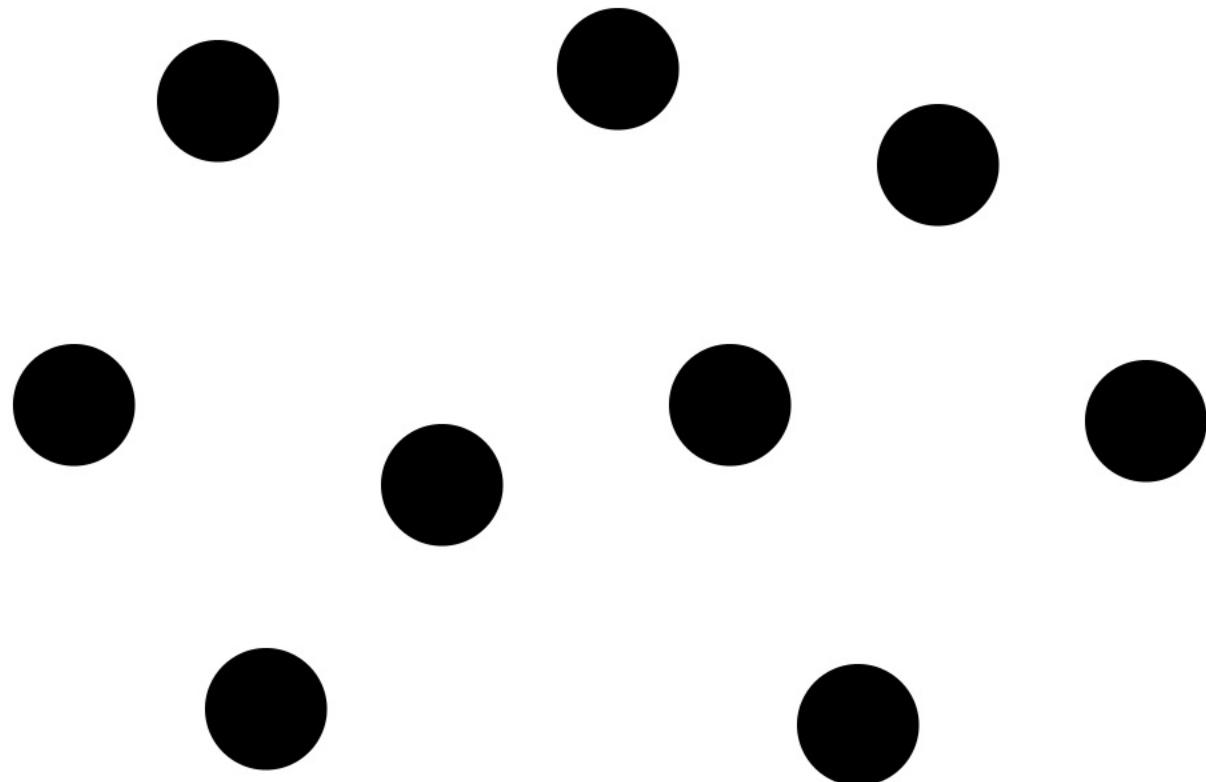
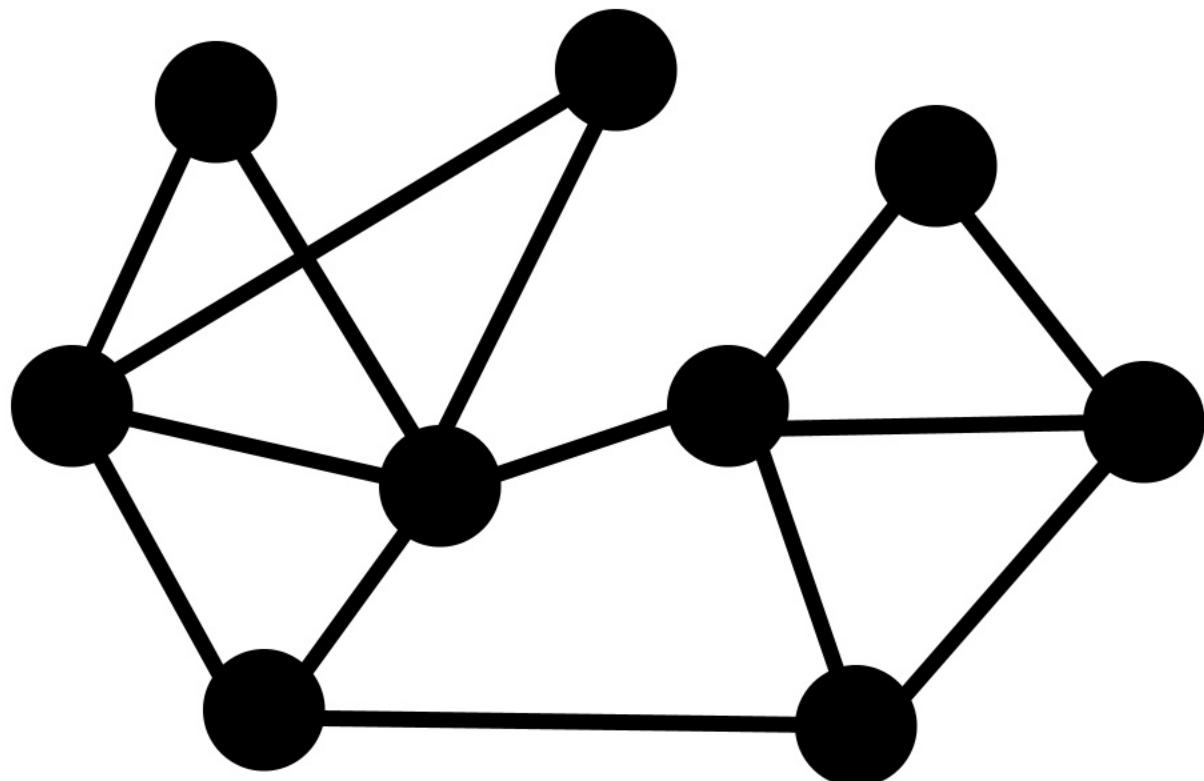


# Machine Learning with Graphs

# Why Graphs?

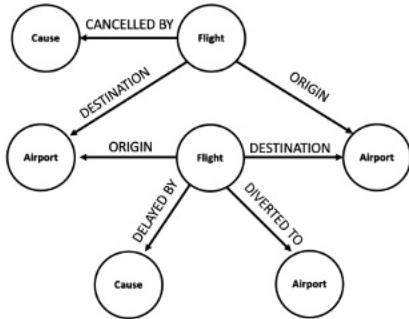
Graphs are a general language for describing and analyzing entities with relations/interactions





# Graph

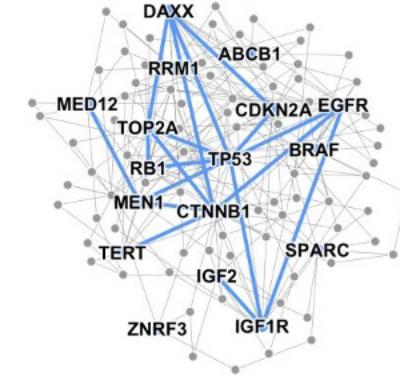
# Many Types of Data are Graphs (1)



## Event Graphs



## Computer Networks



## Disease Pathways

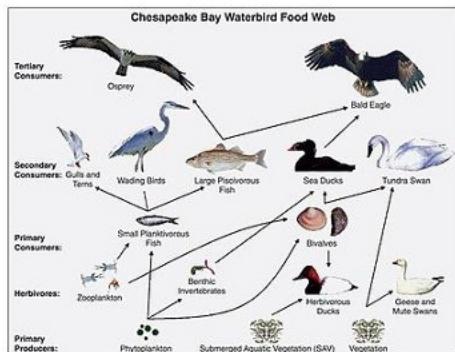


Image credit: [Wikipedia](#)

## Food Webs



Image credit: [Pinterest](#)

## Particle Networks



Image credit: [visitlondon.com](#)

## Underground Networks

# Many Types of Data are Graphs (2)



Image credit: [Medium](#)

## Social Networks

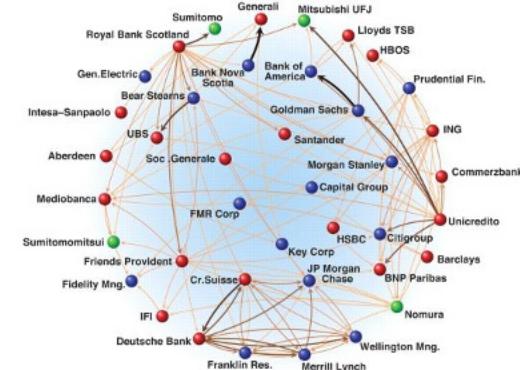


Image credit: [Science](#)



Image credit: [Lumen Learning](#)

## Economic Networks



Image credit: [Missoula Current News](#)

## Citation Networks

## Internet

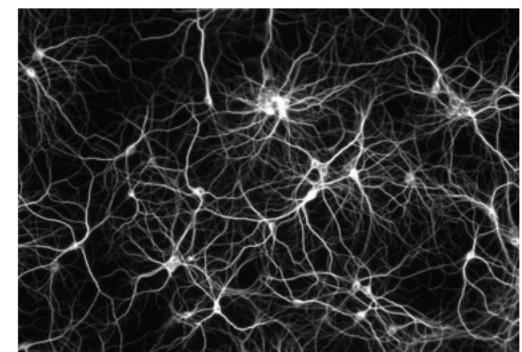


Image credit: [The Conversation](#)

## Networks of Neurons

# Many Types of Data are Graphs (3)

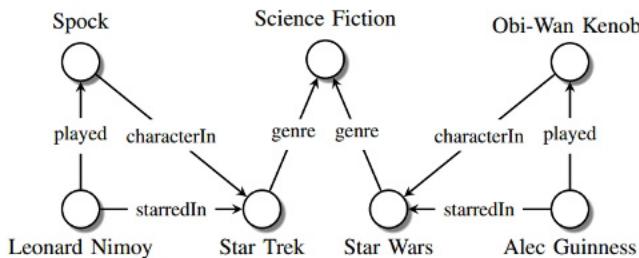


Image credit: [Maximilian Nickel et al](#)

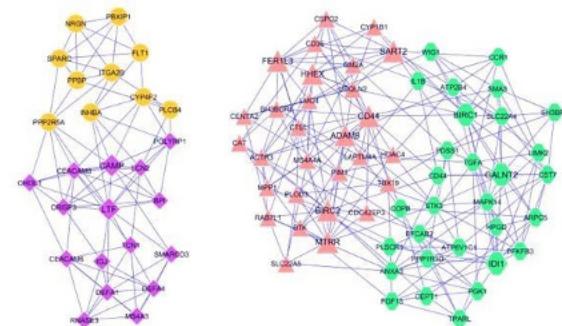


Image credit: [ese.wustl.edu](#)

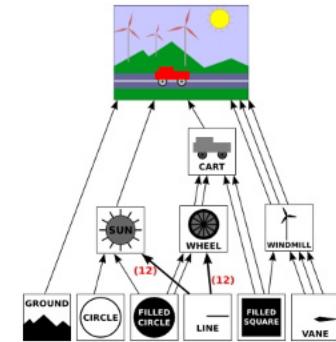


Image credit: [math.hws.edu](#)

## Knowledge Graphs

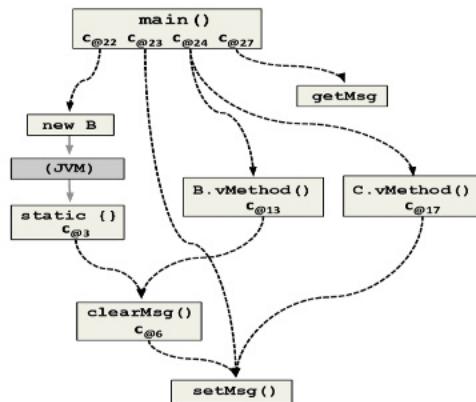


Image credit: [ResearchGate](#)

## Code Graphs

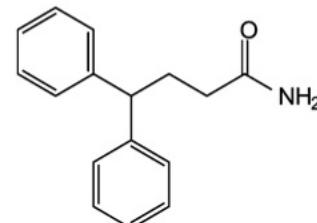


Image credit: [MDPI](#)

## Molecules

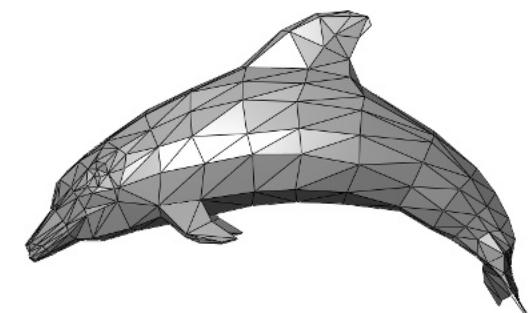
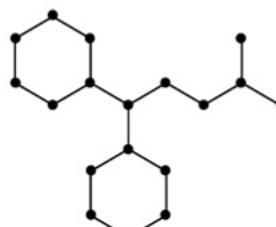


Image credit: [Wikipedia](#)

## 3D Shapes

# Graphs and Relational Data

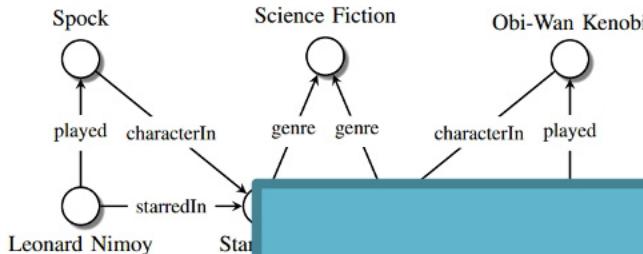


Image credit:

## Known

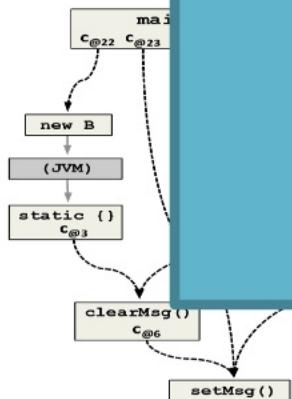


Image credit: ResearchGate

## Code Graphs

Main question:

How do we take advantage of relational structure for better prediction?

Image credit: MDPI

## Molecules

Graphs

3D Shapes

Graphs

Image credit: Wikipedia

## 3D Shapes

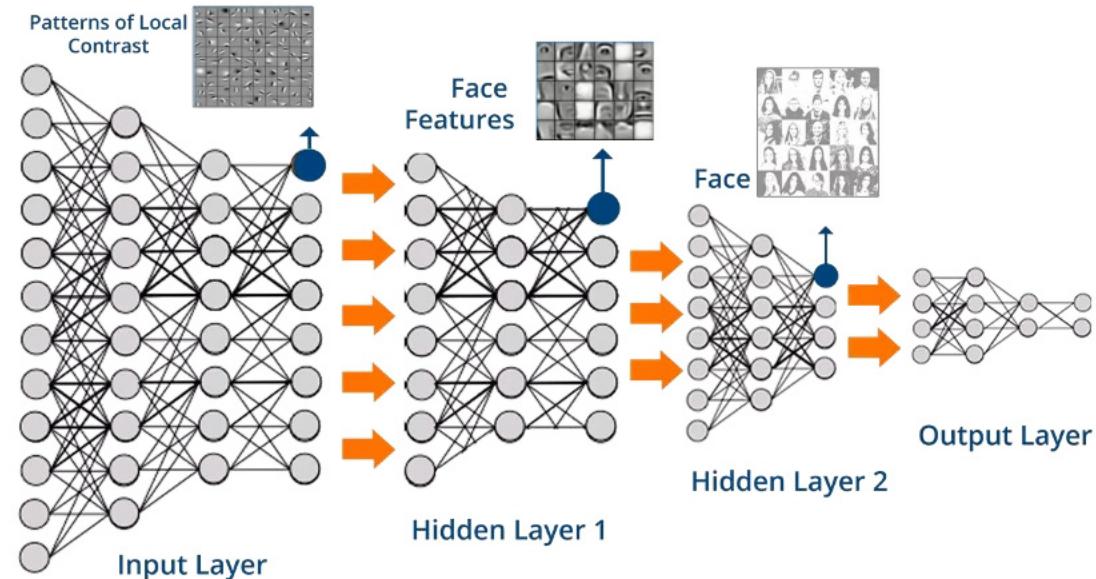
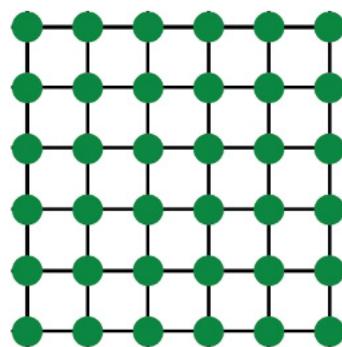


# Graphs: Machine Learning

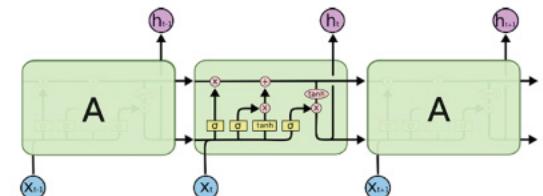
Complex domains have a rich relational structure, which can be represented as a **relational graph**

**By explicitly modeling relationships we achieve better performance!**

# Today: Modern ML Toolbox



**Text/Speech**



Modern deep learning toolbox is designed  
for simple sequences & grids

Doubt thou the stars are fire,  
Doubt that the sun doth move;  
Doubt truth to be a liar;  
But never doubt I love...

Text



Audio signals



Images

Modern  
deep learning toolbox  
is designed for  
sequences & grids

Not everything  
can be represented as  
a sequence or a grid

**How can we develop neural  
networks that are much more  
broadly applicable?**

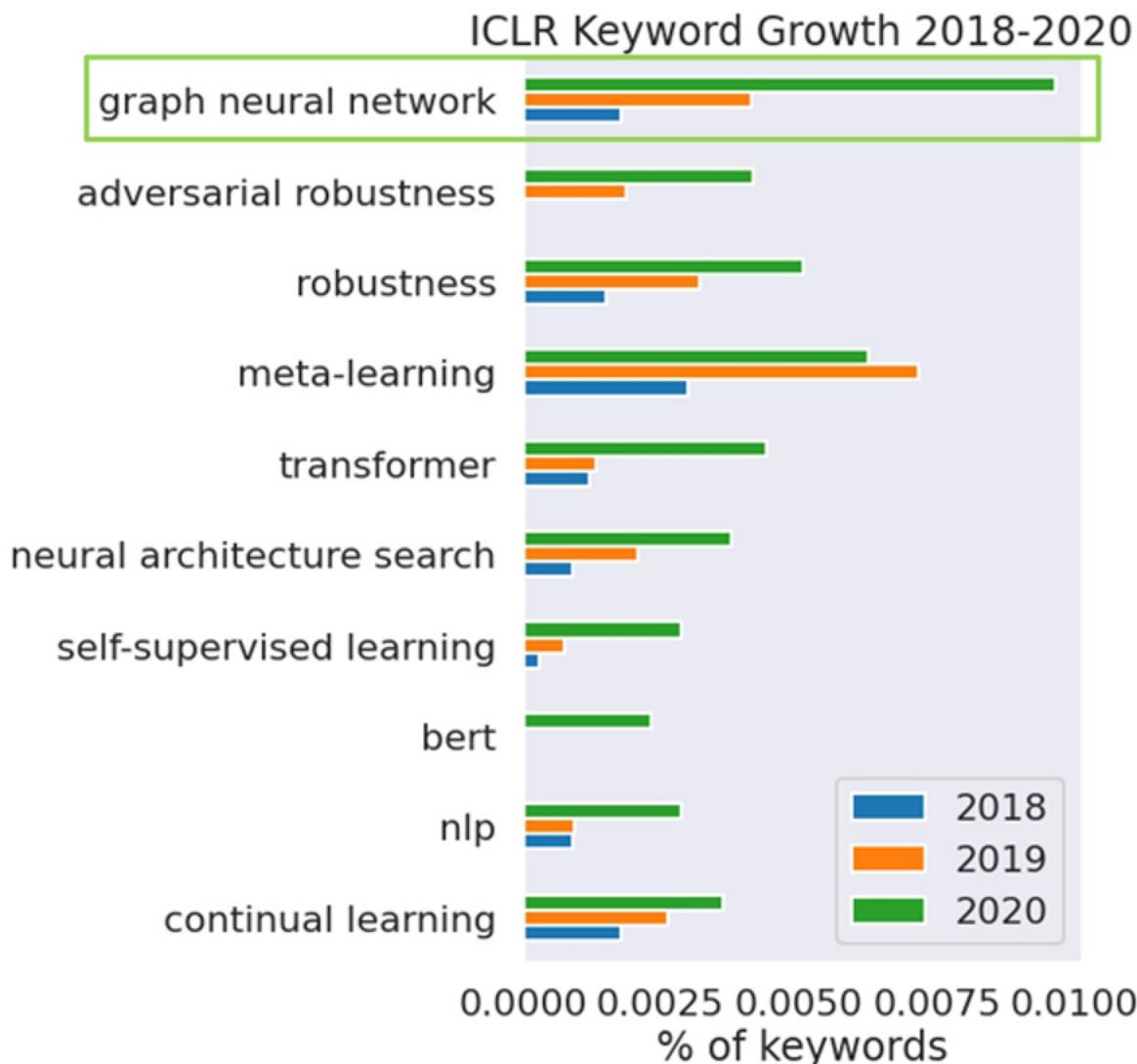
New frontiers beyond classic neural  
networks that only learn on images  
and sequences

# This Class

Graphs are the new frontier  
of deep learning

Graphs connect things.

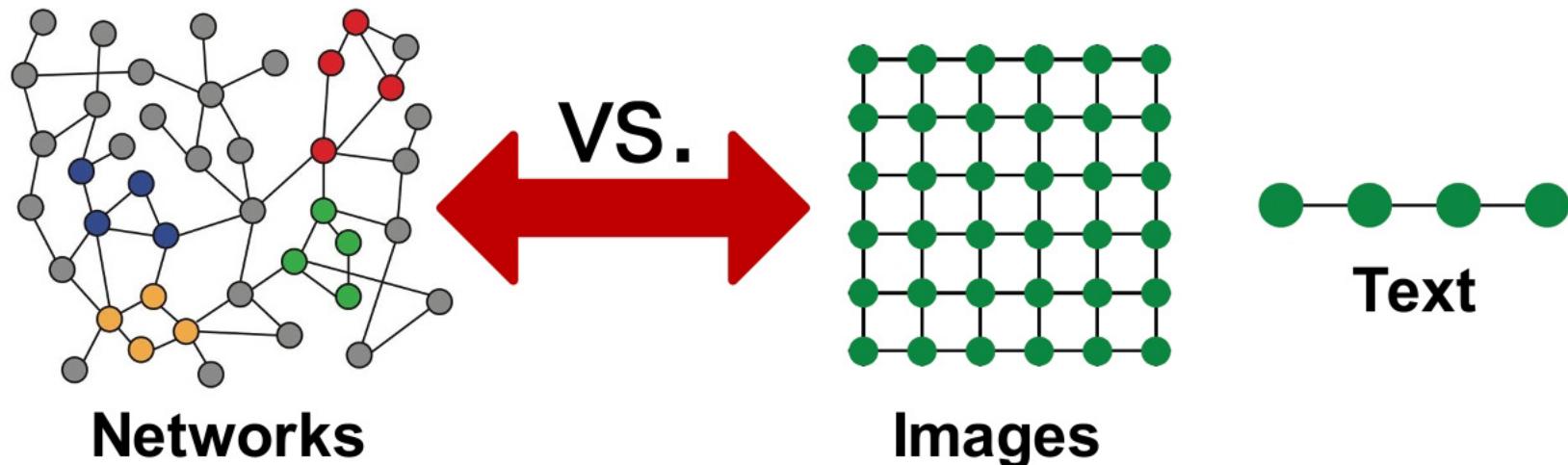
# The hottest subfield in ML



# Why is Graph Deep Learning Hard?

## Networks are complex.

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



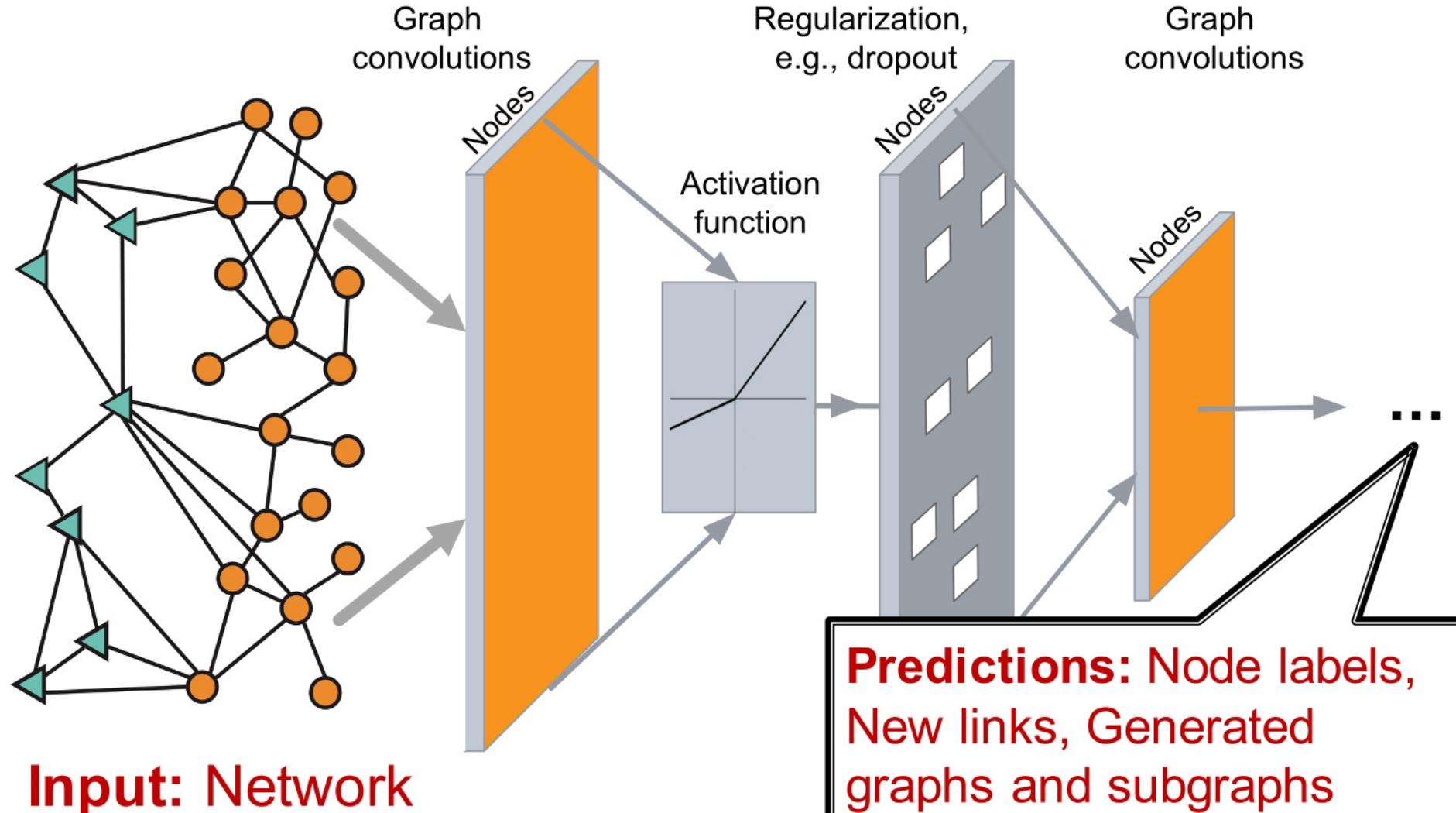
- No fixed node ordering or reference point
- Often dynamic and have multimodal features

# This Course

How can we develop neural networks  
that are much more broadly  
applicable?

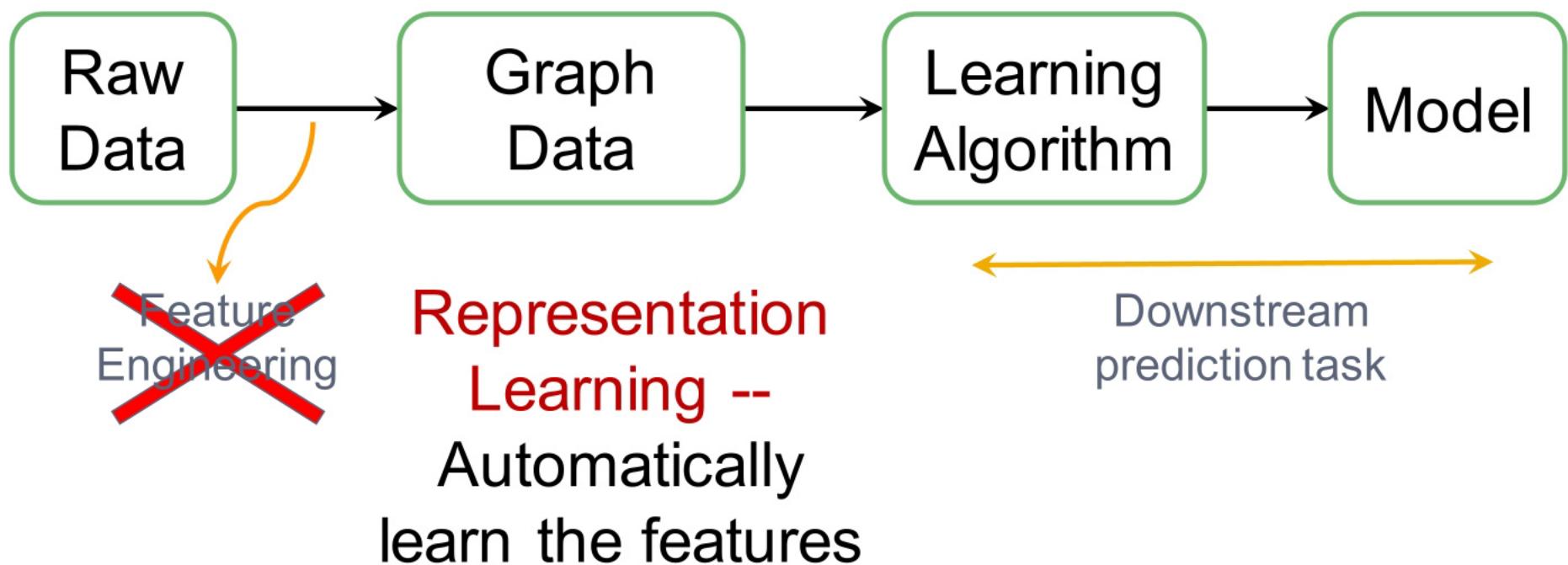
Graphs are the new frontier  
of deep learning

# CS224W: Deep Learning in Graphs



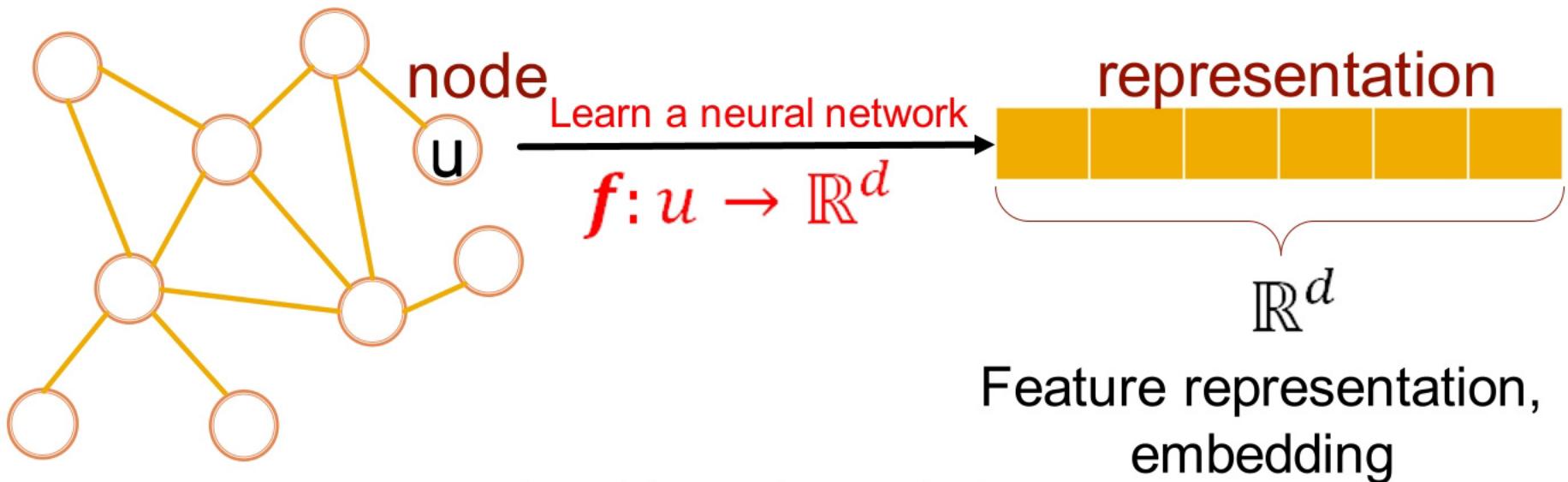
# CS224W & Representation Learning

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



# CS224W & Representation Learning

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



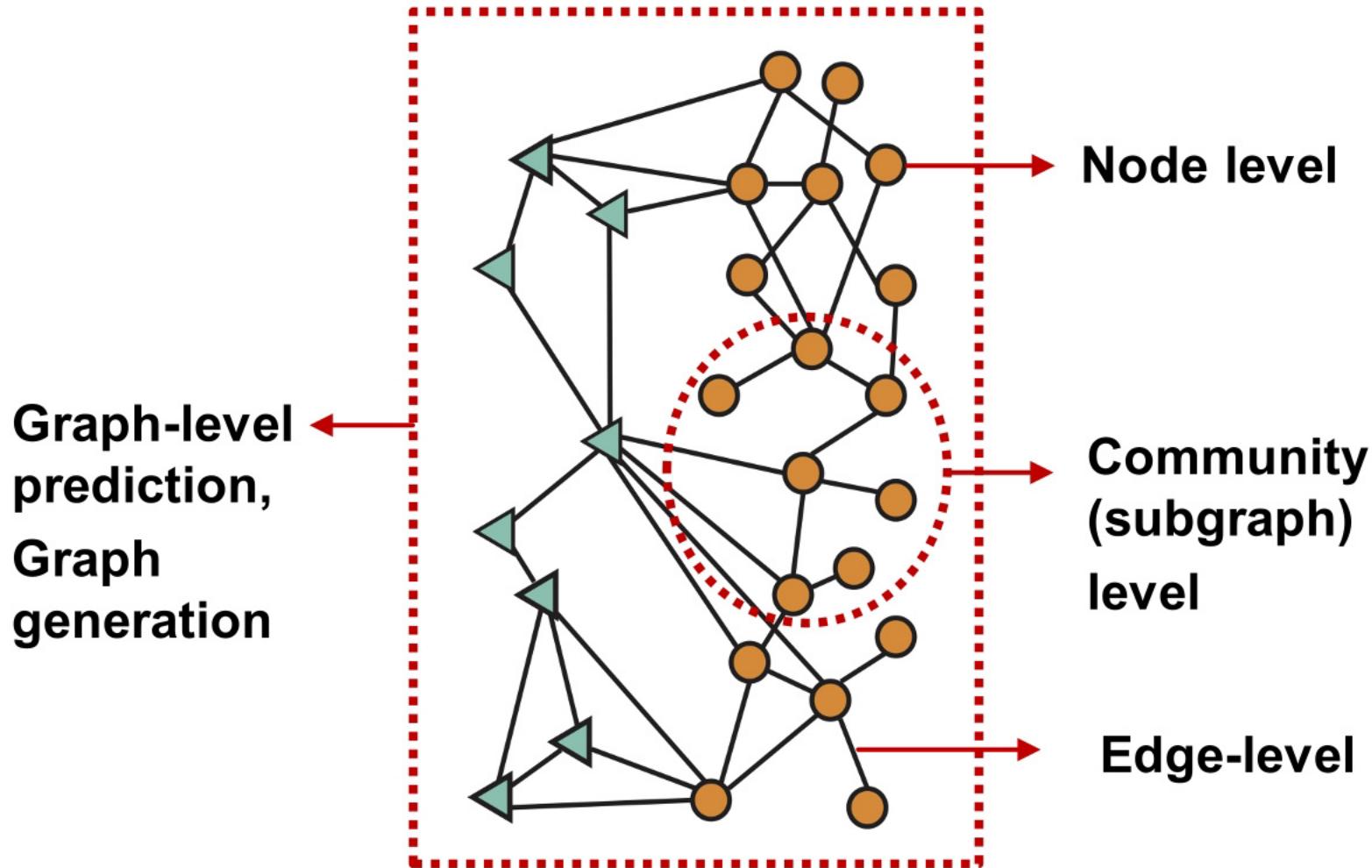
# CS224W Course Outline

We are going to cover various topics in Machine Learning and Representation Learning for graph structured data:

- Traditional methods: Graphlets, Graph Kernels
- Methods for node embeddings: DeepWalk, Node2Vec
- Graph Neural Networks: GCN, GraphSAGE, GAT, Theory of GNNs
- Knowledge graphs and reasoning: TransE, BetaE
- Deep generative models for graphs: GraphRNN
- Applications to Biomedicine, Science, Industry

# **Applications of Graph ML**

# Different Types of Tasks



# Classic Graph ML Tasks

- **Node classification**: Predict a property of a node
  - **Example**: Categorize online users / items
- **Link prediction**: Predict whether there are missing links between two nodes
  - **Example**: Knowledge graph completion
- **Graph classification**: Categorize different graphs
  - **Example**: Molecule property prediction
- **Clustering**: Detect if nodes form a community
  - **Example**: Social circle detection
- **Other tasks**:
  - **Graph generation**: Drug discovery
  - **Graph evolution**: Physical simulation

# Classic Graph ML Tasks

- **Node classification:** Predict a property of a node
  - Example: Categorize online users / items
- **Link prediction:** Predict whether there are missing links
  - Example
- **Graph classification:** Predict a property of a graph
  - Example
- **Clustering:** Group nodes into clusters
  - Example
- **Others:**
  - **Graph generation:** Drug discovery
  - **Graph evolution:** Physical simulation

These Graph ML tasks lead to high-impact applications!

# Example of Node-level ML Tasks

# Example (1): Protein Folding

A protein chain acquires its native 3D structure

Every protein is made up of a sequence of amino acids bonded together

These amino acids interact locally to form shapes like helices and sheets

These shapes fold up on larger scales to form the full three-dimensional protein structure

Proteins can interact with other proteins, performing functions such as signalling and transcribing DNA

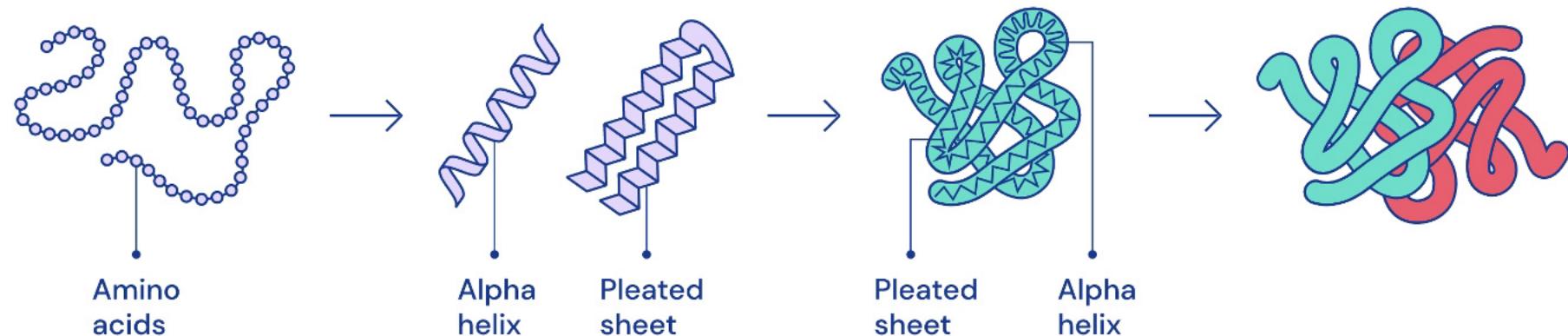
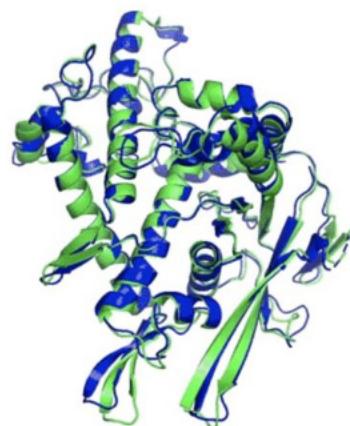


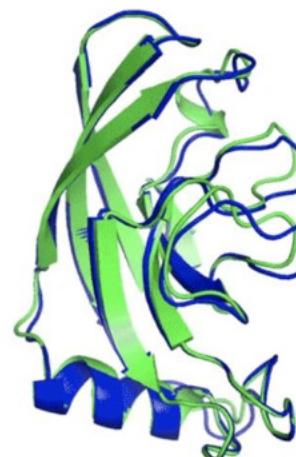
Image credit: [DeepMind](#)

# The Protein Folding Problem

Computationally predict a protein's 3D structure  
based solely on its amino acid sequence



T1037 / 6vr4  
90.7 GDT  
(RNA polymerase domain)

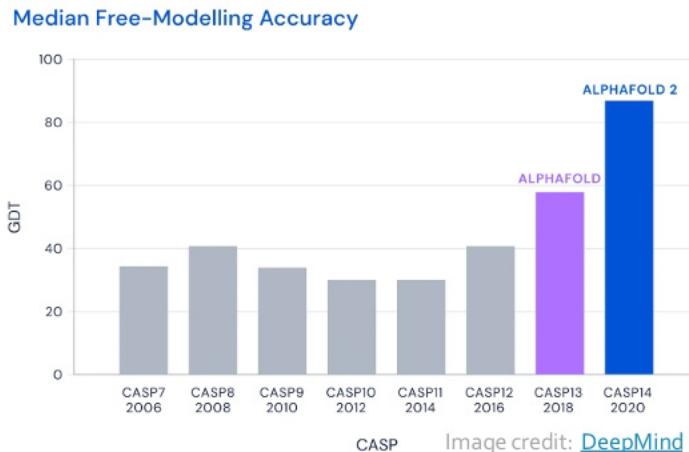


T1049 / 6y4f  
93.3 GDT  
(adhesin tip)

- Experimental result
- Computational prediction

Image credit: [DeepMind](#)

# AlphaFold: Impact



**AlphaFold's AI could change the world of biological science as we know it**

DeepMind's latest AI breakthrough can accurately predict the way proteins fold

**Has Artificial Intelligence 'Solved' Biology's Protein-Folding Problem?**

12-14-20

**DeepMind's latest AI breakthrough could turbocharge drug discovery**

# AlphaFold: Solving Protein Folding

- Key idea: “Spatial graph”
  - **Nodes:** Amino acids in a protein sequence
  - **Edges:** Proximity between amino acids (residues)

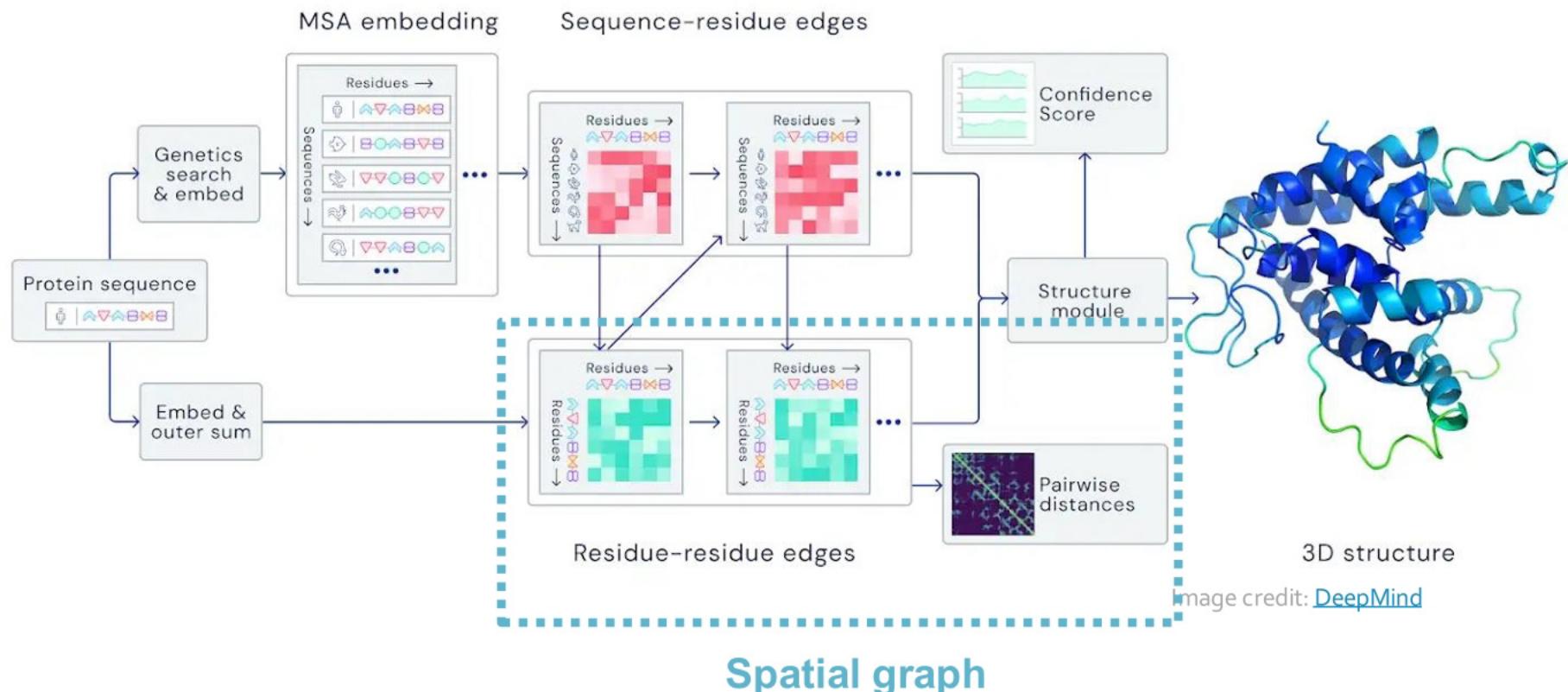


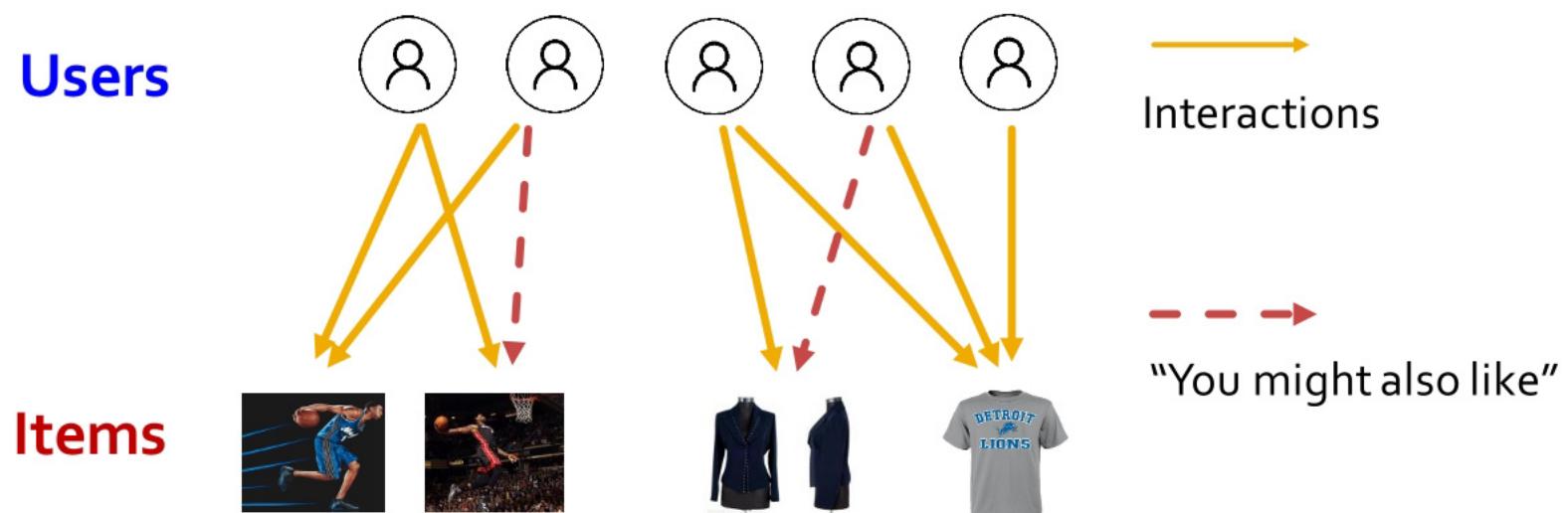
Image credit: [DeepMind](#)

Spatial graph

# Examples of Edge-level ML Tasks

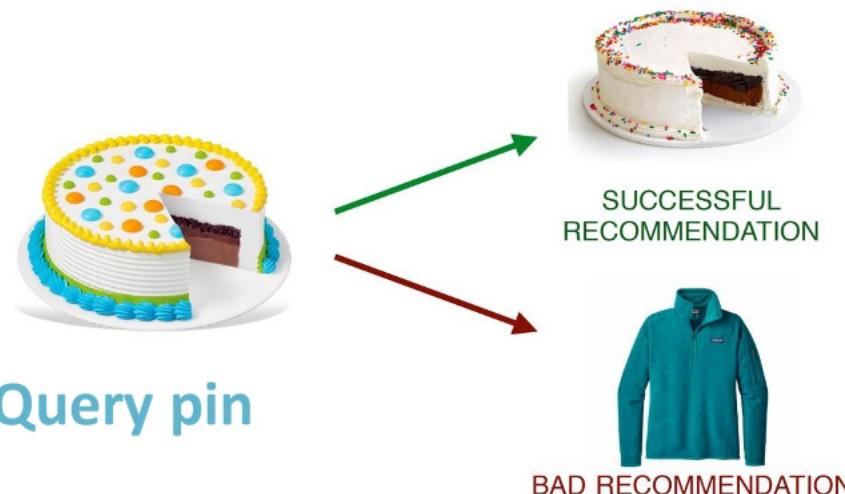
# Example (2): Recommender Systems

- **Users interacts with items**
  - Watch movies, buy merchandise, listen to music
  - **Nodes:** Users and items
  - **Edges:** User-item interactions
- **Goal: Recommend items users might like**



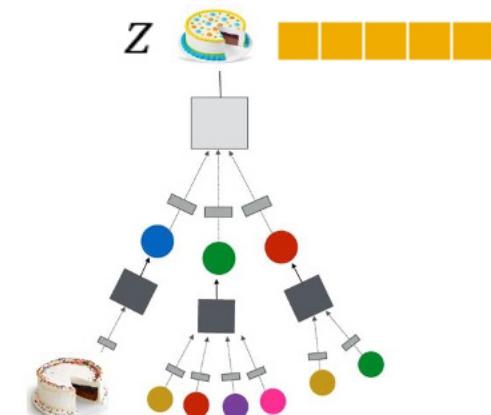
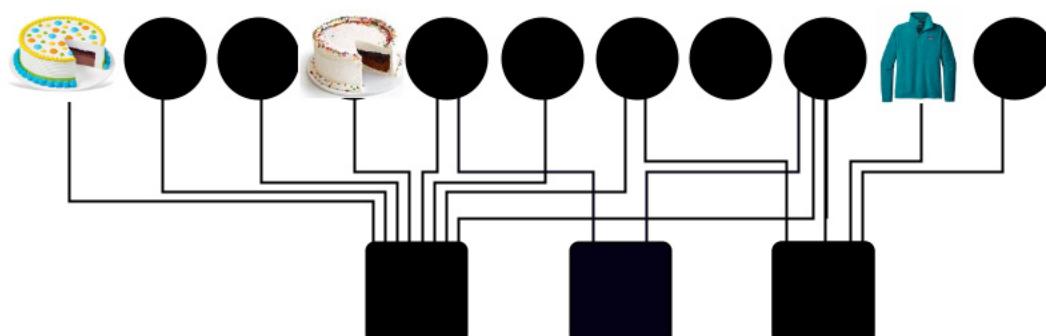
# PinSage: Graph-based Recommender

**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**

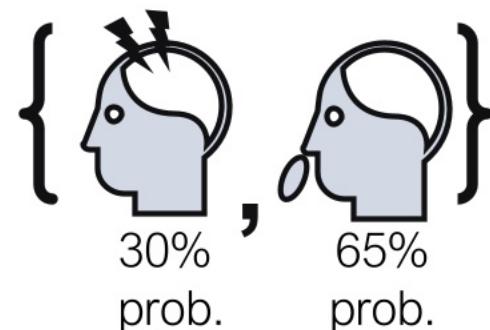
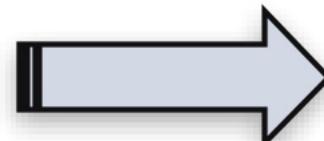


# Example (3): Drug Side Effects

Many patients **take multiple drugs** to treat  
**complex or co-existing diseases:**

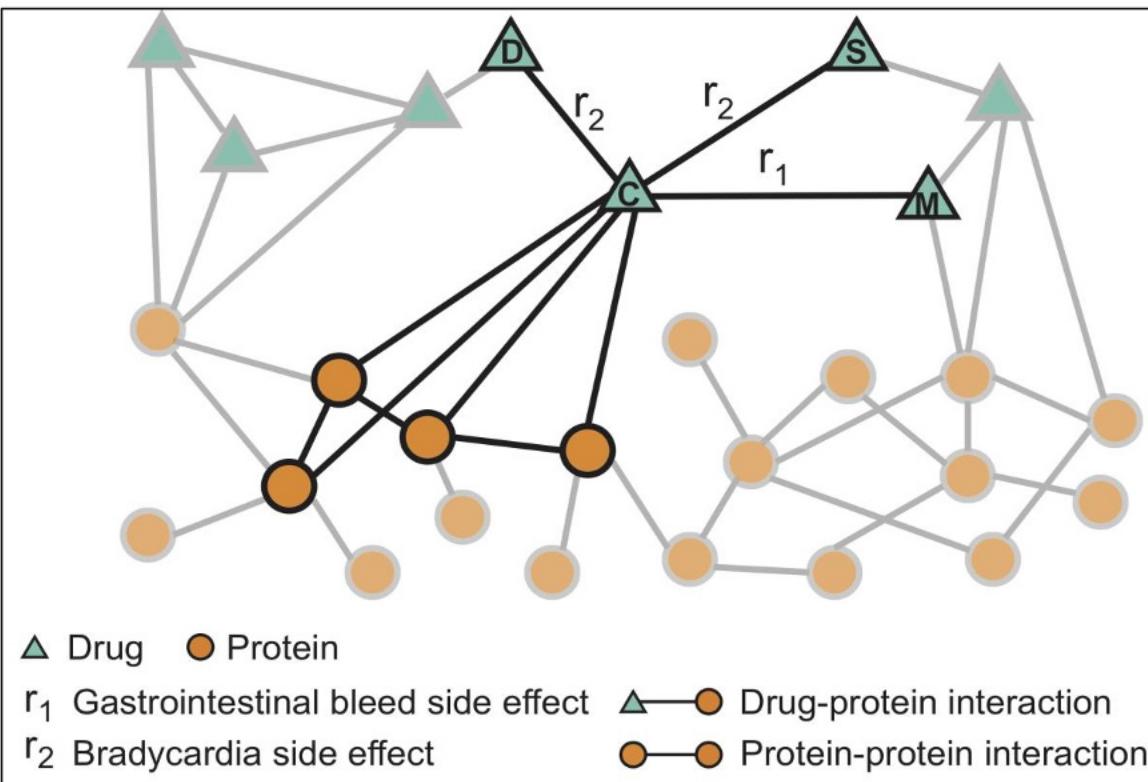
- 46% of people ages 70-79 take more than 5 drugs
- Many patients take more than 20 drugs to treat heart disease, depression, insomnia, etc.

**Task: Given a pair of drugs predict  
adverse side effects**

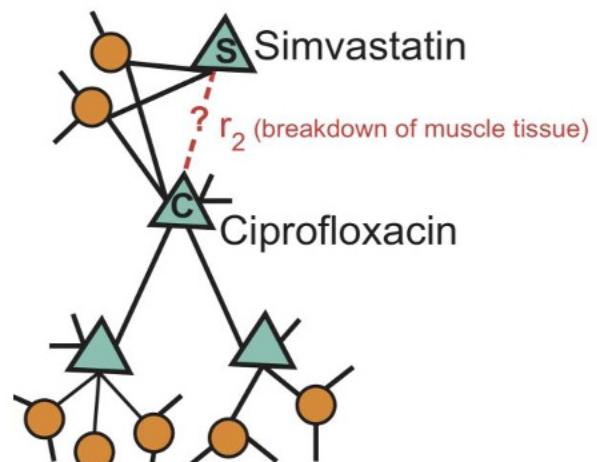


# Biomedical Graph Link Prediction

- **Nodes:** Drugs & Proteins
- **Edges:** Interactions



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



# Results: *De novo* Predictions

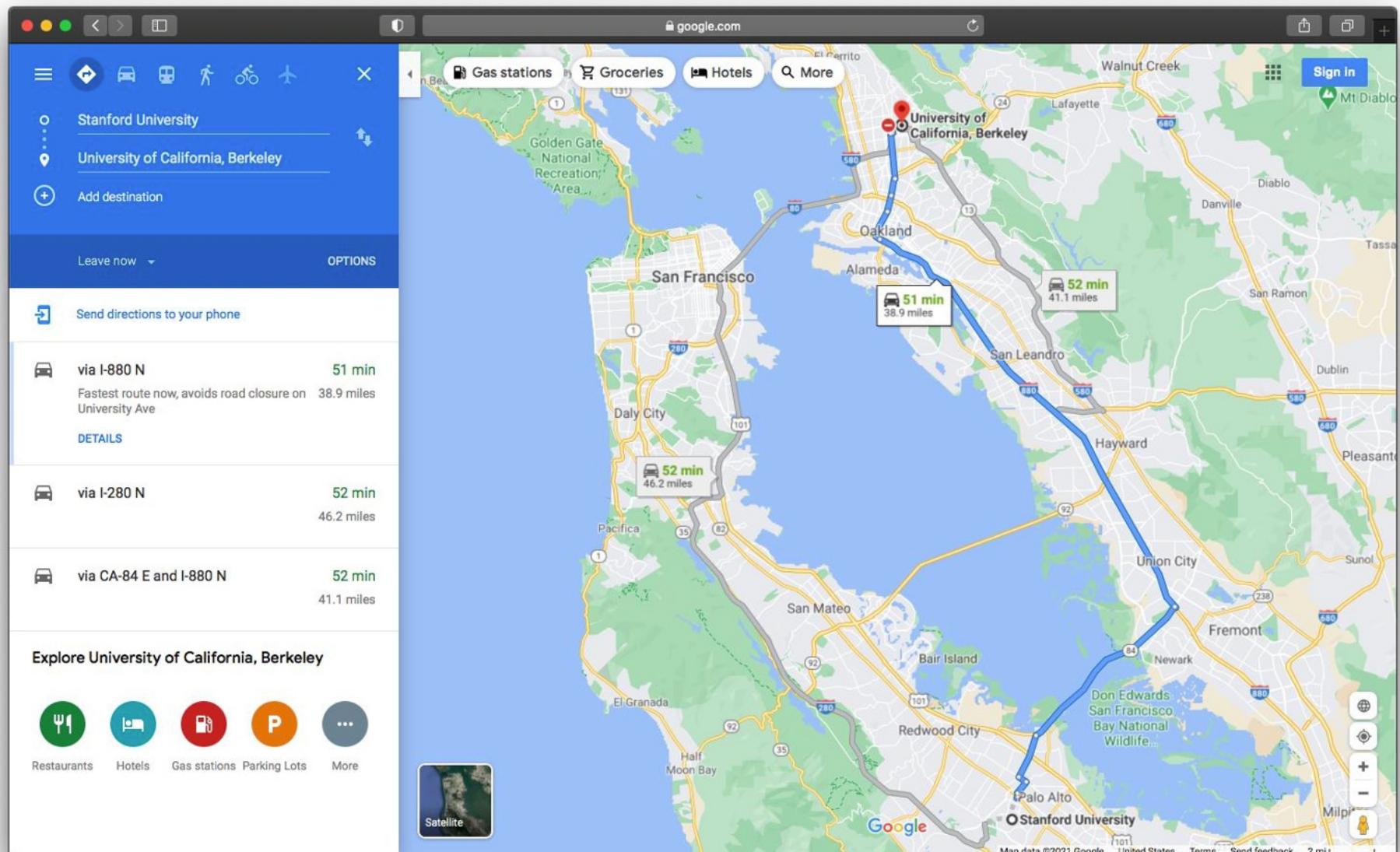
Rank	Drug $c$	Drug $d$	Side effect $r$	Evidence found
1	Pyrimethamine	Aliskiren	Sarcoma	<a href="#">Stage et al. 2015</a>
2	Tigecycline	Bimatoprost	Autonomic neuropathy	
3	Omeprazole	Dacarbazine	Telangiectases	
4	Tolcapone	Pyrimethamine	Breast disorder	<a href="#">Bicker et al. 2017</a>
5	Minoxidil	Paricalcitol	Cluster headache	
6	Omeprazole	Amoxicillin	Renal tubular acidosis	<a href="#">Russo et al. 2016</a>
7	Anagrelide	Azelaic acid	Cerebral thrombosis	
8	Atorvastatin	Amlodipine	Muscle inflammation	<a href="#">Banakh et al. 2017</a>
9	Aliskiren	Tioconazole	Breast inflammation	<a href="#">Parving et al. 2012</a>
10	Estradiol	Nadolol	Endometriosis	

*Case Report*

**Severe Rhabdomyolysis due to Presumed Drug Interactions  
between Atorvastatin with Amlodipine and Ticagrelor**

# Examples of Subgraph-level ML Tasks

# Example (4): Traffic Prediction



# Road Network as a Graph

- **Nodes:** Road segments
- **Edges:** Connectivity between road segments
- **Prediction:** Time of Arrival (ETA)

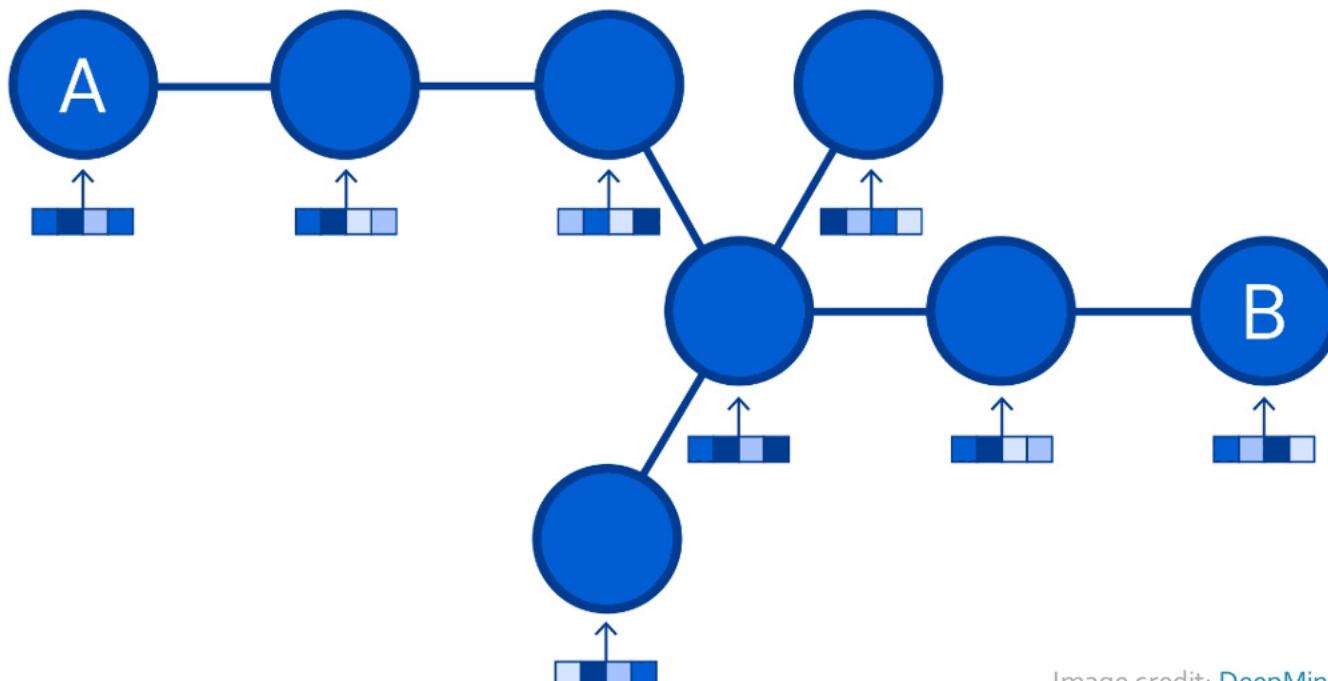
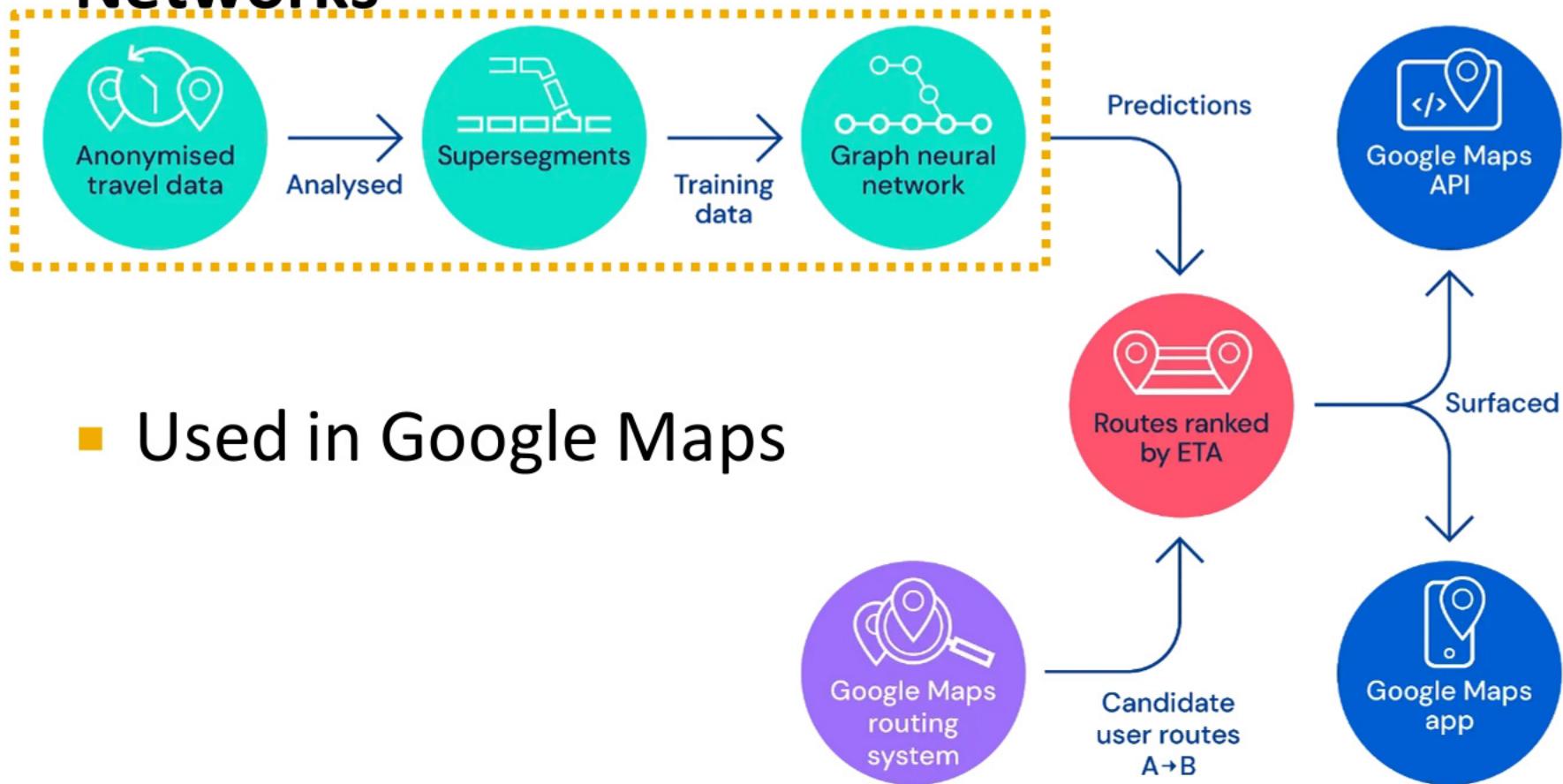


Image credit: [DeepMind](#)

# Traffic Prediction via GNN

## Predicting Time of Arrival with Graph Neural Networks

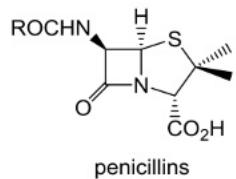


- Used in Google Maps

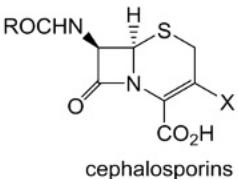
# Examples of Graph-level ML Tasks

# Example (5): Drug Discovery

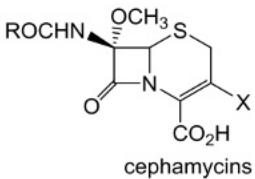
- Antibiotics are small molecular graphs
  - **Nodes:** Atoms
  - **Edges:** Chemical bonds



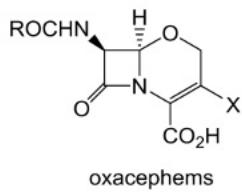
penicillins



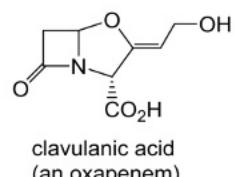
cephalosporins



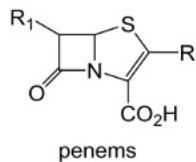
cephamycins



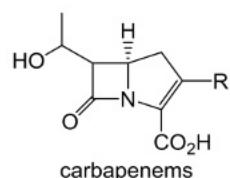
oxacephems



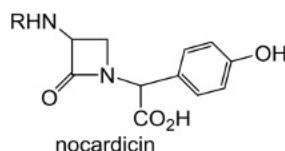
clavulanic acid  
(an oxapenem)



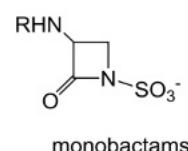
penems



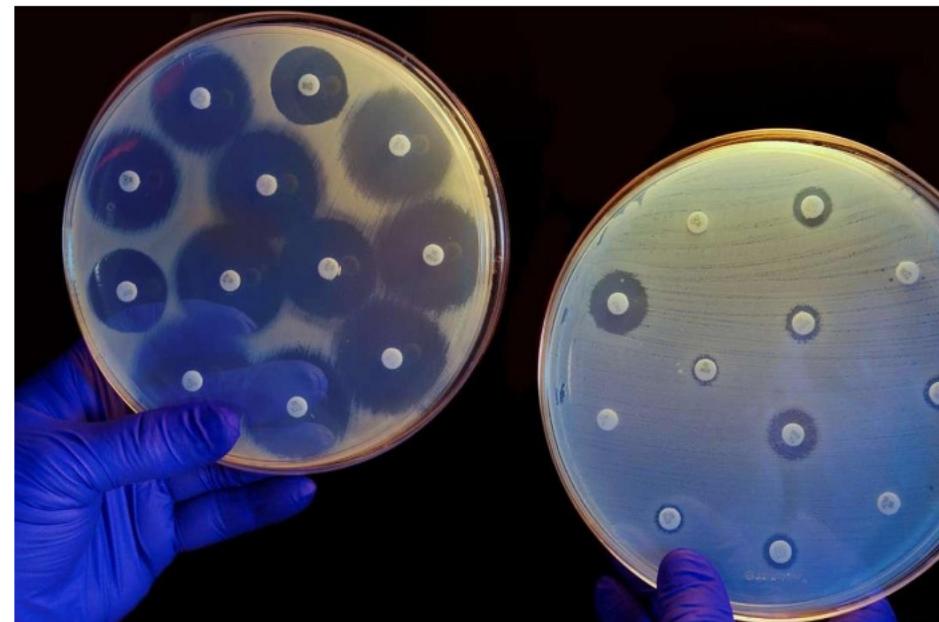
carbapenems



nocardicin



monobactams

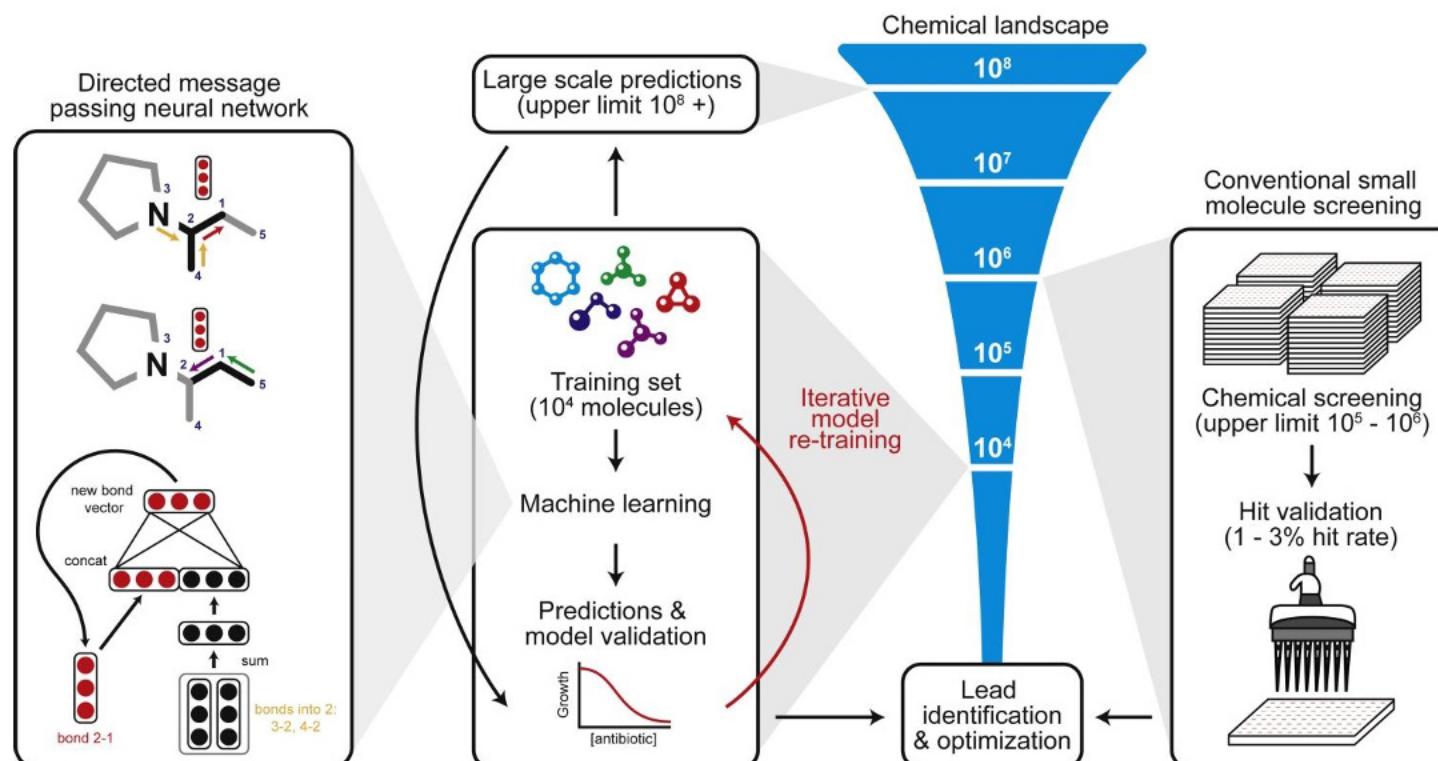


Konaklieva, Monika I. "Molecular targets of  $\beta$ -lactam-based antimicrobials: beyond the usual suspects." *Antibiotics* 3.2 (2014): 128-142.

Image credit: [CNN](#)

# Deep Learning for Antibiotic Discovery

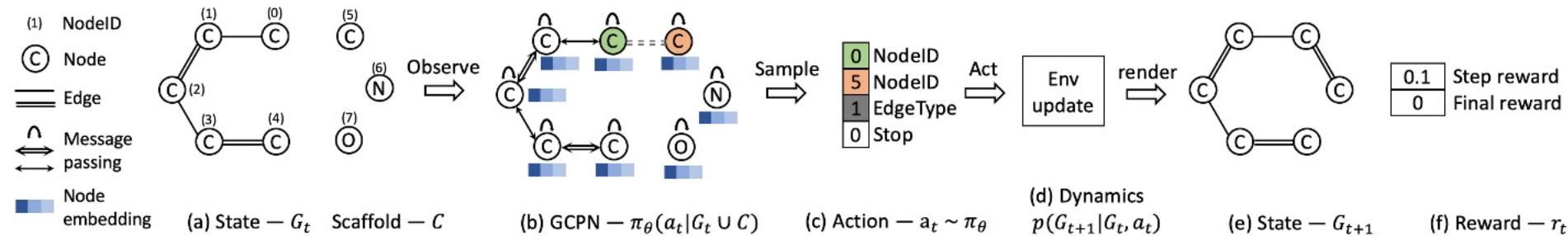
- A Graph Neural Network **graph classification model**
- Predict promising molecules from a pool of candidates



Stokes, Jonathan M., et al. "A deep learning approach to antibiotic discovery." Cell 180.4 (2020): 688-702.

# Molecule Generation / Optimization

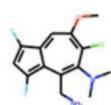
## Graph generation: Generating novel molecules



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



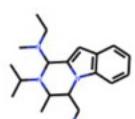
0.948



0.945



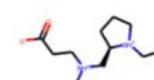
0.944



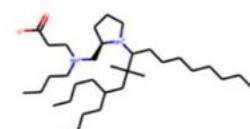
0.941

**Drug likeness**

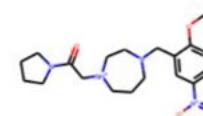
**Use case 2: Optimize existing molecules to have desirable properties**



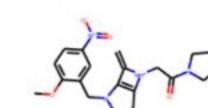
-8.32



-0.71



-5.55

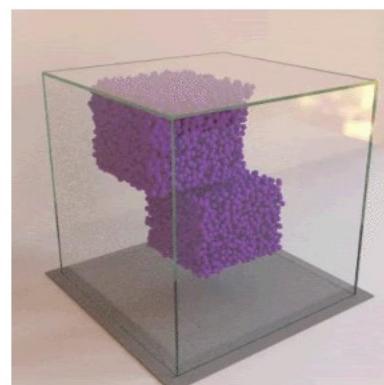
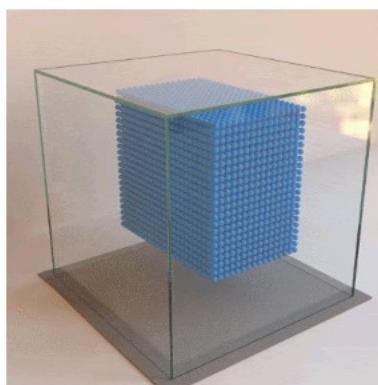


-1.78

# Example (6): Physics Simulation

Physical simulation as a graph:

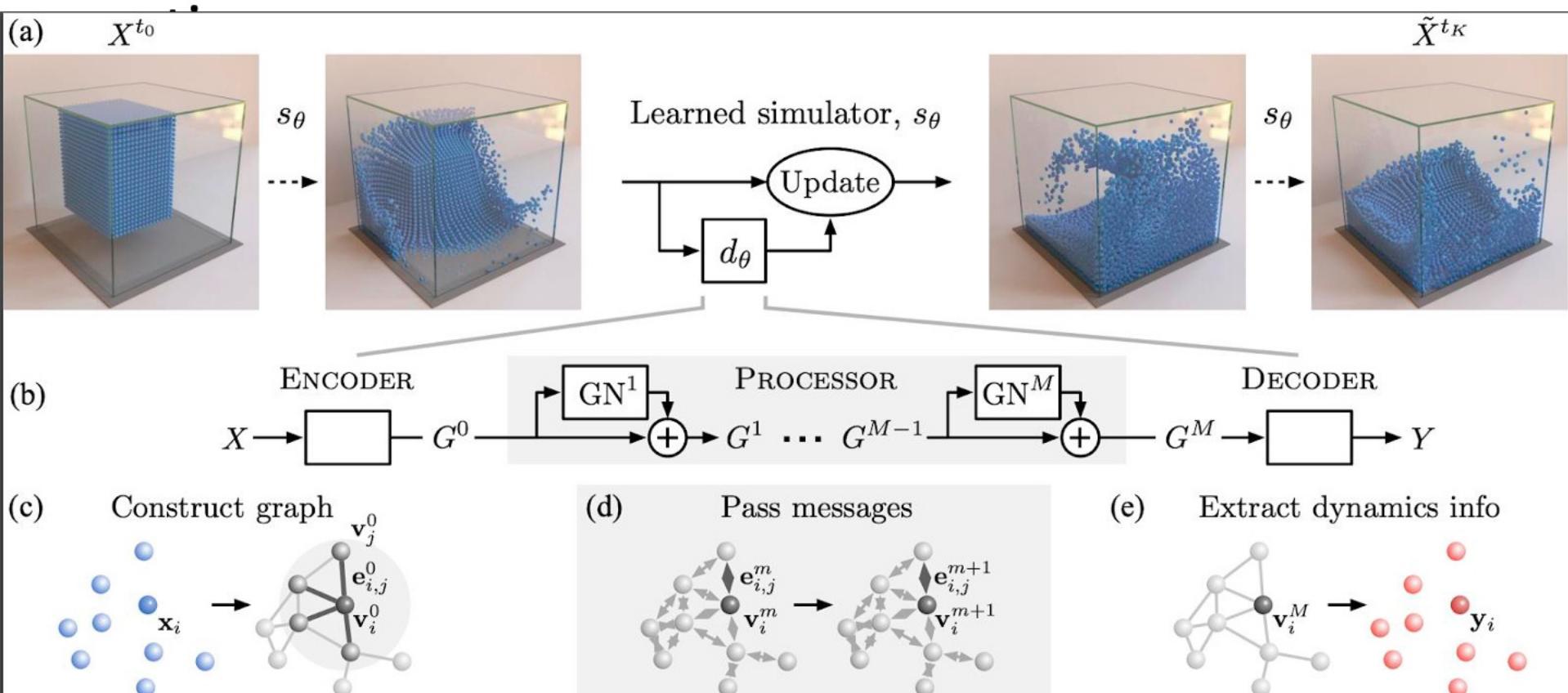
- **Nodes:** Particles
- **Edges:** Interaction between particles



# Simulation Learning Framework

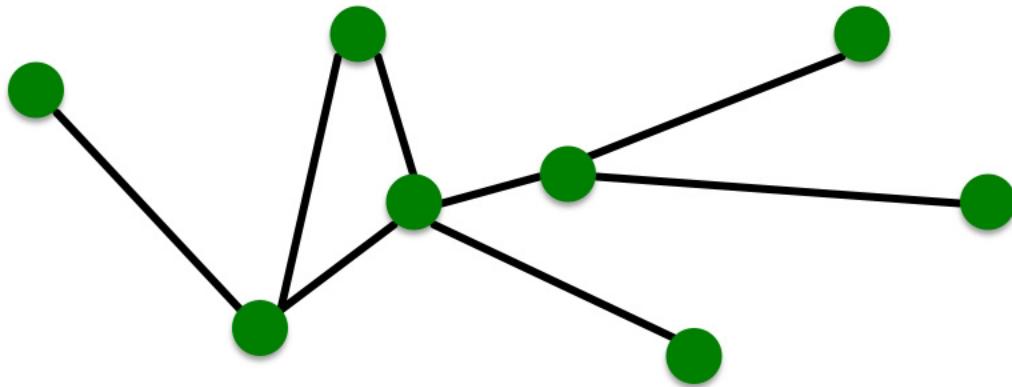
## A graph evolution task:

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



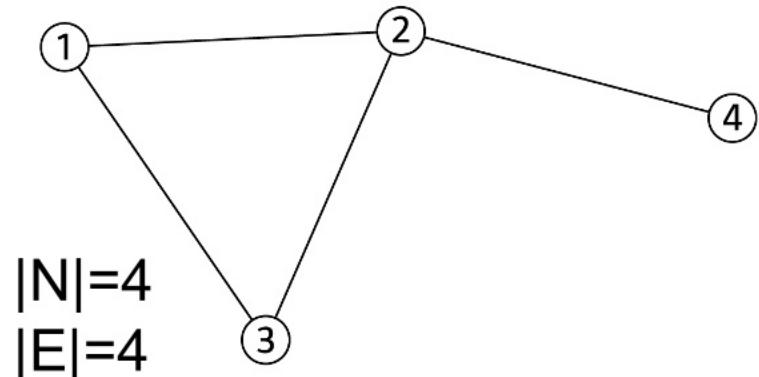
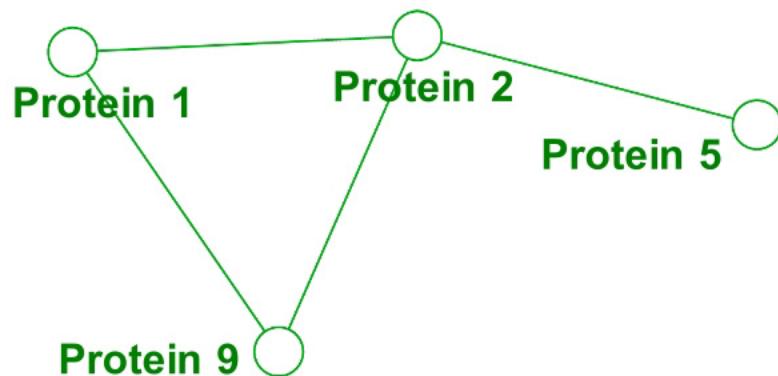
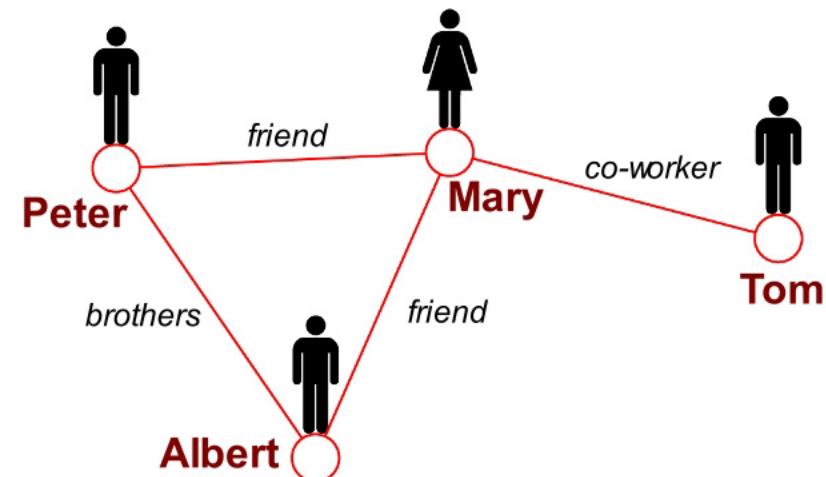
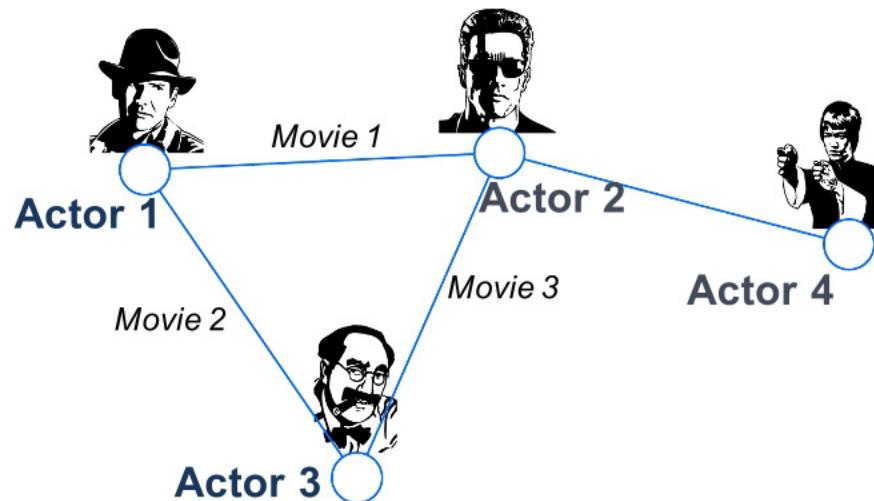
# Choice of Graph Representation

# Components of a Network



- **Objects:** nodes, vertices  $N$
- **Interactions:** links, edges  $E$
- **System:** network, graph  $G(N,E)$

# Graphs: A Common Language



# Choosing a Proper Representation

- If you connect individuals that work with each other, you will explore a **professional network**
- If you connect those that have a sexual relationship, you will be exploring **sexual networks**
- If you connect scientific papers that cite each other, you will be studying the **citation network**
- **If you connect all papers with the same word in the title, what will you be exploring?** It is a network, nevertheless



Image credit: [Euro Scientists](#)

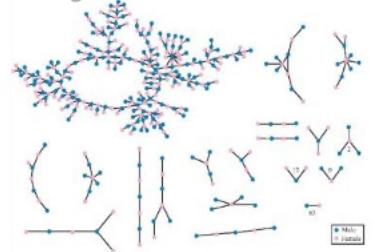
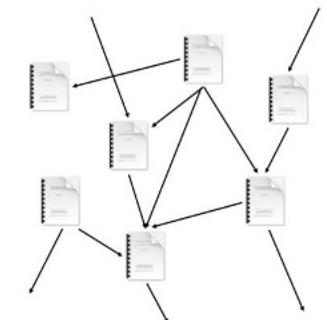


Image credit: [ResearchGate](#)



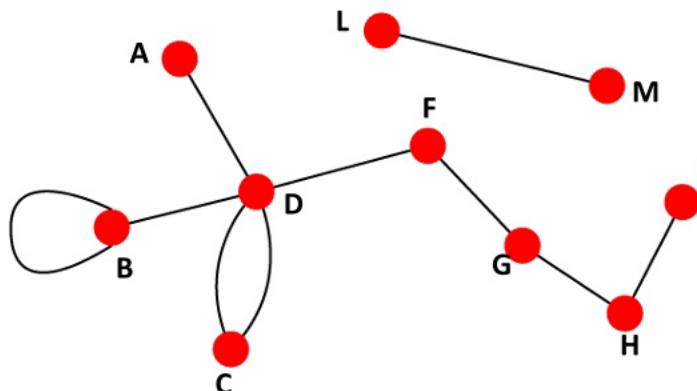
# How do you define a graph?

- **How to build a graph:**
  - What are nodes?
  - What are edges?
- **Choice of the proper network representation of a given domain/problem determines our ability to use networks successfully:**
  - In some cases, there is a unique, unambiguous representation
  - In other cases, the representation is by no means unique
  - The way you assign links will determine the nature of the question you can study

# Directed vs. Undirected Graphs

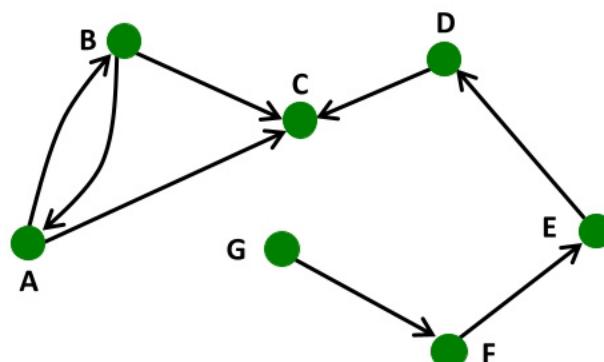
## Undirected

- Links: undirected  
(symmetrical, reciprocal)



## Directed

- Links: directed  
(arcs)



## Examples:

- Collaborations
- Friendship on Facebook

## Examples:

- Phone calls
- Following on Twitter

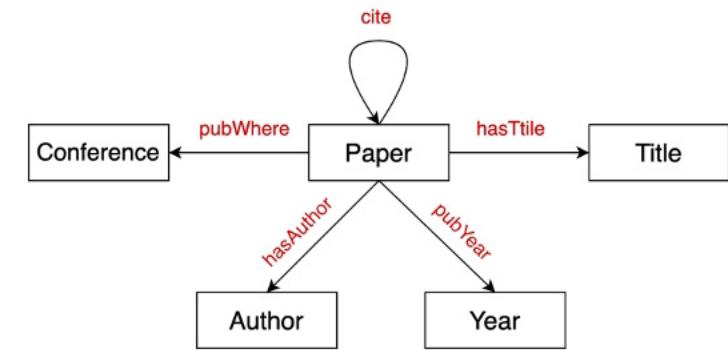
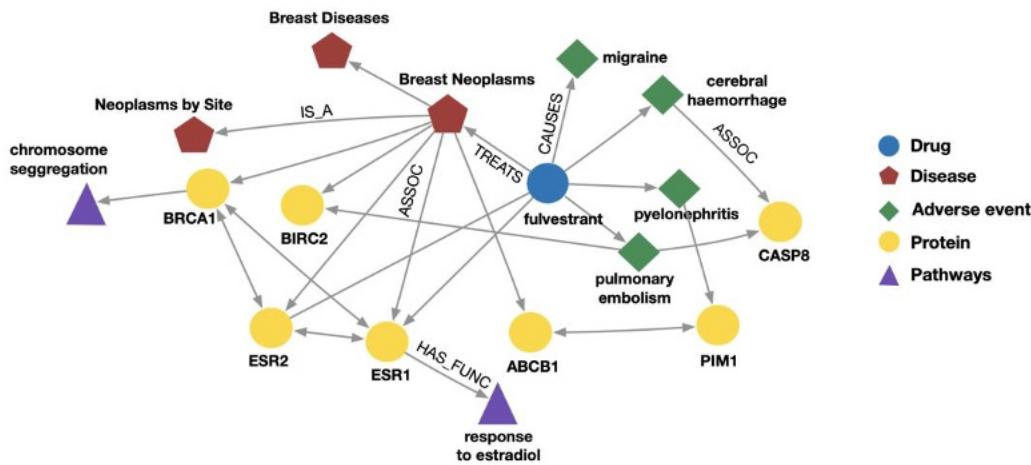
# Heterogeneous Graphs

- A heterogeneous graph is defined as

$$G = (V, E, R, T)$$

- Nodes with node types  $v_i \in V$
- Edges with relation types  $(v_i, r, v_j) \in E$
- Node type  $T(v_i)$
- Relation type  $r \in R$

# Many Graphs are Heterogeneous Graphs



## Biomedical Knowledge Graphs

Example node: Migraine

Example edge: (fulvestrant, Treats, Breast Neoplasms)

Example node type: Protein

Example edge type (relation): Causes

## Academic Graphs

Example node: ICML

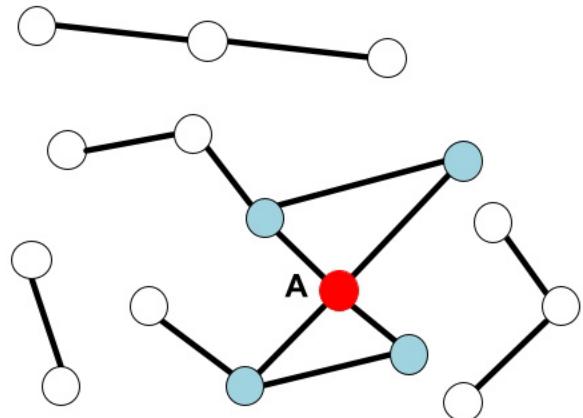
Example edge: (GraphSAGE, NeurIPS)

Example node type: Author

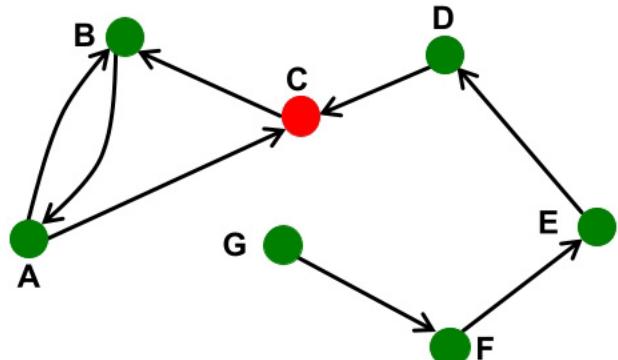
Example edge type (relation): pubYear

# Node Degrees

Undirected



Directed



**Source:** Node with  $k^{in} = 0$

**Sink:** Node with  $k^{out} = 0$

**Node degree,  $k_i$ :** the number of edges adjacent to node  $i$

$$k_A = 4$$

**Avg. degree:**  $\bar{k} = \langle k \rangle = \frac{1}{N} \sum_{i=1}^N k_i = \frac{2E}{N}$

In directed networks we define an **in-degree** and **out-degree**. The (total) degree of a node is the sum of in- and out-degrees.

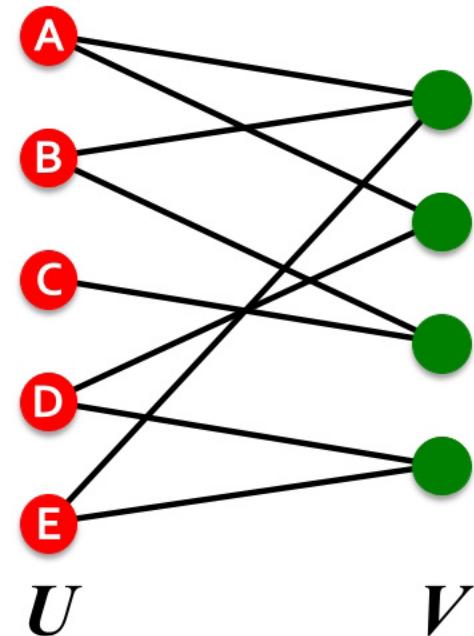
$$k_C^{in} = 2 \quad k_C^{out} = 1 \quad k_C = 3$$

$$\bar{k} = \frac{E}{N}$$

$$\bar{k}^{in} = \bar{k}^{out}$$

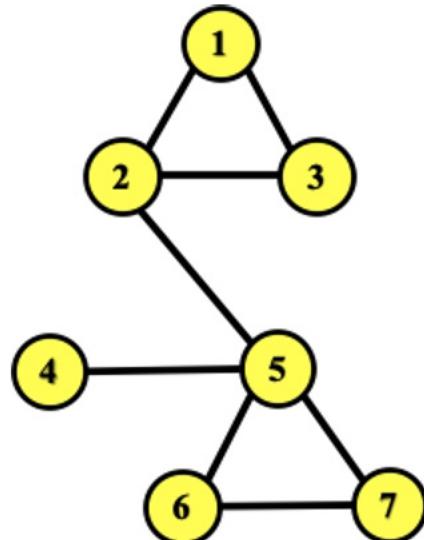
# Bipartite Graph

- **Bipartite graph** is a graph whose nodes can be divided into two disjoint sets  $U$  and  $V$  such that every link connects a node in  $U$  to one in  $V$ ; that is,  $U$  and  $V$  are **independent sets**
- **Examples:**
  - Authors-to-Papers (they authored)
  - Actors-to-Movies (they appeared in)
  - Users-to-Movies (they rated)
  - Recipes-to-Ingredients (they contain)
- **“Folded” networks:**
  - Author collaboration networks
  - Movie co-rating networks

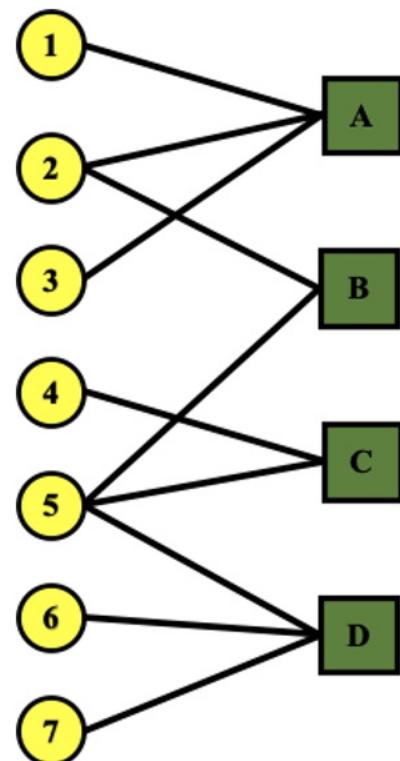


# Folded/Projected Bipartite Graphs

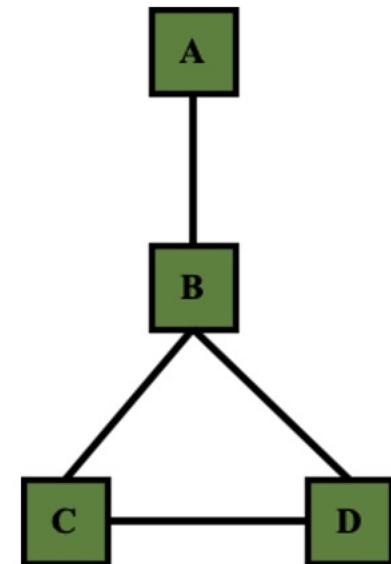
Projection U



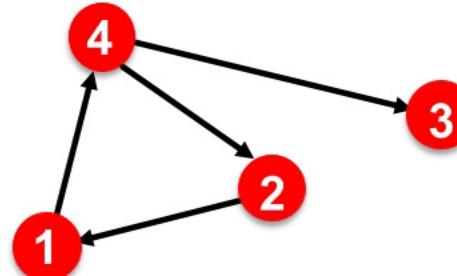
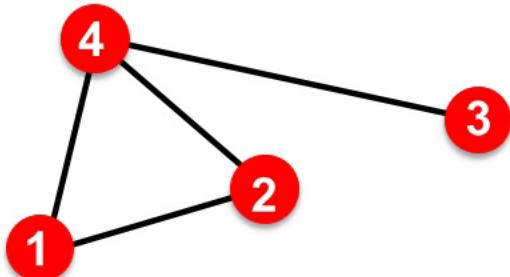
U      V



Projection V



# Representing Graphs: Adjacency Matrix



$A_{ij} = 1$  if there is a link from node  $i$  to node  $j$

$A_{ij} = 0$  otherwise

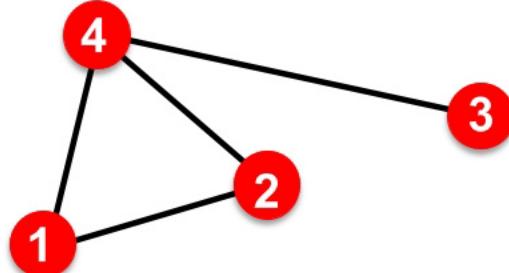
$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

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

Note that for a directed graph (right) the matrix is not symmetric.

# Adjacency Matrix

Undirected



$$A_{ij} = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

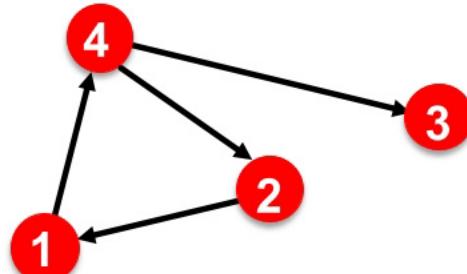
$$\begin{aligned} A_{ij} &= A_{ji} \\ A_{ii} &= 0 \end{aligned}$$

$$k_i = \sum_{j=1}^N A_{ij}$$

$$k_j = \sum_{i=1}^N A_{ij}$$

$$L = \frac{1}{2} \sum_{i=1}^N k_i = \frac{1}{2} \sum_{ij}^N A_{ij}$$

Directed



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

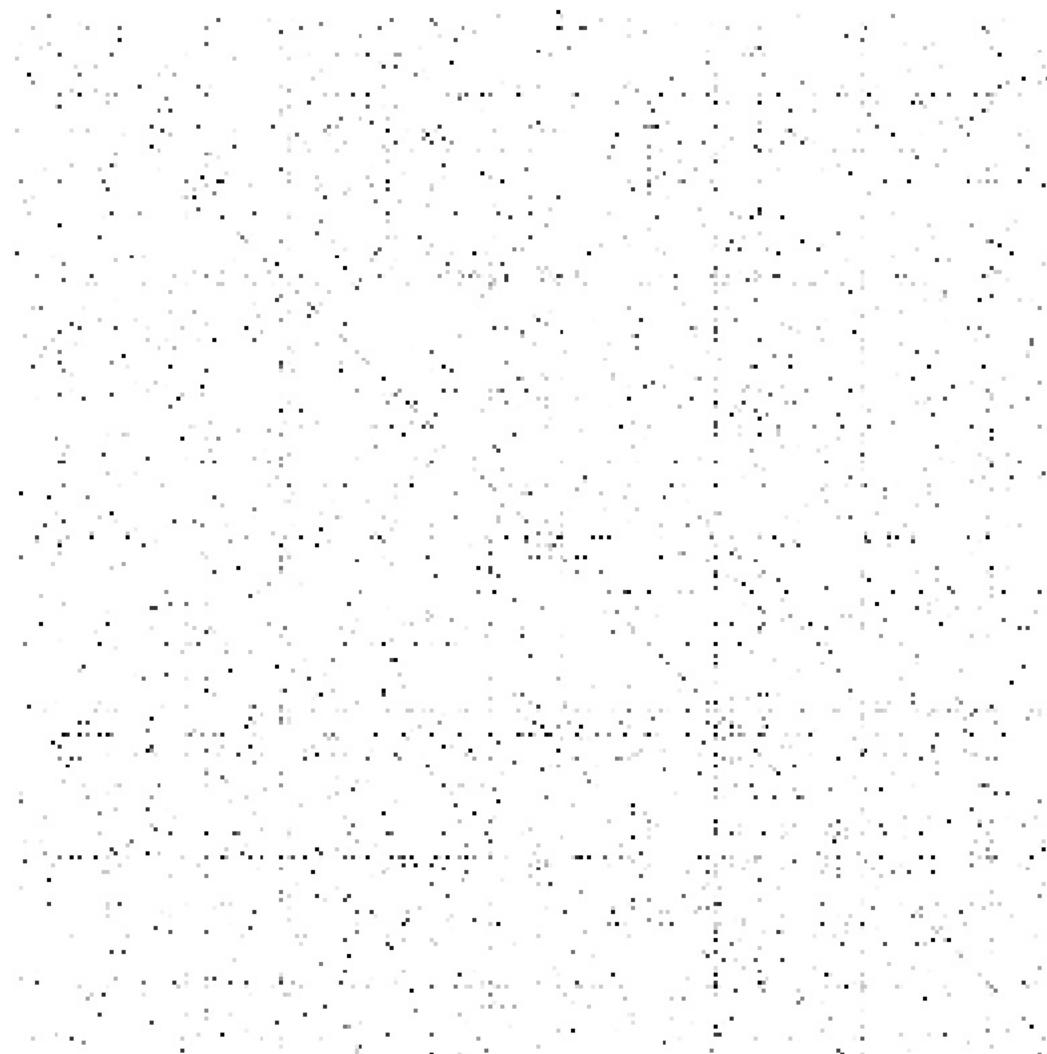
$$\begin{aligned} A_{ij} &= A_{ji} \\ A_{ii} &= 0 \end{aligned}$$

$$k_i^{out} = \sum_{j=1}^N A_{ij}$$

$$k_j^{in} = \sum_{i=1}^N A_{ij}$$

$$L = \sum_{i=1}^N k_i^{in} = \sum_{j=1}^N k_j^{out} = \sum_{i,j}^N A_{ij}$$

# Adjacency Matrices are Sparse



# Networks are Sparse Graphs

Most real-world networks are **sparse**

$$E \ll E_{\max} \text{ (or } k \ll N-1)$$

NETWORK	NODES	LINKS	DIRECTED/ UNDIRECTED	N	L	$\langle k \rangle$
Internet	Routers	Internet connections	Undirected	192,244	609,066	6.33
WWW	Webpages	Links	Directed	325,729	1,497,134	4.60
Power Grid	Power plants, transformers	Cables	Undirected	4,941	6,594	2.67
Phone Calls	Subscribers	Calls	Directed	36,595	91,826	2.51
Email	Email Addresses	Emails	Directed	57,194	103,731	1.81
Science Collaboration	Scientists	Co-authorship	Undirected	23,133	93,439	8.08
Actor Network	Actors	Co-acting	Undirected	702,388	29,397,908	83.71
Citation Network	Paper	Citations	Directed	449,673	4,689,479	10.43
E. Coli Metabolism	Metabolites	Chemical reactions	Directed	1,039	5,802	5.58
Protein Interactions	Proteins	Binding interactions	Undirected	2,018	2,930	2.90

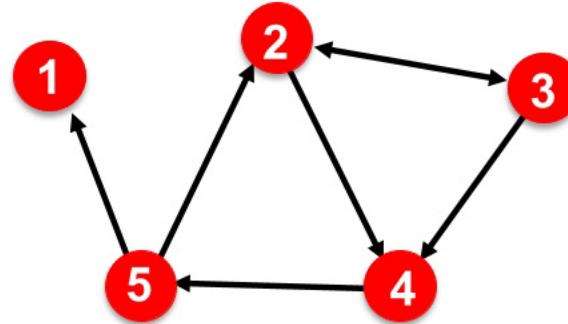
**Consequence:** Adjacency matrix is filled with zeros!

(Density of the matrix ( $E/N^2$ ): WWW=1.51x10<sup>-5</sup>, MSN IM = 2.27x10<sup>-8</sup>)

# Representing Graphs: Edge list

- Represent graph as a **list of edges**:

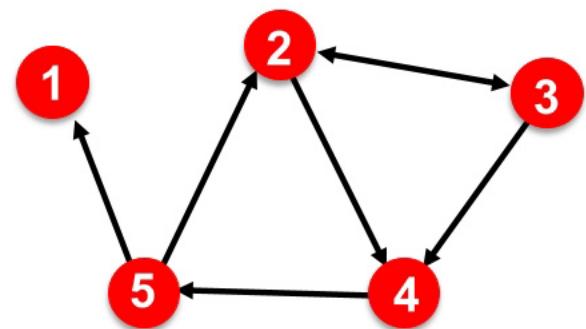
- (2, 3)
- (2, 4)
- (3, 2)
- (3, 4)
- (4, 5)
- (5, 2)
- (5, 1)



# Representing Graphs: Adjacency list

## ■ **Adjacency list:**

- Easier to work with if network is
  - Large
  - Sparse
- Allows us to quickly retrieve all neighbors of a given node
  - 1:
  - 2: 3, 4
  - 3: 2, 4
  - 4: 5
  - 5: 1, 2



# Node and Edge Attributes

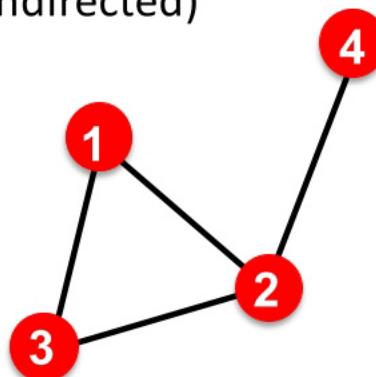
## Possible options:

- Weight (*e.g.*, frequency of communication)
- Ranking (best friend, second best friend...)
- Type (friend, relative, co-worker)
- Sign: Friend vs. Foe, Trust vs. Distrust
- Properties depending on the structure of the rest of the graph: Number of common friends

# More Types of Graphs

## ■ Unweighted

(undirected)



$$A_{ij} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

$$A_{ii} = 0$$

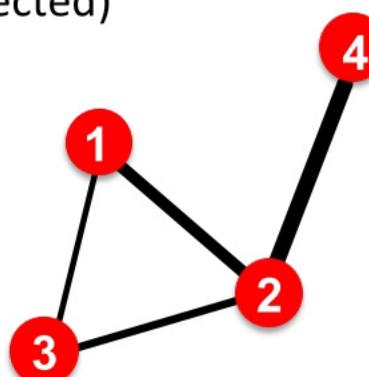
$$A_{ij} = A_{ji}$$

$$E = \frac{1}{2} \sum_{i,j=1}^N A_{ij} \quad \bar{k} = \frac{2E}{N}$$

Examples: Friendship, Hyperlink

## ■ Weighted

(undirected)



$$A_{ij} = \begin{pmatrix} 0 & 2 & 0.5 & 0 \\ 2 & 0 & 1 & 4 \\ 0.5 & 1 & 0 & 0 \\ 0 & 4 & 0 & 0 \end{pmatrix}$$

$$A_{ii} = 0$$

$$A_{ij} = A_{ji}$$

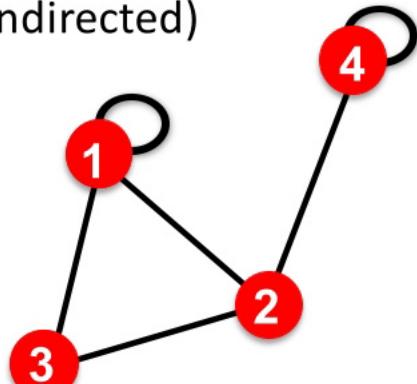
$$E = \frac{1}{2} \sum_{i,j=1}^N \text{nonzero}(A_{ij}) \quad \bar{k} = \frac{2E}{N}$$

Examples: Collaboration, Internet, Roads

# More Types of Graphs

## ■ Self-edges (self-loops)

(undirected)



$$A_{ij} = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

$$A_{ii} \neq 0$$

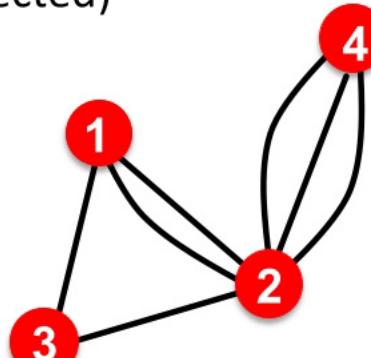
$$A_{ij} = A_{ji}$$

$$E = \frac{1}{2} \sum_{i,j=1, i \neq j}^N A_{ij} + \sum_{i=1}^N A_{ii}$$

Examples: Proteins, Hyperlinks

## ■ Multigraph

(undirected)



$$A_{ij} = \begin{pmatrix} 0 & 2 & 1 & 0 \\ 2 & 0 & 1 & 3 \\ 1 & 1 & 0 & 0 \\ 0 & 3 & 0 & 0 \end{pmatrix}$$

$$A_{ii} = 0$$

$$E = \frac{1}{2} \sum_{i,j=1}^N \text{nonzero}(A_{ij})$$

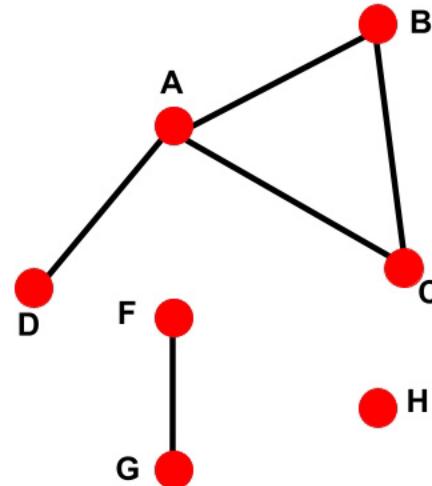
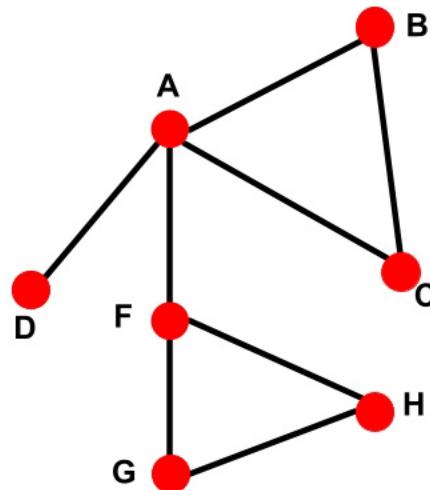
$$A_{ij} = A_{ji}$$

$$\bar{k} = \frac{2E}{N}$$

Examples: Communication, Collaboration

# Connectivity of Undirected Graphs

- **Connected (undirected) graph:**
  - Any two vertices can be joined by a path
- A disconnected graph is made up by two or more connected components



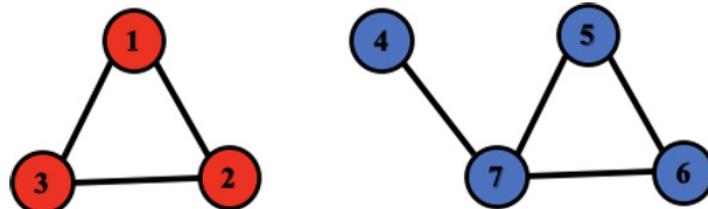
Largest Component:  
**Giant Component**

Isolated node (node H)

# Connectivity: Example

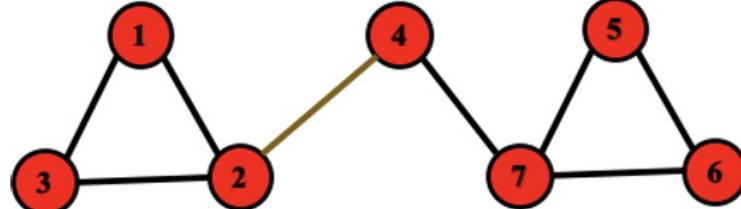
The adjacency matrix of a network with several components can be written in a block-diagonal form, so that nonzero elements are confined to squares, with all other elements being zero:

Disconnected



$$\begin{pmatrix} \textcolor{red}{\begin{matrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{matrix}} & \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix} \\ \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix} & \textcolor{blue}{\begin{matrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{matrix}} \end{pmatrix}$$

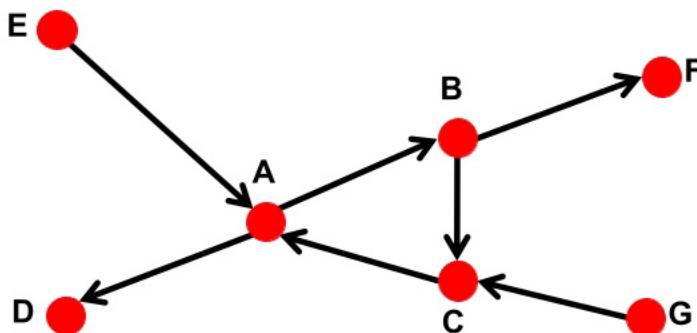
Connected



$$\begin{pmatrix} \textcolor{red}{\begin{matrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{matrix}} & \begin{matrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{matrix} \\ \begin{matrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix} & \textcolor{blue}{\begin{matrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{matrix}} \end{pmatrix}$$

# Connectivity of Directed Graphs

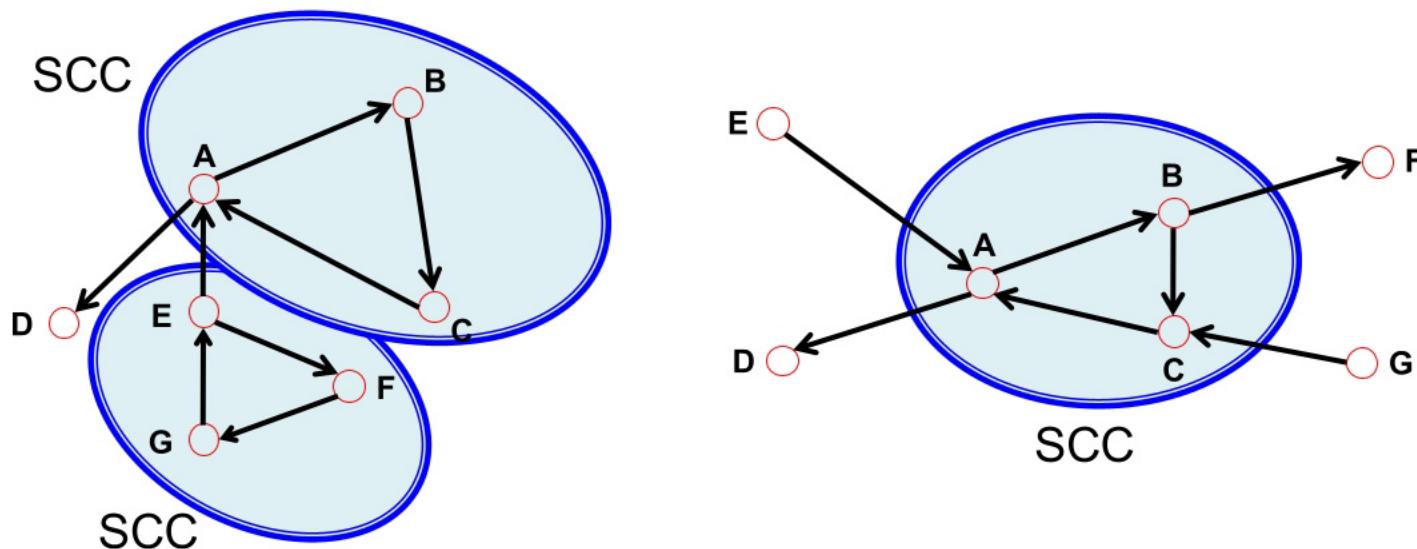
- **Strongly connected directed graph**
  - has a path from each node to every other node and vice versa (e.g., A-B path and B-A path)
- **Weakly connected directed graph**
  - is connected if we disregard the edge directions



Graph on the left is connected but not strongly connected (e.g., there is no way to get from F to G by following the edge directions).

# Connectivity of Directed Graphs

- Strongly connected components (SCCs) can be identified, but not every node is part of a nontrivial strongly connected component.

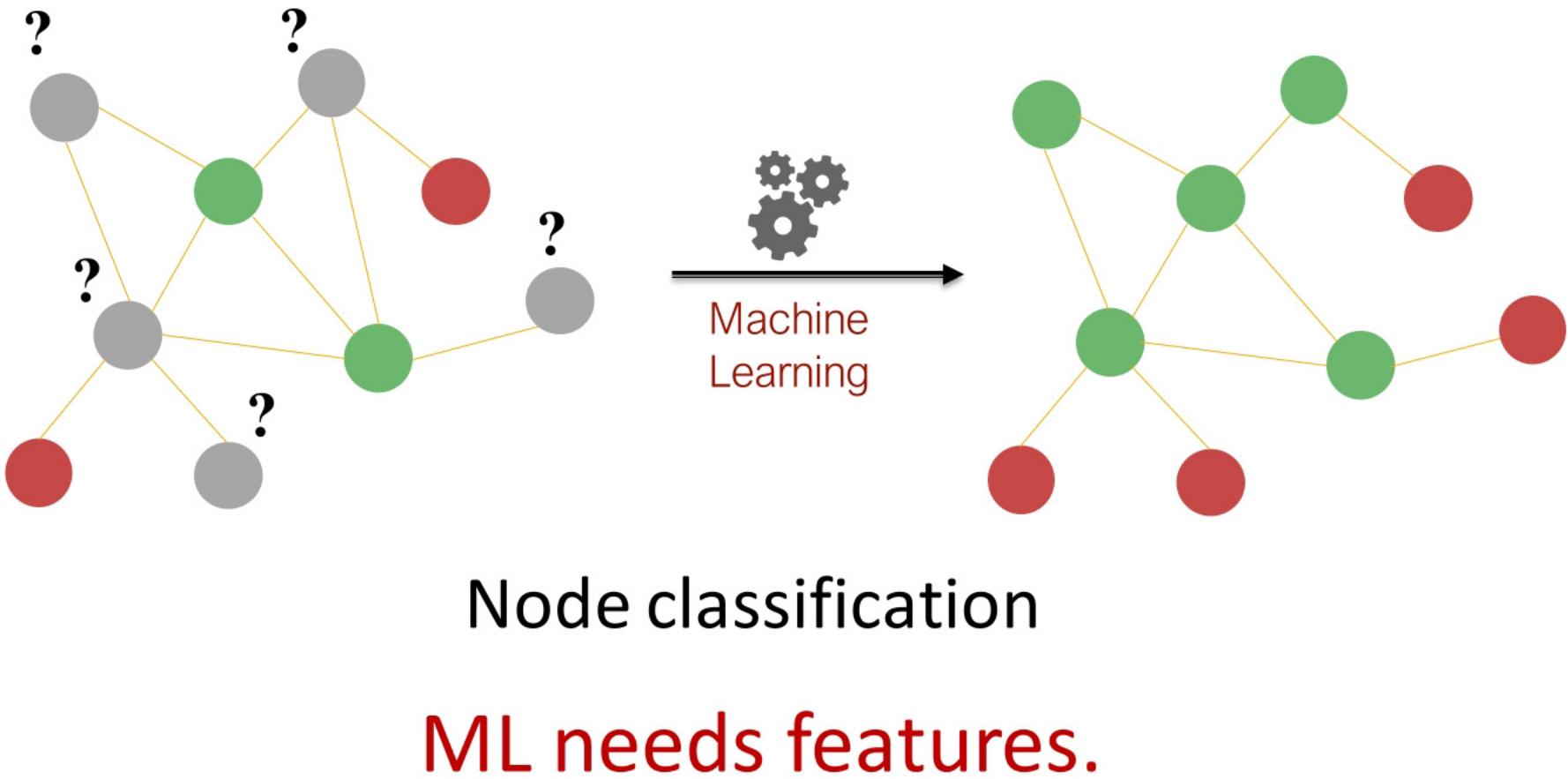


**In-component:** nodes that can reach the SCC,

**Out-component:** nodes that can be reached from the SCC.

# **Node-Level Tasks and Features**

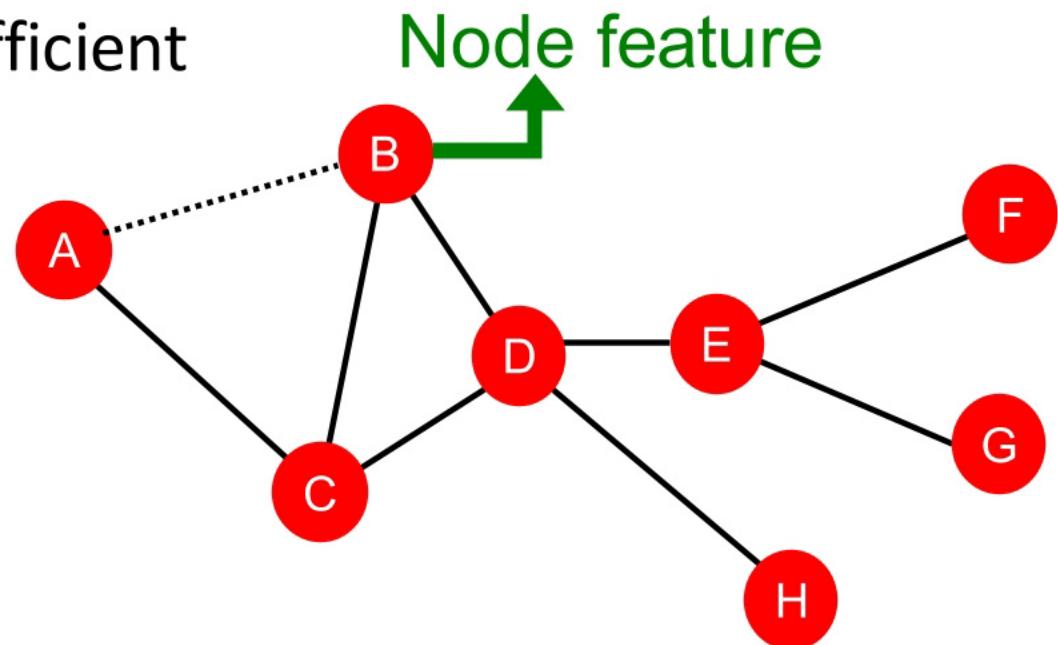
# Node-Level Tasks



# Node-Level Features: Overview

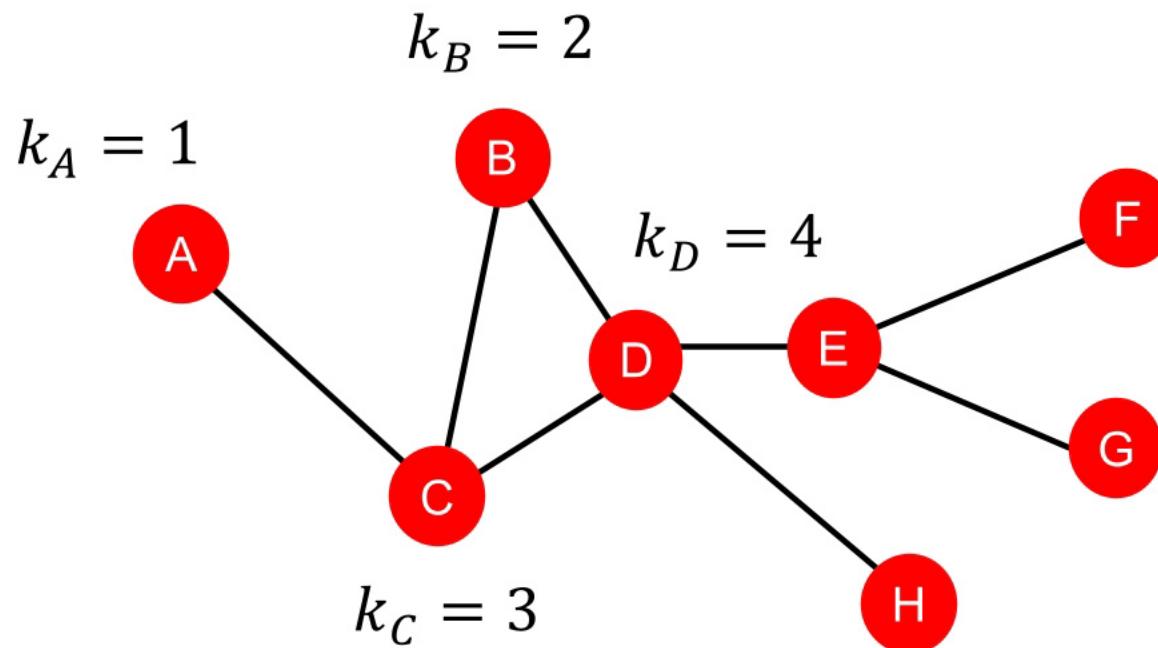
**Goal:** Characterize the structure and position of a node in the network:

- Node degree
- Node centrality
- Clustering coefficient
- Graphlets



# Node Features: Node Degree

- The degree  $k_v$  of node  $v$  is the number of edges (neighboring nodes) the node has.
- Treats all neighboring nodes equally.



# Node Features: Node Centrality

- Node degree counts the neighboring nodes without capturing their importance.
- Node centrality  $c_v$ , takes the node importance in a graph into account
- **Different ways to model importance:**
  - Eigenvector centrality
  - Betweenness centrality
  - Closeness centrality
  - and many others...

# Node Centrality (1)

## ■ Eigenvector centrality:

- A node  $v$  is important if **surrounded by important neighboring nodes**  $u \in N(v)$ .
- We model the centrality of node  $v$  as **the sum of the centrality of neighboring nodes**:

$$c_v = \frac{1}{\lambda} \sum_{u \in N(v)} c_u$$

$\lambda$  is normalization constant (it will turn out to be the largest eigenvalue of  $A$ )

- Notice that the above equation models centrality in a **recursive manner**. **How do we solve it?**

# Node Centrality (1)

## ■ Eigenvector centrality:

- Rewrite the recursive equation in the matrix form.

$$c_v = \frac{1}{\lambda} \sum_{u \in N(v)} c_u \quad \longleftrightarrow \quad \lambda c = A c$$

$\lambda$  is normalization const  
(largest eigenvalue of  $A$ )

- $A$ : Adjacency matrix  
 $A_{uv} = 1$  if  $u \in N(v)$
- $c$ : Centrality vector
- $\lambda$ : Eigenvalue

- We see that centrality  $c$  is the **eigenvector of  $A$ !**
- The largest eigenvalue  $\lambda_{max}$  is always positive and unique (by Perron-Frobenius Theorem).
- The eigenvector  $c_{max}$  corresponding to  $\lambda_{max}$  is used for centrality.

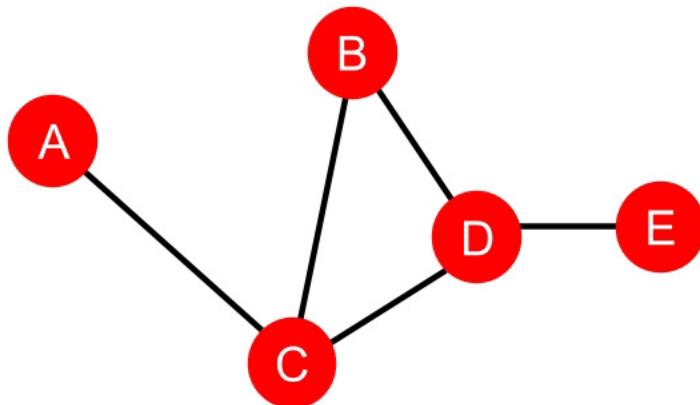
# Node Centrality (2)

## ■ Betweenness centrality:

- A node is important if it lies on many shortest paths between other nodes.

$$c_v = \sum_{s \neq v \neq t} \frac{\#(\text{shortest paths between } s \text{ and } t \text{ that contain } v)}{\#(\text{shortest paths between } s \text{ and } t)}$$

- Example:



$$\begin{aligned} c_A &= c_B = c_E = 0 \\ c_C &= 3 \\ &\quad (\text{A-C-B, A-C-D, A-C-D-E}) \\ c_D &= 3 \\ &\quad (\text{A-C-D-E, B-D-E, C-D-E}) \end{aligned}$$

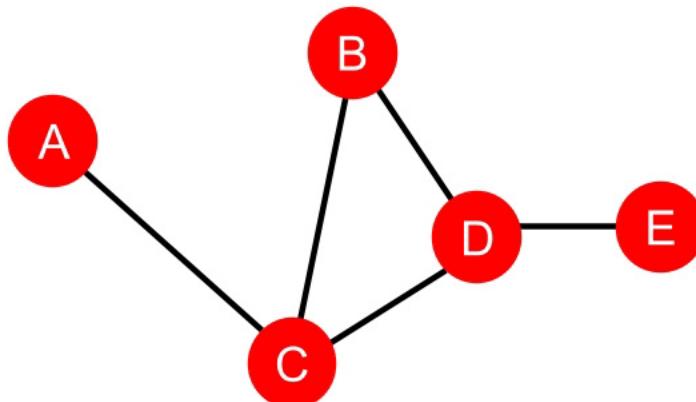
# Node Centrality (3)

## ■ Closeness centrality:

- A node is important if it has small shortest path lengths to all other nodes.

$$c_v = \frac{1}{\sum_{u \neq v} \text{shortest path length between } u \text{ and } v}$$

- Example:



$$c_A = 1/(2 + 1 + 2 + 3) = 1/8$$

(A-C-B, A-C, A-C-D, A-C-D-E)

$$c_D = 1/(2 + 1 + 1 + 1) = 1/5$$

(D-C-A, D-B, D-C, D-E)

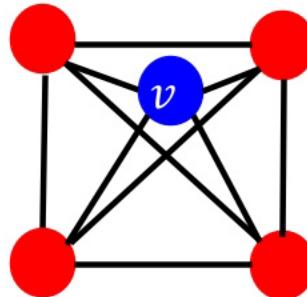
# Node Features: Clustering Coefficient

- Measures how connected  $v$ 's neighboring nodes are:

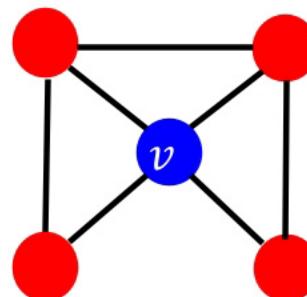
$$e_v = \frac{\text{\#(edges among neighboring nodes)}}{\binom{k_v}{2}} \in [0,1]$$

#(node pairs among  $k_v$  neighboring nodes)  
In our examples below the denominator is 6 (4 choose 2).

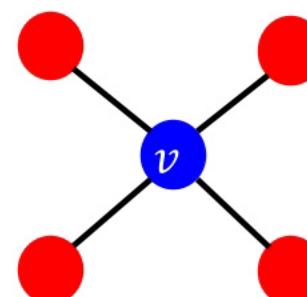
- Examples:



$$e_v = 1$$



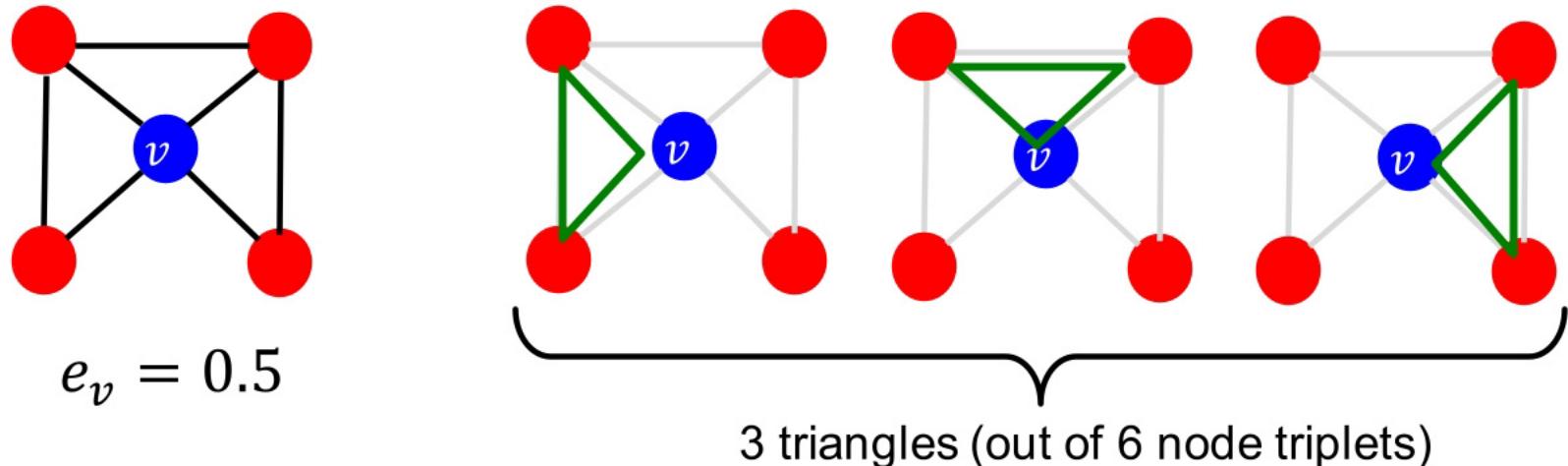
$$e_v = 0.5$$



$$e_v = 0$$

# Node Features: Graphlets

- **Observation:** Clustering coefficient counts the #(triangles) in the ego-network ☐



- We can generalize the above by counting #(pre-specified subgraphs, i.e., **graphlets**).

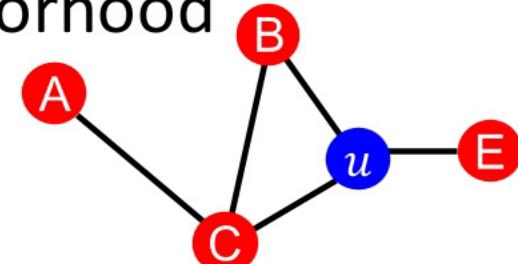
# Node Features: Graphlets

- **Goal:** Describe network structure around node  $u$

- **Graphlets** are small subgraphs that describe the structure of node  $u$ 's network neighborhood

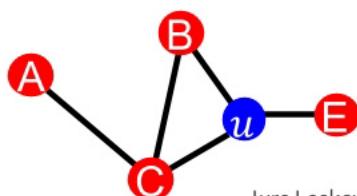
Analogy:

- **Degree** counts **#(edges)** that a node touches
- **Clustering coefficient** counts **#(triangles)** that a node touches.
- **Graphlet Degree Vector (GDV)**: Graphlet-base features for nodes
  - **GDV** counts **#(graphlets)** that a node touches



# Node Features: Graphlets

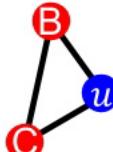
- Considering graphlets of size 2-5 nodes we get:
  - **Vector of 73 coordinates** is a signature of a node that describes the topology of node's neighborhood
- Graphlet degree vector provides a measure of a **node's local network topology**:
  - Comparing vectors of two nodes provides a more detailed measure of local topological similarity than node degrees or clustering coefficient.



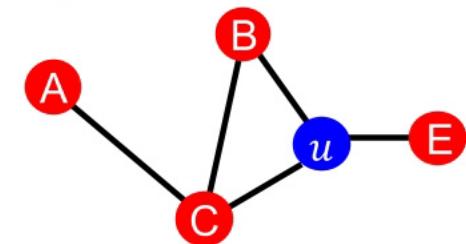
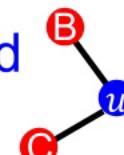
# Induced Subgraph & Isomorphism

- Def: Induced subgraph is another graph, formed from a subset of vertices and *all* of the edges connecting the vertices in that subset.

Induced subgraph:

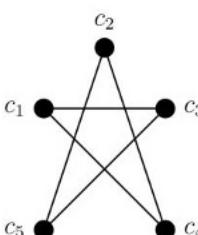
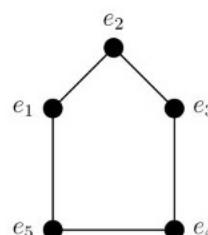


Not induced subgraph:



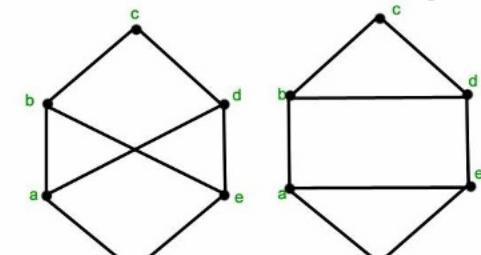
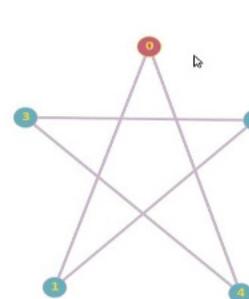
- Def: Graph Isomorphism

- Two graphs which contain the same number of nodes connected in the same way are said to be isomorphic.



Isomorphic

Node mapping: (e2,c2), (e1, c5),  
(e3,c4), (e5,c3), (e4,c1)



Non-Isomorphic

The right graph has cycles of length 3 but the left graph does not, so the graphs cannot be isomorphic.

Source: Mathoverflow

# Node Features: Graphlets

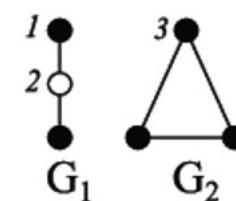
**Graphlets:** Rooted connected induced non-isomorphic subgraphs:

Take some nodes and all the edges between them.

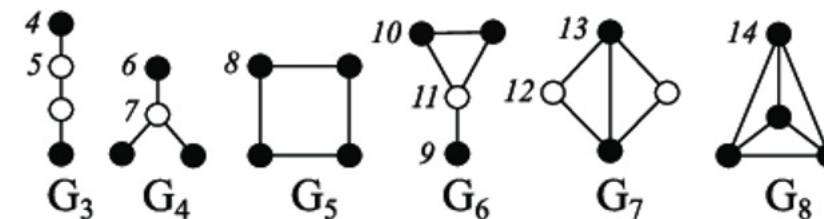
2-node graphlet



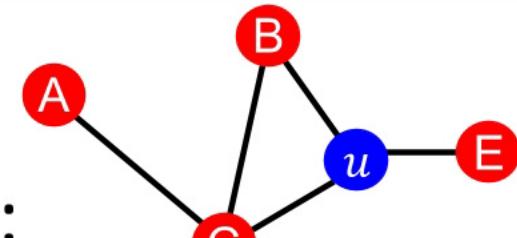
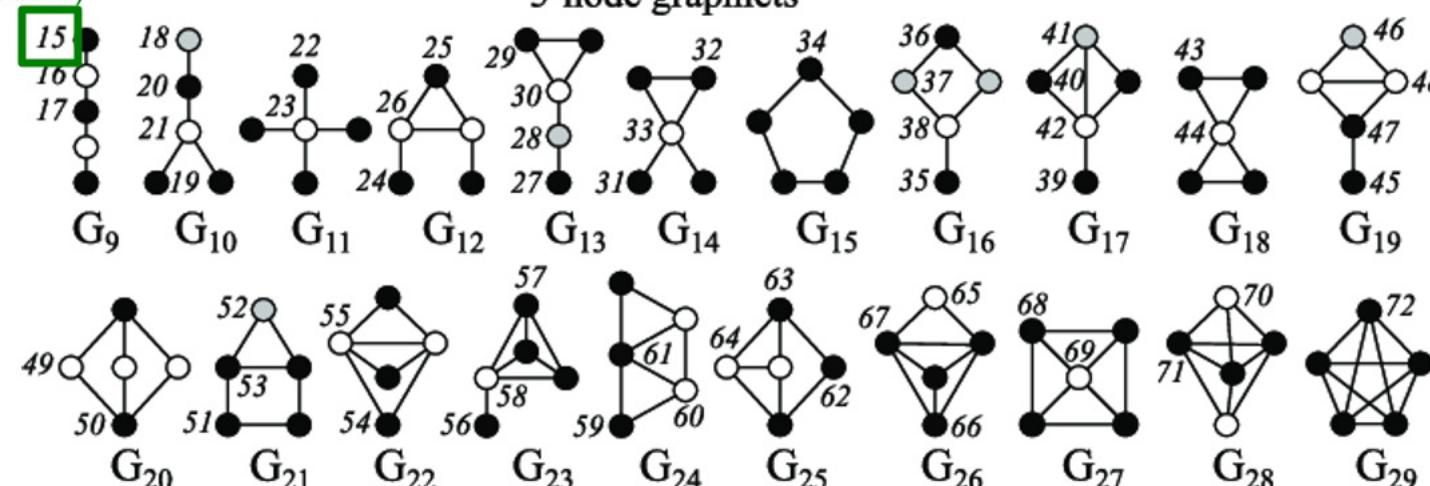
3-node graphlets



4-node graphlets



Graphlet id (Root/“position” of node  $u$ )



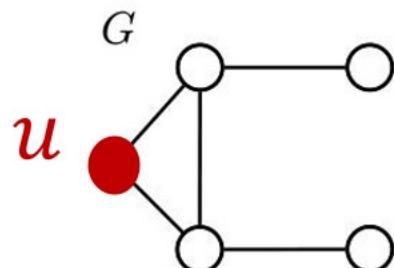
$u$  has graphlets: 0, 1, 2, 3, 5, 10, 11, ...

There are 73 different graphlets on up to 5 nodes

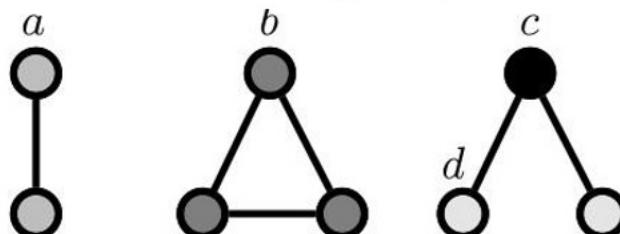
# Node Features: Graphlets

- **Graphlet Degree Vector (GDV):** A count vector of graphlets rooted at a given node.

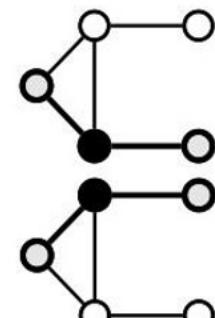
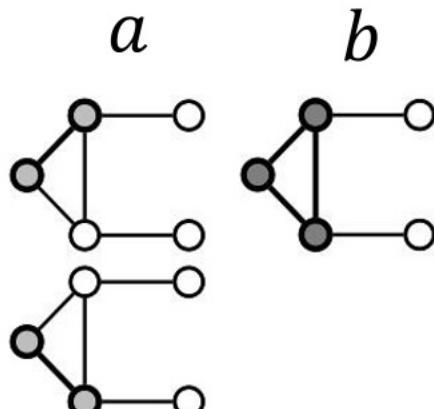
- **Example:**



Possible graphlets up to size 3



Graphlet instances of node  $u$ :



GDV of node  $u$ :  
 $a, b, c, d$   
 $[2, 1, 0, 2]$

# Node-Level Feature: Summary

- We have introduced different ways to obtain node features.
- They can be categorized as:
  - Importance-based features:
    - Node degree
    - Different node centrality measures
  - Structure-based features:
    - Node degree
    - Clustering coefficient
    - Graphlet count vector

# Node-Level Feature: Summary

- **Importance-based features**: capture the importance of a node in a graph
  - Node degree:
    - Simply counts the number of neighboring nodes
  - Node centrality:
    - Models **importance of neighboring nodes** in a graph
    - Different modeling choices: eigenvector centrality, betweenness centrality, closeness centrality
- Useful for predicting influential nodes in a graph
  - **Example**: predicting celebrity users in a social network

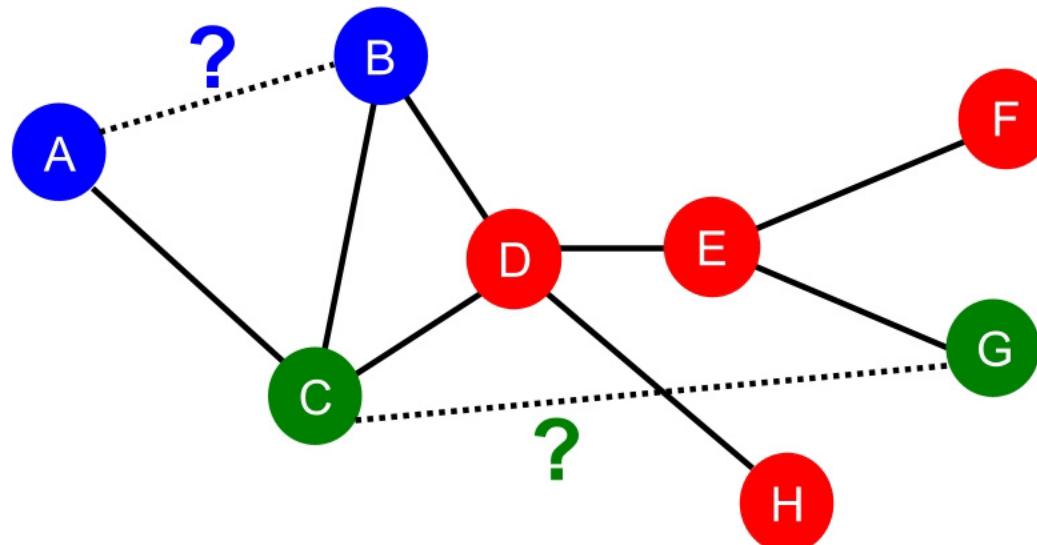
# Node-Level Feature: Summary

- **Structure-based features:** Capture topological properties of local neighborhood around a node.
  - **Node degree:**
    - Counts the number of neighboring nodes
  - **Clustering coefficient:**
    - Measures how connected neighboring nodes are
  - **Graphlet degree vector:**
    - Counts the occurrences of different graphlets
- **Useful for predicting a particular role a node plays in a graph:**
  - **Example:** Predicting protein functionality in a protein-protein interaction network.

# Link Prediction Task and Features

# Link-Level Prediction Task: Recap

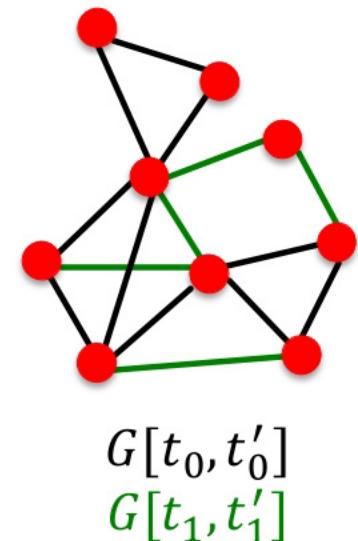
- The task is to predict **new links** based on the existing links.
- At test time, node pairs (with no existing links) are ranked, and top  $K$  node pairs are predicted.
- **The key is to design features for a pair of nodes.**



# Link Prediction as a Task

Two formulations of the link prediction task:

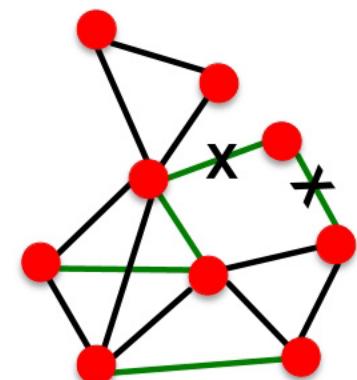
- 1) Links missing at random:
  - Remove a random set of links and then aim to predict them
- 2) Links over time:
  - Given  $G[t_0, t'_0]$  a graph defined by edges up to time  $t'_0$ , **output a ranked list  $L$**  of edges (not in  $G[t_0, t'_0]$ ) that are predicted to appear in time  $G[t_1, t'_1]$
  - **Evaluation:**
    - $n = |E_{new}|$ : # new edges that appear during the test period  $[t_1, t'_1]$
    - Take top  $n$  elements of  $L$  and count correct edges



# Link Prediction via Proximity

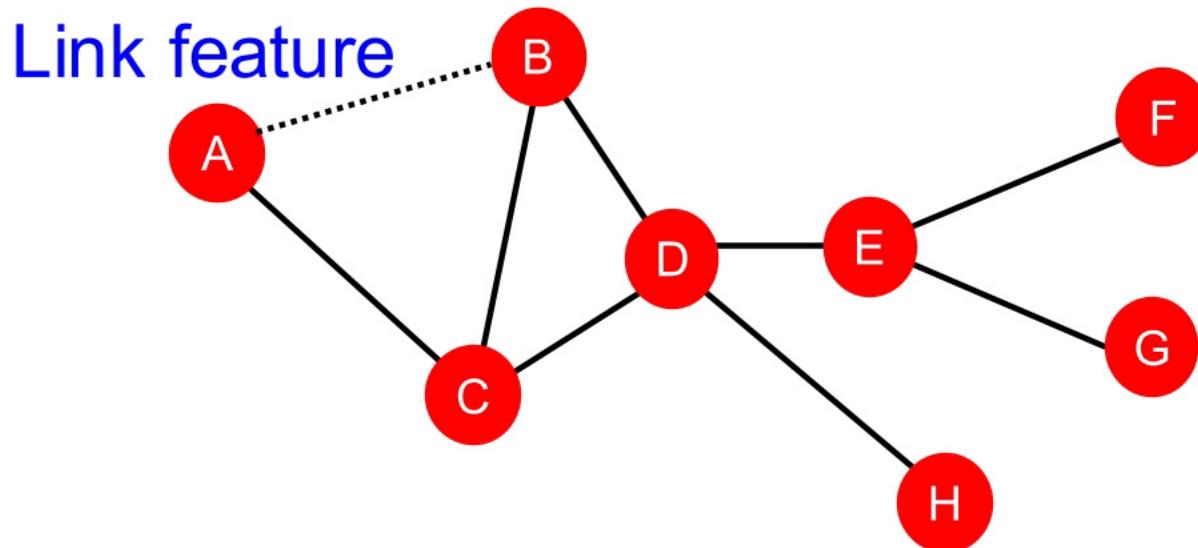
## ■ Methodology:

- For each pair of nodes  $(x,y)$  compute score  $c(x,y)$ 
  - For example,  $c(x,y)$  could be the # of common neighbors of  $x$  and  $y$
- Sort pairs  $(x,y)$  by the decreasing score  $c(x,y)$
- **Predict top  $n$  pairs as new links**
- **See which of these links actually appear in  $G[t_1, t'_1]$**



# Link-Level Features: Overview

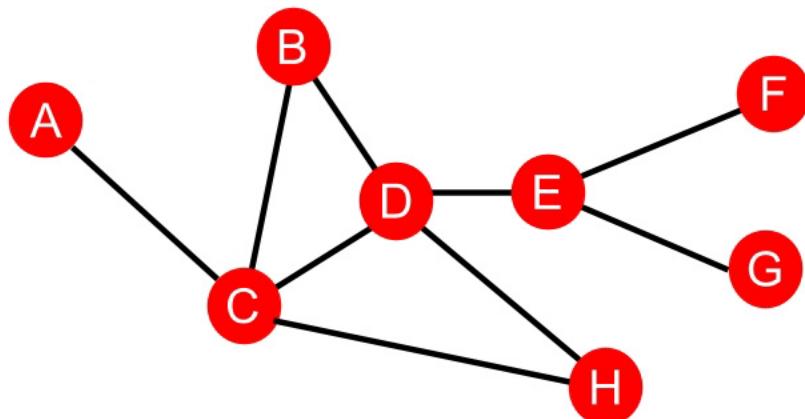
- Distance-based feature
- Local neighborhood overlap
- Global neighborhood overlap



# Distance-Based Features

## Shortest-path distance between two nodes

- Example:



$$S_{BH} = S_{BE} = S_{AB} = 2$$
$$S_{BG} = S_{BF} = 3$$

- However, this does not capture the degree of neighborhood overlap:
  - Node pair  $(B, H)$  has 2 shared neighboring nodes, while pairs  $(B, E)$  and  $(A, B)$  only have 1 such node.

# Local Neighborhood Overlap

Captures # neighboring nodes shared between two nodes  $v_1$  and  $v_2$ :

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

- Example:  $|N(A) \cap N(B)| = |\{C\}| = 1$

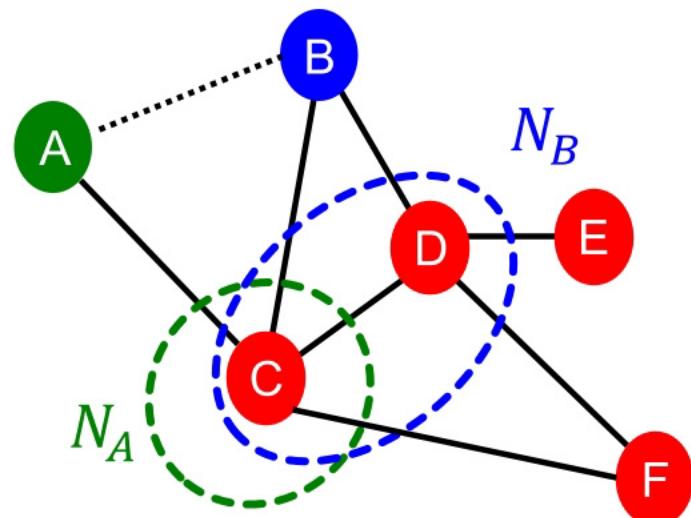
- Jaccard's coefficient:  $\frac{|N(v_1) \cap N(v_2)|}{|N(v_1) \cup N(v_2)|}$

- Example:  $\frac{|N(A) \cap N(B)|}{|N(A) \cup N(B)|} = \frac{|\{C\}|}{|\{A, B, C, D\}|} = \frac{1}{2}$

- Adamic-Adar index:

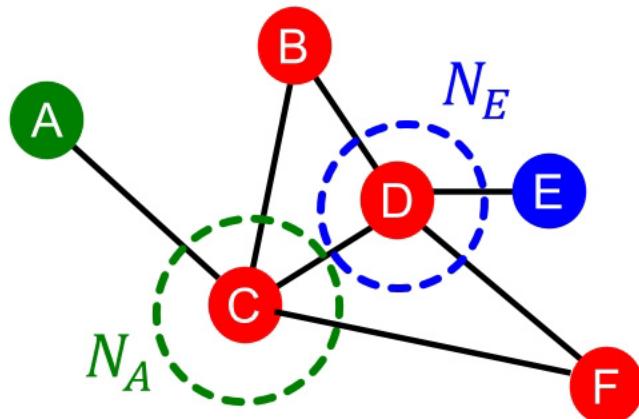
$$\sum_{u \in N(v_1) \cap N(v_2)} \frac{1}{\log(k_u)}$$

- Example:  $\frac{1}{\log(k_C)} = \frac{1}{\log 4}$



# Global Neighborhood Overlap

- **Limitation of local neighborhood features:**
  - Metric is always zero if the two nodes do not have any neighbors in common.



$$N_A \cap N_E = \phi$$
$$|N_A \cap N_E| = 0$$

- However, the two nodes may still potentially be connected in the future.
- **Global neighborhood overlap metrics** resolve the limitation by considering the entire graph.

# Global Neighborhood Overlap

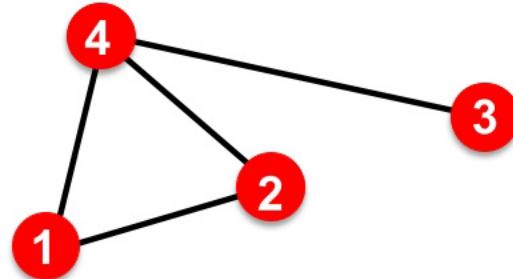
- **Katz index:** count the number of walks of all lengths between a given pair of nodes.
- **Q: How to compute #walks between two nodes?**
- Use **powers of the graph adjacency matrix!**

# Intuition: Powers of Adj Matrices

## ■ Computing #walks between two nodes

- Recall:  $A_{uv} = 1$  if  $u \in N(v)$
- Let  $P_{uv}^{(K)} = \# \text{walks of length } K \text{ between } u \text{ and } v$
- We will show  $P^{(K)} = A^k$
- $P_{uv}^{(1)} = \# \text{walks of length 1 (direct neighborhood)} \text{ between } u \text{ and } v = A_{uv}$

$$P_{12}^{(1)} = A_{12}$$



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

# Intuition: Powers of Adj Matrices

- How to compute  $P_{uv}^{(2)}$  ?
  - Step 1: Compute #walks of length 1 between each of  $u$ 's neighbor and  $v$
  - Step 2: Sum up these #walks across  $u$ 's neighbors
  - $P_{uv}^{(2)} = \sum_i A_{ui} * P_{iv}^{(1)} = \sum_i A_{ui} * A_{iv} = A_{uv}^2$

Node 1's neighbors      #walks of length 1 between  
Node 1's neighbors and Node 2       $P_{12}^{(2)} = A_{12}^2$

$$A^2 = \begin{matrix} \text{Power of} \\ \text{adjacency} \end{matrix} \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 3 \end{pmatrix}$$

Diagram illustrating the computation of  $A^2$ . The first matrix shows Node 1's neighbors (highlighted in blue) as columns 1 and 4. The second matrix shows the #walks of length 1 between Node 1's neighbors and Node 2 (highlighted in green). The result is  $A_{12}^2$ , where the 12th entry is highlighted in red.

# Global Neighborhood Overlap

- **Katz index:** count the number of walks of all lengths between a pair of nodes.
- How to compute #walks between two nodes?
- Use **adjacency matrix powers!**
  - $A_{uv}$  specifies #walks of length 1 (direct neighborhood) between  $u$  and  $v$ .
  - $A_{uv}^2$  specifies #walks of **length 2** (neighbor of neighbor) between  $u$  and  $v$ .
  - And,  $A_{uv}^l$  specifies #walks of **length  $l$** .

# Global Neighborhood Overlap

- **Katz index** between  $v_1$  and  $v_2$  is calculated as

**Sum over all walk lengths**

$$S_{v_1 v_2} = \sum_{l=1}^{\infty} \beta^l A_{v_1 v_2}^l$$

#walks of length  $l$   
between  $v_1$  and  $v_2$

$\beta$ : discount factor

- Katz index matrix is computed in closed-form:

$$\begin{aligned} S &= \sum_{i=1}^{\infty} \beta^i A^i = \underbrace{(\mathbf{I} - \beta \mathbf{A})^{-1}}_{=} - \mathbf{I}, \\ &= \sum_{i=0}^{\infty} \beta^i A^i \\ &\text{by geometric series of matrices} \end{aligned}$$

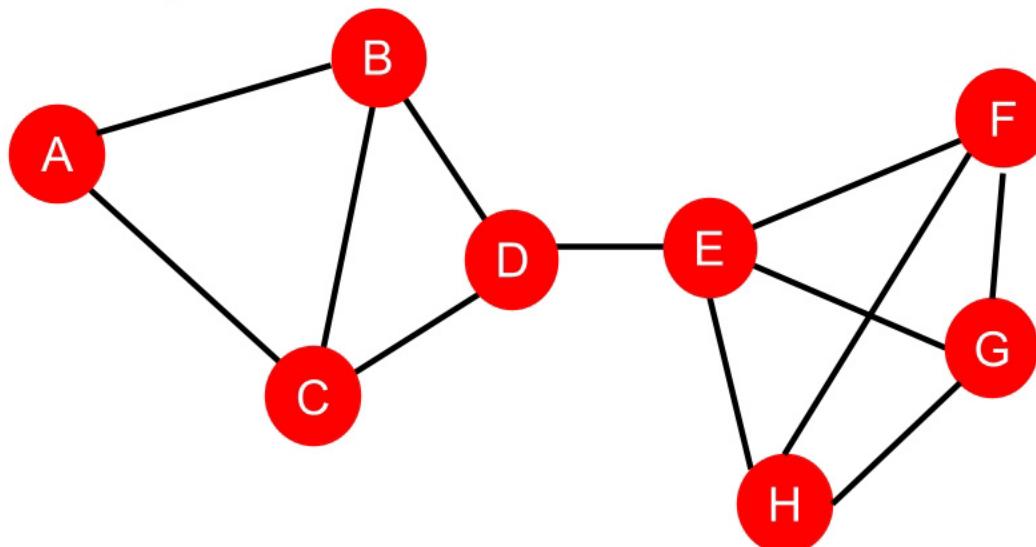
# Link-Level Features: Summary

- **Distance-based features:**
  - Uses the shortest path length between two nodes but does not capture how neighborhood overlaps.
- **Local neighborhood overlap:**
  - Captures how many neighboring nodes are shared by two nodes.
  - Becomes zero when no neighbor nodes are shared.
- **Global neighborhood overlap:**
  - Uses global graph structure to score two nodes.
  - Katz index counts #walks of all lengths between two nodes.

# **Graph-Level Features and Graph Kernels**

# Graph-Level Features

- **Goal:** We want features that characterize the structure of an entire graph.
- **For example:**



# Background: Kernel Methods

- **Kernel methods** are widely-used for traditional ML for graph-level prediction.
- **Idea: Design kernels instead of feature vectors.**
- **A quick introduction to Kernels:**
  - Kernel  $K(G, G') \in \mathbb{R}$  measures similarity b/w data
  - Kernel matrix  $\mathbf{K} = (K(G, G'))_{G, G'}$ , must always be positive semidefinite (i.e., has positive eigenvalues)
  - There exists a feature representation  $\phi(\cdot)$  such that  $K(G, G') = \phi(G)^T \phi(G')$
  - Once the kernel is defined, off-the-shelf ML model, such as **kernel SVM**, can be used to make predictions.

# Graph-Level Features: Overview

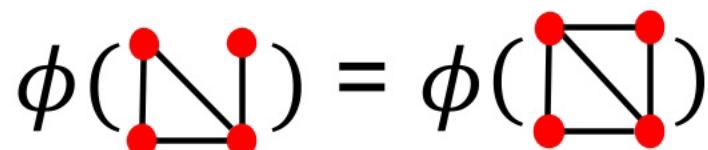
- **Graph Kernels:** Measure similarity between two graphs:
  - Graphlet Kernel [1]
  - Weisfeiler-Lehman Kernel [2]
  - Other kernels are also proposed in the literature (beyond the scope of this lecture)
    - Random-walk kernel
    - Shortest-path graph kernel
    - And many more...

[1] Shervashidze, Nino, et al. "Efficient graphlet kernels for large graph comparison." Artificial Intelligence and Statistics. 2009.

[2] Shervashidze, Nino, et al. "Weisfeiler-lehman graph kernels." Journal of Machine Learning Research 12.9 (2011).

# Graph Kernel: Key Idea

- **Goal:** Design graph feature vector  $\phi(G)$
- **Key idea:** Bag-of-Words (BoW) for a graph
  - **Recall:** BoW simply uses the word counts as features for documents (no ordering considered).
  - Naïve extension to a graph: **Regard nodes as words.**
  - Since both graphs have **4 red nodes**, we get the same feature vector for two different graphs...

$$\phi(\text{graph 1}) = \phi(\text{graph 2})$$
The diagram shows two graphs side-by-side. Both graphs have four red circular nodes. The left graph has three nodes in a horizontal row, with the fourth node connected to the first and second nodes. The right graph has four nodes in a square arrangement, with all four nodes connected to each other. Both graphs represent the same set of 4 red nodes, which is the key point being illustrated.

# Graph Kernel: Key Idea

What if we use Bag of node degrees?

Deg1: ● Deg2: ● Deg3: ●

$$\phi(\text{graph}) = \text{count}(\text{graph}) = [1, 2, 1]$$

Obtains different features  
for different graphs!

$$\phi(\text{graph}) = \text{count}(\text{graph}) = [0, 2, 2]$$

- Both Graphlet Kernel and Weisfeiler-Lehman (WL) Kernel use **Bag-of-\*** representation of graph, where \* is more sophisticated than node degrees!

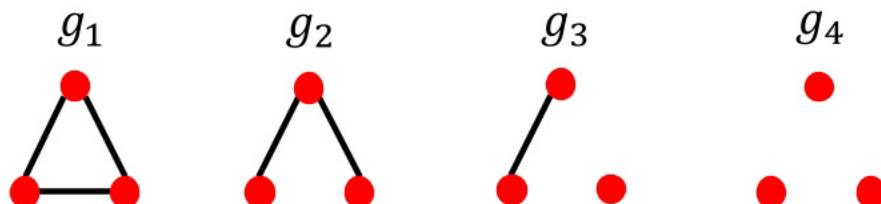
# Graphlet Features

- **Key idea:** Count the number of different graphlets in a graph.
- **Note:** Definition of graphlets here is slightly different from node-level features.
- The two differences are:
  - Nodes in graphlets here do **not need to be connected** (allows for isolated nodes)
  - The graphlets here are not rooted.
  - Examples in the next slide illustrate this.

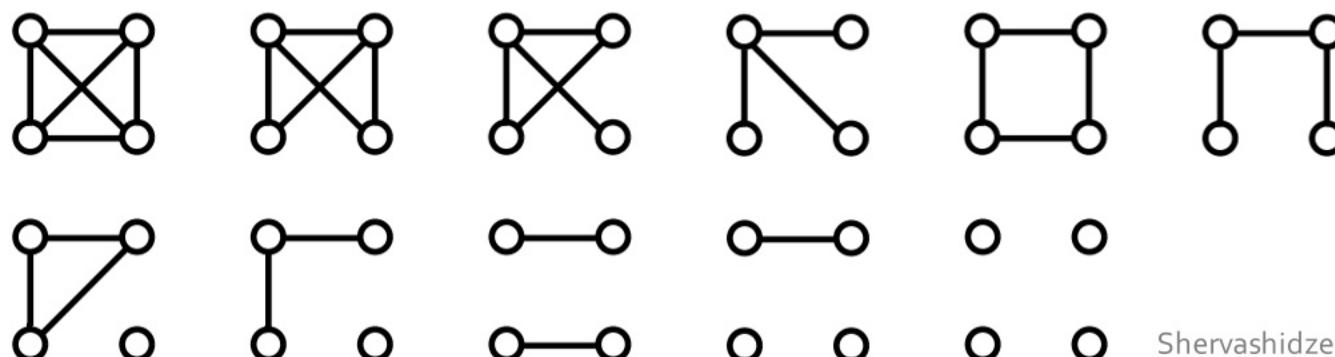
# Graphlet Features

Let  $\mathcal{G}_k = (g_1, g_2, \dots, g_{n_k})$  be a list of graphlets of size  $k$ .

- For  $k = 3$ , there are 4 graphlets.



- For  $k = 4$ , there are 11 graphlets.



Shervashidze et al., AISTATS 2011

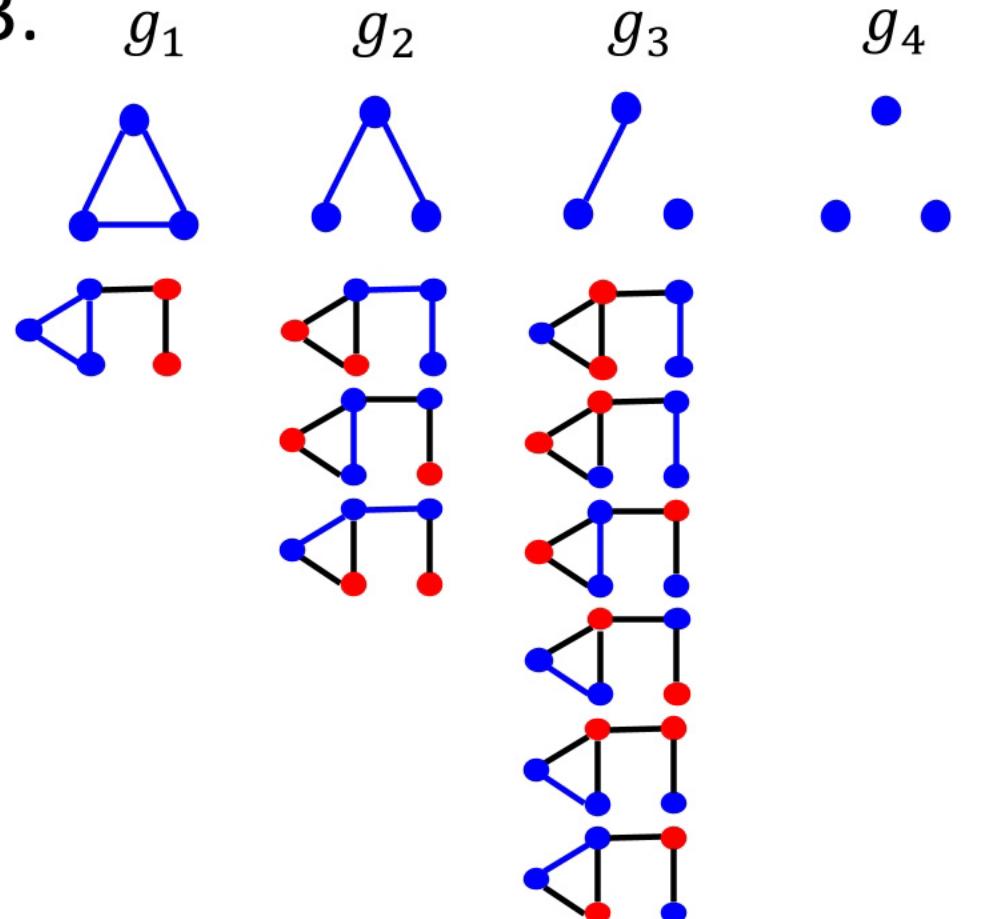
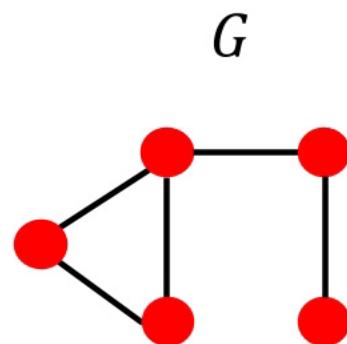
# Graphlet Features

- Given graph  $G$ , and a graphlet list  $\mathcal{G}_k = (g_1, g_2, \dots, g_{n_k})$ , define the graphlet count vector  $f_G \in \mathbb{R}^{n_k}$  as

$$(f_G)_i = \#(g_i \subseteq G) \text{ for } i = 1, 2, \dots, n_k.$$

# Graphlet Features

- Example for  $k = 3$ .



$$\mathbf{f}_G = (1, 3, 6, 0)^T$$

# Graphlet Kernel

- Given two graphs,  $G$  and  $G'$ , graphlet kernel is computed as

$$K(G, G') = \mathbf{f}_G^T \mathbf{f}_{G'}$$

- Problem:** if  $G$  and  $G'$  have different sizes, that will greatly skew the value.
- Solution:** normalize each feature vector

$$\mathbf{h}_G = \frac{\mathbf{f}_G}{\text{Sum}(\mathbf{f}_G)} \quad K(G, G') = \mathbf{h}_G^T \mathbf{h}_{G'}$$

# Graphlet Kernel

**Limitations:** Counting graphlets is **expensive!**

- Counting size- $k$  graphlets for a graph with size  $n$  by enumeration takes  $n^k$ .
- This is unavoidable in the worst-case since **subgraph isomorphism test** (judging whether a graph is a subgraph of another graph) is **NP-hard**.
- If a graph's node degree is bounded by  $d$ , an  $O(nd^{k-1})$  algorithm exists to count all the graphlets of size  $k$ .

**Can we design a more efficient graph kernel?**

# Weisfeiler-Lehman Kernel

- **Goal:** Design an efficient graph feature descriptor  $\phi(G)$
- **Idea:** Use neighborhood structure to iteratively enrich node vocabulary.
  - Generalized version of **Bag of node degrees** since node degrees are one-hop neighborhood information.
- **Algorithm to achieve this:**

**Color refinement**

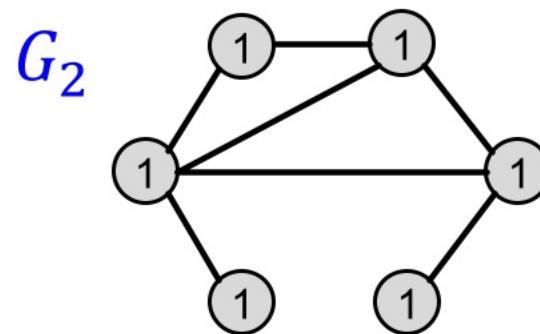
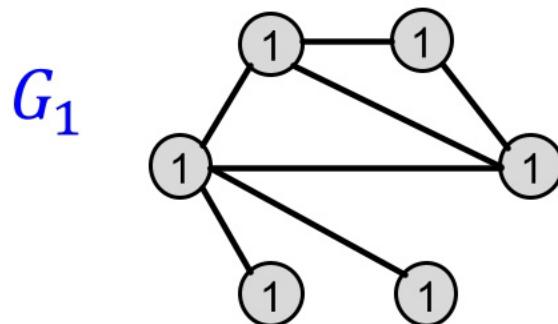
# Color Refinement

- **Given:** A graph  $G$  with a set of nodes  $V$ .
  - Assign an initial color  $c^{(0)}(v)$  to each node  $v$ .
  - Iteratively refine node colors by
$$c^{(k+1)}(v) = \text{HASH} \left( \left\{ c^{(k)}(v), \left\{ c^{(k)}(u) \right\}_{u \in N(v)} \right\} \right),$$
where **HASH** maps different inputs to different colors.
- After  $K$  steps of color refinement,  $c^{(K)}(v)$  summarizes the structure of  $K$ -hop neighborhood

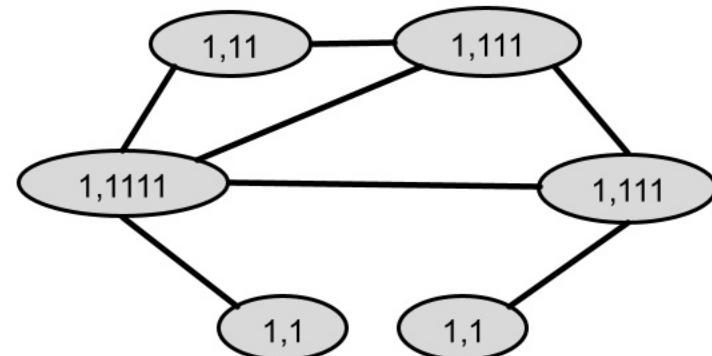
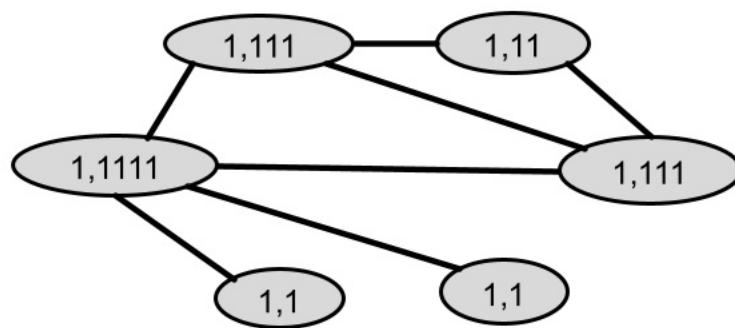
# Color Refinement (1)

## Example of color refinement given two graphs

- Assign initial colors



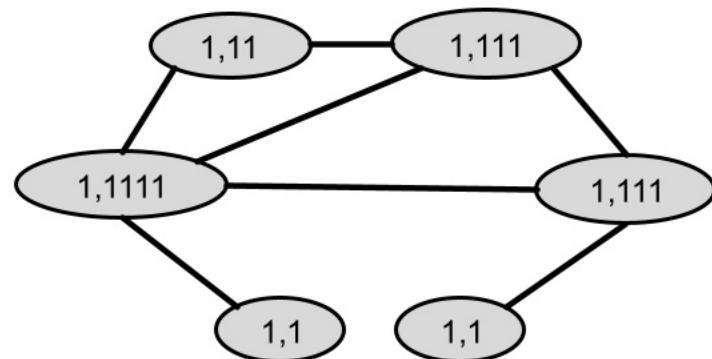
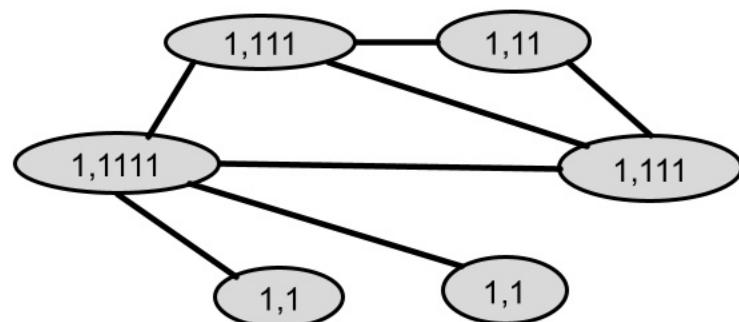
- Aggregate neighboring colors



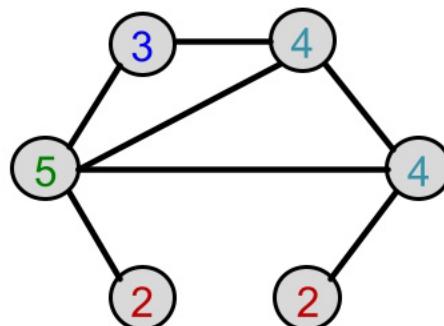
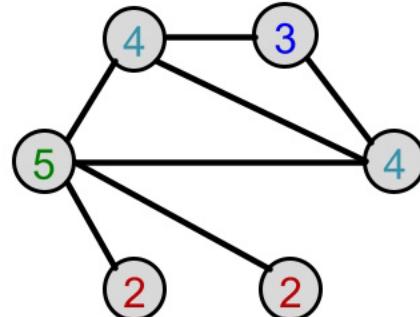
# Color Refinement (2)

## Example of color refinement given two graphs

- Aggregated colors



- Hash aggregated colors



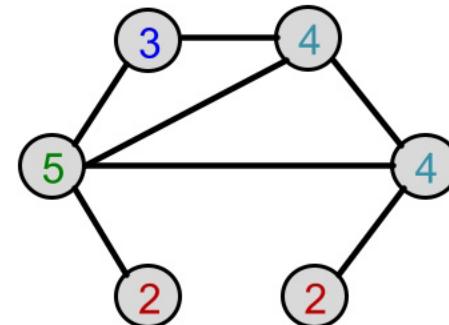
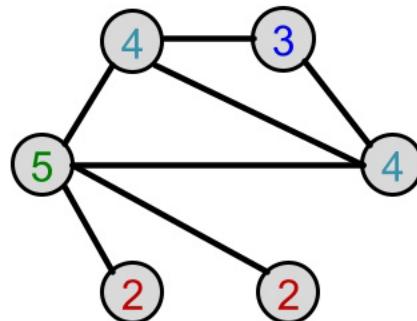
Hash table

1,1	-->	2
1,11	-->	3
1,111	-->	4
1,1111	-->	5

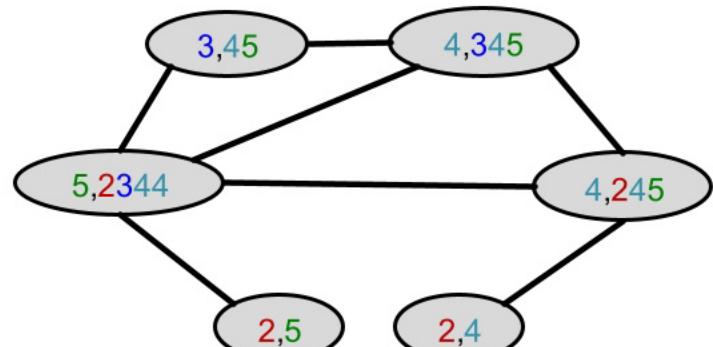
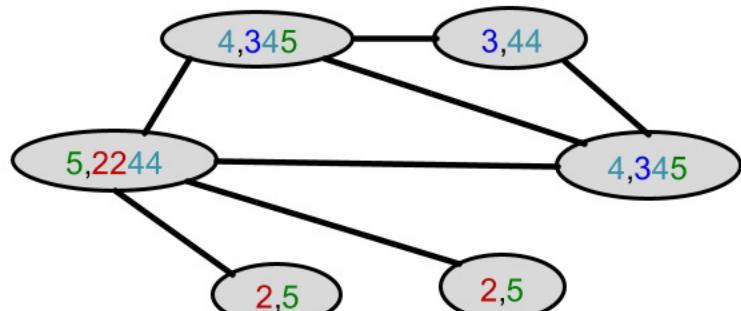
# Color Refinement (3)

Example of color refinement given two graphs

- Aggregated colors



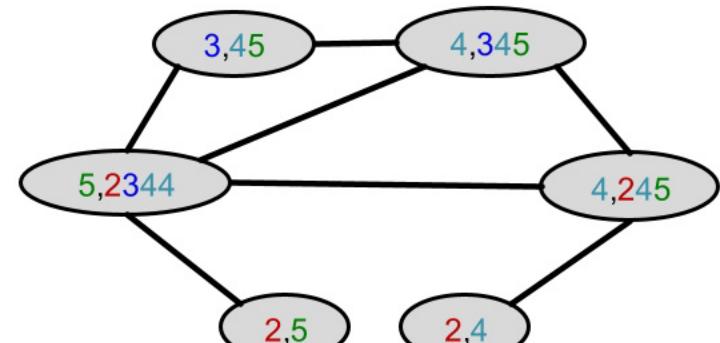
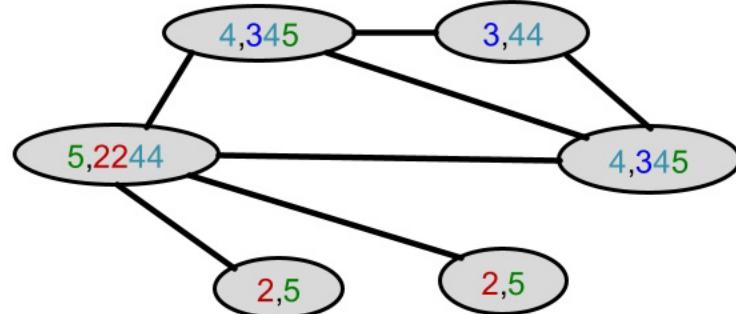
- Hash aggregated colors



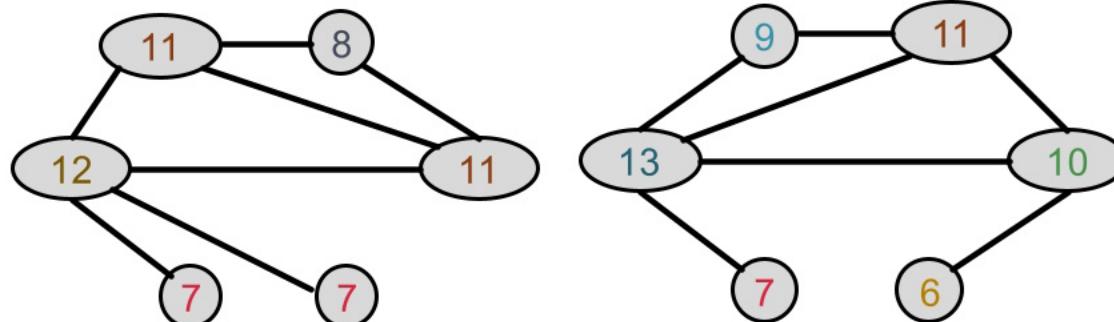
# Color Refinement (4)

## Example of color refinement given two graphs

- Aggregated colors



- Hash aggregated colors

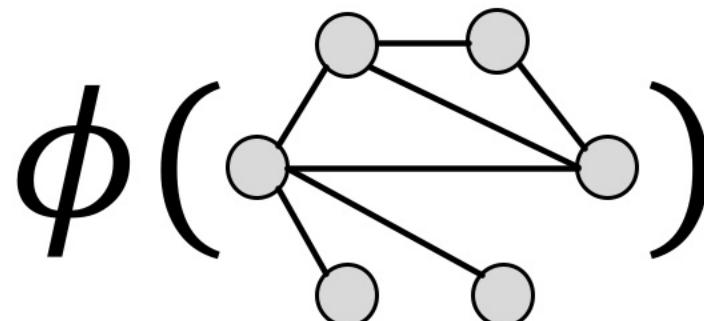


Hash table

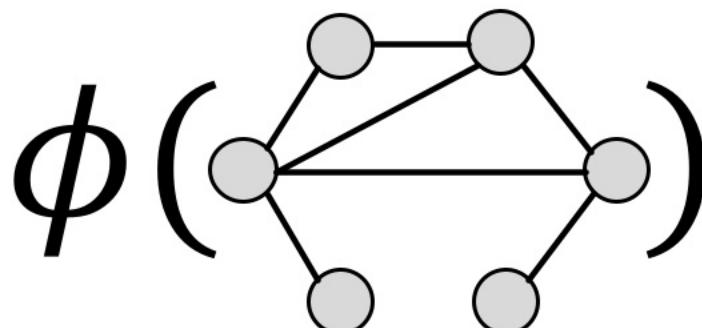
2,4	-->	6
2,5	-->	7
3,44	-->	8
3,45	-->	9
4,245	-->	10
4,345	-->	11
5,2244	-->	12
5,2344	-->	13

# Weisfeiler-Lehman Graph Features

After color refinement, WL kernel counts number of nodes with a given color.



Colors  
1,2,3,4,5,6,7,8,9,10,11,12,13  
= [6,2,1,2,1,0,2,1,0,0, 0, 2, 1]  
Counts



1,2,3,4,5,6,7,8,9,10,11,12,13  
= [6,2,1,2,1,1,1,0,1,1, 1, 0, 1]

# Weisfeiler-Lehman Kernel

The WL kernel value is computed by the inner product of the color count vectors:

$$\begin{aligned} K(&\text{graph}_1, \text{graph}_2) \\ &= \phi(\text{graph}_1)^T \phi(\text{graph}_2) \\ &= 49 \end{aligned}$$

# Weisfeiler-Lehman Kernel

- WL kernel is **computationally efficient**
  - The time complexity for color refinement at each step is linear in #(edges), since it involves aggregating neighboring colors.
- When computing a kernel value, only colors appeared in the two graphs need to be tracked.
  - Thus, #(colors) is at most the total number of nodes.
- Counting colors takes linear-time w.r.t. #(nodes).
- In total, time complexity is **linear in #(edges)**.

# Graph-Level Features: Summary

- **Graphlet Kernel**

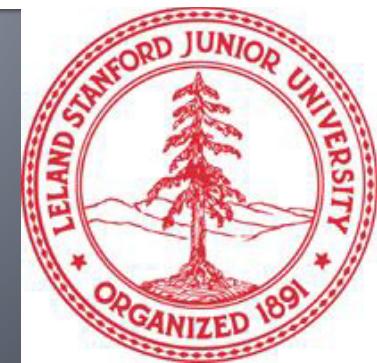
- Graph is represented as **Bag-of-graphlets**
- **Computationally expensive**

- **Weisfeiler-Lehman Kernel**

- Apply  $K$ -step color refinement algorithm to enrich node colors
  - Different colors capture different  $K$ -hop neighborhood structures
- Graph is represented as **Bag-of-colors**
- **Computationally efficient**
- Closely related to Graph Neural Networks (as we will see!)

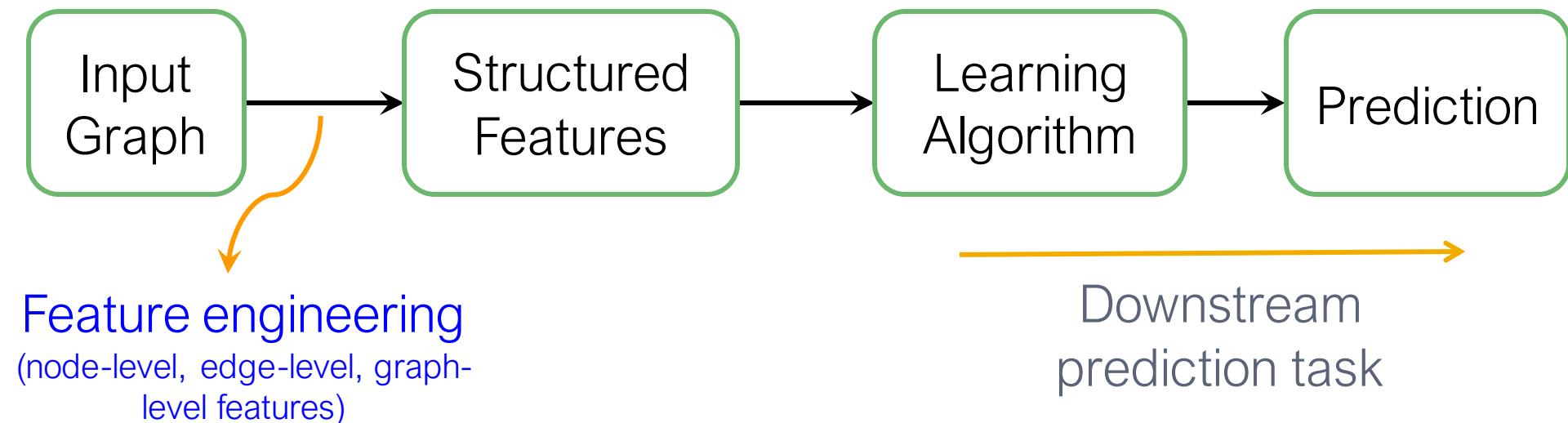
# Stanford CS224W: Node Embeddings

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



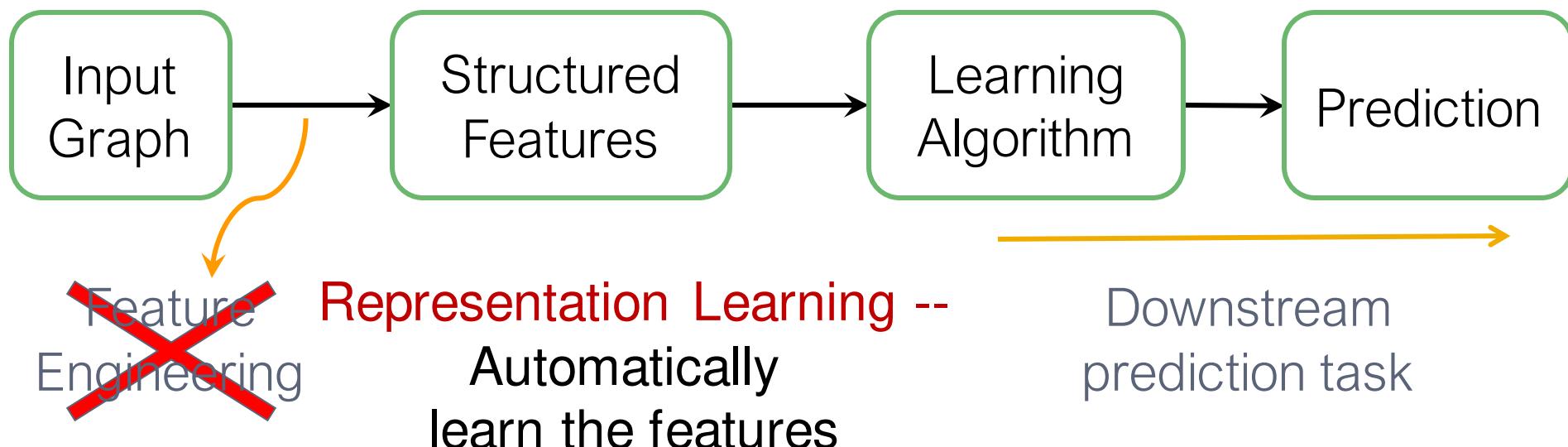
# Recap: Traditional ML for Graphs

Given an input graph, extract node, link and graph-level features, learn a model (SVM, neural network, etc.) that maps features to labels.



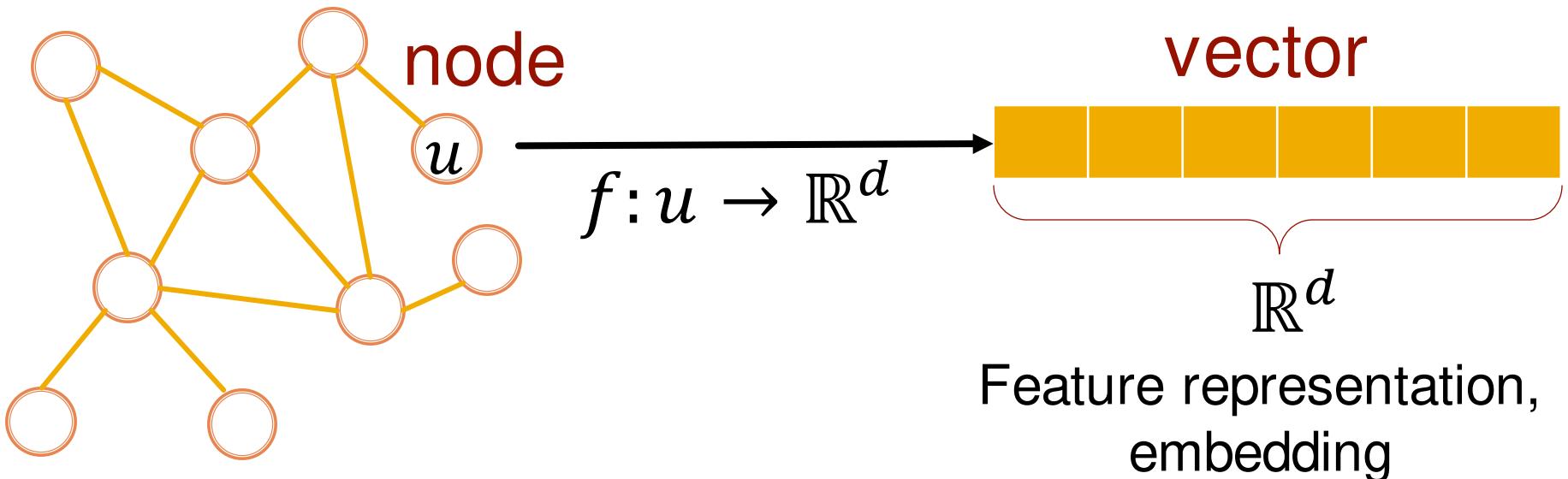
# Graph Representation Learning

**Graph Representation Learning alleviates the need to do feature engineering **every single time.****



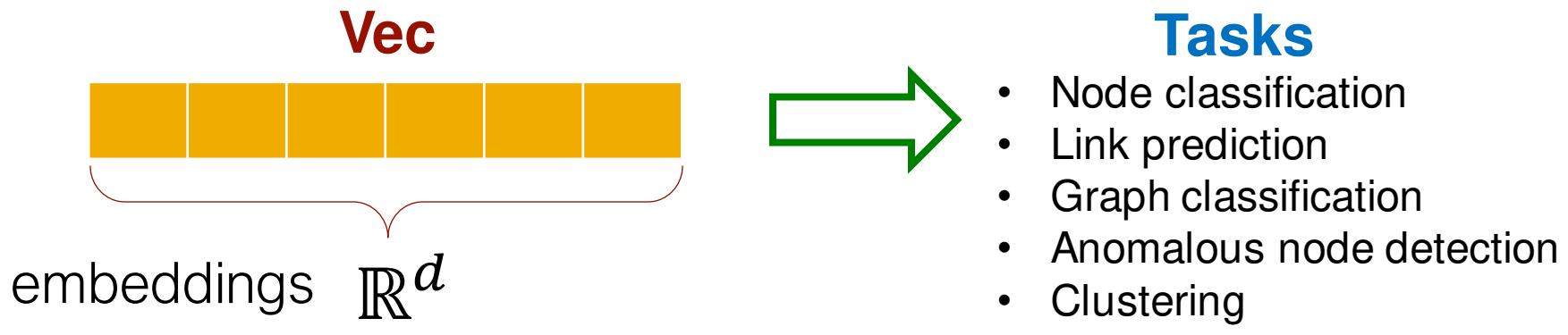
# Graph Representation Learning

**Goal:** Efficient task-independent feature learning for machine learning with graphs!



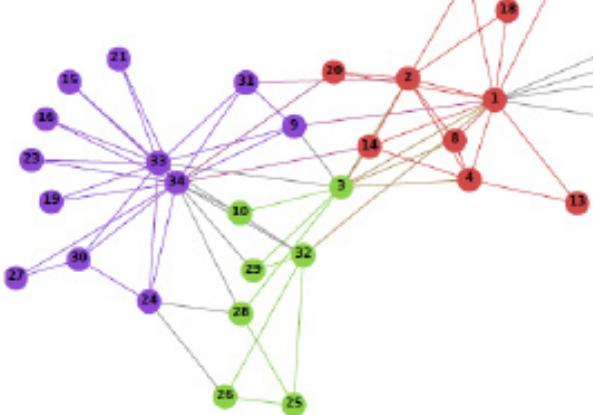
# Why Embedding?

- **Task: Map nodes into an embedding space**
  - Similarity of embeddings between nodes indicates their similarity in the network. For example:
    - Both nodes are close to each other (connected by an edge)
  - Encode network information
  - Potentially used for many downstream predictions

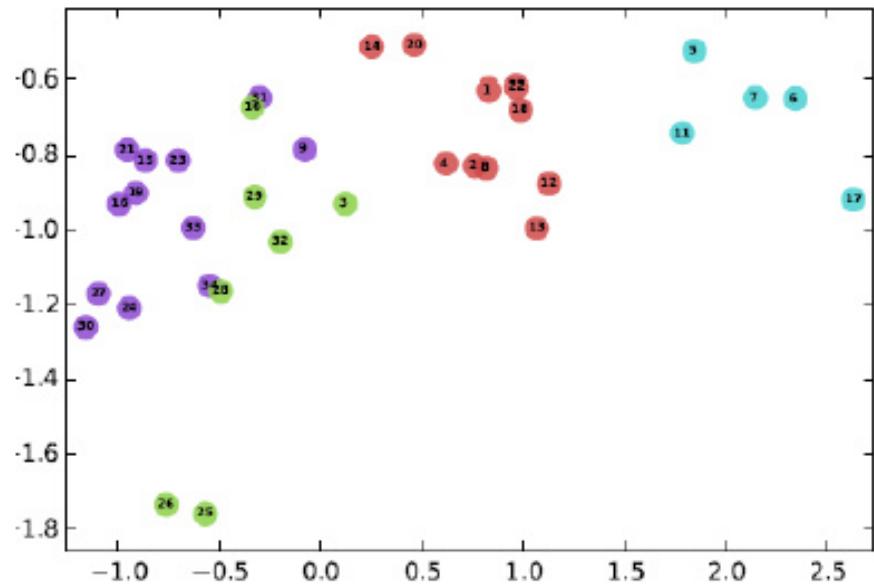


# Example Node Embedding

- 2D embedding of nodes of the Zachary's Karate Club network:



Input

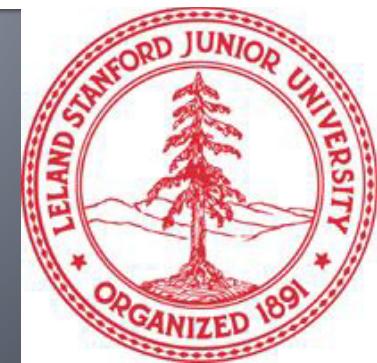


Output

Image from: [Perozzi et al.](#). DeepWalk: Online Learning of Social Representations. *KDD 2014*.

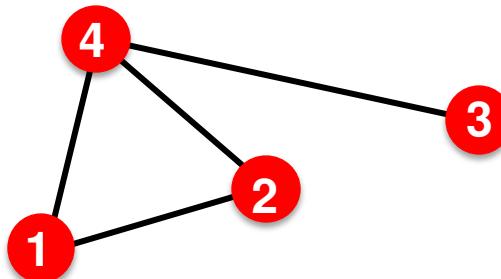
# **Stanford CS224W:** **Node Embeddings:** **Encoder and Decoder**

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



# Setup

- Assume we have a graph  $G$ :
  - $V$  is the vertex set.
  - $A$  is the adjacency matrix (assume binary).
  - **For simplicity: No node features or extra information is used**

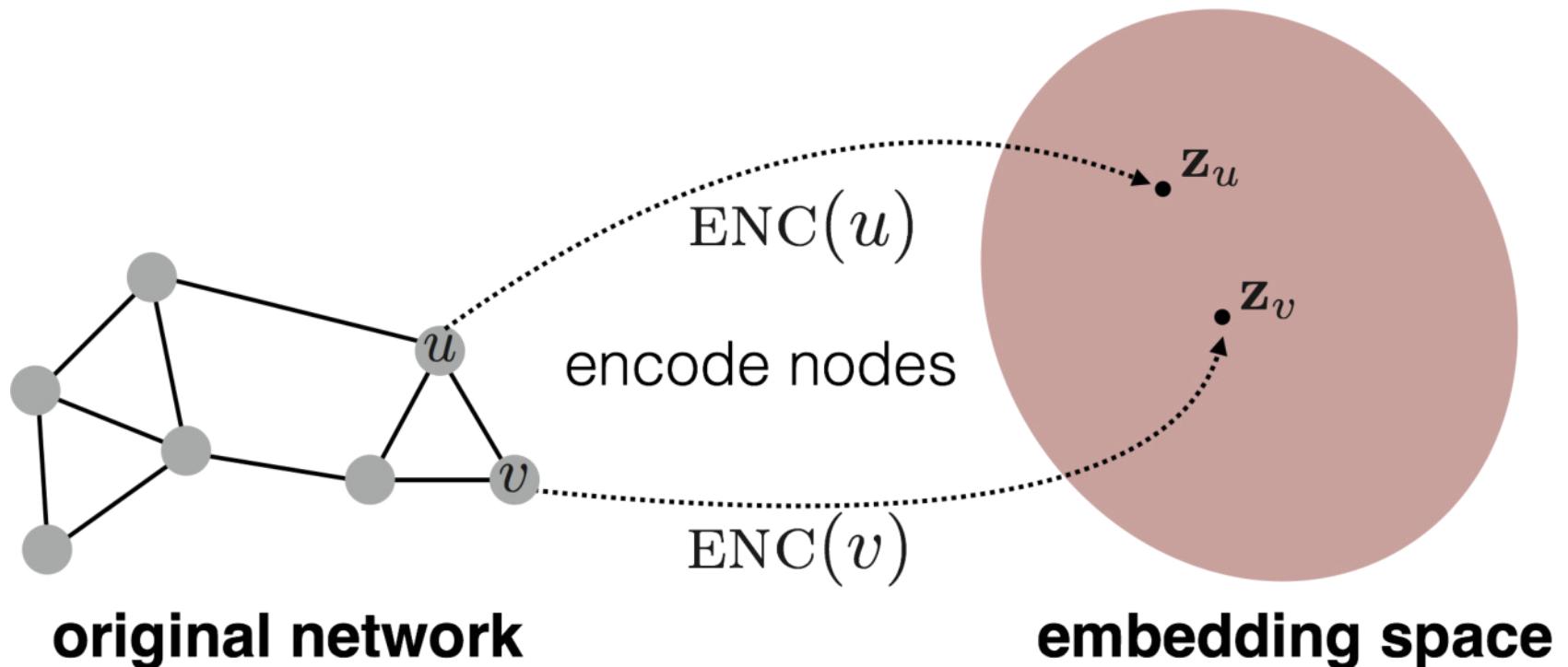


$V: \{1, 2, 3, 4\}$

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

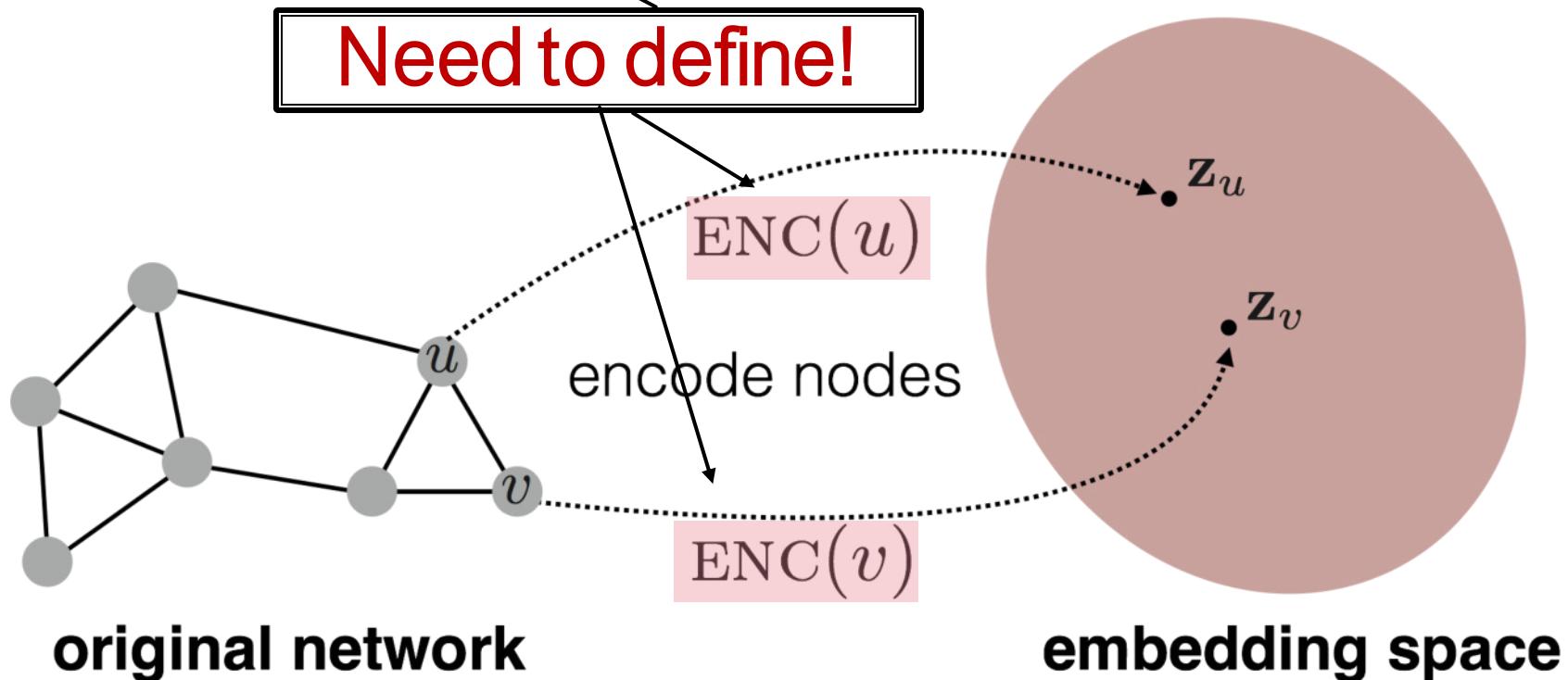
# Embedding Nodes

- Goal is to encode nodes so that **similarity in the embedding space (e.g., dot product)** approximates **similarity in the graph**



# Embedding Nodes

Goal:  $\text{similarity}(u, v) \approx \mathbf{z}_v^T \mathbf{z}_u$   
in the original network      Similarity of the embedding



# Learning Node Embeddings

1. **Encoder** maps from nodes to embeddings
2. Define a node similarity function (i.e., a measure of similarity in the original network)
3. **Decoder DEC** maps from embeddings to the similarity score
4. Optimize the parameters of the encoder so that:

$$\text{DEC}(\mathbf{z}_v^T \mathbf{z}_u)$$

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

in the original network

Similarity of the embedding

# Two Key Components

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

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

node in the input graph

$d$ -dimensional embedding

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

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

Similarity of  $u$  and  $v$  in the original network

dot product between node embeddings

**Decoder**

# “Shallow” Encoding

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

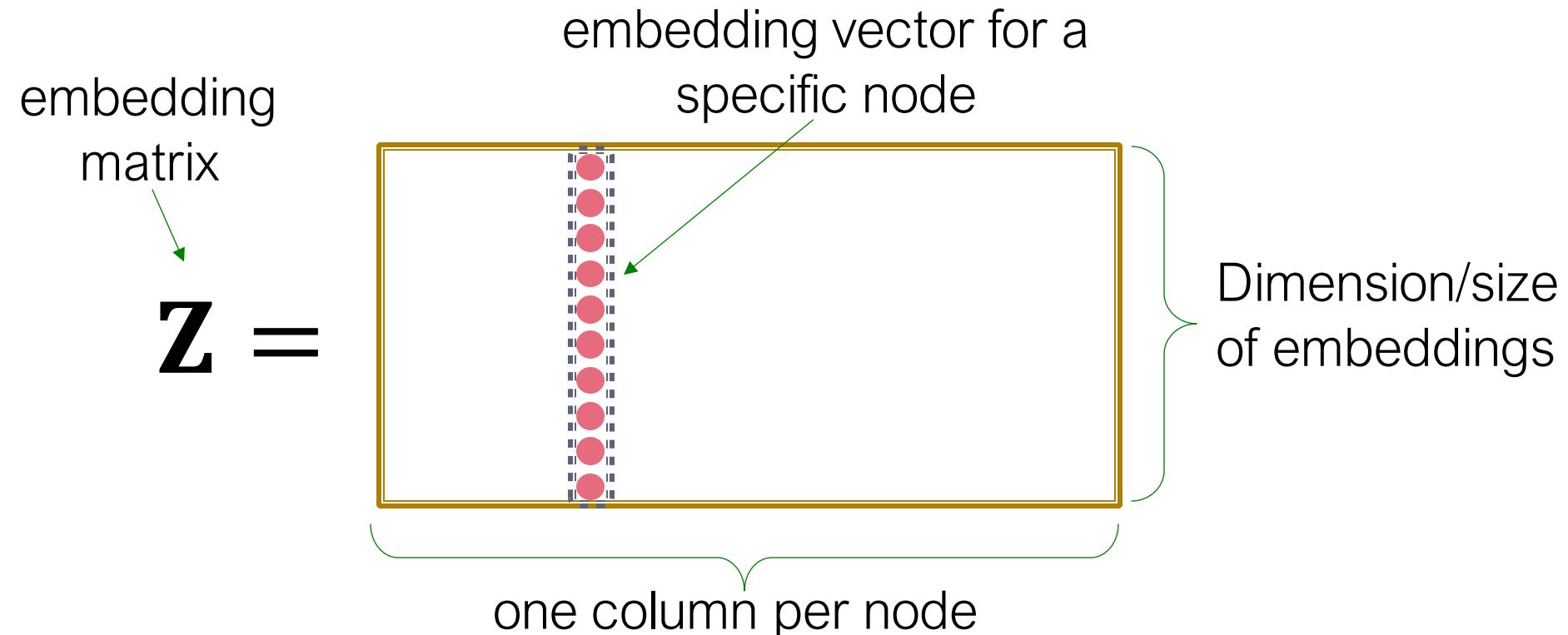
$$\text{ENC}(v) = z_v = Z \cdot v$$

$Z \in \mathbb{R}^{d \times |\mathcal{V}|}$  matrix, each column is a node embedding [what we learn / optimize]

$v \in \mathbb{I}^{|\mathcal{V}|}$  indicator vector, all zeroes except a one in column indicating node  $v$

# “Shallow” Encoding

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



# “Shallow” Encoding

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

**Each node is assigned a unique  
embedding vector**

(i.e., we directly optimize  
the embedding of each node)

Many methods: DeepWalk, node2vec

# Framework Summary

## ■ Encoder + Decoder Framework

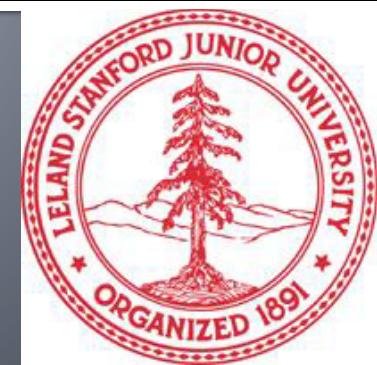
- Shallow encoder: embedding lookup
- Parameters to optimize:  $\mathbf{Z}$  which contains node embeddings  $\mathbf{z}_u$  for all nodes  $u \in V$
- We will cover deep encoders (GNNs) in Lecture 6
- **Decoder:** based on node similarity.
- **Objective:** maximize  $\mathbf{z}_v^T \mathbf{z}_u$  for 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”?
- We will now learn node similarity definition that uses  **random walks**, and how to optimize embeddings for such a similarity measure.

# **Stanford CS224W: Random Walk Approaches for Node Embeddings**

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



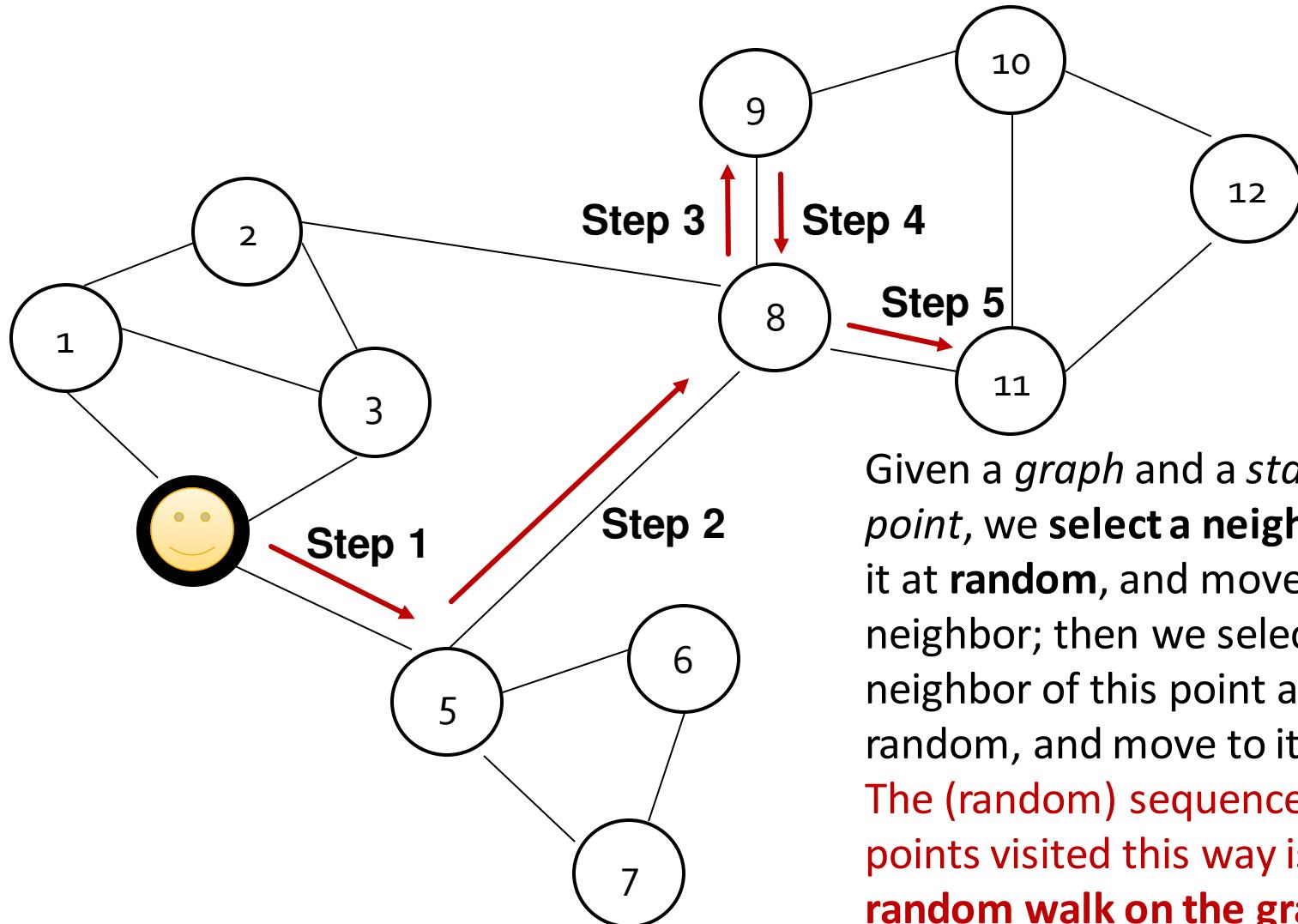
# Notation

- **Vector  $\mathbf{z}_u$ :**
    - The embedding of node  $u$  (what we aim to find).
  - **Probability  $P(v | \mathbf{z}_u)$** :  Our model prediction based on  $\mathbf{z}_u$ 
    - The **(predicted) probability** of visiting node  $v$  on random walks starting from node  $u$ .
- 

Non-linear functions used to produce predicted **probabilities**

- **Softmax** function:
  - Turns vector of  $K$  real values (model predictions) into  $K$  probabilities that sum to 1:  $\sigma(\mathbf{z})[i] = \frac{e^{\mathbf{z}[i]}}{\sum_{j=1}^K e^{\mathbf{z}[j]}}$
- **Sigmoid** function:
  - S-shaped function that turns real values into the range of  $(0, 1)$ .  
Written as  $S(x) = \frac{1}{1+e^{-x}}$ .

# 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**.

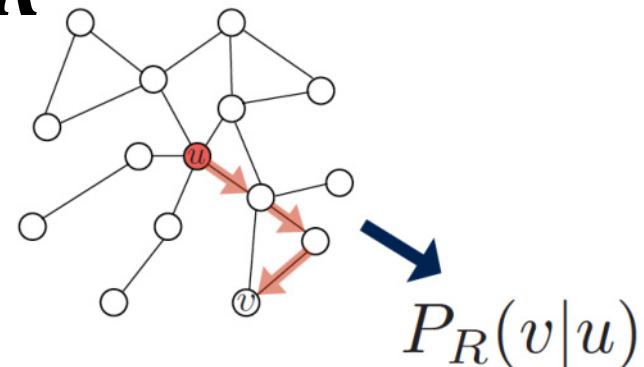
# Random-Walk Embeddings

$$\mathbf{z}_u^T \mathbf{z}_v \approx$$

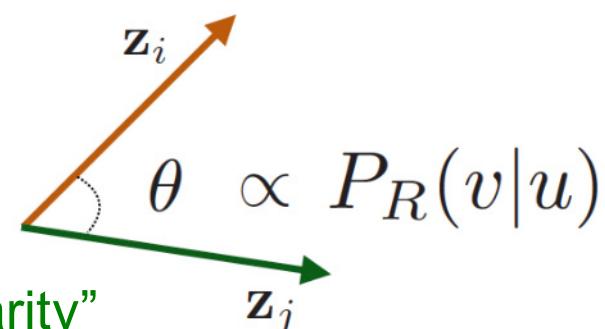
probability that  $u$  and  $v$  co-occur on a random walk over the graph

# 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:



Similarity in embedding space (Here:  
dot product= $\cos(\theta)$ ) encodes random walk “similarity”

# 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  $R$

# Feature Learning as Optimization

- Given  $G = (V, E)$ ,
- Our goal is to learn a mapping  $f: u \rightarrow \mathbb{R}^d$ :  
$$f(u) = \mathbf{z}_u$$
- Log-likelihood objective:

$$\max_f \sum_{u \in V} \log P(N_R(u) | \mathbf{z}_u)$$

- $N_R(u)$  is the neighborhood of node  $u$  by strategy  $R$
- Given node  $u$ , we want to learn feature representations that are predictive of the nodes in its random walk neighborhood  $N_R(u)$ .

# 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  $N_R(u)$ .**

$$\max_f \sum_{u \in V} \log P(N_R(u) | \mathbf{z}_u) \implies \text{Maximum likelihood objective}$$

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

# Random Walk Optimization

Equivalently,

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

- **Intuition:** Optimize embeddings  $\mathbf{z}_u$  to maximize the likelihood of random walk co-occurrences.
- **Parameterize  $P(v|\mathbf{z}_u)$  using softmax:**

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

**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:

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

sum over all nodes  $u$

sum over nodes  $v$  seen on random walks starting from  $u$

predicted probability of  $u$  and  $v$  co-occurring on random walk

Optimizing random walk embeddings =

Finding embeddings  $\mathbf{z}_u$  that minimize  $\mathcal{L}$

# Random Walk Optimization

But doing this naively is too expensive!

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



Nested sum over nodes gives  
 $O(|V|^2)$  complexity!

# Random Walk Optimization

But doing this naively is too expensive!

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

The normalization term from the softmax is the culprit... can we approximate it?

# Negative Sampling

## ■ Solution: Negative sampling

$$\log\left(\frac{\exp(\mathbf{z}_u^\top \mathbf{z}_v)}{\sum_{n \in V} \exp(\mathbf{z}_u^\top \mathbf{z}_n)}\right)$$

$$\approx \log\left(\sigma(\mathbf{z}_u^\top \mathbf{z}_v)\right) - \sum_{i=1}^k \log\left(\sigma(\mathbf{z}_u^\top \mathbf{z}_{n_i})\right), n_i \sim P_V$$

sigmoid function  
(makes each term a “probability” between 0 and 1)

random distribution over nodes

### Why is the approximation valid?

Technically, this is a different objective. But Negative Sampling is a form of Noise Contrastive Estimation (NCE) which approx. maximizes the log probability of softmax.

New formulation corresponds to using a logistic regression (sigmoid func.) to distinguish the target node  $v$  from nodes  $n_i$  sampled from background distribution  $P_v$ .

More at <https://arxiv.org/pdf/1402.3722.pdf>

Instead of normalizing w.r.t. all nodes, just normalize against  $k$  random “negative samples”  $n_i$

- Negative sampling allows for quick likelihood calculation.

# Negative Sampling

$$\log\left(\frac{\exp(\mathbf{z}_u^\top \mathbf{z}_v)}{\sum_{n \in V} \exp(\mathbf{z}_u^\top \mathbf{z}_n)}\right)$$

random distribution  
over nodes

$$\approx \log\left(\sigma(\mathbf{z}_u^\top \mathbf{z}_v)\right) - \sum_{i=1}^k \log\left(\sigma(\mathbf{z}_u^\top \mathbf{z}_{n_i})\right), n_i \sim P_V$$

- Sample  $k$  negative nodes each with prob. proportional to its degree
  - Two considerations for  $k$  (# negative samples):
    1. Higher  $k$  gives more robust estimates
    2. Higher  $k$  corresponds to higher bias on negative events
- In practice  $k = 5-20$ .

Can negative sample be any node or only the nodes not on the walk? People often use any nodes (for efficiency). However, the most “correct” way is to use nodes not on the walk.

# Stochastic Gradient Descent

- After we obtained the objective function, how do we optimize (minimize) it?

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

- Gradient Descent**: a simple way to minimize  $\mathcal{L}$ :

- Initialize  $z_u$  at some randomized value for all nodes  $u$ .
- Iterate until convergence:
  - For all  $u$ , compute the derivative  $\frac{\partial \mathcal{L}}{\partial z_u}$ .
  - For all  $u$ , make a step in reverse direction of derivative:  $z_u \leftarrow z_u - \eta \frac{\partial \mathcal{L}}{\partial z_u}$ .

$\eta$ : learning rate



# Stochastic Gradient Descent

- **Stochastic Gradient Descent:** Instead of evaluating gradients over all examples, evaluate it for each **individual** training example.
  - Initialize  $z_u$  at some randomized value for all nodes  $u$ .
  - Iterate until convergence:
$$\mathcal{L}^{(u)} = \sum_{v \in N_R(u)} -\log(P(v|\mathbf{z}_u))$$
    - Sample a node  $u$ , for all  $v$  calculate the derivative  $\frac{\partial \mathcal{L}^{(u)}}{\partial z_v}$ .
    - For all  $v$ , update:  $z_v \leftarrow z_v - \eta \frac{\partial \mathcal{L}^{(u)}}{\partial z_v}$ .

# Random Walks: Summary

1. Run **short fixed-length** random walks starting from each node on the graph
2. For each node  $u$  collect  $N_R(u)$ , the multiset of nodes visited on random walks starting from  $u$ .
3. Optimize embeddings using Stochastic Gradient Descent:

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

We can efficiently approximate this using negative sampling!

# How should we randomly walk?

- So far we have described how to optimize embeddings given a random walk strategy  $R$
- **What strategies should we use to run these random walks?**
  - Simplest idea: **Just run fixed-length, unbiased random walks starting from each node** (i.e., [DeepWalk from Perozzi et al., 2013](#))
    - The issue is that such notion of similarity is too constrained
- **How can we generalize this?**

Reference: Perozzi et al. 2014. [DeepWalk: Online Learning of Social Representations](#). *KDD*.

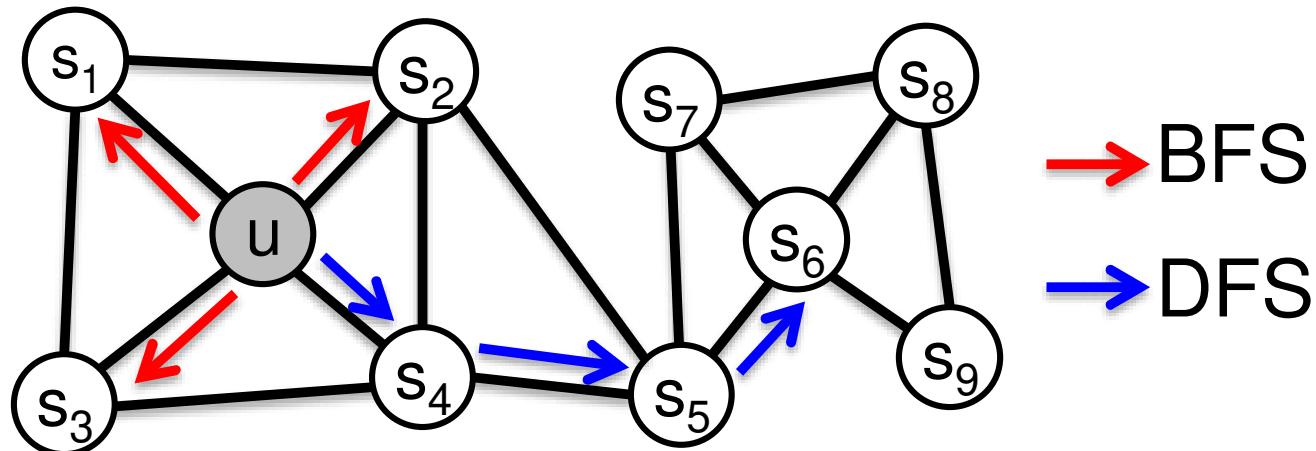
# Overview of node2vec

- **Goal:** Embed nodes with similar network neighborhoods close in the feature space.
- We frame this goal as a maximum likelihood optimization problem, independent to the downstream prediction task.
- **Key observation:** Flexible notion of network neighborhood  $N_R(u)$  of node  $u$  leads to rich node embeddings
- Develop biased 2<sup>nd</sup> order random walk  $R$  to generate network neighborhood  $N_R(u)$  of node  $u$

Reference: Grover et al. 2016. [node2vec: Scalable Feature Learning for Networks](#). KDD.

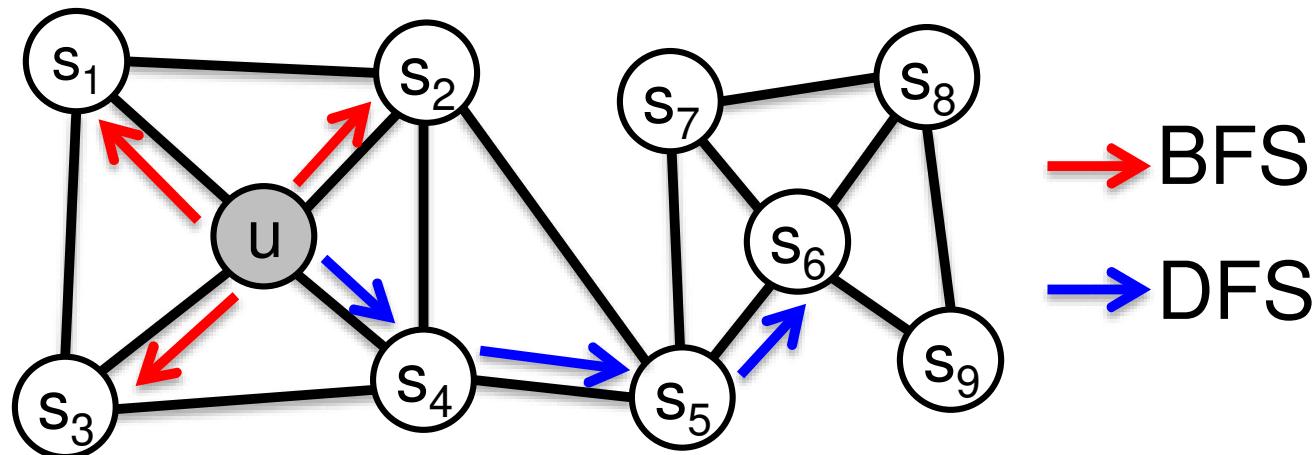
# node2vec: Biased Walks

❑ 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  $u$ :

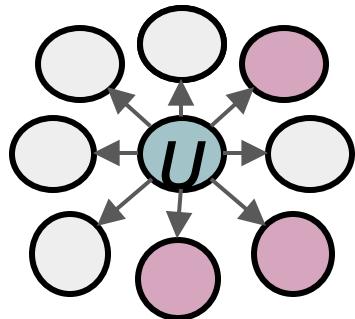


Walk of length 3 ( $N_R(u)$  of size 3):

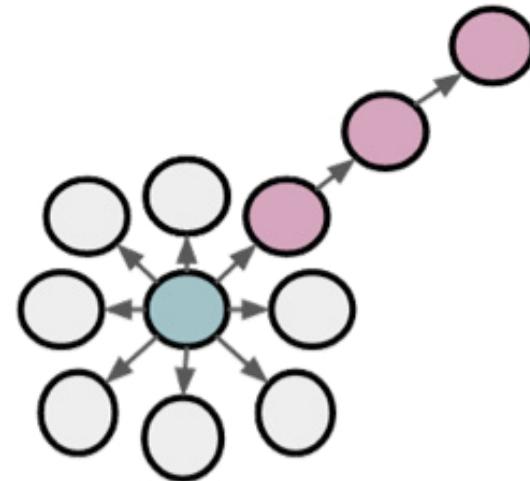
$$N_{BFS}(u) = \{S_1, S_2, S_3\} \quad \text{Local microscopic view}$$

$$N_{DFS}(u) = \{S_4, S_5, S_6\} \quad \text{Global macroscopic view}$$

# BFS vs. DFS



BFS:  
Micro-view of  
neighbourhood



DFS:  
Macro-view of  
neighbourhood

# Interpolating BFS and DFS

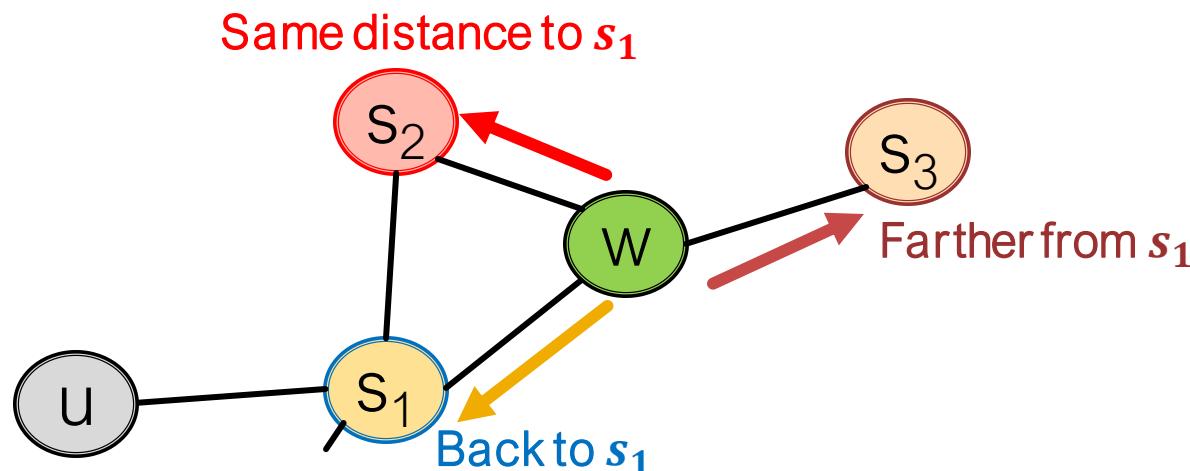
Biased fixed-length random walk  $R$  that given a node  $u$  generates neighborhood  $N_R(u)$

- Two parameters:
  - **Return parameter  $p$ :**
    - Return back to the previous node
  - **In-out parameter  $q$ :**
    - Moving outwards (DFS) vs. inwards (BFS)
    - Intuitively,  $q$  is the “ratio” of BFS vs. DFS

# Biased Random Walks

Biased 2<sup>nd</sup>-order random walks explore network neighborhoods:

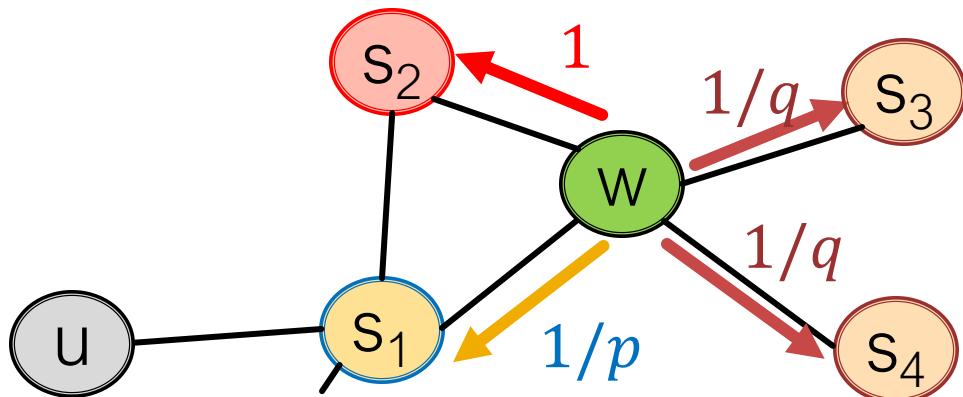
- Rnd. walk just traversed edge  $(s_1, w)$  and is now at  $w$
- **Insight:** Neighbors of  $w$  can only be:



**Idea:** Remember where the walk came from

# Biased Random Walks

- Walker came over edge  $(s_1, w)$  and is at **w**. Where to go next?

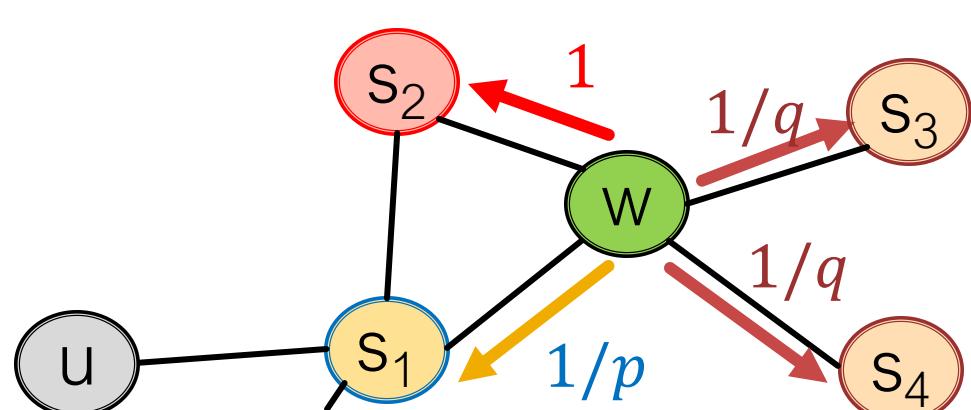


$1/p, 1/q, 1$  are unnormalized probabilities

- $p, q$  model transition probabilities
  - $p$  ... return parameter
  - $q$  ... "walk away" parameter

# Biased Random Walks

- Walker came over edge  $(s_1, w)$  and is at  $w$ .  
Where to go next?



Target $t$	Prob.	Dist. $(s_1, t)$
$s_1$	$1/p$	0
$s_2$	1	1
$s_3$	$1/q$	2
$s_4$	$1/q$	2

Unnormalized  
transition prob.  
segmented based  
on distance from  $s_1$

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

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

# node2vec algorithm

- 1) Compute random walk probabilities
- 2) Simulate  $r$  random walks of length  $l$  starting from each node  $u$
- 3) Optimize the node2vec objective using Stochastic Gradient Descent
- **Linear-time complexity**
- All 3 steps are **individually parallelizable**

# Other Random Walk Ideas

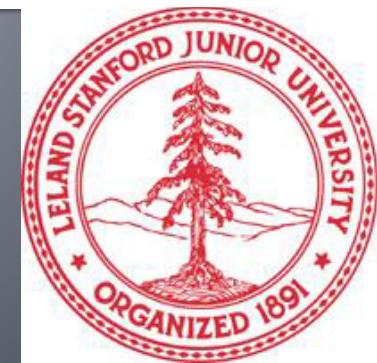
- **Different kinds of biased random walks:**
  - Based on node attributes ([Dong et al., 2017](#)).
  - Based on learned weights ([Abu-El-Haija et al., 2017](#))
- **Alternative optimization schemes:**
  - Directly optimize based on 1-hop and 2-hop random walk probabilities (as in [LINE from Tang et al. 2015](#)).
- **Network preprocessing techniques:**
  - Run random walks on modified versions of the original network (e.g., [Ribeiro et al. 2017's struct2vec](#), [Chen et al. 2016's HARP](#)).

# **Stanford CS224W:** **Embedding Entire Graphs**

CS224W: Machine Learning with Graphs

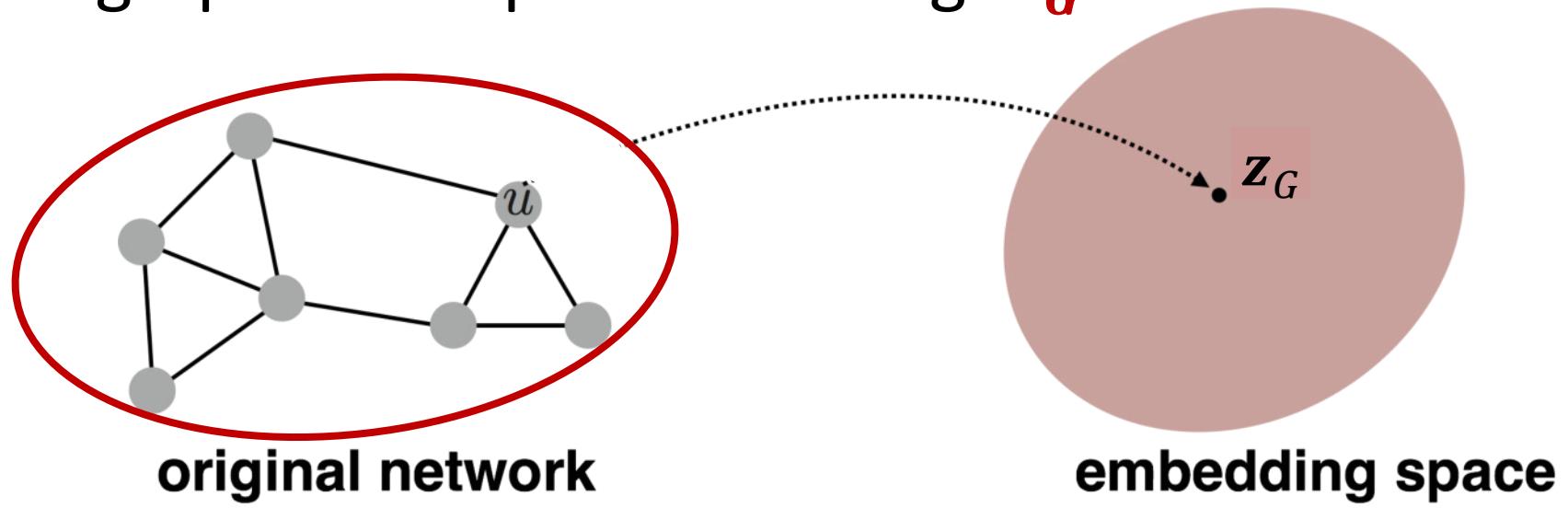
Jure Leskovec, Stanford University

<http://cs224w.stanford.edu>



# Embedding Entire Graphs

- **Goal:** Want to embed a subgraph or an entire graph  $G$ . Graph embedding:  $\mathbf{z}_G$ .



- **Tasks:**
  - Classifying toxic vs. non-toxic molecules
  - Identifying anomalous graphs

# Approach 1

## Simple (but effective) approach 1:

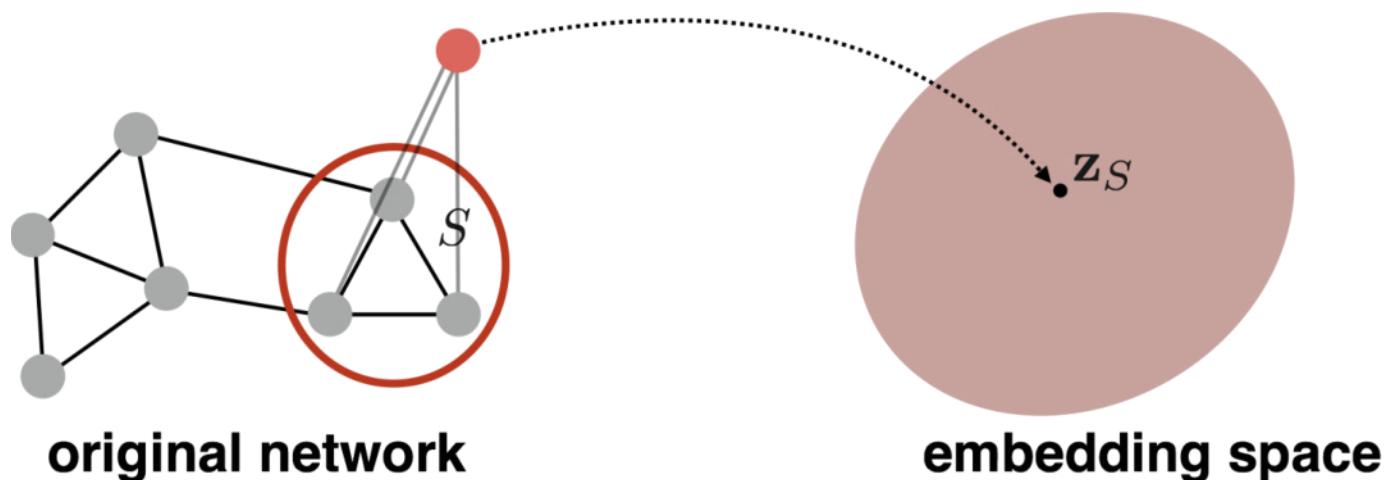
- Run a standard graph embedding technique *on* the (sub)graph  $G$ .
- Then just sum (or average) the node embeddings in the (sub)graph  $G$ .

$$\mathbf{z}_G = \sum_{v \in G} \mathbf{z}_v$$

- Used by Duvenaud et al., 2016 to classify molecules based on their graph structure

# Approach 2

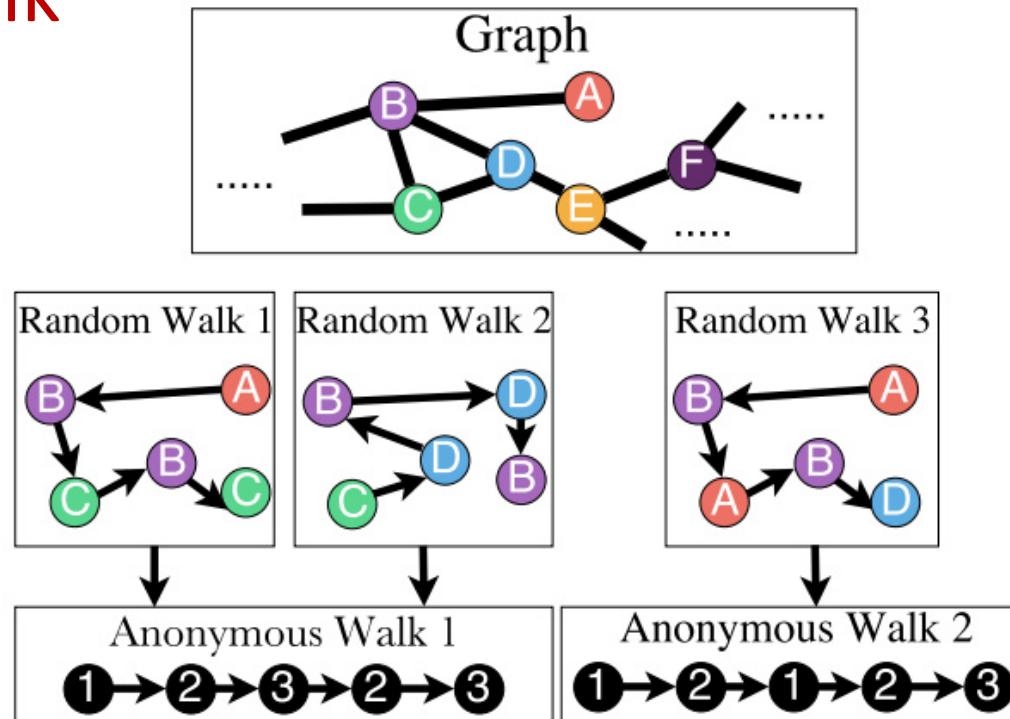
- **Approach 2:** Introduce a “**virtual node**” to represent the (sub)graph and run a standard graph embedding technique



- Proposed by Li et al., 2016 as a general technique for subgraph embedding

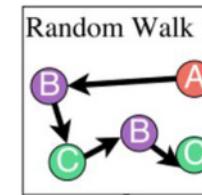
# Approach 3: Anonymous Walk Embeddings

States in **anonymous walks** correspond to the index of the **first time** we visited the node in a random walk



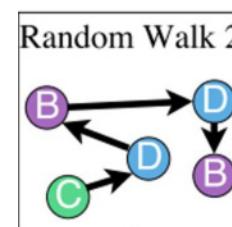
# Approach 3: Anonymous Walk Embeddings

- Agnostic to the identity of the nodes visited (hence anonymous)
- **Example:** Random walk  $w_1$ :

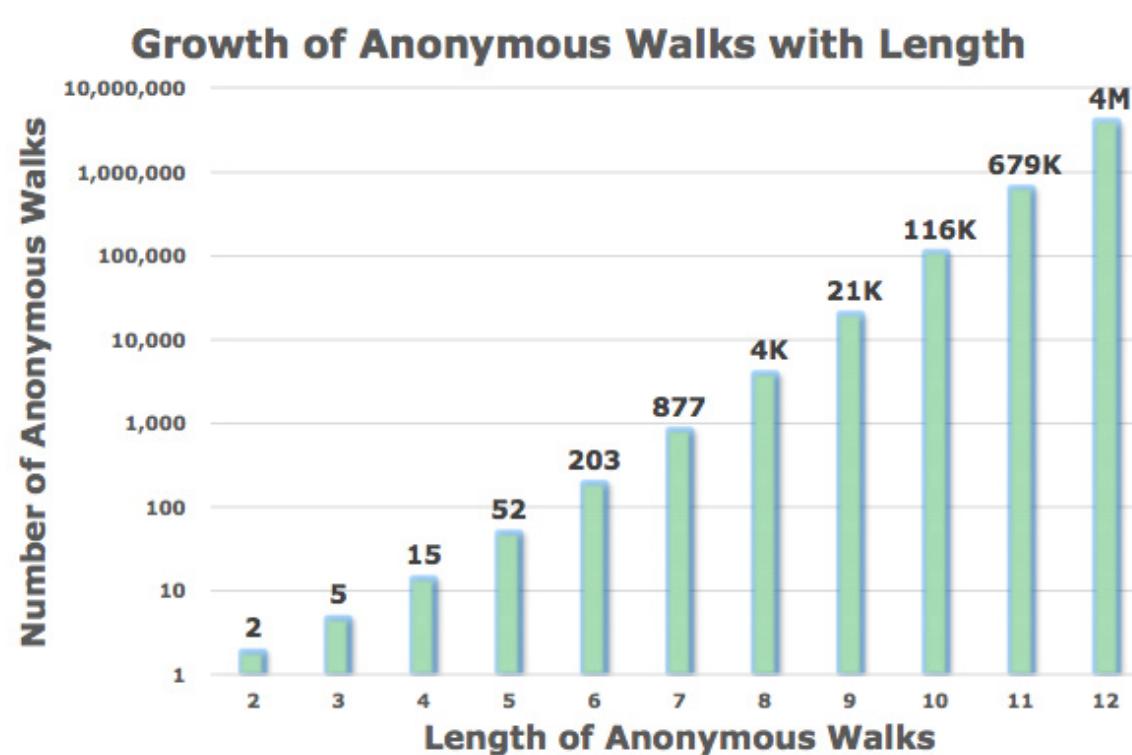


- Step 1: node A  $\longrightarrow$  node 1
- Step 2: node B  $\longrightarrow$  node 2 (different from node 1)
- Step 3: node C  $\longrightarrow$  node 3 (different from node 1, 2)
- Step 4: node B  $\longrightarrow$  node 2 (same as the node in step 2)
- Step 5: node C  $\longrightarrow$  node 3 (same as the node in step 3)

- **Note:** Random walk  $w_2$  gives the same anonymous walk:



# Number of Walks Grows



**Number of anonymous walks grows exponentially:**

- There are 5 anon. walks  $w_i$  of length 3:

$$w_1=111, w_2=112, w_3=121, w_4=122, w_5=123$$

# Simple Use of Anonymous Walks

- Simulate anonymous walks  $w_i$  of  $l$  steps and record their counts.
- Represent the graph as a probability distribution over these walks.
- For example:
  - Set  $l = 3$
  - Then we can represent the graph as a 5-dim vector
    - Since there are 5 anonymous walks  $w_i$  of length 3: 111, 112, 121, 122, 123
  - $\mathbf{z}_G[i] = \text{probability of anonymous walk } w_i \text{ in graph } G.$

# Sampling Anonymous Walks

- **Sampling anonymous walks:** Generate independently a set of  $m$  random walks.
- Represent the graph as a probability distribution over these walks.



## How many random walks $m$ do we need?

- We want the distribution to have error of more than  $\varepsilon$  with prob. less than  $\delta$ :

$$m = \left\lceil \frac{2}{\varepsilon^2} (\log(2^\eta - 2) - \log(\delta)) \right\rceil$$

where:  $\eta$  is the total number of anon. walks of length  $l$ .

**For example:**  
There are  $\eta = 877$  anonymous walks of length  $l = 7$ . If we set  $\varepsilon = 0.1$  and  $\delta = 0.01$  then we need to generate  $m = 122,500$  random walks

# New idea: Learn Walk Embeddings



Rather than simply representing each walk by the fraction of times it occurs, we **learn embedding  $\mathbf{z}_i$  of anonymous walk  $w_i$ .**

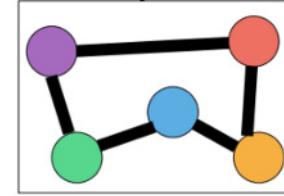
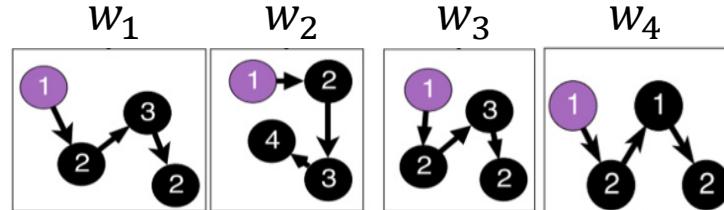
- Learn a graph embedding  $\mathbf{z}_G$  together with all the anonymous walk embeddings  $\mathbf{z}_i$   
 $\mathbf{Z} = \{\mathbf{z}_i : i = 1 \dots \eta\}$ , where  $\eta$  is the number of sampled anonymous walks.

## How to embed walks?

- Idea: Embed walks s.t. the next walk can be predicted.

# Learn Walk Embeddings

- Output: A vector  $\mathbf{z}_G$  for input graph  $G$ 
  - The embedding of entire graph to be learned
- Starting from **node 1**: Sample anonymous random walks, e.g.



- **Learn to predict walks that co-occur in  $\Delta$ -size window** (e.g., predict  $w_3$  given  $w_1, w_2$  if  $\Delta = 2$ )
- Objective:

$$\max_{\mathbf{z}_G} \sum_{t=\Delta+1}^T \log P(w_t | w_{t-\Delta}, \dots, w_{t-1}, \mathbf{z}_G)$$

- Where  $T$  is the total number of walks

# Learn Walk Embeddings



- Run  $T$  different random walks from  $u$  each of length  $l$ :  
$$N_R(u) = \{w_1^u, w_2^u \dots w_T^u\}$$
- Learn to predict walks that co-occur in  $\Delta$ -size window
- Estimate embedding  $\mathbf{z}_i$  of anonymous walk  $w_i$ .  
Let  $\eta$  be number of all possible walk embeddings.

Objective: 
$$\max_{\mathbf{z}_i, \mathbf{z}_G} \frac{1}{T} \sum_{t=\Delta}^T \log P(w_t | \{w_{t-\Delta}, \dots, w_{t-1}, \mathbf{z}_G\})$$

- $P(w_t | \{w_{t-\Delta}, \dots, w_{t-1}, \mathbf{z}_G\}) = \frac{\exp(y(w_t))}{\sum_{i=1}^{\eta} \exp(y(w_i))}$
- $y(w_t) = b + U \cdot \left( \text{cat}\left(\frac{1}{\Delta} \sum_{i=1}^{\Delta} \mathbf{z}_i, \mathbf{z}_G\right) \right)$ 
  - $\text{cat}\left(\frac{1}{\Delta} \sum_{i=1}^{\Delta} \mathbf{z}_i, \mathbf{z}_G\right)$  means an average of anonymous walk embeddings  $\mathbf{z}_i$  in the window, concatenated with the graph embedding  $\mathbf{z}_G$ .
  - $b \in \mathbb{R}$ ,  $U \in \mathbb{R}^D$  are learnable parameters. This represents a linear layer.

# Learn Walk Embeddings

- We obtain the graph embedding  $\mathbf{z}_G$  (learnable parameter) after the optimization.

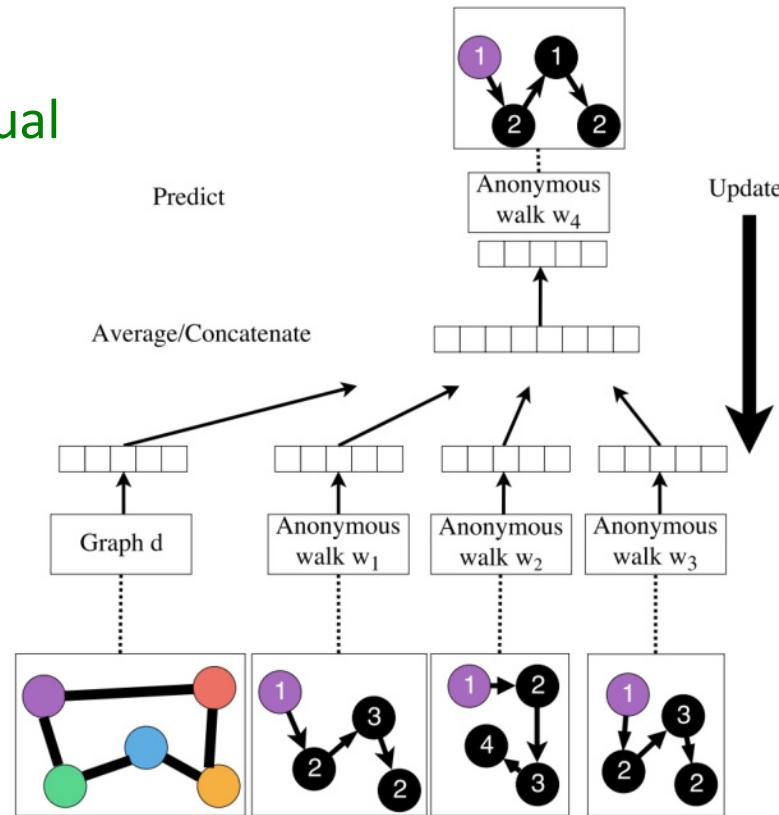
- Is  $\mathbf{z}_G$  simply the sum over walk embeddings  $\mathbf{z}_i$ ? Or is  $\mathbf{z}_G$  the residual embedding next to  $\mathbf{z}_i$ ?
- According to the paper,  $\mathbf{z}_G$  is a separately optimized vector parameter, just like other  $\mathbf{z}_i$ 's.

- Use  $\mathbf{z}_G$  to make predictions (e.g., graph classification):

- **Option1:** Inner product Kernel  $\mathbf{z}_{G_1}^T \mathbf{z}_{G_2}$  (Lecture 2)

- **Option2:** Use a neural network that takes  $\mathbf{z}_G$  as input to classify  $G$ .

## Overall Architecture

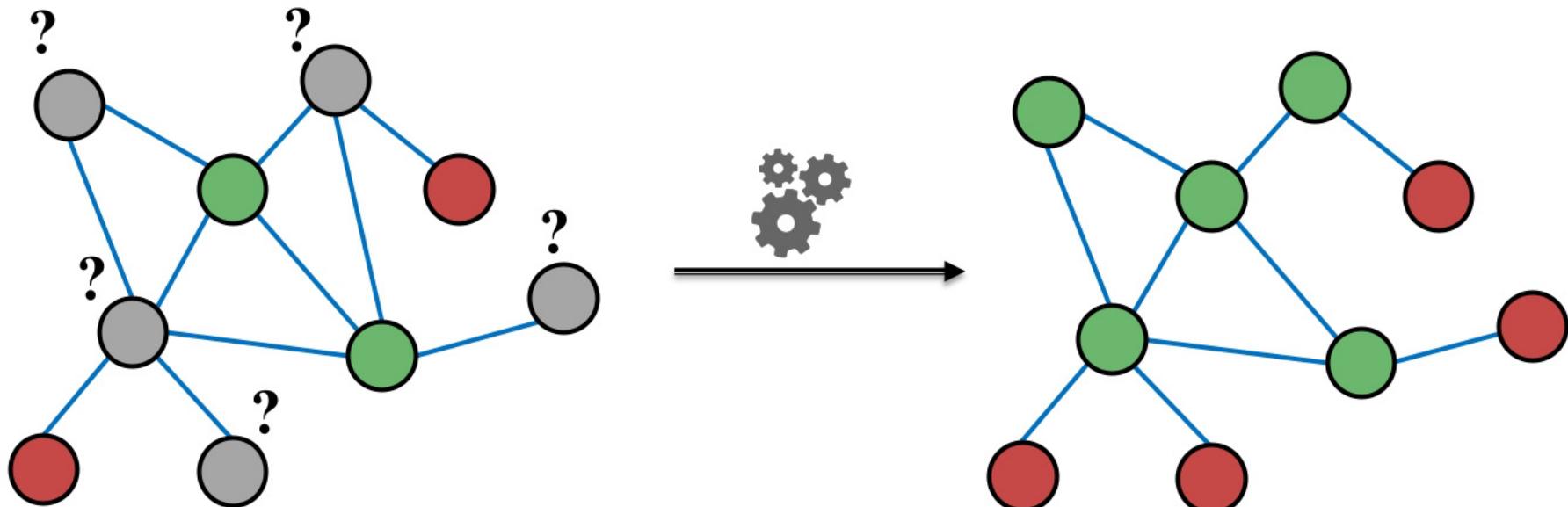


# Message Passing and Node Classification

# Today's Lecture: Outline

- **Main question today:** Given a network with labels on some nodes, how do we assign labels to all other nodes in the network?
- **Example:** In a network, some nodes are fraudsters, and some other nodes are fully trusted. **How do you find the other fraudsters and trustworthy nodes?**
- We already discussed node embeddings as a method to solve this in Lecture 3

# Example: Node Classification



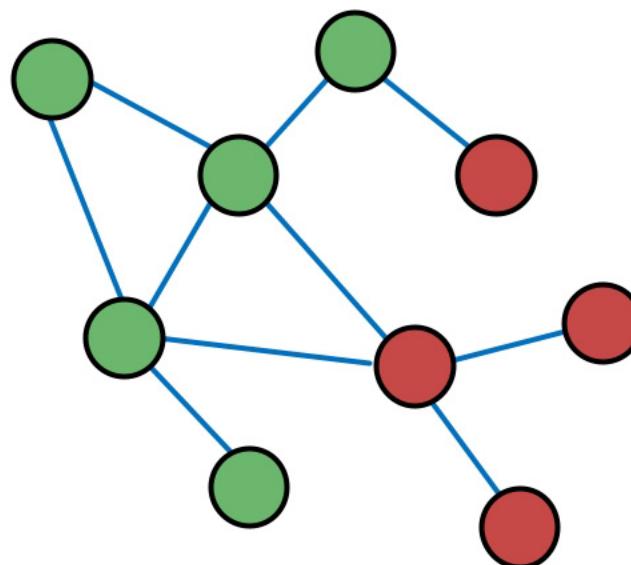
- Given labels of some nodes
- Let's predict labels of unlabeled nodes
- This is called **semi-supervised node classification**

# Today's Lecture: Outline

- **Main question today:** Given a network with labels on some nodes, how do we assign labels to all other nodes in the network?
- **Today we will discuss an alternative framework: Message passing**
- **Intuition:** **Correlations (dependencies)** exist in networks.
  - In other words: Similar nodes are connected.
  - Key concept is **collective classification**: Idea of assigning labels to all nodes in a network together.
- **We will look at three techniques today:**
  - **Relational classification**
  - **Iterative classification**
  - **Correct & Smooth**

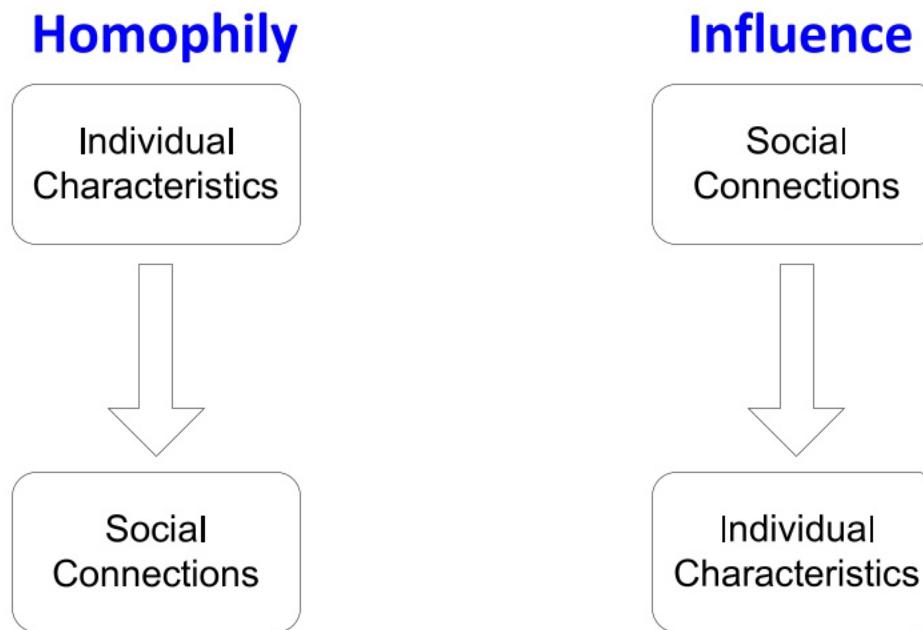
# Correlations Exist in Networks

- Behaviors of nodes are **correlated** across the links of the network
- Correlation:** Nearby nodes have the same color (belonging to the same class)



# Correlations Exist in Networks

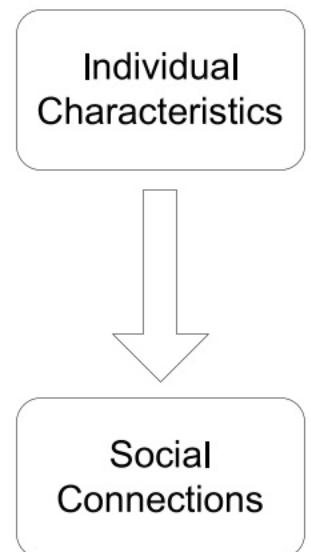
- Two explanations for why behaviors of nodes in networks are correlated:



# Social Homophily

- **Homophily:** The tendency of individuals to associate and bond with similar others
  - “*Birds of a feather flock together*”
  - It has been observed in a vast array of network studies, based on a variety of attributes (e.g., age, gender, organizational role, etc.)
  - **Example:** Researchers who focus on the same research area are **more likely to establish a connection** (meeting at conferences, interacting in academic talks, etc.)

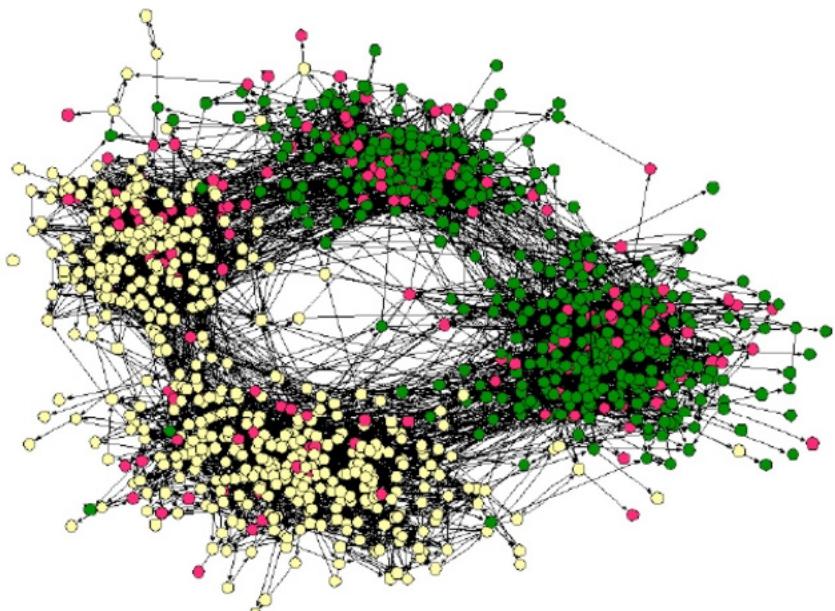
**Homophily**



# Homophily: Example

## Example of homophily

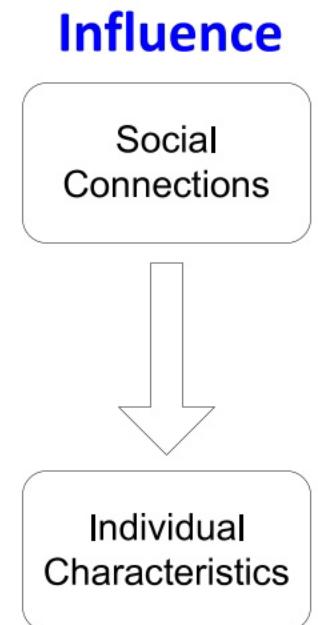
- Online social network
  - Nodes = people
  - Edges = friendship
  - Node color = interests (sports, arts, etc.)
- People with the same interest are more closely connected due to homophily



(Easley and Kleinberg, 2010)

# Social Influence: Example

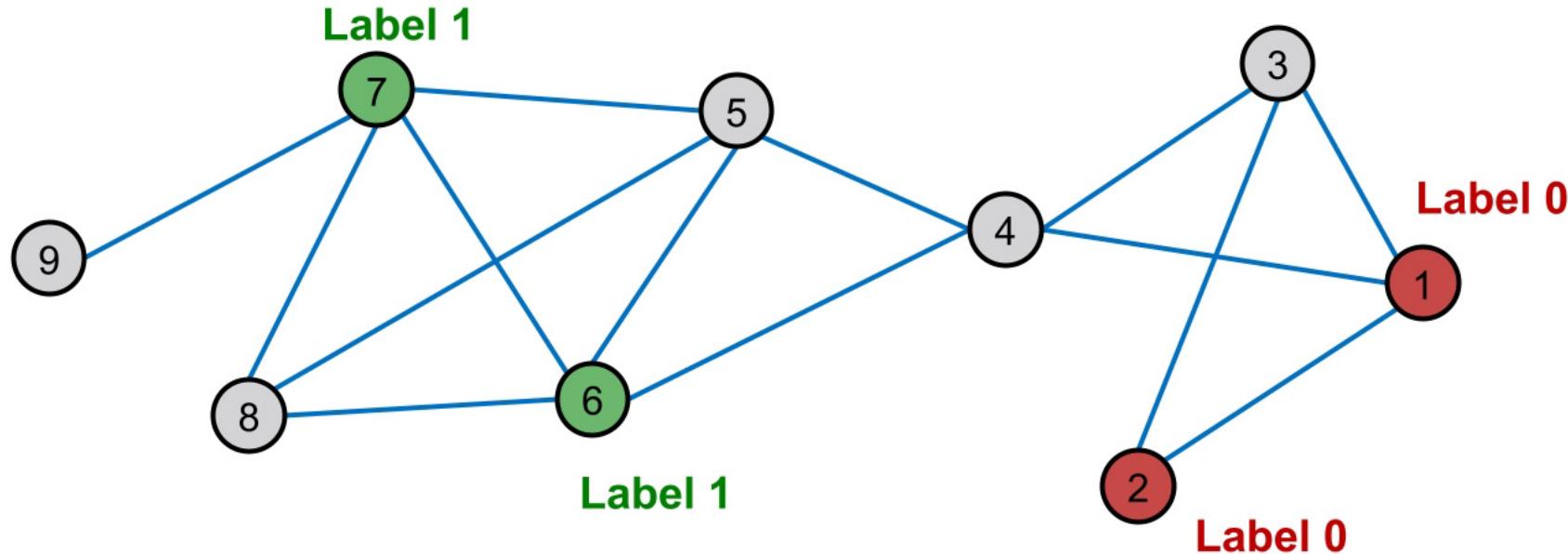
- **Influence:** Social connections can influence the individual characteristics of a person.
  - **Example:** I recommend my musical preferences to my friends, until one of them grows to like my same favorite genres!



**How do we leverage node correlations in networks?**

# Classification with Network Data

- How do we leverage this correlation observed in networks to help predict node labels?



How do we predict the labels for the nodes in grey?

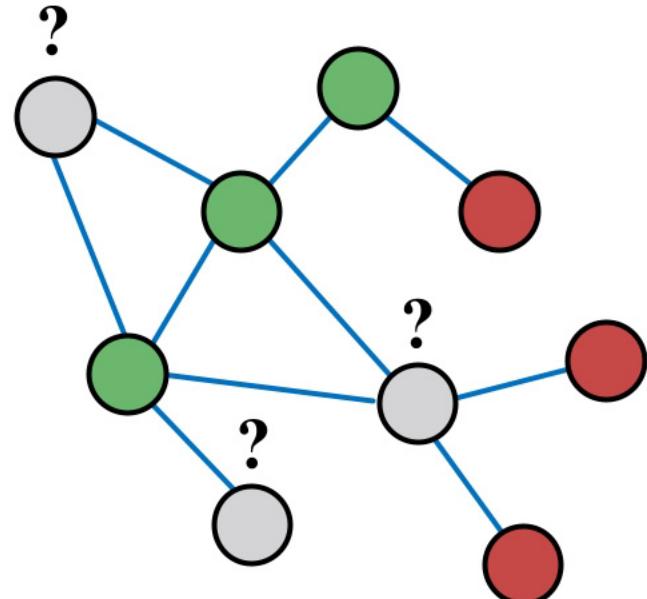
# Motivation (1)

- Similar nodes are typically close together or directly connected in the network:
  - **Guilt-by-association**: If I am connected to a node with label  $X$ , then I am likely to have label  $X$  as well.
  - **Example: Malicious/benign web page**:  
Malicious web pages link to one another to increase visibility, look credible, and rank higher in search engines

# Motivation (2)

- Classification label of a node  $v$  in network may depend on:
  - Features of  $v$
  - Labels of the nodes in  $v$ 's neighborhood
  - Features of the nodes in  $v$ 's neighborhood

# Semi-supervised Learning (1)



**Formal setting:**

**Given:**

- Graph
- Few labeled nodes

**Find:** Class (**red/green**) of remaining nodes

**Main assumption:** There is homophily in the network

# Semi-supervised Learning (2)

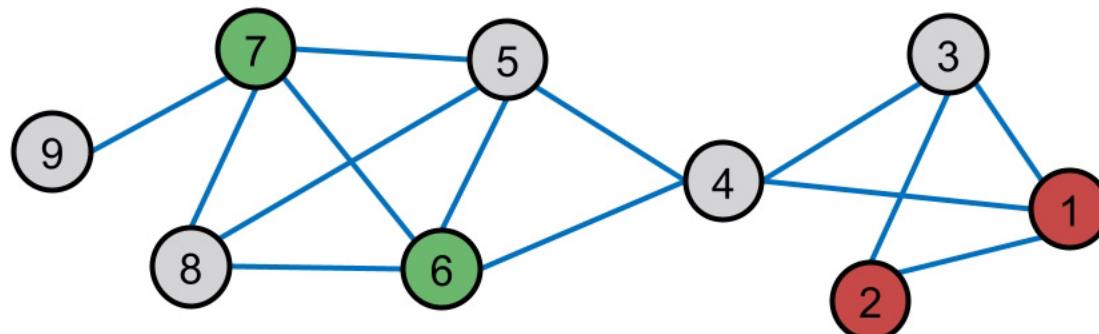
## Example task:

- Let  $A$  be a  $n \times n$  adjacency matrix over  $n$  nodes
- Let  $Y = \{0, 1\}^n$  be a vector of **labels**:
  - $Y_v = 1$  belongs to **Class 1**
  - $Y_v = 0$  belongs to **Class 0**
  - There are **unlabeled** node needs to be classified
- **Goal:** Predict which **unlabeled** nodes are likely **Class 1**, and which are likely **Class 0**

# Problem Setting

- How to predict the labels  $Y_v$  for the unlabeled nodes  $v$  (in grey color)?
- Each node  $v$  has a feature vector  $f_v$
- Labels for some nodes are given (1 for green, 0 for red)
- Task: Find  $P(Y_v)$  given all features and the network

$$P(Y_v) = ?$$



# Example applications:

- **Many applications under this setting:**
  - Document classification
  - Part of speech tagging
  - Link prediction
  - Optical character recognition
  - Image/3D data segmentation
  - Entity resolution in sensor networks
  - Spam and fraud detection

# Overview of What is Coming

- We focus on **semi-supervised binary node classification**
- We will introduce three approaches:
  - Relational classification
  - Iterative classification
  - Correct & Smooth

# **Relational Classification**

# Probabilistic Relational Classifier (1)

- **Idea:** Propagate node labels across the network
  - Class probability  $Y_v$  of node  $v$  is a weighted average of class probabilities of its neighbors.
- For **labeled nodes**  $v$ , initialize label  $Y_v$  with ground-truth label  $Y_v^*$ .
- For **unlabeled nodes**, initialize  $Y_v = 0.5$ .
- **Update** all nodes in a random order until convergence or until maximum number of iterations is reached.

# Probabilistic Relational Classifier (2)

- **Update** for each node  $v$  and label  $c$  (e.g. 0 or 1)

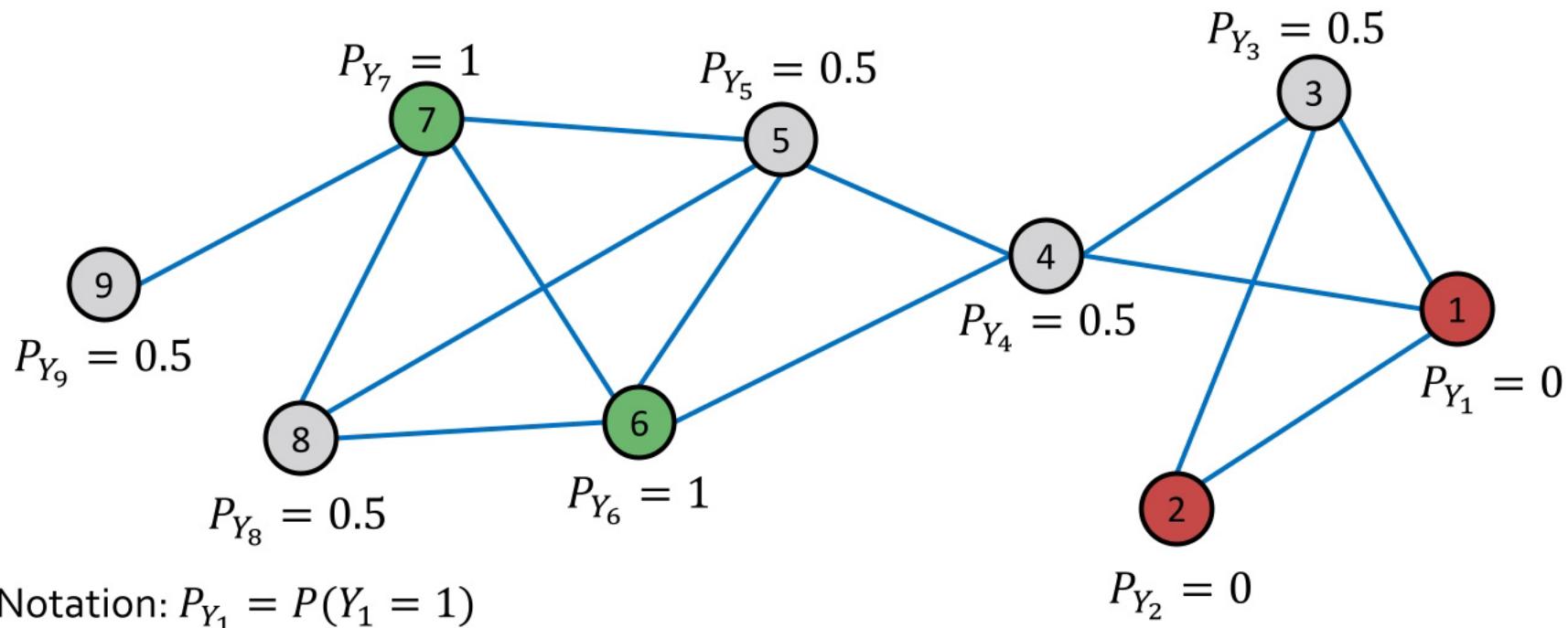
$$P(Y_v = c) = \frac{1}{\sum_{(v,u) \in E} A_{v,u}} \sum_{(v,u) \in E} A_{v,u} P(Y_u = c)$$

- If edges have strength/weight information,  $A_{v,u}$  can be the edge weight between  $v$  and  $u$
- $P(Y_v = c)$  is the probability of node  $v$  having label  $c$
- **Challenges:**
  - Convergence is not guaranteed
  - Model cannot use node feature information

# Example: Initialization

## Initialization:

- All labeled nodes with their labels
- All unlabeled nodes 0.5 (belonging to class 1 with probability 0.5)

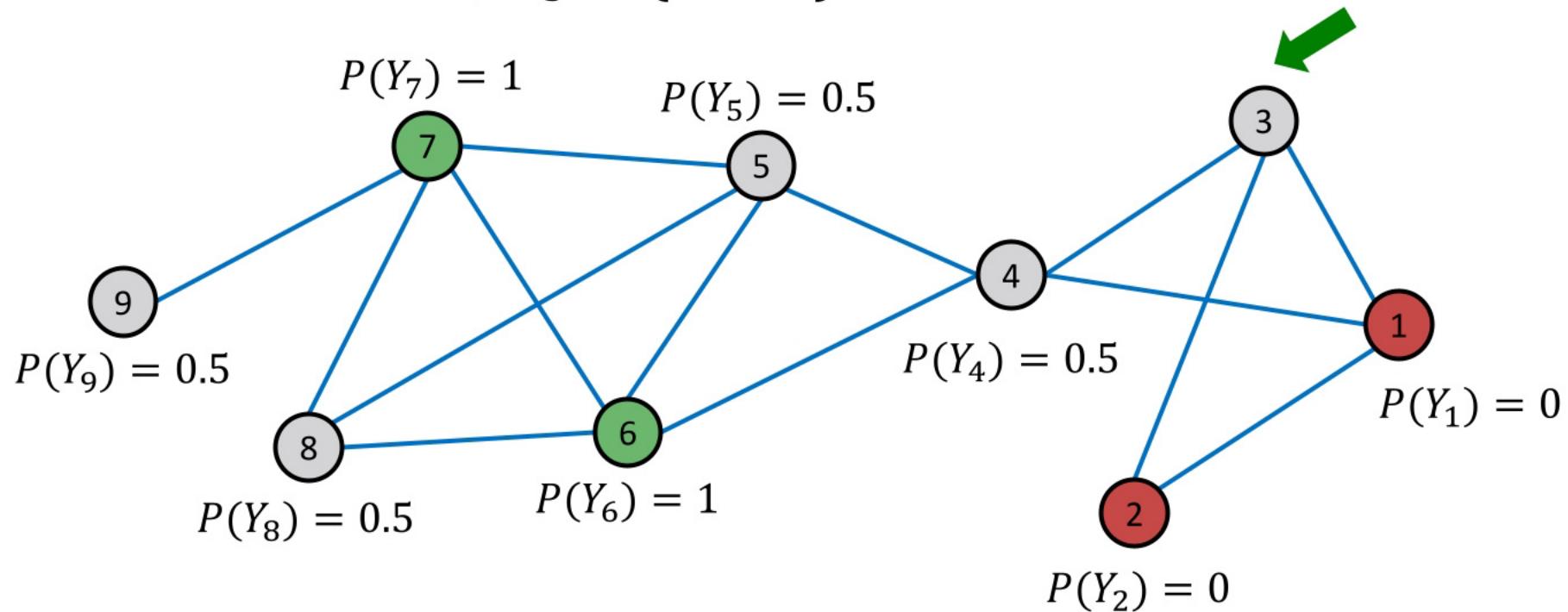


# Example: 1<sup>st</sup> Iteration, Update Node 3

- Update for the 1<sup>st</sup> Iteration:

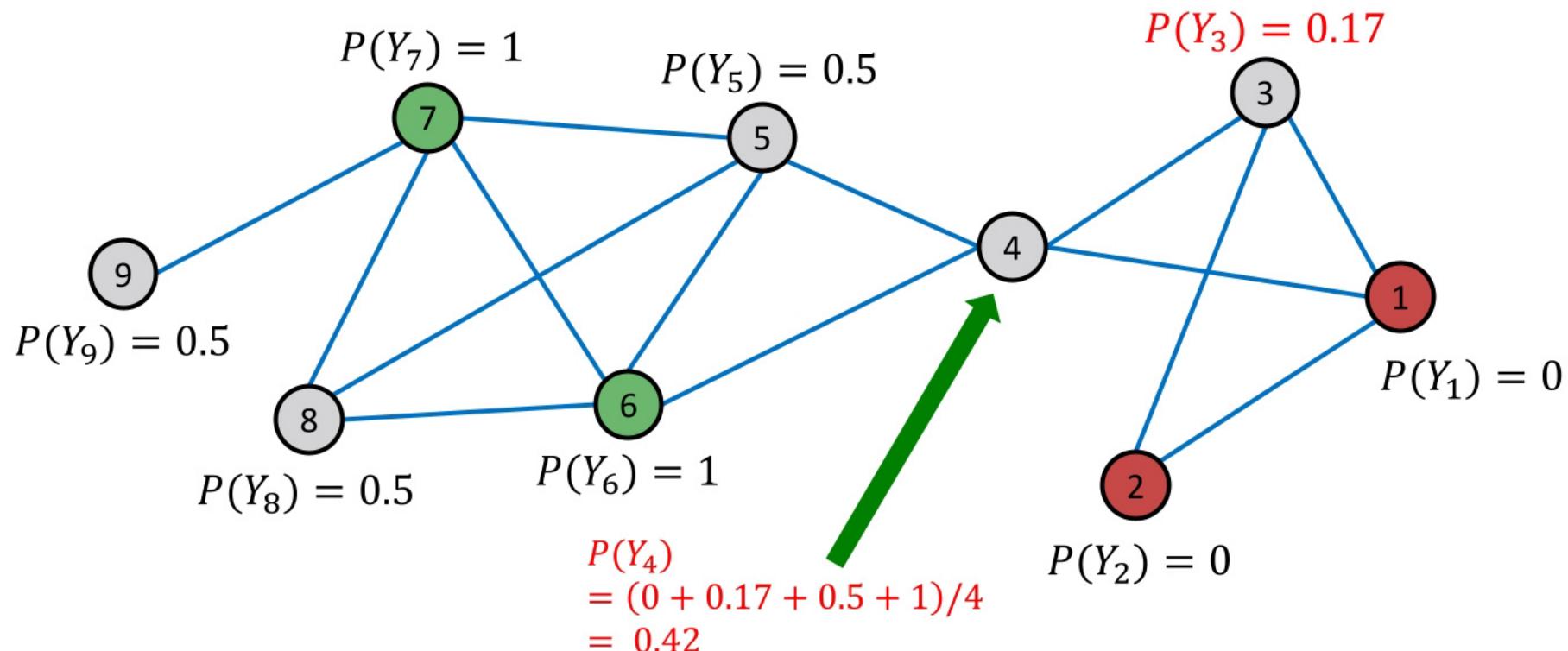
- For node 3,  $N_3 = \{1, 2, 4\}$

$$P(Y_3) = (0 + 0 + 0.5)/3 = 0.17$$



# Example: 1<sup>st</sup> Iteration, Update Node 4

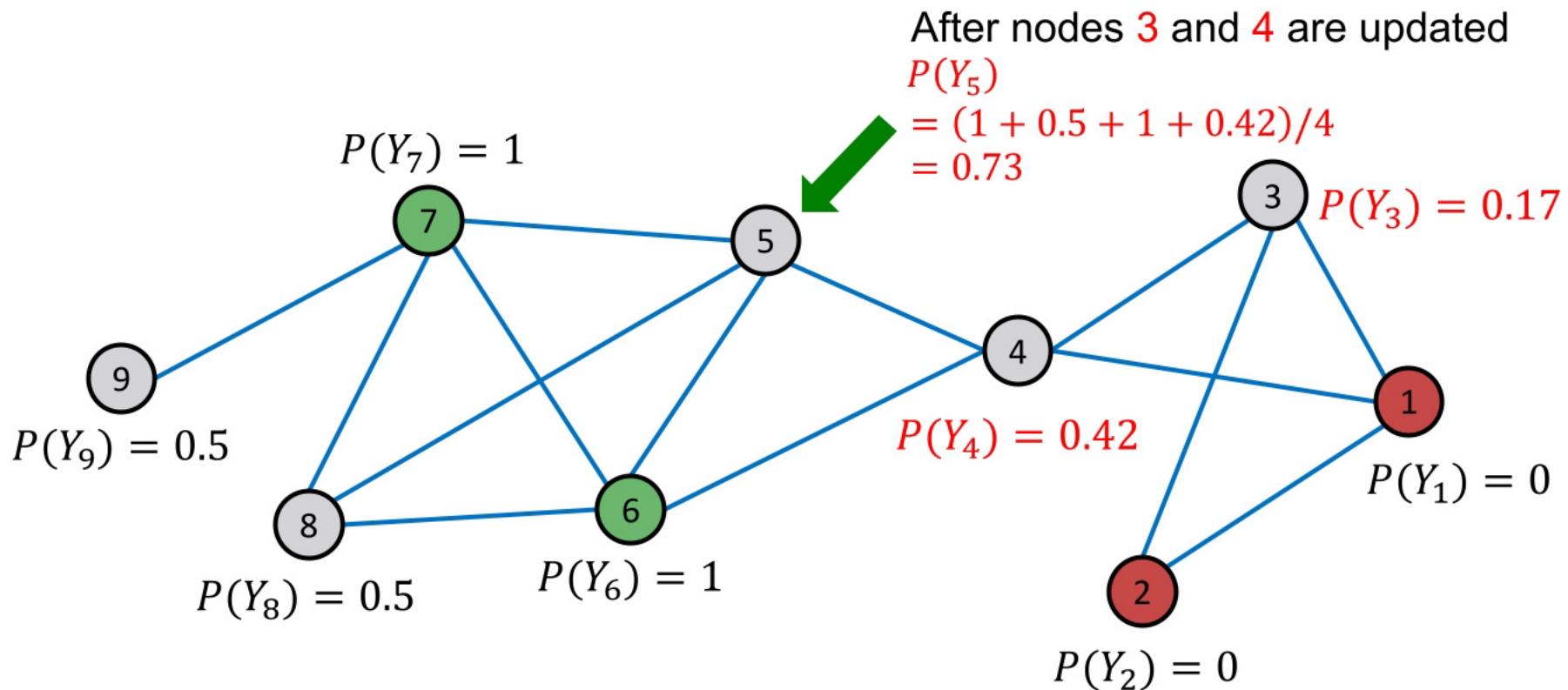
- Update for the 1<sup>st</sup> Iteration:
  - For node 4,  $N_4 = \{1, 3, 5, 6\}$



After Node 3 is updated

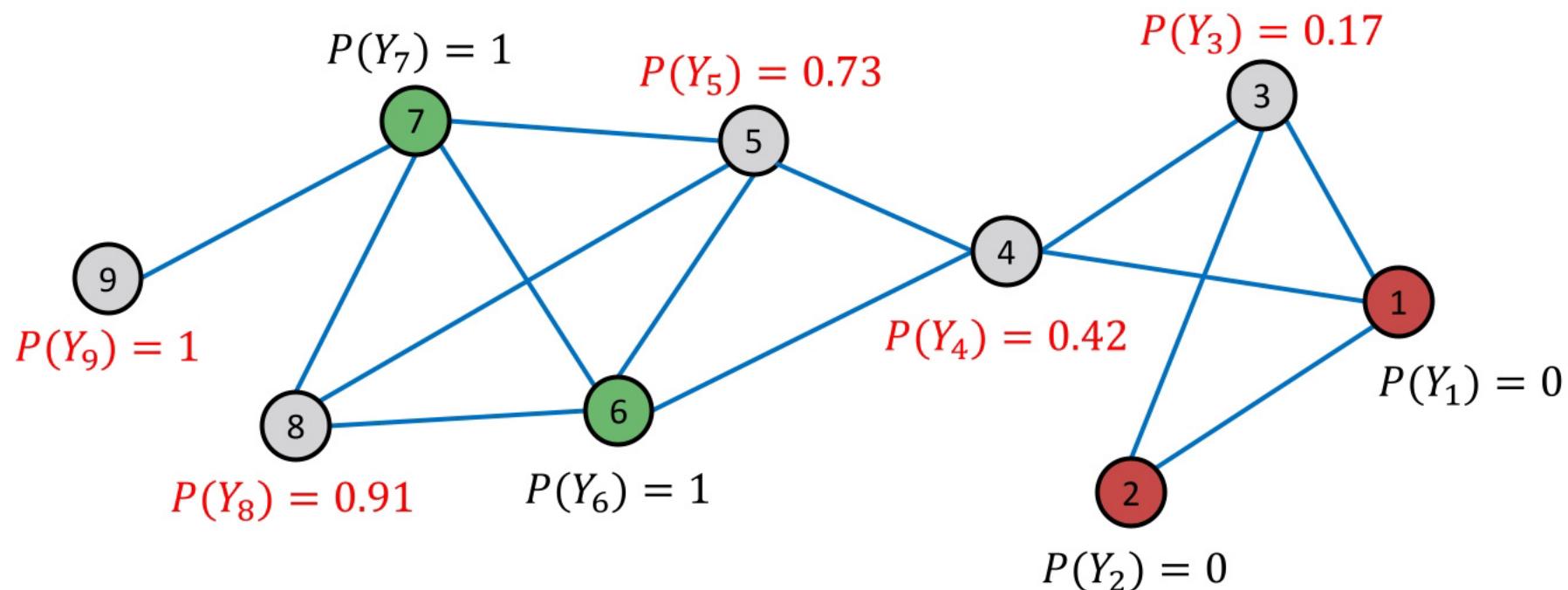
# Example: 1<sup>st</sup> Iteration, Update Node 5

- Update for the 1<sup>st</sup> Iteration:
  - For node 5,  $N_5 = \{4, 6, 7, 8\}$



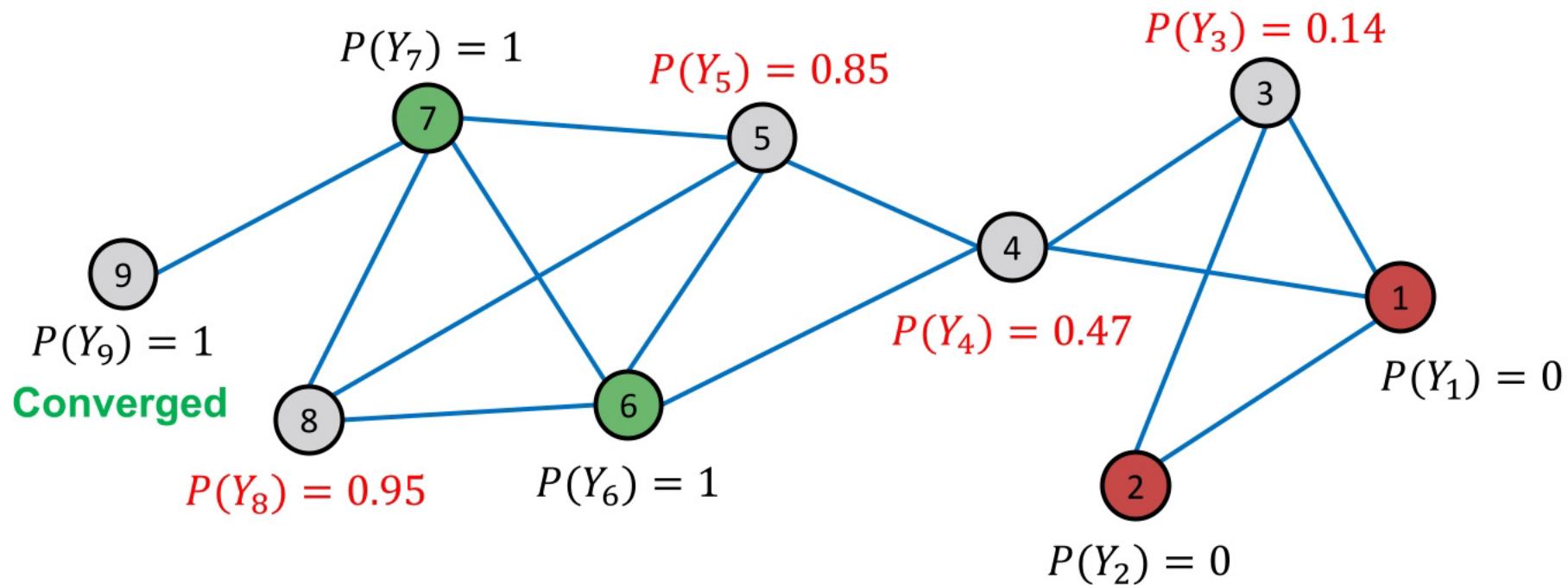
# Example: After 1<sup>st</sup> Iteration

After Iteration 1 (a round of updates for all unlabeled nodes)



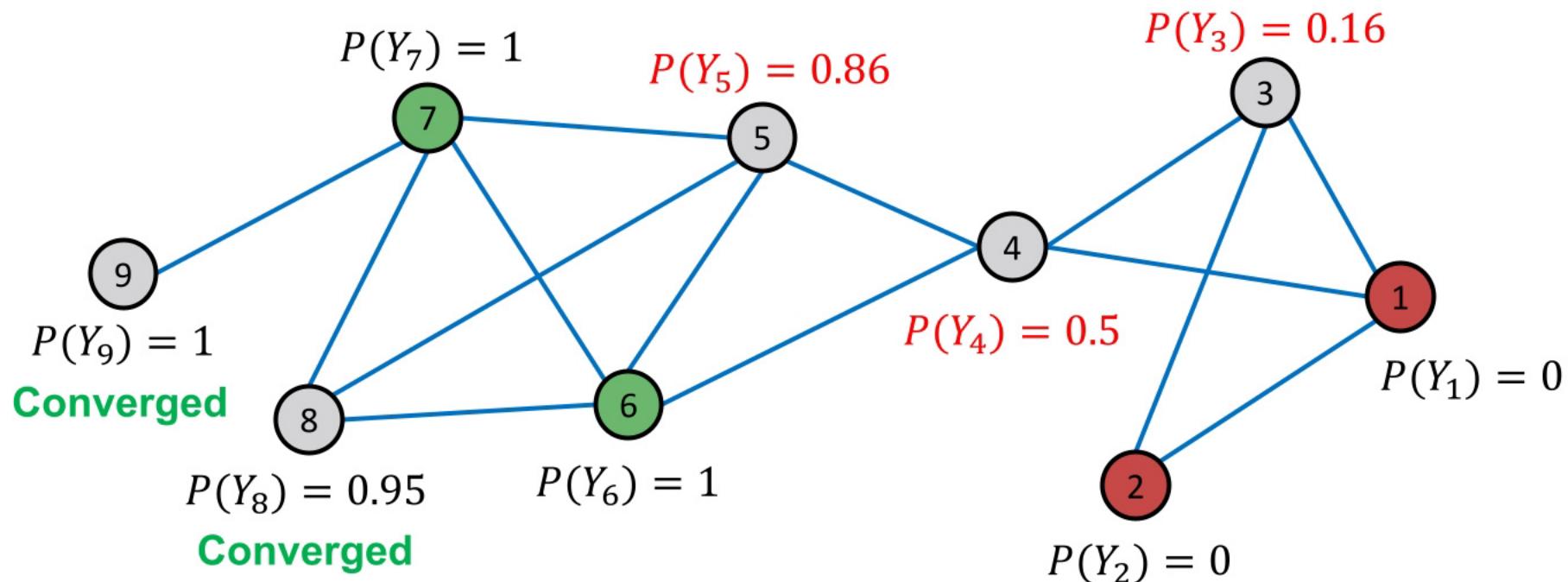
# Example: After 2<sup>nd</sup> Iteration

After Iteration 2



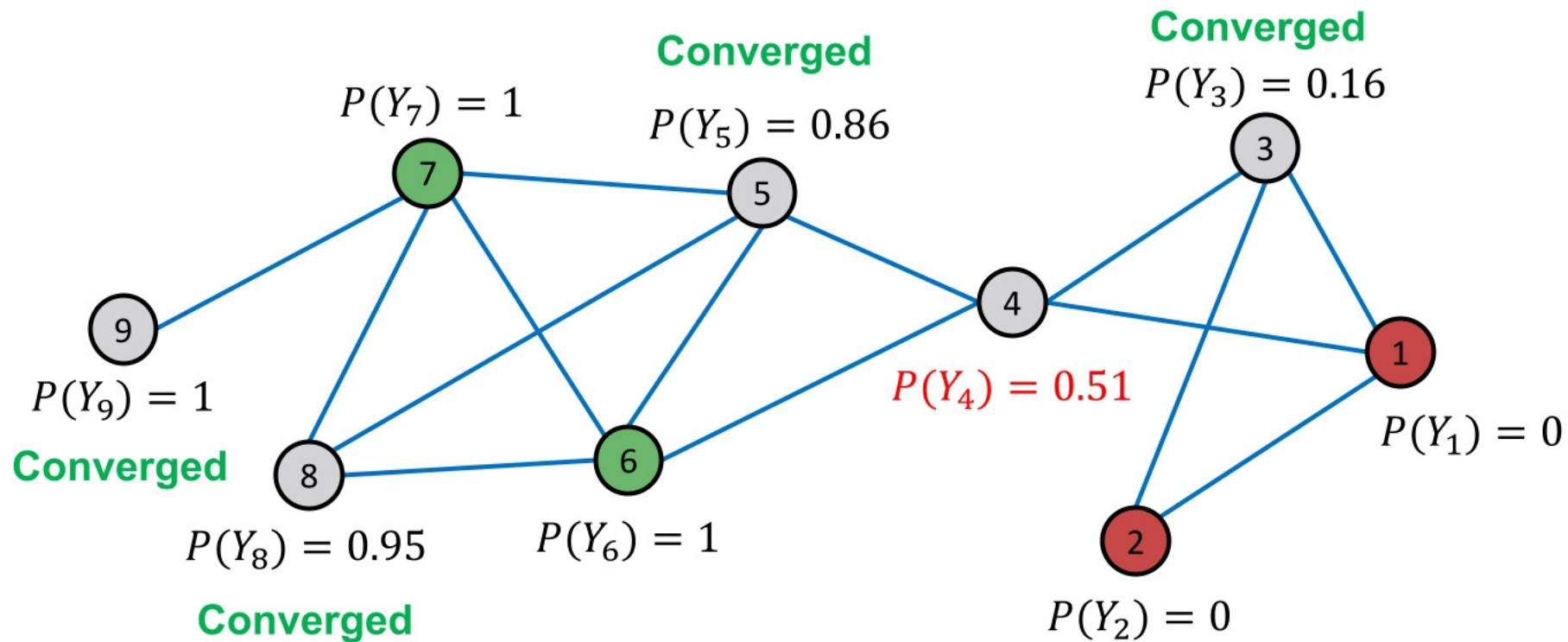
# Example: After 3<sup>rd</sup> Iteration

After Iteration 3



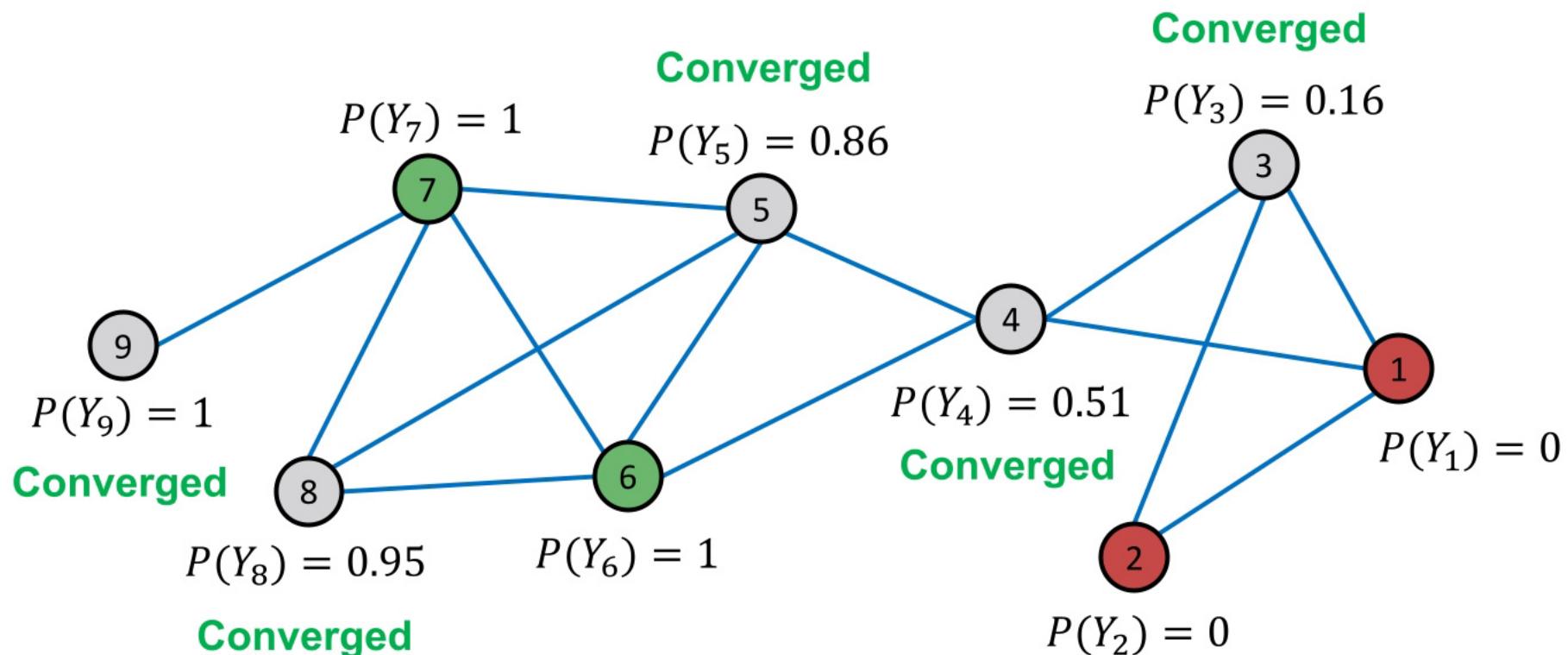
# Example: After 4<sup>th</sup> Iteration

After Iteration 4



# Example: Convergence

- All scores stabilize after 4 iterations. We therefore predict:
  - Nodes 4, 5, 8, 9 belong to class 1 ( $P_{Y_v} > 0.5$ )
  - Nodes 3 belongs to class 0 ( $P_{Y_v} < 0.5$ )



# Iterative Classification

# Iterative Classification

- Relational classifier **does not use node attributes.**
- How can one leverage them?
- **Main idea of iterative classification:** Classify node  $v$  based on its **attributes**  $f_v$ , as well as **labels**  $\mathbf{z}_v$  of neighbor set  $N_v$ .

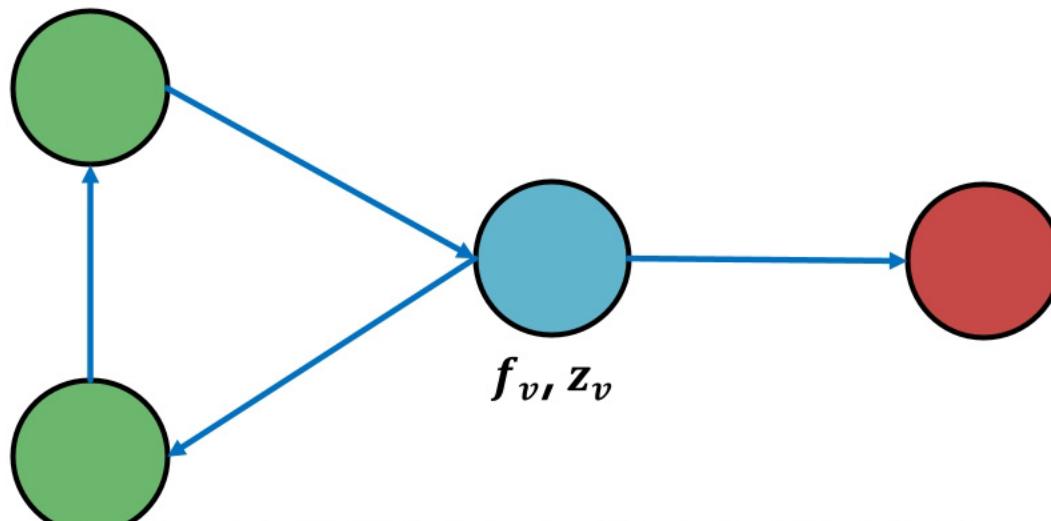
# Iterative Classification

- **Input: Graph**
  - $f_v$  : feature vector for node  $v$
  - Some nodes  $v$  are labeled with  $Y_v$
- **Task:** Predict label of unlabeled nodes
- **Approach: Train two classifiers:**
  - $\phi_1(f_v)$  = Predict node label based on node feature vector  $f_v$ . This is called **base classifier**.
  - $\phi_2(f_v, z_v)$  = Predict label based on node feature vector  $f_v$  and **summary**  $z_v$  of labels of  $v$ 's neighbors. This is called **relational classifier**.

# Computing the Summary $z_v$

How do we compute the summary  $z_v$  of labels of  $v$ 's neighbors  $N_v$ ?

- $z_v$  = vector that captures labels around node  $v$ 
  - Histogram of the number (or fraction) of each label in  $N_v$
  - Most common label in  $N_v$
  - Number of different labels in  $N_v$



# Architecture of Iterative Classifiers

- **Phase 1: Classify based on node attributes alone**
  - On the labeled **training set**, train two classifiers:
    - **Base classifier:**  $\phi_1(f_v)$  to predict  $Y_v$  based on  $f_v$
    - **Relational classifier:**  $\phi_2(f_v, z_v)$  to predict  $Y_v$  based on  $f_v$  and summary  $z_v$  of labels of  $v$ 's neighbors
- **Phase 2: Iterate till convergence**
  - On **test set**, set labels  $Y_v$  based on the classifier  $\phi_1$ , compute  $z_v$  and **predict the labels with**  $\phi_2$
  - **Repeat** for each node  $v$ :
    - Update  $z_v$  based on  $Y_u$  for all  $u \in N_v$
    - Update  $Y_v$  based on the new  $z_v$  ( $\phi_2$ )
  - Iterate until class labels stabilize or max number of iterations is reached
  - **Note:** Convergence is not guaranteed

# **Collective Classification: Correct & Smooth**

# Correct & Smooth

- Last, we introduce **C&S**, recent state-of-the-art collective classification method.

## Leaderboard for ogbn-products

The classification accuracy on the test and validation sets. The higher, the better.

Package:  $\geq 1.1.1$

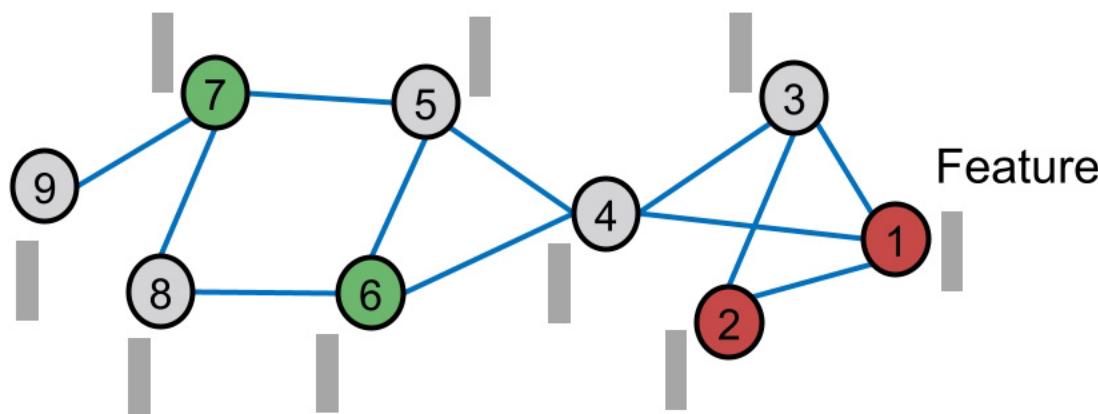
Rank	Method	Test Accuracy	Validation Accuracy	Contact	References	#Params	Hardware	Date
1	SAGN+SLE (4 stages)+C&S	0.8485 ± 0.0010	0.9302 ± 0.0003	Chuxiong Sun (CTRI)	Paper, Code	2,179,678	Tesla V100 (16GB GPU)	Sep 21, 2021
2	SAGN+SLE (4 stages)	0.8468 ± 0.0012	0.9309 ± 0.0007	Chuxiong Sun (CTRI)	Paper, Code	2,179,678	Tesla V100 (16GB GPU)	Sep 21, 2021
3	GAMLP+RLU	0.8459 ± 0.0010	0.9324 ± 0.0005	Wentao Zhang (PKU Tencent Joint Lab)	Paper, Code	3,335,831	Tesla V100 (32GB)	Aug 19, 2021
4	Spec-MLP-Wide + C&S	0.8451 ± 0.0006	0.9132 ± 0.0010	Huixuan Chi (AML@ByteDance)	Paper, Code	406,063	Tesla V100 (32GB)	Jul 27, 2021
5	SAGN+SLE	0.8428 ± 0.0014	0.9287 ± 0.0003	Chuxiong Sun	Paper, Code	2,179,678	Tesla V100 (16GB GPU)	Apr 19, 2021
6	MLP + C&S	0.8418 ± 0.0007	0.9147 ± 0.0009	Horace He (Cornell)	Paper, Code	96,247	GeForce RTX 2080 (11GB GPU)	Oct 27, 2020

**C&S** tops the current OGB leaderboard!

[OGB leaderboard](#) snapshot at Oct 1<sup>st</sup>, 2021

# Correct & Smooth

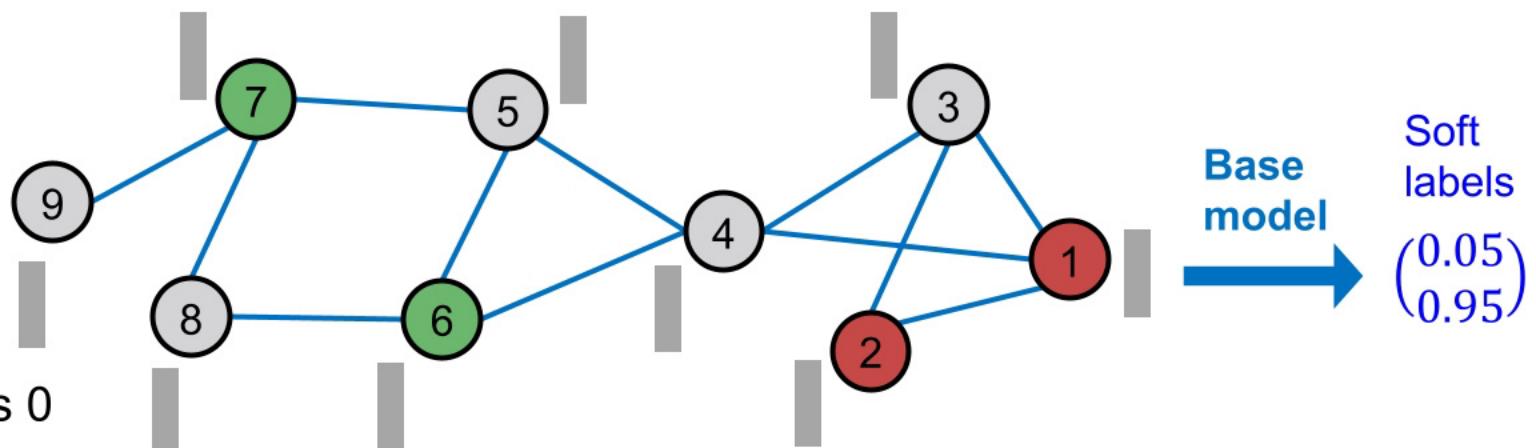
- **Setting:** A partially labeled graph and features over nodes.



- **C&S follows the three-step procedure:**
  1. Train base predictor
  2. Use the base predictor to predict soft labels of all nodes.
  3. **Post-process the predictions using graph structure** to obtain the final predictions of all nodes.

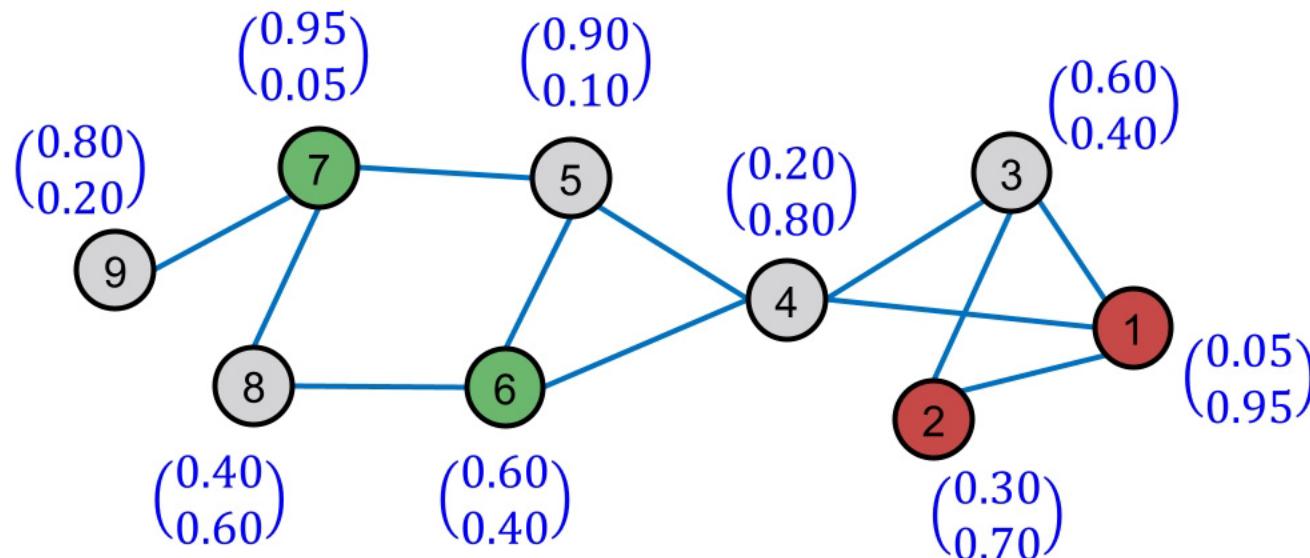
# C&S: (1) Train Base Predictor

- (1) Train a **base predictor** that predict **soft labels** (class probabilities) over all nodes.
  - Labeled nodes are used for train/validation data.
  - Base predictor can be simple:
    - Linear model/Multi-Layer-Perceptron(MLP) over node features



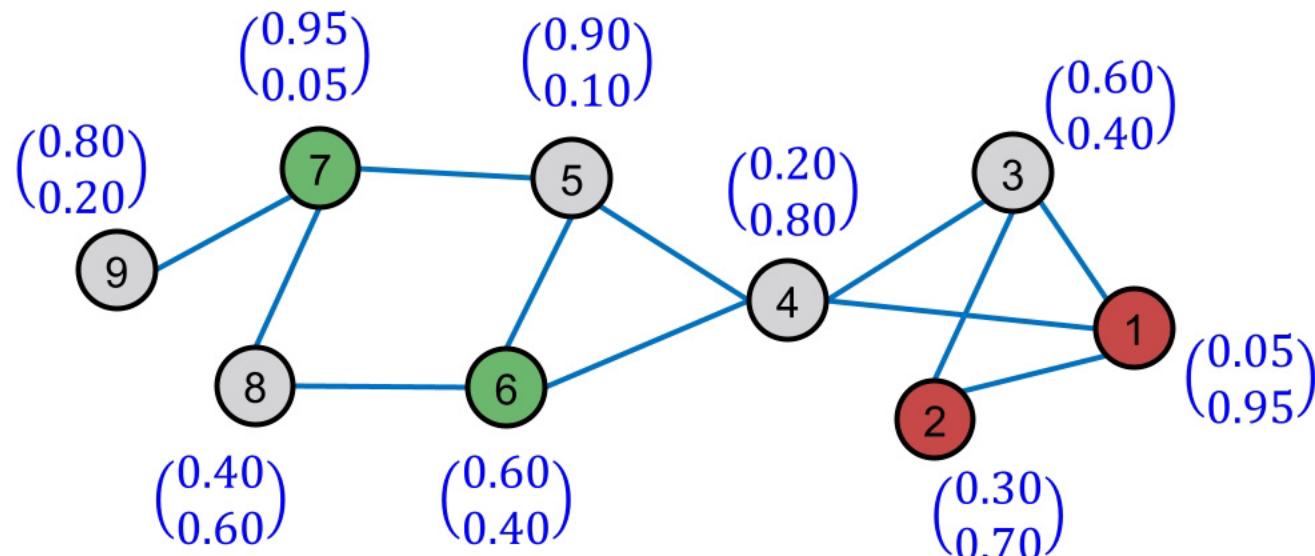
# C&S: (2) Predict Over All Nodes

- (2) Given a trained **base predictor**, we apply it to obtain **soft labels** for all the nodes.
  - We expect these soft labels to be decently accurate.
  - **Can we use graph structure to post-process the predictions to make them more accurate?**



# C&S: (3) Post-Process Predictions

- (3) C&S uses the 2-step procedure to post-process the **soft predictions**.
  1. Correct step
  2. Smooth step



# C&S Post-Processing: Correct Step



- The key idea is that we expect errors in the base prediction to be positively correlated along edges in the graph.
  - In other words, an error at node  $u$  increases the chance of a similar error at neighbors of  $u$ .
  - Thus, we should “spread” such uncertainty over the graph.

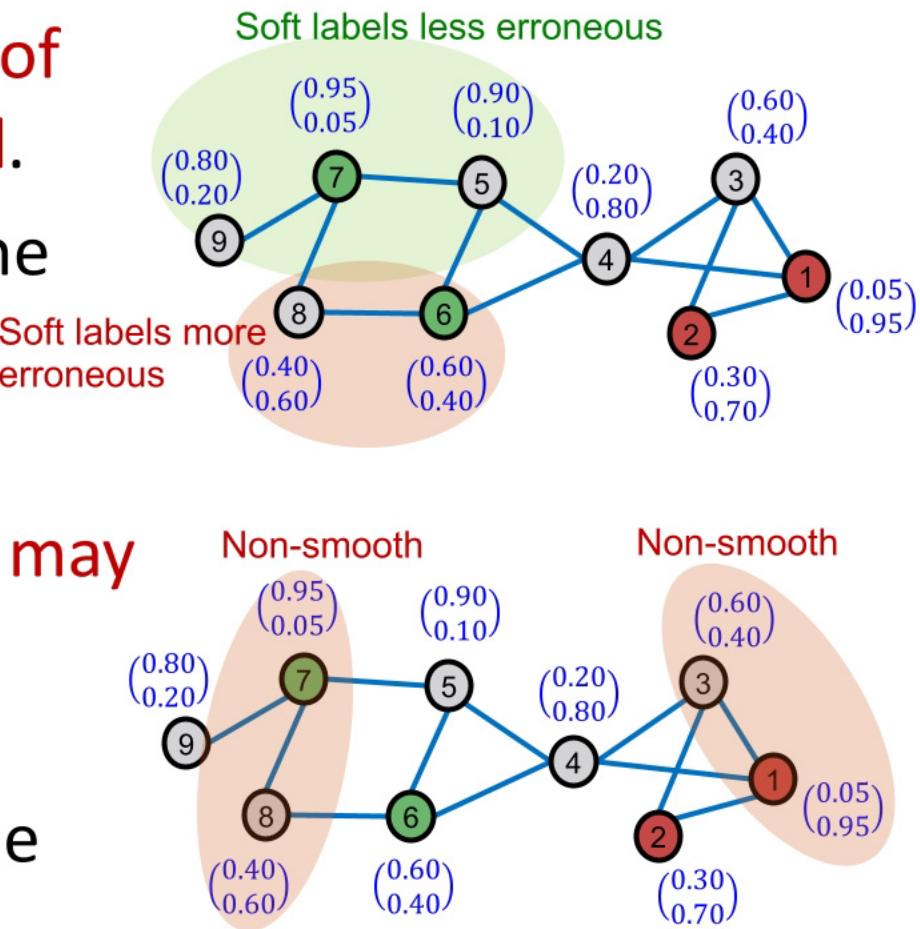
# Intuition of Correct & Smooth

## ■ Correct step

- The degree of the errors of the soft labels are **biased**.
- We need to correct for the error bias.

## ■ Smooth step

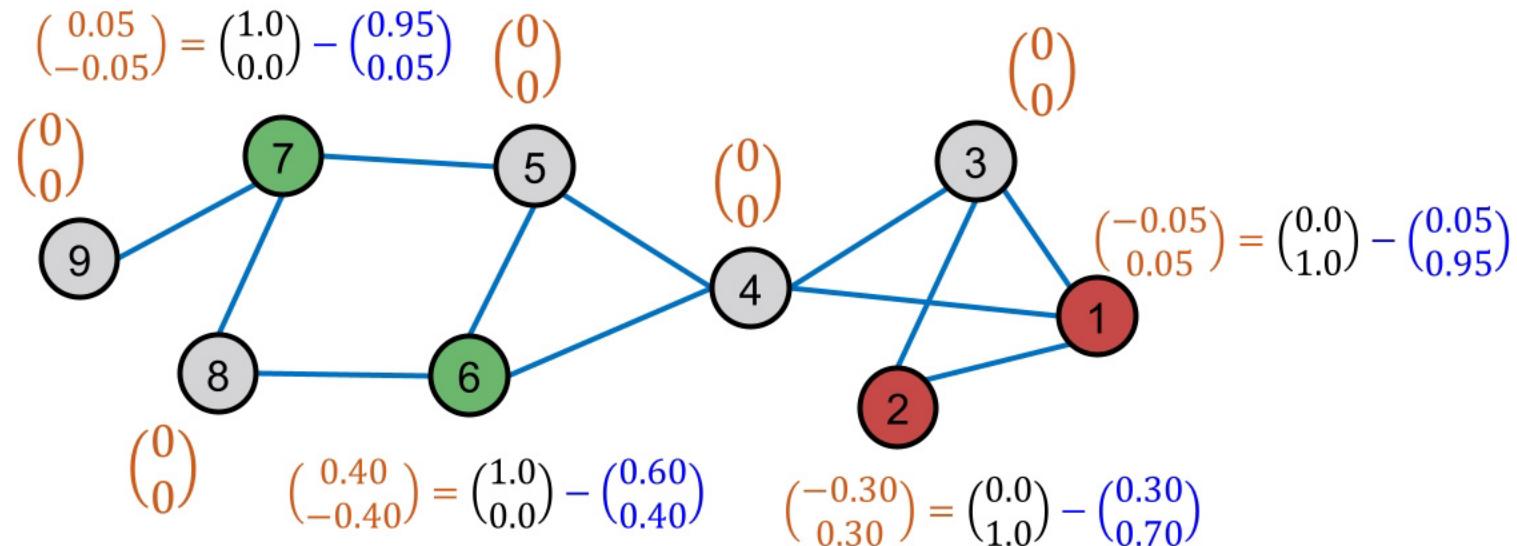
- The predicted soft labels may **not be smooth** over the graph.
- We need to smoothen the soft labels.



# C&S Post-Processing: Correct Step (1)

## ■ Correct step:

- Compute training errors of nodes.
  - Training error:** Ground-truth label minus soft label.  
Defined as 0 for unlabeled nodes.

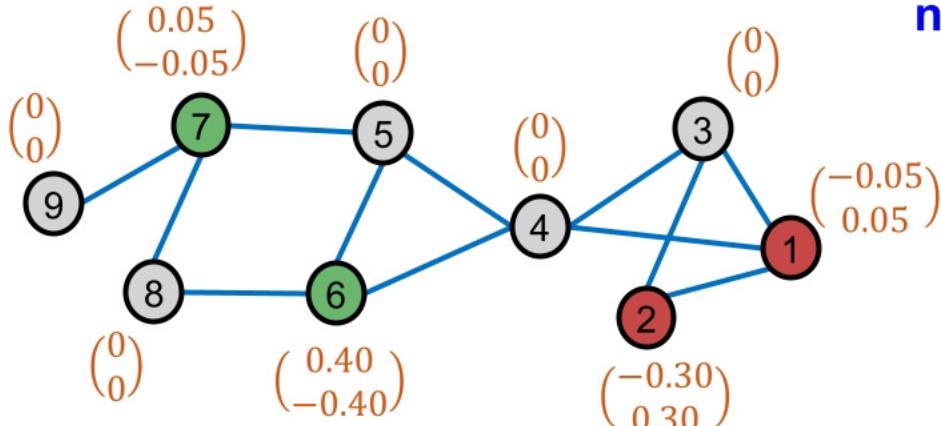


# C&S Post-Processing: Correct Step (2)

## ■ Correct step (contd.):

- Diffuse **training errors**  $E^{(0)}$  along the edges.
- Let  $A$  be the adjacency matrix,  $\tilde{A}$  be the **diffusion matrix** (defined in the next slide).
- $E^{(t+1)} \leftarrow (1 - \boxed{\alpha}) \cdot E^{(t)} + \alpha \cdot \boxed{\tilde{A}E^{(t)}}.$ 
  - Similar to PageRank.

Diffuse training errors along the edges  
**Assumption: errors are similar for nearby nodes**

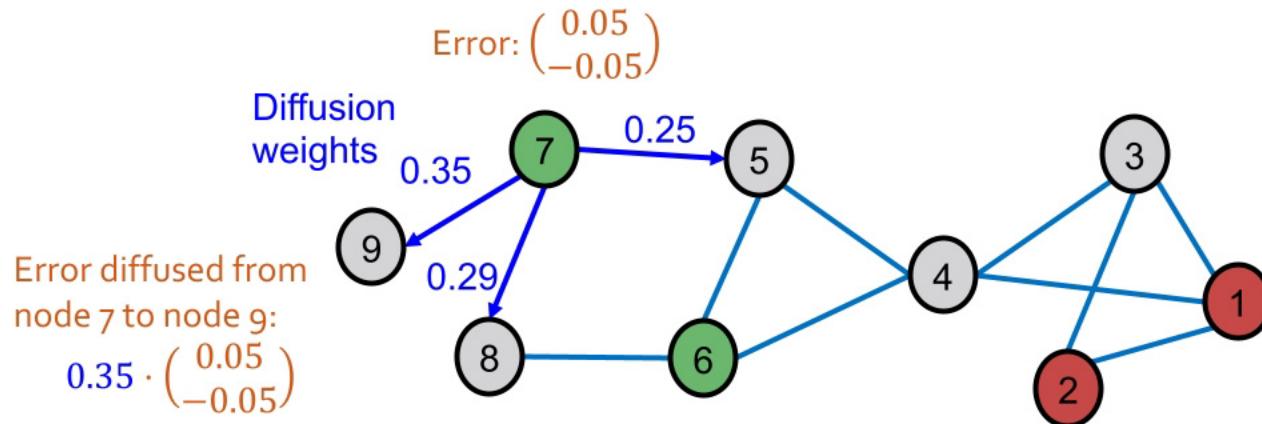


Initial  
training  
error  
matrix

$$E^{(0)} = \begin{pmatrix} -0.05 & 0.05 \\ -0.30 & 0.30 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0.40 & -0.40 \\ 0.05 & -0.05 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

# Diffusion Matrix $\hat{A}$

- Normalized diffusion matrix  $\tilde{A} \equiv D^{-1/2} A D^{-1/2}$ 
  - Add self-loop to the adjacency matrix  $A$ , i.e.,  $A_{ii} = 1$ .
  - Let  $D \equiv \text{Diag}(d_1, \dots, d_N)$  be the degree matrix.
  - See [Zhu et al. ICML 2013](#) for details.

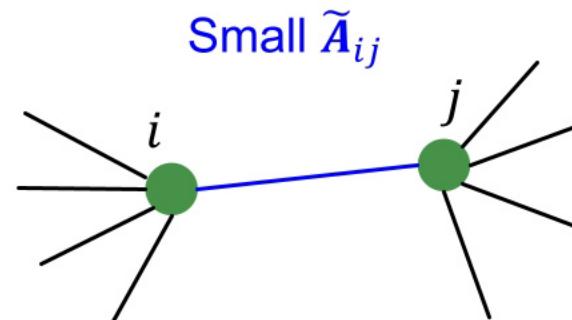
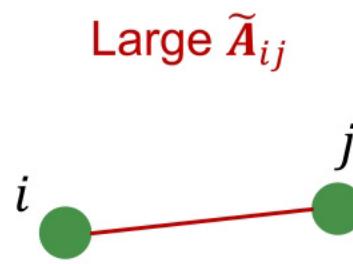


# Theoretical Motivation for $\tilde{A}$

- **Normalized diffusion matrix**  $\tilde{A} \equiv D^{-1/2}AD^{-1/2}$
- All the eigenvalues  $\lambda$ 's are in the range of [-1,1].
  - Eigenvector for  $\lambda = 1$  is  $D^{1/2}\mathbf{1}$  ( $\mathbf{1}$  is an all-one vector).
    - Proof:  $\tilde{A}D^{-1/2}\mathbf{1} = D^{-1/2}AD^{-1/2}D^{1/2}\mathbf{1} = D^{-1/2}A\mathbf{1} = D^{-1/2}D\mathbf{1} = 1 \cdot D^{1/2}\mathbf{1}$ .
  - The power of  $\tilde{A}$  (i.e.,  $\tilde{A}^K$ ) is well-behaved for any  $K$ .
    - The eigenvalues of  $\tilde{A}^K$  are always in the range of [-1,1].
    - The largest eigenvalue is always 1.

# Intuition for $\tilde{A}$

- If  $i$  and  $j$  are connected, the weight  $\tilde{A}_{ij}$  is  $\frac{1}{\sqrt{d_i}\sqrt{d_j}}$
- Intuition:
  - Large if  $i$  and  $j$  are connected only with each other (no other nodes are connected to  $i$  and  $j$ ).
  - Small if  $i$  and  $j$  are connected also connected with many other nodes.

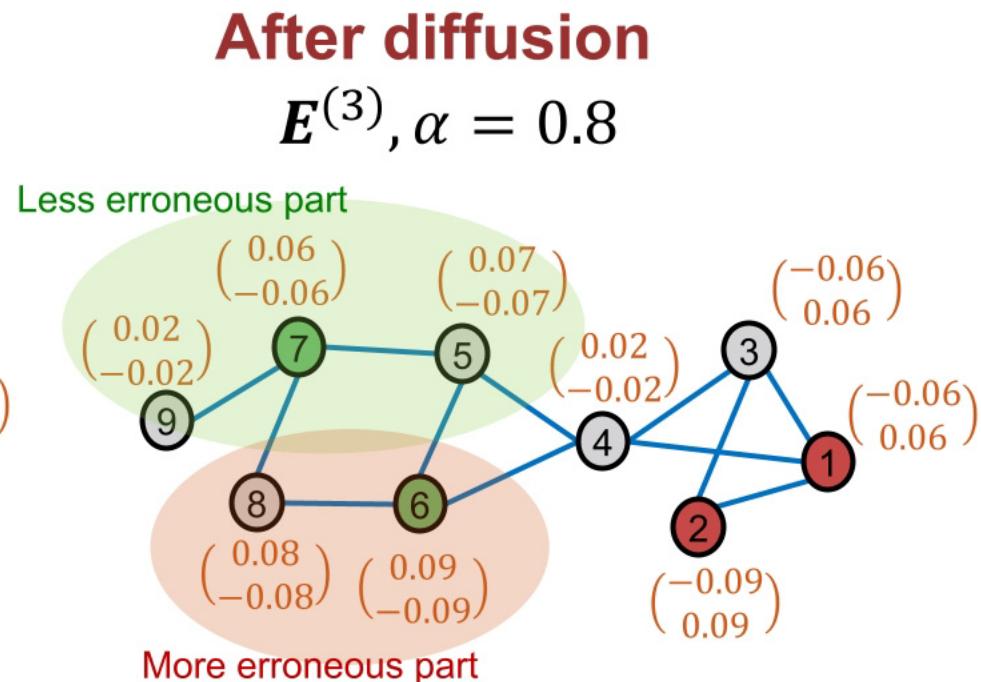
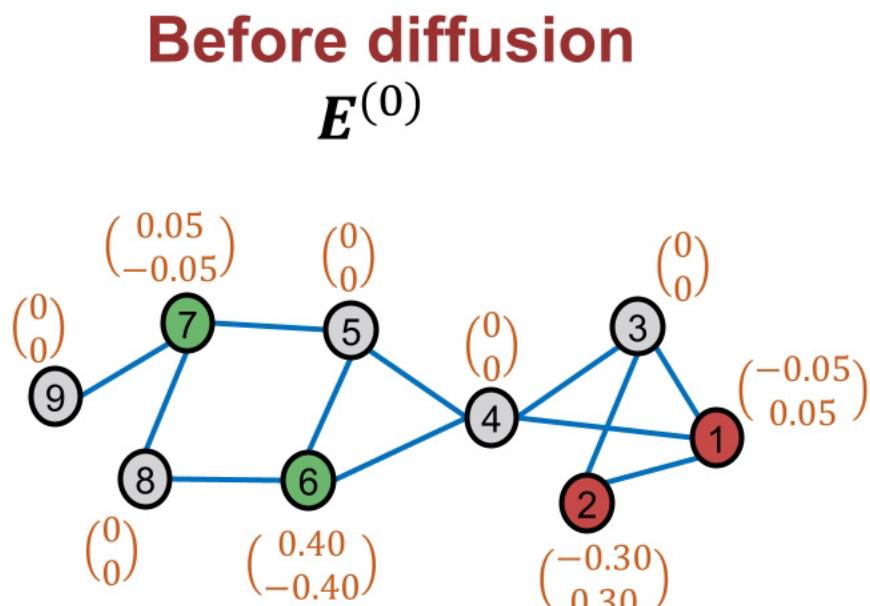


# C&S Post-Processing: Correct Step (3)

## Diffusion of training errors:

$$E^{(t+1)} \leftarrow (1 - \alpha) \cdot E^{(t)} + \alpha \cdot \tilde{A}E^{(t)}$$

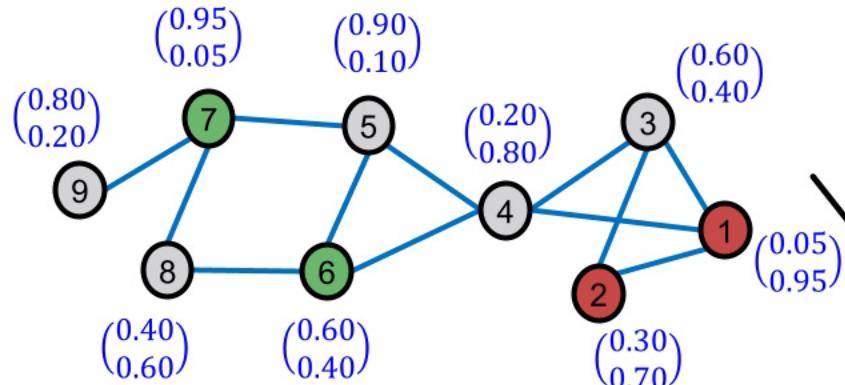
**Assumption:** Prediction errors are similar for nearby nodes.



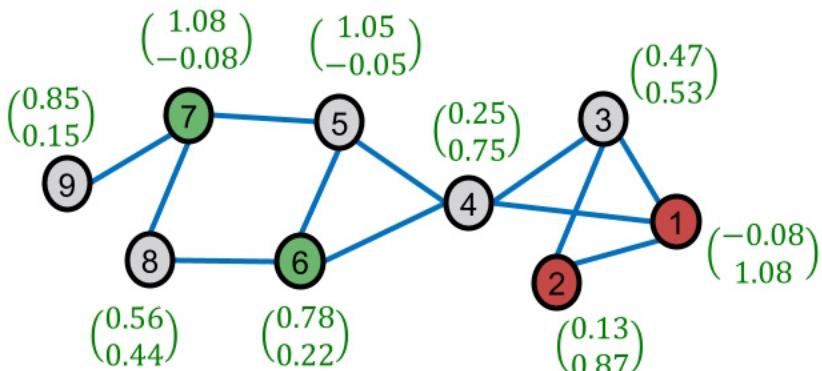
# C&S Post-Processing: Correct Step (4)

- Add the scaled diffused training errors into the predicted soft labels

Soft labels



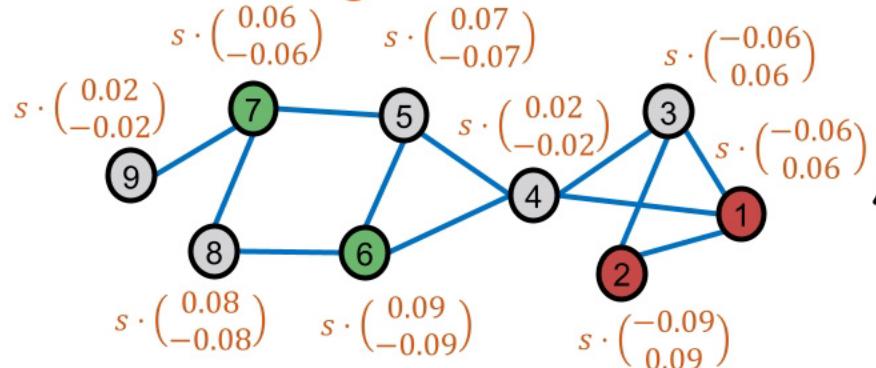
Output after the correct step ( $s = 2$ )



+

Scale by  $s$   
(hyper-parameter)

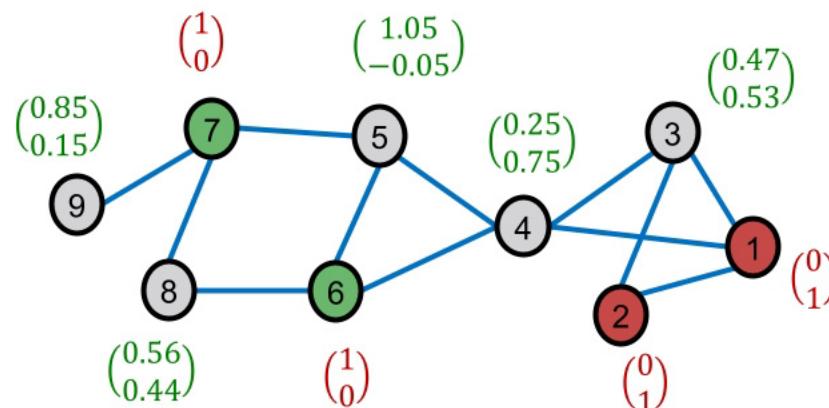
Diffused training errors



# C&S Post-Processing: Smooth Step

- Smoothen the corrected soft labels along the edges.
- **Assumption:** Neighboring nodes tend to share the same labels.
- **Note:** For training nodes, we use the **ground-truth hard labels** instead of the soft labels.

Input to the smooth step:



# C&S Post-Processing: Smooth Step (1)

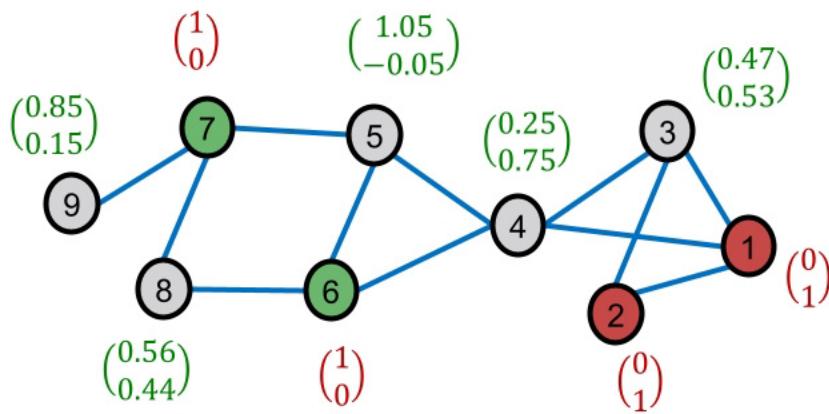
## ■ Smooth step:

- Diffuse label  $Z^{(0)}$  along the graph structure.

$$Z^{(t+1)} \leftarrow (1 - \boxed{\alpha}) \cdot Z^{(t)} + \alpha \cdot \boxed{\tilde{A}Z^{(t)}}.$$

Hyper-parameter

Diffuse labels along the edges



Corrected  
label  
matrix

$$Z^{(0)} = \begin{pmatrix} 0 & 1 \\ 0 & 1 \\ 0.47 & 0.53 \\ 0.25 & 0.75 \\ 1.05 & -0.05 \\ 1 & 0 \\ 1 & 0 \\ 0.56 & 0.44 \\ 0.85 & 0.15 \end{pmatrix}$$

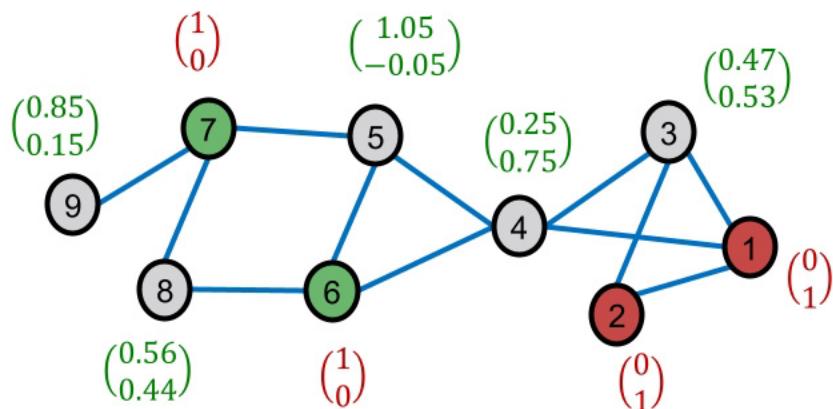
# C&S Post-Processing: Correct Step (2)

## ■ Smooth step:

$$\mathbf{Z}^{(t+1)} \leftarrow (1 - \alpha) \cdot \mathbf{Z}^{(t)} + \alpha \cdot \tilde{\mathbf{A}}\mathbf{Z}^{(t)}$$

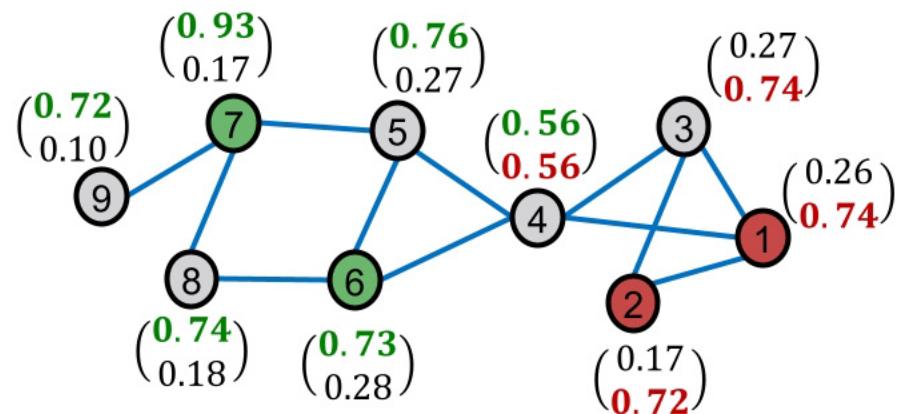
Before smoothing

$\mathbf{Z}^{(0)}$



After smoothing

$\mathbf{Z}^{(3)}, \alpha = 0.8$



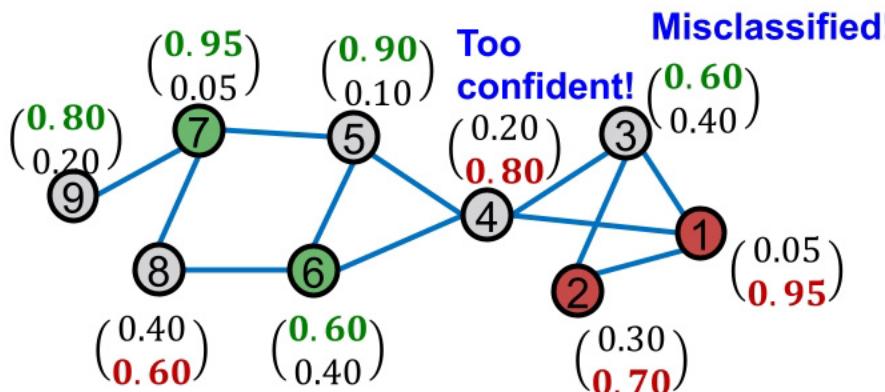
The final class prediction of C&S is the class with the maximum  $\mathbf{Z}^{(3)}$  score.

**Note:** The  $\mathbf{Z}^{(3)}$  scores do not have direct probabilistic interpretation (e.g., not sum to 1 for each node), but larger scores indicate the classes are more likely.

# C&S: Toy Example Summary

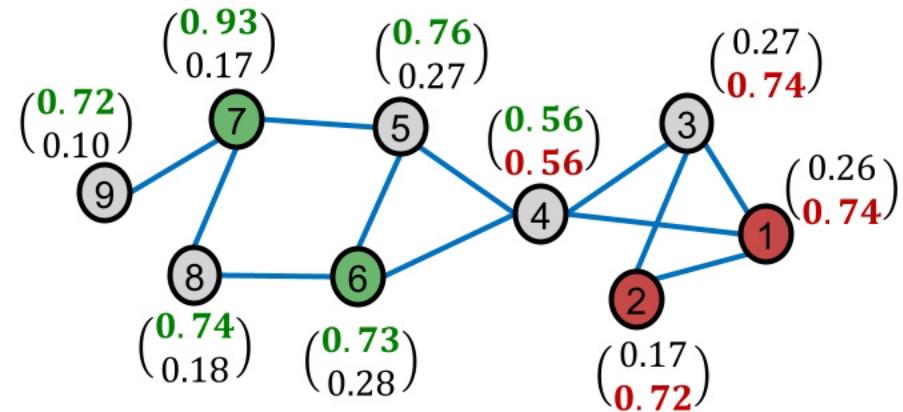
- Our toy example shows that C&S successfully improves base model performance using graph structure.

Prediction of the base model



Misclassified!

After C&S



# C&S on a Real-World Dataset

- C&S significantly improves the performance of the base model (MLP).
- C&S outperforms Smooth-only (no correct step) baseline.

Method	Classification accuracy (%) on ogbn-products dataset
MLP (base model)	63.41
MLP + smooth only	80.34
<b>MLP + C&amp;S</b>	<b>84.18</b>

# Correct & Smooth: Summary

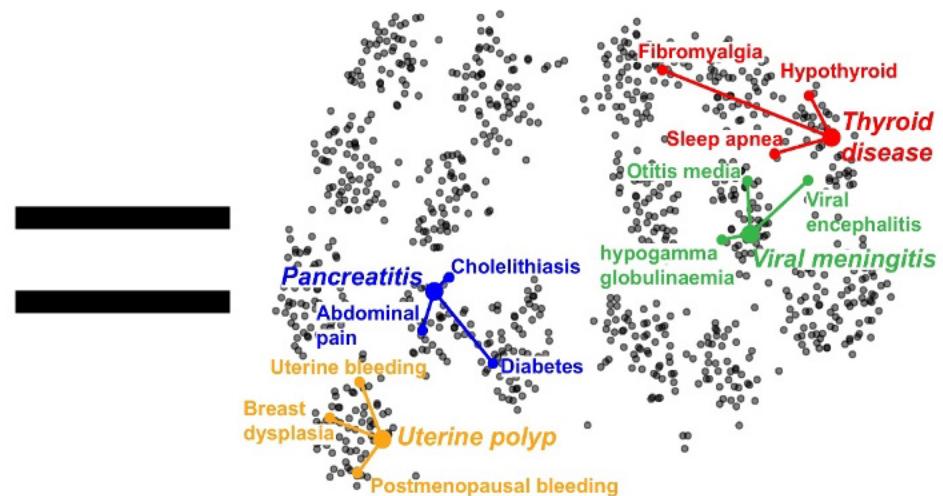
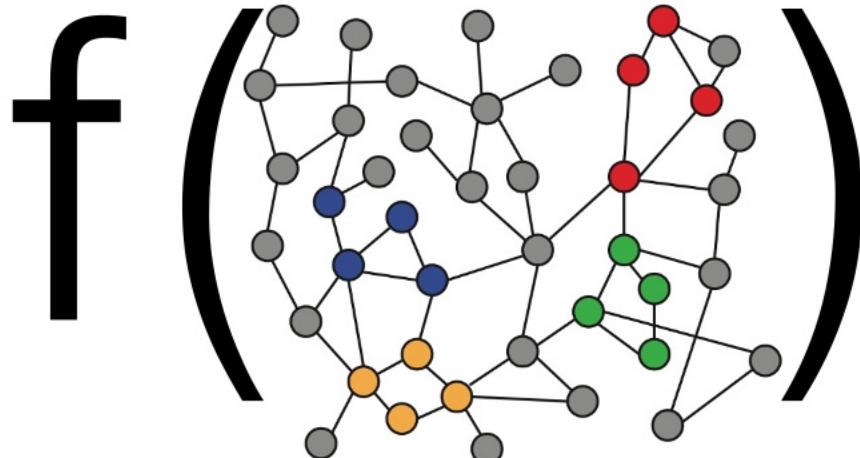


- Correct & Smooth (C&S) **uses graph structure to post-process** the soft node labels predicted by any base model.
- **Correction step:** Diffuse and correct for the training errors of the base predictor.
- **Smooth step:** Smoothen the prediction of the base predictor.
- C&S achieves strong performance on semi-supervised node classification.

# Graph Neural Networks

# Recap: Node Embeddings

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

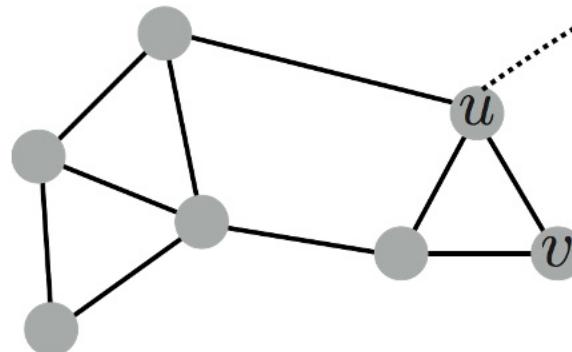


How to learn mapping function  $f$ ?

# Recap: Node Embeddings

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

Need to define!



encode nodes

$\text{ENC}(u)$

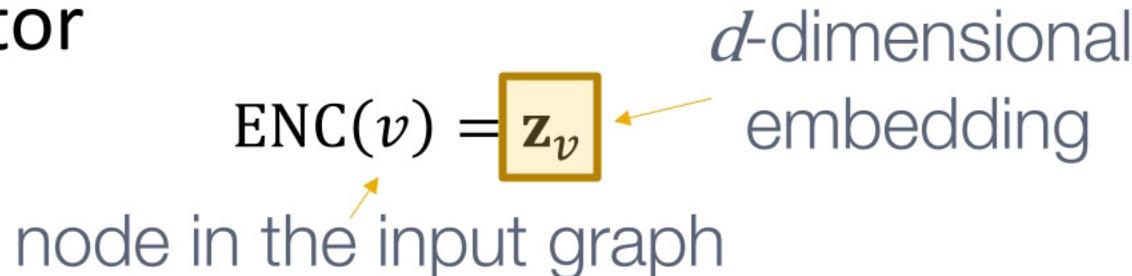
$\text{ENC}(v)$

Input network

d-dimensional  
embedding space

# Recap: Two Key Components

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



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

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

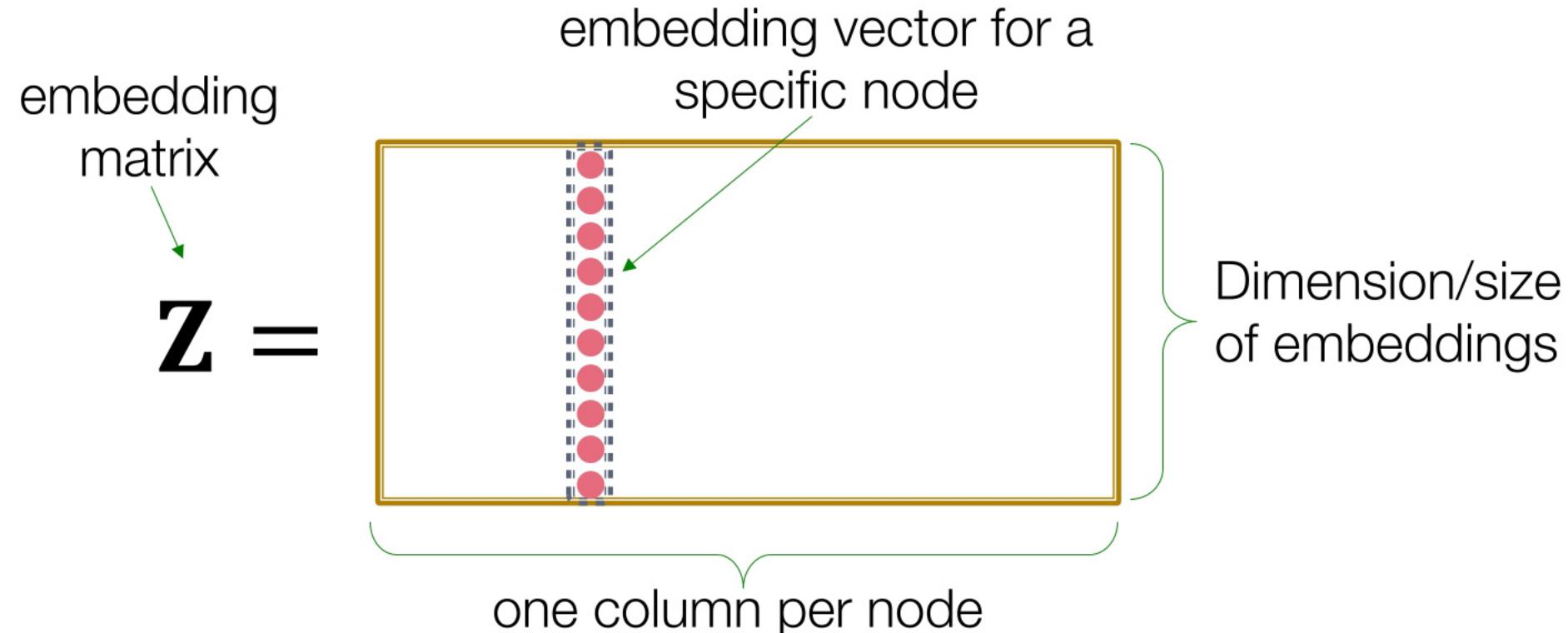
A diagram showing the similarity function. An arrow points from the text "Similarity of  $u$  and  $v$  in the original network" to the word "similarity". Another arrow points from the text "dot product between node embeddings" to the term  $\mathbf{z}_v^T \mathbf{z}_u$ . To the right of the equation, the word "Decoder" is written in red.

Similarity of  $u$  and  $v$  in  
the original network

dot product between node  
embeddings

# Recap: “Shallow” Encoding

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



# Recap: Shallow Encoders

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

# Today: Deep Graph Encoders

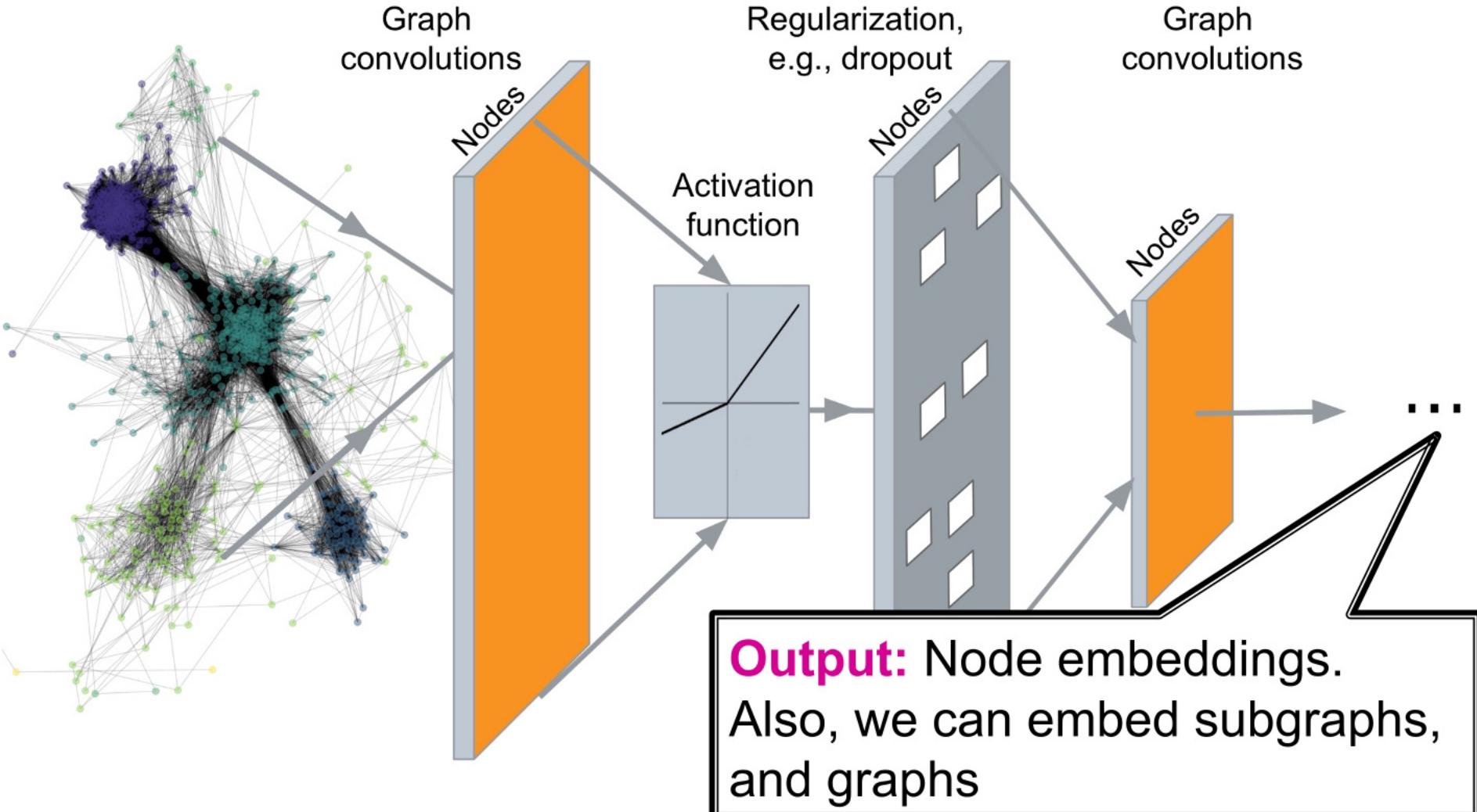
- **Today:** We will now discuss deep learning methods based on **graph neural networks (GNNs)**:

$$\text{ENC}(\nu) =$$

**multiple layers of  
non-linear transformations  
based on graph structure**

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

# Deep Graph Encoders

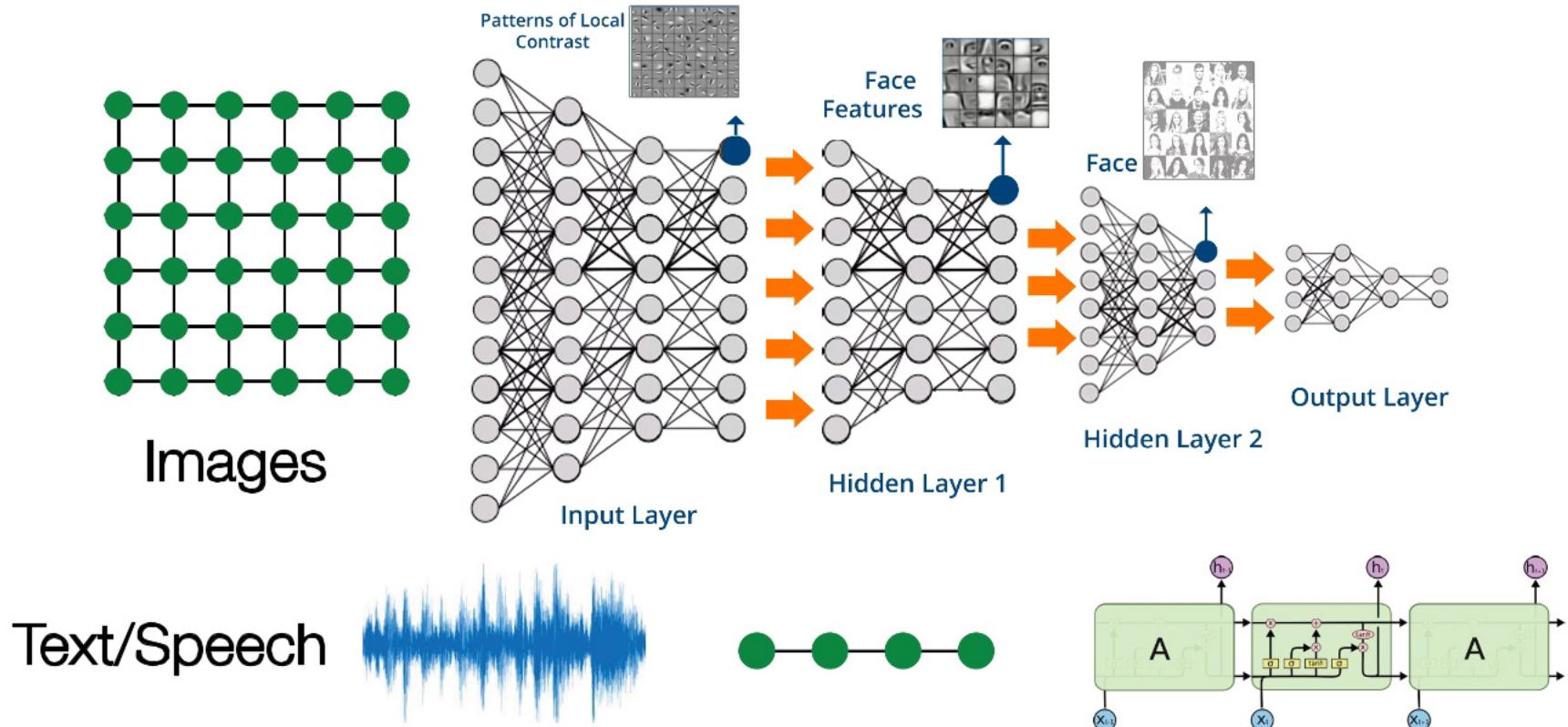


# Tasks on Networks

## Tasks we will be able to solve:

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

# Modern ML Toolbox

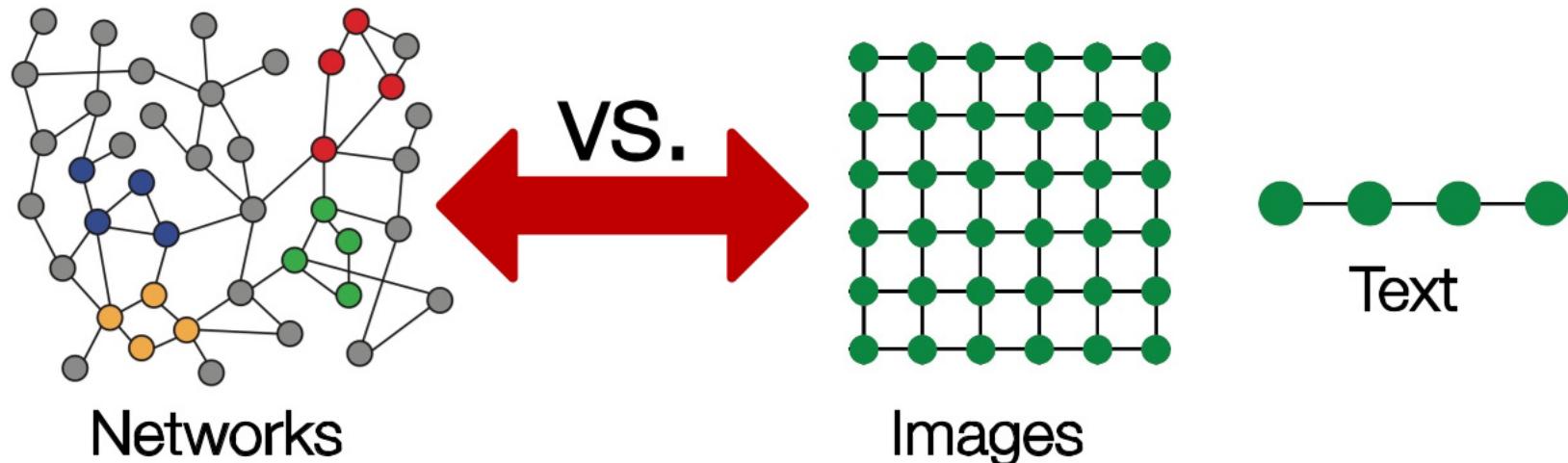


Modern deep learning toolbox is designed  
for simple sequences & grids

# Why is it Hard?

**But networks are far more complex!**

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



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

# Outline of Today's Lecture

**1. Basics of deep learning**



**2. Deep learning for graphs**

**3. Graph Convolutional Networks**

**4. GNNs subsume CNNs and  
Transformers**

# Basics of Deep Learning

# Machine Learning as Optimization

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

# Machine Learning as Optimization

- Formulate the task as an optimization problem:

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

Objective function

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

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

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

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

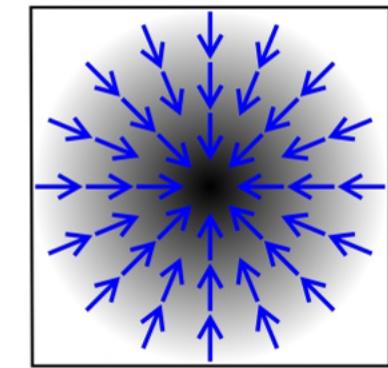
# Loss Function Example

- One common loss for classification: **cross entropy (CE)**
- Label  $\mathbf{y}$  is a categorical vector (**one-hot encoding**)
  - e.g.  $\mathbf{y} = \begin{array}{c|c|c|c|c} \text{o} & \text{o} & \text{1} & \text{o} & \text{o} \end{array}$   $\mathbf{y}$  is of class "3"
- $f(\mathbf{x}) = \text{Softmax}(g(\mathbf{x}))$ 
  - Recall from lecture 3:  $f(\mathbf{x})_i = \frac{e^{g(\mathbf{x})_i}}{\sum_{j=1}^C e^{g(\mathbf{x})_j}}$   $g(\mathbf{x})_i$  denotes  $i$ -th coordinate of the vector output of func.  $g(\mathbf{x})$
  - where  $C$  is the number of classes.
  - e.g.  $f(\mathbf{x}) = \begin{array}{c|c|c|c|c} \text{0.1} & \text{0.3} & \text{0.4} & \text{0.1} & \text{0.1} \end{array}$
- $\text{CE}(\mathbf{y}, f(\mathbf{x})) = -\sum_{i=1}^C (y_i \log f(\mathbf{x})_i)$ 
  - $y_i$  and  $f(\mathbf{x})_i$  are the **actual** and **predicted** values of the  $i$ -th class.
  - **Intuition:** the lower the loss, the closer the prediction is to one-hot
- **Total loss over all training examples:**
  - $\mathcal{L} = \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{T}} \text{CE}(\mathbf{y}, f(\mathbf{x}))$ 
    - $\mathcal{T}$ : training set containing all pairs of data and labels  $(\mathbf{x}, \mathbf{y})$

# Machine Learning as Optimization

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

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



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

- $\Theta_1, \Theta_2 \dots$  : components of  $\Theta$
- Recall **directional derivative** of a multi-variable function (e.g.  $\mathcal{L}$ ) along a given vector represents the instantaneous rate of change of the function along the vector.
- Gradient is the directional derivative in the **direction of largest increase**.

# Gradient Descent

- **Iterative algorithm:** repeatedly update weights in the (opposite) direction of gradients until convergence  
$$\Theta \leftarrow \Theta - \eta \nabla_{\Theta} \mathcal{L}$$
- **Training:** Optimize  $\Theta$  iteratively
  - **Iteration:** 1 step of gradient descent
- **Learning rate (LR)  $\eta$ :**
  - Hyperparameter that controls the size of gradient step
  - Can vary over the course of training (LR scheduling)
- **Ideal termination condition:** gradient =  $\mathbf{0}$ 
  - In practice, we stop training if it no longer improves performance on **validation set** (part of dataset we hold out from training).

# Stochastic Gradient Descent (SGD)

## ■ Problem with gradient descent:

- Exact gradient requires computing  $\nabla_{\Theta} \mathcal{L}(y, f(\mathbf{x}))$ , where  $\mathbf{x}$  is the **entire** dataset!
  - This means summing gradient contributions over all the points in the dataset
  - Modern datasets often contain billions of data points
  - Extremely expensive for every gradient descent step

## ■ Solution: Stochastic gradient descent (SGD)

- At every step, pick a different **minibatch**  $\mathcal{B}$  containing a subset of the dataset, use it as input  $\mathbf{x}$

# Minibatch SGD

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

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

# Neural Network Function

- **Objective:**  $\min_{\Theta} \mathcal{L}(y, f(\mathbf{x}))$
- In deep learning, function  $f$  can be very complex
- **Example:**
  - To start simple, consider linear function  
$$f(\mathbf{x}) = \mathbf{W} \cdot \mathbf{x}, \quad \Theta = \{\mathbf{W}\}$$
  - Then, if  $f$  returns a scalar, then  $\mathbf{W}$  is a learnable **vector**  
$$\nabla_{\mathbf{W}} f = \left( \frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}, \frac{\partial f}{\partial w_3}, \dots \right)$$
  - But, if  $f$  returns a vector, then  $\mathbf{W}$  is the **weight matrix**

$$\nabla_{\mathbf{W}} f = \begin{bmatrix} \frac{\partial f_1}{\partial w_{11}} & \frac{\partial f_2}{\partial w_{12}} \\ \frac{\partial f_1}{\partial w_{21}} & \frac{\partial f_2}{\partial w_{22}} \end{bmatrix}$$

 **Jacobian**  
matrix of  $f$

# Intuition: Back Propagation

- **Goal:**  $\min_{\Theta} \mathcal{L}(y, f(\mathbf{x}))$

- To minimize  $\mathcal{L}$ , we need to evaluate the gradient:

$$\nabla_{\mathbf{W}} \mathcal{L} = \left( \frac{\partial f}{\partial \mathbf{w}_1}, \frac{\partial f}{\partial \mathbf{w}_2}, \frac{\partial f}{\partial \mathbf{w}_3} \dots \right)$$

which means we need to derive derivative of  $\mathcal{L}$ .

- **Overview of Back-propagation:**

- $\mathcal{L}$  is composed from some set of predefined building block functions  $g(\cdot)$
  - For each such  $g$  we also have its derivative  $g'$
  - Then we can automatically compute  $\nabla_{\Theta} \mathcal{L}$  by evaluating appropriate funcs.  $g'$  on the minibatch  $\mathcal{B}$ .

# Back-propagation

- How about a more complex function:

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

- Recall chain rule:

$$\frac{df}{dx} = \frac{dg}{dh} \cdot \frac{dh}{dx} \text{ or } f'(x) = g'(h(x))h'(x)$$

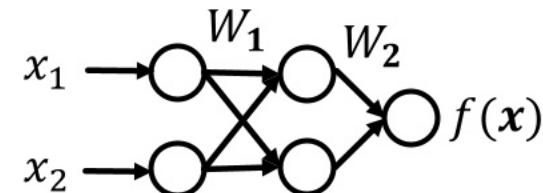
- Example:  $\nabla_{\mathbf{x}} f = \frac{\partial f}{\partial (W_1 \mathbf{x})} \cdot \frac{\partial (W_1 \mathbf{x})}{\partial \mathbf{x}}$

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

In other words:  
 $f(\mathbf{x}) = W_2(W_1 \mathbf{x})$   
 $h(x) = W_1 \mathbf{x}$   
 $g(z) = W_2 z$

# Back-propagation Example (1)

- **Example:** Simple 2-layer linear network
- $f(\mathbf{x}) = g(h(x)) = W_2(W_1 \mathbf{x})$
- $\mathcal{L} = \sum_{(x,y) \in \mathcal{B}} \left\| (y, -f(x)) \right\|_2$ 
  - The loss  $\mathcal{L}$  sums the L2 loss in a minibatch  $\mathcal{B}$ .
- **Hidden layer:**
  - Intermediate representation of input  $\mathbf{x}$
  - Here we use  $h(x) = W_1 \mathbf{x}$  to denote the hidden layer
  - $f(\mathbf{x}) = W_2 h(\mathbf{x})$



# Back-propagation Example (2)

## ■ Forward propagation:

Compute loss starting from input

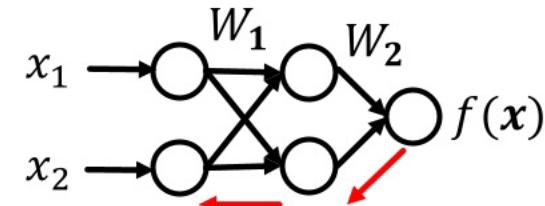


Remember:

$$f(x) = W_2(W_1 x)$$

$$h(x) = W_1 x$$

$$g(z) = W_2 z$$



## ■ Back-propagation to compute gradient of

$$\Theta = \{W_1, W_2\}$$

Start from loss, compute the gradient

$$\frac{\partial \mathcal{L}}{\partial W_2} = \underbrace{\frac{\partial \mathcal{L}}{\partial f} \cdot \frac{\partial f}{\partial W_2}}_{\text{Compute backwards}},$$

$$\frac{\partial \mathcal{L}}{\partial W_1} = \underbrace{\frac{\partial \mathcal{L}}{\partial f} \cdot \frac{\partial f}{\partial W_2} \cdot \frac{\partial W_2}{\partial W_1}}_{\text{Compute backwards}}$$

Compute backwards

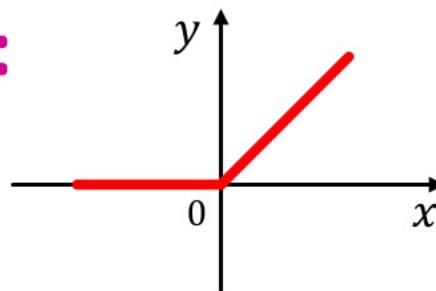
Compute backwards

# Non-linearity

- Note that in  $f(\mathbf{x}) = W_2(W_1 \mathbf{x})$ ,  $W_2 W_1$  is another matrix (vector, if we do binary classification)
  - Hence  $f(\mathbf{x})$  is still linear w.r.t.  $\mathbf{x}$  no matter how many weight matrices we compose
- We introduce non-linearity:

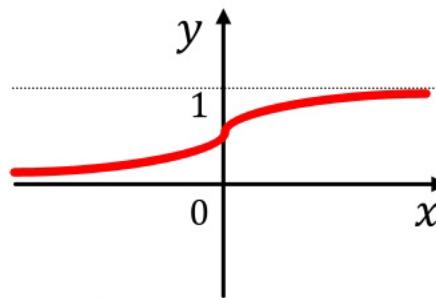
- Rectified linear unit (ReLU)

$$\text{ReLU}(x) = \max(x, 0)$$



- Sigmoid

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

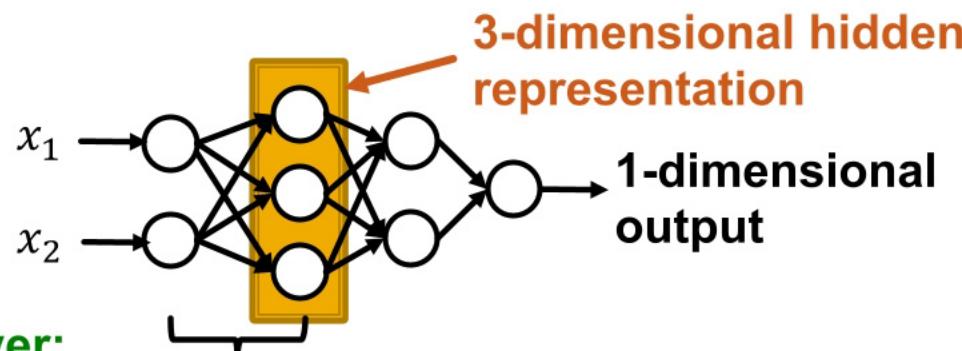


# Multi-layer Perceptron (MLP)

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

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

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



Every layer:  
Linear transformation +  
non-linearity

# Summary

- **Objective function:**

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

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

# Outline of Today's Lecture

1. Basics of deep learning



2. Deep learning for graphs



3. Graph Convolutional Networks

4. GNNs subsume CNNs and  
Transformers

# Deep Learning for Graphs

# Content

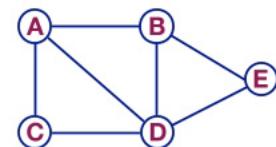
- **Local network neighborhoods:**
  - Describe aggregation strategies
  - Define computation graphs
- **Stacking multiple layers:**
  - Describe the model, parameters, training
  - How to fit the model?
  - Simple example for unsupervised and supervised training

# Setup

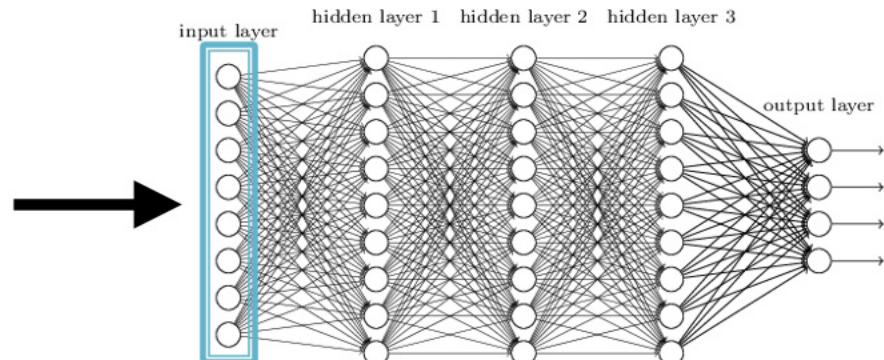
- Assume we have a graph  $G$ :
  - $V$  is the **vertex set**
  - $A$  is the **adjacency matrix** (assume binary)
  - $X \in \mathbb{R}^{m \times |V|}$  is a matrix of **node features**
  - $v$ : a node in  $V$ ;  $N(v)$ : the set of neighbors of  $v$ .
  - **Node features:**
    - Social networks: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

# A Naïve Approach

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



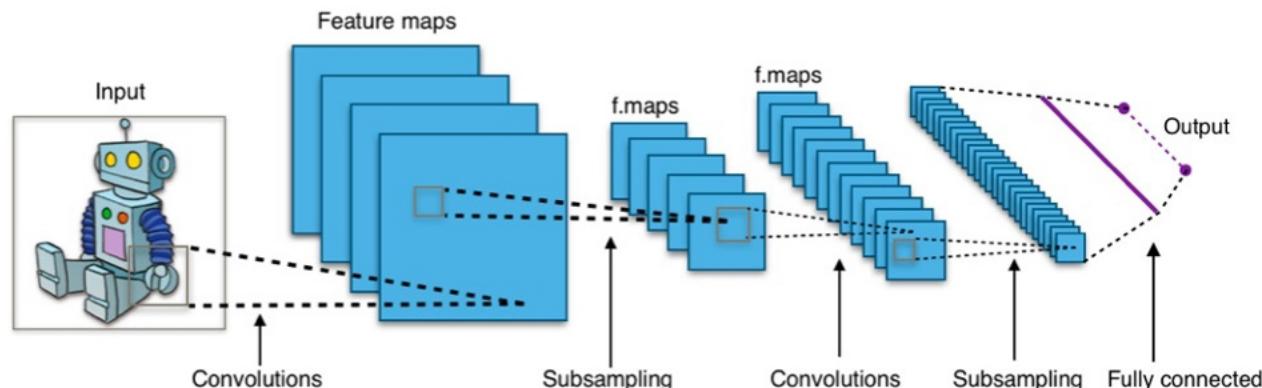
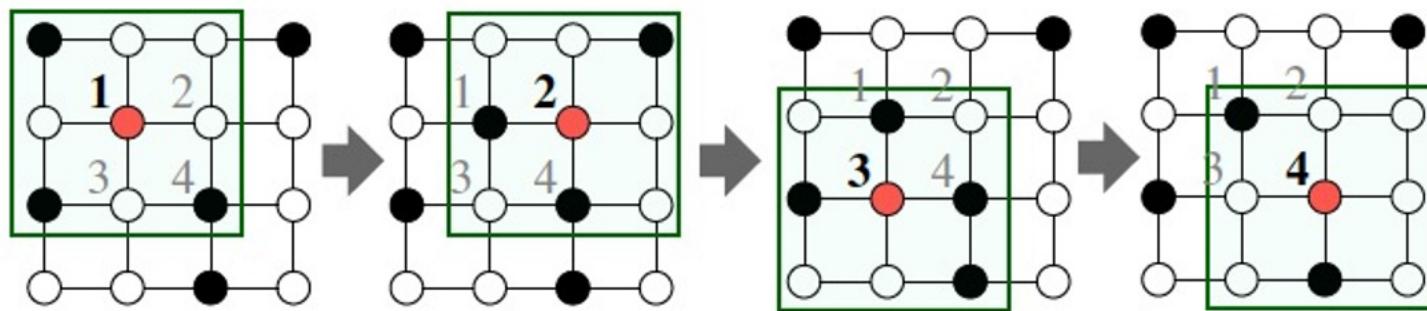
	A	B	C	D	E	Feat
A	0	1	1	1	0	1 0
B	1	0	0	1	1	0 0
C	1	0	0	1	0	0 1
D	1	1	1	0	1	1 1
E	0	1	0	1	0	1 0



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

# Idea: Convolutional Networks

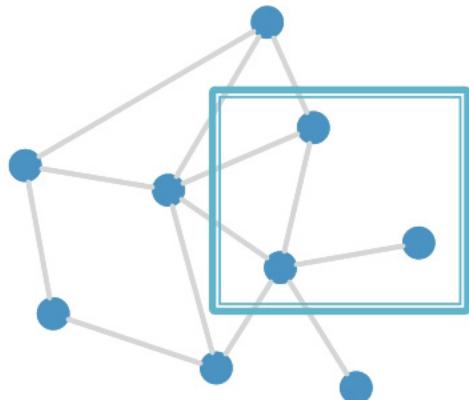
CNN on an image:



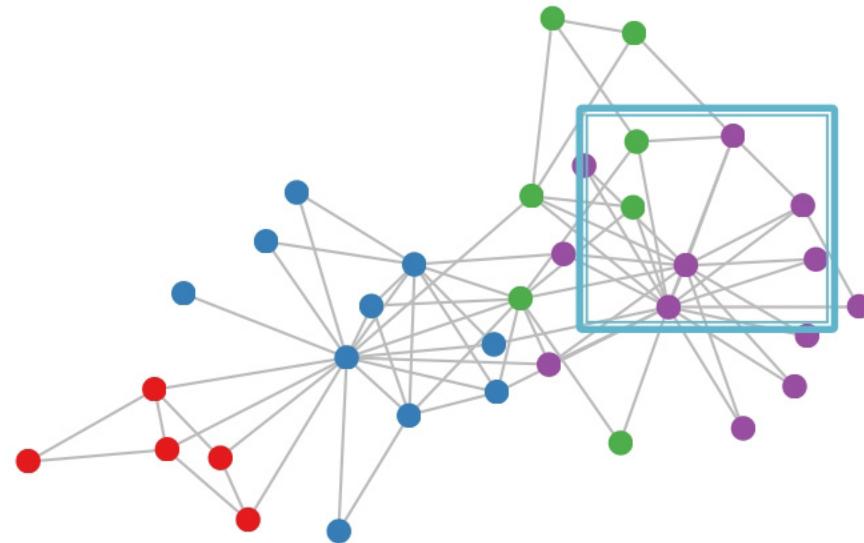
Goal is to generalize convolutions beyond simple lattices  
Leverage node features/attributes (e.g., text, images)

# Real-World Graphs

But our graphs look like this:



or this:



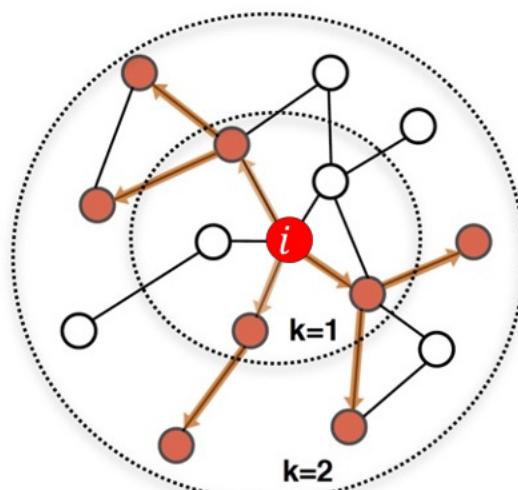
- There is no fixed notion of locality or sliding window on the graph
- Graph is permutation invariant

# Permutation Invariance

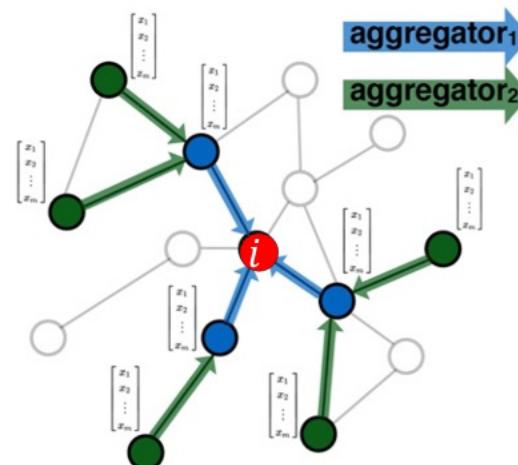
- Graph does not have a canonical order of the nodes!
- We can have many different order plans.

# Graph Convolutional Networks

**Idea:** Node's neighborhood defines a computation graph



Determine node computation graph

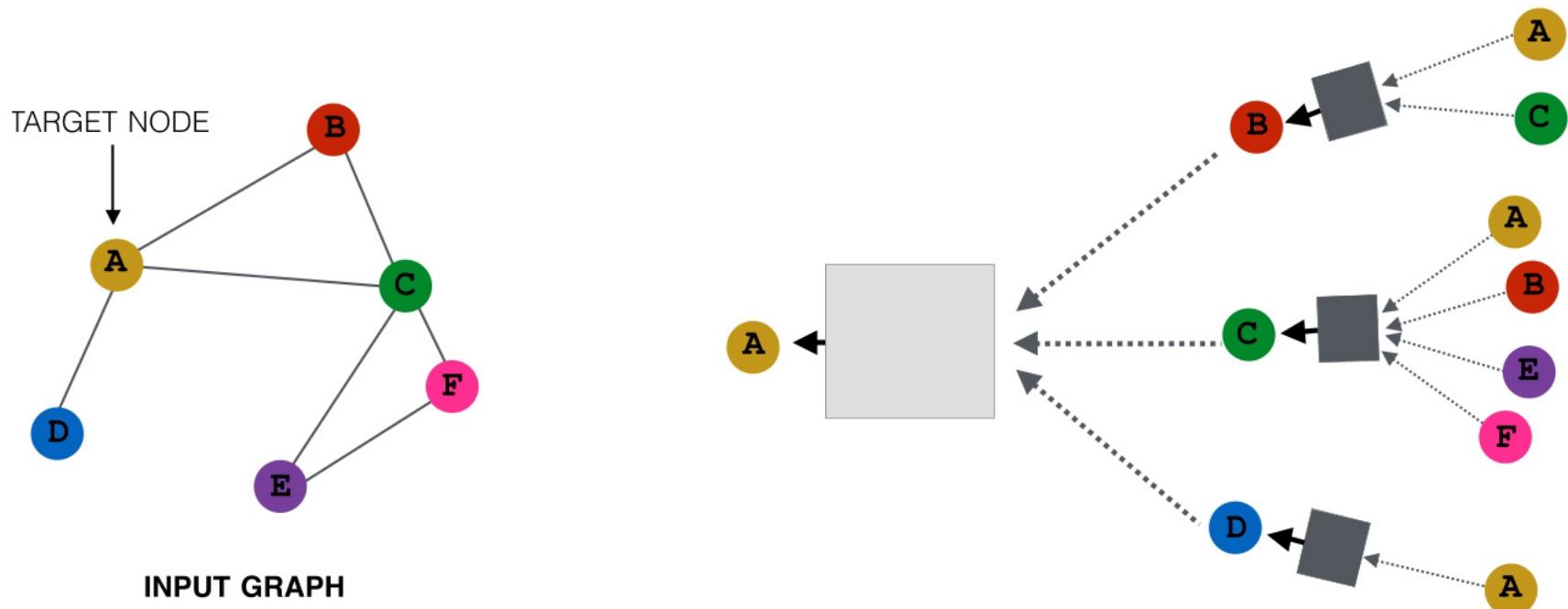


Propagate and transform information

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

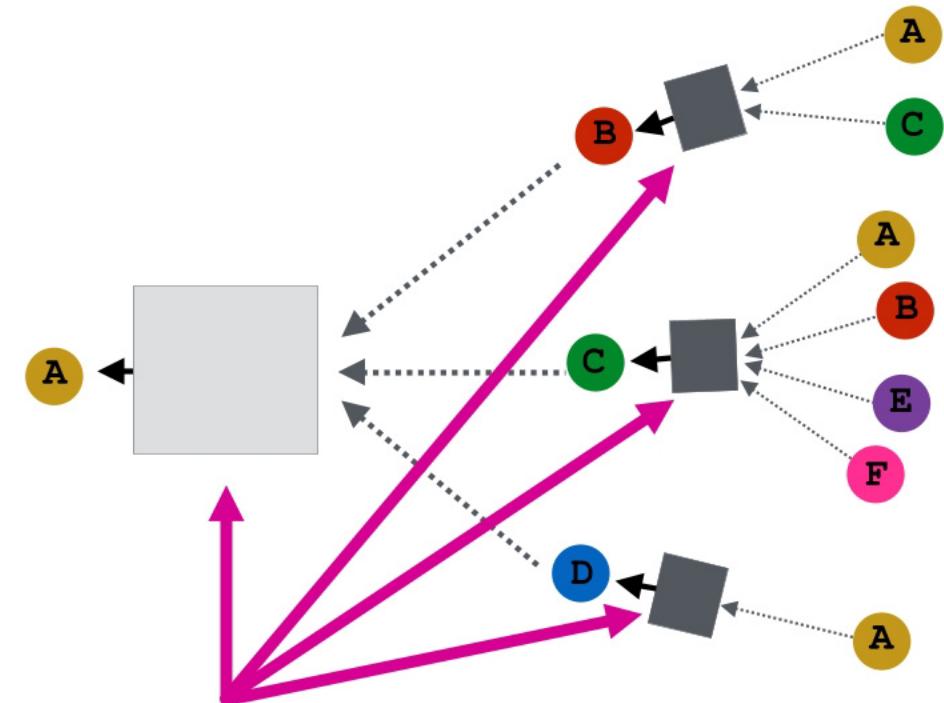
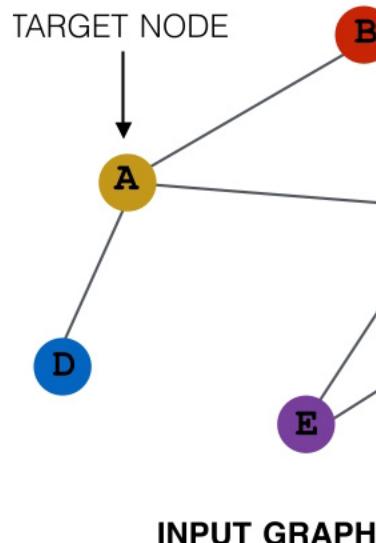
# Idea: Aggregate Neighbors

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



# Idea: Aggregate Neighbors

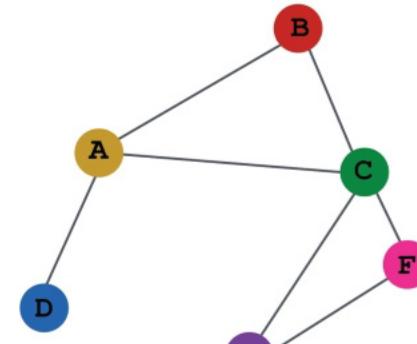
- **Intuition:** Nodes aggregate information from their neighbors using neural networks



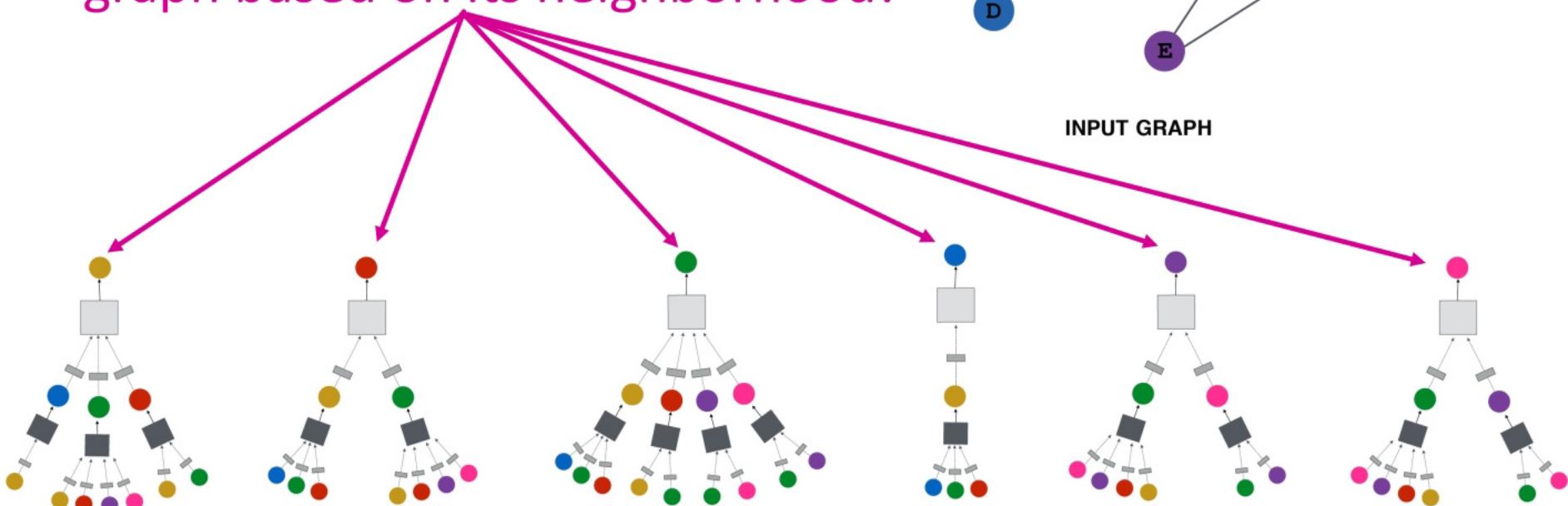
# Idea: Aggregate Neighbors

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

Every node defines a computation graph based on its neighborhood!

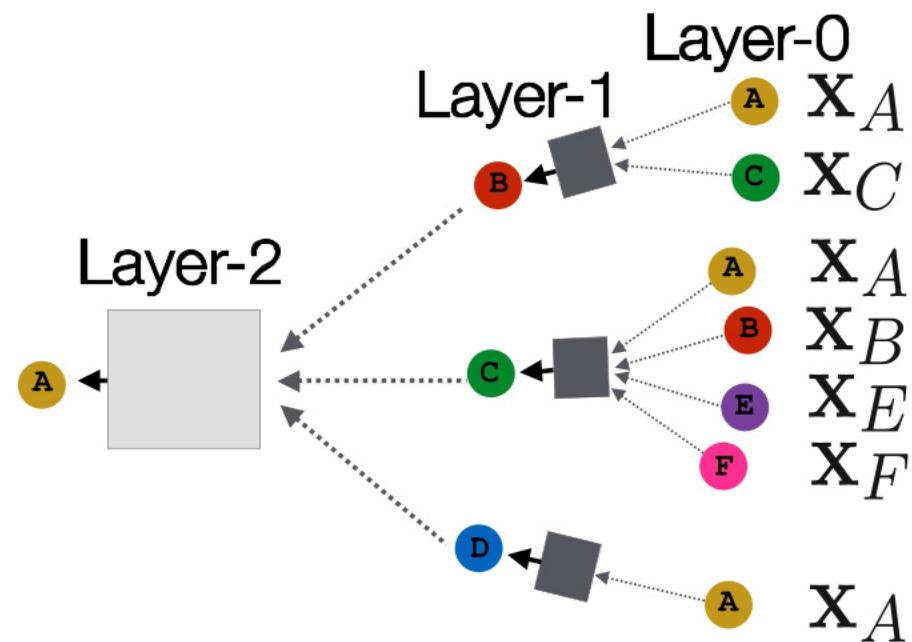
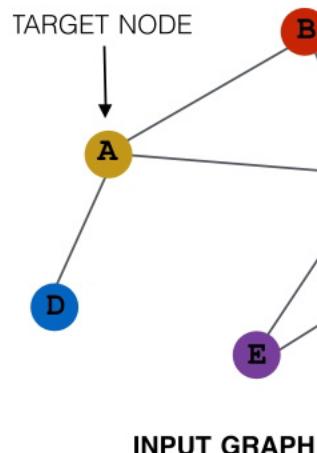


INPUT GRAPH



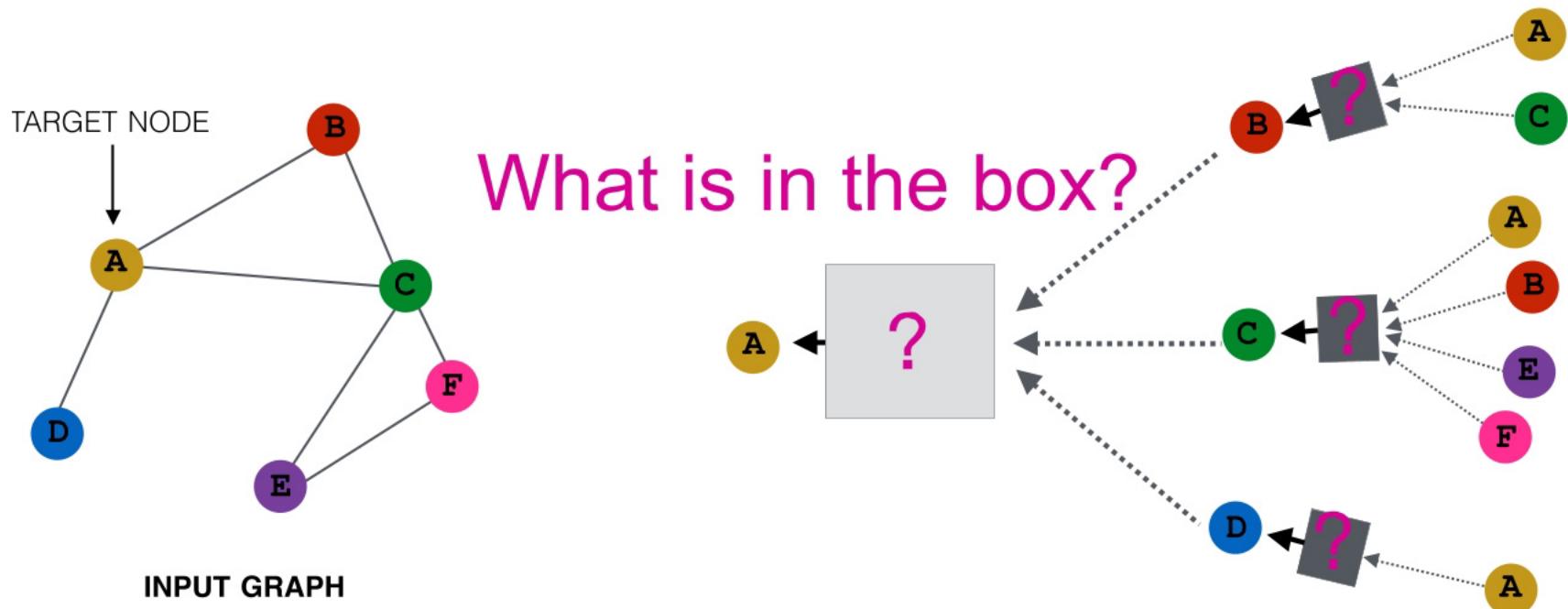
# Deep Model: Many Layers

- 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



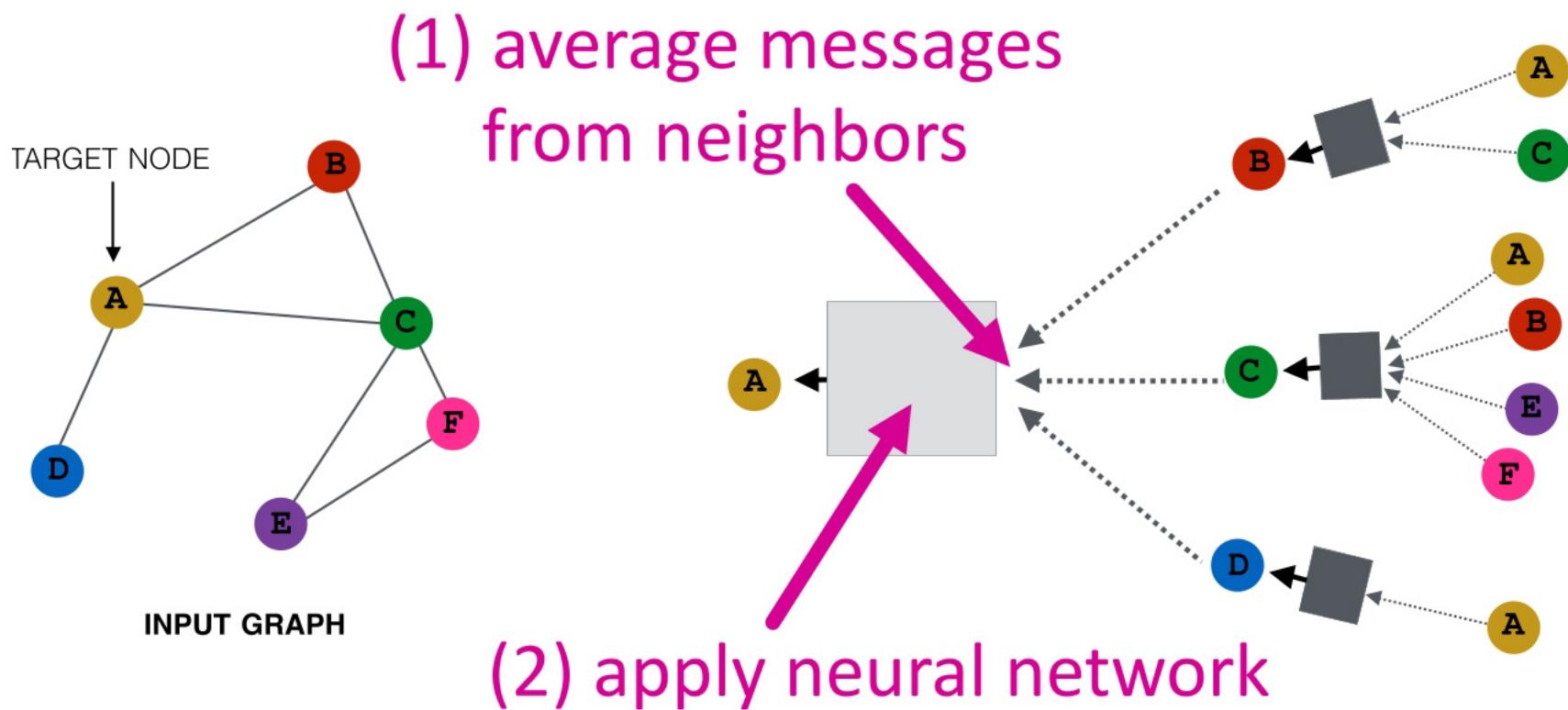
# Neighborhood Aggregation

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



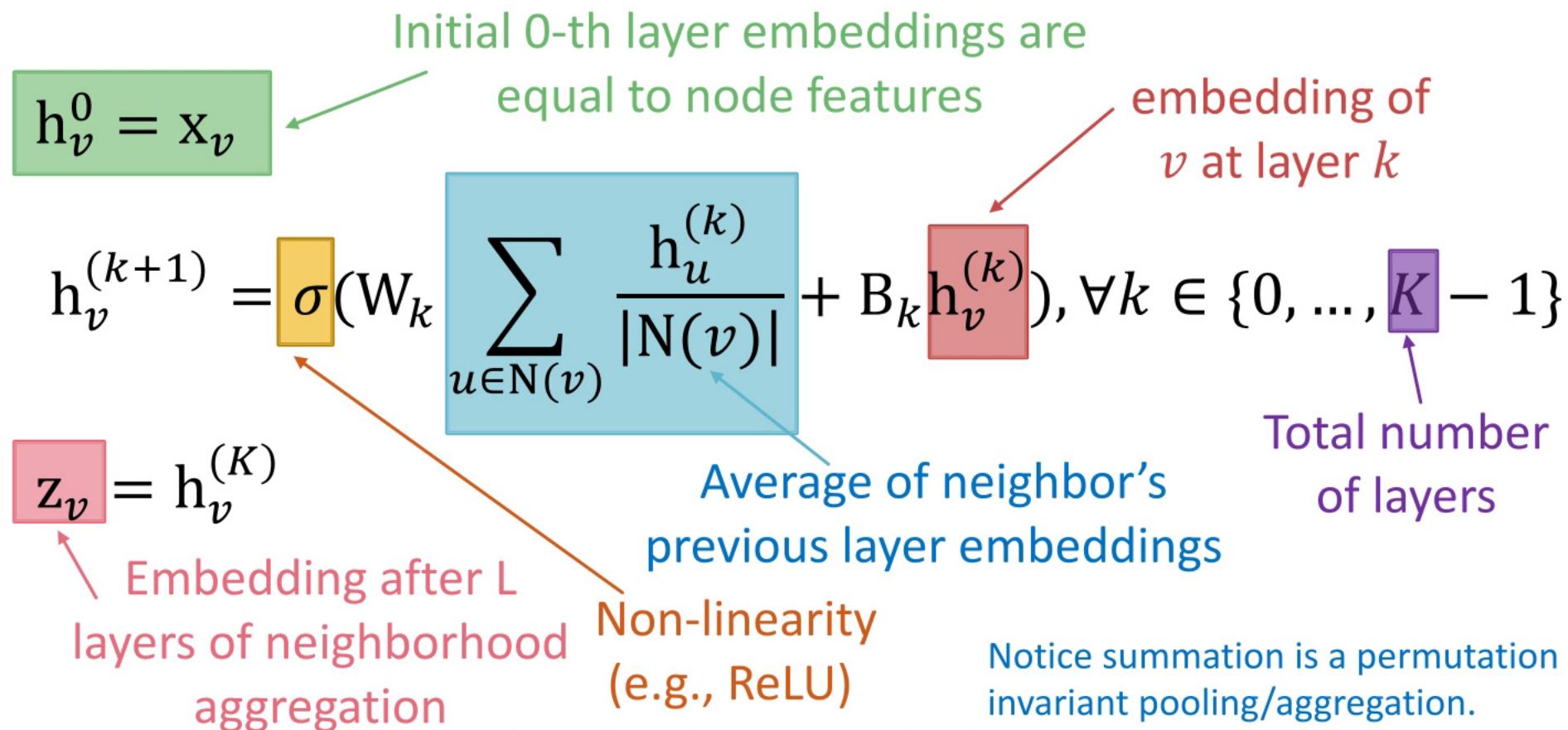
# Neighborhood Aggregation

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



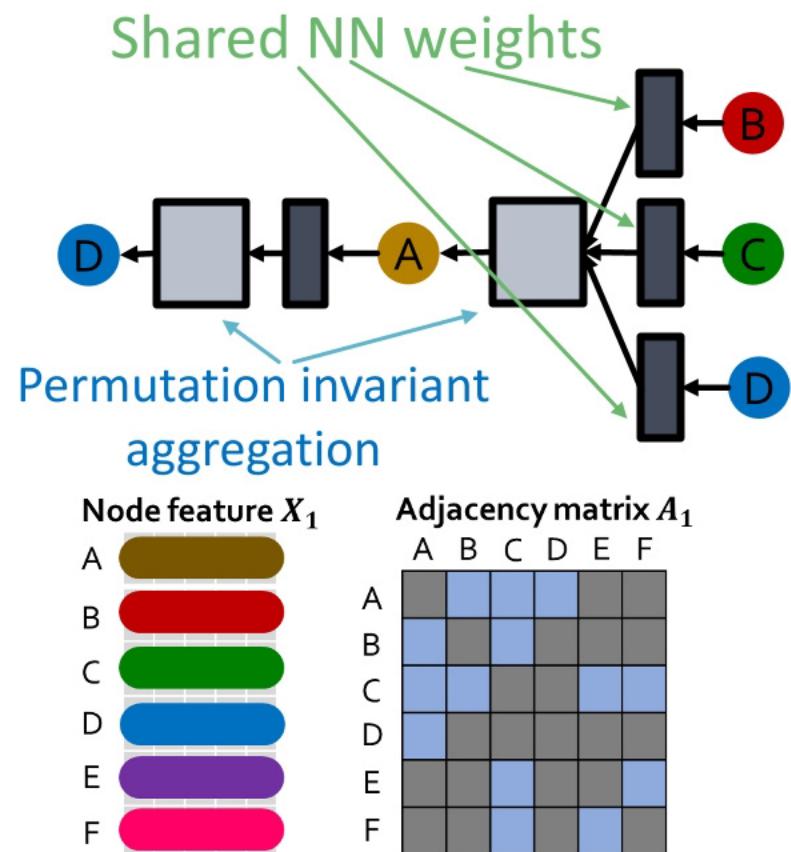
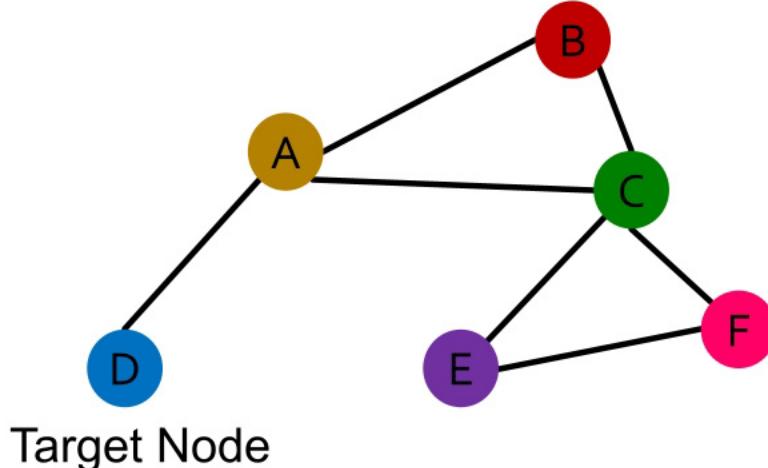
# The Math: Deep Encoder

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



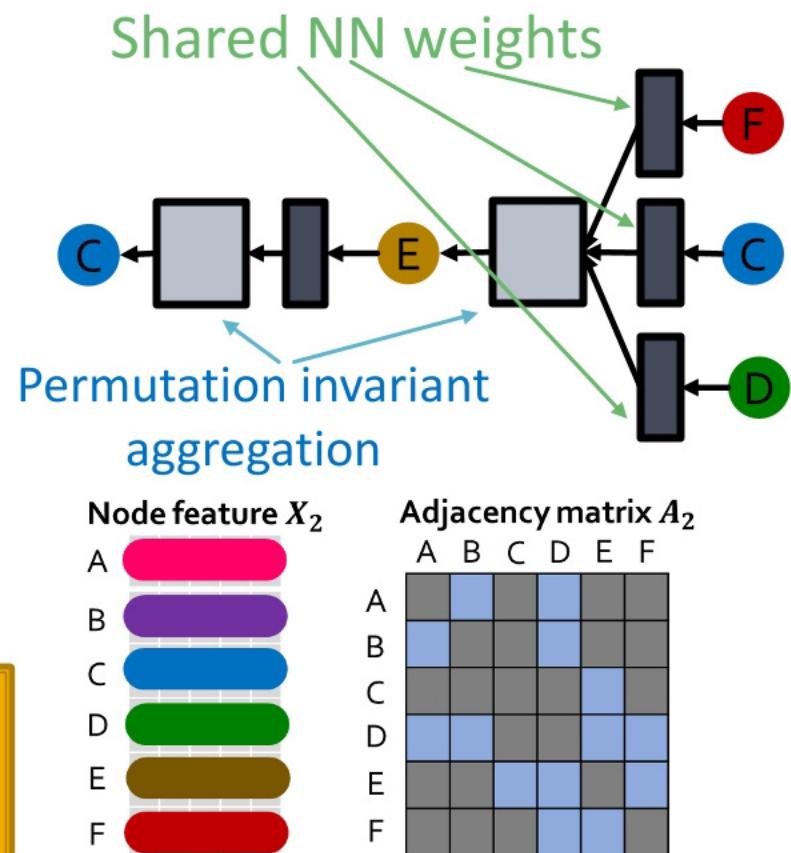
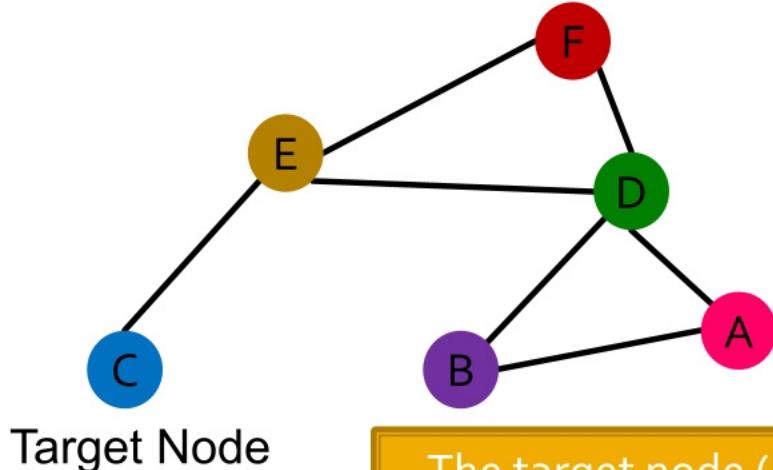
# Equivariant Property

Message passing and neighbor aggregation in graph convolution networks is permutation equivariant.



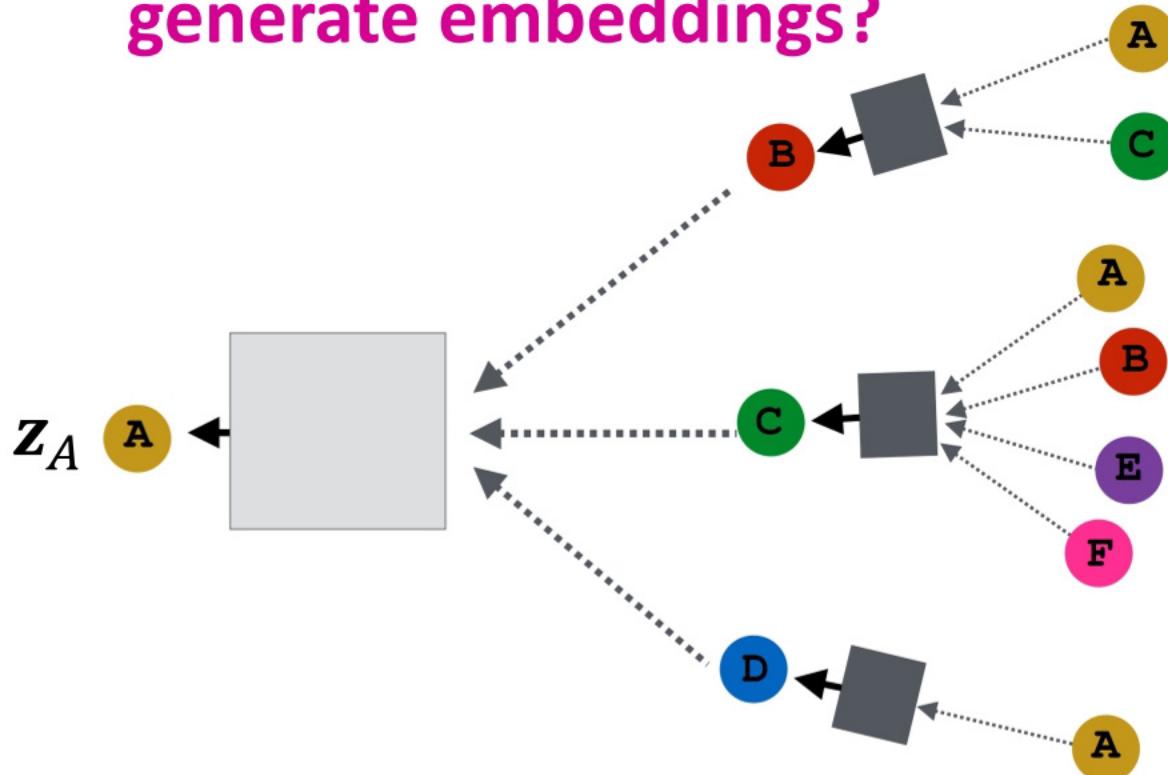
# Equivariant Property

Message passing and neighbor aggregation in graph convolution networks is permutation equivariant.



# Training the Model

How do we train the GCN to generate embeddings?



Need to define a loss function on the embeddings.

# Model Parameters

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

$$\begin{aligned} h_v^{(0)} &= x_v \\ h_v^{(k+1)} &= \sigma(W_k \sum_{u \in N(v)} \frac{h_u^{(k)}}{|N(v)|} + B_k h_v^{(k)}), \forall k \in \{0..K-1\} \\ z_v &= h_v^{(K)} \end{aligned}$$

Final node embedding

```
graph TD; x_v[x_v] --> h0[h_v^(0)]; h0 --> hkplus1[h_v^(k+1)]; hkplus1 --> zk[z_v];
```

The diagram shows the flow of information from the initial node features  $x_v$  through  $K$  layers of processing to the final node embedding  $z_v$ . Each layer  $k$  takes the hidden representation  $h_v^{(k)}$  from the previous layer and applies a transformation involving a weight matrix  $W_k$  and a bias  $B_k$  to produce the next hidden representation  $h_v^{(k+1)}$ .

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

# Matrix Formulation (1)

- Many aggregations can be performed efficiently by (sparse) matrix operations

- Let  $H^{(k)} = [h_1^{(k)} \dots h_{|V|}^{(k)}]^T$
- Then:  $\sum_{u \in N_v} h_u^{(k)} = A_{v,:} H^{(k)}$

- Let  $D$  be diagonal matrix where

$$D_{v,v} = \text{Deg}(v) = |N(v)|$$

- The inverse of  $D$ :  $D^{-1}$  is also diagonal:

$$D_{v,v}^{-1} = 1/|N(v)|$$

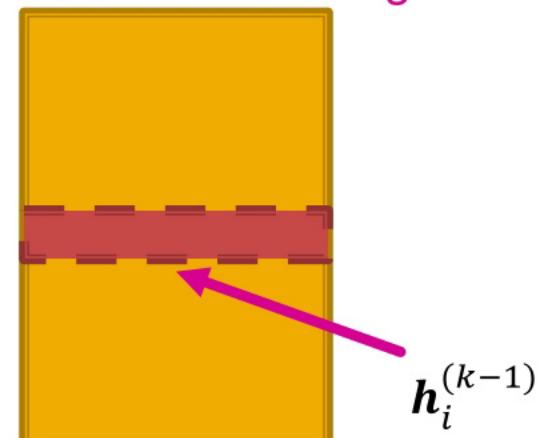
- Therefore,

$$\sum_{u \in N(v)} \frac{h_u^{(k-1)}}{|N(v)|}$$



$$H^{(k+1)} = D^{-1} A H^{(k)}$$

Matrix of hidden embeddings  $H^{(k-1)}$

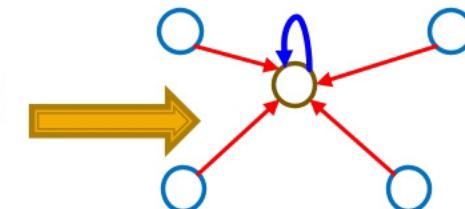


# Matrix Formulation (2)

- Re-writing update function in matrix form:

$$H^{(k+1)} = \sigma(\tilde{A}H^{(k)}W_k^T + H^{(k)}B_k^T)$$

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



$$H^{(k)} = [h_1^{(k)} \dots h_{|V|}^{(k)}]^T$$

- Red: neighborhood aggregation
- Blue: self transformation
- In practice, this implies that efficient sparse matrix multiplication can be used ( $\tilde{A}$  is sparse)
- **Note:** not all GNNs can be expressed in matrix form, when aggregation function is complex

# How to Train A GNN

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

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

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

# Unsupervised Training

- “Similar” nodes have similar embeddings

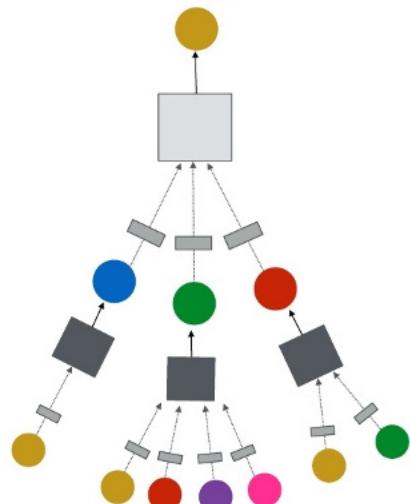
$$\mathcal{L} = \sum_{z_u, z_v} \text{CE}(y_{u,v}, \text{DEC}(z_u, z_v))$$

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

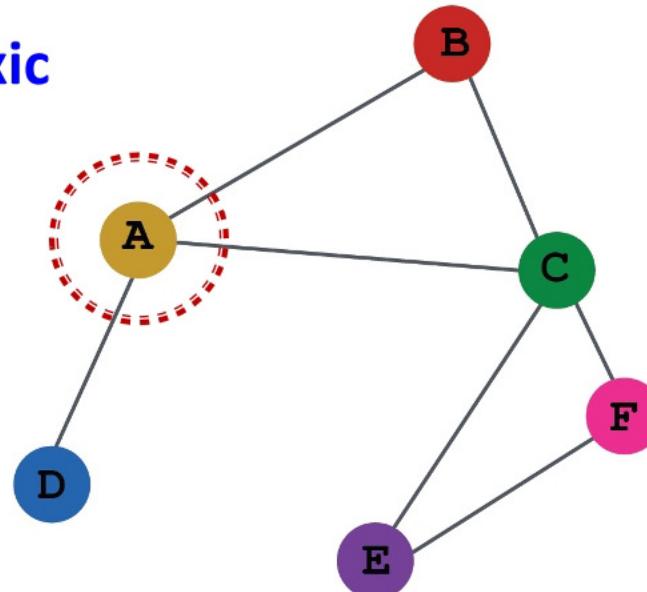
# Supervised Training

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

Safe or toxic  
drug?



Safe or toxic  
drug?

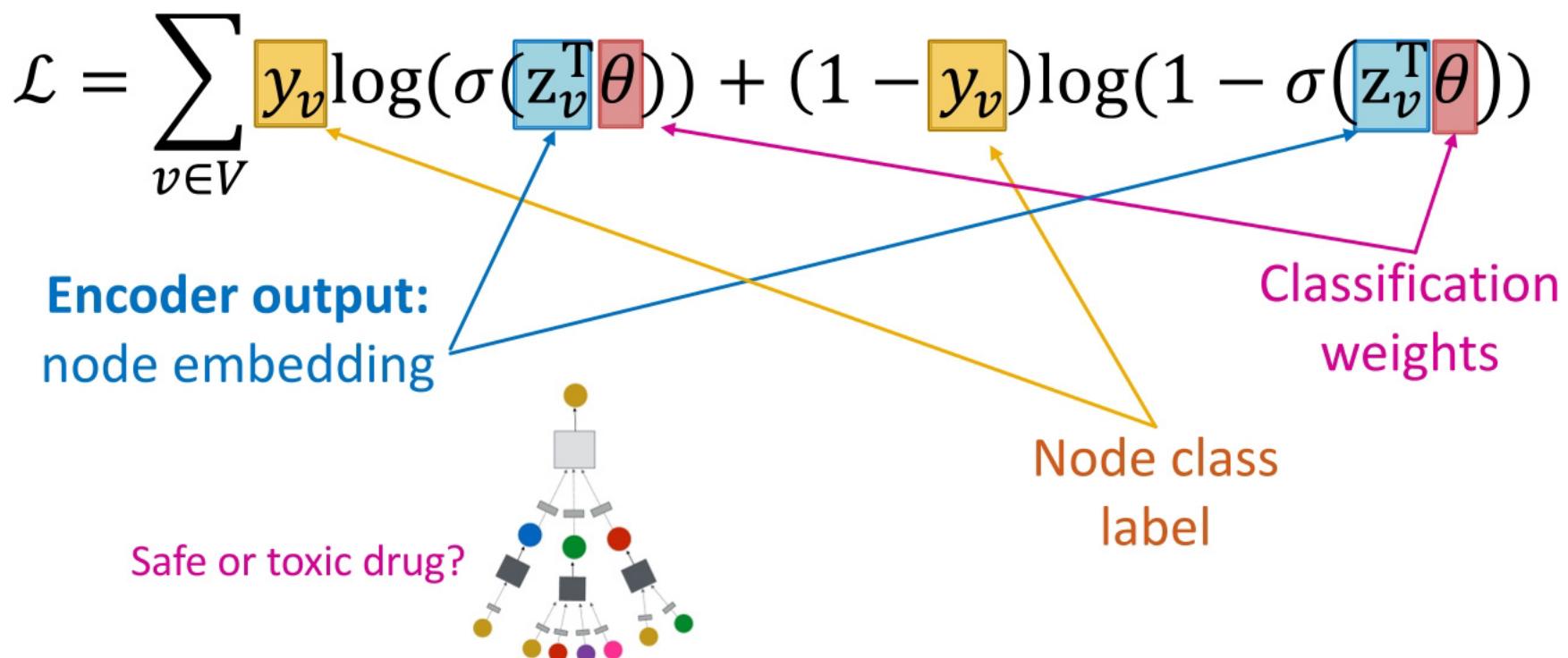


E.g., a drug-drug  
interaction network

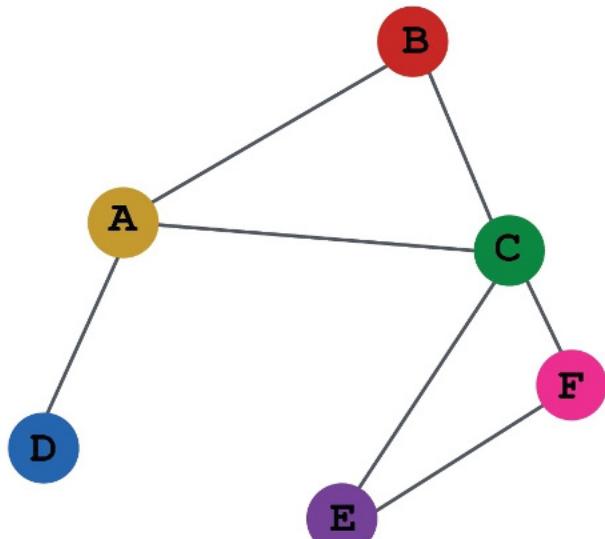
# Supervised Training

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

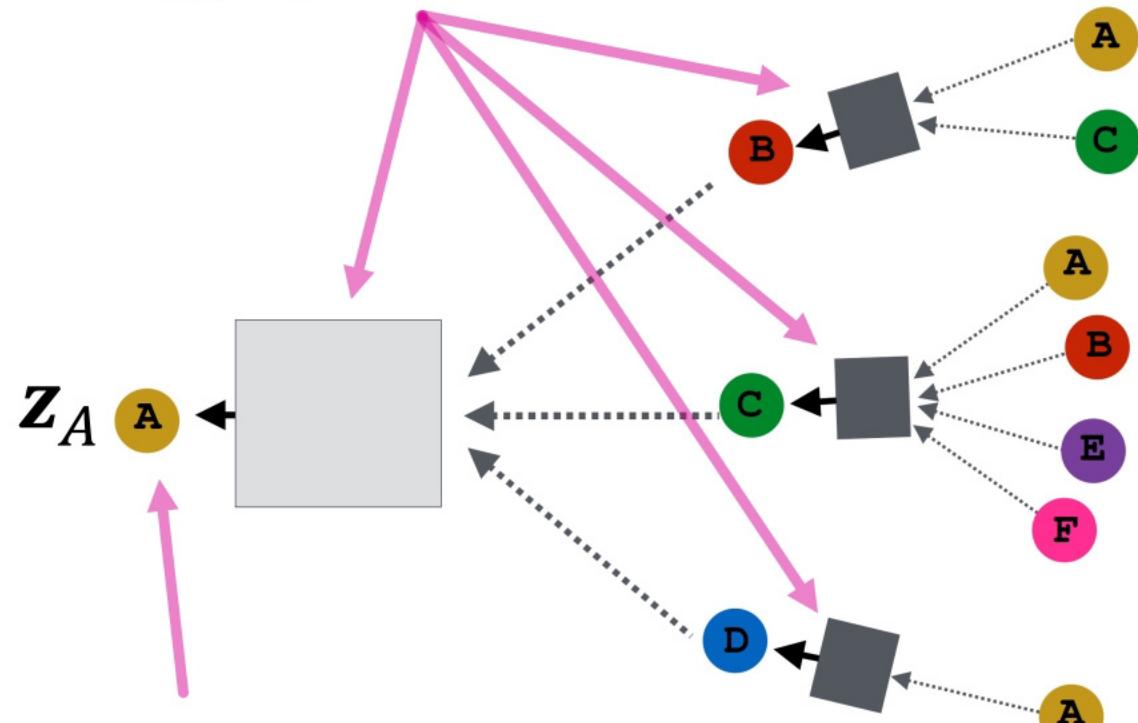
- Use cross entropy loss (Slide 16)



# Model Design: Overview

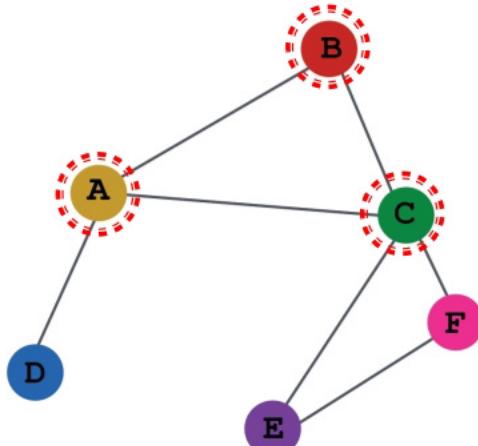


(1) Define a neighborhood aggregation function



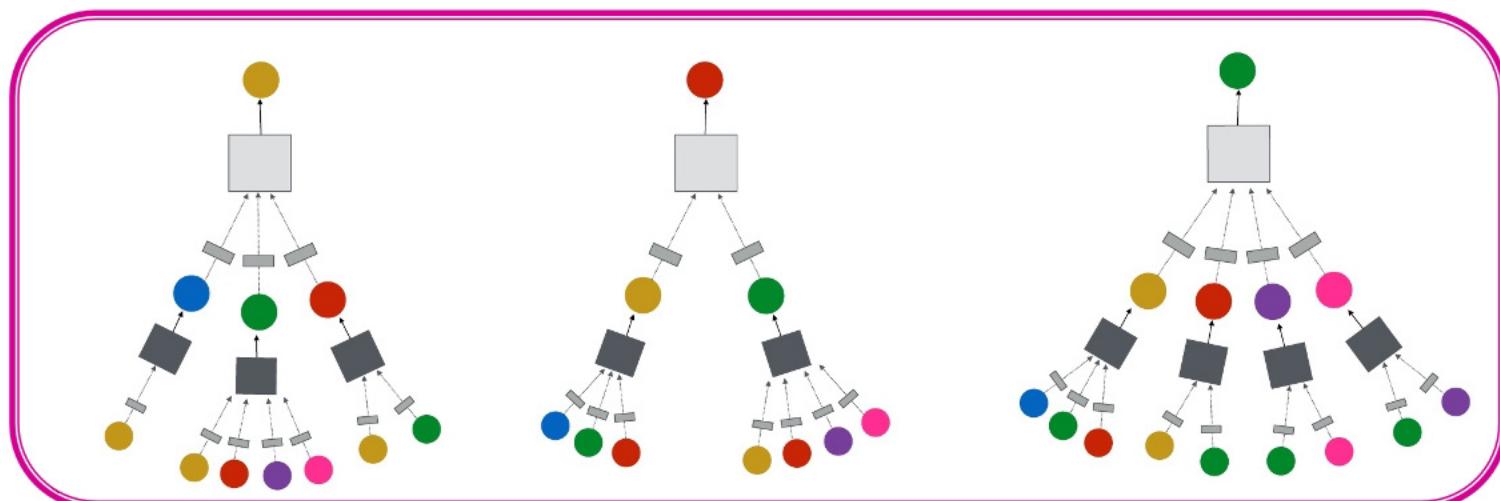
(2) Define a loss function on the embeddings

# Model Design: Overview

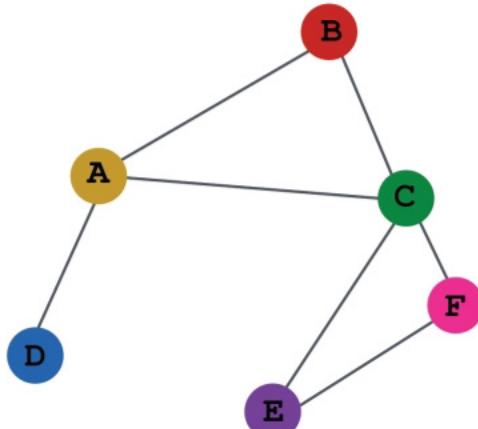


INPUT GRAPH

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



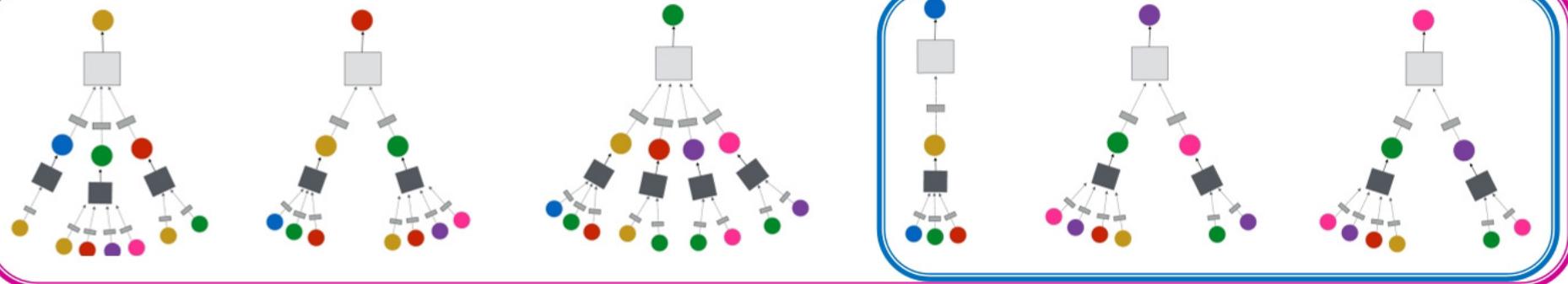
# Model Design: Overview



INPUT GRAPH

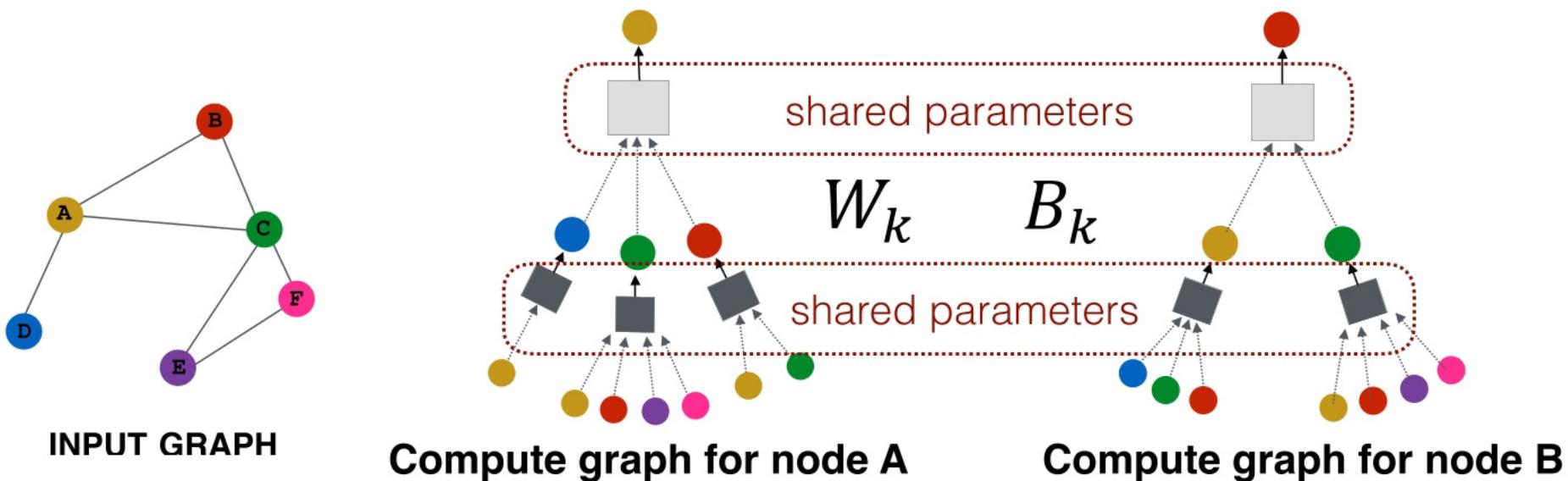
(4) Generate embeddings  
for nodes as needed

Even for nodes we never  
trained on!

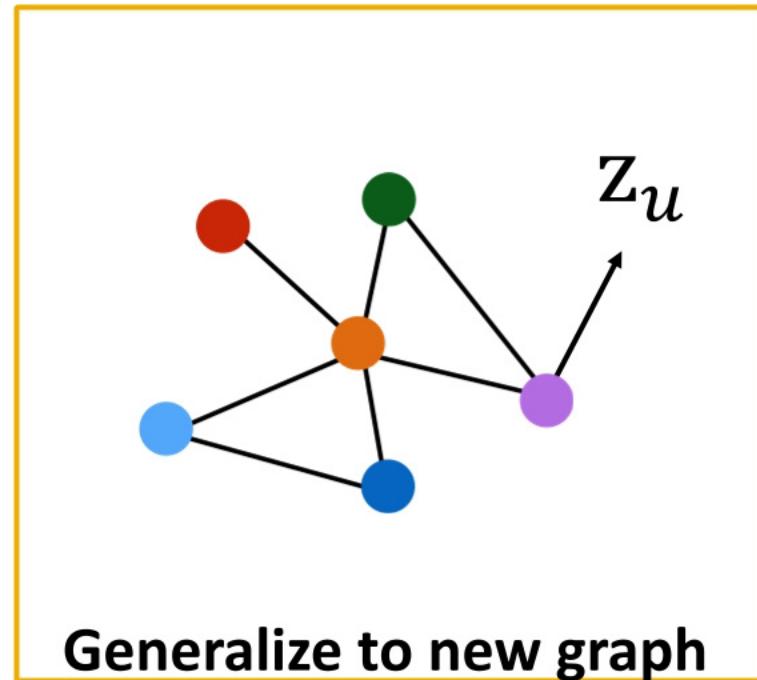
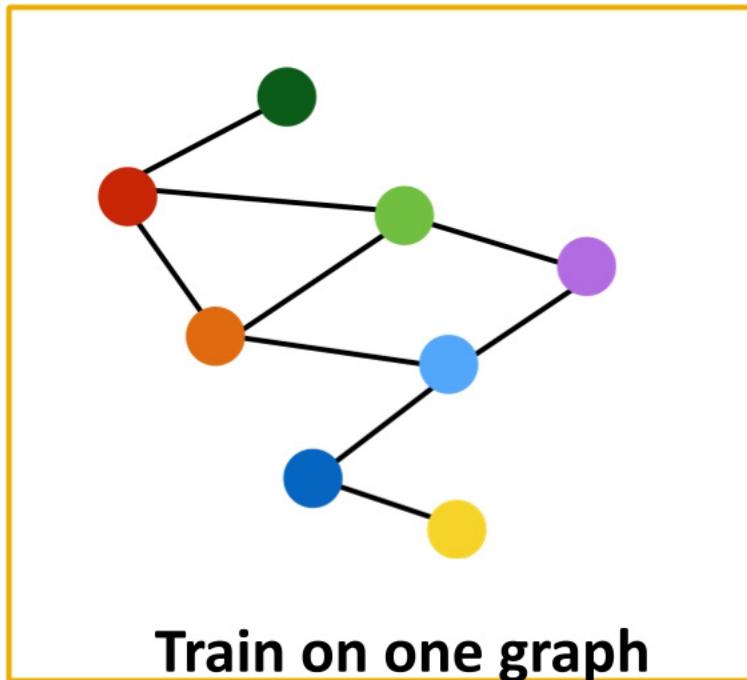


# Inductive Capability

- 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!**



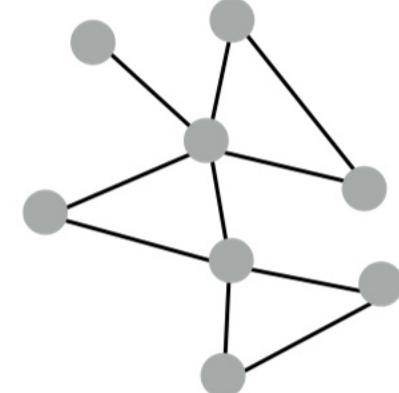
# Inductive Capability: New Graphs



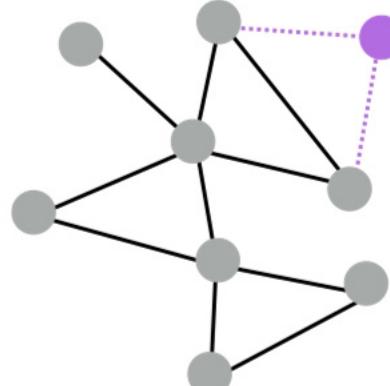
Inductive node embedding → Generalize to entirely unseen graphs

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

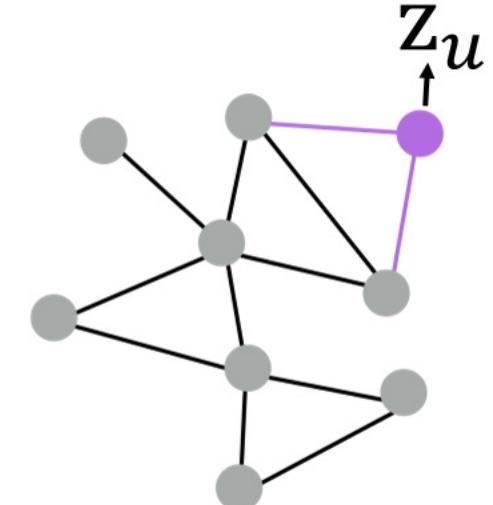
# Inductive Capability: New Nodes



Train with snapshot



New node arrives



Generate embedding  
for new node

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

# Outline of Today's Lecture

1. Basics of deep learning



2. Deep learning for graphs



3. Graph Convolutional Networks



4. GNNs subsume CNNs and  
Transformers

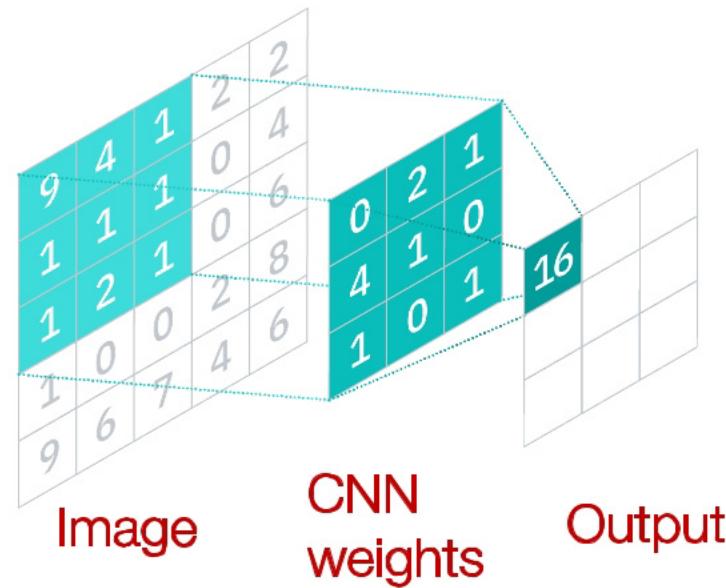
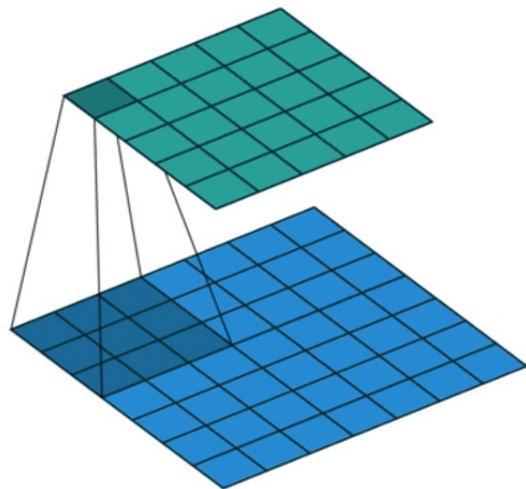


# Architecture Comparison

- How does GNNs compare to prominent architectures such as Convolutional Neural Nets, and Transformers?

# Convolutional Neural Network

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

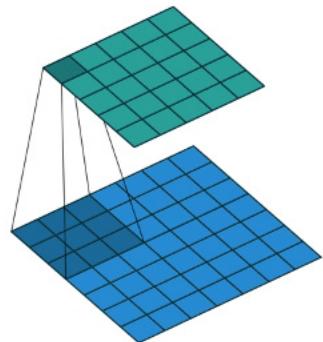


$$\text{CNN formulation: } h_v^{(l+1)} = \sigma(\sum_{u \in N(v) \cup \{v\}} W_l^u h_u^{(l)}), \quad \forall l \in \{0, \dots, L-1\}$$

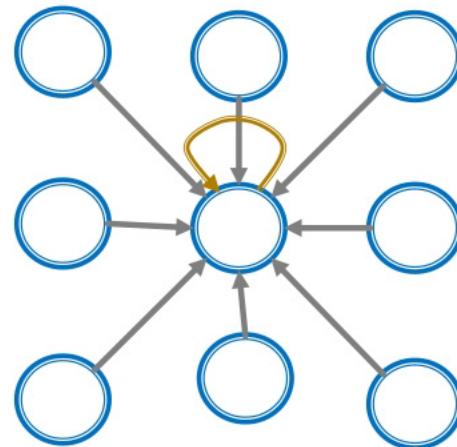
**$N(v)$  represents the 8 neighbor pixels of  $v$ .**

# GNN vs. CNN

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



Image

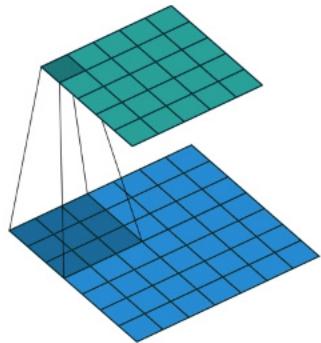


Graph

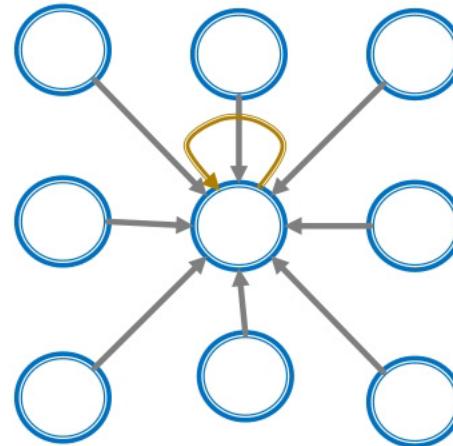
- GNN formulation (previous slide):  $h_v^{(l+1)} = \sigma(\mathbf{W}_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)})$ ,  $\forall l \in \{0, \dots, L-1\}$
- CNN formulation:
  - if we rewrite:
$$h_v^{(l+1)} = \sigma(\sum_{u \in N(v) \cup \{v\}} W_l^u h_u^{(l)})$$
$$h_v^{(l+1)} = \sigma(\sum_{u \in N(v)} \mathbf{W}_l^u h_u^{(l)} + B_l h_v^{(l)})$$
,  $\forall l \in \{0, \dots, L-1\}$

# GNN vs. CNN

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



Image



Graph

$$\text{GNN formulation: } h_v^{(l+1)} = \sigma(\mathbf{W}_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)}), \forall l \in \{0, \dots, L-1\}$$

$$\text{CNN formulation: } h_v^{(l+1)} = \sigma(\sum_{u \in N(v)} \mathbf{W}_l^u h_u^{(l)} + B_l h_v^{(l)}), \forall l \in \{0, \dots, L-1\}$$



**Key difference:** We can learn different  $W_l^u$  for different “neighbor”  $u$  for pixel  $v$  on the image. The reason is we can pick an order for the 9 neighbors using **relative position** to the center pixel:  $\{(-1, -1), (-1, 0), (-1, 1), \dots, (1, 1)\}$

# GNN vs. CNN

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



- CNN can be seen as a special GNN with fixed neighbor size and ordering:
  - The size of the filter is pre-defined for a CNN.
  - The advantage of GNN is it processes arbitrary graphs with different degrees for each node.

**Key difference:** We can learn different  $W_l^u$  for different “neighbor”  $u$  for pixel  $v$  on the image. The reason is we can pick an order for the 9 neighbors using **relative position** to the center pixel:  $\{(-1, -1), (-1, 0), (-1, 1), \dots, (1, 1)\}$

# GNN vs. CNN

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

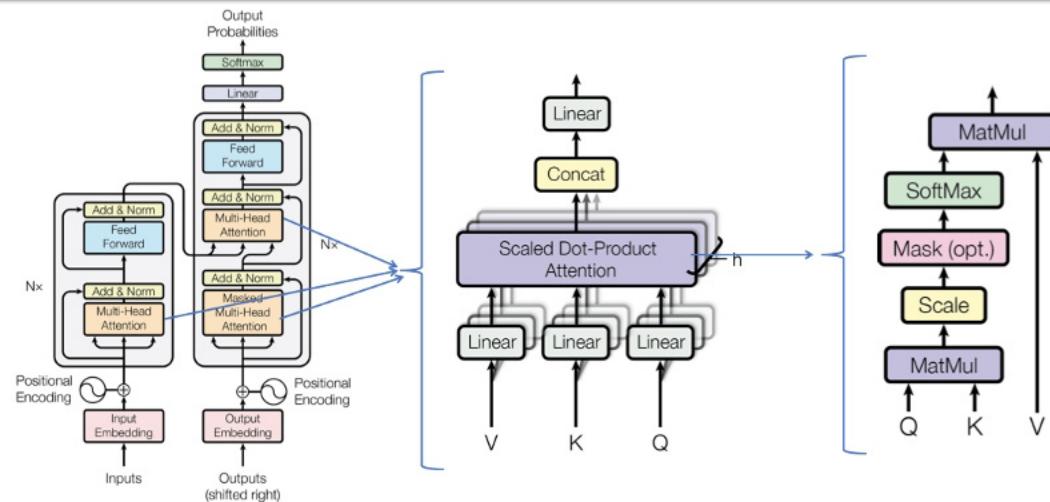


- CNN can be seen as a special GNN with fixed neighbor size and ordering.
- CNN is not permutation equivariant.
  - Switching the order of pixels will leads to different outputs.

**Key difference:** We can learn different  $W_l^u$  for different “neighbor”  $u$  for pixel  $v$  on the image. The reason is we can pick an order for the 9 neighbors using **relative position** to the center pixel:  $\{(-1, -1), (-1, 0), (-1, 1), \dots, (1, 1)\}$

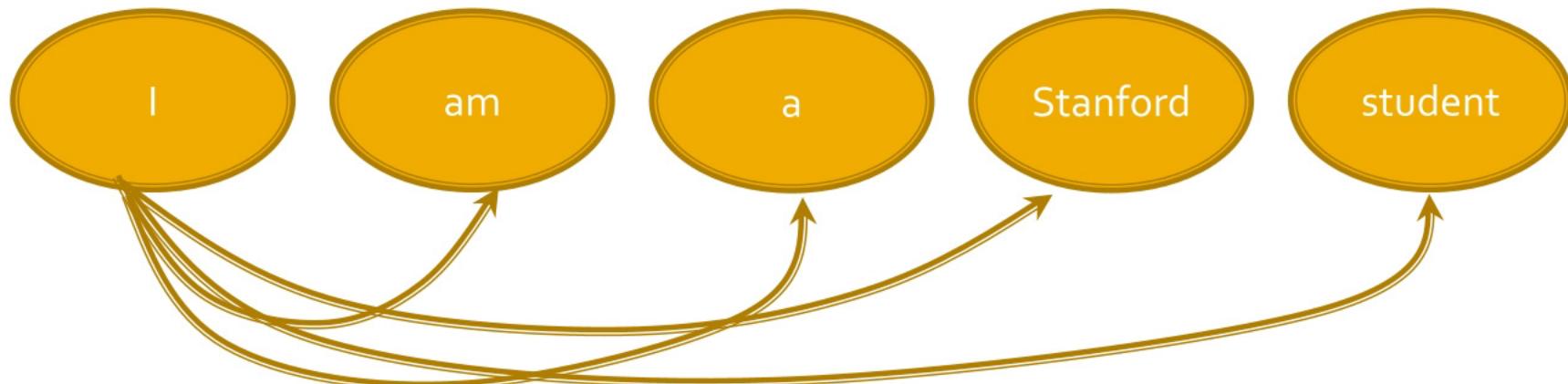
# Transformer

Transformer is one of the most popular architectures that achieves great performance in many sequence modeling tasks.



## Key component: self-attention

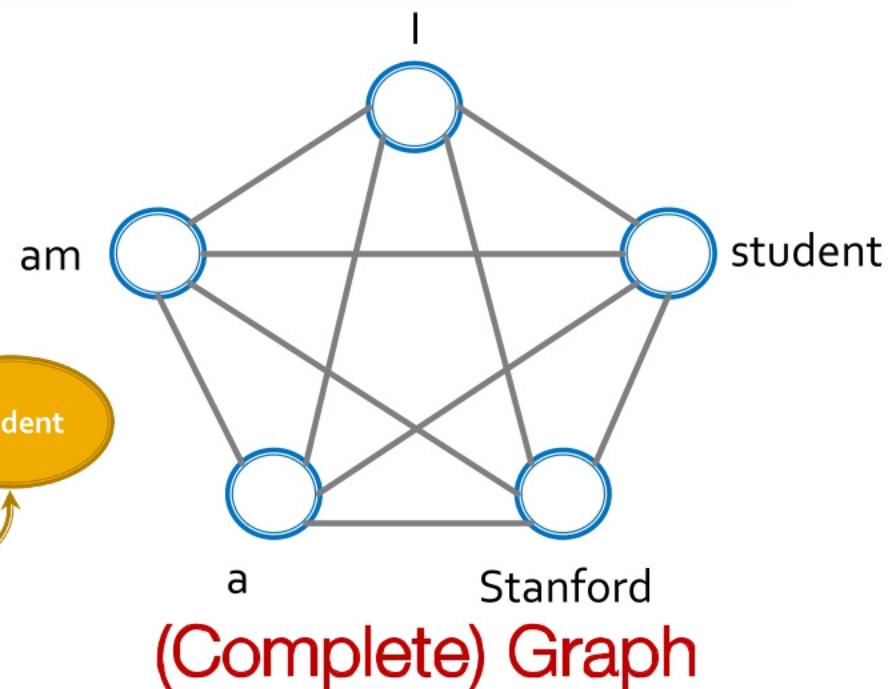
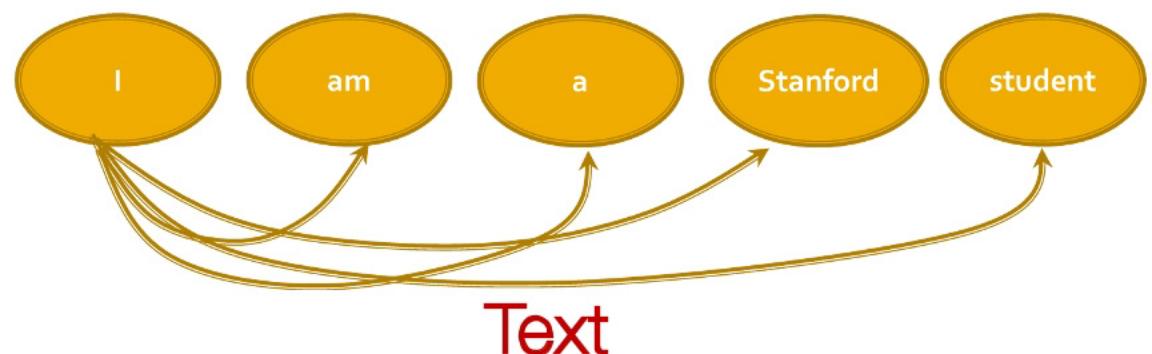
- Every token/word attends to all the other tokens/words via matrix calculation.



# GNN vs. Transformer

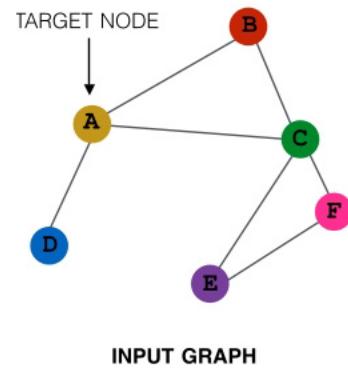
Transformer layer can be seen as a special GNN that runs on a fully-connected “word” graph!

Since each word attends to **all the other words**, **the computation graph** of a transformer layer is identical to that of a GNN on the **fully-connected “word” graph**.



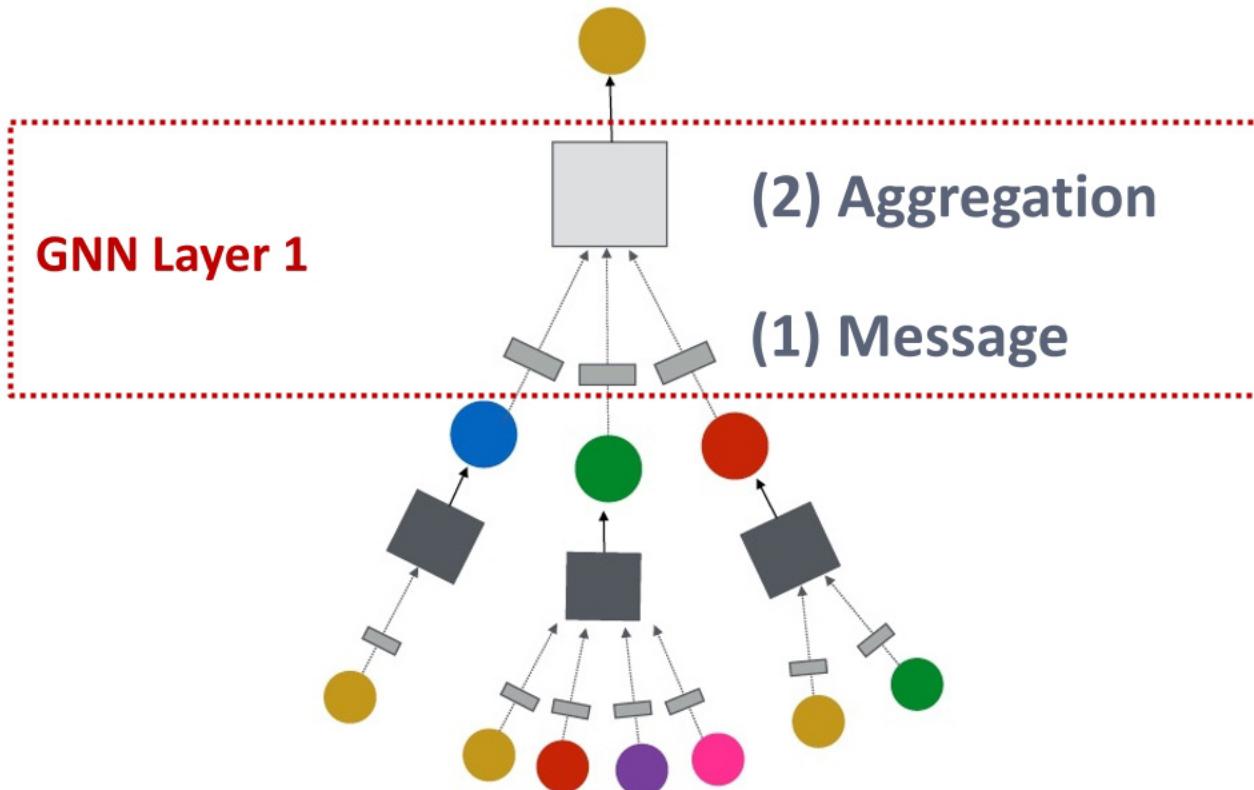
# A General Perspective on Graph Neural Networks

# A General GNN Framework (1)

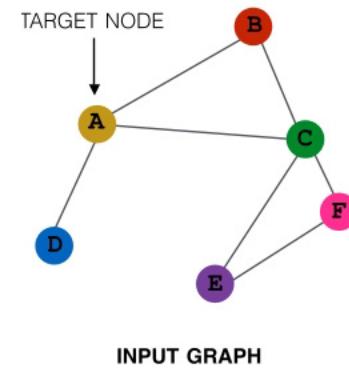


**GNN Layer = Message + Aggregation**

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



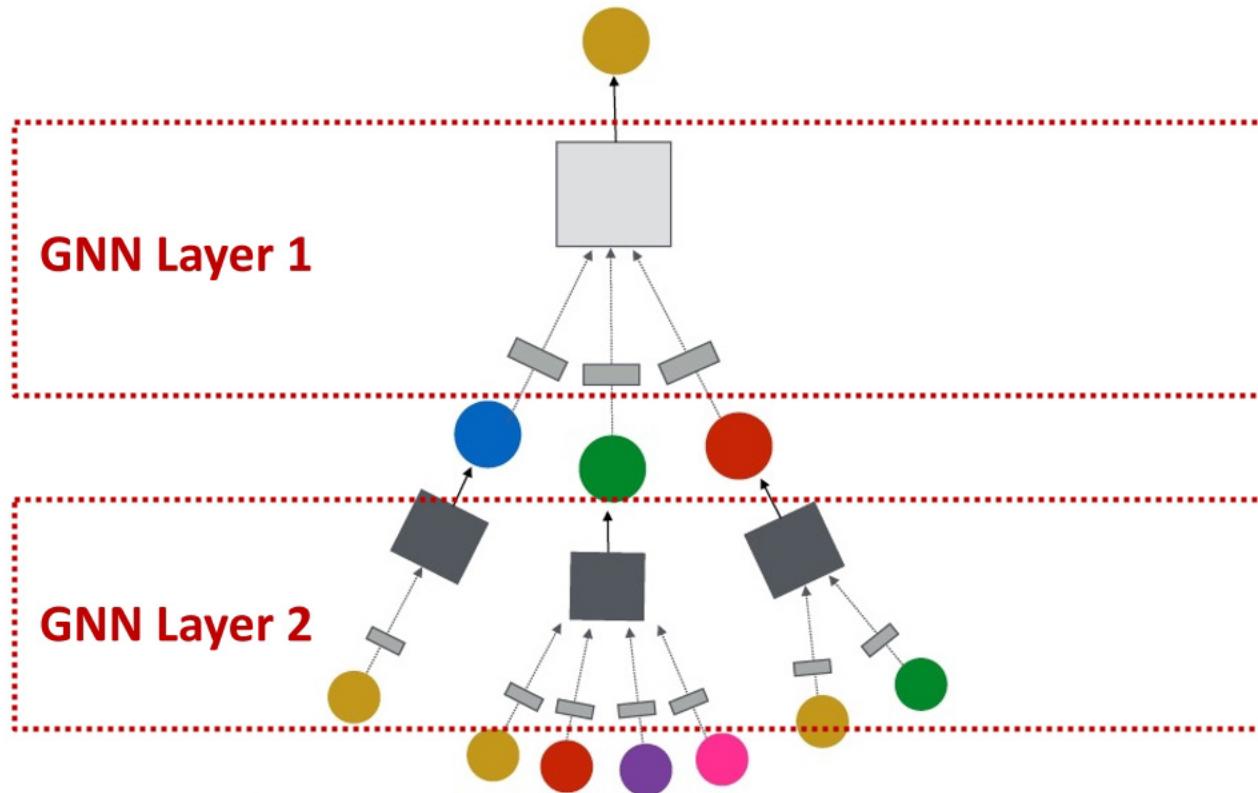
# A General GNN Framework (2)



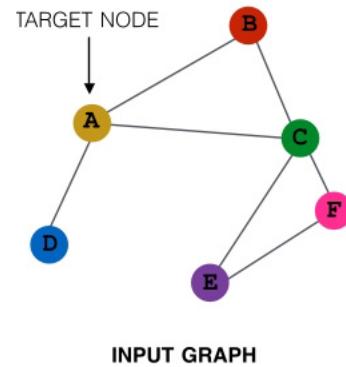
## Connect GNN layers into a GNN

- Stack layers sequentially
- Ways of adding skip connections

(3) Layer connectivity

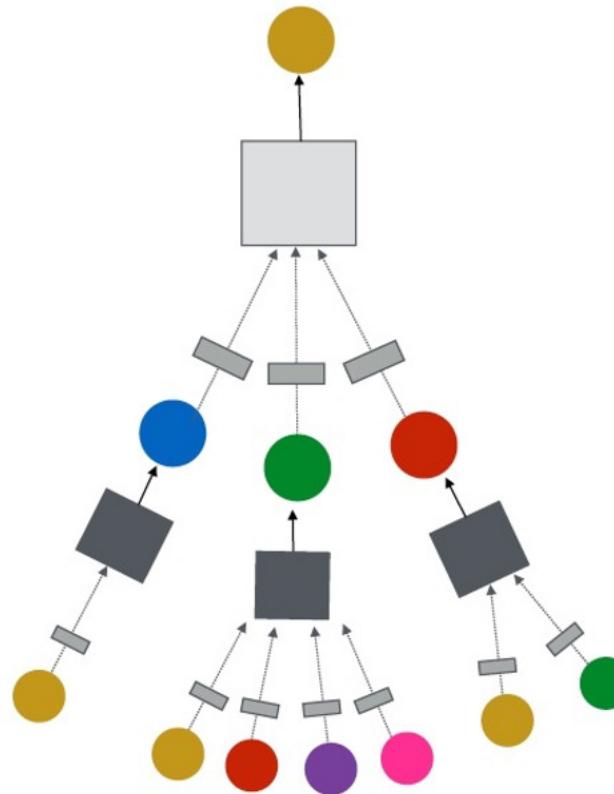


# A General GNN Framework (3)



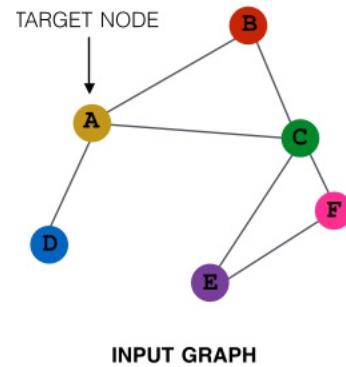
Idea: Raw input graph  $\neq$  computational graph

- Graph feature augmentation
- Graph structure augmentation



(4) Graph augmentation

# A General GNN Framework (4)

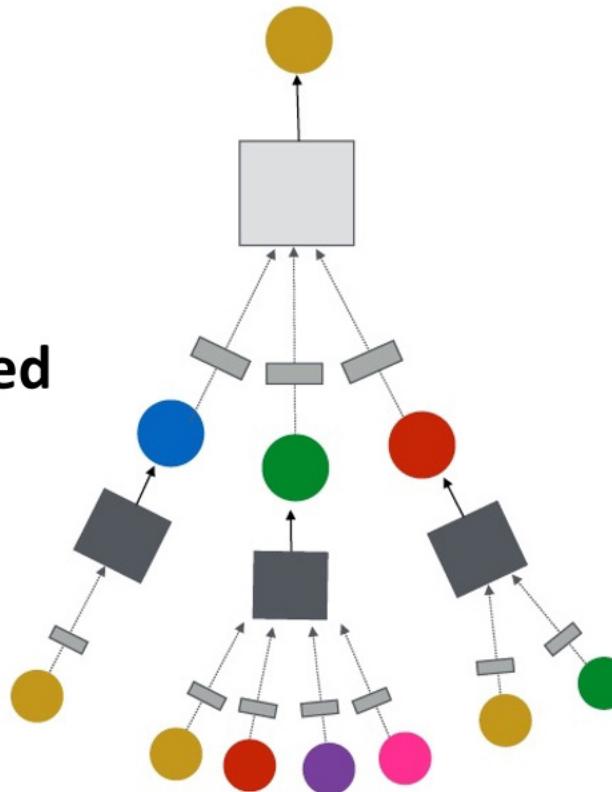


## (5) Learning objective

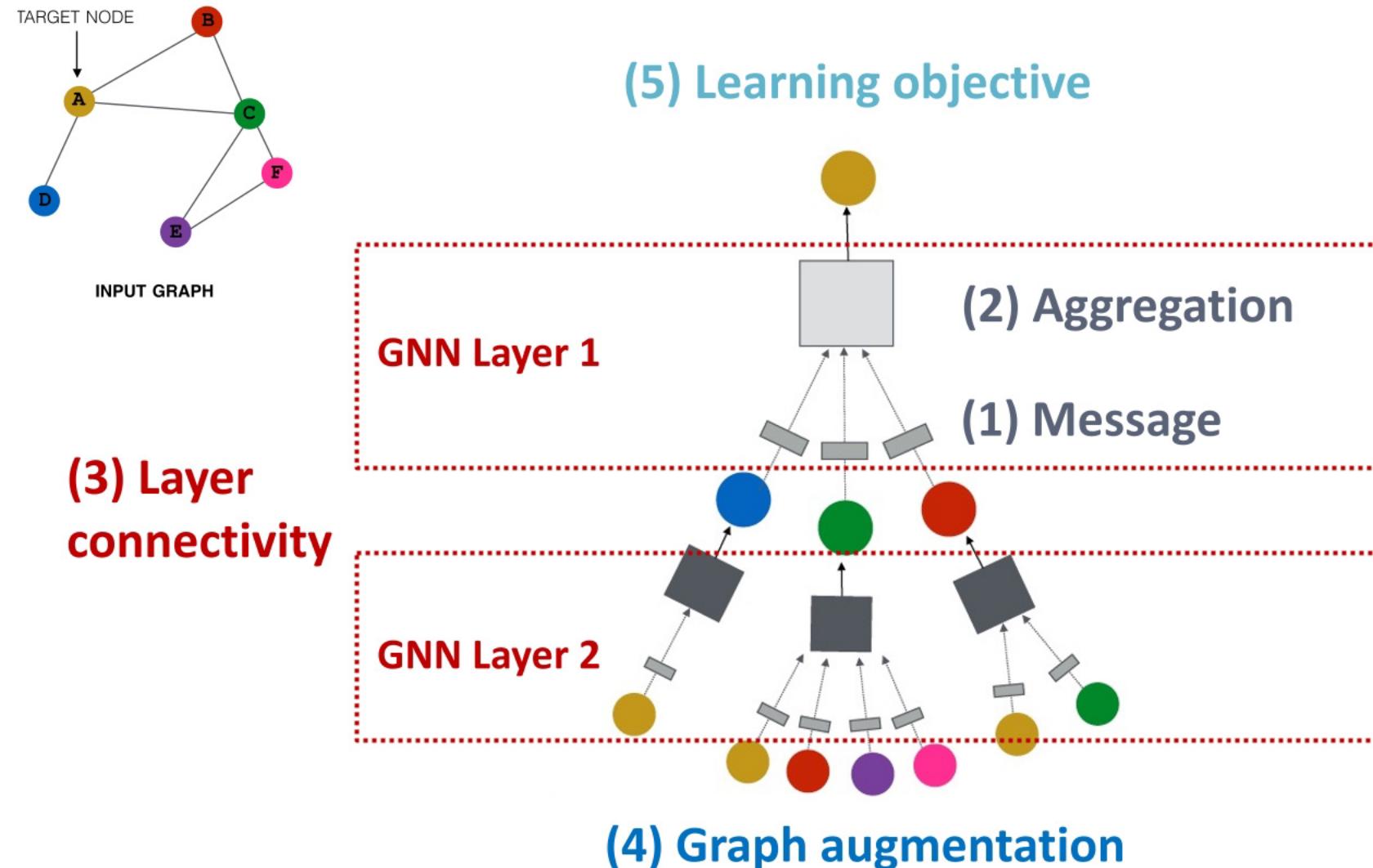
### How do we train a GNN

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

(We will discuss all of these later in class)

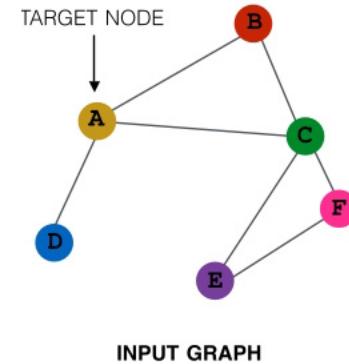


# GNN Framework: Summary



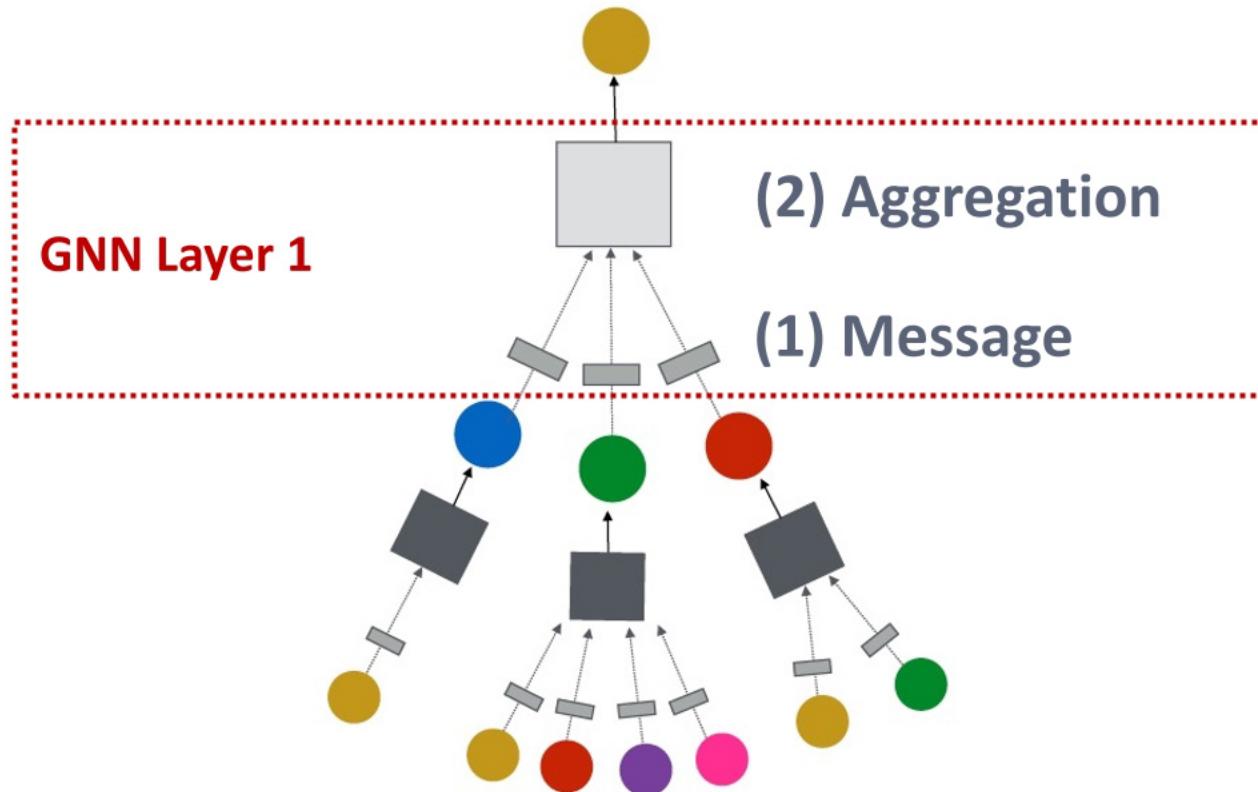
# A Single Layer of a GNN

# A GNN Layer



**GNN Layer = Message + Aggregation**

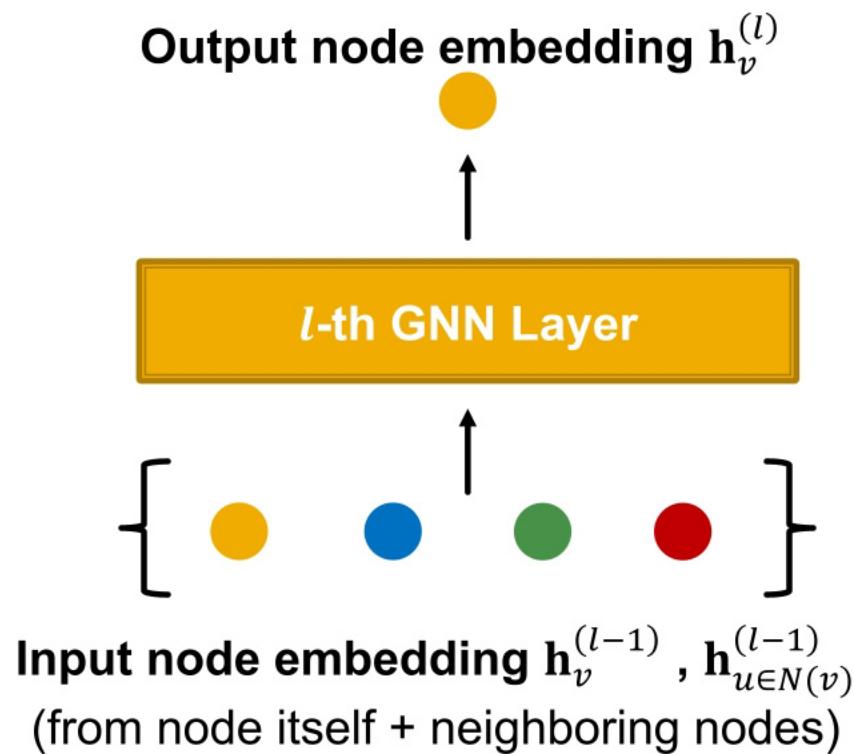
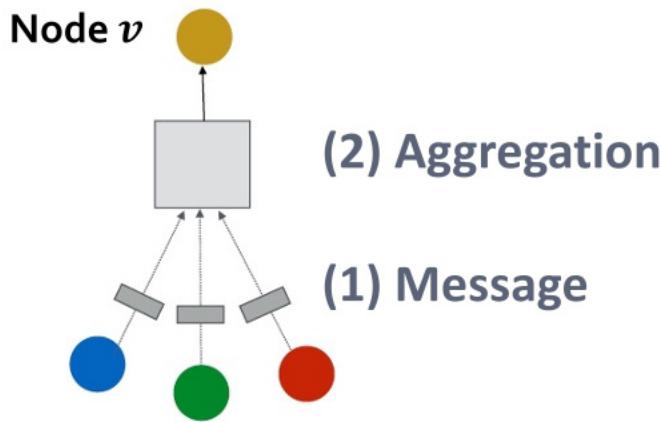
- Different instantiations under this perspective
- GCN, GraphSAGE, GAT, ...



# A Single GNN Layer

## ■ Idea of a GNN Layer:

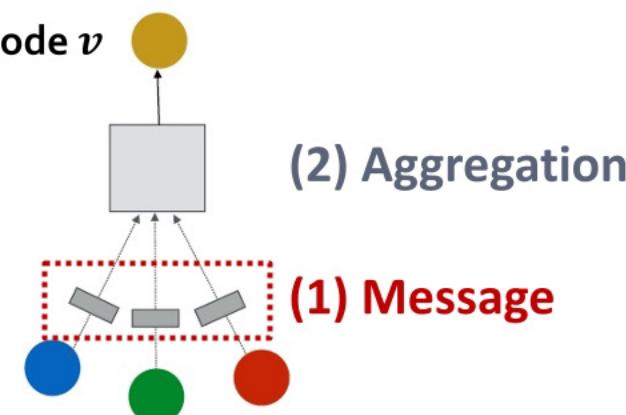
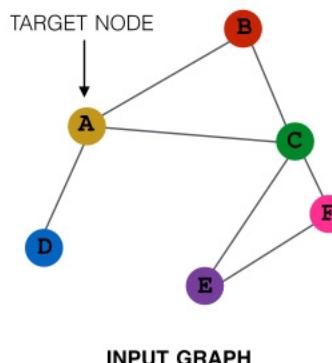
- Compress a set of vectors into a single vector
- Two-step process:
  - (1) Message
  - (2) Aggregation



# Message Computation

## ■ (1) Message computation

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



# Message Aggregation

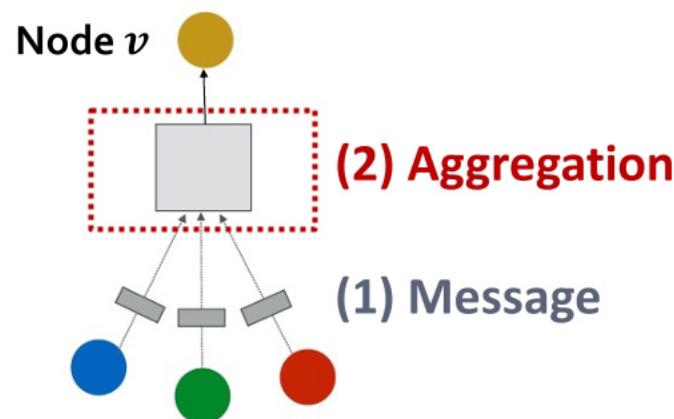
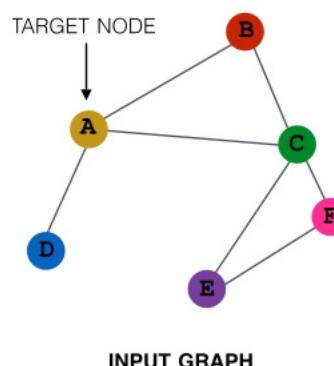
## ■ (2) Aggregation

- **Intuition:** Each node will aggregate the messages from node  $v$ 's neighbors

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

- **Example:** Sum( $\cdot$ ), Mean( $\cdot$ ) or Max( $\cdot$ ) aggregator

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



# Message Aggregation: Issue

- **Issue:** Information from node  $v$  itself **could get lost**
  - Computation of  $\mathbf{h}_v^{(l)}$  does not directly depend on  $\mathbf{h}_v^{(l-1)}$
- **Solution:** Include  $\mathbf{h}_v^{(l-1)}$  when computing  $\mathbf{h}_v^{(l)}$ 
  - **(1) Message:** compute message from node  $v$  itself
    - Usually, a **different message computation** will be performed



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



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

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

Then aggregate from node itself

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

First aggregate from neighbors

# A Single GNN Layer

## ■ Putting things together:

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

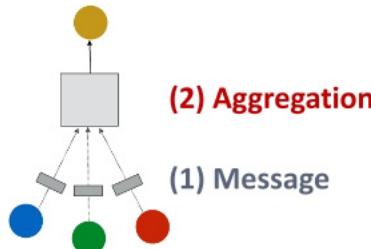
$$\mathbf{m}_u^{(l)} = \text{MSG}^{(l)}\left(\mathbf{h}_u^{(l-1)}\right), u \in \{N(v) \cup v\}$$

- (2) **Aggregation**: aggregate messages from neighbors

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

- **Nonlinearity (activation)**: Adds expressiveness

- Often written as  $\sigma(\cdot)$ : ReLU( $\cdot$ ), Sigmoid( $\cdot$ ) , ...
- Can be added to **message or aggregation**



# Classical GNN Layers: GCN (1)

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

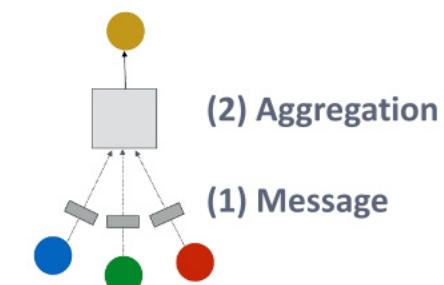
$$\mathbf{h}_v^{(l)} = \sigma \left( \mathbf{W}^{(l)} \sum_{u \in N(v)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|} \right)$$

- How to write this as Message + Aggregation?

**Message**

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

**Aggregation**

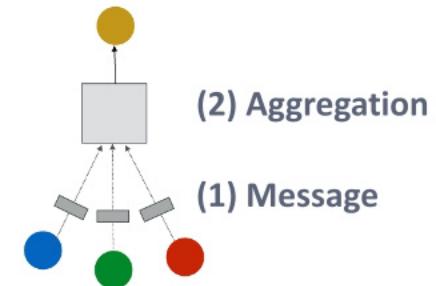


# Classical GNN Layers: GCN (2)



## ■ (1) Graph Convolutional Networks (GCN)

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



### ■ Message:

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

Normalized by node degree  
(In the GCN paper they use a slightly different normalization)

### ■ Aggregation:

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

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

# Classical GNN Layers: GraphSAGE

- (2) GraphSAGE

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

- How to write this as Message + Aggregation?

- Message is computed within the AGG( $\cdot$ )

- Two-stage aggregation

- Stage 1: Aggregate from node neighbors

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

- Stage 2: Further aggregate over the node itself

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

# GraphSAGE Neighbor Aggregation

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

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

AggregationMessage computation

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

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

AggregationMessage computation

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

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

Aggregation

# GraphSAGE: L<sub>2</sub> Normalization



## ■ $\ell_2$ Normalization:

- **Optional:** Apply  $\ell_2$  normalization to  $\mathbf{h}_v^{(l)}$  at every layer
- $\mathbf{h}_v^{(l)} \leftarrow \frac{\mathbf{h}_v^{(l)}}{\|\mathbf{h}_v^{(l)}\|_2} \quad \forall v \in V \text{ where } \|u\|_2 = \sqrt{\sum_i u_i^2} \text{ ( $\ell_2$ -norm)}$
- Without  $\ell_2$  normalization, the embedding vectors have different scales ( $\ell_2$ -norm) for vectors
- In some cases (not always), normalization of embedding results in performance improvement
- After  $\ell_2$  normalization, all vectors will have the same  $\ell_2$ -norm

# Classical GNN Layers: GAT (1)



## ■ (3) Graph Attention Networks

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

Attention weights

## ■ In GCN / GraphSAGE

- $\alpha_{vu} = \frac{1}{|N(v)|}$  is the **weighting factor (importance)** of node  $u$ 's message to node  $v$
- $\Rightarrow \alpha_{vu}$  is defined **explicitly** based on the **structural properties** of the graph (node degree)
- $\Rightarrow$  All neighbors  $u \in N(v)$  are **equally important** to node  $v$

# Classical GNN Layers: GAT (2)

## ■ (3) Graph Attention Networks

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

Attention weights

**Not all node's neighbors are equally important**

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

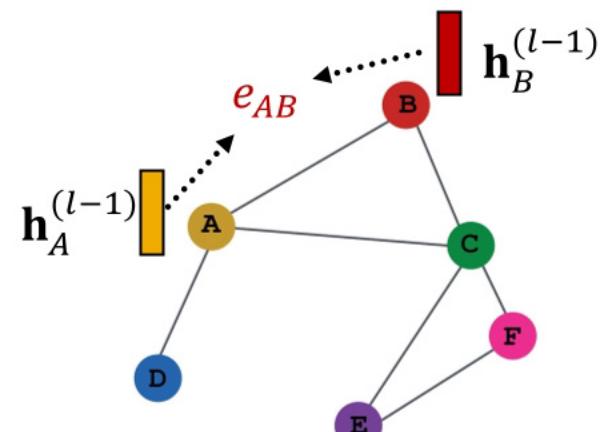
Can we do better than simple neighborhood aggregation?

Can we let weighting factors  $\alpha_{vu}$  to be learned?

- **Goal:** Specify **arbitrary importance** to different neighbors of each node in the graph
- **Idea:** Compute embedding  $h_v^{(l)}$  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  $\alpha_{vu}$  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:
$$e_{vu} = a(\mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}, \mathbf{W}^{(l)} \mathbf{h}_v^{(l-1)})$$
    - $e_{vu}$  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**  $\alpha_{vu}$

- Use the **softmax** function, so that  $\sum_{u \in N(v)} \alpha_{vu} = 1$ :

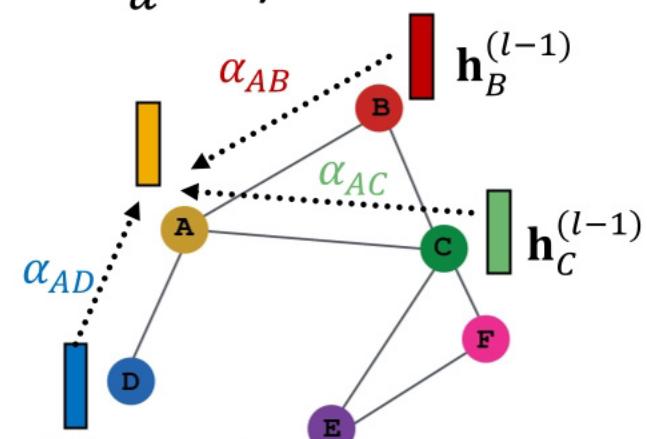
$$\alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})}$$

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

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

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

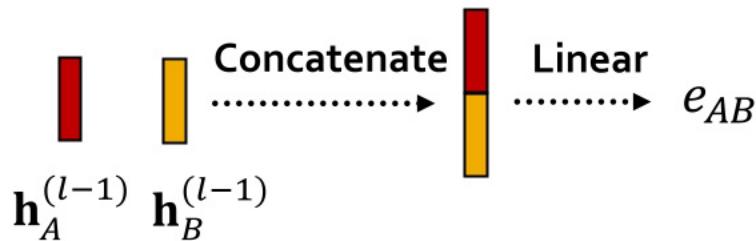
$$\begin{aligned} \mathbf{h}_A^{(l)} = \sigma(&\alpha_{AB} \mathbf{W}^{(l)} \mathbf{h}_B^{(l-1)} + \alpha_{AC} \mathbf{W}^{(l)} \mathbf{h}_C^{(l-1)} + \\ &\alpha_{AD} \mathbf{W}^{(l)} \mathbf{h}_D^{(l-1)}) \end{aligned}$$



# 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)



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

- 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):

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

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

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

- **Outputs are aggregated:**

- By concatenation or summation

- $\mathbf{h}_v^{(l)} = \text{AGG}(\mathbf{h}_v^{(l)}[1], \mathbf{h}_v^{(l)}[2], \mathbf{h}_v^{(l)}[3])$

# Benefits of Attention Mechanism



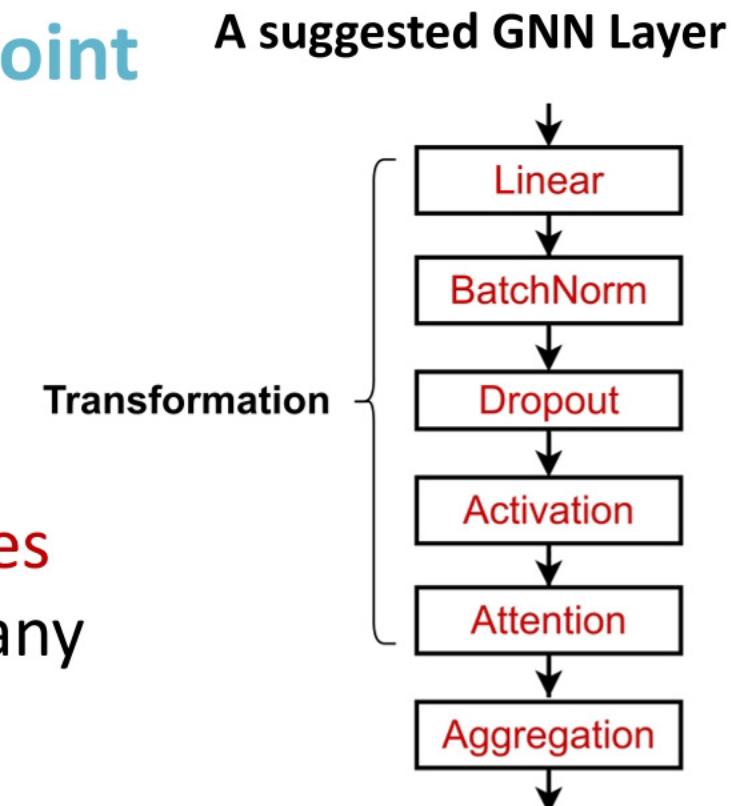
- **Key benefit:** Allows for (implicitly) specifying **different importance values** ( $\alpha_{vu}$ ) **to different neighbors**
- **Computationally efficient:**
  - Computation of attentional coefficients can be parallelized across all edges of the graph
  - Aggregation may be parallelized across 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 Layers in Practice

# GNN Layer in Practice

- In practice, these classic GNN layers are a great starting point

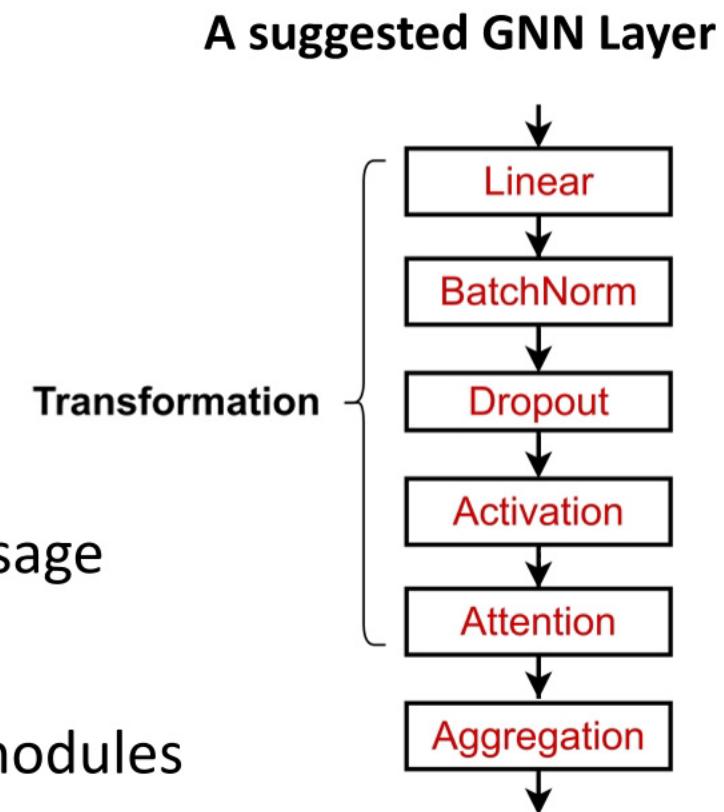
- We can often get better performance by considering a general GNN layer design
- Concretely, we can include modern deep learning modules that proved to be useful in many domains



# GNN Layer in Practice

- Many modern deep learning modules can be incorporated into a GNN layer

- Batch Normalization:
  - Stabilize neural network training
- Dropout:
  - Prevent overfitting
- Attention/Gating:
  - Control the importance of a message
- More:
  - Any other useful deep learning modules



# Batch Normalization



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

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

$N$  node embeddings

**Trainable Parameters:**

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

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

Normalized node embeddings

**Step 1:**  
**Compute the**  
**mean and variance**  
**over  $N$  embeddings**

$$\mu_j = \frac{1}{N} \sum_{i=1}^N \mathbf{x}_{i,j}$$

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (\mathbf{x}_{i,j} - \mu_j)^2$$

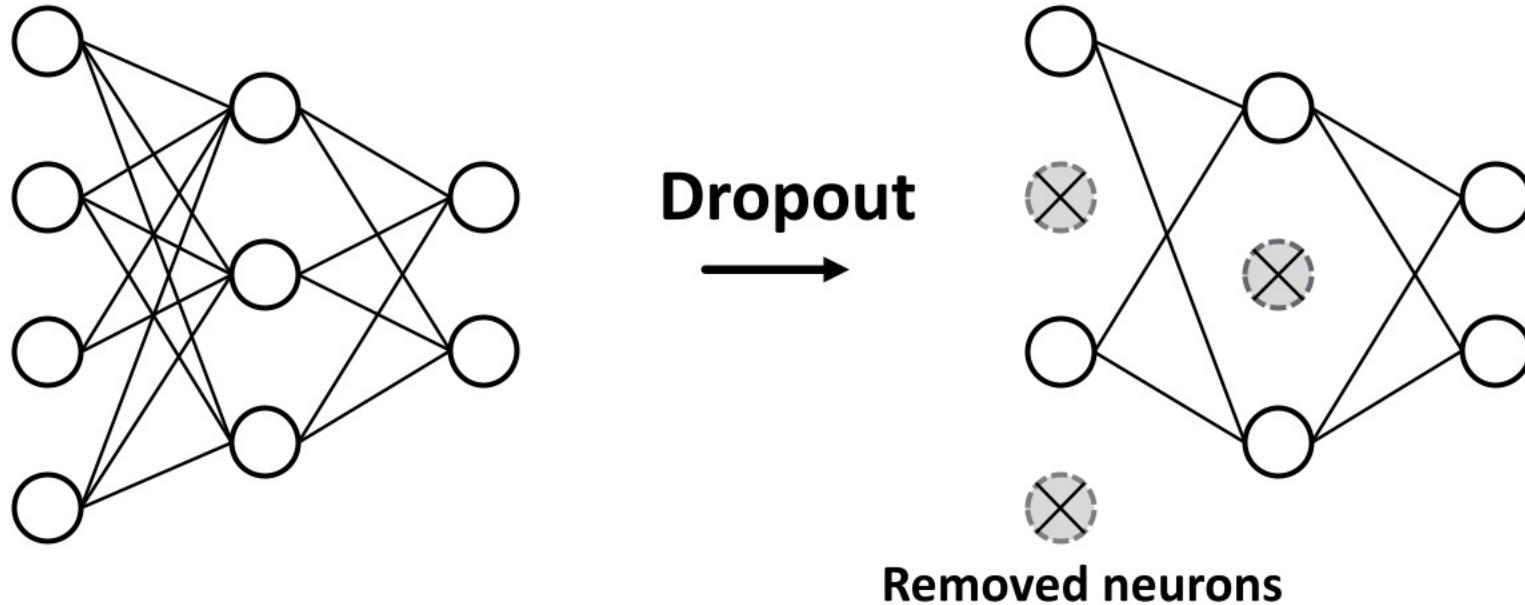
**Step 2:**  
**Normalize the feature**  
**using computed mean**  
**and variance**

$$\hat{\mathbf{x}}_{i,j} = \frac{\mathbf{x}_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

$$y_{i,j} = \gamma_j \hat{\mathbf{x}}_{i,j} + \beta_j$$

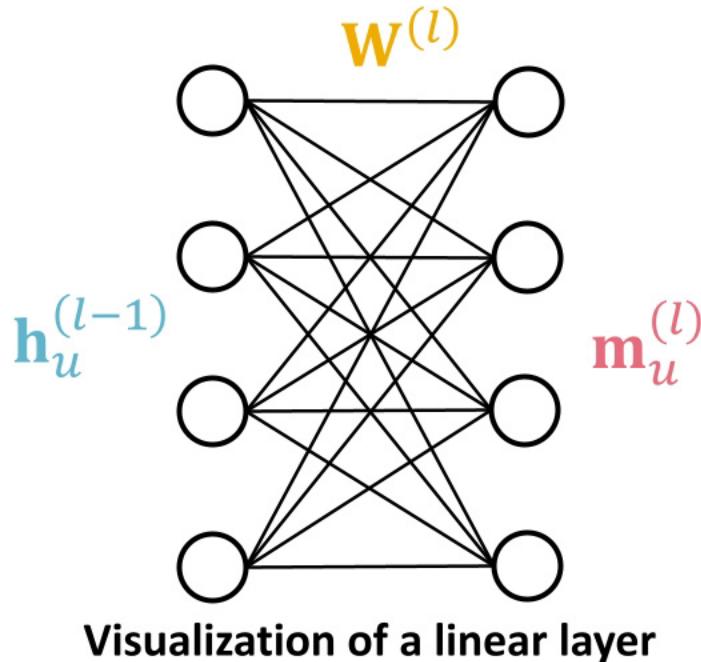
# Dropout

- **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

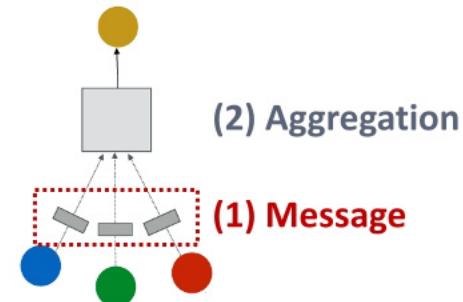
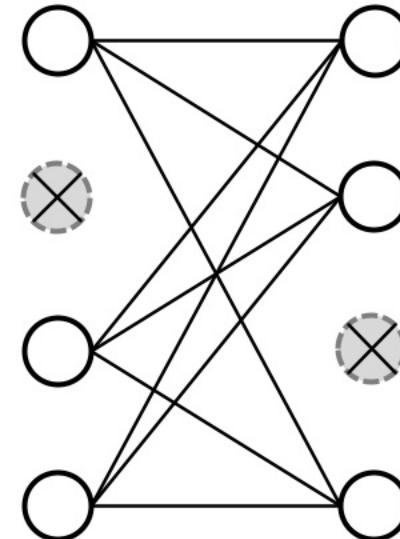


# Dropout for GNNs

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



Dropout  
→



# Activation (Non-linearity)

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

- Rectified linear unit (ReLU)

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

- Most commonly used

- Sigmoid

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

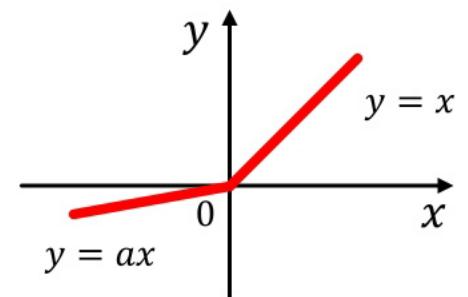
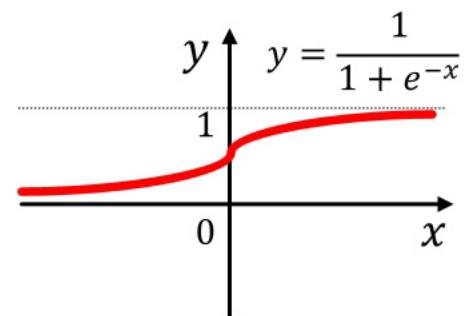
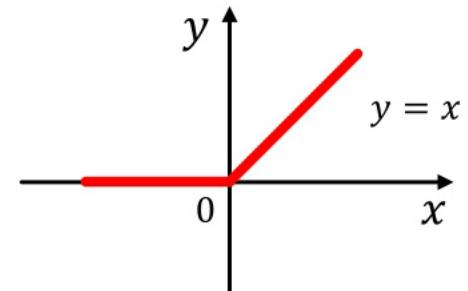
- Used only when you want to restrict the range of your embeddings

- Parametric ReLU

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

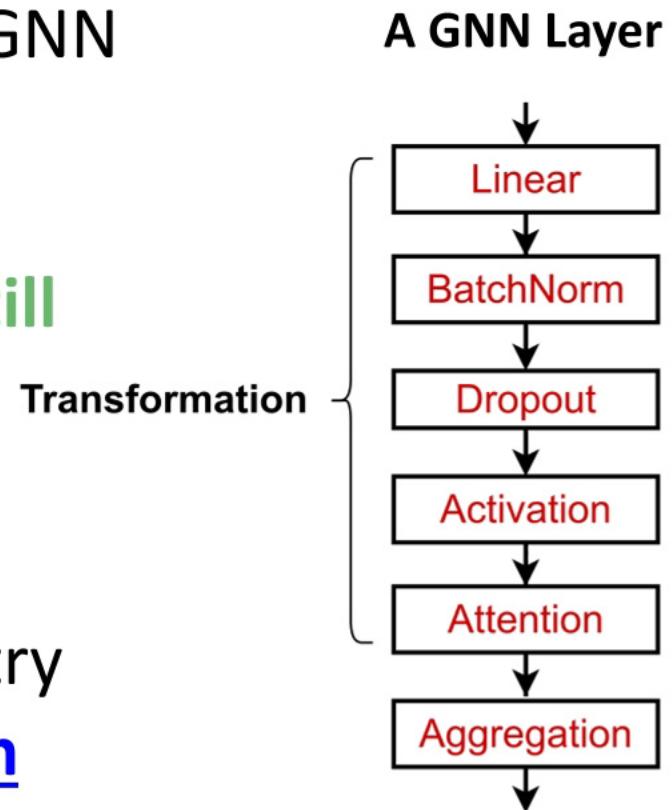
$a_i$  is a trainable parameter

- Empirically performs better than ReLU



# GNN Layer in Practice

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

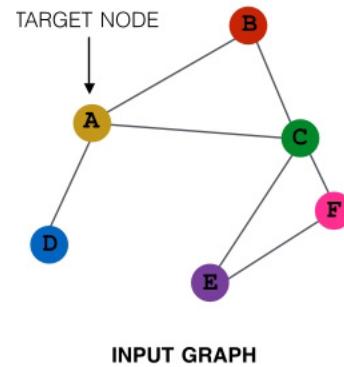


# Stacking Layers of a GNN

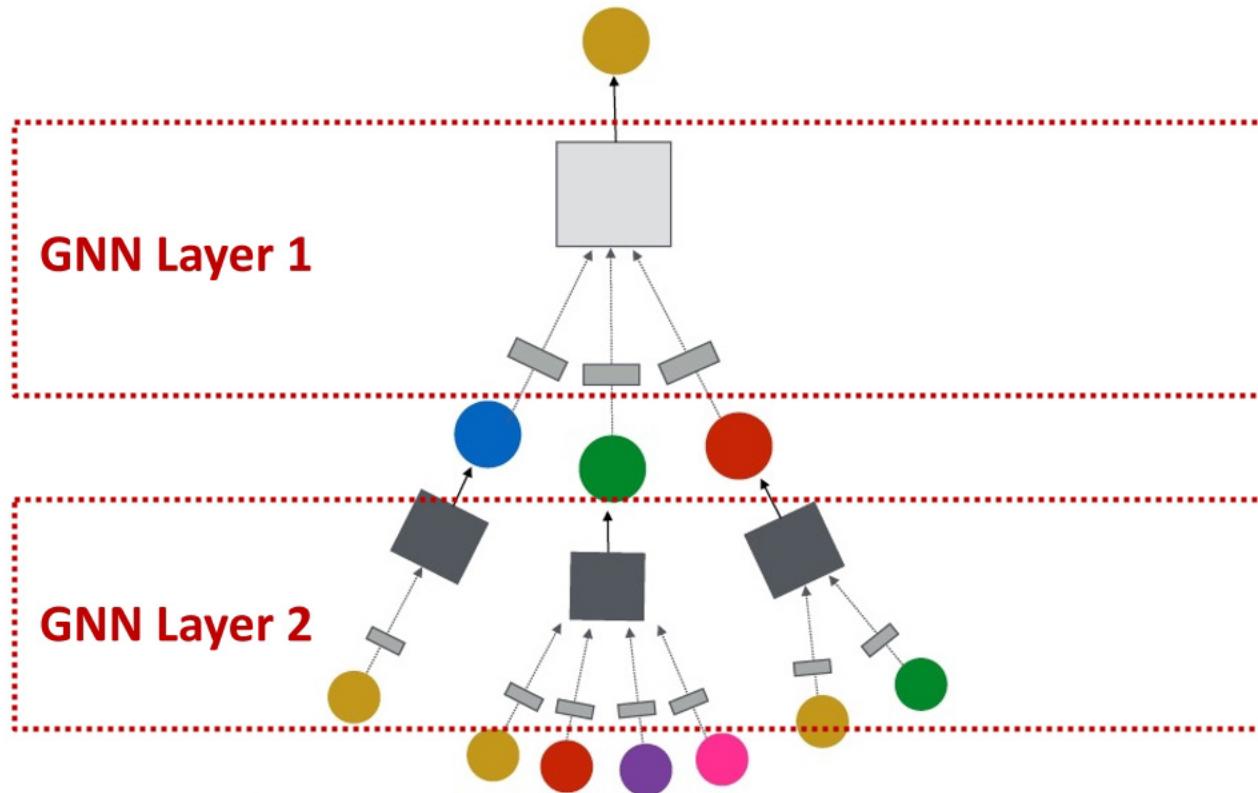
# Stacking GNN Layers

**How to connect GNN layers into a GNN?**

- Stack layers sequentially
- Ways of adding skip connections

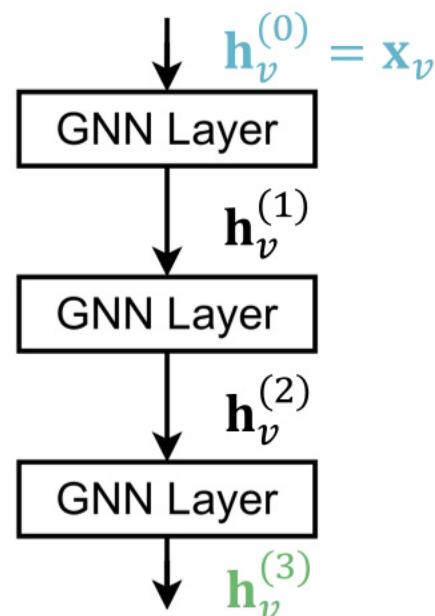


**(3) Layer connectivity**



# 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



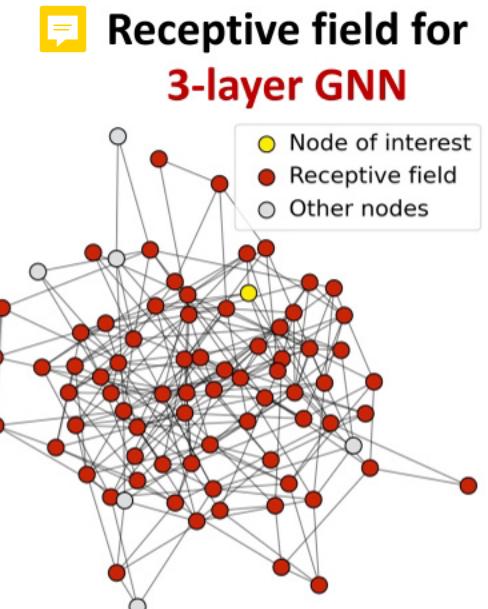
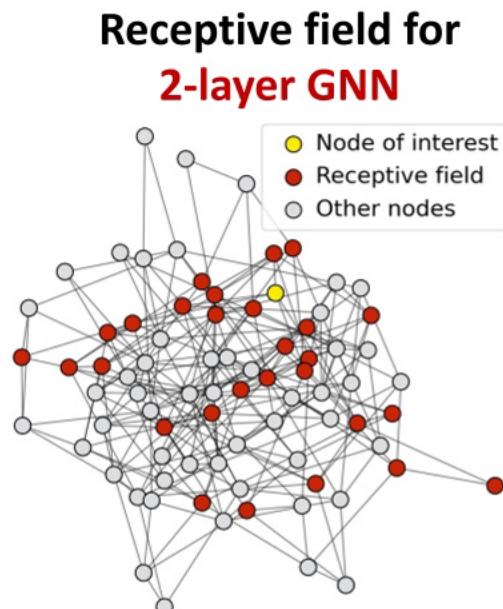
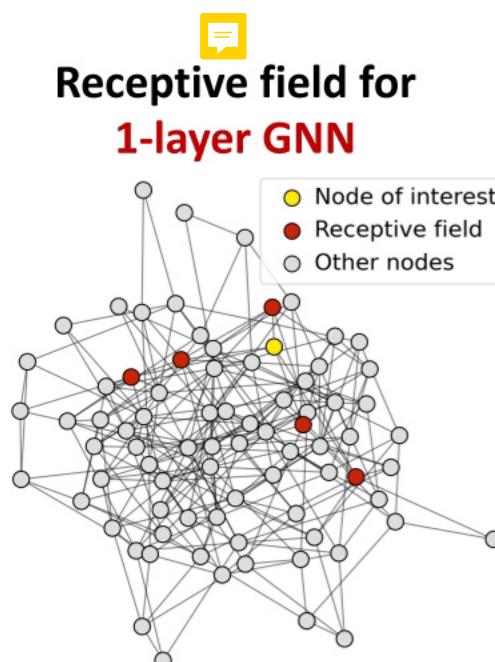
# The Over-smoothing Problem



- **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



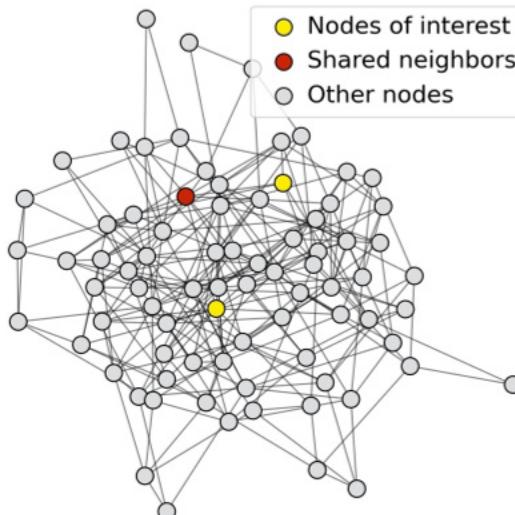
# Receptive Field of a GNN



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

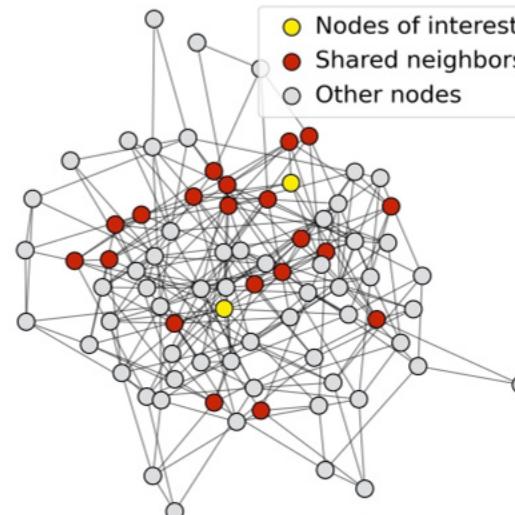
## 1-hop neighbor overlap

Only 1 node



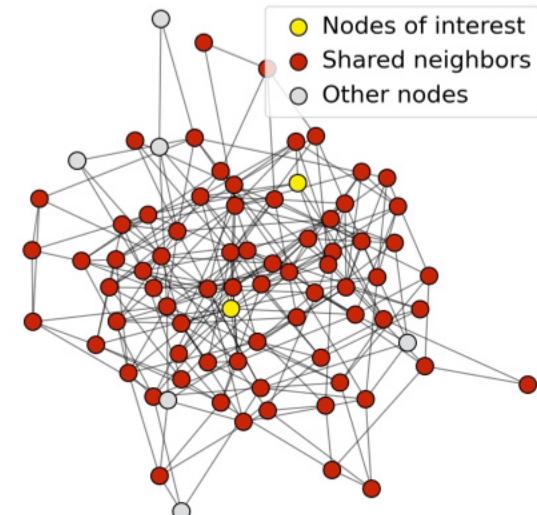
## 2-hop neighbor overlap

About 20 nodes



## 3-hop neighbor overlap

Almost all the nodes!



# Receptive Field & Over-smoothing



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

# Design GNN Layer Connectivity

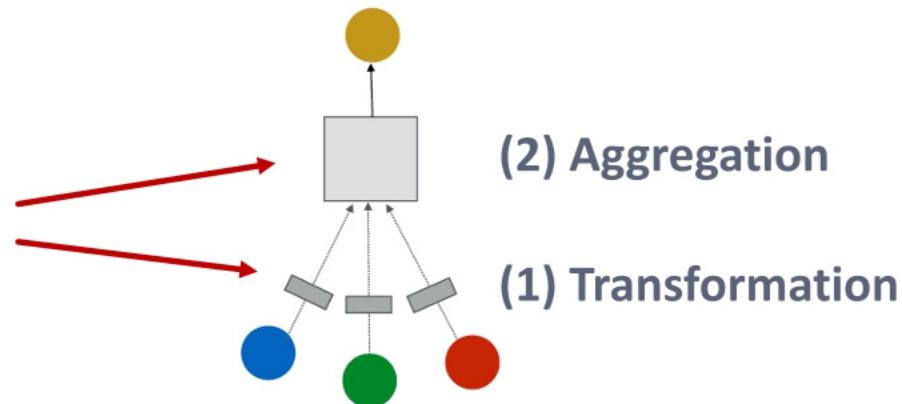
- **What do we learn from the over-smoothing problem?**
- **Lesson 1: Be cautious when adding GNN layers**
  - Unlike neural networks in other domains (CNN for image classification), **adding more GNN layers do not always help**
  - **Step 1:** Analyze the necessary receptive field to solve your problem.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

# Expressive Power for Shallow GNNs



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

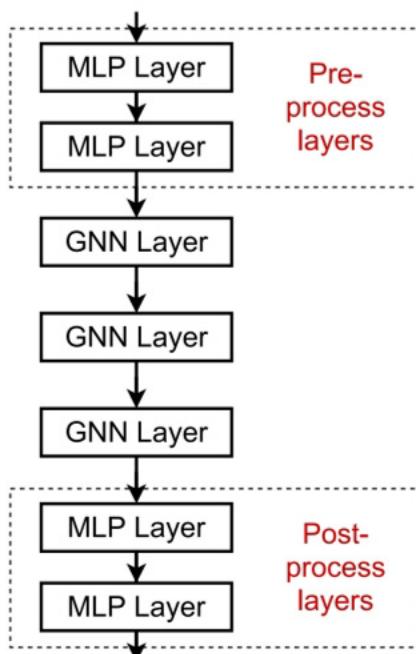
If needed, each box could include a 3-layer MLP



# Expressive Power for Shallow GNNs



- How to make a shallow GNN more expressive?
- Solution 2: Add layers that do not pass messages
  - A GNN does not necessarily only contain GNN layers
    - E.g., we can add **MLP layers** (applied to each node) before and after GNN layers, as **pre-process layers** and **post-process layers**



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

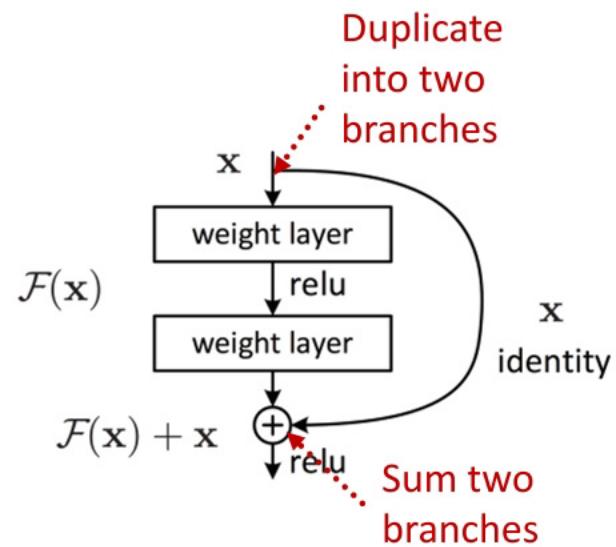
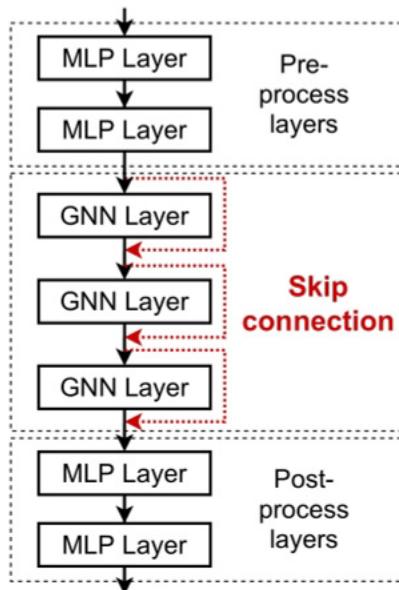
**Post-processing layers:** Important when reasoning / transformation over node embeddings are needed  
E.g., graph classification, knowledge graphs

In practice, adding these layers works great!

# Design GNN Layer Connectivity



- What if my problem still requires many GNN layers?
- Lesson 2: Add skip connections in GNNs
  - Observation from over-smoothing: Node embeddings in earlier GNN layers can sometimes better differentiate nodes
  - Solution: We can increase the impact of earlier layers on the final node embeddings, by adding shortcuts in GNN



**Idea of skip connections:**

Before adding shortcuts:

$$\mathcal{F}(x)$$

After adding shortcuts:

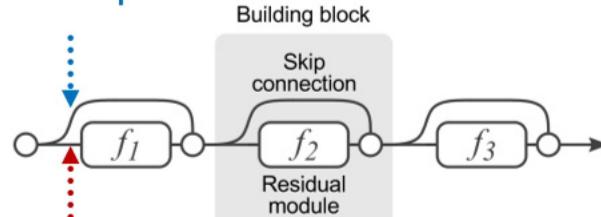
$$\mathcal{F}(x) + x$$

# Idea of Skip Connections



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

Path 2: skip this module

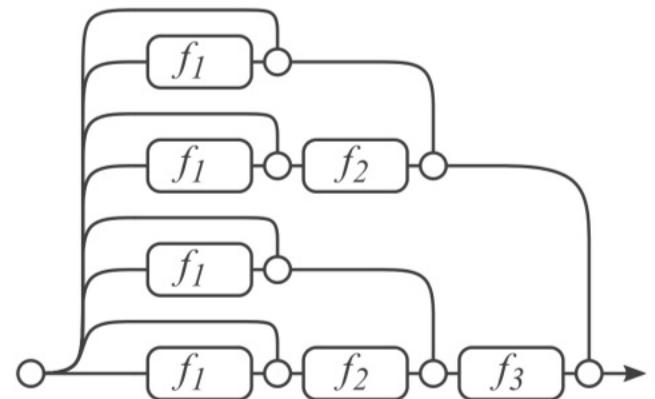


Path 1: include this module

(a) Conventional 3-block residual network

All the possible paths:

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



(b) Unraveled view of (a)

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

# Example: GCN with Skip Connections

- A standard GCN layer

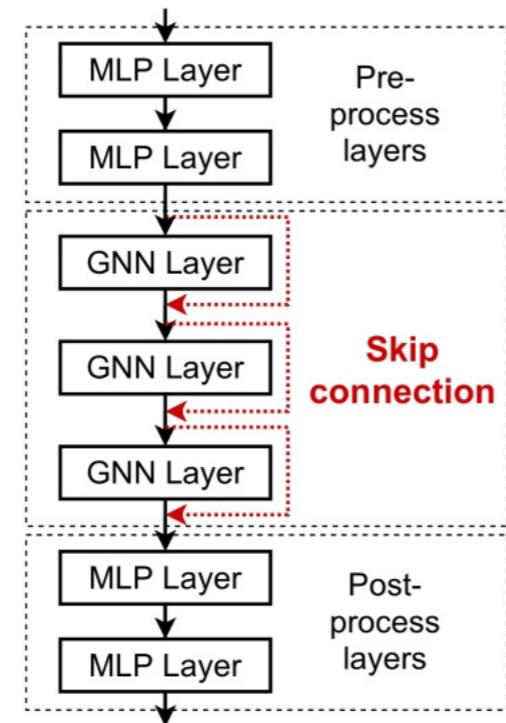
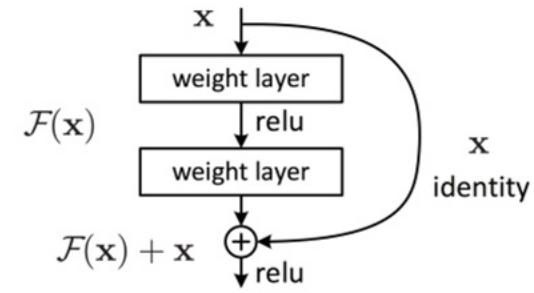
$$\mathbf{h}_v^{(l)} = \sigma \left( \sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|} \right)$$

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

- A GCN layer with skip connection

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

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



# Other Options of Skip Connections

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

