

# Graph Neural Networks

---

Lecturer: Dr. Zhang Handuo

<https://handuo.top/>

[handuo.zhang@shanda.com](mailto:handuo.zhang@shanda.com)

# About me

---

## Zhang Handuo

AI Scientist

Shanda AI Research Institute

<https://handuo.top>

[handuo.zhang@shanda.com](mailto:handuo.zhang@shanda.com)

[zhanghanduo@gmail.com](mailto:zhanghanduo@gmail.com)



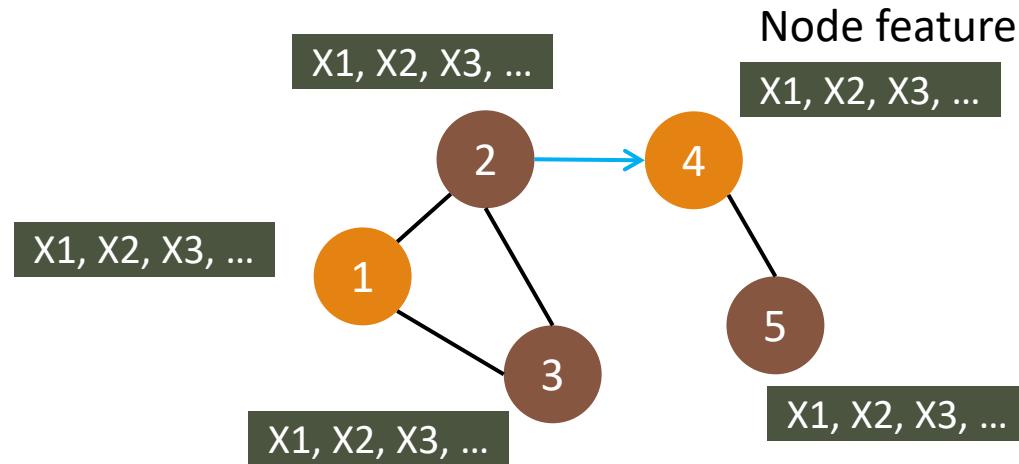
# Outline

---

- 1 Graph Representation
- 2 Machine Learning Types
- 3 Graph Neural Networks: Basics
- 4 GNN variants: GCN, GAT, GraphSage
- 5 GNN Layers in Practice
- 6 Knowledge Graph

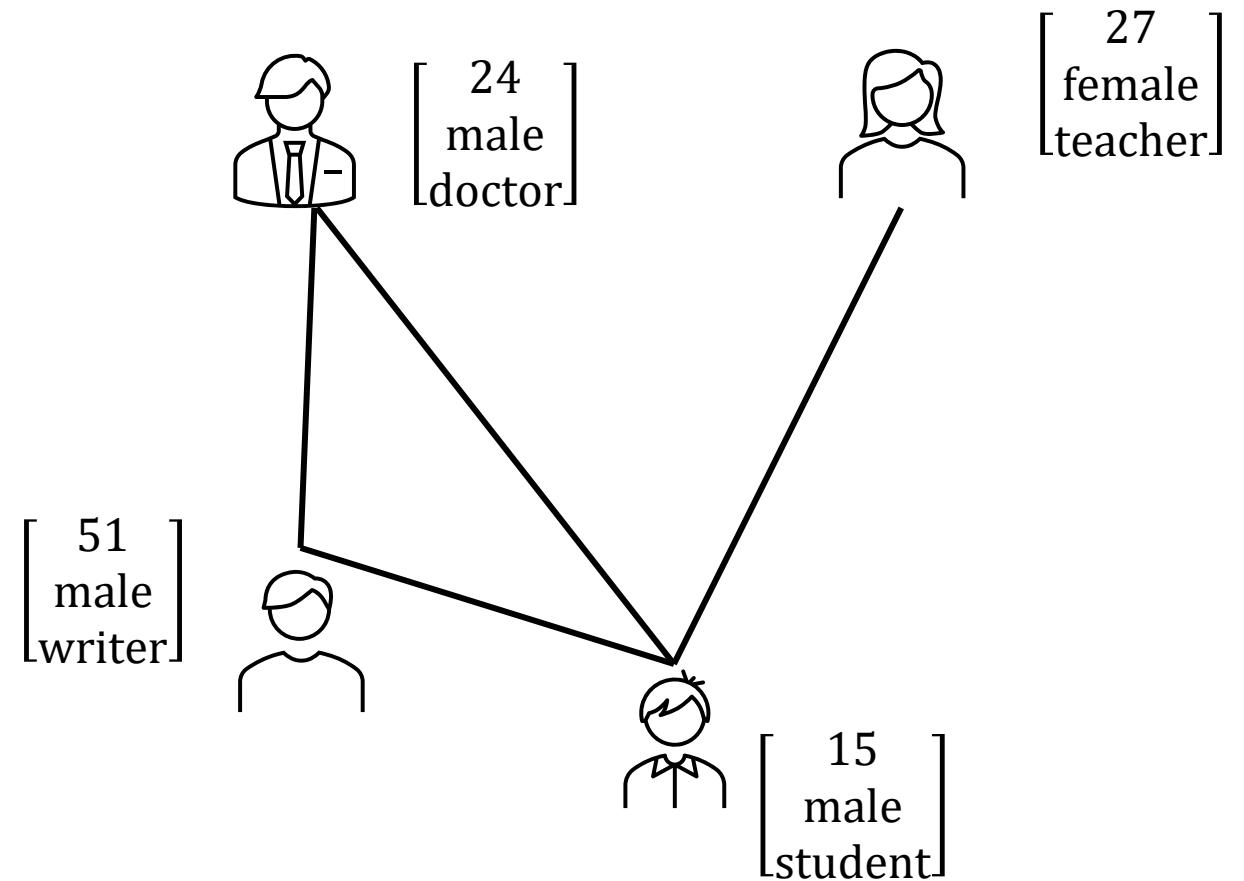
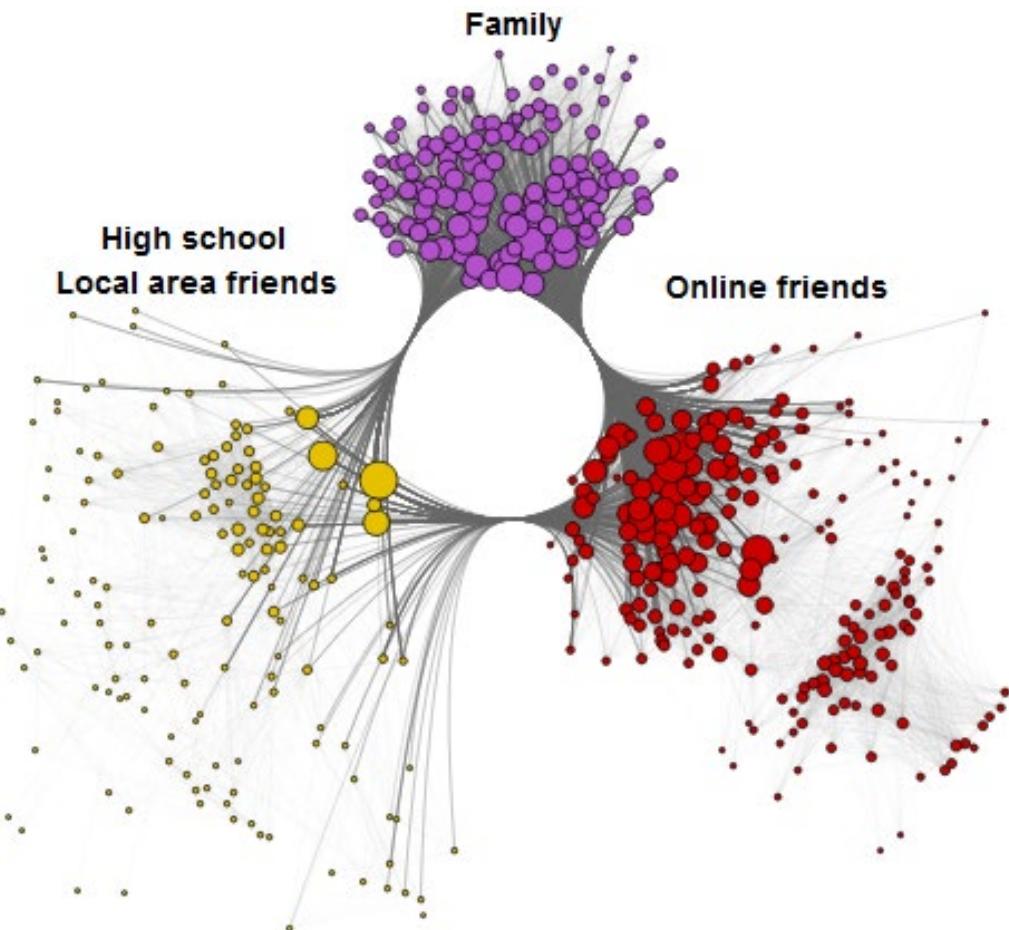
# What is a Graph

## 1 Graph Representation



- **Nodes:** entities, vertices
  - **Interactions:** links, edges
  - **Graph:** network
- $$\mathbf{V} = \{v_1, \dots, v_N\}$$
- $$\mathbf{E} = \{e_1, \dots, e_M\}$$
- $$G = (\mathbf{V}, \mathbf{E})$$
- **Node Features:**  $\mathbf{X} = \{x_1, \dots, x_N\}$  associated with each node

# Examples



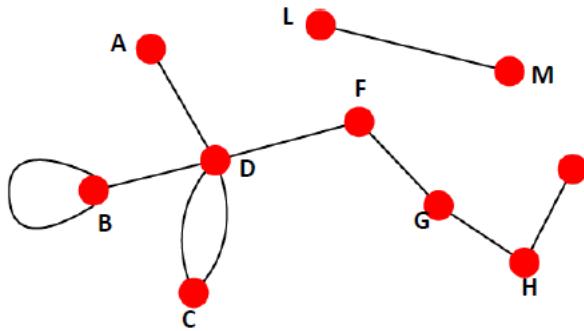
# Graph Edge Types

- **Directed vs. Undirected:** Edges may have a direction (ordered pair) or be bidirectional.
- **Weighted vs. Unweighted:** Edges might carry a weight or strength (e.g., distance, similarity) or just denote a connection.
- **Simple vs. Multi-graphs:** A simple graph has at most one edge between any two nodes; multi-graphs can have multiple parallel edges.
- **Homogeneous vs. Heterogeneous Graphs:** Nodes and edges in homogeneous graphs have same types, while nodes and edges have different types in heterogeneous graphs

# Directed vs. Undirected Graphs

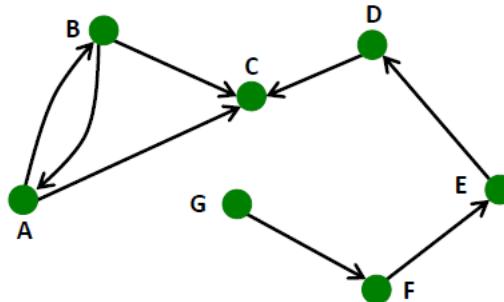
## Undirected

- **Links:** undirected  
(symmetrical, reciprocal)



## Directed

- **Links:** directed



- **Examples:**

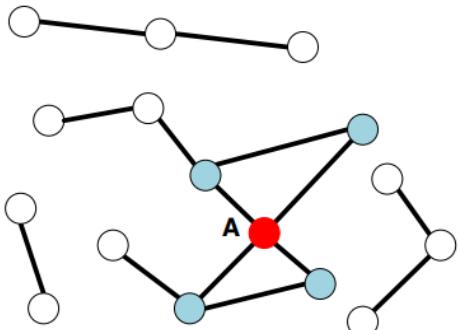
- Collaborations
- Friendship on Facebook

- **Examples:**

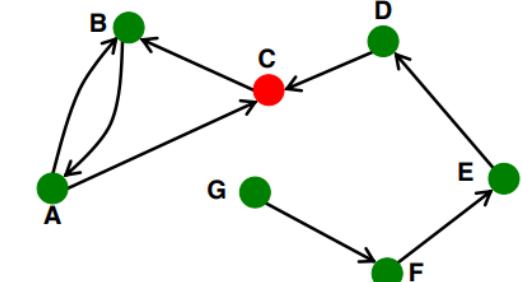
- Phone calls
- Following on Twitter

# Directed vs. Undirected Graphs

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.

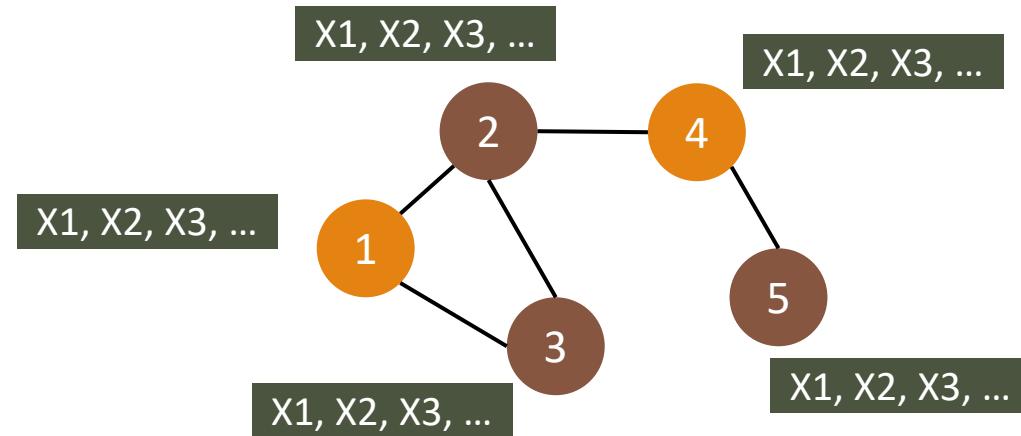
$$k_C^{in} = 2 \quad k_C^{out} = 1 \quad k_C = 3$$

$$\bar{k} = \frac{E}{N} \qquad \qquad \bar{k}^{in} = \bar{k}^{out}$$

# Adjacency Matrix

## 1 Graph Representation

Standard Network architectures don't work on such data



- Entities: nodes, vertices  $V$
- Interactions: links, edges  $E$
- System: network, graph  $G = (V, E)$

Representing Graphs

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

$A_{ij} = 0$  otherwise

Adjacency Matrix  $|V| \times |V|$

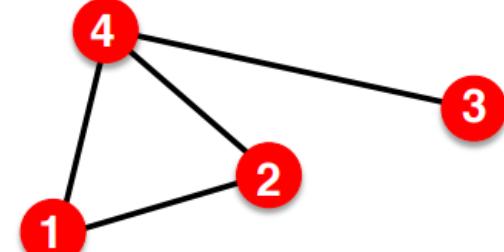
	v1	v2	v3	v4	v5	...
v1	0	1	1	0	0	...
v2	1	0	1	1	0	...
v3	1	1	0	0	0	...
v4	0	1	0	0	1	...
v5	0	0	0	1	0	...
...	...	...	...	...	...	...

Note that for an undirected graph the adjacent matrix is symmetric

# Adjacency Matrix

## 1 Graph Representation

Undirected



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

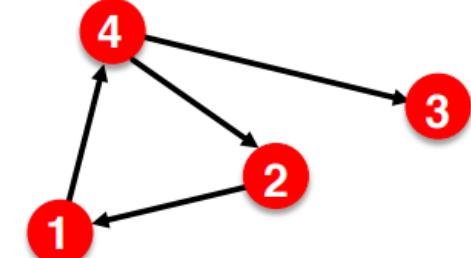
$$\begin{aligned} A_{ij} &= A_{ji} \\ A_{ii} &= 0 \end{aligned}$$

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

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

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

Directed



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

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

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

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

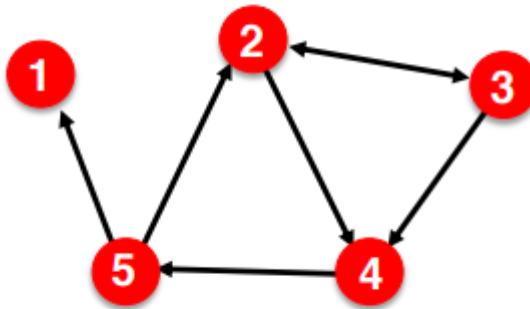
$$L = \sum_{i=1}^N k_i^{in} = \sum_{j=1}^N k_j^{out} = \sum_{i,j} A_{ij}$$

In real world applications, most graphs are sparse -- adjacency matrix is filled with zeros

# Edge list

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

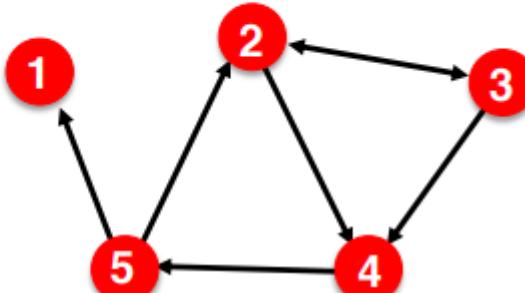
- (2, 3)
- (2, 4)
- (3, 2)
- (3, 4)
- (4, 5)
- (5, 2)
- (5, 1)



# 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



# Comparisons of the 3 representations

Aspect	Adjacency Matrix	Edge List	Adjacency List
<b>Best for</b>	Dense graphs	Sparse graphs, iterating over edges	Most situations, especially sparse graphs
<b>Memory Usage</b>	High ( $O(V^2)$ )	Low ( $O(E)$ )	Moderate ( $O(V + E)$ )
<b>Edge Existence Check</b>	Very efficient ( $O(1)$ )	Inefficient ( $O(E)$ )	Moderate ( $O(V)$ in worst case)
<b>Add/Remove Edge</b>	Efficient ( $O(1)$ )	Varies ( $O(1)$ to $O(E)$ depending on implementation)	Efficient ( $O(1)$ if implemented as a linked list)
<b>Add/Remove Vertex</b>	Inefficient (requires matrix resizing)	Efficient ( $O(1)$ for adding)	Efficient for adding ( $O(1)$ ), varies for removing
<b>Iterate over Neighbors</b>	Efficient ( $O(V)$ )	Inefficient ( $O(E)$ in worst case)	Efficient ( $O(\text{degree of vertex})$ )
<b>Space Efficiency</b>	Inefficient for sparse graphs	Efficient for sparse graphs	Efficient for sparse graphs
<b>Operations Suited</b>	Matrix operations, checking adjacency	Algorithms iterating over edges (e.g., Kruskal's)	Adjacency queries, adding/removing edges

# New Drug Discovery in 2020

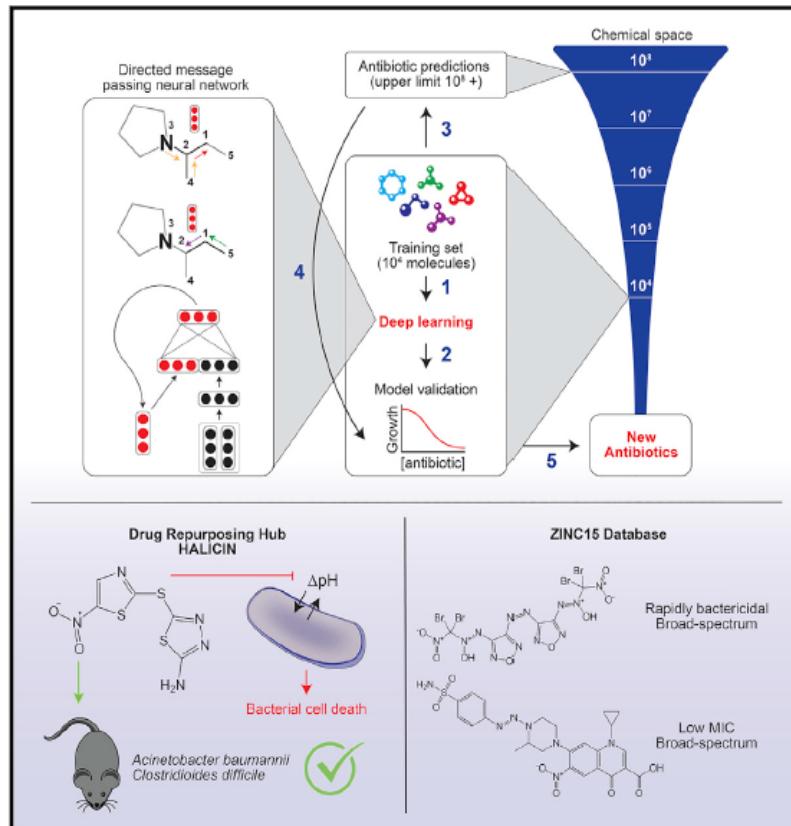
2 Machine Learning Types

Cell

Article

## A Deep Learning Approach to Antibiotic Discovery

### Graphical Abstract



### Authors

Jonathan M. Stokes, Kevin Yang,  
Kyle Swanson, ..., Tommi S. Jaakkola,  
Regina Barzilay, James J. Collins

### Correspondence

regina@csail.mit.edu (R.B.),  
jimjc@mit.edu (J.J.C.)

### In Brief

A trained deep neural network predicts antibiotic activity in molecules that are structurally different from known antibiotics, among which *Halicin* exhibits efficacy against broad-spectrum bacterial infections in mice.

## Authors

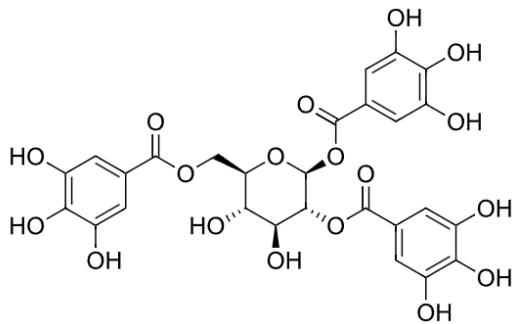
- Jonathan M. Stokes, Kevin Yang,
- Kyle Swanson, ..., Tommi S. Jaakkola,
- Regina Barzilay, James J. Collins

## In Brief

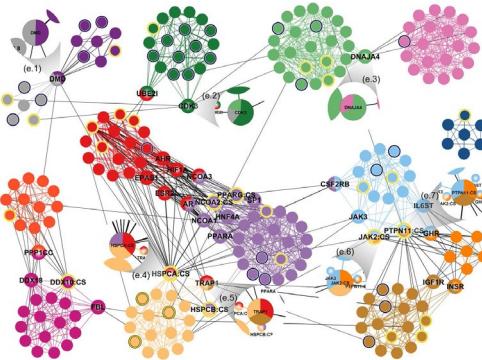
A deep neural network predicts antibiotic activity in molecules that are structurally different from known antibiotics, among which *Halicin* exhibits efficacy against broad-spectrum bacterial infections in mice.

# Graph Data is everywhere

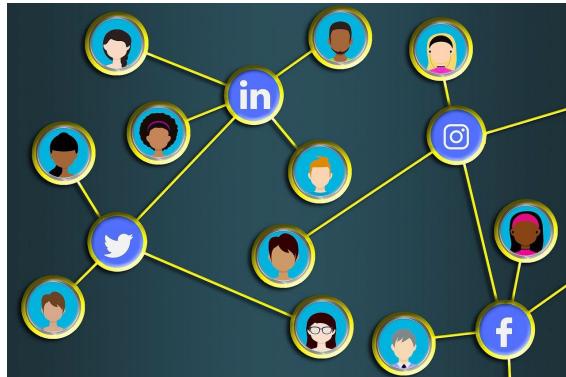
2 Machine Learning Types



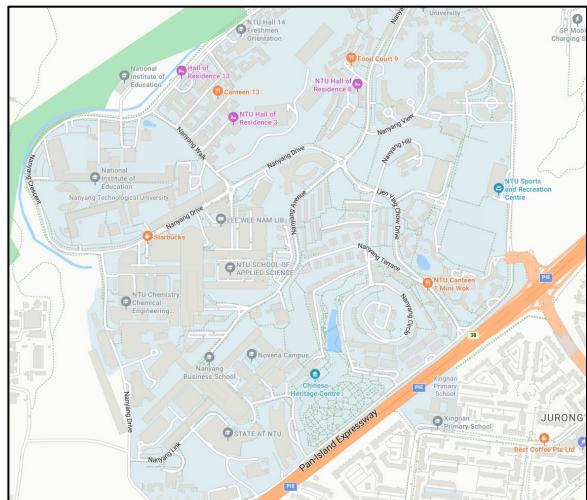
Molecules



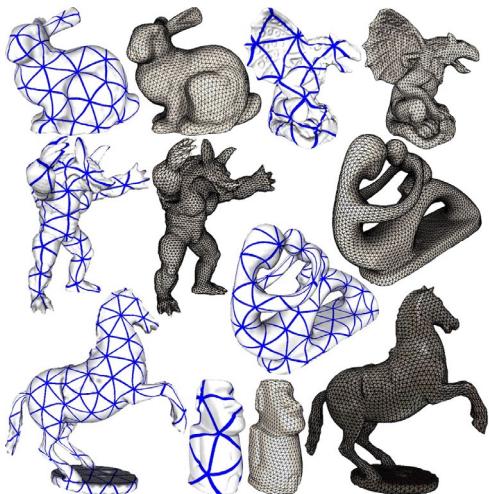
Medicine / Pharmacy



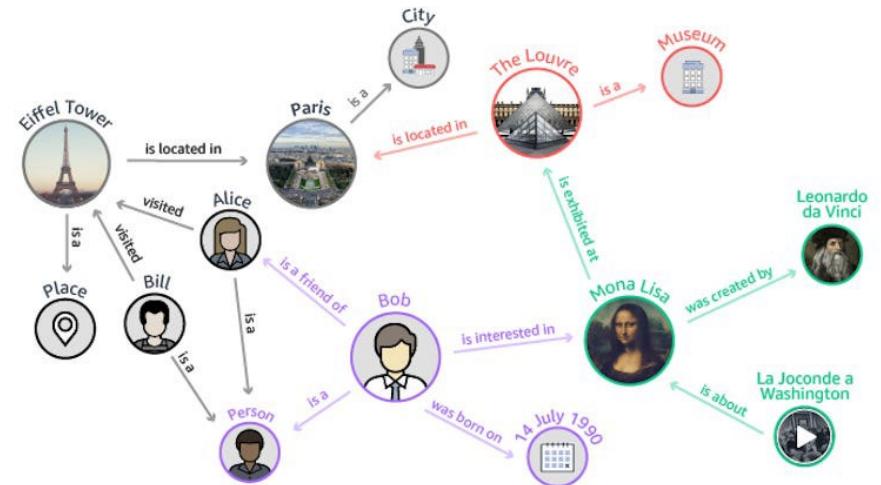
Social Networks



Map of NTU Campus



3D Games / Meshes



Knowledge Graph

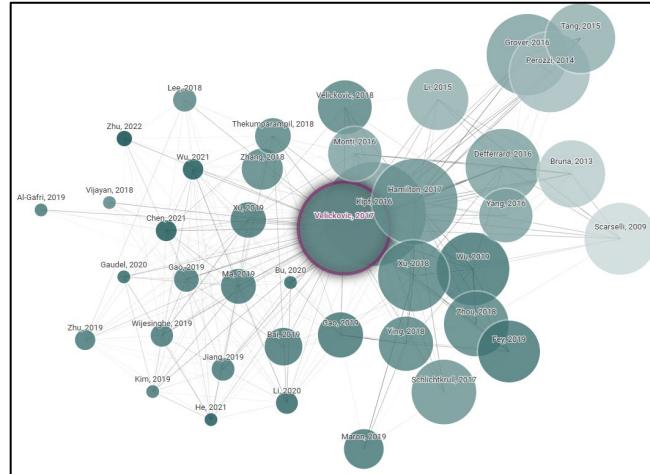
Most real-world data is not as structured and regular as grid data

# Graph Data is everywhere

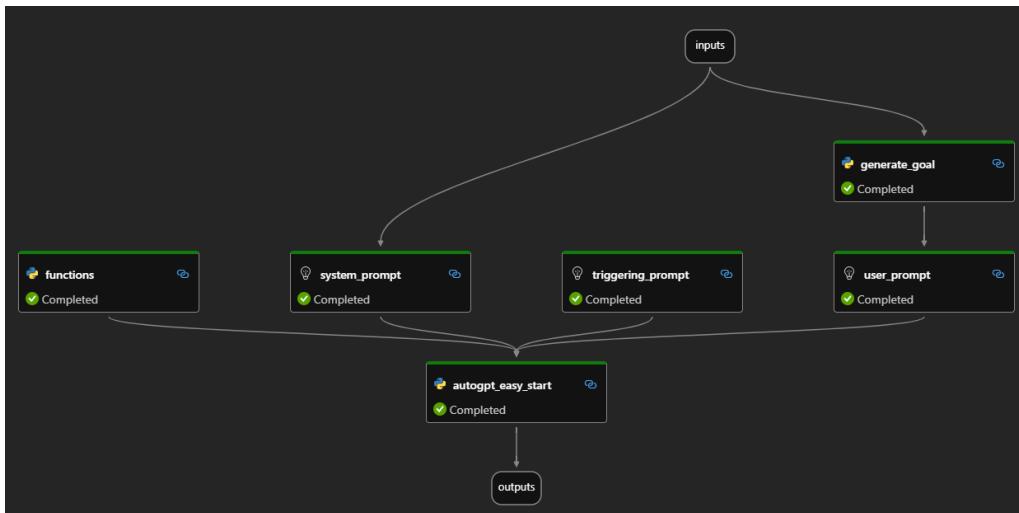
2 Machine Learning Types



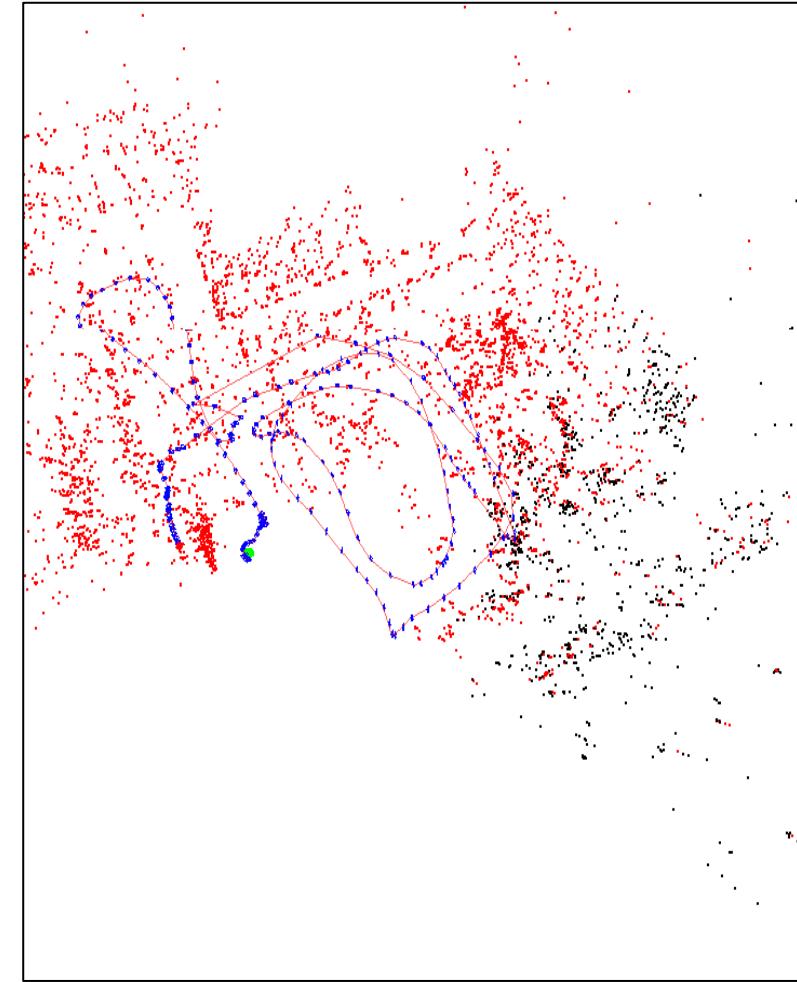
Crime board pinned with Evidence



Citation Network



Code Graph

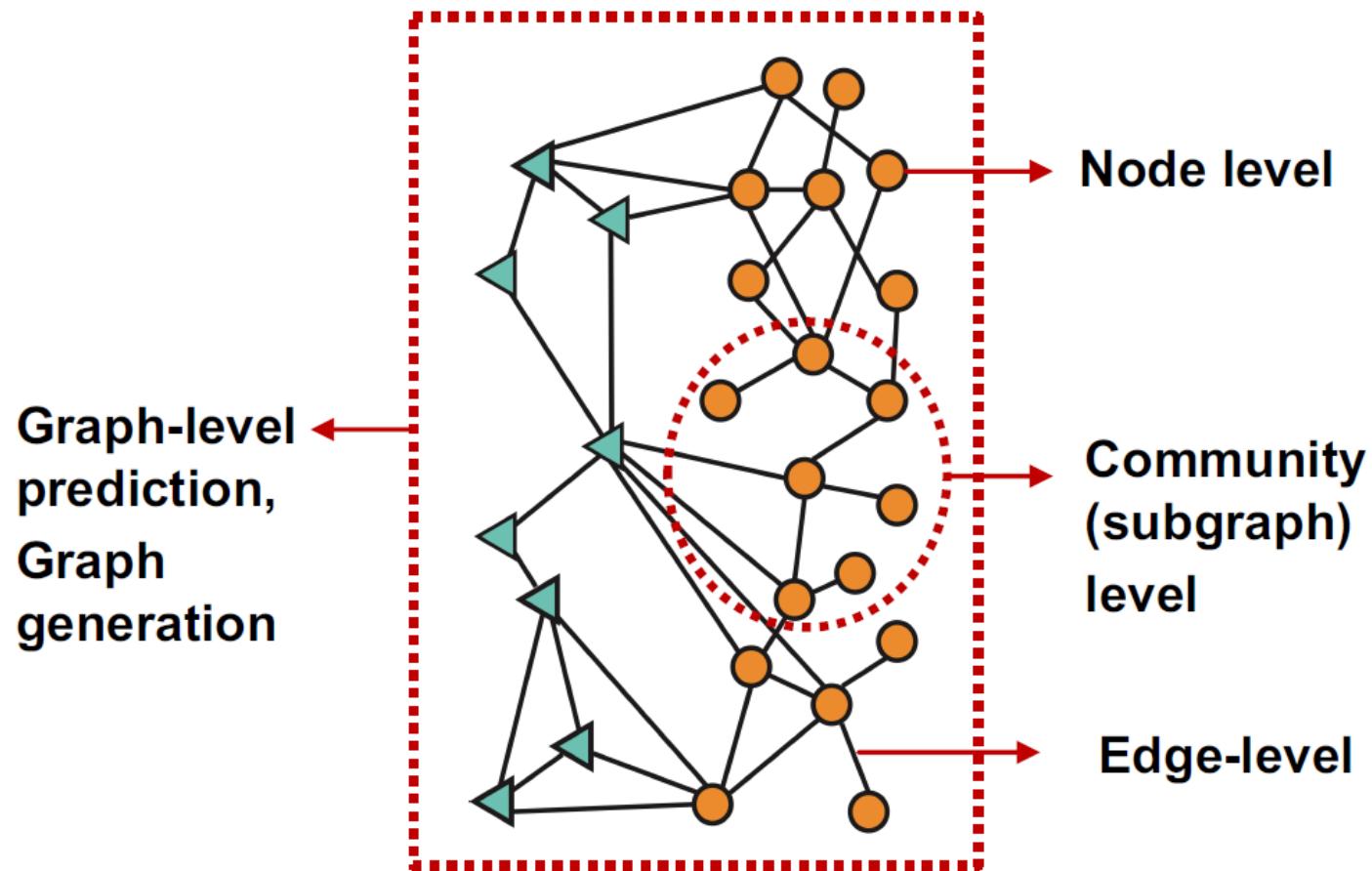


Robot Landmark Map

Most real-world data is not as structured and regular as grid data

# ML Problems with Graph Data

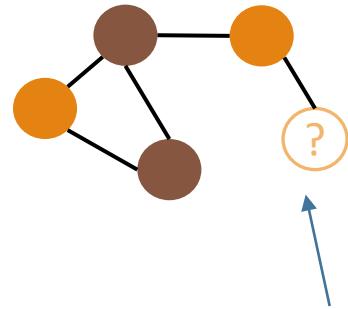
2 Machine Learning Types



# ML Problems with Graph Data

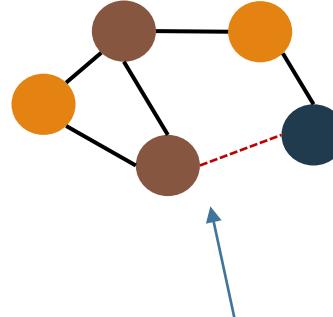
# 2 Machine Learning Types

## $f(\bullet)$ Node-level predictions



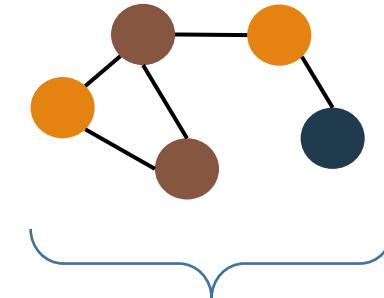
Does this person smoke?  
(Unlabeled node)

$f(\text{---})$



Facebook this person you might know?  
(Friend recommendation)

# $f(\text{graph})$ Graph-level predictions



Is this molecule a suitable drug?

## Predict a property of a node

## Categorize online users / items

# Predict whether there are missing links between two nodes

## Knowledge graph completion

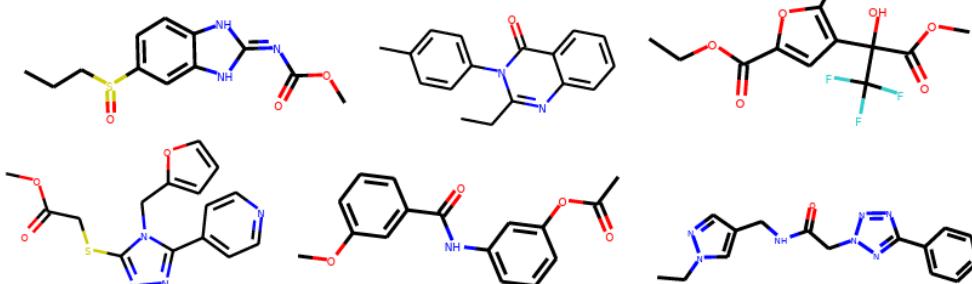
## Categorize different graphs

## Molecule property prediction

# ML Problems with Graph Data

2 Machine Learning Types

## ■ Node Regression: ZINC molecule graph dataset [1]



Model	$L$	AQSOL[2]				
		#Param	TestMAE±s.d.	TrainMAE±s.d.	Epochs	Epoch/Total
MLP	4	114525	1.744±0.016	1.413±0.042	85.75	0.61s/0.02hr
<i>vanilla</i> GCN	4	108442	1.483±0.014	0.791±0.034	110.25	1.14s/0.04hr
	16	511443	1.458±0.011	0.567±0.027	121.50	2.83s/0.10hr
GraphSage	4	109620	1.431±0.010	0.666±0.027	106.00	1.51s/0.05hr
	16	509078	1.402±0.013	0.402±0.013	110.50	3.20s/0.10hr
GCN	4	108442	1.372±0.020	0.593±0.030	135.00	1.28s/0.05hr
	16	511443	1.333±0.013	0.382±0.018	137.25	3.31s/0.13hr
MoNet	4	109332	1.395±0.027	0.557±0.022	125.50	1.68s/0.06hr
	16	507750	1.501±0.056	0.444±0.024	110.00	3.62s/0.11hr
GAT	4	108289	1.441±0.023	0.678±0.021	104.50	1.92s/0.06hr
	16	540673	1.403±0.008	0.386±0.014	111.75	4.44s/0.14hr
GatedGCN	4	108325	1.352±0.034	0.576±0.056	142.75	2.28s/0.09hr
	16	507039	1.355±0.016	0.465±0.038	99.25	5.52s/0.16hr
GatedGCN-E	4	108535	1.295±0.016	0.544±0.033	116.25	2.29s/0.08hr
	16	507273	1.308±0.013	0.367±0.012	110.25	5.61s/0.18hr
GatedGCN-E-PE	16	507663	<b>0.996±0.008</b>	0.372±0.016	105.25	5.70s/0.30hr

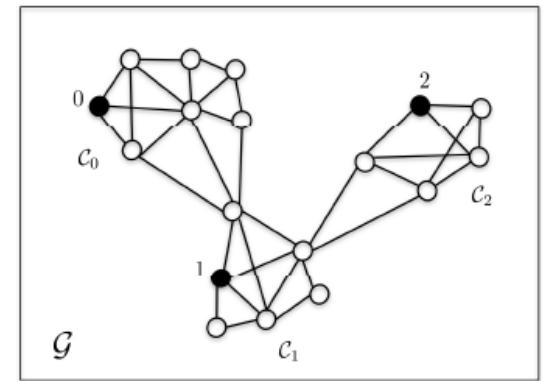
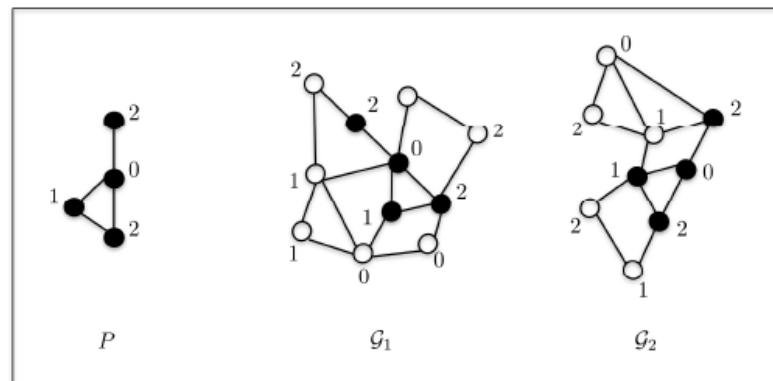
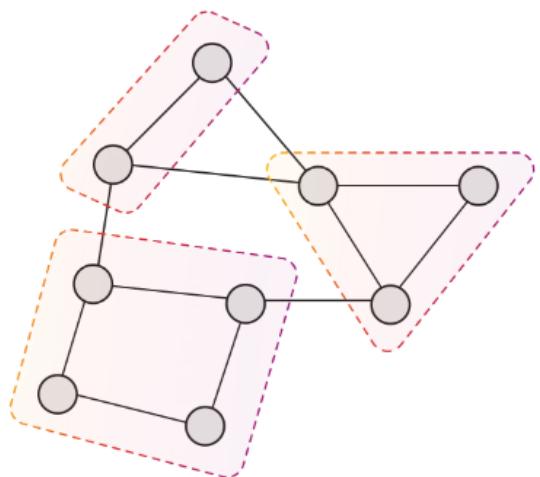
[1] Sterling, Teague, and John J. Irwin. "ZINC 15-ligand discovery for everyone." *Journal of chemical information and modeling* 55.11 (2015): 2324-2337.

[2] Dwivedi, Vijay Prakash, et al. "Benchmarking graph neural networks." *Journal of Machine Learning Research* 24.43 (2023): 1-48.

# ML Problems with Graph Data

2 Machine Learning Types

- Clustering
- Subgraph: Stochastic Block Model dataset [1]



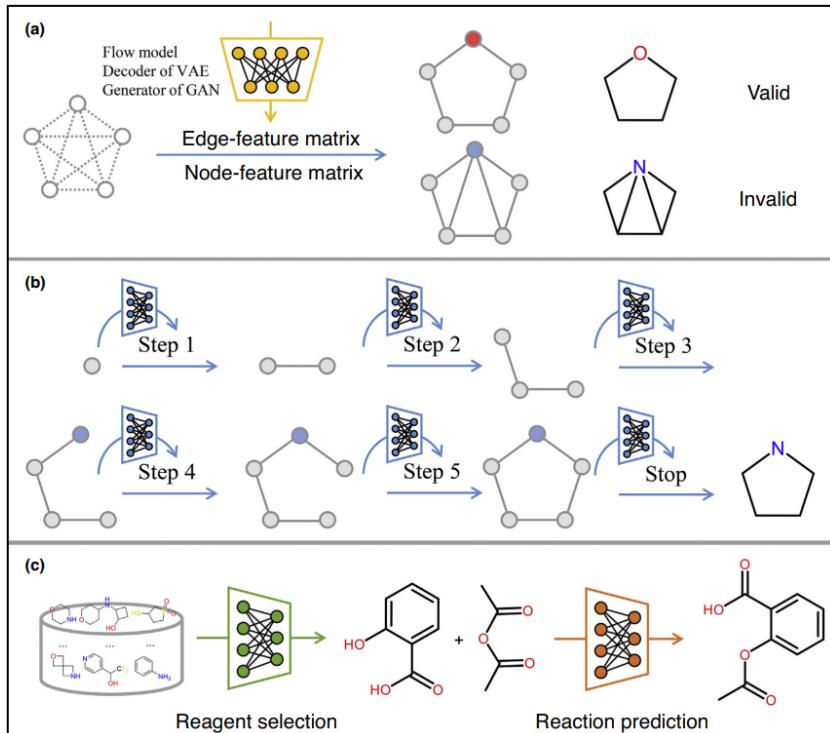
**Detect if nodes form a community,**  
like social circle detection

[1] Bresson, Xavier, and Thomas Laurent. "Residual gated graph convnets." *arXiv preprint arXiv:1711.07553* (2017).

# ML Problems with Graph Data

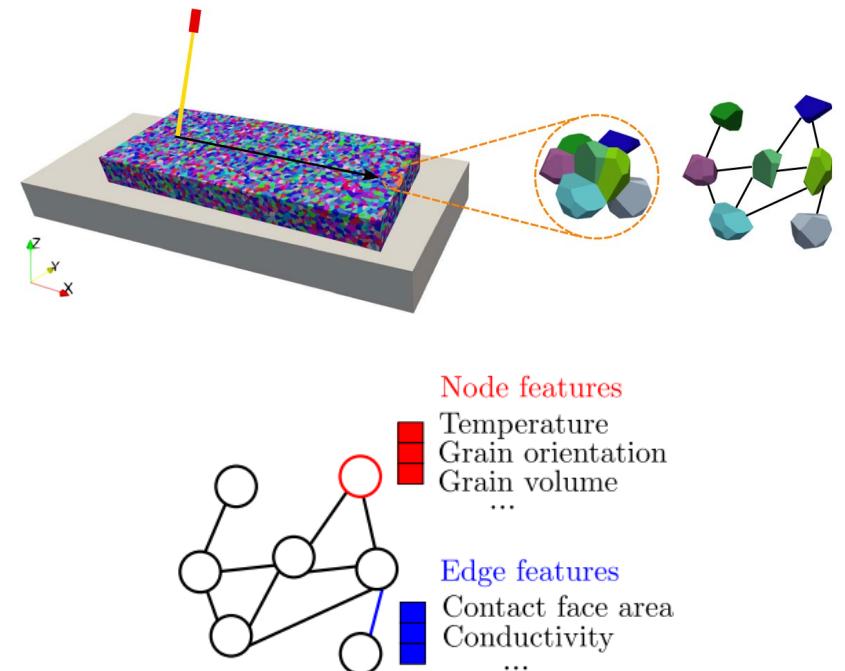
2 Machine Learning Types

## Graph Generation



Generate novel, realistic graphs with desired properties,  
Drug discovery [1]

## Graph Evolution



Generate simulations of complex physics  
over time, Physical simulation [2]

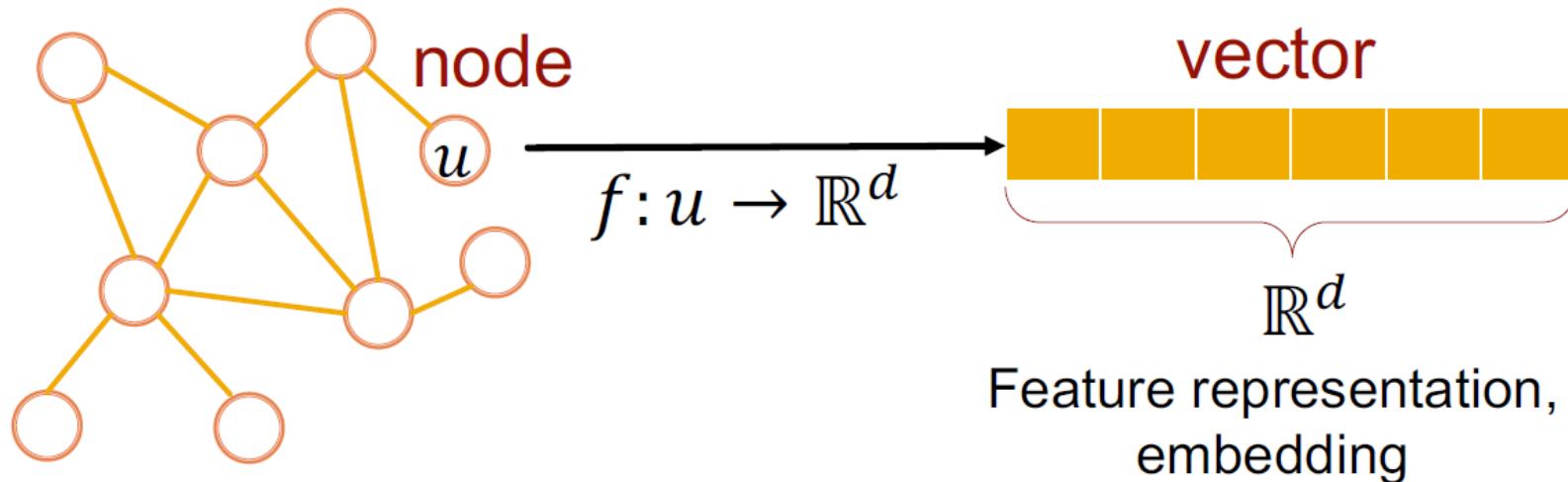
[1] Xiong, Jiacheng, et al. "Graph neural networks for automated de novo drug design." *Drug Discovery Today* 26.6 (2021): 1382-1393.

[2] Sanchez-Gonzalez, Alvaro, et al. "Learning to simulate complex physics with graph networks." *International conference on machine learning*. PMLR, 2020.

# Graph Representation Learning

2 Machine Learning Types

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



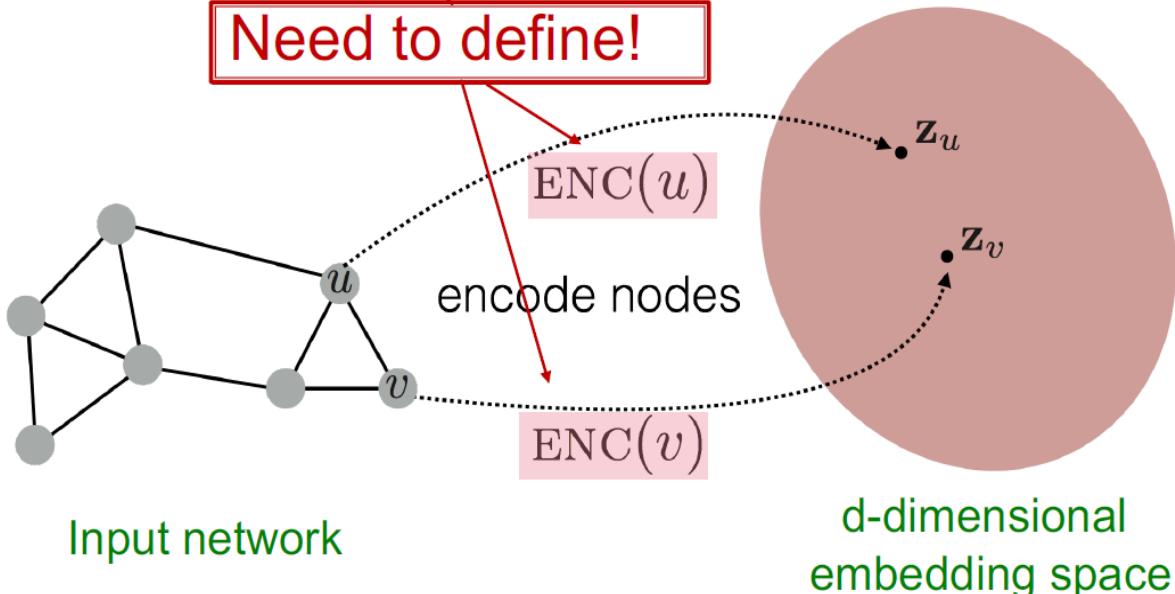
How to learn the mapping function  $f$ ?

# Graph Representation Learning

2 Machine Learning Types

Goal:  $\text{similarity}(u, v) \approx z_v^T z_u$

Need to define!



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

$\text{ENC}(v) = z_v$   $d$ -dimensional embedding  
node in the input graph

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

$$\text{similarity}(u, v) \approx z_v^T z_u$$

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

**Decoder**  
dot product between node embeddings

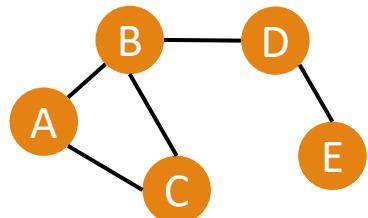
Shallow encoding: encoder is just an embedding-lookup

# Graph Representation Learning

- 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
- Inductive
  - Train on fully labelled graph; Test on unknown nodes added.
- Transduction learning
  - Train on partially labelled graph; Test on partially unlabelled graph.
  - Transduction predict specific output for specific examples within the same data distribution as the training data.

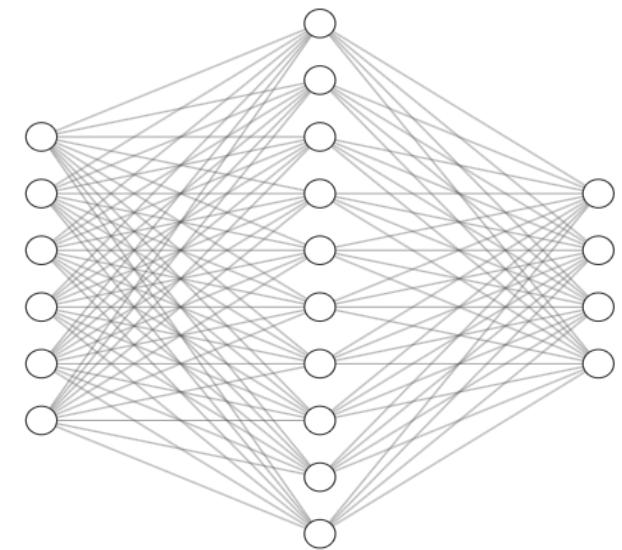
# Deep Graph Encoders

- We will discuss deep neural networks on graph: **Graph Neural Networks (GNNs)**
  - $\text{ENC}(v)$  = Multiple layers of perceptrons based on graph structure



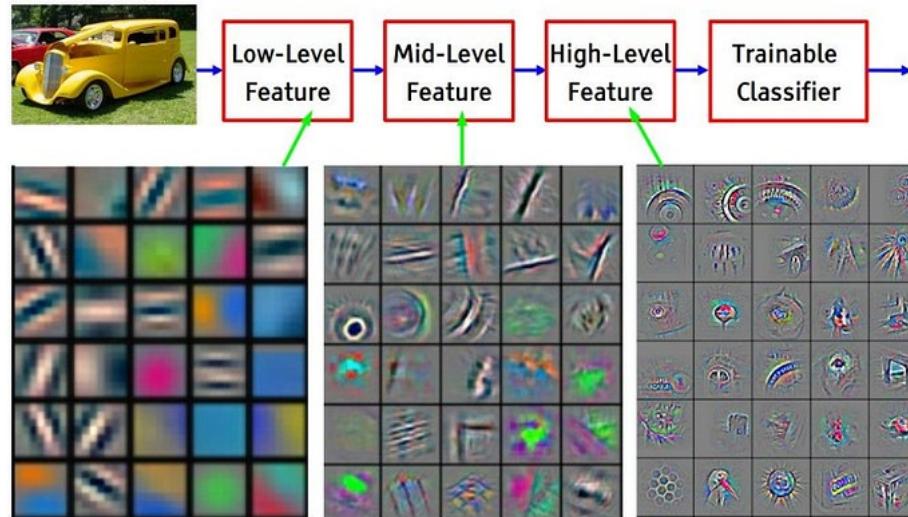
$$G = (V, E)$$

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



# Neural Networks we have learned

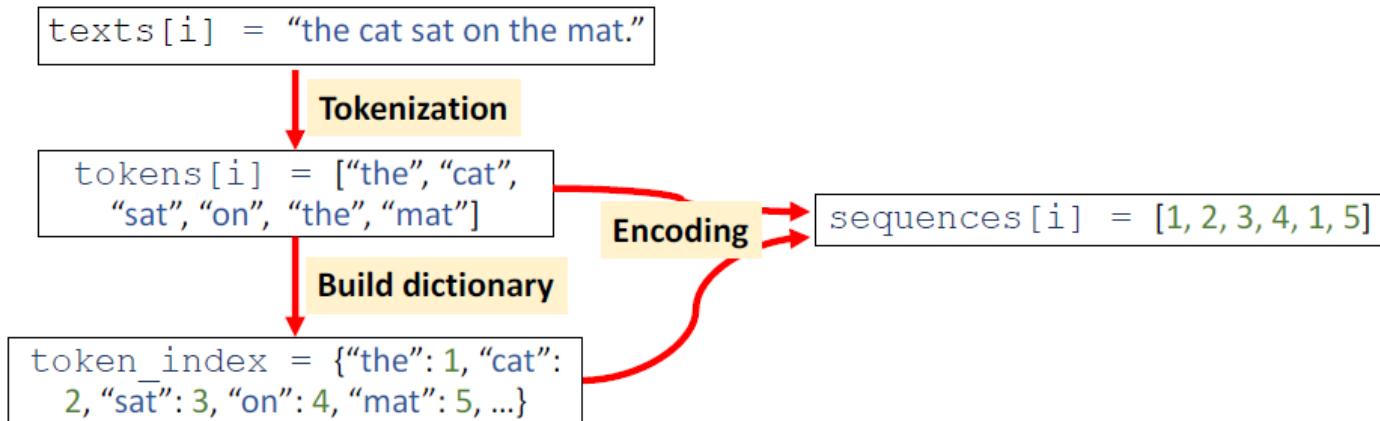
## 3 Graph Neural Networks: Basics



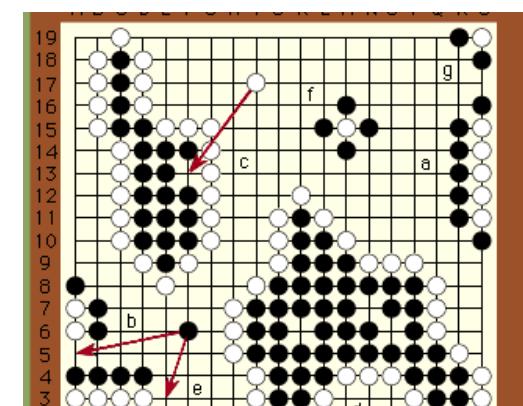
Feature map of image data: convolution



Speech data: spectrum

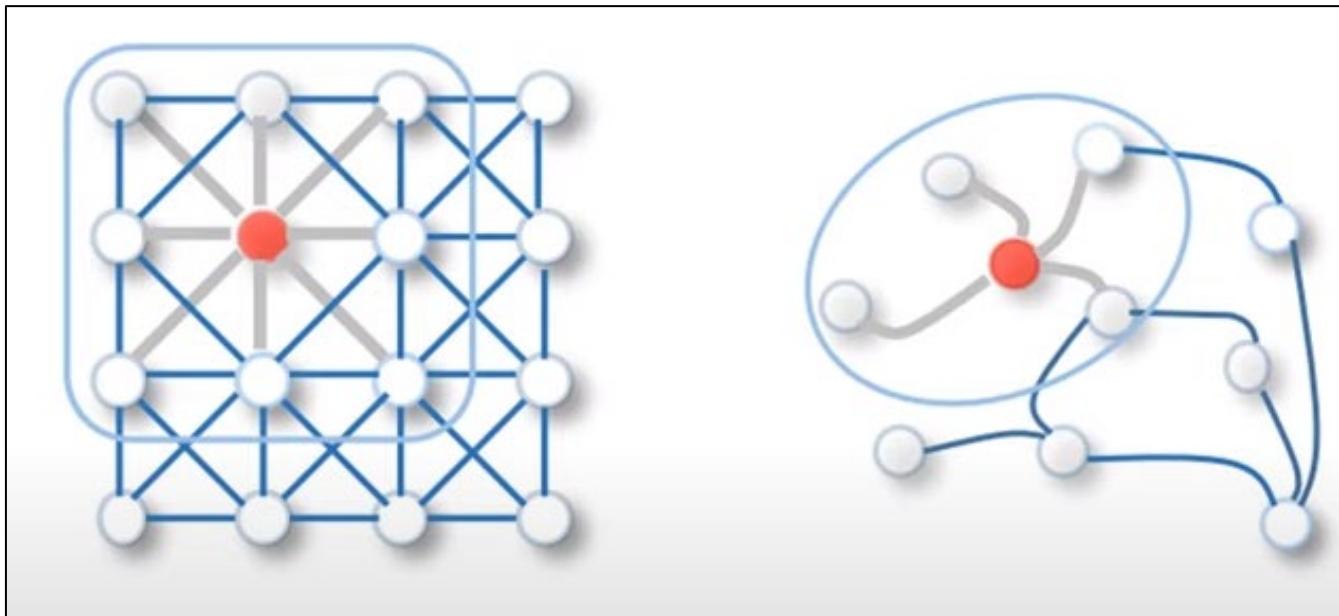


Natural language processing: tokenization and encoding



Grid based games: Go

# Differences with image and graph data



- Arbitrary size
  - Complex topological structure
  - No fixed node ordering
  - Often dynamic and have multimodal features
- 
- There is no fixed notion of locality or sliding window on the graph
  - Graph is **permutation invariant**

# Setup (Notation)

- Assume we have a graph  $G$ :

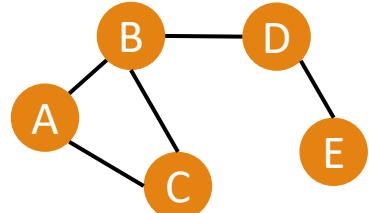
- $V$  is the **vertex set**
- $A$  is the **adjacency matrix** (assume binary)
- $X \in \mathbb{R}^{|V| \times m}$  is a matrix of **node features**
- $v$ : a node in  $V$ ;  $N(v)$ : the set of neighbors of  $v$ .
- **Node features:**
  - Social networks: User profile, User image
  - Biological networks: 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:  $[1, 1, \dots, 1]$

We often denote each neighbour of node  $v$  as  $u \in N(v)$

This is called feature augmentation

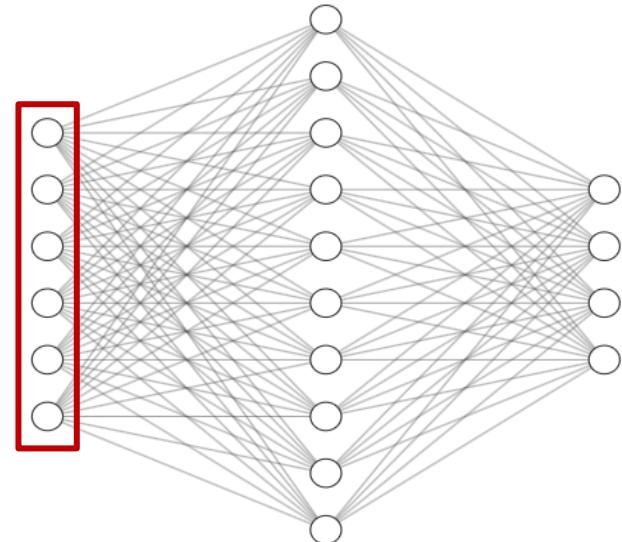
# Naïve MLP Approach

- Concatenate adjacency matrix and features, then feed into a deep neural network



$$G = (V, E)$$

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

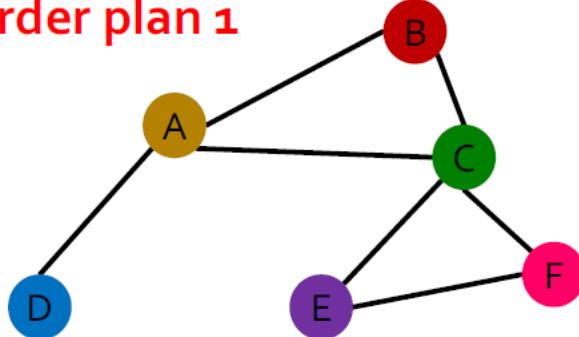


- Issues
  - Parameters  $O(|V|)$
  - Not applicable to graphs of different sizes
  - Sensitive to node ordering

# Permutation Invariance

## 3 Graph Neural Networks: Basics

Order plan 1



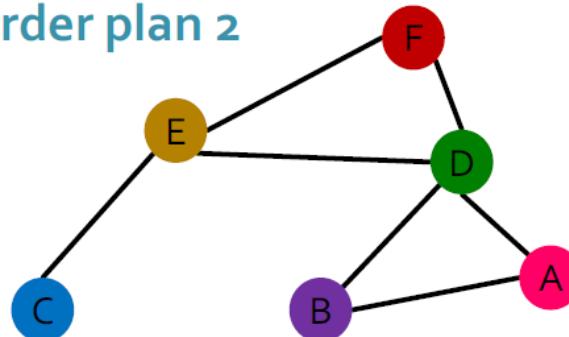
Node features  $X_1$

A	[Color bar]
B	[Color bar]
C	[Color bar]
D	[Color bar]
E	[Color bar]
F	[Color bar]

Adjacency matrix  $A_1$

A	B	C	D	E	F
A	1	1	0	0	0
B	0	1	1	0	0
C	1	0	1	1	0
D	0	0	1	0	1
E	0	0	0	1	1
F	0	0	0	0	1

Order plan 2



Node features  $X_2$

A	[Color bar]
B	[Color bar]
C	[Color bar]
D	[Color bar]
E	[Color bar]
F	[Color bar]

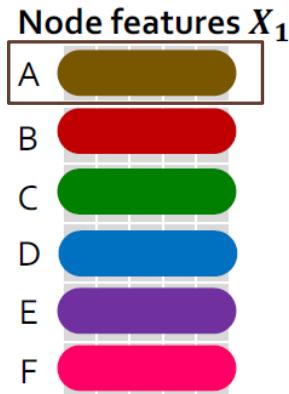
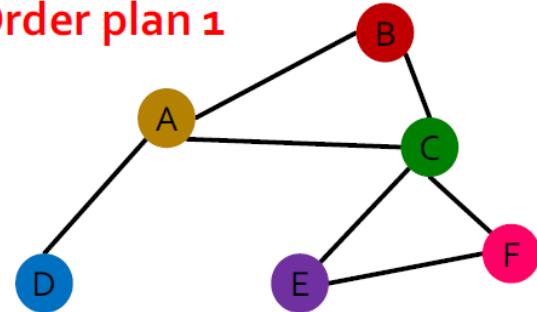
Adjacency matrix  $A_2$

A	B	C	D	E	F
A	1	1	0	0	0
B	0	1	1	0	0
C	1	0	1	1	0
D	0	0	1	0	1
E	0	0	0	1	1
F	0	0	0	0	1

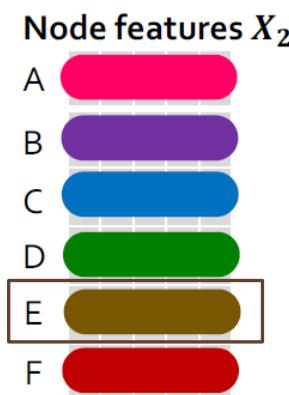
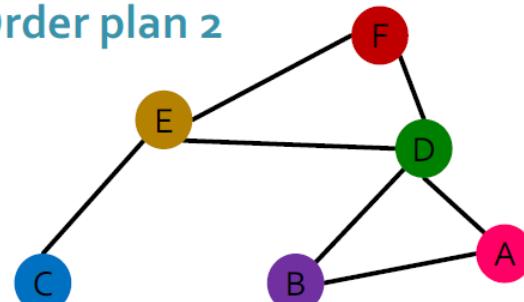
- Graph representations should be the same for Order plan 1 and Order plan 2
- Suppose a function  $f$  maps a graph to a  $d$ -dim embedding, then  $f(A_1, X_1) = f(A_2, X_2)$
- Definition:** For any graph function  $f$ , it is **permutation-invariant** if  $f(A, X) = f(PAP^T, PX)$  for any permutation  $P$ .
- Permute the input, the output stays the same.

# Permutation Equivariance

Order plan 1

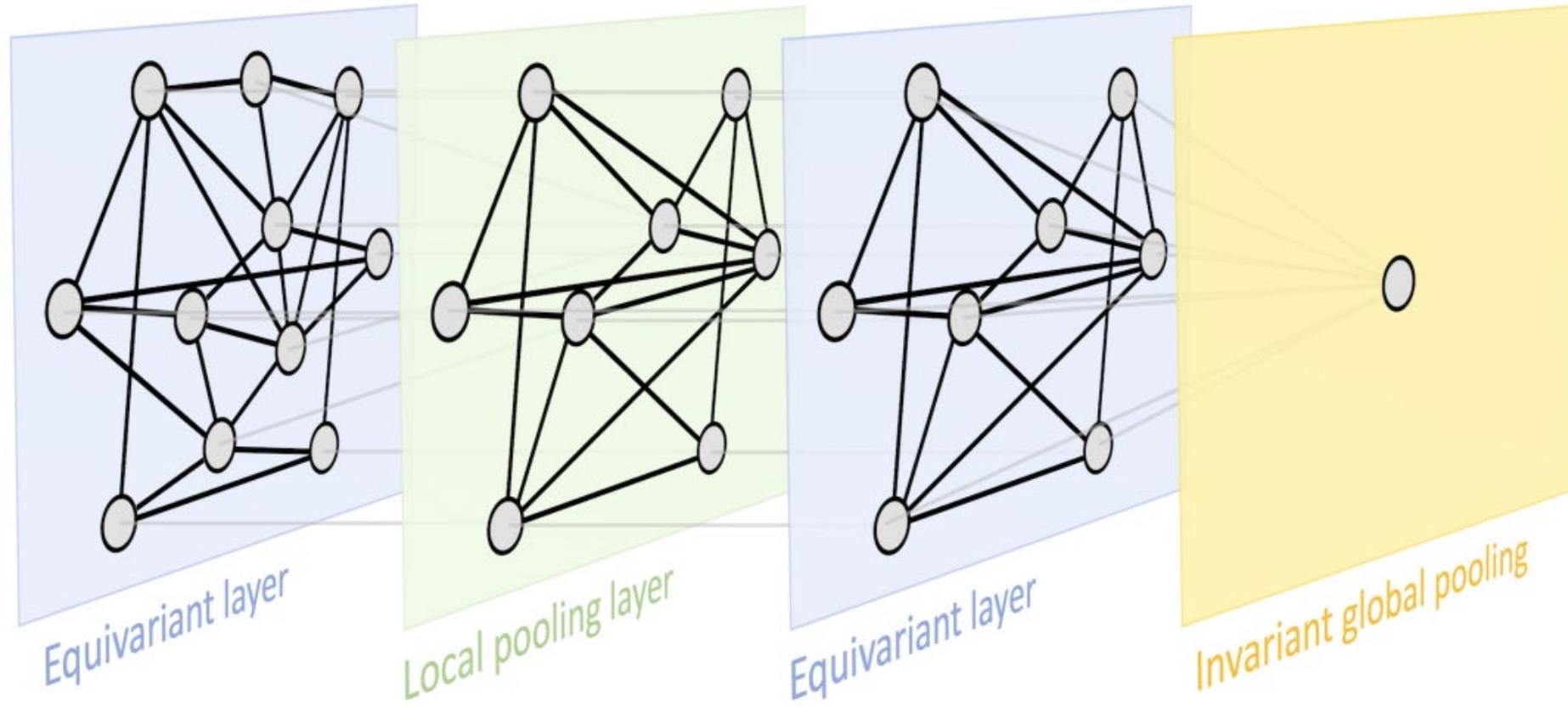


Order plan 2



- **Node representations:** Learn a function  $f$  that maps nodes of  $G = (A, X)$  to a matrix  $\mathbb{R}^{|V| \times d}$
- For two order plans, the vector of node at the same position in the graph is the same.
- If the output vector of a node at the same position in the graph remains unchanged for any order, we say  $f$  is **permutation equivariant**.
  
- **Definition:** For any **node** function  $f$ , it is **permutation-equivariant** if  $Pf(A, X) = f(PAP^T, PX)$  for any permutation  $P$ .
- Permute the input, output also permutes accordingly.

# Graph Neural Network Overview

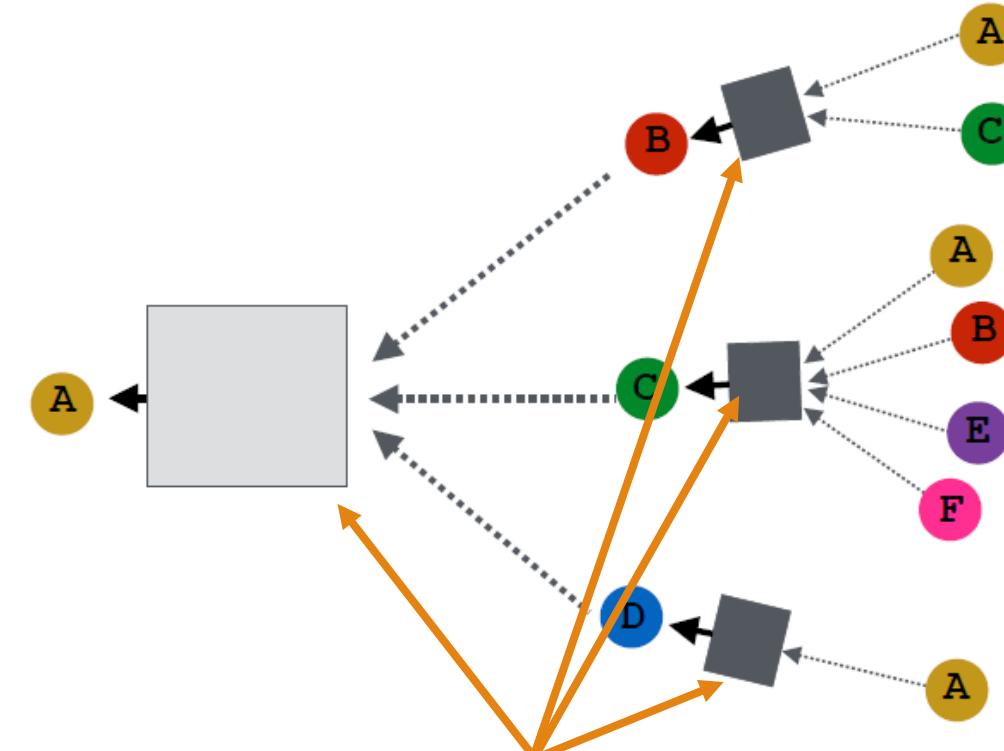
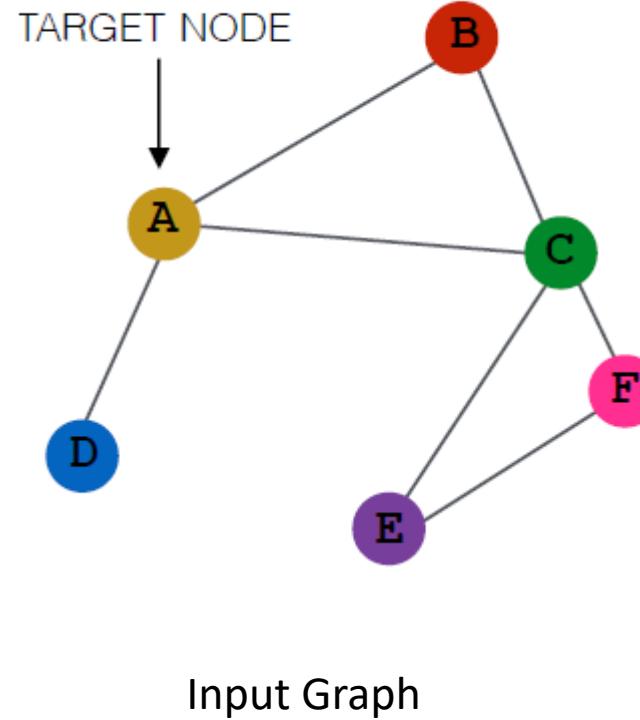


Geometric Deep Learning blueprint, exemplified on a graph. A typical Graph Neural Network architecture may contain permutation equivariant layers (computing node-wise features), local pooling (graph coarsening), and a permutation-invariant global pooling layer (readout layer) [1].

[1] Bronstein, Michael M., et al. "Geometric deep learning: Grids, groups, graphs, geodesics, and gauges." *arXiv preprint arXiv:2104.13478* (2021).

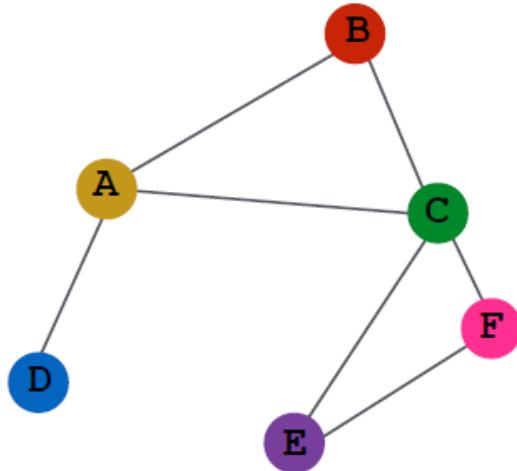
# Idea: Aggregate neighbours

- **Key Idea:** Generate node embeddings based on local network neighbourhoods

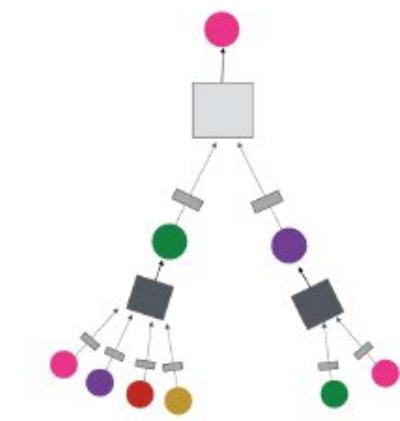
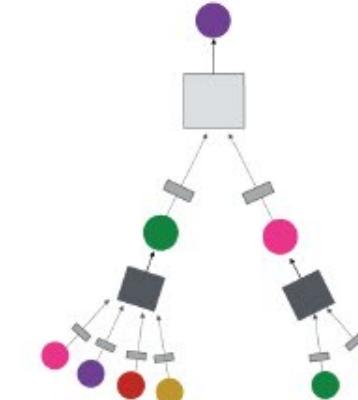
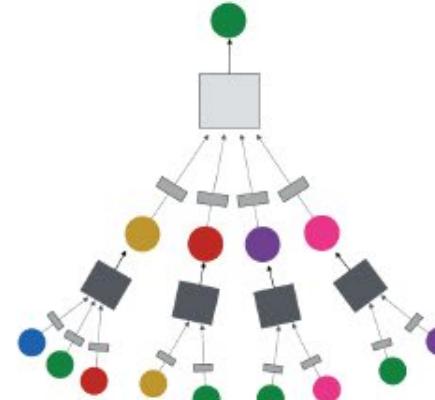
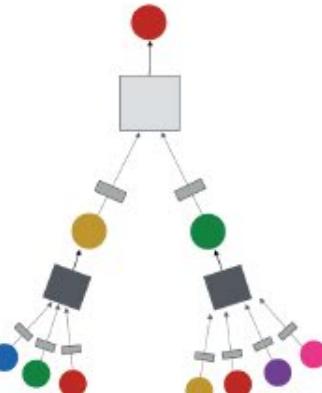
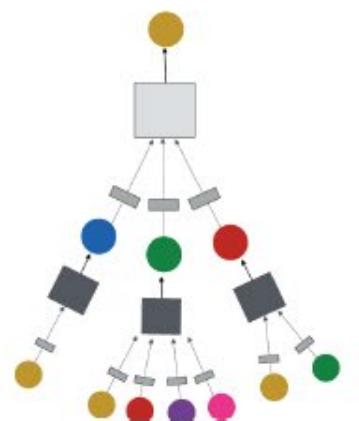


- **Via neural networks**

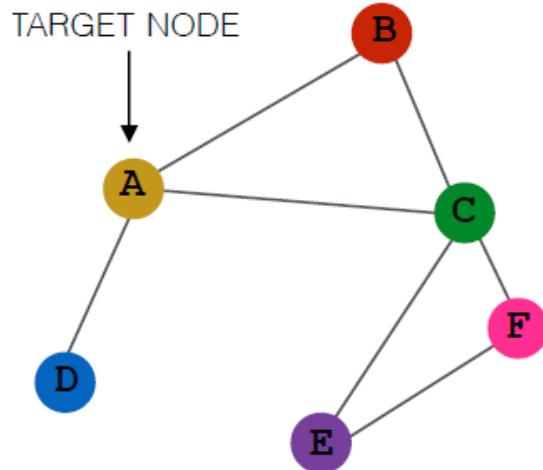
# Idea: Aggregate neighbours from 1 layer



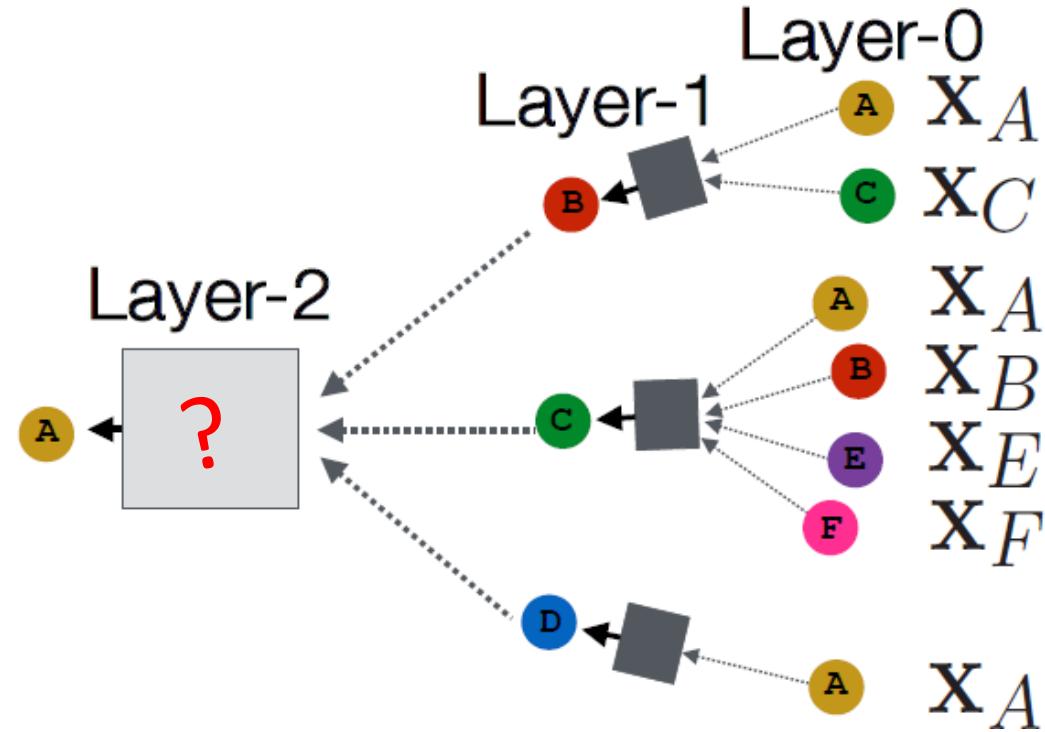
- Every node defines a computation graph based on its neighbourhood



# Idea: Aggregate neighbours across layers



- Model can be of arbitrary depth
  - Layer-0 embedding of node  $v$  is the input feature  $x_v$
  - Layer- $k$  embedding gets information from nodes  $k$  hops away



# What is a GNN/DGN

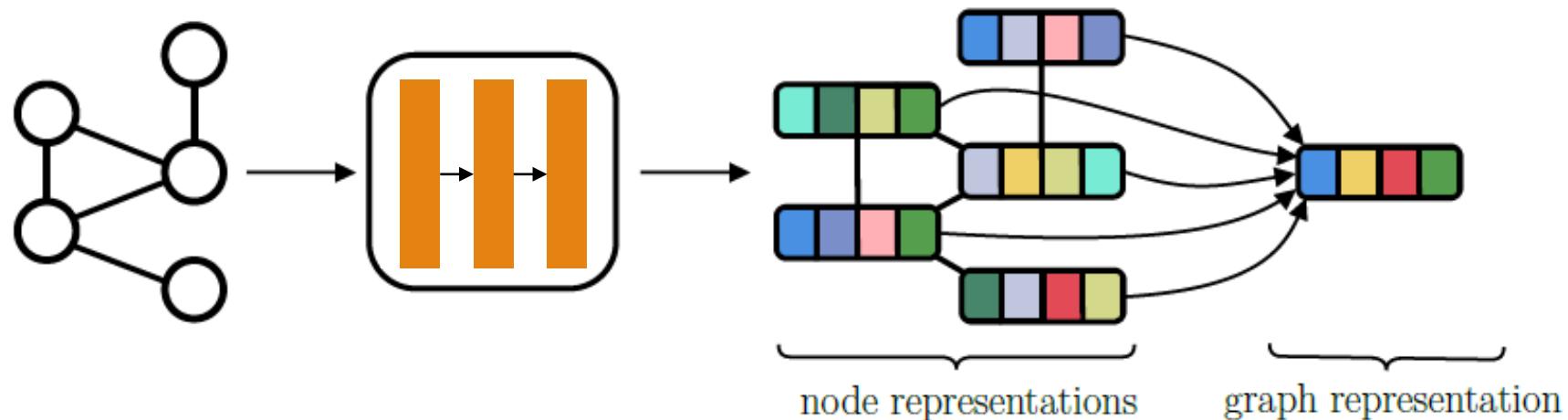


Figure 2: The bigger picture that all graph learning methods share. A “Deep Graph Network” takes an input graph and produces node representations  $h_v \forall v \in \mathcal{V}_g$ . Such representations can be aggregated to form a single graph representation  $h_g$ .

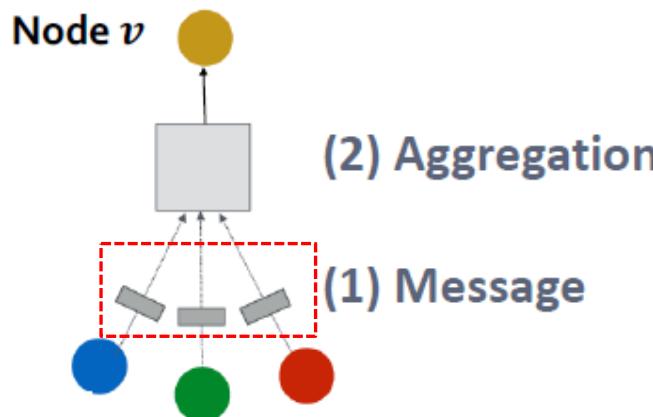
By passing and aggregating information from neighbours, we could design permutation invariant / equivariant networks!

Via Message Passing Layers

# A Single GNN Layer

## 3 Graph Neural Networks: Basics

- Construct a GNN Layer
  - 1) Message passing
  - 2) Aggregation
  - 3) Update
  - 4) Readout (Graph-level)



### ■ (1) Message computation

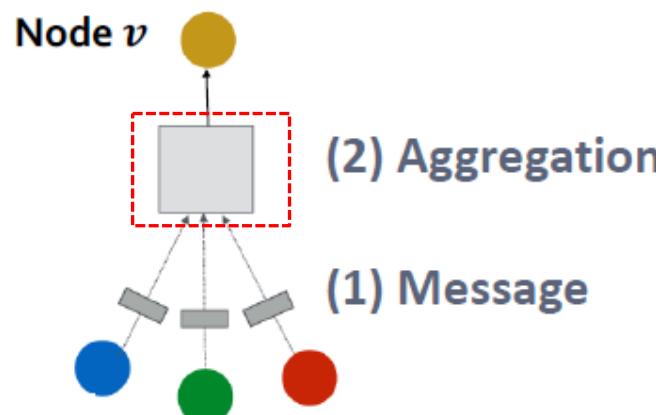
- **Message function:**  $\mathbf{m}_u^{(l)} = \text{MSG}^{(l)}(\mathbf{h}_u^{(l-1)})$ 
  - **Intuition:** Each node will create a message, which will be sent to other nodes later
  - **Example:** A Linear layer  $\mathbf{m}_u^{(l)} = \mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}$ 
    - Multiply node features with weight matrix  $\mathbf{W}^{(l)}$
  - To avoid information loss from node  $v$ , also include  $\mathbf{h}_v^{(l-1)}$
- **Message:** compute message from node  $v$  itself
  - Usually, a **different message computation** will be performed

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

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

# A Single GNN Layer

- Construct a GNN Layer
  - 1) Message passing
  - 2) Aggregation
  - 3) Update
  - 4) Readout (Graph-level)



## ■ (2) Aggregation

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

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

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

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

## ■ (3) Update

- After aggregating from neighbors, we aggregate the message from node  $v$  itself, with an “update” operation.

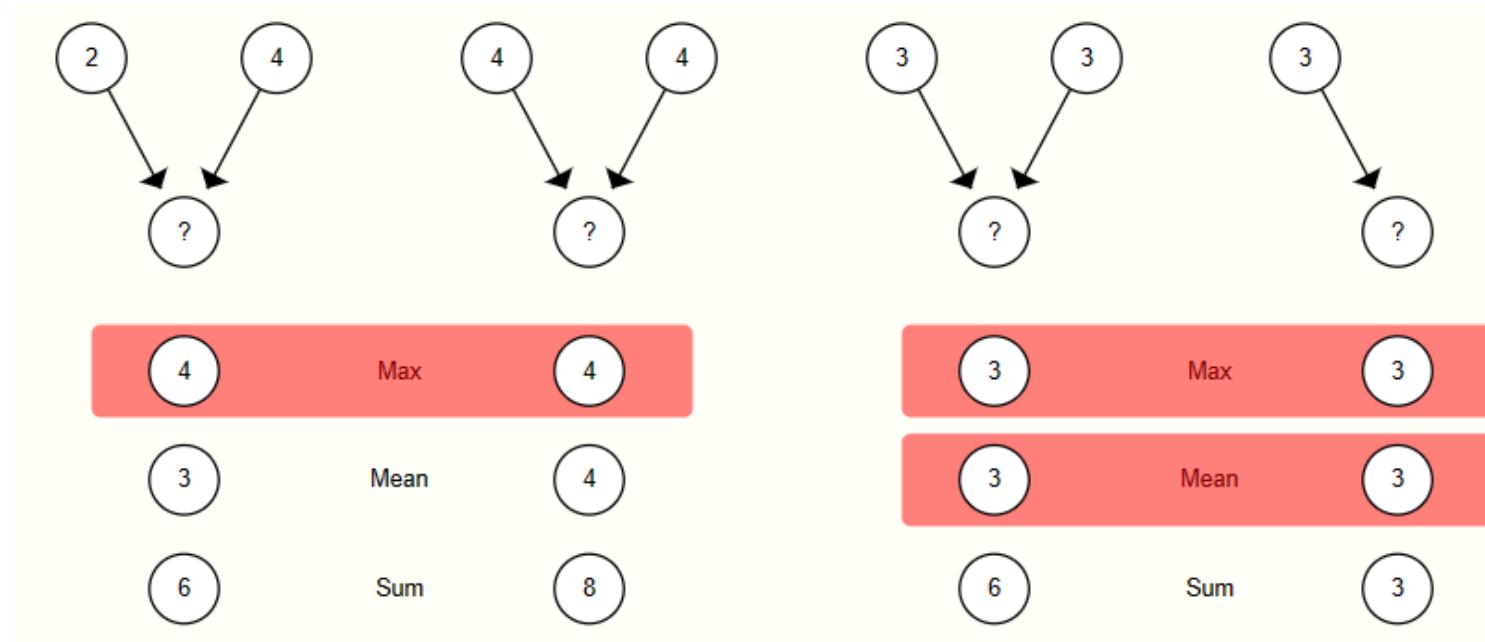
Then prepare the node itself

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

Update to new state of node  $v$

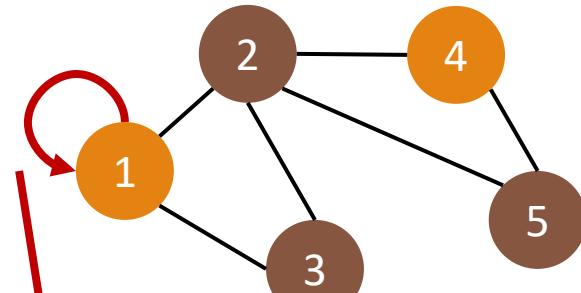
First aggregate from neighbours

# Aggregation Operations



There is no operation that is uniformly the best choice.

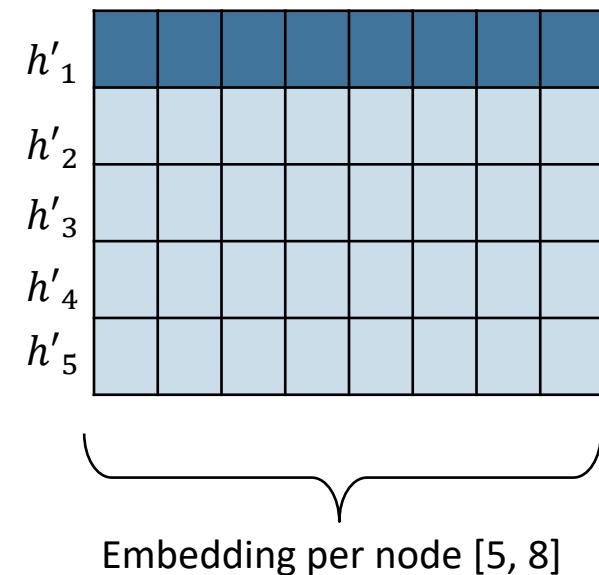
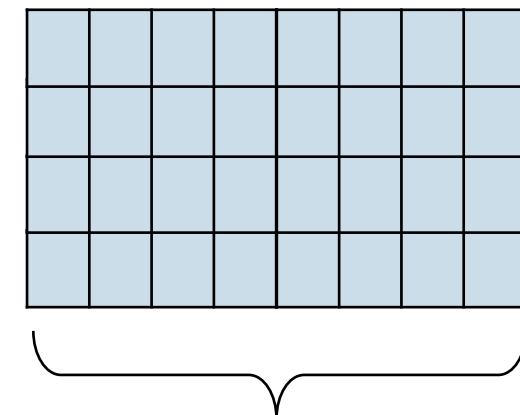
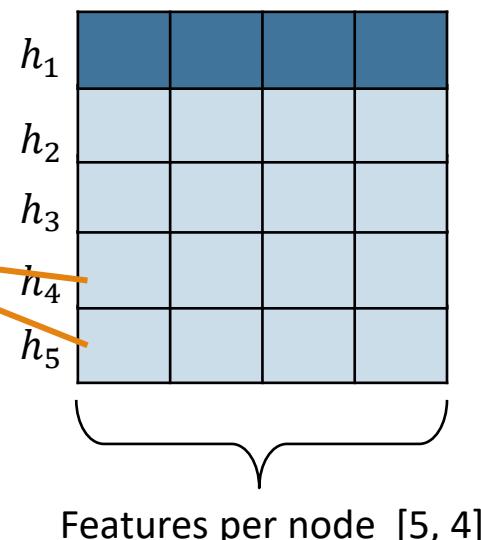
# Matrix Multiplication Process



Adjacency Matrix [5, 5]

1	1	1	0	0
1	1	1	1	1
1	1	1	0	0
0	1	0	1	1
0	1	0	1	1

$$\mathbf{h}'_v = \sigma \left( \sum_{u \in N(v)} \mathbf{W} * \mathbf{h}_u \right)$$



# A Single GNN Layer

## 3 Graph Neural Networks: Basics

### ■ Putting things together:

- (1) Message: each node computes a message

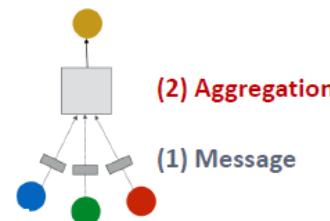
$$\mathbf{m}_u^{(l)} = \text{MSG}^{(l)} \left( \mathbf{h}_u^{(l-1)} \right), u \in \{N(v) \cup v\}$$

- (2) Aggregation: aggregate messages from neighbors

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

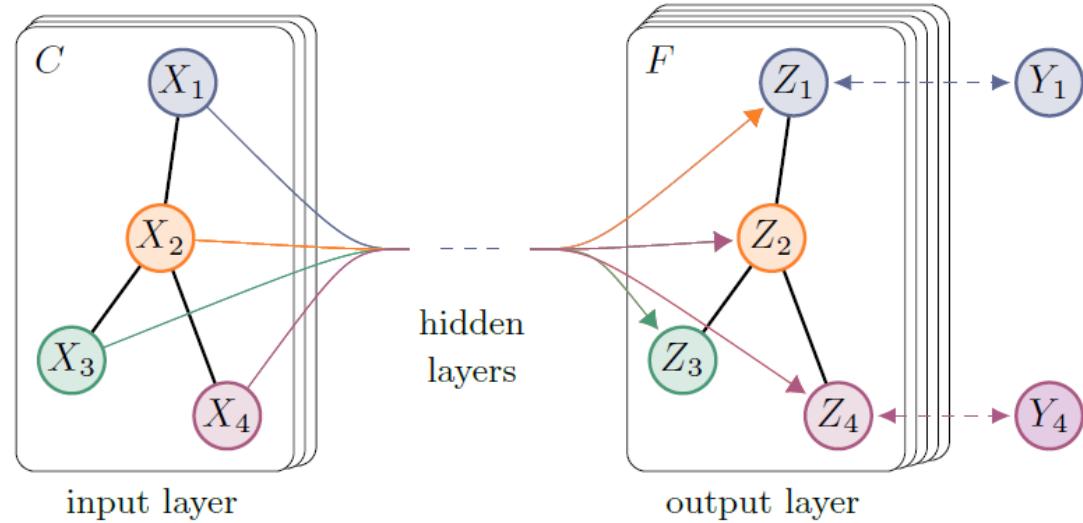
- Nonlinearity (activation): Adds expressiveness

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



# 4.1 Graph Convolution Networks

4 GNN Variants



- Multi-layer Graph Convolutional Network with  $C$  input channels and  $F$  feature maps in the output layer (3-layer)
  - The graph structure is shared over layers
  - Labels denoted by  $Y_i$
- Input-to-hidden weight matrix  $W^{(0)} \in \mathbb{R}^{C \times H}$  for a hidden layer with  $H$  feature maps
- Hidden-to-output weight matrix  $W^{(1)} \in \mathbb{R}^{H \times F}$ .

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



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

# 4.1 Graph Convolution Networks

4 GNN Variants

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



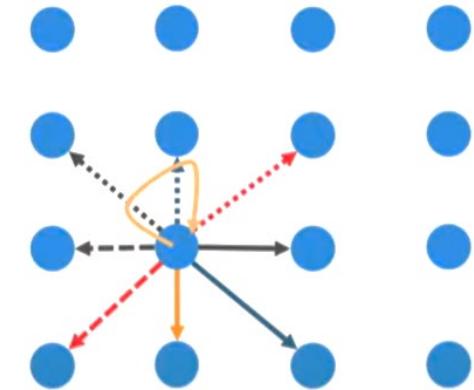
$$\mathbf{h}_v^{(l)} = \sigma \left( \sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|} \right)$$

## ■ Message:

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

Normalized by node degree

(This is a simplified version. Not the optimal form in the GCN paper<sup>[1]</sup>)



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

# 4.1 Graph Convolution Networks

4 GNN Variants

Aggregations can be performed by matrix operations

Let  $\mathbf{H}^{(l)} = \begin{bmatrix} h_1^{(l)} & \dots & h_{|V|}^{(l)} \end{bmatrix}^T$

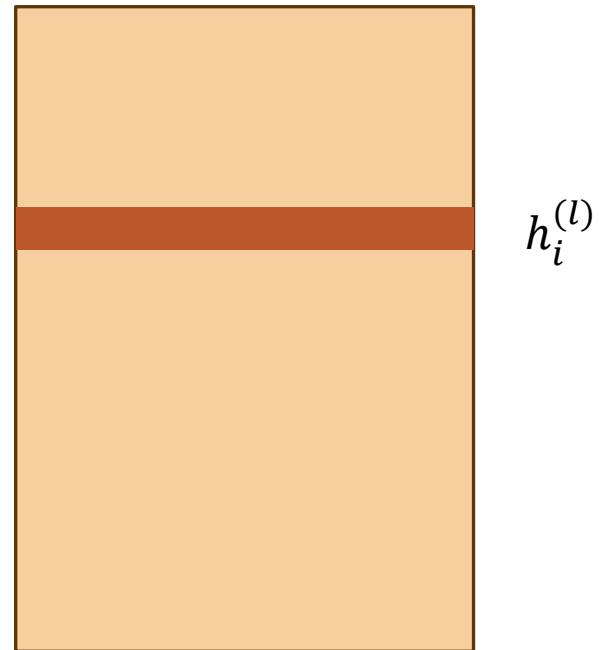
Then  $\sum_{u \in N_v} h_u^{(l)} = \mathbf{A}_{v,:} \mathbf{H}^{(l)}$

Let  $\mathbf{D}$  be diagonal matrix where  $D_{v,v} = \text{Deg}(v) = |N(v)|$

The inverse of  $\mathbf{D}$ :  $\mathbf{D}^{-1}$  is also diagonal:  $D_{v,v}^{-1} = 1/|N(v)|$

$$\sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} \quad \longrightarrow \quad H^{(l)} = \mathbf{D}^{-1} \mathbf{A} \mathbf{H}^{(l)}$$

Matrix of hidden embeddings  $\mathbf{H}^{(l)}$



# 4.1 Graph Convolution Networks

$$\begin{aligned}\mathbf{h}_v^{(l)} &= \sigma \left( \mathbf{W}^{(l)} \sum_{u \in N(v)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|} \right) \\ &= \sigma(\widehat{\mathbf{D}}^{-1} \widehat{\mathbf{A}} \mathbf{H}^{(l-1)} \mathbf{W}^l)\end{aligned}$$

where  $\widehat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ ,  $\mathbf{I}$  is the identity matrix indicating the self-loop of node  $v$ ,  $\widehat{\mathbf{D}}$  is the diagonal node degree matrix

$$\mathbf{h}_v^{(l)} = \sigma(\widehat{\mathbf{D}}^{-\frac{1}{2}} \widehat{\mathbf{A}} \widehat{\mathbf{D}}^{-\frac{1}{2}} \mathbf{W}^l)$$

- To counteract the scale discrepancy and ensure that each neighbor's contribution is considered fairly, normalization is applied. Normalizing  $\mathbf{A}$  such that all rows sum to one gets rid of the issue<sup>[1]</sup>.
- Normalization in GCNs is akin to saying, "Let's not let the quantity of connections overshadow the quality of information each neighbor brings."
- In practice, we use a symmetric normalization  $\mathbf{D}^{-\frac{1}{2}} \widehat{\mathbf{A}} \mathbf{D}^{-\frac{1}{2}}$  to the adjacency matrix. This form is derived from the symmetric Laplacian in spectral graph theory. We will skip the derivation of the formulas here.
- For details, please refer to [2] (the original classic paper)

[1] <https://tkipf.github.io/graph-convolutional-networks/>

[2] Kipf, Thomas N., and Max Welling. "Semi-supervised classification with graph convolutional networks." ICLR 2017 (arXiv 2016).

# Discussion: Does GCN uses mean aggregation?

The aggregation step involves both normalization and summing. Specifically, the normalization factor used is the square root of the degree of the node itself and the degree of its neighbors.

$$\mathbf{H}^{(l+1)} = \sigma(\hat{\mathbf{D}}^{-\frac{1}{2}} \hat{\mathbf{A}} \hat{\mathbf{D}}^{-\frac{1}{2}} \mathbf{W}^{(l)} \mathbf{H}^{(l)})$$

Where:

- $\mathbf{H}^{(l+1)}$  is the node feature matrix after applying layer  $l$
- $\sigma$  is a non-linear activation function like ReLU.
- $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$  is the adjacency matrix of the graph  $\mathbf{G}$  with added self-connections  $I_N$ .
- $\hat{\mathbf{D}}$  is the degree matrix of  $\hat{\mathbf{A}}$ , and  $\hat{\mathbf{D}}^{-\frac{1}{2}}$  is used to normalize.
- $\mathbf{W}^{(l)}$  is the weight matrix for layer  $l$ .

The normalization  $\hat{\mathbf{D}}^{-\frac{1}{2}} \hat{\mathbf{A}} \hat{\mathbf{D}}^{-\frac{1}{2}}$  effectively normalizes the features of each node by a factor related to the degrees of the node itself and its neighbors, ensuring that nodes with a high degree don't have disproportionately large influence on the feature aggregation process. This normalization allows the model to better generalize across different parts of the graph by preventing the scale of feature values from escalating too quickly and reducing the risk of overfitting on graphs with highly variable node degrees.

# Discussion: Why high degree node matters so much?

- Think of the global picture
- Without normalization, this aggregation process is essentially a sum. The more neighbors a node has, the larger the sum, hence, the higher the aggregated feature values.
- Escalate the feature values for nodes with high degrees -> contribute their features to all their neighbors -> potentially dominating the feature space.
- By normalizing the aggregation based on the degrees of both the node and its neighbors, it ensures that no single node's features, regardless of how many connections (degree) it has, can unduly influence the aggregated feature vector of a node.

# 4.1 Graph Convolution Networks

4 GNN Variants

- Key drawbacks of GCN:
  - 1. During the aggregation process, all neighbors are equally important to node  $v$ , so the edge weights are fixed, lacking flexibility.
  - 2. Scalability is poor because it involves graph convolution fusion over the entire graph. Gradient updates are performed on the entire graph, which becomes too slow and unsuitable when the graph is relatively large.
  - 3. As the number of layers increases, the results tend to become overly smooth, with the feature representations of individual nodes becoming highly similar.

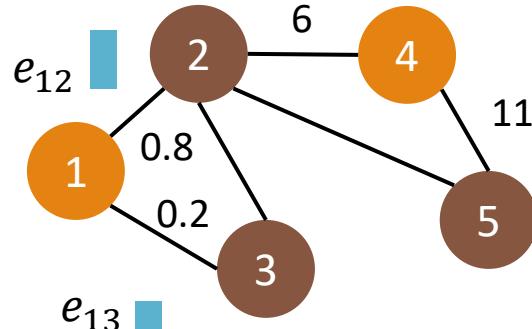
# 4.2 Graph Attention Networks

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

- In GCN,  $\alpha_{vu} = \frac{1}{|N(v)|}$  is **weighting factor** for every neighboring node  $u$ 's message to node  $v$  (**importance**)
- Not all nodes' neighbors are equally important in GAT idea
  - Attention is inspired by the popular NLP attention mechanism -> **learning** message importance
  - $\alpha_{vu}$  is called **Attention Weights** in GAT
  - The attention  $\alpha_{vu}$  focuses on the important parts of the input data and ignores the rest.
  - Which part of the data is more important depends on the context and is **learned** during training.
- Attention mechanism  $a$  computes **attention coefficients**  $e_{vu}$  across pairs of nodes  $u, v$ 
  - Take it as a weight for each edge of the neighbors that tells us how much attention we should pay to that specific node.
  - $e_{vu} = a(\mathbf{W}\mathbf{h}_v, \mathbf{W}\mathbf{h}_u)$
- In fact, NLP on text data and GNN on graph data is similar. We can see the sentence as a graph, the words as nodes, and the word vectors as node feature vectors.

# 4.2 Attention Weights

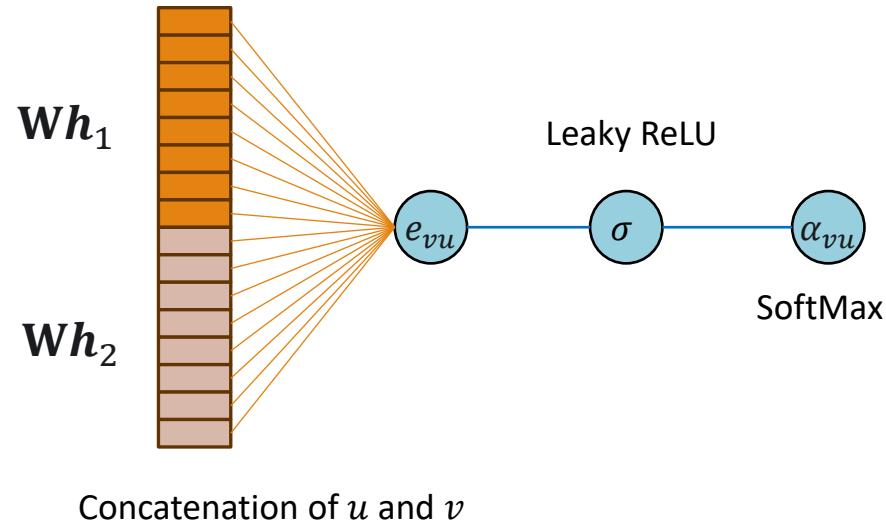
4 GNN Variants



$$e_{vu} = a(\mathbf{Wh}_v, \mathbf{Wh}_u)$$

$$\alpha_{vu} = \text{softmax}_v(e_{vu}) = \frac{\text{ex}}{\sum_{k \in N(v)}}$$

- We call the attention method “mask nodes that are not 1-neighbor to the target node”

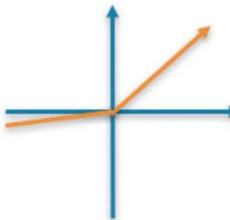


$$\vec{a} \in \mathbb{R}^{2F'}$$

$$\frac{\mathbf{Wh}_u)}{\mathbf{i}_v, \mathbf{Wh}_k)}$$

$$\text{leakyrelu}(z) = \begin{cases} 0.01z & \text{for } z < 0 \\ z & \text{for } z \geq 0 \end{cases}$$

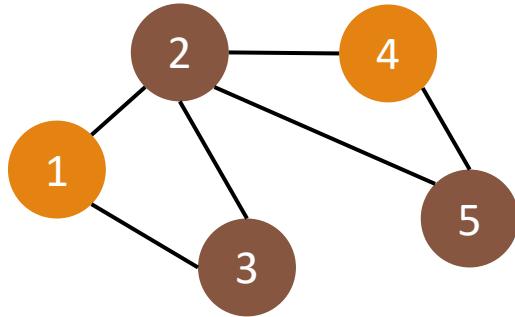
Range: -infinity to infinity



$$\alpha_{vu} = \frac{\exp(\text{LeakyReLU}(\vec{a}^T [\mathbf{Wh}_v || \mathbf{Wh}_u]))}{\sum_{k \in N(v)} \exp(\text{LeakyReLU}(\vec{a}^T [\mathbf{Wh}_v || \mathbf{Wh}_k]))}$$

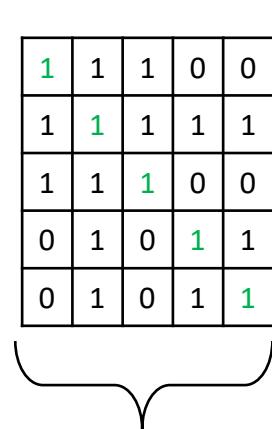
# 4.2 Graph Attention Networks

4 GNN Variants

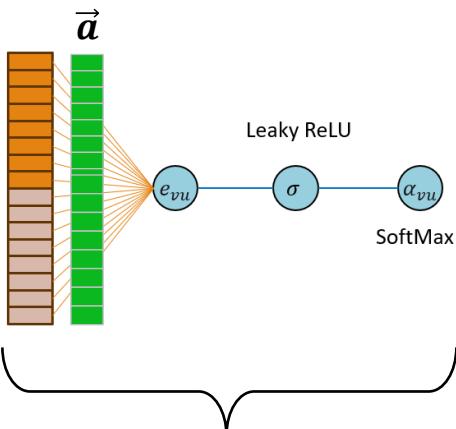


$$\mathbf{h}'_v = \sigma \left( \sum_{u \in N(v)} \alpha_{vu} \mathbf{W} * \mathbf{h}_u \right)$$

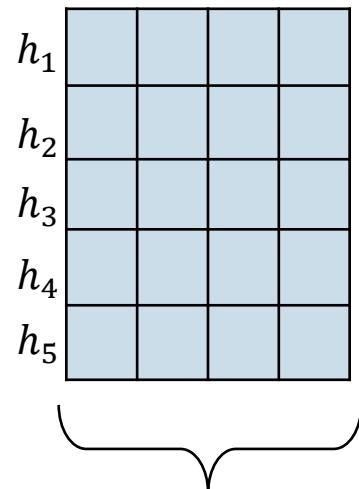
1	1	1	0	0
1	1	1	1	1
1	1	1	0	0
0	1	0	1	1
0	1	0	1	1



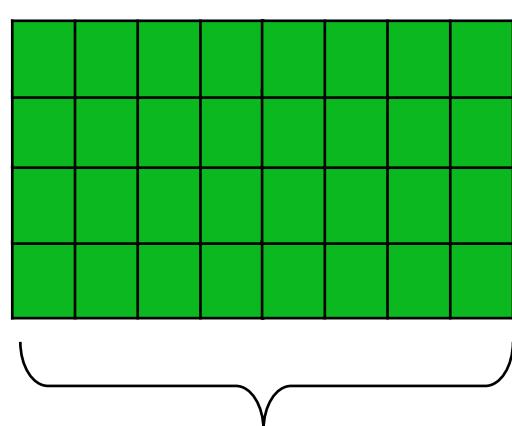
Adjacency  
matrix [5, 5]



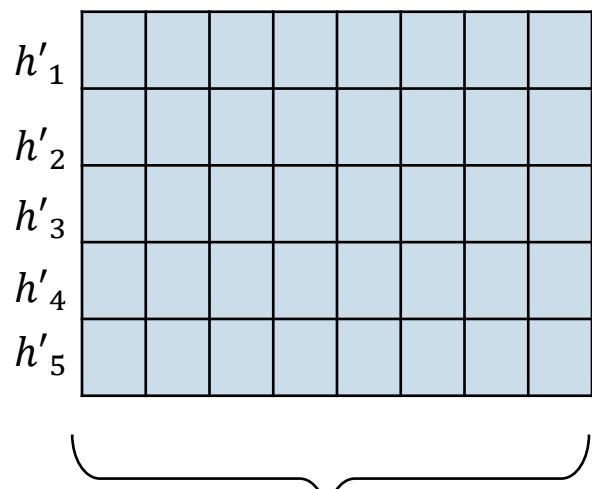
Masked attention



Features per node  
[5, 4]



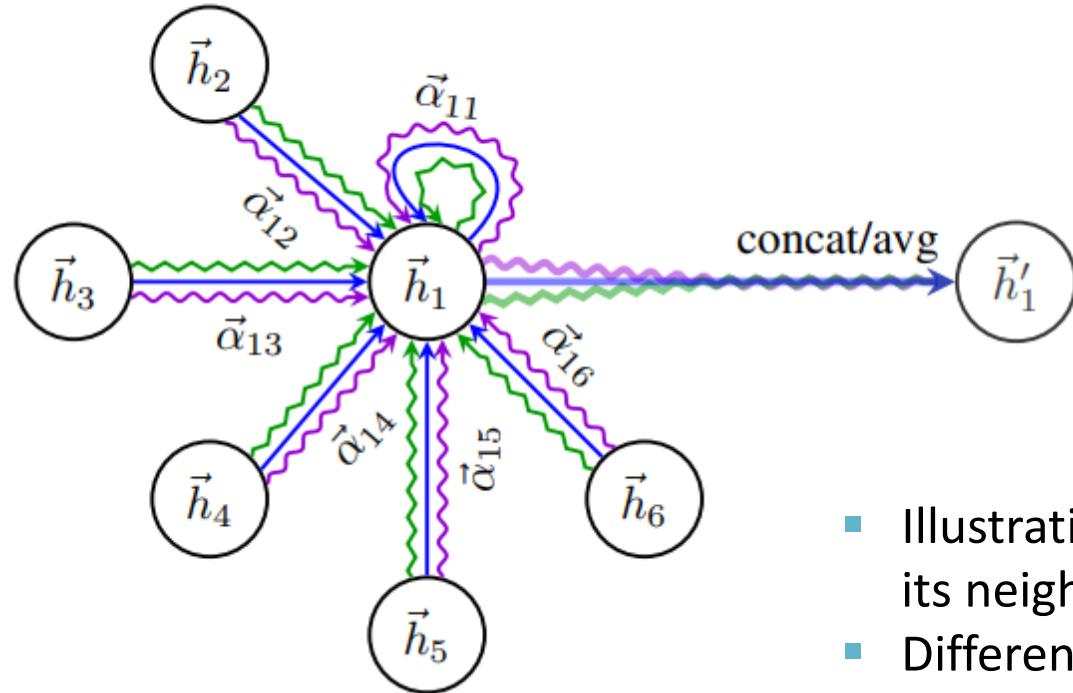
Learnable weight  
matrix [4, 8]



Embedding per node  
[5, 8]

# 4.2 Graph Attention Networks

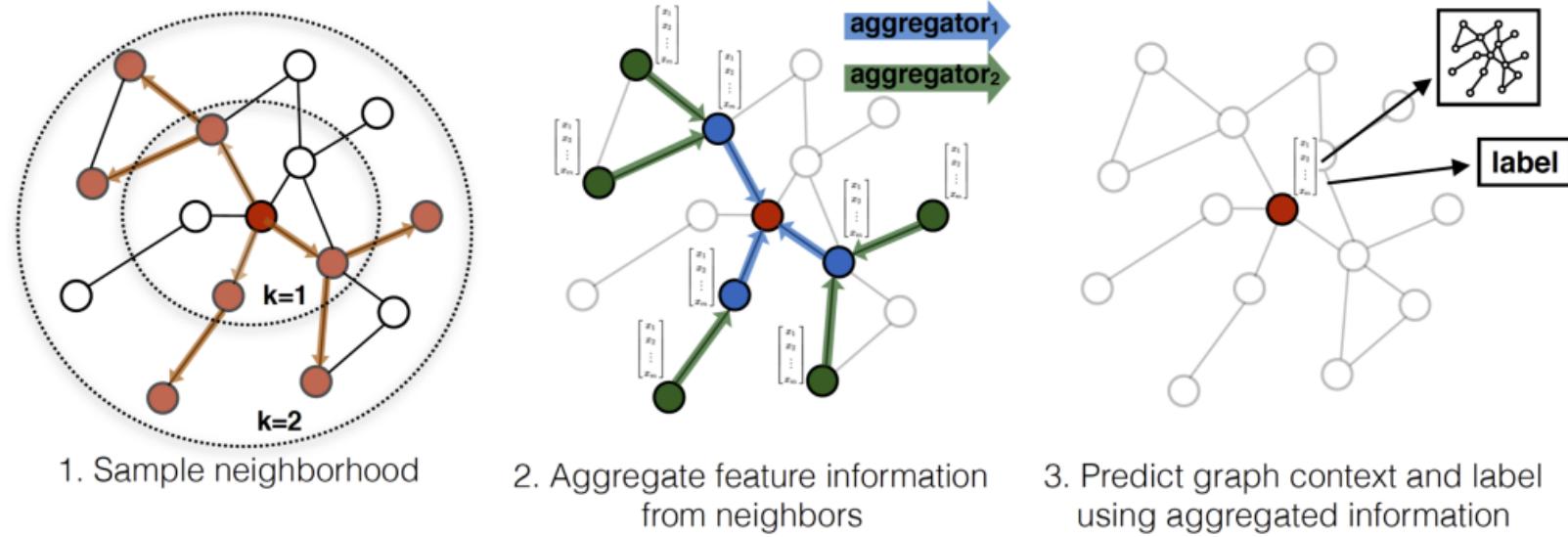
4 GNN Variants



- Illustration of multi-head attention (with  $K = 3$  heads) by node 1 on its neighborhood.
- Different arrow styles and colors denote independent attention computations.
- The aggregated features from each head are concatenated or averaged to obtain the final  $\vec{h}'_1$

# 4.3 GraphSAGE

- Short for **Graph Sample and AggreGatE** [1]
  - **Scalability:** By sampling and aggregating information from a node's local neighborhood, GraphSAGE can generate embeddings without processing the **entire** graph. Suitable for large-scale graphs.
  - **Inductiveness:** GraphSAGE's learned function can generate embeddings for new nodes without retraining, as long as it has access to the node's attributes and its neighborhood structure. Suitable for dynamic graphs where new nodes and edges are continuously added.
  - **Four Steps**
    - Node Sampling
    - Feature Aggregation
    - Update
    - Normalization



# 4.3 GraphSAGE

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

- Concatenate the self-node embeddings with the aggregated neighbor embeddings to increase expressive power on top of GCN<sup>[1]</sup>.
- Then in **most** scenarios, we apply  $\ell_2$  normalization to  $\mathbf{h}_v^{(l)}$  at every layer (the Euclidean length of the embedding vector  $\left\| \mathbf{h}_v^{(l)} \right\|_2$  will always be equal to 1), which improves great performance.
- Inductive vs. Transductive Learning
  - Transductive learning models are trained on a specific graph and are designed to make predictions for nodes or edges that are part of the same graph seen during training. These models do not generalize to unseen nodes or graphs.
  - Inductive learning models are trained in a way that allows them to generalize to unseen nodes or entirely new graphs. They learn a function that can be applied to nodes not seen during training, based on node features and local graph structure.

# 4.3 GraphSAGE Aggregation

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

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

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

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

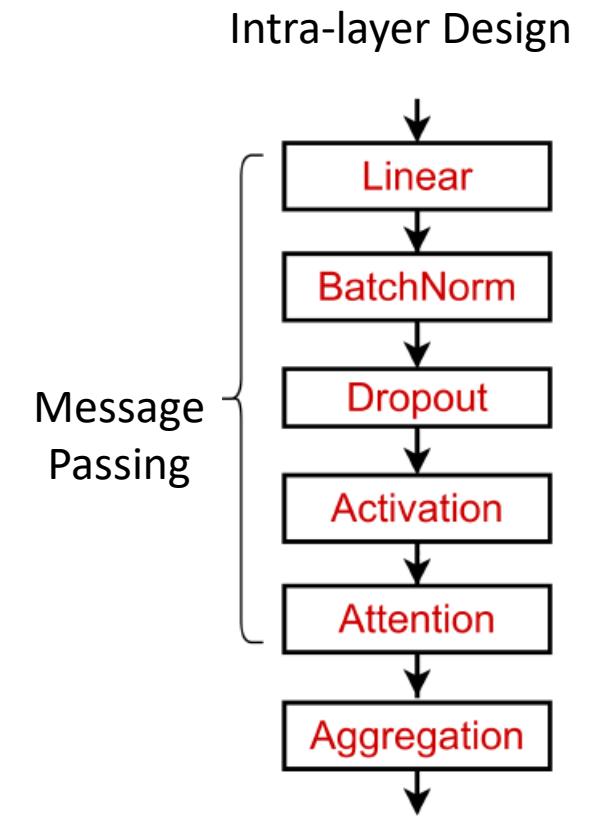
- **LSTM:** Apply LSTM to the reshuffled neighbors recursively

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

- When we apply sequence model to the messages from the neighbours, we have to bear in mind that it is not order invariant
- During training, we keep permuting the orderings so that we teach the sequence model to ignore the ordering of the received messages

# 5.1 GNN Layer Design

- In practice, we introduce modern deep learning modules to improve the performance<sup>[1]</sup>.
- Batch Normalization<sup>[2]</sup>
  - Re-center the node embeddings into zero mean
  - Re-scale the variance into unit variance
  - Stabilize neural network training process
- Dropout
  - Prevent overfitting
- Attention or Gating
  - Control the importance of a message
- And more ...

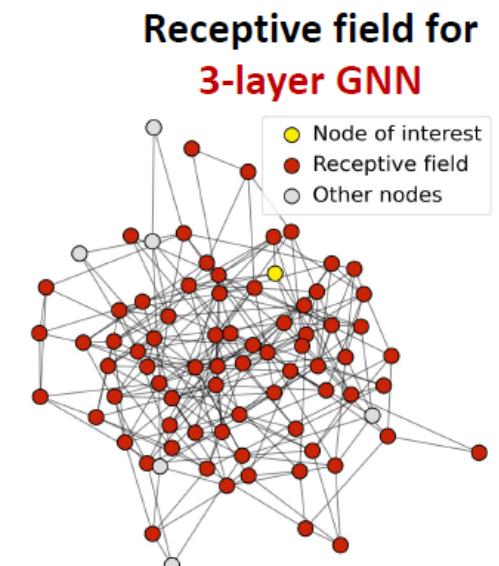
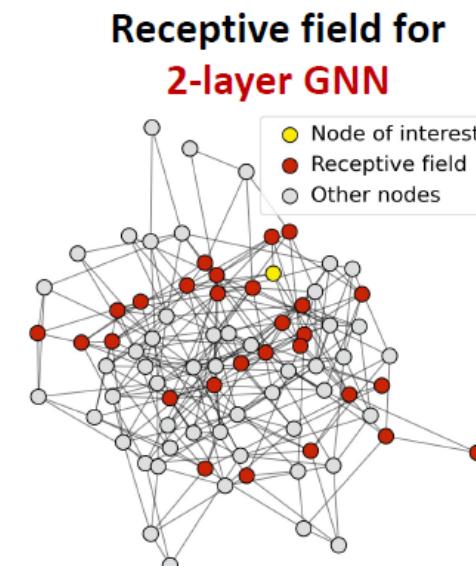
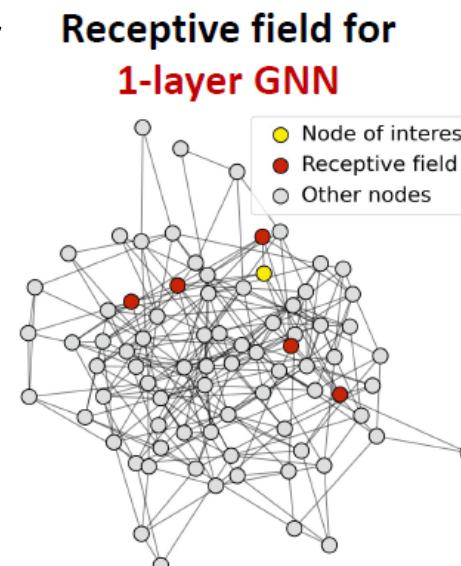


[1] J. You, R. Ying, J. Leskovec. Design Space for Graph Neural Networks, NeurIPS 2020

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

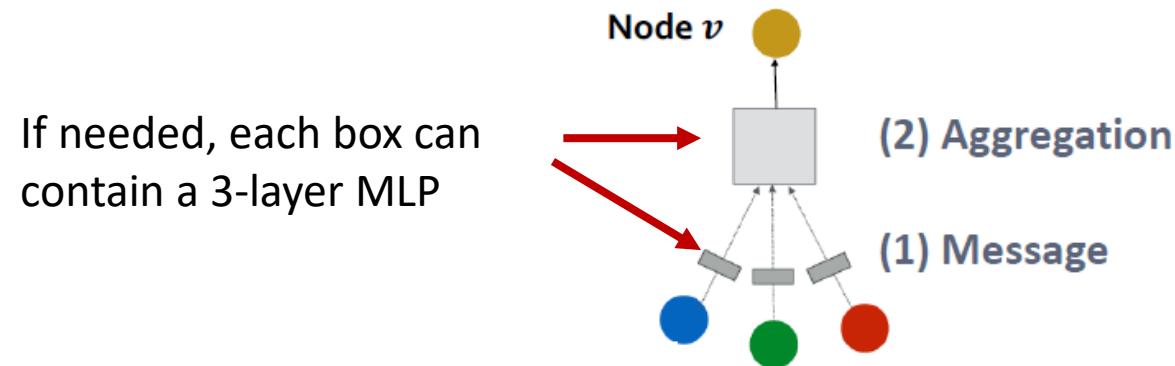
# 5.1 GNN Layer Design

- GNN suffers from over-smoothing problem when stacking many GNN layers
  - All the node embeddings converge to the same value
- 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
  - The shared neighbours quickly grows when layers increase



# 5.1 GNN Layer Design

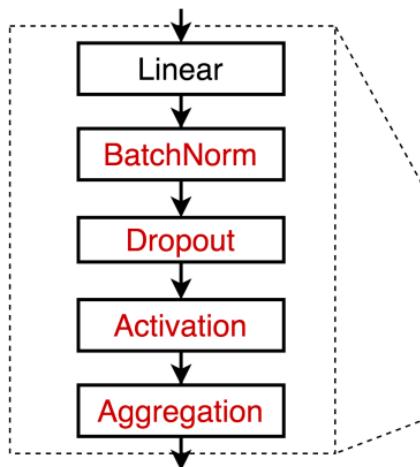
- We have to make a shallow but a powerful GNN 😞
- **Solution:** Increase the expressive power within each GNN layer
  - We can make message passing / aggregation process a deep neural network!



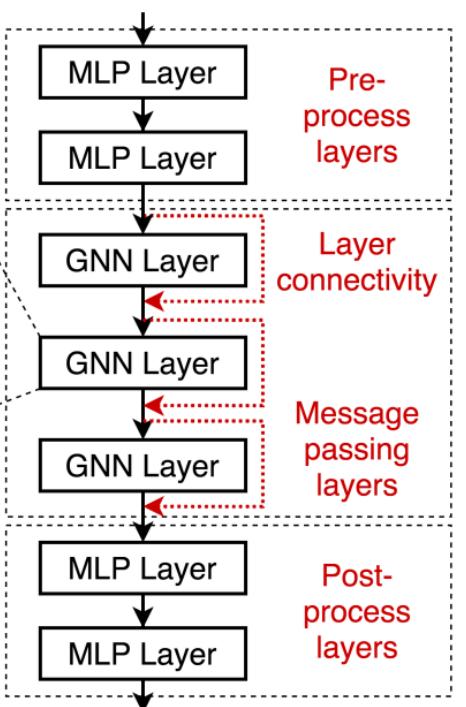
# 5.1 GNN Layer Design

- We have to make a shallow but a powerful GNN 😞
- Solution 2:** Add layers that do not pass messages
  - Not all layers in a GNN have to be GNN layers
  - We can have MLP layers (applied to each node) before and after GNN layers

Intra-layer Design: 4 dims



Inter-layer Design: 4 dims



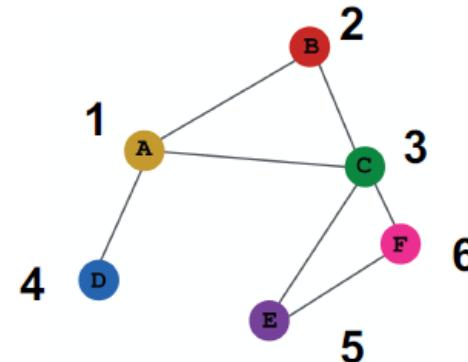
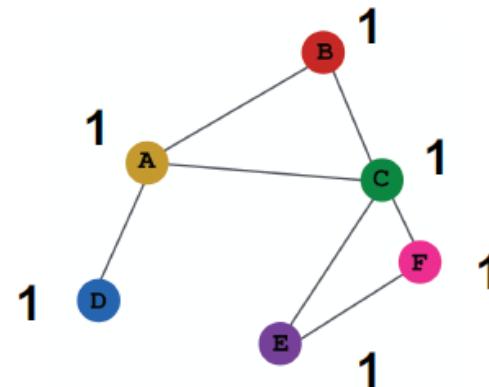
Learning Configuration: 4 dims

Batch size  
Learning rate  
Optimizer  
Training epochs

- Pre-processing layers
  - Important when encoding node features is necessary
  - E.g., when node represent images / text
- Post-processing layers:
  - Important when reasoning / transformation over node embeddings are needed
  - E.g., graph classification, knowledge graphs
- Add skip connections
  - Like residual network, adding **shortcuts** in GNN can increase the impact of earlier layers

# 5.2 Graph Feature Manipulation

- Sometimes the input graph does not have node features
  - Quite common scenario when we only have adjacency matrix
- **Solution 1:** Assign constant values to nodes
- **Solution 2:** Assign unique IDs to nodes (**one-hot vectors**)

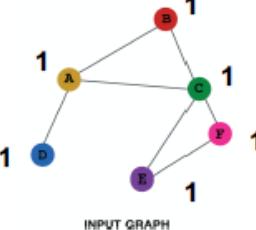
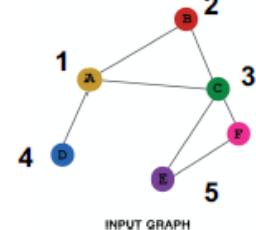


One-hot vector for node ID=5

ID = 5  
↓  
[0, 0, 0, 0, 1, 0]  
Total number of IDs = 6

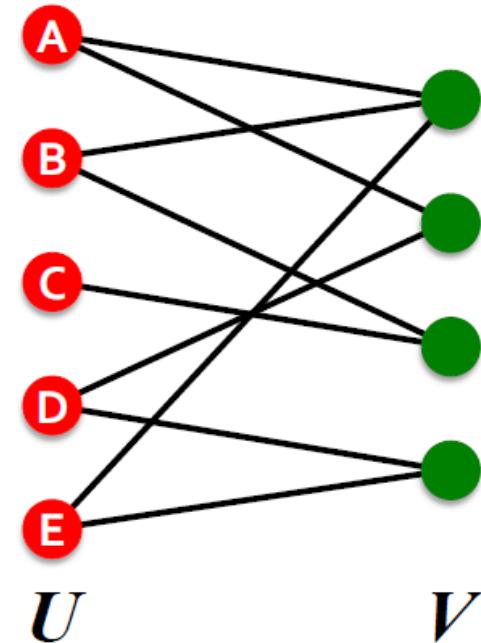
# 5.2 Graph Feature Manipulation

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

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

# Bipartite Graph

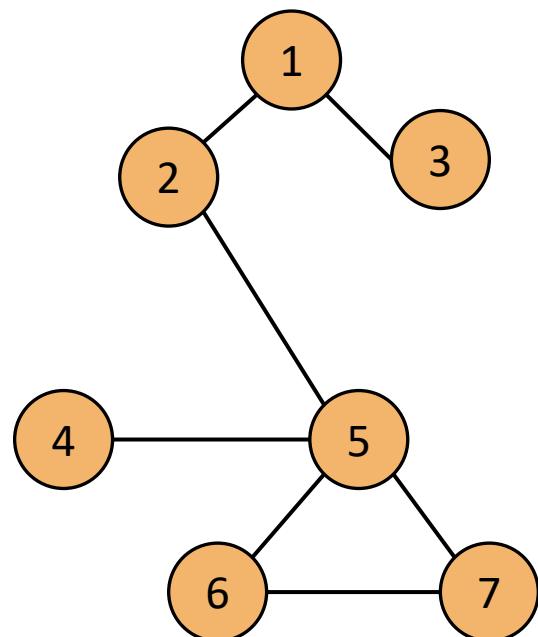
- **Bipartite graph** is a graph whose nodes can be divided into two disjoint sets  $U$  and  $V$  such that every link connects a node in  $U$  to one in  $V$ ; that is,  $U$  and  $V$  are **independent sets**
- **Examples:**
  - Authors-to-Papers (they authored)
  - Actors-to-Movies (they appeared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



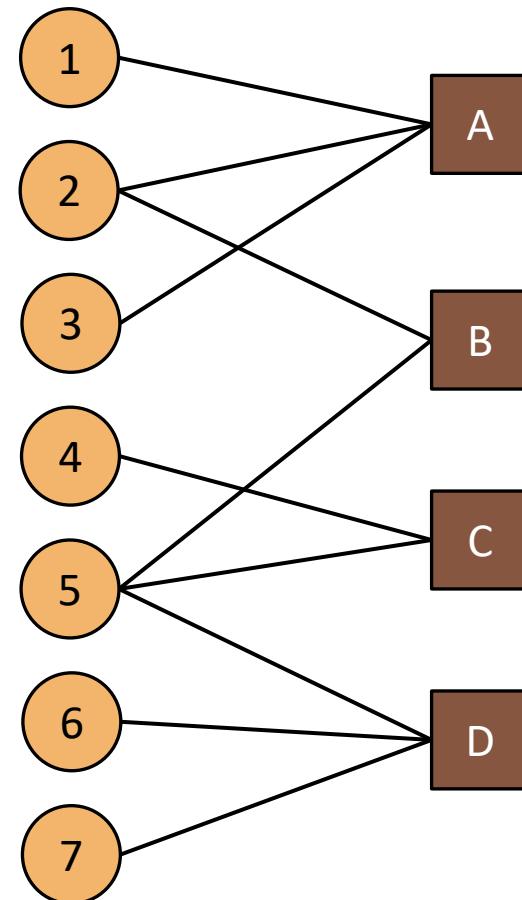
# Projected Bipartite Graph

5 GNN Layer in Practice

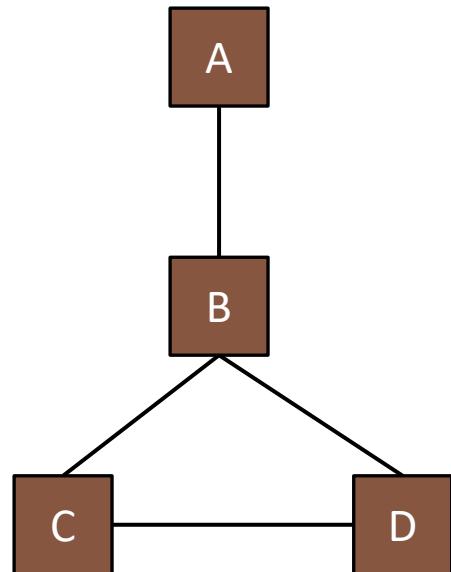
**Projection U**



**U**      **V**

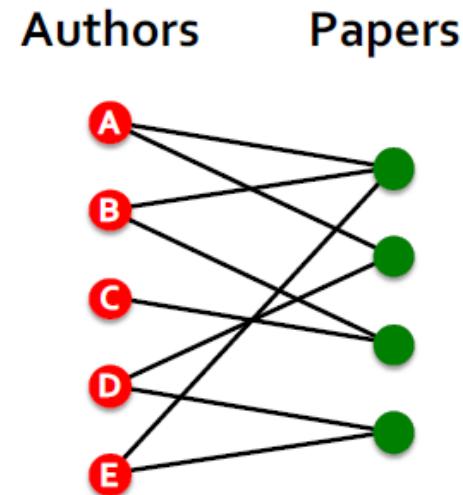


**Projection V**



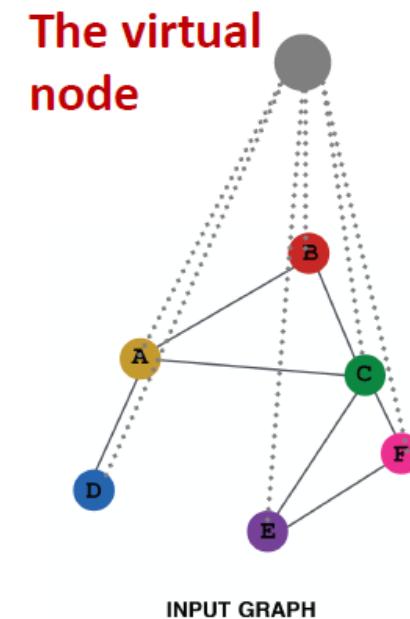
# 5.3 Graph Structure Manipulation

- Augment sparse graphs
- **Solution 1:** Add virtual edges
  - **Common approach:** Connect 2-hop neighbors via virtual edges
  - **Intuition:** Instead of using adj. matrix  $A$  for GNN computation, use  $A + A^2$
  - **Use cases:** Bipartite graphs
    - Author-to-papers (they authored)
    - 2-hop virtual edges make an author-author collaboration graph



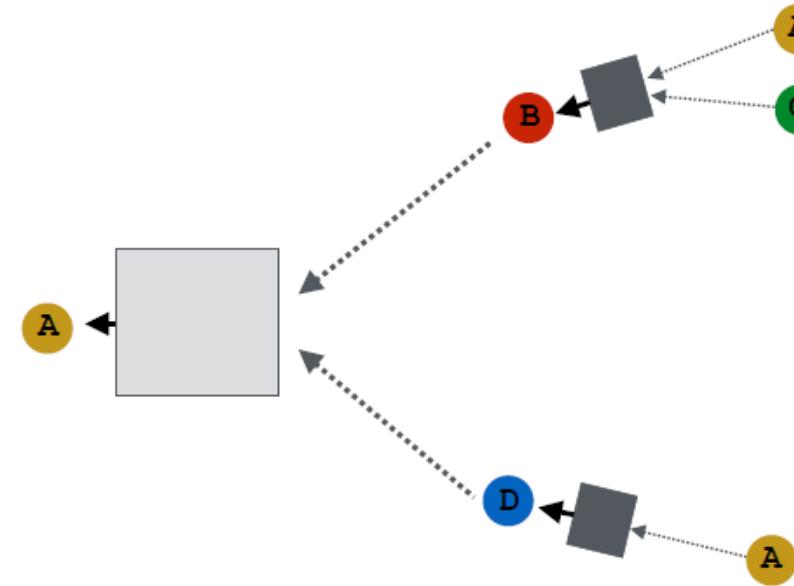
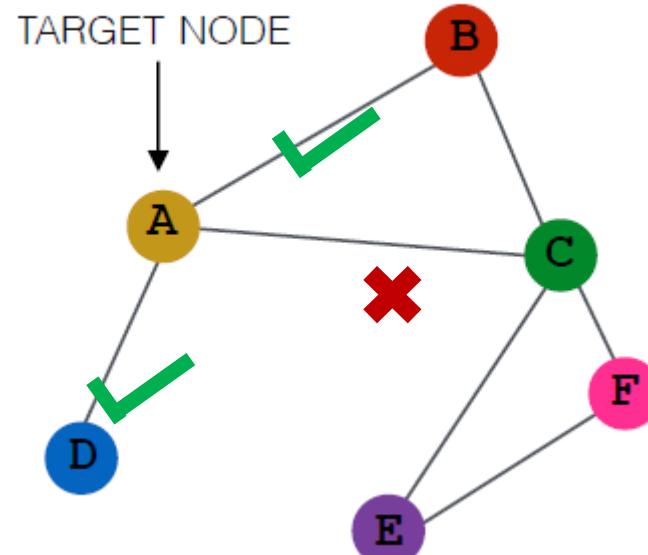
# 5.3 Graph Structure Manipulation

- Augment sparse graphs
- **Solution 2:** Add virtual nodes
  - The virtual node will connect to all the nodes in the graph
    - Suppose in a sparse graph, two nodes have shortest path distance of 10
    - After adding the virtual node, **all the nodes will have a distance of 2**
      - Node A – Virtual node – Node B
  - Greatly improves **message passing efficiency** in sparse graphs



# 5.3 Graph Structure Manipulation

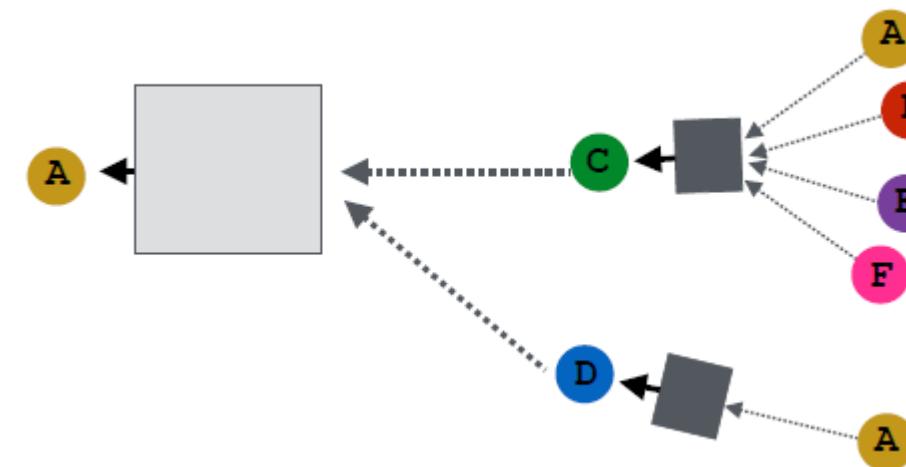
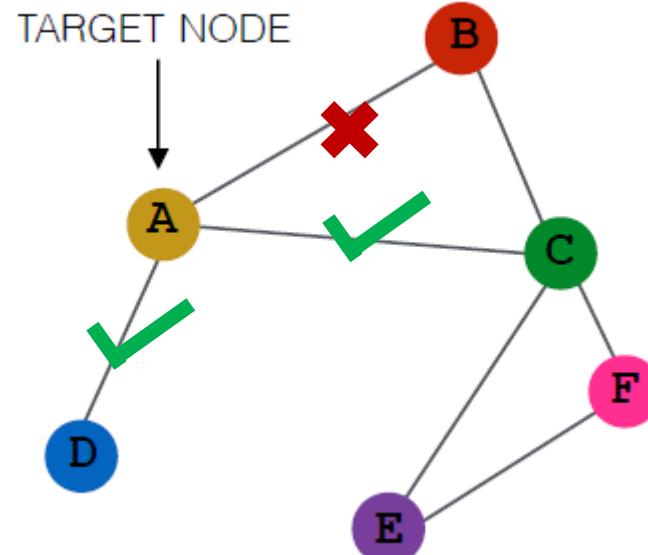
- For dense / large graphs, high-degree nodes
- Solution:** not all the neighbors are used for message passing<sup>[1]</sup>
  - Randomly determine a node's neighborhood for message passing
  - For example, we can randomly choose 2 neighbors to pass messages
    - Only nodes *B* and *D* will pass message to *A*



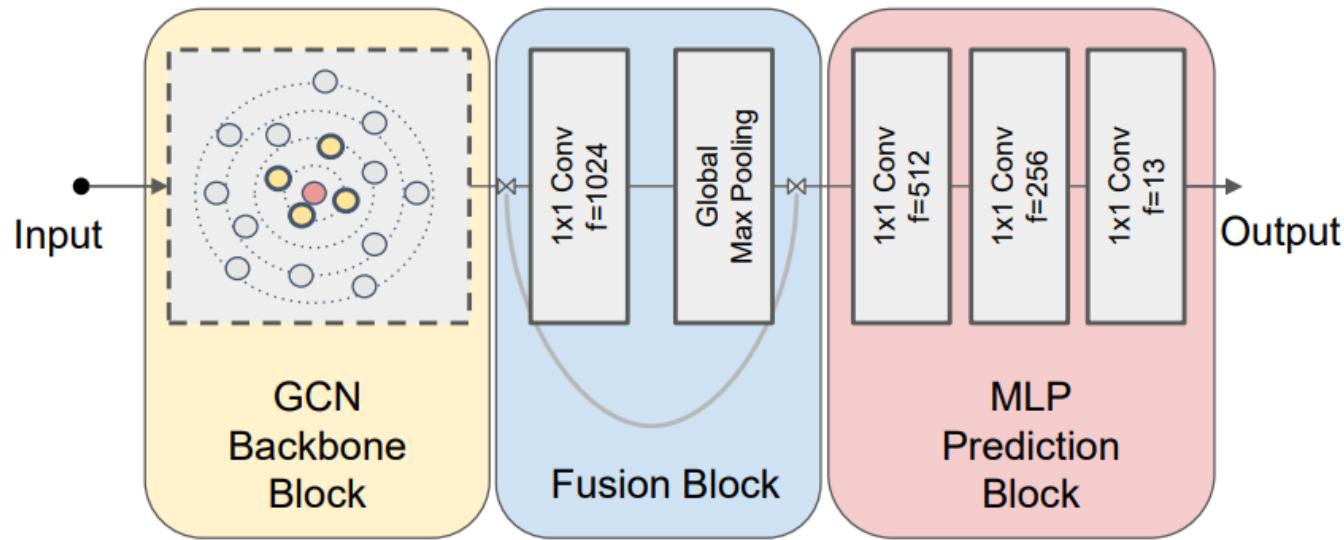
[1] Ying et al. Graph Convolutional Neural Networks for Web-Scale Recommender Systems, KDD 2018

# 5.3 Graph Structure Manipulation

- For dense / large graphs, high-degree nodes
- Solution:** not all the neighbors are used for message passing
  - Randomly determine a node's neighborhood for message passing
  - Next time when computing embeddings, we sample different neighbors
  - It greatly reduce computational cost while still maintaining accuracy

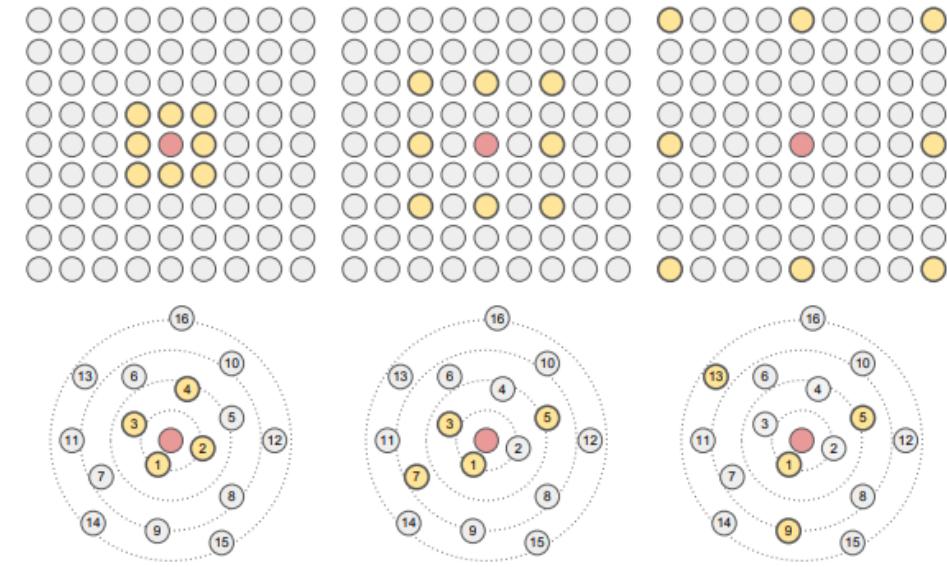


# 5.4 GCN for downstream tasks



3 blocks for point cloud semantic segmentation<sup>[1]</sup>:

- GCN backbone block (feature transformation of input point cloud)
- Fusion block (global feature generation and fusion)
- MLP prediction block (point-wise label prediction)

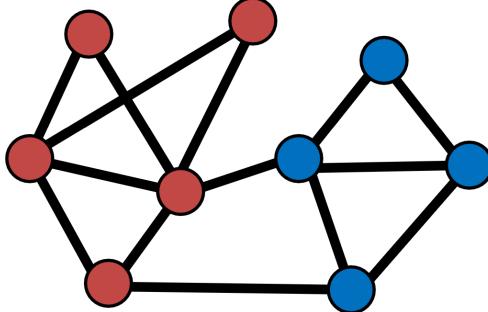


**Figure 3. Dilated Convolution in GCNs.** Visualization of dilated convolution on a structured graph arranged in a grid (e.g. 2D image) and on a general structured graph. (top) 2D convolution with kernel size 3 and dilation rate 1, 2, 4 (left to right). (bottom) Dynamic graph convolution with dilation rate 1, 2, 4 (left to right).

[1] Li, Guohao, et al. "DeepGCNs: Can GCNs Go as Deep as CNN?." Proceedings of the IEEE/CVF international conference on computer vision. 2019

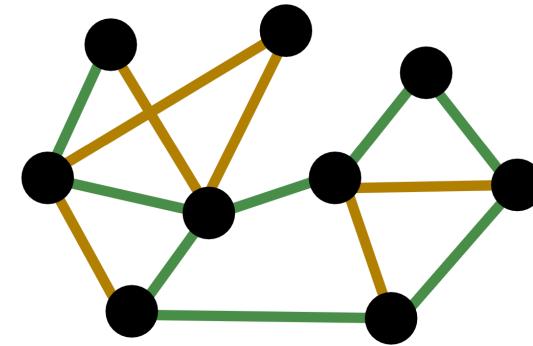
# 6.1 Heterogeneous Graphs

- Graphs with multiple nodes or edge types



**2 types of nodes:**

- Node type A: Paper nodes
- Node type B: Author nodes

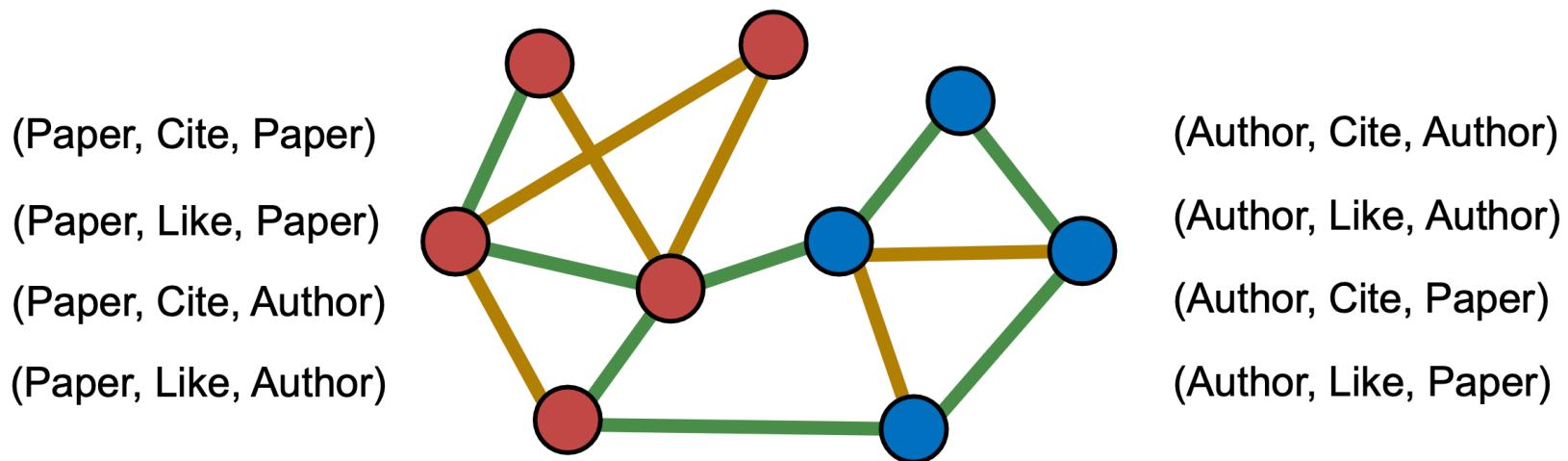


**2 types of edges:**

- Edge type A: Cite
- Edge type B: Like

# 6.1 Heterogeneous Graphs

- 8 possible relation types with 2 node types and 2 edge types
  - We use **relation types** to describe an edge, instead of edge types
  - (**head**, **relation**, **tail**)

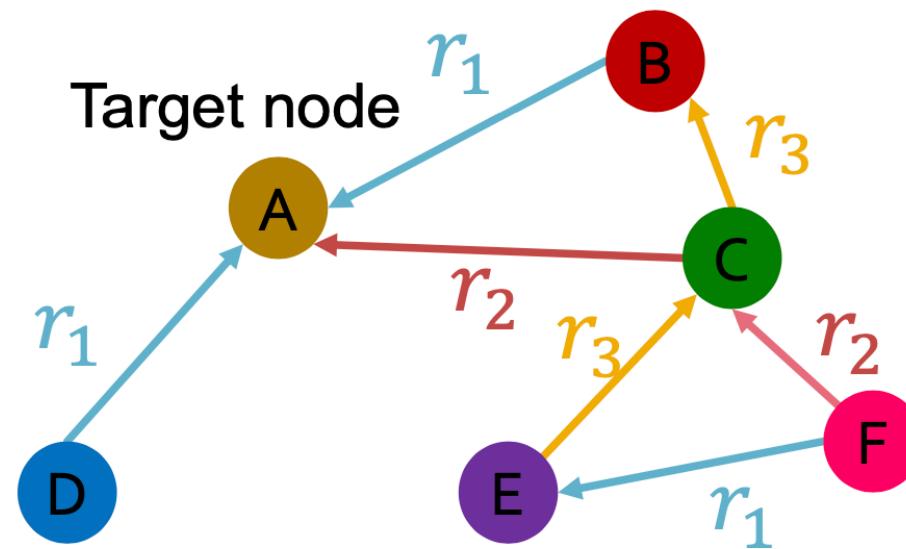


# 6.1 Heterogeneous Graphs

- A heterogeneous graph is defined as  $G = (V, E, \tau, \phi)$ 
  - Node type for node  $v$ :  $\tau(v)$  where  $v \in V$
  - Edge type for edge  $(u, v)$ :  $\phi(u, v)$  where  $(u, v) \in E$
  - Relation type for edge is a tuple:  $r(u, v) = (\tau(u), \phi(u, v), \tau(v))$
- When do we need a heterogeneous graph?
  - Case 1: Different node/edge types have different feature shapes
  - Case 2: We know different relation types represent different interactions
    - (English, translate, French) and (English, translate, Chinese) require different models

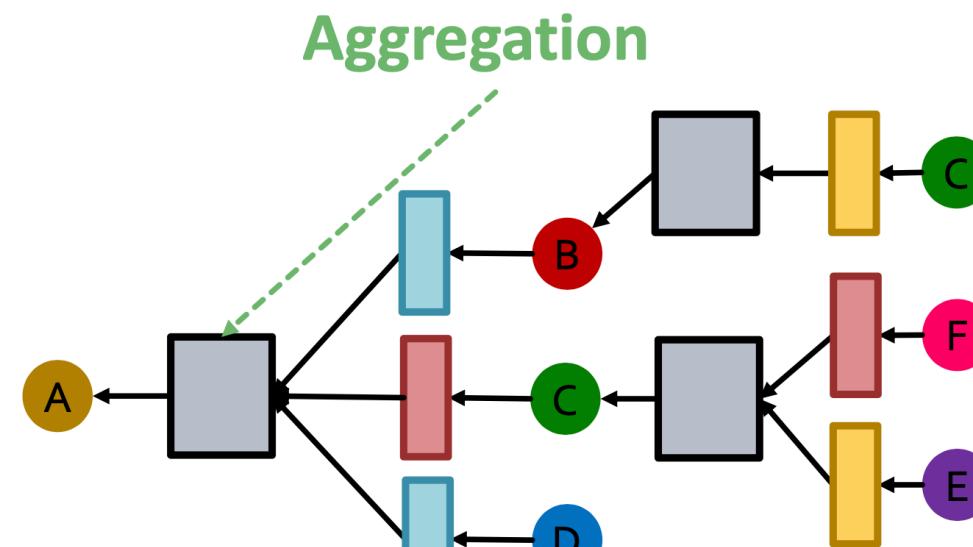
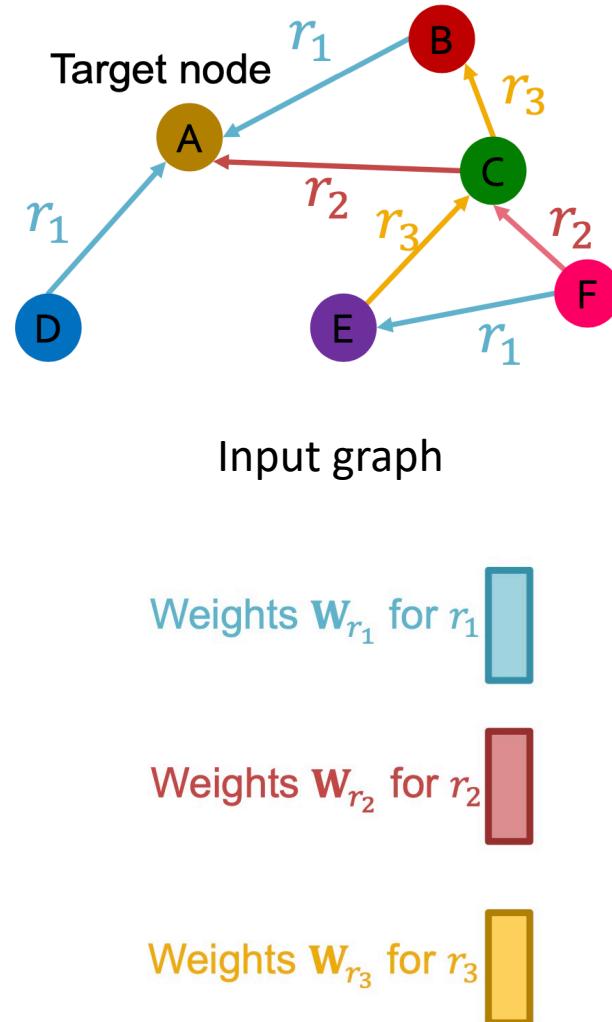
# 6.1 Heterogeneous Graphs

- We try to simplify a heterogeneous graph into a homogeneous graph
  - Extend GCN to **Relational GCN** to handle such issues
  - Use different neural network weights for different relation types



# 6.1 Heterogeneous Graphs

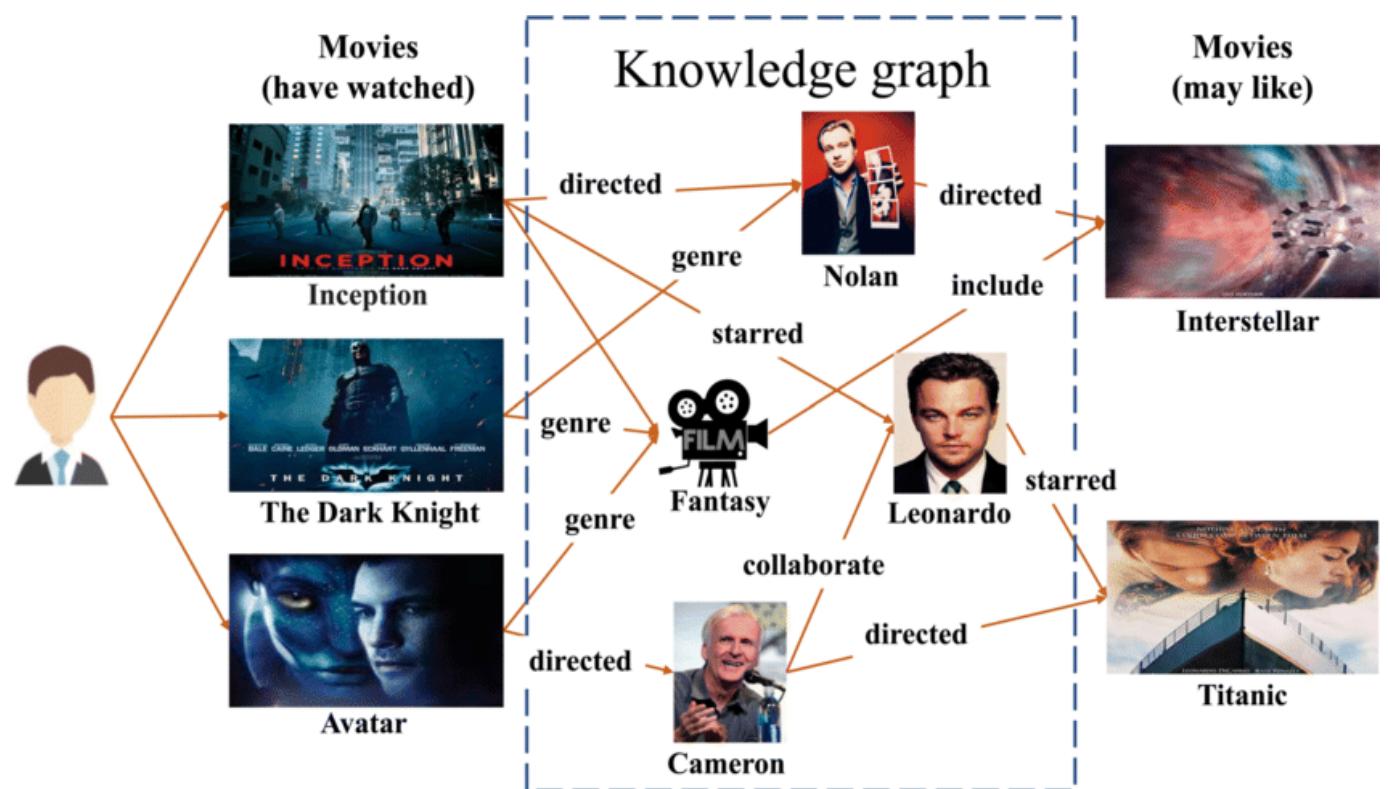
- When do we need a heterogeneous graph?



Neural networks

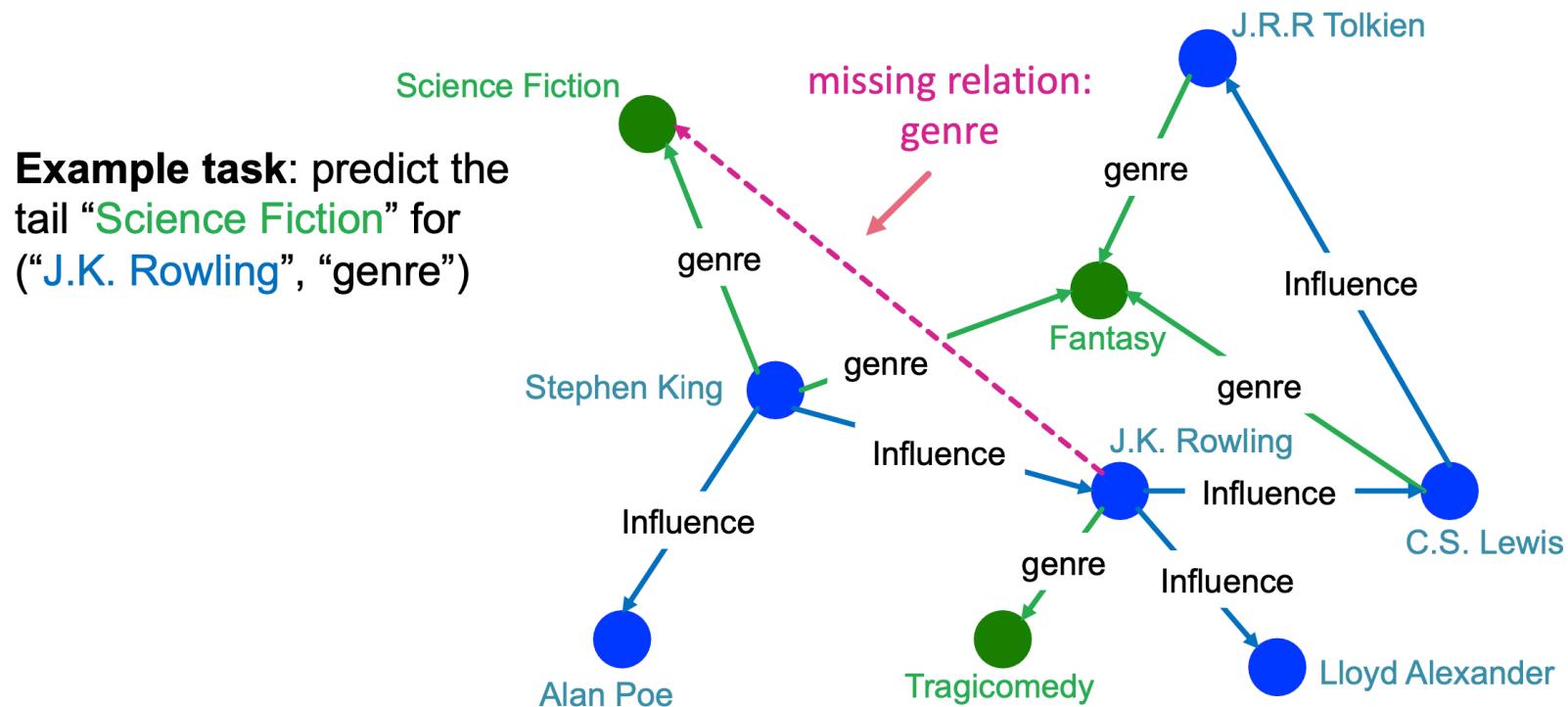
# 6.2 Knowledge Graphs

- Knowledge in a graph is a heterogeneous graph:
  - Capture entities, types and relationships
  - Nodes are **entities**
  - Nodes are labelled with the **types**
  - Edges between two nodes capture **relationships**



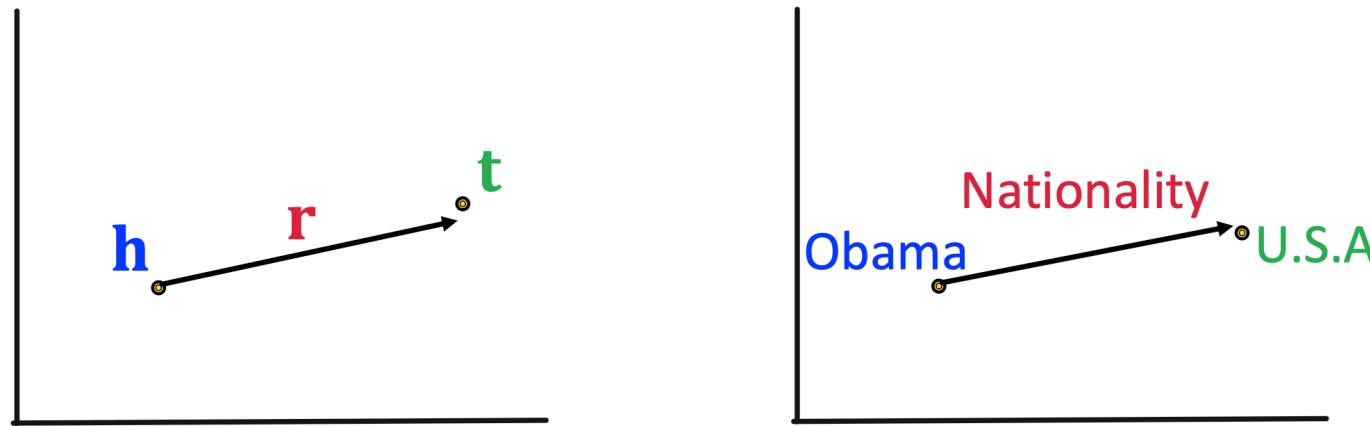
# 6.2 Knowledge Graphs

- Challenges
  - Massive: Millions of nodes and edges (like Wikidata)
  - Incomplete: Many true edges are missing
- KG Completion Task
  - Given an enormous KG, with a given (head, relation), we predict missing tails.



## 6.3 TransE

- Edges in KG are represented as **triples**  $(h, r, t)$  – (head, relation, tail)
- Key Idea
  - Model entities and relations in embedding space  $\mathbb{R}^d$  (using shallow embedding)
  - Given a triple  $(h, r, t)$ , the goal is that the embedding of  $(h, r)$  should be close to the embedding of  $t$
- Translation – **TransE**<sup>[1]</sup>
  - $h + r \approx t$  if the given link exists else  $h + r \neq t$
  - **Entity scoring function:**  $f(h, t) = -\|h + r - t\|$



# 6.4 Relation Patterns

## ■ Symmetric (Antisymmetric) Relations:

$$r(h, t) \Rightarrow r(t, h) \quad (r(h, t) \Rightarrow \neg r(t, h)) \quad \forall h, t$$

### ■ Example:

- Symmetric: Family, Roommate
- Antisymmetric: Hypernym (a word with a broader meaning: poodle vs. dog)

## ■ Inverse Relations:

$$r_2(h, t) \Rightarrow r_1(t, h)$$

### ■ Example : (Advisor, Advisee)

## ■ Composition (Transitive) Relations:

$$r_1(x, y) \wedge r_2(y, z) \Rightarrow r_3(x, z) \quad \forall x, y, z$$

### ■ Example: My mother's husband is my father.

## ■ 1-to-N relations:

$r(h, t_1), r(h, t_2), \dots, r(h, t_n)$  are all True.

### ■ Example: $r$ is “StudentsOf”

# 6.4 Relation Patterns

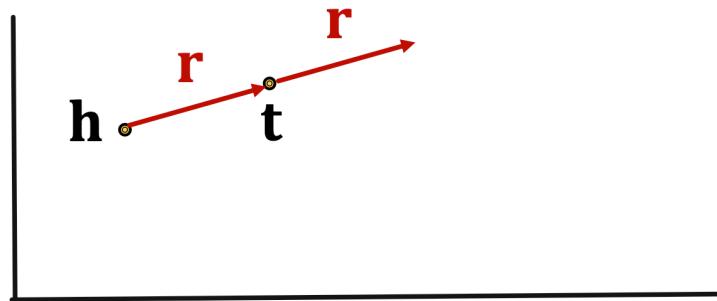
## ■ Antisymmetric Relations:

$$r(h, t) \Rightarrow \neg r(t, h) \quad \forall h, t$$

- Example: Hypernym (a word with a broader meaning: poodle vs. dog)

## ■ TransE can model antisymmetric relations ✓

- $\mathbf{h} + \mathbf{r} = \mathbf{t}$ , but  $\mathbf{t} + \mathbf{r} \neq \mathbf{h}$



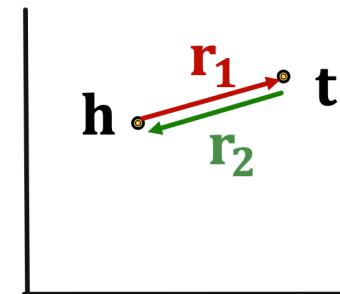
## ■ Inverse Relations:

$$\mathbf{r}_2(h, t) \Rightarrow \mathbf{r}_1(t, h)$$

- Example : (Advisor, Advisee)

## ■ TransE can model inverse relations ✓

- $\mathbf{h} + \mathbf{r}_2 = \mathbf{t}$ , we can set  $\mathbf{r}_1 = -\mathbf{r}_2$



# 6.4 Relation Patterns

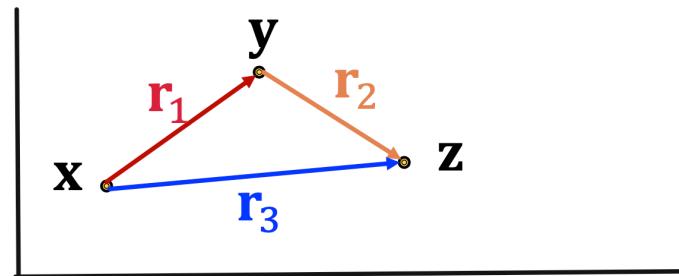
## ■ Composition (Transitive) Relations:

$$r_1(x, y) \wedge r_2(y, z) \Rightarrow r_3(x, z) \quad \forall x, y, z$$

- Example: My mother's husband is my father.

- TransE can model composition relations ✓

$$\mathbf{r}_3 = \mathbf{r}_1 + \mathbf{r}_2$$

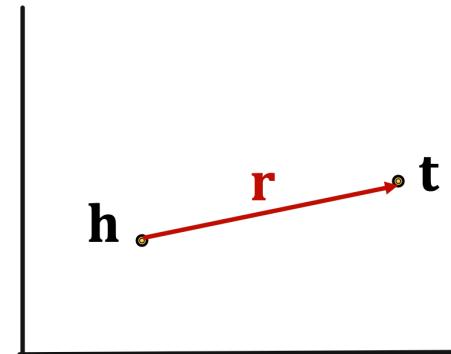


## ■ Symmetric Relations:

$$r(h, t) \Rightarrow r(t, h) \quad \forall h, t$$

- Example: Family, Roommate

- TransE cannot model symmetric relations ✗  
only if  $\mathbf{r} = 0$ ,  $\mathbf{h} = \mathbf{t}$



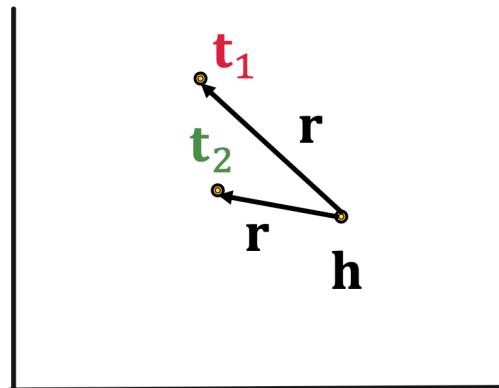
For all  $h, t$  that satisfy  $r(h, t)$ ,  $r(t, h)$  is also True, which means  $\|\mathbf{h} + \mathbf{r} - \mathbf{t}\| = 0$  and  $\|\mathbf{t} + \mathbf{r} - \mathbf{h}\| = 0$ . Then  $\mathbf{r} = 0$  and  $\mathbf{h} = \mathbf{t}$ , however  $h$  and  $t$  are two different entities and should be mapped to different locations.

# 6.4 Relation Patterns

## ■ 1-to-N Relations:

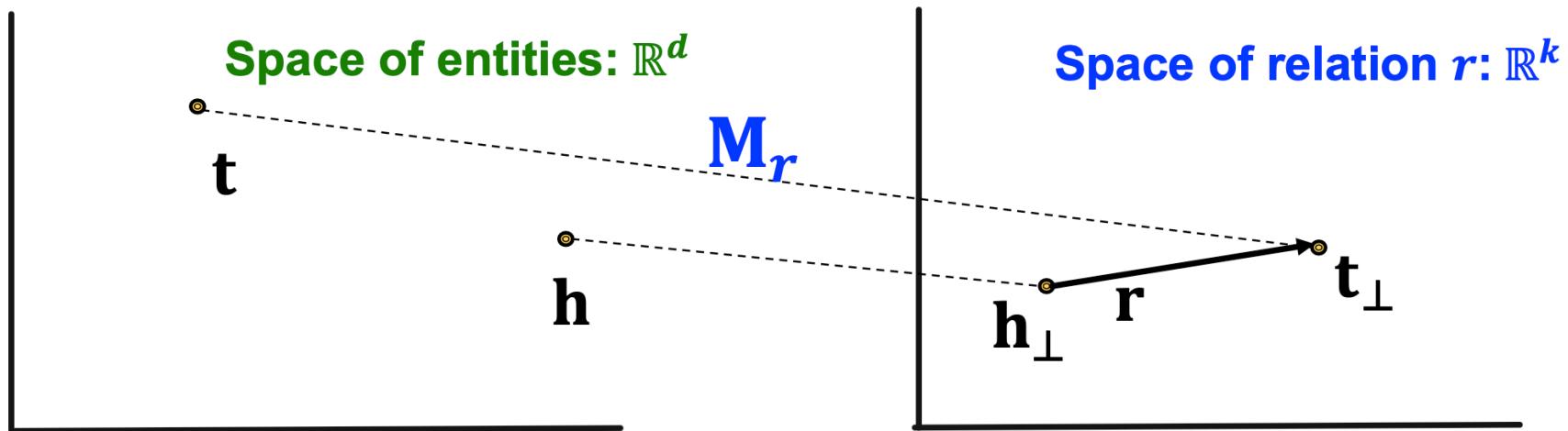
- Example:  $(h, r, t_1)$  and  $(h, r, t_2)$  both exist in the knowledge graph, e.g.,  $r$  is “StudentsOf”
- TransE cannot model 1-to-N relations ✗
  - $t_1$  and  $t_2$  will map to the same vector, although they are different entities

- $t_1 = h + r = t_2$
- $t_1 \neq t_2$       contradictory!



# 6.5 TransR

- TransE models translation of any relation in the **same** embedding space.
- Can we design a new space for each relation and do translation in relation-specific space?
- TransR<sup>[2]</sup>
  - Model **entities** as vectors in the entity space  $\mathbb{R}^d$  and model each **relation** as vector in relation space  $\mathbb{R}^k$  with  $M_r \in \mathbb{R}^{k \times d}$  as the projection matrix.
  - Use  $M_r$  to project from entity space  $\mathbb{R}^d$  to relation space  $\mathbb{R}^k$
  - Score function:  $f(h, t) = -\|h_\perp + r - t_\perp\|$  where  $h_\perp = M_r h, t_\perp = M_r t$



# 6.5 TransR

## 6 Knowledge Graph

- **Symmetric Relations:**

$$r(h, t) \Rightarrow r(t, h) \quad \forall h, t$$

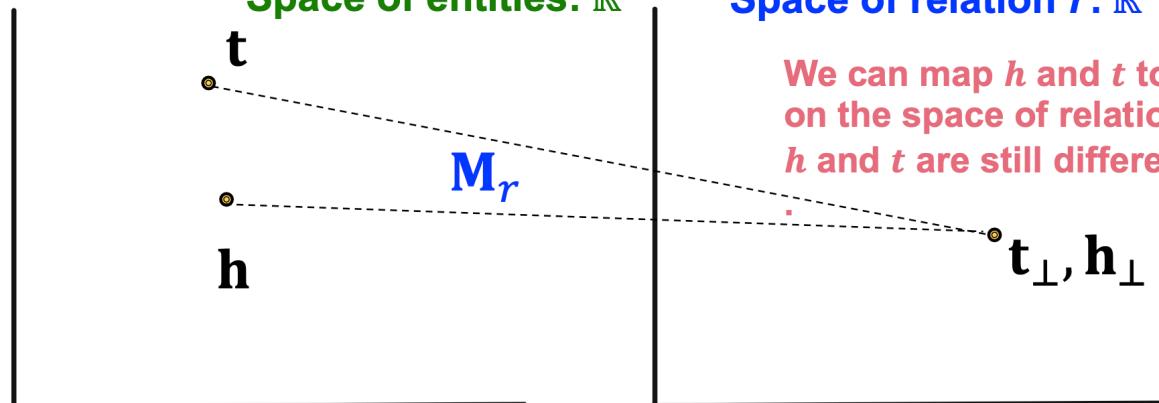
- **Example:** Family, Roommate

- **TransR** can model symmetric relations

$$\mathbf{r} = 0, \quad \mathbf{h}_\perp = \mathbf{M}_r \mathbf{h} = \mathbf{M}_r \mathbf{t} = \mathbf{t}_\perp \checkmark$$

Note different  
symmetric  
relations may  
have different  $\mathbf{M}_r$

What about anti-symmetric relations?



We can map  $h$  and  $t$  to the same location  
on the space of relations  $r$ . Then  $r = 0$ .  
 $h$  and  $t$  are still different in the entity space,

# 6.5 TransR

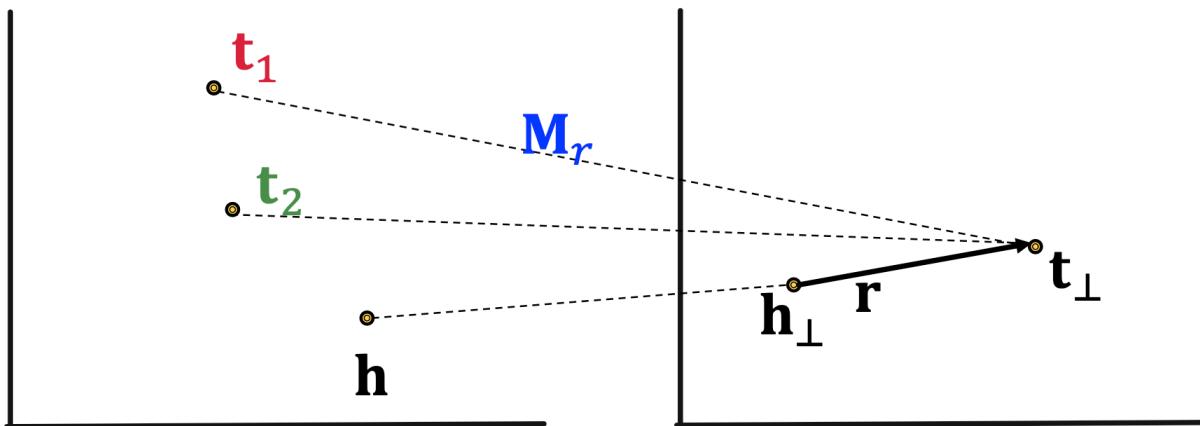
## 6 Knowledge Graph

### ■ 1-to-N Relations:

- **Example:** If  $(h, r, t_1)$  and  $(h, r, t_2)$  exist in the knowledge graph.

### ■ TransR can model 1-to-N relations ✓

- We can learn  $\mathbf{M}_r$  so that  $\mathbf{t}_\perp = \mathbf{M}_r \mathbf{t}_1 = \mathbf{M}_r \mathbf{t}_2$
- Note that  $\mathbf{t}_1$  does not need to be equal to  $\mathbf{t}_2$ !



# References

---

- **Deep Learning on Graphs:** [www.cse.msu.edu/~mayao4/dlg\\_book/dlg\\_book.pdf](http://www.cse.msu.edu/~mayao4/dlg_book/dlg_book.pdf)
- **Graph Representation Learning:** [https://www.cs.mcgill.ca/~wlh/grl\\_book/files/GRL\\_Book.pdf](https://www.cs.mcgill.ca/~wlh/grl_book/files/GRL_Book.pdf)
- GitHub curation: [thunlp/GNNPapers: Must-read papers on graph neural networks \(GNN\) \(github.com\)](https://github.com/thunlp/GNNPapers)
- Popular graph database: neo4j, nebula

# GNN Libraries

Library Name	Comment	Supported Framework
PyTorch Geometric (PyG) <sup>1</sup>	De-facto standard framework for GNNs in PyTorch ecosystem. PyG provides automatic batching and established community, with the ease of integration of common benchmark datasets.	Pytorch
Deep Graph Library (DGL) <sup>2</sup>	Focused on high-performance and scalable development, with a useful higher-level abstraction, allowing for auto-batching.	Pytorch, TF, MxNet
TF-GNN <sup>3</sup>	Designed to scale to large graphs, also support distributed training.	TF2
Spektral <sup>4</sup>	TF2 framework but without batching support.	TF2

1: <https://pytorch-geometric.readthedocs.io/>

2: <https://www.dgl.ai/>

3: <https://github.com/tensorflow/gnn>

4: <https://graphneural.network/>

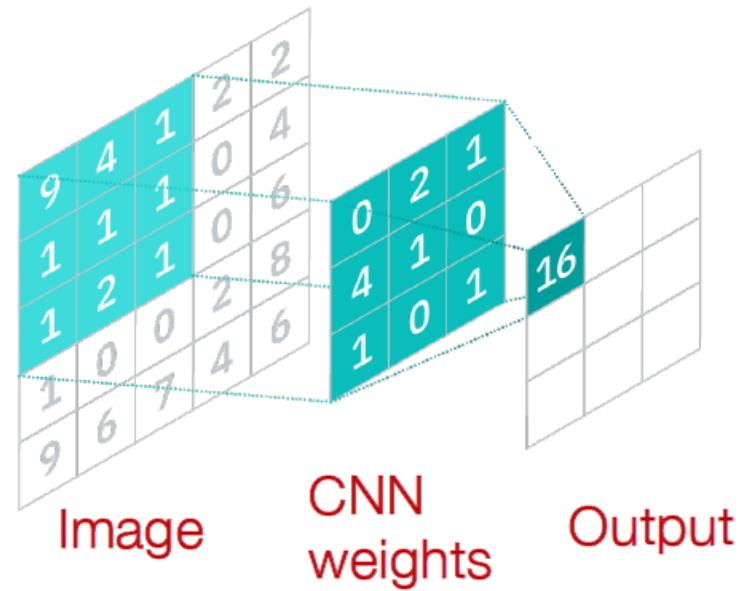
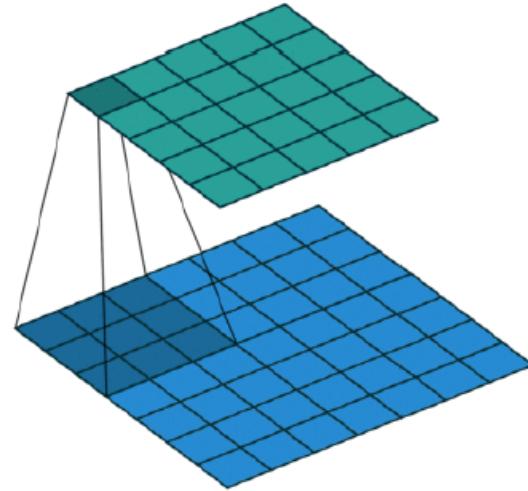


Thank you!

# Convolutional Neural Network

Supplementary Material

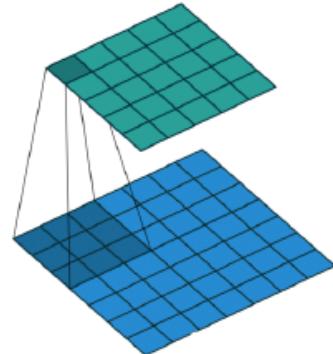
Convolutional neural network (CNN) layer with 3x3 filter:



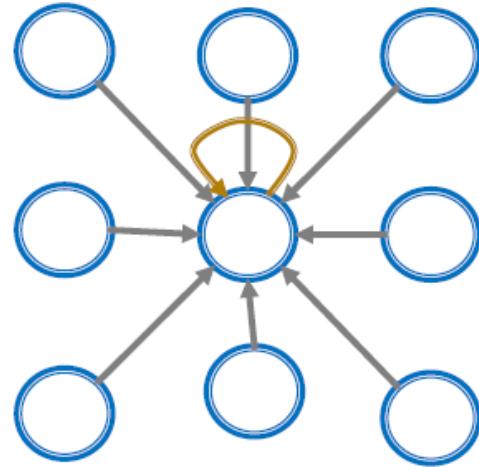
$$\text{CNN formulation: } h_v^{(l+1)} = \sigma(\sum_{u \in N(v) \cup \{v\}} W_l^u h_u^{(l)}), \quad \forall l \in \{0, \dots, L-1\}$$

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

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



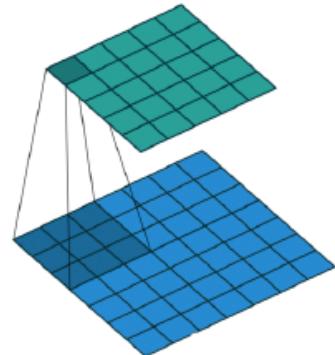
Image



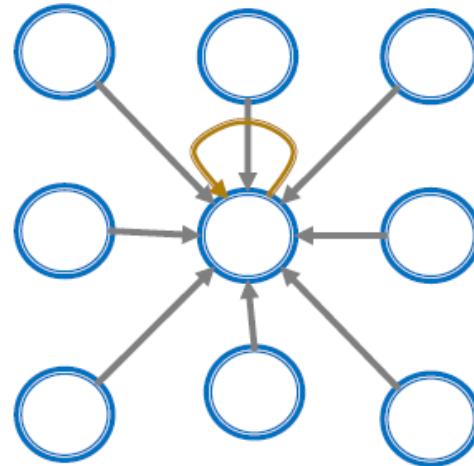
Graph

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

## 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:  $\{(-1, -1), (-1, 0), (-1, 1), \dots, (1, 1)\}$

- 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
  - We implement resizing and padding to make all input samples unified size
  - The advantage of GNN is it processes arbitrary graphs with different degrees for each node
- CNN is not permutation invariant / equivariant
  - Switching the order of pixels leads to different outputs