BOSTON UNIVERSITY

Deep Learning for Data Science
DS 542

https://dl4ds.github.io/fa2025/

Graph Neural Networks

Original slides by Tom Gardos,
other content from Understanding Deep Learning unless otherwise cited

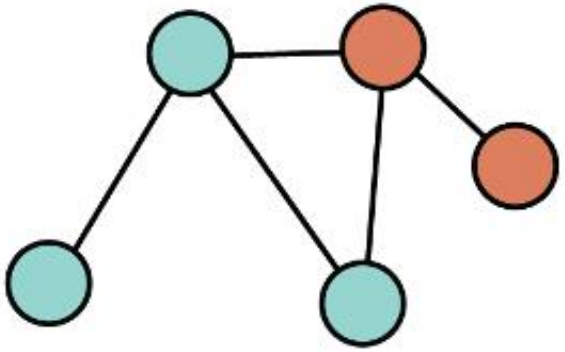
Plan for Today

- Basic definition and examples
- Graph representation
- Properties of Adjacency Matrix
- Graph neural network, tasks and loss functions
- Graph convolutional network
- Graph & Node classification
- Edge graphs

Project 4

Colors

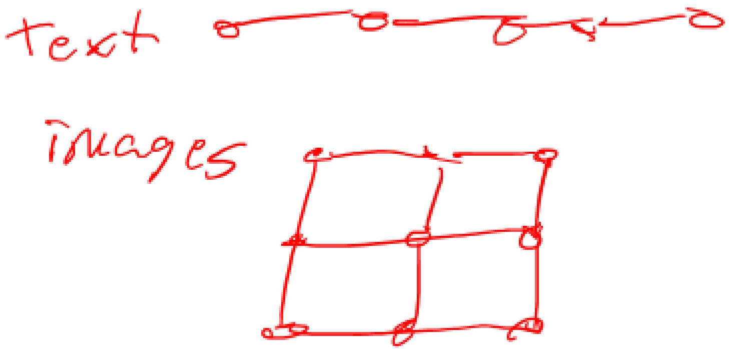
Graph Neural Networks



Neural architectures that process graphs.

Three challenges:

1. Variable topology
2. Size (billions of nodes)
3. Single monolithic graph



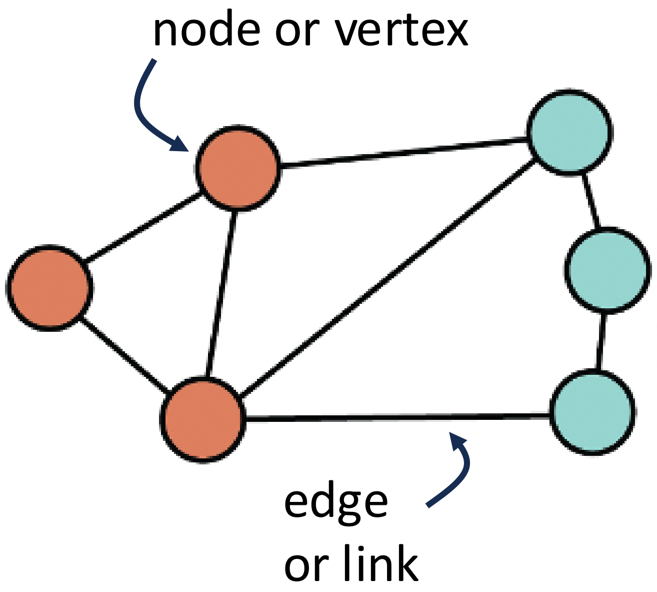
more flexible

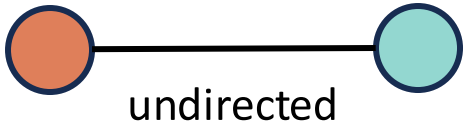
Graph (Network)

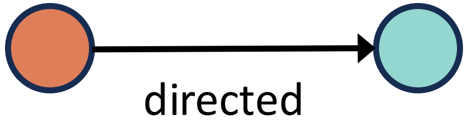
- general structure composed of *nodes* (vertices) and *edges* (links)

- edges can be *undirected* or *directed*

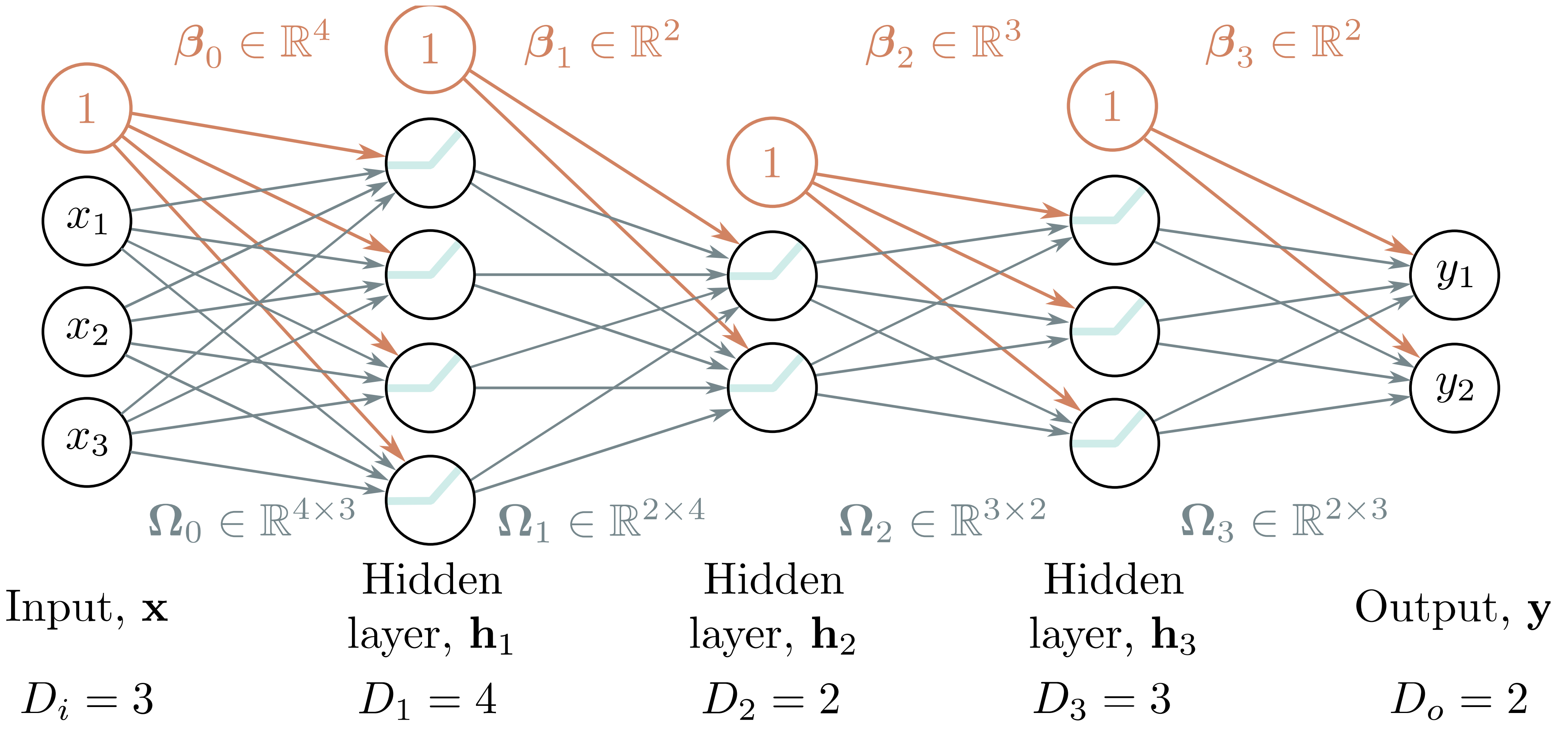
- a graph with directed edges and no cycles (no loops) is called *directed acyclic graph* (DAG)



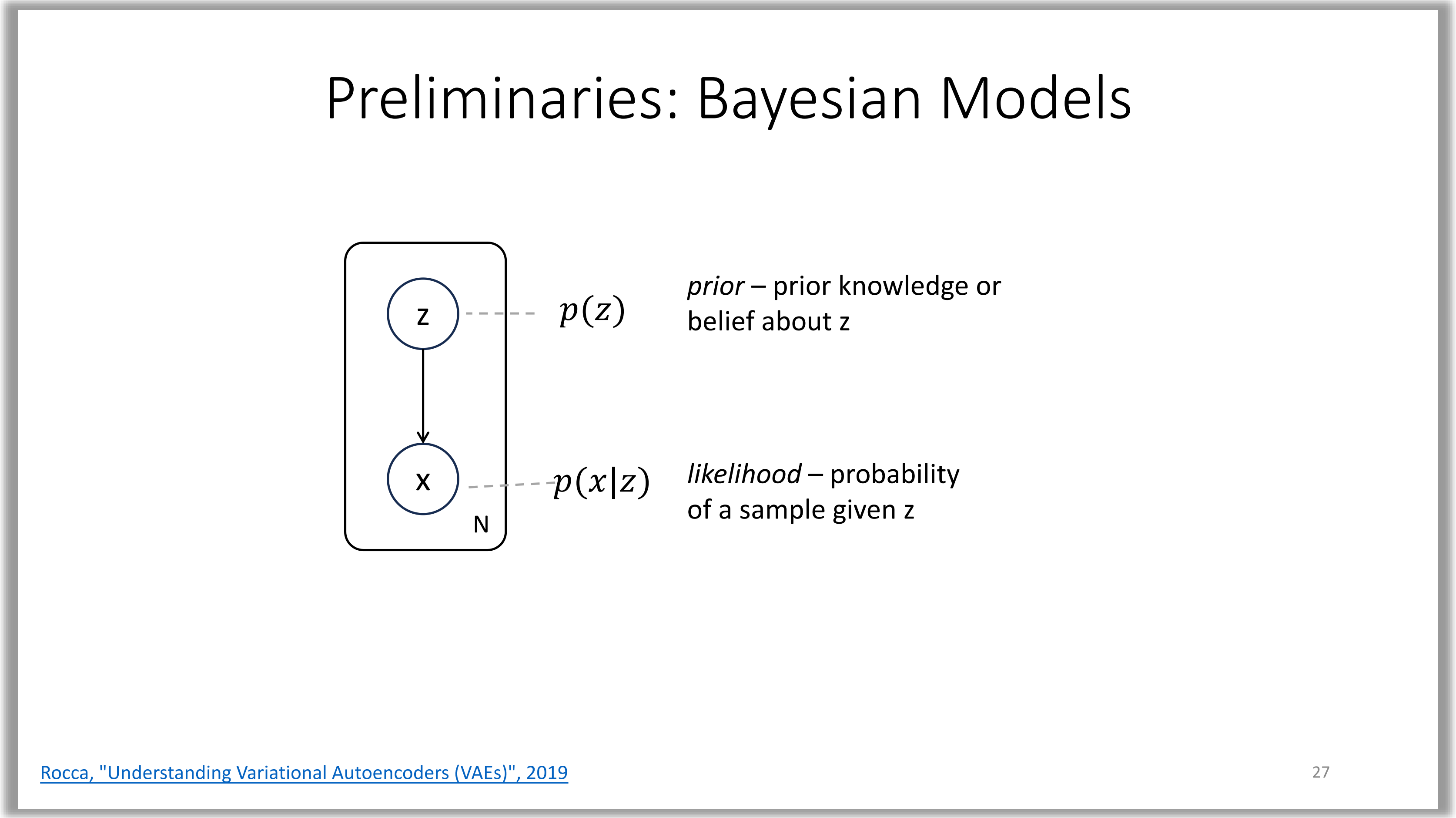




Directed Example – Feed Forward Network


$\beta_0 \in \mathbb{R}^4$
$\beta_1 \in \mathbb{R}^2$
$\beta_2 \in \mathbb{R}^3$
$\beta_3 \in \mathbb{R}^2$
1
1
1
1
x_1
x_2
x_3
y_1
y_2
$\Omega_0 \in \mathbb{R}^{4\times 3}$
$\Omega_1 \in \mathbb{R}^{2\times 4}$
$\Omega_2 \in \mathbb{R}^{3\times 2}$
$\Omega_3 \in \mathbb{R}^{2\times 3}$
Input, $\mathbf{x}$
Hidden layer, $\mathbf{h}_1$
Hidden layer, $\mathbf{h}_2$
Hidden layer, $\mathbf{h}_3$
Output, $\mathbf{y}$
$D_i = 3$
$D_1 = 4$
$D_2 = 2$
$D_3 = 3$
$D_o = 2$

Directed Example – Bayesian Graphical Model



Preliminaries: Bayesian Models

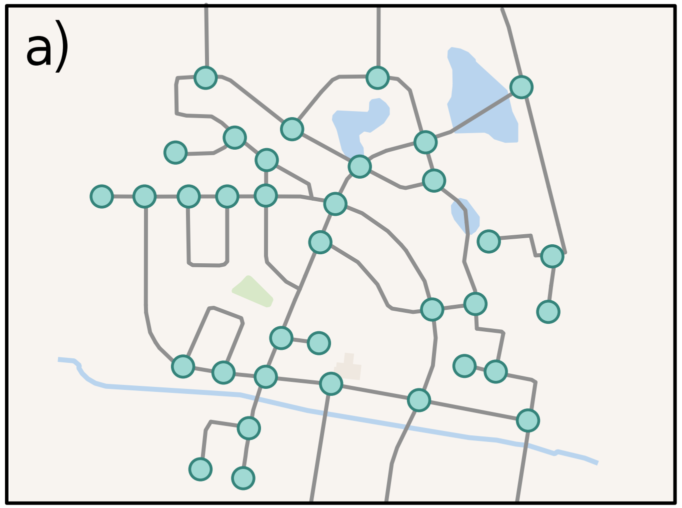
$p(z)$ *prior* – prior knowledge or belief about z

$p(x|z)$ *likelihood* – probability of a sample given z

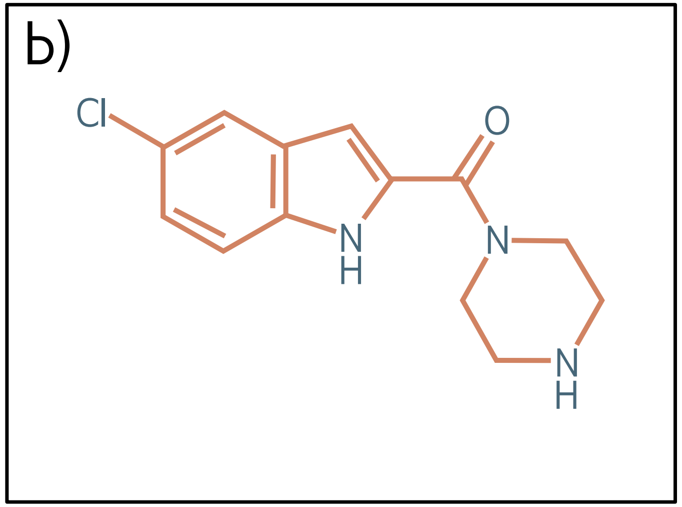
Rocca, "Understanding Variational Autoencoders (VAEs)", 2019

27

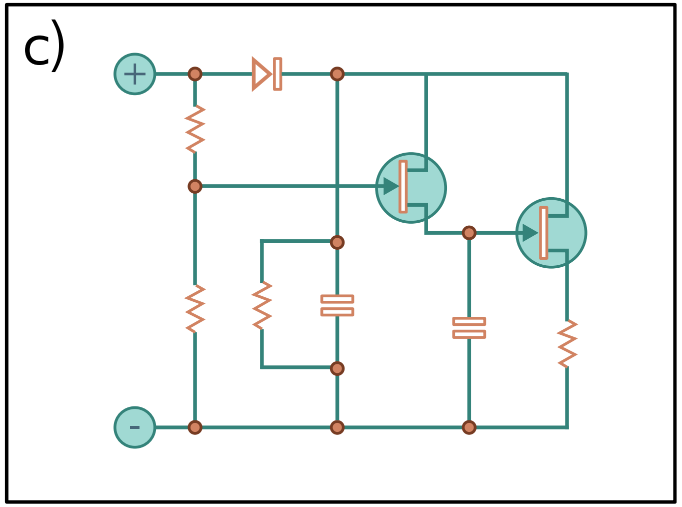
Undirected Examples



road networks
nodes: physical locations or landmarks
edges: connecting roads

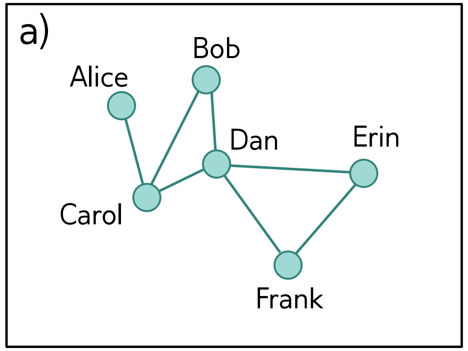


chemical molecules
nodes: atoms
edges: chemical bonds

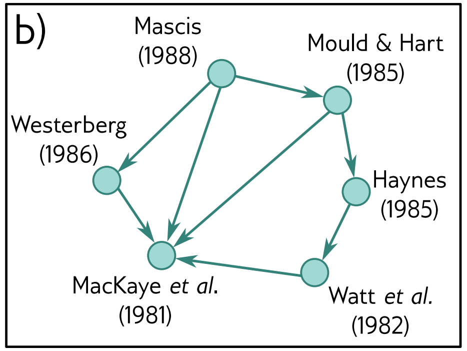


electrical circuits
nodes: components or junctions
edges: wires/electrical connections

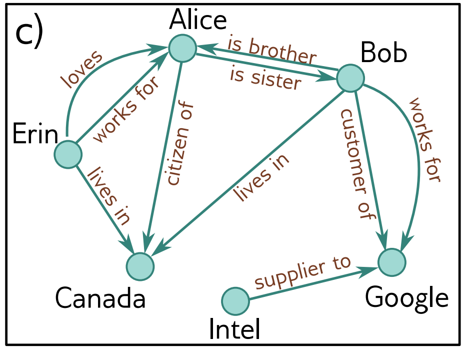
Examples



social networks
nodes: people
edges: friendships
(undirected)

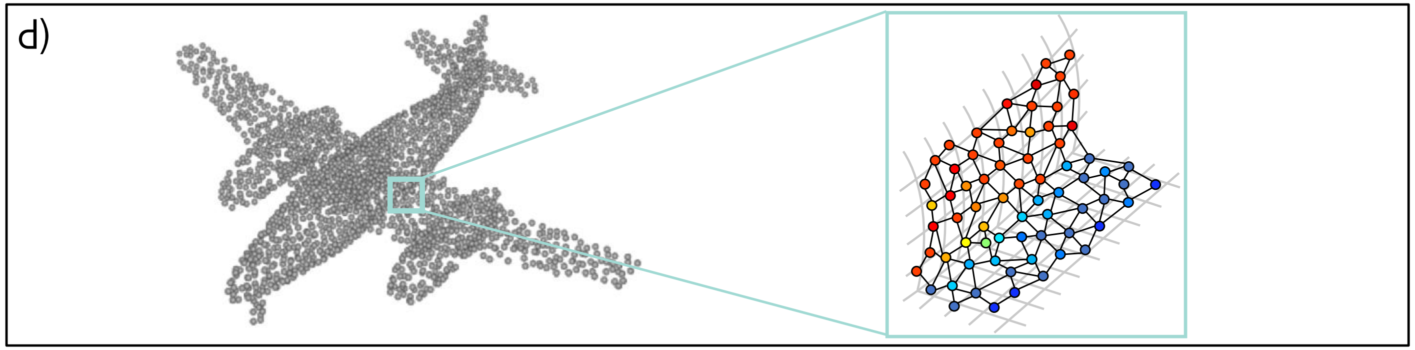


science literature
nodes: papers
edges: citations
(acyclic directed)



knowledge graph
nodes: objects
edges: named relationship
(cyclic directed)

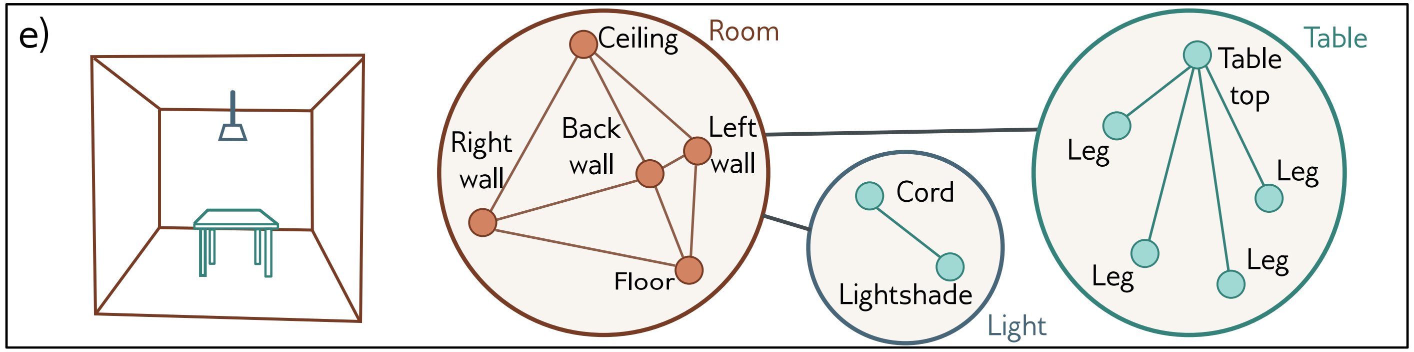
Example – Geometric Point Cloud



nodes: positions in 3D space (vertex in 3D graphics)
edges: connections to nearby points
(undirected)

https://en.wikipedia.org/wiki/Vertex_(computer_graphics)

Example – Scene Graph



hierarchical graph showing relationship between objects in a 3D scene

nodes: composite graphs or objects in 3D space
edges: connections to nearby points
(undirected)

Fernandez-Madrigal and Gonzalez, "Multi-hierarchical graph search," 2002
Armeni et al, “3D Scene Graph: A structure for unified semantics, 3D space and camera,” 2020?
Wald et al, “Learning 3D Semantic Scene Graphs with Instance Embeddings,” 2022

Other examples

- Wikipedia – nodes are articles, edges are hyperlinks between articles
- Computer programs – nodes are syntax tokens, edges are computation between tokens (tensor graph from Gradients lecture)
- Protein interactions – nodes are proteins, edges exist where two proteins interface
- Set or list – every element is connected to every other element
- image – each pixel is a node with edges to the eight adjacent pixels

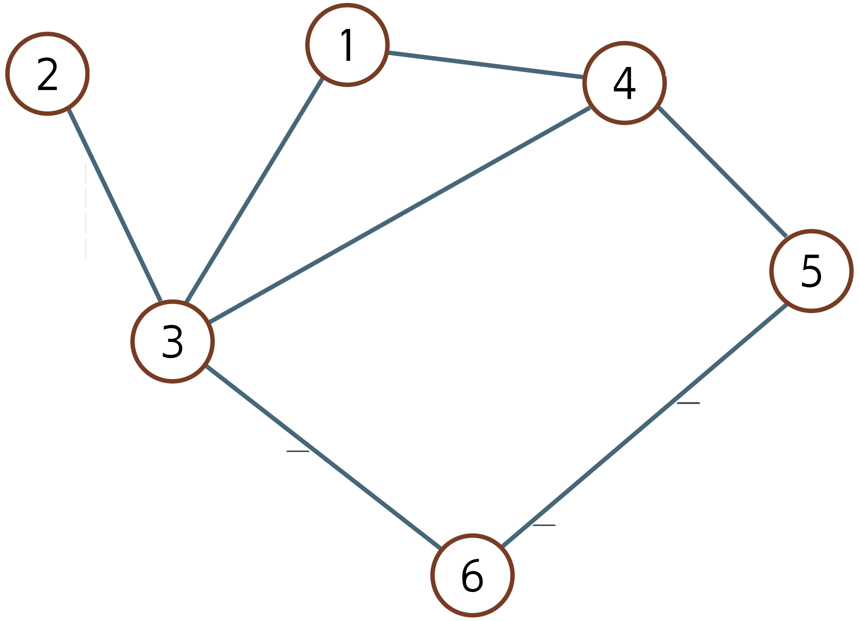
Any Questions?

???

Moving on

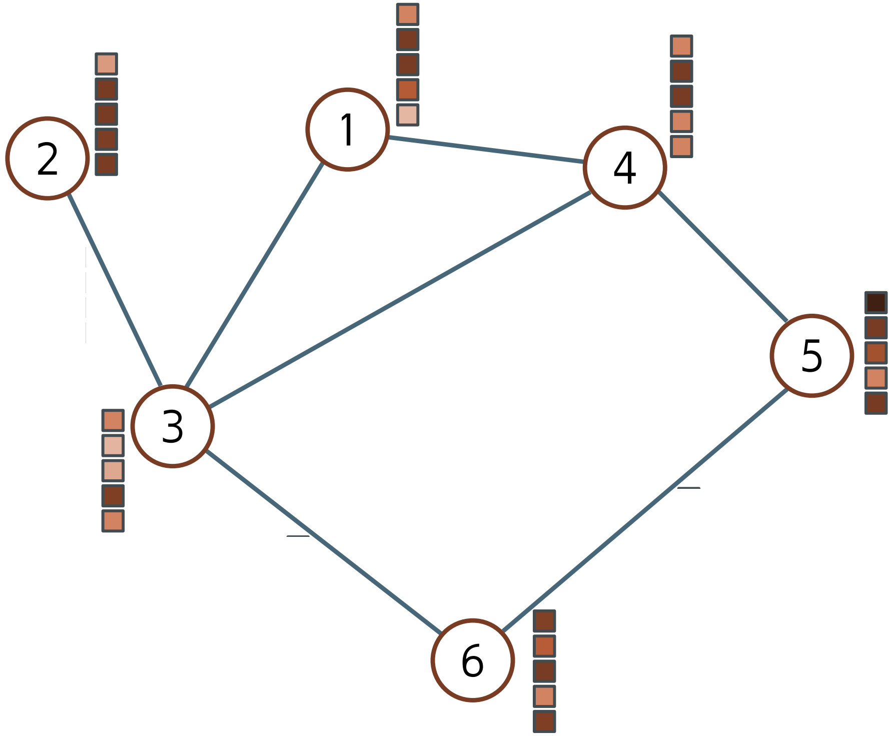
- Basic definition and examples
- Graph representation
- Properties of Adjacency Matrix
- Graph neural network, tasks and loss functions
- Graph convolutional network
- Graph & Node classification
- Edge graphs

Graph representation



Example undirected graph with 6 nodes

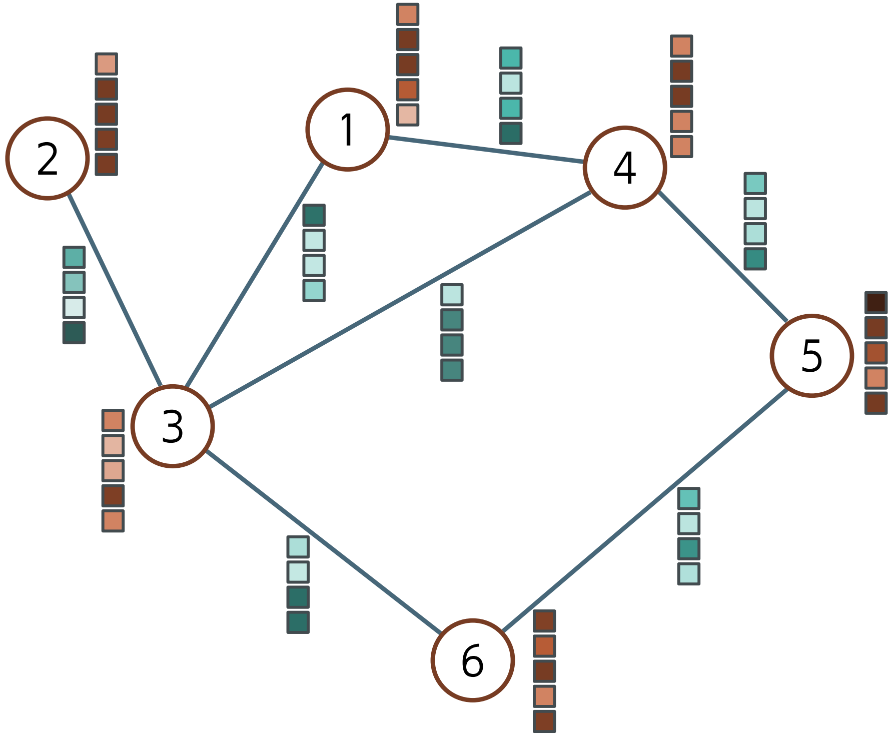
Graph representation – node embedding



Example undirected graph with 6 nodes

Information about a node is stored in a *node embedding* → initial data?

Graph representation – edge embedding



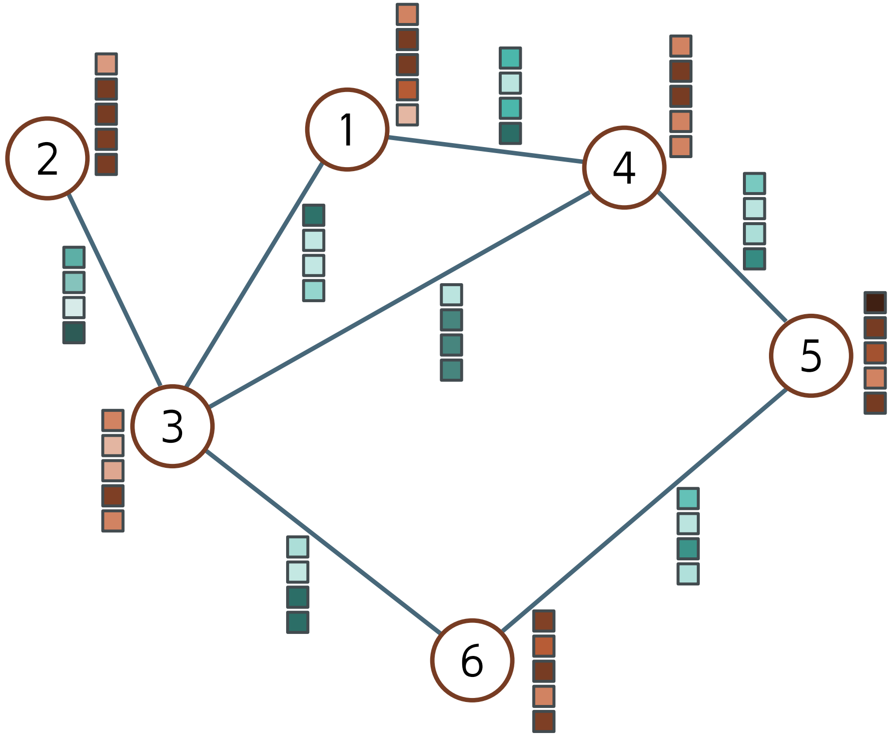
Example undirected graph with 6 nodes

Information about a node is stored in a *node embedding*

Information about an edge is stored in an *edge embedding*

what does this connection represent?

Graph representation – adjacency matrix



Adjacency matrix, $\mathbf{A}$

$N \times N$

	1	2	3	4	5	6
1	□	□	■	■	□	□
2	□	□	■	□	□	□
3	■	■	□	■	□	■
4	■	□	■	□	■	□
5	□	□	□	■	□	■
6	□	□	■	□	■	□

Assume we have N nodes

The graph connections can be represented by an *adjacency matrix*

Where a value of $\overset{0}{1}$ at (m, n) represents a connection between nodes m and n.

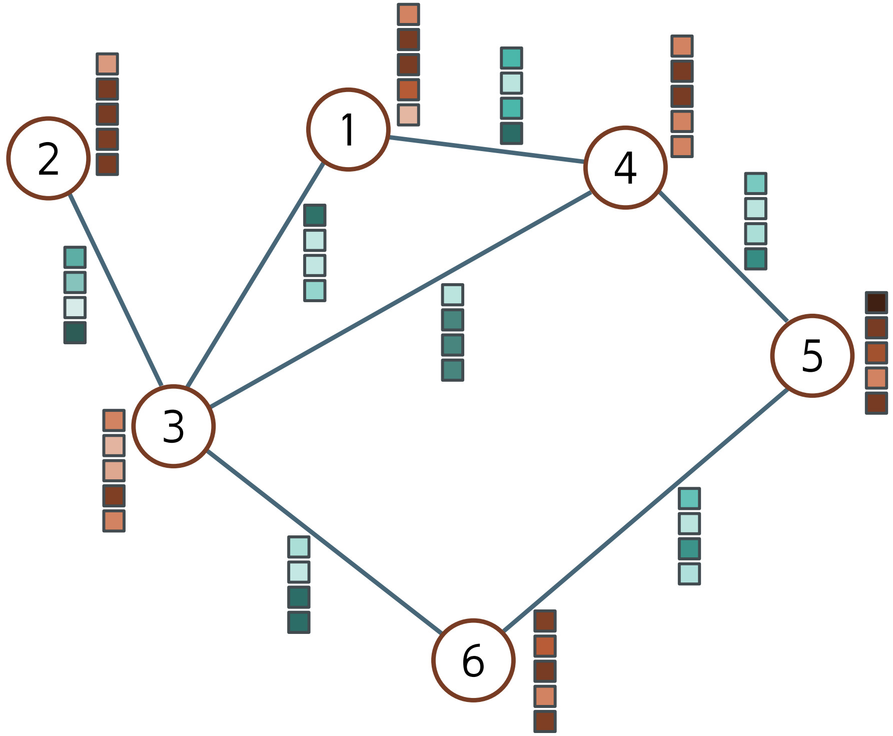
For undirected graphs the matrix is always symmetric about the diagonal

Diagonal is zero – no edge to itself
no self

Can be very sparse

0 if no edge.

Graph representation – node data matrix



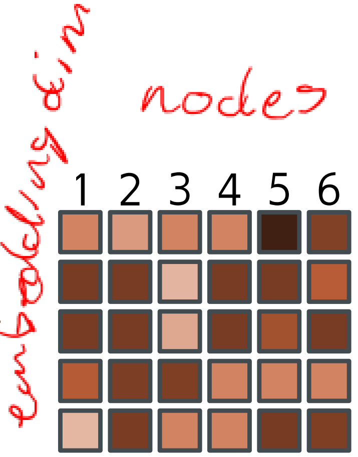
Adjacency matrix, $\mathbf{A}$

$N \times N$

	1	2	3	4	5	6
1	□	□	■	■	□	□
2	□	□	■	□	□	□
3	■	■	□	■	□	■
4	■	□	■	□	■	□
5	□	□	□	■	□	■
6	□	□	■	□	■	□

Node data, $\mathbf{X}$

$D \times N$



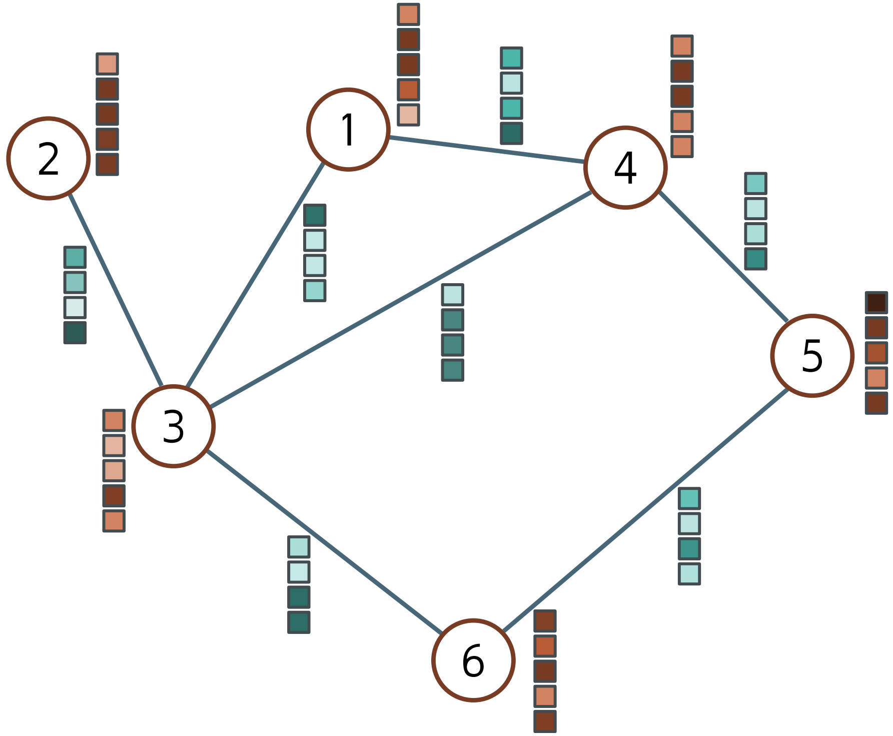
All the node data in the form of node embeddings can represented by a *Node data matrix*

Where D is the dimension of the note embedding and

N is the number of nodes

will be analogous to token embeddings for LLMs

Graph representation – edge data matrix



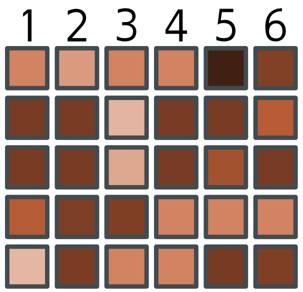
Adjacency matrix, $\mathbf{A}$

$N \times N$

	1	2	3	4	5	6
1	□	□	■	■	□	□
2	□	□	■	□	□	□
3	■	■	□	■	□	■
4	■	□	■	□	■	□
5	□	□	□	■	□	■
6	□	□	■	□	■	□

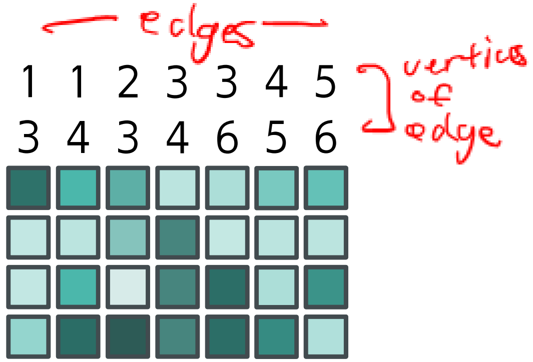
Node data, $\mathbf{X}$

$D \times N$



Edge data, $\mathbf{E}$

$D_E \times E$



Similarly, all the edge embedding information can be stored in an *Edge data matrix*, where:

D_E is the dimension of the edge embedding vector and

E is the number of edges

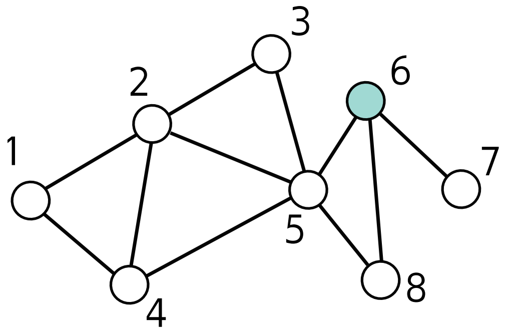
Any Questions?

???

Moving on

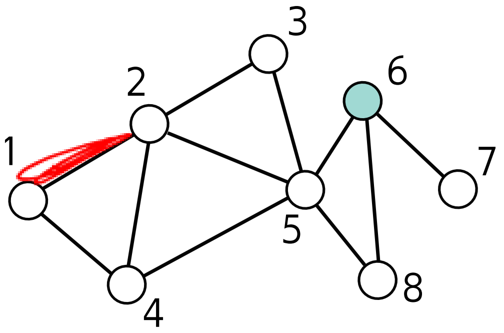
- Basic definition and examples
- Graph representation
- Properties of Adjacency Matrix
- Graph neural network, tasks and loss functions
- Graph convolutional network
- Graph & Node classification
- Edge graphs

Adjacency Matrix



Assume we have an 8-node undirected graph

Adjacency Matrix

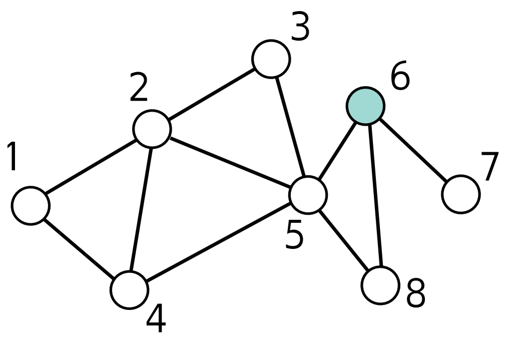


vertex 1 + 2 connected

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

Adjacency matrix for this graph.

Adjacency Matrix



$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

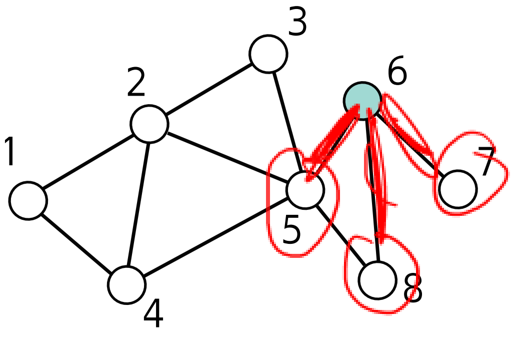
adjacency matrix

node representation (selection)

$$\mathbf{x} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

We can one hot encode representation of node 6

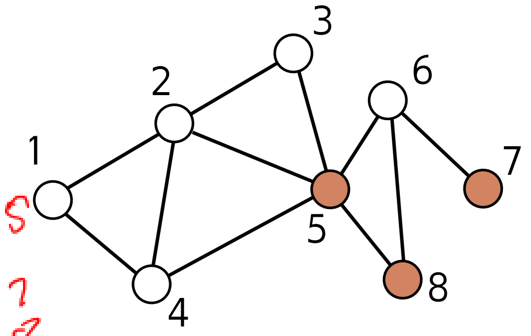
Adjacency Matrix



$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

$$\mathbf{x} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

$$\mathbf{Ax} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} \begin{array}{l} \\ \\ \\ \\ 5 \\ \\ 7 \\ 8 \end{array}$$

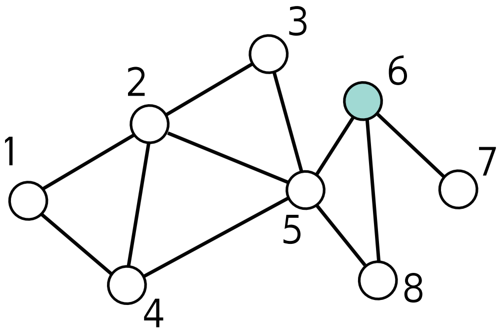


If we pre-multiply the one-hot encoded data node vector x by adjacency matrix A we get the 6th column of A indicating direct connections to other nodes

same as selecting LLM token embedding w/ 1 hot

One-hot encoding vector of all nodes directly connected node 6

Adjacency Matrix

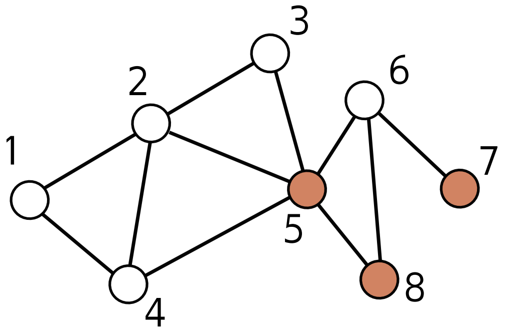


$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

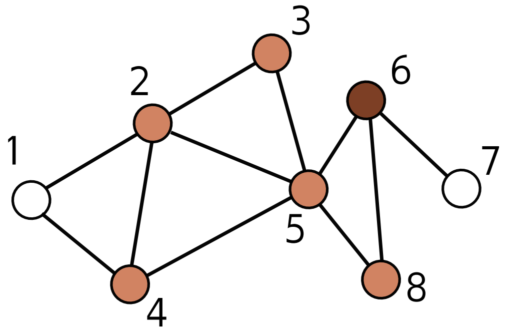
If we pre-multiply again by A, we get a vector showing the number of times we can get to each node in 2 steps.

$$\mathbf{x} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

$$\mathbf{Ax} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$



$$\mathbf{A}^2\mathbf{x} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 3 \\ 0 \\ 1 \end{bmatrix}$$



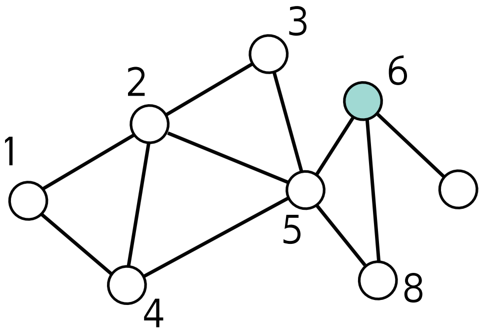
Graph showing all nodes that can be reached in *exactly* 2 steps.

NOT ≤2, EXACTLY 2

Adjacency Matrix

Pre-multiplying x by A twice is equivalent to the matrix A^2

Shows how many times you can get from node m to node n in 2 steps

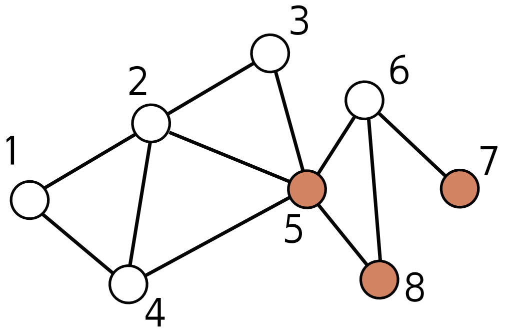


$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

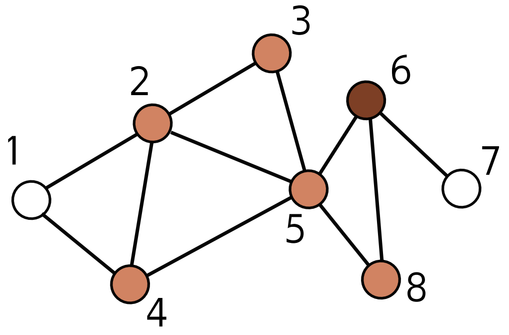
$$\mathbf{A}^2 = \begin{bmatrix} 2 & 1 & 1 & 1 & 2 & 0 & 0 & 0 \\ 1 & 4 & 1 & 2 & 2 & 1 & 0 & 1 \\ 1 & 1 & 2 & 2 & 1 & 1 & 0 & 1 \\ 1 & 2 & 2 & 3 & 1 & 1 & 0 & 1 \\ 2 & 2 & 1 & 1 & 5 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 3 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 2 \end{bmatrix}$$

$$\mathbf{x} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

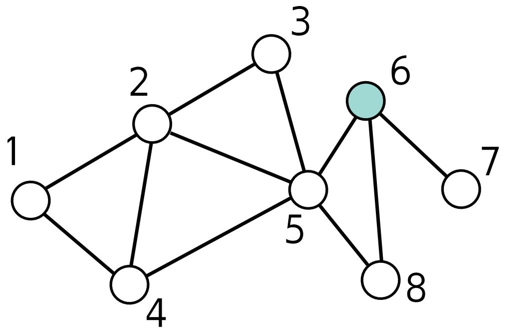
$$\mathbf{Ax} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$



$$\mathbf{A}^2\mathbf{x} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 3 \\ 0 \\ 1 \end{bmatrix}$$



Adjacency Matrix



$$\mathbf{A}^2 = \begin{bmatrix} 2 & 1 & 1 & 1 & 2 & 0 & 0 & 0 \\ 1 & 4 & 1 & 2 & 2 & 1 & 0 & 1 \\ 1 & 1 & 2 & 2 & 1 & 1 & 0 & 1 \\ 1 & 2 & 2 & 3 & 1 & 1 & 0 & 1 \\ 2 & 2 & 1 & 1 & 5 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 3 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 2 \end{bmatrix}$$

Example for $L = 2$

When you raise the adjacency matrix to the power of L (pre-multiply L-1 times),

the entry at position (m, n) of $\mathbf{A}^L$ contains the number of unique walks of length L from node n to node m

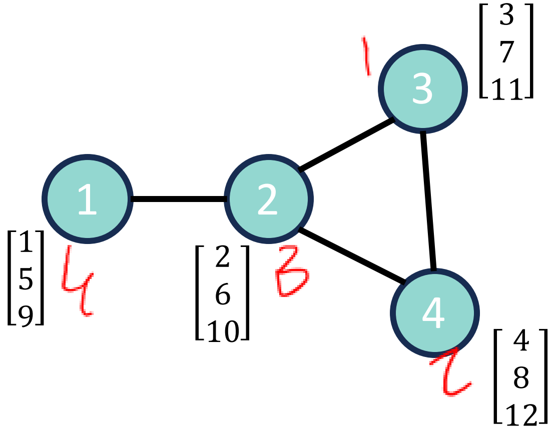
Note: this is not the same as the number of unique paths since it includes routes that visit the same node more than once.

a non-zero entry at position (m, n) indicates that the distance from m to n must be less than or equal to L.

See Notebook 13.1 – Encoding Graphs

Permutation of node indices

Since node indexing is arbitrary, we can permute the node indices



$$\mathbf{X} = \begin{array}{c} \begin{matrix} (1 & 2 & 3 & 4) \end{matrix} \\ \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \end{array}$$

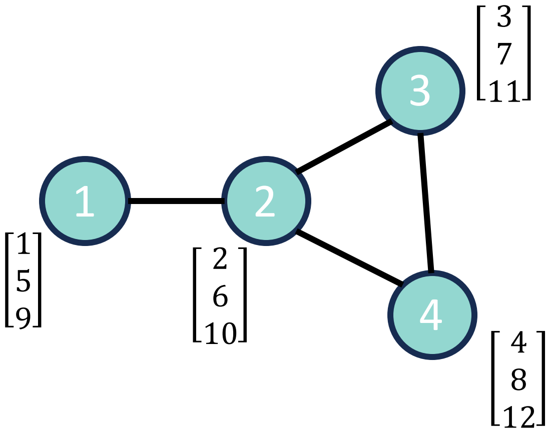
node data

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

adjacency matrix

Permutation of node indices

Since node indexing is arbitrary, we can permute the node indices



$$\mathbf{X} = \begin{array}{cccc} (1 & 2 & 3 & 4) \\ \hline 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{array}$$

node data

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

adjacency matrix

$$\mathbf{P} = \begin{array}{cccc} (3 & 4 & 2 & 1) \\ \hline 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{array}$$

We can express this mathematically with a permutation matrix, $\mathbf{P}$

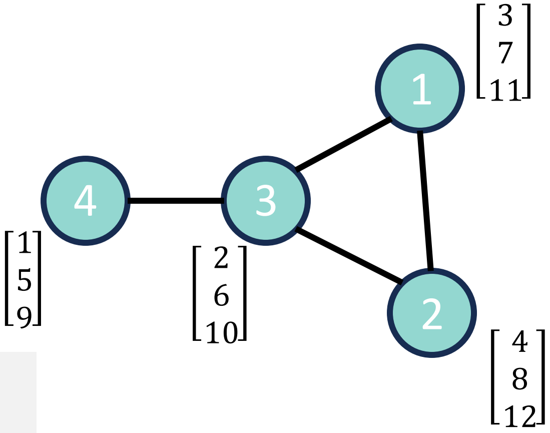
$$\begin{array}{lcccc} \text{New:} & (1 & 2 & 3 & 4) \\ \text{Old:} & (3 & 4 & 2 & 1) \end{array}$$

$$\mathbf{X}' = \mathbf{X}\mathbf{P} = \begin{bmatrix} 3 & 4 & 2 & 1 \\ 7 & 8 & 6 & 5 \\ 11 & 12 & 10 & 9 \end{bmatrix}$$

Permute the columns of the Node data matrix

$$\mathbf{A}' = \mathbf{P}^{\mathbf{T}}\mathbf{A}\mathbf{P} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Permute both the rows and column of the Adjacency matrix



Any Questions?

???

Moving on

- Basic definition and examples
- Graph representation
- Properties of Adjacency Matrix
- Graph neural network, tasks and loss functions
- Graph convolutional network
- Graph & Node classification
- Edge graphs

Graph Neural Network

- A graph neural network is a model that takes the node embeddings $\mathbf{X}$ and the adjacency matrix $\mathbf{A}$ as inputs and passes them through a series of K layers. Sounds like layers of convolutions or attention
- The node embeddings are updated at each layer to create intermediate "hidden" representations $\mathbf{H}_K$ before finally computing output embeddings $\mathbf{H}_K$.
- At the start of this network, each column of the input node embeddings $\mathbf{X}$ just contains information about the node itself.
- At the end, each column of the model output $\mathbf{H}_K$ includes information about the node and its context within the graph. like attention
- This is like word embeddings passing through a transformer network. These represent words at the start but represent the word meanings in the context of the sentence at the end.

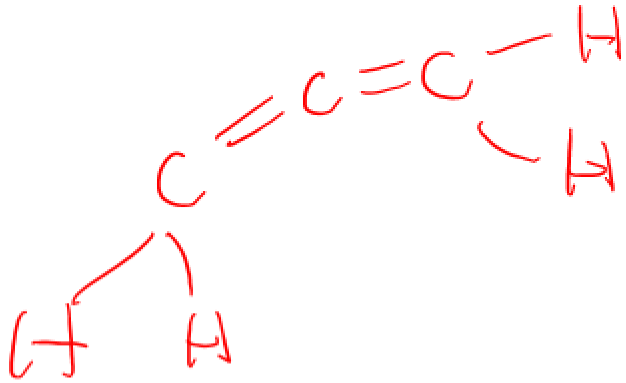
Graph Level Tasks

describe whole graph

Determine

- class categories, e.g. molecule is poisonous
- regression values, e.g. molecule boiling and freezing point

based on graph structure and node embeddings



For graph-level tasks, the output node embeddings are combined (e.g., by averaging), and the resulting vector is mapped via a linear transformation or neural network to a fixed-size vector

Per node $H_K \rightarrow \text{mean}(H_K) \rightarrow$ prediction

$D \times N \rightarrow D \rightarrow 1$ or C

Typical Three Types of Models

- Graph level regression & classification just talked about this
- Node level regression & classification per node output

  sometimes generalizing from partial data
- Edge prediction

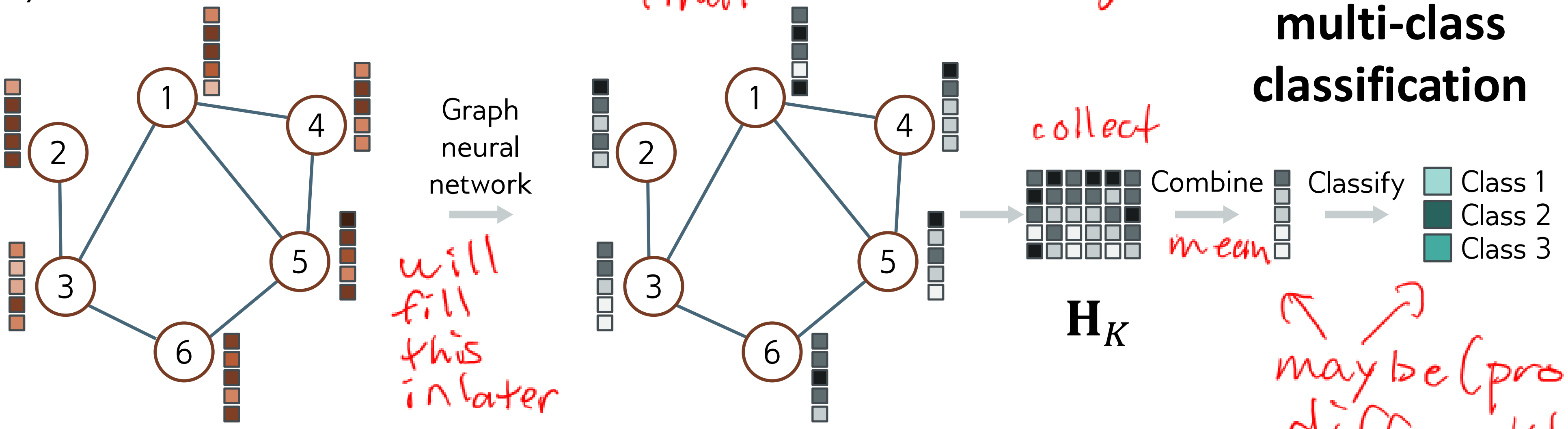
  e.g. are these nodes supposed to be connected?

Look at prediction heads first.

Graph level regression & classification

final node embeddings

multi-class classification



will fill this in later

collect

mean

maybe (probably) different lengths

Last layer (Regression): $$Pr(y|\mathbf{X},\mathbf{A}) = \beta_K + \omega_K \mathbf{H}_K \mathbf{1}\ /N$$

Last layer (Classification): $$Pr(y=1|\mathbf{X},\mathbf{A}) = \text{sigmoid}[\beta_K + \omega_K \mathbf{H}_K \underbrace{\mathbf{1}\ /N]}_{\text{Mean pooling}}$$

β_K is scalar

ω_K is $1 \times D$ row vector

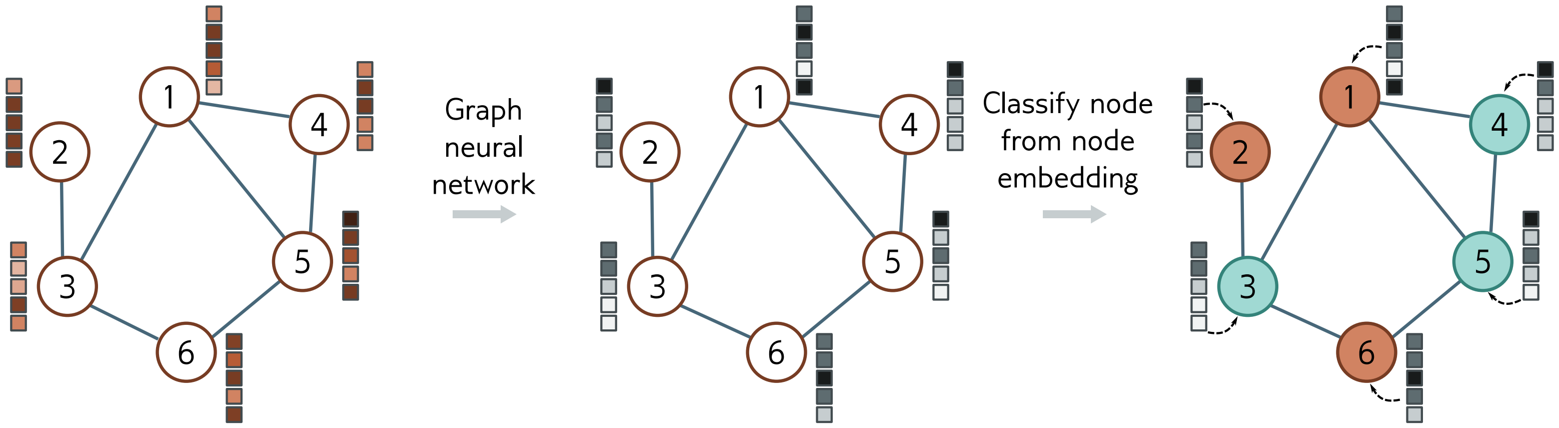
$\mathbf{H}_K$ is the $D \times N$ output embedding matrix

$\mathbf{1}$ is an $N \times 1$ column vector of 1s

Regression Loss Function: Least Squares Loss
Classification Loss Function: (Binary) Cross Entropy

binary classification. can do softmax for multiclass.

Node level binary regression & classification



Last layer (Regression): $\Pr\left(y^{(n)} \middle| \mathbf{X}, \mathbf{A}\right) = \beta_K + \omega_K \mathbf{h}_K^{(n)}$

embedding of node n
(after K layers of processing)

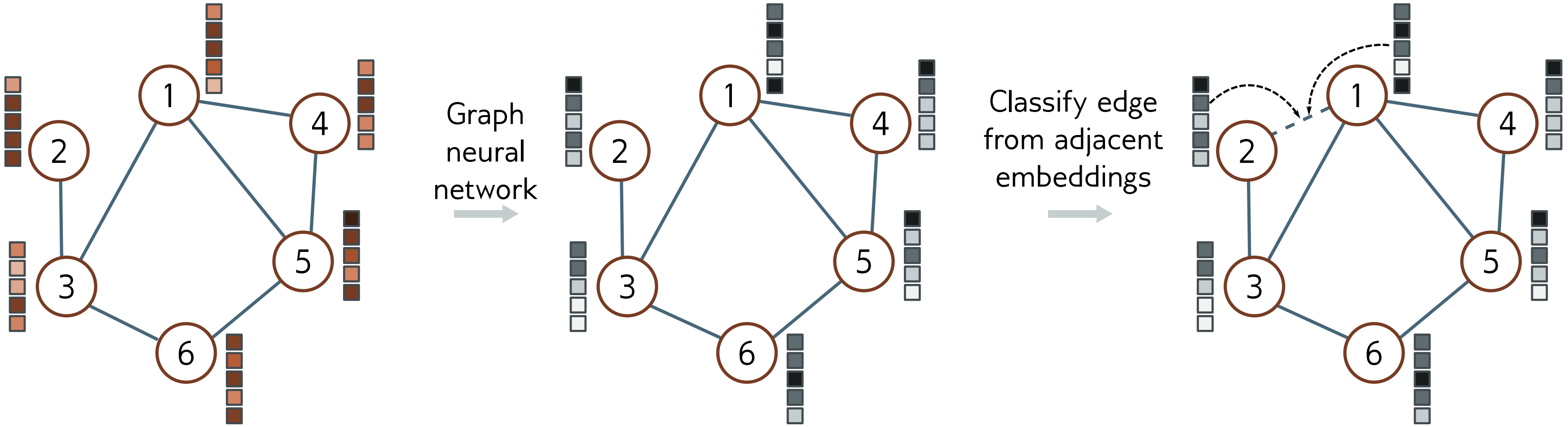
Last layer (Classification): $\Pr\left(y^{(n)} = 1 \middle| \mathbf{X}, \mathbf{A}\right) = \text{sigmoid}[\beta_K + \omega_K \mathbf{h}_K^{(n)}]$

$\mathbf{h}_K^{(n)}$ is the $D \times 1$ output embedding vector node for n

Regression Loss Function: Least Squares Loss
Classification Loss Function: (Binary) Cross Entropy

Edge prediction (classification)

Predict whether edge should exist or not.



Last layer: $\Pr\left(y^{(mn)} = 1 \middle| \mathbf{X}, \mathbf{A}\right) = \text{sigmoid}[\mathbf{h}_K^{(m)T} \mathbf{h}_K^{(n)}]$

$[1 \times D][D \times 1]$

dot product of node embedding → sigmoid input

Classification Loss Function: Binary Cross Entropy

Any Questions?

???

Moving on

- Basic definition and examples
- Graph representation
- Properties of Adjacency Matrix
- Graph neural network, tasks and loss functions
- Graph convolutional network
- Graph & Node classification
- Edge graphs

Graph convolutional network

These models are convolutional in that they update each node by aggregating information from nearby nodes.

As such, they induce a relational inductive bias (i.e., a bias toward prioritizing information from neighbors).

multiple layers of similar transforms

$$\begin{aligned}
\mathbf{H}_1 &= \mathbf{F}[\mathbf{X}, \mathbf{A}, \phi_0] \\
\mathbf{H}_2 &= \mathbf{F}[\mathbf{H}_1, \mathbf{A}, \phi_1] \\
\mathbf{H}_3 &= \mathbf{F}[\mathbf{H}_2, \mathbf{A}, \phi_2] \\
\vdots &= \vdots \\
\mathbf{H}_K &= \mathbf{F}[\mathbf{H}_{K-1}, \mathbf{A}, \phi_{K-1}],
\end{aligned}$$

A function $F[\cdot]$ with parameters ϕ_i that takes the node embeddings and adjacency matrix and outputs new node embeddings

each old embeddings → new embeddings

both are per node

Equivariance and Invariance

Every layer should be *equivariant* to index permutations

$$\mathbf{H}_{k+1}\mathbf{P} = \mathbf{F}[\mathbf{H}_k\mathbf{P}, \mathbf{P}^T\mathbf{A}\mathbf{P}, \phi_k]$$

And for node classification and edge prediction the output should be *invariant* to index permutations

$$y = \text{sigmoid}[\beta_K + \omega_K \mathbf{H}_K \overset{\text{dropping to ØD by summing}}{\mathbf{1}}/N\,] = \text{sigmoid}[\beta_K + \omega_K \mathbf{H}_K \underset{\text{add perm.}}{\boxed{\mathbf{P}}}\mathbf{1}/N\,]$$

TLDR permutations move outputs but do not change them

Example Graph Convolution Network (GCN) layer

At each node n in layer k, aggregate information from neighboring nodes

a specific neighbor

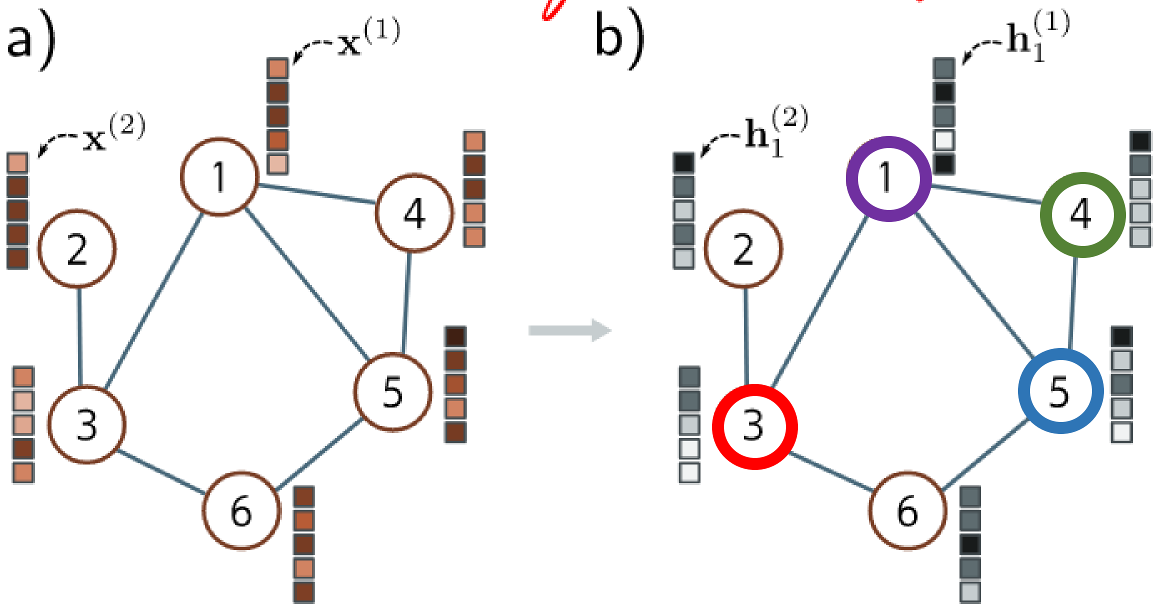
$$\text{agg}[n,k] = \sum_{m\in\text{ne}[n]} \mathbf{h}_k^{(m)}$$

vectors of neighbors of n

where $\text{ne}[n]$ returns the set of indices of the neighbors of node n.

neighbor function

SUM not MEAN



$$\text{ne}[1] = \{4, 5, 3\}$$

$$\text{agg}[n=1, k=1] = \mathbf{h}_1^{(4)} + \mathbf{h}_1^{(5)} + \mathbf{h}_1^{(3)}$$

Example Graph Convolution Network (GCN) layer

At each node n in layer k, aggregate information from neighboring nodes

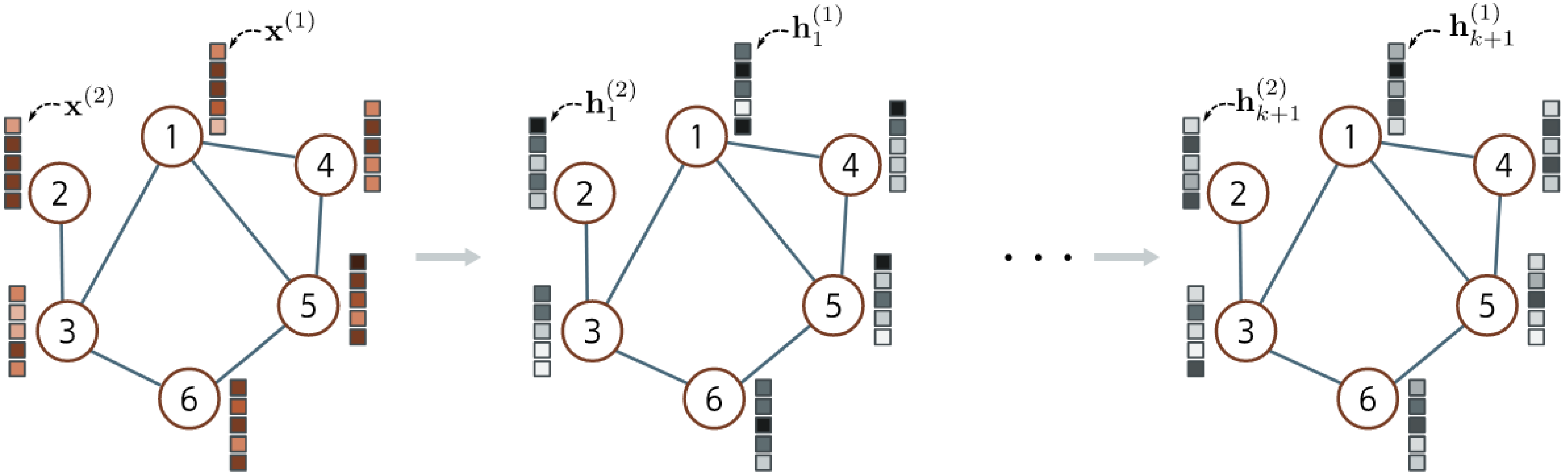
$$\mathrm{agg}[n,k] = \sum_{m \in \mathrm{ne}[n]} \mathbf{h}_k^{(m)}$$

where $\mathrm{ne}[n]$ returns the set of indices of the neighbors of node n.

Then a linear transform to the current node vector and the aggregate for the current node and add a bias.

$$\mathbf{h}_{k+1}^{(n)} = \mathbf{a}\left[\underset{D\times 1}{\boldsymbol{\beta}_k} + \underset{D\times D}{\boldsymbol{\Omega}_k} \cdot \underset{D\times 1}{\mathbf{h}_k^{(n)}} + \underset{D\times D}{\boldsymbol{\Omega}_k} \cdot \underset{D\times 1}{\mathrm{agg}[n,k]}\right]$$

Graph convolution layers



$$\mathbf{h}_1^{(n)} = \mathbf{a}\left[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0\mathbf{x}_1^{(n)} + \boldsymbol{\Omega}_0\mathbf{agg}[n]\right]$$

$$\mathbf{h}_{k+1}^{(n)} = \mathbf{a}\left[\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k\mathbf{h}_k^{(n)} + \boldsymbol{\Omega}_k\mathbf{agg}[n]\right]$$

Input

1st Layer

k+1st Layer

Example Graph Convolution Network (GCN) layer

We apply the following equation

$$\mathbf{h}_{k+1}^{(n)} = \mathbf{a}\left[\beta_k + \Omega_k \cdot \mathbf{h}_k^{(n)} + \Omega_k \cdot \text{agg}[n,k]\right]$$

← previous embedding put together (so $h_k^{()}$ ...)

to the entire node hidden layers matrix, $\mathbf{H}_k$, by noting that $\mathbf{H}_k\mathbf{A}$ produces a matrix where the n^{th} column is $\text{agg}[n,k]$.

$$\begin{aligned}\mathbf{H}_{k+1} &= \mathbf{a}\left[\boldsymbol{\beta}_k\mathbf{1}^T + \boldsymbol{\Omega}_k\mathbf{H}_k + \boldsymbol{\Omega}_k\mathbf{H}_k\mathbf{A}\right] \\ &= \mathbf{a}\left[\boldsymbol{\beta}_k\mathbf{1}^T + \boldsymbol{\Omega}_k\mathbf{H}_k(\mathbf{A}+\mathbf{I})\right],\end{aligned}$$

← $H_k A$ = concat of $\text{agg}(n,k)$

~ linear function + activation function

"~" b/c Adjacency list is input, not fixed

Example Graph Convolution Network (GCN) layer

We apply the following equation

$$\mathbf{h}_{k+1}^{(n)} = \mathbf{a}\left[\beta_k + \Omega_k \cdot \mathbf{h}_k^{(n)} + \Omega_k \cdot \text{agg}[n,k]\right]$$

to the entire node hidden layers matrix, $\mathbf{H}_k$, by noting that $\mathbf{H}_k\mathbf{A}$ produces a matrix where the n^{th} column is $\text{agg}[n,k]$.

$$\begin{aligned} \mathbf{H}_{k+1} &= \mathbf{a}\left[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \mathbf{H}_k + \boldsymbol{\Omega}_k \mathbf{H}_k \mathbf{A}\right] \\ &= \mathbf{a}\left[\boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \mathbf{H}_k (\mathbf{A} + \mathbf{I})\right], \end{aligned}$$

Note that this is (1) equivariant to permutations, (2) handles arbitrary number of neighbors, (3) exploits graph structure and (4) share parameters

Any Questions?

???

Moving on

- Basic definition and examples
- Graph representation
- Properties of Adjacency Matrix
- Graph neural network, tasks and loss functions
- Graph convolutional network
- Graph & Node classification
- Edge graphs

Graph classification example

We can put it all together and add a sigmoid layer

$$\begin{aligned}
\mathbf{H}_1 &= \mathbf{a}\left[\boldsymbol{\beta}_0\mathbf{1}^T + \boldsymbol{\Omega}_0\mathbf{X}(\mathbf{A}+\mathbf{I})\right]\\
\mathbf{H}_2 &= \mathbf{a}\left[\boldsymbol{\beta}_1\mathbf{1}^T + \boldsymbol{\Omega}_1\mathbf{H}_1(\mathbf{A}+\mathbf{I})\right]\\
\vdots &= \vdots\\
\mathbf{H}_K &= \mathbf{a}\left[\boldsymbol{\beta}_{K-1}\mathbf{1}^T + \boldsymbol{\Omega}_{K-1}\mathbf{H}_{k-1}(\mathbf{A}+\mathbf{I})\right]\\
\mathrm{f}[\mathbf{X},\mathbf{A},\boldsymbol{\Phi}] &= \mathrm{sig}\left[\beta_K + \boldsymbol{\omega}_K\mathbf{H}_K\underbrace{\mathbf{1}/N}_{\text{Mean pooling}}\right],
\end{aligned}$$

For classification on molecules,

$X \in \mathbb{R}^{118\times N}$: one hot encoding of 118 elements

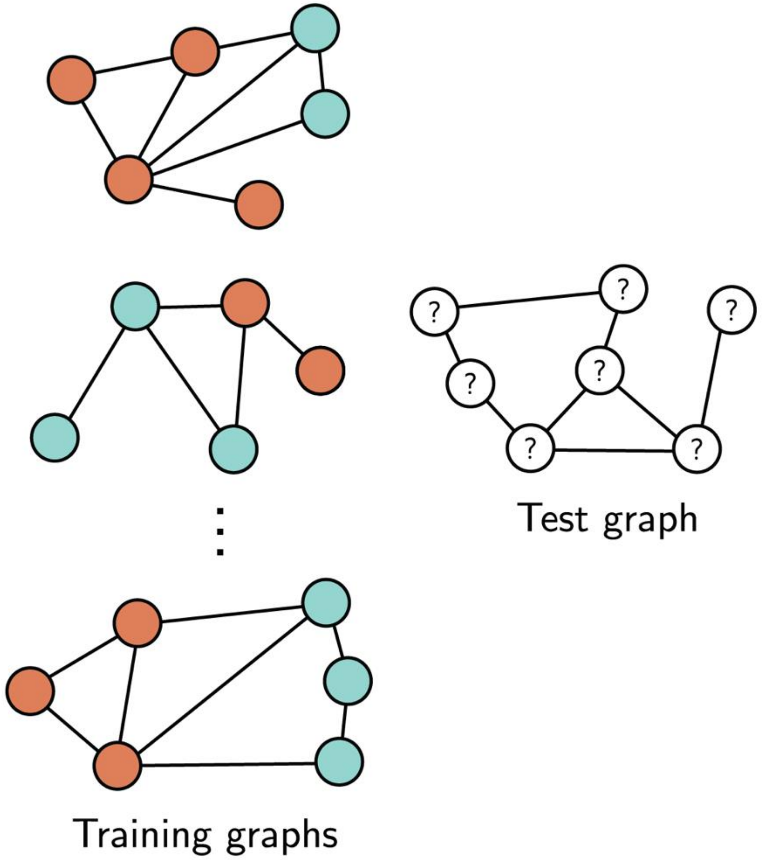
$\Omega_0 \in \mathbb{R}^{D\times 118}$: convert to D-dimensional embeddings

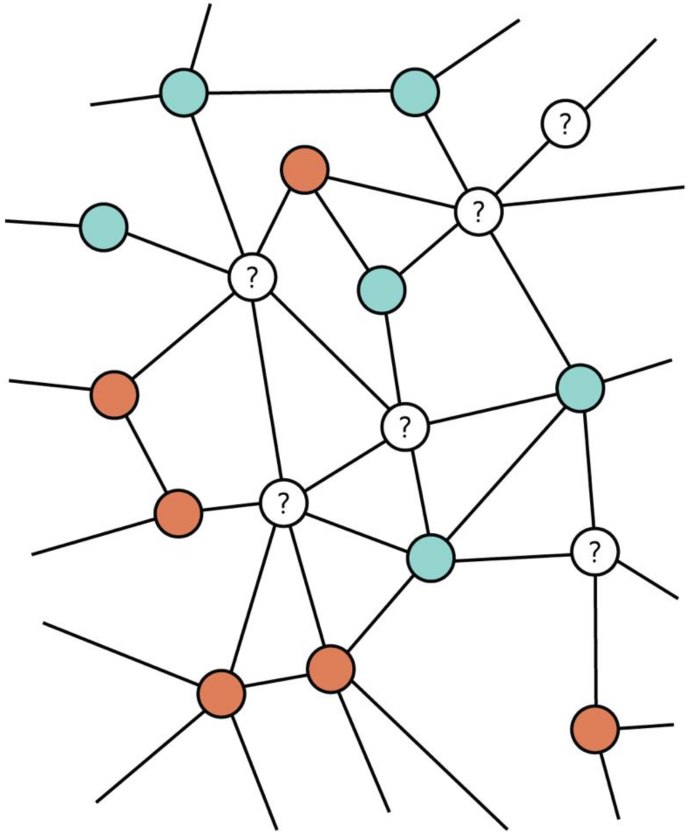
β_K: is a scalar

ω_K: a $1 \times D$ parameters row vector

See notebook 13.2.

Inductive vs. Transductive





<u>supervised learning</u>: train with the labeled graphs and then run inference on the unlabeled (test) graphs

<u>semi-supervised learning</u>: train with the labeled nodes, then run inference to determine label for unlabeled nodes

Node classification example

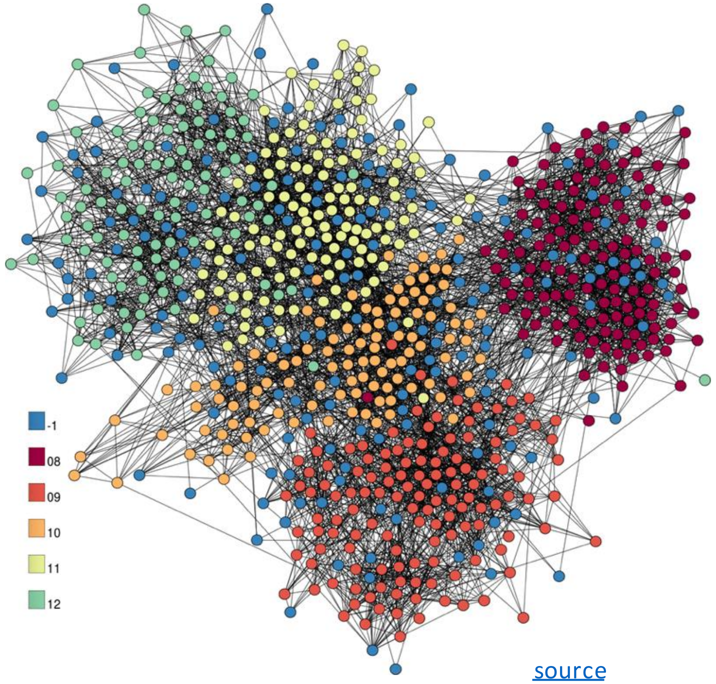
Assume *transductive* binary node *classification* with millions of nodes, *partially labeled*.

Same network body as graph classification, but different head:

$$\mathbf{f}[\mathbf{X}, \mathbf{A}, \boldsymbol{\Phi}] = \text{sigmoid}[\beta_K \mathbf{1}^T + \boldsymbol{\omega}_K \mathbf{H}_K]$$

No mean pooling. Output is $1 \times N$.

Train with binary cross-entropy loss on nodes with labels.



source

Node classification example

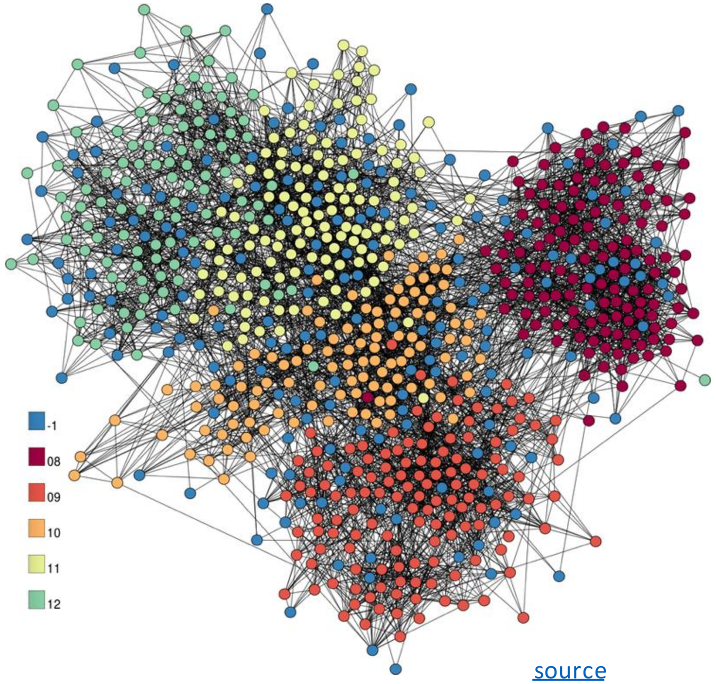
Assume *transductive* binary node *classification* with millions of nodes, *partially labeled*.

Challenges:

too big?

1. memory limitations: need to store every node and hidden layer embedding during training
2. how to perform SGD with basically one batch!

works but weird?



source

Solutions: Choosing batches for graphs

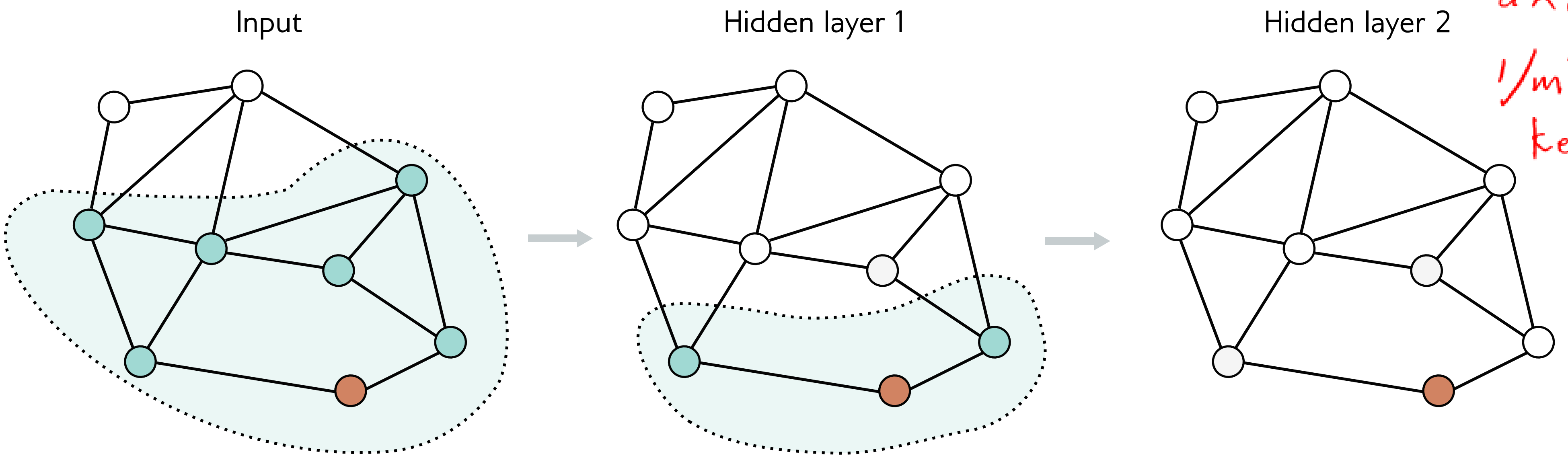
1. Choose random subset of nodes

2. Neighborhood sampling

3. Graph partitioning

Batches: Random subset

Just picking random connections has problems:
- if average(edges/node) ≤ 10 and sampling 1/million nodes, kept ~ 0 edges



You can pick a random batch of labeled nodes at each training step,

And only include them and their “k-hop neighborhoods”.

input → $H_1 \to H_2 \to \dots H_k$

sees 1 away

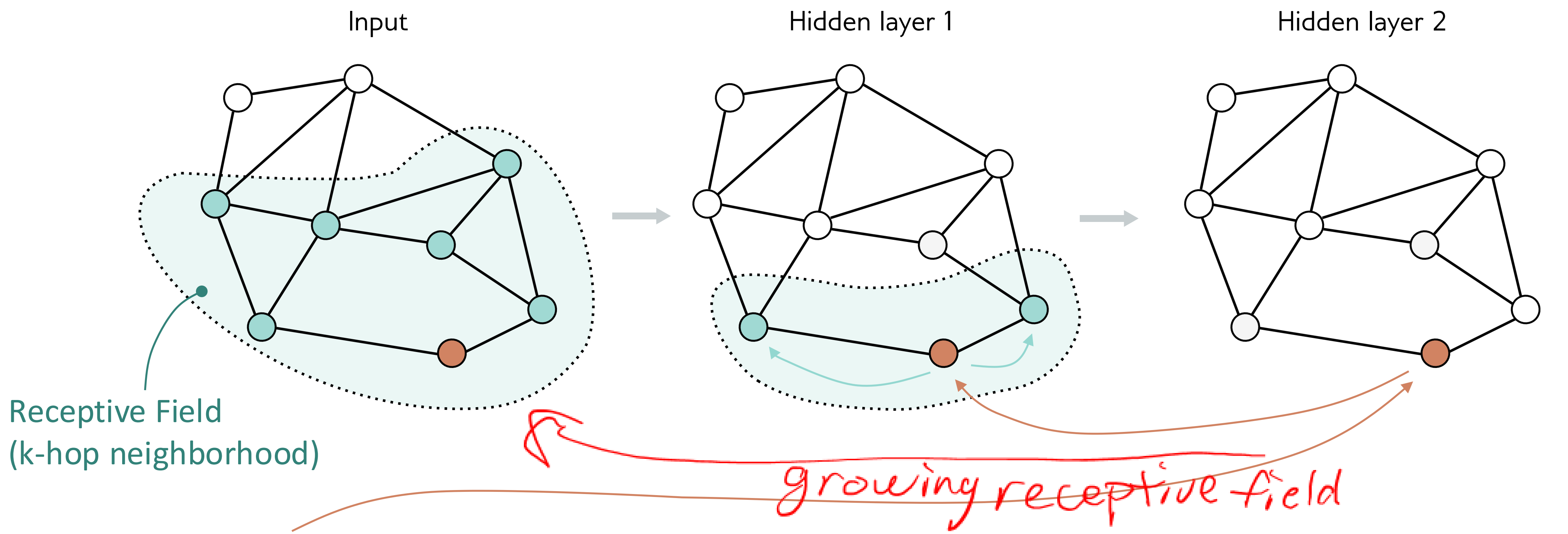
sees 2 away

sees k away

handles neighbor dropping.

k is from # of layers in GNN

Batches: Random subset

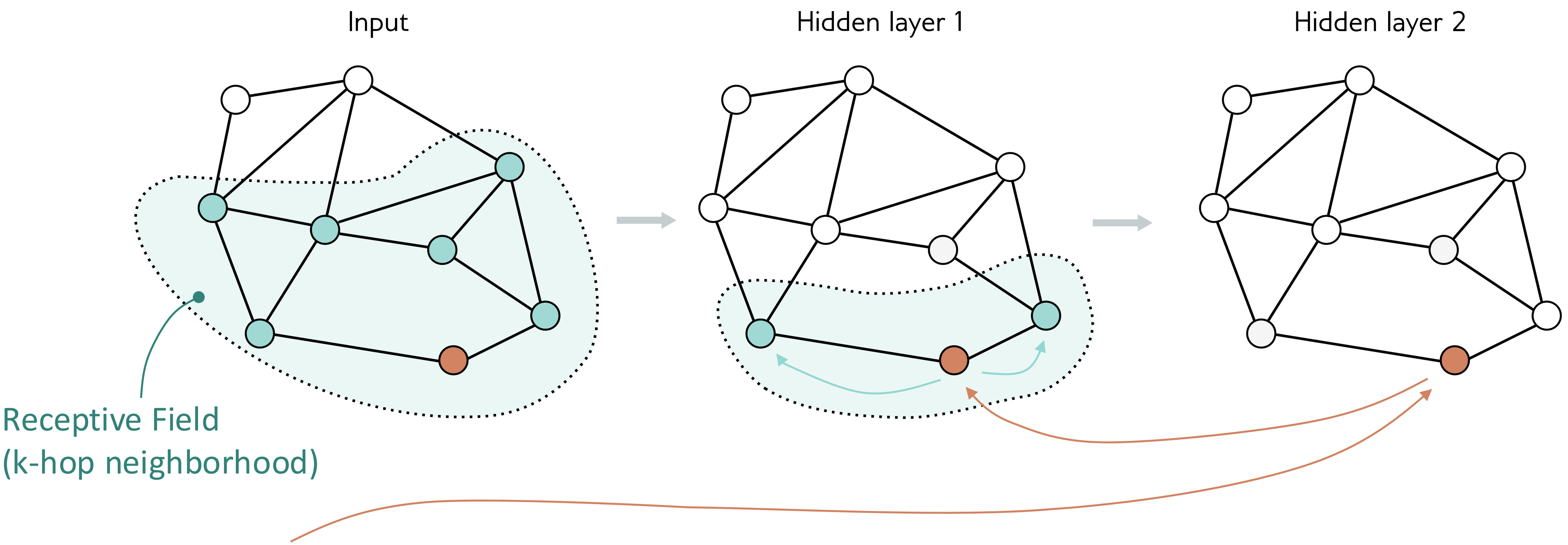


Receptive Field
(k-hop neighborhood)

Each node is dependent on the same node in the previous layer and its neighbors because of agg[]

$$\mathbf{h}_{k+1}^{(n)} = \mathbf{a}\left[\beta_k + \Omega_k \cdot \mathbf{h}_k^{(n)} + \Omega_k \cdot \mathrm{agg}[n,k]\right]$$

Batches: Random subset

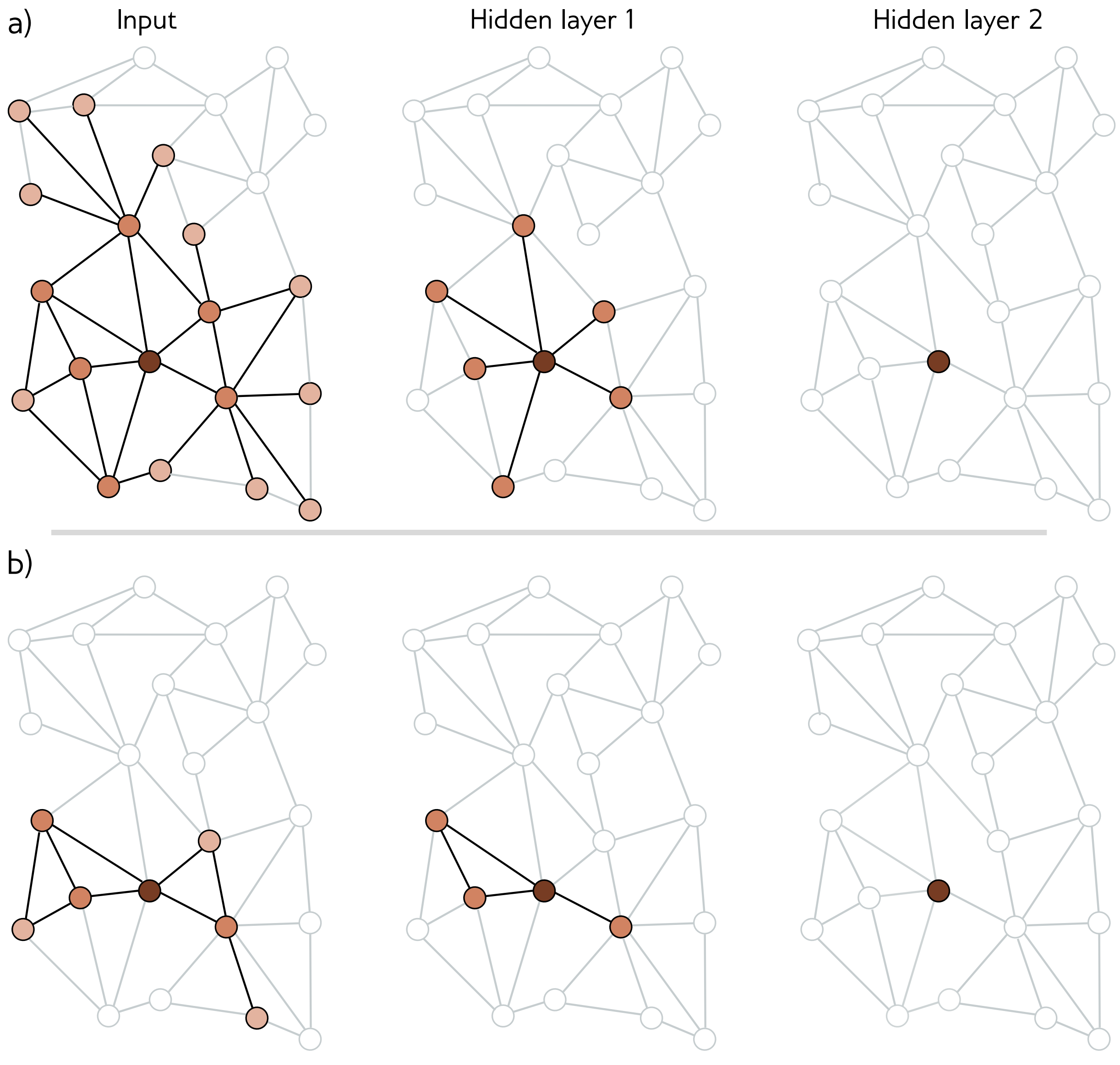


Each node is dependent on the same node in the previous layer and its neighbors because of agg[].

$$\mathbf{h}_{k+1}^{(n)} = \mathbf{a}\left[\beta_k + \Omega_k \cdot \mathbf{h}_k^{(n)} + \Omega_k \cdot \mathrm{agg}[n,k]\right]$$

With many layers and dense connection, it can quickly expand to encompass every node.

Neighborhood Sampling



Random Sampling:

Use all the neighbors

Neighborhood Sampling:

Use max n of the neighbors.

limit fan out

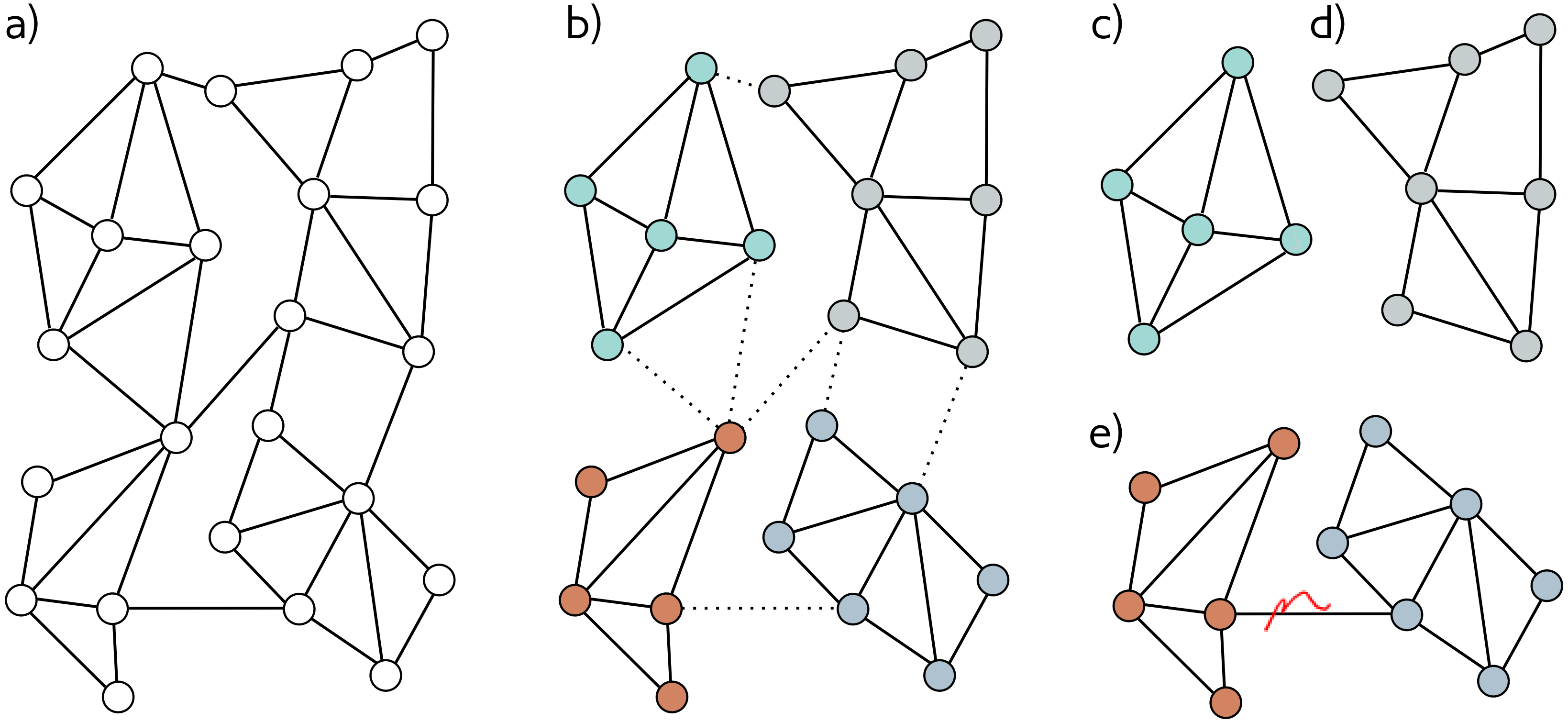
Here $n = 3$.

limits samplesize to ~n^k

See Notebook 13.3

Graph Partitioning

all internal edges survive.



Disconnect edges of the original to create maximally connected disjoint subsets

Split into train, test and validation sets and train just like in the inductive setting.

Alternatives to Mean Pooling for Node Combinations

- **Diagonal enhancement**: current node is multiplied by $(1+\epsilon_k)$, where ϵ_k is a learned scalar for each layer

$$\mathbf{H}_{k+1} = \mathbf{a}[\beta_k \mathbf{1}^T + \mathbf{\Omega}_k \mathbf{H}_k(\mathbf{A} + (1+\epsilon_k)\mathbf{I})]$$

- **Residual connections**: Include the current node in the sum

$$\mathbf{H}_{k+1} = \underline{\mathbf{a}[\beta_k \mathbf{1}^T + \mathbf{\Omega}_k \mathbf{H}_k \mathbf{A})]} + \mathbf{H}_k$$

- **Mean aggregation**: take average instead of sum of neighbors

$$\mathrm{agg}[n] = \frac{1}{|\mathrm{ne}[n]|} \sum_{m \in \mathrm{ne}[n]} \mathbf{h}_m$$

- **Kipf normalization**: downweight neighboring nodes with a lot of neighbors

$$\mathrm{agg}[n] = \sum_{m \in \mathrm{ne}[n]} \frac{h_m}{\sqrt{|\mathrm{ne}[n]|}|\mathrm{ne}[m]|}$$

- **Max pool aggregation**: element-wise max of all neighbors to current node

$$\mathrm{agg}[n] = \max_{m \in \mathrm{ne}[n]} [\mathbf{h}_m]$$

Aggregation by Attention

Weights depend on data at the nodes.

Apply linear transform to current node:

$$\mathbf{H}'_k = \boldsymbol{\beta}_k \mathbf{1}^T + \boldsymbol{\Omega}_k \mathbf{H}$$

Then the similarity s_{mn} of each transformed node embedding $\mathbf{h}'_m$ to the transformed node embedding $\mathbf{h}'_n$ is computed by concatenating the pairs, taking a dot product with a column vector ϕ_k of learned parameters, and applying an activation function:

$$s_{mn} = \text{a}\left[\phi_k^T \begin{bmatrix}\mathbf{h}'_m \\ \mathbf{h}'_n\end{bmatrix}\right]$$

two node vectors

dotproduct

new activation

learned parameters

$$\mathbf{H}_{k+1} = \mathbf{a}[\mathbf{H}'_k \cdot \text{Softma}\cancel{\text{s}}\text{k}[\mathbf{S}, \mathbf{A} + \mathbf{I}]$$

Softmask[S, A+I]

LLM's had Masked Self Attention to block ~~using~~ predicting token w/itself or future tokens.

The function Softmask[S, A+I]

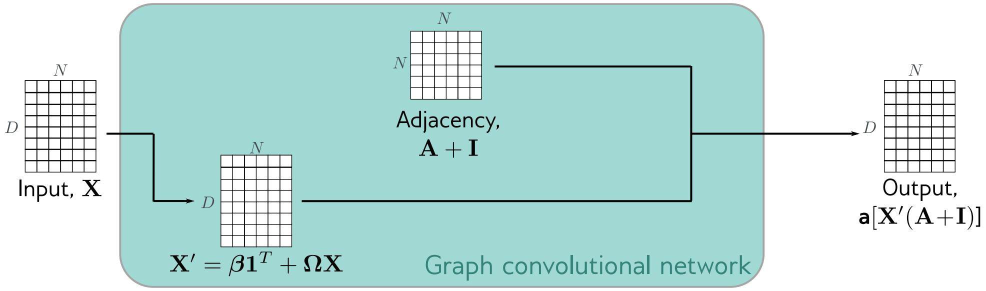
- computes the attention values by applying softmax operation separately to each column of its first argument S,
- but only after setting values where the second argument A + I is zero to negative infinity, so they do not contribute.
- This ensures that the attention to non-neighboring nodes is zero.

+non-self

to make softmax output be zero via $\frac{e^{-\infty}}{\Sigma \cdots} = 0$

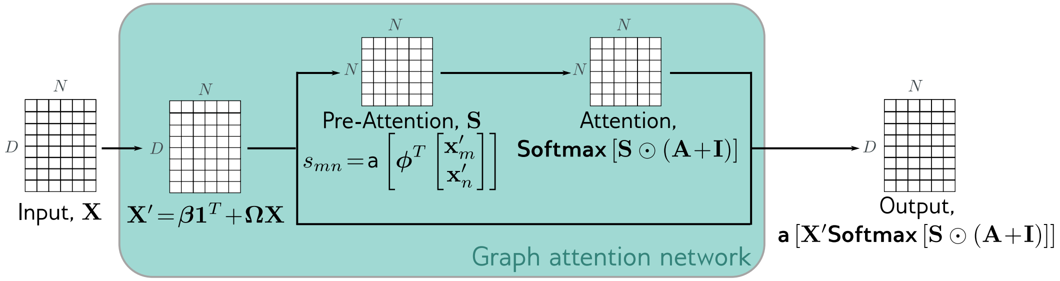
A makes neighbors non-zero
I makes self non-zero

Graph Attention


N
D
Input, $\mathbf{X}$
N
N
Adjacency,
$\mathbf{A}+\mathbf{I}$
N
D
$\mathbf{X}' = \beta\mathbf{1}^T + \mathbf{\Omega}\mathbf{X}$
Graph convolutional network
N
D
Output,
$\mathbf{a}[\mathbf{X}'(\mathbf{A}+\mathbf{I})]$

Regular graph convolution

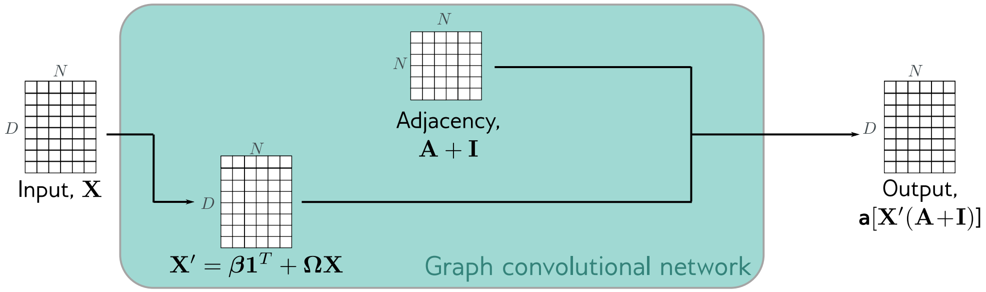
just adding together w/equal weight


N
D
Input, $\mathbf{X}$
N
D
$\mathbf{X}'=\beta\mathbf{1}^T+\mathbf{\Omega}\mathbf{X}$
N
N
Pre-Attention, $\mathbf{S}$
$s_{mn}=\mathbf{a}\left[\phi^T\begin{bmatrix}\mathbf{x}'_m\\\mathbf{x}'_n\end{bmatrix}\right]$
N
N
Attention,
$\mathbf{Softmax}\left[\mathbf{S}\odot(\mathbf{A}+\mathbf{I})\right]$
Graph attention network
N
D
Output,
$\mathbf{a}\left[\mathbf{X}'\mathbf{Softmax}\left[\mathbf{S}\odot(\mathbf{A}+\mathbf{I})\right]\right]$

Graph attention

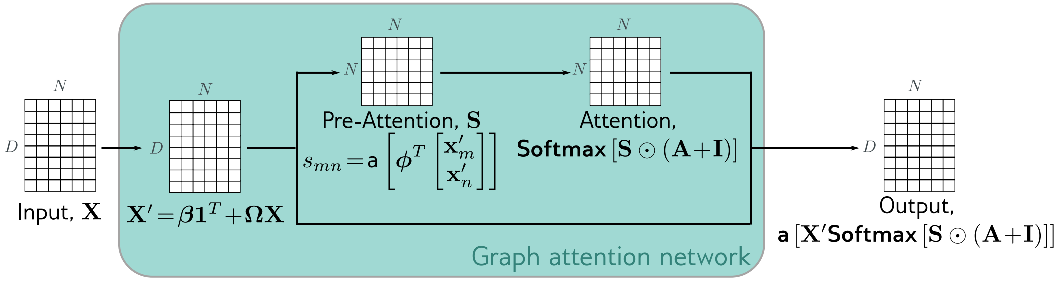
Attention weighted combination

Graph Attention



N
D
Input, $\mathbf{X}$
N
N
Adjacency,
$\mathbf{A}+\mathbf{I}$
N
D
$\mathbf{X}' = \boldsymbol{\beta}\mathbf{1}^T + \boldsymbol{\Omega}\mathbf{X}$
Graph convolutional network
N
D
Output,
$\mathbf{a}[\mathbf{X}'(\mathbf{A}+\mathbf{I})]$

Regular graph convolution

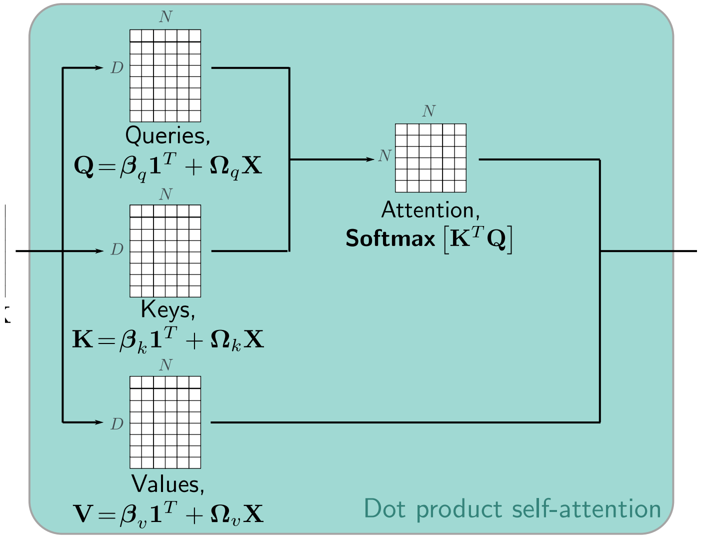


N
D
Input, $\mathbf{X}$
N
D
$\mathbf{X}' = \boldsymbol{\beta}\mathbf{1}^T + \boldsymbol{\Omega}\mathbf{X}$
N
N
Pre-Attention, $\mathbf{S}$
$s_{mn} = \mathbf{a}\left[\boldsymbol{\phi}^T \begin{bmatrix}\mathbf{x}'_m \\ \mathbf{x}'_n\end{bmatrix}\right]$
N
N
Attention,
Softmax $[\mathbf{S} \odot (\mathbf{A}+\mathbf{I})]$
Graph attention network
N
D
Output,
$\mathbf{a}\left[\mathbf{X}'\text{Softmax}\left[\mathbf{S} \odot (\mathbf{A}+\mathbf{I})\right]\right]$

Graph attention

Similar to Transformer Self Attention, *except*

- K, Q and V are all the same
- Different similarity measure
- Only attends to neighbors



N
D
Queries,
$\mathbf{Q} = \boldsymbol{\beta}_q\mathbf{1}^T + \boldsymbol{\Omega}_q\mathbf{X}$
N
D
Keys,
$\mathbf{K} = \boldsymbol{\beta}_k\mathbf{1}^T + \boldsymbol{\Omega}_k\mathbf{X}$
N
D
Values,
$\mathbf{V} = \boldsymbol{\beta}_v\mathbf{1}^T + \boldsymbol{\Omega}_v\mathbf{X}$
N
N
Attention,
Softmax $[\mathbf{K}^T\mathbf{Q}]$
Dot product self-attention

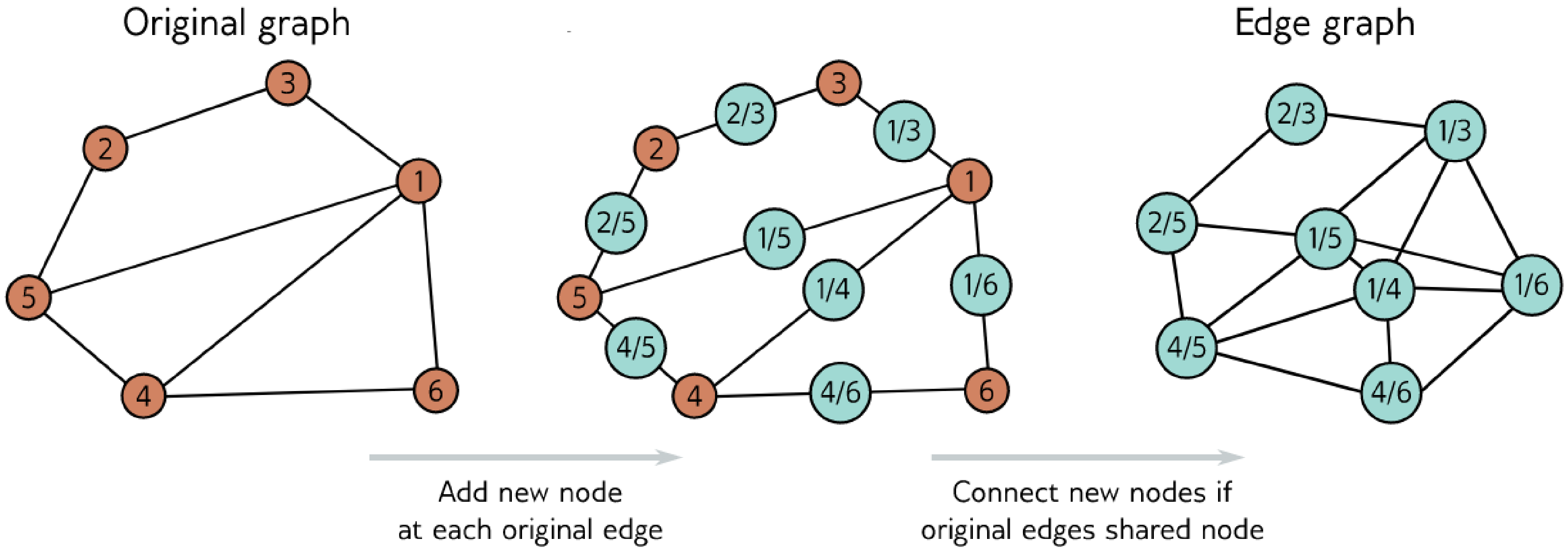
Any Questions?

???

Moving on

- Basic definition and examples
- Graph representation
- Properties of Adjacency Matrix
- Graph neural network, tasks and loss functions
- Graph convolutional network
- Graph & Node classification
- Edge graphs

Edge Graphs



Handled by simple transformation from node graphs.

Then process as node graph.

Transform back to edge graph.

Any Questions?

???

Moving on

- Basic definition and examples
- Graph representation
- Properties of Adjacency Matrix
- Graph neural network, tasks and loss functions
- Graph convolutional network
- Graph & Node classification
- Edge graphs