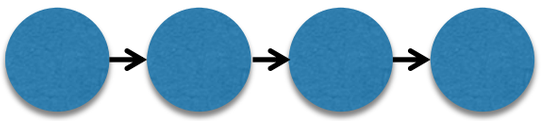
Deep Learning

LM Computer Science, Data Science, Cybersecurity
2nd semester - 6 CFU

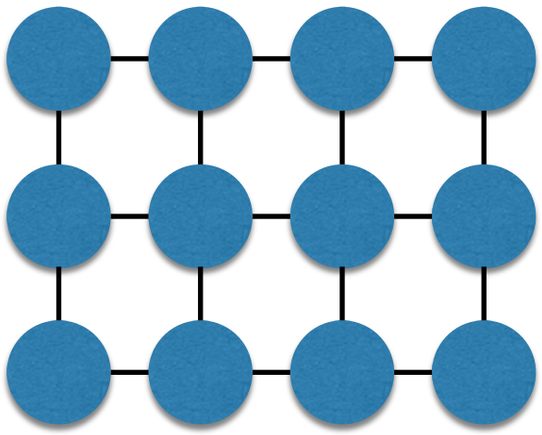
References:
Graph Neural Networks book chapter (on Moodle)

Graph Convolutional Networks

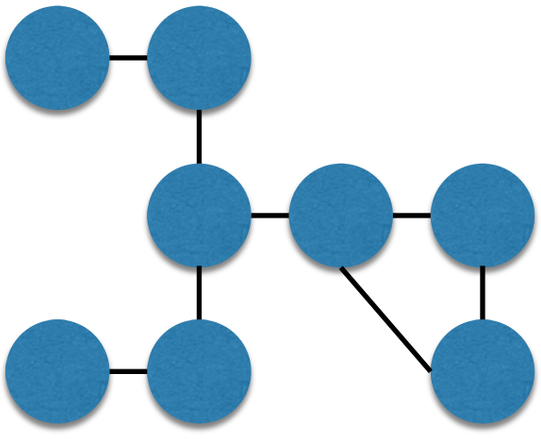
Graphs as irregular grids



Sequences



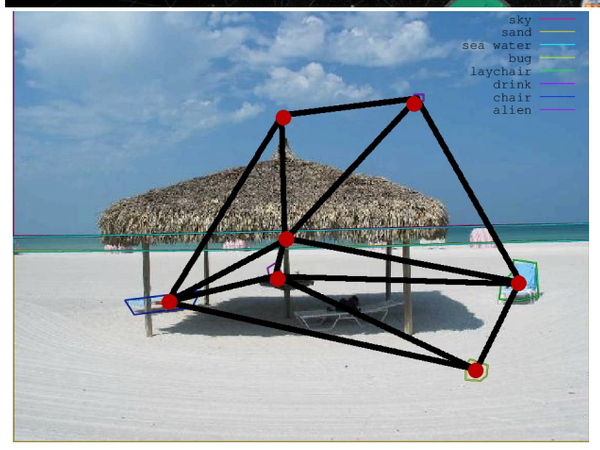
Images



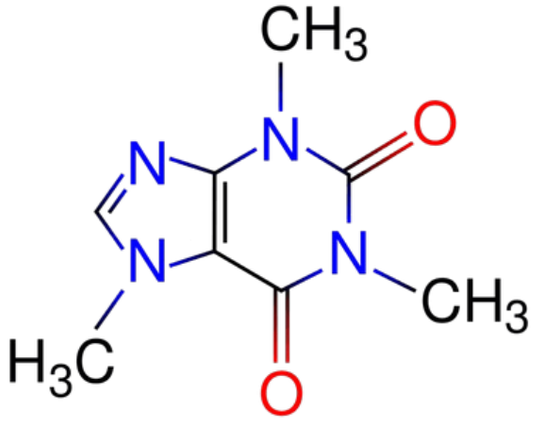
Graphs

Graphs are everywhere..

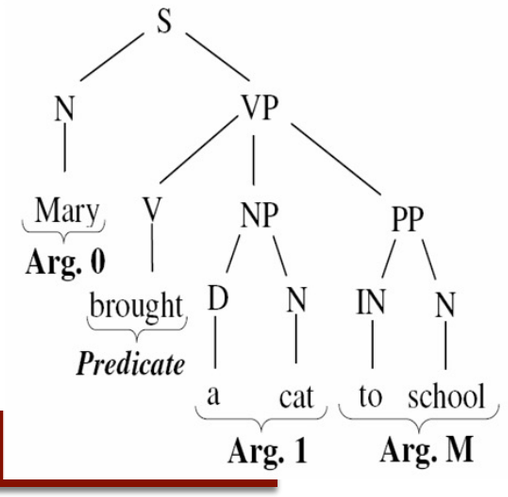
- In several settings it is natural to represent data as graphs.
- We may want to predict some property:
  - over Graphs or
  - over Graph Nodes



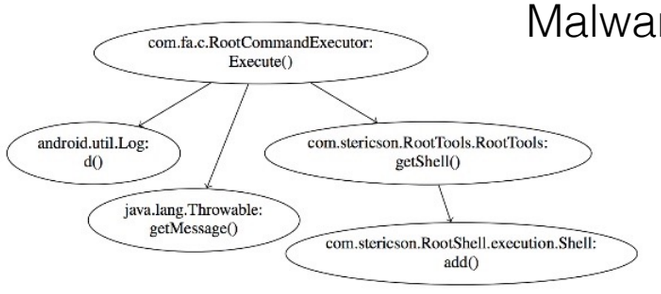
Chemical compounds



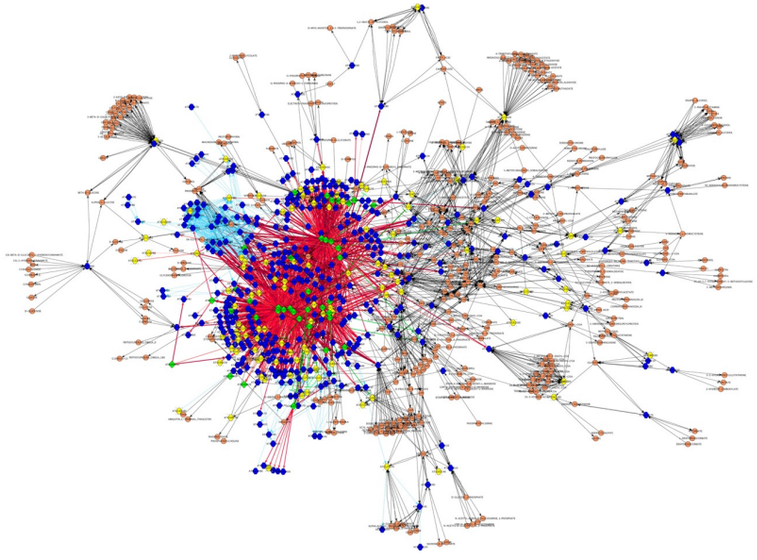
Parse trees



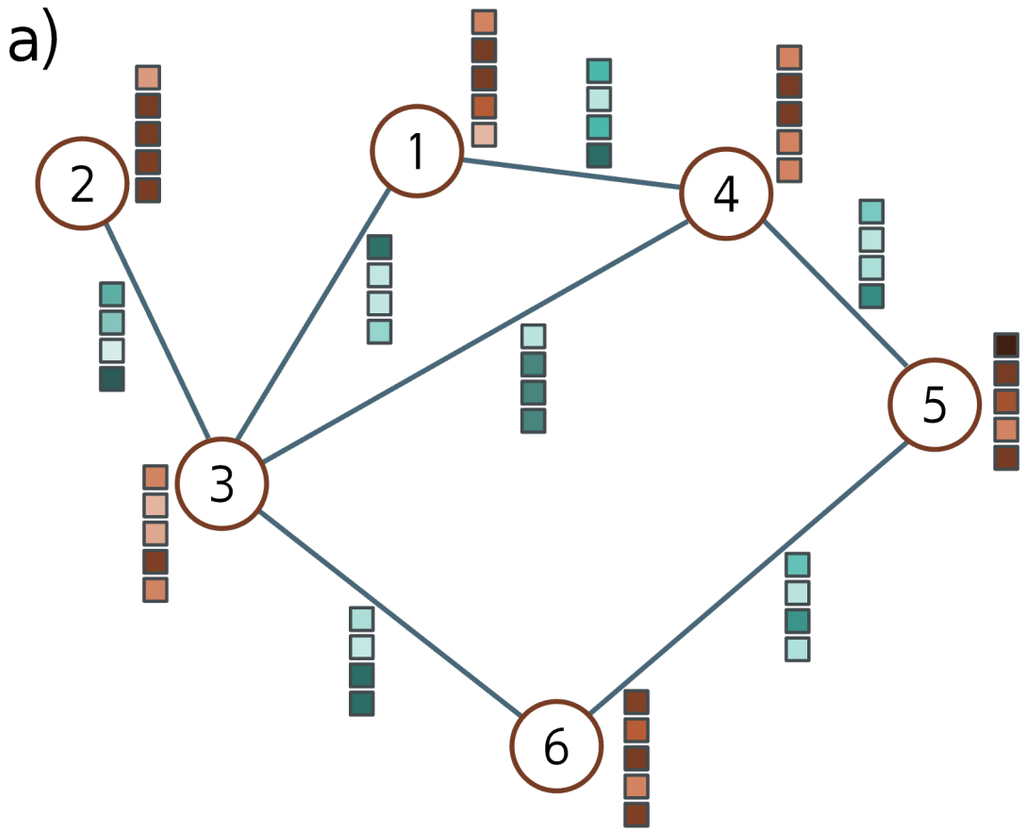
Malware API calls

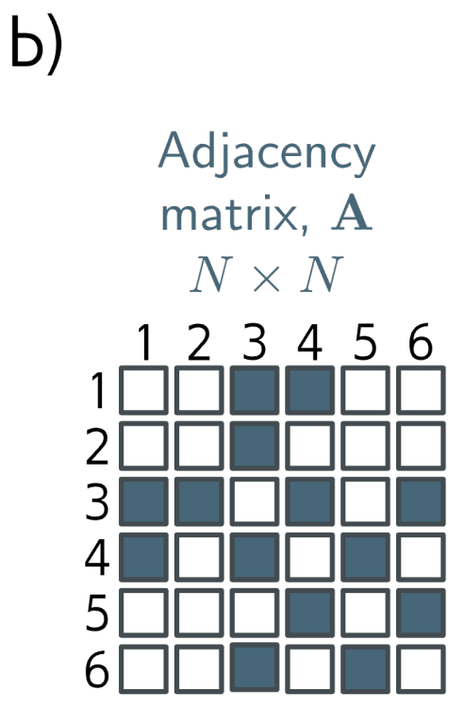


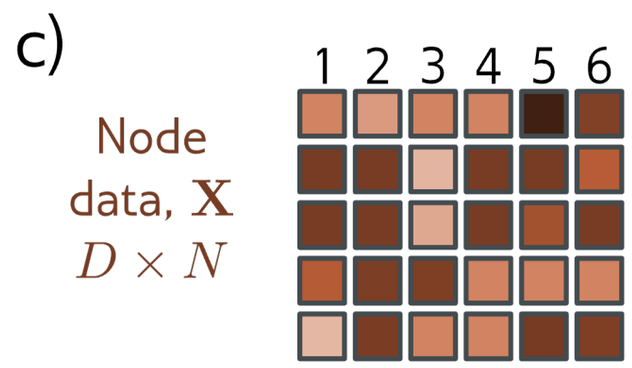
Gene networks

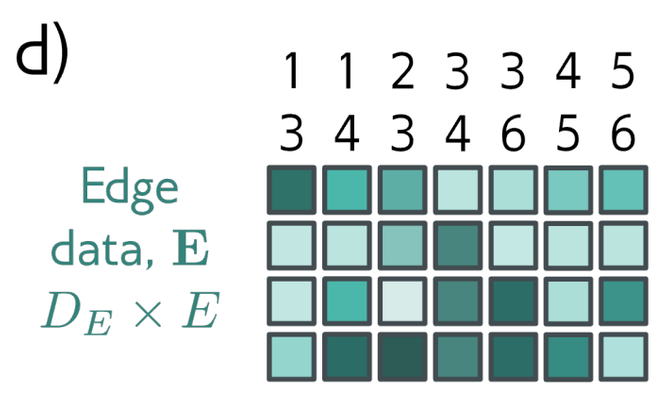


Representing Graphs



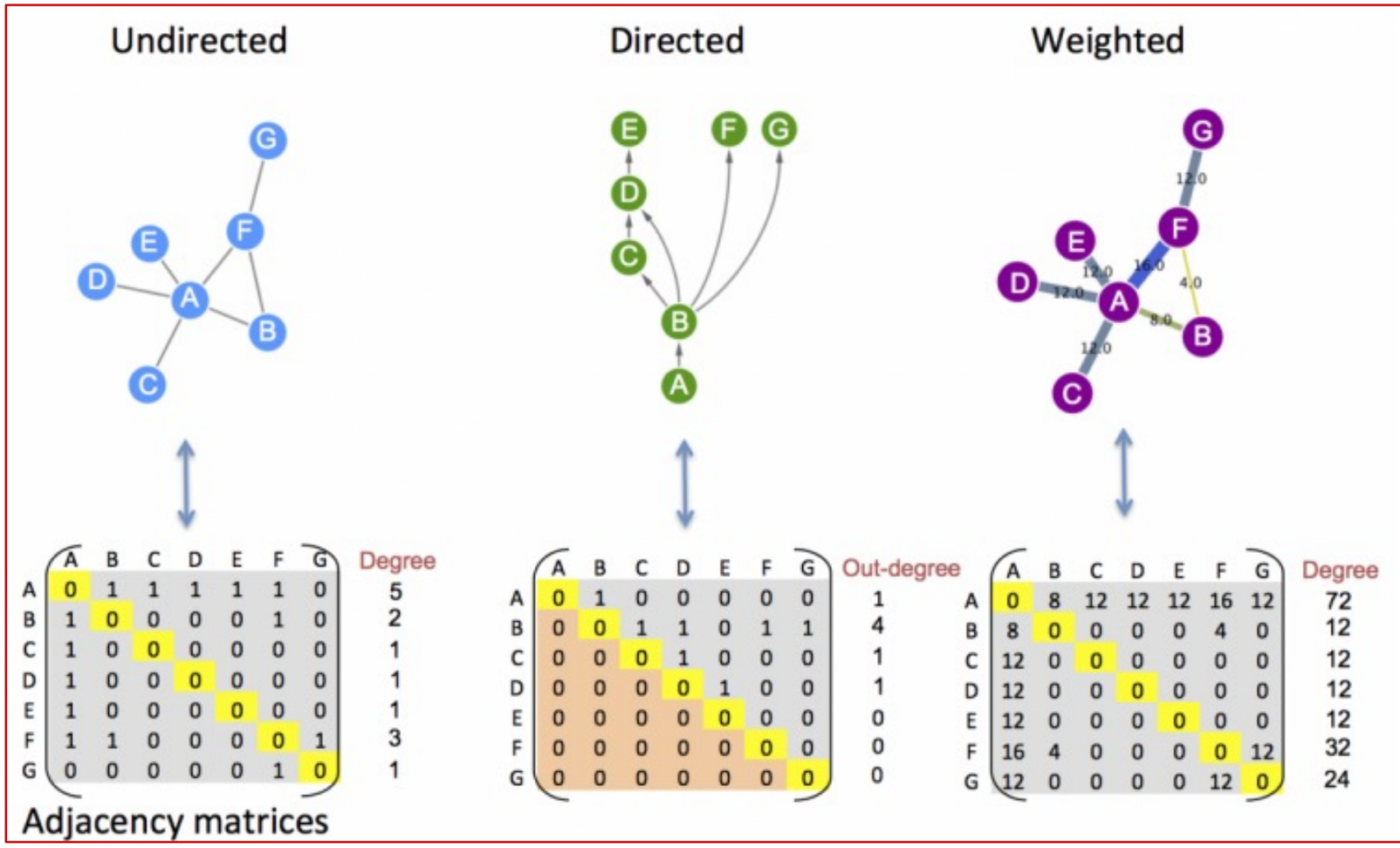






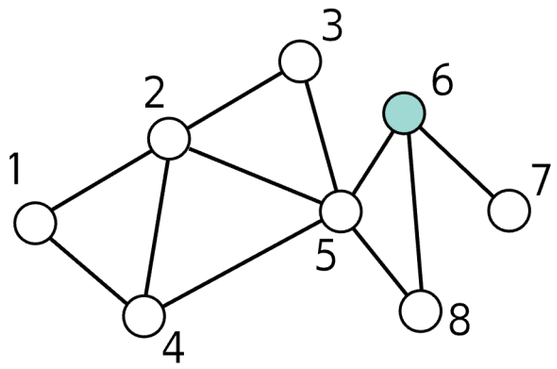
Adjacency matrix

- Shortest paths
- Random walks



Exponentiating the adjacency matrix

a)



b)

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

c)

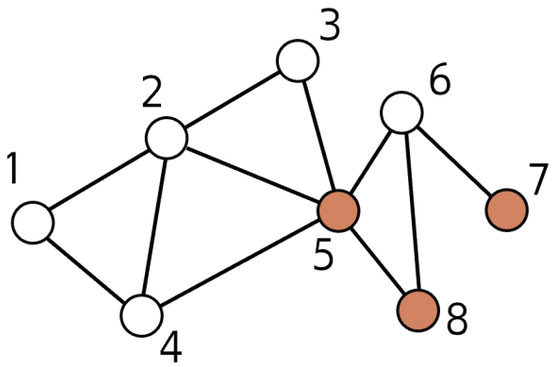
$$\mathbf{A}^2 = \begin{bmatrix} 2 & 1 & 1 & 1 & 2 & 0 & 0 & 0 \\ 1 & 4 & 1 & 2 & 2 & 1 & 0 & 1 \\ 1 & 1 & 2 & 2 & 1 & 1 & 0 & 1 \\ 1 & 2 & 2 & 3 & 1 & 1 & 0 & 1 \\ 2 & 2 & 1 & 1 & 5 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 3 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 2 \end{bmatrix}$$

d)

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

e)

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



f)

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

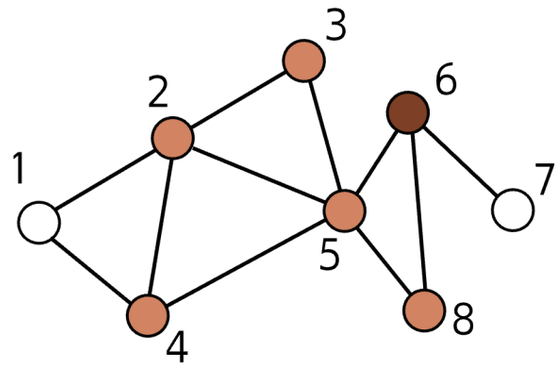


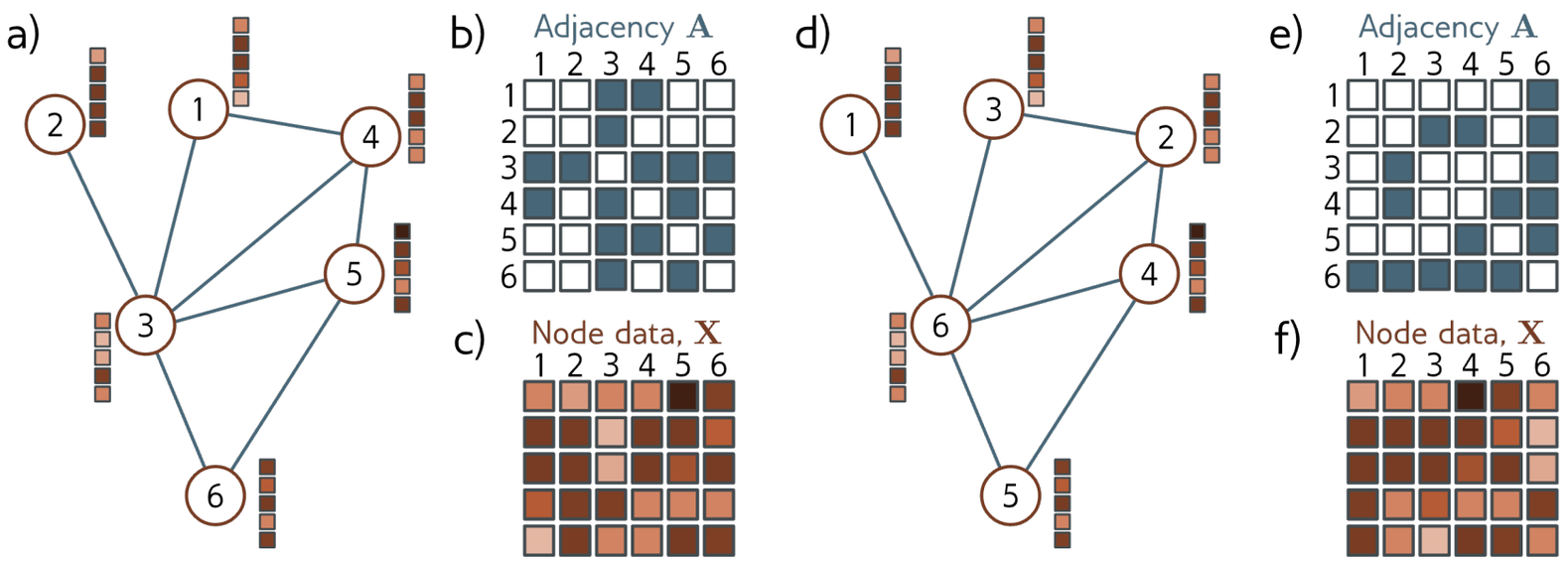
Figure 13.4 Properties of the adjacency matrix. a) Example graph. b) Position (m, n) of the adjacency matrix $\mathbf{A}$ contains the number of walks of length one from node m to node n. c) Position (m, n) of the squared adjacency matrix $\mathbf{A}^2$ contains the number of walks of length two from node m to node n. d) One hot vector representing node six, which was highlighted in panel (a). e) When we pre-multiply this vector by $\mathbf{A}$, the result contains the number of walks of length one from node six to each node; we can reach nodes five, seven, and eight in one move. f) When we pre-multiply this vector by $\mathbf{A}^2$, the resulting vector contains the number of walks of length two from node six to each node; we can reach nodes two, three, four, five, and eight in two moves, and we can return to the original node in three different ways (via nodes five, seven, and eight).

Permutation Invariance/Equivariance

Node indexing in graphs is arbitrary:

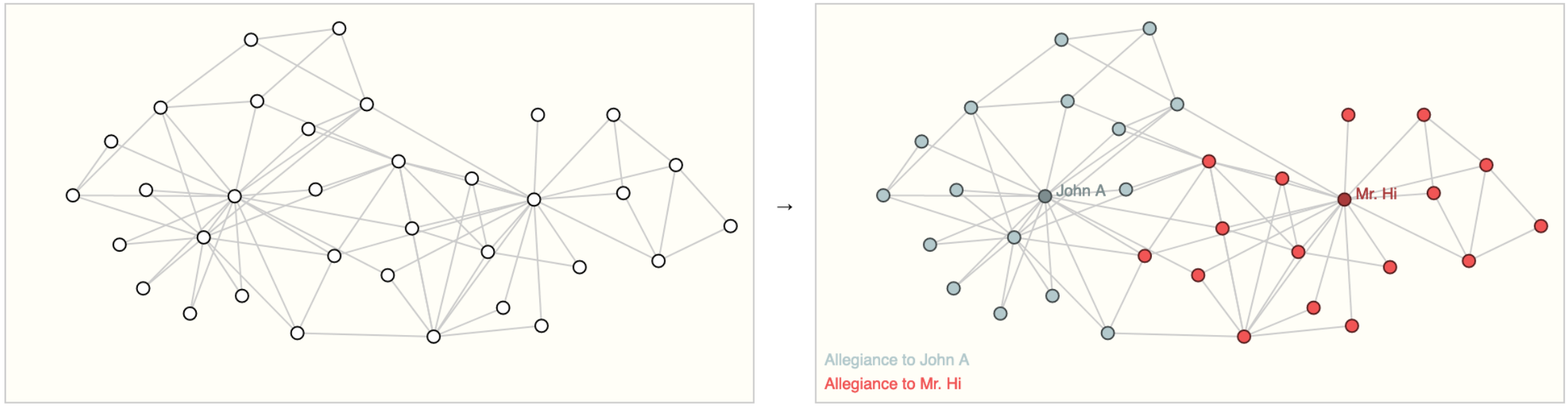
- Differently from e.g. images, permuting the node indices results in a permutation of the columns of X and a permutation of both the rows and columns of A. However, the underlying graph is unchanged.
- Given a permutation matrix $\boldsymbol{P}$, we get a different representation of the same graph

$$\boldsymbol{X' = XP}$$
$$\boldsymbol{A' = P^T AP}$$



Node-level Classification

- Dataset: a single huge graph (e.g. social network):
  - n vertices, e edges
  - d attributes associated with nodes: $x_v \in \mathbb{R}^d$
  - Target y_v associated to a small subset of nodes
- Given an unseen node u , the task is to predict the correct target y_u

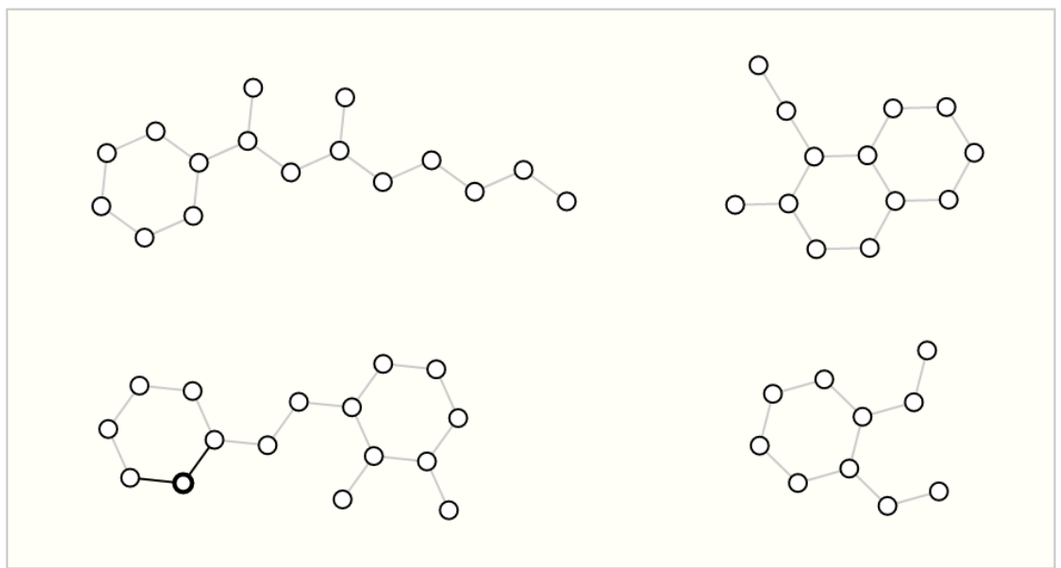


Input: graph with unlabled nodes

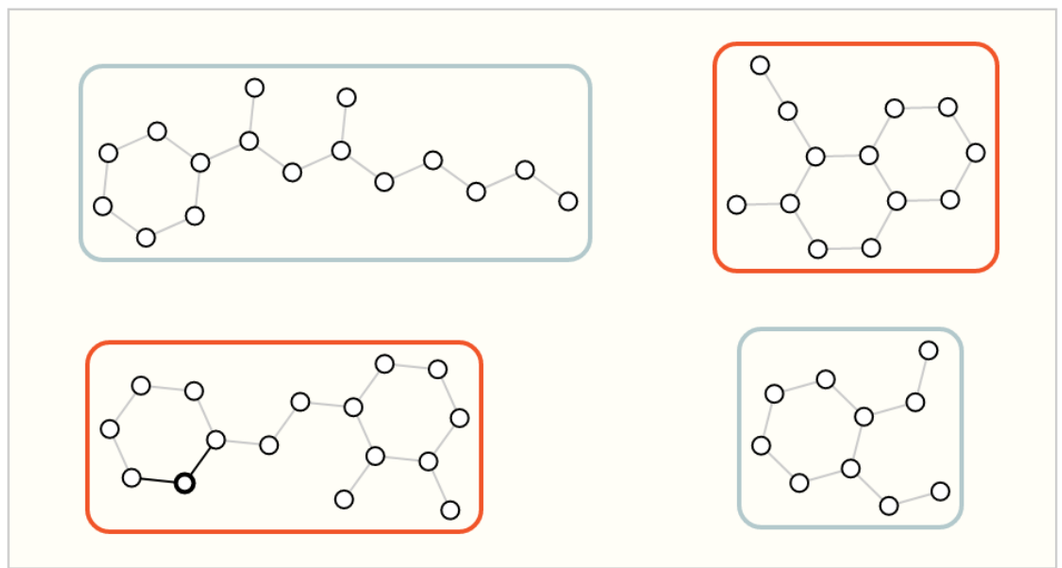
Output: graph node labels

Graph-level Classification

- Dataset composed of N pairs $\{(G_i, y_i), 1 \leq i \leq N\}$
  - Each graph:
    - n_i vertices, e_i edges
    - label associated to each graph: $\mathrm{y_i}$
    - d vectorial attributes associated to each node: $x_v \in \mathbb{R}^d$, $X \in \mathbb{R}^{n_i \times d}$
- Given an unseen graph G, the task is to predict the correct target label



Input: graphs

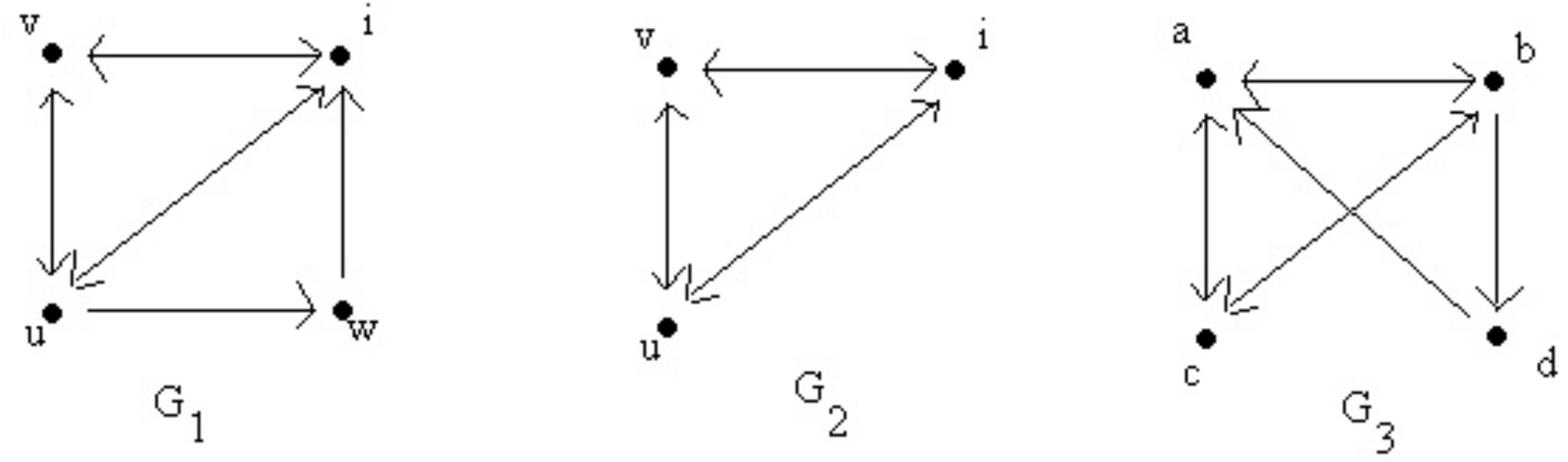


Output: labels for each graph, (e.g., "does the graph contain two rings?")

Learning on Graphs is Difficult

Definition: Isomorphism

A graph $G_1 = (V_1, E_1)$ is isomorphic to $G_2 = (V_2, E_2)$ if there exists a bijective mapping $f : V_1 \times V_2$ such that $\forall (v_1, v_1') \in E_1 \Leftrightarrow (f(v_1), f(v_1')) \in E_2$
$(\forall v \in V_1,\ L_{G_1}(v) = L_{G_2}(f(v)))$

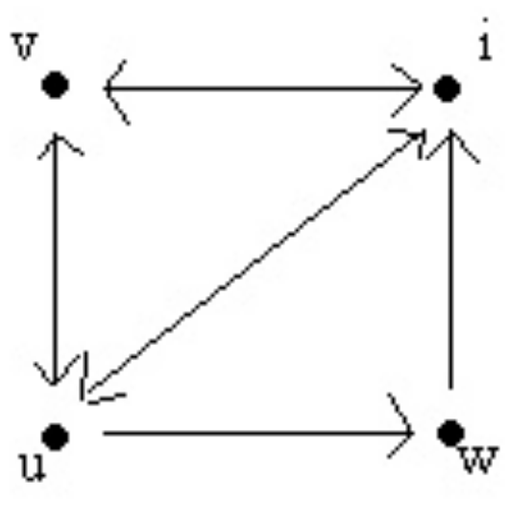


G_1 and G_3 are isomorphic: $f(i) = a,\ f(u) = b,\ f(v) = c,\ f(w) = d$

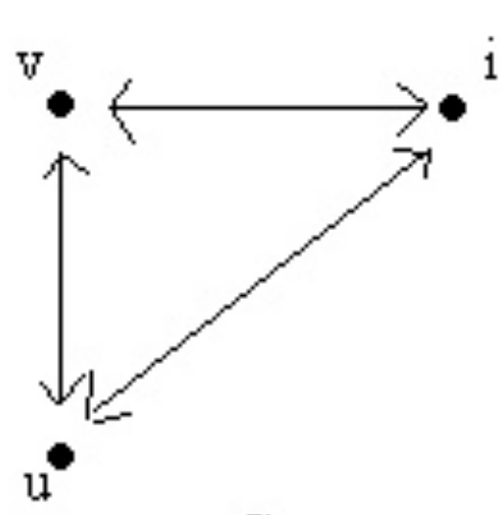
Learning on Graphs is Difficult

Definition: Subgraph

A subgraph $G_2 = (V_2, E_2)$ of $G_1 = (V_1, E_1)$ is a graph for which $V_2 \subseteq V_1$, $E_2 \subseteq E_1 \cap (V_2 \times V_2)$.



G_1



G_2

Learning on Graphs is Difficult

Two general problems

- Same graph can be represented in different ways ...
  (any graph learning algorithm has to consider that!)
- How to recognize that a given graph G_2 is a subgraph of G_1 ?
  (learning makes sense only if the similarity between two graphs can be assessed ...)

- It is not known whether Graph Isomorphism is NP-Complete
  (no polynomial-time algorithm is known, neither is it known to be NP-Complete)
- Subgraph Isomorphism is NP-Complete !!
  ... runtime may grow exponentially with the number of nodes

Learning on Graphs is Difficult

Two specific problems for Neural Networks

- **Problem 1:** How to represent graphs of different sizes (i.e., different number of nodes) into fixed-size vectors ?
  (. . . without loosing expressiveness, i.e., different graphs should get different representations)
- **Problem 2:** How to avoid explosion in the number of parameters with the size of the graphs?
  (number of parameters should not grow too much with the size of the graphs . . .)

- Problem 1 is commonly faced by using recursive models that exploit a causal state space [Sperduti & Starita., TNN 1997]
- Problem 2 is commonly faced by exploiting shared parameters

Graph Neural Networks -1

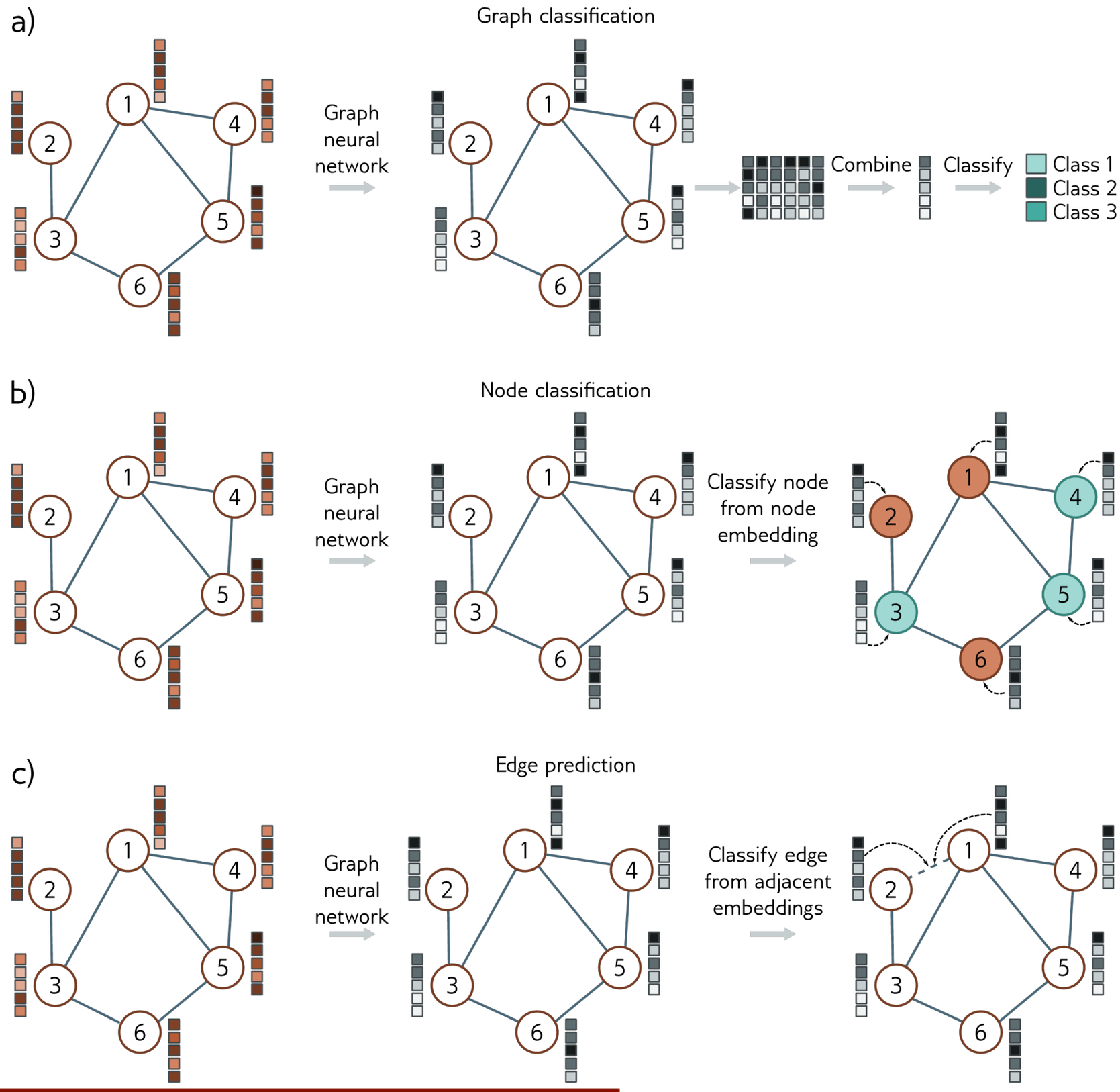
Idea: network architecture receives in input a graph (adj. matrix and node repr. for simplicity) and passes it through a series of k layers

- **each layer** computes a hidden representation for each node
- the **last layer** computes the final node embeddings $\boldsymbol{H}_k$
- similarly to CNNs, each node representation includes information about the node and its context within the graph
- Node-level tasks: output computed from $\boldsymbol{H}_k$
- GNNs are *permutation equivariant* by construction

Graph Neural Networks -2

- 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
  - the resulting network will be permutation invariant
- Link prediction tasks: the network predicts whether or not there should be an edge between nodes
  - This is a binary classification task where the two node embeddings must be mapped to a single number representing the probability that the edge is present.
  - E.g. dot product of the node embeddings and pass the result through a sigmoid function to create a probability

Graph Neural Networks



Convolution operator (on images)

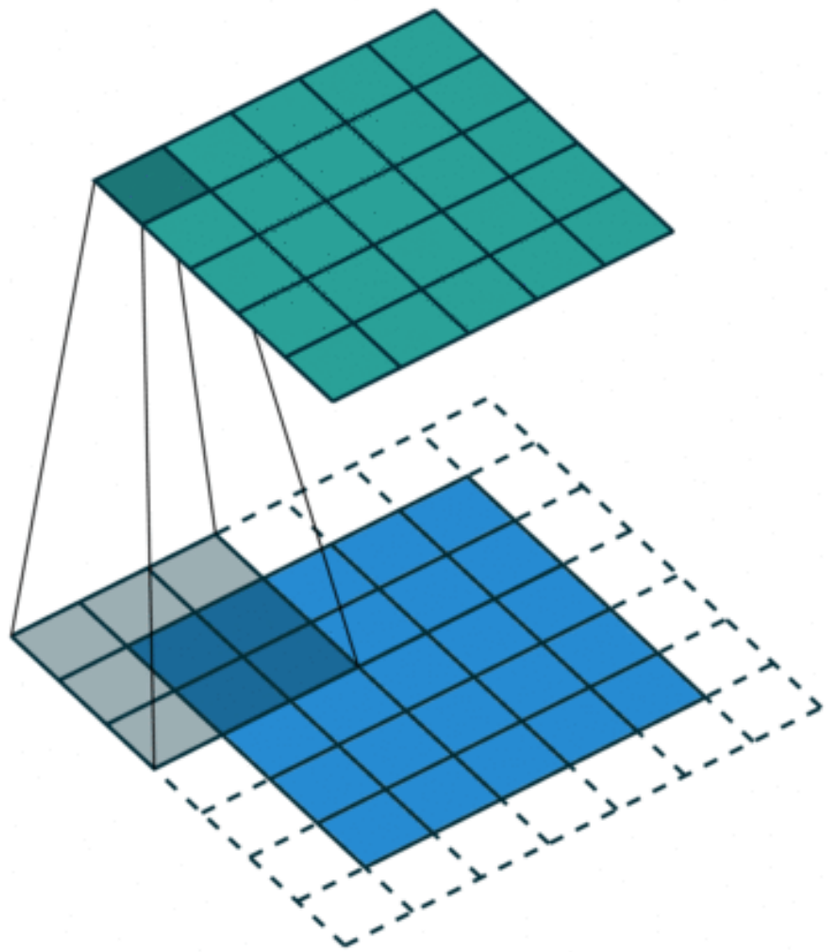
General definition for filter f and signal x:

$$(f * x)(t) = \int_{-\infty}^{\infty} f(\tau)x(t-\tau)\,d\tau$$

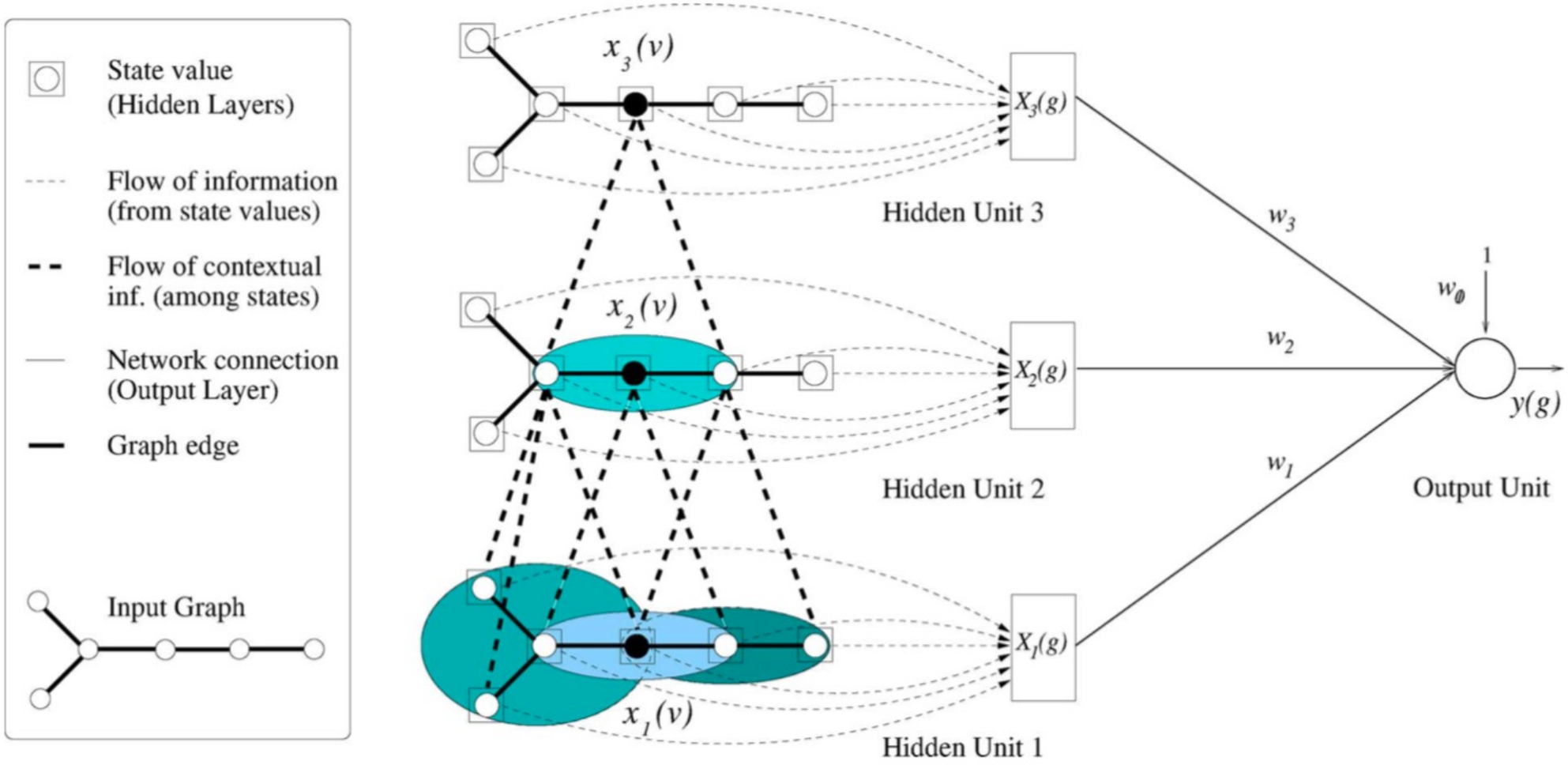
In images, it corresponds to

$$(f * x)\,(i,j) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} f(s,t)x(i-s,j-t)$$

Where f is a $2a \times 2b$ filter and x an image



NN4G by Micheli



- Each convolution takes as input the representation of all previous layers
- Trained layer-wise (cascade correlation)
- Readout: a representation per-graph per-layer is computed using the sum

(Simplified) NN4G formulation

- The representation learned for a node after layer i

$$\boldsymbol{h}_v^1 = \sigma(\bar{W}^1 \boldsymbol{x}_v)$$

$$\boldsymbol{h}_v^i = \sigma\left(\bar{W}^i \boldsymbol{x}_v + W^i \sum_{u \in N(v)} \boldsymbol{h}_u^{i-1}\right), i > 1$$

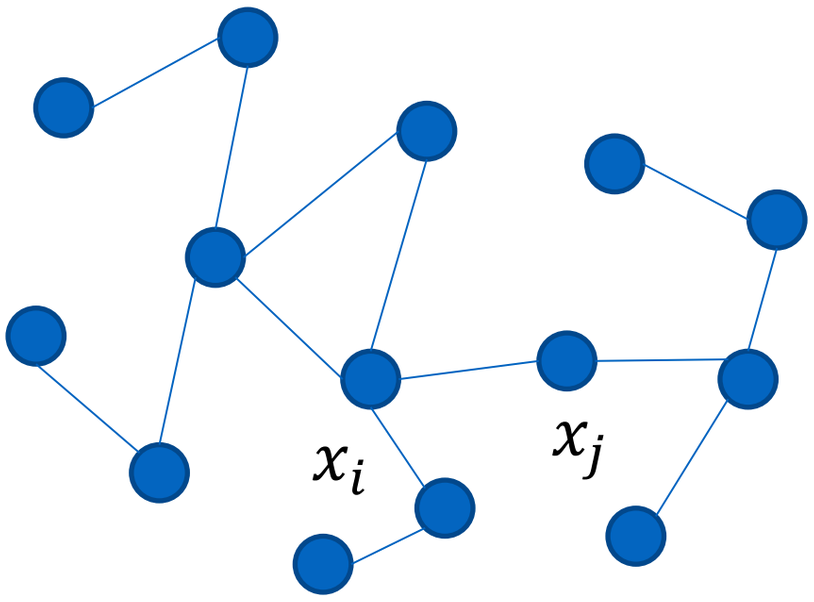
- Actual formulation with skip-connections (see GNN book chapter)

- Equivalent formulation

$$H^1 = \sigma(\bar{W}^1 X)$$

$$H^i = \sigma\left(\bar{W}^i X + W^i A H\right), i > 1$$

How to define convolution on graphs?



Consider a simple setting:

- single undirected graph
- x: $V \rightarrow \mathbb{R}$: a signal on the nodes of a graph
  - Represented as vector $x \in \mathbb{R}^n$

The convolution operator is difficult to define in the vertex domain

<u>Convolution Theorem:</u>

- Convolution in one domain (time, space)
- corresponds to pointwise multiplication in frequency domain

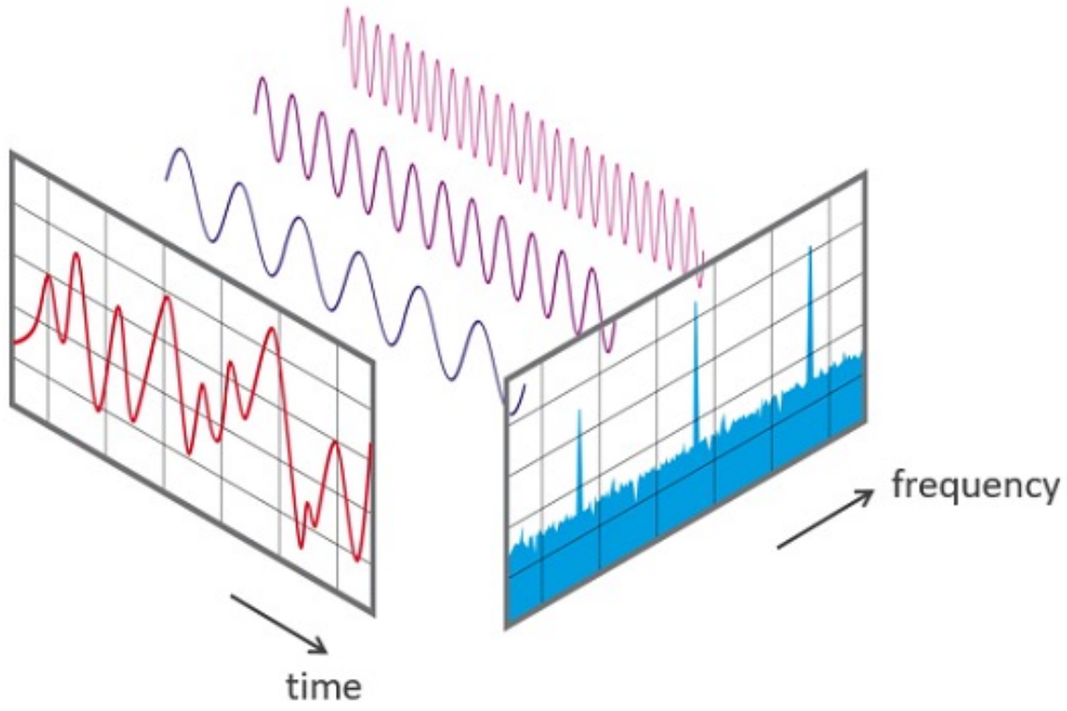
$$\widehat{f * g} = \hat{f} \odot \hat{g}$$

$\hat{f}$: Fourier transform of f

- $\odot$ Hadamard (element-wise) product

Fourier Transform

”An arbitrary function, continuous or with discontinuities, defined in a finite interval by an arbitrarily capricious graph can always be expressed as a sum of sinusoids“.
Jean Joseph Fourier (1768—1830)



In many classical settings of Fourier analysis, the Fourier transform can be defined in terms of the eigenvalues and eigenvectors of the Laplace operator.

Graph Fourier Transform

- Graph (normalized) Laplacian

$$L = I_n - D^{-1/2} A D^{-1/2}$$

$$L = U \Lambda U^T$$

- where $\Lambda = diag([\lambda_0, \dots, \lambda_{n-1}])$ and U is the Fourier basis of the graph
- Given a spatial signal x
  - $\hat{x} = U^T x$ is its graph Fourier Transform
  - $x = U \hat{x}$ is the inverse Fourier transform

Graph Convolution

- Convolution operator in the **node** domain has a correspondence in the Graph **frequency** domain ($\odot$ Hadamard product)

$$\boldsymbol{x}_1 *_G \boldsymbol{x}_2 = U\big((U^T\boldsymbol{x}_1) \odot (U^T\boldsymbol{x}_2)\big)$$

- **Convolution** between a **parametric filter** and a **signal**:

$$\text{y}=\boldsymbol{f}_\theta *_G \boldsymbol{x} = U\big((U^T\boldsymbol{f}_\theta) \odot (U^T\boldsymbol{x})\big)$$

- Easier to define the filter in spectral domain
- Inverse Fourier transform of f_θ

$$\boldsymbol{f}_\theta = U\widehat{\boldsymbol{f}_\theta}$$

- N.B. $U^T U = I$

$$\mathbf{y} = \boldsymbol{f}_\theta *_G \boldsymbol{x} = U\big(\widehat{\boldsymbol{f}_\theta} \odot U^T\,\boldsymbol{x}\,\big)$$

Graph Convolution

$$\mathbf{y} = \boldsymbol{f}_\theta *_G \boldsymbol{x} = U(\widehat{\boldsymbol{f}_\theta} \odot U^T \boldsymbol{x})$$

- Hadamard product -> Matrix multiplication

$$\mathbf{y} = \boldsymbol{f}_\theta *_G \boldsymbol{x} = U\hat{F}_\theta U^T \boldsymbol{x}$$

$$\boldsymbol{a} \odot \boldsymbol{b} = A\boldsymbol{b}$$
where $A = diag(\boldsymbol{a})$

We can design the diagonal $\hat{F}_\theta$ in several ways

- Non-parametric filters: $\widehat{F}_\theta = diag(\theta)$
  - Not localized
  - n parameters
- Polynomial filters: $\widehat{F}_\theta = \sum_{k=0}^{K} \theta_k \Lambda^k$
  - exactly K-localized
  - K parameters
  - We can learn directly in the graph domain $g_\theta(L) = \sum_{k=0}^{K} \theta_k L^k$

$$\mathbf{y} = \boldsymbol{f}_\theta *_G \boldsymbol{x} = U\hat{F}_\theta U^T \boldsymbol{x} = \sum_{k=0}^{K} \theta_k U\Lambda^k U^T \boldsymbol{x} = \sum_{k=0}^{K} \theta_k L^k \boldsymbol{x}$$

Graph Convolution

- Chebyshev polynomials:

$$T_0(x) = 1,\ T_1(x) = x,\ \ T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)$$

- Chebyshev K-localized polynomial filter (evaluated in [-1,1]):
  - Filter in Freq domain: (where the scaled $\tilde{\Lambda} = \frac{2}{\lambda_{max}}\Lambda - I_n$)

$$\widehat{F}_\theta = \sum_{k=0}^{K} \theta_k T_k(\tilde{\Lambda})$$

  - Convolution in node domain:

$$\boldsymbol{f}_\theta *_G \boldsymbol{x} = \sum_{k=0}^{K} \theta_k T_k(\tilde{L})x \qquad \text{where } \tilde{L} = \frac{2}{\lambda_{max}}L - I_n)$$

- If we limit $K = 1$ and assume $\lambda_{max} \approx 2$

$$\boldsymbol{f}_\theta *_G \boldsymbol{x} \approx \theta_0\boldsymbol{x} - \theta_1 D^{-1/2}AD^{-1/2}\boldsymbol{x}$$

- Forcing $\theta = \theta_0 = -\theta_1$

$$\boldsymbol{f}_\theta *_G \boldsymbol{x} \approx \theta\left(I_n + D^{-1/2}AD^{-1/2}\right)\boldsymbol{x}$$

- *renormalization trick*:
  - $I_n + D^{-1/2}AD^{1/2} \rightarrow \check{D}^{-1/2}\check{A}\check{D}^{-1/2}$ with $\check{A} = A + I_n$
- For d input and F output channels: ($X \in \mathbb{R}^{n\times d}$ and $\Theta \in \mathbb{R}^{d\times F}$)

$$Y = \check{D}^{-1/2}\check{A}\check{D}^{-1/2}X\Theta$$

- Finally, stack multiple layers and introduce nonlinearity

Graph Convolution - Summary

Main steps:

- Graph Fourier Transform
  - Fourier Basis are eigenvectors of normalized Graph Laplacian

$$L = U\Lambda U^T$$

- We can then define the graph convolution in the frequency domain

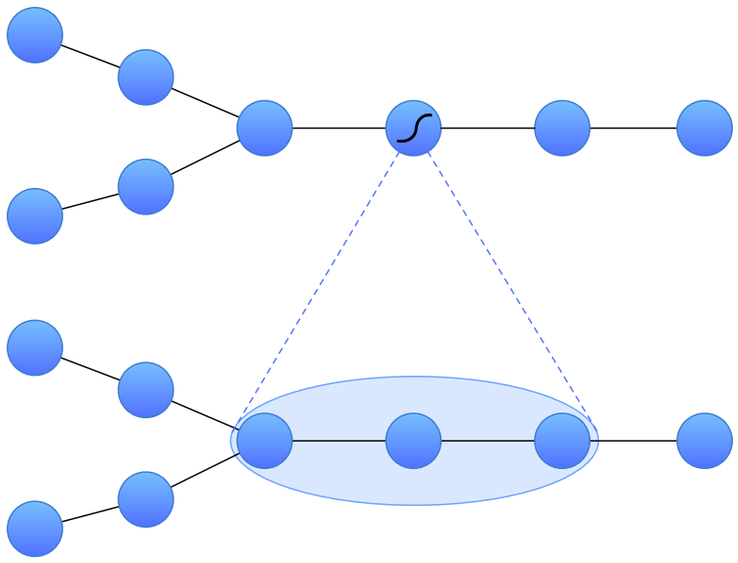
$$f *_G x = U\hat{F}U^T x$$

where $\hat{F} = diag(\hat{f})$

- For some choice of filters, e.g. polynomials of the spectral matrix
  - the convolution can be computed in the node space directly

$$\hat{F}_\Theta = \sum_{k=0}^{K} \theta_k \Lambda^k \rightarrow f *_G x = \sum_{k=0}^{K} \theta_k L^k x$$

Graph Convolution - Summary



- **1-localized** GCN maps multisets of representations (node and neighbours at the previous layer) to a new one:

$$\boldsymbol{h}_v^{l+1} = f\left(\{\boldsymbol{h}_v^l, \boldsymbol{h}_u^l, \forall u \in ne(v)\}\right)$$

  where $H^0 = X$

- Where f is a linear mapping & non-linear activation function, e.g.

$$H^{l+1} = \sigma\left(\breve{D}^{-1/2}\breve{A}\breve{D}^{-1/2}H^l\Theta^l\right)$$

The convolution operator can be generalized to be more expressive than 1-WL [6]

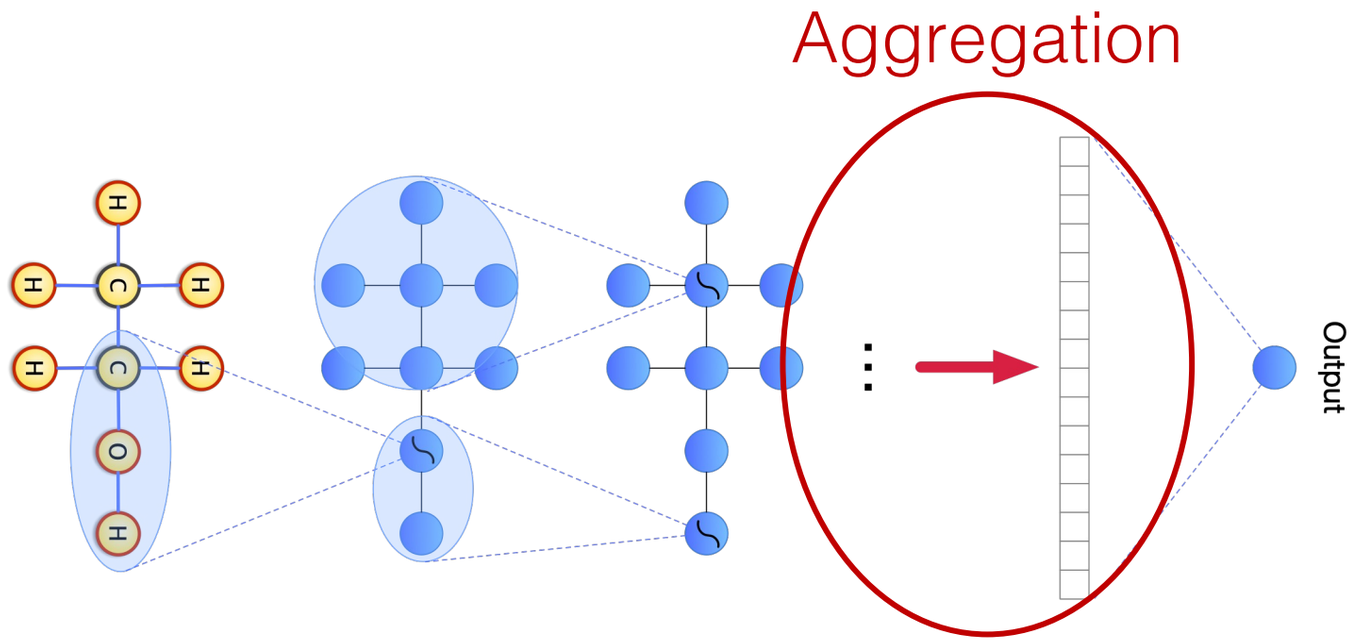
If f is expressive enough (and with an injective readout), a multilayer 1-localized GCN is as expressive ad the 1-dim WL isomorphism test

[6] Tran, D. V., Navarin, N., & Sperduti, A. (2018). On Filter Size in Graph Convolutional Networks. *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*, 1534–1541.

Aggregation Layer for graph classification

With GC we have a representation for each graph node.
How can we map node representations to a graph-level representation?

- **Aggregation (pooling)** function:
  - Maps a (multi) **set** of node representations to a **graph-level representation**
  - Differentiable
- Naïve solutions:
  - **sum** (or average) of node representations
  - More complex alternatives: Universal readout (DeepSets) [7]



[7] Navarin, N., Tran, D. Van, & Sperduti, A. (2019). Universal Readout for Graph Convolutional Neural Networks. *International Joint Conference on Neural Networks*. Budapest, Hungary.

Brief History of GNNs

- The idea of Neural Networks for structured data dates back to ’97 [Sperduti & Starita, 1997]
- In the ‘00s, two proposals:
  - Graph Neural Network Model [Scarselli, Gori et al., 2009]
    - Recurrent model, contraction mapping
  - Neural Networks for Graphs [Micheli, 2009]
    - Convolutional model, layer-wise training
- Recently, many works proposing slight modifications, e.g. :
  - [Li, Tarlow et al., 2016] Extends [Scarselli, Gori et al., 2009]
    - no contraction mapping, GRUs
  - [Kipf & Welling, 2017] proposes an approach similar to [Micheli, 2009] for node classification
    - end-to-end

Sperduti & Starita (1997). *Supervised neural networks for the classification of structures*. *IEEE TNNs.*
Scarselli, Gori et al. (2009). *The Graph Neural Network Model*. *IEEE TNNs.*
Micheli (2009). *Neural network for graphs: A contextual constructive approach*. *IEEE TNNs.*
Kipf & Welling(2017). *Semi-Supervised Classification with Graph Convolutional Networks*. In *ICLR.*
Tarlow et al. (2016). *Gated Graph Sequence Neural Networks*. In *ICLR*.

Graph Recurrent Neural Networks

Graph Neural Network Model [Scarselli, Gori et al., 2009]:

- Assumes the graph is fixed.
- instead of stacking multiple layers, a single recurrent layer can be adopted

$$\boldsymbol{h}_v^{t+1} = \sum_{u \in ne(v)} f(\boldsymbol{h}_v^t, \boldsymbol{x}_v \boldsymbol{x}_u)$$

Where f is a function (e.g. a neural network) with shared parameters across all the nodes and all the time steps.

- If f is a contraction mapping, the recursion converges to a fixed point $\boldsymbol{h}_v^*$ (that does not depend on the initialization of $\boldsymbol{h}$)
- Trained with an efficient version of backpropagation through time

Graph Recurrent Neural Networks

Gated Graph Neural Networks [Li, Tarlow et al., 2016]:

- the graph is fixed
- remove the constraint for the recurrent system to be a contraction mapping
- implemented this idea by adopting recurrent neural networks (GRU) to define the recurrent function f
- training with backpropagation through time for a fixed number of time steps

$$\mathbf{h}_v^{(1)} = [\boldsymbol{x}_v^\top, \mathbf{0}]^\top \tag{1}$$

$$\mathbf{a}_v^{(t)} = \mathbf{A}_{v:}^\top \left[\mathbf{h}_1^{(t-1)\top} \dots \mathbf{h}_{|\mathcal{V}|}^{(t-1)\top}\right]^\top + \mathbf{b} \tag{2}$$

$$\mathbf{z}_v^t = \sigma\left(\mathbf{W}^z \mathbf{a}_v^{(t)} + \mathbf{U}^z \mathbf{h}_v^{(t-1)}\right) \tag{3}$$

$$\mathbf{r}_v^t = \sigma\left(\mathbf{W}^r \mathbf{a}_v^{(t)} + \mathbf{U}^r \mathbf{h}_v^{(t-1)}\right) \tag{4}$$

$$\widetilde{\mathbf{h}_v^{(t)}} = \tanh\left(\mathbf{W}\mathbf{a}_v^{(t)} + \mathbf{U}\left(\mathbf{r}_v^t \odot \mathbf{h}_v^{(t-1)}\right)\right) \tag{5}$$

$$\mathbf{h}_v^{(t)} = (1 - \mathbf{z}_v^t) \odot \mathbf{h}_v^{(t-1)} + \mathbf{z}_v^t \odot \widetilde{\mathbf{h}_v^{(t)}}. \tag{6}$$

Graph Recurrent Neural Networks

Gated Graph Sequence Neural Networks [Li, Tarlow et al., 2016]:

- Extension to sequences of graphs where the node labels change
- With a sequence of outputs, one per time step
- Uses two GG-NNs: one for predicting the output, and one to predict the next state ($\boldsymbol{X}^{t+1}$)

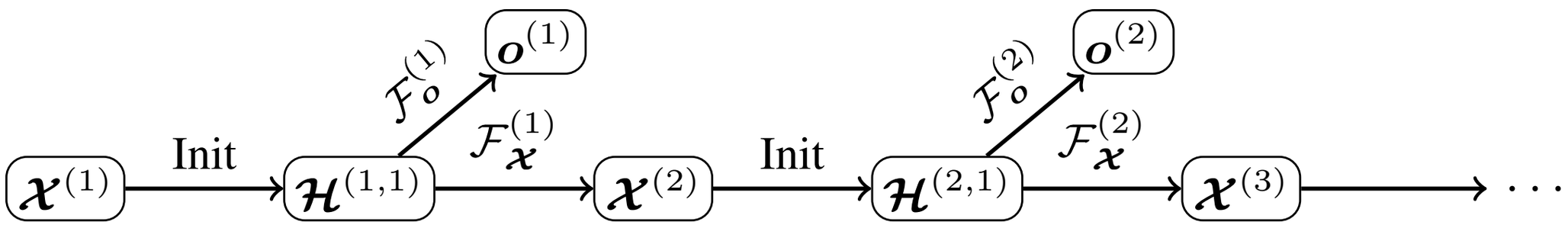


Figure 2: Architecture of GGS-NN models.

GNNs for dynamic graphs

The graph can change over time:

- Change in node attributes
- Change in edge attributes
- Change in graph topology (addition/removal of nodes and edges)

A simple case:

- Nodes and features are fixed
- Only the adjacency matrix changes over time
- A simple solution:
- Weighted aggregation of adjacency matrices

$$A = \sum_{t=0}^{T} \varphi(t, T) A^t$$

e.g. exponential decay scheme

- we obtain a weighted adjacency matrix for each time step that incorporates past information
- Similar approach when X changes over time

Temporal unrolling

- Similar idea: create a static graph from a dynamic sequence
- Consider only the last k time steps

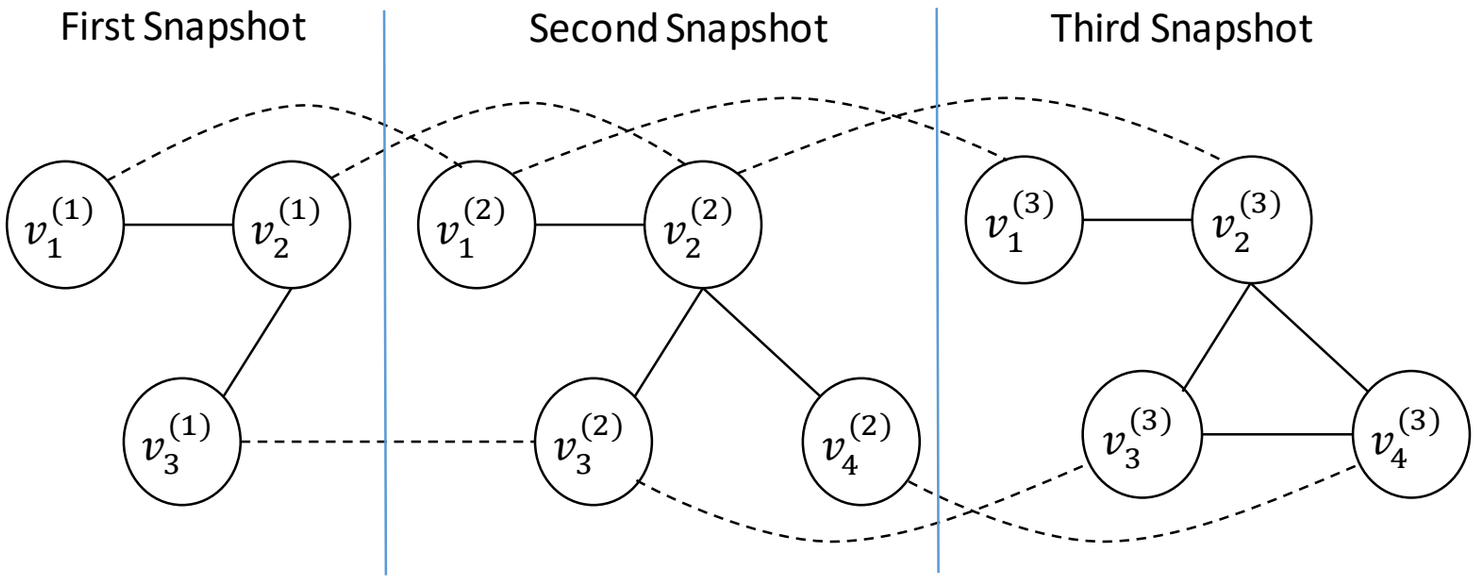


Fig. 15.4: An example of converting a DTDG into a static graph through temporal unrolling. Solid lines represent the edges in the graph at different timestamps and dashed lines represent the added edges. In this example, each node is connected to the node corresponding to the same entity only in the previous timestamp (i.e. $\omega = 1$).

Graph sequences: GNN-RNN

- Apply a GNN to the graphs at each time step
- For each node, we obtain a sequence of representaions over time

$$Z = [Z^0, \dots, Z^{T-1}]$$

- Use a RNN model to learn a mapping from the sequence of node representations to a single node representation

Thank you!

- 3rd part of the course already started!