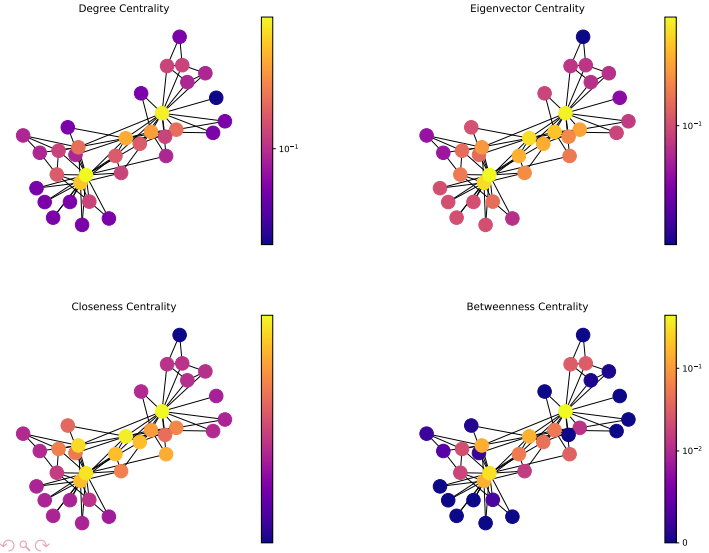
Graph (convolutional) Neural Networks (GNNs)

APM_5DS30_TP, Machine Learning with Graphs

Jhony H. Giraldo (Enseignant-Chercheur).
jhony.giraldo@telecom-paris.fr
October 10, 2025

Recap


Degree Centrality
10^{-1}
Eigenvector Centrality
10^{-1}
Closeness Centrality
Betweenness Centrality
10^{-1}
10^{-2}
0

Classical machine learning on graphs.

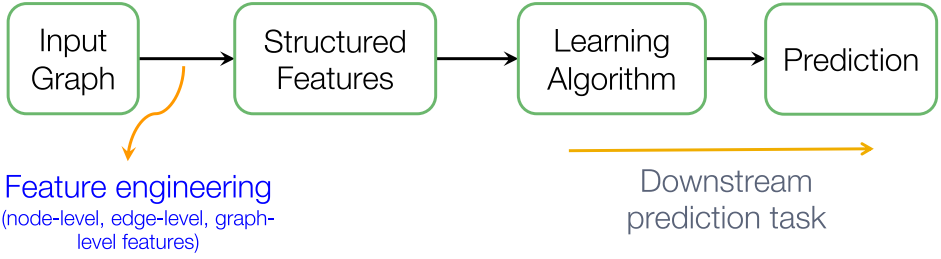
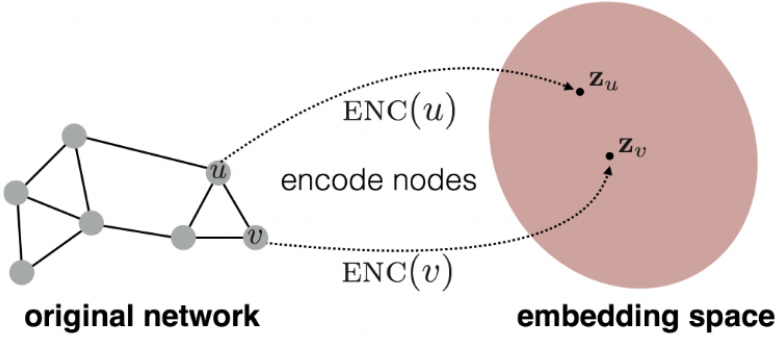


Image from Leskovec's lecture at Stanford

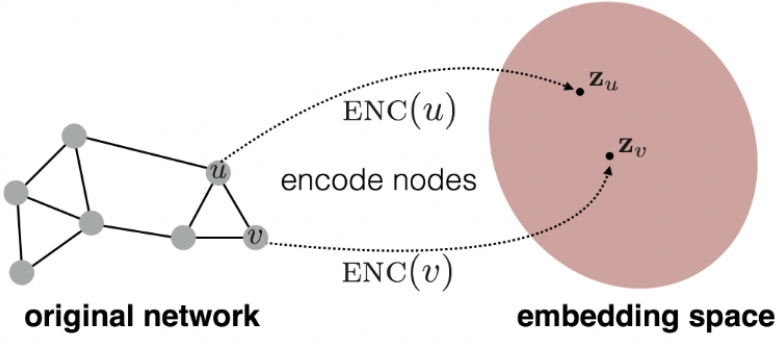
Recap

We want to do graph representation learning. We saw last week the DeepWalk method.



Recap

We want to do graph representation learning. We saw last week the DeepWalk method.



We know that DeepWalk doesn't scale well and should be retrained every time we have new data.

Outline

Objectives of the Lectures (2 and 3) in GNNs

- Graph Neural Networks (GNNs) are exciting tools with broad applicability and interesting properties
- The two objectives of the 2 lectures in GNNs are:

Applications

Develop the ability to use GNNs in practical applications

Fundamentals

Understand the fundamental properties and challenges of GNNs

Objectives of the Lectures (2 and 3) in GNNs

- Graph Neural Networks (GNNs) are exciting tools with broad applicability and interesting properties
- The two objectives of the 2 lectures in GNNs are:

Applications

Develop the ability to use GNNs in practical applications

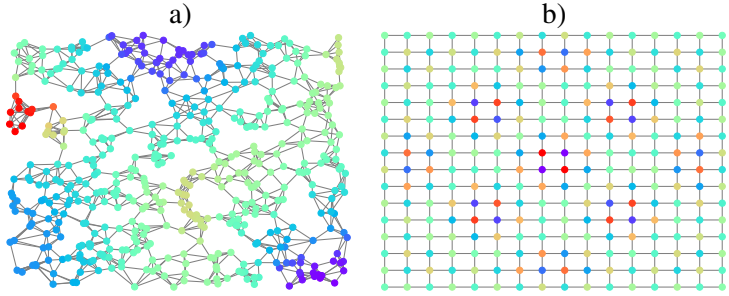
Fundamentals

Understand the fundamental properties and challenges of GNNs

- Identify situations where GNNs have potential. Formulate problems with GNNs. Develop solutions → TP.

Why using Neural Networks for Machine Learning on Graphs

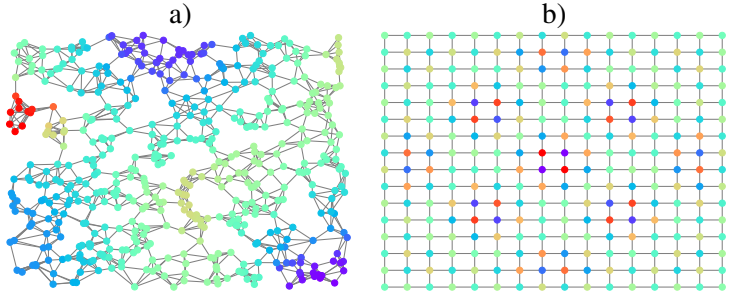
- Strong evidence (empirical and theoretical) supports the use of Neural Networks (NNs) - maybe that's why you're enrolled in this master.



Our objective is to run a NN over a). We are pretty good at running NNs over b)

Why using Neural Networks for Machine Learning on Graphs

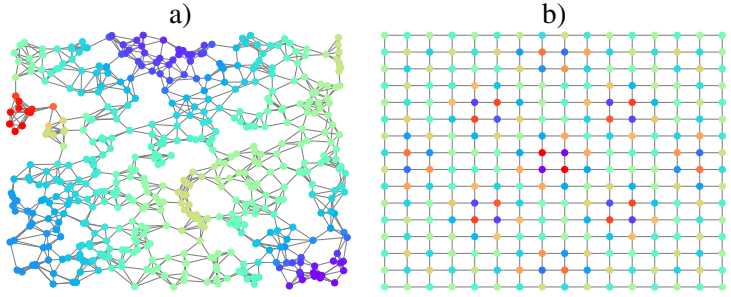
- Strong evidence (empirical and theoretical) supports the use of Neural Networks (NNs) - maybe that's why you're enrolled in this master.



Our objective is to run a NN over a). We are pretty good at running NNs over b)

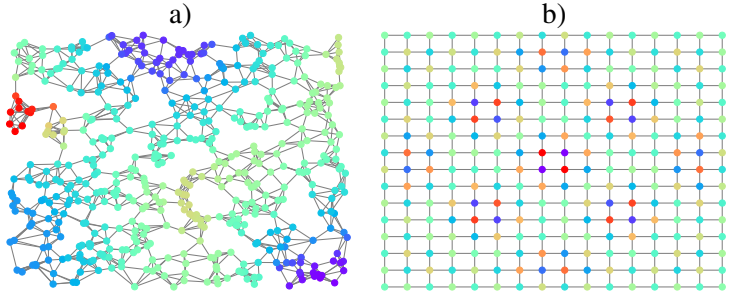
- Classical NNs (MLPs) do not scale to large dimensions (Why is that?). However, Convolutional Neural Networks (CNNs) scale well.

- CNNs are composed of layers with convolutional filters and point-wise non-linearities (and other stuff).



Process a) with graph convolutional NNs. Process b) with with convolutional NNs

- CNNs are composed of layers with convolutional filters and point-wise non-linearities (and other stuff).



Process a) with graph convolutional NNs. Process b) with with convolutional NNs

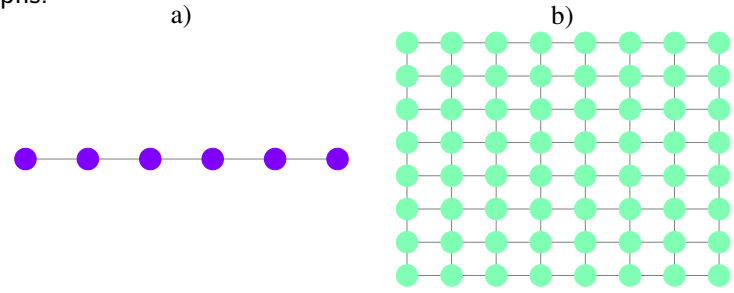
- We need to generalize convolutions to graphs. Then, we can composed graph filters with point-wise non-linearities.

Convolutions in Time, in Space, and on Graph

Convolutions in Time, in Space, and on Graphs

We Can Represent Time and Space Data as Graphs

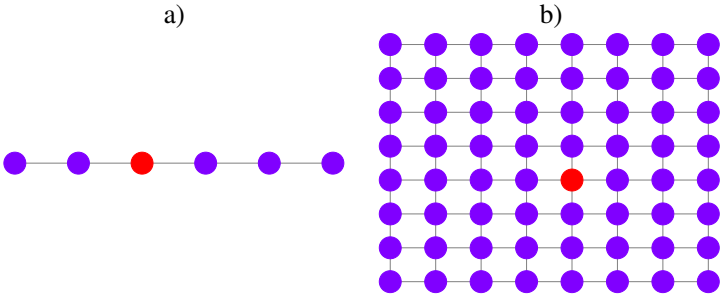
- We can describe discrete time (1D) and space (2D) structures using graphs.



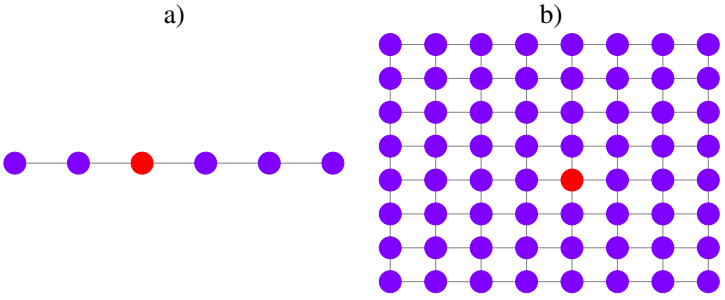
a) Description of time (1D) with a directed line graph, b) description of images (2D) with a grid graph

- Line graph represents adjacency of points in time (1D). Grid graph represents adjacency of points in space (2D).

- Use line and grid graphs to write convolutions as polynomials on respective shift operators (adjacency matrices) $\mathbf{S}$.

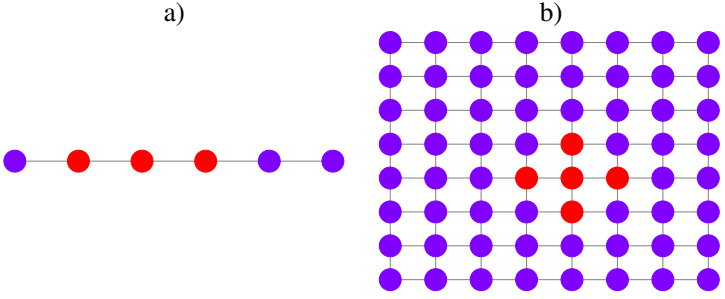


- Use line and grid graphs to write convolutions as polynomials on respective shift operators (adjacency matrices) $\mathbf{S}$.



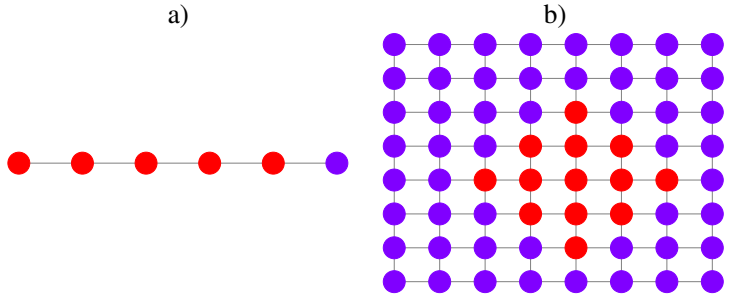
Filter with coefficients $h_k \rightarrow$
Output $\mathbf{z} = h_0 \mathbf{S}^0 \mathbf{x}$

- Use line and grid graphs to write convolutions as polynomials on respective shift operators (adjacency matrices) $\mathbf{S}$.



Filter with coefficients $h_k \rightarrow$
Output $\mathbf{z} = h_0\mathbf{S}^0\mathbf{x} + h_1\mathbf{S}^1\mathbf{x}$

- Use line and grid graphs to write convolutions as polynomials on respective shift operators (adjacency matrices) $\mathbf{S}$.



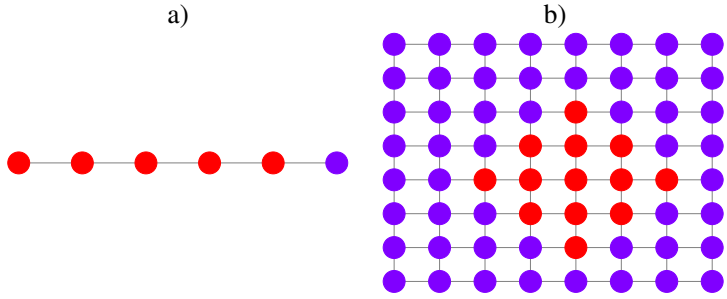
Filter with coefficients $h_k \rightarrow$
Output $\mathbf{z} = h_0\mathbf{S}^0\mathbf{x} + h_1\mathbf{S}^1\mathbf{x} + h_2\mathbf{S}^2\mathbf{x} + \cdots = \sum_{k=0}^{\infty} h_k\mathbf{S}^k\mathbf{x}$

- Time (1D) and space (2D) structures are important, but still a (very) limited class of signals.

Use graphs as generic descriptors of structures with signal values associated to nodes, and edges expressing the similarity between the nodes

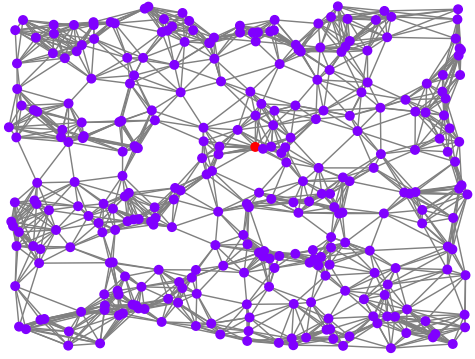
- (Particles) nodes are particles, signals are particle-based state representations, and edges capture the relationship between particles.

- We saw that convolutions in time and space are polynomials on adjacency matrices.



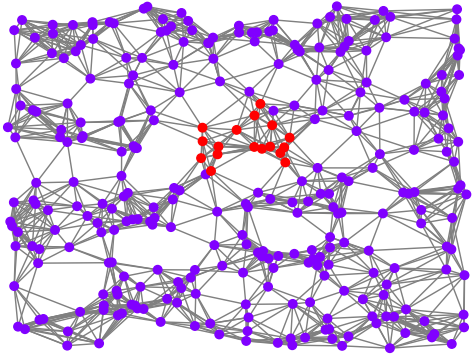
- Filter with coefficients $h_k \rightarrow$
  Output $\mathbf{z} = h_0\mathbf{S}^0\mathbf{x} + h_1\mathbf{S}^1\mathbf{x} + h_2\mathbf{S}^2\mathbf{x} + \cdots = \sum_{k=0}^{\infty} h_k\mathbf{S}^k\mathbf{x}$

- For graph signals, we define graph convolutions as polynomials on matrix representations of graphs.



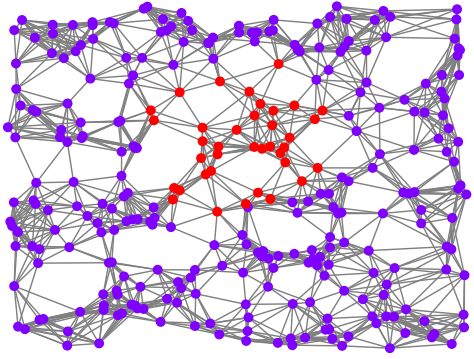
- Filter with coefficients $h_k \rightarrow$
  Output $\mathbf{z} = h_0 \mathbf{S}^0 \mathbf{x}$

- For graph signals, we define graph convolutions as polynomials on matrix representations of graphs.



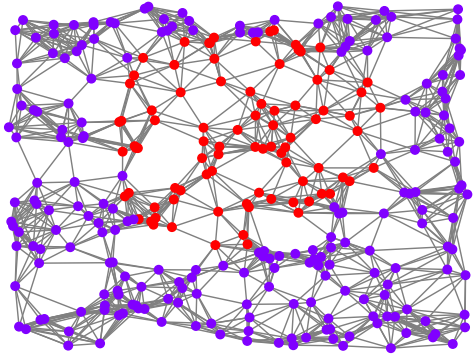
- Filter with coefficients $h_k \rightarrow$
  Output $\mathbf{z} = h_0\mathbf{S}^0\mathbf{x} + h_1\mathbf{S}^1\mathbf{x}$

- For graph signals, we define graph convolutions as polynomials on matrix representations of graphs.



- Filter with coefficients $h_k \rightarrow$
  Output $\mathbf{z} = h_0\mathbf{S}^0\mathbf{x} + h_1\mathbf{S}^1\mathbf{x} + h_2\mathbf{S}^2\mathbf{x}$

- For graph signals, we define graph convolutions as polynomials on matrix representations of graphs.

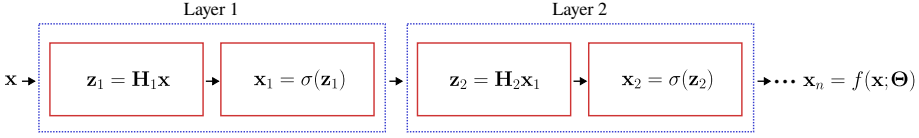


- Filter with coefficients $h_k \rightarrow$
  Output $\mathbf{z} = h_0\mathbf{S}^0\mathbf{x} + h_1\mathbf{S}^1\mathbf{x} + h_2\mathbf{S}^2\mathbf{x} + \cdots = \sum_{k=0}^{\infty} h_k\mathbf{S}^k\mathbf{x}$

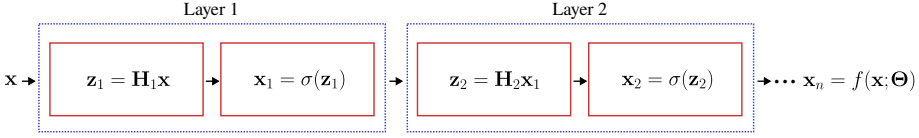
Graph Convolutional Neural Networks

- A NN is a composite of functions (layers)

- A NN is a composite of functions (layers)
- Each layer is a composition of linear maps with point-wise non-linearities (like ReLU or sigmoid)



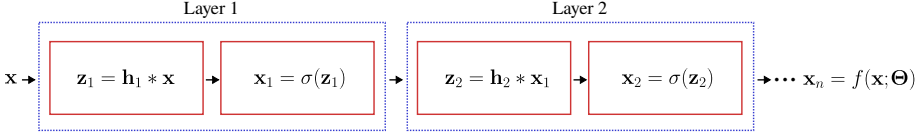
- A NN is a composite of functions (layers)
- Each layer is a composition of linear maps with point-wise non-linearities (like ReLU or sigmoid)



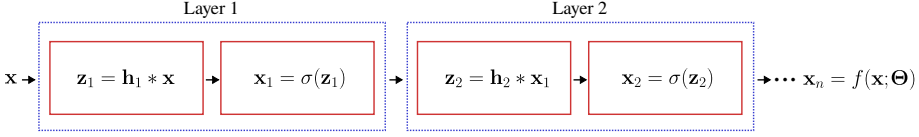
- NNs do not scale to large dimensional signals $\mathbf{x}$

- A CNN is also a composite of functions (layers)

- A CNN is also a composite of functions (layers)
- Each layer is a composition of convolutions with point-wise non-linearities



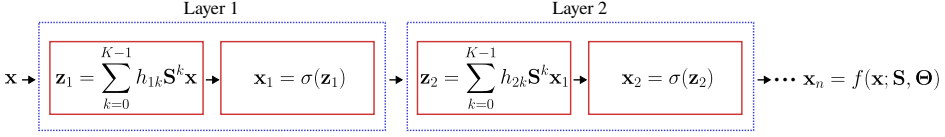
- A CNN is also a composite of functions (layers)
- Each layer is a composition of convolutions with point-wise non-linearities



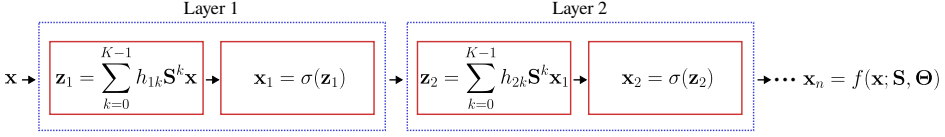
- CNNs scale well. CNNs are widely used in Deep Learning

- A GNN is also a composite of functions (layers)

- A GNN is also a composite of functions (layers)
- Each layer is a composition of graph convolutions with point-wise non-linearities



- A GNN is also a composite of functions (layers)
- Each layer is a composition of graph convolutions with point-wise non-linearities



- GNNs recover a CNN if $\mathbf{S}$ describes a directed line graph

Graphs, Basic Definitions (Recap)

- $G = (\mathcal{V}, \mathcal{E})$, $\mathcal{V}$ set of nodes, and $\mathcal{E}$ set of edges

Graphs, Basic Definitions (Recap)

- $G = (\mathcal{V}, \mathcal{E})$, $\mathcal{V}$ set of nodes, and $\mathcal{E}$ set of edges
- $x : \mathcal{V} \to \mathbb{R}$ graph signal, $\mathbf{x} \in \mathbb{R}^N$

Graphs, Basic Definitions (Recap)

- $G = (\mathcal{V}, \mathcal{E})$, $\mathcal{V}$ set of nodes, and $\mathcal{E}$ set of edges
- $x : \mathcal{V} \to \mathbb{R}$ graph signal, $\mathbf{x} \in \mathbb{R}^N$
- $\mathbf{A} \in \mathbb{R}^{N \times N}$ adjacency matrix
  - a_{ij} is the weight between nodes i and j when the graph is weighted
- $\mathbf{D} \in \mathbb{R}^{N \times N}$ degree matrix
  - $\mathbf{D} = \mathrm{diag}(\mathbf{D1})$, *i.e.*, $\mathbf{D}(i,i) = \sum_{j=1}^{N} \mathbf{A}(i,j)$
- $\mathbf{L} = \mathbf{D} - \mathbf{A}$ combinatorial Laplacian matrix

Graphs, Basic Definitions (Recap)

- $G = (\mathcal{V}, \mathcal{E})$, $\mathcal{V}$ set of nodes, and $\mathcal{E}$ set of edges
- $x : \mathcal{V} \to \mathbb{R}$ graph signal, $\mathbf{x} \in \mathbb{R}^N$
- $\mathbf{A} \in \mathbb{R}^{N \times N}$ adjacency matrix
  - a_{ij} is the weight between nodes i and j when the graph is weighted
- $\mathbf{D} \in \mathbb{R}^{N \times N}$ degree matrix
  - $\mathbf{D} = \mathrm{diag}(\mathbf{D1})$, *i.e.*, $\mathbf{D}(i,i) = \sum_{j=1}^{N} \mathbf{A}(i,j)$
- $\mathbf{L} = \mathbf{D} - \mathbf{A}$ combinatorial Laplacian matrix
- $\hat{\mathbf{A}} = \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$ normalized adjacency matrix

Graphs, Basic Definitions (Recap)

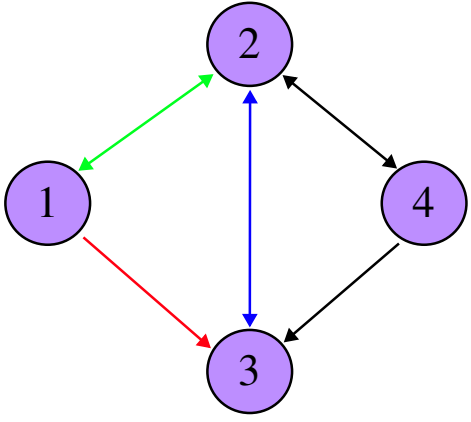
- $G = (\mathcal{V}, \mathcal{E})$, $\mathcal{V}$ set of nodes, and $\mathcal{E}$ set of edges
- $x : \mathcal{V} \to \mathbb{R}$ graph signal, $\mathbf{x} \in \mathbb{R}^N$
- $\mathbf{A} \in \mathbb{R}^{N \times N}$ adjacency matrix
  - a_{ij} is the weight between nodes i and j when the graph is weighted
- $\mathbf{D} \in \mathbb{R}^{N \times N}$ degree matrix
  - $\mathbf{D} = \mathrm{diag}(\mathbf{D1})$, *i.e.*, $\mathbf{D}(i,i) = \sum_{j=1}^{N} \mathbf{A}(i,j)$
- $\mathbf{L} = \mathbf{D} - \mathbf{A}$ combinatorial Laplacian matrix
- $\hat{\mathbf{A}} = \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$ normalized adjacency matrix
- $\hat{\mathbf{L}} = \mathbf{D}^{-1/2}\mathbf{L}\mathbf{D}^{-1/2}$ normalized Laplacian matrix

Graphs, Basic Definitions (Recap)

- $G = (\mathcal{V}, \mathcal{E})$, $\mathcal{V}$ set of nodes, and $\mathcal{E}$ set of edges
- $x : \mathcal{V} \to \mathbb{R}$ graph signal, $\mathbf{x} \in \mathbb{R}^N$
- $\mathbf{A} \in \mathbb{R}^{N \times N}$ adjacency matrix
  - a_{ij} is the weight between nodes i and j when the graph is weighted
- $\mathbf{D} \in \mathbb{R}^{N \times N}$ degree matrix
  - $\mathbf{D} = \mathrm{diag}(\mathbf{D1})$, *i.e.*, $\mathbf{D}(i,i) = \sum_{j=1}^{N} \mathbf{A}(i,j)$
- $\mathbf{L} = \mathbf{D} - \mathbf{A}$ combinatorial Laplacian matrix
- $\hat{\mathbf{A}} = \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$ normalized adjacency matrix
- $\hat{\mathbf{L}} = \mathbf{D}^{-1/2}\mathbf{L}\mathbf{D}^{-1/2}$ normalized Laplacian matrix
- $\mathbf{A}, \mathbf{L}, \hat{\mathbf{A}}, \hat{\mathbf{L}}$ are graph shift operators $\mathbf{S}$, the specific choice of graph shift operators matters in practice

Graphs, Basic Definitions (Recap)

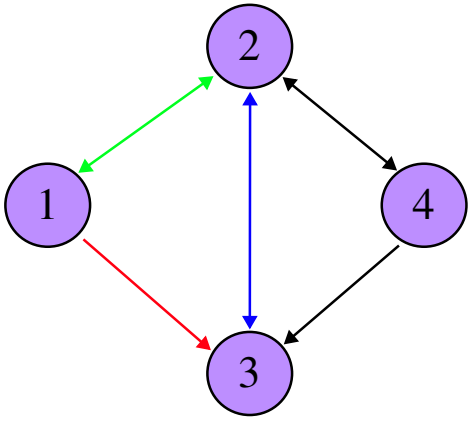
Example



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

Graphs, Basic Definitions (Recap)

Example



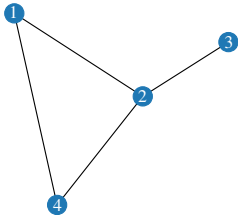
$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

- Is this graph undirected, directed, weighted, unweighted?

- Graphs have relatively few connections per node $\rightarrow$ adjacency matrices are **sparse** $\rightarrow$ **space inefficient**
- Imagine we have a graph with $N = 25'000,000$ and we use a full adjacency matrix to represent this graph
- If we represent each number of the matrix as a double tensor (in PyTorch for example), what's going to be the total size of the matrix?
- Hint: we use a 64-bit floating point to represent a double tensor in Pytorch
- Answer: $64/8 = 8$ bytes per number, so $8 \times N^2 = 8 \times 25'000,000^2 = 5e + 15$, *i.e.*, 5 Petabytes of memory RAM!
- The fastest computer on earth (in the US) has 4 Petabytes of memory RAM

Graphs, Basic Definitions (Recap)

The List of Edges Representation

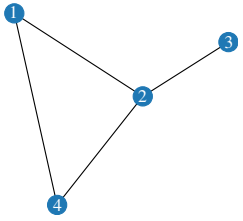


- In PyTorch Geometric (the tool we're gonna use in our TP), we use the list of edges representation
- For the graph in the figure we have

$$\begin{bmatrix} 1 & 1 & 2 & 2 & 2 & 3 & 4 & 4 \\ 2 & 4 & 1 & 3 & 4 & 2 & 1 & 2 \end{bmatrix}$$

Graphs, Basic Definitions (Recap)

The List of Edges Representation



- In PyTorch Geometric (the tool we're gonna use in our TP), we use the list of edges representation
- For the graph in the figure we have

$$\begin{bmatrix} 1 & 1 & 2 & 2 & 2 & 3 & 4 & 4 \\ 2 & 4 & 1 & 3 & 4 & 2 & 1 & 2 \end{bmatrix}$$

- We saw last week that with this representation we can save a lot of memory if the graph is sparse (which is typically the case)

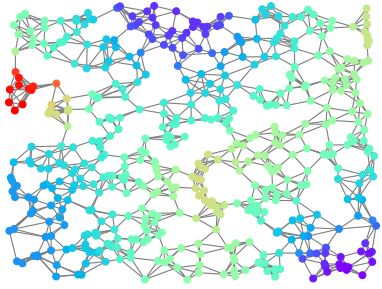
Graph Signals

Graph Signals

- A graph signal is a function that goes from the set of nodes to some vector space, *e.g.* $x : \mathcal{V} \to \mathbb{R}^m$

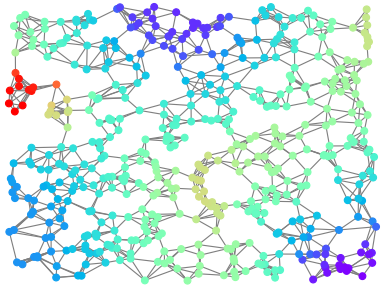
Graph Signals

- A graph signal is a function that goes from the set of nodes to some vector space, *e.g.* $x : \mathcal{V} \to \mathbb{R}^m$
- In graph signal processing, we use very often the space of real numbers, *i.e.*, $x : \mathcal{V} \to \mathbb{R}$. Thus we can represent the graph signal as $\mathbf{x} \in \mathbb{R}^N$



Graph Signals

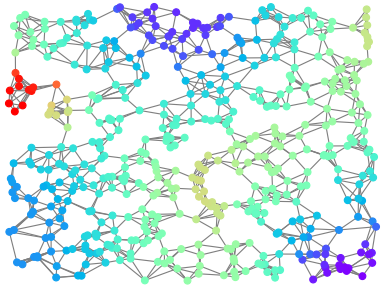
- A graph signal is a function that goes from the set of nodes to some vector space, *e.g.* $x : \mathcal{V} \to \mathbb{R}^m$
- In graph signal processing, we use very often the space of real numbers, *i.e.*, $x : \mathcal{V} \to \mathbb{R}$. Thus we can represent the graph signal as $\mathbf{x} \in \mathbb{R}^N$



- For example, the previous graph shows a sensor network, where the graph signal (colors) is the temperature at some instant in that graph.

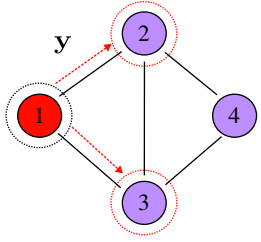
Graph Signals

- A graph signal is a function that goes from the set of nodes to some vector space, *e.g.* $x : \mathcal{V} \to \mathbb{R}^m$
- In graph signal processing, we use very often the space of real numbers, *i.e.*, $x : \mathcal{V} \to \mathbb{R}$. Thus we can represent the graph signal as $\mathbf{x} \in \mathbb{R}^N$



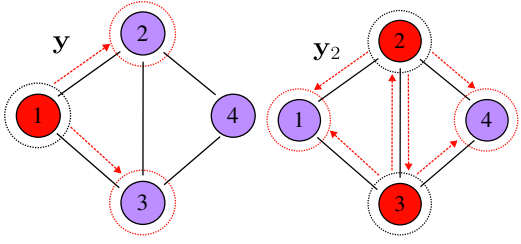
- For example, the previous graph shows a sensor network, where the graph signal (colors) is the temperature at some instant in that graph.
- The graph signal could be some feature representation.

- Multiplication by the graph shift operator (like the adjacency matrix) implements diffusion of the signal over the graph
- Diffused signal $\mathbf{y} = \mathbf{A}\mathbf{x}$, *i.e.*, $\mathbf{y}(i) = \sum_{j=1}^{N} a_{ij}\mathbf{x}(j) = \sum_{j \in \mathcal{N}_i} a_{ij}\mathbf{x}(j)$, where $\mathcal{N}_i$ is the neighborhood of node i.



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

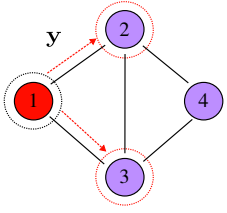
- Multiplication by the graph shift operator (like the adjacency matrix) implements diffusion of the signal over the graph
- Diffused signal $\mathbf{y} = \mathbf{A}\mathbf{x}$, *i.e.*, $\mathbf{y}(i) = \sum_{j=1}^{N} a_{ij}\mathbf{x}(j) = \sum_{j\in\mathcal{N}_i} a_{ij}\mathbf{x}(j)$, where $\mathcal{N}_i$ is the neighborhood of node i.



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

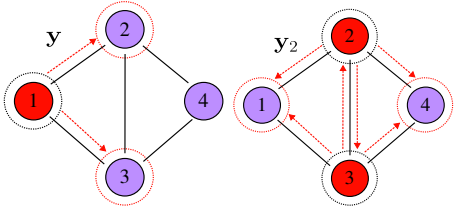
$$\mathbf{y}_2 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 1 \\ 2 \end{bmatrix}$$

- This diffusion codifies a local operation where components are mixed with components of neighboring nodes
- When graph are weighted, stronger weights contribute more to the diffusion output



$$\mathbf{y} = \begin{bmatrix} 0 & 0.5 & 1 & 0 \\ 0.5 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0.5 \\ 1 \\ 0 \end{bmatrix}$$

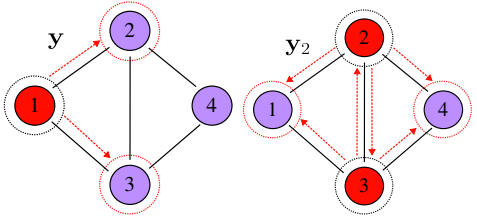
- This diffusion codifies a local operation where components are mixed with components of neighboring nodes
- When graph are weighted, stronger weights contribute more to the diffusion output



$$\mathbf{y} = \begin{bmatrix} 0 & 0.5 & 1 & 0 \\ 0.5 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0.5 \\ 1 \\ 0 \end{bmatrix}$$

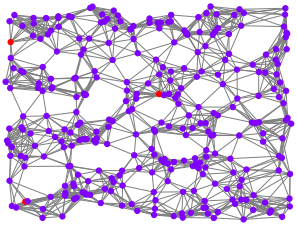
$$\mathbf{y}_2 = \begin{bmatrix} 0 & 0.5 & 1 & 0 \\ 0.5 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0.5 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1.25 \\ 1 \\ 0.5 \\ 1.5 \end{bmatrix}$$

- We compose the diffusion operator to produce the diffusion sequence $\mathbf{x}^{(k+1)} = \mathbf{S}\mathbf{x}^{(k)}$, with $\mathbf{x}^{(0)} = \mathbf{x}$
- We can unroll the recursion and write the diffusion sequence as the power sequence $\mathbf{x}^{(k)} = \mathbf{S}^k\mathbf{x}$

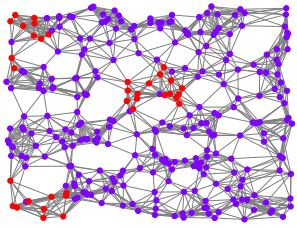


$$\mathbf{y}_2 = \begin{bmatrix} 0 & 0.5 & 1 & 0 \\ 0.5 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}^2 \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1.25 \\ 1 \\ 0.5 \\ 1.5 \end{bmatrix}$$

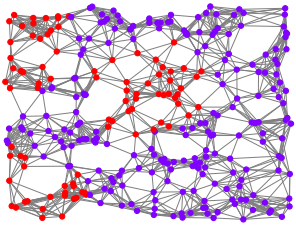
- The kth element of the diffusion sequence $\mathbf{x}^{(k)}$ diffuses information to k-hop neighborhoods
- We have a recursive definition and another one using powers of $\mathbf{S}$. Regarding implementation, always implement the recursive definition



$\mathbf{x}^{(0)} = \mathbf{x} = \mathbf{S}^0\mathbf{x}$



$\mathbf{x}^{(1)} = \mathbf{S}\mathbf{x}^{(0)} = \mathbf{S}^1\mathbf{x}$



$\mathbf{x}^{(2)} = \mathbf{S}\mathbf{x}^{(1)} = \mathbf{S}^2\mathbf{x}$

Graph Convolutional Filters

- A graph filter is a polynomial on the graph shift operator $\mathbf{S}$

$$\mathbf{H}(\mathbf{S}) = \sum_{k=0}^{\infty} h_k \mathbf{S}^k$$

- A graph filter is a polynomial on the graph shift operator $\mathbf{S}$

$$\mathbf{H}(\mathbf{S}) = \sum_{k=0}^{\infty} h_k \mathbf{S}^k$$

- The result of applying the filter $\mathbf{H}(\mathbf{S})$ to the signal $\mathbf{x}$ is a signal $\mathbf{y}$

$$\mathbf{y} = \mathbf{H}(\mathbf{S})\mathbf{x} = \sum_{k=0}^{\infty} h_k \mathbf{S}^k \mathbf{x} \tag{1}$$

- A graph filter is a polynomial on the graph shift operator $\mathbf{S}$

$$\mathbf{H}(\mathbf{S}) = \sum_{k=0}^{\infty} h_k \mathbf{S}^k$$

- The result of applying the filter $\mathbf{H}(\mathbf{S})$ to the signal $\mathbf{x}$ is a signal $\mathbf{y}$

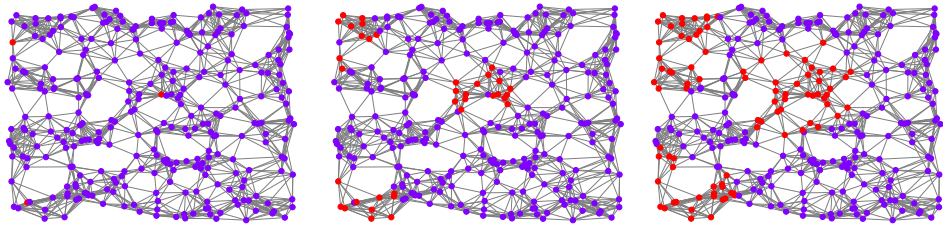
$$\mathbf{y} = \mathbf{H}(\mathbf{S})\mathbf{x} = \sum_{k=0}^{\infty} h_k \mathbf{S}^k \mathbf{x} \tag{1}$$

- The signal $\mathbf{y} = \mathbf{h}_{*\mathbf{S}}\mathbf{x}$ is the graph convolution of the filter $\mathbf{h} = \{h_k\}_{k=0}^{\infty}$ with the signal $\mathbf{x}$. We will see later that the most famous GNN uses an approximation of (1)

Graph Convolutional Filters

From Local to Global Information

- Graph convolutions aggregate information growing from local to global neighborhoods (the diffusion sequence)
- Given a graph signal $\mathbf{x}$, shift operator $\mathbf{S}$, and a filter $\mathbf{h} = \{h_k\}_{k=0}^{\infty}$:



- The graph convolution output is given by
$\mathbf{y} = h_0\mathbf{S}^0\mathbf{x} + h_1\mathbf{S}^1\mathbf{x} + h_2\mathbf{S}^2\mathbf{x} + h_3\mathbf{S}^3\mathbf{x} + \cdots = \sum_{k=0}^{\infty} h_k\mathbf{S}^k\mathbf{x}$

Wooclap

Graph Neural Networks (GNNs)

Desired Properties on GNNs

- Properties we want on GNNs:

Desired Properties on GNNs

- Properties we want on GNNs:
  - **Computational and storage efficiency**, proportional to $\mathcal{O}(|\mathcal{E}|)$

Desired Properties on GNNs

- Properties we want on GNNs:
  - **Computational and storage efficiency**, proportional to $\mathcal{O}(|\mathcal{E}|)$
  - **Fixed** number of parameters (independent of the number of nodes)

Desired Properties on GNNs

- Properties we want on GNNs:
  - **Computational and storage efficiency**, proportional to $\mathcal{O}(|\mathcal{E}|)$
  - **Fixed** number of parameters (independent of the number of nodes)
  - **Localization** (acts on the local neighborhood of a node)

Desired Properties on GNNs

- Properties we want on GNNs:
  - **Computational and storage efficiency**, proportional to $\mathcal{O}(|\mathcal{E}|)$
  - **Fixed** number of parameters (independent of the number of nodes)
  - **Localization** (acts on the local neighborhood of a node)
  - We can apply our model in **inductive problems**

Desired Properties on GNNs

- Properties we want on GNNs:
  - **Computational and storage efficiency**, proportional to $\mathcal{O}(|\mathcal{E}|)$
  - **Fixed** number of parameters (independent of the number of nodes)
  - **Localization** (acts on the local neighborhood of a node)
  - We can apply our model in **inductive problems**
- Images have quite a regular structure, and we have decades of research on convolutions from the field of digital signal processing

Desired Properties on GNNs

- Properties we want on GNNs:
  - **Computational and storage efficiency**, proportional to $\mathcal{O}(|\mathcal{E}|)$
  - **Fixed** number of parameters (independent of the number of nodes)
  - **Localization** (acts on the local neighborhood of a node)
  - We can apply our model in **inductive problems**
- Images have quite a regular structure, and we have decades of research on convolutions from the field of digital signal processing
- We already know how to do graph convolutions. However, computing the recursive diffusion sequence is expensive (let's come back to Wooclap)

Desired Properties on GNNs

- Properties we want on GNNs:
  - **Computational and storage efficiency**, proportional to $\mathcal{O}(|\mathcal{E}|)$
  - **Fixed** number of parameters (independent of the number of nodes)
  - **Localization** (acts on the local neighborhood of a node)
  - We can apply our model in **inductive problems**
- Images have quite a regular structure, and we have decades of research on convolutions from the field of digital signal processing
- We already know how to do graph convolutions. However, computing the recursive diffusion sequence is expensive (let's come back to Wooclap)

Desired Properties on GNNs

- Properties we want on GNNs:
  - **Computational and storage efficiency**, proportional to $\mathcal{O}(|\mathcal{E}|)$
  - **Fixed** number of parameters (independent of the number of nodes)
  - **Localization** (acts on the local neighborhood of a node)
  - We can apply our model in **inductive problems**
- Images have quite a regular structure, and we have decades of research on convolutions from the field of digital signal processing
- We already know how to do graph convolutions. However, computing the recursive diffusion sequence is expensive (let's come back to Wooclap)
- This is because we cannot parallelize our NN

Desired Properties on GNNs

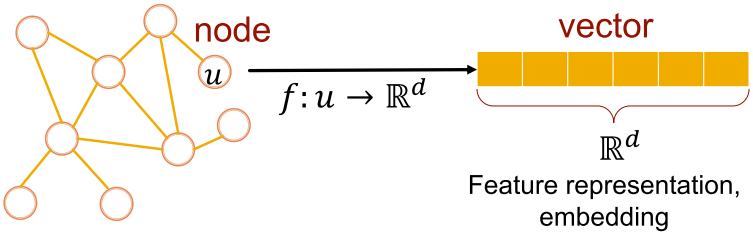
- Properties we want on GNNs:
  - **Computational and storage efficiency**, proportional to $\mathcal{O}(|\mathcal{E}|)$
  - **Fixed** number of parameters (independent of the number of nodes)
  - **Localization** (acts on the local neighborhood of a node)
  - We can apply our model in **inductive problems**
- Images have quite a regular structure, and we have decades of research on convolutions from the field of digital signal processing
- We already know how to do graph convolutions. However, computing the recursive diffusion sequence is expensive (let's come back to Wooclap)
- This is because we cannot parallelize our NN
- Computing the powers of $\mathbf{S}$ is computationally and memory expensive as well

Desired Properties on GNNs

- Properties we want on GNNs:
  - **Computational and storage efficiency**, proportional to $\mathcal{O}(|\mathcal{E}|)$
  - **Fixed** number of parameters (independent of the number of nodes)
  - **Localization** (acts on the local neighborhood of a node)
  - We can apply our model in **inductive problems**
- Images have quite a regular structure, and we have decades of research on convolutions from the field of digital signal processing
- We already know how to do graph convolutions. However, computing the recursive diffusion sequence is expensive (let's come back to Wooclap)
- This is because we cannot parallelize our NN
- Computing the powers of $\mathbf{S}$ is computationally and memory expensive as well
- We need to make some approximations in practice

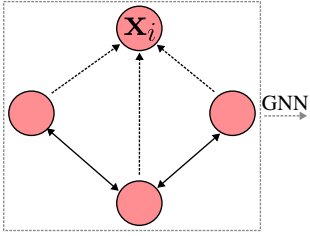
Graph Representation Learning: Node Embedding (Recap)

- Goal efficient **task-independent** feature learning for machine learning with graphs aka **node embedding**

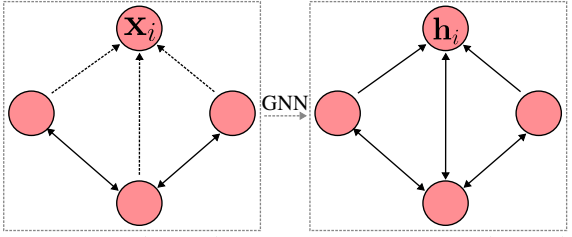


- Indeed, our input node could have also a feature $\mathbb{R}^F$ attached to it
- We'll learn in this lecture how to learn the function f

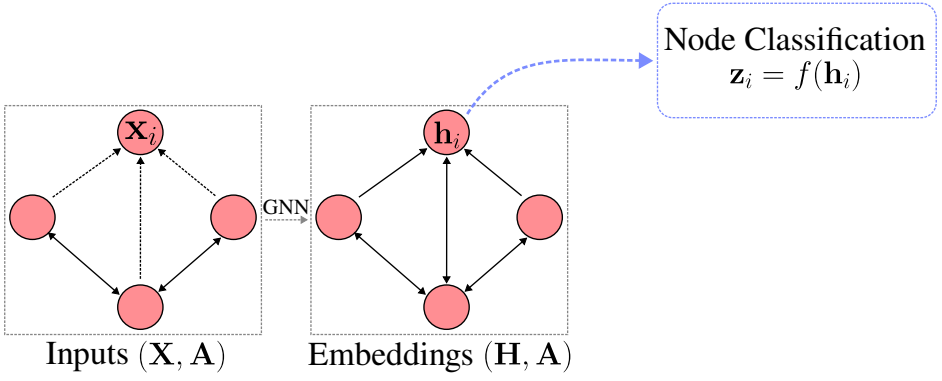
What we can do with the Outputs of a GNN?



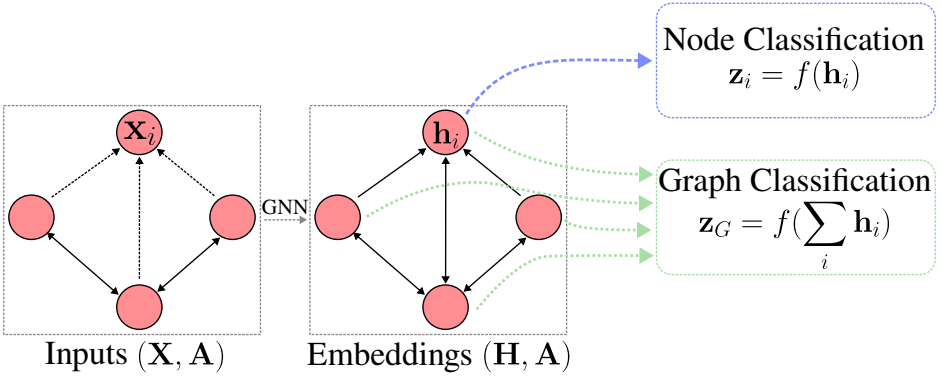
What we can do with the Outputs of a GNN?



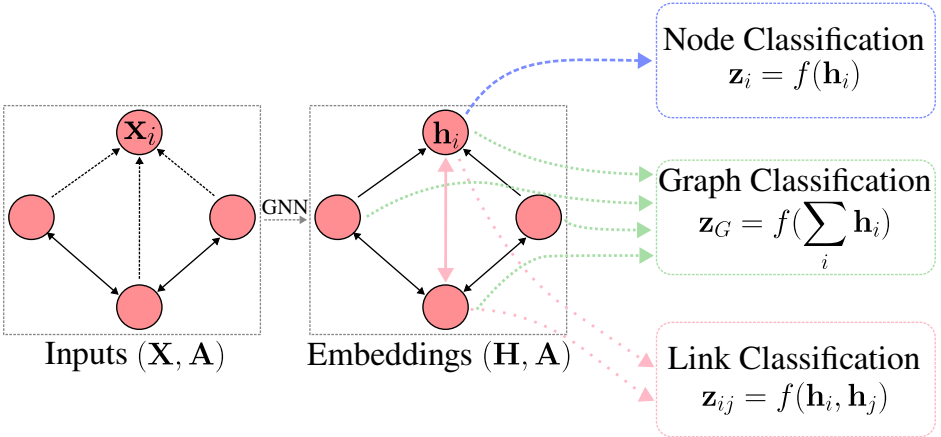
What we can do with the Outputs of a GNN?



What we can do with the Outputs of a GNN?

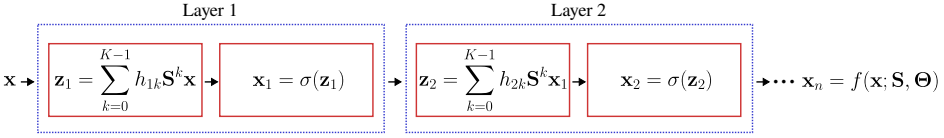


What we can do with the Outputs of a GNN?



Graph Convolutional Networks

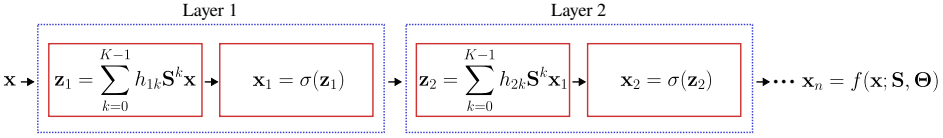
What is the Problem with the Graph Filtering?



- $\mathbf{y} = h_0\mathbf{S}^0\mathbf{x} + h_1\mathbf{S}^1\mathbf{x} + h_2\mathbf{S}^2\mathbf{x} + h_3\mathbf{S}^3\mathbf{x} + \cdots = \sum_{k=0}^{\infty} h_k\mathbf{S}^k\mathbf{x}$

Graph Convolutional Networks

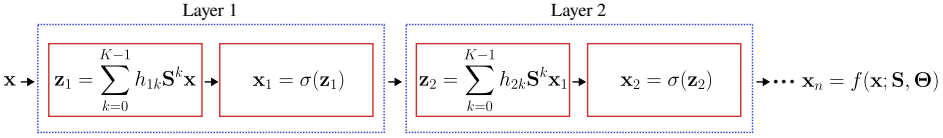
What is the Problem with the Graph Filtering?



- $\mathbf{y} = h_0\mathbf{S}^0\mathbf{x} + h_1\mathbf{S}^1\mathbf{x} + h_2\mathbf{S}^2\mathbf{x} + h_3\mathbf{S}^3\mathbf{x} + \cdots = \sum_{k=0}^{\infty} h_k\mathbf{S}^k\mathbf{x}$
- There has been some architectures based on this graph filters, *e.g.* [Defferrard et al.(2016)]

Graph Convolutional Networks

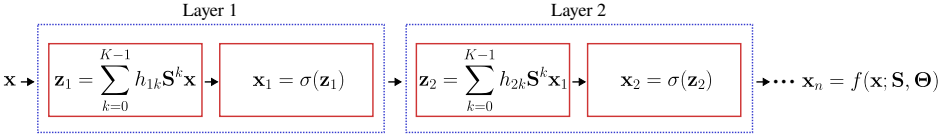
What is the Problem with the Graph Filtering?



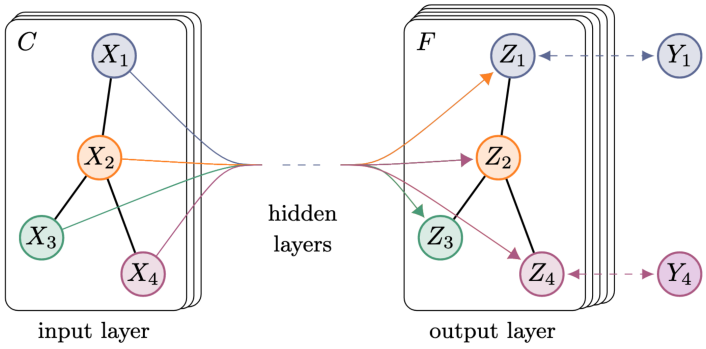
- $\mathbf{y} = h_0\mathbf{S}^0\mathbf{x} + h_1\mathbf{S}^1\mathbf{x} + h_2\mathbf{S}^2\mathbf{x} + h_3\mathbf{S}^3\mathbf{x} + \cdots = \sum_{k=0}^{\infty} h_k\mathbf{S}^k\mathbf{x}$
- There has been some architectures based on this graph filters, *e.g.* [Defferrard et al.(2016)]
- Problem: $\mathbf{S}$ becomes denser with powers (computational and memory problems for large-scale graphs)

Graph Convolutional Networks

What is the Problem with the Graph Filtering?



- $\mathbf{y} = h_0\mathbf{S}^0\mathbf{x} + h_1\mathbf{S}^1\mathbf{x} + h_2\mathbf{S}^2\mathbf{x} + h_3\mathbf{S}^3\mathbf{x} + \cdots = \sum_{k=0}^{\infty} h_k\mathbf{S}^k\mathbf{x}$
- There has been some architectures based on this graph filters, *e.g.* [Defferrard et al.(2016)]
- Problem: $\mathbf{S}$ becomes denser with powers (computational and memory problems for large-scale graphs)
- Problem: Using the recursive definition imposes a bottleneck for the implementation



Graph Convolutional Network (GCN). **Source:** [Kipf and Welling(2017)]

- GCN is the most widely used (and most cited work) in the GNN community (close to $50,000$ citations on October 2025)
- Its success is due to its simplicity and computational efficiency

- GCN is an approximation of the graph filter with $h_1 = 1$, $h_i = 0 \; \forall \; i \neq 1$, $\mathbf{S} = \hat{\mathbf{A}} = \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$ (normalized adjacency matrix), and another trick

- GCN is an approximation of the graph filter with $h_1 = 1$, $h_i = 0 \; \forall \; i \neq 1$, $\mathbf{S} = \hat{\mathbf{A}} = \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$ (normalized adjacency matrix), and another trick
- Most precisely, the propagation rule of GCN is given by $\mathbf{H}^{(l+1)} = \sigma(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{H}^{(l)}\mathbf{W}^{(l)})$, where $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$, $\tilde{\mathbf{D}}$ is the degree matrix of $\tilde{\mathbf{A}}$, $\mathbf{H}^{(l)}$ is the matrix of activations in layer l such that $\mathbf{H}^{(0)} = \mathbf{X}$ (data matrix), $\mathbf{W}^{(l)}$ is the matrix of trainable weights in layer l, and $\sigma(\cdot)$ is an activation function

- GCN is an approximation of the graph filter with $h_1 = 1$, $h_i = 0 \; \forall \; i \neq 1$, $\mathbf{S} = \hat{\mathbf{A}} = \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$ (normalized adjacency matrix), and another trick
- Most precisely, the propagation rule of GCN is given by $\mathbf{H}^{(l+1)} = \sigma(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{H}^{(l)}\mathbf{W}^{(l)})$, where $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$, $\tilde{\mathbf{D}}$ is the degree matrix of $\tilde{\mathbf{A}}$, $\mathbf{H}^{(l)}$ is the matrix of activations in layer l such that $\mathbf{H}^{(0)} = \mathbf{X}$ (data matrix), $\mathbf{W}^{(l)}$ is the matrix of trainable weights in layer l, and $\sigma(\cdot)$ is an activation function
- Notice that the trick is such that $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$. This corresponds to add self-loops, and it was done to avoid exploding/vanishing gradients

- GCN is an approximation of the graph filter with $h_1 = 1$, $h_i = 0 \; \forall \; i \neq 1$, $\mathbf{S} = \hat{\mathbf{A}} = \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$ (normalized adjacency matrix), and another trick
- Most precisely, the propagation rule of GCN is given by $\mathbf{H}^{(l+1)} = \sigma(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{H}^{(l)}\mathbf{W}^{(l)})$, where $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$, $\tilde{\mathbf{D}}$ is the degree matrix of $\tilde{\mathbf{A}}$, $\mathbf{H}^{(l)}$ is the matrix of activations in layer l such that $\mathbf{H}^{(0)} = \mathbf{X}$ (data matrix), $\mathbf{W}^{(l)}$ is the matrix of trainable weights in layer l, and $\sigma(\cdot)$ is an activation function
- Notice that the trick is such that $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$. This corresponds to add self-loops, and it was done to avoid exploding/vanishing gradients
- Notice that GCN reduces to a linear layer when $\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}} = \mathbf{I}$ (no graph structure)

- Let's take a closer look to the propagation rule in the first layer $\mathbf{H}^{(1)} = \sigma(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{X}\mathbf{W}^{(0)})$

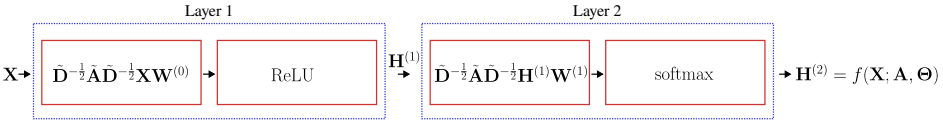
- Let's take a closer look to the propagation rule in the first layer $\mathbf{H}^{(1)} = \sigma(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{X}\mathbf{W}^{(0)})$
- $\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}} \in \mathbb{R}^{N\times N}$, $\mathbf{X} \in \mathbb{R}^{N\times F}$, and $\mathbf{W}^{(0)} \in \mathbb{R}^{F\times H_0}$, where N is the number of nodes, F is the dimension of the input features, and H_0 is the number of hidden units of the layer 0 (this is a hyperparameter)

- Let's take a closer look to the propagation rule in the first layer $\mathbf{H}^{(1)} = \sigma(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{X}\mathbf{W}^{(0)})$
- $\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}} \in \mathbb{R}^{N\times N}$, $\mathbf{X} \in \mathbb{R}^{N\times F}$, and $\mathbf{W}^{(0)} \in \mathbb{R}^{F\times H_0}$, where N is the number of nodes, F is the dimension of the input features, and H_0 is the number of hidden units of the layer 0 (this is a hyperparameter)
- Thus, the dimensions of $\mathbf{H}^{(1)} \in \mathbb{R}^{N\times H_0}$

- Let's take a closer look to the propagation rule in the first layer $\mathbf{H}^{(1)} = \sigma(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{X}\mathbf{W}^{(0)})$
- $\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}} \in \mathbb{R}^{N\times N}$, $\mathbf{X} \in \mathbb{R}^{N\times F}$, and $\mathbf{W}^{(0)} \in \mathbb{R}^{F\times H_0}$, where N is the number of nodes, F is the dimension of the input features, and H_0 is the number of hidden units of the layer 0 (this is a hyperparameter)
- Thus, the dimensions of $\mathbf{H}^{(1)} \in \mathbb{R}^{N\times H_0}$
- If we have another layer $\mathbf{H}^{(2)} = \sigma(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{H}^{(1)}\mathbf{W}^{(1)})$, where $\mathbf{W}^{(1)} \in \mathbb{R}^{H_0\times H_1}$, where H_1 is the number of hidden units of the layer 1

- Let's take a closer look to the propagation rule in the first layer $\mathbf{H}^{(1)} = \sigma(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{X}\mathbf{W}^{(0)})$
- $\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}} \in \mathbb{R}^{N\times N}$, $\mathbf{X} \in \mathbb{R}^{N\times F}$, and $\mathbf{W}^{(0)} \in \mathbb{R}^{F\times H_0}$, where N is the number of nodes, F is the dimension of the input features, and H_0 is the number of hidden units of the layer 0 (this is a hyperparameter)
- Thus, the dimensions of $\mathbf{H}^{(1)} \in \mathbb{R}^{N\times H_0}$
- If we have another layer $\mathbf{H}^{(2)} = \sigma(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{H}^{(1)}\mathbf{W}^{(1)})$, where $\mathbf{W}^{(1)} \in \mathbb{R}^{H_0\times H_1}$, where H_1 is the number of hidden units of the layer 1
- The vanilla (two layers) GCN is given by: $\bar{\mathbf{Y}} = f(\mathbf{X}; \mathbf{A}, \mathbf{\Theta}) = \mathrm{softmax}\left(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathrm{ReLU}\left(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{X}\mathbf{W}^{(0)}\right)\mathbf{W}^{(1)}\right)$

- Too much math? Let's look at GCN visually.



Block diagram of Vanilla GCN for node classification

- We're just missing the loss function and the optimizer to train our GCN

- We're just missing the loss function and the optimizer to train our GCN
- Let's assume we're performing node classification, so we can use the cross-entropy loss as follows:

$$\mathcal{L} = -\frac{1}{|\mathcal{S}|} \sum_{i \in \mathcal{S}} \sum_{j=1}^{C} \mathbf{Y}(i,j) \ln \bar{\mathbf{Y}}(i,j),$$

where $\mathcal{S}$ is the set of indices of the training samples, C is the number of classes in our problem, and $\mathbf{Y} \in \{0,1\}^{N \times C}$ is the matrix of N one-hot encoded vectors representing the classes of the nodes $\mathcal{V}$

- We're just missing the loss function and the optimizer to train our GCN
- Let's assume we're performing node classification, so we can use the cross-entropy loss as follows:

$$\mathcal{L} = -\frac{1}{|\mathcal{S}|} \sum_{i \in \mathcal{S}} \sum_{j=1}^{C} \mathbf{Y}(i,j) \ln \bar{\mathbf{Y}}(i,j),$$

  where $\mathcal{S}$ is the set of indices of the training samples, C is the number of classes in our problem, and $\mathbf{Y} \in \{0,1\}^{N \times C}$ is the matrix of N one-hot encoded vectors representing the classes of the nodes $\mathcal{V}$
- In practice, you just need to use the function CrossEntropyLoss ("torch.nn.CrossEntropyLoss" in PyTorch)

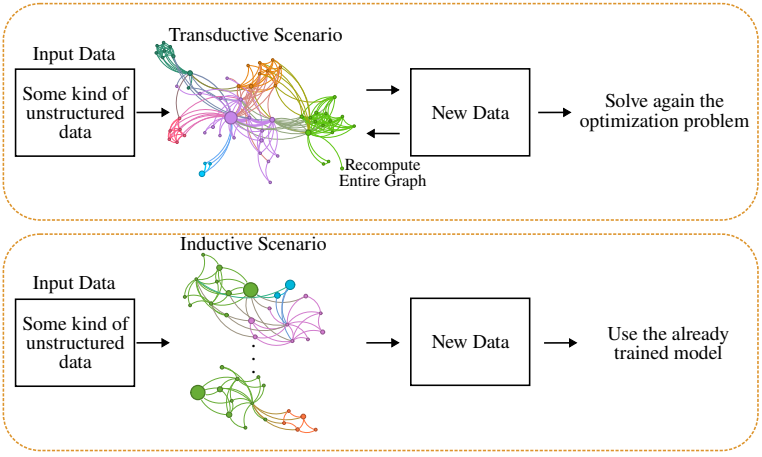
- We're just missing the loss function and the optimizer to train our GCN
- Let's assume we're performing node classification, so we can use the cross-entropy loss as follows:

$$\mathcal{L} = -\frac{1}{|\mathcal{S}|} \sum_{i \in \mathcal{S}} \sum_{j=1}^{C} \mathbf{Y}(i,j) \ln \bar{\mathbf{Y}}(i,j),$$

  where $\mathcal{S}$ is the set of indices of the training samples, C is the number of classes in our problem, and $\mathbf{Y} \in \{0,1\}^{N \times C}$ is the matrix of N one-hot encoded vectors representing the classes of the nodes $\mathcal{V}$
- In practice, you just need to use the function CrossEntropyLoss ("torch.nn.CrossEntropyLoss" in PyTorch)
- Finally, you can train your GCN with SGD or Adam for example

Graph Convolutional Networks

Inductive Learning

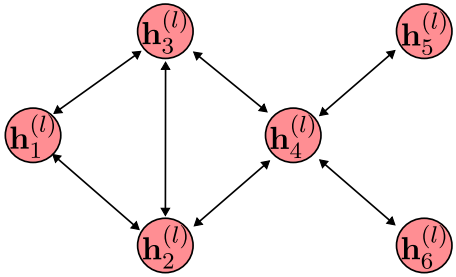


GCNs works in the inductive scenario!

Message Passing Neural Network (MPNN)

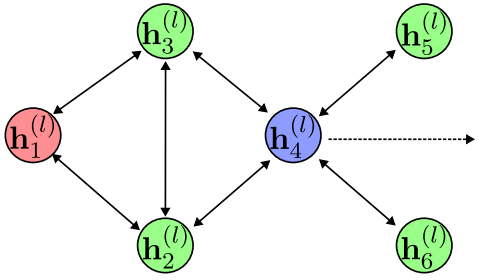
Message Passing Neural Network (MPNN) Framework

MPNN



Message Passing Neural Network (MPNN) Framework

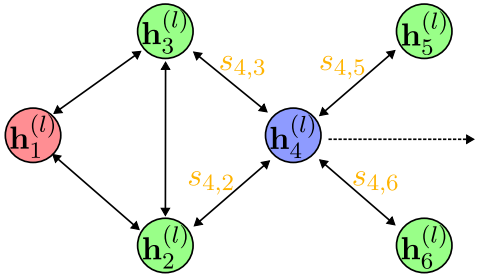
MPNN



$$\mathbf{m}_4^{(l+1)} = \sum_{j \in \mathcal{N}_4} \mathbf{S}(4,j)\psi_l(\mathbf{h}_4^{(l)}, \mathbf{h}_j^{(l)})$$

Message Passing Neural Network (MPNN) Framework

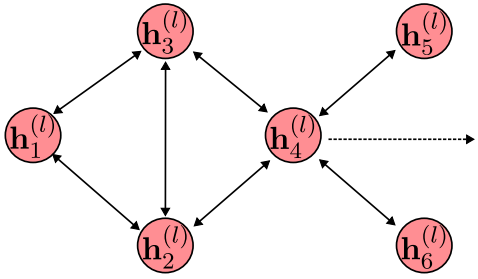
MPNN



$$\mathbf{m}_4^{(l+1)} = \sum_{j \in \mathcal{N}_4} \mathbf{S}(4,j)\psi_l(\mathbf{h}_4^{(l)}, \mathbf{h}_j^{(l)})$$

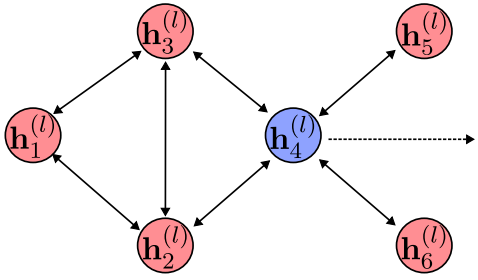
Message Passing Neural Network (MPNN) Framework

MPNN



$$\mathbf{m}_4^{(l+1)} = \sum_{j \in \mathcal{N}_4} \mathbf{S}(4,j)\psi_l(\mathbf{h}_4^{(l)}, \mathbf{h}_j^{(l)})$$

$$\mathbf{h}_4^{(l+1)} = \phi_l\left(\mathbf{h}_4^{(l)}, \mathbf{m}_4^{(l+1)}\right)$$



$$\mathbf{m}_4^{(l+1)} = \sum_{j \in \mathcal{N}_4} \mathbf{S}(4,j)\psi_l(\mathbf{h}_4^{(l)}, \mathbf{h}_j^{(l)})$$

$$\mathbf{h}_4^{(l+1)} = \phi_l\left(\mathbf{h}_4^{(l)}, \mathbf{m}_4^{(l+1)}\right)$$

Message Passing Neural Network (MPNN) Framework

MPNN

- Let G be a graph with a set of input features $\mathbf{X} \in \mathbb{R}^{N \times F_1} = [\mathbf{x}_1, \ldots, \mathbf{x}_N]^\mathsf{T}$

Message Passing Neural Network (MPNN) Framework

MPNN

- Let G be a graph with a set of input features $\mathbf{X} \in \mathbb{R}^{N \times F_1} = [\mathbf{x}_1, \ldots, \mathbf{x}_N]^{\mathsf{T}}$
- The output of a generic MPNN is defined as follows [Gilmer et al.(2017)]:

$$\mathbf{h}_i^{(l+1)} = \phi_l \left(\mathbf{h}_i^{(l)}, \sum_{j \in \mathcal{N}_i} \mathbf{S}(i,j) \psi_l(\mathbf{h}_i^{(l)}, \mathbf{h}_j^{(l)}) \right),$$

Message Passing Neural Network (MPNN) Framework

MPNN

- Let G be a graph with a set of input features $\mathbf{X} \in \mathbb{R}^{N \times F_1} = [\mathbf{x}_1, \ldots, \mathbf{x}_N]^\mathsf{T}$
- The output of a generic MPNN is defined as follows [Gilmer et al.(2017)]:

$$\mathbf{h}_i^{(l+1)} = \phi_l \left(\mathbf{h}_i^{(l)}, \sum_{j \in \mathcal{N}_i} \mathbf{S}(i,j) \psi_l(\mathbf{h}_i^{(l)}, \mathbf{h}_j^{(l)}) \right),$$

- where $\mathbf{H}^{(l)} \in \mathbb{R}^{N \times F_l} = [\mathbf{h}_1^{(l)}, \ldots, \mathbf{h}_N^{(l)}]^\mathsf{T}$ is the F_l-dimensional embeddings after l layers such that each $\mathbf{h}_i^{(l)} \in \mathbb{R}^{F_l}$ and $\mathbf{H}^{(1)} = \mathbf{X}$

- Let G be a graph with a set of input features $\mathbf{X} \in \mathbb{R}^{N \times F_1} = [\mathbf{x}_1, \ldots, \mathbf{x}_N]^\mathsf{T}$
- The output of a generic MPNN is defined as follows [Gilmer et al.(2017)]:

$$\mathbf{h}_i^{(l+1)} = \phi_l \left(\mathbf{h}_i^{(l)}, \sum_{j \in \mathcal{N}_i} \mathbf{S}(i,j) \psi_l(\mathbf{h}_i^{(l)}, \mathbf{h}_j^{(l)}) \right),$$

- where $\mathbf{H}^{(l)} \in \mathbb{R}^{N \times F_l} = [\mathbf{h}_1^{(l)}, \ldots, \mathbf{h}_N^{(l)}]^\mathsf{T}$ is the F_l-dimensional embeddings after l layers such that each $\mathbf{h}_i^{(l)} \in \mathbb{R}^{F_l}$ and $\mathbf{H}^{(1)} = \mathbf{X}$
- $\psi_l : \mathbb{R}^{F_l} \times \mathbb{R}^{F_l} \to \mathbb{R}^{F_l'}$ is a family of message functions, $\mathbf{S}$ is a shift operator, and $\phi_l : \mathbb{R}^{F_l} \times \mathbb{R}^{F_l'} \to \mathbb{R}^{F_{l+1}}$ is an update function

- Remember that the propagation rule of GCN is given by:

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

- Remember that the propagation rule of GCN is given by:

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

- We can rewrite GCN as a MPNN as follows:

$$\mathbf{h}_i^{(l+1)} = \sigma\left(\mathbf{W}^{\mathsf{T}} \sum_{j \in \mathcal{N}_i \cup \{i\}} \frac{a_{i,j}}{\sqrt{(d_j+1)(d_i+1)}} \mathbf{h}_j^{(l)}\right), \tag{3}$$

where d_i is the degree of node i

- Remember that the propagation rule of GCN is given by:

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

- We can rewrite GCN as a MPNN as follows:

$$\mathbf{h}_i^{(l+1)} = \sigma\left(\mathbf{W}^{\mathsf{T}} \sum_{j \in \mathcal{N}_i \cup \{i\}} \frac{a_{i,j}}{\sqrt{(d_j+1)(d_i+1)}} \mathbf{h}_j^{(l)}\right), \tag{3}$$

  where d_i is the degree of node i

- Notice that we are doing a weighted sum of the embeddings in the neighborhood of node i ($\mathcal{N}_i$), plus the current embedding of node i ($\mathcal{N}_i \cup \{i\}$). Then, we apply a MLP ($\mathbf{W}^{\mathsf{T}}$) and an activation function $\sigma(\cdot)$

- We cannot process graphs that contains edge features
- When we compute the message passing mechanism, we assume that the importance (weight) of each node is given by $a_{i,j}/\sqrt{(d_j+1)(d_i+1)}$
- We are assuming that the importance of each node in the message passing function is given by the structure of the graph and the predefined weights $a_{i,j}$
- In some applications, the importance of each node in the message passing step could depend of other aspects

Graph Attention Network (GAT)

- If you look at GCN with the MPNN setting, we know that the importance of each node is given by $a_{i,j}/\sqrt{(d_j+1)(d_i+1)}$ so that our layer is given by:

$$\mathbf{h}_i^{(l+1)} = \sigma\left(\mathbf{W}^\mathsf{T} \sum_{j \in \mathcal{N}_i \cup \{i\}} \frac{a_{i,j}}{\sqrt{(d_j+1)(d_i+1)}} \mathbf{h}_j^{(l)}\right), \tag{4}$$

i.e., the weight is determined by the degree of the nodes

- If you look at GCN with the MPNN setting, we know that the importance of each node is given by $a_{i,j}/\sqrt{(d_j+1)(d_i+1)}$ so that our layer is given by:

$$\mathbf{h}_i^{(l+1)} = \sigma\left(\mathbf{W}^\mathsf{T} \sum_{j \in \mathcal{N}_i \cup \{i\}} \frac{a_{i,j}}{\sqrt{(d_j+1)(d_i+1)}} \mathbf{h}_j^{(l)}\right), \tag{4}$$

  i.e., the weight is determined by the degree of the nodes

- General idea of GAT: we want to replace this fixed weight with some function $\alpha_{i,j} = f(\mathbf{h}_i, \mathbf{h}_j)$

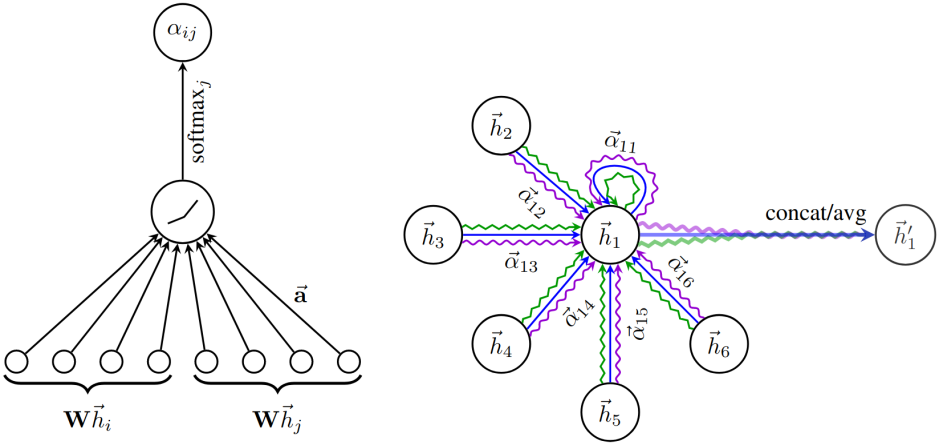
- If you look at GCN with the MPNN setting, we know that the importance of each node is given by $a_{i,j}/\sqrt{(d_j+1)(d_i+1)}$ so that our layer is given by:

$$\mathbf{h}_i^{(l+1)} = \sigma\left(\mathbf{W}^\mathsf{T} \sum_{j \in \mathcal{N}_i \cup \{i\}} \frac{a_{i,j}}{\sqrt{(d_j+1)(d_i+1)}} \mathbf{h}_j^{(l)}\right), \tag{4}$$

  i.e., the weight is determined by the degree of the nodes
- General idea of GAT: we want to replace this fixed weight with some function $\alpha_{i,j} = f(\mathbf{h}_i, \mathbf{h}_j)$
- This new function is learnable, so the weight depends on more than the number of neighbors

Graph Attention Network (GAT)

GAT



Graph Attention Network (GAT). **Source:** [Veličković et al.(2018)]

- We're gonna adopt temporarily the same notation as in [Veličković et al.(2018)]

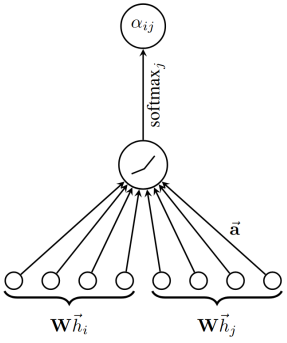
- We're gonna adopt temporarily the same notation as in [Veličković et al.(2018)]
- Let $\mathbf{h} = \{\vec{h}_1, \vec{h}_2, \ldots, \vec{h}_N\}$ be the node features where $\vec{h}_i \in \mathbb{R}^F$

- We're gonna adopt temporarily the same notation as in [Veličković et al.(2018)]
- Let $\mathbf{h} = \{\vec{h}_1, \vec{h}_2, \ldots, \vec{h}_N\}$ be the node features where $\vec{h}_i \in \mathbb{R}^F$
- The layer produces a node feature (embedding) $\mathbf{h}' = \{\vec{h}'_1, \vec{h}'_2, \ldots, \vec{h}'_N\}$ where $\vec{h}'_i \in \mathbb{R}^{F'}$

- We're gonna adopt temporarily the same notation as in [Veličković et al.(2018)]
- Let $\mathbf{h} = \{\vec{h}_1, \vec{h}_2, \ldots, \vec{h}_N\}$ be the node features where $\vec{h}_i \in \mathbb{R}^F$
- The layer produces a node feature (embedding) $\mathbf{h}' = \{\vec{h}'_1, \vec{h}'_2, \ldots, \vec{h}'_N\}$ where $\vec{h}'_i \in \mathbb{R}^{F'}$
- The first step is to transform every embedding $\vec{h}_i$ with a linear transformation, using a weight matrix $\mathbf{W} \in \mathbb{R}^{F' \times F}$, *i.e.*, $\mathbf{W}\vec{h}_i$

- We're gonna adopt temporarily the same notation as in [Veličković et al.(2018)]
- Let $\mathbf{h} = \{\vec{h}_1, \vec{h}_2, \ldots, \vec{h}_N\}$ be the node features where $\vec{h}_i \in \mathbb{R}^F$
- The layer produces a node feature (embedding) $\mathbf{h}' = \{\vec{h}'_1, \vec{h}'_2, \ldots, \vec{h}'_N\}$ where $\vec{h}'_i \in \mathbb{R}^{F'}$
- The first step is to transform every embedding $\vec{h}_i$ with a linear transformation, using a weight matrix $\mathbf{W} \in \mathbb{R}^{F' \times F}$, *i.e.*, $\mathbf{W}\vec{h}_i$
- Now we have linear transformation from the input node features, and the next step is to compute the attention mechanism

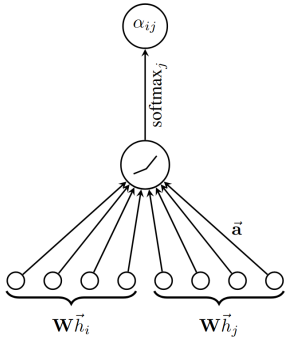
- The attention mechanism is computed as the $\mathrm{softmax}$ function between every pair of connected nodes in the graph



- The attention mechanism is computed as the $\mathrm{softmax}$ function between every pair of connected nodes in the graph
- The attention coefficient is given by:

$$\alpha_{i,j} = \frac{\exp\left(\mathrm{LeakyReLU}\left(\vec{\mathbf{a}}^{\top}[\mathbf{W}\vec{h}_i, \mathbf{W}\vec{h}_j]\right)\right)}{\sum_{k\in\mathcal{N}_i}\exp\left(\mathrm{LeakyReLU}\left(\vec{\mathbf{a}}^{\top}[\mathbf{W}\vec{h}_i, \mathbf{W}\vec{h}_k]\right)\right)}$$

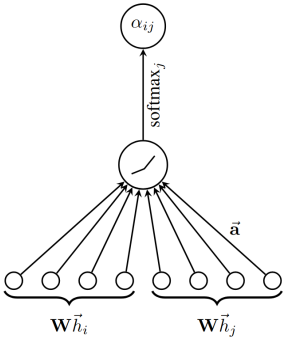
where $\vec{\mathbf{a}} \in \mathbb{R}^{2F'}$, and $[\cdot,\cdot]$ is regular concatenation



- The attention mechanism is computed as the $\mathrm{softmax}$ function between every pair of connected nodes in the graph
- The attention coefficient is given by:

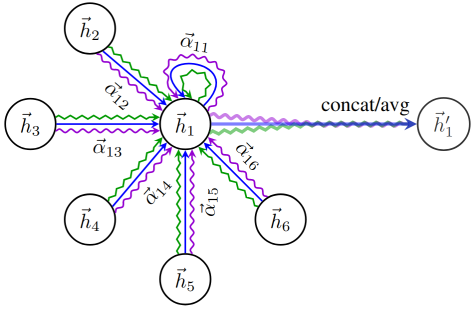
$$\alpha_{i,j} = \frac{\exp\left(\mathrm{LeakyReLU}\left(\vec{\mathbf{a}}^{\top}[\mathbf{W}\vec{h}_i, \mathbf{W}\vec{h}_j]\right)\right)}{\sum_{k\in\mathcal{N}_i}\exp\left(\mathrm{LeakyReLU}\left(\vec{\mathbf{a}}^{\top}[\mathbf{W}\vec{h}_i, \mathbf{W}\vec{h}_k]\right)\right)}$$

  where $\vec{\mathbf{a}} \in \mathbb{R}^{2F'}$, and $[\cdot,\cdot]$ is regular concatenation
- Now, we just need to aggregate all embeddings in the neighborhood to get the output of our layer



- Once we have all our attention coefficient we get the output as:

$$\vec{h}_i' = \sigma\left(\sum_{j\in\mathcal{N}_i} \alpha_{i,j}\mathbf{W}_t\vec{h}_j\right)$$

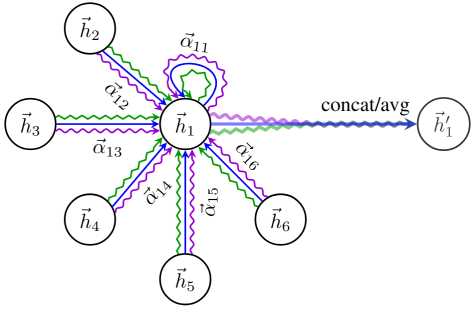


- Once we have all our attention coefficient we get the output as:

$$\vec{h}'_i = \sigma\left(\sum_{j\in\mathcal{N}_i} \alpha_{i,j}\mathbf{W}_t\vec{h}_j\right)$$

- We can also employ *multi-head attention*, where we apply k independent graph attention layers and then we aggregate them all (concatenation, average, etc):

$$\vec{h}'_i = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j\in\mathcal{N}_i} \alpha^k_{i,j}\mathbf{W}^k\vec{h}_j\right)$$

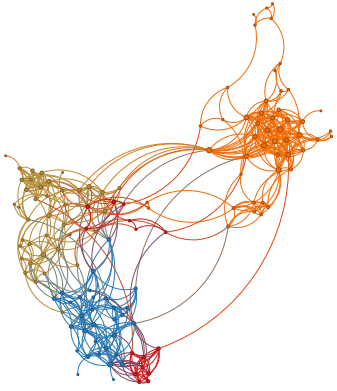


Summary of GCN and GAT

- GCNs aggregate information using **fixed weights**
- The weights depend directly on $\mathbf{A}$
- Useful for homophilous graphs and scaling up
- GATs aggregate information with **implicit weights** (via attention)
- The weights are computed with an attention mechanism
- Useful as *middle ground* w.r.t. capacity and scale

Limitations in GNNs

- Since the commonly used GNNs are approximations of the graph filtering operation, we have some limitations



Limitations in GNNs

Homophily and Heterophily

- Since the commonly used GNNs are approximations of the graph filtering operation, we have some limitations
- We usually get good performance in graphs where homophily is high when using MPNNs

- Since the commonly used GNNs are approximations of the graph filtering operation, we have some limitations
- We usually get good performance in graphs where homophily is high when using MPNNs
- High homophily means that nodes in a neighborhood share the same label (as the graph on the right)

- There are several formal definitions of homophily in the literature, the simplest one is:

$$H(G) = \frac{1}{|\mathcal{V}|} \sum_{i \in \mathcal{V}} \frac{|\mathcal{N}_i^s|}{|\mathcal{N}_i|},$$

where $\mathcal{N}_i^s$ is the set of nodes that share the same labels as the ith node

- There are several formal definitions of homophily in the literature, the simplest one is:

$$H(G) = \frac{1}{|\mathcal{V}|} \sum_{i \in \mathcal{V}} \frac{|\mathcal{N}_i^s|}{|\mathcal{N}_i|},$$

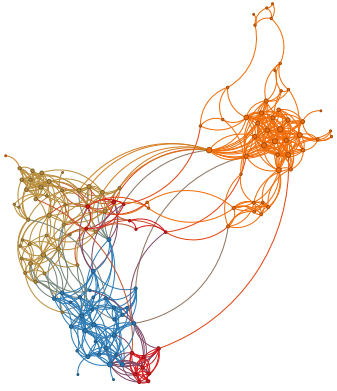
  where $\mathcal{N}_i^s$ is the set of nodes that share the same labels as the ith node
- In the hypothetic case where $|\mathcal{N}_i^s| = |\mathcal{N}_i|$ for all nodes, $H(G) = 1$

- There are several formal definitions of homophily in the literature, the simplest one is:

$$H(G) = \frac{1}{|\mathcal{V}|} \sum_{i \in \mathcal{V}} \frac{|\mathcal{N}_i^s|}{|\mathcal{N}_i|},$$

  where $\mathcal{N}_i^s$ is the set of nodes that share the same labels as the ith node
- In the hypothetic case where $|\mathcal{N}_i^s| = |\mathcal{N}_i|$ for all nodes, $H(G) = 1$
- $H(G) \to 1$ when we have graphs with strong homophily, while $H(G) \to 0$ for graphs with strong heterophily

- Intuitively, common GNNs perform well in scenarios of strong homophily because we usually aggregate information from the neighborhood of each node (that's the case for GCN and GAT)



- Intuitively, common GNNs perform well in scenarios of strong homophily because we usually aggregate information from the neighborhood of each node (that's the case for GCN and GAT)
- More formally, since common GNNs are an approximation of the graph filtering operation, we can only compute low-pass filters.

- Intuitively, common GNNs perform well in scenarios of strong homophily because we usually aggregate information from the neighborhood of each node (that's the case for GCN and GAT)
- More formally, since common GNNs are an approximation of the graph filtering operation, we can only compute low-pass filters.
- The key message for you is that if your GNN is not performing well in your data, low homophily could be the reason

- You may have seen in other courses that deep neural networks perform better than shallow ones, from there the name of Deep Learning

Limitations in GNNs

Over-smoothing

- You may have seen in other courses that deep neural networks perform better than shallow ones, from there the name of Deep Learning
- This is (still) not necessarily true in GNNs, and one of the problems we've found in the literature is over-smoothing

Limitations in GNNs

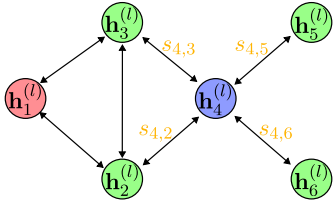
Over-smoothing

- You may have seen in other courses that deep neural networks perform better than shallow ones, from there the name of Deep Learning
- This is (still) not necessarily true in GNNs, and one of the problems we've found in the literature is over-smoothing
- At a very high level, over-smoothing means that as we stack more layers in our GNNs, the nodes embeddings start to converge to the same values, washing away all the feature information (*i.e.*, we loose the discriminative power of our GNN)

Limitations in GNNs

Over-smoothing

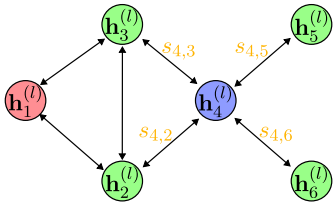
- Intuitively, we can see the message-passing algorithm as smoothing the neighborhood features (we're aggregating the information in the neighborhood, making it homogeneous)



Limitations in GNNs

Over-smoothing

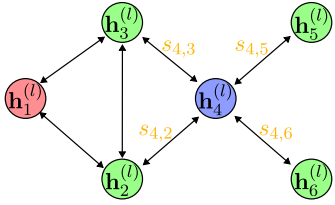
- Intuitively, we can see the message-passing algorithm as smoothing the neighborhood features (we're aggregating the information in the neighborhood, making it homogeneous)
- Performing this operation many times, *i.e.*, stacking several layers, will over-smooth the neighborhood nodes



Limitations in GNNs

Over-smoothing

- Intuitively, we can see the message-passing algorithm as smoothing the neighborhood features (we're aggregating the information in the neighborhood, making it homogeneous)
- Performing this operation many times, *i.e.*, stacking several layers, will over-smooth the neighborhood nodes
- If you like theory, this can be explained as interpreting the graph as a Markov chain and looking at the mixing time of this chain, *i.e.*, the number of steps (layers) required to reach convergence to the stationary distribution

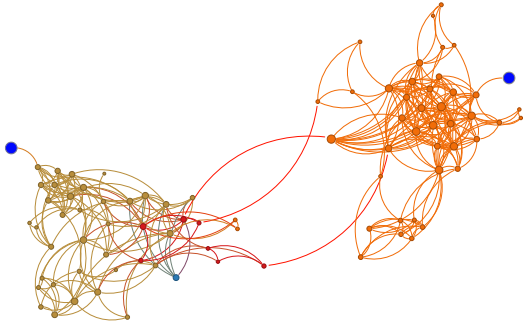


- We still do not know if we indeed need deep GNNs, and the answer to this question is an active topic of research

- We still do not know if we indeed need deep GNNs, and the answer to this question is an active topic of research
- The hints we have is that we need deep GNNs where there are long-range interactions between nodes in the graph

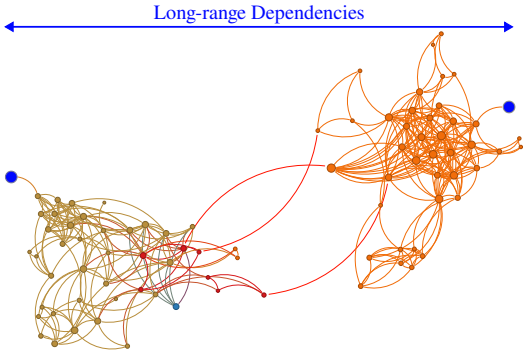
Limitations in GNNs

Over-squashing



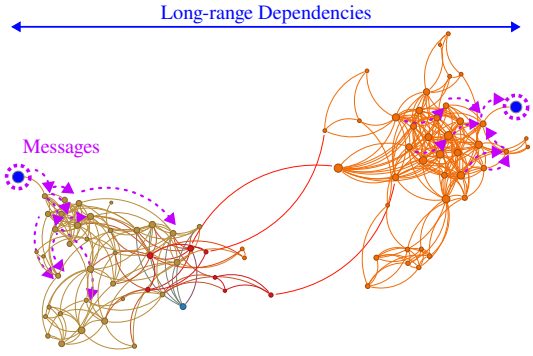
Limitations in GNNs

Over-squashing



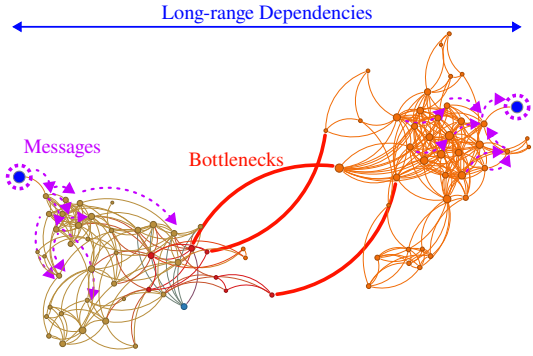
Limitations in GNNs

Over-squashing



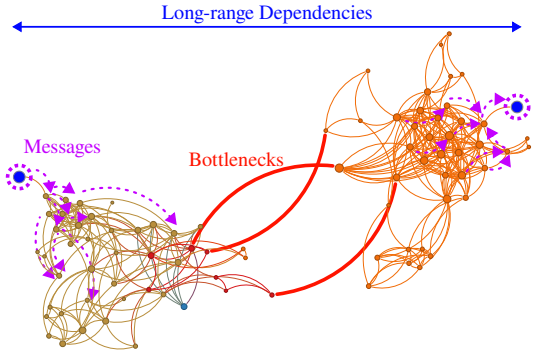
Limitations in GNNs

Over-squashing



Limitations in GNNs

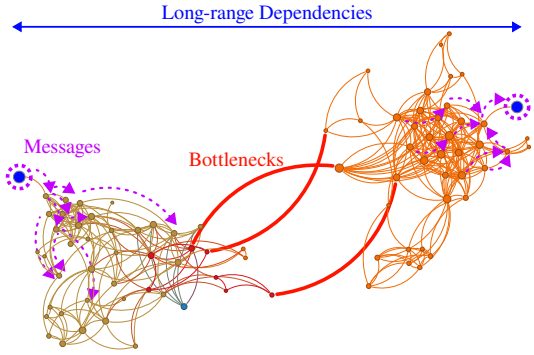
Over-squashing



- When we have graphs with **long-range dependencies**, and we want to build deep graph neural networks we face **over-smoothing** and **over-squashing**

Limitations in GNNs

Over-squashing



- When we have graphs with **long-range dependencies**, and we want to build deep graph neural networks we face **over-smoothing** and **over-squashing**
- If you encounter over-smoothing or over-squashing in your projects, let's see some practical solutions to it

- One simple way to avoid over-smoothing is simply to prevent the embeddings to become the same

- One simple way to avoid over-smoothing is simply to prevent the embeddings to become the same
- This can be achieved with a normalization layer called PairNorm [Zhao and Akoglu(2020)]

- One simple way to avoid over-smoothing is simply to prevent the embeddings to become the same
- This can be achieved with a normalization layer called PairNorm [Zhao and Akoglu(2020)]
- PairNorm is composed of centering and scaling of the data:

$$\mathbf{x}_i^c = \mathbf{x}_i - \frac{1}{N}\sum_{i\in\mathcal{V}} \mathbf{x}_i \quad \text{(Centering)},$$

$$\mathbf{x}_i' = s\frac{\mathbf{x}_i^c}{\sqrt{\frac{1}{N}\sum_{i\in\mathcal{V}} \|\mathbf{x}_i^c\|_2^2}} \quad \text{(Scaling)},$$

where s is a scaling factor

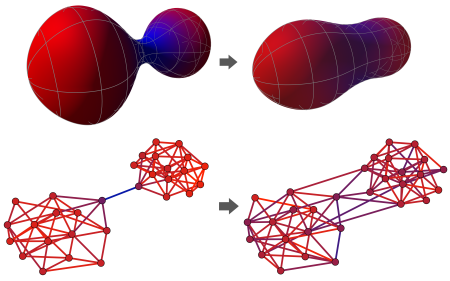
- Another effective method to fight over-smoothing is DropEdge [Rong et al.(2020)]

- Another effective method to fight over-smoothing is DropEdge [Rong et al.(2020)]
- The idea of DropEdge is quite simple, you randomly drop a certain amount of edges from the graph while training the GNN

- Another effective method to fight over-smoothing is DropEdge [Rong et al.(2020)]
- The idea of DropEdge is quite simple, you randomly drop a certain amount of edges from the graph while training the GNN
- To explain why DropEdge reduces over-smoothing we should study the spectral gap in graphs and the mixing time in Markov chains (which is beyond the scope of this course)

- Another effective method to fight over-smoothing is DropEdge [Rong et al.(2020)]
- The idea of DropEdge is quite simple, you randomly drop a certain amount of edges from the graph while training the GNN
- To explain why DropEdge reduces over-smoothing we should study the spectral gap in graphs and the mixing time in Markov chains (which is beyond the scope of this course)
- Besides fighting over-smoothing, DropEdge also reduces the computational complexity for training the GNN since we have to pass less messages during the training (the computational complexity of popular GNNs is proportional to $\mathcal{O}(|\mathcal{E}|)$)

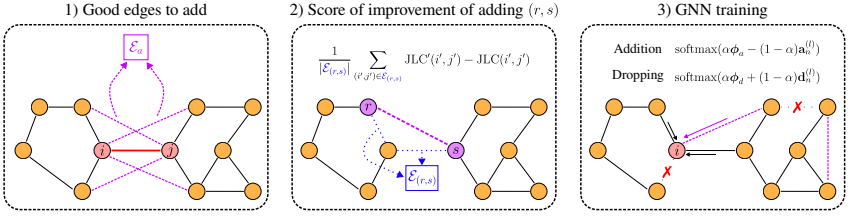
- We know from [Topping et al.(2022)] the relationship between over-squashing and the Ricci flow curvature
- More precisely we know that if we have positive Ricci curvature everywhere, we ensure that the receptive field of each node in a deep GNN will be polynomial in the hop distance rather than exponential (no over-squashing)
- We can achieve this by rewiring the graph
- We look for the most curved edges in the graph, and add edges in the neighborhood



Source: [Topping et al.(2022)]

Limitations in GNNs

Solving Over-squashing (Self-study)



- In 2023, we proposed an algorithm to alleviate over-squashing called SJLR [Giraldo et al.(2023)]
- SJLR first computes a bank of potential good edges to add $\mathcal{E}_a$, where the JLC metric is calculated (for all $(i, j) \in \mathcal{E}$) and sorted (in asc. order)
- Therefore, we associate a score to every edge $(r, s) \in \mathcal{E}_a$, which is computed as the average improvement of curvature from adding that edge (r, s) to the graph
- Finally, during the GNN training for each layer we use the JLC metrics and Euclidean distances to add and remove stochastically some edges

Pytorch Geometric (PyG)

Pytorch Geometric (PyG)

Introduction of Pytorch Geometric https://pyg.org/

- We're going to use PyTorch Geometric (PyG) for our TP
- PyG is an open-source library built upon PyTorch to easily write and train GNNs for a wide range of applications related to structured data
- Since PyG is built upon PyTorch, we can easily transfer the knowledge we have in PyTorch to PyG
- PyG was initiated by a Ph.D. student from TU Dortmund University, and now it's widely used by several startups and big companies like Microsoft, Nvidia, Spotify, AstraZeneca, Airbus, Lyft, and Intel, among others

Pytorch Geometric (PyG)

Creating a GCN with PyG

```
import torch
import torch.nn.functional as functional

from torch_geometric.nn import GCNConv


class GCN(torch.nn.Module):
    def __init__(self, in_channels, out_channels, hidden_units, number_layers):
        super(GCN, self).__init__()

        self.convs = torch.nn.ModuleList()
        if number_layers > 1:
            self.convs.append(GCNConv(in_channels, hidden_units))
        for _ in range(number_layers - 2):
            self.convs.append(GCNConv(hidden_units, hidden_units))
        if number_layers > 1:
            self.convs.append(GCNConv(hidden_units, out_channels))
        else:
            self.convs.append(GCNConv(in_channels, out_channels))

    def forward(self, data):
        x, edge_index, edge_attr = data.x, data.edge_index, data.edge_attr
        for i, conv in enumerate(self.convs[:-1]):
            x = conv(x, edge_index, edge_weight=edge_attr)
            x = functional.relu(x)
        x = self.convs[-1](x, edge_index, edge_attr=edge_attr)
        return x.softmax(dim=-1)
```

Pytorch Geometric (PyG)

Creating a GAT with PyG

```
import torch
import torch.nn.functional as functional

from torch_geometric.nn import GATConv


class GAT(torch.nn.Module):
    def __init__(self, in_channels, out_channels, hidden_units, number_layers):
        super(GAT, self).__init__()

        self.convs = torch.nn.ModuleList()
        if number_layers > 1:
            self.convs.append(GATConv(in_channels, hidden_units, heads=2, concat=False))
            # The multi-head attentions are averaged instead of concatenated
        for _ in range(number_layers - 2):
            self.convs.append(GATConv(hidden_units, hidden_units, heads=2, concat=False))
        if number_layers > 1:
            self.convs.append(GATConv(hidden_units, out_channels, heads=2, concat=False))
        else:
            self.convs.append(GATConv(in_channels, out_channels, heads=2, concat=False))

    def forward(self, data):
        x, edge_index, edge_attr = data.x, data.edge_index, data.edge_attr
        for i, conv in enumerate(self.convs[:-1]):
            x = conv(x, edge_index, edge_weight=edge_attr)
            x = functional.relu(x)
        x = self.convs[-1](x, edge_index, edge_attr=edge_attr)
        return x.softmax(dim=-1)
```

Main takeaways

- Deep learning on graphs tries to leverage the power of neural networks into graphs

Main takeaways

- Deep learning on graphs tries to leverage the power of neural networks into graphs
- We want to build NNs for graph data that scale well

Main takeaways

- Deep learning on graphs tries to leverage the power of neural networks into graphs
- We want to build NNs for graph data that scale well
- We know that CNNs scale well, so we try to do graph convolutions

Main takeaways

- Deep learning on graphs tries to leverage the power of neural networks into graphs
- We want to build NNs for graph data that scale well
- We know that CNNs scale well, so we try to do graph convolutions
- We can build GNNs by composing layers with graph filters

Main takeaways

- Deep learning on graphs tries to leverage the power of neural networks into graphs
- We want to build NNs for graph data that scale well
- We know that CNNs scale well, so we try to do graph convolutions
- We can build GNNs by composing layers with graph filters
- Computing a proper graph filter could be quite expensive

Main takeaways

- We need to build efficient GNNs to use them in real-world applications

Main takeaways

- We need to build efficient GNNs to use them in real-world applications
- We can use GNNs for node classification, graph classification, and link classification (or prediction)

Main takeaways

- We need to build efficient GNNs to use them in real-world applications
- We can use GNNs for node classification, graph classification, and link classification (or prediction)
- The graph convolutional filter is not scalable, we need to do some approximations

Main takeaways

- We need to build efficient GNNs to use them in real-world applications
- We can use GNNs for node classification, graph classification, and link classification (or prediction)
- The graph convolutional filter is not scalable, we need to do some approximations
- GCN is quite effective and fast, it should be one of the first architectures you test before moving to more complex stuff

Main takeaways

- We need to build efficient GNNs to use them in real-world applications
- We can use GNNs for node classification, graph classification, and link classification (or prediction)
- The graph convolutional filter is not scalable, we need to do some approximations
- GCN is quite effective and fast, it should be one of the first architectures you test before moving to more complex stuff
- We still have plenty of limitations and open problems in GNNs–a lot of things to explore if you like research ;)

Activity

This lecture is quite important since it lays down the foundation of what follows in the course. Please take the time to reflect on what we learned today, and on a piece of paper, please:

1. Write the most important things you learned today
2. If you have questions about the lecture, you can write them and I'll try to answer a couple in the following lectures
3. Please sign it if you want to (participation), and give it to me before leaving

Thank you!

Jhony H. Giraldo: jhony.giraldo@telecom-paris.fr
https://jhonygiraldo.github.io/

References I

M. Defferrard et al.
Convolutional neural networks on graphs with fast localized spectral filtering.
In *Advances in Neural Information Processing Systems*, 2016.

J. Gilmer et al.
Neural message passing for quantum chemistry.
In *International Conference on Machine Learning*, 2017.

J. H. Giraldo et al.
On the trade-off between over-smoothing and over-squashing in deep graph neural networks.
In *ACM International Conference on Information and Knowledge Management*, 2023.

References II


T. N. Kipf and M. Welling.
Semi-supervised classification with graph convolutional networks.
In *International Conference on Learning Representations*, 2017.


Y. Rong et al.
DropEdge: Towards deep graph convolutional networks on node classification.
In *International Conference on Learning Representations*, 2020.


Jake Topping et al.
Understanding over-squashing and bottlenecks on graphs via curvature.
In *International Conference on Learning Representations*, 2022.

References III


P. Veličković et al.
Graph attention networks.
In *International Conference on Learning Representations*, 2018.


Lingxiao Zhao and Leman Akoglu.
PairNorm: Tackling oversmoothing in GNNs.
In *International Conference on Learning Representations*, 2020.