Graph Neural Networks

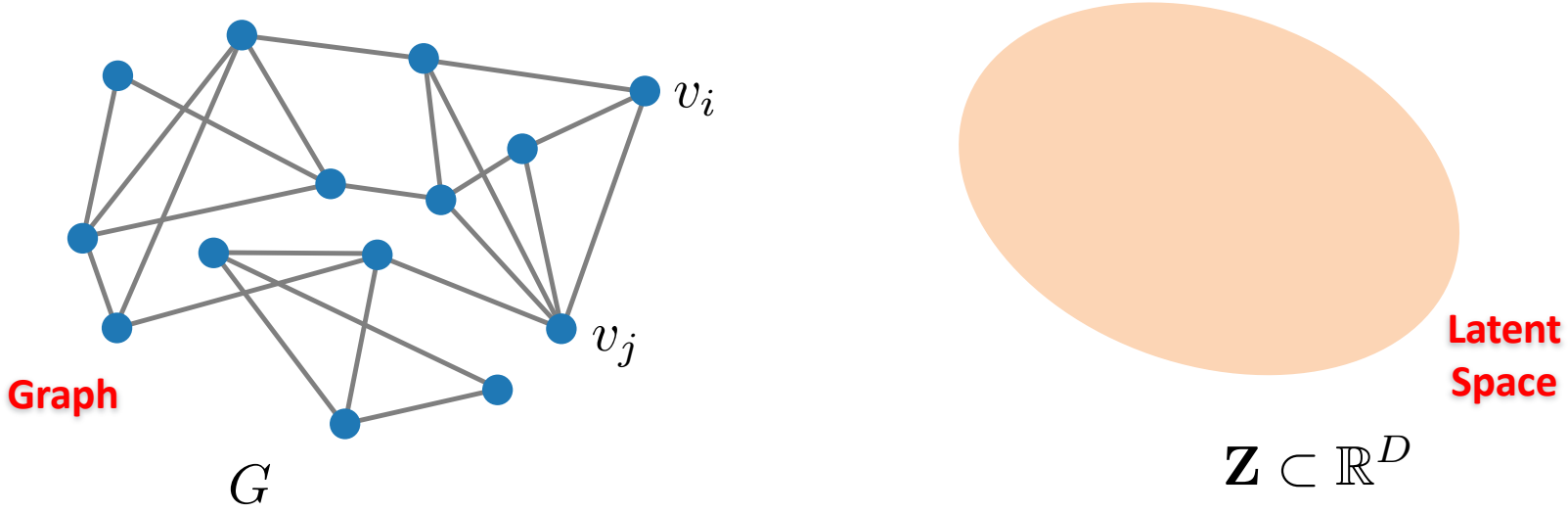
ACMS 80770: Deep Learning with Graphs

Instructor: Navid Shervani-Tabar

Department of Applied and Comp Math and Stats

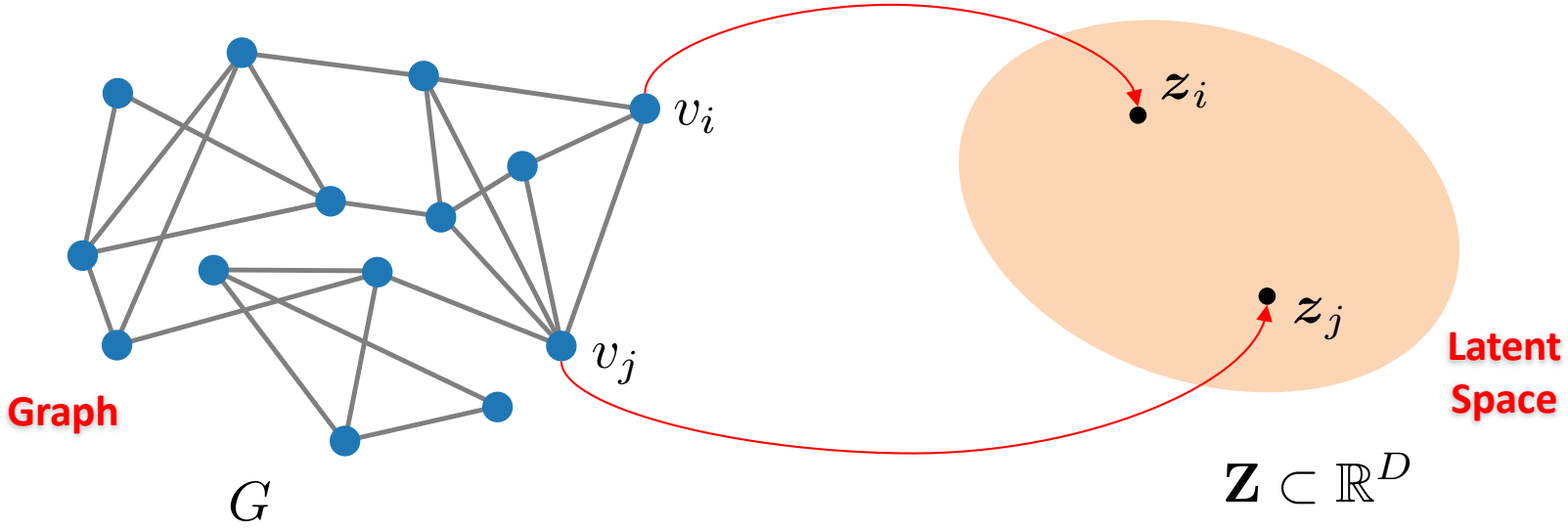
Node Embedding

- In the previous lectures, we discussed learning node feature representations.
- Node feature vectors $\boldsymbol{z}_i$, also called a **node embedding**, are low-dimensional vectors that represent the node on a low-dimensional space.



Node Embedding

- In the previous lectures, we discussed learning node feature representations.
- Node feature vectors $\boldsymbol{z}_i$, also called a **node embedding**, are low-dimensional vectors that represent the node on a low-dimensional space.



Shallow Embedding

- We have so far relied on **shallow embedding** method to learn node representations.
- In this method, an embedding **look up** takes the node index i as input and returns an embedding $\boldsymbol{z}_i \in \mathbb{R}^D$ for the node.

$$\mathbf{Z} = \begin{bmatrix} \begin{bmatrix} z_{11} \\ \vdots \\ z_{1D} \end{bmatrix} & \cdots & \begin{bmatrix} z_{|V|1} \\ \vdots \\ z_{|V|D} \end{bmatrix} \end{bmatrix}$$

- We **directly** learn node embeddings as model parameters.
- Model parameters are $|V|$ vectors of dimension D.

Graph ML Tasks

- The learned node embeddings can be used in **node-level** tasks.
  - Node classification
- One can construct a feature vectors $\boldsymbol{z}_{(i,j)}$ corresponding to edges (v_i, v_j) by integrating pairs of node embeddings $\boldsymbol{z}_i$ and $\boldsymbol{z}_j$, to perform **edge-level** tasks.
  - Link prediction
- Similarly, dummy node approach can be used to generate **graph** or **subgraph-level** embeddings.
  - Graph-level tasks such as graph regression and graph classification.

Shallow Embedding

- Shallow embedding methods have a few shortcomings:
  - Embedding $\boldsymbol{Z}$ is learned as **model parameter** and nodes do not **share parameters** within the encoder.
  - They are **transductive** and can't learn embedding on nodes that were not seen during the training.
  - They don't leverage **node attributes**.

Graph-based Neural Network

- Shallow embedding methods have a few shortcomings:
  - Embedding $\boldsymbol{Z}$ is learned as **model parameter** and nodes do not **share parameters** within the encoder.
  - They are **transductive** and can't learn embedding on nodes that were not seen during the training.
  - They don't leverage **node attributes**.
- More **sophisticated** encoders based on **deep learning** models alleviate these limitations.
- Graph neural networks incorporate information from the **structure** of the graph, as well as the **attributes** on the nodes of the graph.

Graph-based Neural Network

- Shallow embedding methods have a few shortcomings:
  - Embedding $\boldsymbol{Z}$ is learned as **model parameter** and nodes do not **share parameters** within the encoder.
  - They are **transductive** and can't learn embedding on nodes that were not seen during the training.
  - They don't leverage **node attributes**.
- More **sophisticated** encoders based on **deep learning** models alleviate these limitations.
- Graph neural networks incorporate information from the **structure** of the graph, as well as the **attributes** on the nodes of the graph.
- To design a graph-based neural network, we need to define a **layer-wise propagation rule** for graph structured data.

Fully-Connected Layer

- One approach could be to pass a vector representation $\boldsymbol{x}_G$ of the graph G to a simple fully connected network f.
- To pass a graph to a neural network, we need to convert the graph to a vector representation.
- However, it is not clear how to construct this vector representation.
- One can use a flattened adjacency matrix as input

$$\mathbf{x}_G = \mathbf{A}_1 \oplus \mathbf{A}_2 \oplus \cdots \oplus \mathbf{A}_{|V|}$$

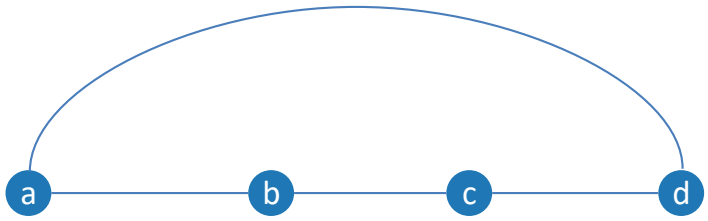
Permutation

- To pass a graph to a neural network, we need to convert the graph to a vector representation.
- However, it is not clear how to construct this vector representation.
- One can use a flattened adjacency matrix as input

$$\mathbf{x}_G = \mathbf{A}_1 \oplus \mathbf{A}_2 \oplus \cdots \oplus \mathbf{A}_{|V|}$$

- Consider a graph G, with its adjacency matrix defined as

$$\mathbf{A} = \begin{array}{c} \begin{matrix} a & b & c & d \end{matrix} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$



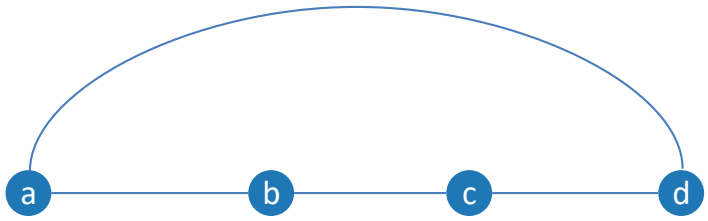
Permutation

- To pass a graph to a neural network, we need to convert the graph to a vector representation.
- However, it is not clear how to construct this vector representation.
- One can use a flattened adjacency matrix as input

$$\mathbf{x}_G = \mathbf{A}_1 \oplus \mathbf{A}_2 \oplus \cdots \oplus \mathbf{A}_{|V|}$$

- Consider a graph G, with its adjacency matrix defined as

$$\mathbf{A} = \begin{array}{c} \begin{matrix} a & b & c & d \end{matrix} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$



$$\mathbf{x}_G = (0,1,0,1,1,0,1,0,0,1,0,1,1,0,1,0)$$

Permutation

- Let $\boldsymbol{P}$ define a permutation matrix

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

- We can construct a permuted graph G' with adjacency matrix.

Permutation

- Let $\boldsymbol{P}$ define a permutation matrix

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

- We can construct a permuted graph G' with adjacency matrix

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

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

Permutation

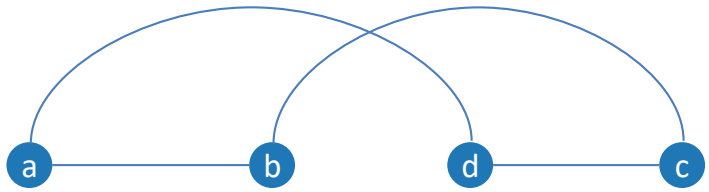
- Let $\boldsymbol{P}$ define a permutation matrix

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

- We can construct a permuted graph G' with adjacency matrix

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

$$\mathbf{A}' = \begin{matrix} a & b & d & c \end{matrix} \\ \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$



Permutation

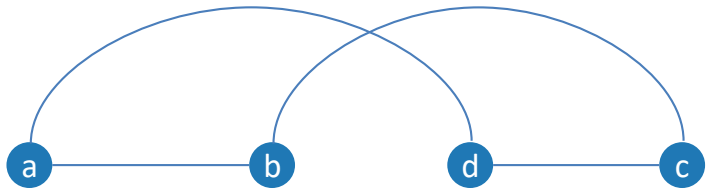
- Let $\boldsymbol{P}$ define a permutation matrix

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

- We can construct a permuted graph G' with adjacency matrix

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

$$\mathbf{A}' = \begin{matrix} a & b & d & c \end{matrix} \\ \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$



$$\mathbf{x}_{G'} = (0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0)$$

Permutation Invariance and Equivariance

- There are 24 different permutations of a simple graph of size 4.

$$\mathbf{x}_G = (0,1,0,1,1,0,1,0,0,1,0,1,1,0,1,0)$$

$$\mathbf{x}_{G'} = (0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0)$$

- The output $f(\boldsymbol{x}_G)$ constructed in this way depends on ordering of the nodes.
- To avoid this, the propagation rule f should satisfy either
  - **Permutation invariance:**
  - **Permutation equivariance:**

Permutation Invariance and Equivariance

- There are 24 different permutations of a simple graph of size 4.

$$\mathbf{x}_G = (0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0)$$

$$\mathbf{x}_{G'} = (0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0)$$

- The output $f(\boldsymbol{x}_G)$ constructed in this way depends on ordering of the nodes.
- To avoid this, the propagation rule f should satisfy either
  - **Permutation invariance:**
  - Output is not affected by the permutation of the input graph

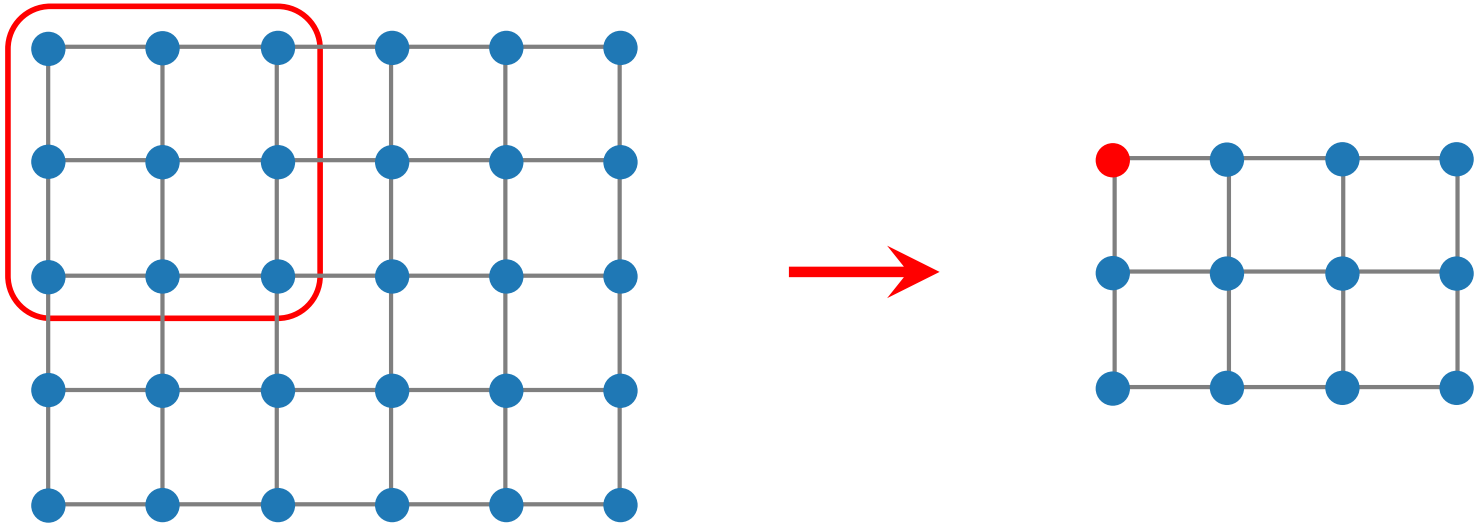
$$f(\mathbf{P}\mathbf{A}\mathbf{P}^T) = f(\mathbf{A})$$

  - **Permutation equivariance:**
  - Output is permuted in the same order as the input matrix.

$$f(\mathbf{P}\mathbf{A}\mathbf{P}^T) = \mathbf{P}f(\mathbf{A})$$

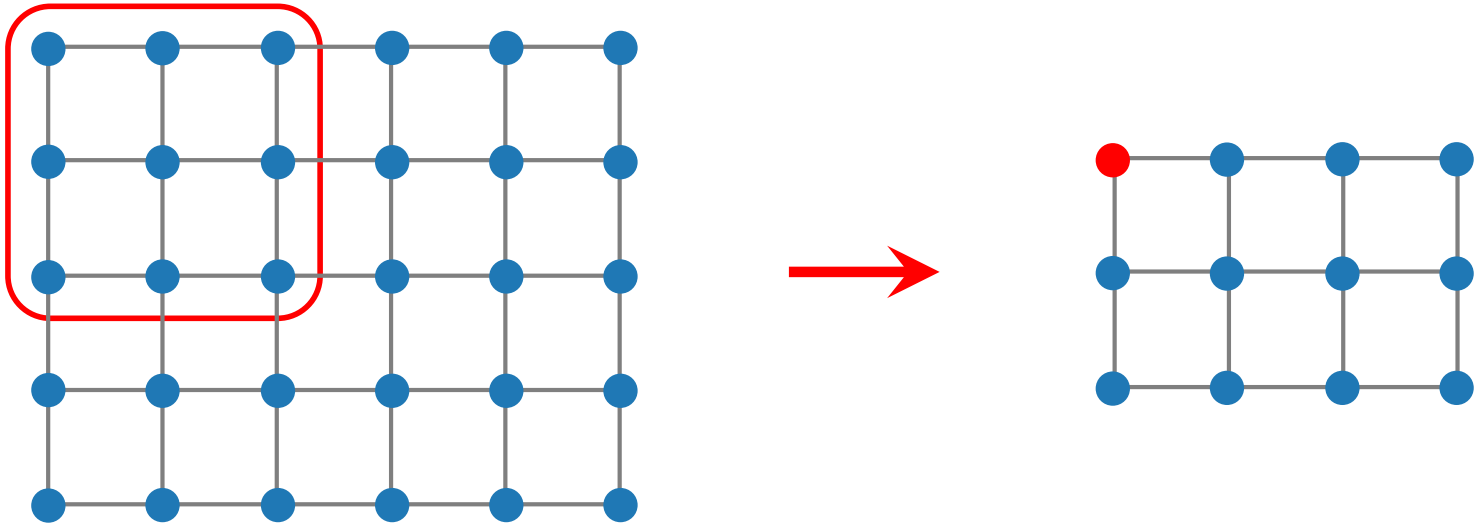
Convolutional Layer

- Another approach could be to use a **convolutional** neural network.
- In this model, a kernel or **filter** is convolved with grid structured data to **extract** features from the input signal.



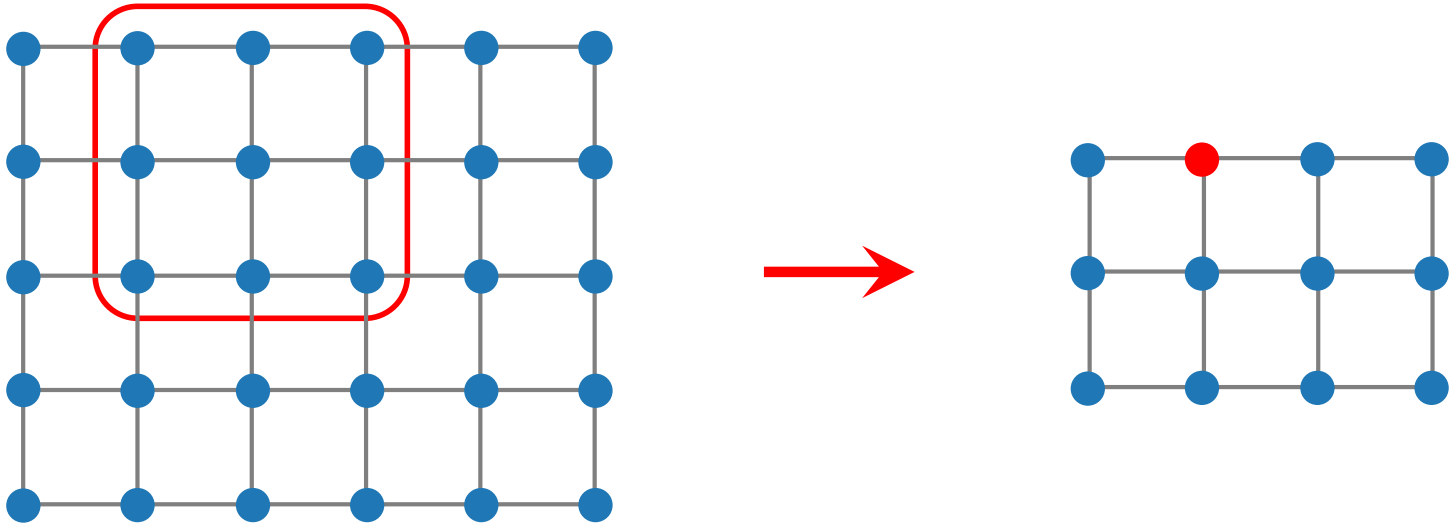
Convolutional Layer

- Another approach could be to use a **convolutional** neural network.
- In this model, a kernel or **filter** is convolved with grid structured data to **extract** features from the input signal.
- Convolution on **structured grid** performed by **sliding** the filter over the grid structures.



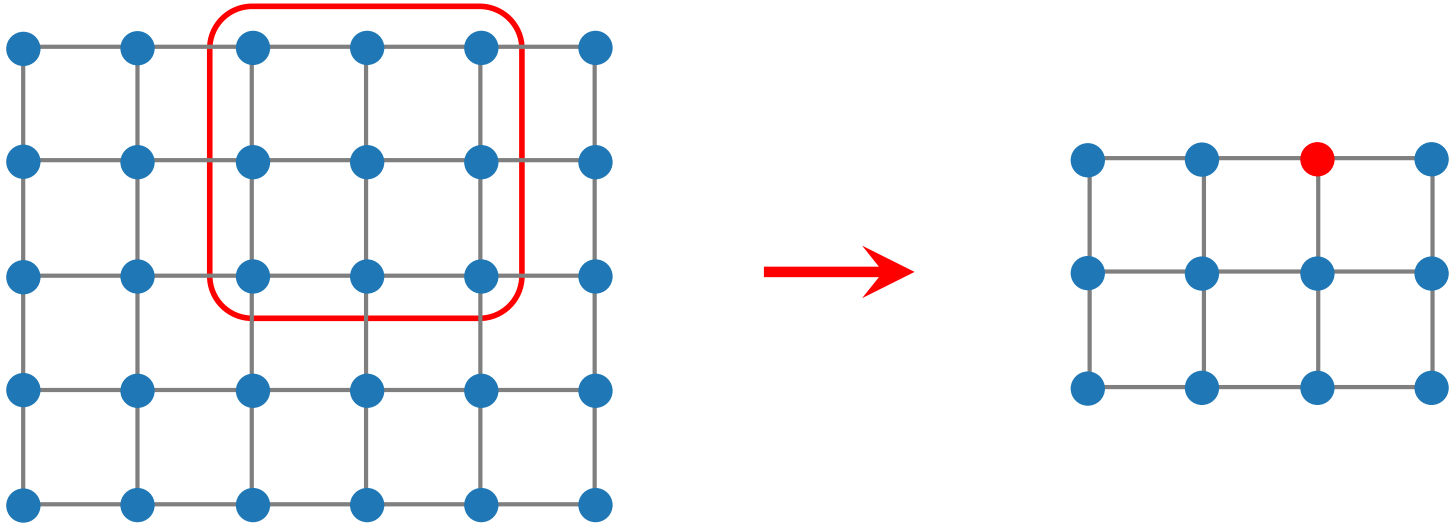
Convolutional Layer

- Another approach could be to use a **convolutional** neural network.
- In this model, a kernel or **filter** is convolved with grid structured data to **extract** features from the input signal.
- Convolution on **structured grid** performed by **sliding** the filter over the grid structures.



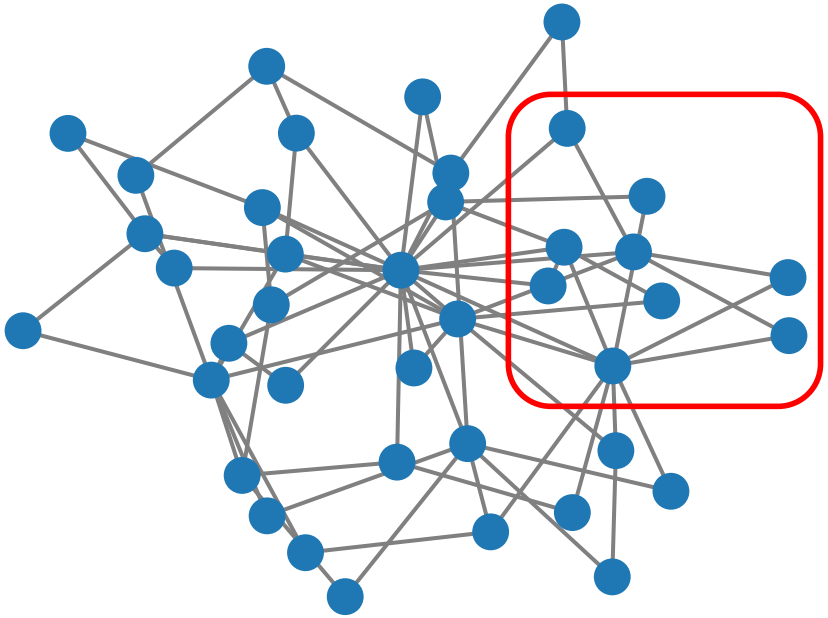
Convolutional Layer

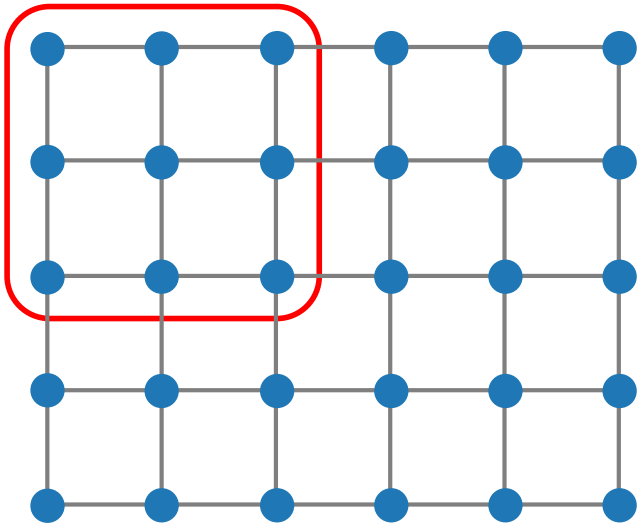
- Another approach could be to use a **convolutional** neural network.
- In this model, a kernel or **filter** is convolved with grid structured data to **extract** features from the input signal.
- Convolution on **structured grid** performed by **sliding** the filter over the grid structures.



Convolutional Layer

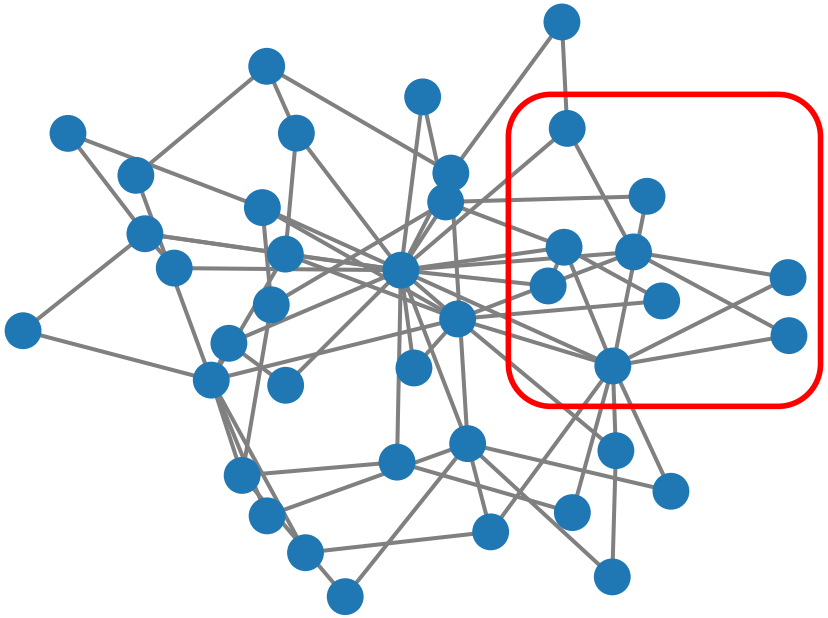
- When translating this to graphs, it is not clear how to directly operate **convolution on graphs**.
- One issue is how to slide the kernel window over the graph.

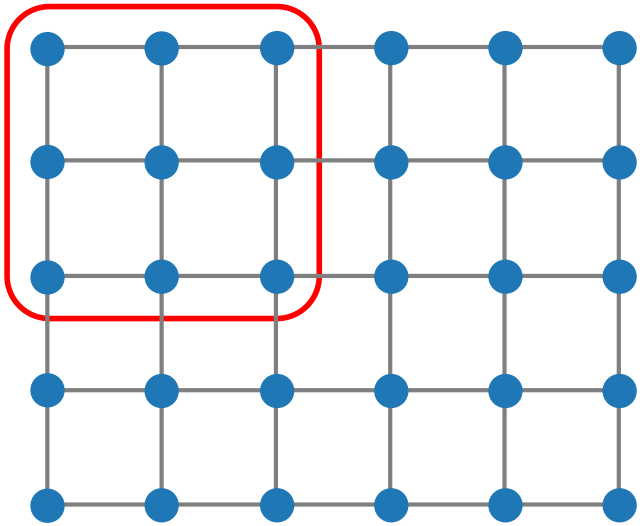




Convolutional Layer

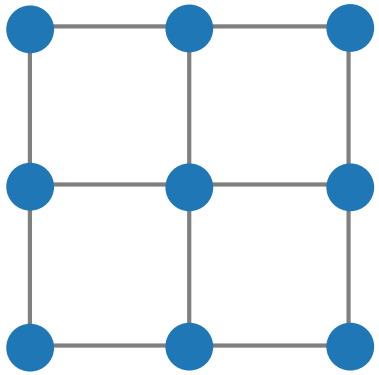
- When translating this to graphs, it is not clear how to directly operate **convolution on graphs**.
- One issue is how to slide the kernel window over the graph.
  - How to select nodes when sliding
  - How many nodes in the window after sliding





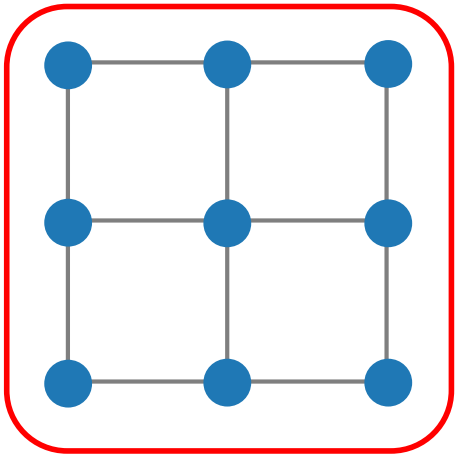
Convolutional Layer

- To address this, we resort to the notion of **gathering local information** in the convolutional layers.



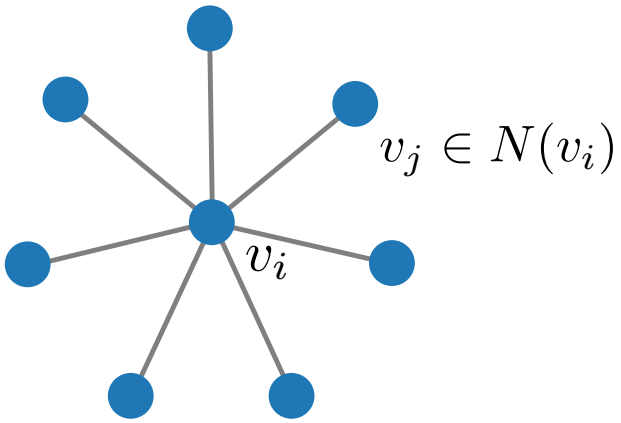
Convolutional Layer

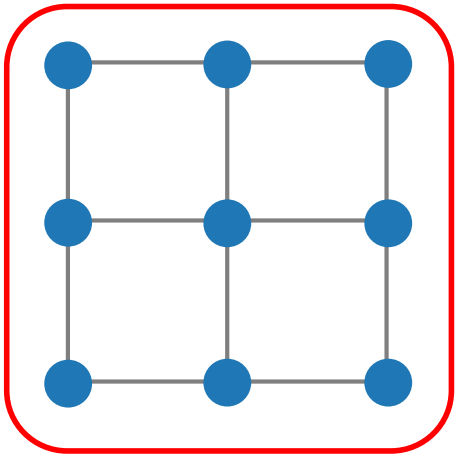
- To address this, we resort to the notion of **gathering local information** in the convolutional layers.
- A convolution filter summarizes the information in a small-sized, spatially-defined **window**.



Convolutional Layer

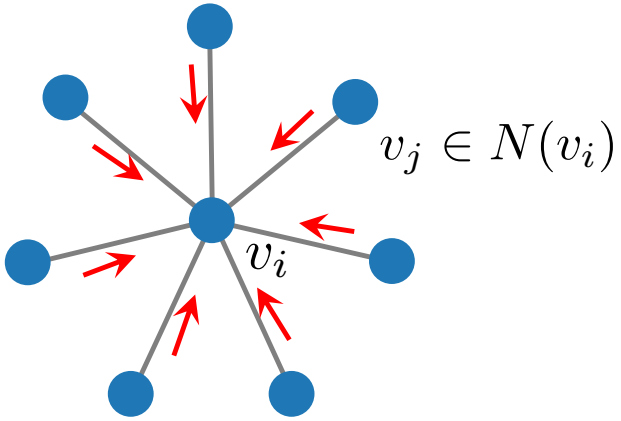
- To address this, we resort to the notion of **gathering local information** in the convolutional layers.
- A convolution filter summarizes the information in a small-sized, spatially-defined **window**.
- This can be extended to graphs using the notation of **local neighborhood** of a node.

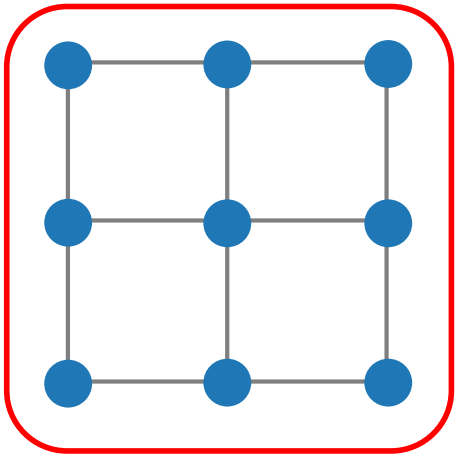




Message Passing

- To address this, we resort to the notion of **gathering local information** in the convolutional layers.
- A convolution filter summarizes the information in a small-sized, spatially-defined **window**.
- This can be extended to graphs using the notation of **local neighborhood** of a node.
- At each iteration, node v_i **collects** information from a neighborhood $N(v_i)$.





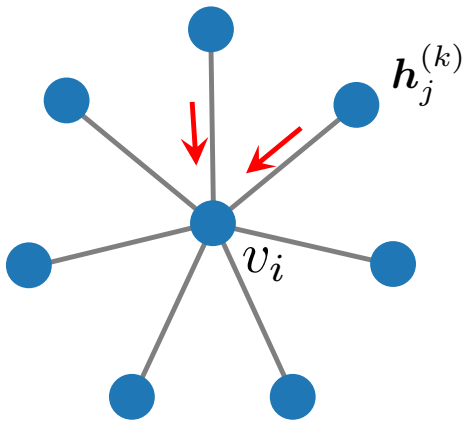
Neural Message Passing

- The basic component of graph neural networks is **neural message passing**.
- In this algorithm, nodes **pass messages** to each other and then they are updated using **neural networks**.
- For a graph $G = (V, E)$, the goal is to construct node embedding $\boldsymbol{z}_i \in \mathbb{R}^{d'}$ for all nodes $v_i \in \mathrm{V}$ through neural message passing.
- This propagation rule uses the graph **structure** $\boldsymbol{A}$ and node **attributes** $\boldsymbol{X} \in \mathbb{R}^{d \times |V|}$ and returns embeddings $\boldsymbol{z}_i \in \mathbb{R}^{d'}$.

Neural Message Passing

- At each iteration k, hidden activation $\boldsymbol{h}_j^{(k)}$ of neighbor nodes $v_j \in N(v_i)$ are collected through an **aggregation** operator.
- An aggregate operator is a **differentiable** function that receives the neighborhood information and summarizes them into a message.
- Mathematically put

$$m_{N_i \to i}^{(k)} = \texttt{aggregate}\left(\left\{\boldsymbol{h}_j^{(k)} \mid v_j \in N(v_i)\right\}\right)$$



Neural Message Passing

- At each iteration k, hidden activation $\boldsymbol{h}_j^{(k)}$ of neighbor nodes $v_j \in N(v_i)$ are collected through an **aggregation** operator.
- An aggregate operator is a **differentiable** function that receives the neighborhood information and summarizes them into a message.
- Mathematically put

$$m_{N_i \to i}^{(k)} = \texttt{aggregate}\left(\left\{\boldsymbol{h}_j^{(k)} \mid v_j \in N(v_i)\right\}\right)$$

- Then, each node is modified using its neighborhood messages through an **update operator**.
- The update operator is a **differentiable** function parametrized by a neural network that **combines** the neighborhood messages with the current state of each node $\boldsymbol{h}_i^{(k)}$.

Neural Message Passing

- A graph neural network's propagation rule is a combination of two differentiable functions:
  - **Aggregate:**
  - Summarizes the hidden activations $h_j^{(k)}$ in the neighborhood $v_j \in N(v_i)$ of node v_i in a message $m_{N_i \to i}^{(k)}$

$$m_{N_i \to i}^{(k)} = \texttt{aggregate}(\{h_j^{(k)} | v_j \in N(v_i)\})$$

  - **Update:**
  - Combines the message $m_{N_i \to i}$ with the hidden activation $h_i^{(k)}$ of node v_i.

$$h_i^{(k+1)} = \texttt{update}(h_i^{(k)}, m_{N_i \to i}^{(k)})$$

Neural Message Passing

- In iteration $k = 0$, the information on the nodes $v_i \in V$ are the **attributes** $\boldsymbol{x}_i \in \mathbb{R}^d$ associated with each node.

$$\boldsymbol{h}_i^{(0)} = \boldsymbol{x}_i$$

- If the input graph does not contain node attributes, one can initialize $h_i^{(0)}$ by measures of **node importance**.
- Alternatively, one can initialize nodes with one-hot vectors of **node indices** $\boldsymbol{e}^{(i)} \in \{0,1\}^{|V|}$.
- The latter approach renders the method **transductive**.

Neural Message Passing

- In iteration $k = 0$, the information on the nodes $v_i \in V$ are the **attributes** $\boldsymbol{x}_i \in \mathbb{R}^d$ associated with each node.

$$\boldsymbol{h}_i^{(0)} = \boldsymbol{x}_i$$

- If the input graph does not contain node attributes, one can initialize $h_i^{(0)}$ by measures of **node importance**.
- Alternatively, one can initialize nodes with one-hot vectors of **node indices** $\boldsymbol{e}^{(i)} \in \{0,1\}^{|V|}$.
- The latter approach renders the method **transductive**.
- In the final message passing iteration K, the update function returns the **node embeddings** $\boldsymbol{z}_i \in \mathbb{R}^{d'}$ generated by the graph neural networks.

$$\boldsymbol{z}_i = \boldsymbol{h}_i^{(K)}$$

Basic GNN

- The basic graph neural network is represented as

$$\boldsymbol{h}_i^{(k)} = \sigma\left(\mathbf{W}_v^{(k)}\boldsymbol{h}_i^{(k-1)} + \mathbf{W}_N^{(k)}\sum_{v_j \in N(v_i)}\boldsymbol{h}_j^{(k-1)}\right)$$

  where $\boldsymbol{W}_v^{(k)}, \boldsymbol{W}_N^{(k)} \in \mathbb{R}^{d^{(k-1)} \times d^{(k)}}$ are model parameters in iteration k and σ is the non-linear activation function.

- This index notation represents a basic GNN in the **node level**.

Basic GNN

- The basic graph neural network is represented as

$$\boldsymbol{h}_i^{(k)} = \sigma\left(\mathbf{W}_v^{(k)}\boldsymbol{h}_i^{(k-1)} + \mathbf{W}_N^{(k)}\sum_{v_j \in N(v_i)} \boldsymbol{h}_j^{(k-1)}\right)$$

  where $\boldsymbol{W}_v^{(k)}, \boldsymbol{W}_N^{(k)} \in \mathbb{R}^{d^{(k-1)} \times d^{(k)}}$ are model parameters in iteration k and σ is the non-linear activation function.

- This index notation represents a basic GNN in the **node level**.
- We can represent a **graph-level** relation using matrix notation

$$\boldsymbol{H}^{(k)} = \sigma\left(\boldsymbol{H}^{(k-1)}\mathbf{W}_v^{(k)} + \mathbf{A}\boldsymbol{H}^{(k-1)}\mathbf{W}_N^{(k)}\right)$$

  where $\boldsymbol{H}^{(k)} \in \mathbb{R}^{|V| \times d}$ is the hidden activation.

Basic GNN

- We can rewrite this using the aggregate and update framework as

$$\texttt{aggregate}: \quad m_{N_i \to i}^{(k)} = \sum_{v_j \in N(v_i)} \boldsymbol{h}_j^{(k)}$$

and

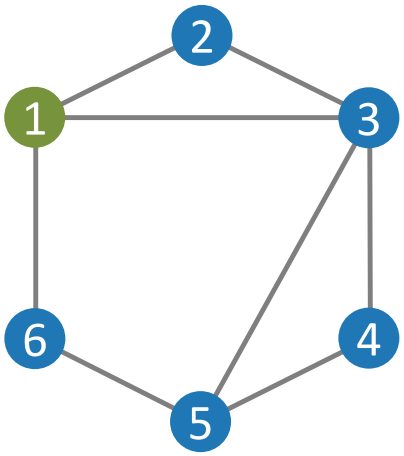
$$\texttt{update} \ : \quad \boldsymbol{h}_i^{(k+1)} = \sigma\left(\mathbf{W}_v^{(k+1)} \boldsymbol{h}_i^{(k)} + \mathbf{W}_N^{(k+1)} m_{N_i \to i}^{(k)}\right)$$

where $\boldsymbol{W}_i^{(k)}$ and $\boldsymbol{W}_N^{(k)}$ are model parameters in iteration k.

- This representation of basic GNN as linear combination of inputs followed by a non-linear layer is analogues to feed-forward networks.
- Therefore, the update operator is a linear combination of embedding $\boldsymbol{h}_i$ and messages $m_{N_i \to i}$ followed by a non-linearity.

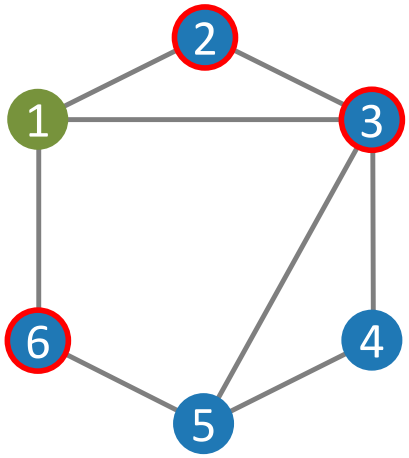
Subtree Pattern

- One can demonstrate the message passing in terms of the unfolded subtree pattern of each node v_i.



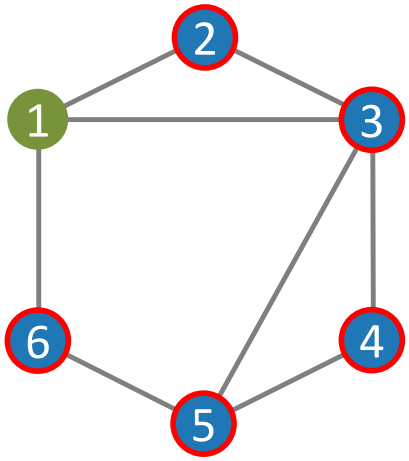
Subtree Pattern

- One can demonstrate the message passing in terms of the **unfolded subtree pattern** rooted at each node v_i.
- In one iteration of message passing, node v_1 receives information from its 1 −hop neighborhood.



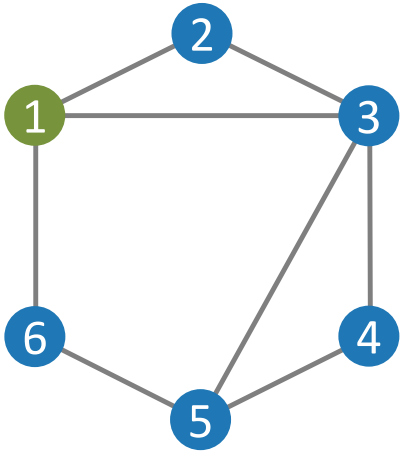
Subtree Pattern

- One can demonstrate the message passing in terms of the **unfolded subtree pattern** rooted at each node v_i.
- In one iteration of message passing, node v_1 receives information from its 1 −hop neighborhood.
- In iteration $k = 2$, v_1 receives information from its 2 −hop neighbors.



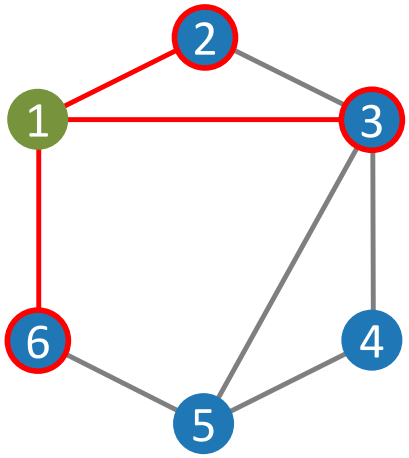
Subtree Pattern

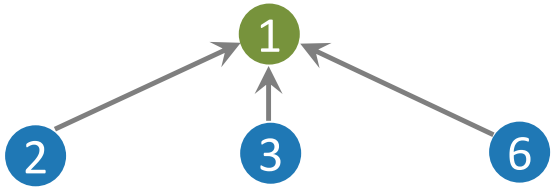
- One can demonstrate the message passing in terms of the **unfolded subtree pattern** rooted at each node v_i.
- In one iteration of message passing, node v_1 receives information from its 1 −hop neighborhood.
- In iteration $k = 2$, v_1 receives information from its 2 −hop neighbors.
- This can be shown using an unfolded height 2 subtree pattern rooted at v_1.



Subtree Pattern

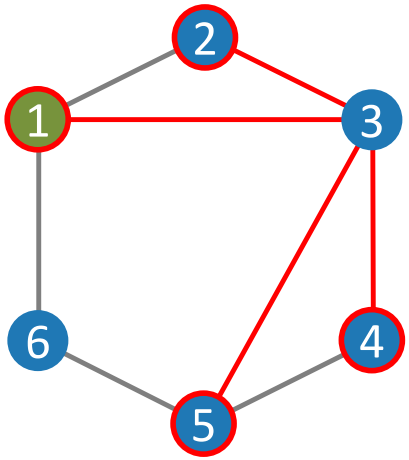
- One can demonstrate the message passing in terms of the **unfolded subtree pattern** rooted at each node v_i.
- In one iteration of message passing, node v_1 receives information from its 1 −hop neighborhood.
- In iteration $k = 2$, v_1 receives information from its 2 −hop neighbors.
- This can be shown using an unfolded height 2 subtree pattern rooted at v_1.

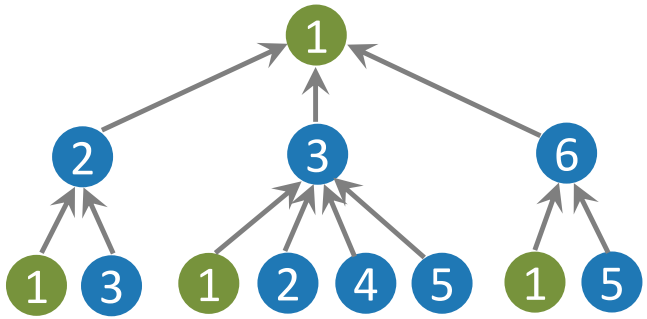




Subtree Pattern

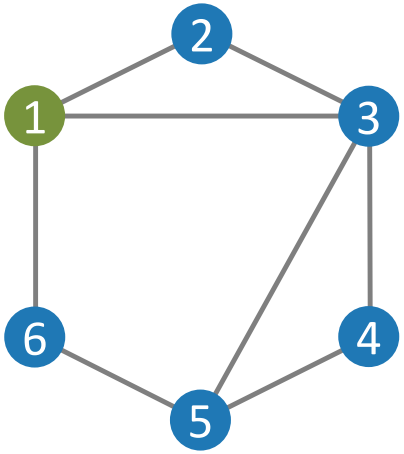
- One can demonstrate the message passing in terms of the **unfolded subtree pattern** rooted at each node v_i.
- In one iteration of message passing, node v_1 receives information from its 1 −hop neighborhood.
- In iteration $k = 2$, v_1 receives information from its 2 −hop neighbors.
- This can be shown using an unfolded height 2 subtree pattern rooted at v_1.

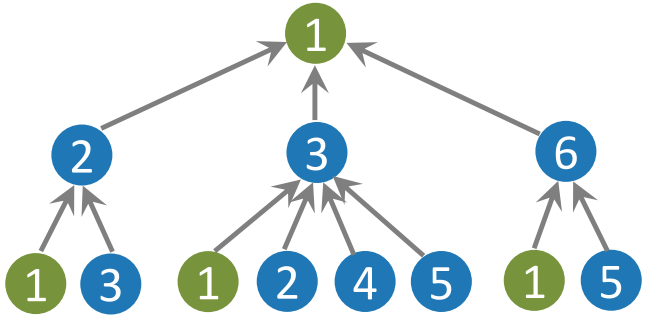




Neural Message Passing

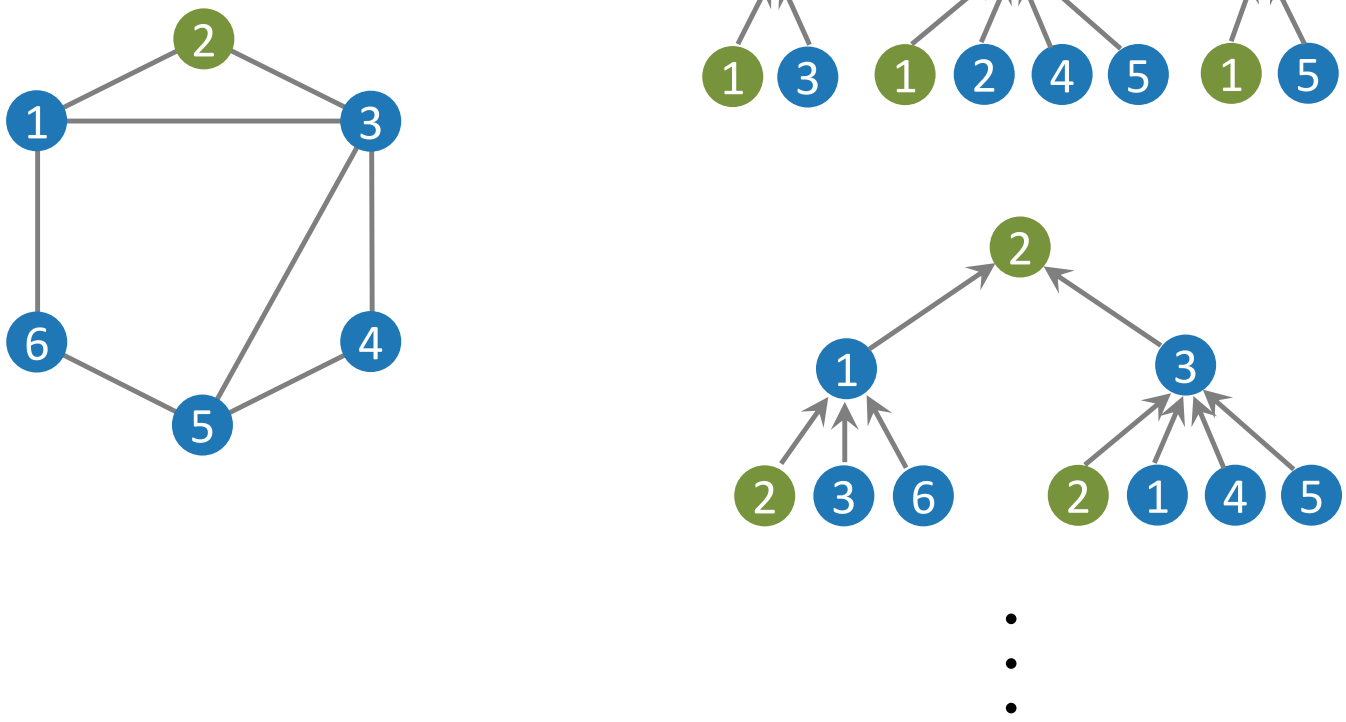
- A neural message passing propagation rule uses an unfolded, height k subtree pattern rooted at node v_i to construct node embeddings.





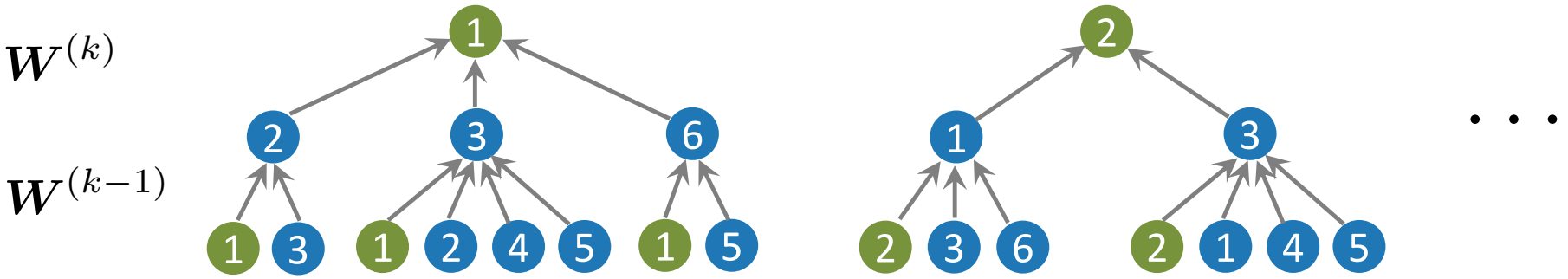
Neural Message Passing

- A neural message passing propagation rule uses an unfolded, height k subtree pattern rooted at node v_i to construct node embeddings.



Neural Message Passing

- A neural message passing propagation rule uses an unfolded, height k subtree pattern rooted at node v_i to construct node embeddings.
- All such iterations rooted at target nodes $v_i \in V$ are parameterized by shared parameters $\boldsymbol{W}^{(k)}$.



Neural Message Passing

- There are various ways to this layer-wise propagation rule
  - Spectral graph convolutions.
  - Differentiable loopy belief propagation.
  - Classical graph isomorphism tests.
- Here, we use the **Weisfeiler-leman** algorithm to motivate the neural message passing propagation rule in graph-based neural networks.

Weisfeilier-Leman Algorithm

- Given a label function h, WL algorithm follows the steps below:
  - Set initial label $h_i^{(0)}$ for each node $v_i \in V$.
  - At each iteration k, for $v_i \in V$,

Weisfeilier-Leman Algorithm

- Given a label function h, WL algorithm follows the steps below:
  - Set initial label $h_i^{(0)}$ for each node $v_i \in V$.
  - At each iteration k, for $v_i \in V$,
    - Aggregate and sort labels of neighboring nodes.

$$\texttt{sort}\left(\left\{\left\{h_j^{(k-1)} \mid v_j \in N\left(v_i\right)\right\}\right\}\right)$$

Weisfeilier-Leman Algorithm

- Given a label function h, WL algorithm follows the steps below:
  - Set initial label $h_i^{(0)}$ for each node $v_i \in V$.
  - At each iteration k, for $v_i \in V$,
    - Aggregate and sort labels of neighboring nodes.
    - Construct a tuple of the current label $h_i^{(k-1)}$ and the multiset of the sorted neighborhood labels.

$$\left(h_i^{(k-1)}, \texttt{sort}\left(\left\{\left\{h_j^{(k-1)} \mid v_j \in N\left(v_i\right)\right\}\right\}\right)\right)$$

Weisfeilier-Leman Algorithm

- Given a label function h, WL algorithm follows the steps below:
  - Set initial label $h_i^{(0)}$ for each node $v_i \in V$.
  - At each iteration k, for $v_i \in V$,
    - Aggregate and sort labels of neighboring nodes.
    - Construct a tuple of the current label $h_i^{(k-1)}$ and the multiset of the sorted neighborhood labels.
    - Hash each unique tuple as $h_i^{(k)}$.

$$\texttt{hash}\left(h_i^{(k-1)}, \texttt{sort}\left(\left\{\left\{h_j^{(k-1)} \mid v_j \in N(v_i)\right\}\right\}\right)\right)$$

Weisfeilier-Leman Algorithm

- Given a label function h, WL algorithm follows the steps below:
  - Set initial label $h_i^{(0)}$ for each node $v_i \in V$.
  - At each iteration k, for $v_i \in V$,
    - Aggregate and sort labels of neighboring nodes.
    - Construct a tuple of the current label $h_i^{(k-1)}$ and the multiset of the sorted neighborhood labels.
    - Hash each unique tuple as $h_i^{(k)}$.
    - Set $h_i^{(k)}$ as the new label for v_i.

$$h_i^{(k)} \leftarrow \texttt{hash}\left(h_i^{(k-1)}, \texttt{sort}\left(\left\{\left\{h_j^{(k-1)} \mid v_j \in N(v_i)\right\}\right\}\right)\right)$$

Neural Message Passing

- The graph-based propagation rule replaces the sort and hash functions with **differentiable functions**.

- In other words, the presented propagation rule is a **differentiable** and **parametrized** version of the WL algorithm.

Neural Message Passing

- The graph-based propagation rule replaces the sort and hash functions with **differentiable functions**.

$$\sum_{v_j \in N(v_i)} \boldsymbol{h}_j^{(k-1)}$$

$$\texttt{sort}\left(\left\{\left\{\boldsymbol{h}_j^{(k-1)} \mid v_j \in N\left(v_i\right)\right\}\right\}\right)$$

- In other words, the presented propagation rule is a **differentiable** and **parametrized** version of the WL algorithm.

Neural Message Passing

- The graph-based propagation rule replaces the sort and hash functions with **differentiable functions**.

$$\boldsymbol{h}_i^{(k-1)} \qquad \sum_{v_j \in N(v_i)} \boldsymbol{h}_j^{(k-1)}$$

$$\left(\boldsymbol{h}_i^{(k-1)}, \texttt{sort}\left(\left\{\left\{\boldsymbol{h}_j^{(k-1)} \mid v_j \in N\left(v_i\right)\right\}\right\}\right)\right)$$

- In other words, the presented propagation rule is a **differentiable** and **parametrized** version of the WL algorithm.

Neural Message Passing

- The graph-based propagation rule replaces the sort and hash functions with **differentiable functions**.

$$\boldsymbol{h}_i^{(k)} \leftarrow \sigma\left(\mathbf{W}_v^{(k)}\boldsymbol{h}_i^{(k-1)} + \mathbf{W}_N^{(k)} \sum_{v_j \in N(v_i)} \boldsymbol{h}_j^{(k-1)}\right)$$

$$\boldsymbol{h}_i^{(k)} \leftarrow \texttt{hash}\left(\boldsymbol{h}_i^{(k-1)}, \texttt{sort}\left(\left\{\left\{\boldsymbol{h}_j^{(k-1)} \mid v_j \in N\left(v_i\right)\right\}\right\}\right)\right)$$

- In other words, the presented propagation rule is a **differentiable** and **parametrized** version of the WL algorithm.

Message Passing with Self-loop

- In some cases, the update equation can be combined with the aggregate by introducing **self-loops** to the graph.
- Let G be a graph with adjacency matrix $\boldsymbol{A}$.
- We can add self-loop to all nodes in the graph and redefine the adjacency matrix as

$$\bar{\mathbf{A}} = I + \mathbf{A}$$

- Then, the aggregate function written in the matrix form

$$\boldsymbol{H}^{(k)} = \sigma\left(\bar{\mathbf{A}}\boldsymbol{H}^{(k-1)}\boldsymbol{W}^{(k)}\right)$$

  would also combine the hidden state h_i with neighborhood message.
- This removes the need to explicitly define an update function.

Summary

- Node Embedding shortcomings
- Permutation
- Permutation invariance
- Permutation equivariance
- Neural message passing
- Aggregate-update framework
- Basic GNN
- Unfolded subtree pattern
- Analogy based on graph isomorphism tests
- Message Passing with Self-loop