Graph Neural Networks

Prof. Mirko Polato - mirko.polato@unito.it

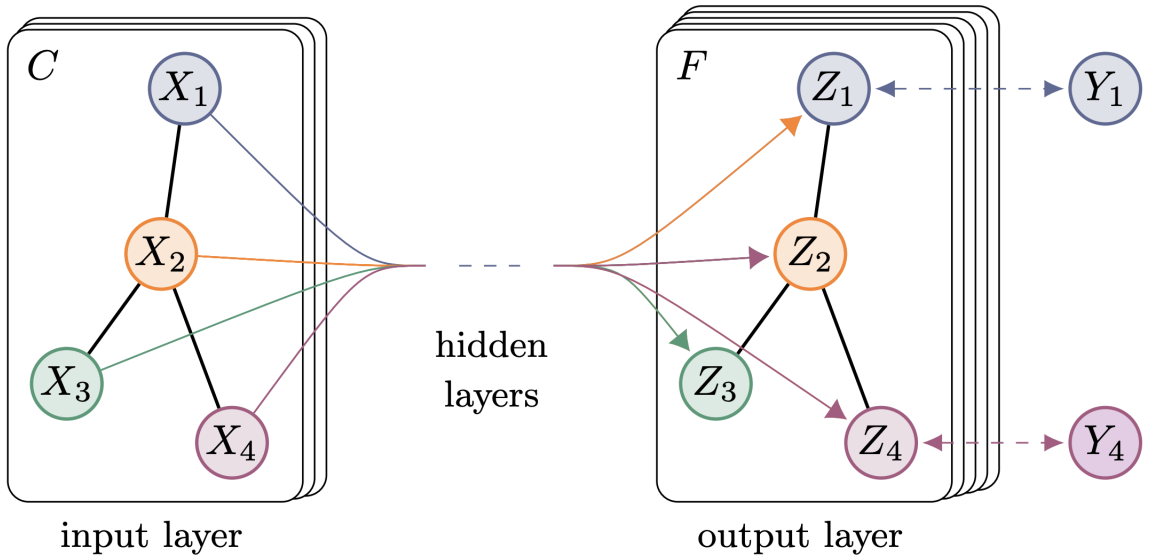


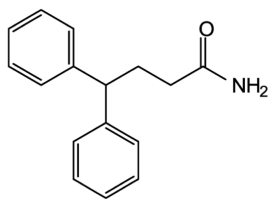
Image from the Kipf, T. N., & Welling, M. (2016). Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907.
These slides are based on the material of the course "Machine Learning with Graphs" by Prof. Jure Leskovec, Stanford University.

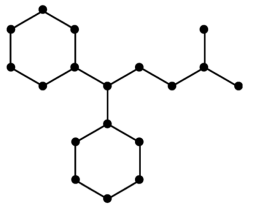
Why do we care about graphs?

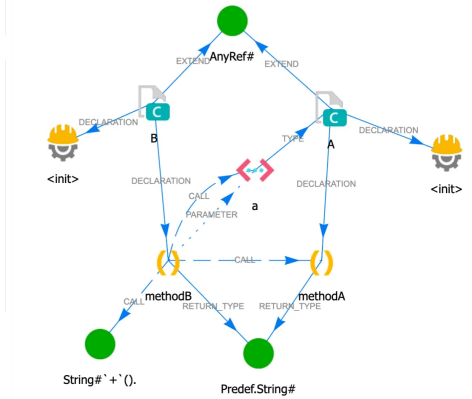
Graphs are a general language for describing and analyzing entities with relations/interactions

Graphs are everywhere

- **Social networks**
- **Biological networks**
- **Transportation networks**
- **Citation networks**
- **Chemical compounds**
- …





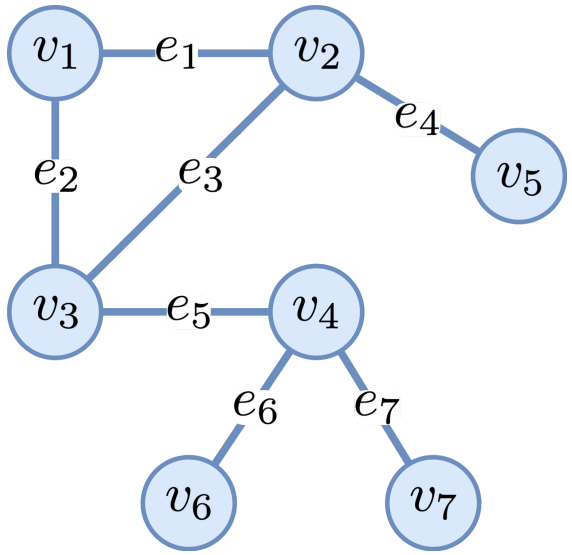


- Social network image from Medium
- Compund image from MDPI
- Code network image from Medium
- Citation network image from ResearchGate

Graph: Definition

A graph $\mathcal{G}$ is defined as a tuple of a set of **nodes/vertices**, and a set of **edges/links**. Each edge is a pair of two vertices, and represents a connection between them.

$$\mathcal{G} = (\mathcal{V}, \mathcal{E})$$

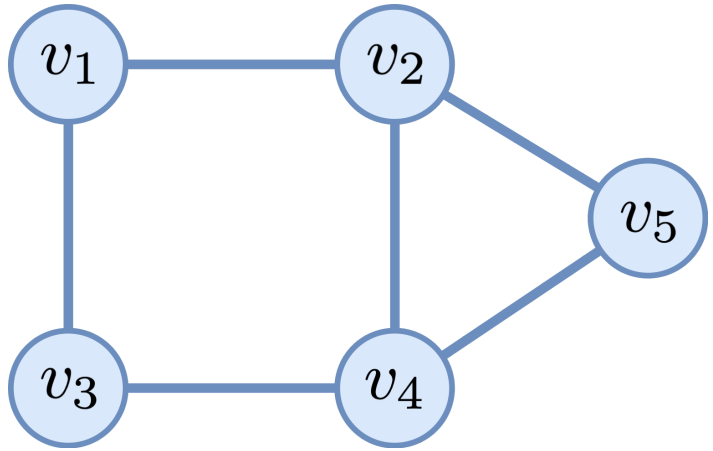


$$\mathcal{V} \equiv \{v_1, \ldots, v_7\}$$
$$\mathcal{E} \equiv \{e_1, \ldots, e_7\}$$

Graph adjacency matrix

Assume we have an (undirected) graph $\mathcal{G}$ with n nodes. The (binary) adjacency matrix $\mathbf{A}^{\mathcal{G}}$ is a square matrix of size $n \times n$, where each entry $\mathbf{A}^{\mathcal{G}}_{ij}$ is 1 if there is an edge between node i and node j, and 0 otherwise.

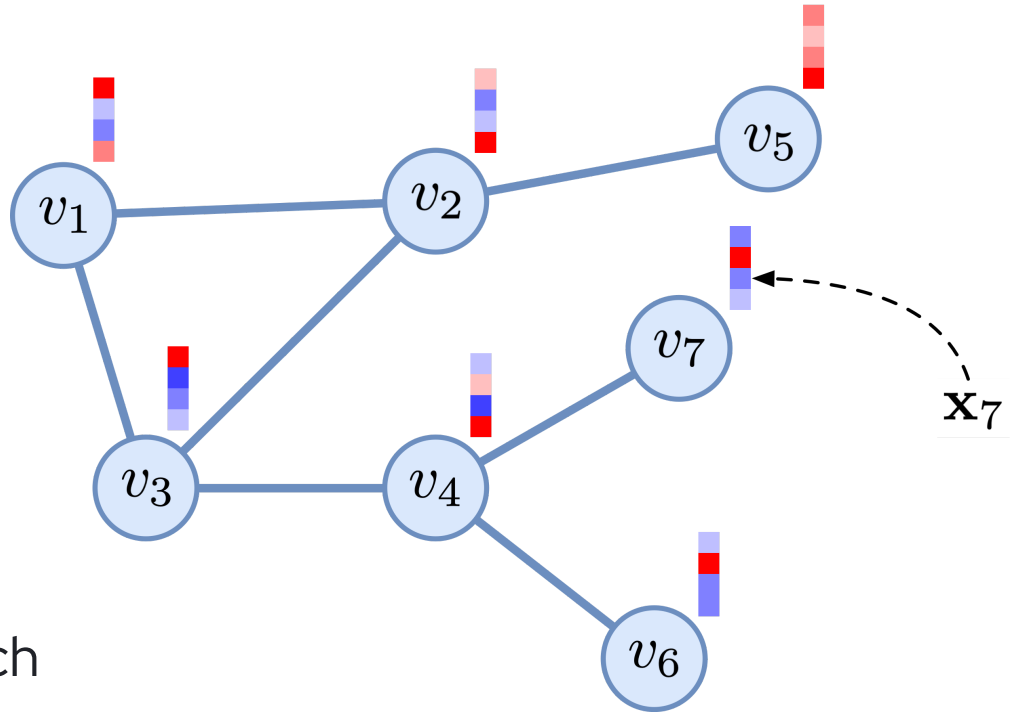
$$\mathbf{A}^{\mathcal{G}} = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$



Setup

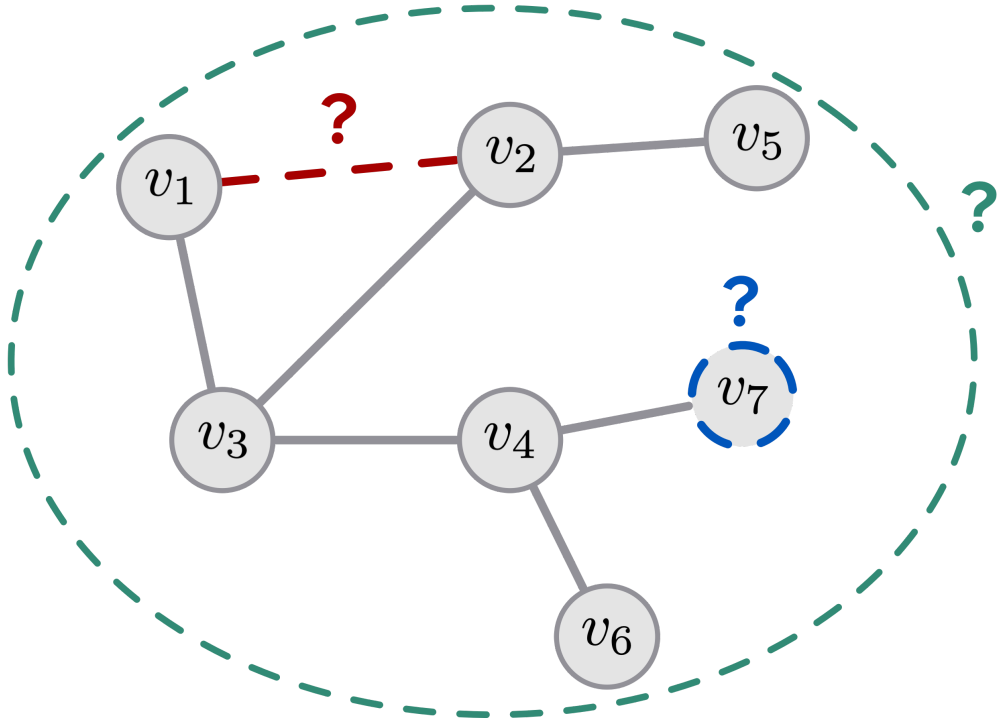
Graph: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$

- $\mathcal{V}$: set of nodes/vertices
- $\mathcal{E}$: set of edges
- $\mathbf{A}^{\mathcal{G}}$ or simply $\mathbf{A}$: adjacency matrix
- $\mathcal{N}(v)$: set of neighbors of node $v \in \mathcal{V}$
- We assume that each node v has an associated feature vector $\mathbf{x}_v \in \mathbb{R}^d$
  - if not, we can set $\mathbf{x}_v = \mathbf{1}$
  - or we can assign an indicator vector to each node (one-hot encoding)
- $\mathcal{G} = (\mathbf{A}, \mathbf{X})$



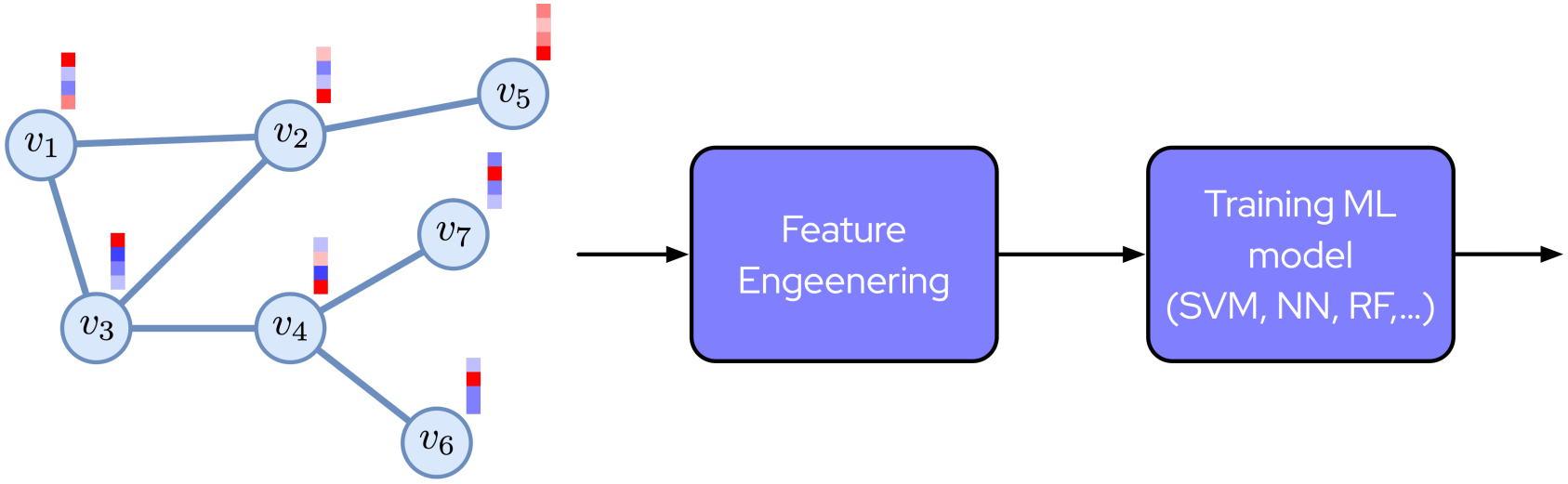
Which tasks can involve graphs?

- **Node-level** prediction.
  (E.g., is a drug toxic? [drug-drug interaction])
- **Link-level** prediction
  (E.g., recommendation)
- **Graph-level** prediction
  (E.g., time of arrival)



Graphs and Machine Learning

- **Traditional method**: Extract features from the graph (e.g., node degree, centrality, subgraph patterns, etc.) and use them as input to a machine learning model



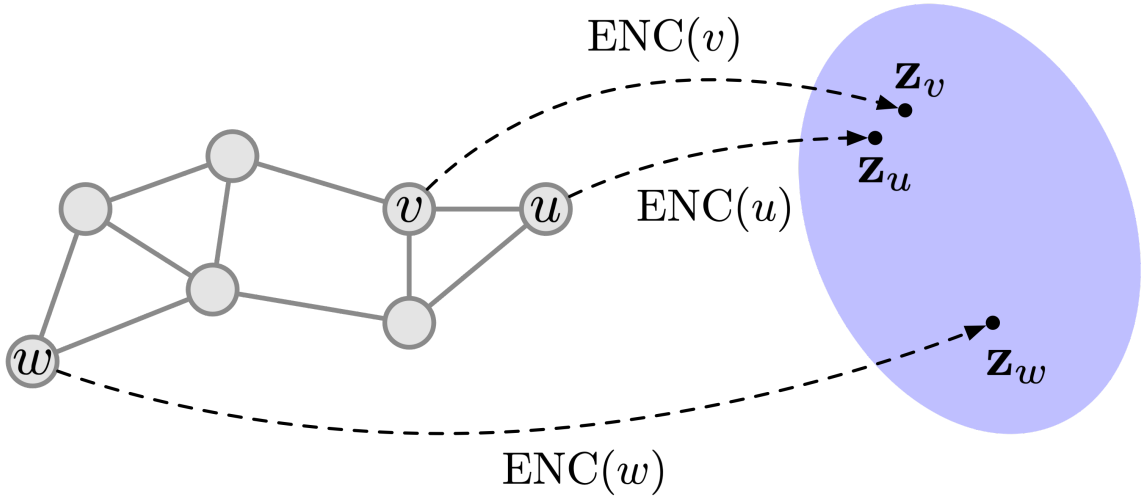
Graph Representation Learning

(Deep) Neural Networks have changed the way we represent and learn from data. Feature engineering is replaced by learning representations from data in the so-called **representation learning** paradigm:

1. Learn a function that maps a graph to a vector representation;
2. Use the learned representation for downstream tasks (e.g., node classification, link prediction, etc.)

Early tries: Node embeddings

Goal is to encode nodes so that **similarity in the embedding space** (e.g., dot product) approximates **similarity in the graph**, for some definition of similarity.



Original graph

emedding space

Note: a simple way to get graph embeddings is to aggregate (e.g., averege) the node embeddings.

Encoder-Decoder framework

Key components:

- **ENC**: a function that maps nodes to vector representations

$$\mathbf{z}_v = \text{ENC}(v)$$

- **Similarity**: function that measures the similarity between two nodes in the network space

$$\text{sim}(v, u)$$

- **DEC**: function that compute the similarity between two nodes in the embedding space

$$\mathbf{y}_{vu} = \text{DEC}(\mathbf{z}_v, \mathbf{z}_u)$$

Encoder-Decoder framework

Usually, the decoder is a function of the dot product between the embeddings of two nodes:

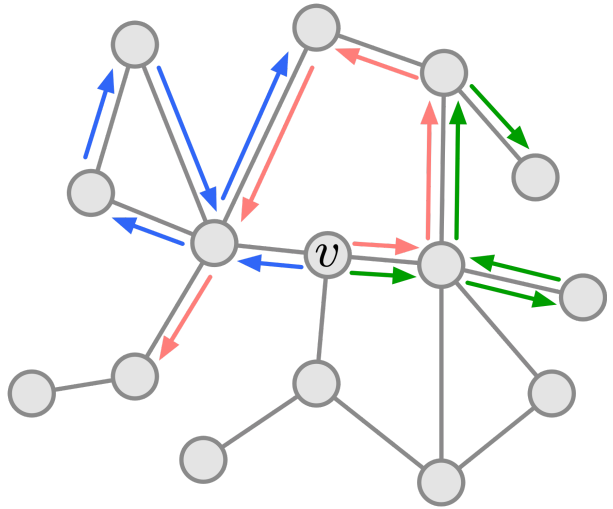
$$\mathbf{y}_{vu} = \text{DEC}(\mathbf{z}_v, \mathbf{z}_u) = f(\mathbf{z}_v^\top \mathbf{z}_u)$$

Once fixed DEC, the goal is to learn the function ENC such that the similarity in the embedding space approximates the **similarity in the graph space**.

Random-Walk as a similarity measure

A random walk on a graph is a sequence of steps where at each step, a node is chosen randomly from the neighboring nodes of the current node.

Estimate probability of visiting node u on a random walk starting from node v using some random walk strategy.



RW is a flexible stochastic definition of node similarity that incorporates both local and higher-order neighborhood information

Optimization

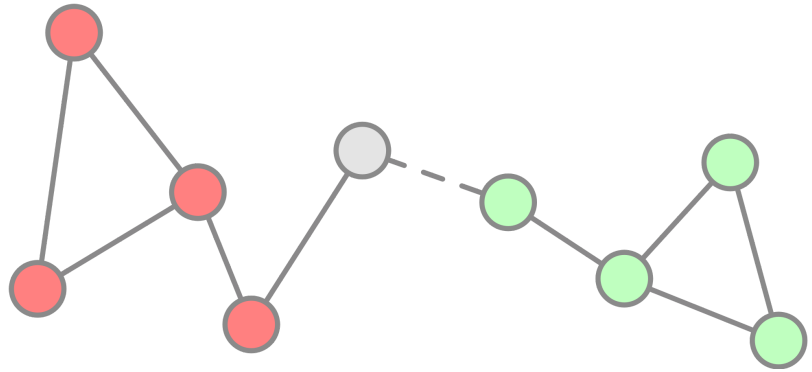
Learning strategy: we want to learn the (parametrized) function $\mathbf{ENC}_\Theta$ such that $\mathbf{z}_v^\top \mathbf{z}_u = \mathbf{ENC}_\Theta(v)^\top \mathbf{ENC}_\Theta(u) \approx$ probability that u and v co-occur on a random walk over the graph.

$$\arg\min_\Theta \sum_{v \in \mathcal{V}} \sum_{u \in \mathcal{N}_R(v)} -\log\left(\frac{\exp\left(\mathbf{z}_v^\mathrm{T} \mathbf{z}_u\right)}{\sum_{w \in \mathcal{V}} \exp\left(\mathbf{z}_v^\mathrm{T} \mathbf{z}_w\right)}\right)$$

which can be optimized using gradient-based methods.

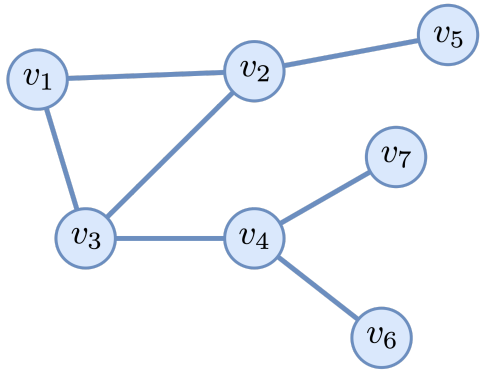
Limitations of these kind of approaches

- **Transductive** (not inductive) method, i.e., cannot obtain embeddings for nodes not in the training set $\Rightarrow$ Cannot be applied to new graphs
- Cannot capture structural similarity



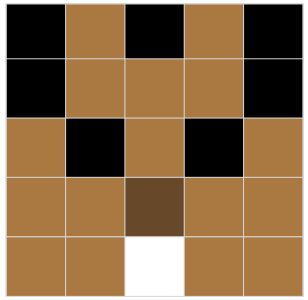
- Cannot utilize node, edge and graph features

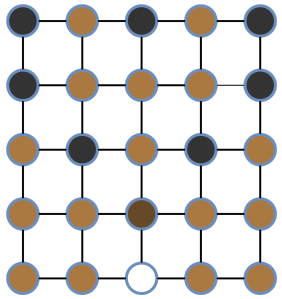
Graph Neural Network - Is it really that hard?



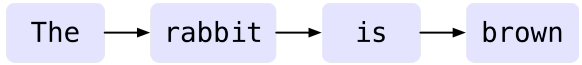
VS

Image

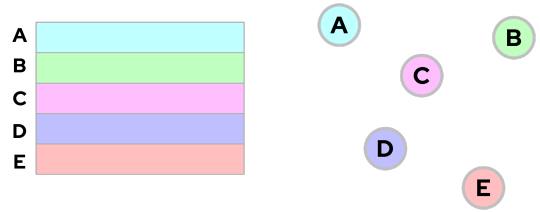




Text

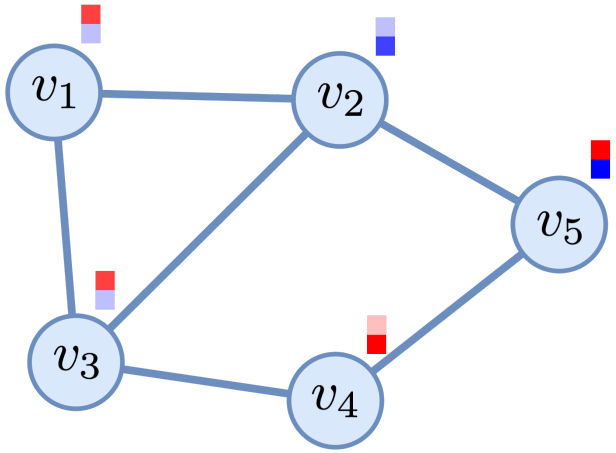


Tabular data



- Arbitrary size and complex topological structure (i.e., no spatial locality like grids)
- No fixed node ordering or reference point
- Often dynamic and have multimodal features

Graph Neural Network - A Naïve approach

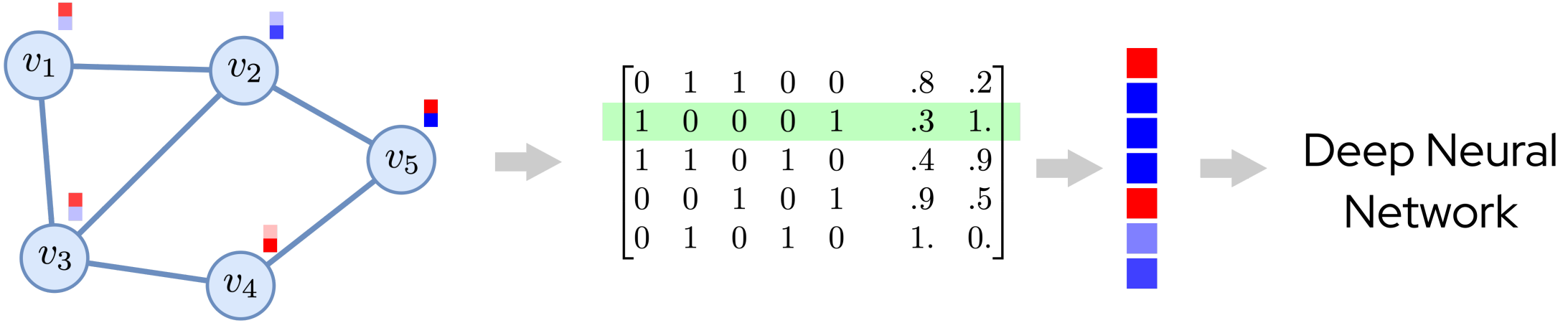


$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & .8 & .2 \\ 1 & 0 & 0 & 0 & 1 & .3 & 1. \\ 1 & 1 & 0 & 1 & 0 & .4 & .9 \\ 0 & 0 & 1 & 0 & 1 & .9 & .5 \\ 0 & 1 & 0 & 1 & 0 & 1. & 0. \end{bmatrix}$$

Deep Neural Network

What is the problem with this approach?

Graph Neural Network - A Naïve approach



What is the problem with this approach?

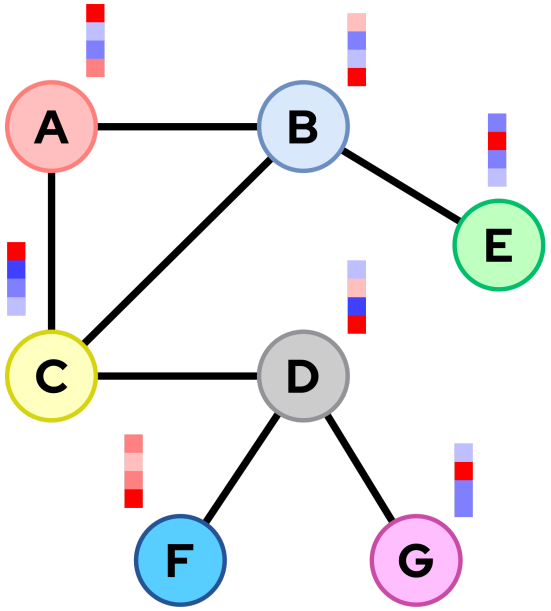
Two main issues:

- **PROBLEM 1**: Sensitive to node ordering
- **PROBLEM 2**: Not applicable to graphs of different sizes

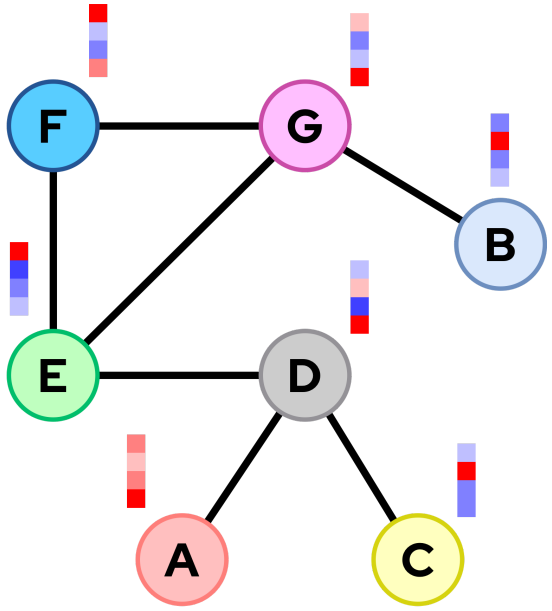
Permutation invariance

Graph does not have a canonical order of the nodes!

Order plan **1**



Order plan **2**

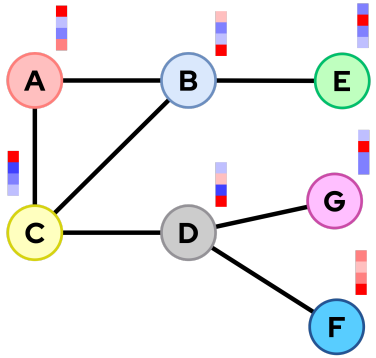


Graph and node **representations should be the same** for Order plan 1 and Order plan 2

Permutation invariance

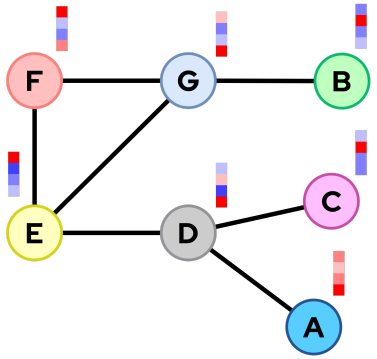
Order plan **1**

$\mathbf{A}_1, \mathbf{X}_1$



Order plan **2**

$\mathbf{A}_2, \mathbf{X}_2$

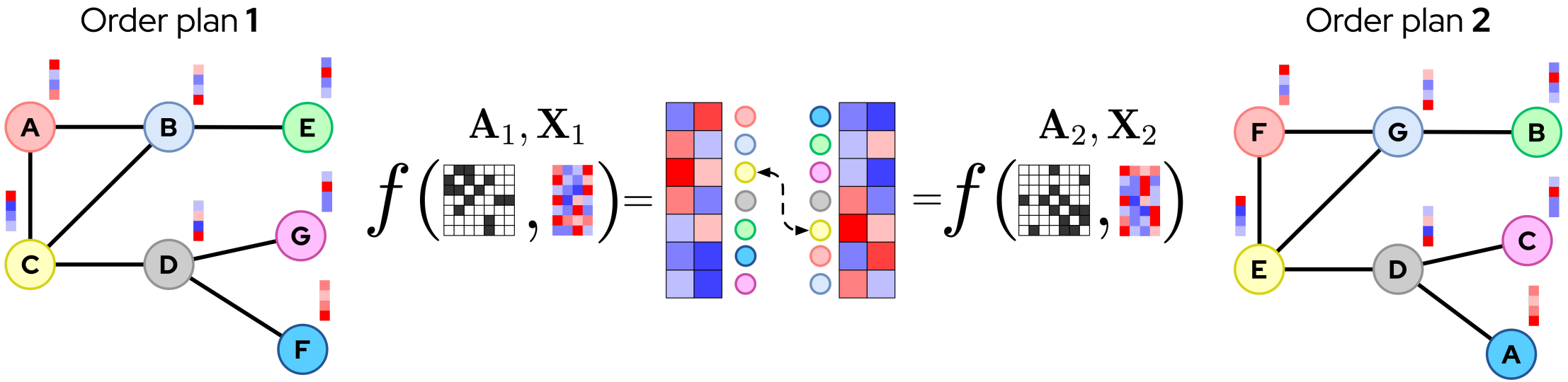


Consider we learn a function f that maps a graph $\mathcal{G} = (\mathbf{A}, \mathbf{X})$ to a vector $\mathbf{z} \in \mathbb{R}^d$, then $f(\mathbf{A}_1, \mathbf{X}_1) = f(\mathbf{A}_2, \mathbf{X}_2)$.

Permutation-invariant

For any graph function $f : \mathbb{R}^{n \times n} \times \mathbb{R}^{n \times m} \to \mathbb{R}^d$, and for any permutation matrix $\mathbf{P}$, f is permutation-invariant if $f(\mathbf{A}, \mathbf{X}) = f(\mathbf{P}\mathbf{A}\mathbf{P}^\top, \mathbf{P}\mathbf{X})$.

Permutation equivariance



Permutation-invariant

For any node function $f : \mathbb{R}^{n\times n} \times \mathbb{R}^{n\times m} \to \mathbb{R}^{n\times d}$, and for any permutation matrix $\mathbf{P}$, f is permutation-equivariant if $\mathbf{P}f(\mathbf{A}, \mathbf{X}) = f(\mathbf{PAP}^\top, \mathbf{PX})$.

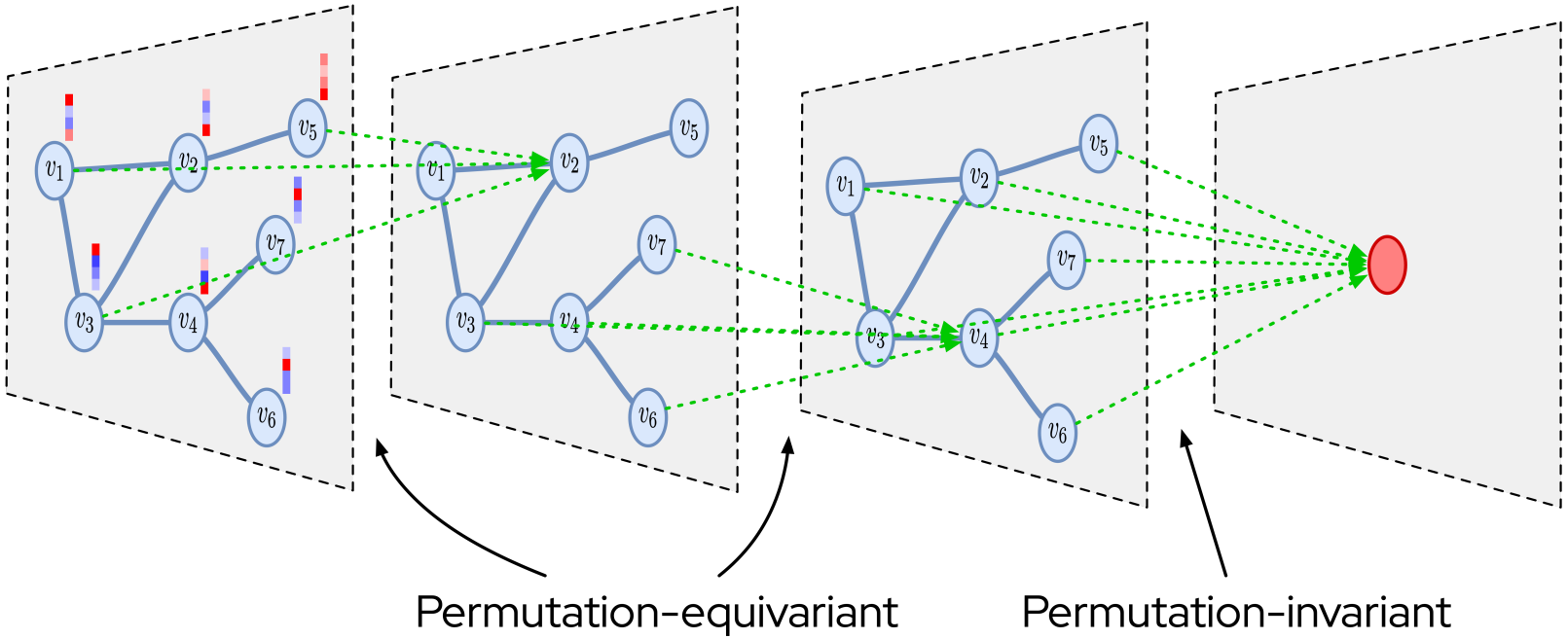
Permutation invariant and equivariant examples

- $f(\mathbf{A}, \mathbf{X}) = \mathbf{1}^\top \mathbf{X} = \sum_i \mathbf{x}_i$ is permutation-invariant
  Proof: $f(\mathbf{P}\mathbf{A}\mathbf{P}^\top, \mathbf{X}) = \mathbf{1}^\top \mathbf{P}\mathbf{X} = \sum_i \mathbf{x}_i = f(\mathbf{A}, \mathbf{X})$
- $f(\mathbf{A}, \mathbf{X}) = \mathbf{X}$ is permutation-equivariant
  Proof: $f(\mathbf{P}\mathbf{A}\mathbf{P}^\top, \mathbf{P}\mathbf{X}) = \mathbf{P}\mathbf{X} = \mathbf{P}f(\mathbf{A}, \mathbf{X})$
- $f(\mathbf{A}, \mathbf{X}) = \mathbf{A}\mathbf{X}$ is permutation-equivariant
  Proof: $f(\mathbf{P}\mathbf{A}\mathbf{P}^\top, \mathbf{P}\mathbf{X}) = \mathbf{P}\mathbf{A}\mathbf{P}^\top\mathbf{P}\mathbf{X} = \mathbf{P}\mathbf{A}\mathbf{X} = \mathbf{P}f(\mathbf{A}, \mathbf{X})$
- **A MLP is neither permutation-invariant nor permutation-equivariant!**

GNN - The vision

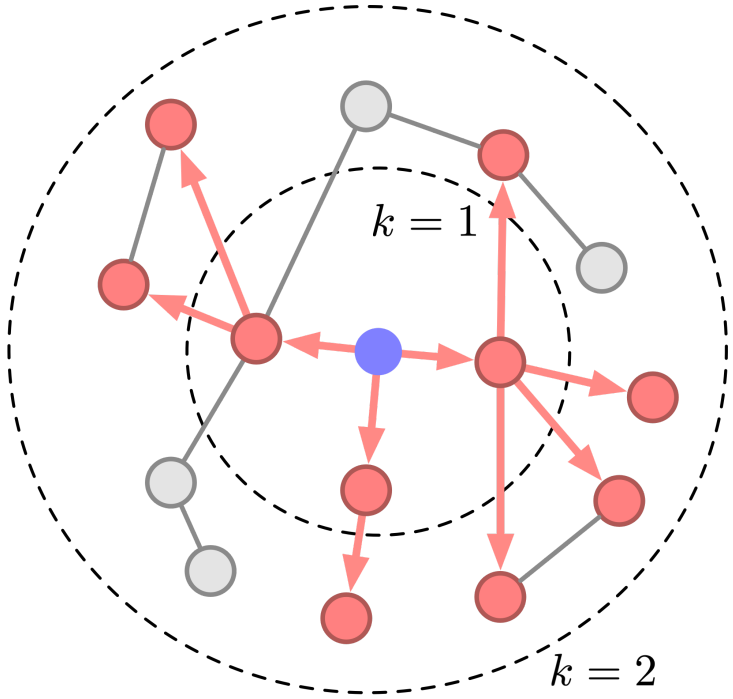
A GNN is an optimizable transformation on all attributes of the graph (nodes, edges, global-context) that preserves graph symmetries (permutation invariances).

Goal: Design graph neural networks that are permutation invariant / equivariant by passing and aggregating information from neighbors.

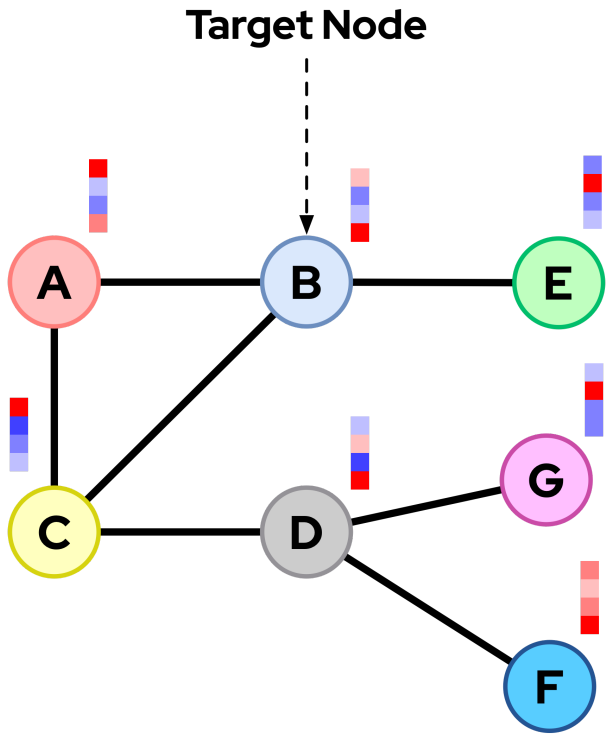


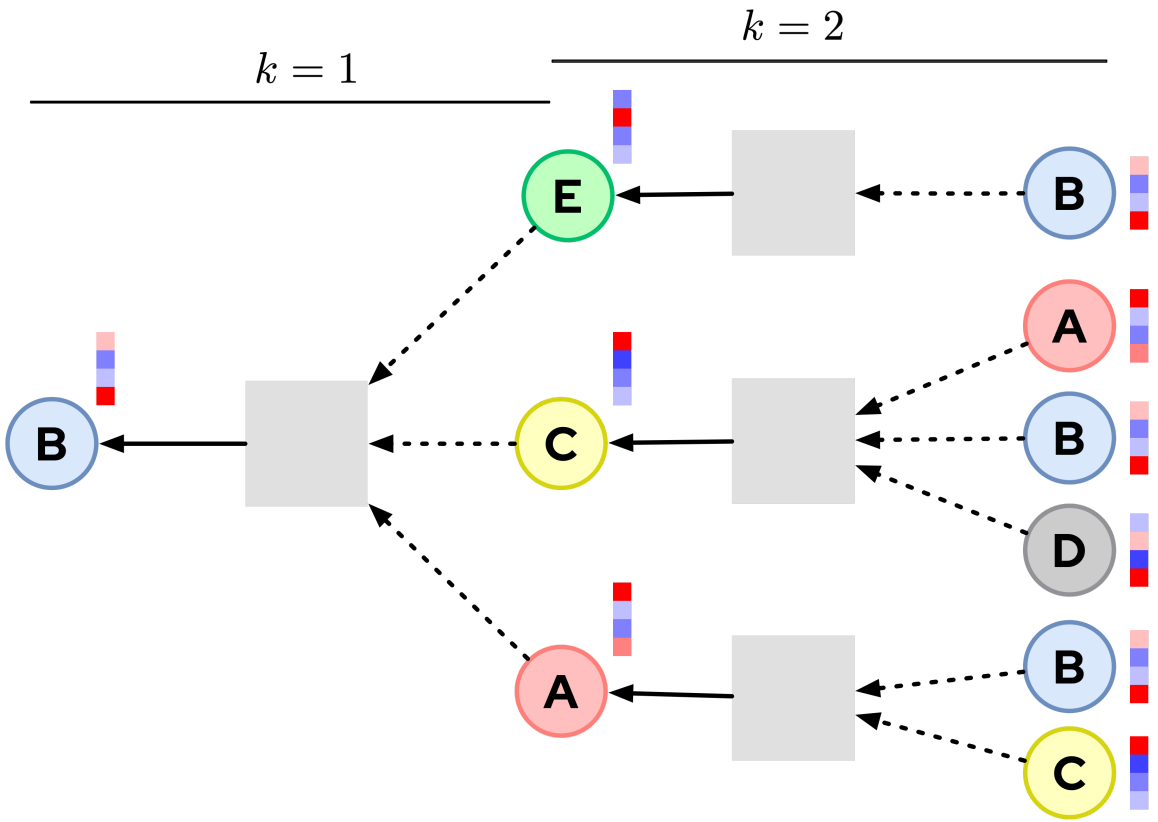
Neighbour Aggregation

Generate node embeddings based on local network neighborhoods (do you remember the random walk?!)

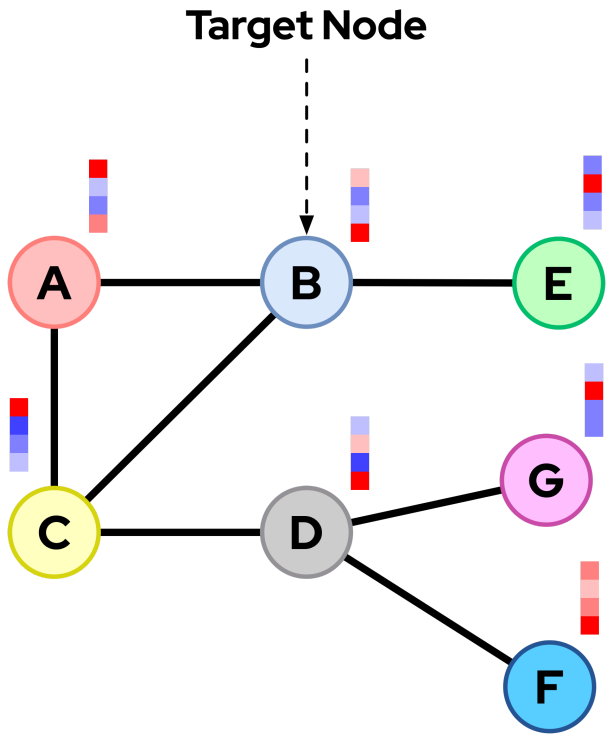


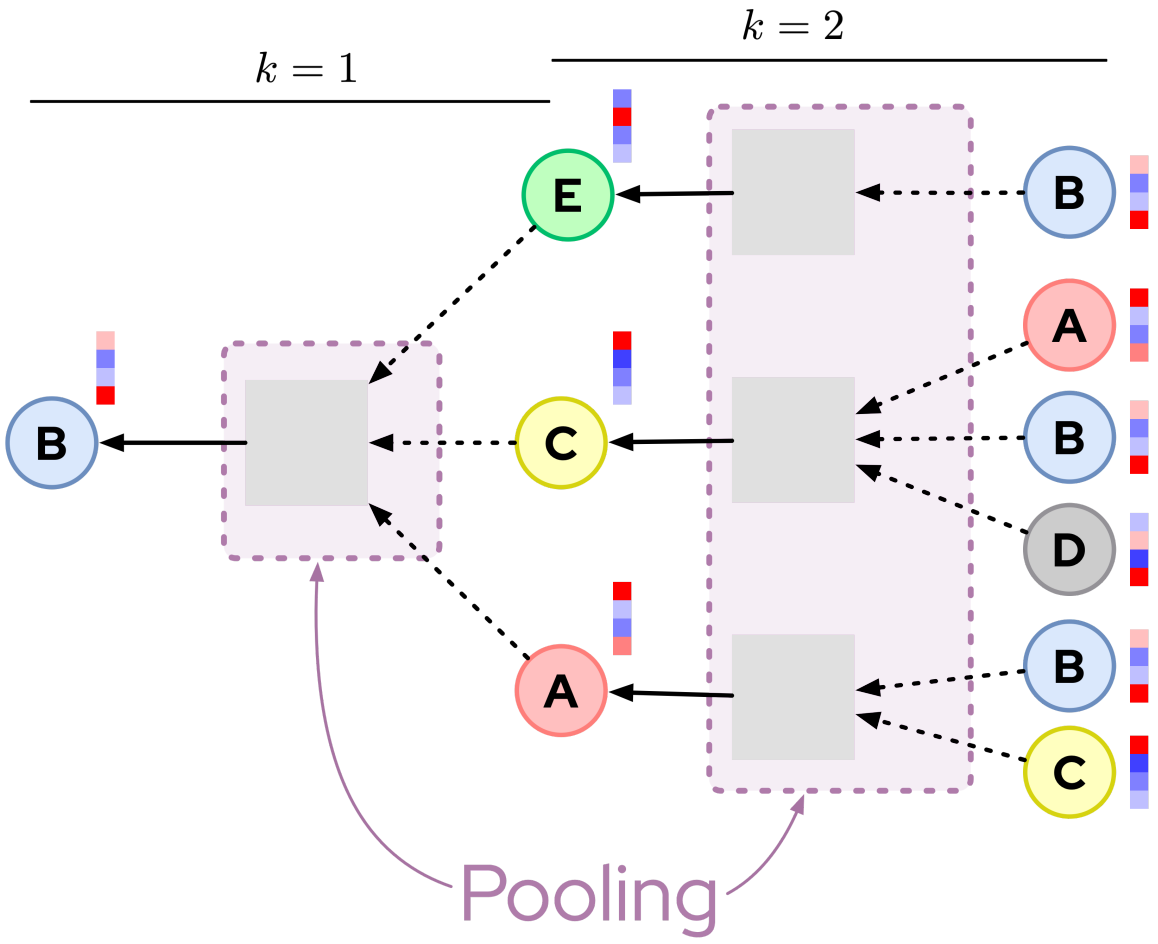
Neighbour Aggregation as a computational graph





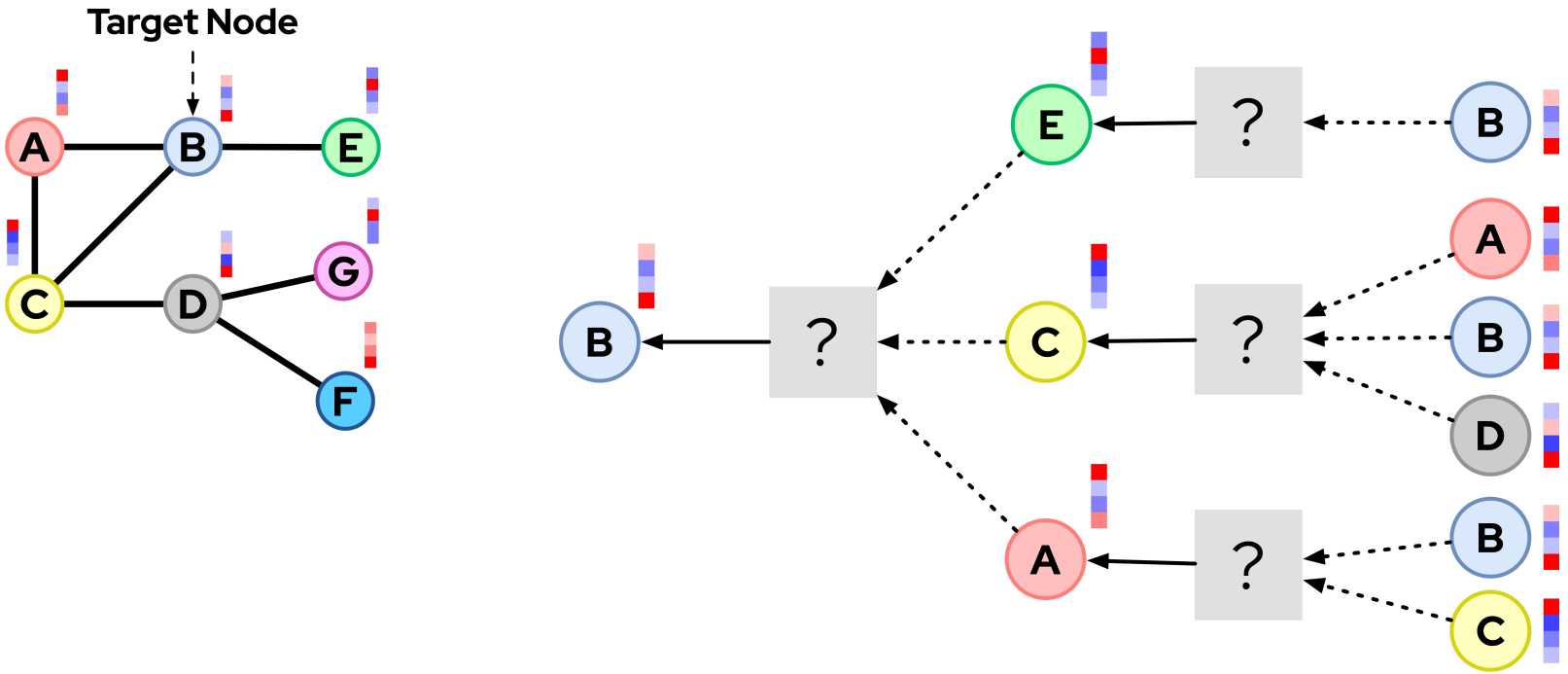
Neighbour Aggregation as a computational graph





Neighbour Aggregation

The key distinctions are in how different approaches aggregate information across the layers.

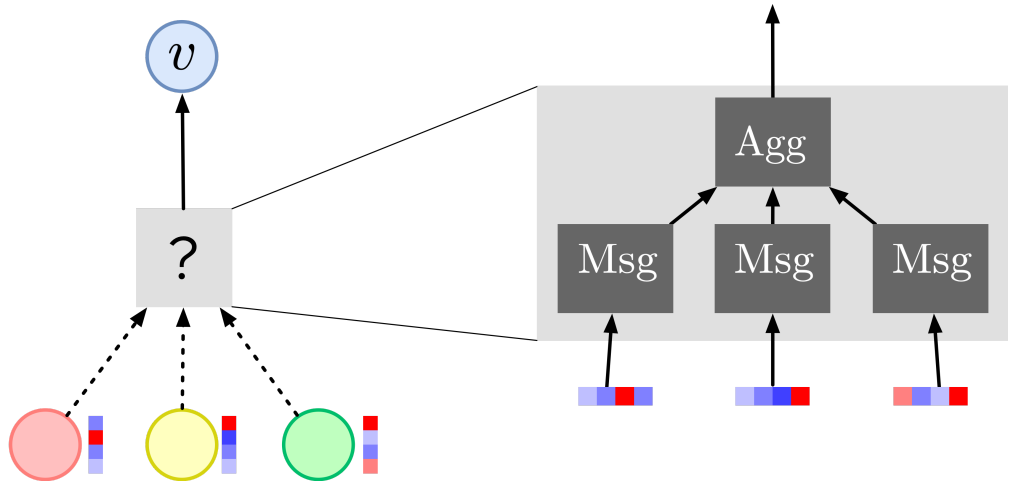


What's in the boxes?

GNN - The Message Passing Framework

Consider a single GNN layer. The message passing framework is a general way to define the update rule for each node, and it consists of two main steps:

1. **Message computation**: Each node v sends a message to its neighbors, based on its own features and the features of its neighbors.
2. **Message aggregation**: Each node v aggregates the messages received from its neighbors, and updates its own features.



$$\mathbf{x}_v = \mathbf{h}_v^{(0)} \rightsquigarrow \mathbf{h}_v^{(0)} \rightsquigarrow \mathbf{h}_v^{(1)} \rightsquigarrow \mathbf{h}_v^{(2)} \rightsquigarrow \cdots \rightsquigarrow \mathbf{h}_v^{(L)}$$

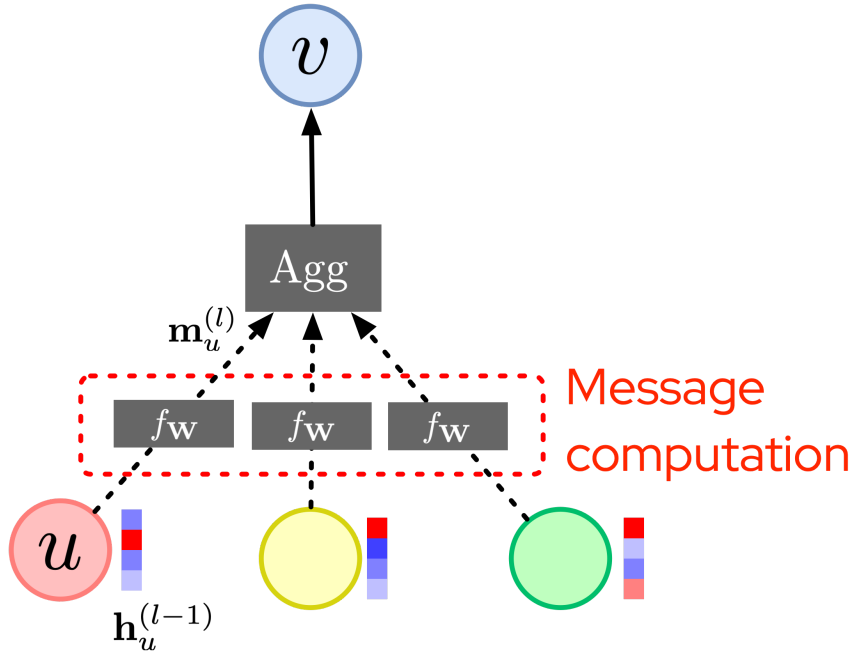
MPF - Message computation

$$\mathbf{m}_u^{(l)} = \mathrm{MSG}^{(l)}(\mathbf{h}_u^{(l-1)})$$

Each node will create a message, which will be sent to other nodes later.

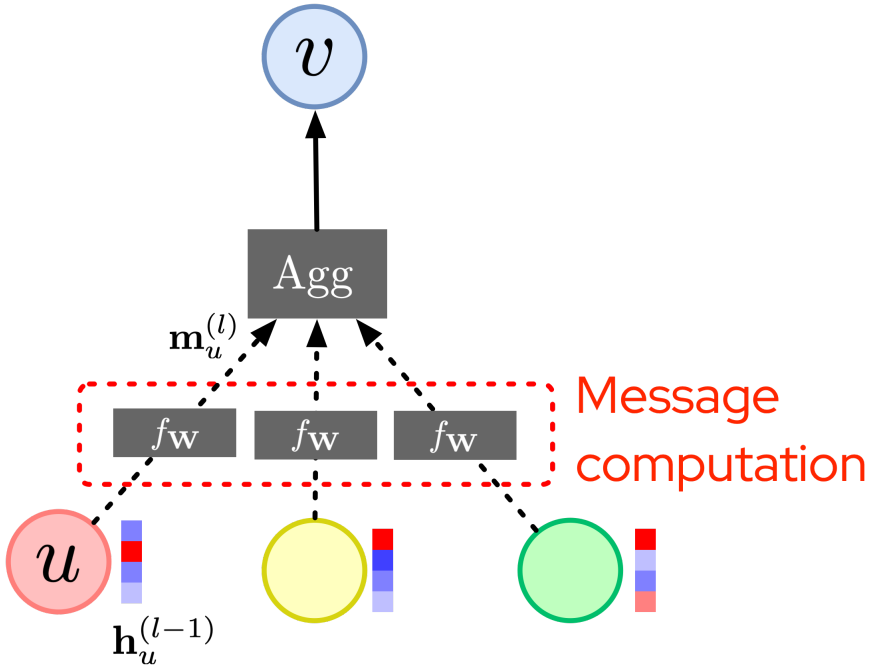
The function $\mathrm{MSG}^{(l)}$ is usually a neural network that takes as input the features of node u at layer $l-1$, and returns a message $\mathbf{m}_u^{(l)}$. E.g.,

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



MPF - Message computation

- The parameters of the message computation function $\mathrm{MSG}^{(l)}$ are shared across the nodes.
- The message computation function is **permutation-equivariant**, because the same function is applied to all nodes in the graph independently.



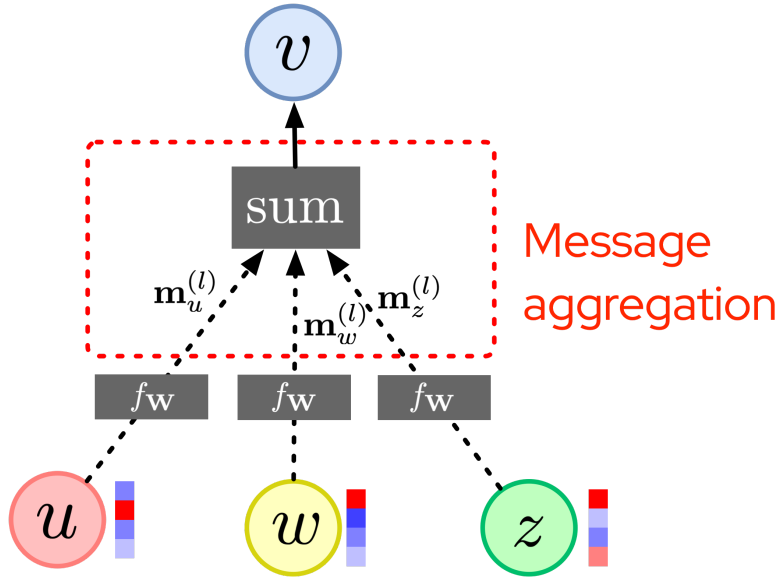
Both properties are crucial to ensure that the GNN cen be applied to graphs of different sizes!

MPF - Message aggregation

$$\mathbf{h}_v^{(l)} = \mathbf{Agg}^{(l)}(\{\mathbf{m}_u^{(l)}, \forall u \in \mathcal{N}(v)\})$$

Node v will aggregate the messages from its neighbors. The aggregation function must be **permutation-invariant**! E.g., $\mathbf{sum}(\cdot), \mathbf{mean}(\cdot), \mathbf{max}(\cdot), \ldots$

$$\mathbf{h}_v^{(l)} = \mathrm{sum}(\mathbf{m}_u, \forall u \in \mathcal{N}(v))$$



Note: potential parameters associated with the aggregation function $\mathbf{Agg}^{(l)}$ are shared across the nodes.

Issue with messagge aggregation

Information from node v itself could get lost: the computation of $\mathbf{h}_v^{(l)}$ does not directly depend on $\mathbf{h}_v^{(l-1)}$. Let's fix this:

- MESSAGE: compute message from node v itself

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

- AGGREGATION: after aggregating from neighbors, we can aggregate the message from node v itself

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

MPF as a general GNN framework

Many of the most famous and successful GNN models can be cast in the MPF. Specifically, we will cover:

- GCN : Graph Convolutional Network
- GraphSAGE : Graph Sample and Aggregation
- GAT : Graph Attention Network

Graph Convolutional Network

A single GCN layer can be expressed as follows:

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

Let's cast the GCN in the MPF framework:

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

Message

Aggregation

GCN

A single GCN layer can be expressed in matrix form as follows[1]:

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

where $\widetilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$, $\mathbf{D}$ the diagonal degree matrix, and $\widetilde{D}_{ii} = \sum_j \tilde{A}_{ij}$.

The original GCN paper assume that there is a self-loop in each node, and the formulation above is slightly different from the one presented in the previous slide:

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

As stated previously, the parameters associated with the node v couold potentially be different from the parameters associated with the neighbors of v.

1. Kipf, T. N., & Welling, M. (2016). Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907. Not that this formulation is slightly different wrt the one presented above.

GraphSAGE

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

Let's cast the GraphSAGE formulation in the MPF framework. A couple of observations are needed:

1. The message computation is "hidden" in the aggregation function $\text{Agg}^{(l)}$.
2. **The aggregation happens in two steps**: first, the messages of the neighbours are aggregated, and then they are concatenated with the node features.

$$\mathbf{h}_{\mathcal{N}(v)}^{(l)} = \text{Agg}^{(l)}\left(\left\{\mathbf{h}_u^{(l-1)}, \forall u \in N(v)\right\}\right) \quad \mathbf{h}_v^{(l)} = \sigma\left(\mathbf{W}^{(l)} \cdot \text{Concat}\left(\mathbf{h}_v^{(l-1)}, \mathbf{h}_{\mathcal{N}(v)}^{(l)}\right)\right)$$

GraphSAGE - Aggregation function

The first part of the aggregation happens in the $\mathbf{Agg}^{(l)}$ function along with the message computation. Let's see some examples:

MEAN aggregator

$$\mathbf{h}_{\mathcal{N}(v)}^{(l)} = \frac{1}{|\mathcal{N}(v)|} \sum_{u \in \mathcal{N}(v)} \mathbf{h}_u^{(l-1)} = \underbrace{\sum_{u \in \mathcal{N}(v)}}_{\text{Aggregation}} \underbrace{\frac{\mathbf{h}_u^{(l-1)}}{|\mathcal{N}(v)|}}_{\text{Message}}$$

POOLING aggregator

$$\mathbf{h}_{\mathcal{N}(v)}^{(l)} = \underbrace{\max}_{\text{Aggregation}} \left(\left\{ \underbrace{\sigma\left(\mathbf{W}_{\text{pool}}^{(l)} \mathbf{h}_u^{(l-1)} + \mathbf{b}\right)}_{\text{Message}}, \forall u \in \mathcal{N}(v) \right\} \right)$$

Graph ATtention Network

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

GAT introduces the concept of attention mechanism in the message computation and its formulation "generalizes" both GCN and GraphSAGE:

- $\alpha_{vu} = \frac{1}{|\mathcal{N}(v)|}$ is the weighting factor (i.e., the importance) of node u's message to node v
- The importance is solely **based on the structural properties** of the graph (node degree)
- All neighbors are **equally important** to v

GAT - Attention mechanism

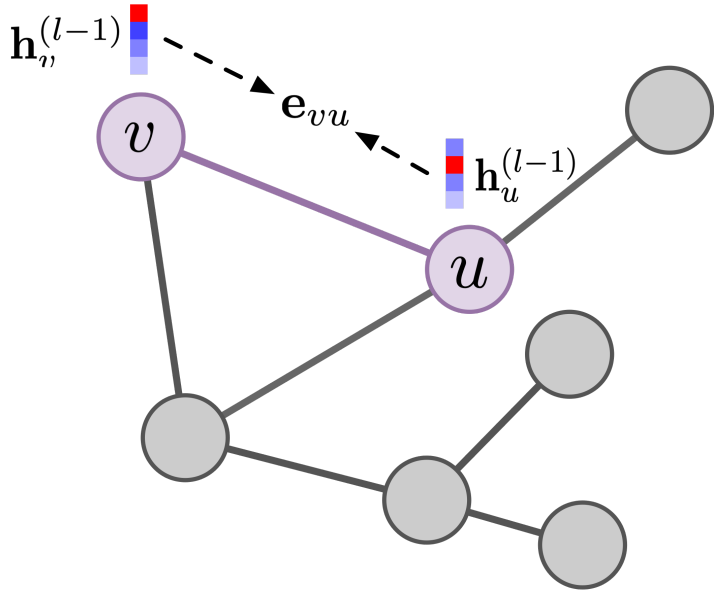
IDEA: **not all neighbors are equally important**. The importance of a neighbor u to node v should be learned from the data. Which part of the data is more important depends on the context and is learned through training.

GAT - Attention mechanism

Let α_{vu} be computed as a byproduct of an attention mechanism Att.

- Let Att compute attention coefficients e_{vu} across pairs of nodes v, u based on their messages:

$$e_{vu} = \mathrm{Att}(\mathbf{W}^{(l)}\mathbf{h}_v^{(l-1)}, \mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)})$$



GAT - Attention mechanism

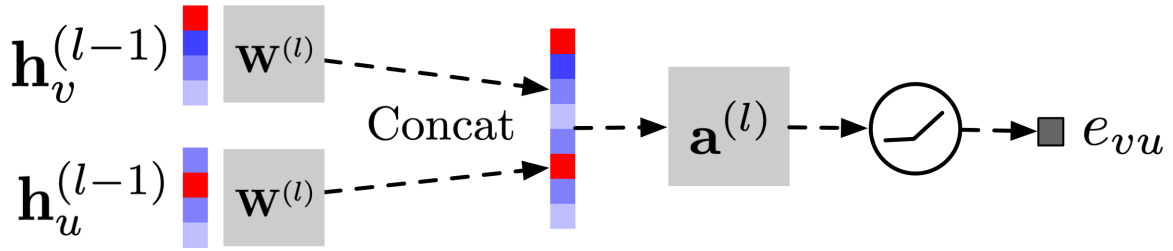
- **Normalize the attention coefficients** (to avoid scaling issues) across all neighbors of v using the softmax function, s.t. $\sum_{u \in \mathcal{N}(v)} \alpha_{vu} = 1$:

$$\alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{w \in \mathcal{N}(v)} \exp(e_{vw})}$$

GAT - Attention mechanism

The attention mechanism $\mathbf{Att}$ can be implemented in different ways, but the overall appraoch is agnostic to the specific implementation. The attention mechanism can be implemented as a simple feedforward neural network, e.g.,

$$e_{vu} = \mathbf{LeakyReLU}\left(\mathbf{a}^{(l)\top}\left[\mathbf{W}^{(l)}\mathbf{h}_v^{(l)} \| \mathbf{W}^{(l)}\mathbf{h}_u^{(l)}\right]\right)$$



GNN Framework hierarchy

$$\mathbf{GCN} \subset \mathbf{GraphSAGE} \subset \mathbf{GAT} \subset \mathbf{MPF}$$

Training a GNN

- Given a GNN with L layers, the embedding of a node v at the last layer is given by:

$$\mathbf{z}_v = \mathbf{h}_v^L = \mathrm{ENC}_\Theta(\mathbf{A}, \mathbf{x}_v)$$

  where ENC_Θ is the GNN model (e.g., GCN, GraphSAGE, GAT, etc.).

- On top of the GNN, we define **further layers** (e.g., fully connected layers) to perform the **downstram task** we want to solve (e.g., node classification, link prediction, etc.):

$$\mathbf{y}_v = \mathrm{DEC}_{\Theta'}(\mathbf{z}_v)$$

- To train the GNN, we need to define a **loss function** and optimize it using gradient-based methods.

Supervised learning with GNNs

Assuming a node-level task, we aim to minimize the following generic loss function:

$$\min_{\Theta,\Theta'} \sum_{v} \mathcal{L}(\mathbf{y}_v, \mathrm{DEC}_{\Theta'}(\mathrm{ENC}_{\Theta}(\mathbf{A}, \mathbf{x}_v)))$$

which can be optimized using gradient-based methods. In this case, the examples are the nodes of the graph!

Model parameters

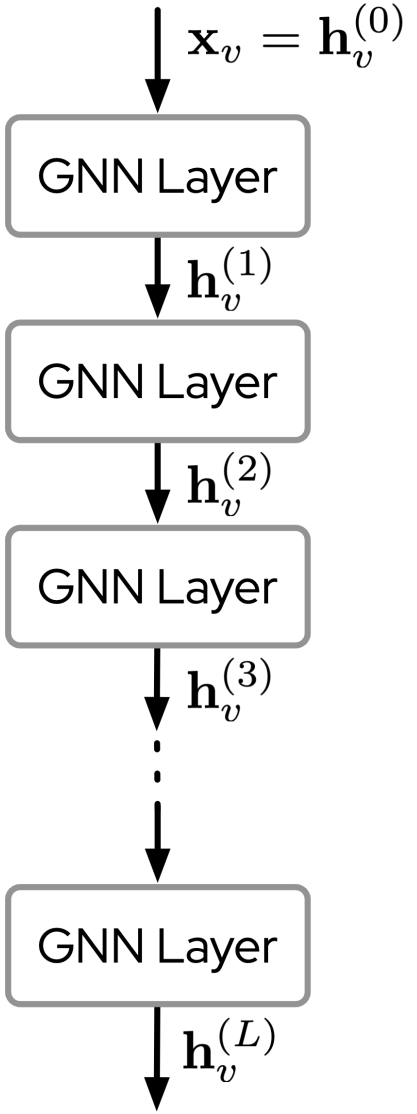
The parameters of the GNN model are the parameters of the message computation and aggregation functions, and the parameters of the network for the downstream task (e.g., fully connected layers).

- GCN: the parameters are the weight matrices $\mathbf{W}^{(l)}$ and the parameters for solving the downstream task.
- GraphSAGE: the parameters are the weight matrices $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$, the parameters of the aggregation function $\mathrm{Agg}^{(l)}$ and the parameters for solving the downstream task.
- GAT: the parameters are the weight matrices $\mathbf{W}^{(l)}$ and the parameters of the attention mechanism $\mathbf{a}^{(l)}$, and the parameters for solving the downstream task.

Stacking GNN layers

We focused on describing how a single GNN layer works. However, in practice, we may stack multiple GNN layers to enhance the expressivity of the model.

Ok, so we are done, right? We can just stack multiple GNN layers and we are good to go, right?



The over-smoothing problem

GNN suffers from the over-smoothing problem.

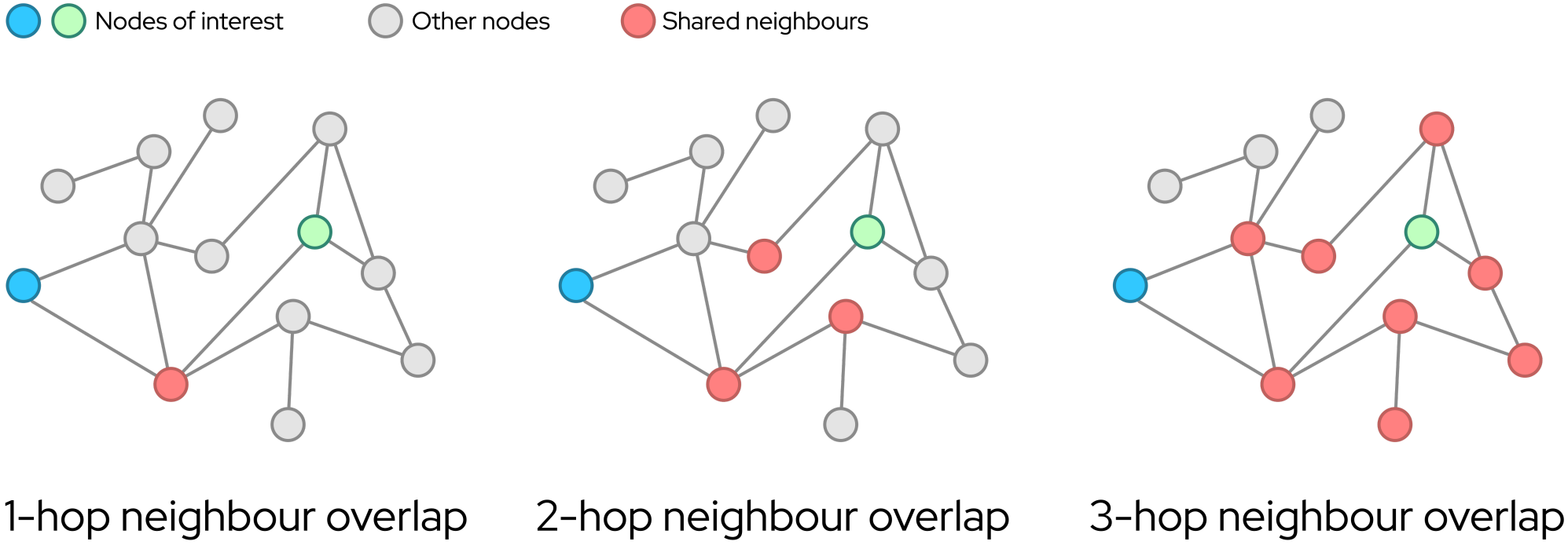
The over-smoothing problem

All the node embeddings converge to the same value.

This means that we are not able to capture the structural properties of the graph, and the model is not able to distinguish between different nodes.

Why the over-smoothing problem?

Stacking multiple layers corresponds to **enlarging the receptive field of a node**. The receptive field of a node is the set of nodes that influence the node's embedding.



Overlapping receptive fields

The **shared neighbors quickly grows** when we increase the number of hops.

Since the node embedding is determined by its receptive field, if two nodes have highly-overlapped receptive fields, then their **embeddings are highly similar**

and so...

Stack many GNN layers ⇒ nodes will have highly overlapped receptive fields ⇒ node embeddings will be highly similar ⇒ suffer from the oversmoothing problem

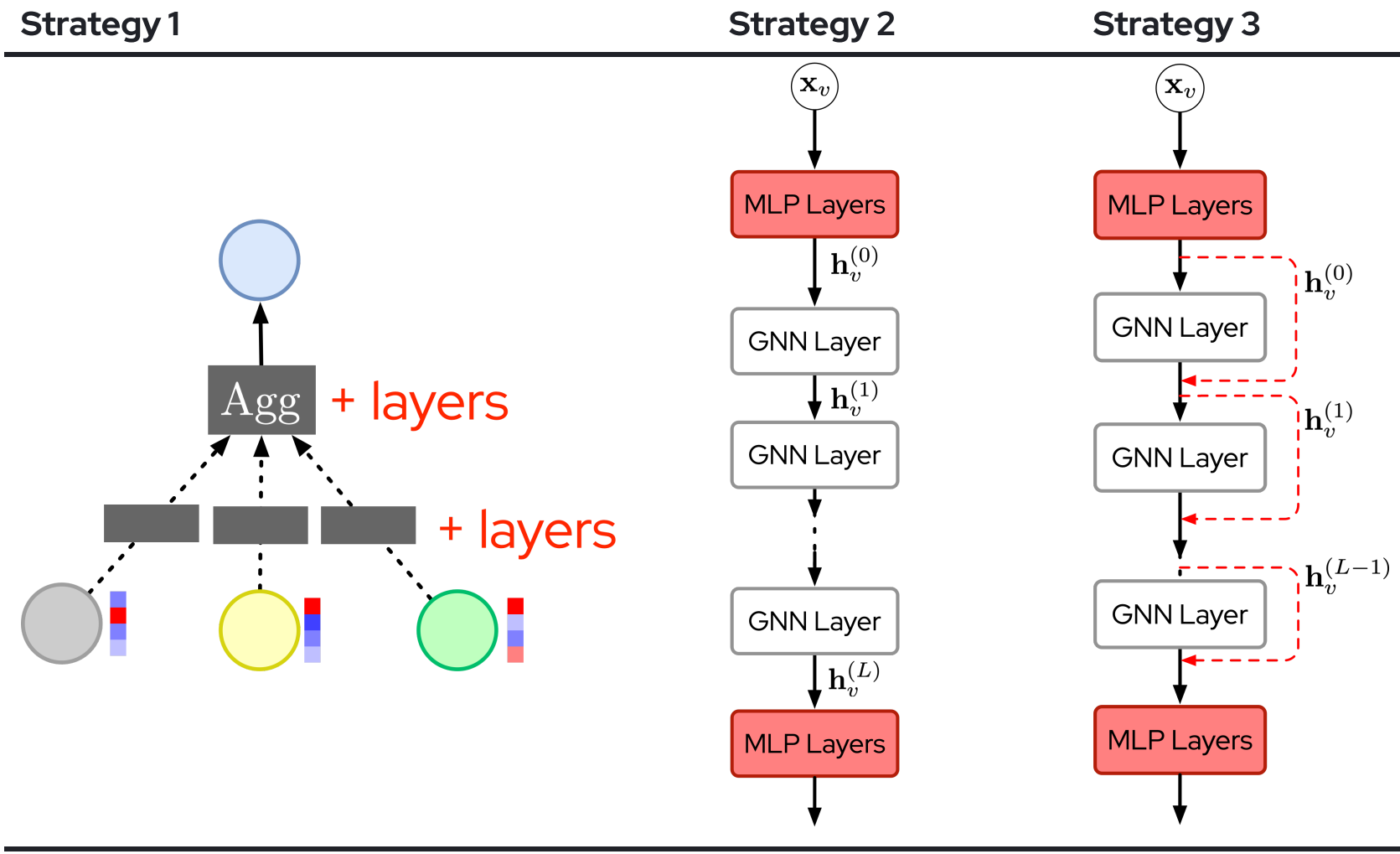
Are we hopeless?

How do we enhance the expressivity of a GNN?

Three main strategies:

1. **Increase the expressive power within each GNN layer** $\Rightarrow$ message computation and aggregation become more complex
2. **Add layers that do not pass messages** $\Rightarrow$ add more layers before and after the GNN layers
3. **Skip connections**: Node embeddings in earlier GNN layers can sometimes better differentiate nodes $\Rightarrow$ add shortcuts in GNN

Enhancing the expressive power of GNNs



References

- Kipf, T. N., & Welling, M. (2016). Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907.
- Hamilton, W. L., Ying, R., & Leskovec, J. (2017). Inductive representation learning on large graphs. In Advances in neural information processing systems (pp. 1024-1034).
- Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., & Bengio, Y. (2017). Graph attention networks. arXiv preprint arXiv:1710.10903.