Graph Neural Networks

Jure Leskovec, Stanford CS224W: Machine Learning with Graphs, http://cs224w.stanford.edu

Graphs

- Data structures:
  - Elements expressed as nodes or vertices
  - Relationships between elements expressed as edges

- Links are key to power of relational-data
  - Allow to learn more about the system
  - Give new tools for analyzing data, and predict new properties from it
  - In contrast to tabular data that is common in traditional learning problems

Graphs

- Many problems in real world are actually graph-based problems in disguise
  - Appearing in nature (molecules), society (social networks), technology (the internet), and everyday settings (roadmaps)

- Need a specialized form of neural network dedicated to work on rich and ubiquitous data type graph
  - Use Graph neural network or GNN

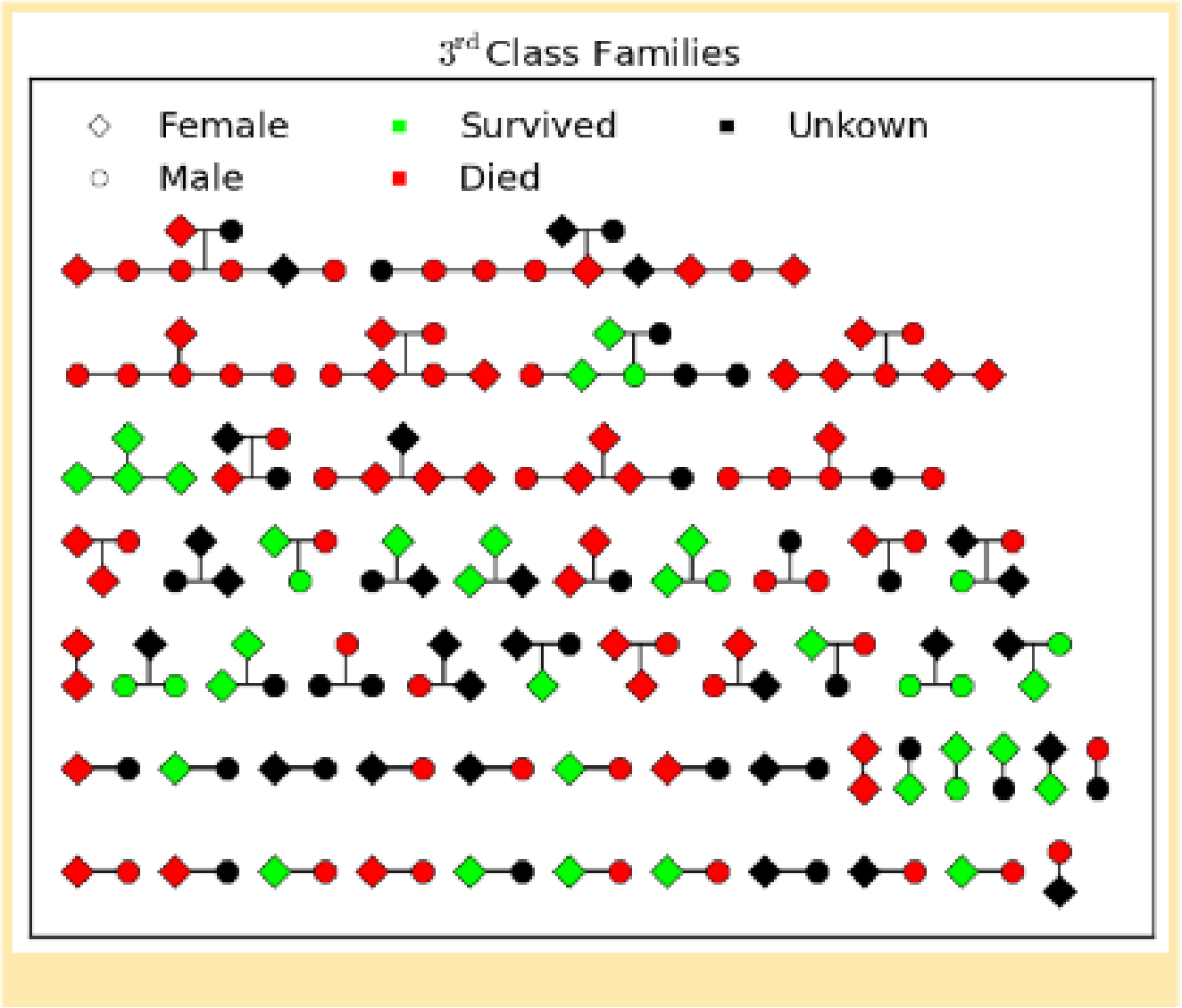
Ex. Titanic Dataset

- Describes passengers of the Titanic ship, many of whom met an untimely end when colliding with an iceberg
  - Dataset historically described in tabular format, but full of unexplored relationships
  - Commonly used for classification and has a target variable (survivorship)
  - Accounts of relationships are typically hidden
  - Ex. hard to work out social links between people
  - Many people on ship share multiple familial types of relationships
    - Including marital and family relations, business and employment relationships, and friendships

https://www.kaggle.com/competitions/titanic/data

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

Tabular versions of the Titanic dataset give high level data on this such as boolean indicators - the alone feature is false if a person had immediate family on the ship) or counts of immediate family relationships



- **The Titanic Dataset: The family relationships of the Titanic visualized as a graph (credit: Matt Hagy).**
- **There was a rich social network as well as many passengers with unknown family ties**

https://livebook.manning.com/book/graph-neural-networks-in-action/

Graph Neural Network (GNN)

- An algorithm that allows to represent and learn from graphs, including their constituent nodes, edges and features
- Provide analogous advantages to conventional neural networks
  - Except that the learning happens on relational data
- Applying traditional machine learning and deep learning methods to graph data structures has proven to be challenging
  - Graph data, when represented in grid-like formats and data structures, can lead to massive repetitions of data

Graph Neural Network (GNN)

- In particular, graph-based learning focuses on approaches that are *permutation invariant*
  - → machine learning method is uninfluenced by ordering of the graph representation
  - →if we shuffle rows and columns of the adjacency matrix then the algorithms performance should not change

GNN

- In a conventional neural network, first need to initialize a set of parameters and functions
  - Include number of layers, size of layers, learning rate, loss function, batch size
  - Then train network by iteratively updating these parameters

A Input data

B Pass data through neural network layers that transform data according to parameters of the layer and an activation rule

C Output a representation from final layer of the network

D Backpropagate the error and adjust parameters accordingly

E Repeat steps a fixed number of epochs or until training has converged

GNN

- For graph-based or relational data, steps are similar except that each epoch relates to one iteration of *message passing*
- Message passing is how GNNs encode information about the graph
  - Graph-updating layers to encode and propagate local information passed to downstream machine learning tasks
  - Still have to initialize neural network parameters, but now also initialize a representation of nodes of the graph

GNN

A Input graph data

B Update node or edge representations using message passing

C Pass node/edge representations to neural network layers

D Output a representation from the final layer of the network

E Backpropagate error and adjust the parameters accordingly

F Repeat steps a fixed number of epochs or until training has converged

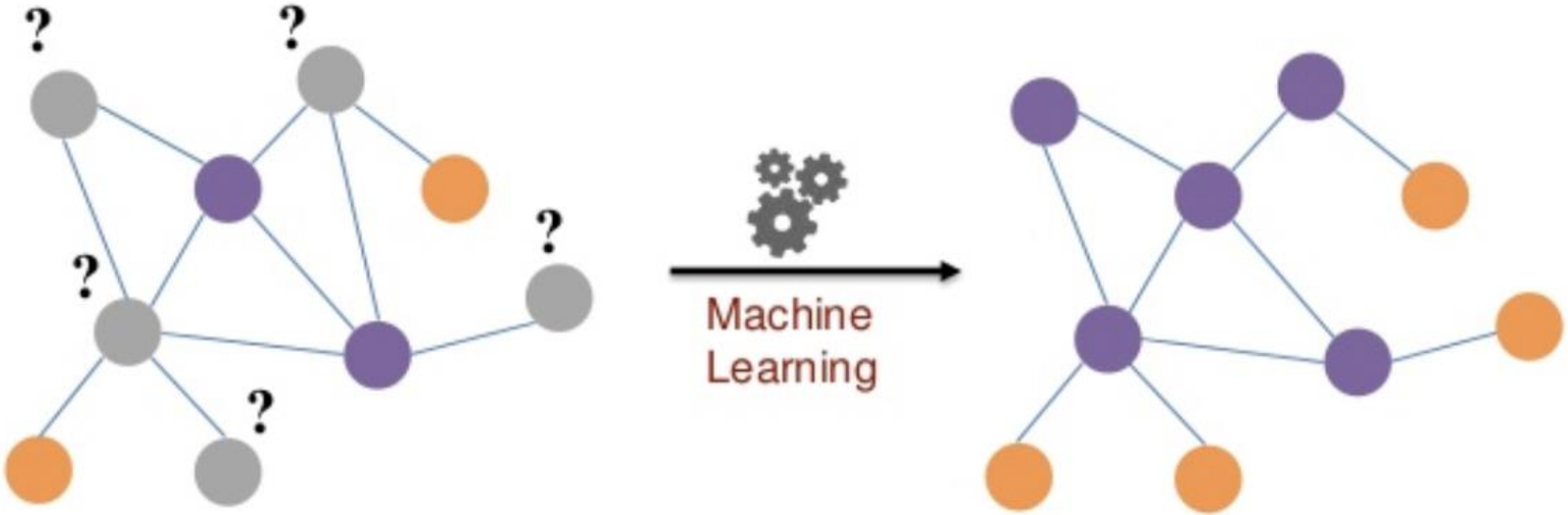
When to Use GNN

Where a GNN may be a good fit:

- Data was modeled as a graph
  - A graph-based model is almost always helpful when relationships are an important characteristic for the situation being modeled
- Nodes and edges have non-trivial information
- A prediction task was involved
  - PageRank, the first algorithm used by Google to order web search results, is probably the most famous example of a graph algorithm used to great success by an enterprise

Node Level Tasks

- Nodes in graphs often have attributes and labels
- Node classification: given a graph with node classes, aim to accurately predict class of an unlabeled node
- Ex., might use node classification if we want to guess whether a passenger is family with another passenger



Edge Level Tasks

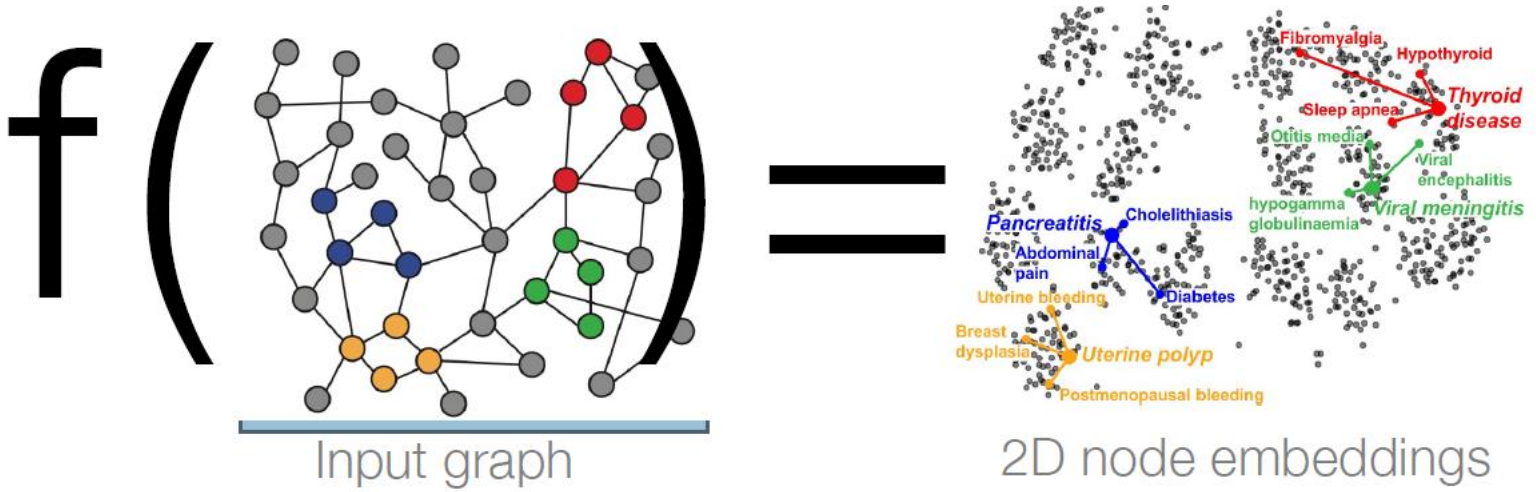
- Edge-level tasks - predicting a link's label in a supervised or semi-supervised way
- Might also want to predict whether two nodes are connected when do not have the full network
  - Known as edge-prediction
  - Describes whether there is an edge between nodes

Graph Level Tasks

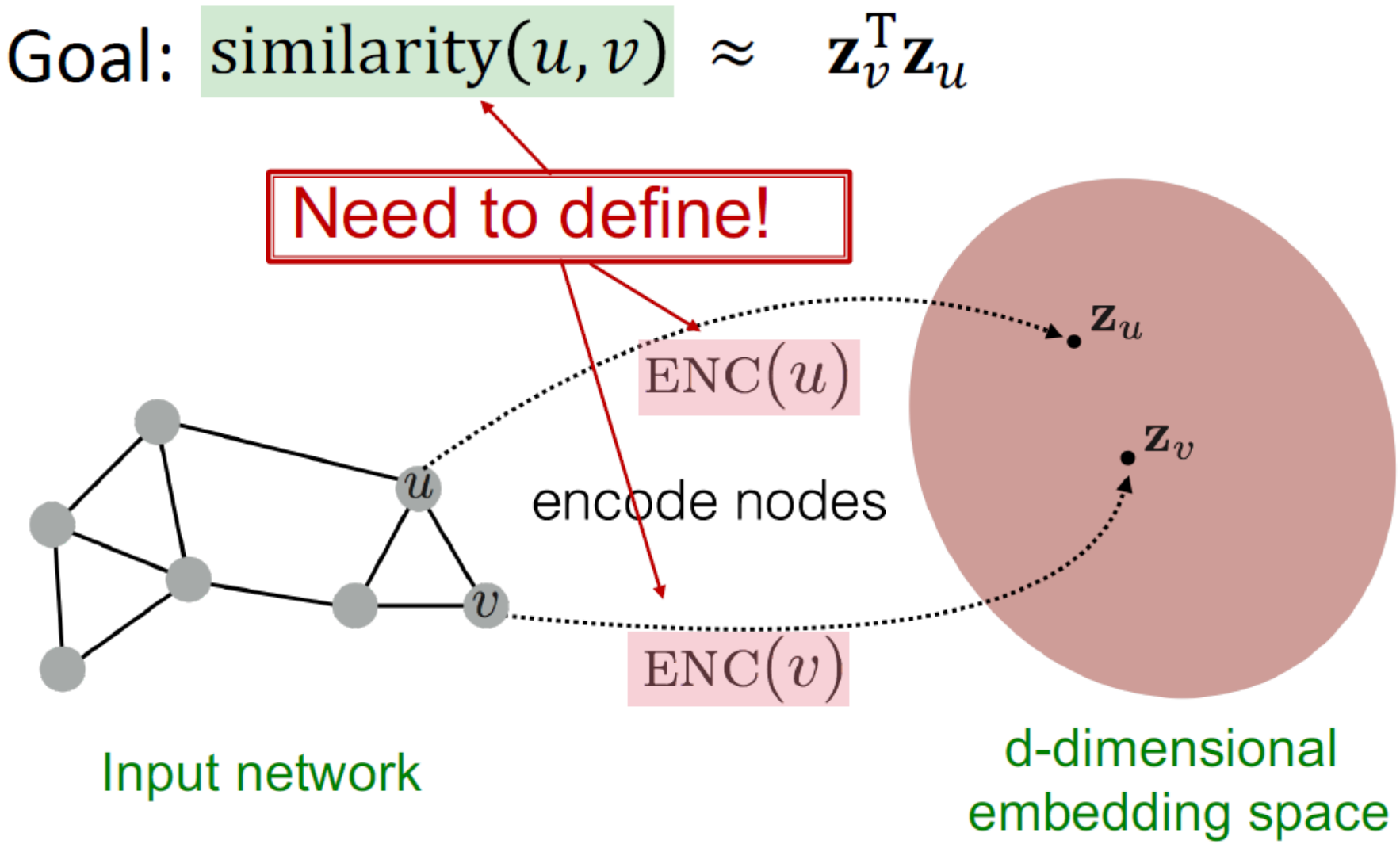
- Graph classification and regression - predicting the label or a quantity associated with the graph
- There are also unsupervised tasks that involve GNNs
  - Involve Graph Auto Encoders that embed graphs and do the opposite process of generating a graph from an embedding

Node Embeddings

- Intuition: Map nodes to *d*-dimensional embeddings such that similar nodes in the graph are embedded close together
- **How to learn mapping function f?**



Node Embeddings



Key Components

- **Encoder:** Maps each node, *v* in input graph to a low-dimensional vector

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

d-dimensional embedding

- **Similarity function:** Specifies how relationships in vector space map to relationships in the original network

$$\text{similarity}(u, v) \approx \mathbf{z}_v^{\mathrm{T}} \mathbf{z}_u$$

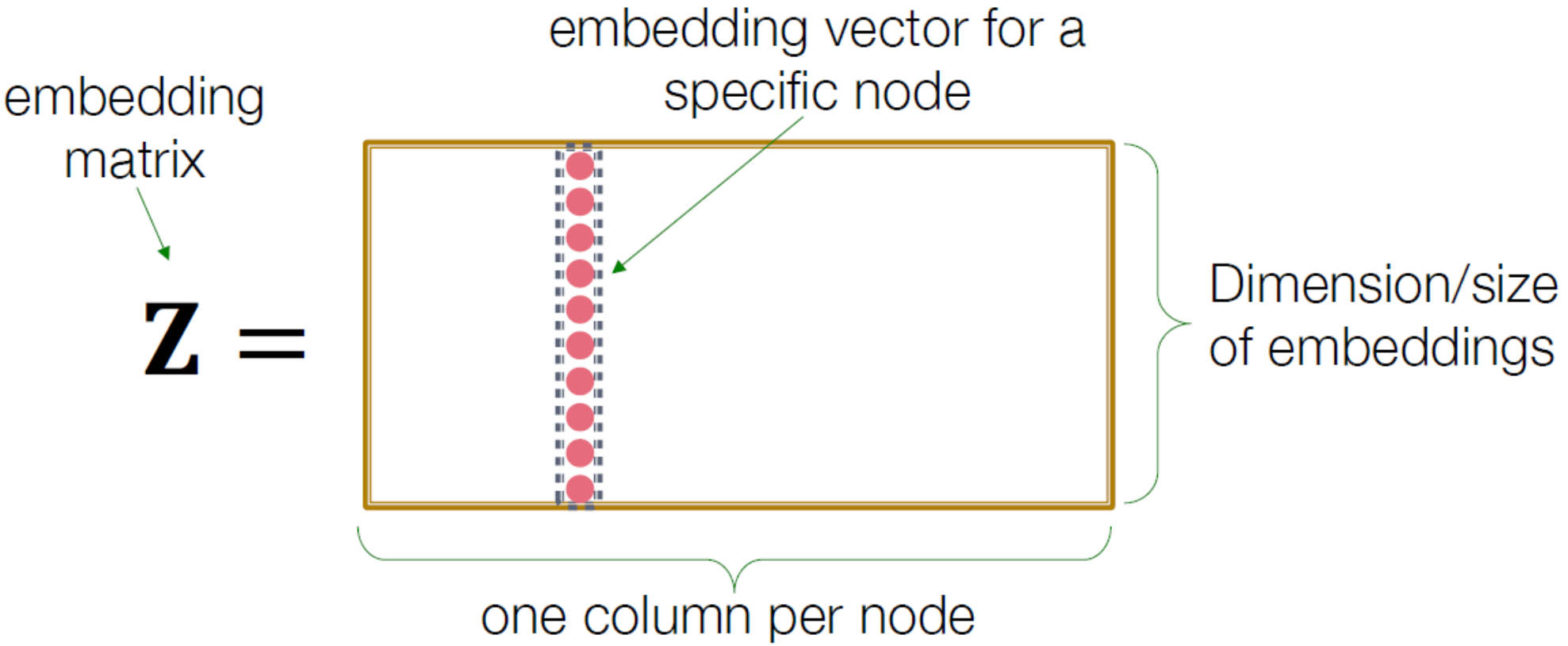
Decoder

Similarity of u and v in the original network

dot product between node embeddings

“Shallow” Encoding

- Simplest encoding approach: **Encoder is just an embedding-lookup**



Shallow Encoders

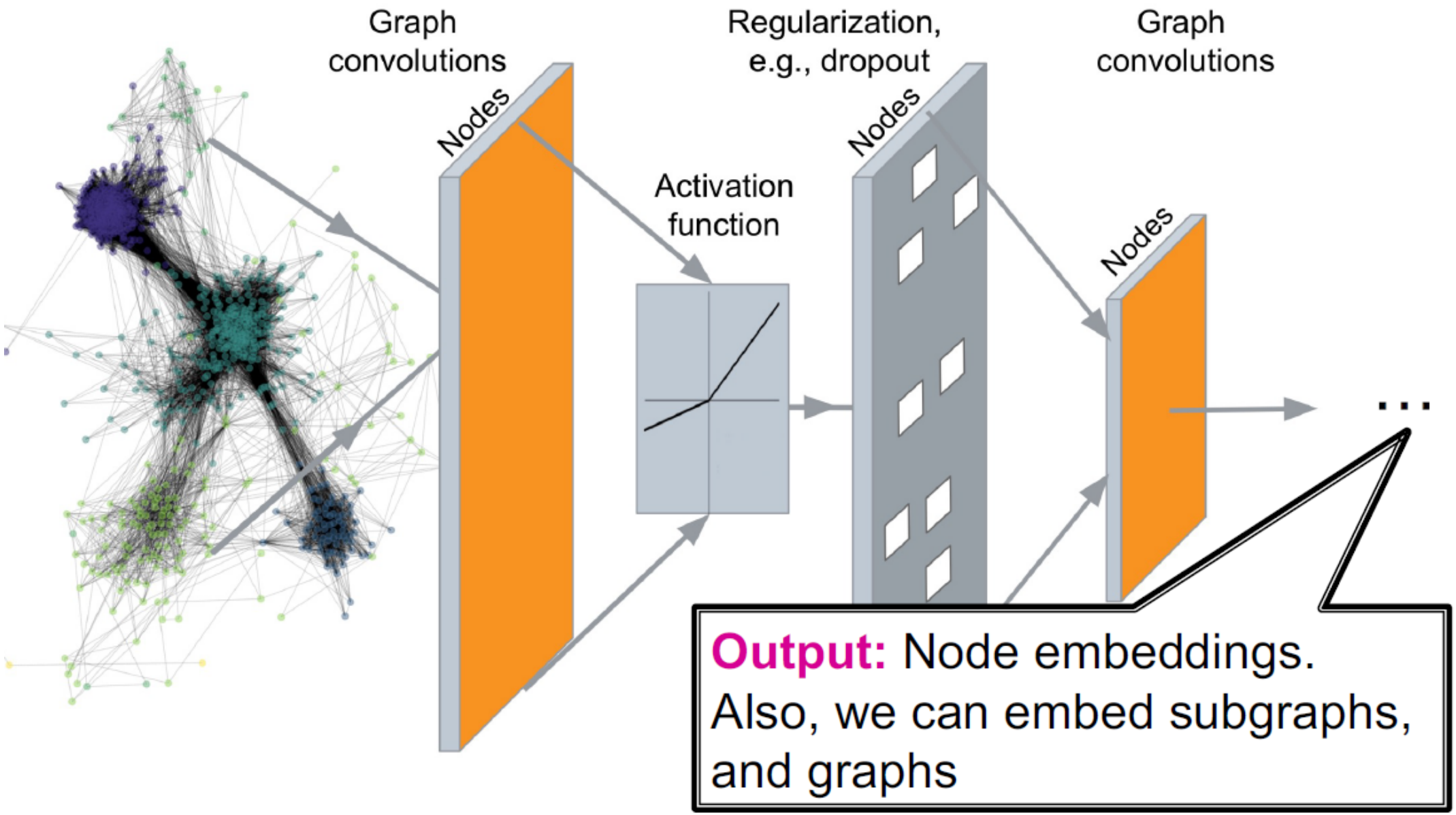
Limitations of shallow embedding methods:

- **$O(|V|d)$ parameters are needed**:
  - No sharing of parameters between nodes
  - Every node has its own unique embedding
- **Inherently "transductive"**:
  - Cannot generate embeddings for nodes that are not seen during training
- **Do not incorporate node features**:
  - Nodes in many graphs have features that we can and should leverage

Deep Graph Encoders

- Deep learning methods based on **graph neural networks (GNNs):**

$$\mathrm{ENC}(v) =$$ **multiple layers of non-linear transformations based on graph structure**

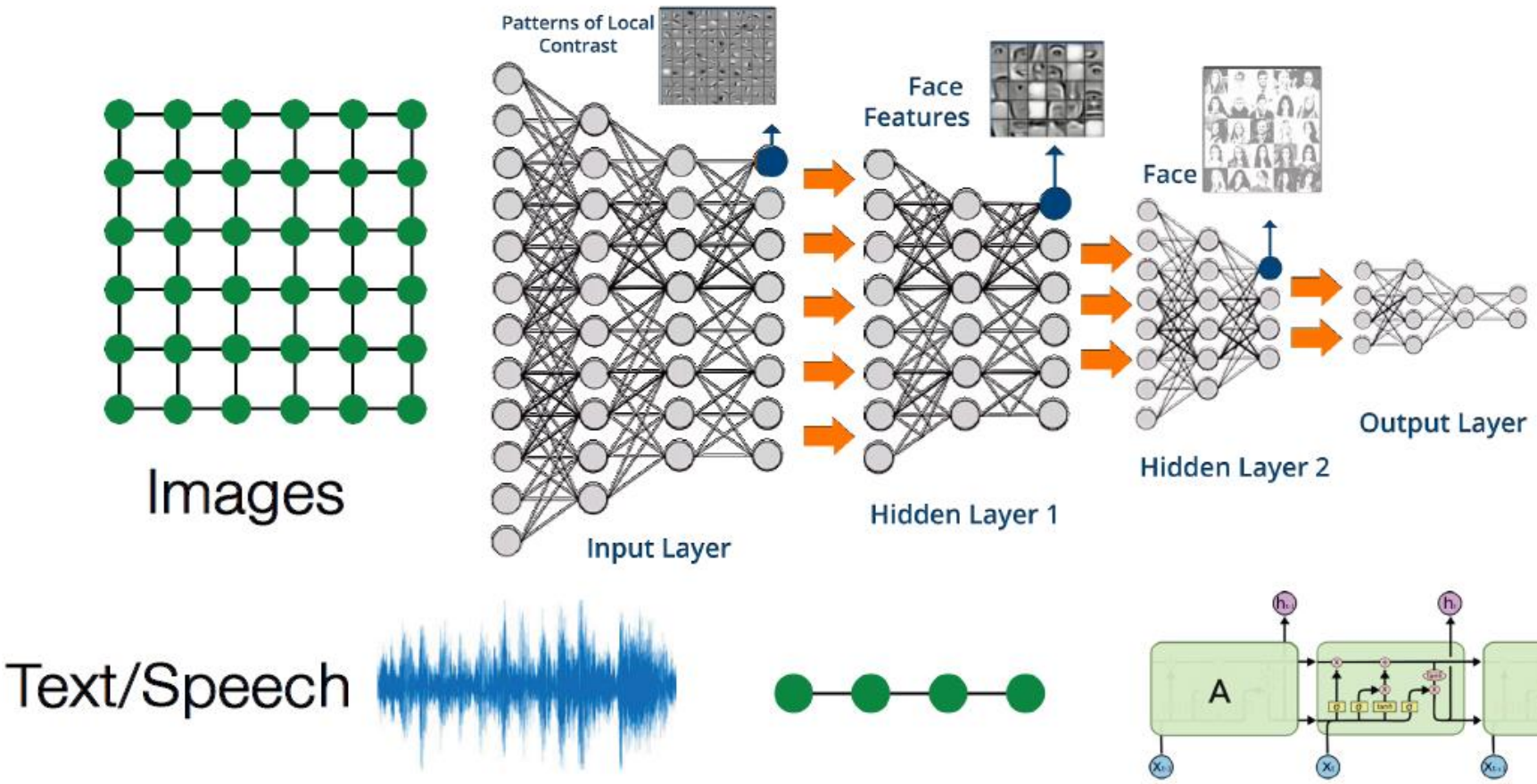

Graph convolutions
Nodes
Activation function
Regularization, e.g., dropout
Nodes
Graph convolutions
Nodes
...
Output: Node embeddings.
Also, we can embed subgraphs, and graphs

Tasks on Networks

Tasks we will be able to solve:

- Node classification
  - Predict the type of a given node
- Link prediction
  - Predict whether two nodes are linked
- Community detection
  - Identify densely linked clusters of nodes
- Network similarity
  - How similar are two (sub)networks

Modern ML Toolbox

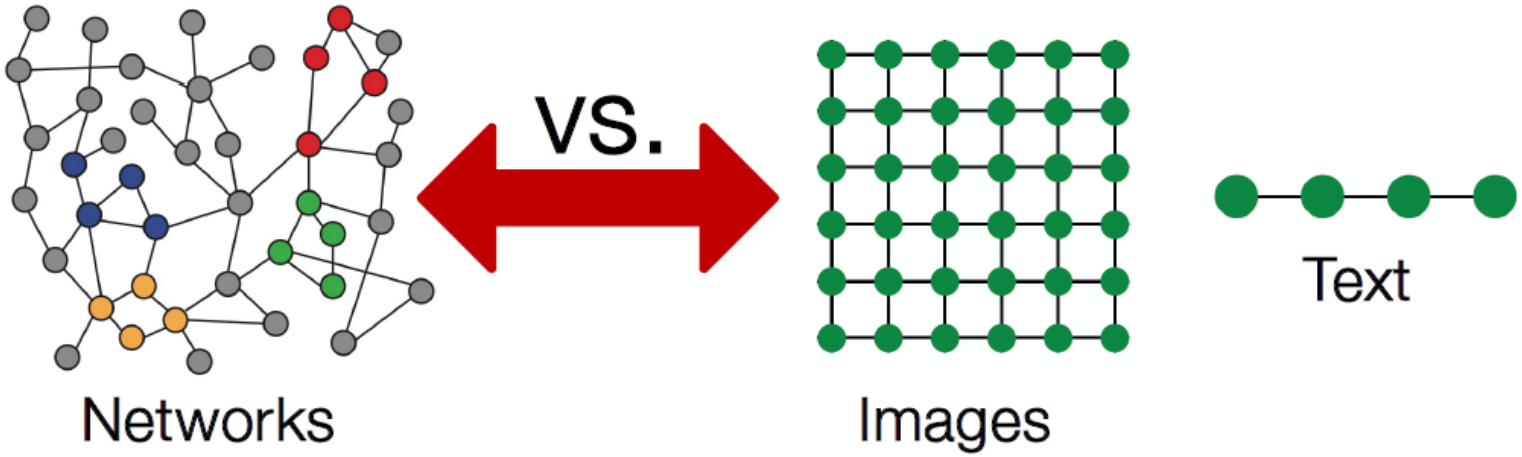


Designed for simple linear sequences and grids

Why is it Hard??

Graph/Networks are far more complex!

- 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



Setup

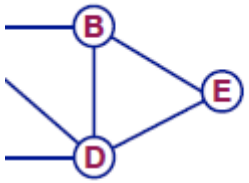
Assume a graph G:

- V: **vertex set**
- $\boldsymbol{A}$: **adjacency matrix** (assume binary → unweighted graph)
- $\boldsymbol{X} \in \mathbb{R}^{|V| \times m}$: matrix of **node features**
- v: node in V; $N(v)$: set of neighbors of v
- **Node features:**
  - Social networks: User profile, User image
  - Biological networks: Gene expression profiles, gene functional information
  - When there is no node feature in the graph dataset:
    - Indicator vectors (one-hot encoding of a node)
    - Vector of constant 1: [1, 1, …, 1]

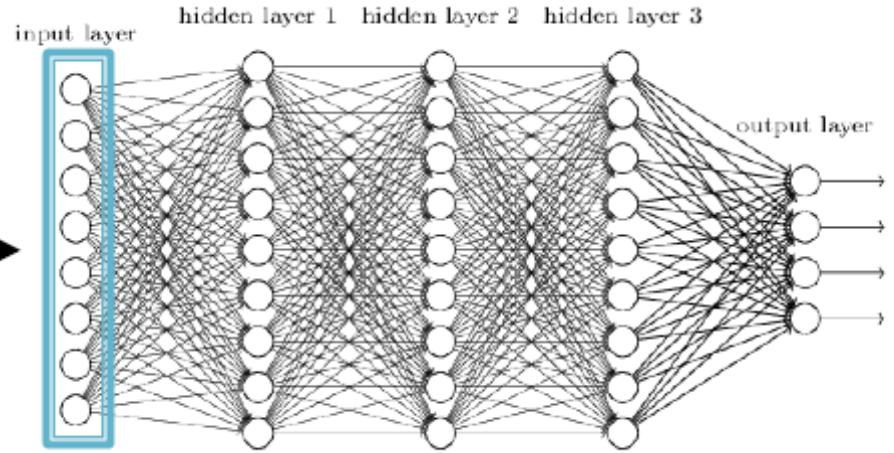
Naïve Approach

Naïve approach to apply deep neural networks to graphs:

- Join adjacency matrix and features
- Feed them into a deep neural net:



	A	B	C	D	E	Feat	
A	0	1	1	1	0	1	0
B	1	0	0	1	1	0	0
C	1	0	0	1	0	0	1
D	1	1	1	0	1	1	1
E	0	1	0	1	0	1	0

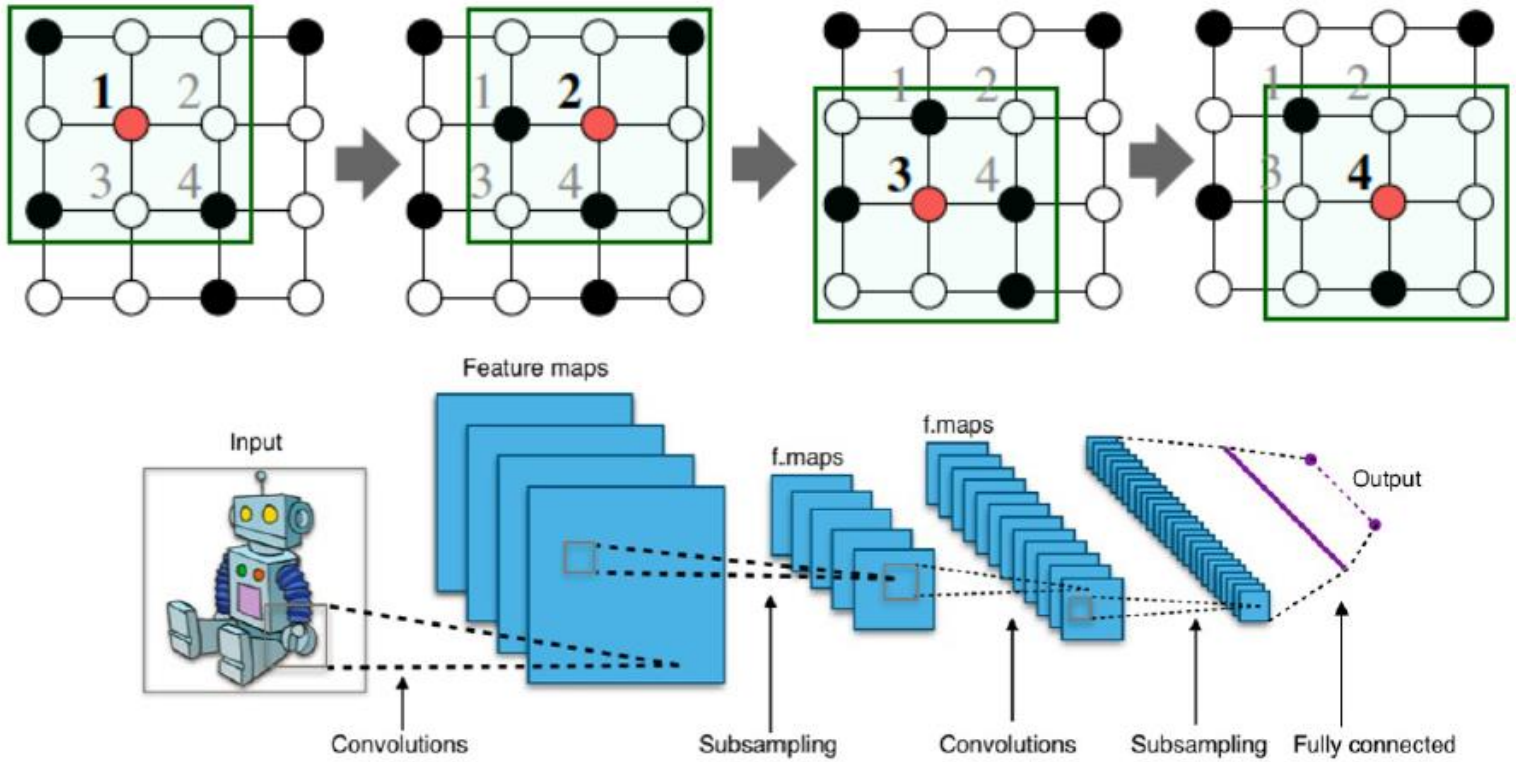


?

- Issues with this idea:
  - $O(|V|)$ parameters
  - Not applicable to graphs of different sizes
  - Sensitive to node ordering

Idea: Convolutional Neural Networks

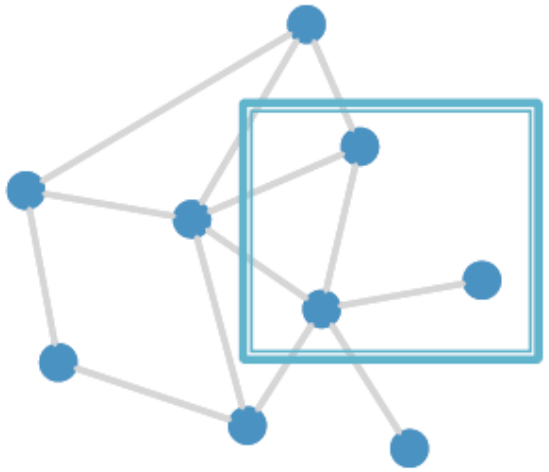
CNN on an image:



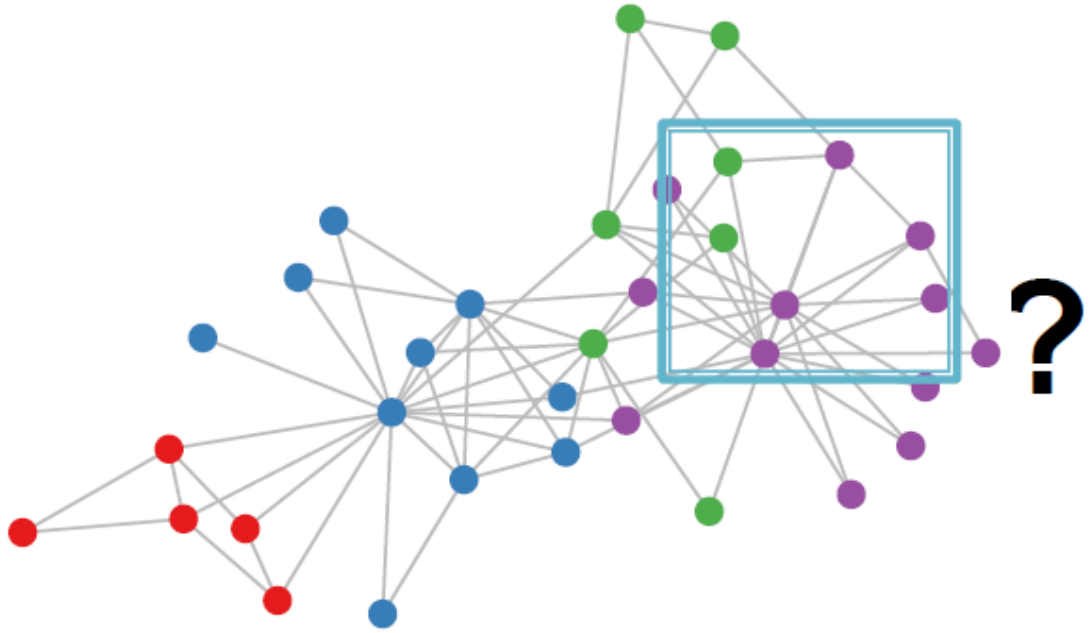
- Goal is to generalize convolutions beyond simple lattices
- Leverage node features/attributes (e.g., text, images)

Real World Graphs

- **But our graphs look like this:**



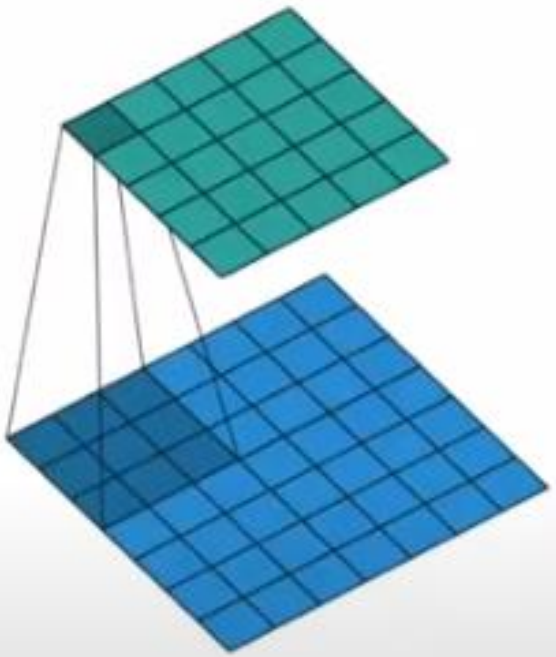
or this:



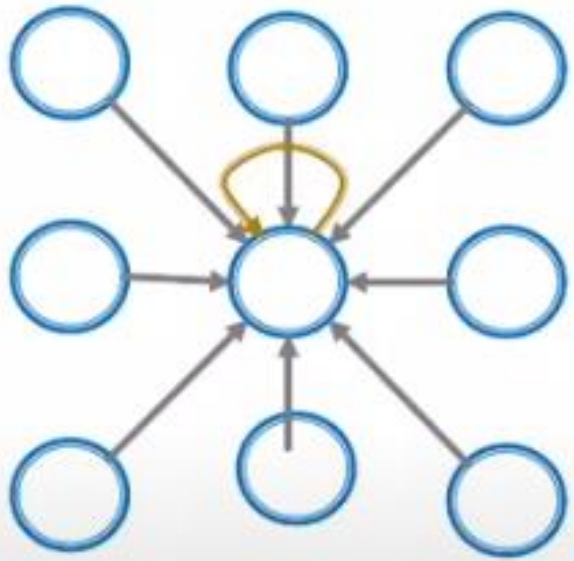
?

- There is no fixed notion of locality or sliding window on the graph
- Graph is permutation invariant

From Images to Graphs



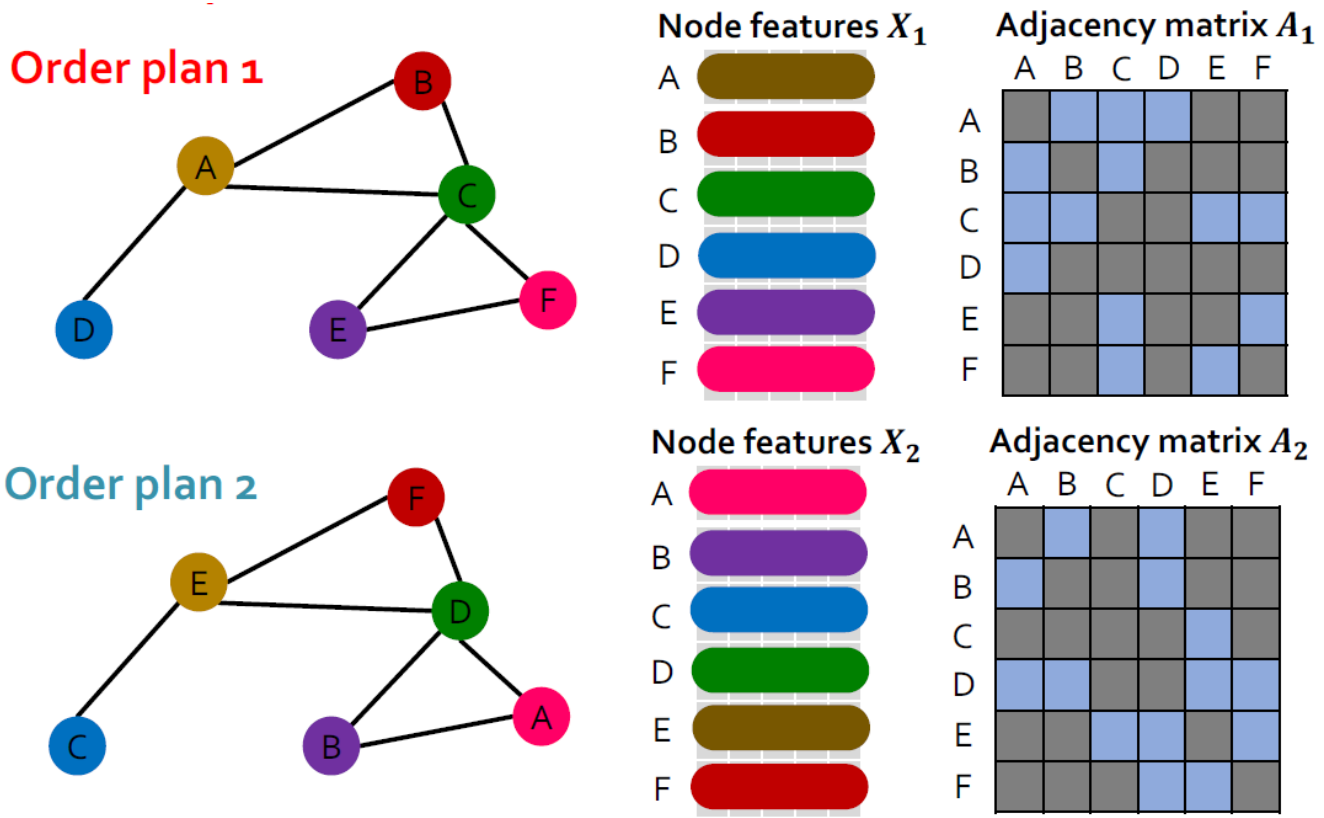
Image



Graph

Permutation Invariance

- Permute the input, the output stays the same (map a graph to a vector)
- **Graph does not have a canonical order of the nodes!**



Graph and node representations should be the same for both plans

Permutation Invariance

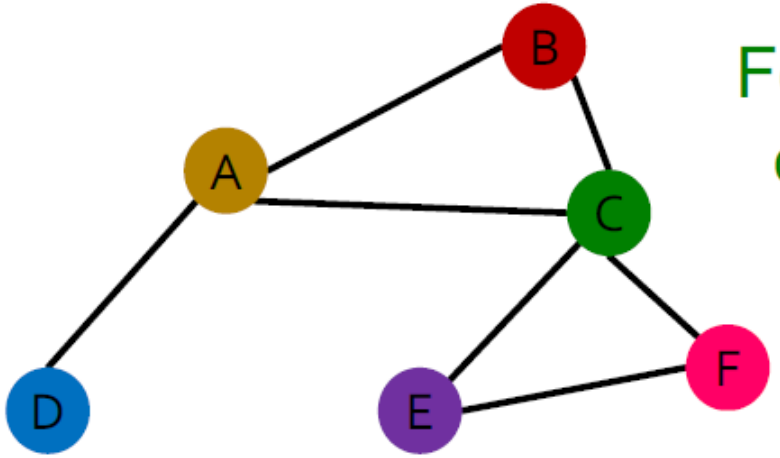
- **What does it mean by "graph representation is same for two order plans"?**
- Consider we learn a function f that maps a graph $G = (\boldsymbol{A}, \boldsymbol{X})$ to a vector $\mathbb{R}^d$:

$f(\boldsymbol{A}_1, \boldsymbol{X}_1) = f(\boldsymbol{A}_2, \boldsymbol{X}_2)$

$\boldsymbol{A}$ is the adjacency matrix

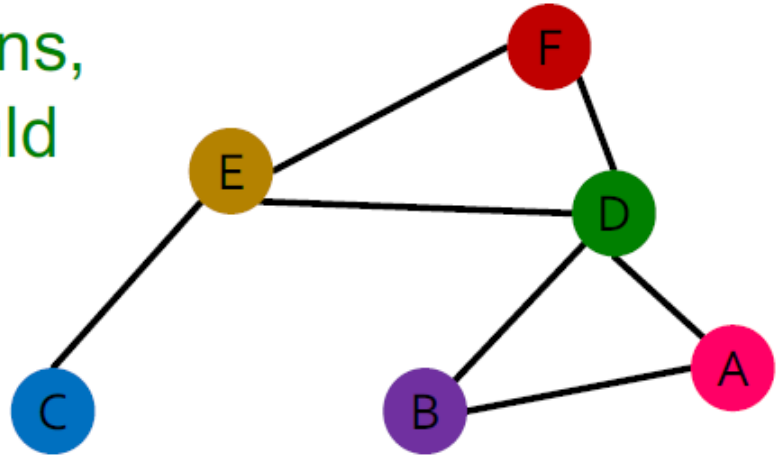
$\boldsymbol{X}$ is the node feature matrix

Order plan 1: A_1, X_1



For two order plans, output of f should be the same!

Order plan 2: A_2, X_2



Permutation Invariance

- Then, if $f(\boldsymbol{A}_i, \boldsymbol{X}_i) = f(\boldsymbol{A}_j, \boldsymbol{X}_j)$ for any order plan i and j, we formally say f is a **permutation invariant function**
  - For a graph with $|V|$ nodes, there are $|V|!$ different order plans
- **Definition:** For any graph function $f: \mathbb{R}^{|V| \times m} \times \mathbb{R}^{|V| \times |V|} \rightarrow \mathbb{R}^d$, f is **permutation-invariant** if

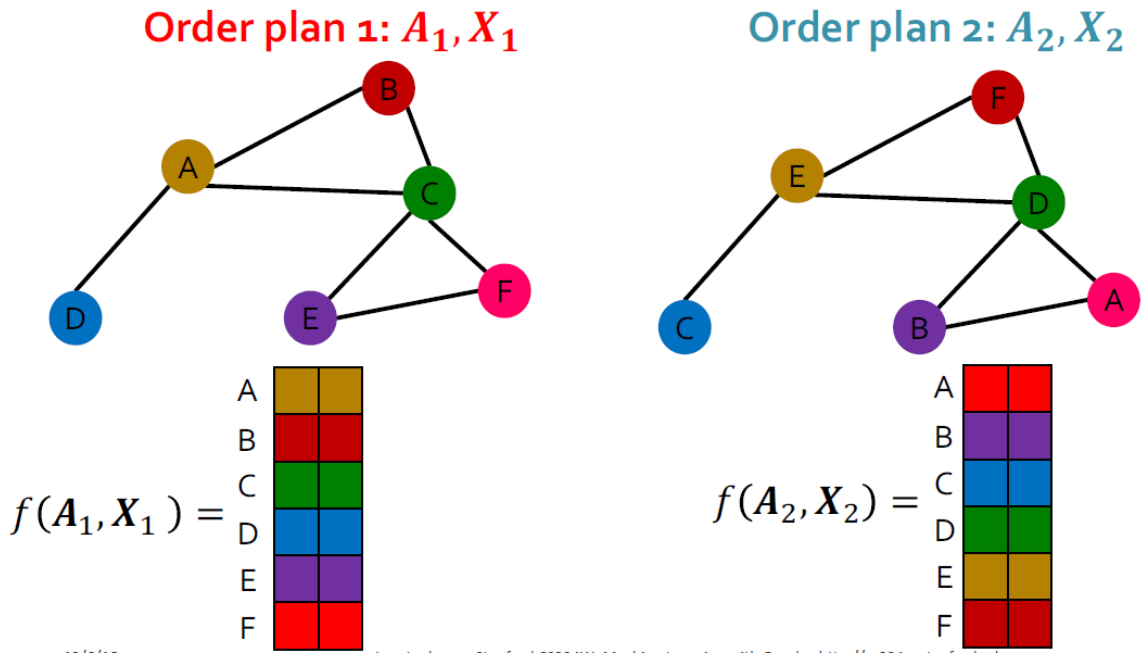
$$f(A, X) = f(PAP^{\mathsf{T}}, PX) \qquad \text{for any permutation } P$$

  - m… each node has a m-dim feature vector associated with it
  - d… output embedding dimensionality of embedding the graph $G = (A, X)$
  - $\boldsymbol{X} \in \mathbb{R}^{|V| \times m}$ $\qquad$ $\boldsymbol{A} \in \mathbb{R}^{|V| \times |V|}$

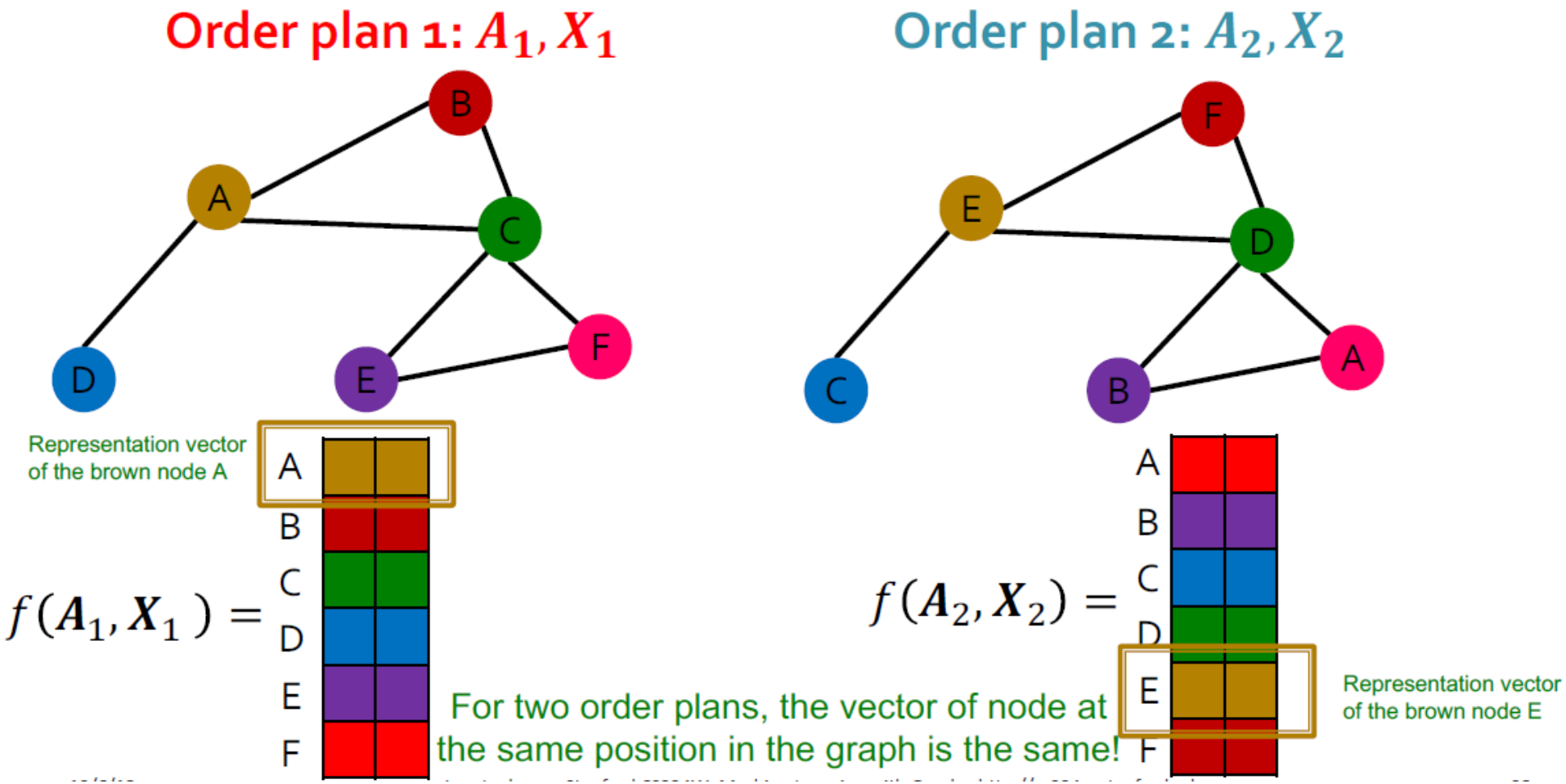
Permutation Equivariance

- Permute input, output also permutes accordingly (map a graph to a matrix)
- **For node representation:** We learn a function f that maps nodes of G to a matrix $\mathbb{R}^{|V| \times d}$

Each node in V is mapped to a d-dim embedding

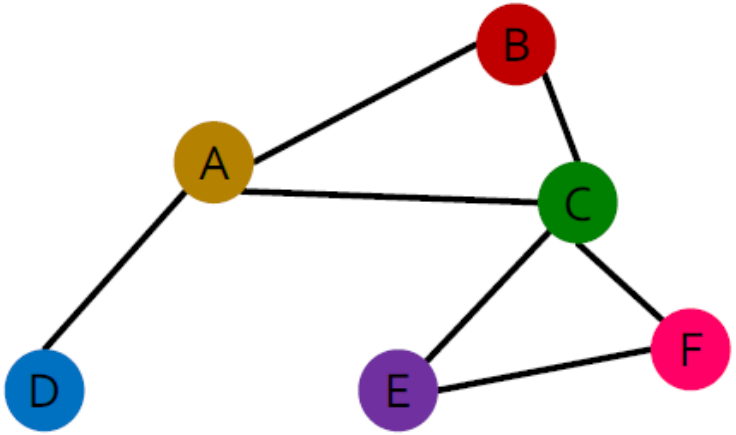


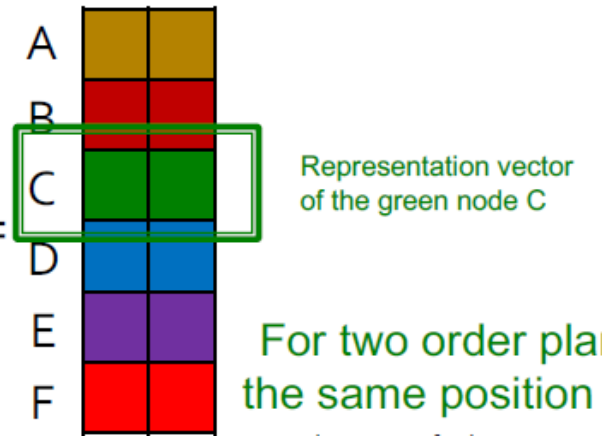
Permutation Equivariance



Permutation Equivariance

Order plan 1: A_1, X_1

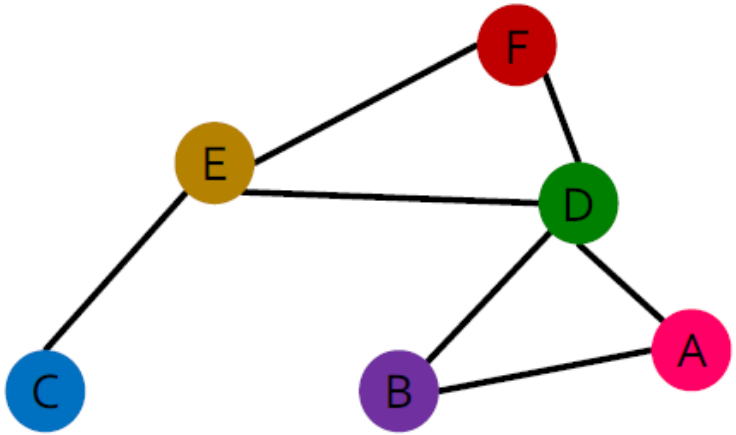


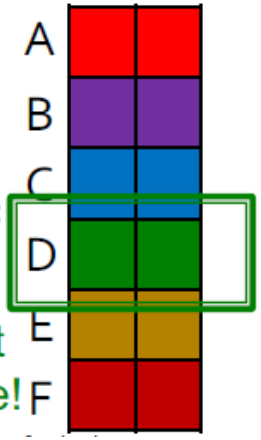


$f(A_1, X_1) =$

Representation vector of the green node C

Order plan 2: A_2, X_2





$f(A_2, X_2) =$

Representation vector of the green node D

For two order plans, the vector of node at the same position in the graph is the same!

Permutation Equivariance

For node representation:

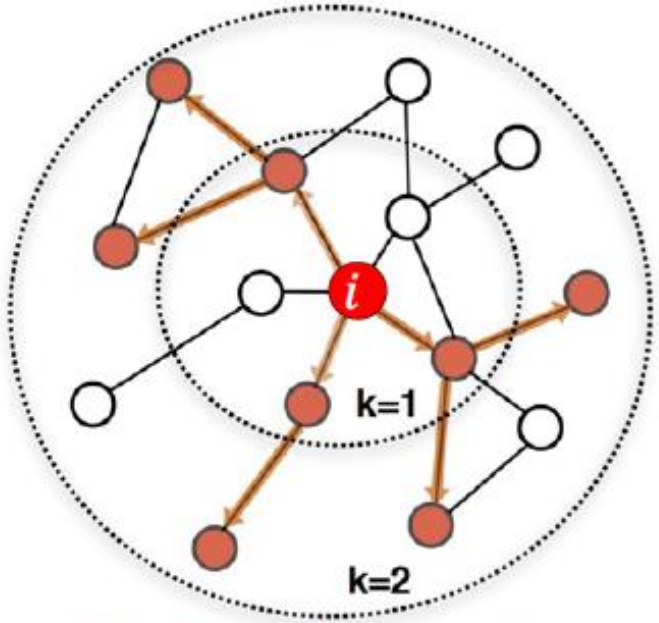
- Consider we learn a function f that maps graph $G = (\boldsymbol{A}, \boldsymbol{X})$ to a matrix $\mathbb{R}^{|V| \times d}$
- If output vector of a node at same position in graph remains unchanged for any order plan, we say f is **permutation equivariant**
- **Definition:** For any node function $f: \mathbb{R}^{|V| \times m} \times \mathbb{R}^{|V| \times |V|} \rightarrow \mathbb{R}^{|V| \times d}$, f is **permutation equivariant** if

$$P f(A, X) = f(PAP^{\mathsf{T}}, PX) \qquad \text{for any permutation } P$$

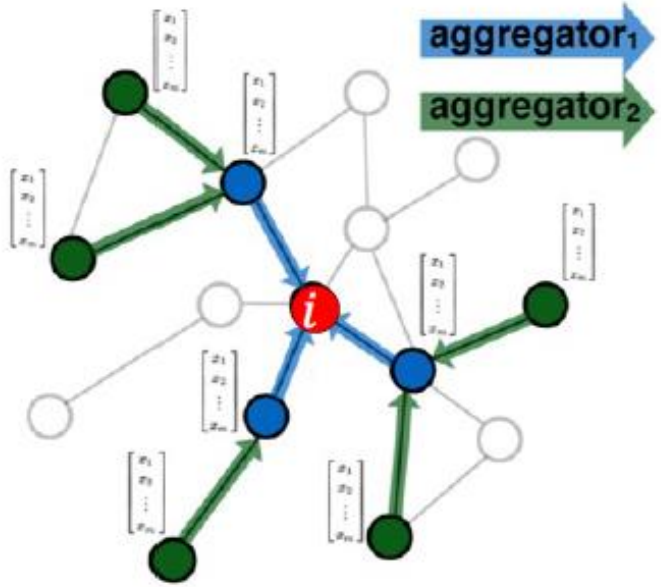
Graph Convolutional Networks

- Idea: Node's neighborhood defines a computation graph

What are computation graphs and why required in deep learning??



Determine node computation graph

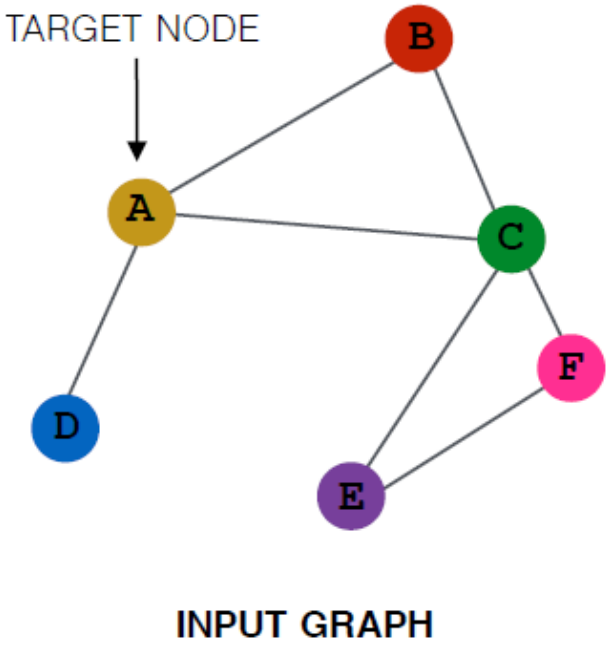


Propagate and transform information

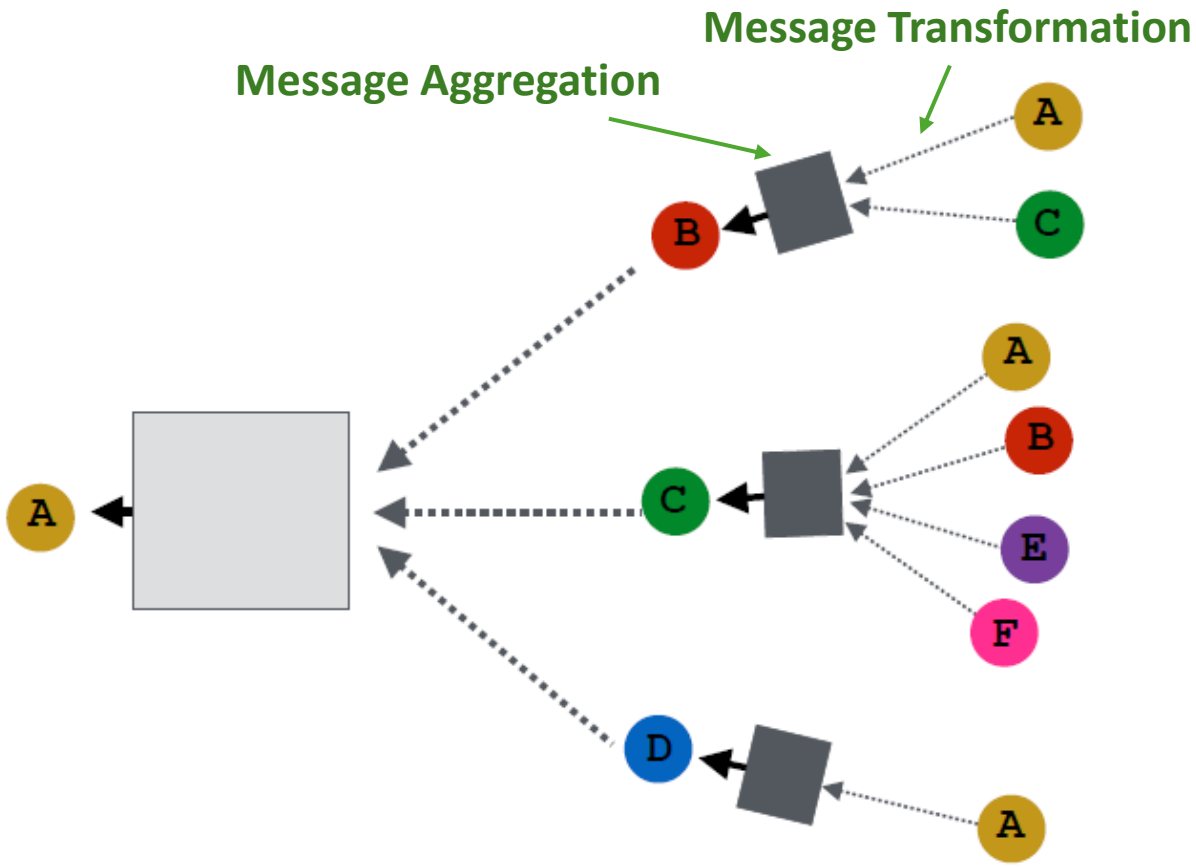
- **Learn how to propagate information across the graph to compute node features**

Idea: Aggregate Neighbours

- **Key idea:** Generate node embeddings based on **local network neighborhoods**

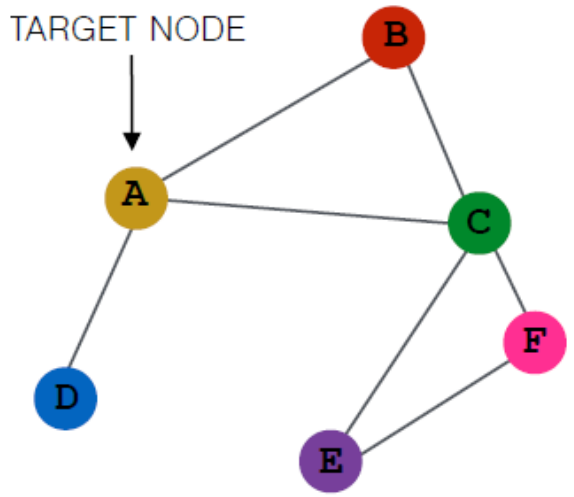


INPUT GRAPH

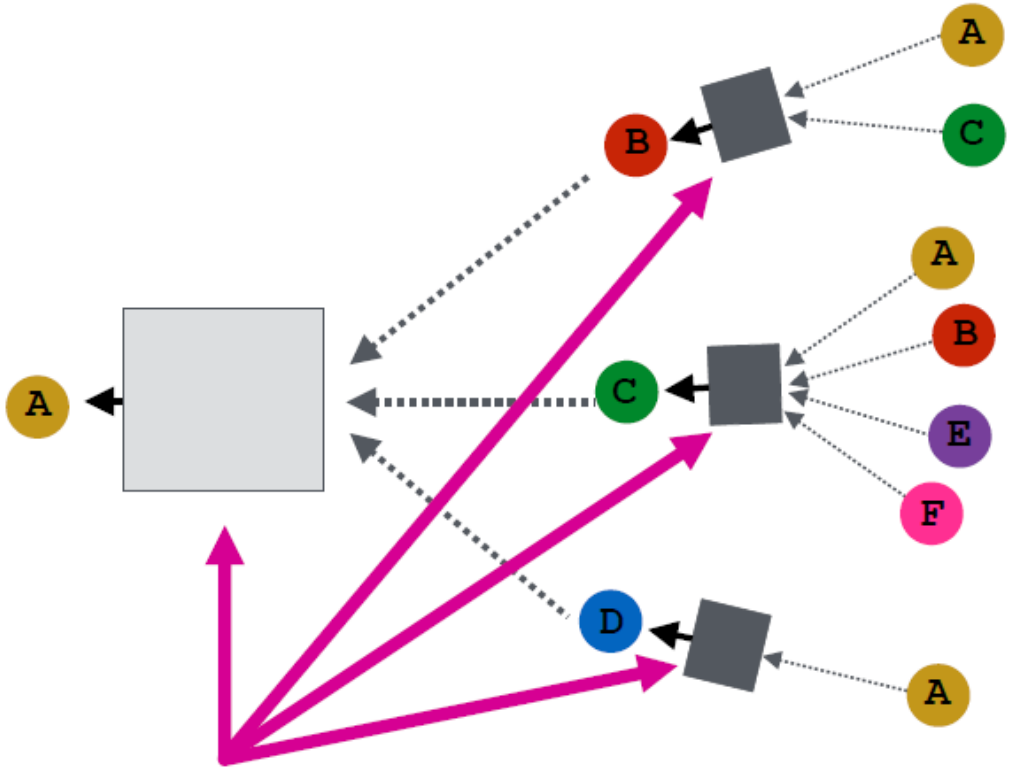


Idea: Aggregate Neighbours

- **Intuition:** Nodes aggregate information from their neighbors using neural networks



INPUT GRAPH



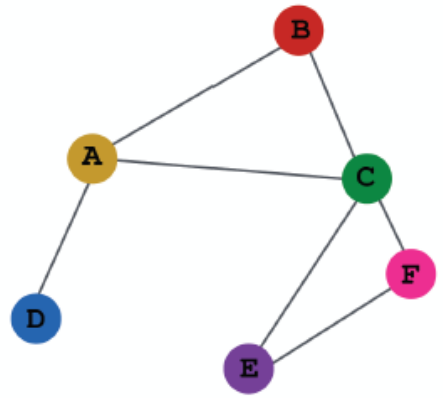
Neural networks

Aggregations and transformations will be learned

Idea: Aggregate Neighbours

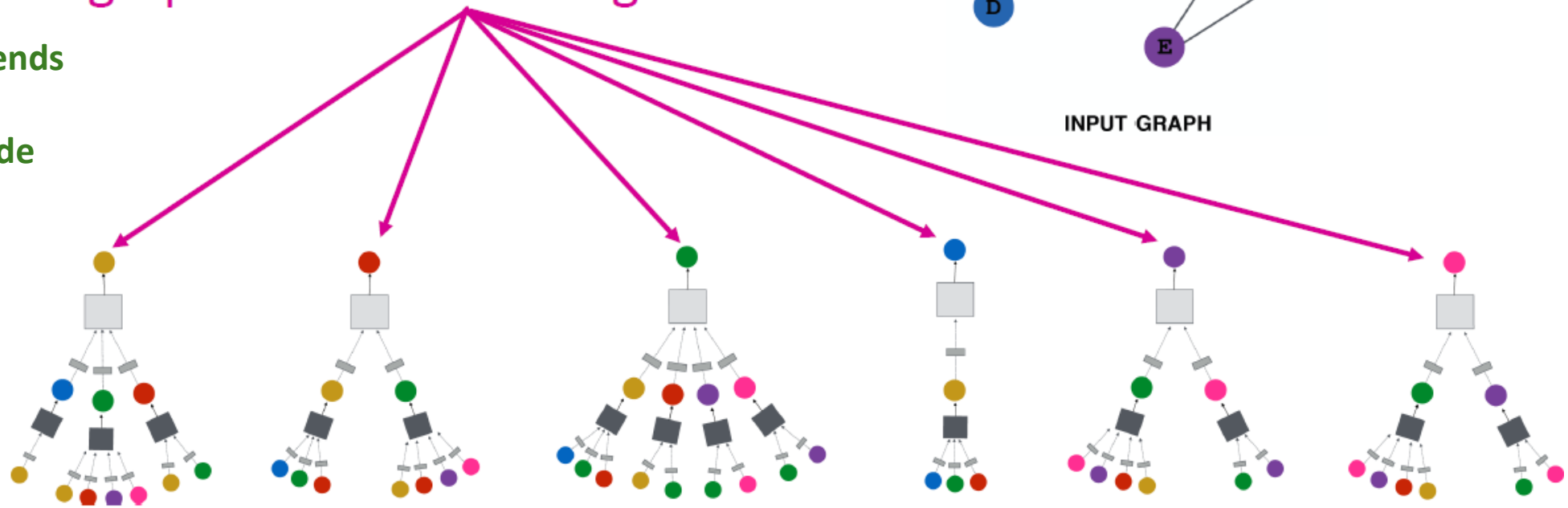
- **Intuition:** Network neighborhood defines a computation graph

Every node defines a computation graph based on its neighborhood!



INPUT GRAPH

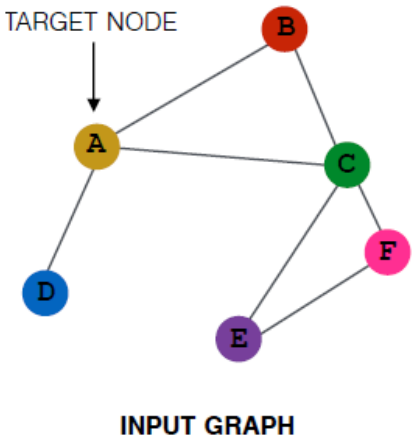
Structure of neural network of a node depends on the structure of the network around the node (its computation graph)

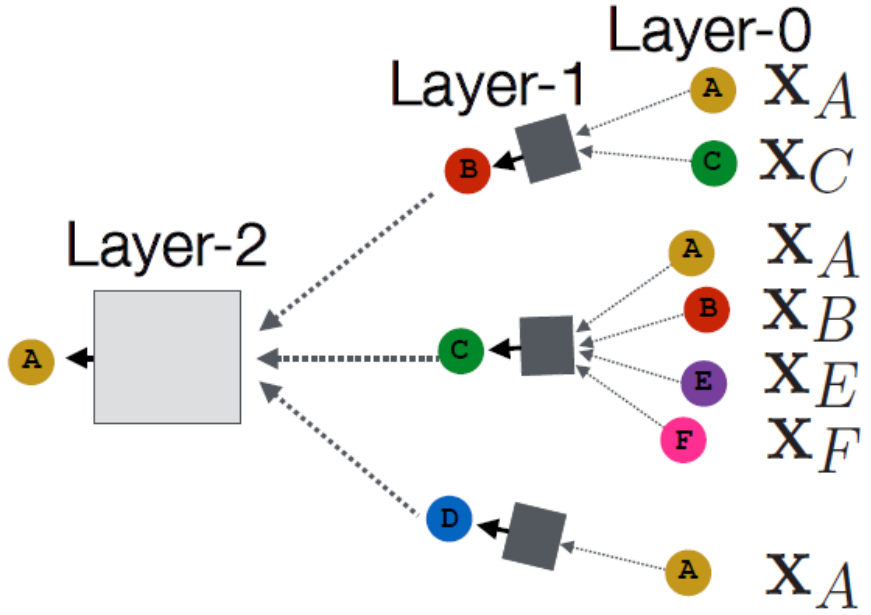


Deep Model: Many Layers

Model can be of arbitrary depth:

- Nodes have embeddings at each layer
- Layer-0 embedding of node v is its input feature, x_v
- Layer-k embedding gets information from nodes that are k hops away



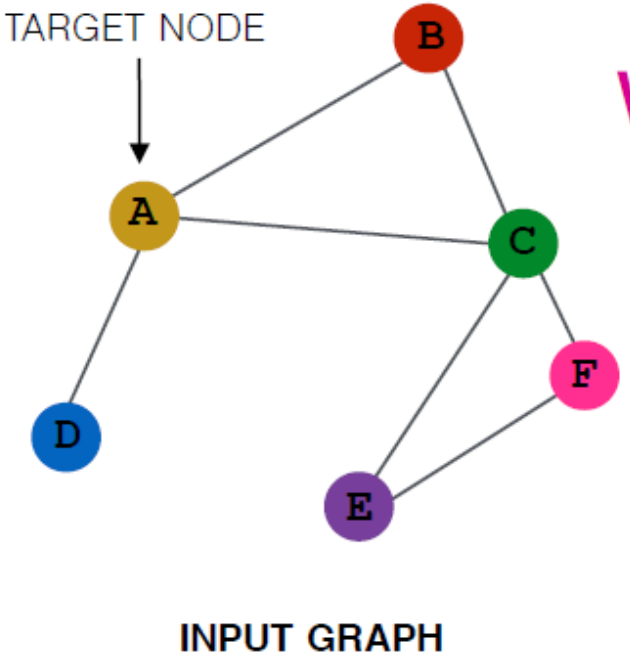


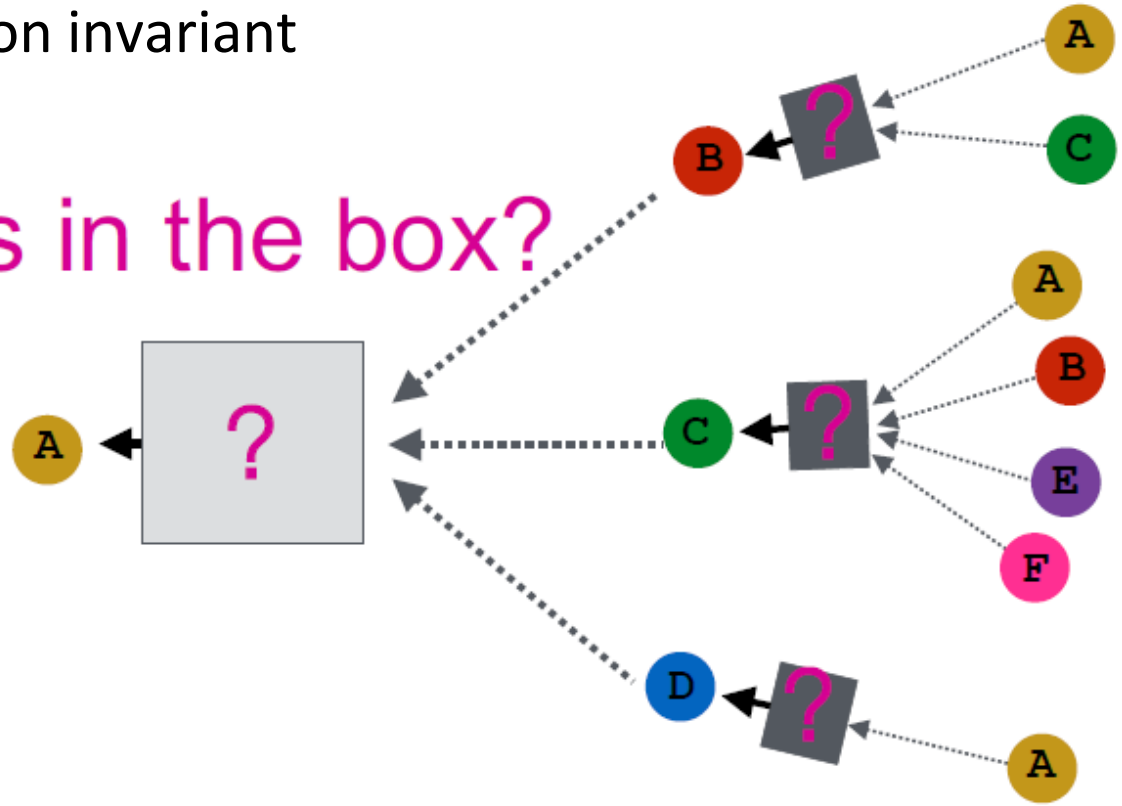
Each step corresponds to one layer of the neural network – corresponds to one hop in the underlying structure

Done only for fixed number of steps..not till convergence

Neighborhood Aggregation

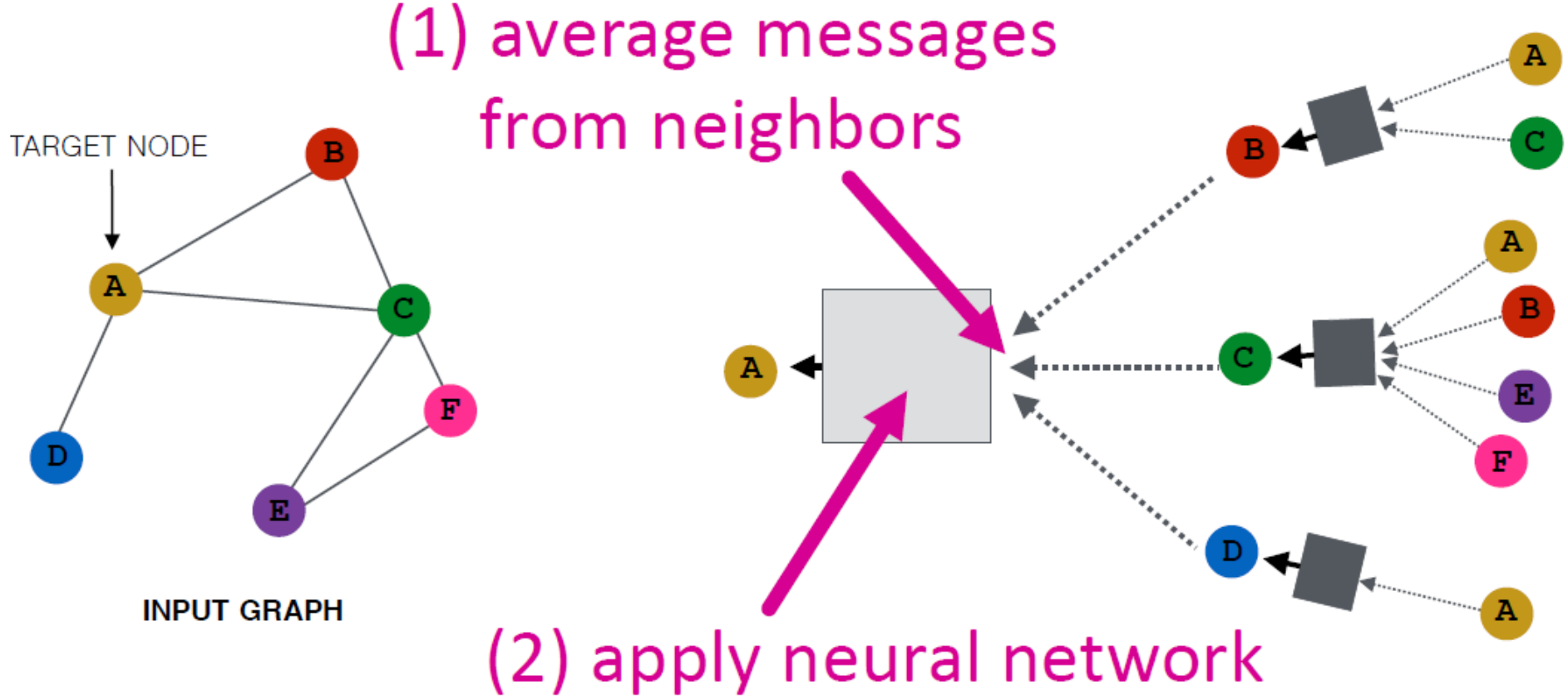
- **Neighborhood aggregation:** Key distinctions are in how different approaches aggregate information across the layers
  - Aggregation should be permutation invariant





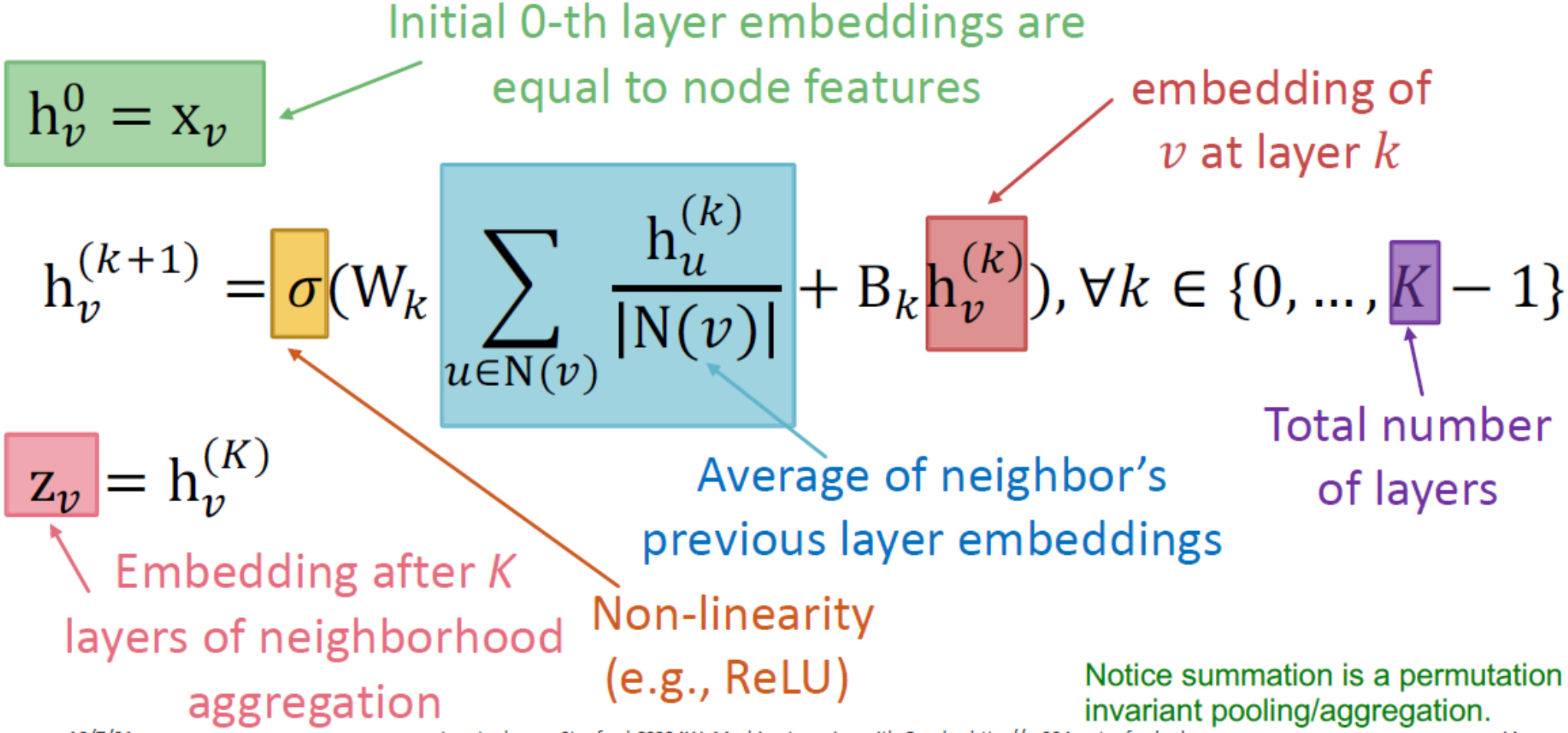
Neighborhood Aggregation

- **Basic approach:** Average information from neighbors and apply a neural network



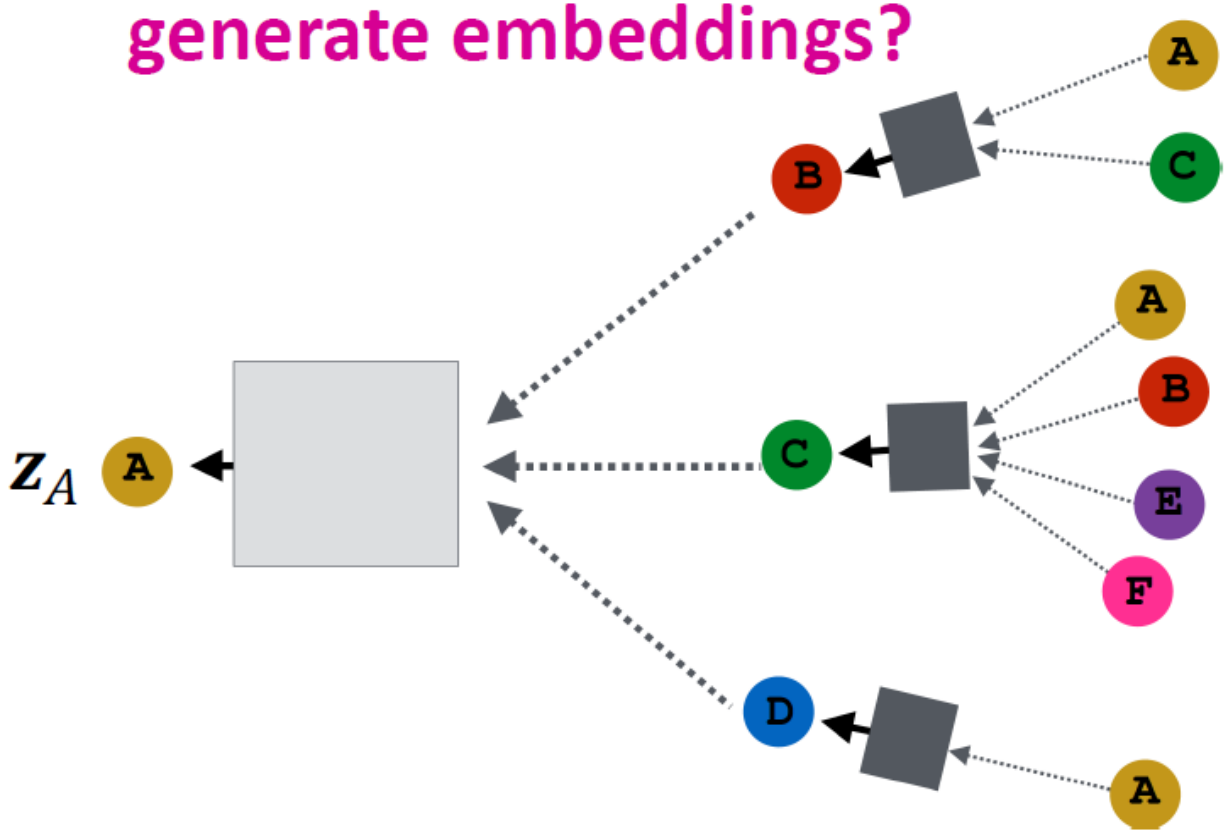
Deep Encoder

- **Basic approach:** Average neighbor messages and apply a neural network



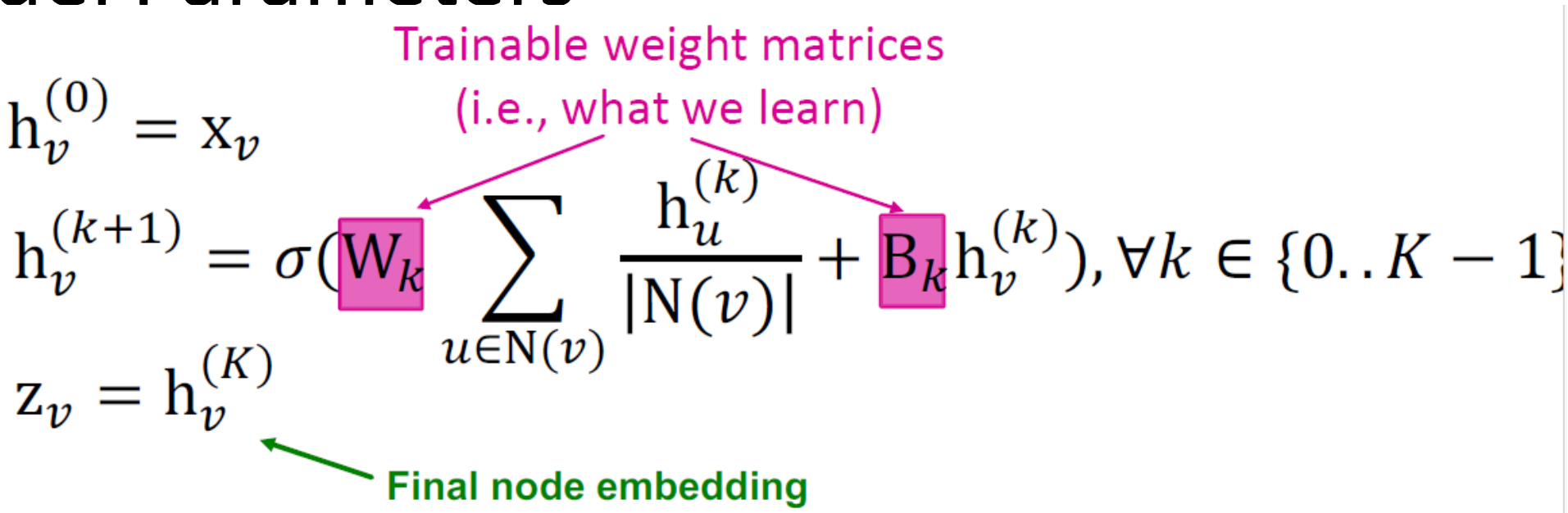
Training the Model

How do we train the GCN to generate embeddings?



Need to define a loss function on the embeddings.

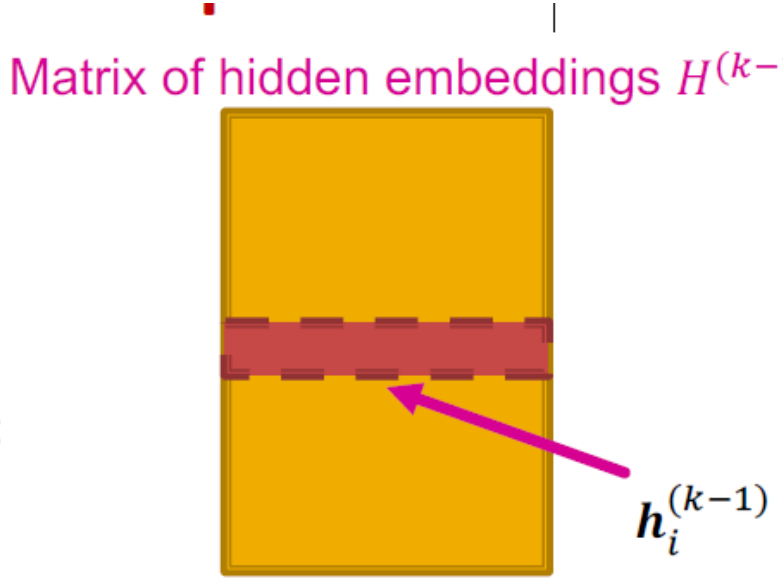
Model Parameters



- We can feed these **embeddings into any loss function** and run SGD to **train the weight parameters**
- h_v^k: the hidden representation of node v at layer k
  - W_k: weight matrix for neighborhood aggregation
  - B_k: weight matrix for transforming hidden vector of self

Matrix Formulation

- **Many aggregations can be performed efficiently by (sparse) matrix operations**
  - Let $H^{(k)} = [h_1^{(k)} \dots h_{|V|}^{(k)}]^{\mathrm{T}}$
  - Then: $\sum_{u \in N_v} h_u^{(k)} = \mathrm{A}_{v,:}\mathrm{H}^{(k)}$
  - Let D be diagonal matrix where $D_{v,v} = \mathrm{Deg}(v) = |N(v)|$
    - The inverse of D: D^{-1} is also diagonal: $D_{v,v}^{-1} = 1/|N(v)|$
  - **Therefore,**



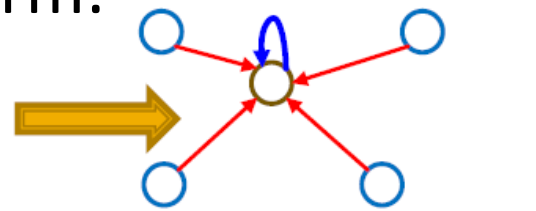
$$\sum_{u \in N(v)} \frac{h_u^{(k-1)}}{|N(v)|} \Rightarrow H^{(k+1)} = D^{-1}AH^{(k)}$$

Matrix Formulation

- Re-writing update function in matrix form:

$$H^{(k+1)} = \sigma(\tilde{A}H^{(k)}W_k^{\mathrm{T}} + H^{(k)}B_k^{\mathrm{T}})$$

where $\tilde{A} = D^{-1}A$



$$H^{(k)} = [h_1^{(k)} \dots h_{|V|}^{(k)}]^T$$

  - Red: neighborhood aggregation
  - Blue: self transformation

- In practice, this implies that efficient sparse matrix multiplication can be used ($\tilde{A}$ is sparse)
- **Note**: not all GNNs can be expressed in a simple matrix form, when aggregation function is complex

How to Train a GNN

- Node embedding $\boldsymbol{z}_v$ is a function of input graph
- **Supervised setting**: We want to minimize loss $\mathcal{L}$:

$$\min_{\Theta} \mathcal{L}(\boldsymbol{y}, f_{\Theta}(\boldsymbol{z}_v))$$

  - $\boldsymbol{y}$: node label
  - $\mathcal{L}$ could be L2 if $\boldsymbol{y}$ is real number, or cross entropy if $\boldsymbol{y}$ is categorical
- **Unsupervised setting:**
  - No node label available
  - **Use the graph structure as the supervision!**
    - Dot product of two nodes has to correspond to their similarity in the network
    - Node similarity can be based on Random Walks, Node proximity in graph, …

Unsupervised Training

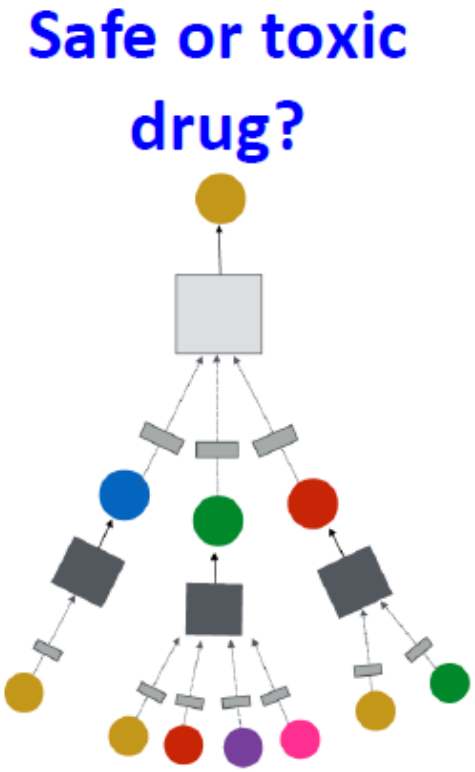
- **One possible idea: "Similar" nodes have similar embeddings:**

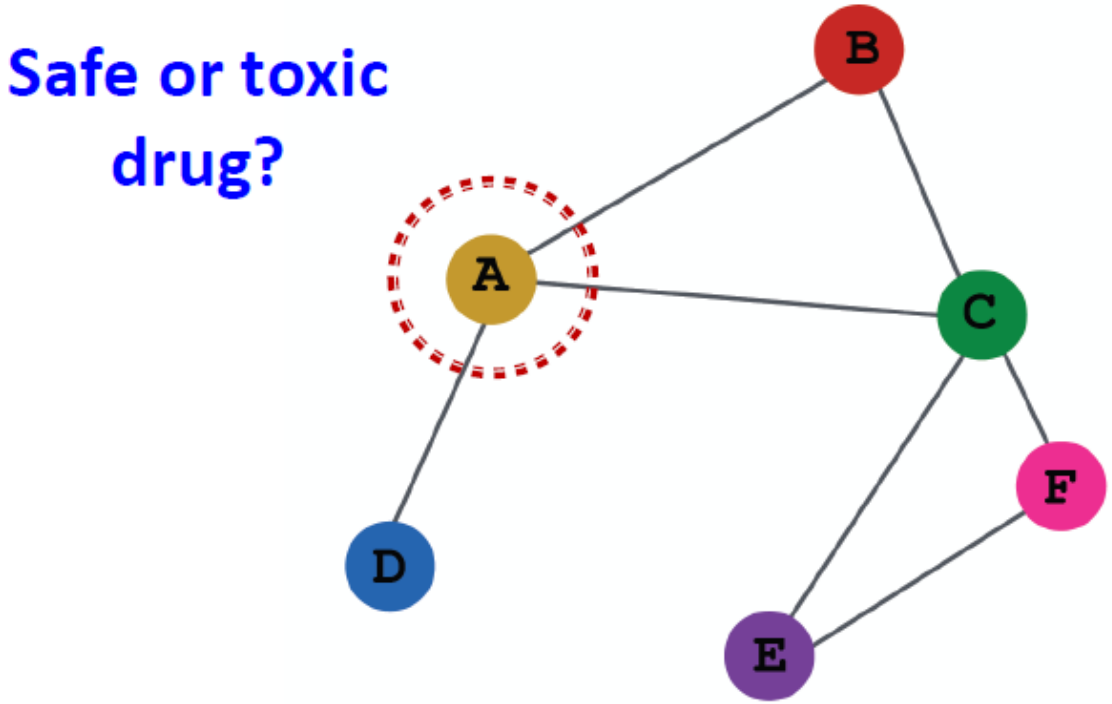
$$\min_{\Theta} \mathcal{L} = \sum_{z_u, z_v} \mathrm{CE}(y_{u,v}, \mathrm{DEC}(z_u, z_v))$$

  - where $y_{u,v} = 1$ when node u and v are **similar**
  - $z_u = f_\Theta(u)$ and $\mathrm{DEC}(\cdot,\cdot)$ is the dot product
- CE is the cross entropy loss:
  - $\mathrm{CE}\big(\boldsymbol{y}, f(\boldsymbol{x})\big) = -\sum_{i=1}^{C}(y_i \log f_\Theta(x)_i)$
    - y_i and $f_\Theta(x)_i$ are the **actual** and **predicted** values of the i-th class.
    - **Intuition:** the lower the loss, the closer the prediction is to one-hot

Supervised Training

- **Directly train** the model for a supervised task (e.g., **node classification**)

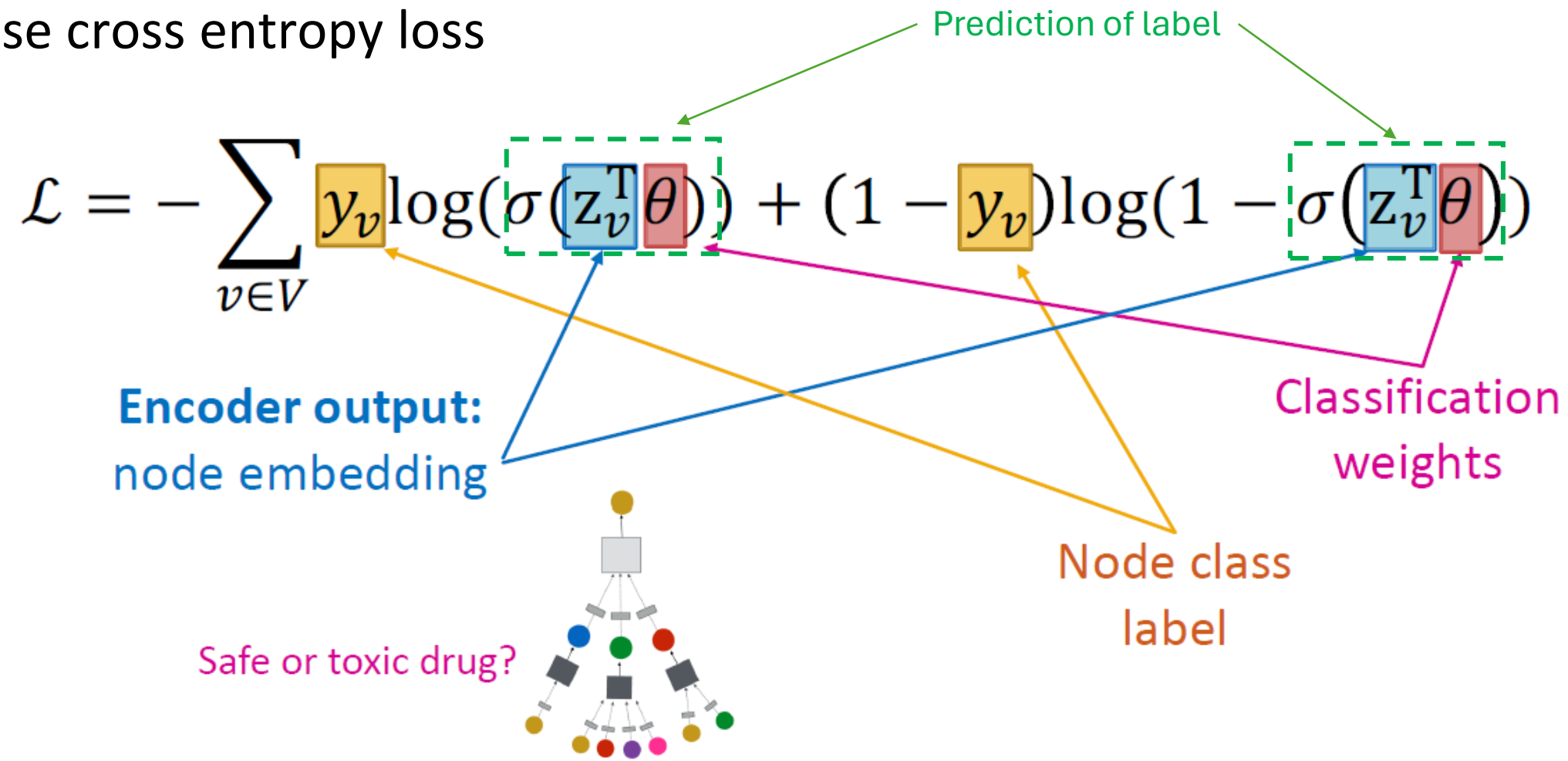




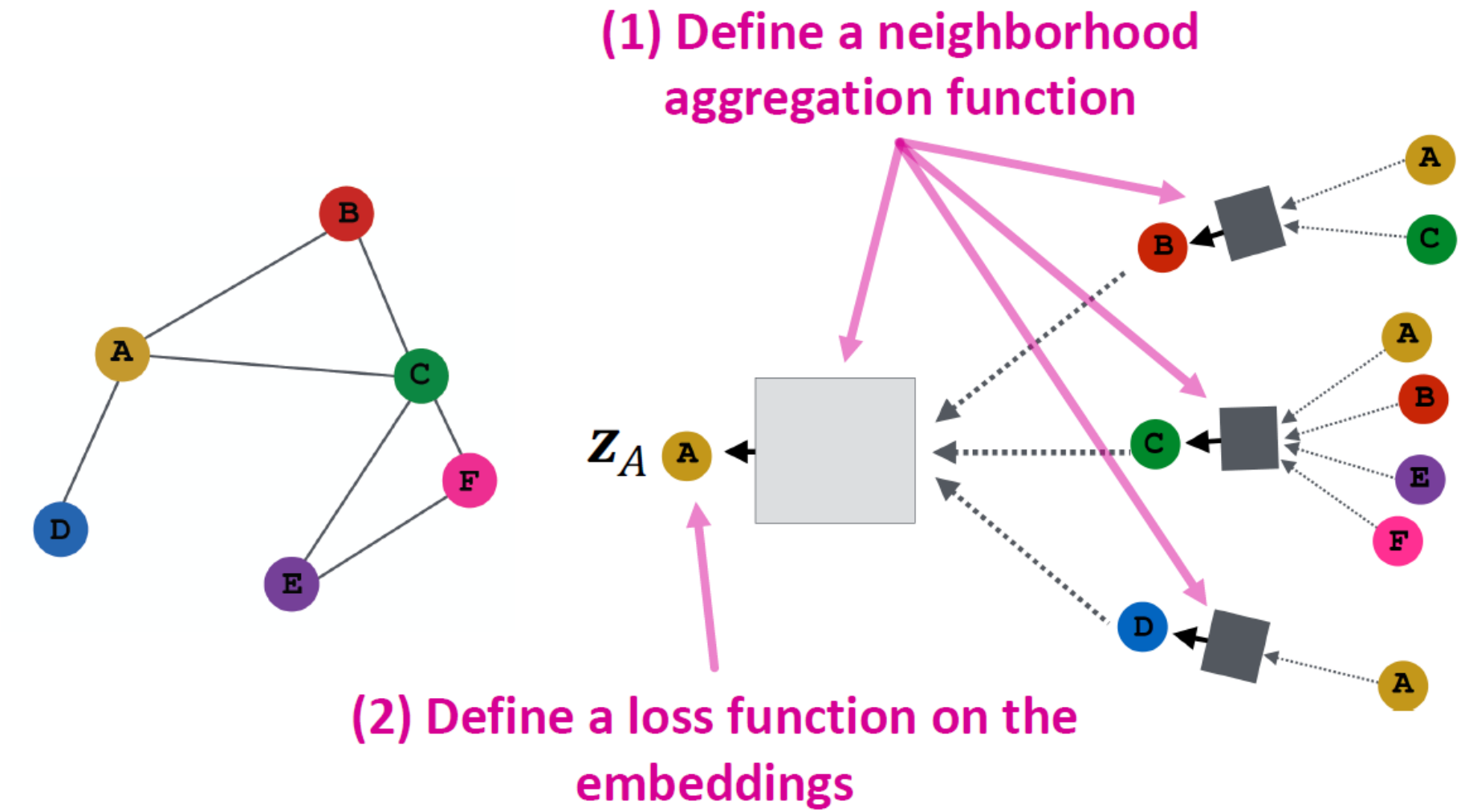
E.g., a drug-drug interaction network

Supervised Training

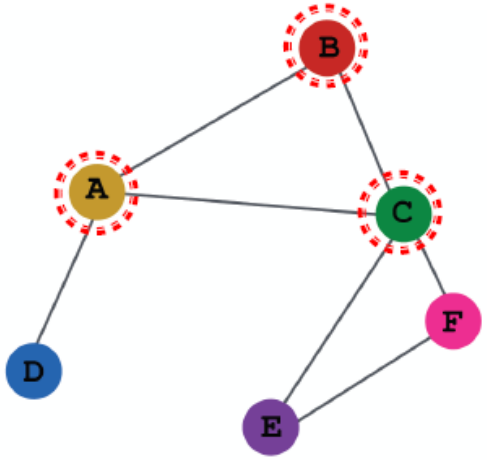
- Use cross entropy loss



Model Design: Overview

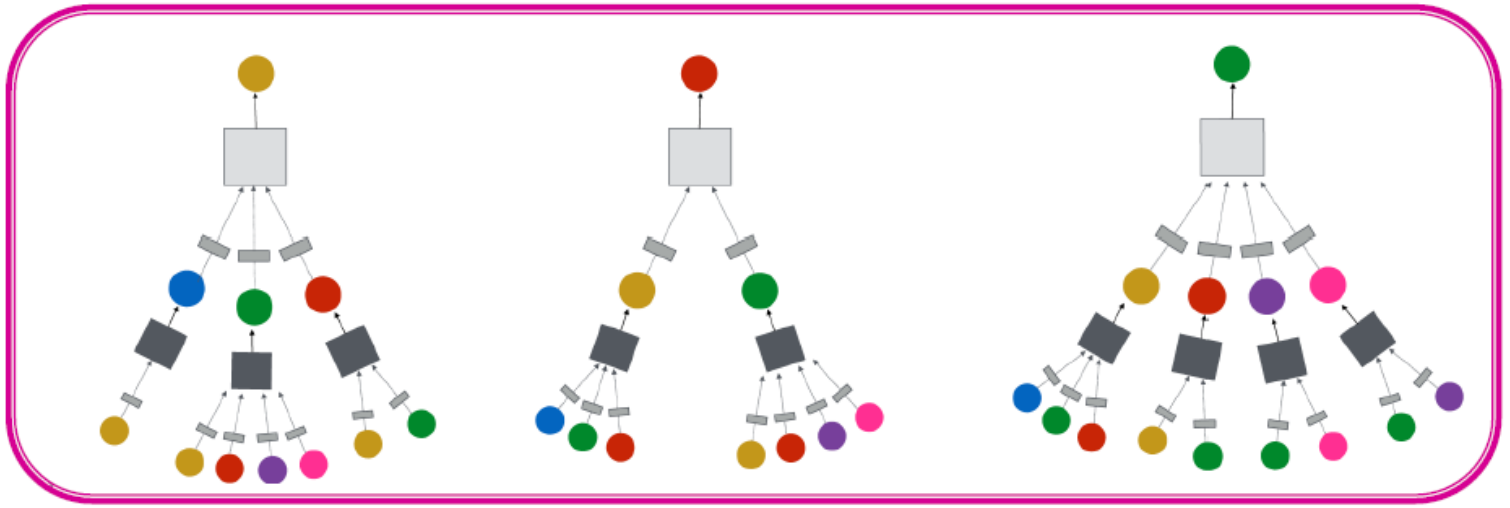


Model Design: Overview

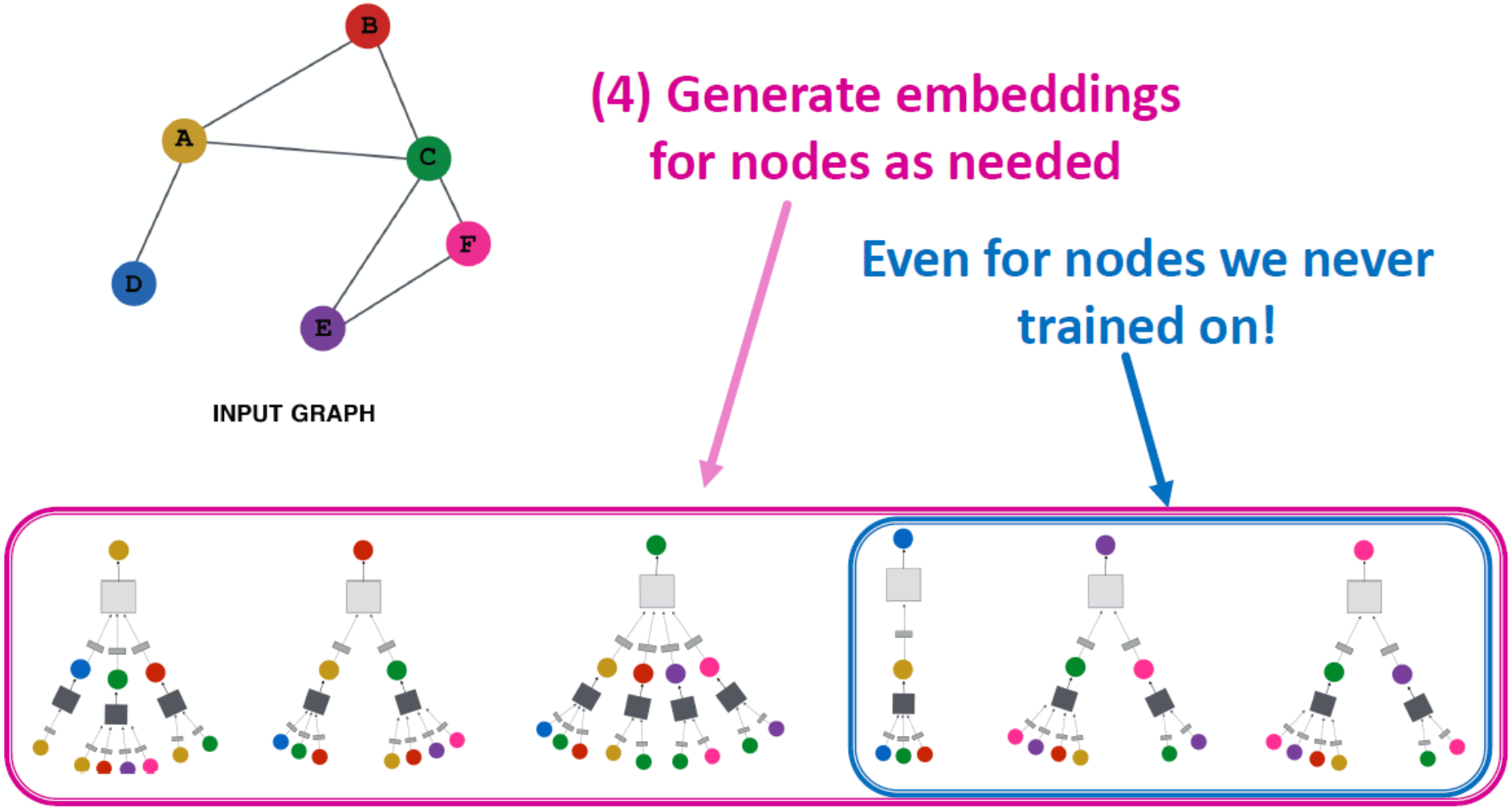


INPUT GRAPH

(3) Train on a set of nodes, i.e., a batch of compute graphs

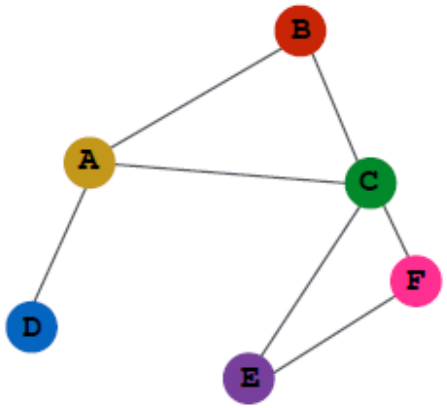


Model Design: Overview

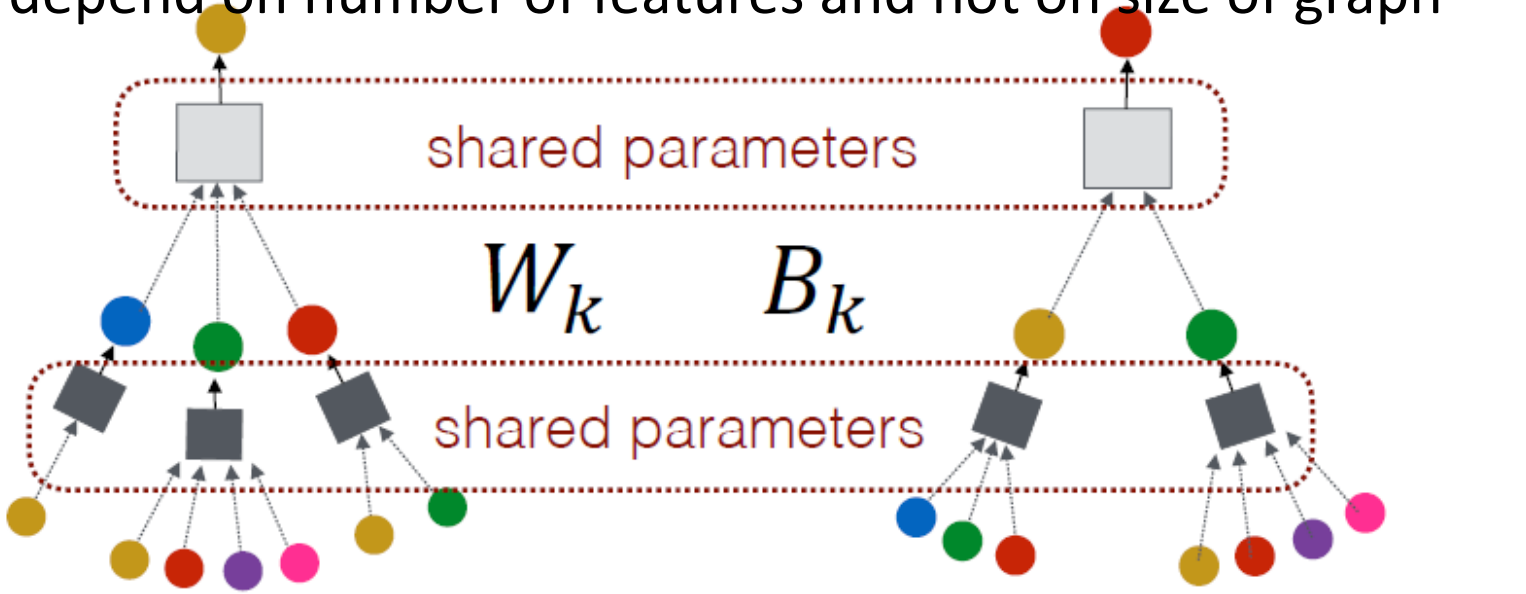


Inductive Capability

- **Same aggregation parameters are shared for all nodes:**
  - Number of model parameters is sublinear in $|V|$ and can **generalize to unseen nodes**!
  - ***W*** and ***B*** parameters depend on number of features and not on size of graph



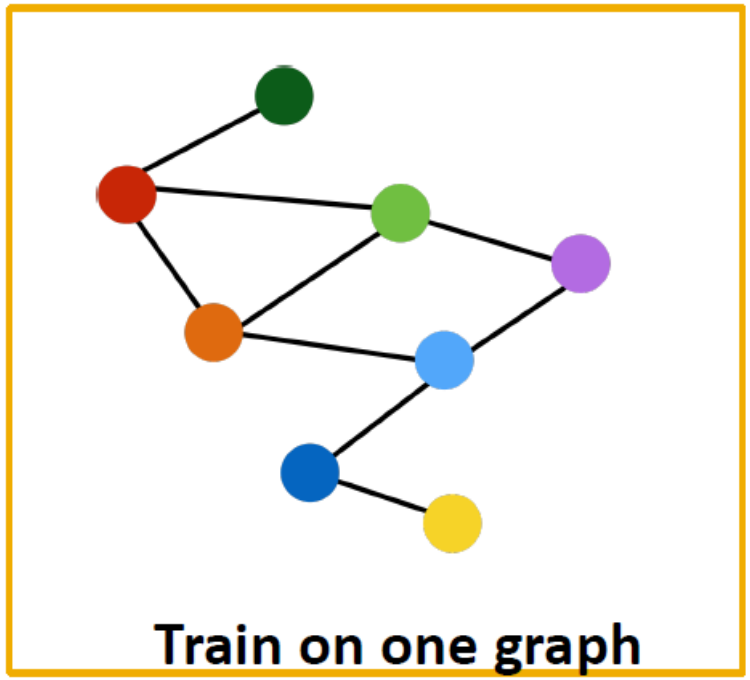
INPUT GRAPH

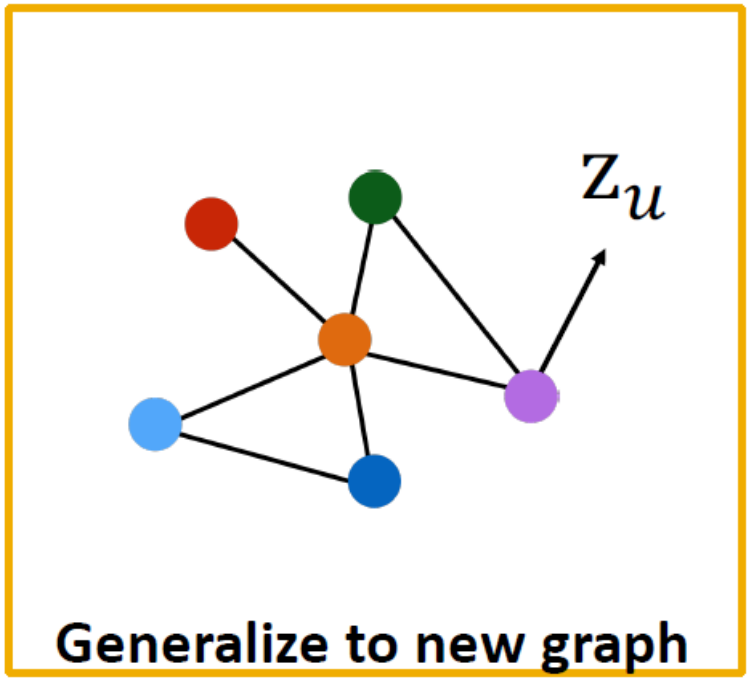


Compute graph for node A

Compute graph for node B

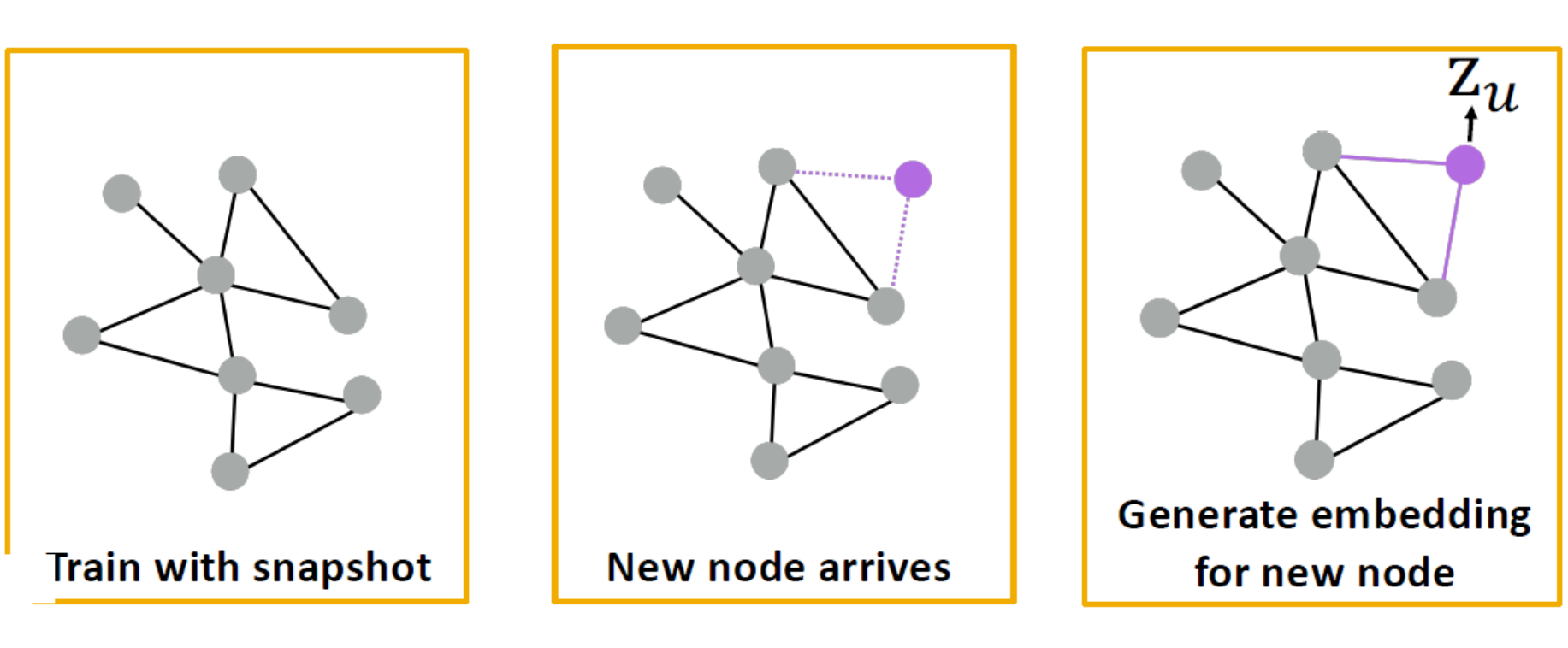
Inductive Capability → New Graphs





- Inductive node embedding Generalize to entirely unseen graphs
- E.g., train on protein interaction graph from model organism A and generate embeddings on newly collected data about organism B

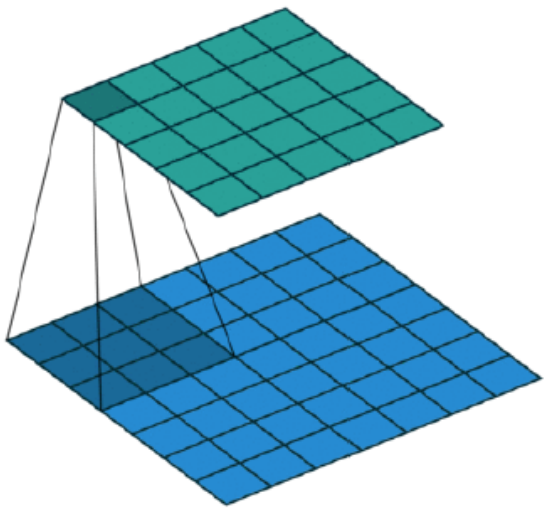
Inductive Capability → New Nodes

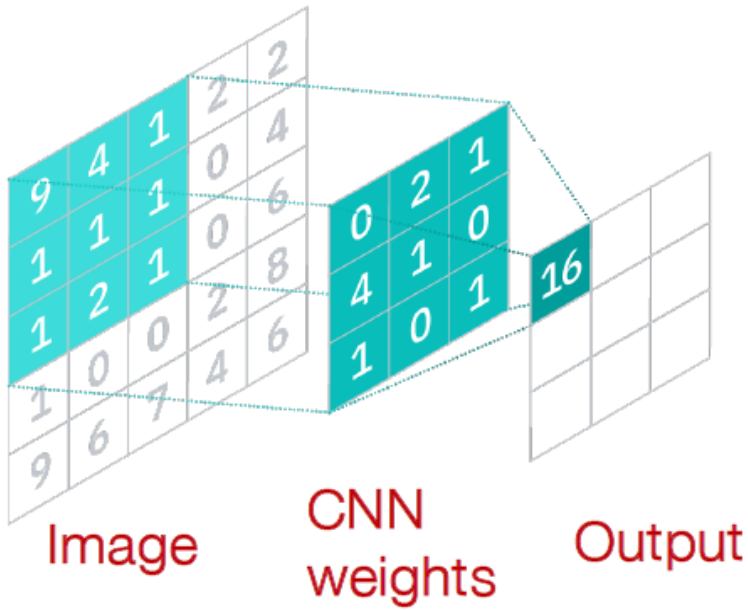


- Many application settings constantly encounter previously unseen nodes:
  - E.g., Reddit, YouTube, Google Scholar
- Need to generate new embeddings "on the fly"

CNN

Convolutional neural network (CNN) layer with 3x3 filter:



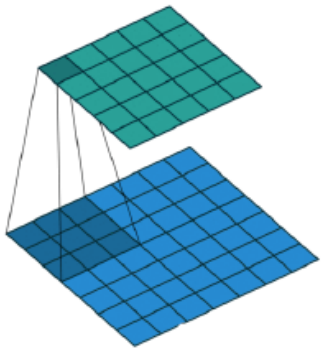


CNN formulation: $\mathrm{h}_v^{(l+1)} = \sigma(\sum_{u \in \mathrm{N}(v) \cup \{v\}} \mathrm{W}_l^u \mathrm{h}_u^{(l)}), \quad \forall l \in \{0, \ldots, L-1\}$

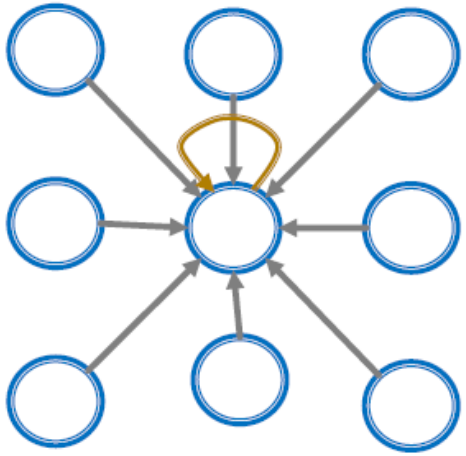
$N(v)$ represents the 8 neighbor pixels of v.

GNN vs. CNN

Convolutional neural network (CNN) layer with 3x3 filter:



Image



Graph

- GNN formulation: $\mathrm{h}_v^{(l+1)} = \sigma(\mathbf{W}_l \sum_{u \in \mathrm{N}(v)} \frac{\mathrm{h}_u^{(l)}}{|\mathrm{N}(v)|} + \mathrm{B}_l \mathrm{h}_v^{(l)}), \forall l \in \{0, \ldots, L-1\}$
- CNN formulation: (previous slide) $\mathrm{h}_v^{(l+1)} = \sigma(\sum_{u \in \mathrm{N}(v) \cup \{v\}} \mathrm{W}_l^u \mathrm{h}_u^{(l)}), \forall l \in \{0, \ldots, L-1\}$

  if we rewrite: $\mathrm{h}_v^{(l+1)} = \sigma(\sum_{u \in \mathrm{N}(v)} \mathbf{W}_l^u \mathrm{h}_u^{(l)} + \mathrm{B}_l \mathrm{h}_v^{(l)}), \forall l \in \{0, \ldots, L-1\}$

GNN vs. CNN

GNN formulation: $\mathrm{h}_v^{(l+1)} = \sigma(\mathbf{W}_l \sum_{u \in \mathrm{N}(v)} \frac{\mathrm{h}_u^{(l)}}{|\mathrm{N}(v)|} + \mathrm{B}_l \mathrm{h}_v^{(l)}), \forall l \in \{0, \ldots, L-1\}$

CNN formulation: $\mathrm{h}_v^{(l+1)} = \sigma(\sum_{u \in \mathrm{N}(v)} \mathbf{W}_l^u \mathrm{h}_u^{(l)} + \mathrm{B}_l \mathrm{h}_v^{(l)}), \forall l \in \{0, \ldots, L-1\}$

Key difference: We can learn different W_l^u for different "neighbor" u for pixel v on the image. The reason is we can pick an order for the 9 neighbors using **relative position** to the center pixel: {(-1,-1). (-1,0), (-1, 1), …, (1, 1)}

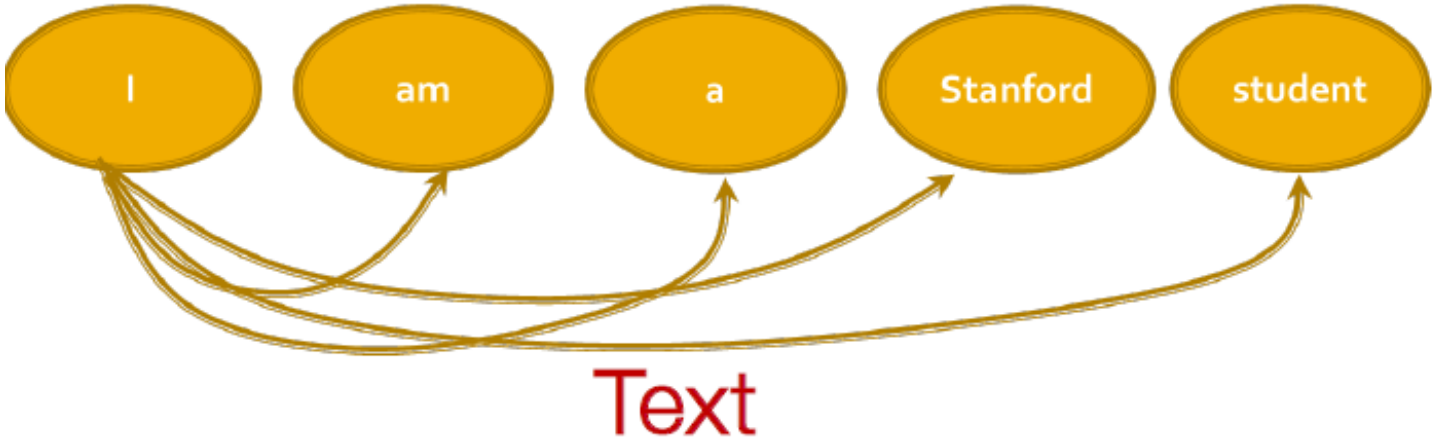
GNN vs. CNN

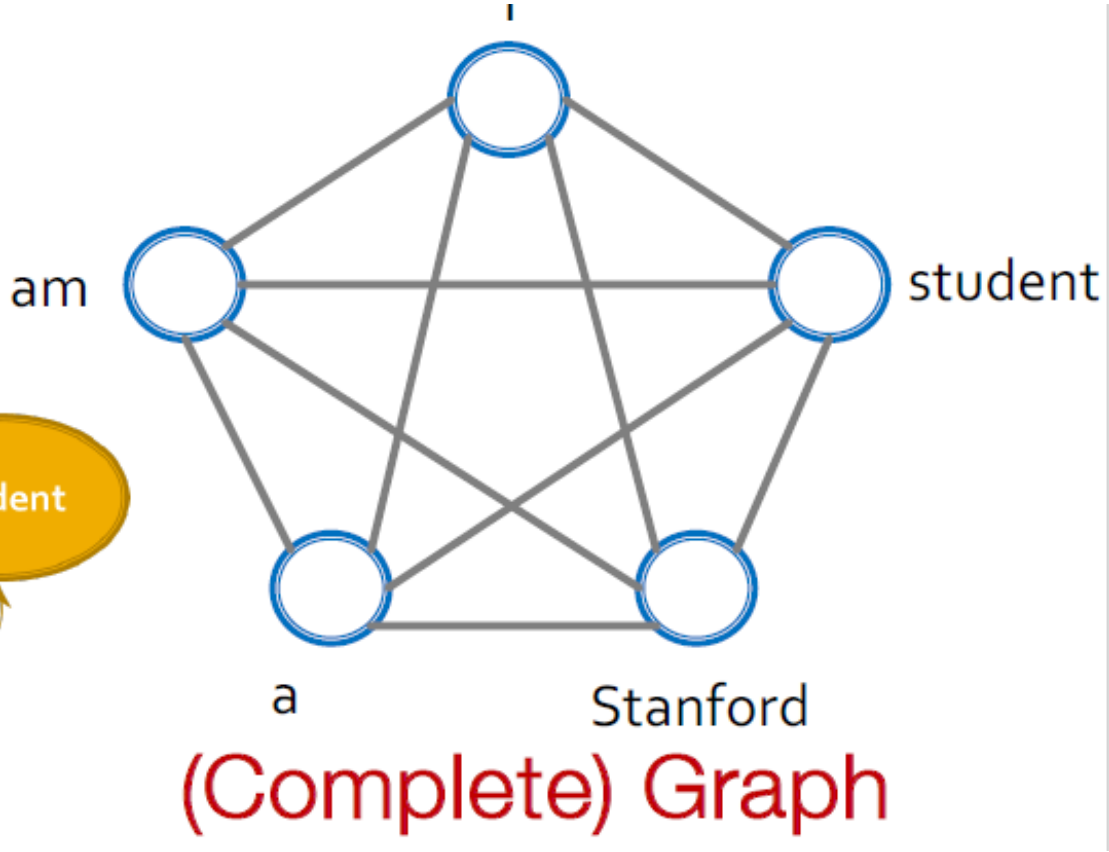
- CNN can be seen as a special GNN with fixed neighbor size and ordering:
  - The size of the filter is pre-defined for a CNN
  - The advantage of GNN is it processes arbitrary graphs with different degrees for each node
- CNN is not permutation invariant/equivariant
  - Switching the order of pixels leads to different outputs

GNN vs. Transformer

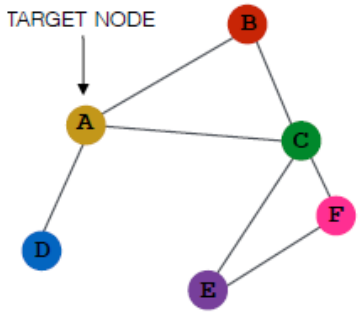
- Transformer layer can be seen as a special GNN that runs on a fully connected "word" graph!

Since each word attends to **all the other words**, **the computation graph** of a transformer layer is identical to that of a GNN on the **fully-connected "word" graph**.



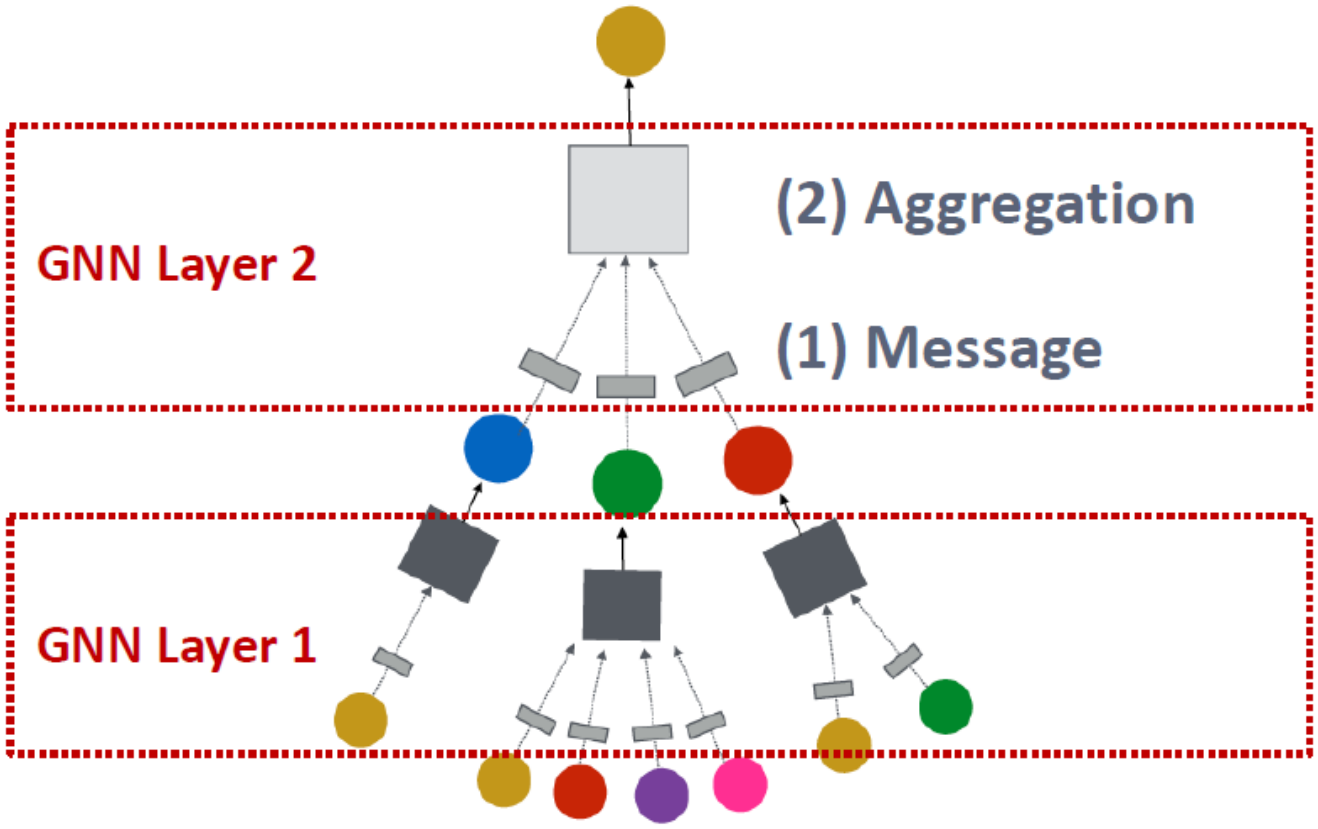


A General GNN Framework



INPUT GRAPH

(5) Learning objective



How to compute message and aggregate?

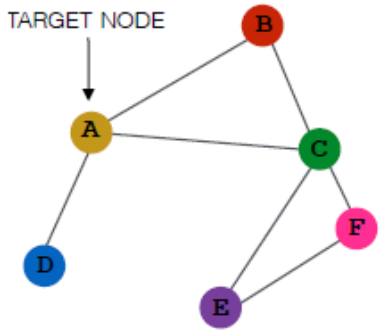
(3) Layer connectivity

Stack layers sequentially?? Add skip connections??

Stack layers sequentially?? Add skip connections??

(4) Graph augmentation

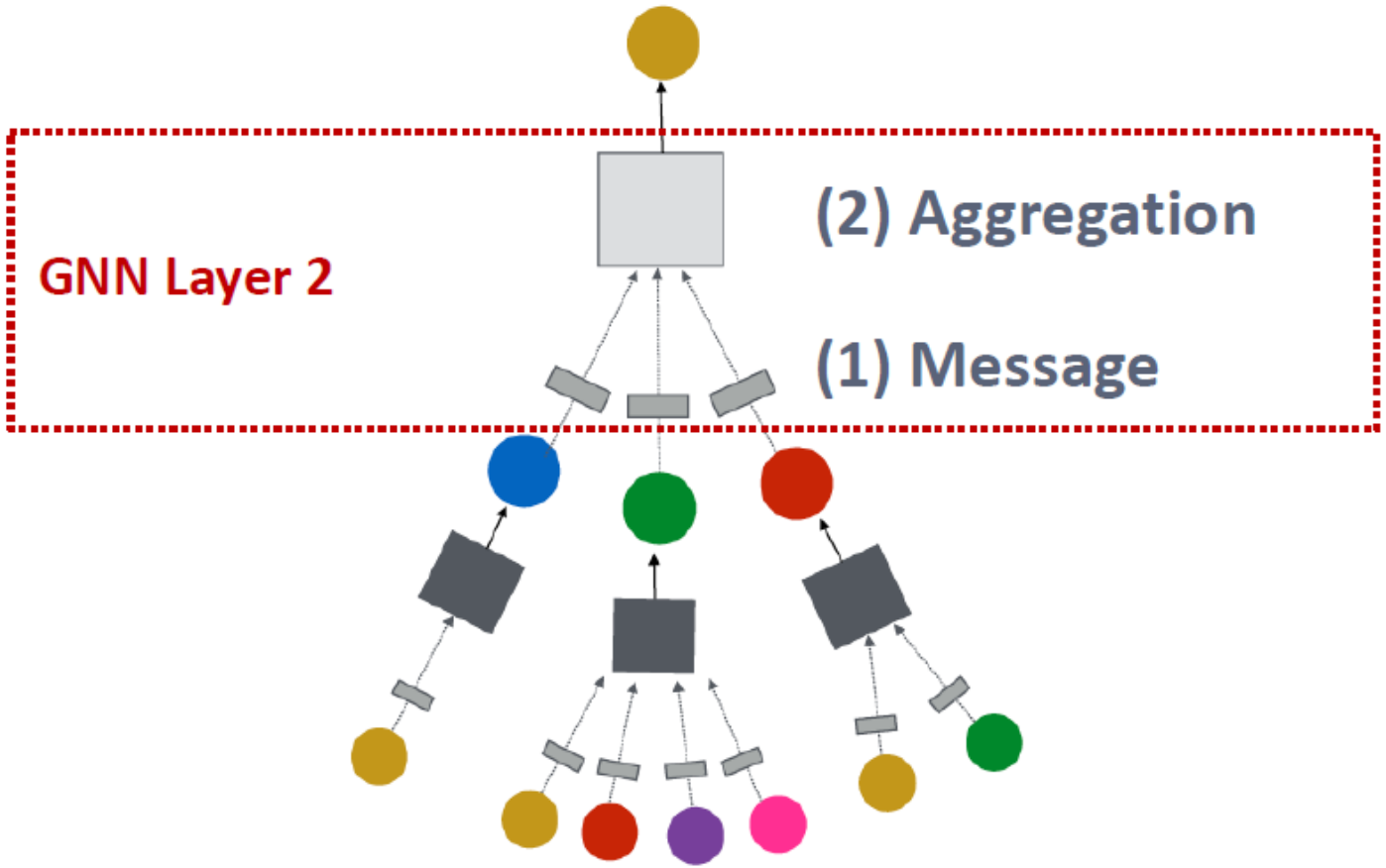
A GNN Layer



INPUT GRAPH

GNN Layer = Message + Aggregation

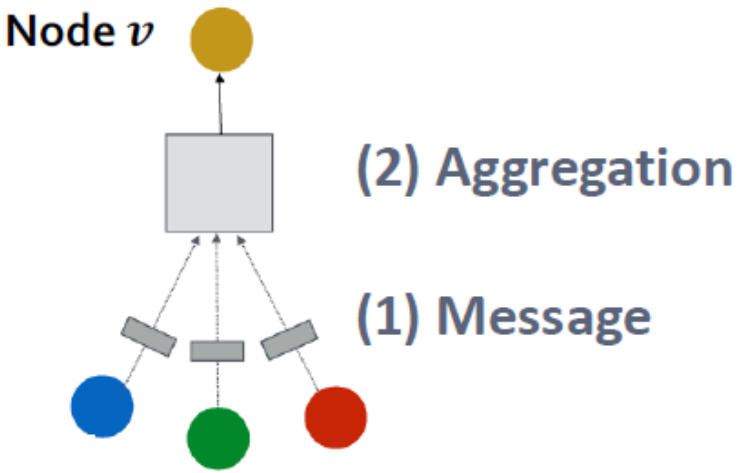
- **Different instantiations under this perspective**
- **GCN, GraphSAGE, GAT, …**

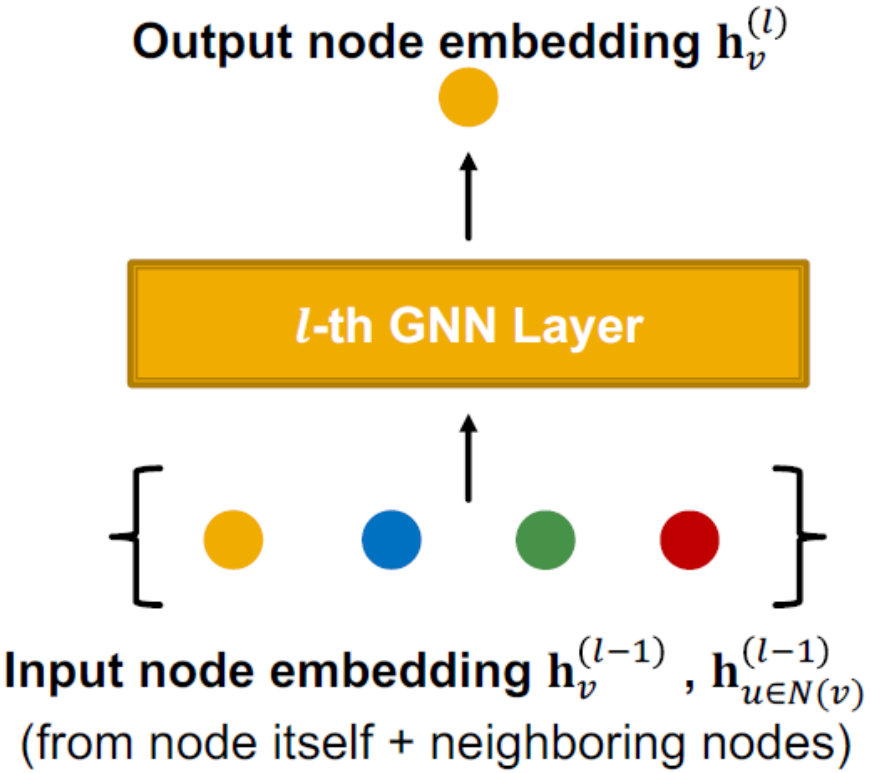


A Single GNN Layer

Idea of a GNN Layer:

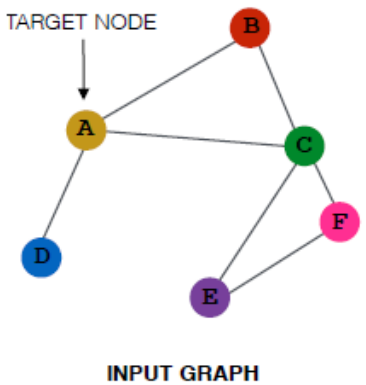
- Compress a set of vectors into a single vector
- **Two-step process:**
  - **(1) Message**
  - **(2) Aggregation**



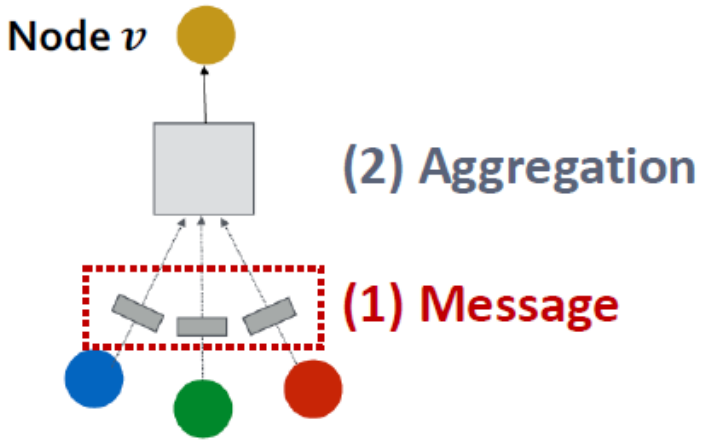


Message Computation

- **(1) Message computation**
  - **Message function:** $\mathbf{m}_u^{(l)} = \mathrm{MSG}^{(l)}\left(\mathbf{h}_u^{(l-1)}\right)$
    - **Intuition:** Each node will create a message, which will be sent to other nodes later
    - **Example:** A Linear layer $\mathbf{m}_u^{(l)} = \mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}$
      - Multiply node features with weight matrix $\mathbf{W}^{(l)}$



INPUT GRAPH

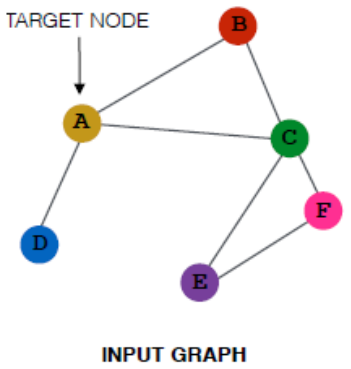


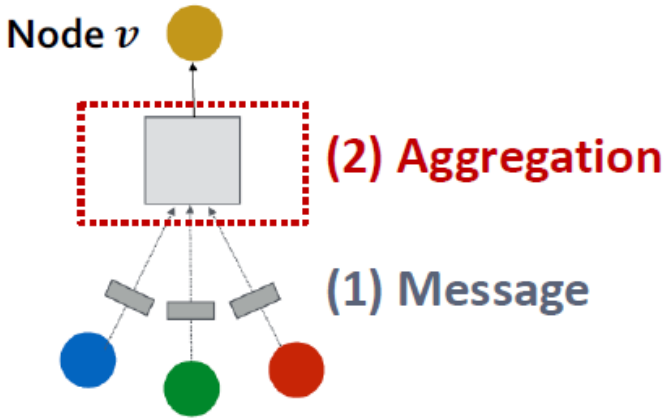
Message Aggregation

- **(2) Aggregation**
  - **Intuition:** Node v will aggregate the messages from its neighbors u:

$$\mathbf{h}_v^{(l)} = \mathrm{AGG}^{(l)}\left(\left\{\mathbf{m}_u^{(l)}, u \in N(v)\right\}\right)$$

  - **Example:** $\mathrm{Sum}(\cdot)$, $\mathrm{Mean}(\cdot)$ or $\mathrm{Max}(\cdot)$ aggregator
    - $\mathbf{h}_v^{(l)} = \mathrm{Sum}(\{\mathbf{m}_u^{(l)}, u \in N(v)\})$





Message Aggregation Issue

Issue: Information from node v itself **could get lost**

- Computation of $\mathbf{h}_v^{(l)}$ does not directly depend on $\mathbf{h}_v^{(l-1)}$
- **Solution:** Include $\mathbf{h}_v^{(l-1)}$ when computing $\mathbf{h}_v^{(l)}$
  - **(1) Message: compute message from node v itself**
  - Usually, a **different message computation** will be performed

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

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

  - **(2) Aggregation:** After aggregating from neighbors, we can **aggregate the message from node v itself**
  - Via **concatenation** or **summation**

Then aggregate from node itself

$$\mathbf{h}_v^{(l)} = \text{CONCAT}\left(\text{AGG}\left(\left\{\mathbf{m}_u^{(l)}, u \in N(v)\right\}\right), \mathbf{m}_v^{(l)}\right)$$

First aggregate from neighbors

A Single GNN Layer

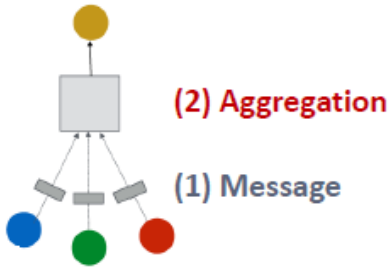
- **Putting things together:**
  - **(1) Message**: each node computes a message

$$\mathbf{m}_u^{(l)} = \mathrm{MSG}^{(l)}\left(\mathbf{h}_u^{(l-1)}\right), u \in \{N(v) \cup v\}$$

  - **(2) Aggregation**: aggregate messages from neighbors

$$\mathbf{h}_v^{(l)} = \mathrm{AGG}^{(l)}\left(\left\{\mathbf{m}_u^{(l)}, u \in N(v)\right\}, \mathbf{m}_v^{(l)}\right)$$

  - **Nonlinearity (activation):** Adds expressiveness
    - Often written as $\sigma(\cdot)$. Examples: $\mathrm{ReLU}(\cdot)$, $\mathrm{Sigmoid}(\cdot)$, ...
    - Can be added to **message or aggregation**



Classical GNN Layers: GCN

- **(1) Graph Convolutional Networks (GCN)**

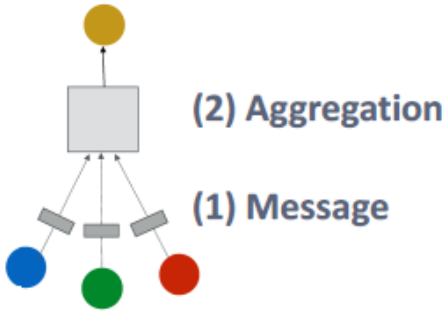
$$\mathbf{h}_v^{(l)} = \sigma\left(\mathbf{W}^{(l)} \sum_{u \in N(v)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}\right)$$

- **How to write this as Message + Aggregation?**

Message

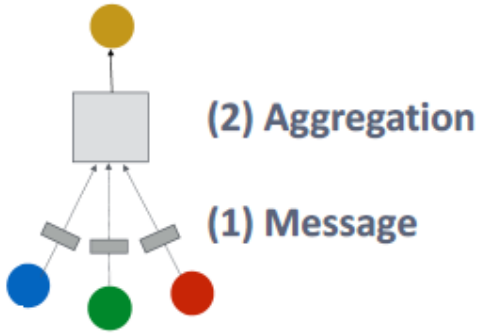
$$\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}\right)$$

Aggregation



Classical GNN Layers: GCN

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



- **Message:**
  - Each Neighbor: $\mathbf{m}_u^{(l)} = \frac{1}{|N(v)|}\mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}$

Normalized by node degree
(In the GCN paper they use a slightly different normalization)

- **Aggregation:**
  - **Sum** over messages from neighbors, then apply activation
  - $\mathbf{h}_v^{(l)} = \sigma\left(\text{Sum}\left(\left\{\mathbf{m}_u^{(l)}, u \in N(v)\right\}\right)\right)$

In GCN the input graph is assumed to have self-edges that are included in the summation.

Classical GNN Layers: GraphSAGE

- **(2) GraphSAGE**

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

- **How to write this as Message + Aggregation?**
  - **Message** is computed within the $\text{AGG}(\cdot)$
  - **Two-stage aggregation**
    - **Stage 1:** Aggregate from node neighbors

$$\mathbf{h}_{N(v)}^{(l)} \leftarrow \text{AGG}\left(\left\{\mathbf{h}_u^{(l-1)}, \forall u \in N(v)\right\}\right)$$

    - **Stage 2:** Further aggregate over the node itself

$$\mathbf{h}_v^{(l)} \leftarrow \sigma\left(\mathbf{W}^{(l)} \cdot \text{CONCAT}(\mathbf{h}_v^{(l-1)}, \mathbf{h}_{N(v)}^{(l)})\right)$$

GraphSAGE Neighbor Aggregation

- **Mean:** Take a weighted average of neighbors

$$\text{AGG} = \sum_{u \in N(v)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}$$

Aggregation — Message computation

- **Pool:** Transform neighbor vectors and apply symmetric vector function $\text{Mean}(\cdot)$ or $\text{Max}(\cdot)$

$$\text{AGG} = \text{Mean}(\{\text{MLP}(\mathbf{h}_u^{(l-1)}), \forall u \in N(v)\})$$

Aggregation — Message computation

- **LSTM:** Apply LSTM to reshuffled of neighbors

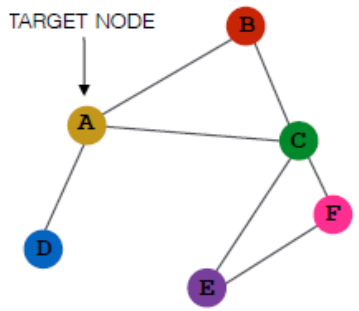
$$\text{AGG} = \text{LSTM}([\mathbf{h}_u^{(l-1)}, \forall u \in \pi(N(v))])$$

Aggregation

GraphSAGE L_2 Normalization

- ℓ_2 **Normalization:**
  - **Optional:** Apply ℓ_2 normalization to $\mathbf{h}_v^{(l)}$ at every layer
  - $\mathbf{h}_v^{(l)} \leftarrow \frac{\mathbf{h}_v^{(l)}}{\left\|\mathbf{h}_v^{(l)}\right\|_2} \forall v \in V$ where $\|u\|_2 = \sqrt{\sum_i u_i^2}$ (ℓ_2-norm)
  - Without ℓ_2 normalization, the embedding vectors have different scales (ℓ_2-norm) for vectors
  - In some cases (not always), normalization of embedding results in performance improvement
  - After ℓ_2 normalization, all vectors will have the same ℓ_2-norm

Stacking GNN Layers

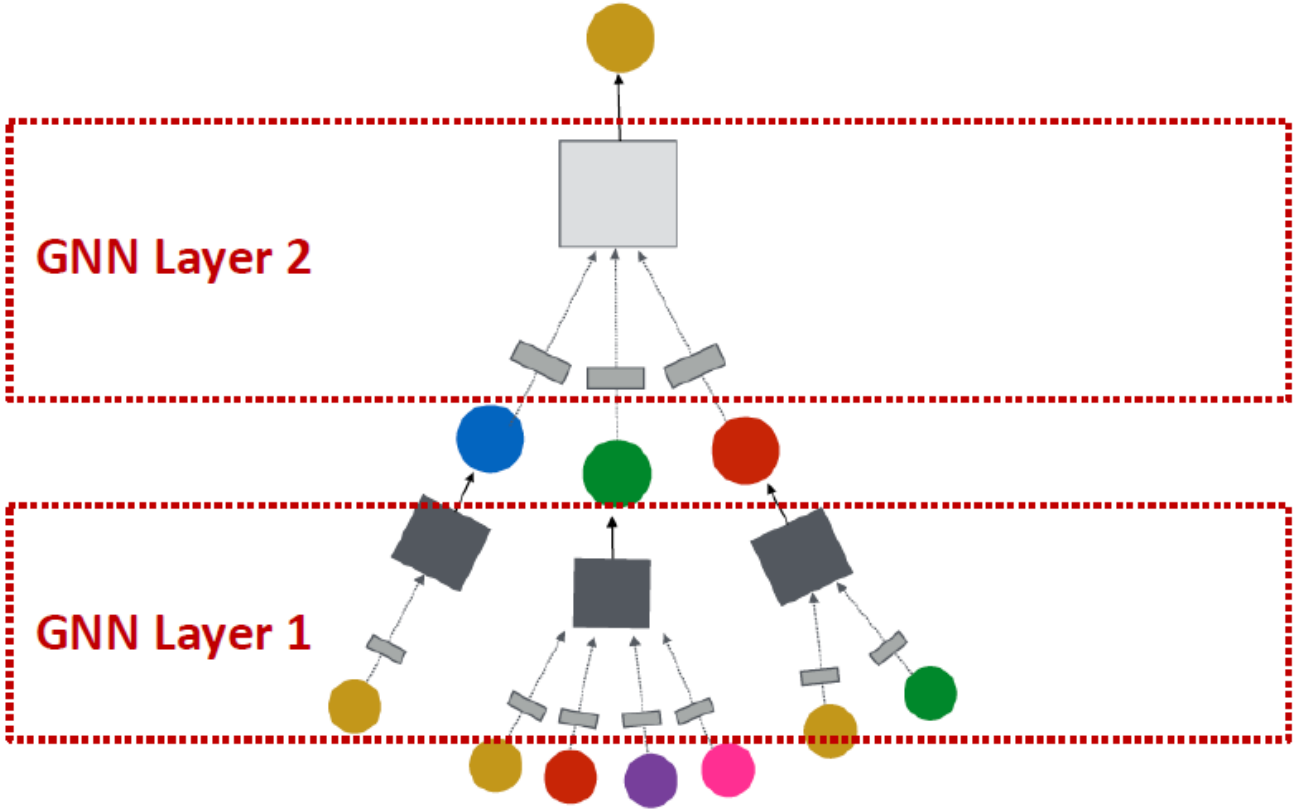


INPUT GRAPH

How to connect GNN layers into a GNN?

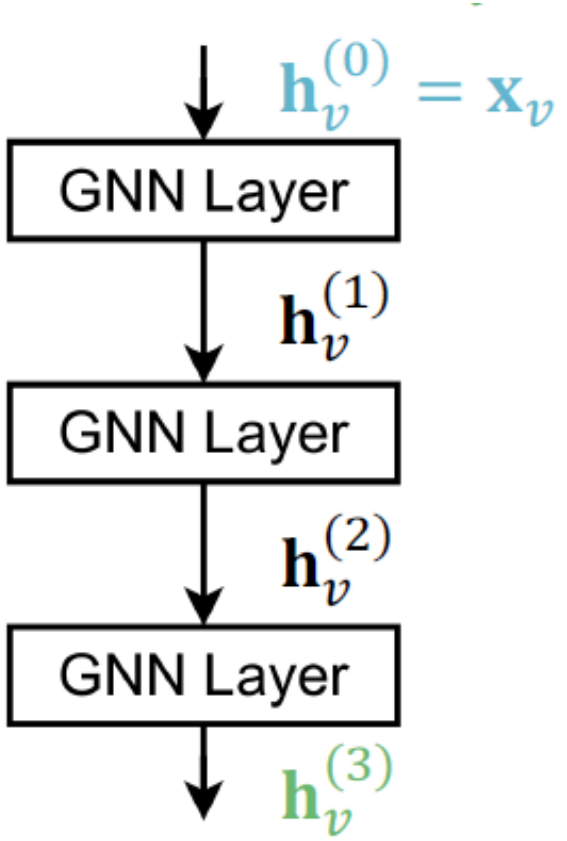
- **Stack layers sequentially**
- **Ways of adding skip connections**

(3) Layer connectivity



Stacking GNN Layers

- **How to construct a Graph Neural Network?**
- **The standard way:** Stack GNN layers sequentially
- **Input:** Initial raw node feature $\mathbf{x}_v$
- **Output:** Node embeddings $\mathbf{h}_v^{(L)}$ after L GNN layers

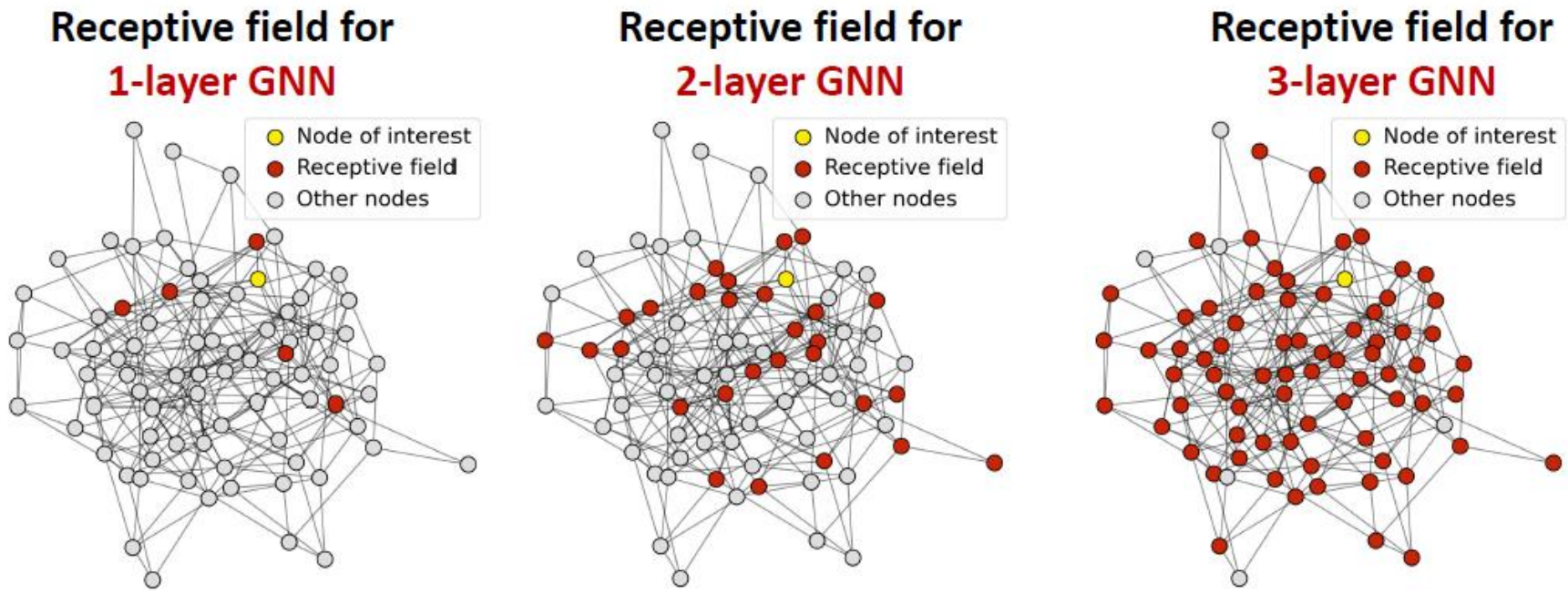


Over-smoothing Problem

- **Issue of stacking many GNN layers**
  - GNN suffers from **over-smoothing problem**
- **The over-smoothing problem: all node embeddings converge to the same value**
  - This is bad because we **want to use node embeddings to differentiate nodes**
- **Why does the over-smoothing problem happen?**

Receptive Field of a GNN

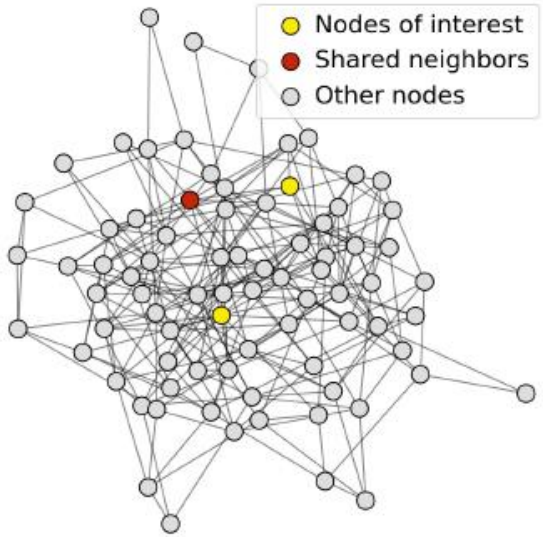
- **Receptive field:** set of nodes that determine embedding of a node of interest
  - **In a K-layer GNN, each node has a receptive field of K-hop neighborhood**



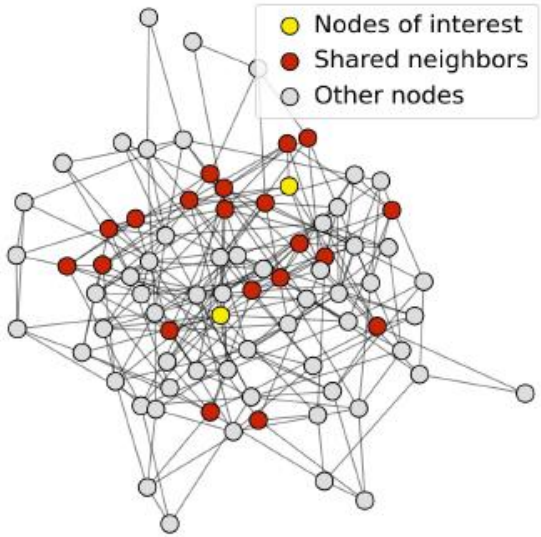
Receptive Field of a GNN

- **Receptive field overlap** for two nodes
- **Shared neighbors quickly grows** when we increase number of hops (num of GNN layers)

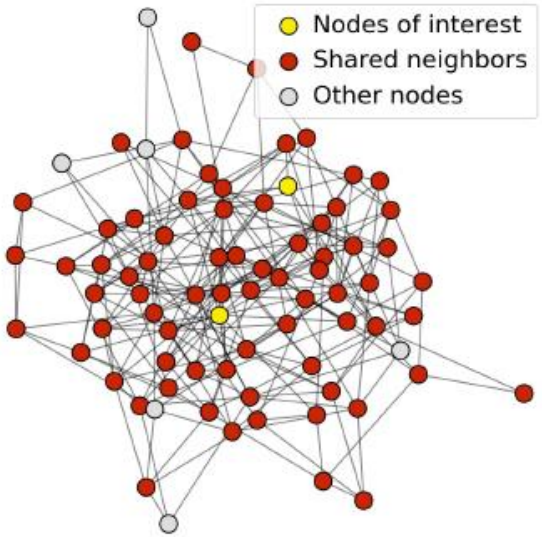
1-hop neighbor overlap
Only 1 node



2-hop neighbor overlap
About 20 nodes



3-hop neighbor overlap
Almost all the nodes!



Receptive Field and Over-smoothing

- **Eembedding of a node is determined by its receptive field**
  - If two nodes **have highly-overlapped receptive fields, then their embeddings are highly similar**
- **Stack many GNN layers → nodes will have highly overlapped receptive fields → node embeddings will be highly similar → suffer from the over-smoothing problem**

Design GNN Layer Connectivity

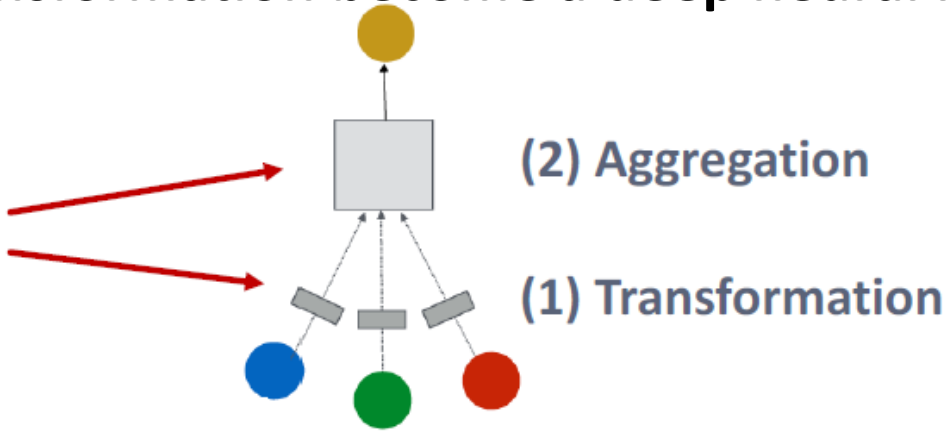
- How do we overcome over-smoothing problem?
- **Lesson 1: Be cautious when adding GNN layers**
  - Unlike neural networks in other domains (CNN for image classification), **adding more GNN layers do not always help**
  - **Step 1: Analyze necessary receptive field** to solve your problem
    - E.g., by computing the diameter of the graph
  - **Step 2:** Set number of GNN layers L to be a bit more than the receptive field we like
    - **Do not set L to be unnecessarily large**!

Expressive Power for Shallow GNNs

How to enhance expressive power of a GNN, if number of GNN layers is small?

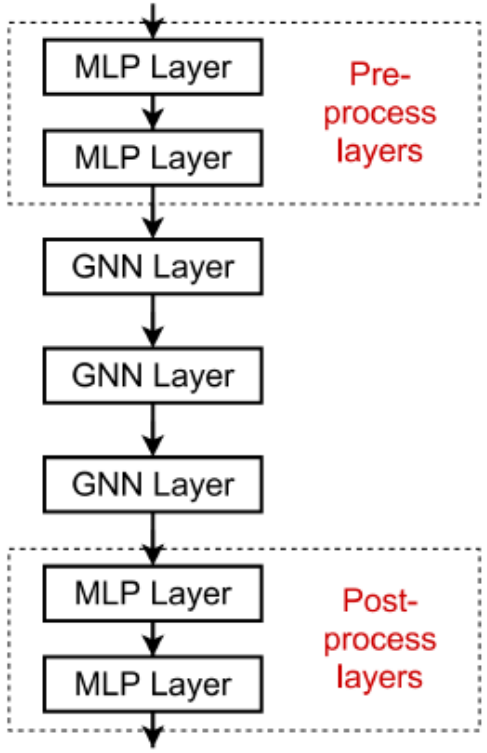
- **How to make a shallow GNN more expressive?**
- **Solution 1:** Increase expressive power **within each GNN layer**
  - In previous examples, each transformation or aggregation function only include one linear layer
  - We can **make aggregation / transformation become a deep neural network**!

If needed, each box could include a 3-layer MLP



Expressive Power for Shallow GNNs

- Solution 2: Add layers that do not pass messages
  - A GNN does not necessarily only contain GNN layers
  - E.g., we can add **MLP layers** (applied to each node) before and after GNN layers, as **pre-process layers** and **post-process layers**



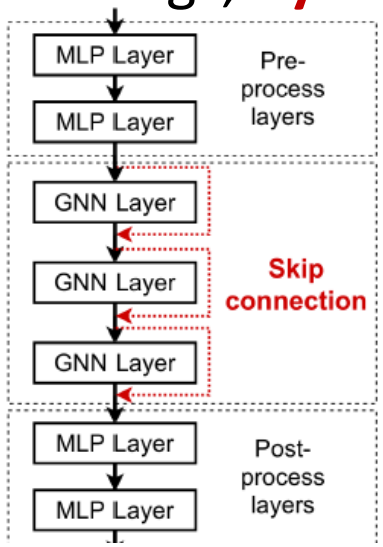
Pre-processing layers: Important when encoding node features is necessary.
E.g., when nodes represent images/text

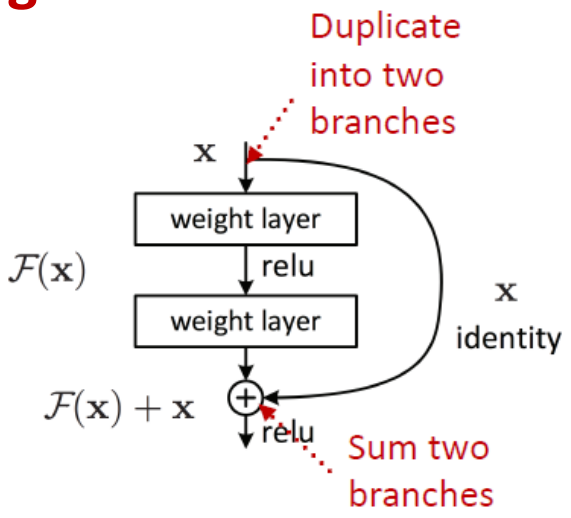
Post-processing layers: Important when reasoning / transformation over node embeddings are needed
E.g., graph classification, knowledge graphs

In practice, adding these layers works great!

Design GNN Layer Connectivity

- **What if my problem still requires many GNN layers?**
- **Lesson 2: Add skip connections in GNNs**
  - **Observation from over-smoothing:** Node embeddings in earlier GNN layers can sometimes better differentiate nodes
  - **Solution:** We can increase the impact of earlier layers on the final node embeddings, **by adding shortcuts in GNN**





Idea of skip connections:

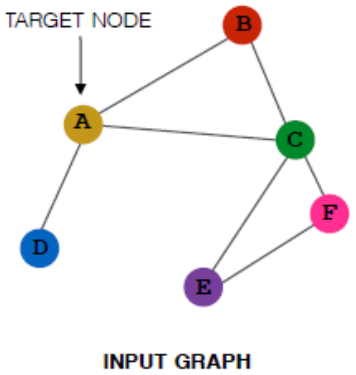
Before adding shortcuts:

$$\boldsymbol{F}(\mathbf{x})$$

After adding shortcuts:

$$\boldsymbol{F}(\mathbf{x}) + \mathbf{x}$$

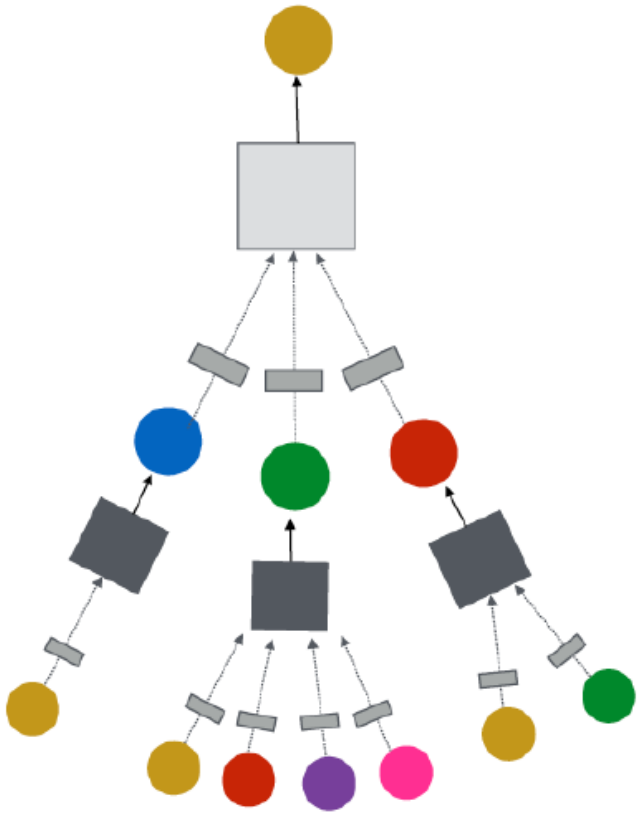
Graph Manipulation



INPUT GRAPH

Idea: Raw input graph ≠ computational graph

- **Graph feature augmentation**
- **Graph structure manipulation**



(4) Graph manipulation

Why Manipulate Graphs?

- **Our assumption so far has been**

  Raw input graph = computational graph

- **Reasons for breaking this assumption**
- **Feature level:**
  - Input graph **lacks features** → feature augmentation
- **Structure level:**
  - Graph is **too sparse** → inefficient message passing
  - Graph is **too dense** → message passing is too costly
  - Graph is **too large** → cannot fit the computational graph into a GPU
- **Unlikely that input graph happens to be optimal computation graph for embeddings**

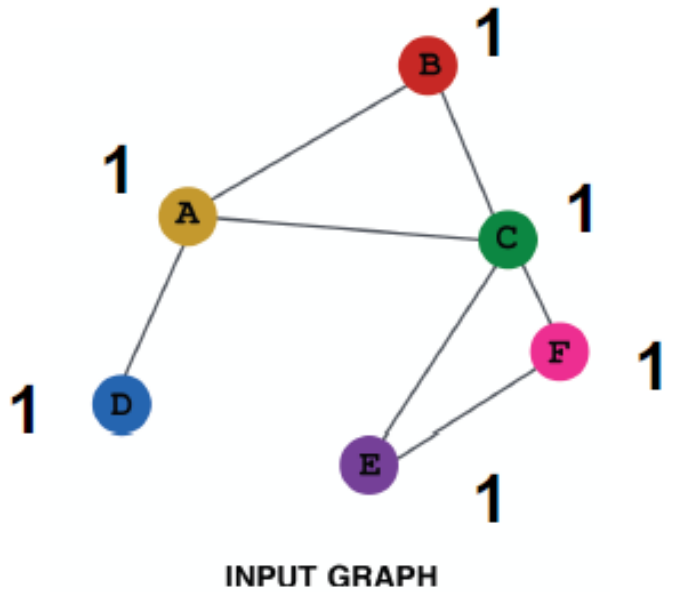
Graph Manipulation Approaches

- **Graph Feature manipulation**
  - Input graph **lacks features** → **feature augmentation**
- **Graph Structure manipulation**
  - Graph is **too sparse** → **Add virtual nodes / edges**
  - Graph is **too dense** → **Sample neighbors when doing message passing**
  - Graph is **too large** → **Sample subgraphs to compute embeddings**

Feature Augmentation on Graphs

Why do we need feature augmentation?

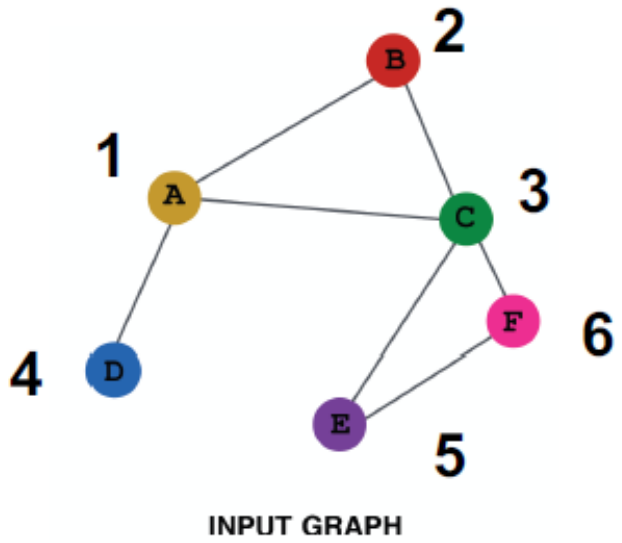
- **(1) Input graph does not have node features**
  - This is common when we only have the adj. matrix
- **Standard approaches:**
  - **a) Assign constant values to nodes**



INPUT GRAPH

Feature Augmentation on Graphs

- **b) Assign unique IDs to nodes**
  - These IDs are converted into **one-hot vectors**



INPUT GRAPH

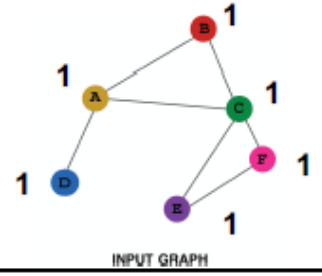
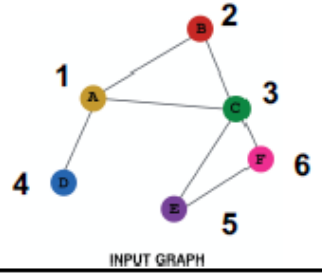
One-hot vector for node with ID=5

ID = 5

[0, 0, 0, 0, 1, 0]

Total number of IDs = 6

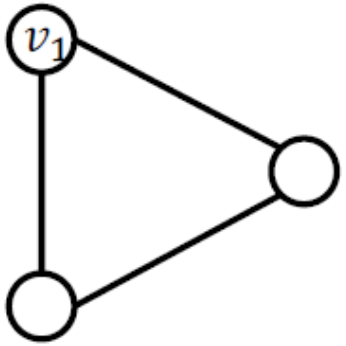
Feature augmentation: **constant** vs. **one-hot**

	Constant node feature	**One-hot node feature**
		
Expressive power	**Medium**. All the nodes are identical, but GNN can still learn from the graph structure	**High**. Each node has a unique ID, so node-specific information can be stored
Inductive learning (Generalize to unseen nodes)	**High**. Simple to generalize to new nodes: we assign constant feature to them, then apply our GNN	**Low**. Cannot generalize to new nodes: new nodes introduce new IDs, GNN doesn't know how to embed unseen IDs
Computational cost	**Low**. Only 1 dimensional feature	**High**. High dimensional feature, cannot apply to large graphs
Use cases	Any graph, inductive settings (generalize to new nodes)	Small graph, transductive settings (no new nodes)

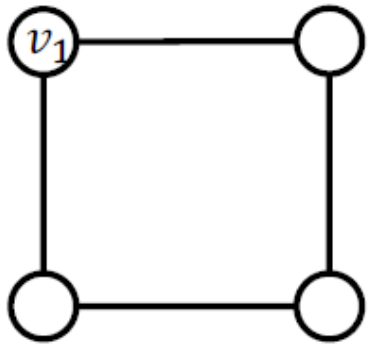
Feature Augmentation on Graphs

- **Why do we need feature augmentation?**
- **(2) Certain structures are hard to learn by GNN**
  - **Example:** Cycle count feature
  - Can GNN learn the length of a cycle that v^* resides in?
  - **Unfortunately, no**

v_1 resides in a cycle with length 3



v_1 resides in a cycle with length 4



Feature Augmentation on Graphs

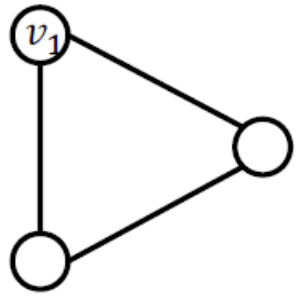
- **Solution:**
- We can use cycle count as augmented node features

We start from cycle with length 0

Augmented node feature for v_1

[0, 0, 0, 1, 0, 0]

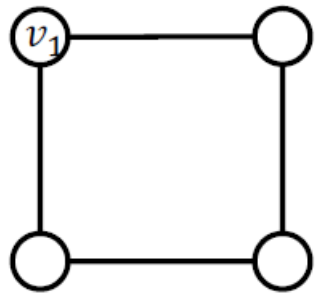
v_1 resides in a cycle with length 3



Augmented node feature for v_1

[0, 0, 0, 0, 1, 0]

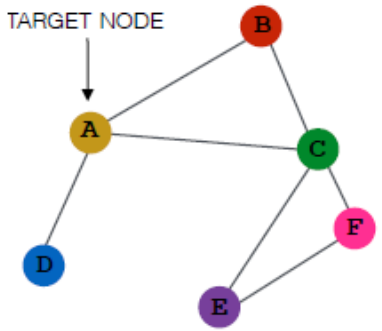
v_1 resides in a cycle with length 4



Feature Augmentation on Graphs

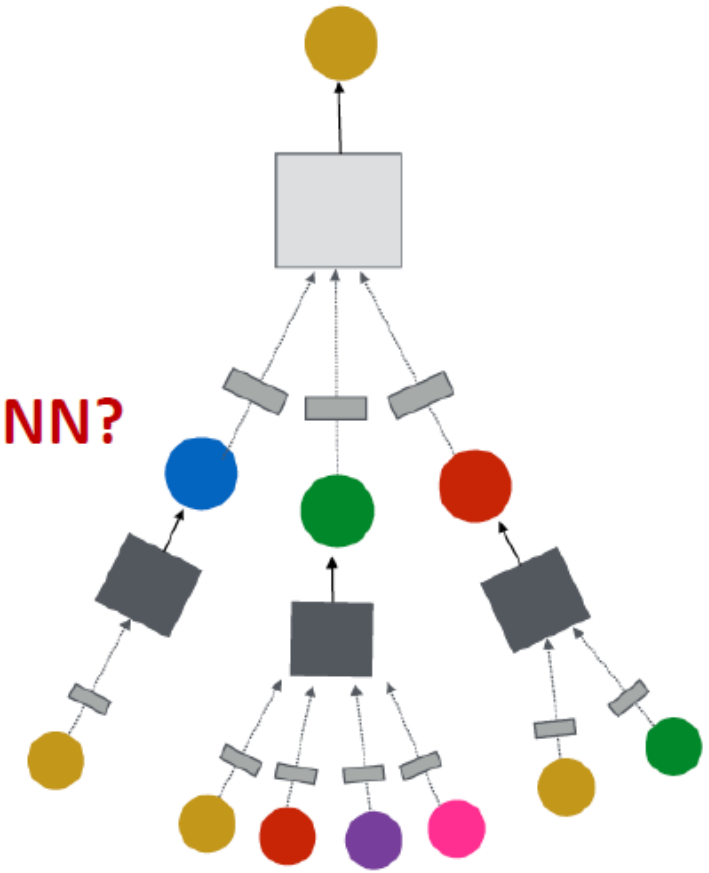
- Other commonly used augmented features:
- **Clustering coefficient**
- **PageRank**
- **Centrality**
- **…**

A General GNN Framework



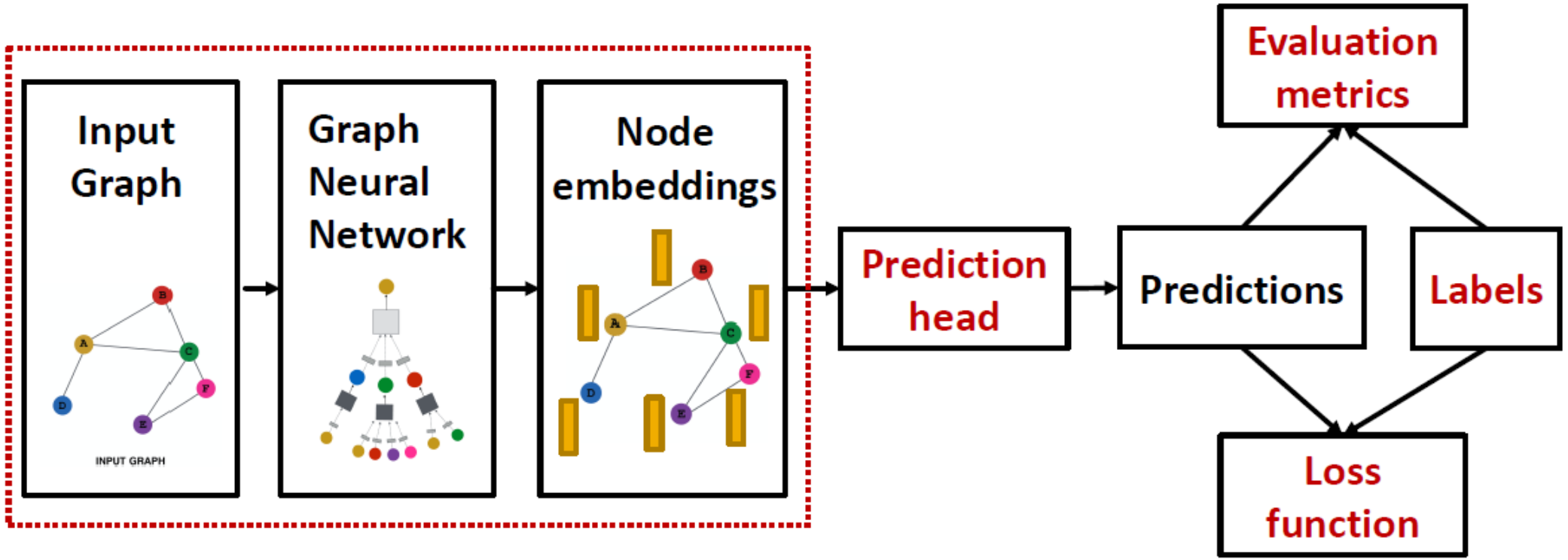
INPUT GRAPH

(5) Learning objective



Next: How do we train a GNN?

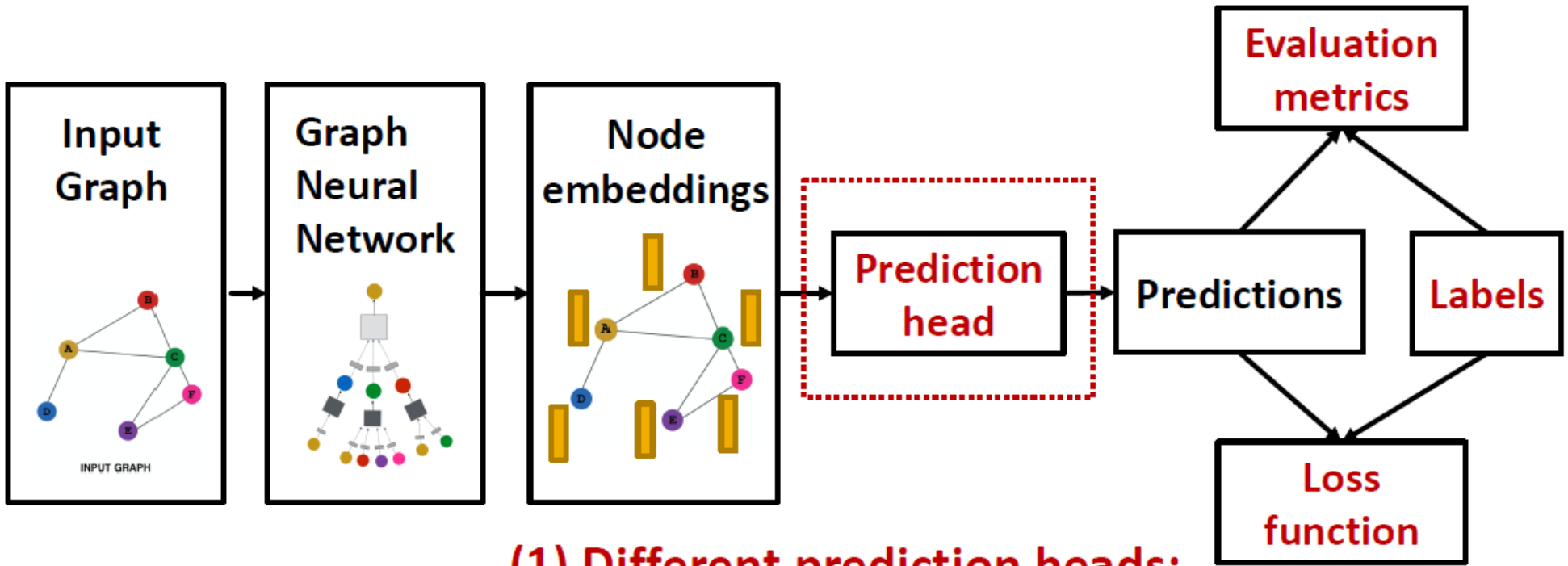
GNN Training Pipeline



Output of a GNN: set of node embeddings

$$\{\mathbf{h}_v^{(L)}, \forall v \in G\}$$

GNN Training Pipeline

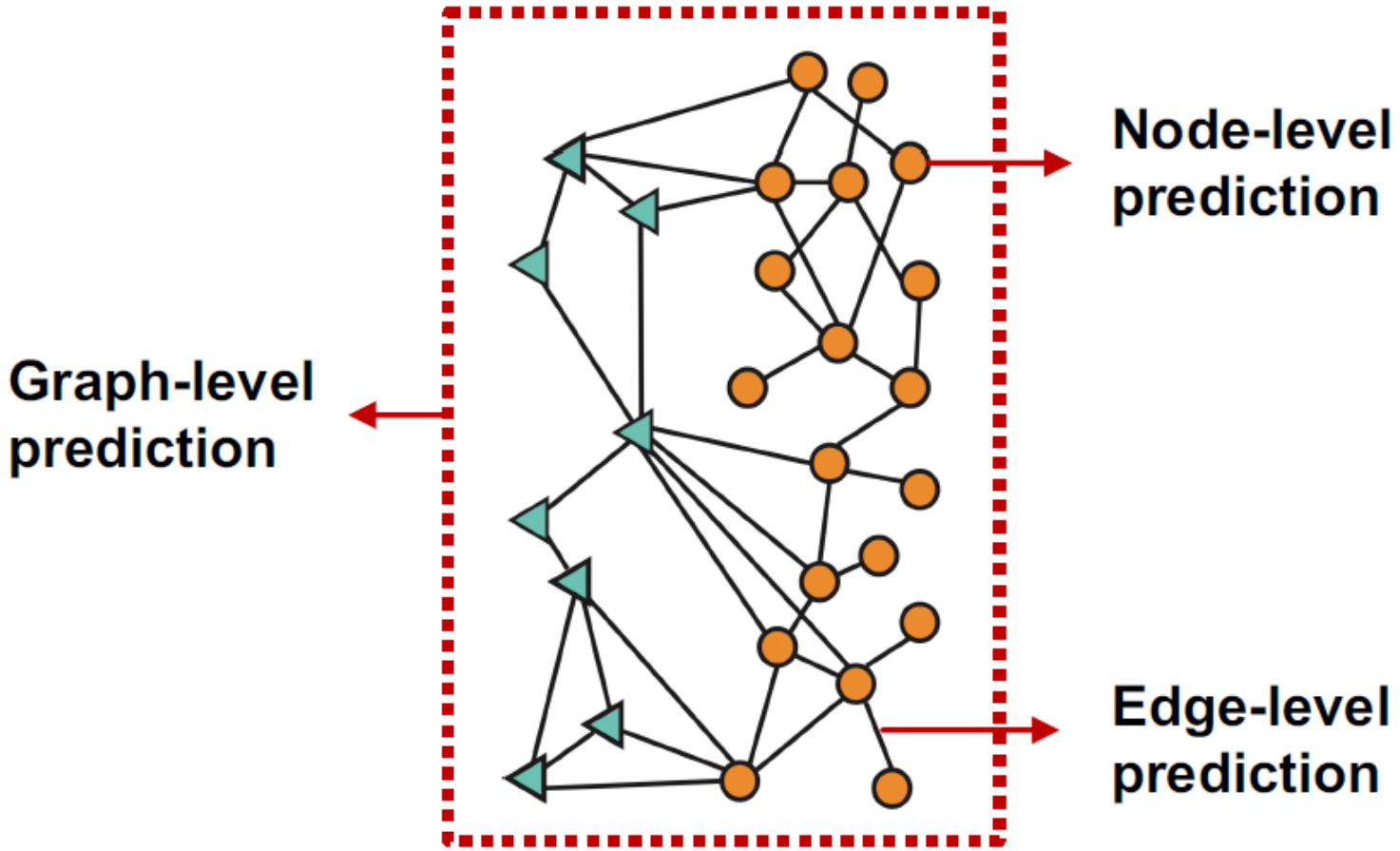


(1) Different prediction heads:

- Node-level tasks
- Edge-level tasks
- Graph-level tasks

GNN Prediction Heads

- **Idea:** Different task levels require different prediction heads



Prediction Heads: Node Level

- **Node-level prediction**: We can directly make prediction using node embeddings!
- After GNN computation, we have d-dim node embeddings:

$$\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\}$$

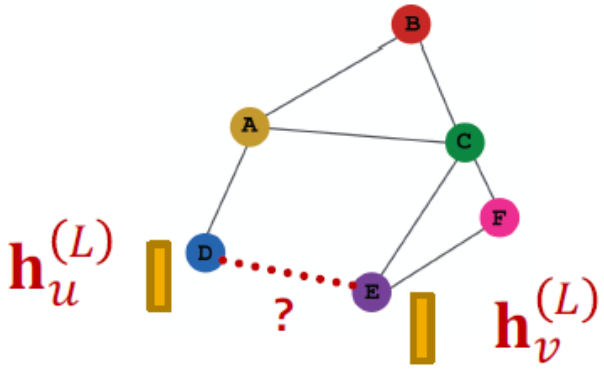
- Suppose we want to make k-way prediction
  - Classification: classify among k categories
  - Regression: regress on k targets

$$\hat{\boldsymbol{y}}_v = \text{Head}_{\text{node}}(\mathbf{h}_v^{(L)}) = \mathbf{W}^{(H)}\mathbf{h}_v^{(L)}$$

  - $\mathbf{W}^{(H)} \in \mathbb{R}^{k \times d}$: We map node embeddings from $\mathbf{h}_v^{(L)} \in \mathbb{R}^d$ to $\hat{\boldsymbol{y}}_v \in \mathbb{R}^k$ so that we can compute the loss

Prediction Heads: Edge Level

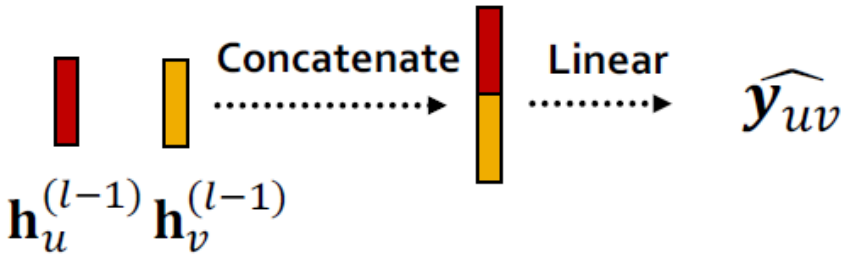
- **Edge-level prediction**: Make prediction using pairs of node embeddings
- Suppose we want to make k-way prediction
  - $\hat{\boldsymbol{y}}_{uv} = \text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$



  - What are the options for $\text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$?

Prediction Heads: Edge Level

- **Options for** $\text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$:
- **(1) Concatenation + Linear**
  - We have seen this in graph attention



  - $\widehat{\boldsymbol{y}}_{\boldsymbol{uv}} = \text{Linear}(\text{Concat}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)}))$
  - Here $\text{Linear}(\cdot)$ will map $2d$-dimensional embeddings (since we concatenated embeddings) to k-dim embeddings (k-way prediction)

Prediction Heads: Edge Level

- **Options for** $\text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$:
- **(2) Dot product**
  - $\hat{y}_{uv} = (\mathbf{h}_u^{(L)})^T \mathbf{h}_v^{(L)}$
  - **This approach only applies to 1-way prediction** (e.g., link prediction: predict the existence of an edge)
  - **Applying to k-way prediction:**
    - Similar to **multi-head attention**: $\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(k)}$ trainable

$$\hat{y}_{uv}^{(1)} = (\mathbf{h}_u^{(L)})^T \mathbf{W}^{(1)} \mathbf{h}_v^{(L)}$$

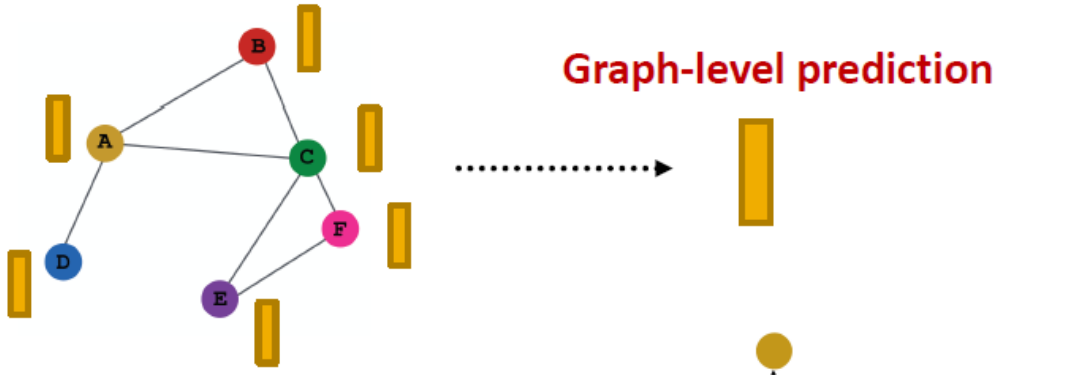
$$\ldots$$

$$\hat{y}_{uv}^{(k)} = (\mathbf{h}_u^{(L)})^T \mathbf{W}^{(k)} \mathbf{h}_v^{(L)}$$

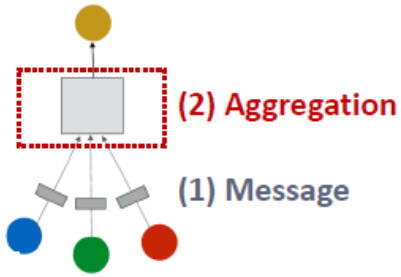
$$\hat{\mathbf{y}}_{uv} = \text{Concat}(\hat{y}_{uv}^{(1)}, \ldots, \hat{y}_{uv}^{(k)}) \in \mathbb{R}^k$$

Prediction Heads: Graph Level

- **Graph-level prediction**: Make prediction using all the node embeddings in our graph
- Suppose we want to make k-way prediction
  - $\hat{\boldsymbol{y}}_G = \text{Head}_{\text{graph}}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$



  - $\text{Head}_{\text{graph}}(\cdot)$ is similar to $\text{AGG}(\cdot)$ in a GNN layer!



Prediction Heads: Graph Level

- Options for $\text{Head}_{\text{graph}}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$
- **(1) Global mean pooling**

$$\hat{\boldsymbol{y}}_G = \text{Mean}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$$

- **(2) Global max pooling**

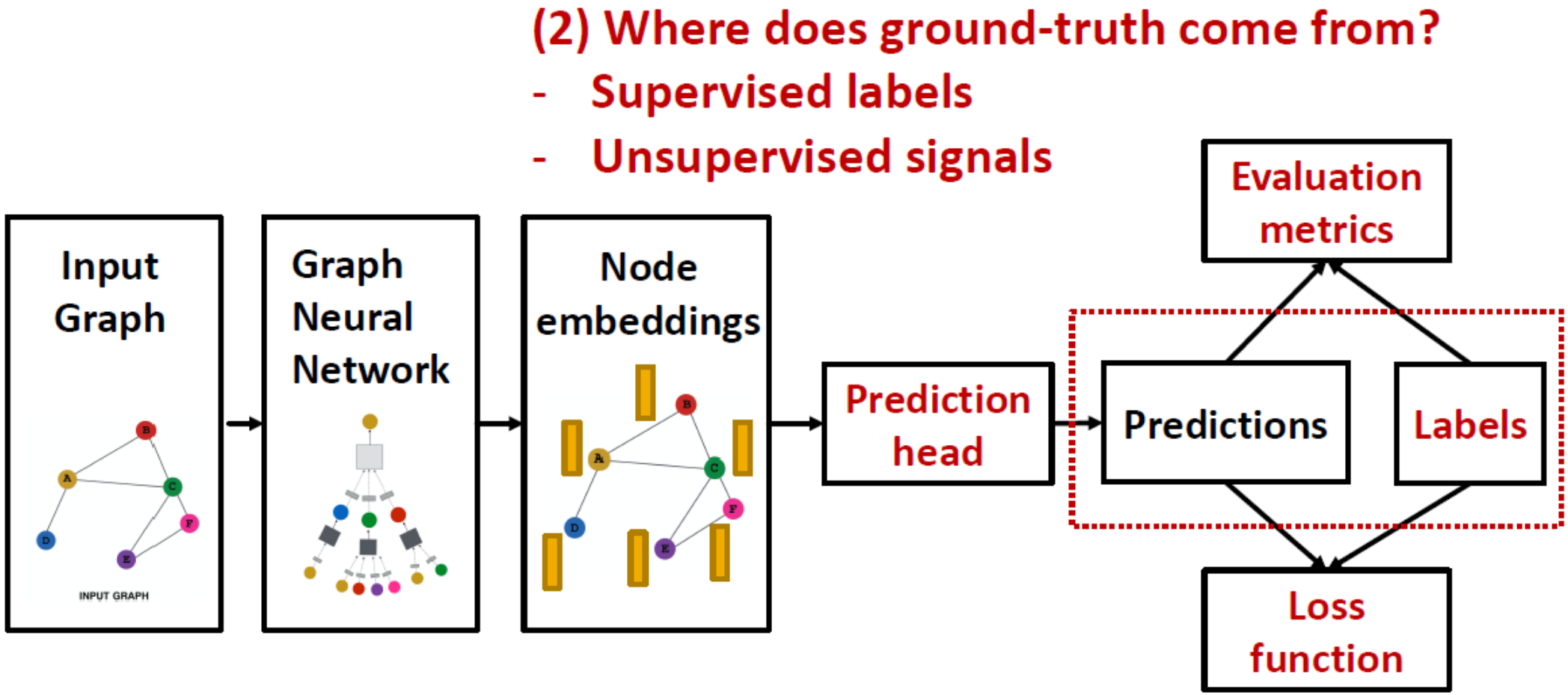
$$\hat{\boldsymbol{y}}_G = \text{Max}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$$

- **(3) Global sum pooling**

$$\hat{\boldsymbol{y}}_G = \text{Sum}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$$

- These options work great for small graphs

GNN Training Pipeline



Supervised vs. Unsupervised

- **Supervised learning on graphs**
  - **Labels come from external sources**
  - E.g., predict drug likeness of a molecular graph
- **Unsupervised learning on graphs**
  - **Signals come from graphs themselves**
  - E.g., link prediction: predict if two nodes are connected
- **Sometimes the differences are blurry**
  - We still have “supervision” in unsupervised learning
  - E.g., train a GNN to predict node clustering coefficient
  - An alternative name for “**unsupervised**” is “**self-supervised**”

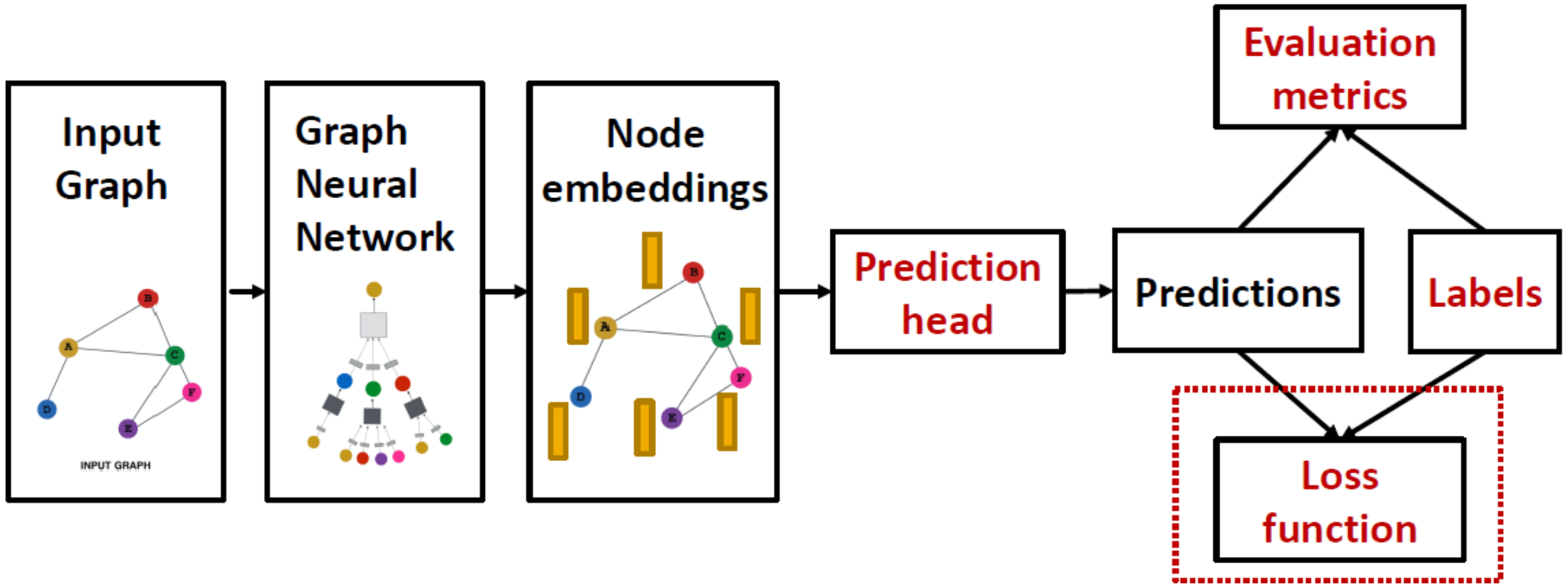
Supervised Labels on Graphs

- **Supervised labels come from the specific use cases**. For example:
  - **Node labels y_v:** in a citation network, which subject area does a node belong to
  - **Edge labels y_{uv}:** in a transaction network, whether an edge is fraudulent
  - **Graph labels y_G:** among molecular graphs, the drug likeness of graphs
- **Advice:** Reduce your task to node / node / edge / graph labels, since they are easy to work with
  - **E.g.,** we knew some nodes form a cluster. We can treat the cluster that a node belongs to as a **node label**

Unsupervised Signals on Graphs

- Sometimes **we only have a graph, without any external labels**
- **The solution:** "self-supervised learning", we can find supervision signals within the graph.
  - For example, we can **let GNN predict the following:**
  - **Node-level y_v.** Node statistics: such as clustering coefficient, PageRank, …
  - **Edge-level y_{uv}.** Link prediction: hide the edge between two nodes, predict if there should be a link
  - **Graph-level y_G.** Graph statistics: for example, predict if two graphs are isomorphic
  - **These tasks do not require any external labels**

GNN Training Pipeline



(3) How do we compute the final loss?

- **Classification loss**
- **Regression loss**

Settings for GNN Training

- **The setting:** We have N data points
  - Each data point can be a node/edge/graph
    - **Node-level**: prediction $\hat{\boldsymbol{y}}_v^{(i)}$, label $\boldsymbol{y}_v^{(i)}$
    - **Edge-level**: prediction $\hat{\boldsymbol{y}}_{uv}^{(i)}$, label $\boldsymbol{y}_{uv}^{(i)}$
    - **Graph-level**: prediction $\hat{\boldsymbol{y}}_G^{(i)}$, label $\boldsymbol{y}_G^{(i)}$
    - We will use prediction $\hat{\boldsymbol{y}}^{(i)}$, label $\boldsymbol{y}^{(i)}$ to refer **predictions at all levels**

Classification Loss

- **Cross entropy (CE)** is a very common loss function in classification
  - K-way prediction for i-th data point:

$$\mathrm{CE}\left(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)}\right) = -\sum_{j=1}^{K} \boldsymbol{y}_j^{(i)} \log\left(\hat{\boldsymbol{y}}_j^{(i)}\right)$$

Label: $\boldsymbol{y}^{(i)}$ — Prediction: $\hat{\boldsymbol{y}}^{(i)}$

i-th data point: (i) — j-th class: j

where:

E.g. | 0 | 0 | 1 | 0 | 0 |

$\boldsymbol{y}^{(i)} \in \mathbb{R}^K$ = one-hot label encoding

$\hat{\boldsymbol{y}}^{(i)} \in \mathbb{R}^K$ = prediction after $\mathrm{Softmax}(\cdot)$

E.g. | 0.1 | 0.3 | 0.4 | 0.1 | 0.1 |

  - Total loss over all N training examples

$$\text{Loss} = \sum_{i=1}^{N} \mathrm{CE}\left(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)}\right)$$

Regression Loss

- For regression tasks can use **Mean Squared Error (MSE)** a.k.a. **L2 loss**
  - K-way regression for data point (i):

$$\text{MSE}\left(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)}\right) = \sum_{j=1}^{K} \left(\boldsymbol{y}_j^{(i)} - \hat{\boldsymbol{y}}_j^{(i)}\right)^2$$

i-th data point

j-th target

where:

E.g. | 1.4 | 2.3 | 1.0 | 0.5 | 0.6 |

$\boldsymbol{y}^{(i)} \in \mathbb{R}^k$ = Real valued vector of targets

$\hat{\boldsymbol{y}}^{(i)} \in \mathbb{R}^k$ = Real valued vector of predictions

E.g. | 0.9 | 2.8 | 2.0 | 0.3 | 0.8 |

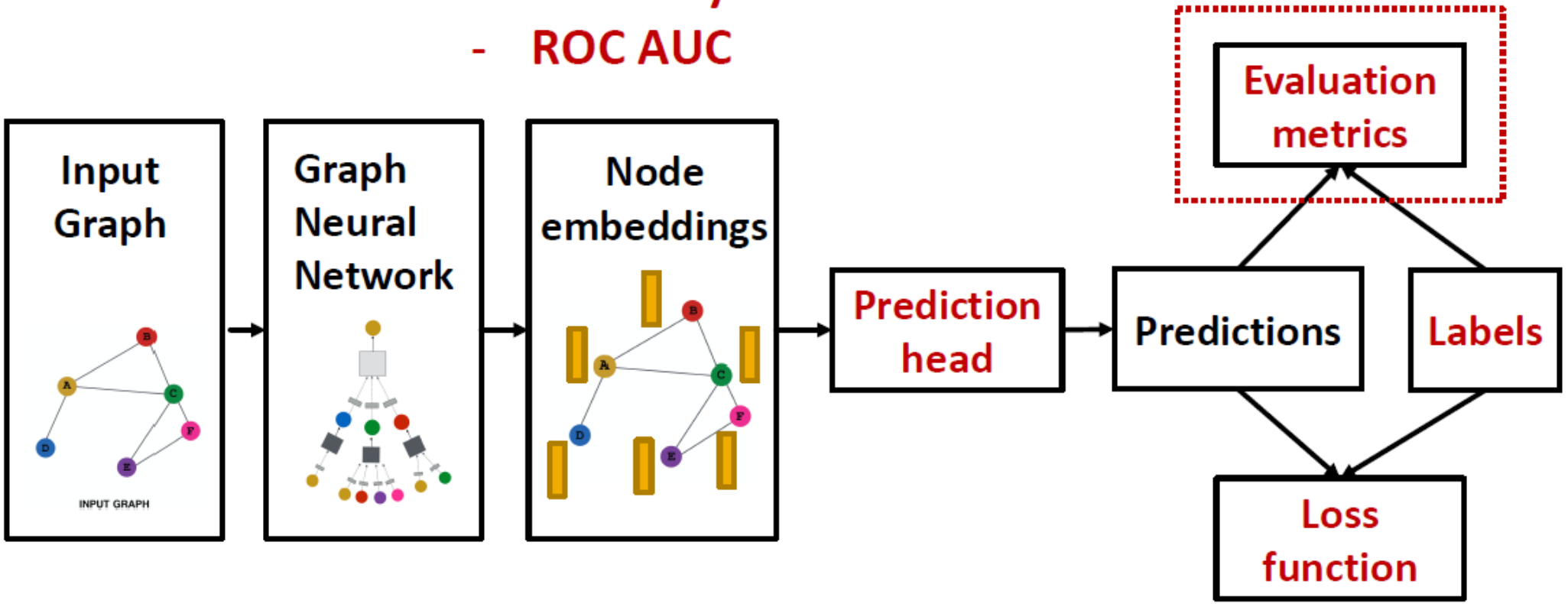
  - Total loss over all N training examples

$$\text{Loss} = \sum_{i=1}^{N} \text{MSE}\left(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)}\right)$$

GNN Training Pipeline

(4) How do we measure the success of a GNN?

- **Accuracy**
- **ROC AUC**



Evaluation Metrics: Regression

- **Root mean square error (RMSE)**

$$\sqrt{\sum_{i=1}^{N} \frac{(\boldsymbol{y}^{(i)} - \widehat{\boldsymbol{y}}^{(i)})^2}{N}}$$

- **Mean absolute error (MAE)**

$$\frac{\sum_{i=1}^{N} \left| \boldsymbol{y}^{(i)} - \widehat{\boldsymbol{y}}^{(i)} \right|}{N}$$

Evaluation Metrics: Classification

- **(1) Multi-class classification**
  - We simply report the accuracy

$$\frac{1\left[\text{argmax}\left(\hat{\boldsymbol{y}}^{(i)}\right) = \boldsymbol{y}^{(i)}\right]}{N}$$

- **(2) Binary classification**
  - Metrics sensitive to classification threshold
    - Accuracy
    - Precision / Recall
    - If the range of prediction is $[0,1]$, we will use 0.5 as threshold
  - Metric Agnostic to classification threshold
    - **ROC AUC**