DSC250: Advanced Data Mining

Graph Neural Networks

Zhiting Hu

Lecture 12, Feb 13, 2025

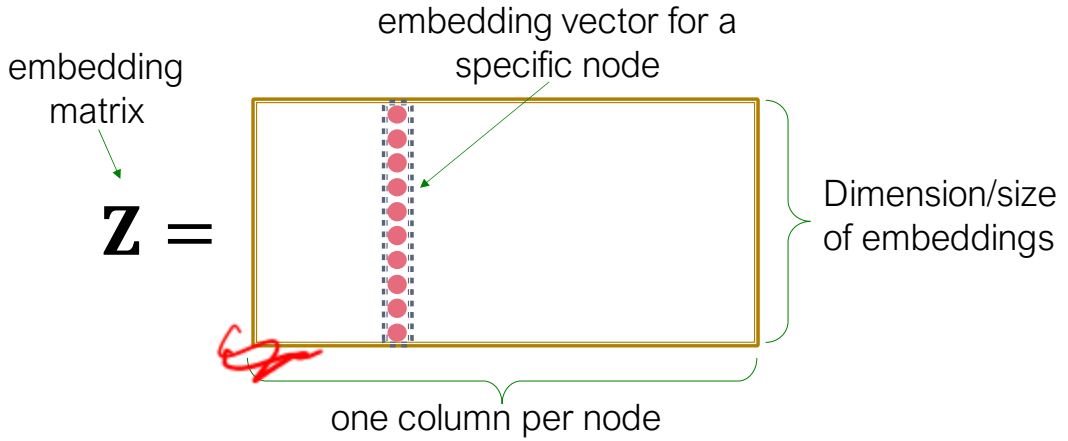
UC San Diego
HALICIOĞLU DATA SCIENCE INSTITUTE

Outline

- Graph neural networks

- Presentation
  - Yongyi Jiang, Haoyun Wang: "Visual Autoregressive Modeling: Scalable Image Generation via Next-Scale Prediction"
  - Nevasini Sasikumar, Kaiming Tao: "DeepSeek-V3 Technical Report"
  - Sarah Borsotto, John Driscoll: "Training Language Models to Generate Text with Citations via Fine-grained Rewards"
  - Sreetama Chowdhury, Chandrima Das: "A New Perspective on ADHD Research: Knowledge Graph Construction with LLMs and Network Based Insights"

Recap: Summary so far



- **Encoder + Decoder Framework**
  - Shallow encoder: embedding lookup
  - Parameters to optimize: $\mathbf{Z}$ which contains node embeddings $\mathbf{z}_u$ for all nodes $u \in V$
  - We will cover deep encoders in the GNNs

  - **Decoder:** based on node similarity.
  - **Objective:** maximize $\mathbf{z}_v^{\mathrm{T}} \mathbf{z}_u$ for node pairs (u, v) that are **similar**

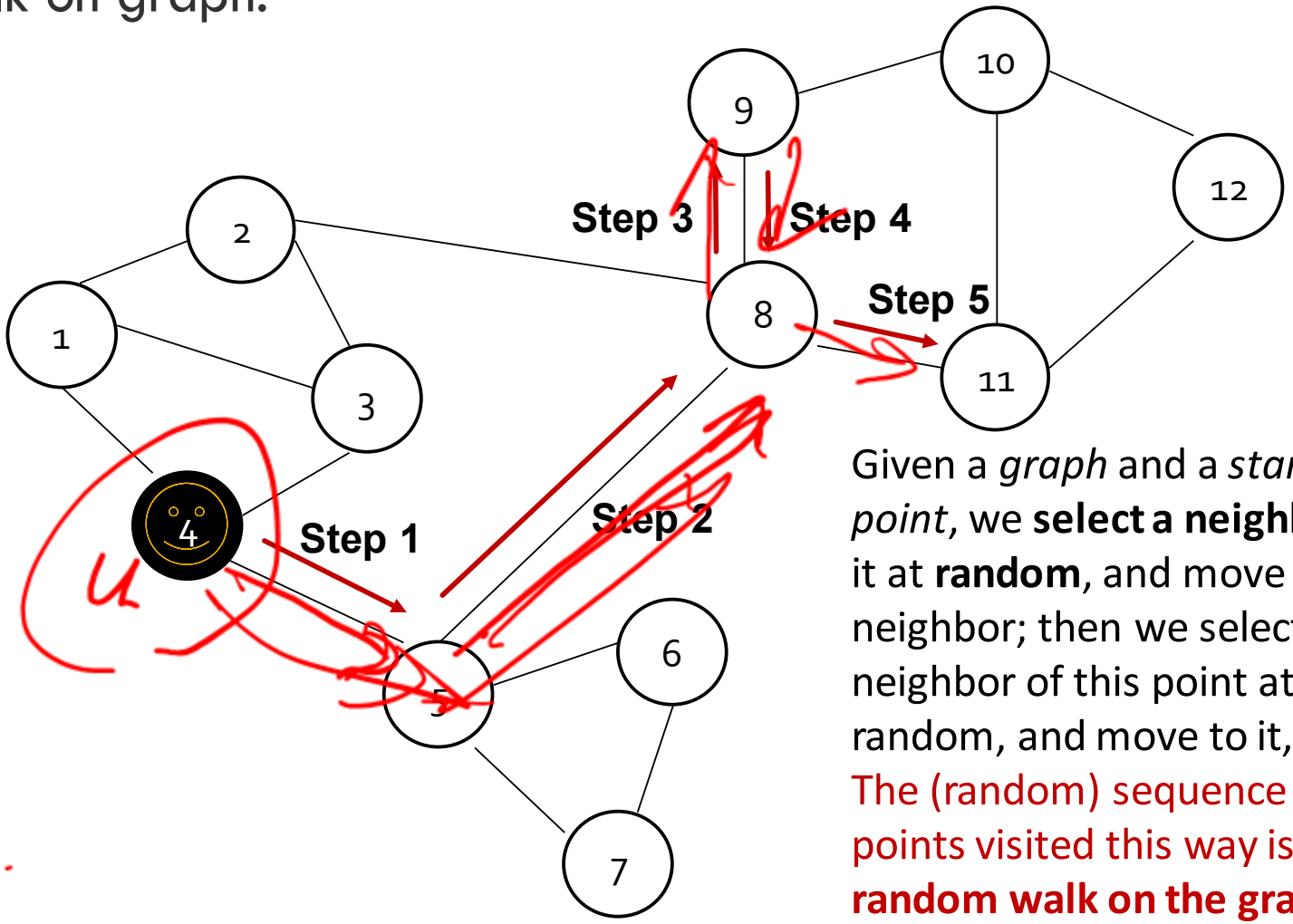
$$\text{similarity}(u, v) \approx \mathbf{z}_v^{\mathrm{T}} \mathbf{z}_u$$

Recap: How to Define Node Similarity?

- Key choice of methods is **how they define node similarity.**
- Should two nodes have a similar embedding if they…
  - are linked?
  - share neighbors?
  - have similar "structural roles"?

Similarity Function based on Random Walk

Random walk on graph:



Given a *graph* and a *starting point*, we **select a neighbor** of it at **random**, and move to this neighbor; then we select a neighbor of this point at random, and move to it, etc. The (random) sequence of points visited this way is a **random walk on the graph**.

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

Similarity Function based on Random Walk

$$\mathbf{z}_u^{\mathrm{T}} \mathbf{z}_v \approx$$

probability that **u** and **v** co-occur on a random walk over the graph

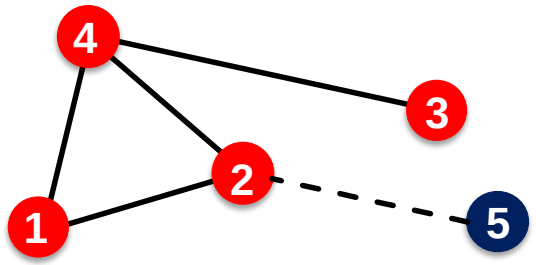
Why Random Walk?

1. **Expressivity:** Flexible stochastic definition of node similarity that incorporates both local and higher-order neighborhood information
   Idea: if random walk starting from node u visits v with high probability, u and v are similar (high-order multi-hop information)

2. **Efficiency:** Do not need to consider all node pairs when training; only need to consider pairs that co-occur on random walks

Limitations of Random Walk Embedding (1)

- Cannot obtain embeddings for nodes not in the training set



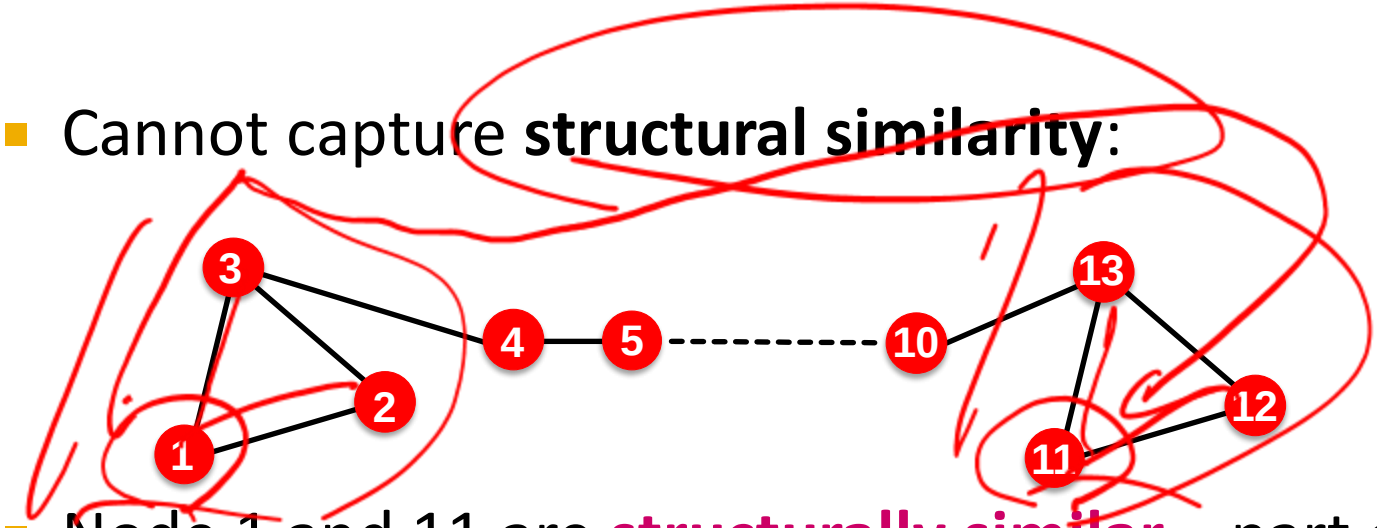
Training set

A newly added node 5 at test time (e.g., new user in a social network)

Cannot compute its embedding with DeepWalk / node2vec. Need to recompute all node embeddings.

Limitations of Random Walk Embedding (2)

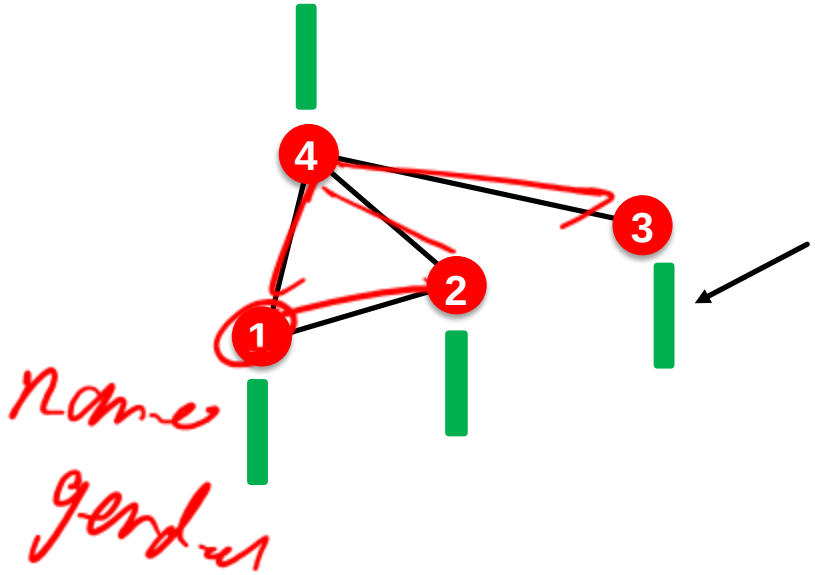
- Cannot capture **structural similarity**:



- Node 1 and 11 are **structurally similar** – part of one triangle, degree 2, …
- However, they have very **different** embeddings.
  - It's unlikely that a random walk will reach node 11 from node 1.

Limitations of Random Walk Embedding (3)

- Cannot utilize node, edge and graph features

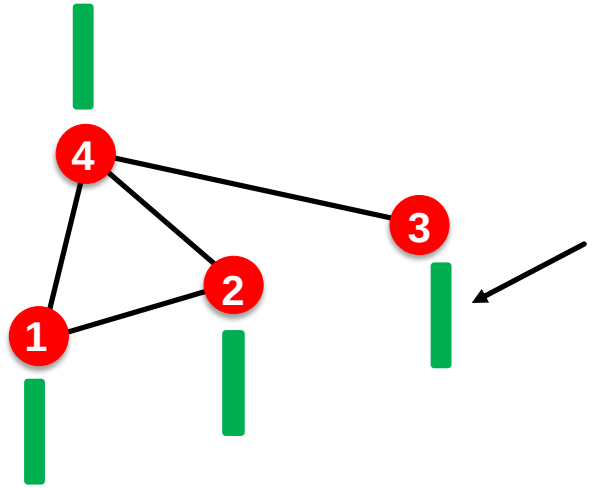


Feature vector
(e.g. protein properties in a protein-protein interaction graph)

DeepWalk / node2vec embeddings do not incorporate such node features

Limitations of Random Walk Embedding (3)

- Cannot utilize node, edge and graph features



Feature vector
(e.g. protein properties in a protein-protein interaction graph)

DeepWalk / node2vec embeddings do not incorporate such node features

Solution to these limitations: Deep Representation Learning and Graph Neural Networks

Summary

- **Encoder + Decoder Framework**
  - Shallow encoder: embedding lookup
  - Parameters to optimize: $\mathbf{Z}$ which contains node embeddings $\mathbf{z}_u$ for all nodes $u \in V$
  - We will cover deep encoders in the GNNs

  - **Decoder:** based on node similarity.
  - **Objective:** maximize $\mathbf{z}_v^{\mathrm{T}} \mathbf{z}_u$ for node pairs (u, v) that are **similar**

random walk

Graph Neural Networks (GNNs)

Slides adapted from:

- Jure Leskovec, Stanford CS224W: Machine Learning with Graphs

Deep Graph Encoders

- Encoding based on graph neural networks

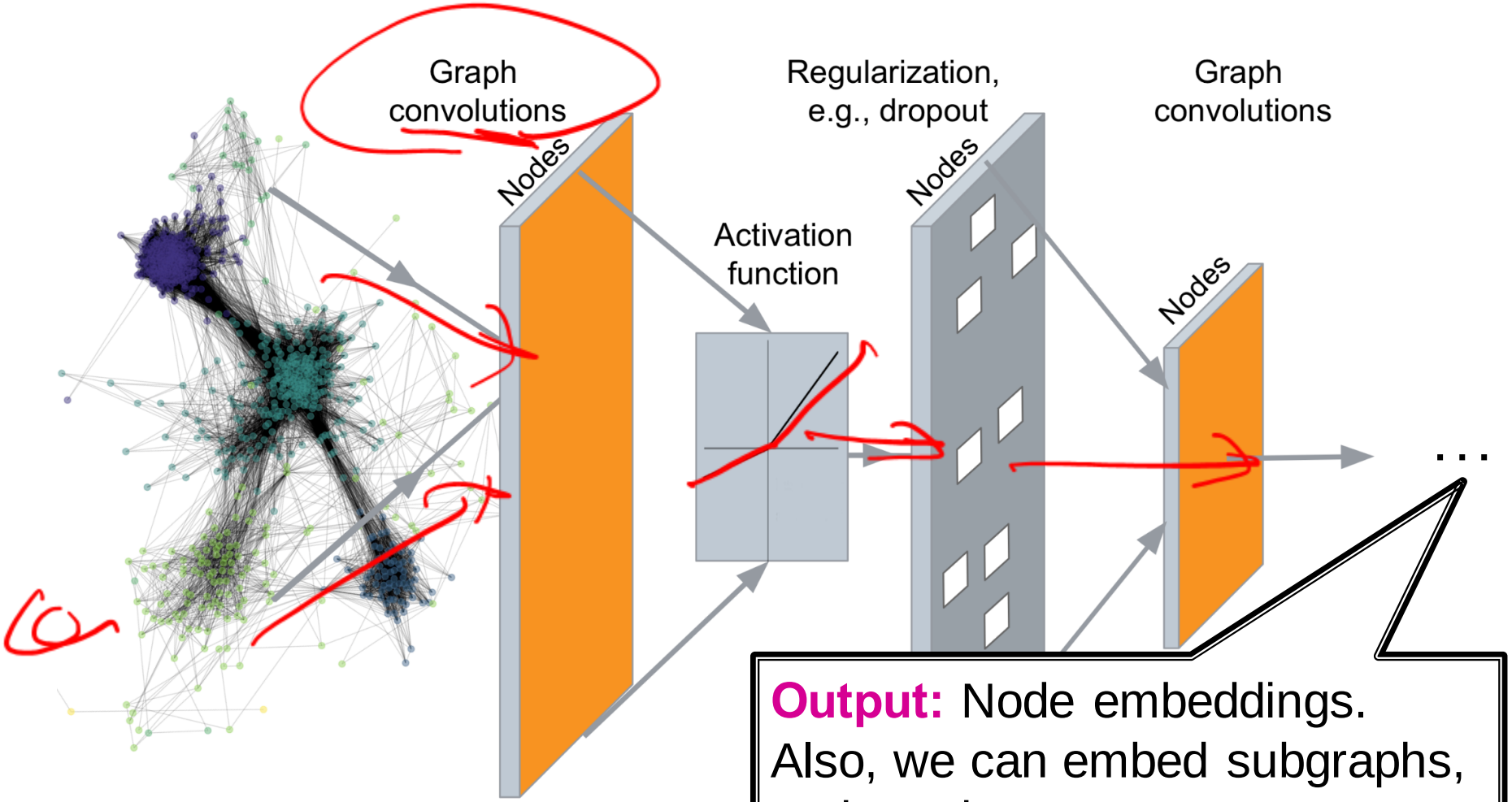
$$\text{ENC}(v) = \quad \text{multiple layers of non-linear transformations based on graph structure}$$

v.s. **Shallow Encoder:**

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

word2vec → BERT

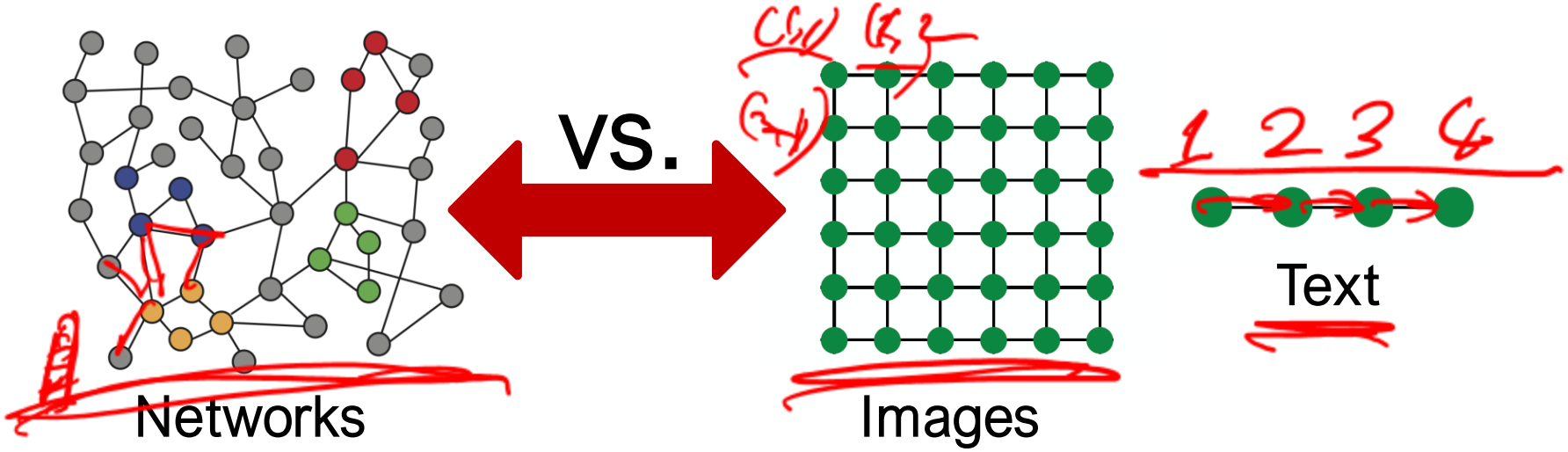
Deep Graph Encoders



Output: Node embeddings.
Also, we can embed subgraphs, and graphs

Graphs are more complex than images / text

- Arbitrary size and complex topological structure (i.e., no spatial locality like grids)



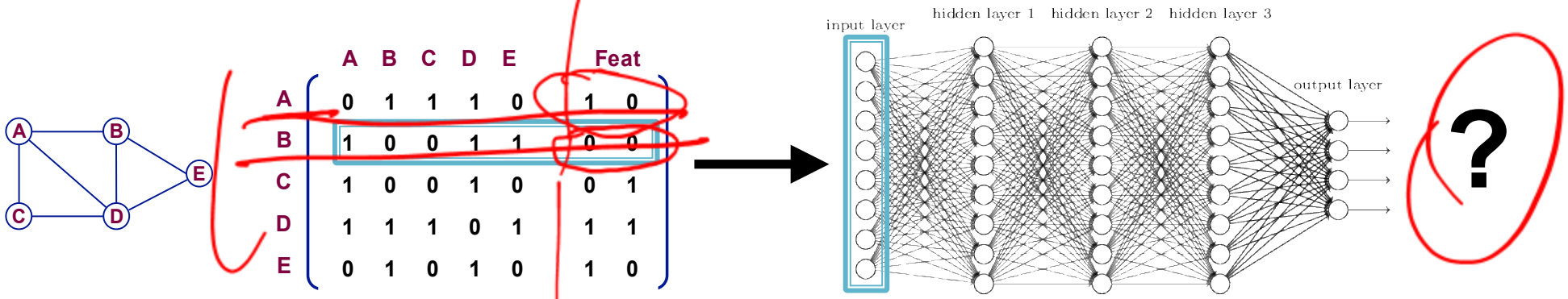
- No fixed node ordering or reference point
- Often dynamic and have multimodal features

Graph Neural Networks: Setup

- **Assume we have a graph G:**
  - V is the **vertex set**
  - $\boldsymbol{A}$ is the **adjacency matrix** (assume binary)
  - $\boldsymbol{X} \in \mathbb{R}^{|V| \times d}$ is a matrix of **node features**
  - v: a node in V; $N(v)$: the 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]

A Naïve Approach

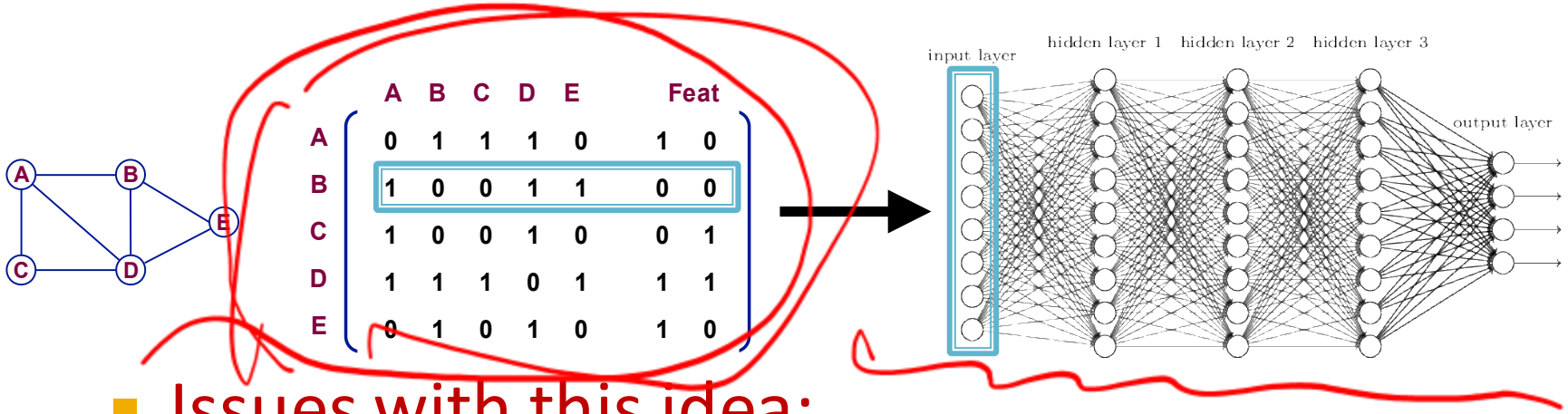
- Join adjacency matrix and features
- Feed them into a deep neural net:



- Issues with this idea:

A Naïve Approach

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

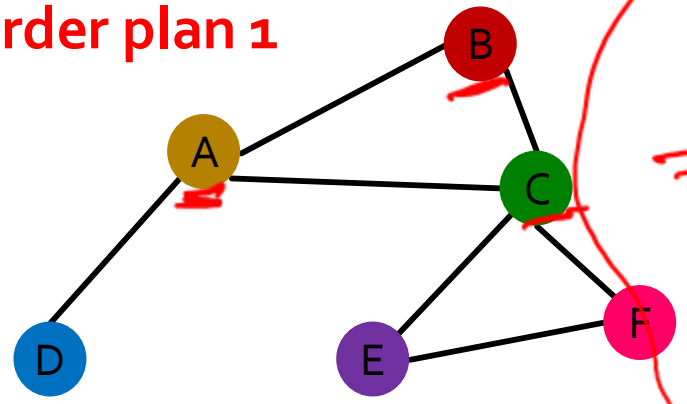


- Issues with this idea:
  - $O(|V|)$ parameters
  - Not applicable to graphs of different sizes
  - Sensitive to node ordering

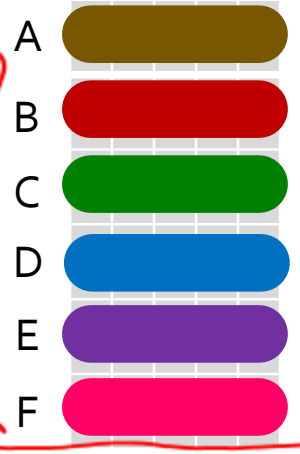
Permutation Invariance

- **Graph does not have a canonical order of the nodes!**

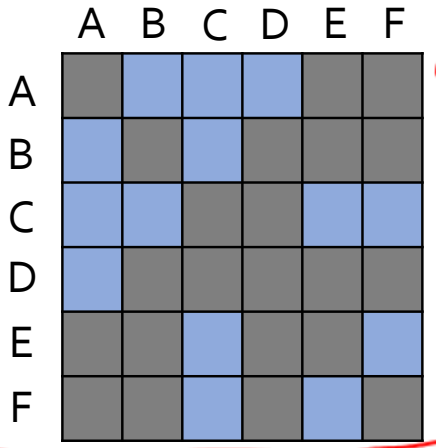
Order plan 1



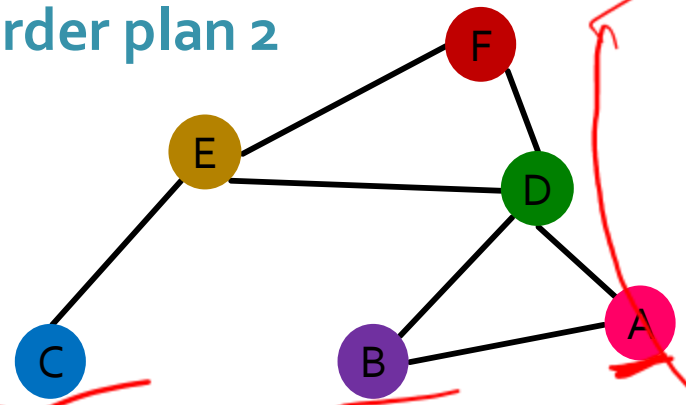
Node features X_1



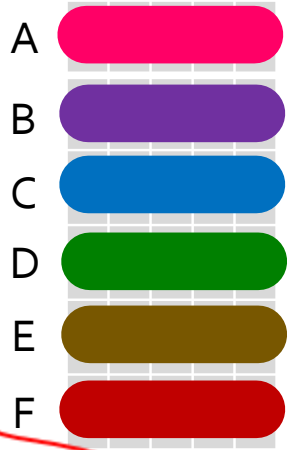
Adjacency matrix A_1



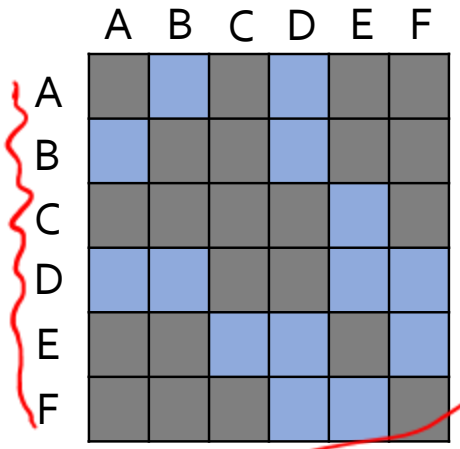
Order plan 2



Node features X_2

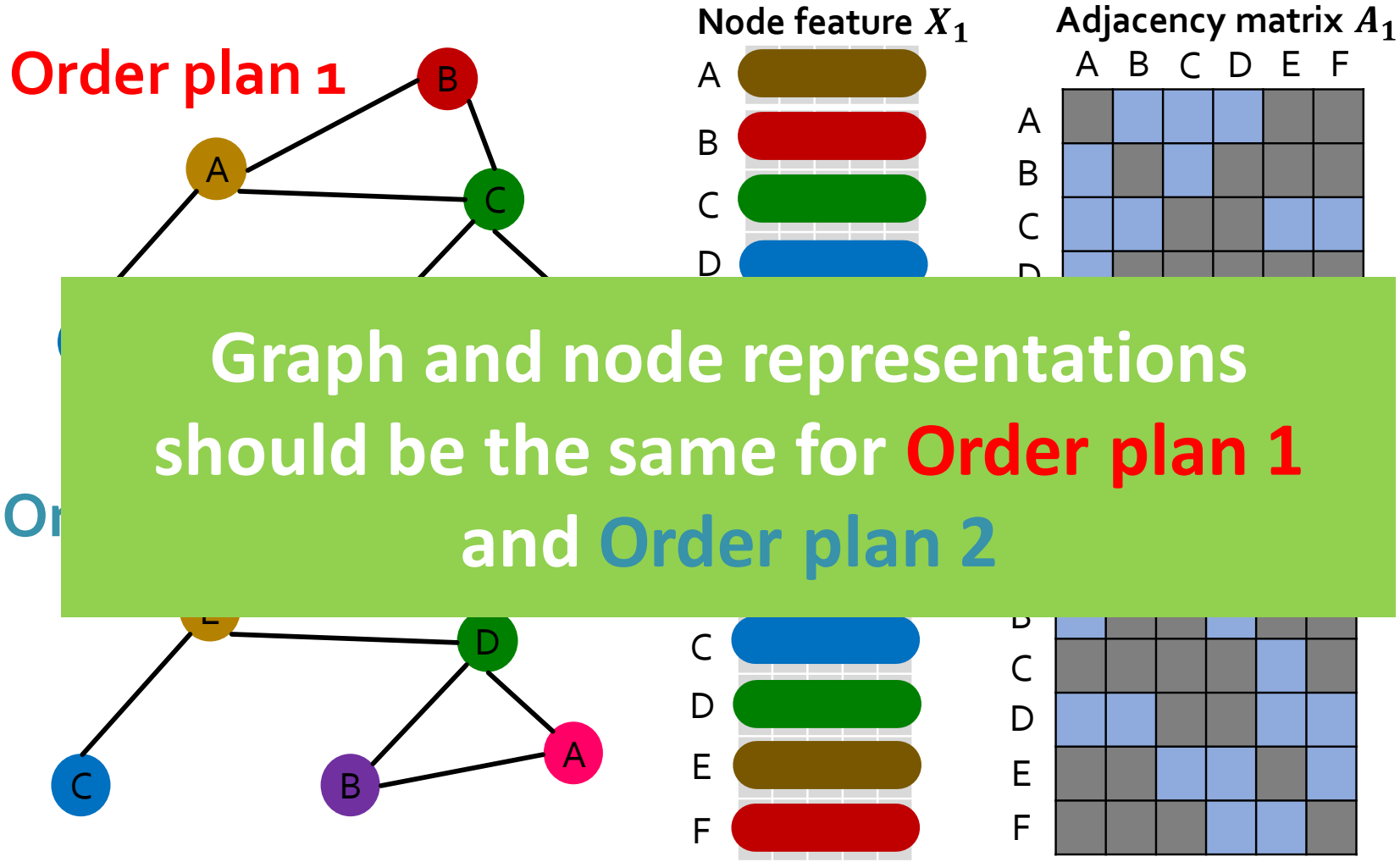


Adjacency matrix A_2



Permutation Invariance

- **Graph does not have a canonical order of the nodes!**



Permutation Invariance

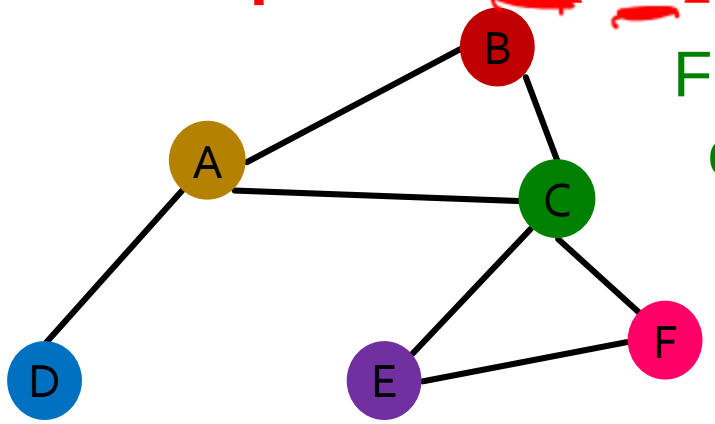
What does it mean by "graph representation is same for two order plans"?

- Consider we learn a function f that maps a graph $G = (\boldsymbol{A}, \boldsymbol{X})$ to a vector $\mathbb{R}^d$ then

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

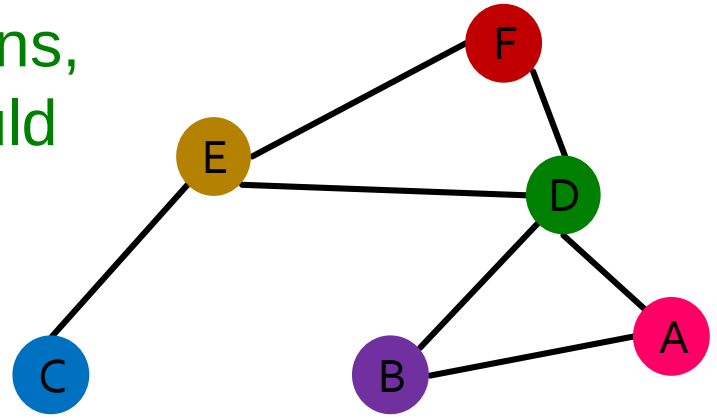
$\boldsymbol{A}$ is the adjacency matrix
$\boldsymbol{X}$ is the node feature matrix

Order plan 1: $\boldsymbol{A}_1, \boldsymbol{X}_1$



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

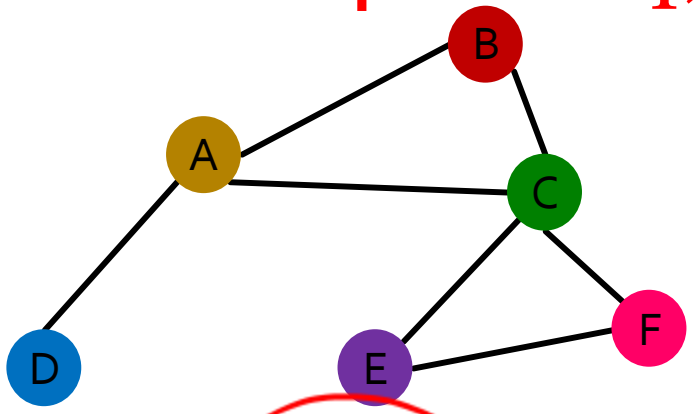
Order plan 2: $\boldsymbol{A}_2, \boldsymbol{X}_2$



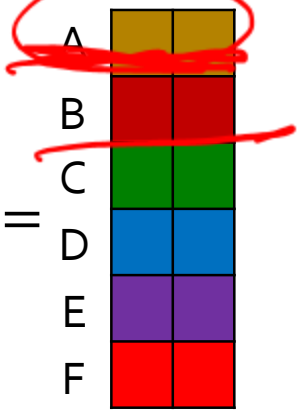
Permutation Equivariance

For node representation: We learn a function f that maps nodes of G to a matrix $\mathbb{R}^{m \times d}$.

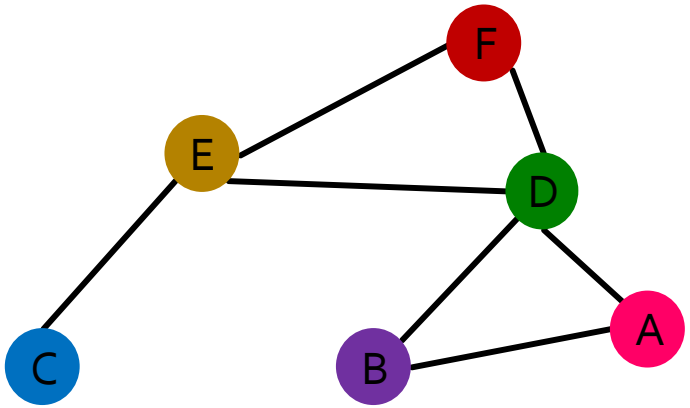
Order plan 1: A_1, X_1



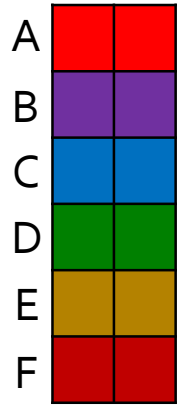
$$f(A_1, X_1) =$$



Order plan 2: A_2, X_2



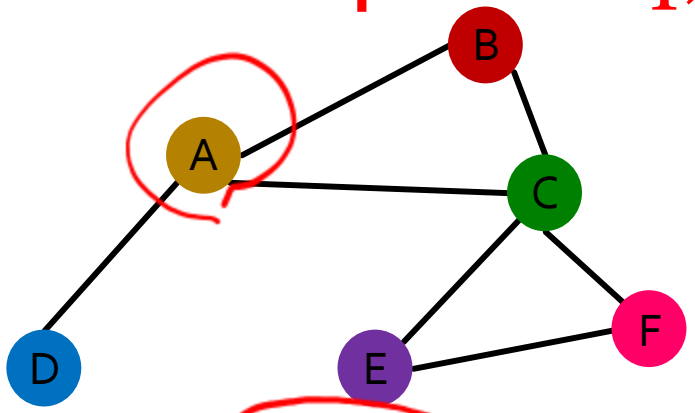
$$f(A_2, X_2) =$$



Permutation Equivariance

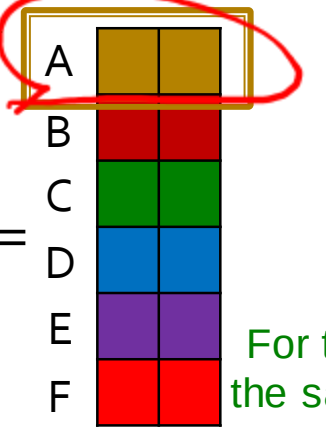
For node representation: We learn a function f that maps nodes of G to a matrix $\mathbb{R}^{m \times d}$.

Order plan 1: A_1, X_1

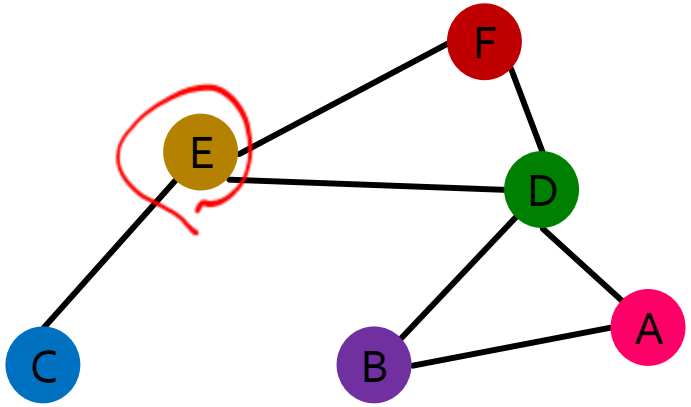


Representation vector of the brown node A

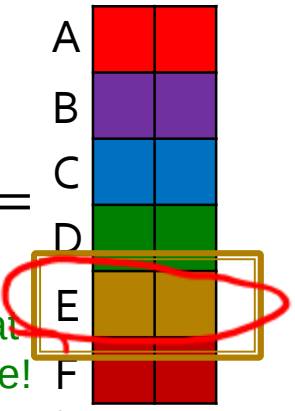
$$f(A_1, X_1) =$$



Order plan 2: A_2, X_2



$$f(A_2, X_2) =$$



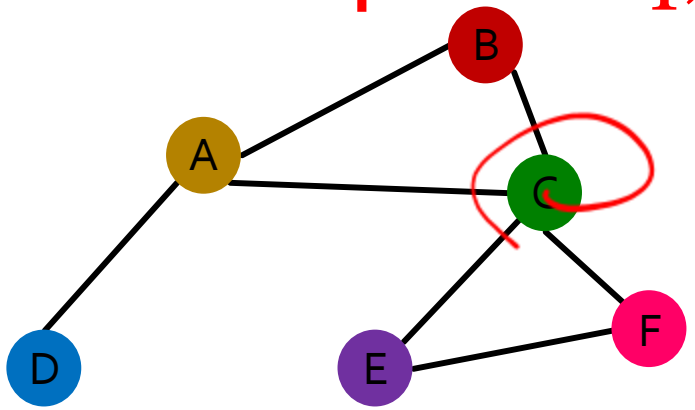
Representation vector of the brown node E

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

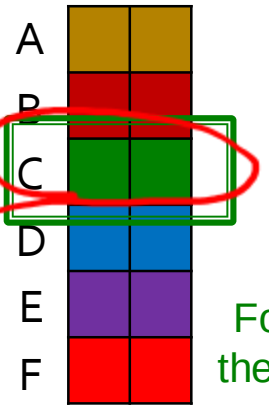
Permutation Equivariance

For node representation: We learn a function f that maps nodes of G to a matrix $\mathbb{R}^{m \times d}$.

Order plan 1: A_1, X_1

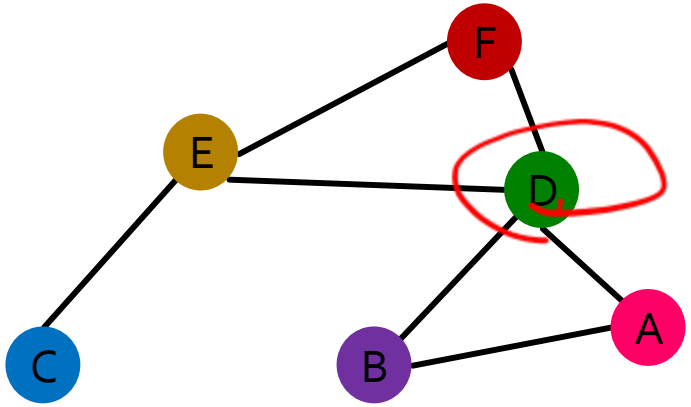


$$f(A_1, X_1) =$$

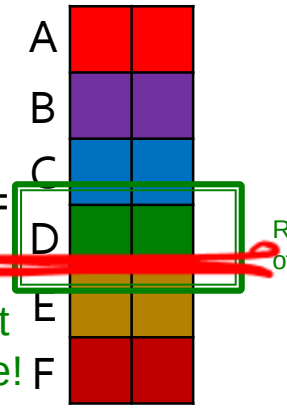


Representation vector of the green node C

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

Summary: Permutation Invariance and Equivariance

- **Permutation-invariant**

$$f(A, X) = f(PAP^T, PX)$$

Permute the input, the output stays the same. (map a graph to a vector)

- **Permutation-equivariant**

$$Pf(A, X) = f(PAP^T, PX)$$

Permute the input, output also permutes accordingly. (map a graph to a matrix)

Summary: Permutation Invariance and Equivariance

- **Permutation-invariant**

$$f(A, X) = f(PAP^T, PX)$$

Permute the input, the output stays the same. (map a graph to a vector)

- **Permutation-equivariant**

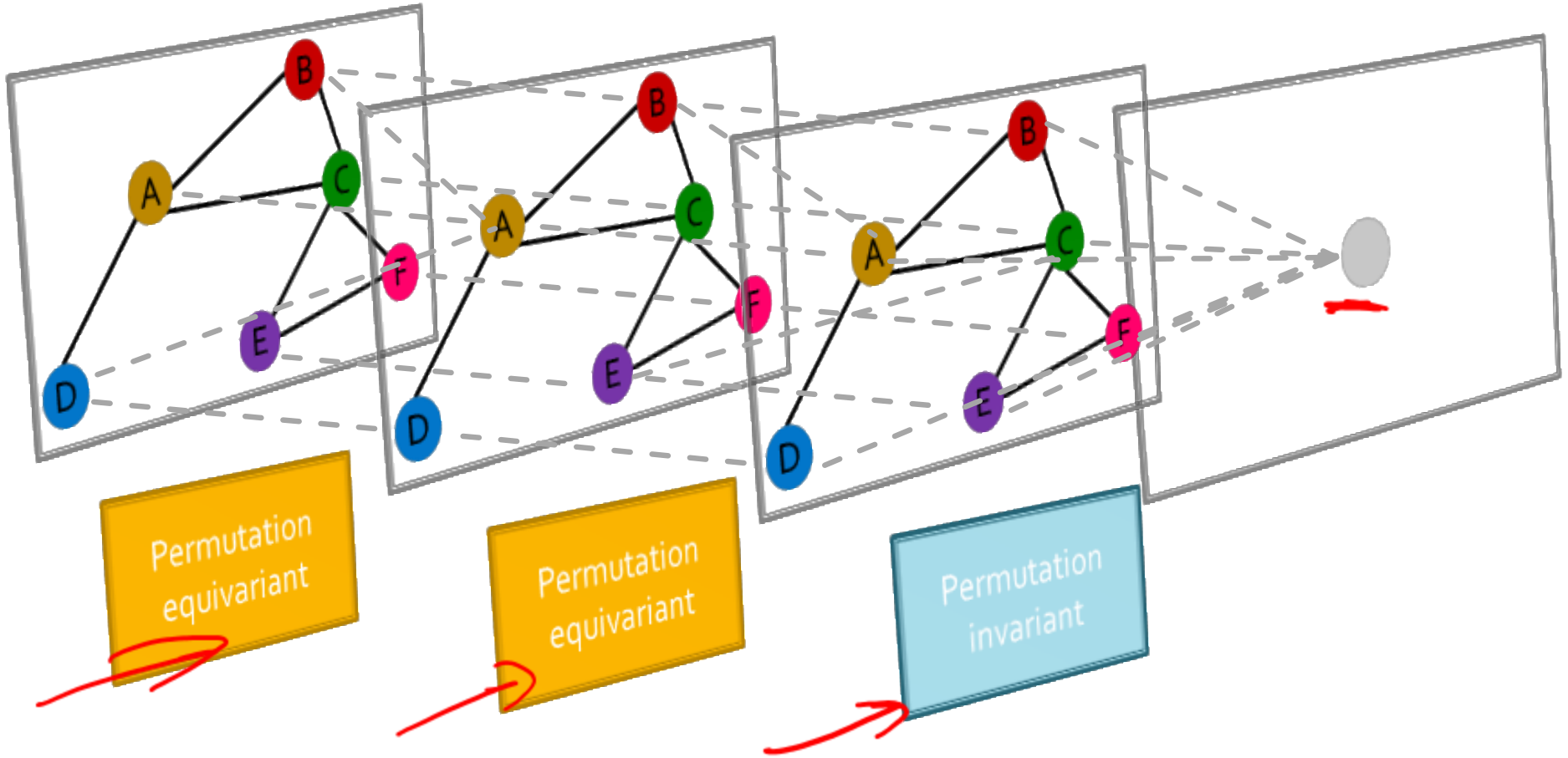
$$Pf(A, X) = f(PAP^T, PX)$$

Permute the input, output also permutes accordingly. (map a graph to a matrix)

- **Examples:**
  - $f(A, X) = 1^T X$: Permutation-**invariant**
    - Reason: $f(PAP^T, PX) = 1^T PX = 1^T X = f(A, X)$
  - $f(A, X) = X$: Permutation-**equivariant**
    - Reason: $f(PAP^T, PX) = PX = Pf(A, X)$
  - $f(A, X) = AX$: Permutation-**equivariant**
    - Reason: $f(PAP^T, PX) = PAP^T PX = PAX = Pf(A, X)$

Graph Neural Networks Overview

- GNNs consist of multiple permutation equivariant / invariant functions



[Bronstein, ICLR 2021 keynote]

Graph Neural Networks Overview

- GNNs consist of multiple permutation equivariant / invariant functions

Are other neural network architectures, e.g., MLPs, permutation invariant / equivariant?

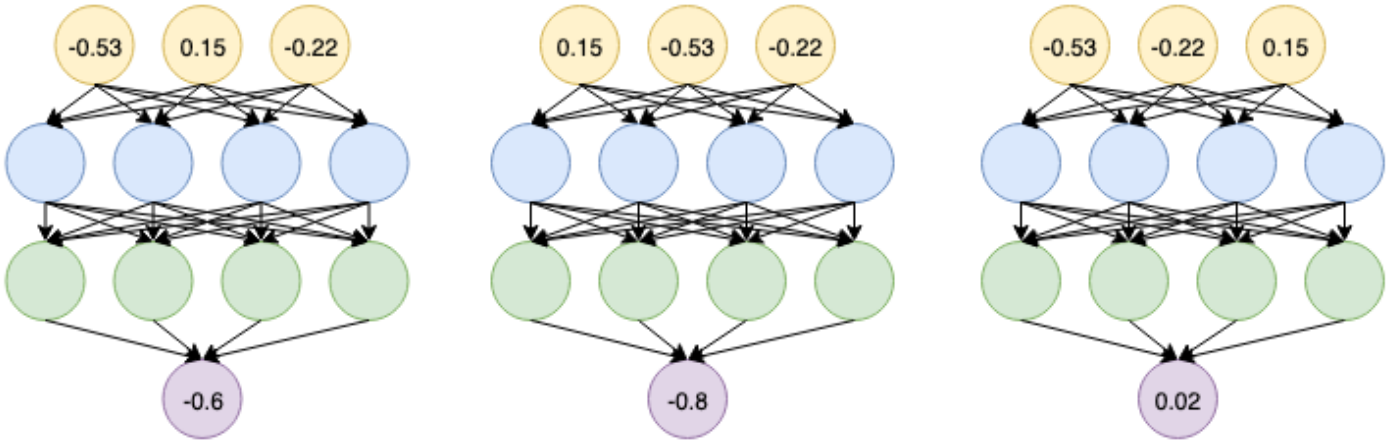
Graph Neural Networks Overview

- GNNs consist of multiple permutation equivariant / invariant functions

Are other neural network architectures, e.g., MLPs, permutation invariant / equivariant?

- **No.**

Switching the order of the input leads to different outputs!

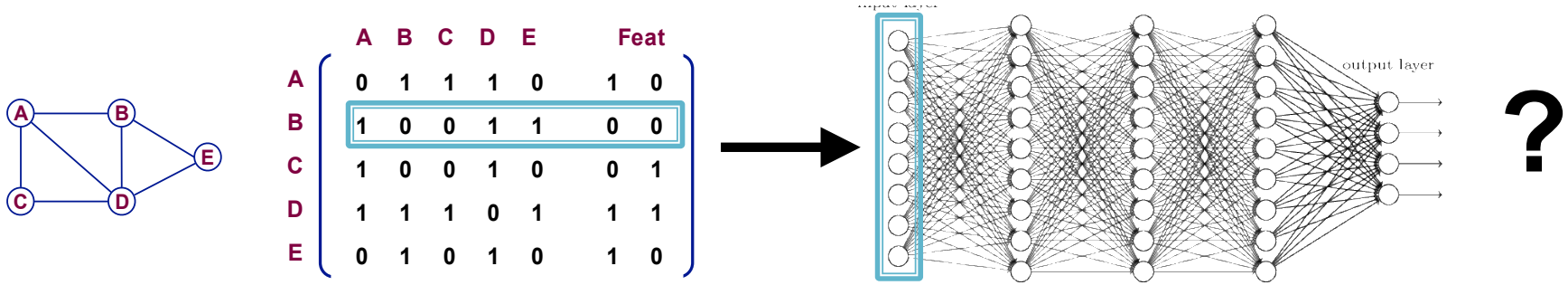


Graph Neural Networks Overview

- GNNs consist of multiple permutation equivariant / invariant functions

Are other neural network architectures, e.g., MLPs, permutation invariant / equivariant?

- **No.**



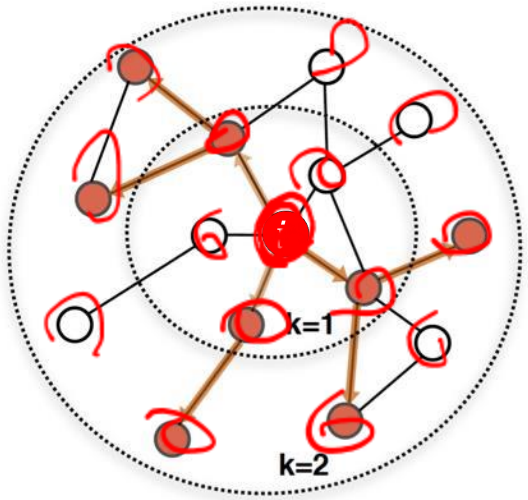
This explains why **the naïve MLP approach fails for graphs**!

Graph Neural Networks Overview

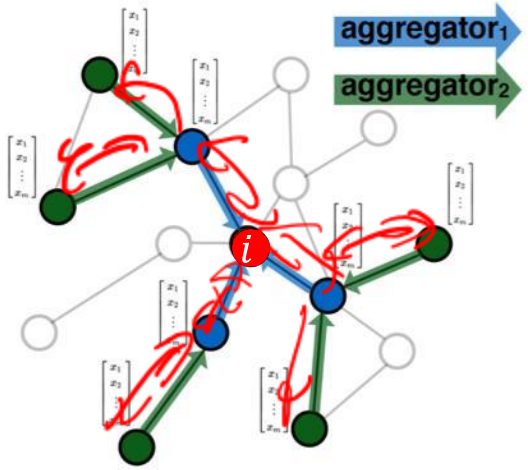
- GNNs consist of multiple permutation equivariant / invariant functions
- Next: Design GNNs that are permutation equivariant / invariant by **passing and aggregating information from neighbors**

Graph Convolutional Networks

Idea: Node's neighborhood defines a computation graph



Determine node computation graph

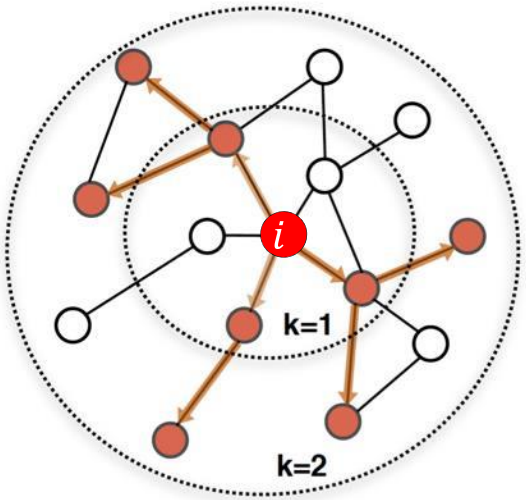


Propagate and transform information

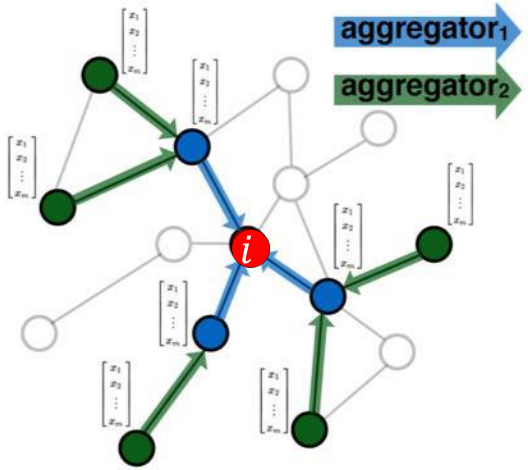
[Kipf and Welling, ICLR 2017]

Graph Convolutional Networks

Idea: Node's neighborhood defines a computation graph



Determine node computation graph



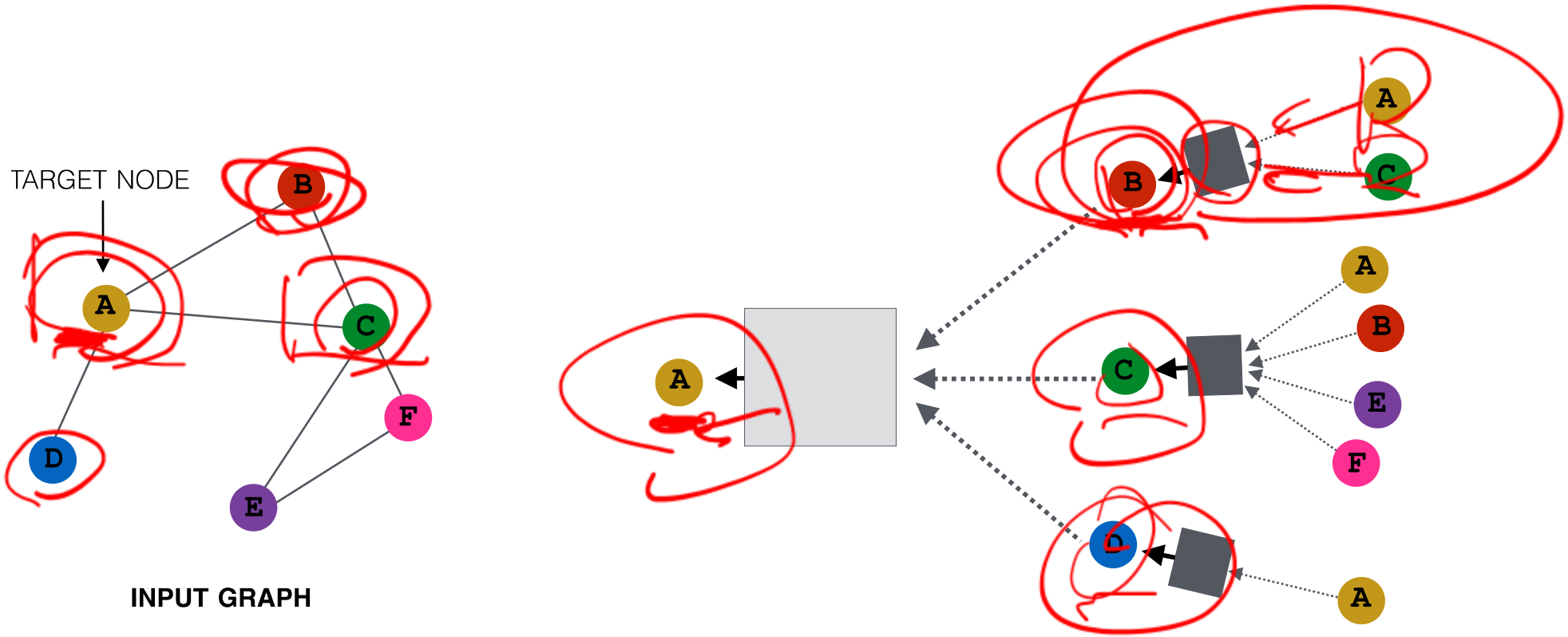
Propagate and transform information

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

[Kipf and Welling, ICLR 2017]

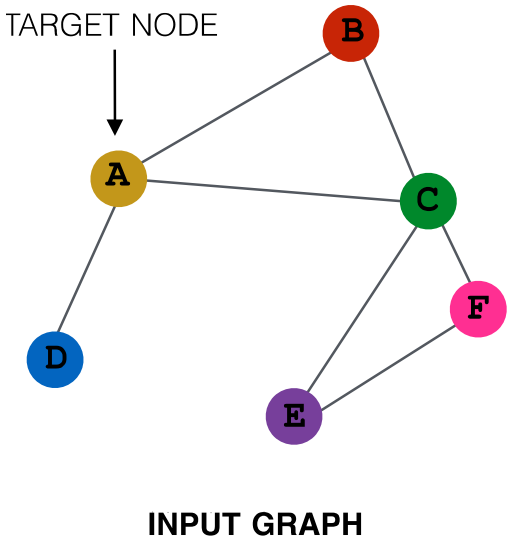
Idea: Aggregate Neighbors

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

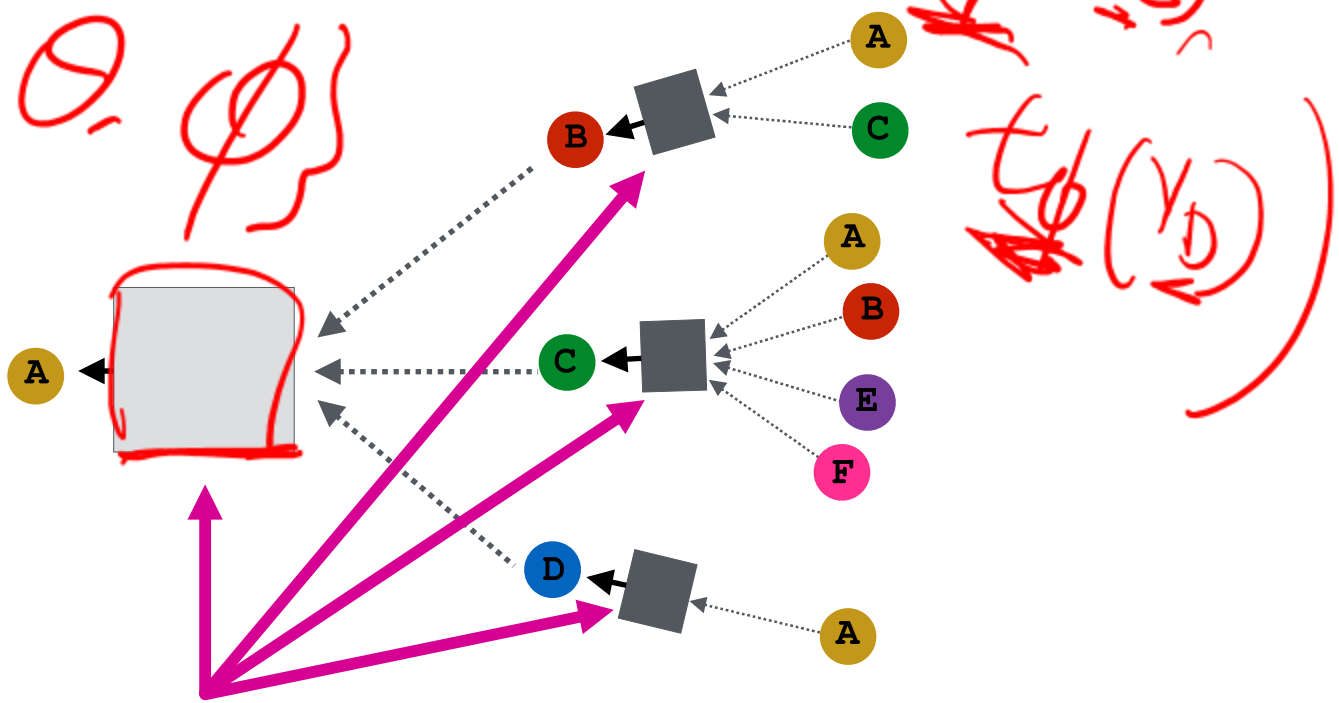


Idea: Aggregate Neighbors

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



INPUT GRAPH

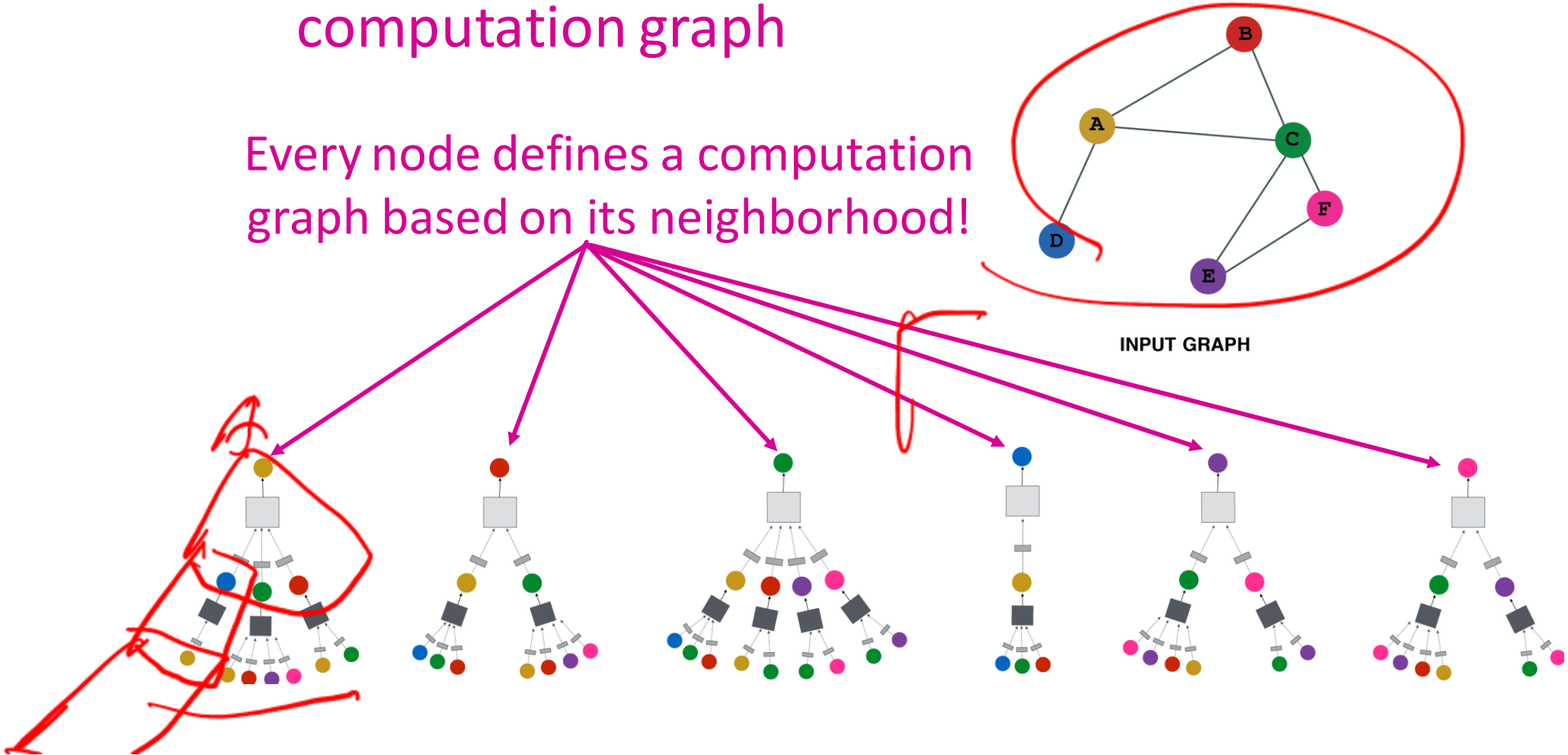


Neural networks

Idea: Aggregate Neighbors

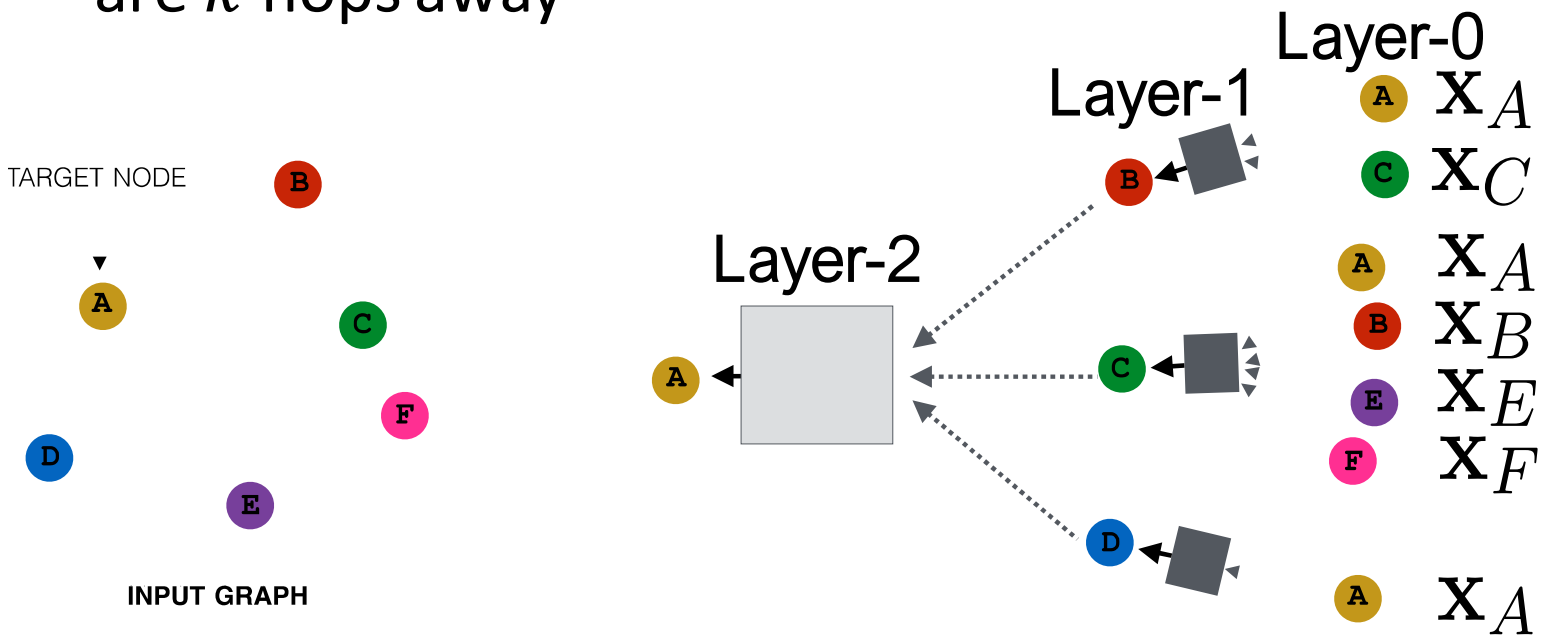
- **Intuition:** Network neighborhood defines a computation graph

Every node defines a computation graph based on its neighborhood!



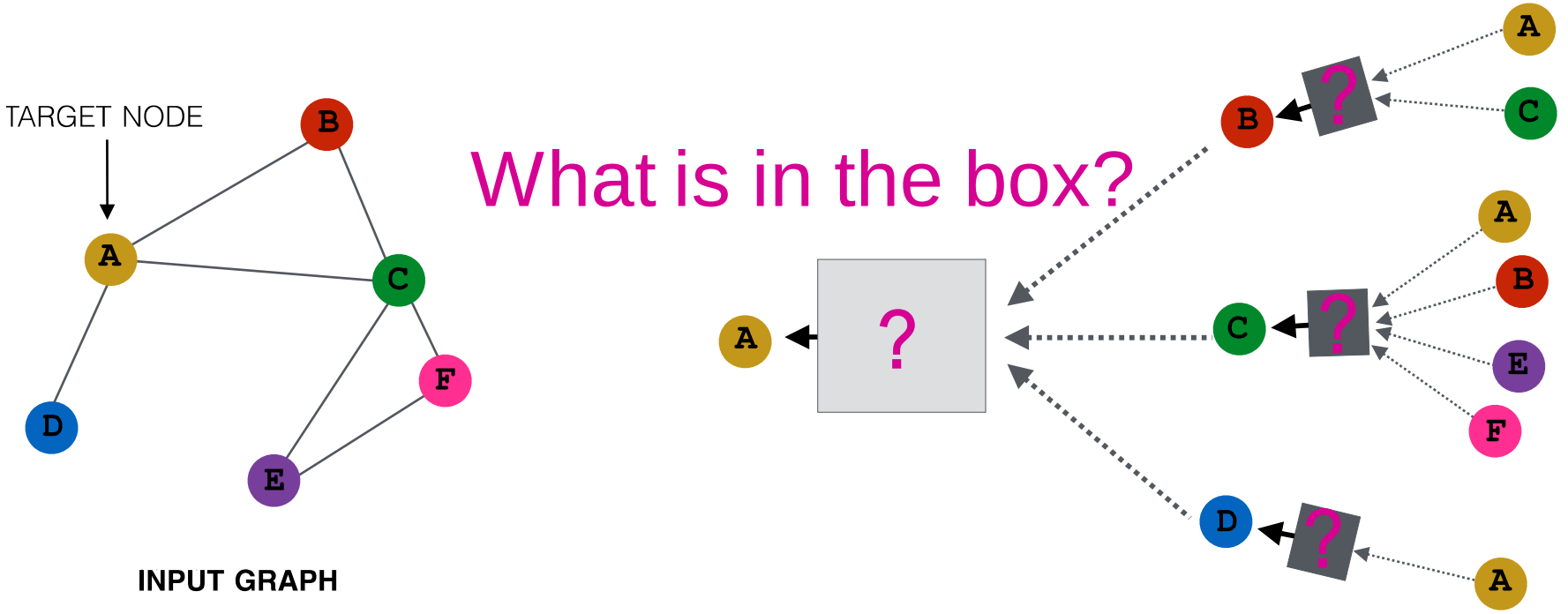
Deep Model: Many Layers

- 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



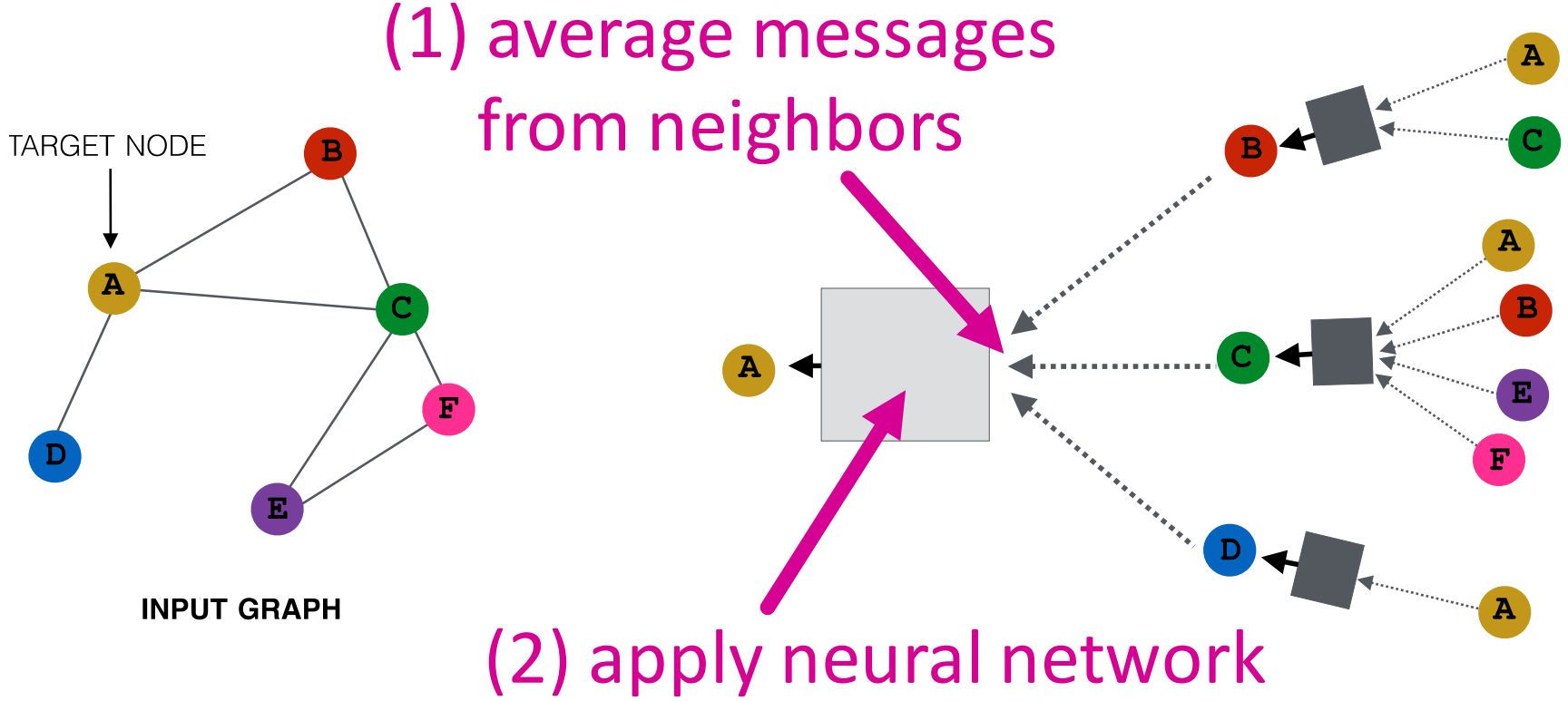
Neighborhood Aggregation

- **Neighborhood aggregation:** Key distinctions are in how different approaches aggregate information across the layers



Neighborhood Aggregation

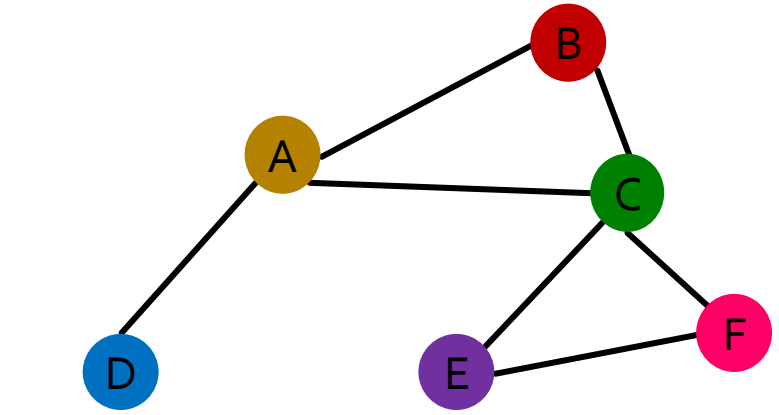
- **Basic approach:** Average information from neighbors and apply a neural network



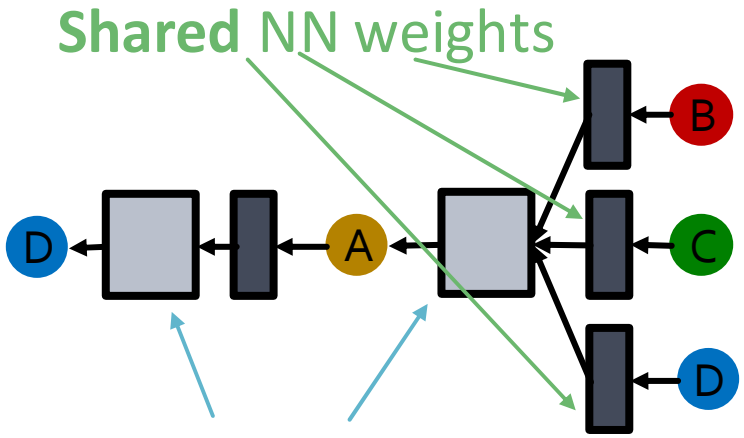
GCN (Graph Convolutional Net): Invariance and Equivariance

What are the invariance and equivariance properties for a GCN?

- **Given a node**, the GCN that computes its embedding is **permutation invariant**



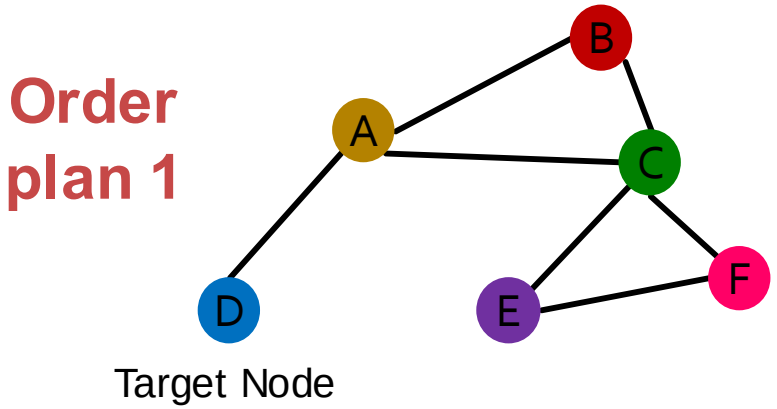
Target Node

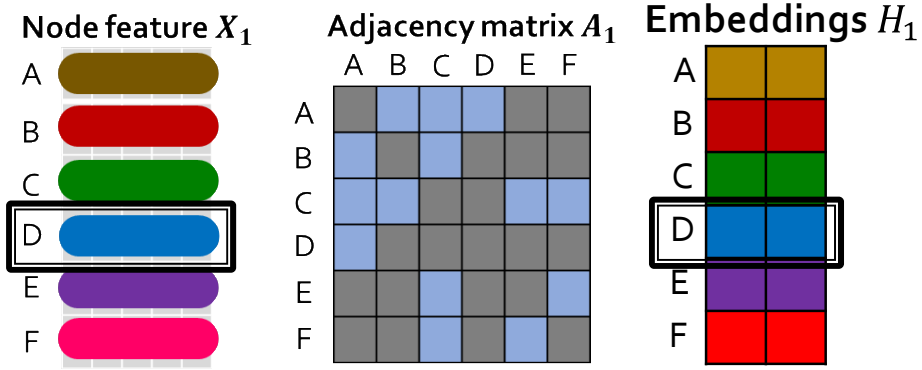


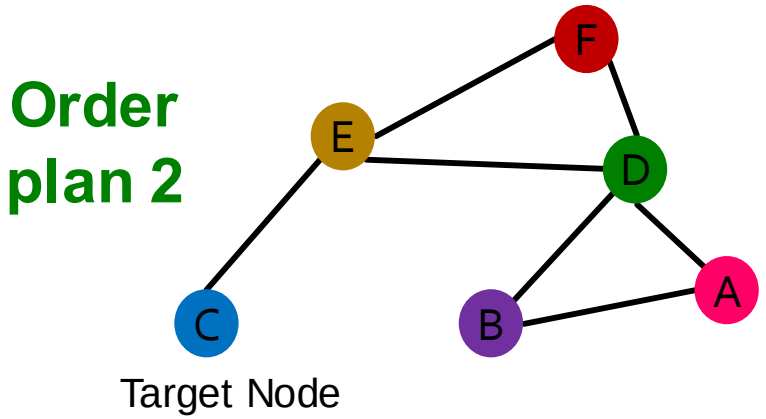
Average of neighbor's previous layer embeddings - **Permutation invariant**

GCN: Invariance and Equivariance

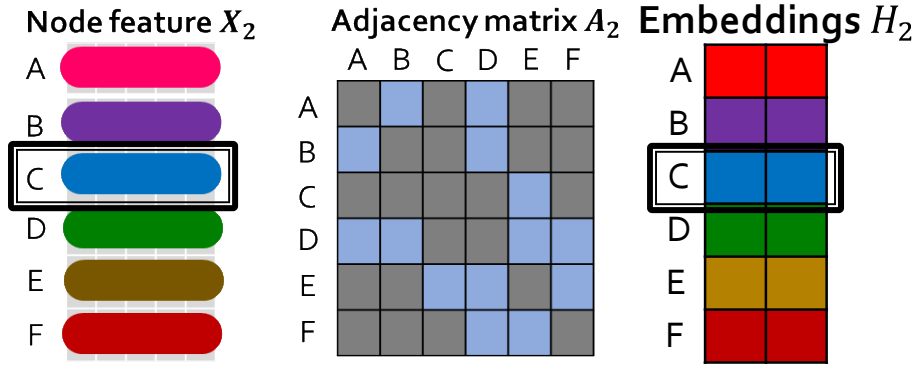
- **Considering all nodes in a graph**, GCN computation is **permutation equivariant**







Permute the input, the output also permutes accordingly - permutation equivariant



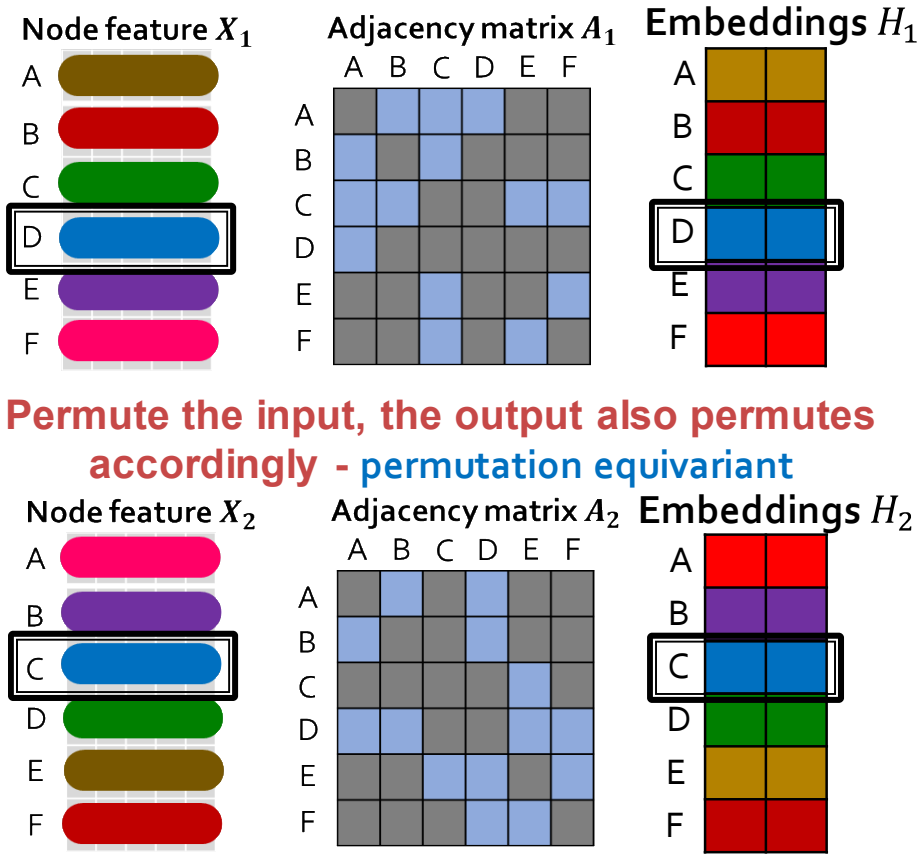
GCN: Invariance and Equivariance

- **Considering all nodes in a graph**, GCN computation is **permutation equivariant**

Detailed reasoning:

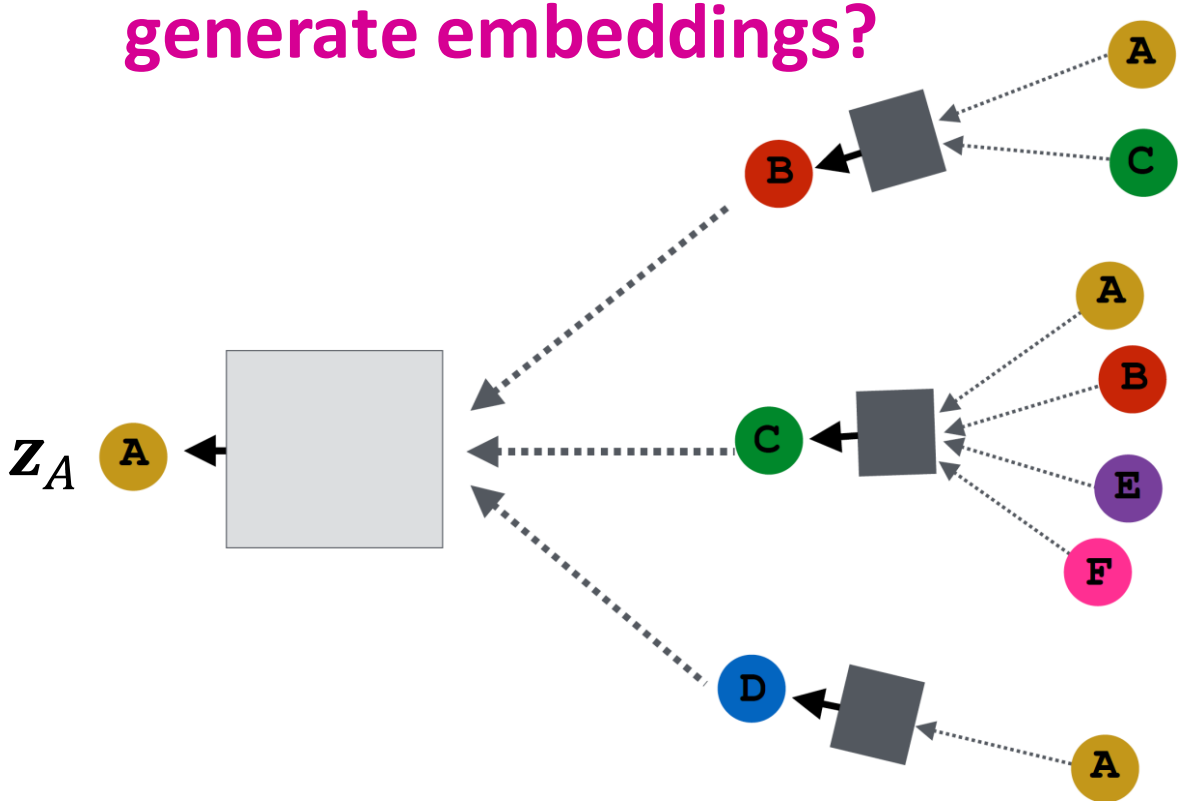
1. The rows of **input node features** and **output embeddings** are **aligned**
2. We know computing the embedding of **a given node** with GCN is **invariant.**
3. So, after permutation, the **location** of **a given node** in the **input node feature** matrix is changed, and the **the output embedding of a given node stays the same** (the colors of node feature and embedding are **matched**)

This is permutation equivariant



How to Train A GNN

How do we train the GCN to generate embeddings?



Need to define a loss function on the embeddings.

How to Train A GNN

- Node embedding $\boldsymbol{z}_v$ is a function of input graph
- **Supervised setting**: we want to minimize the loss $\mathcal{L}$ (see also Slide 15):

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

  - $\boldsymbol{y}$: node label
  - $\mathcal{L}$ could be L2 if $\boldsymbol{y}$ is real number, or cross entropy if $\boldsymbol{y}$ is categorical

How to Train A GNN

- Node embedding $\boldsymbol{z}_v$ is a function of input graph
- **Supervised setting**: we want to minimize the loss $\mathcal{L}$ (see also Slide 15):

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

  - $\boldsymbol{y}$: 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!**

How to Train A GNN

- Node embedding $\boldsymbol{z}_v$ is a function of input graph
- **Supervised setting**: we want to minimize the loss $\mathcal{L}$ (see also Slide 15):

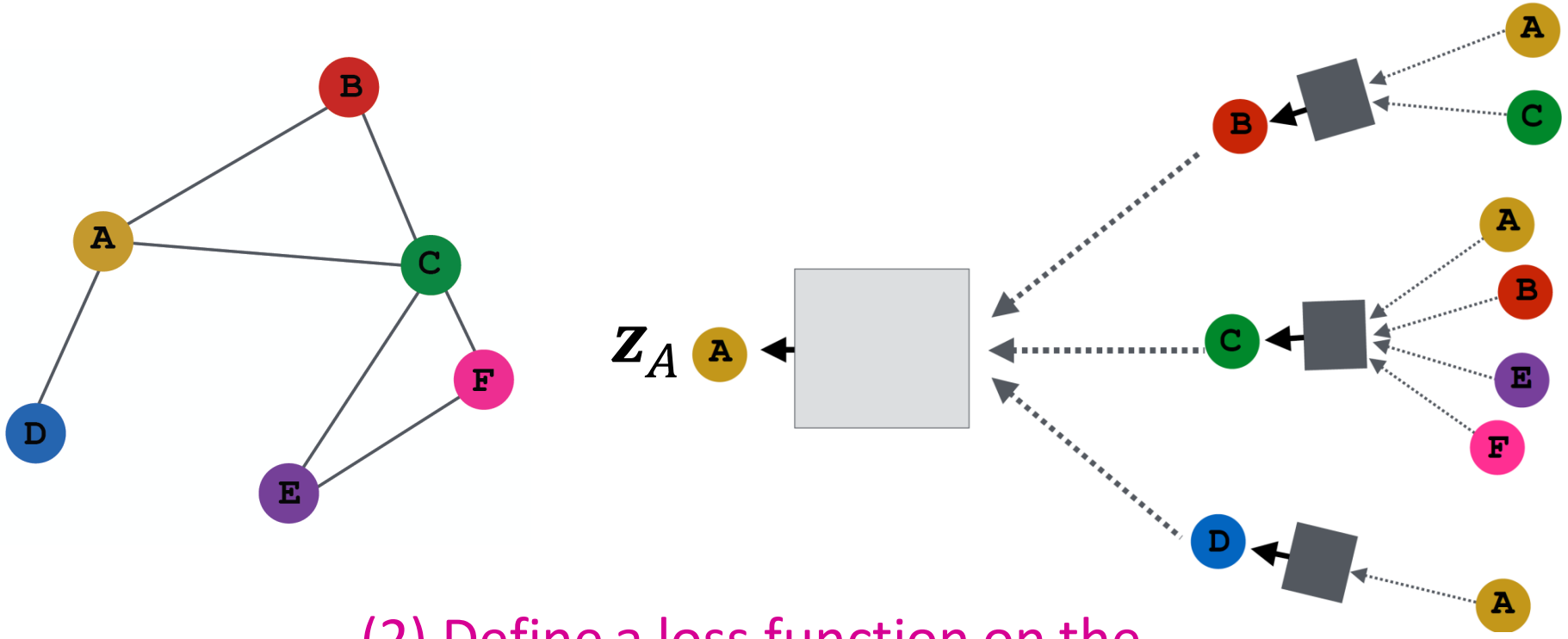
$$\min_{\Theta} \mathcal{L}(\boldsymbol{y}, f(\boldsymbol{z}_v))$$

  - $\boldsymbol{y}$: 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!**

"Similar" nodes have similar embeddings (discussed in last lecture)

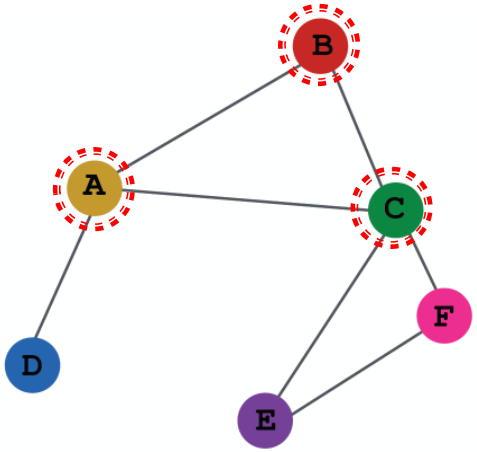
Model Design: Overview

(1) Define a neighborhood aggregation function



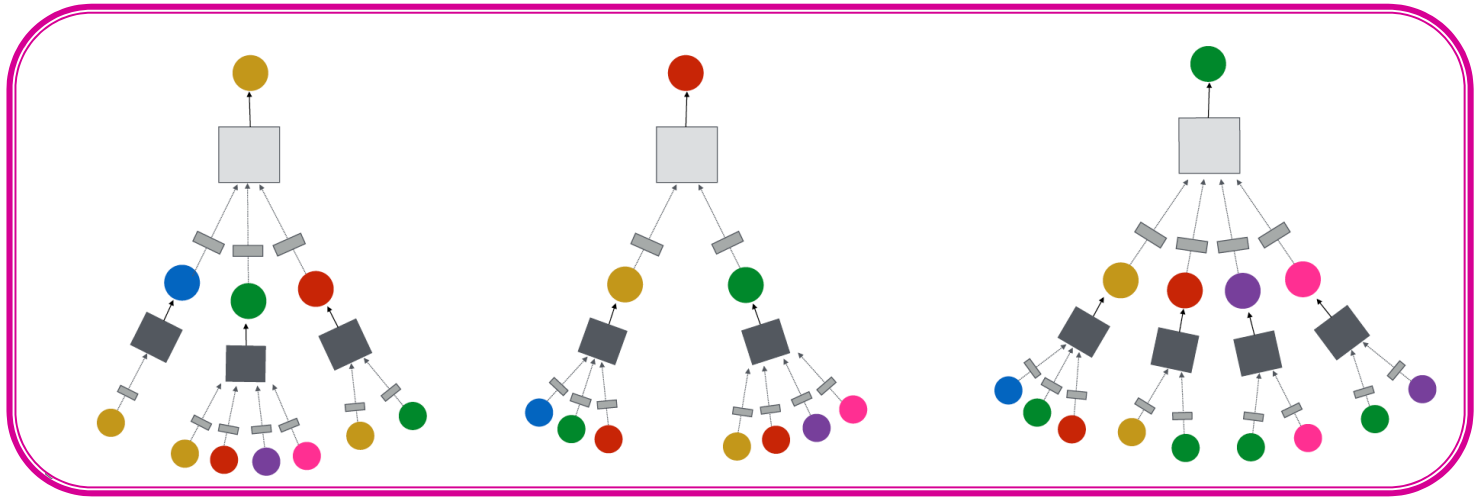
(2) Define a loss function on the embeddings

Model Design: Overview

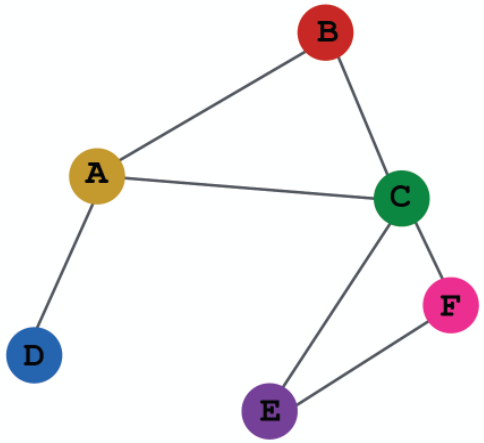


INPUT GRAPH

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



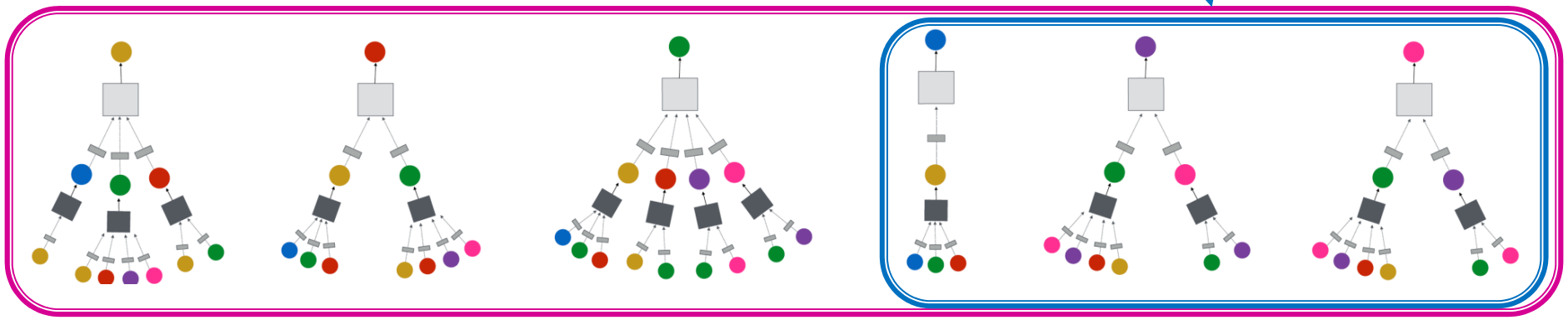
Model Design: Overview



INPUT GRAPH

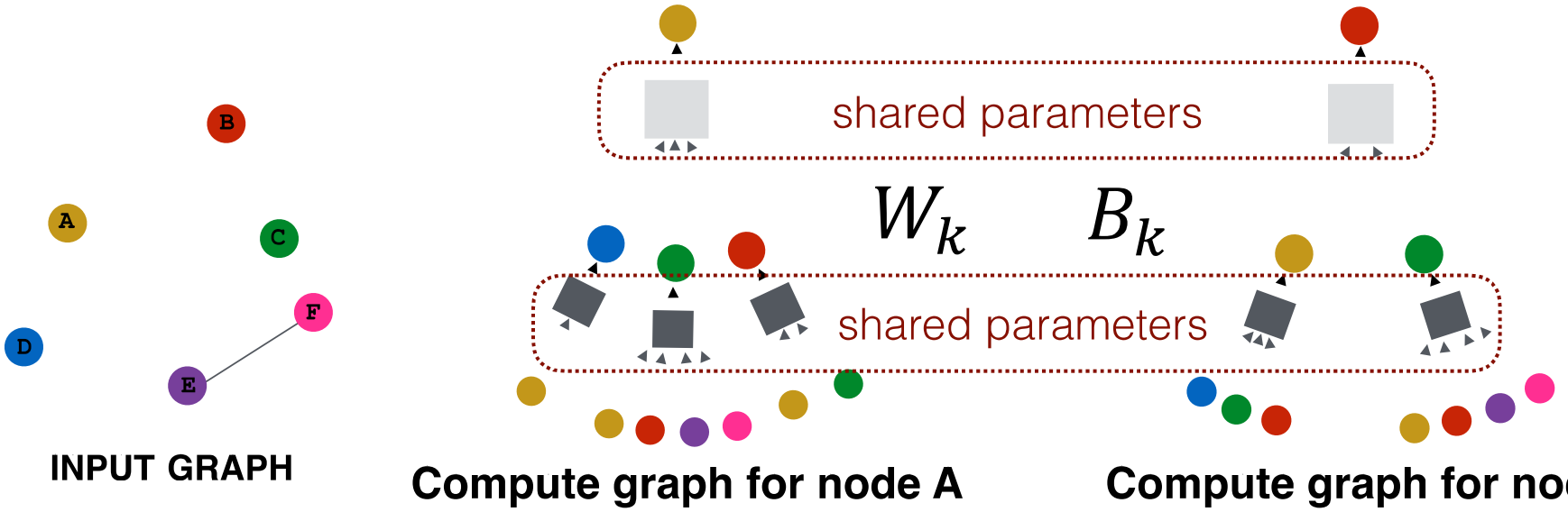
(4) Generate embeddings for nodes as needed

Even for nodes we never trained on!

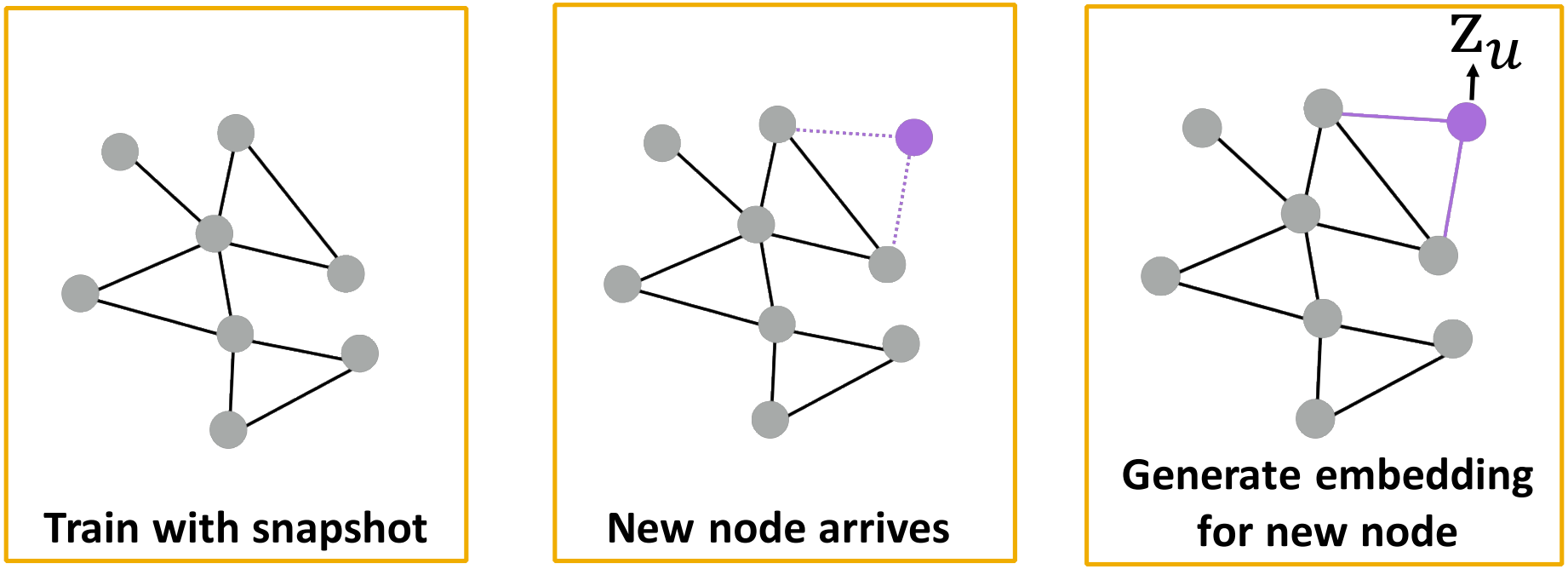


Inductive Capability

- **The same aggregation parameters are shared for all nodes:**
  - The number of model parameters is sublinear in $|V|$ and we can **generalize to unseen nodes**!

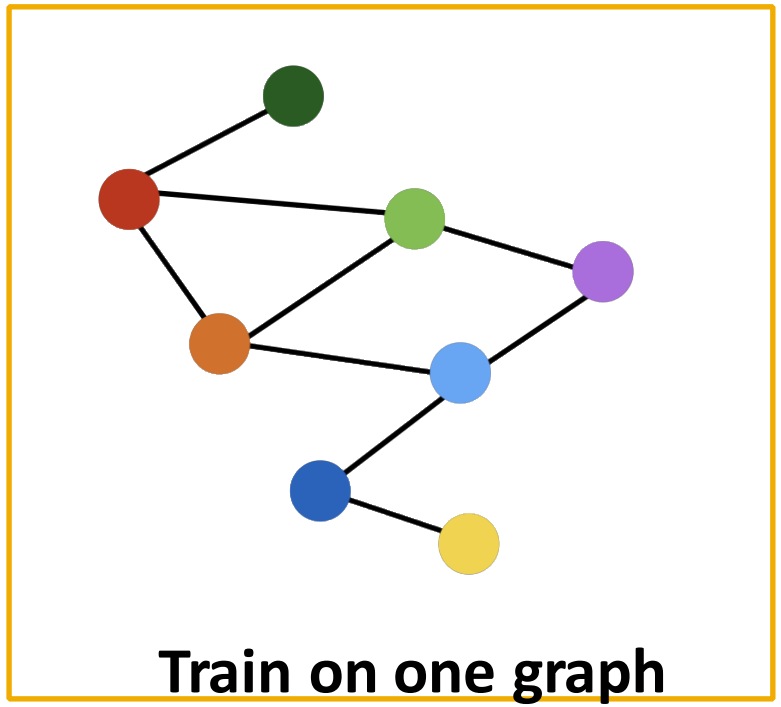


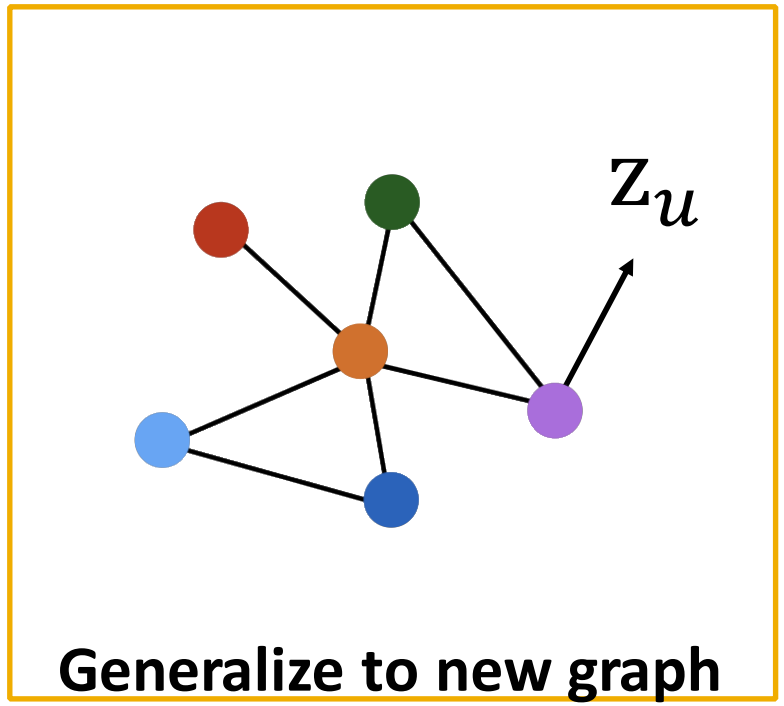
Inductive Capability: New Nodes



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

Inductive Capability: New Graphs

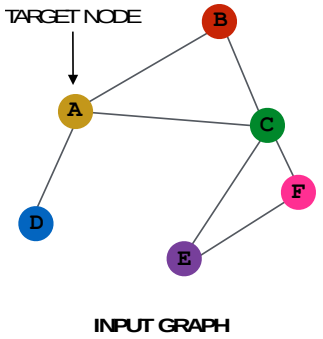




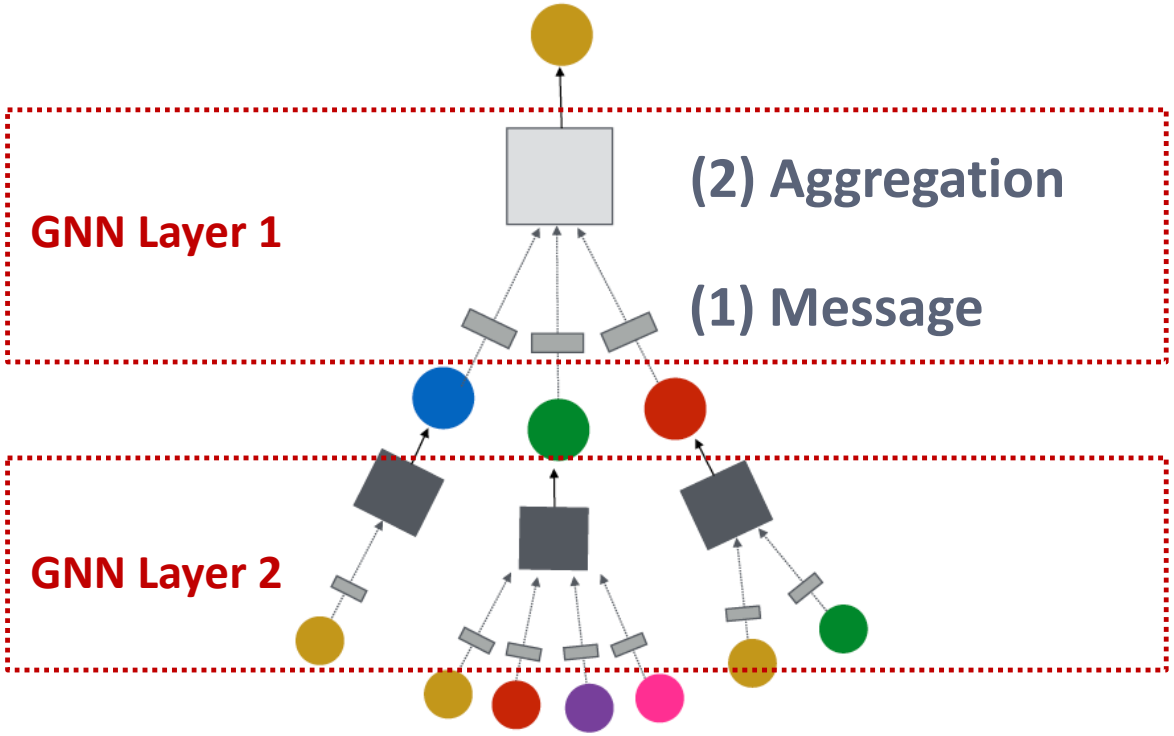
Inductive node embedding → Generalize to entirely unseen graphs

E.g., train on protein interaction graph from model organism A and generate embeddings on newly collected data about organism B

Discussion: Design Space of GNNs



(5) Learning objective



(3) Layer connectivity

(4) Graph augmentation

J. You, R. Ying, J. Leskovec. Design Space of Graph Neural Networks, NeurIPS 2020

Ex1: Connectivity

Our assumption so far has been

- **Raw input graph = computational graph**

Reasons for breaking this assumption

- **Feature level:**
  - The input graph **lacks features** → feature augmentation
- **Structure level:**
  - The graph is **too sparse** à inefficient message passing
  - The graph is **too dense** à message passing is too costly
  - The graph is **too large** à cannot fit the computational graph into a GPU
- It's just **unlikely that the input graph happens to be the optimal computation graph** for embeddings

Ex1: Connectivity

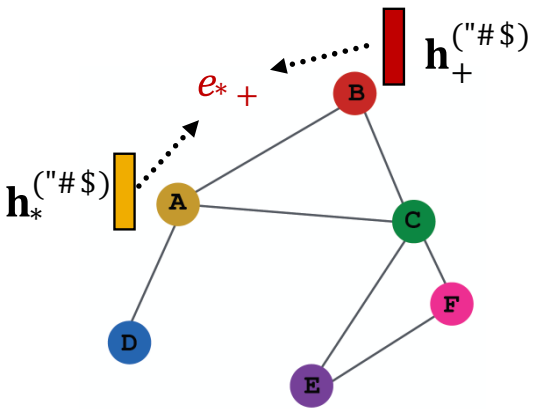
- **Graph Feature manipulation**
  - The input graph **lacks features** → **feature augmentation**
- **Graph Structure manipulation**
  - The graph is **too sparse** → **Add virtual nodes / edges**
  - The graph is **too dense** → **Sample neighbors when doing message passing**
  - The graph is **too large** → **Sample subgraphs to compute embeddings**
    - Will cover later in lecture: Scaling up GNNs

Ex2: Graph Attention Network (GAT)

- **In GCN**
  - $\alpha_{vu} = \frac{1}{|N(v)|}$ is the **weighting factor (importance)** of node u's message to node v
  - $\Longrightarrow$ α_{vu} is defined **explicitly** based on the structural properties of the graph (node degree)
  - $\Longrightarrow$ All neighbors $u \in N(v)$ are equally important to node v

Not all node's neighbors are equally important

- Query, Key, Value
- Alignment $\boldsymbol{e}$
- $\boldsymbol{a} = \mathrm{softmax}(\boldsymbol{e})$



Questions?