Graph Neural Networks: Perspective

Jiaxuan You

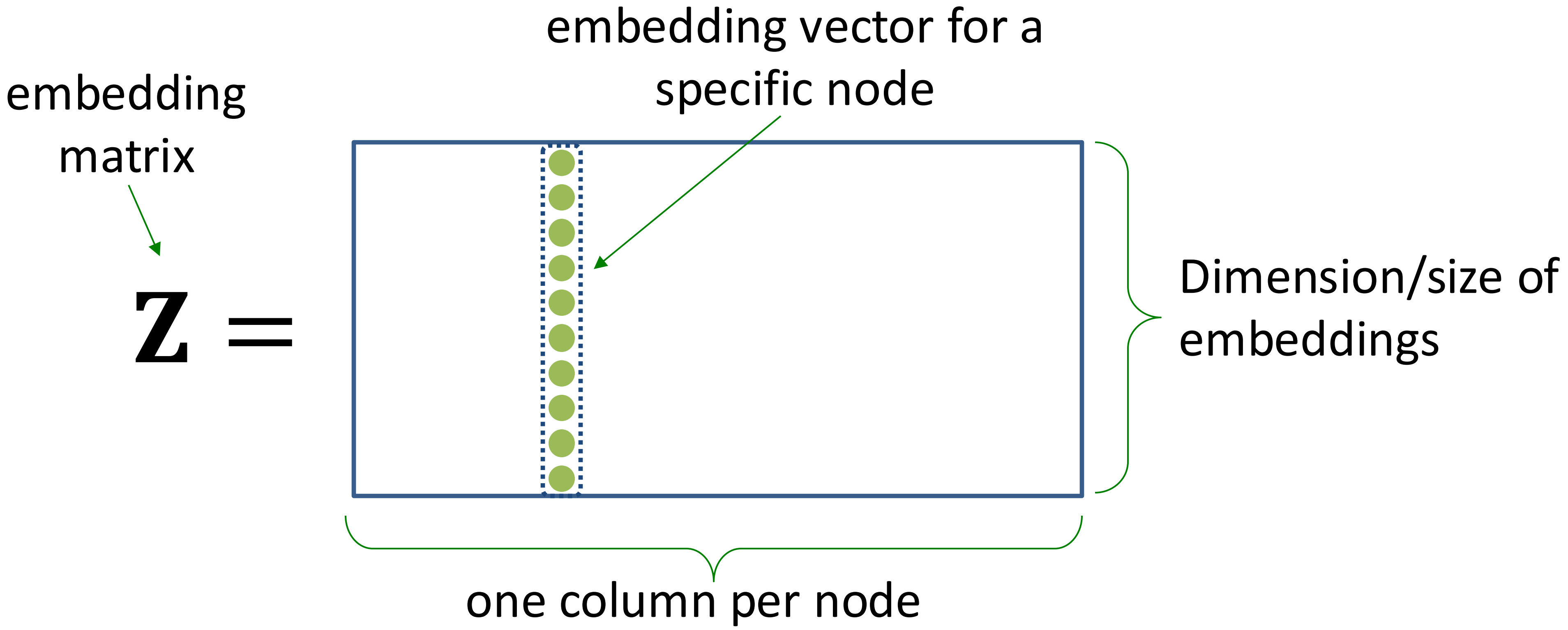
Assistant Professor at UIUC CDS

CS598: Deep Learning with Graphs, 2024 Fall

https://ulab-uiuc.github.io/CS598/

Recap: “Shallow” Encoding

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


embedding matrix
$\mathbf{Z} =$
embedding vector for a specific node
Dimension/size of embeddings
one column per node

Recap: “Shallow” Encoding

- **Limitations** of shallow embedding methods:
  - $\boldsymbol{O(|V|d)}$ **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

Recap: Similarity Based Objective Function

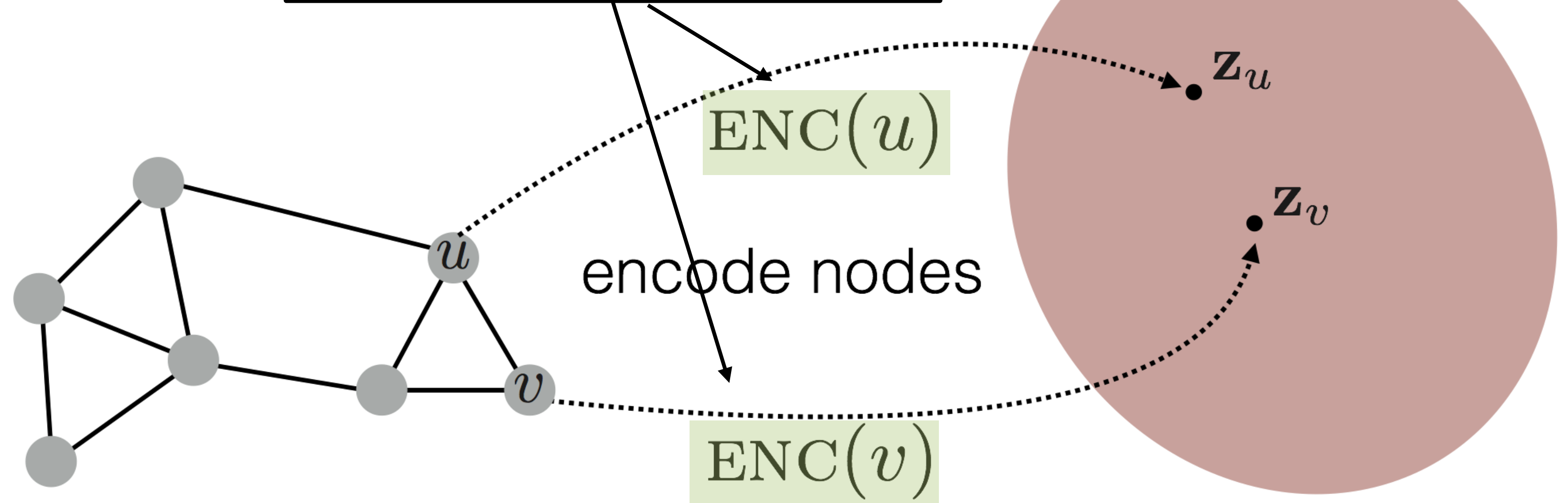
- Requires an **edge-level prediction head**

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

in the original network

similarity of the embedding

Need to define!



original network

embedding space

Today: Deep Graph Encoders

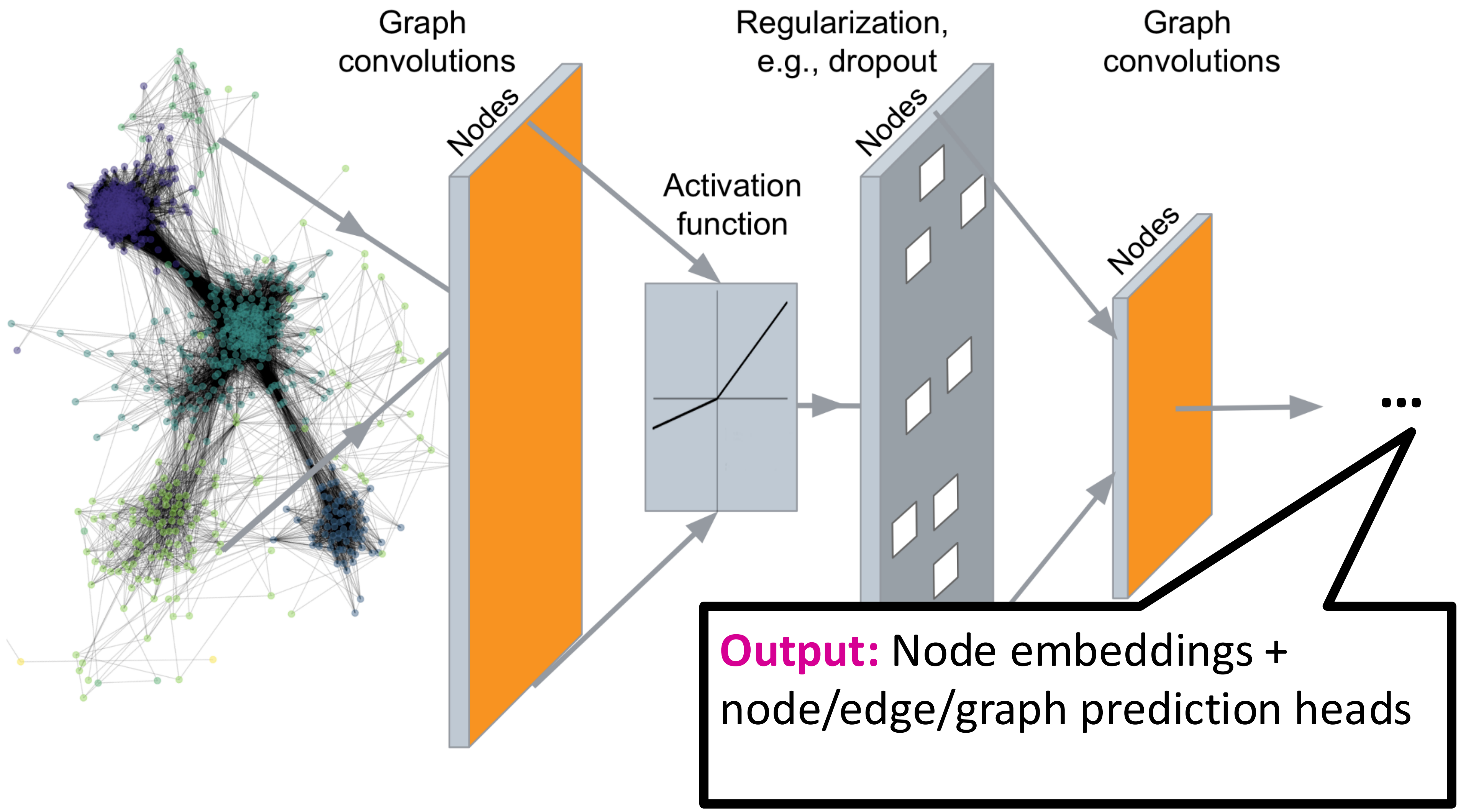
- **Today:** We will now discuss deep learning methods based on **graph neural networks (GNNs):**

$$\mathrm{ENC}(v) =$$

multiple layers of non-linear transformations based on graph structure

- **Note:** All these deep graph encoders still output node embeddings, and can be **combined with different node/edge/graph prediction heads**

Deep Graph Encoders



Output: Node embeddings + node/edge/graph prediction heads

Graph Neural Networks: Perspective

Deep Learning for Graphs

Content

- **Local network neighborhoods:**
  - Describe aggregation strategies
  - Define computation graphs

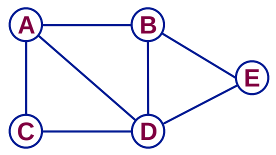
- **Stacking multiple layers:**
  - Describe the model, parameters, training
  - How to fit the model?
  - Simple example for unsupervised and supervised training

Setup

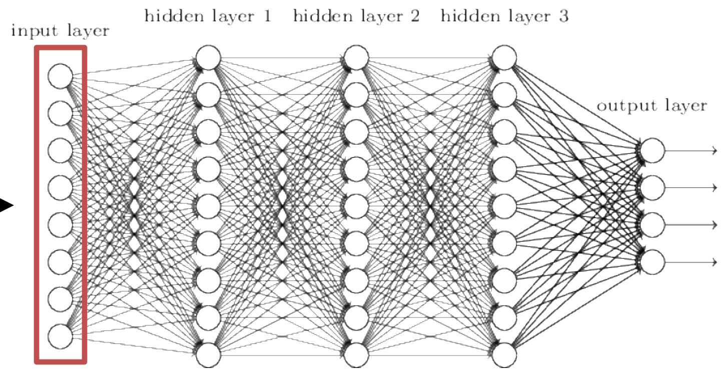
- **Assume we have a graph G:**
  - V is the **vertex set**
  - $\boldsymbol{A}$ is the **adjacency matrix** (assume binary)
  - $\boldsymbol{X} \in \mathbb{R}^{|V| \times d}$ is a matrix of **node features**
  - v: a node in V; $N(v)$: the set of neighbors of v.
  - **Node features:**
    - Social networks: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



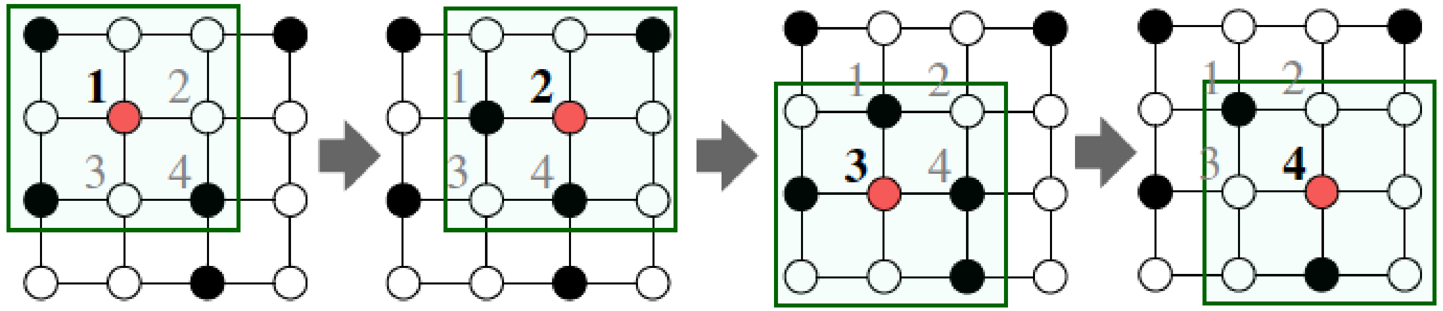
$$\begin{array}{c|ccccc|cc} & A & B & C & D & E & \multicolumn{2}{c}{Feat} \\ A & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ B & \boxed{1} & \boxed{0} & \boxed{0} & \boxed{1} & \boxed{1} & \boxed{0} & \boxed{0} \\ C & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ D & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{array} \longrightarrow$$

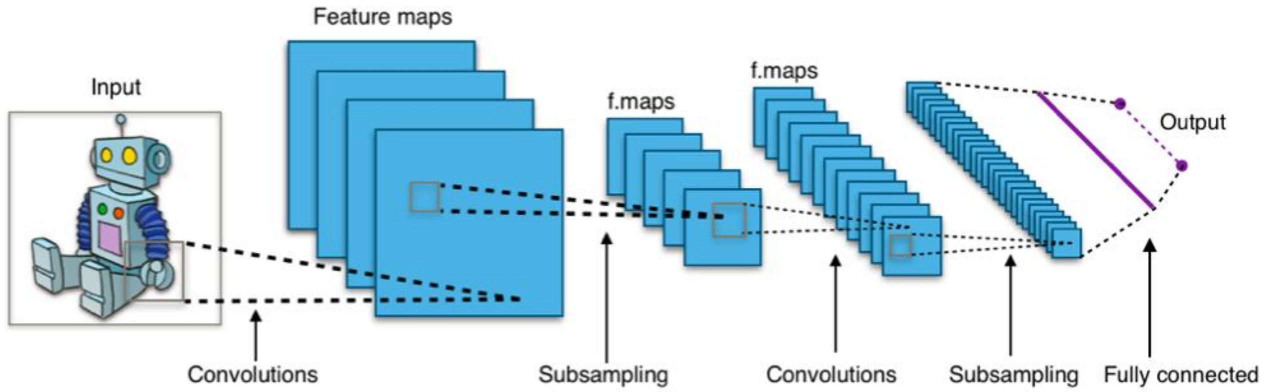


?

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

Idea: Convolutional Networks

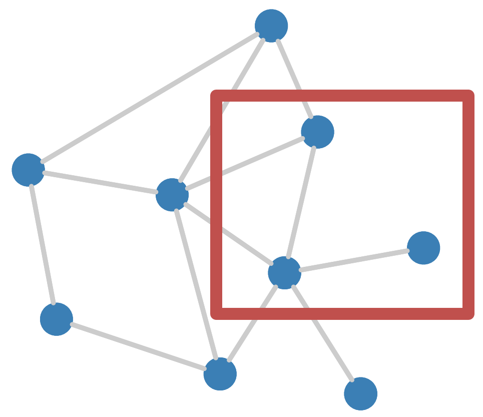




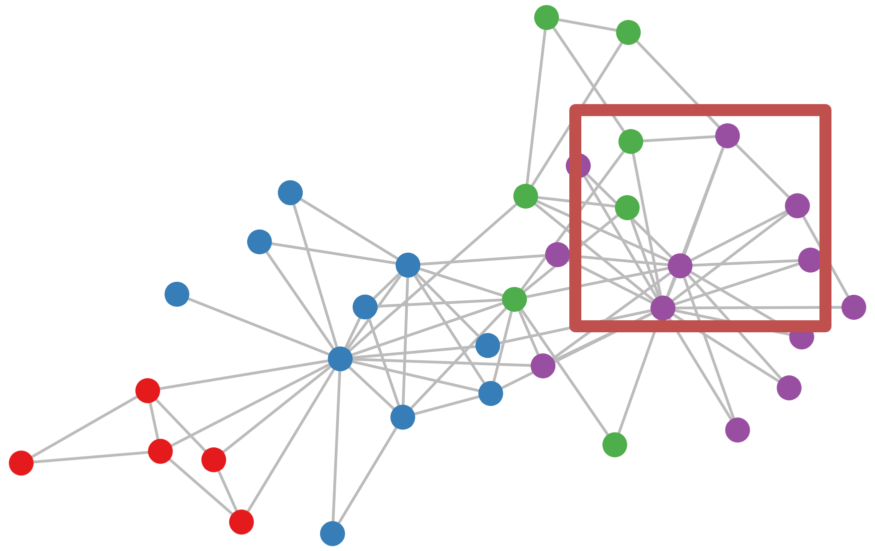
Goal is to generalize convolutions beyond simple lattices
Leverage node features/attributes (e.g., text, images)

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

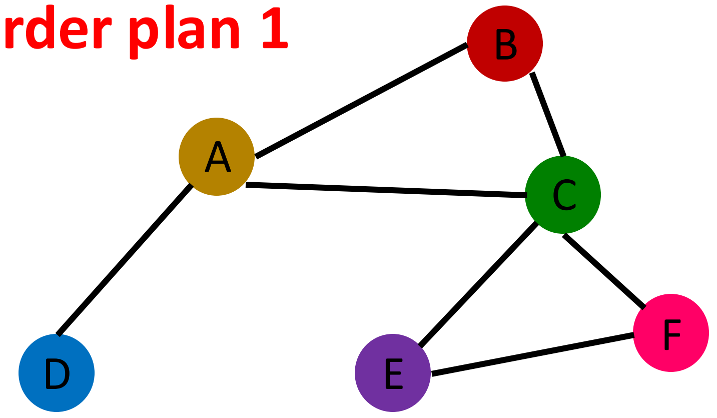
Permutation Invariance

- **Graph does not have a canonical order of the nodes!**
- We can have many different order plans.

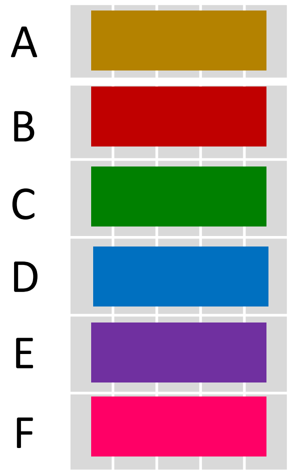
Permutation Invariance

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

Order plan 1



Node features X_1



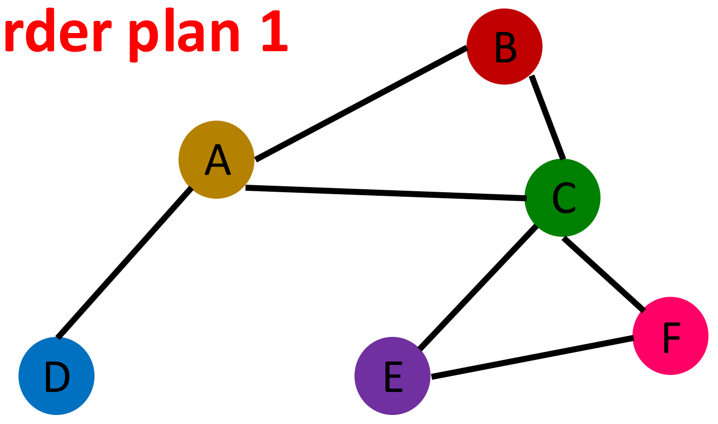
Adjacency matrix A_1

	A	B	C	D	E	F
A	⬛	🟦	🟦	🟦	⬛	⬛
B	🟦	⬛	🟦	⬛	⬛	⬛
C	🟦	🟦	⬛	⬛	🟦	🟦
D	🟦	⬛	⬛	⬛	⬛	⬛
E	⬛	⬛	🟦	⬛	⬛	🟦
F	⬛	⬛	🟦	⬛	🟦	⬛

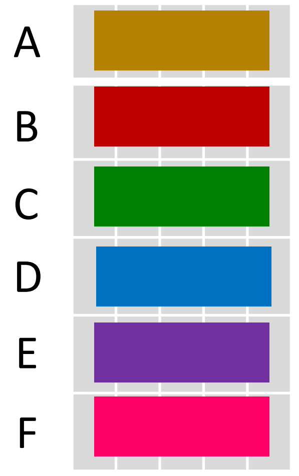
Permutation Invariance

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

Order plan 1



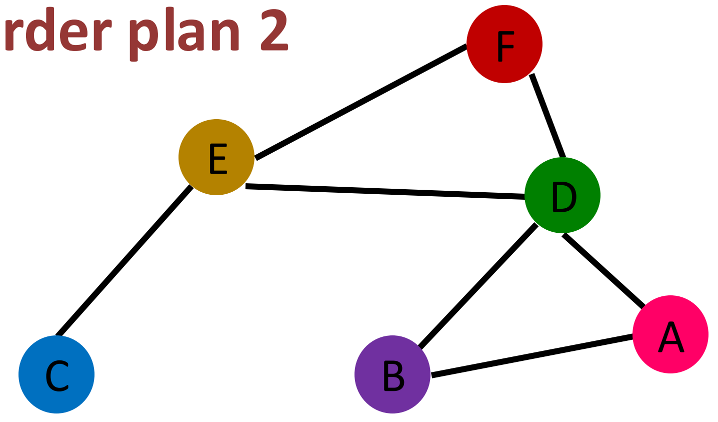
Node features X_1



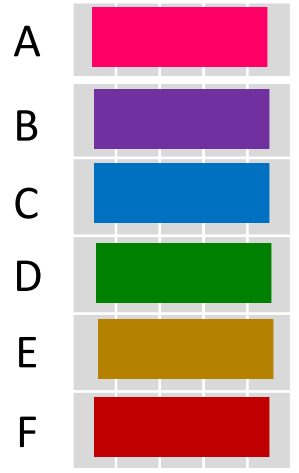
Adjacency matrix A_1

	A	B	C	D	E	F
A	⬛	🟦	🟦	🟦	⬛	⬛
B	🟦	⬛	🟦	⬛	⬛	⬛
C	🟦	🟦	⬛	⬛	🟦	🟦
D	🟦	⬛	⬛	⬛	⬛	⬛
E	⬛	⬛	🟦	⬛	⬛	🟦
F	⬛	⬛	🟦	⬛	🟦	⬛

Order plan 2



Node features X_2

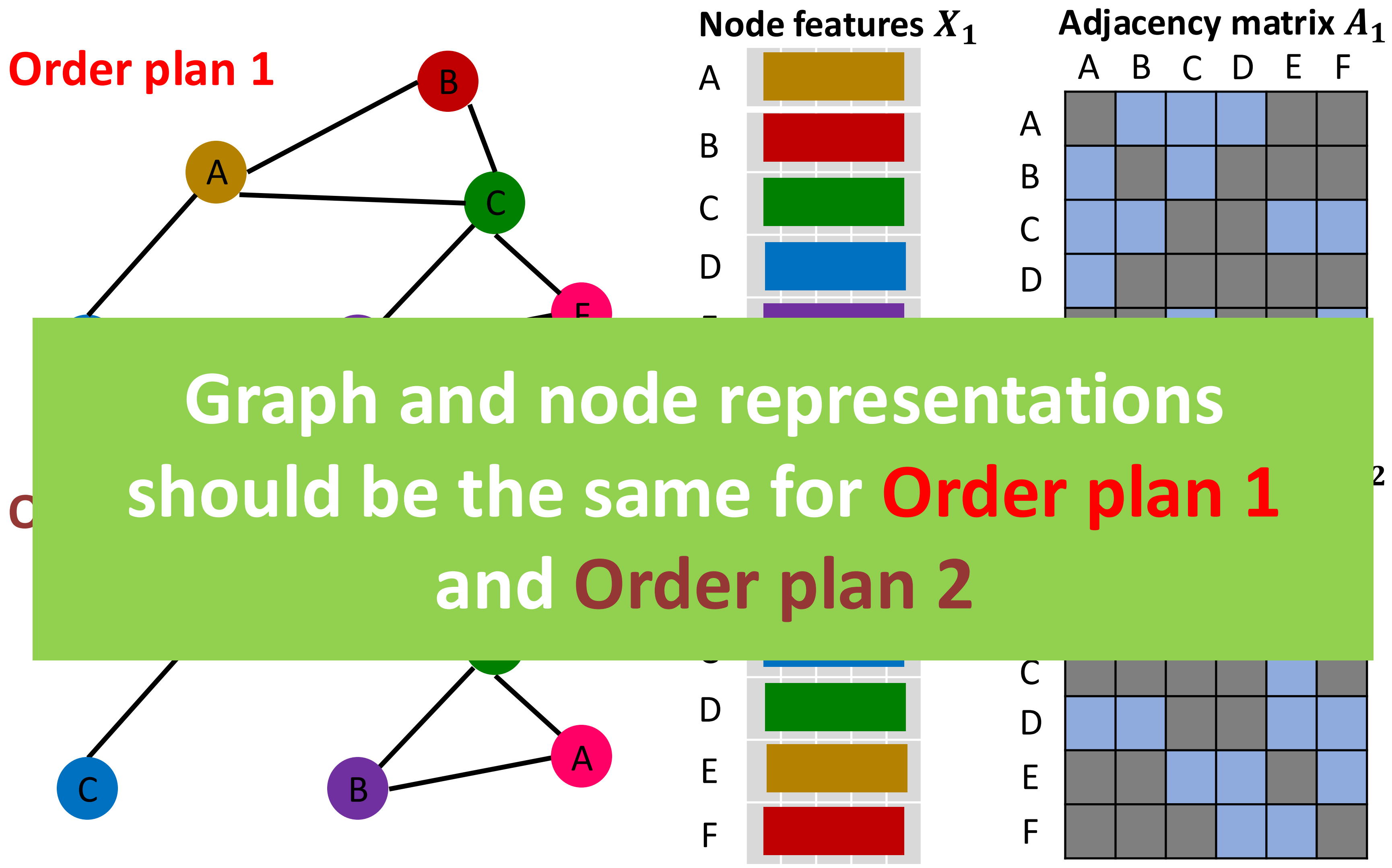


Adjacency matrix A_2

	A	B	C	D	E	F
A	⬛	🟦	⬛	🟦	⬛	⬛
B	🟦	⬛	⬛	🟦	⬛	⬛
C	⬛	⬛	⬛	⬛	🟦	⬛
D	🟦	🟦	⬛	⬛	🟦	🟦
E	⬛	⬛	🟦	🟦	⬛	🟦
F	⬛	⬛	⬛	🟦	🟦	⬛

Permutation Invariance

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



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

Permutation Invariance

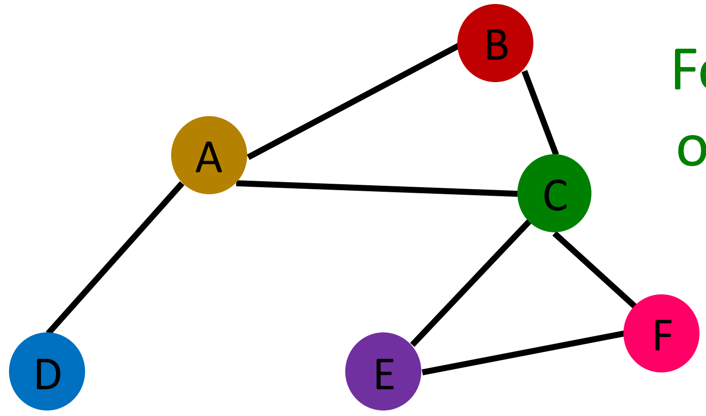
What does it mean by "graph representation is same for two order plans"?

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

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

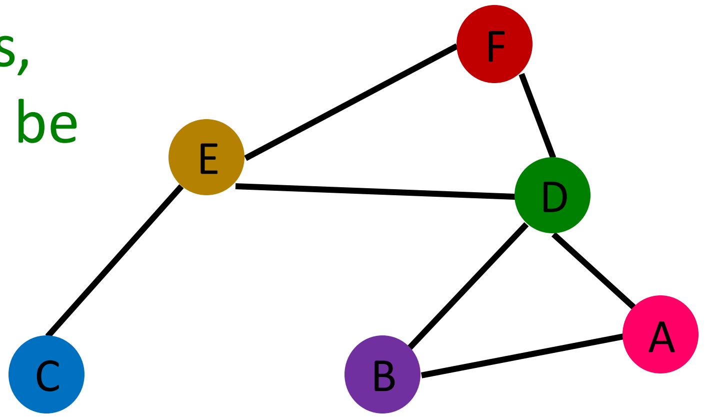
$$f(\boldsymbol{A}_1, \boldsymbol{X}_1) = f(\boldsymbol{A}_2, \boldsymbol{X}_2)$$

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

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, 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**.
- **Definition:** For any graph function $f: \mathbb{R}^{|V| \times m} \times \mathbb{R}^{|V| \times |V|} \to \mathbb{R}^d$, f is **permutation-invariant** if $f(A, X) = f(PAP^T, PX)$ for any permutation P.

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

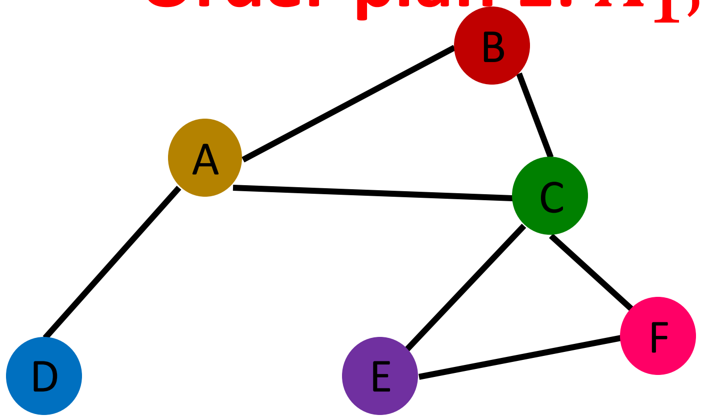
For a graph with $|V|$ nodes, there are $|V|!$ different order plans.

Permutation P: a shuffle of the node order
Example: (A,B,C)->(B,C,A)

Permutation Equivariance

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

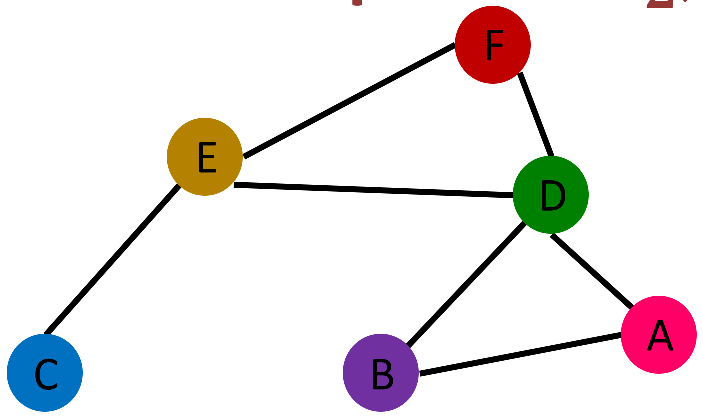
Order plan 1: $\boldsymbol{A}_1, \boldsymbol{X}_1$



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

A	🟫	🟫
B	🟥	🟥
C	🟩	🟩
D	🟦	🟦
E	🟪	🟪
F	🟥	🟥

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



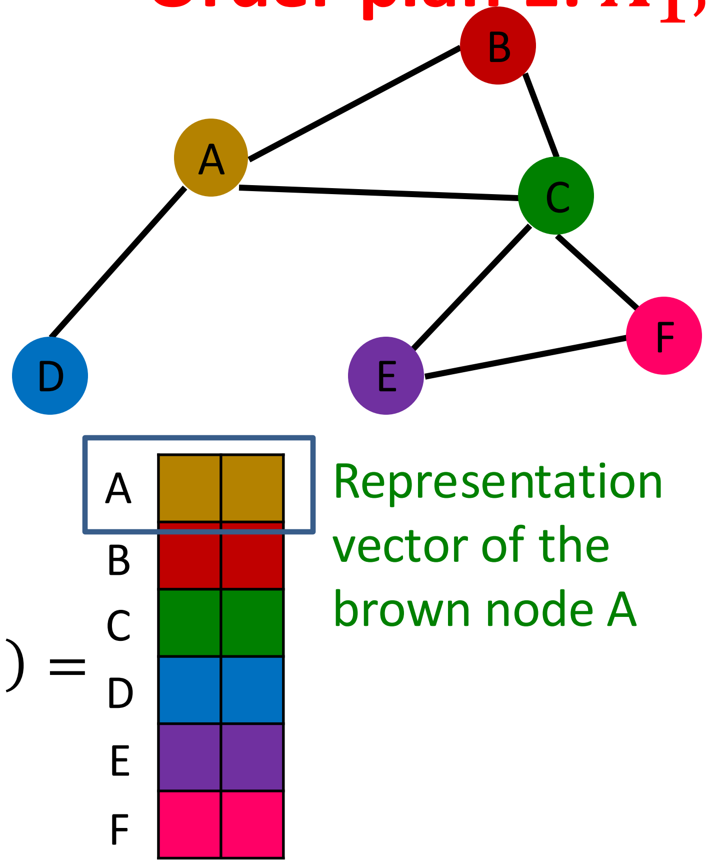
$f(\boldsymbol{A}_2, \boldsymbol{X}_2) =$

A	🟥	🟥
B	🟪	🟪
C	🟦	🟦
D	🟩	🟩
E	🟫	🟫
F	🟥	🟥

Permutation Equivariance

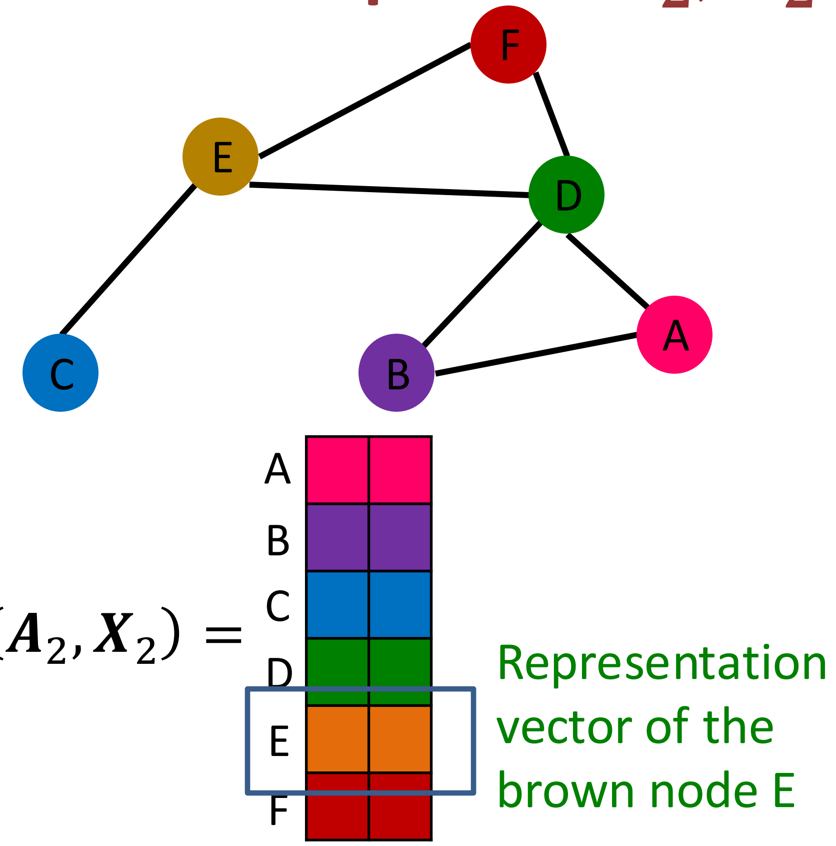
- **For node representation:** We learn a function f that maps nodes of G to a matrix $\mathbb{R}^{m\times d}$.

Order plan 1: $\boldsymbol{A_1}, \boldsymbol{X_1}$



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

Order plan 2: $\boldsymbol{A_2}, \boldsymbol{X_2}$



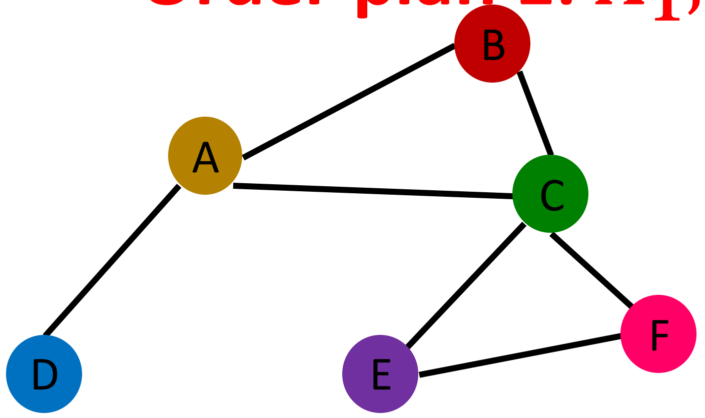
$f(\boldsymbol{A}_2, \boldsymbol{X}_2) =$

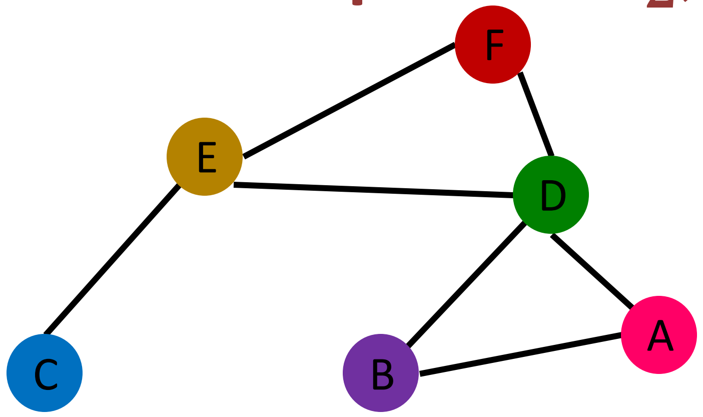
Permutation Equivariance

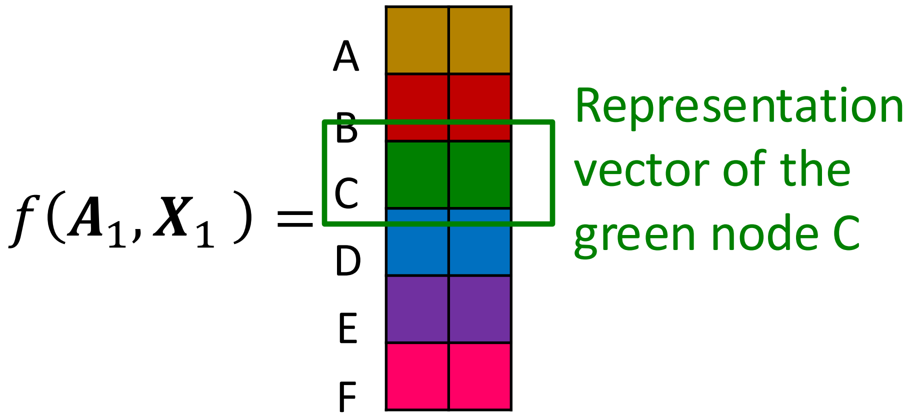
- **For node representation:** We learn a function f that maps nodes of G to a matrix $\mathbb{R}^{m\times d}$.

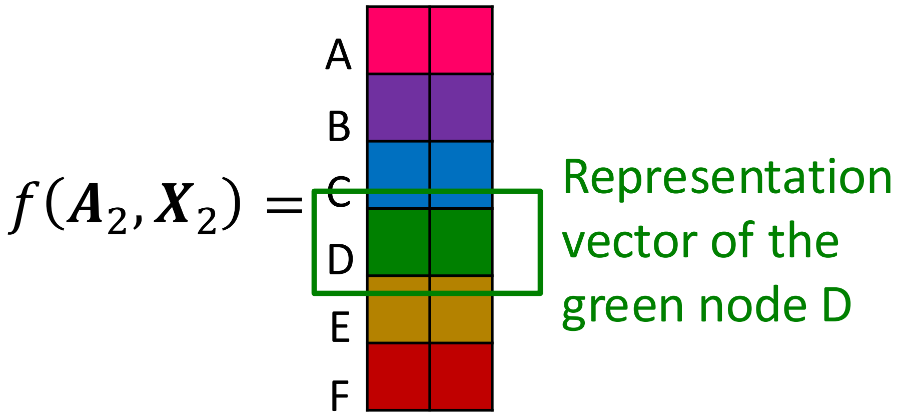
Order plan 1: $\boldsymbol{A_1}, \boldsymbol{X_1}$

Order plan 2: $\boldsymbol{A_2}, \boldsymbol{X_2}$






$f(\boldsymbol{A}_1, \boldsymbol{X}_1) =$
A
B
C
D
E
F
Representation vector of the green node C


$f(\boldsymbol{A}_2, \boldsymbol{X}_2) =$
A
B
C
D
E
F
Representation vector of the green node D

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: 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**

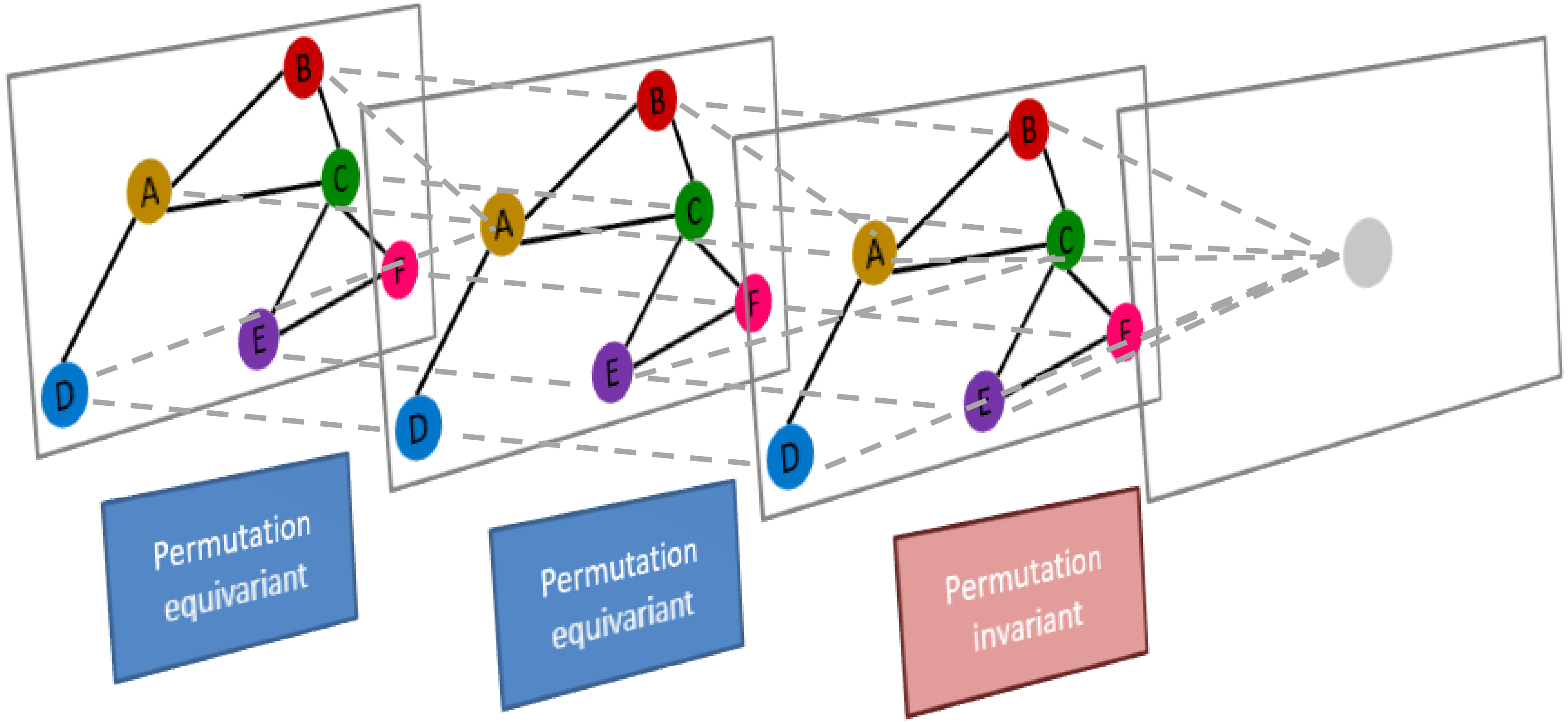
$$Pf(A, X) = f(PAP^T, PX)$$

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 Network Overview

- **Graph neural networks consist of multiple permutation equivariant / invariant functions.**

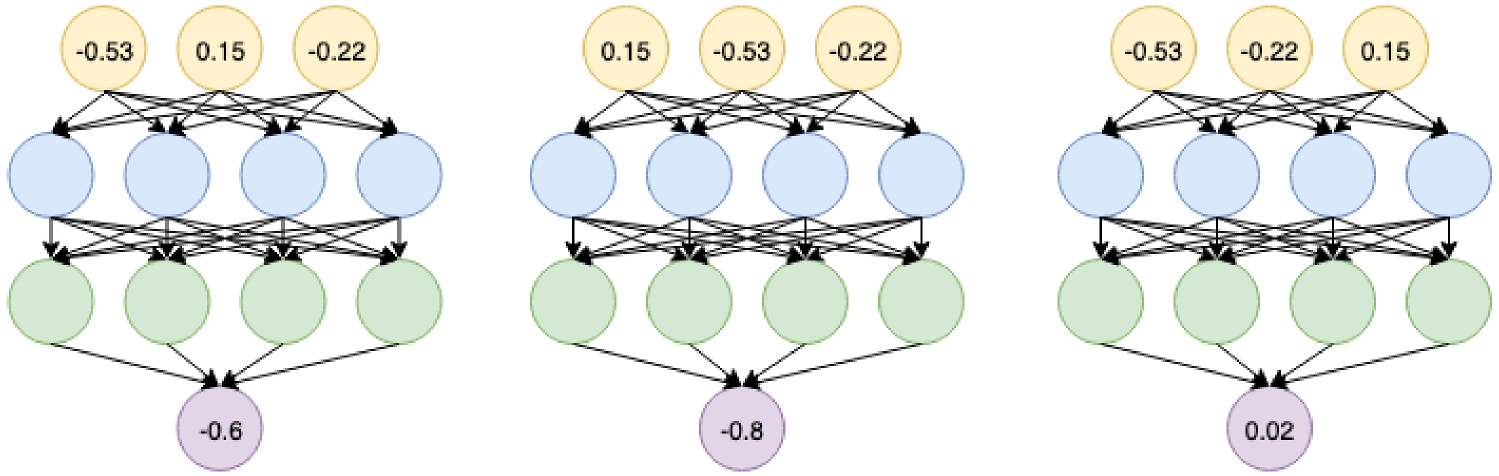


Graph Neural Network Overview

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

- **No.**

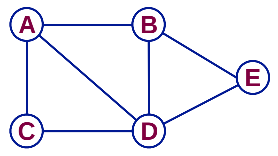
Switching the order of the input leads to different outputs!



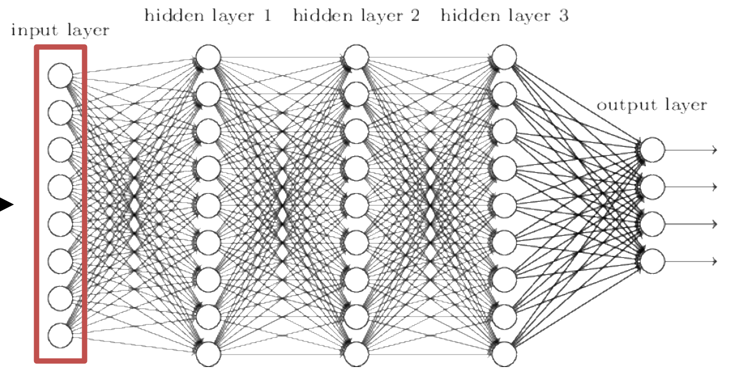
Graph Neural Network Overview

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

- **No.**



$$\begin{array}{c|ccccc|cc} & A & B & C & D & E & \multicolumn{2}{c}{Feat} \\ \hline A & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ B & \boxed{1} & 0 & 0 & 1 & 1 & 0 & 0 \\ C & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ D & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{array} \longrightarrow$$



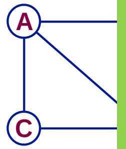
?

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

Graph Neural Network Overview

Are othe **on**
invariant

- **No.**



?

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

fails for graphs!

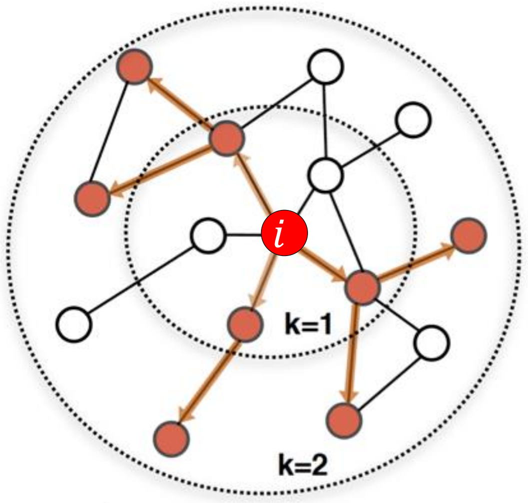
Graph Neural Networks: Perspective

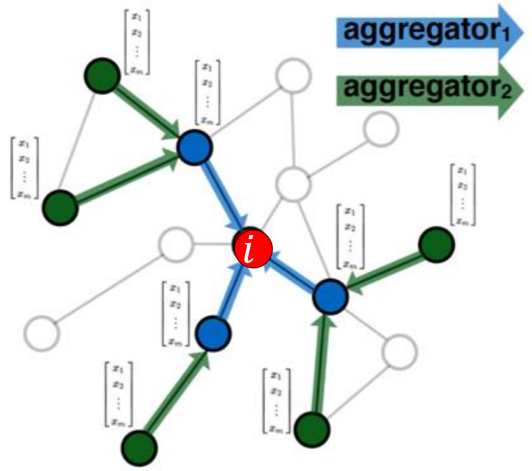
Graph Convolutional Networks

[Kipf and Welling, ICLR 2017]

Graph Convolutional Networks

- Idea: Node's neighborhood defines a computation graph





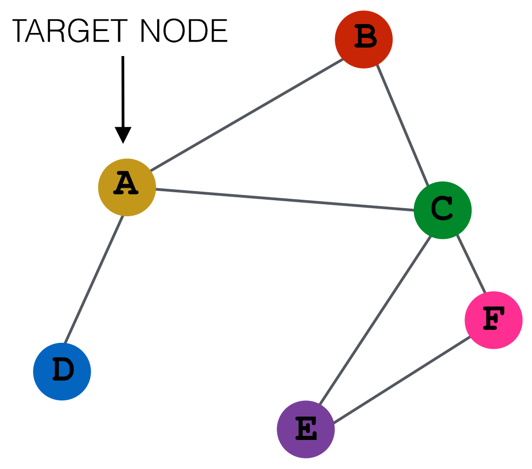
Determine node computation graph

Propagate and transform information

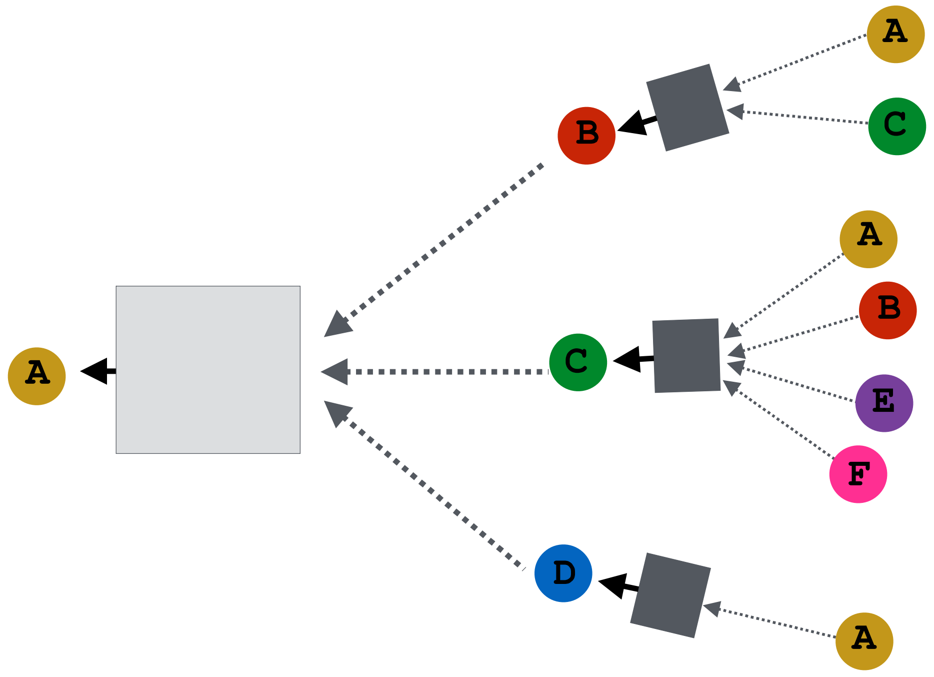
Learn how to propagate information across the graph to compute node features

Idea: Aggregate Neighbors

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

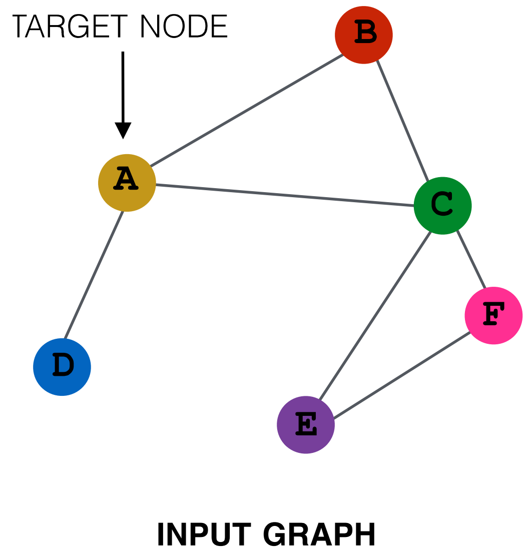


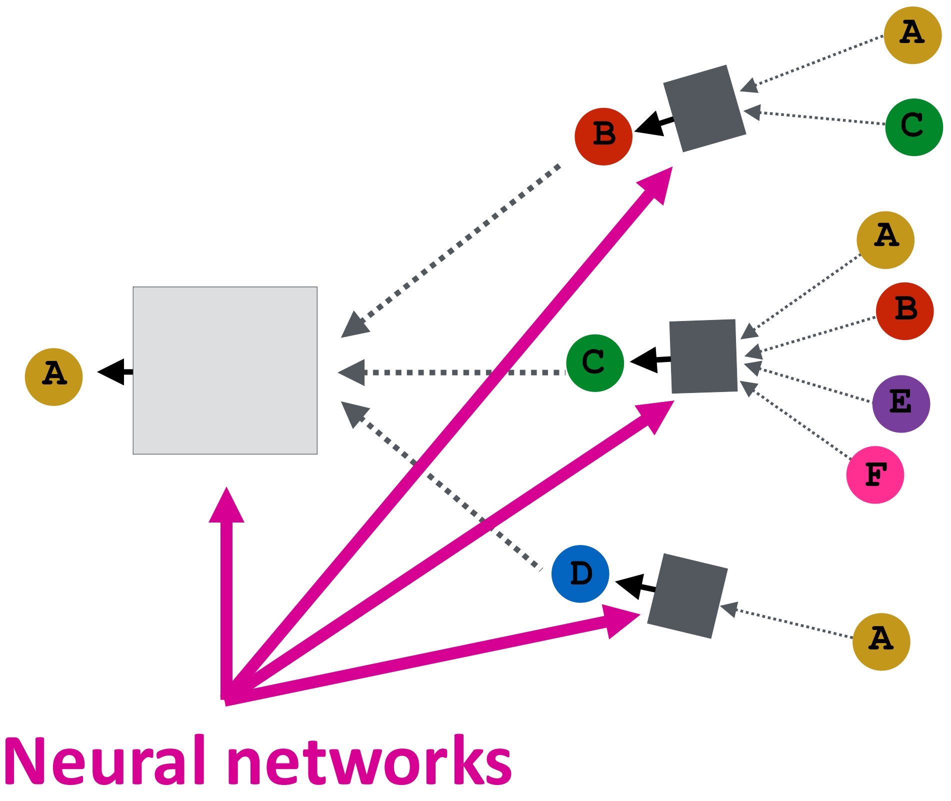
INPUT GRAPH



Idea: Aggregate Neighbors

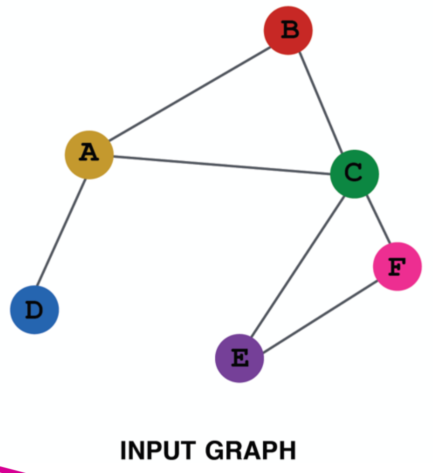
- **Intuition:** Nodes aggregate information from their neighbors using neural networks



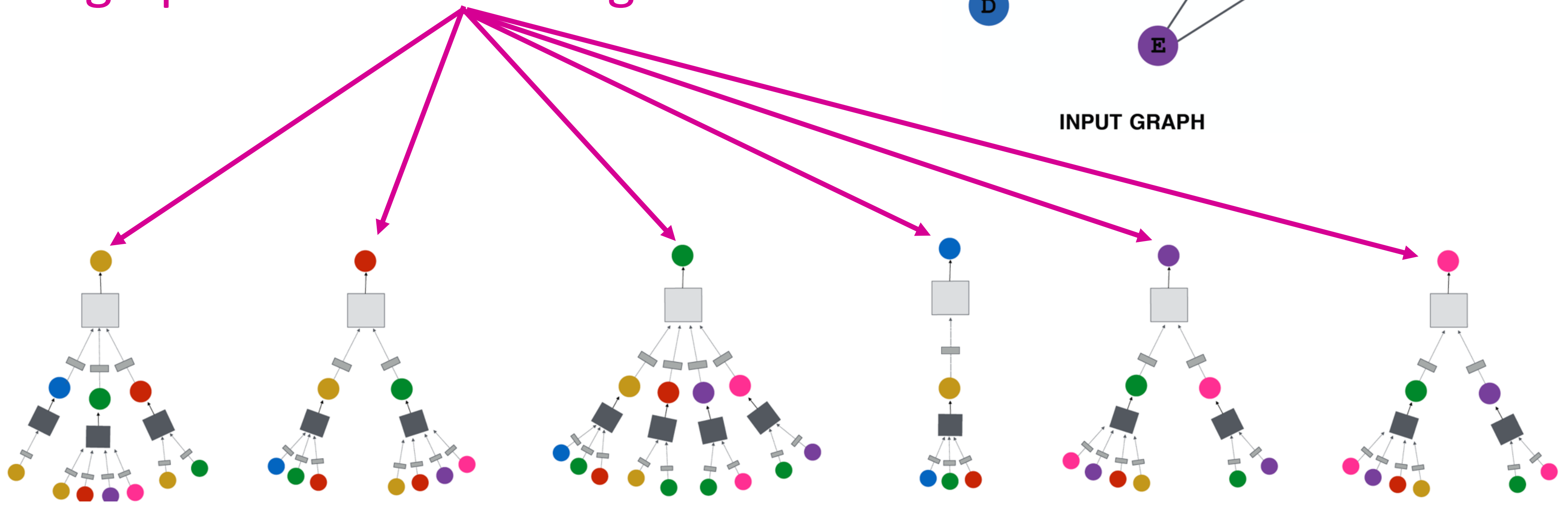


Idea: Aggregate Neighbors

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

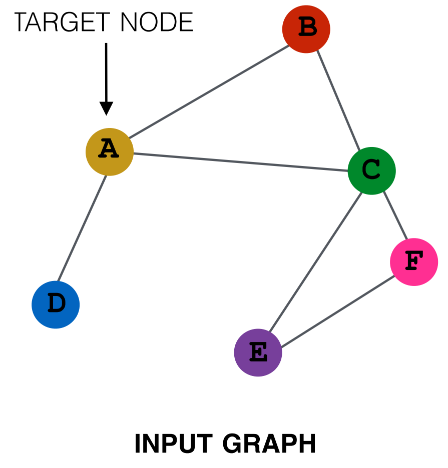


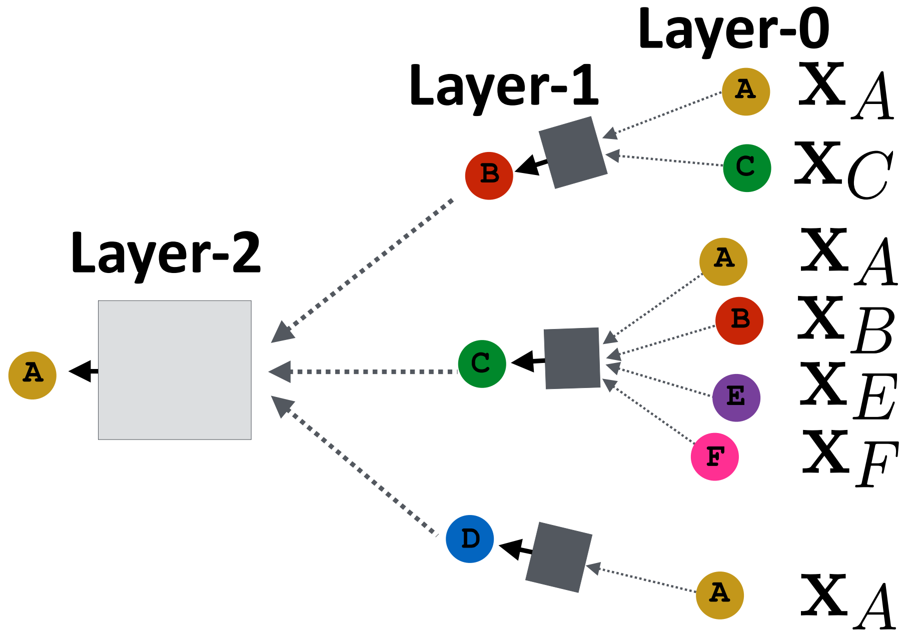
Every node defines a computation graph based on its neighborhood!



Deep Model: Many Layers

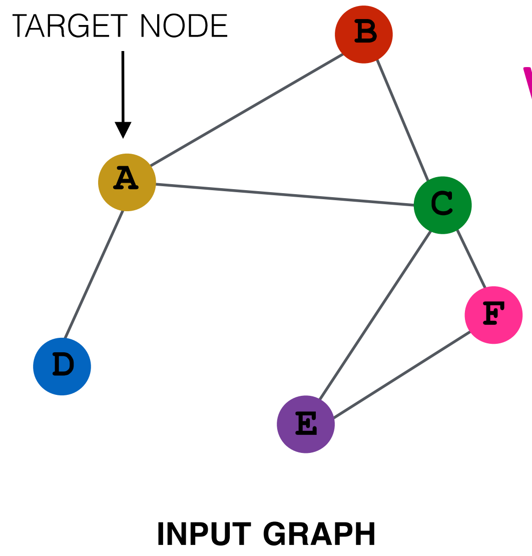
- Model can be of arbitrary depth:
  - Nodes have embeddings at each layer
  - Layer-0 embedding of node v is its input feature, x_v
  - Layer-k embedding gets information from nodes that are k hops away



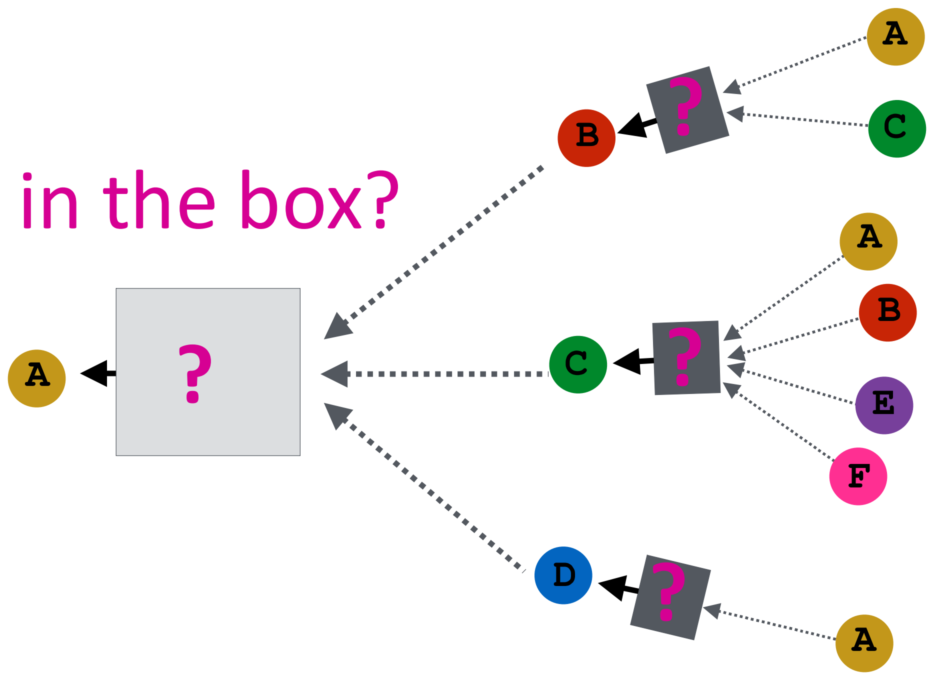


Neighborhood Aggregation

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

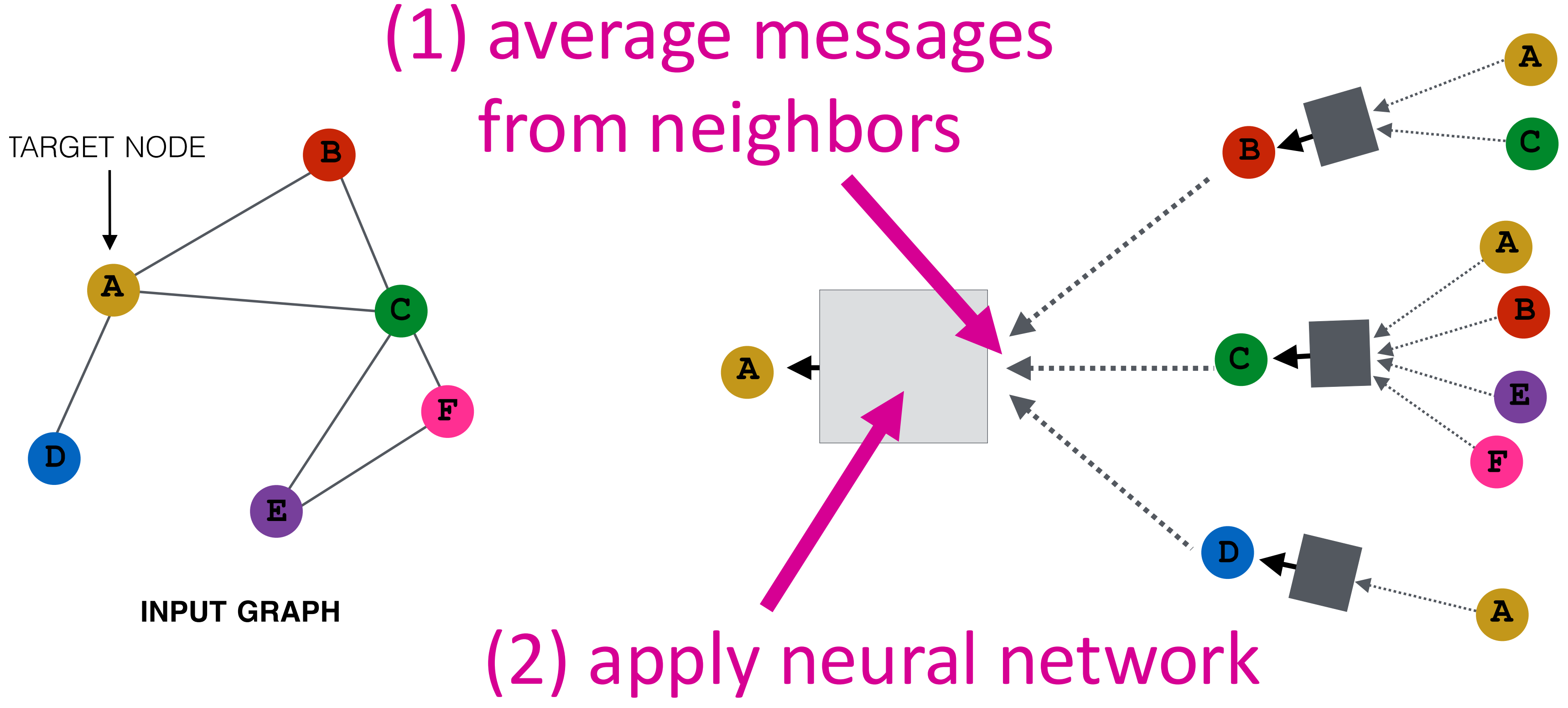


What is in the box?



Neighborhood Aggregation

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



The Math: Deep Encoder

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

Initial 0-th layer embeddings are equal to node features

$$\boxed{h_v^0 = x_v}$$

embedding of v at layer k

$$h_v^{(k+1)} = \boxed{\sigma}\left(W_k \boxed{\sum_{u \in N(v)} \frac{h_u^{(k)}}{|N(v)|}} + B_k \boxed{h_v^{(k)}}\right), \forall k \in \{0, \dots, \boxed{K} - 1\}$$

Average of neighbor's previous layer embeddings

Total number of layers

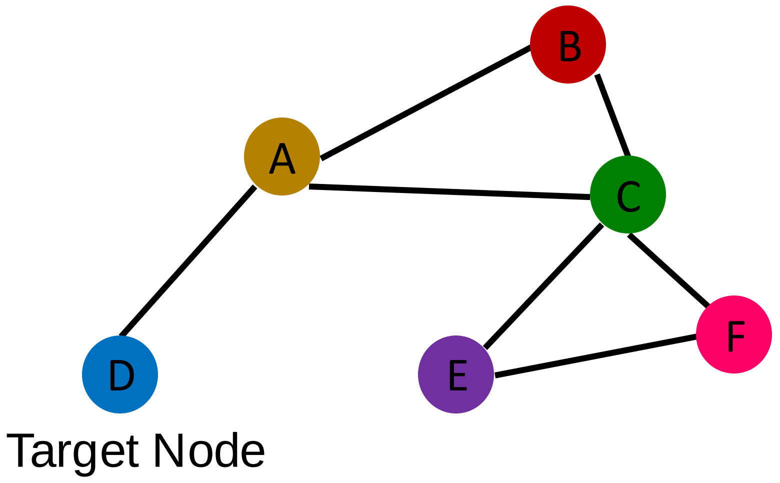
$$\boxed{z_v} = h_v^{(K)}$$

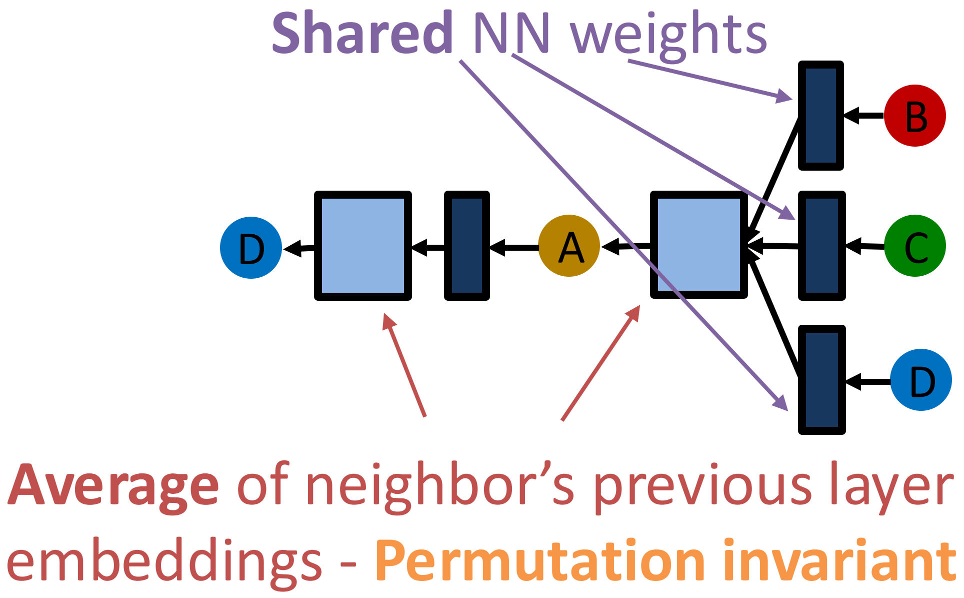
Embedding after K layers of neighborhood aggregation

GCN: 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**

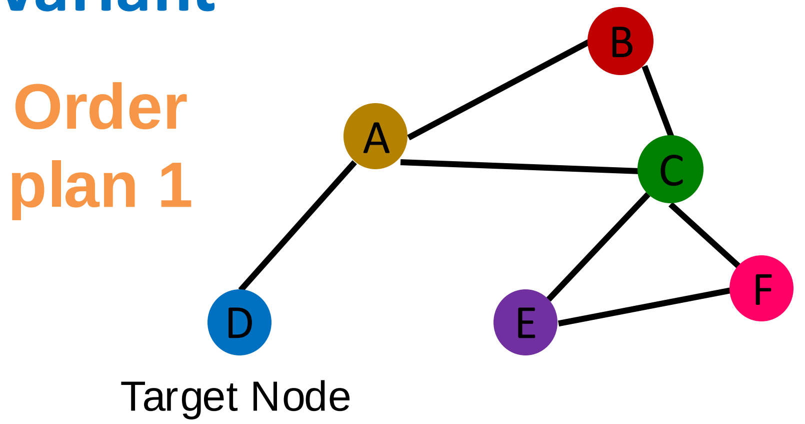


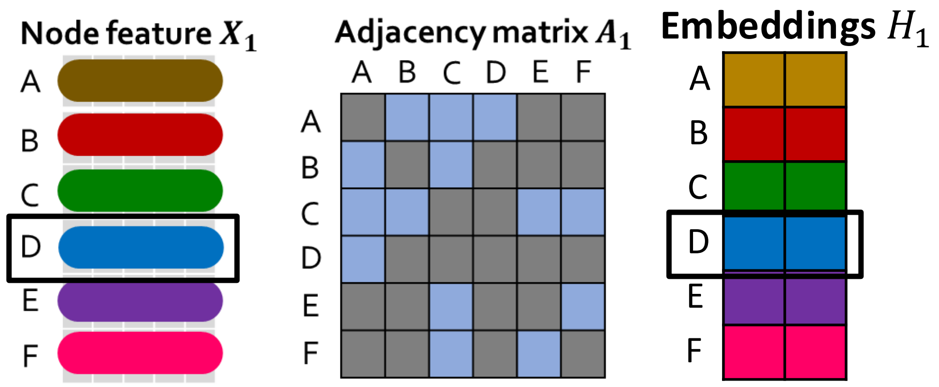


Average of neighbor's previous layer embeddings - **Permutation invariant**

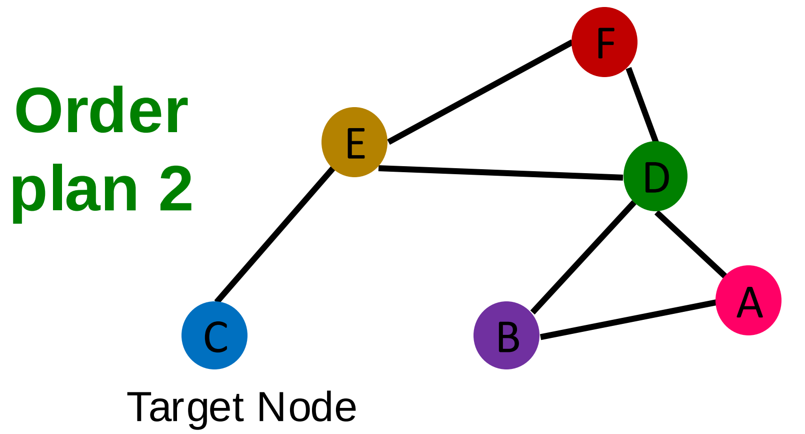
GCN: Invariance and Equivariance

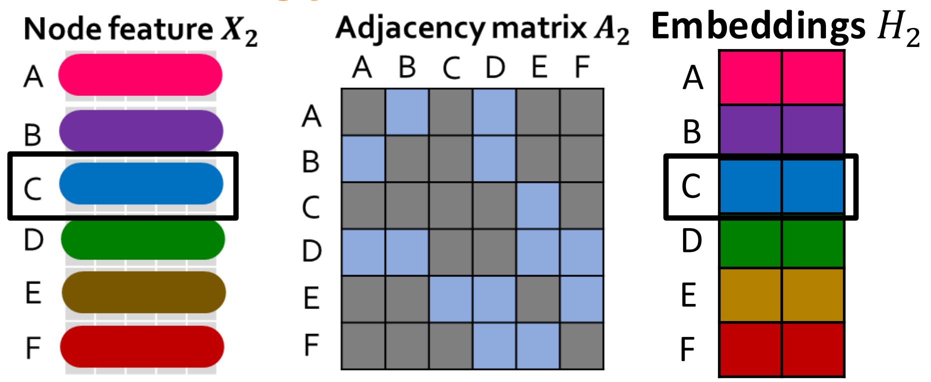
- **Considering all nodes in a graph**, GCN computation is **permutation equivariant**





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





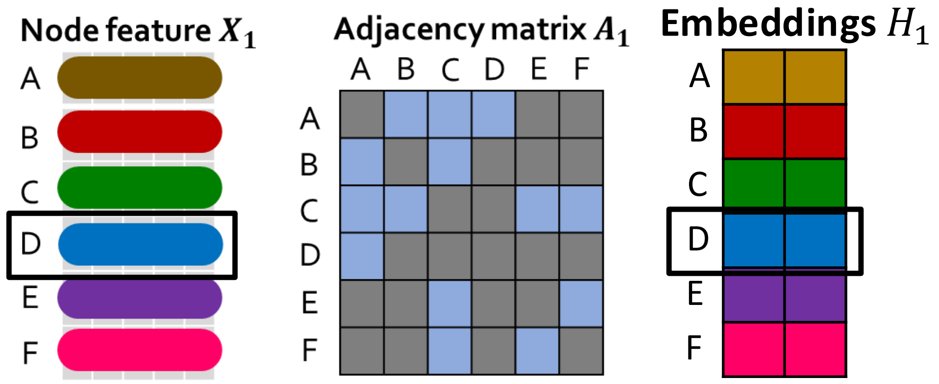
GCN: Invariance and Equivariance

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

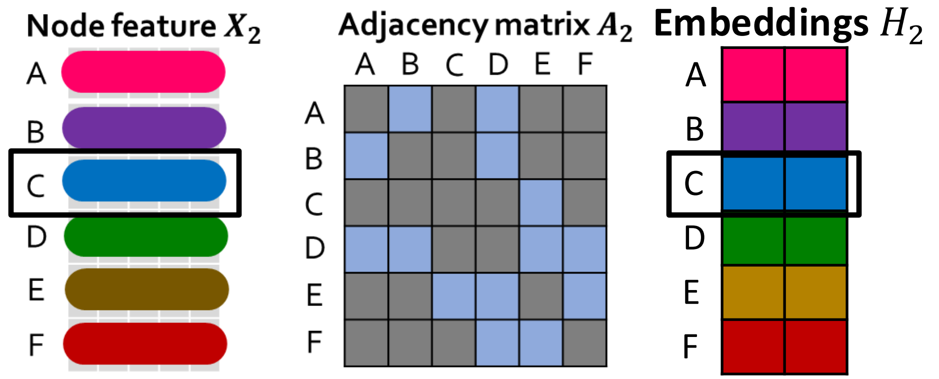
Detailed reasoning:

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

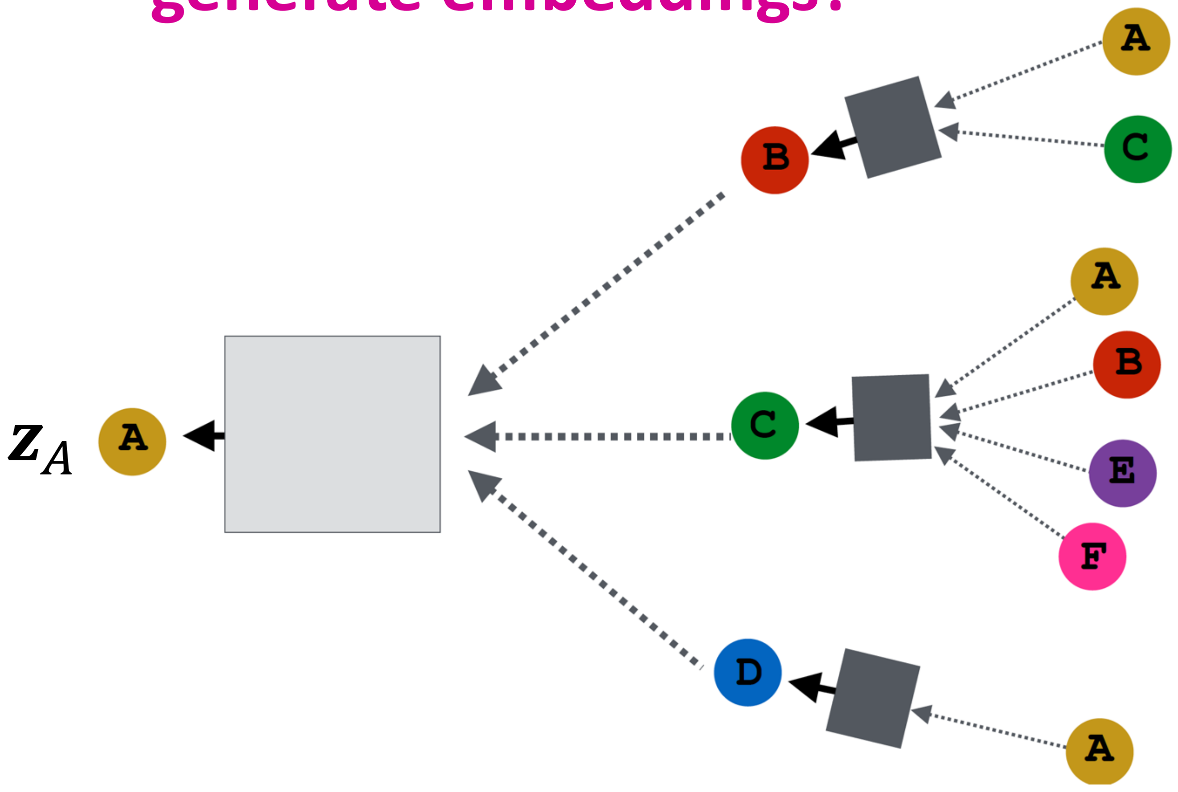


Permute the input, the output also permutes accordingly - permutation equivariant



Training the Model

How do we train the GCN to generate embeddings?



Need to define a loss function on the embeddings.

Model Parameters

Trainable weight matrices (i.e., what we learn)

$$h_v^{(0)} = x_v$$

$$h_v^{(k+1)} = \sigma\left(\boxed{W_k} \sum_{u \in N(v)} \frac{h_u^{(k)}}{|N(v)|} + \boxed{B_k} h_v^{(k)}\right), \forall k \in \{0..K-1\}$$

$$z_v = h_v^{(K)}$$

Final node embedding

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

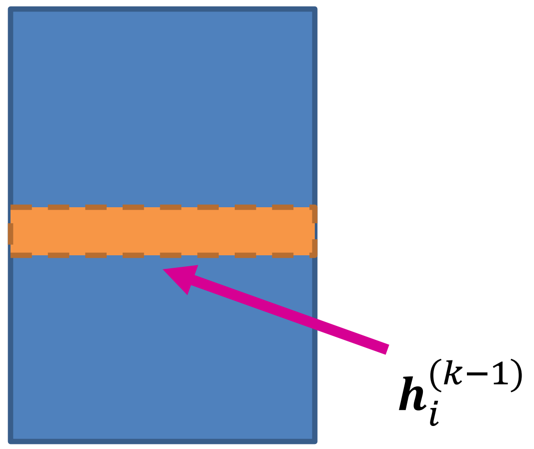
- W_k: weight matrix for neighborhood aggregation
- B_k: weight matrix for transforming hidden vector of self

Matrix Formulation (1)

- **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,**

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

Matrix of hidden embeddings $H^{(k-1)}$

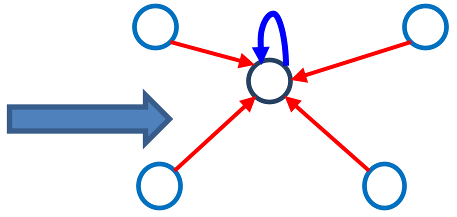


Matrix Formulation (2)

- 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 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 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!**

Unsupervised Training

- **"Similar" nodes have similar embeddings**

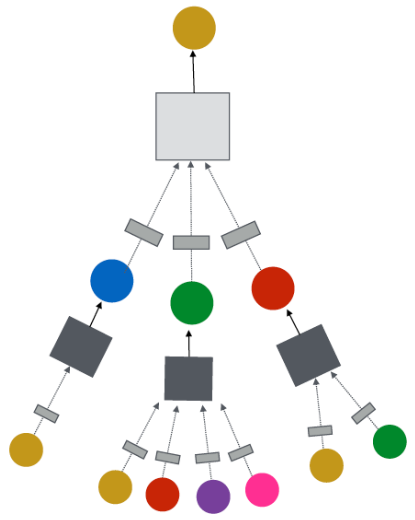
$$\mathcal{L} = \sum_{z_u, z_v} \mathrm{CE}(y_{u,v}, \mathrm{DEC}(z_u, z_v))$$

  - Where $y_{u,v} = 1$ when node u and v are **similar**
  - CE is the cross entropy (Slide 16)
  - DEC is the decoder such as inner product (Lecture 4)
- **Node similarity** can be anything from Lecture 3, e.g., a loss based on:
  - **Random walks** (node2vec, DeepWalk, struc2vec)
  - **Matrix factorization**
  - **Node proximity in the graph**

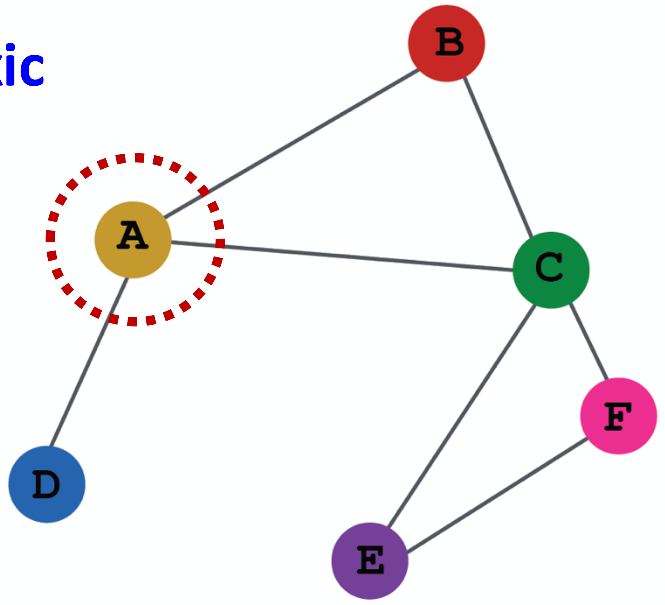
Supervised Training

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

Safe or toxic drug?



Safe or toxic drug?



E.g., a drug-drug interaction network

Supervised Training

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

- Use cross entropy loss (Slide 16)

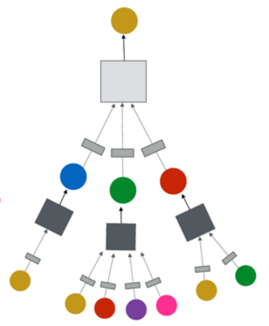
$$\mathcal{L} = -\sum_{v \in V} \boxed{y_v} \log(\sigma(\boxed{\mathrm{z}_v^{\mathrm{T}}}\boxed{\theta})) + (1 - \boxed{y_v}) \log(1 - \sigma(\boxed{\mathrm{z}_v^{\mathrm{T}}}\boxed{\theta}))$$

Encoder output: node embedding

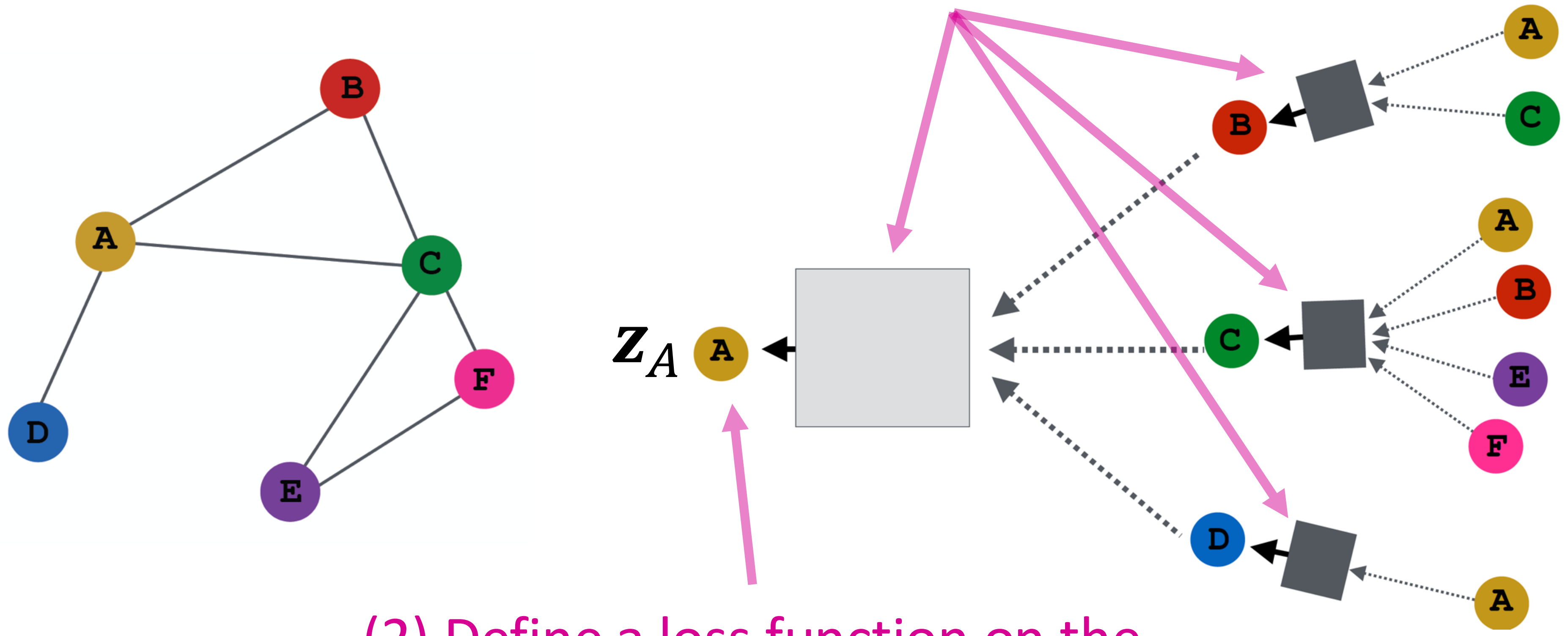
Classification weights

Node class label

Safe or toxic drug?



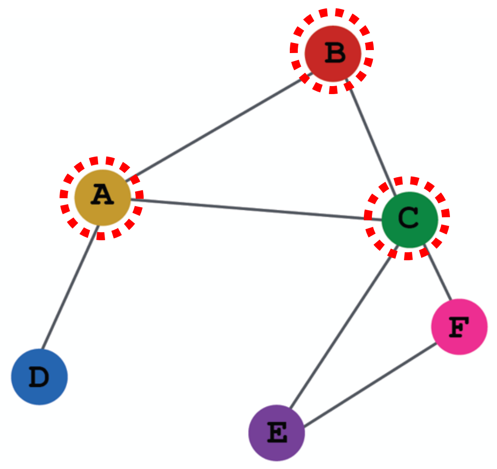
Model Design: Overview



(1) Define a neighborhood aggregation function

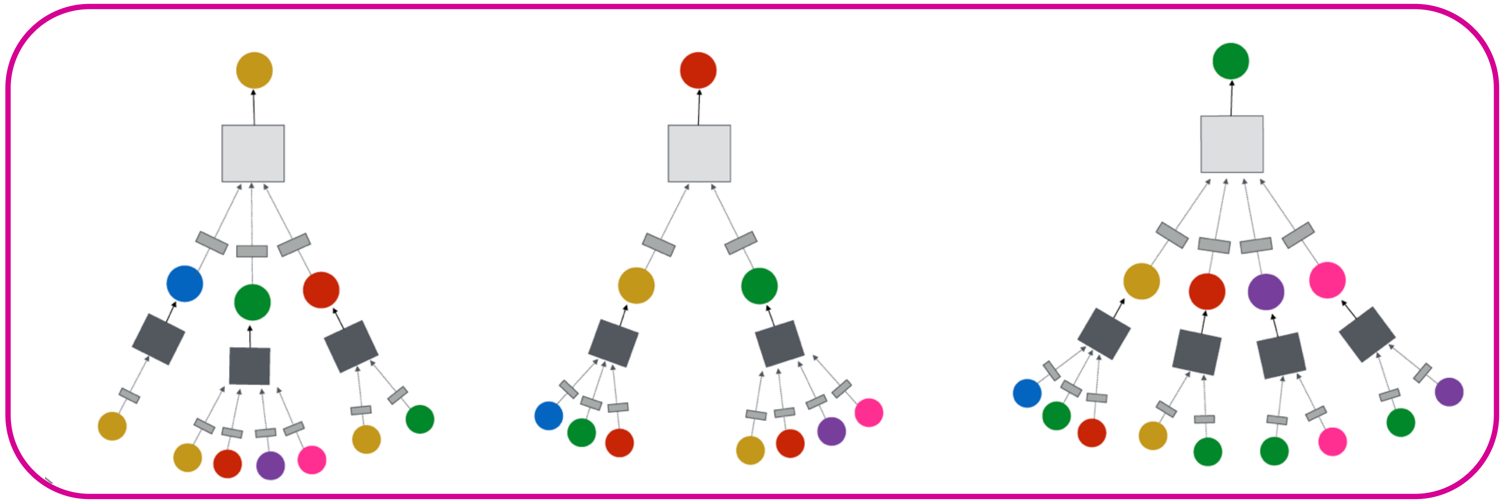
(2) Define a loss function on the embeddings

Model Design: Overview

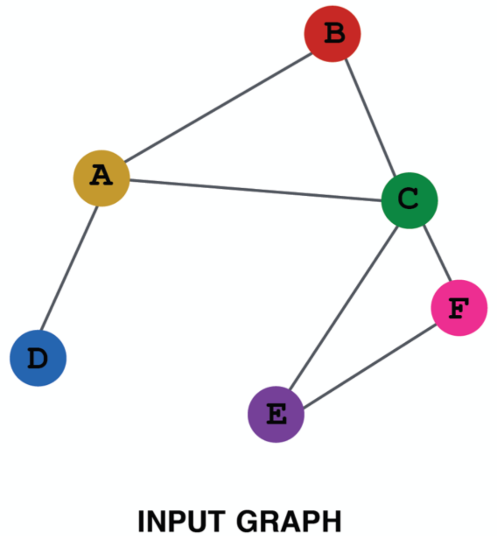


INPUT GRAPH

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

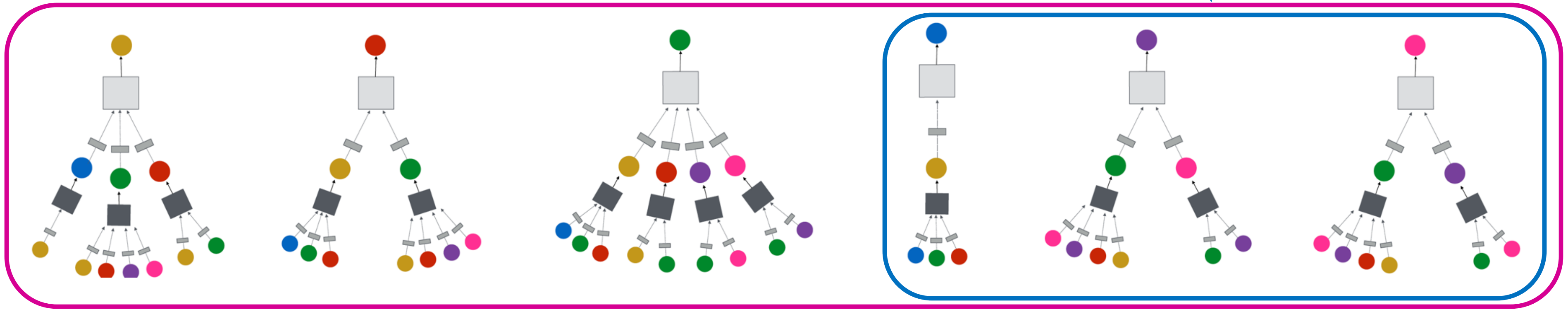


Model Design: Overview



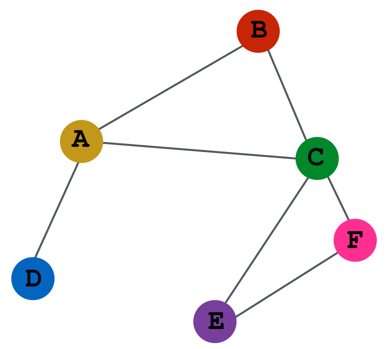
(4) Generate embeddings for nodes as needed

Even for nodes we never trained on!

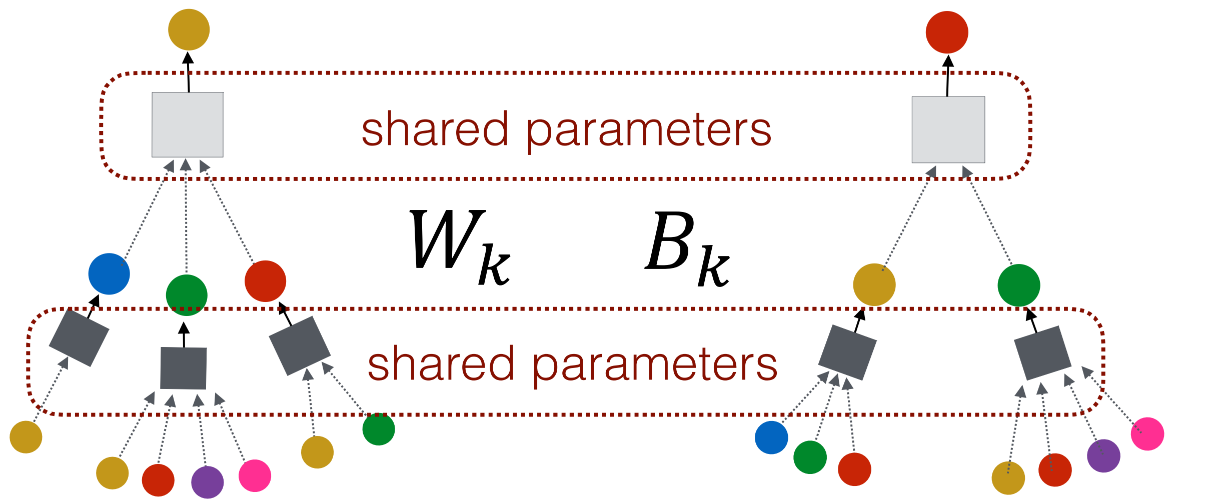


Inductive Capability

- **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**!

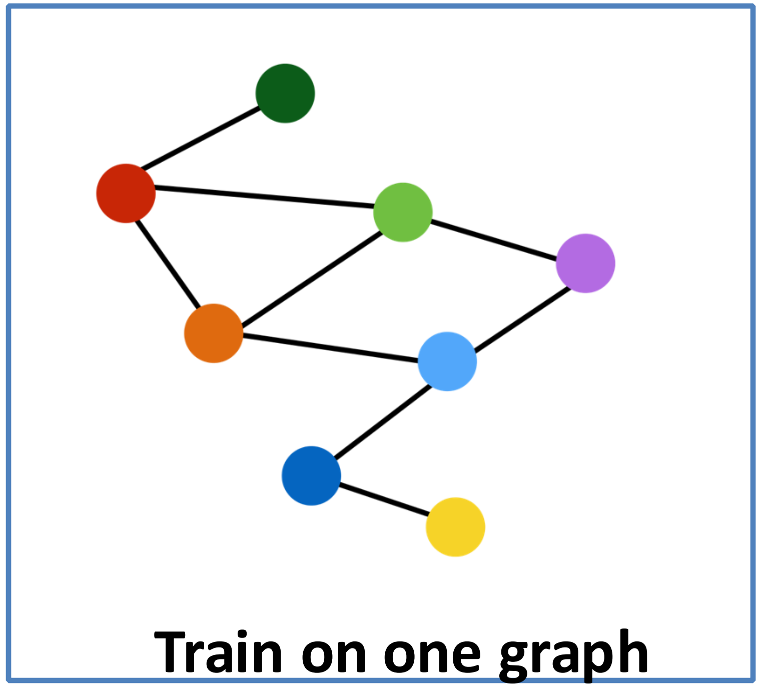


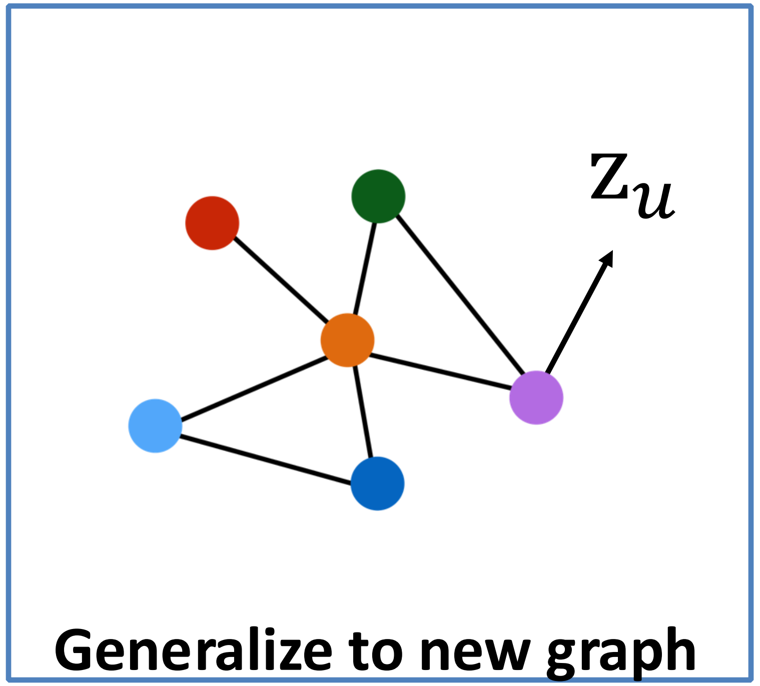
INPUT GRAPH



Compute graph for node A Compute graph for node B

Inductive Capability: New Graphs

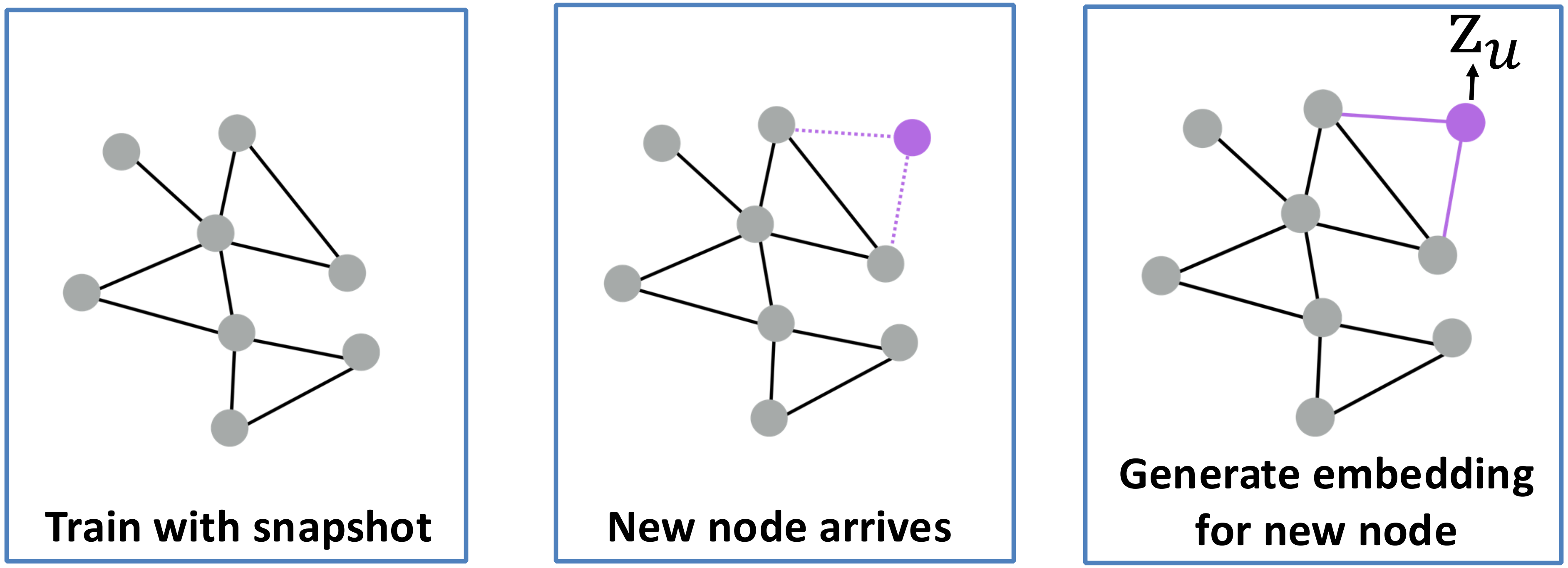




Inductive node embedding → Generalize to entirely unseen graphs

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"

Summary

- **In this lecture, we introduced**
  - Basics of neural networks
    - Loss, Optimization, Gradient, SGD, non-linearity, MLP
  - Idea for Deep Learning for Graphs
    - Multiple layers of embedding transformation
    - At every layer, use the embedding at previous layer as the input
    - Aggregation of neighbors and self-embeddings
  - Graph Convolutional Network
    - Mean aggregation; can be expressed in matrix form
  - GNN is a general architecture
    - CNN can be viewed as a special GNN

Graph Neural Networks: Perspective

Basics of deep learning

Machine Learning as Optimization

- **Supervised learning:** we are given input $\boldsymbol{x}$, and the goal is to predict label $\boldsymbol{y}$.
- **Input x can be:**
  - Vectors of real numbers
  - Sequences (natural language)
  - Matrices (images)
  - Graphs (potentially with node and edge features)
- **We formulate the task as an optimization problem.**

Machine Learning as Optimization

- **Formulate the task as an optimization problem:**

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

Objective function

- Θ: a set of **parameters** we optimize
  - Could contain one or more scalars, vectors, matrices …
  - E.g. $\Theta = \{Z\}$ in the shallow encoder (the embedding lookup)

- $\mathcal{L}$: **loss function**. Example: L2 loss

$$\mathcal{L}\big(\boldsymbol{y}, f(\boldsymbol{x})\big) = \|y - f(x)\|_2$$

  - Other common loss functions:
    - L1 loss, huber loss, max margin (hinge loss), cross entropy …
    - See https://pytorch.org/docs/stable/nn.html#loss-functions

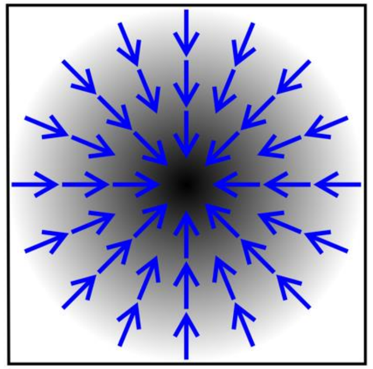
Machine Learning as Optimization

- **How to optimize the objective function?**
- **Gradient vector:** Direction and rate of fastest increase

$$\nabla_{\Theta}\mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial \Theta_1}, \frac{\partial \mathcal{L}}{\partial \Theta_2}, \dots\right)$$

Partial derivative

  - $\Theta_1, \Theta_2 \dots$: components of Θ



https://en.wikipedia.org/wiki/Gradient

- Recall **directional derivative** of a multi-variable function (e.g. $\mathcal{L}$) along a given vector represents the instantaneous rate of change of the function along the vector.
- Gradient is the directional derivative in the **direction of largest increase.**

Gradient Descent

- **Iterative algorithm:** repeatedly update weights in the (opposite) direction of gradients until convergence

$$\Theta \leftarrow \Theta - \eta \nabla_{\Theta} \mathcal{L}$$

- **Training:** Optimize Θ iteratively
  - **Iteration**: 1 step of gradient descent

- **Learning rate (LR) $\boldsymbol{\eta}$:**
  - Hyperparameter that controls the size of gradient step
  - Can vary over the course of training (LR scheduling)
- **Ideal termination condition:** gradient **= 0**
  - In practice, we stop training if it no longer improves performance on **validation set** (part of dataset we hold out from training).

Stochastic Gradient Descent (SGD)

- **Problem with gradient descent:**
  - Exact gradient requires computing $\nabla_{\Theta}\mathcal{L}(\boldsymbol{y}, f(\boldsymbol{x}))$, where $\boldsymbol{x}$ is the **entire** dataset!
    - This means summing gradient contributions over all the points in the dataset
    - Modern datasets often contain billions of data points
    - Extremely expensive for every gradient descent step
- **Solution: Stochastic gradient descent (SGD)**
  - At every step, pick a different **minibatch** $\boldsymbol{\mathcal{B}}$ containing a subset of the dataset, use it as input $\boldsymbol{x}$

Minibatch SGD

- **Concepts:**
  - **Batch size**: the number of data points in a minibatch
    - E.g. number of nodes for node classification task
  - **Iteration**: 1 step of SGD on a minibatch
  - **Epoch**: one full pass over the dataset (# iterations is equal to ratio of dataset size and batch size)

- **SGD is unbiased estimator of full gradient:**
  - But there is no guarantee on the rate of convergence
  - In practice often requires tuning of learning rate
- Common optimizer that improves over SGD:
  - Adam, Adagrad, Adadelta, RMSprop …

Neural Network Function

- **Objective:** $\min_{\Theta} \mathcal{L}(\boldsymbol{y}, f(\boldsymbol{x}))$
- In deep learning, function f can be very complex
- **Example:**
  - To start simple, consider linear function
  $$f(\boldsymbol{x}) = W \cdot \boldsymbol{x}, \qquad \Theta = \{W\}$$
  - Then, if f returns a scalar, then W is a learnable **vector**
  $$\nabla_W f = \left(\frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}, \frac{\partial f}{\partial w_3} \dots\right)$$
  - But, if f returns a vector, then W is the **weight matrix**
  $$\nabla_W f = \begin{bmatrix} \frac{\partial f_1}{\partial w_{11}} & \frac{\partial f_2}{\partial w_{12}} \\ \frac{\partial f_1}{\partial w_{21}} & \frac{\partial f_2}{\partial w_{22}} \end{bmatrix}$$

Jacobian matrix of f

Intuition: Back Propagation

- **Goal:** $\min_{\Theta} \mathcal{L}(\boldsymbol{y}, f(\boldsymbol{x}))$
  - To minimize $\mathcal{L}$, we need to evaluate the gradient: $\nabla_W \mathcal{L} = \left(\frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}, \frac{\partial f}{\partial w_3} \dots\right)$ which means we need to derive derivative of $\mathcal{L}$.
- **Overview of Back-propagation:**
  - $\mathcal{L}$ is composed from some set of predefined building block functions $g(\cdot)$
  - For each such g we also have its derivative g'
  - Then we can automatically compute$\nabla_{\Theta}\mathcal{L}$ by evaluating appropriate funcs. g' on the minibatch $\boldsymbol{\mathcal{B}}$.

Back-propagation

In other words:

$$f(\boldsymbol{x}) = W_2(W_1\boldsymbol{x})$$
$$h(x) = W_1\boldsymbol{x}$$
$$g(z) = W_2 z$$

- **How about a more complex function:**

$$f(\boldsymbol{x}) = W_2(W_1\boldsymbol{x}), \Theta = \{W_1, W_2\}$$

- Recall **chain rule**:

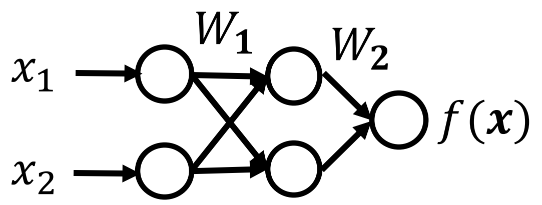
$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\mathrm{d}g}{\mathrm{d}h} \cdot \frac{\mathrm{d}h}{\mathrm{d}x} \text{ or } f'(x) = g'\big(h(x)\big)h'(x)$$

- **Example:** $\nabla_{\boldsymbol{x}} f = \frac{\partial f}{\partial(W_1\boldsymbol{x})} \cdot \frac{\partial(W_1\boldsymbol{x})}{\partial \boldsymbol{x}}$

- **Back-propagation**: Use of **chain rule** to propagate gradients of intermediate steps, and finally obtain gradient of $\mathcal{L}$ w.r.t. Θ.

Back-propagation Example (1)

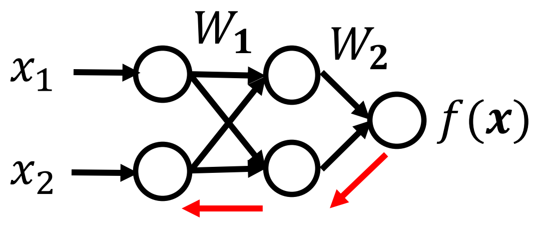
- **Example:** Simple 2-layer linear network
- $f(\boldsymbol{x}) = g\left(h(x)\right) = W_{\mathbf{2}}(W_1\boldsymbol{x})$
- $\mathcal{L} = \sum_{(\boldsymbol{x},\boldsymbol{y})\in\boldsymbol{\mathcal{B}}} \left|\left|\left(\boldsymbol{y}, -f(\boldsymbol{x})\right)\right|\right|_{\mathbf{2}}$
  - The loss $\mathcal{L}$ sums the L2 loss in a minibatch $\boldsymbol{\mathcal{B}}$.
- **Hidden layer:**
  - Intermediate representation of input $\boldsymbol{x}$
  - Here we use $h(x) = W_1\boldsymbol{x}$ to denote the hidden layer
  - $f(\boldsymbol{x}) = W_2 h(x)$



Back-propagation Example (2)

- **Forward propagation:**
  Compute loss starting from input
  - $$\begin{tikzcd} \boldsymbol{x} \arrow[r, "\text{Multiply } W_1"'] & h \arrow[r, "\text{Multiply } W_2"'] & g \arrow[r, "\text{Loss}"'] & \mathcal{L} \end{tikzcd}$$

Remember:
$f(\boldsymbol{x}) = W_2(W_1\boldsymbol{x})$
$h(x) = W_1\boldsymbol{x}$
$g(z) = W_2 z$



- **Back-propagation to compute gradient of**
  $$\Theta = \{W_1, W_2\}$$
  Start from loss, compute the gradient

$$\frac{\partial \mathcal{L}}{\partial W_2} = \frac{\partial \mathcal{L}}{\partial f} \cdot \frac{\partial f}{\partial W_2}, \qquad \frac{\partial \mathcal{L}}{\partial W_1} = \frac{\partial \mathcal{L}}{\partial f} \cdot \frac{\partial f}{\partial W_2} \cdot \frac{\partial W_2}{\partial W_1}$$

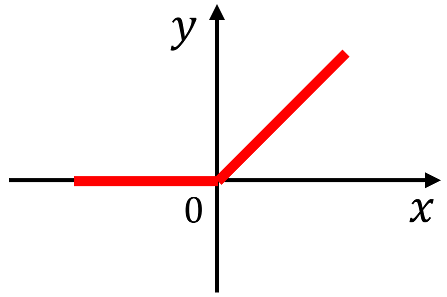
Compute backwards → **Compute backwards** →

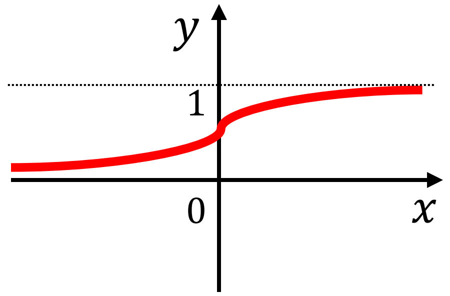
Non-linearity

- Note that in $f(\boldsymbol{x}) = W_2(W_1\boldsymbol{x})$, W_2W_1 is another matrix (vector, if we do binary classification)
  - Hence $f(\boldsymbol{x})$ is still linear w.r.t. $\boldsymbol{x}$ no matter how many weight matrices we compose

- **We introduce non-linearity:**
  - **Rectified linear unit (ReLU)**
    $ReLU(x) = \max(x, 0)$
  - **Sigmoid**

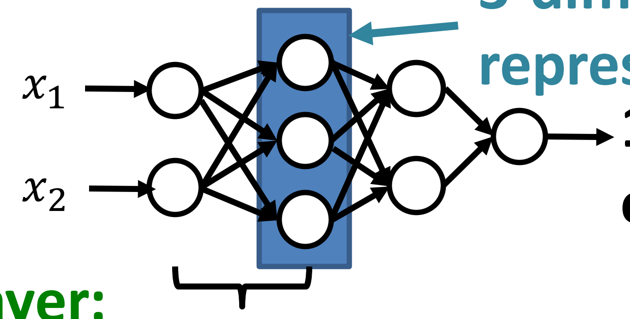
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$





Multi-layer Perceptron (MLP)

- **Each layer of MLP combines linear transformation and non-linearity:**
  - where W_l is weight matrix that transforms hidden representation at layer l to layer $l + 1$
  - b^l is bias at layer l, and is added to the linear transformation of $\boldsymbol{x}^{(\boldsymbol{l})}$
  - σ is non-linearity function (e.g., sigmoid)
- Suppose $\boldsymbol{x}$ is 2-dimensional, with entries x_1 and x_2



Summary

- **Objective function:**

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

- f can be a simple linear layer, an MLP, or other neural networks (e.g., a GNN later)
- Sample a minibatch of input $\boldsymbol{x}$
- **Forward propagation:** Compute $\mathcal{L}$ given $\boldsymbol{x}$
- **Back-propagation:** Obtain gradient $\nabla_{\mathrm{W}}\mathcal{L}$ using a chain rule.
- Use **stochastic gradient descent (SGD)** to optimize for Θ over many iterations.