Graph Neural Networks Introduction

Acknowledgement: Jure Leskovec, Stanford University

Overview

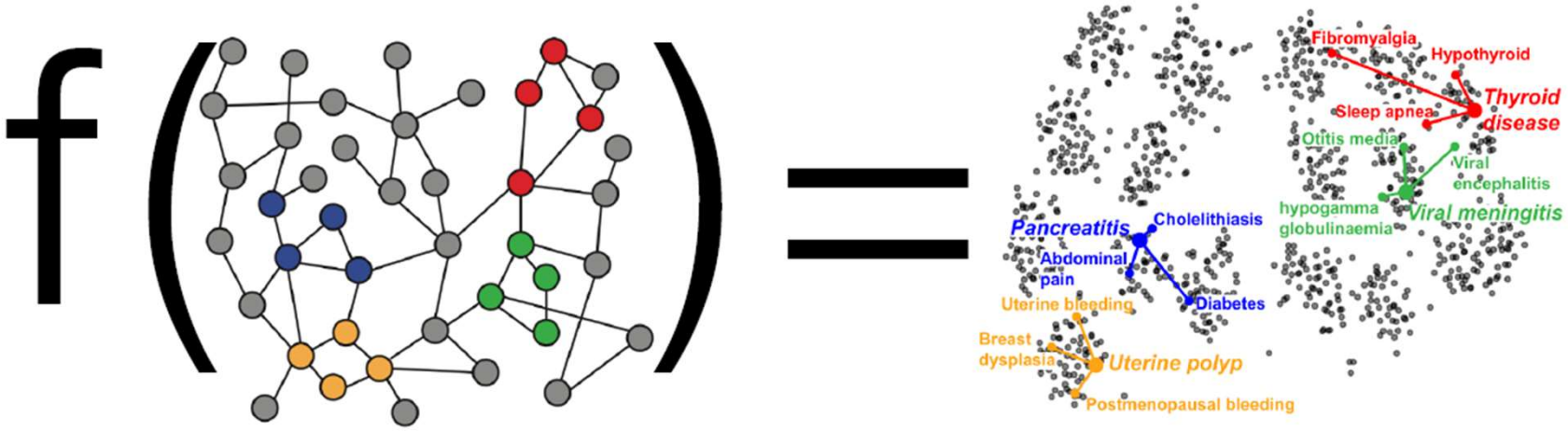
- Shallow Encodings - Limitations
- Deep Learning for Graphs
- Graph Convolution Networks

Overview

- Shallow Encodings - Limitations
- Deep Learning for Graphs
- Graph Convolution Networks

Node Embeddings

- **Intuition:** Map nodes to d-dimensional embeddings such that similar nodes in the graph are embedded close together

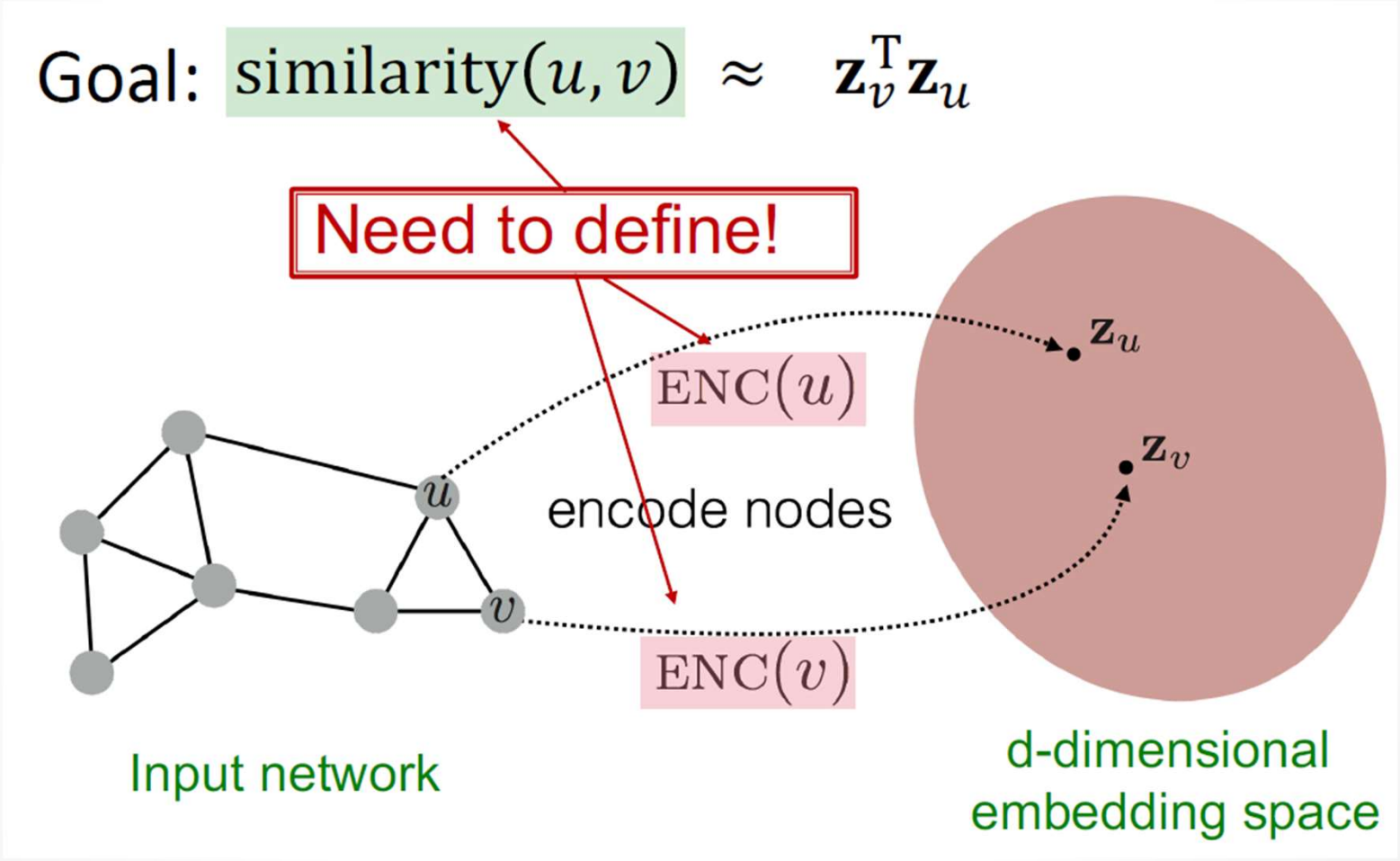


Input graph

2D node embeddings

How to learn mapping function f?

Node Embeddings



Two Key Components

- **Encoder:** maps each node to a low-dimensional vector

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

d-dimensional embedding

node in the input graph

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

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

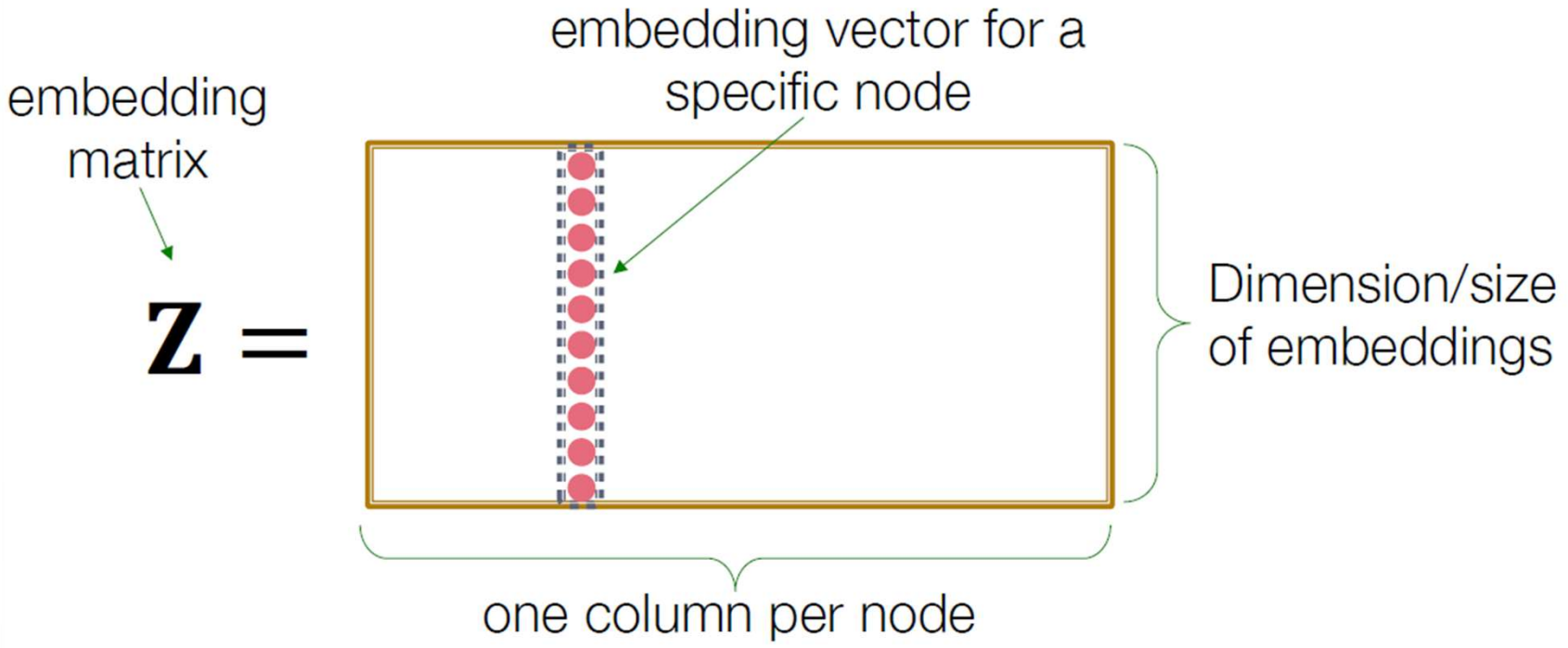
Decoder

Similarity of u and v in the original network

dot product between node embeddings

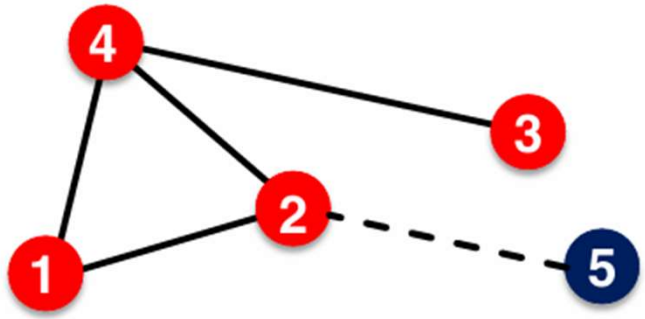
Shallow Encodings

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



Shallow Encodings Limitations - 1

- Cannot obtain embeddings for nodes not in the training set



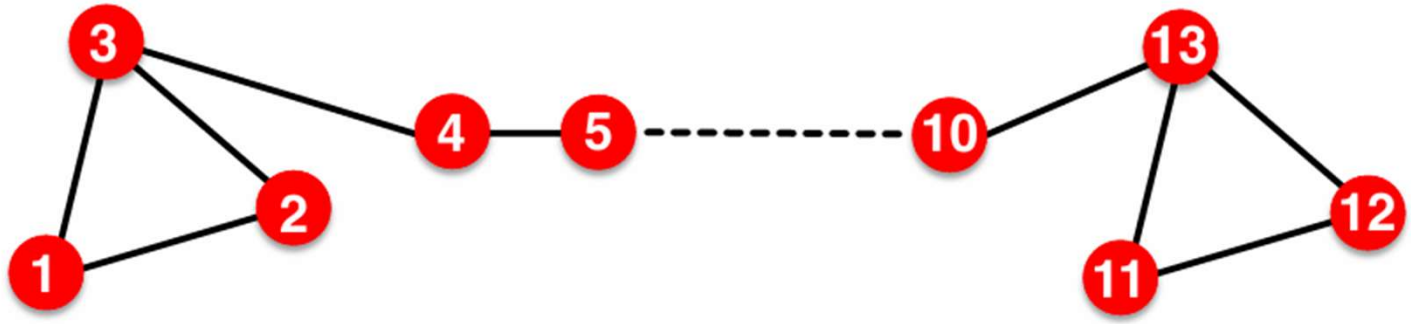
Training set

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.

Shallow Encodings Limitations - 2

- Cannot capture **structural similarity**:

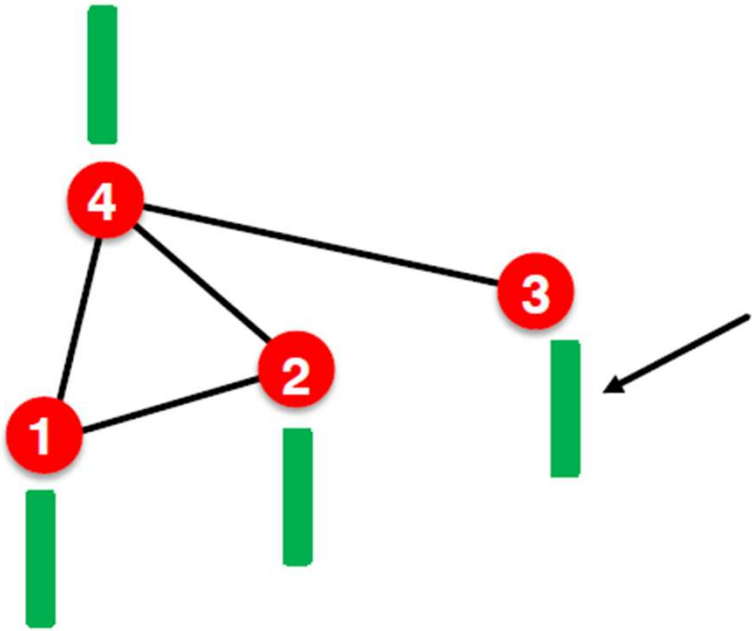


- Node 1 and 11 are **structurally similar** – part of one triangle, degree 2
- However, they have very **different** embeddings
  - It's unlikely that a random walk will reach node 11 from node 1

- **DeepWalk and node2vec do not capture structural similarity**

Shallow Encodings Limitations - 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

Shallow Encodings Limitations - 4

- Every node has its own unique embedding
- No sharing of parameters between nodes
- O(|V|d) parameters needed

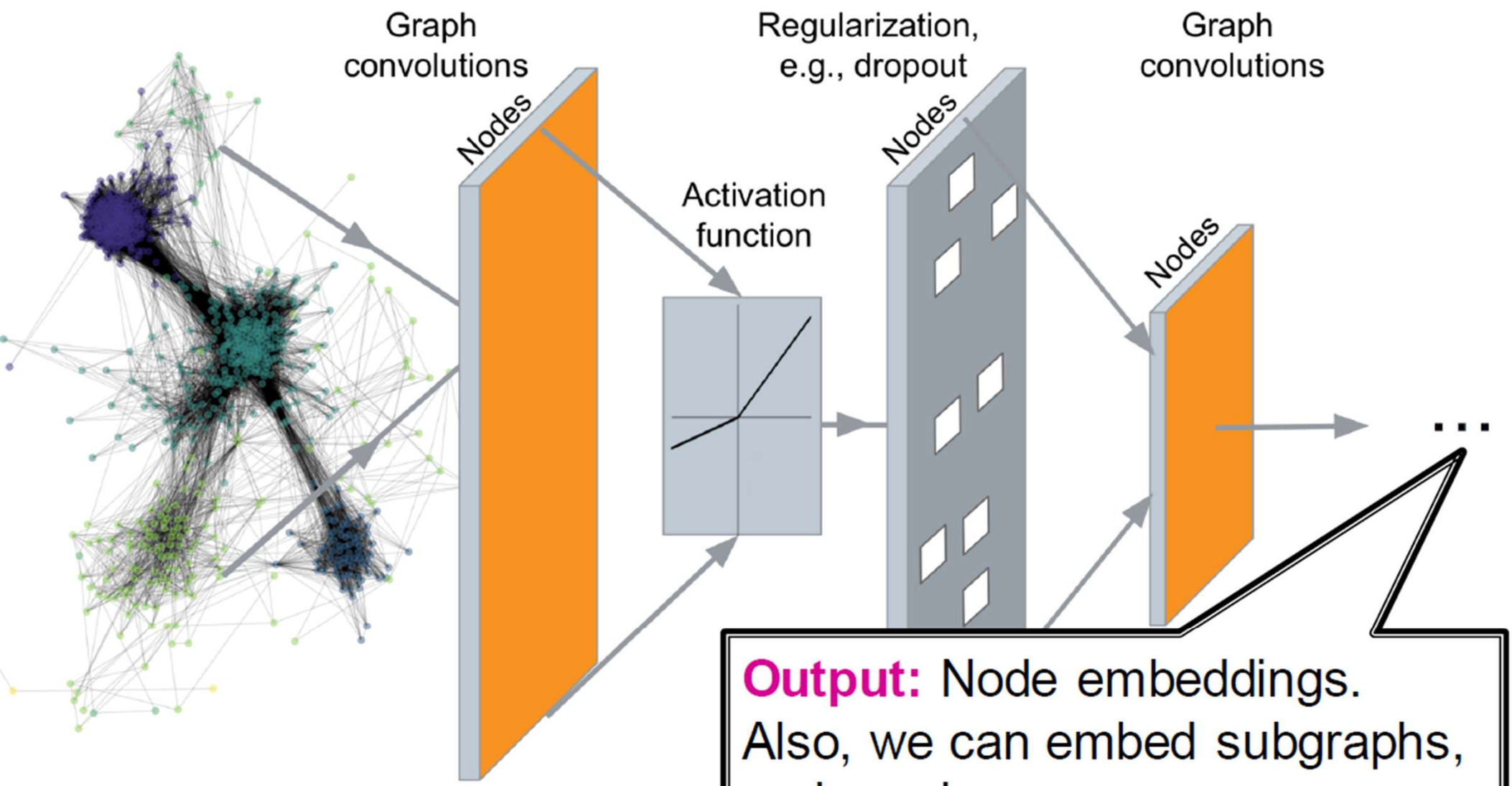
Deep Encoders

- Deep Encoders based on Graph Neural Networks

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

- Deep Encoders can be combined with Graph Similarity functions

Deep Graph Encoders



Tasks on Networks

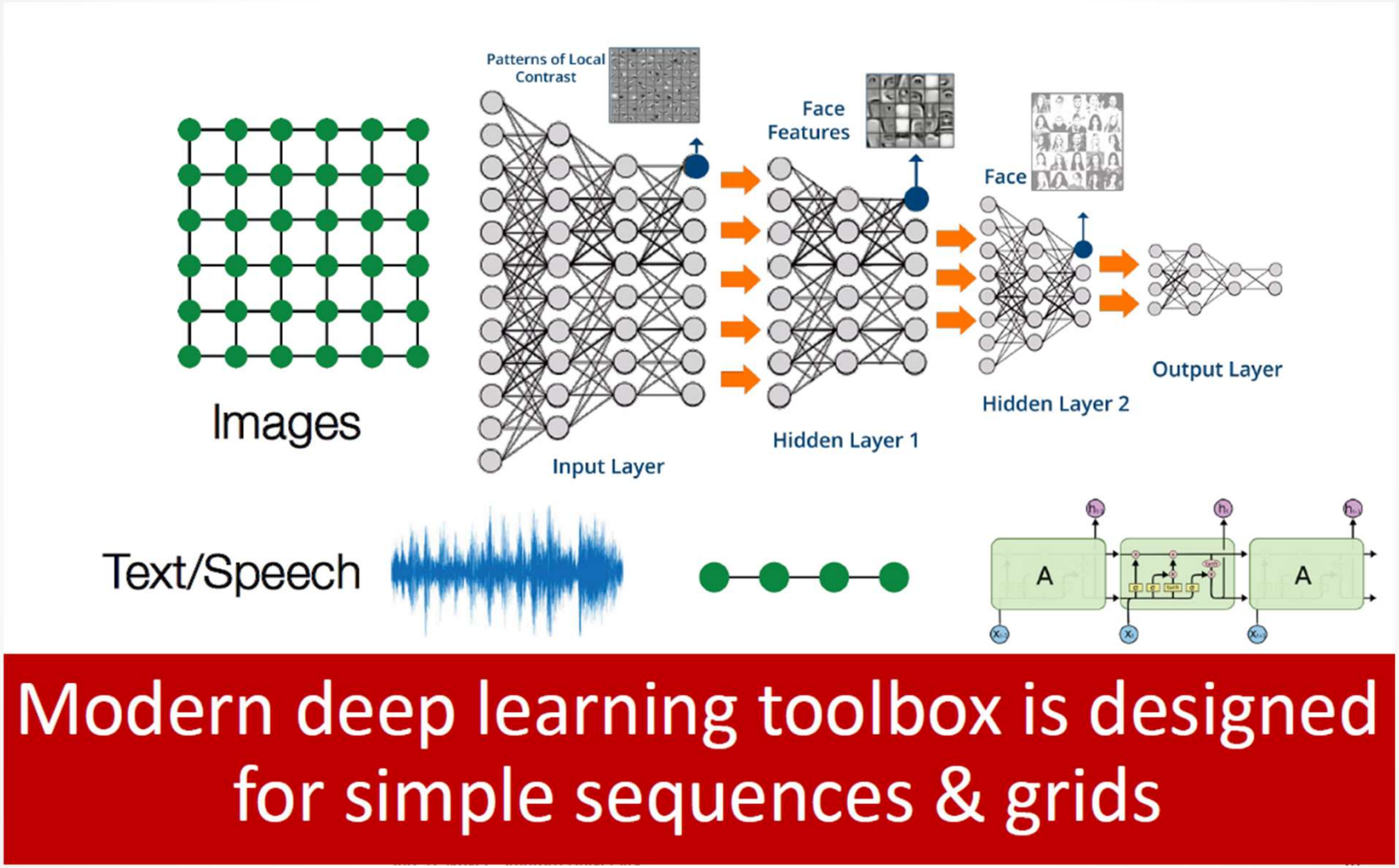
Tasks we will be able to solve:

- 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

Overview

- Shallow Encodings - Limitations
- Deep Learning for Graphs
- Graph Convolution Networks

Modern ML Toolbox



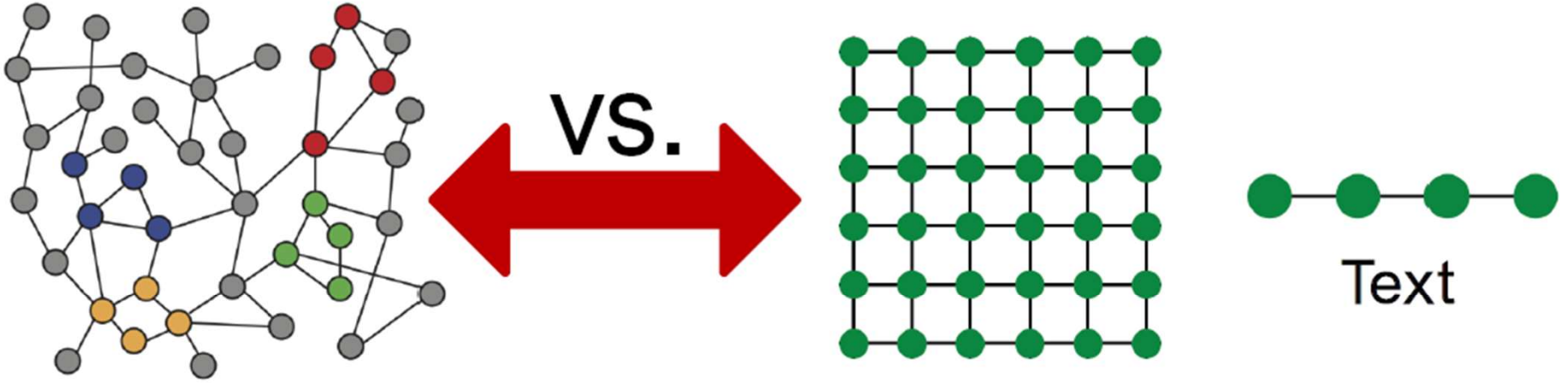
Modern deep learning toolbox is designed for simple sequences & grids

Limitations

- Need new architectures for Graphs

But networks are far more complex!

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



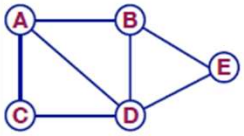
- No fixed node ordering or reference point

Setup

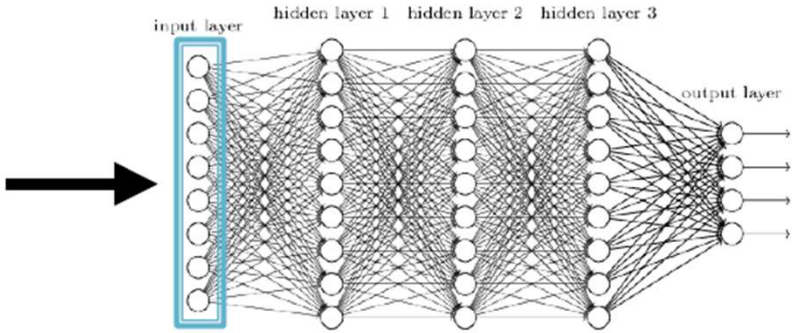
- **Assume we have a graph G:**
  - V is the **vertex set**
  - $\boldsymbol{A}$ is the **adjacency matrix** (assume binary)
  - $\boldsymbol{X} \in \mathbb{R}^{m \times |V|}$ is a matrix of **node features**
  - v: a node in V; $N(v)$: the set of neighbors of v.
  - **Node features:**
    - Social networks: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



	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

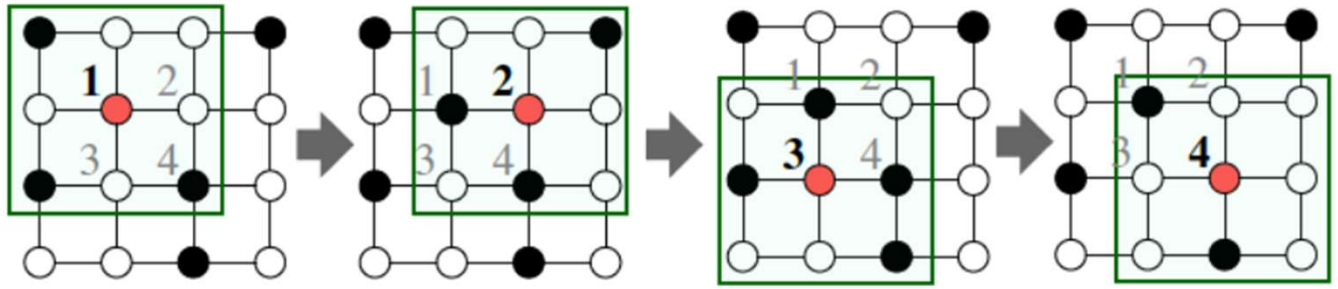


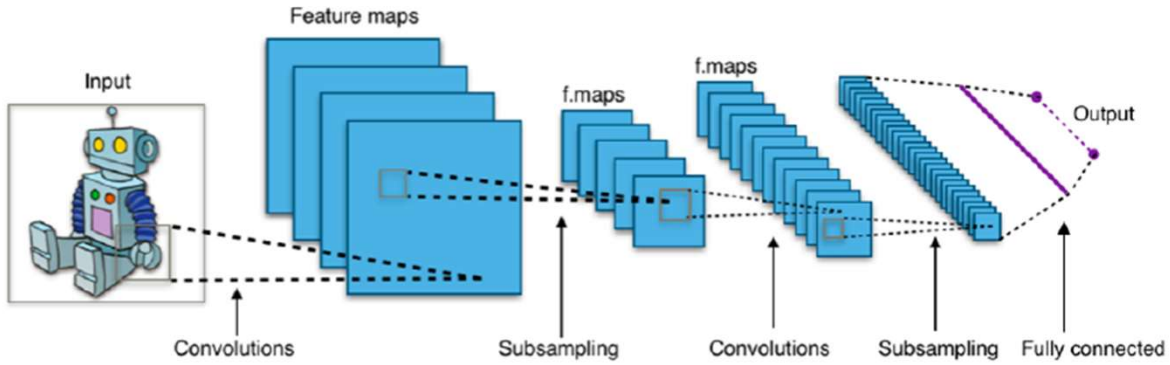
?

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

Idea: Convolutional Network

CNN on an image:





Goal is to generalize convolutions beyond simple lattices

Leverage node features/attributes (e.g., text, images)

Convolutional Neural Network

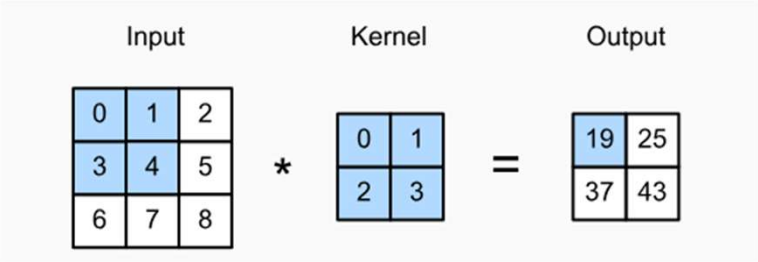
- Inspired by the biology of the visual cortex
  - Local receptive fields are groups of neurons that only respond to a part of what your eyes see (subsampling)
  - They overlap each other to cover the entire visual field (convolutions)
  - They feed into higher layers that identify increasingly complex images
    - Some receptive fields identify horizontal lines, lines at different angles, etc. (filters)
    - These would feed into a layer that identifies shapes
    - Which might feed into a layer that identifies objects
  - For color images, extra layers for red, green, and blue

How do we know that's a Stop sign?

- Individual local receptive fields scan the image looking for edges, and pick up the edges of the stop sign in a layer
- Those edges in turn get picked up by a higher level convolution that identifies the stop sign's shape (and letters, too)
- This shape then gets matched against your pattern of what a stop sign looks like, also using the strong red signal coming from your red layers
- That information keeps getting processed upward until your foot hits the brake!
- A CNN works the same way

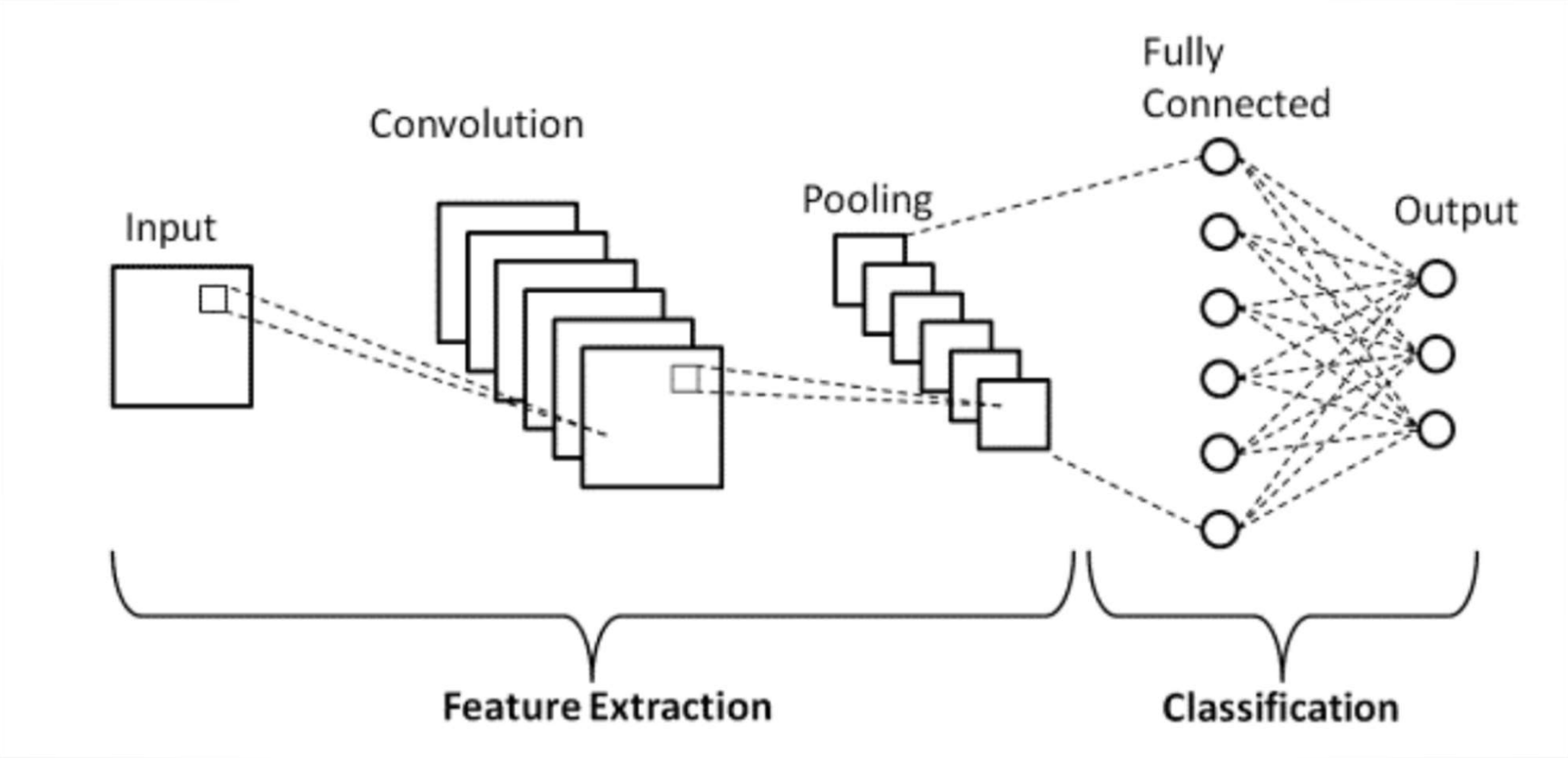
CNN - Components

- **Convolution Layer**
  - This layer is the first layer that is used to extract the various features from the input images. In this layer, the mathematical operation of convolution is performed between the input image and a filter of a particular size MxM. By sliding the filter over the input image, the dot product is taken between the filter and the parts of the input image with respect to the size of the filter (MxM).
  - The output is termed as the Feature map which gives us information about the image such as the corners and edges. Later, this feature map is fed to other layers to learn several other features of the input image.



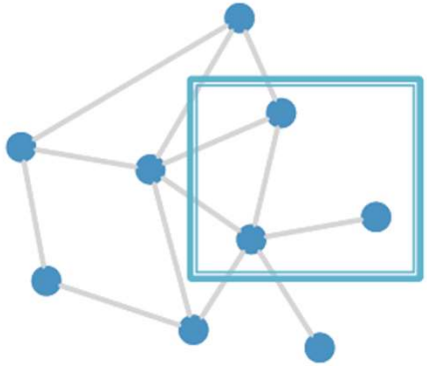
- Pooling Layer
  - A Convolutional Layer is followed by a Pooling Layer. The primary aim of this layer is to decrease the size of the convolved feature map to reduce the computational costs. This is performed by decreasing the connections between layers and independently operates on each feature map. Depending upon method used, there are several types of Pooling operations. It basically summarizes the features generated by a convolution layer.
  - In Max Pooling, the largest element is taken from feature map. Average Pooling calculates the average of the elements in a predefined sized Image section. The total sum of the elements in the predefined section is computed in Sum Pooling. The Pooling Layer usually serves as a bridge between the Convolutional Layer and the FC Layer.
- Fully Connected Layer
  - The Fully Connected (FC) layer consists of the weights and biases along with the neurons and is used to connect the neurons between two different layers. These layers are usually placed before the output layer and form the last few layers of a CNN Architecture.
  - In this, the input image from the previous layers are flattened and fed to the FC layer. The flattened vector then undergoes few more FC layers where the mathematical functions operations usually take place. In this stage, the classification process begins to take place.

CNN - Architecture

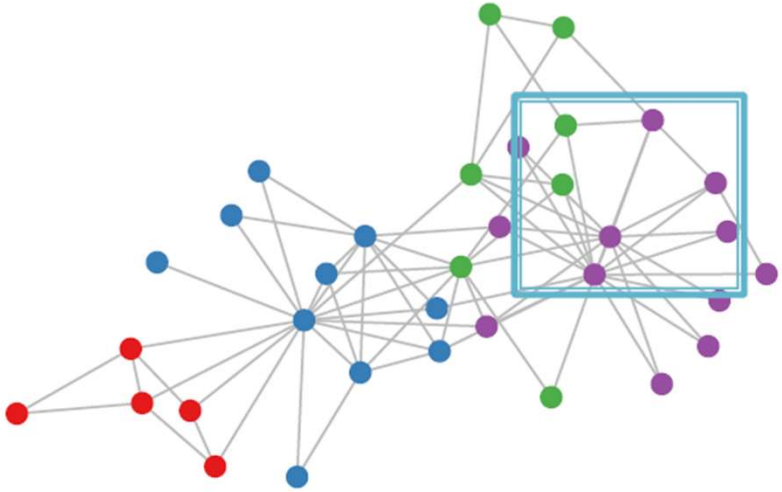


Real-world Graphs

But our graphs look like this:



or this:

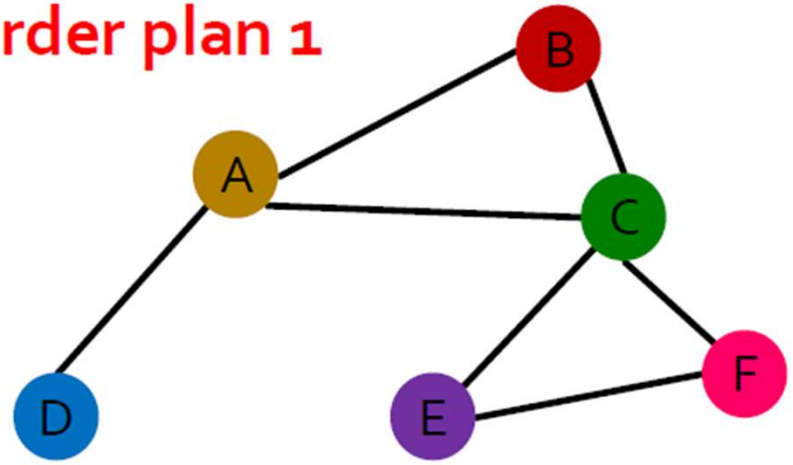


- There is no fixed notion of locality or sliding window on the graph
- Graph is permutation invariant

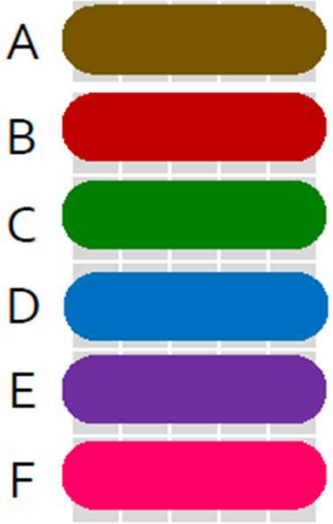
Permutation Invariance

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

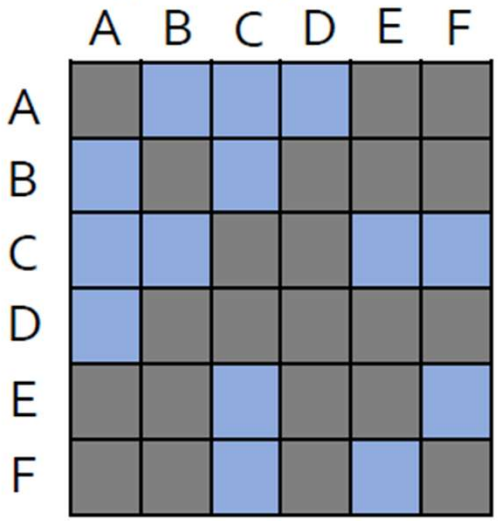
Order plan 1



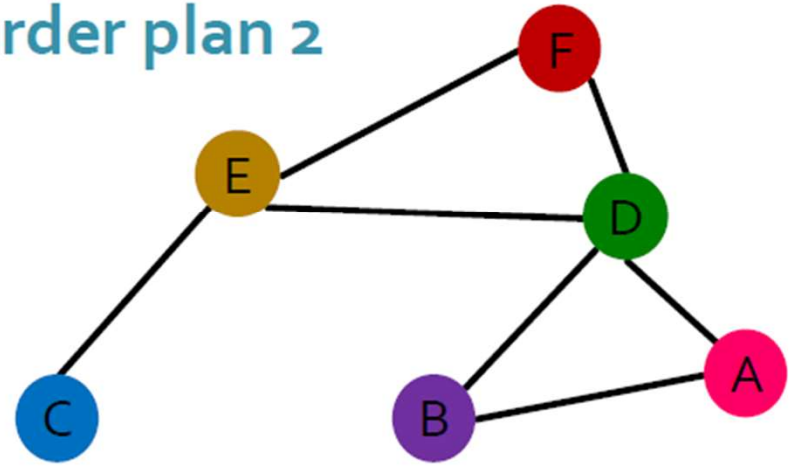
Node features X_1



Adjacency matrix A_1

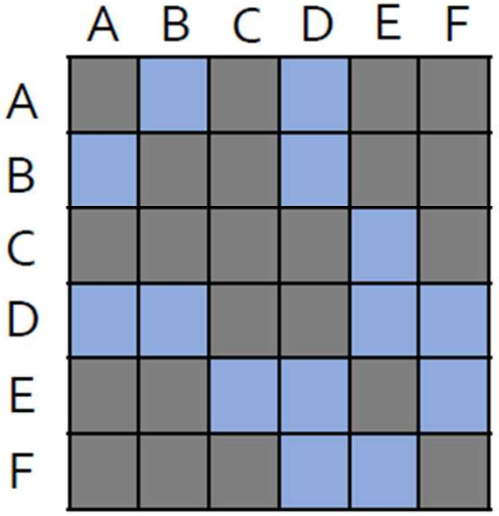


Order plan 2



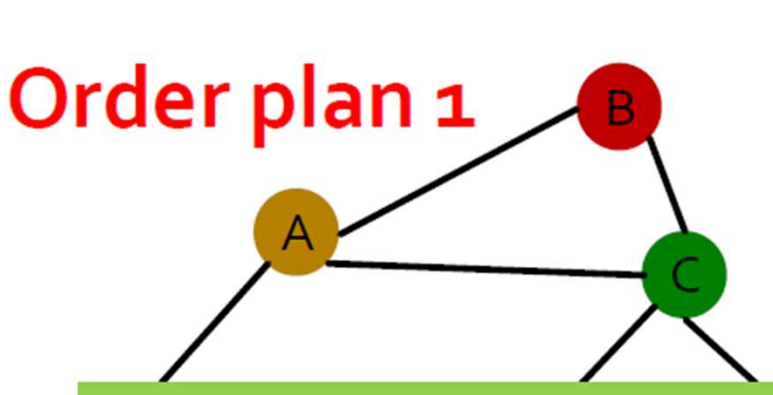
Node features X_2

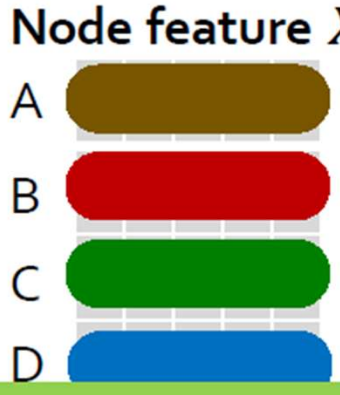
Adjacency matrix A_2

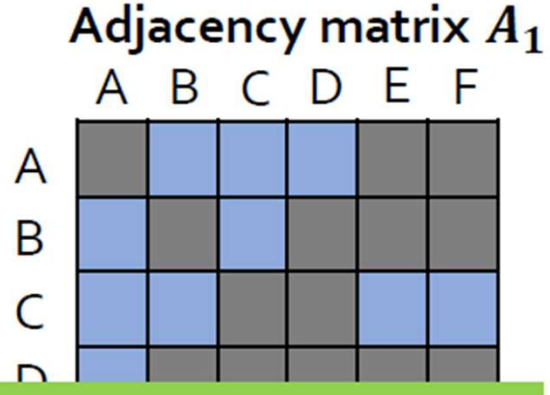


Permutation Invariance

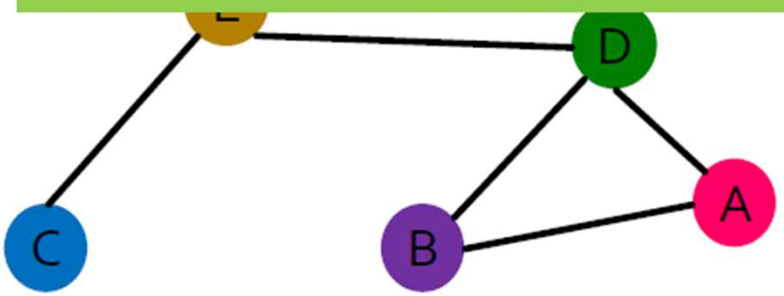
- **Graph does not have a canonical order of the nodes!**

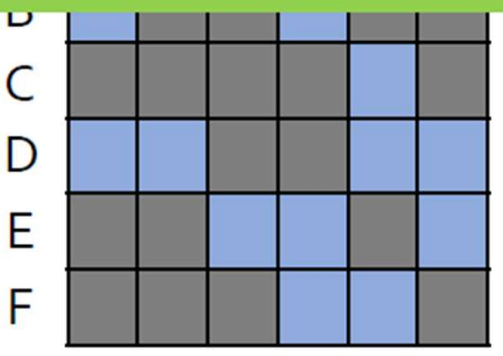






Graph and node representations should be the same for Order plan 1 and Order plan 2





Permutation Invariance

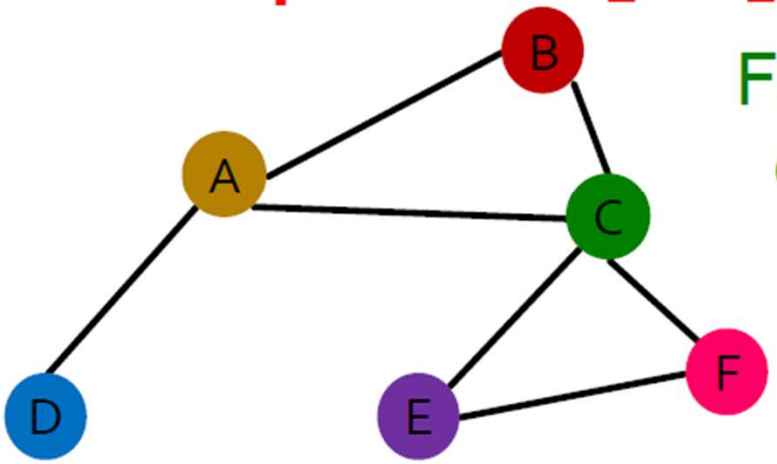
What does it mean by "graph representation is same for two order plans"?

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

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

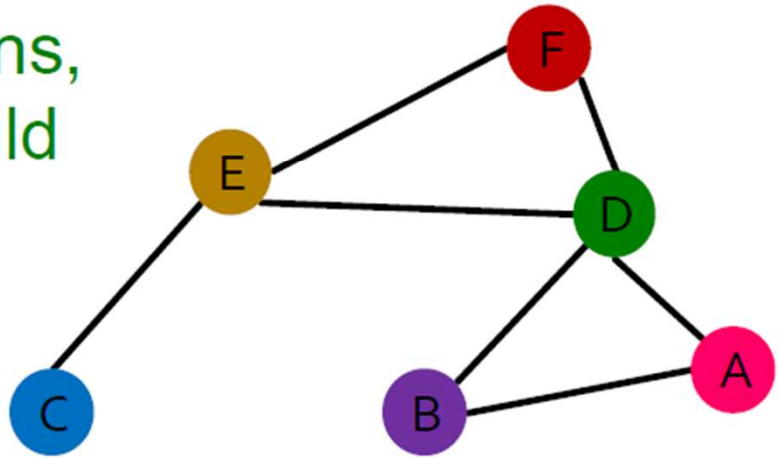
$\boldsymbol{A}$ is the adjacency matrix
$\boldsymbol{X}$ is the node feature matrix

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 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 $\mathbb{R}^d$.

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

- 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.

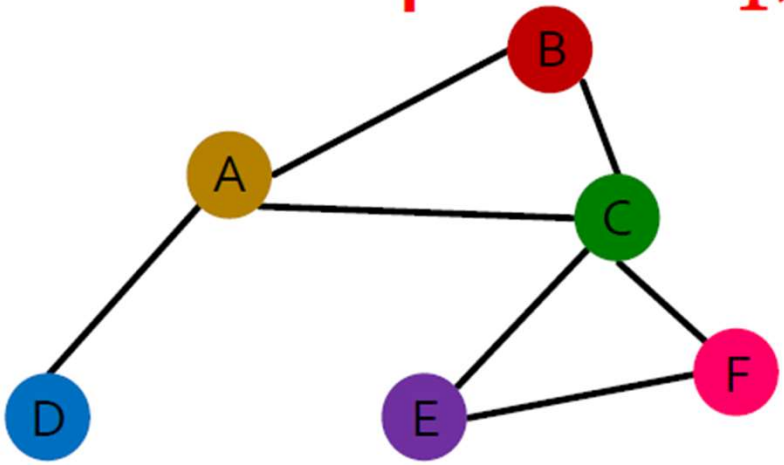
- **Definition:** For any graph function f: $\mathbb{R}^{|V|\times m} \times \mathbb{R}^{|V|\times|V|} \to \mathbb{R}^d$, f is **permutation-invariant** if $f(A, X) = f(PAP^T, PX)$ for any permutation P.

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

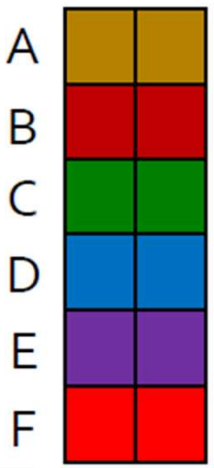
Permutation Equivariance

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

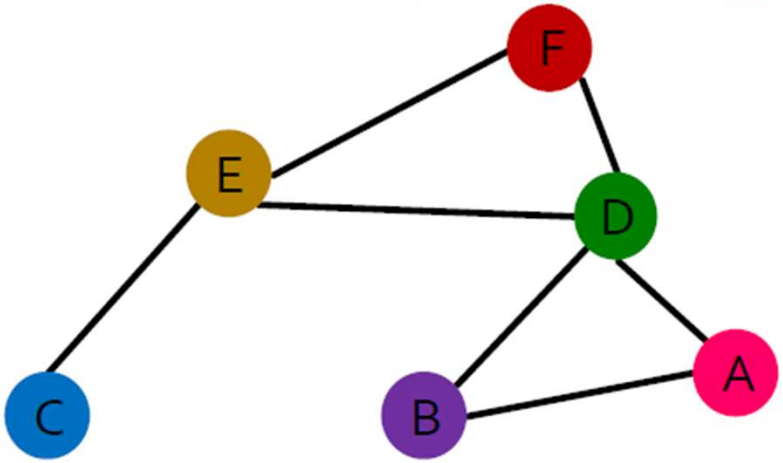
Order plan 1: A_1, X_1



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



Order plan 2: A_2, X_2

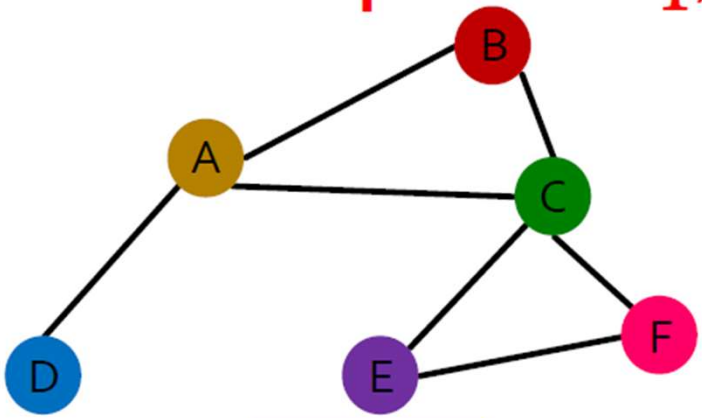


$$f(\boldsymbol{A}_2, \boldsymbol{X}_2) =$$

Permutation Equivariance

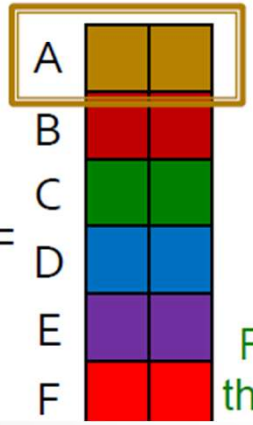
For node representation: We learn a function f that maps nodes of G to a matrix $\mathbb{R}^{m \times d}$.

Order plan 1: A_1, X_1

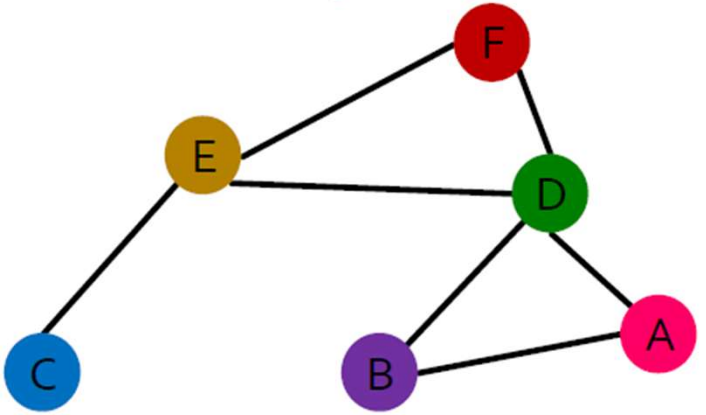


Representation vector of the brown node A

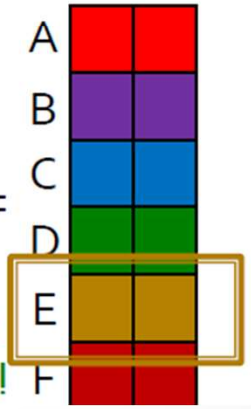
$$f(A_1, X_1) =$$



Order plan 2: A_2, X_2



$$f(A_2, X_2) =$$



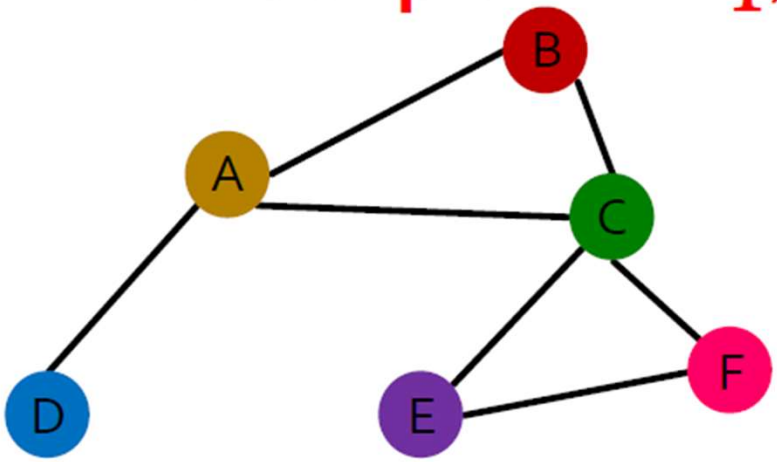
Representation vector of the brown node E

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

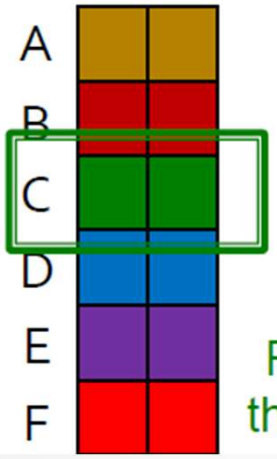
Permutation Equivariance

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

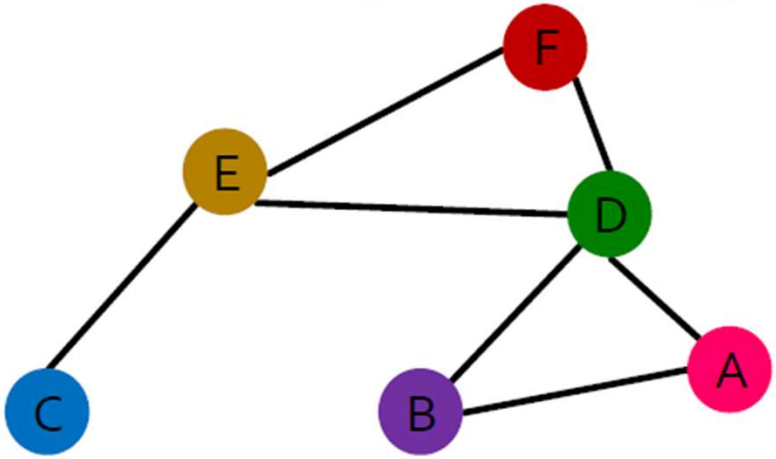


$f(A_1, X_1) =$

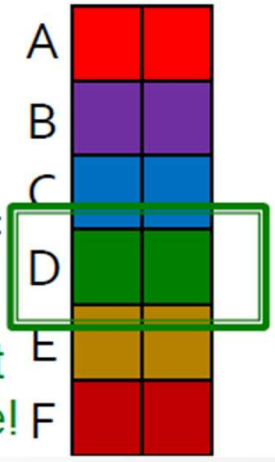


Representation vector of the green node C

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.

Invariance & 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**

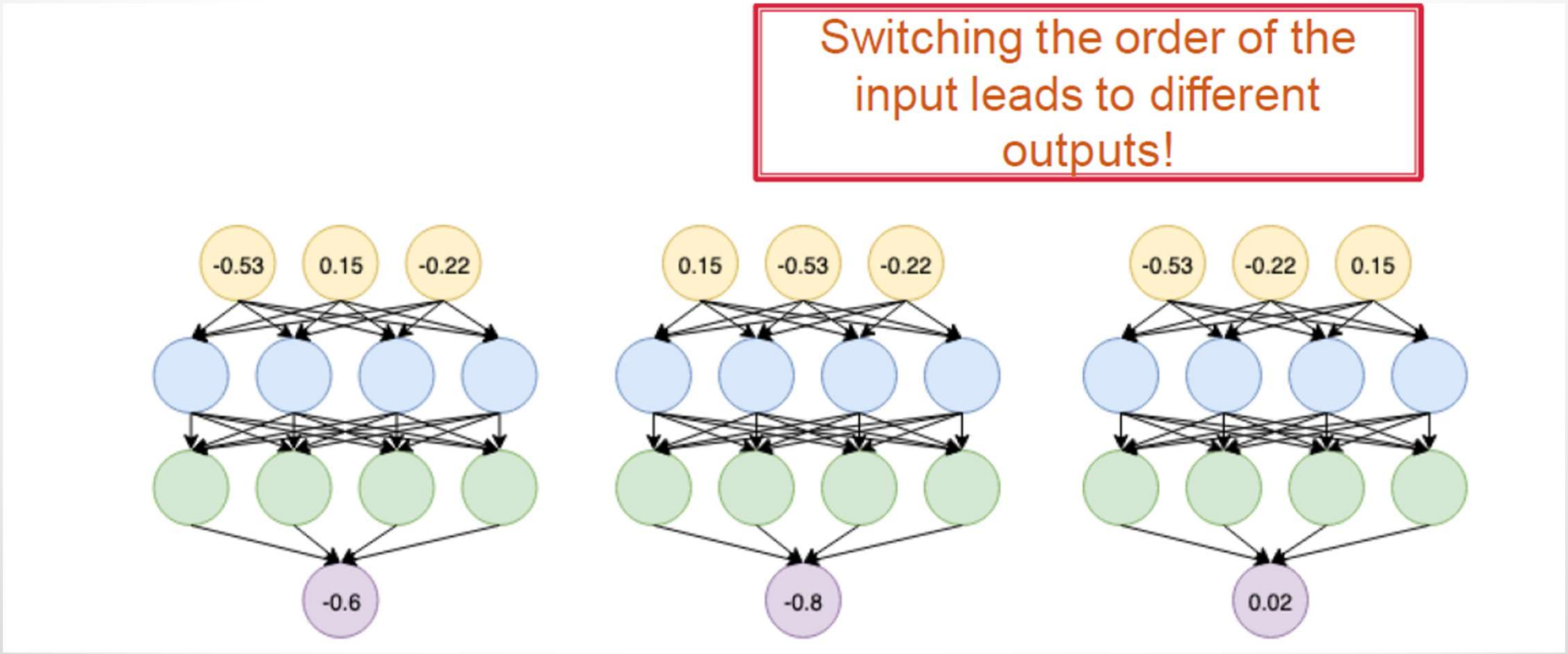
$$Pf(A, X) = f(PAP^T, PX)$$

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

- **Invariance**: Output does not depend on the representation of the input
- **Equivariance**: Output must be permuted when the input is permuted

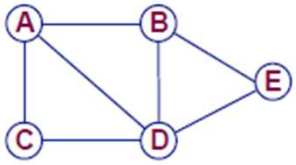
Invariance & Equivariance

- Are other neural network architectures, e.g., MLPs, permutation invariant / equivariant?
- **No.**

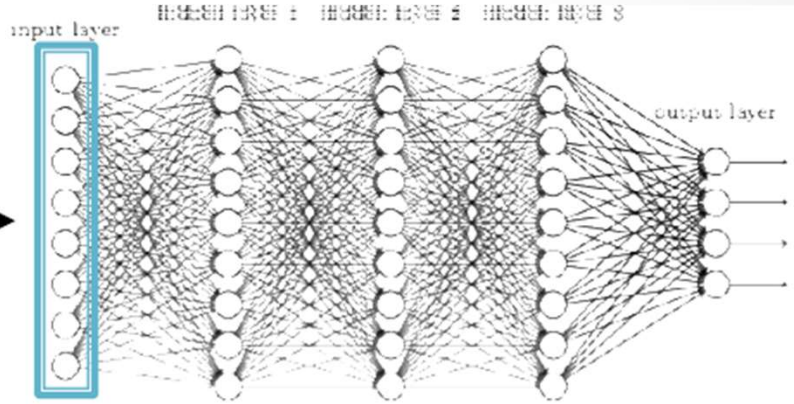


Invariance & Equivariance

- Are other neural network architectures, e.g., MLPs, permutation invariant / equivariant?
- **No.**



	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



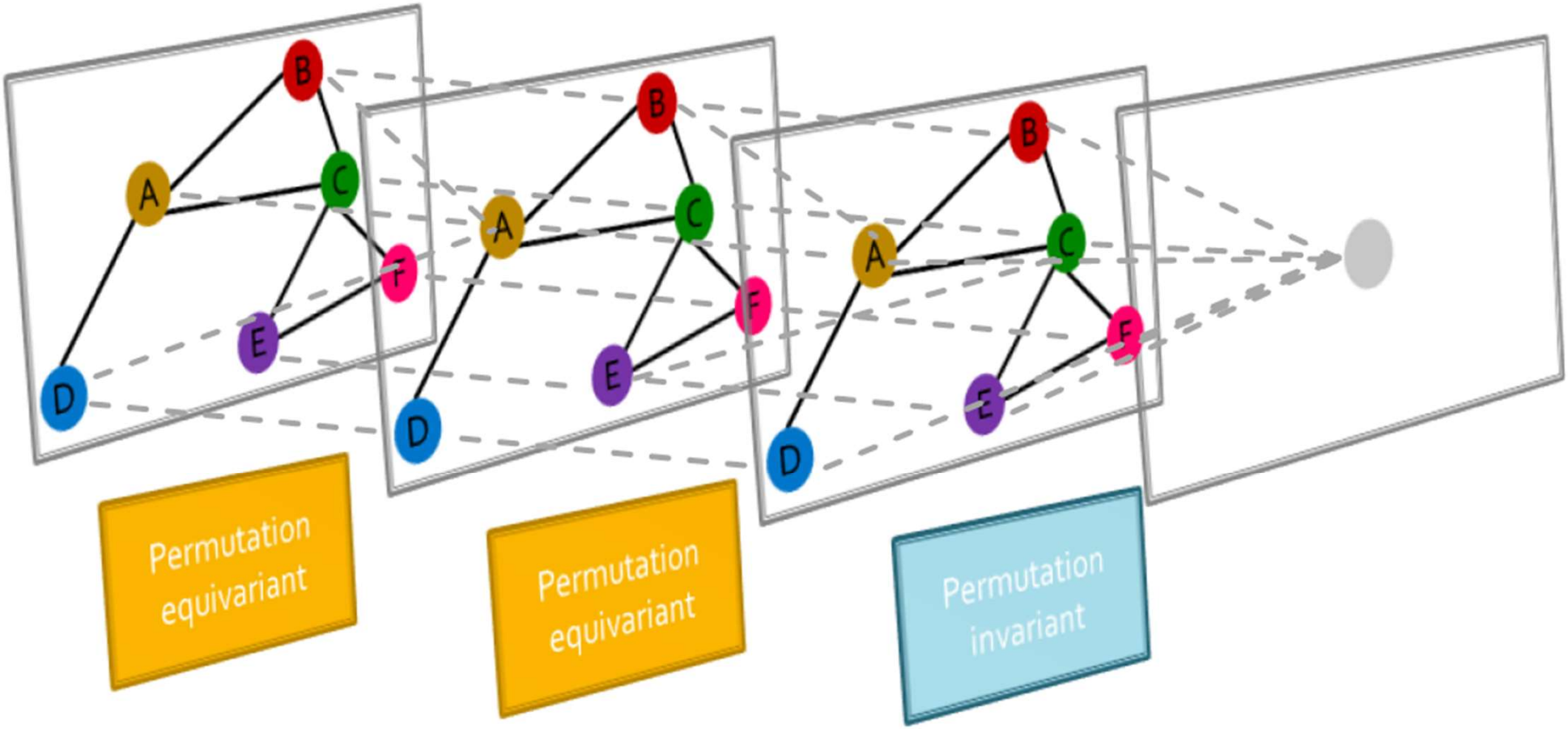
?

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

Design graph neural networks that are permutation invariant / equivariant

Graph Neural Networks

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

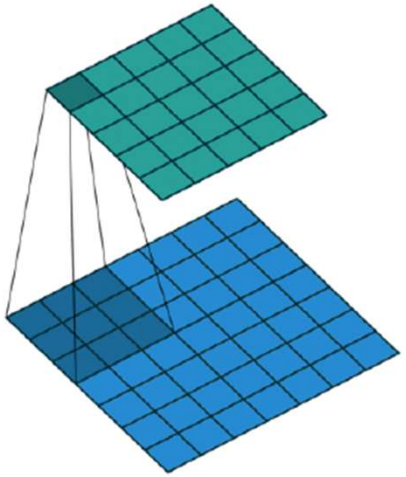


Overview

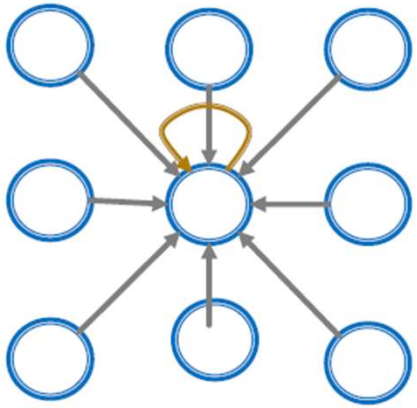
- Shallow Encodings - Limitations
- Deep Learning for Graphs
- Graph Convolution Networks

From Images to Graphs

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



Image



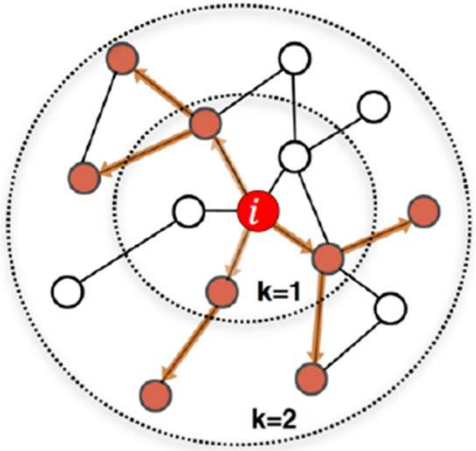
Graph

Idea: transform information at the neighbors and combine it:

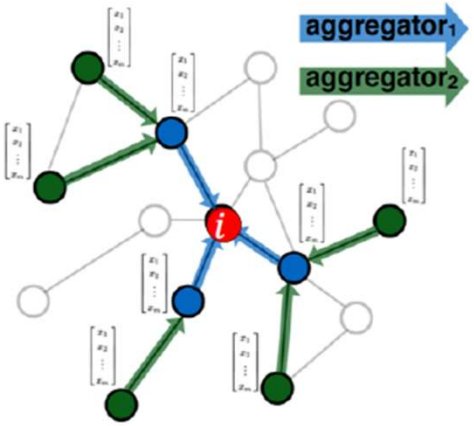
- Transform "messages" h_i from neighbors: $W_i\ h_i$
- Add them up: $\sum_i W_i\ h_i$

Graph Convolutional Network

Idea: Node's neighborhood defines a computation graph



Determine node computation graph



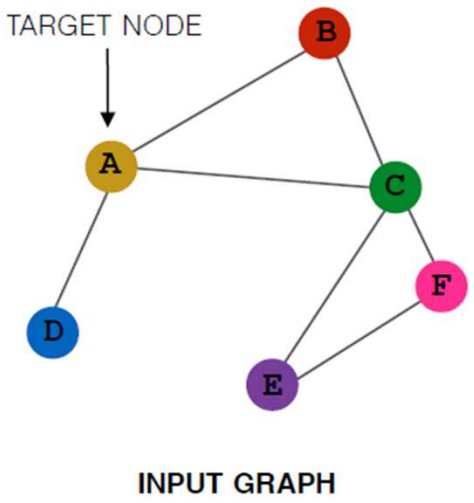
Propagate and transform information

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

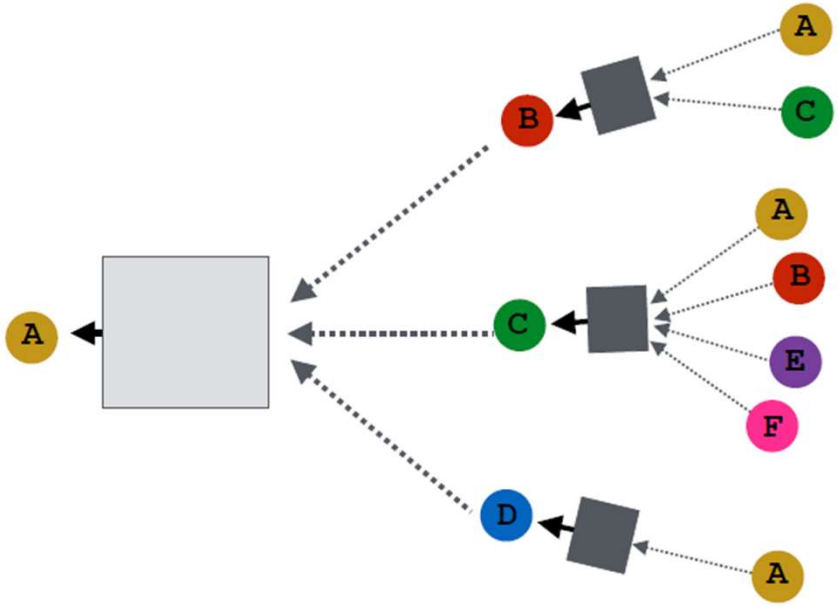
[Kipf and Welling, ICLR 2017]

Idea: Aggregate Neighbors

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

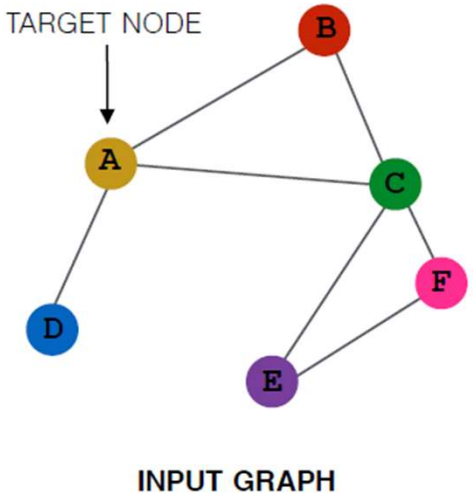


INPUT GRAPH

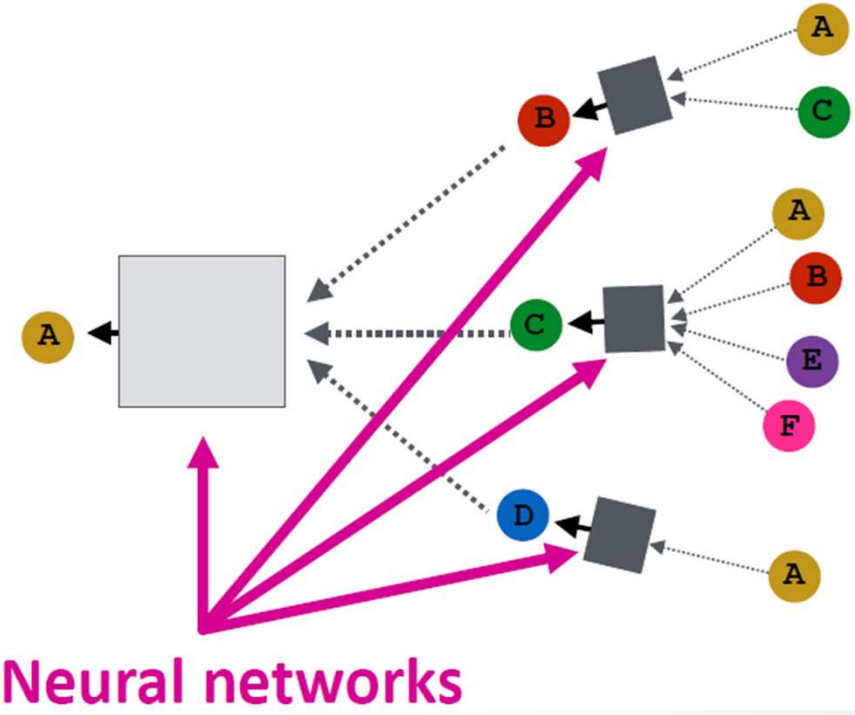


Idea: Aggregate Neighbors

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



INPUT GRAPH

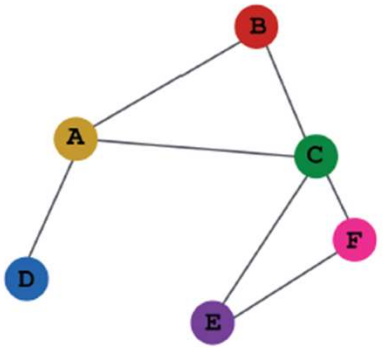


Neural networks

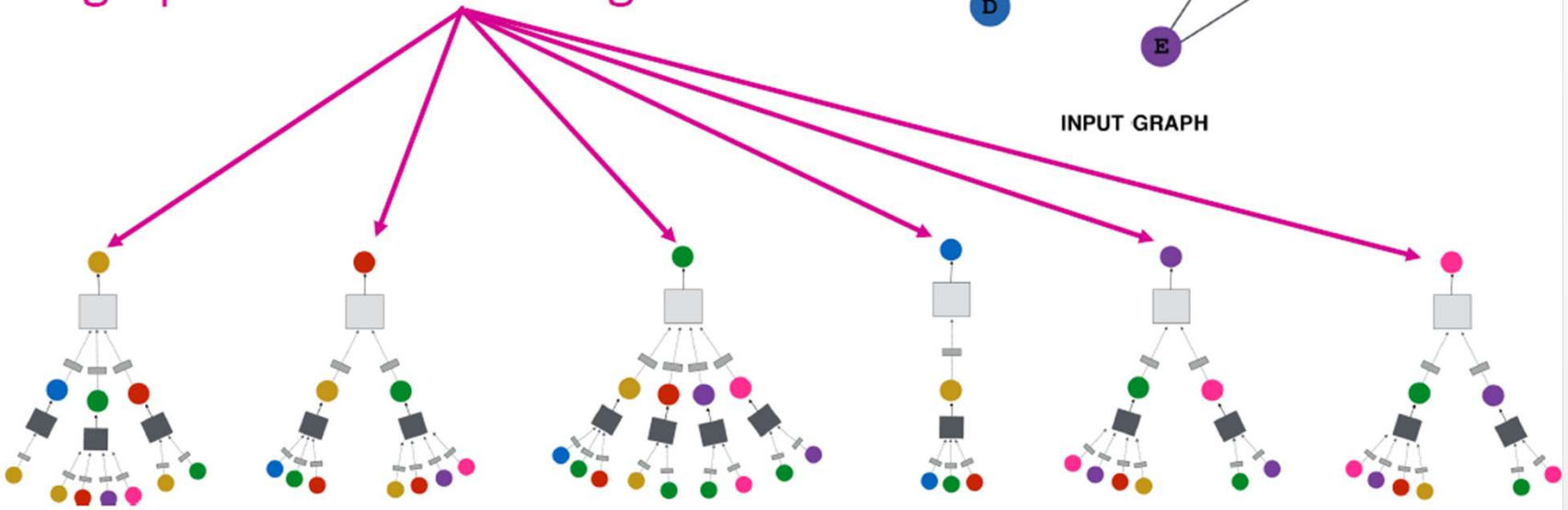
Idea: Aggregate Neighbors

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

Every node defines a computation graph based on its neighborhood!

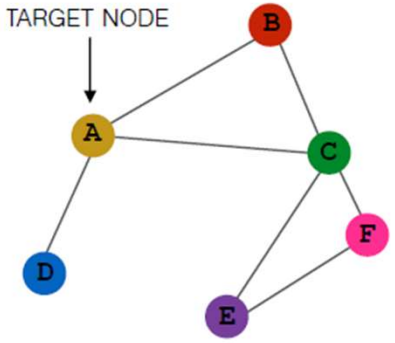


INPUT GRAPH

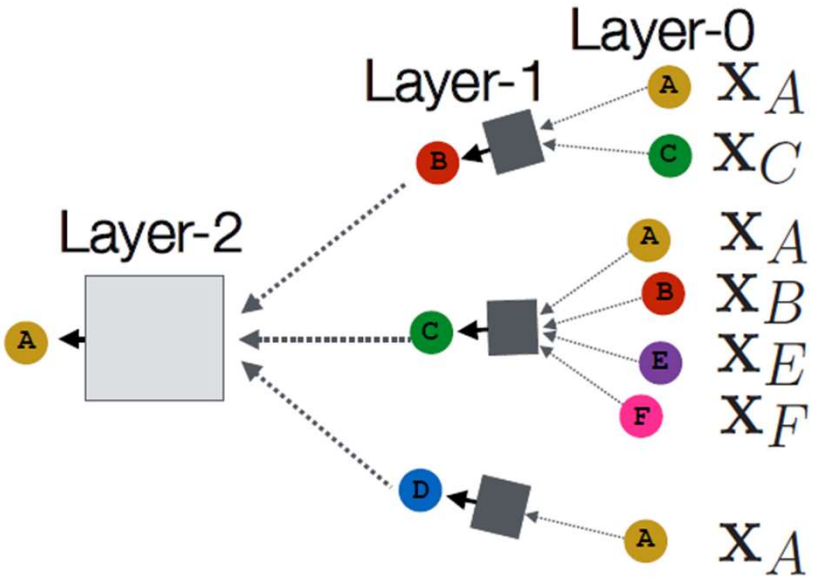


Deep Model: Many Layers

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

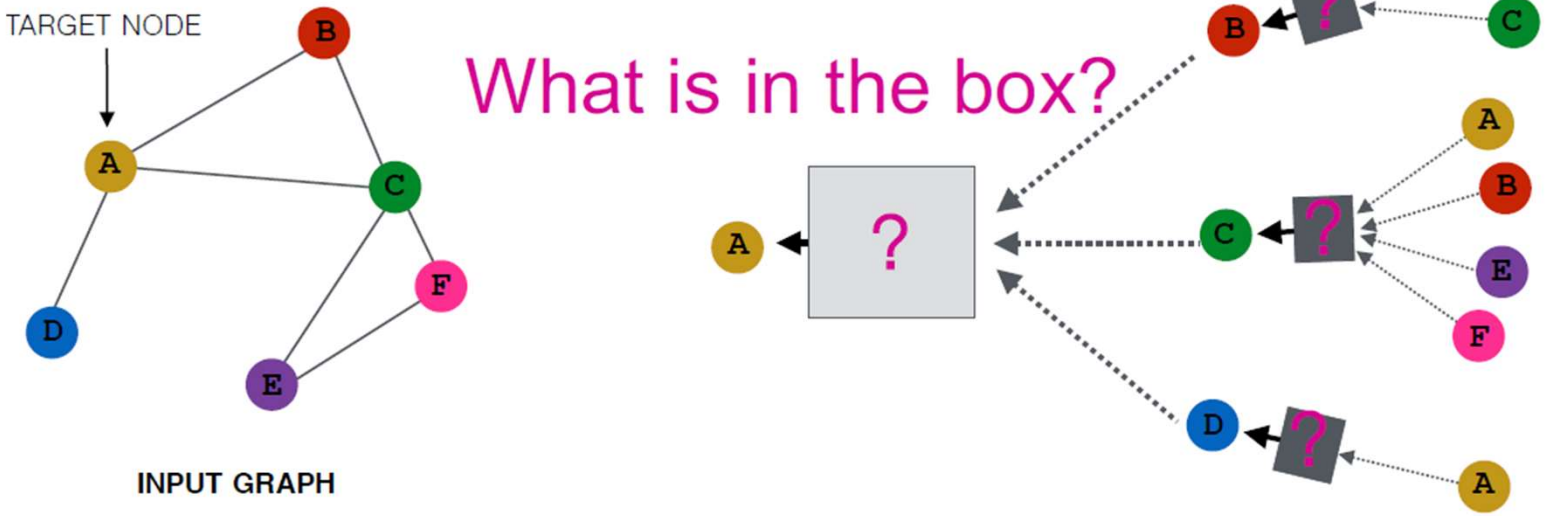


INPUT GRAPH



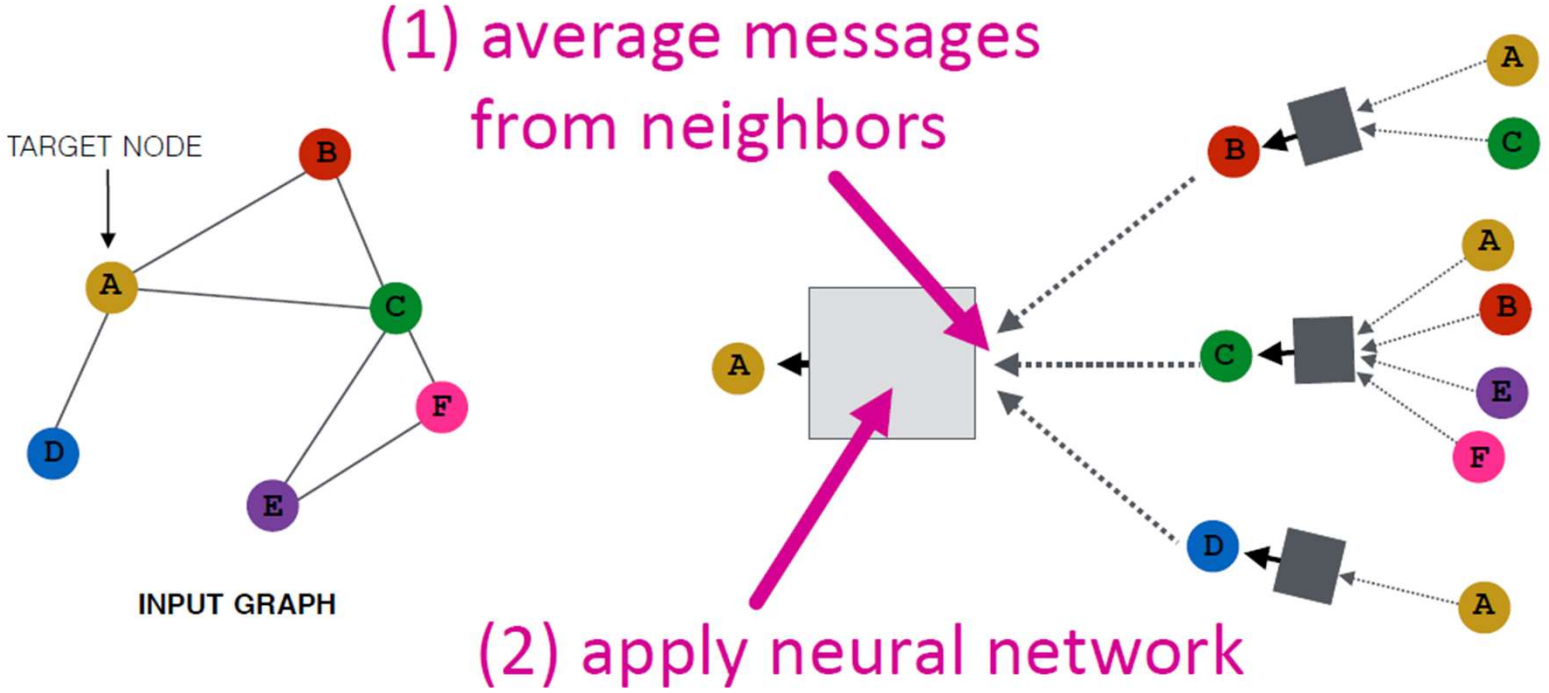
Neighborhood Aggregation

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



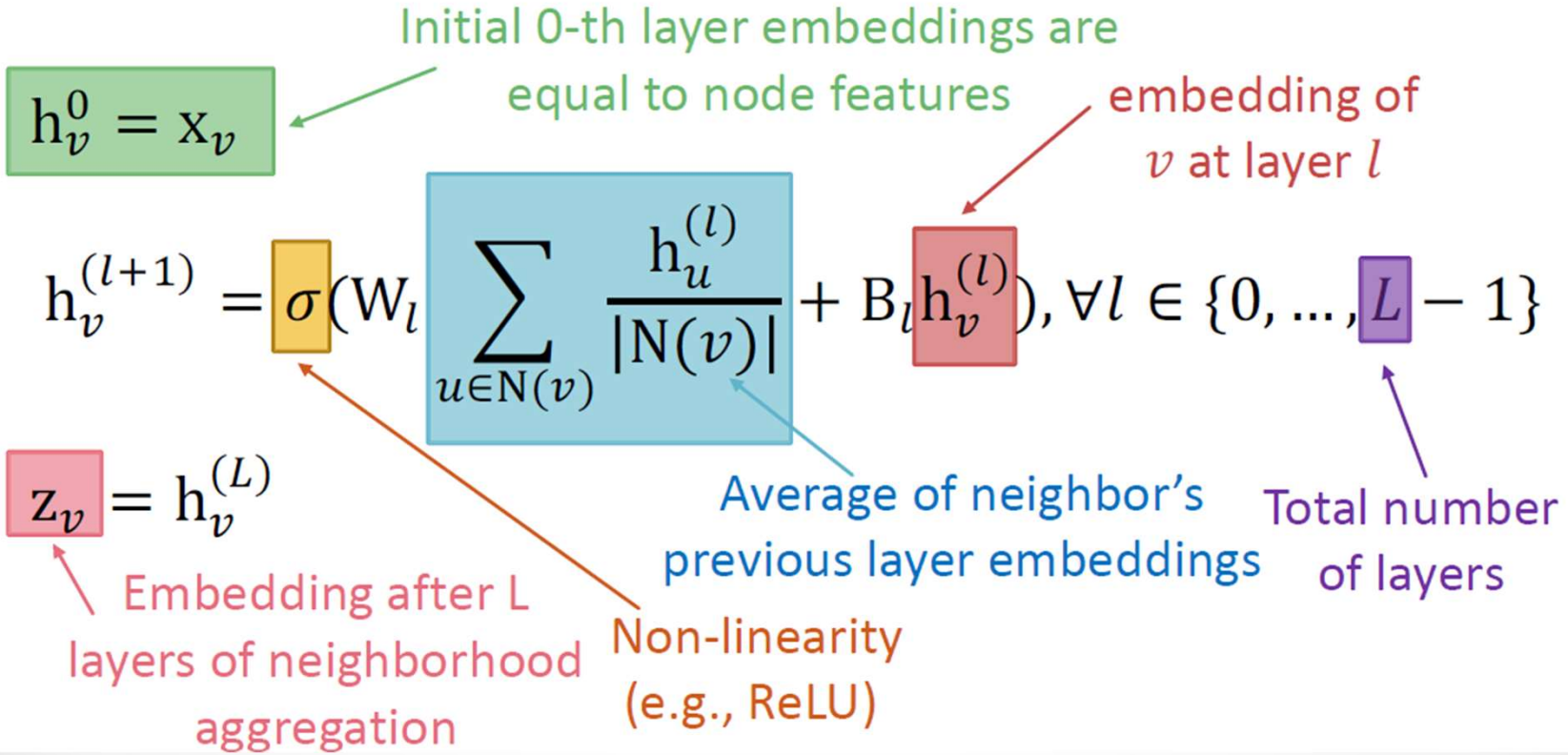
Neighborhood Aggregation

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



The Math: Deep Encoder

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

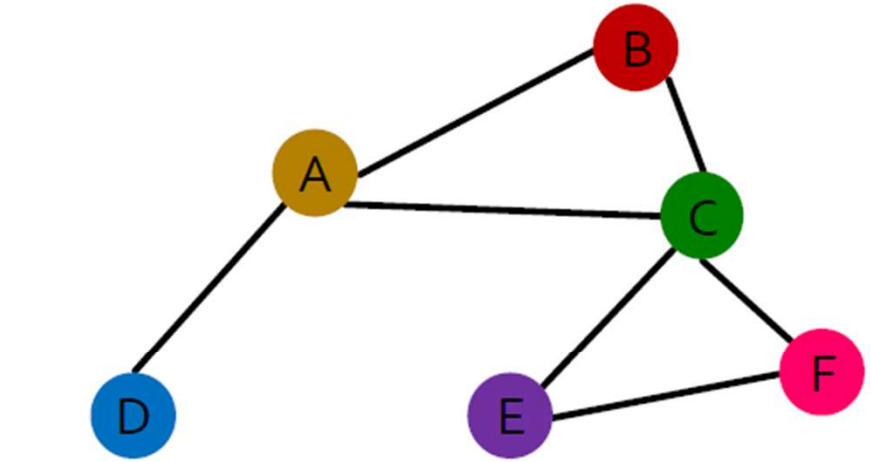


GCN: Invariance & Equivariance

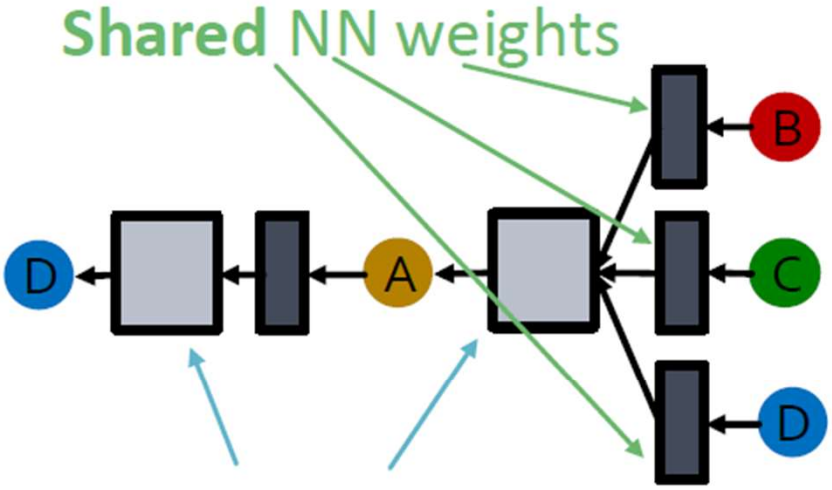
What are the invariance and equivariance properties for a GCN?

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

i) Shared NN wights ii) The aggregation is Permutation Invariant



Target Node

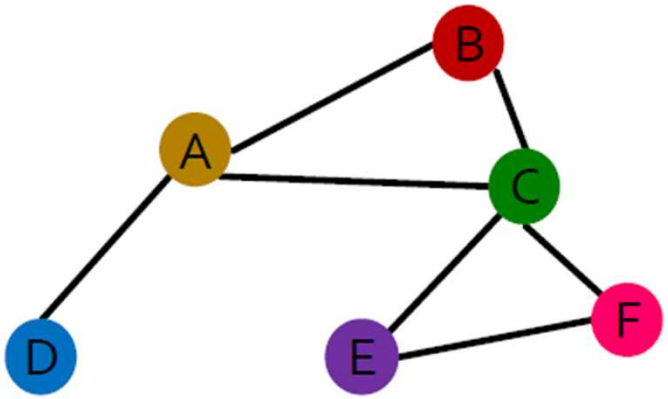


Average of neighbor's previous layer embeddings - **Permutation invariant**

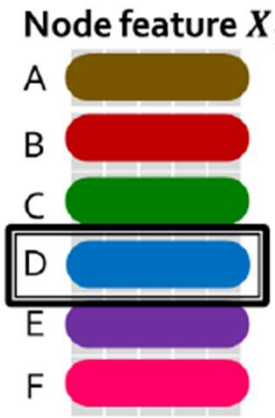
GCN: Invariance & Equivariance

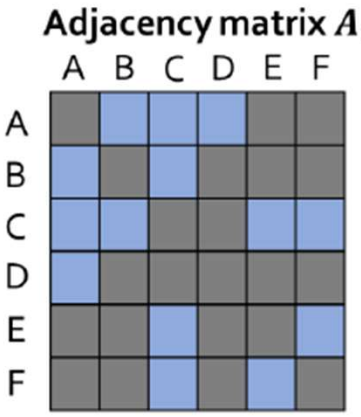
- **Considering all nodes in a graph**, GCN computation is **permutation equivariant**

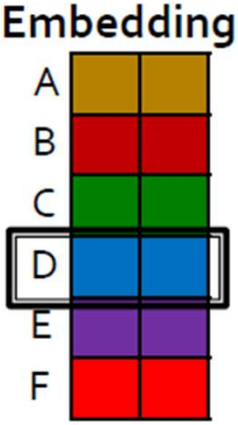
Order plan 1



Target Node

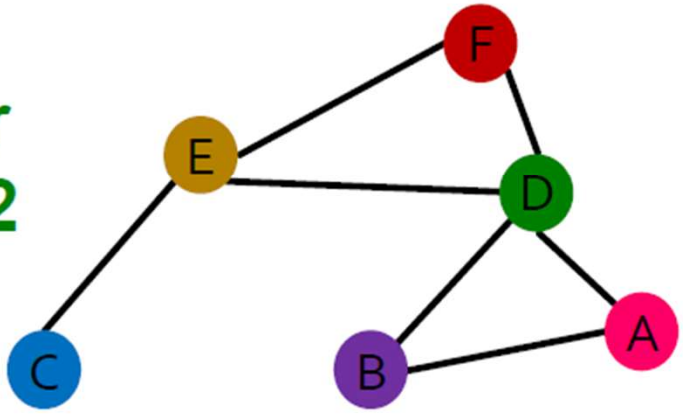




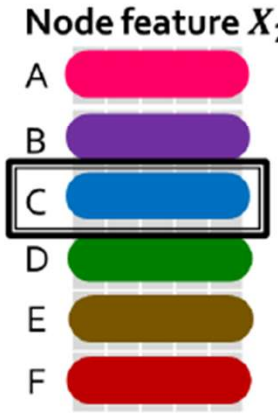


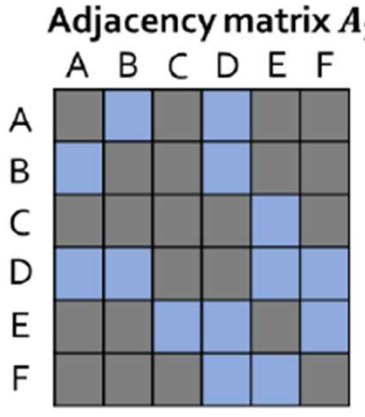
Permute the input, the output also permutes accordingly - permutation equivariant

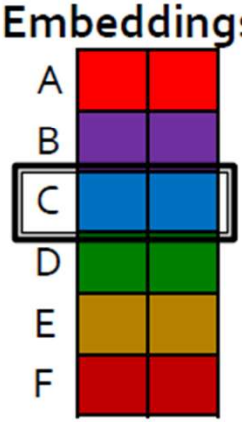
Order plan 2



Target Node



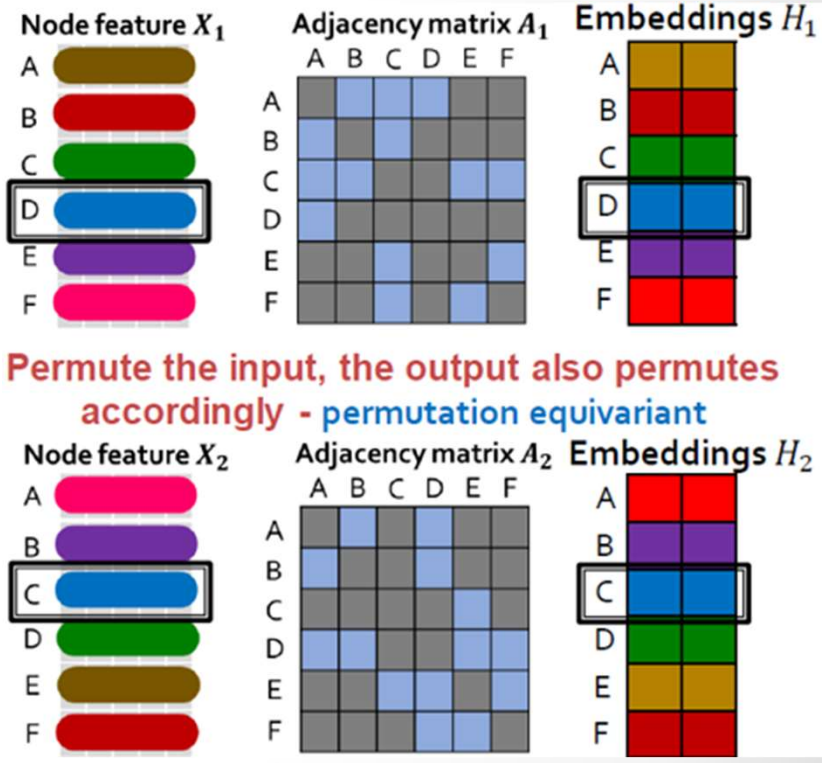




GCN: Invariance & Equivariance

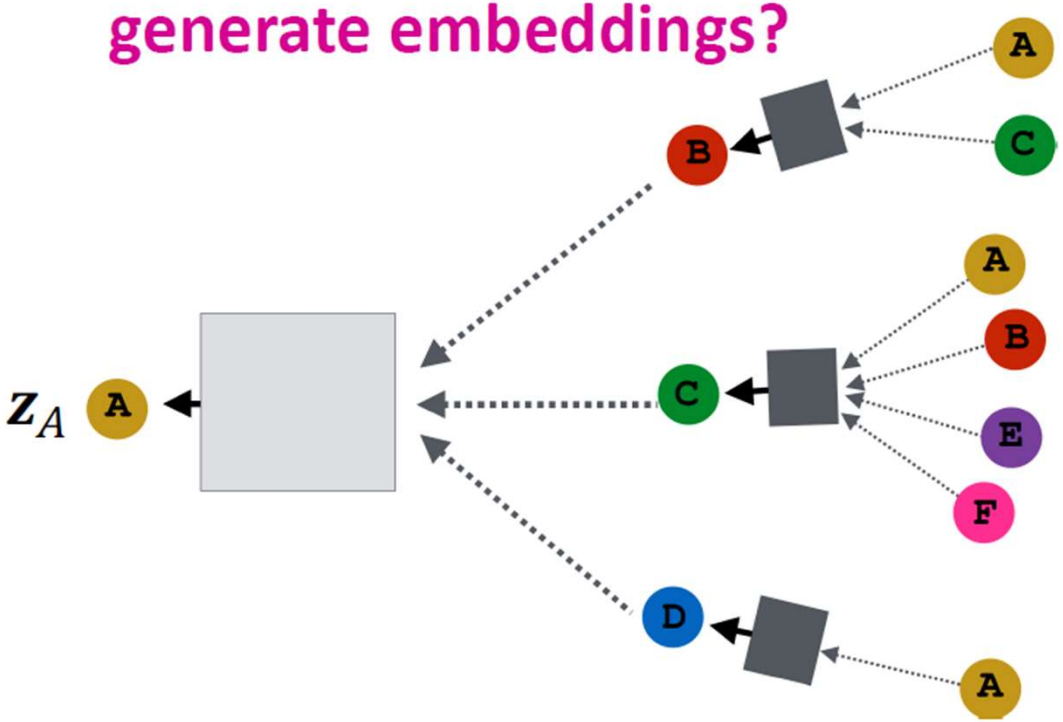
- **Considering all nodes in a graph**, GCN computation is **permutation equivariant**

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 changes in the same way.**



Training the Model

How do we train the model to generate embeddings?



Need to define a loss function on the embeddings

Model Parameters

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

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

$$h_v^{(l+1)} = \sigma\left(W_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)}\right), \forall l \in \{0, \ldots, L-1\}$$

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

Final node embedding

We can feed these **embeddings into any loss function** and run SGD to **train the weight parameters**

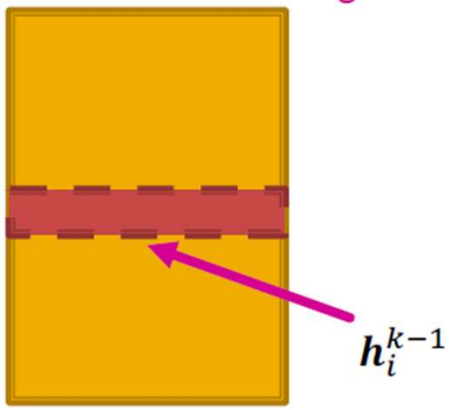
h_v^l: the hidden representation of node v at layer l

- 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 $H^{(l)} = [h_1^{(l)} \dots h_{|V|}^{(l)}]^{\mathrm{T}}$
- Then: $\sum_{u \in N_v} h_u^{(l)} = \mathrm{A}_{v,:}\mathrm{H}^{(l)}$
- 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,**

Matrix of hidden embeddings H^{k-1}



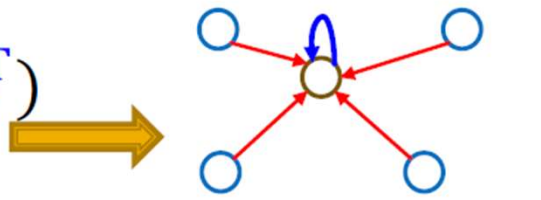
$$\sum_{u \in N(v)} \frac{h_u^{(l-1)}}{|N(v)|} \Rightarrow H^{(l+1)} = D^{-1}AH^{(l)}$$

Matrix Formulation

- Re-writing update function in matrix form:

$$H^{(l+1)} = \sigma(\tilde{A}H^{(l)}W_l^{\mathrm{T}} + H^{(l)}B_l^{\mathrm{T}})$$

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



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

  - 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 z_v is a function of input graph
- **Supervised setting**: we want to minimize the loss

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

  - $\boldsymbol{y}$: 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**

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

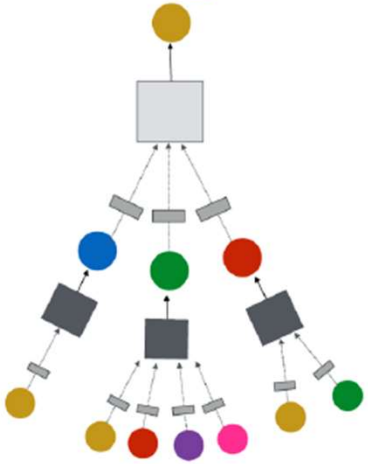
  - Where $y_{u,v} = 1$ when node u and v are **similar**

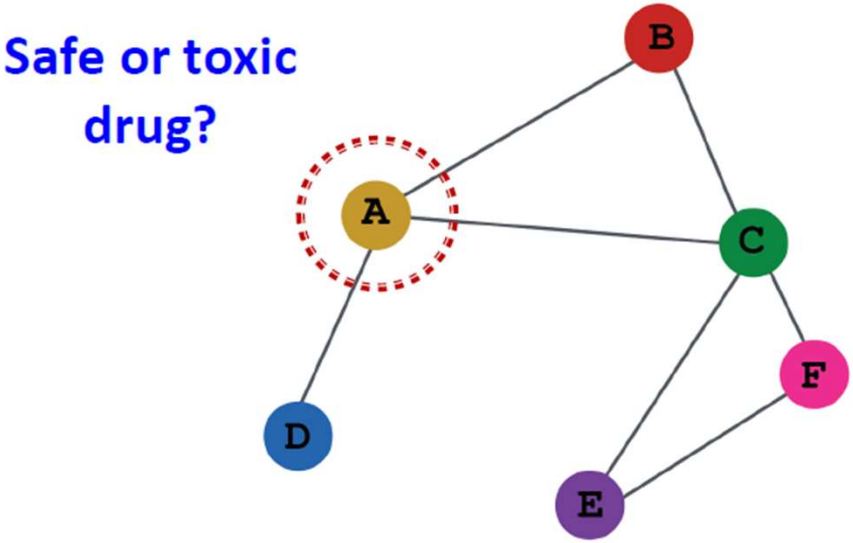
- CE is the cross entropy
- DEC is the decoder such as inner product

- **Node similarity** a loss based on:
  - **Random walks** (node2vec, DeepWalk)
  - **Matrix factorization**
  - **Node proximity in the graph**

Supervised Training

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





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

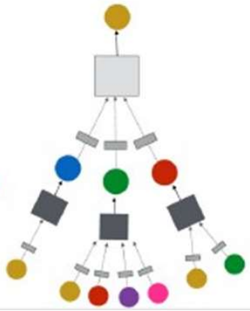
$$\mathcal{L} = -\sum_{v \in V} y_v \log(\sigma(z_v^T \theta)) + (1 - y_v) \log(1 - \sigma(z_v^T \theta))$$

Encoder output: node embedding

Classification weights

Node class label

Safe or toxic drug?



We need to minimize the loss.

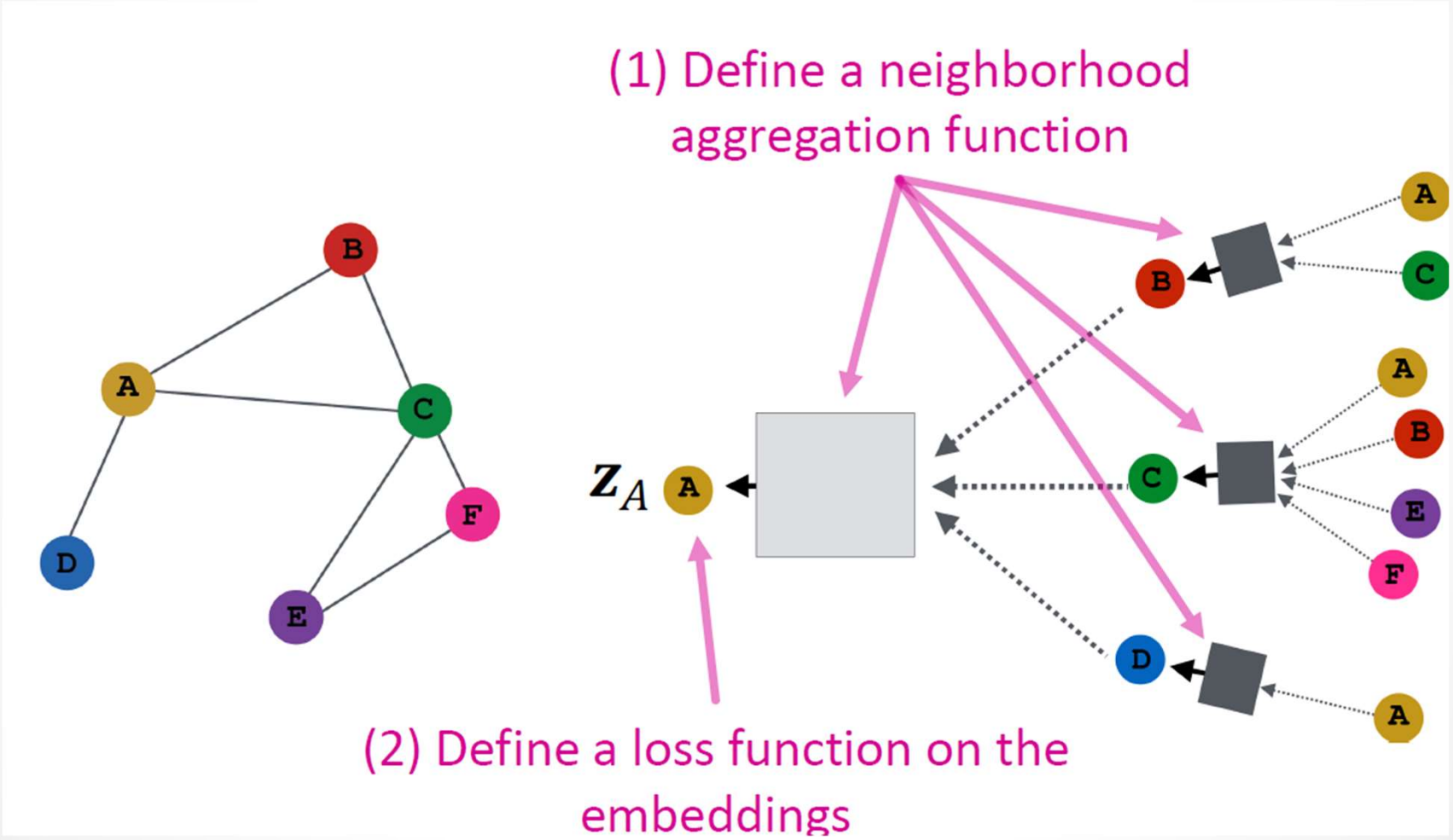
If class label = 1, 1st term needs to be high => encoder output needs to be high

2nd term is 0

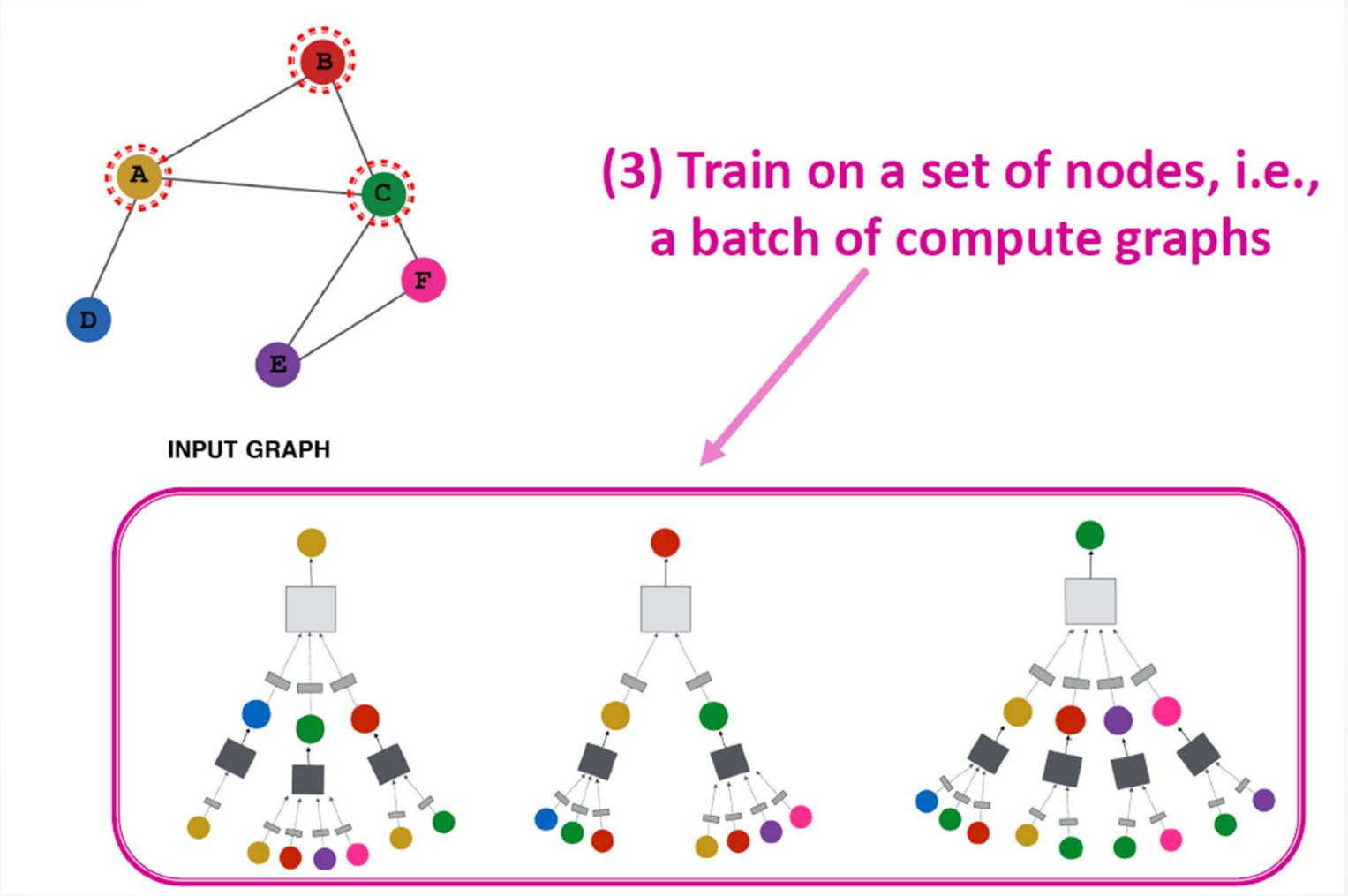
If class label = 0, 2nd term needs to be high => encoder output needs to be low

1st term is 0

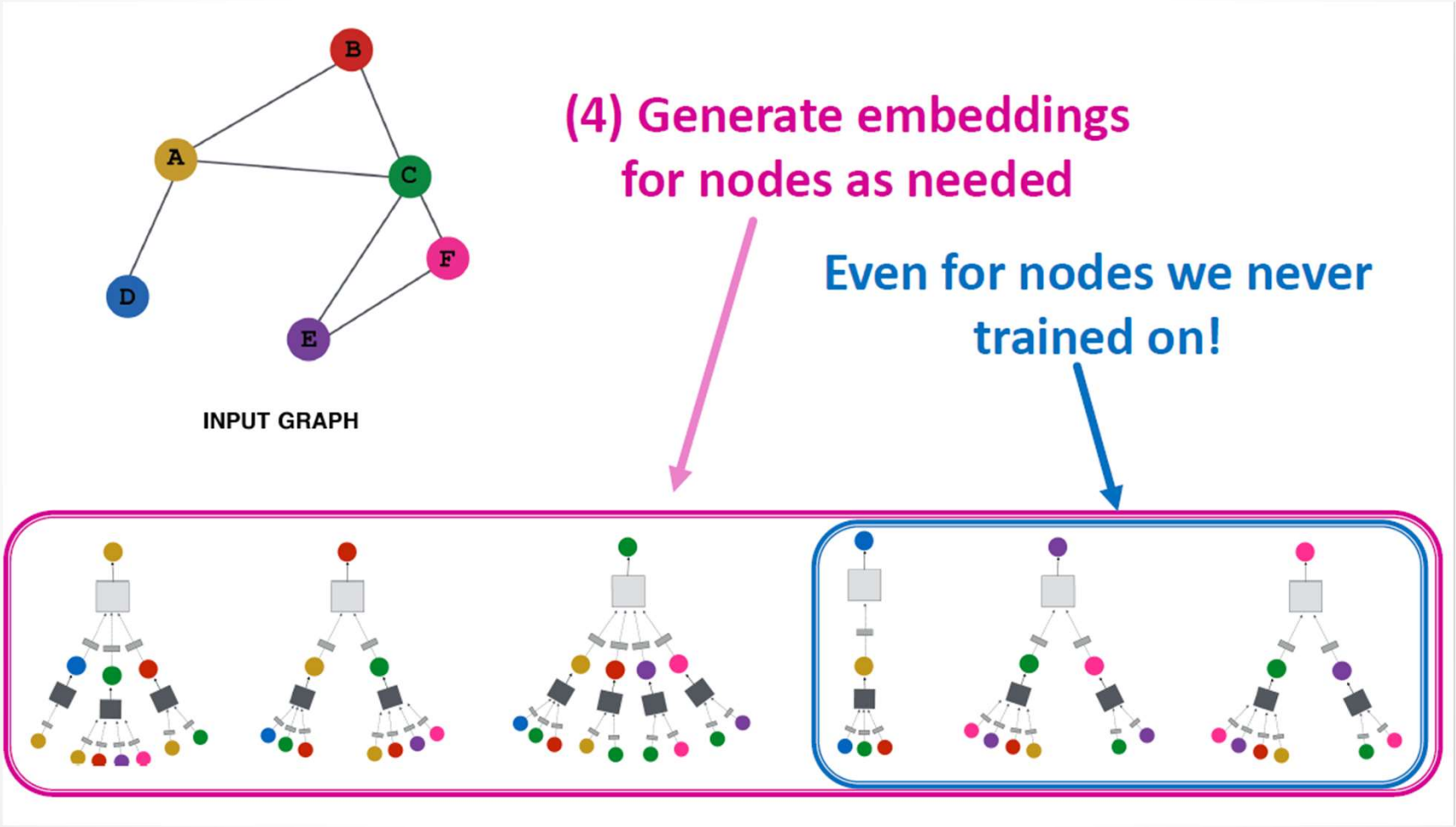
Model Design: Overview



Model Design: Overview

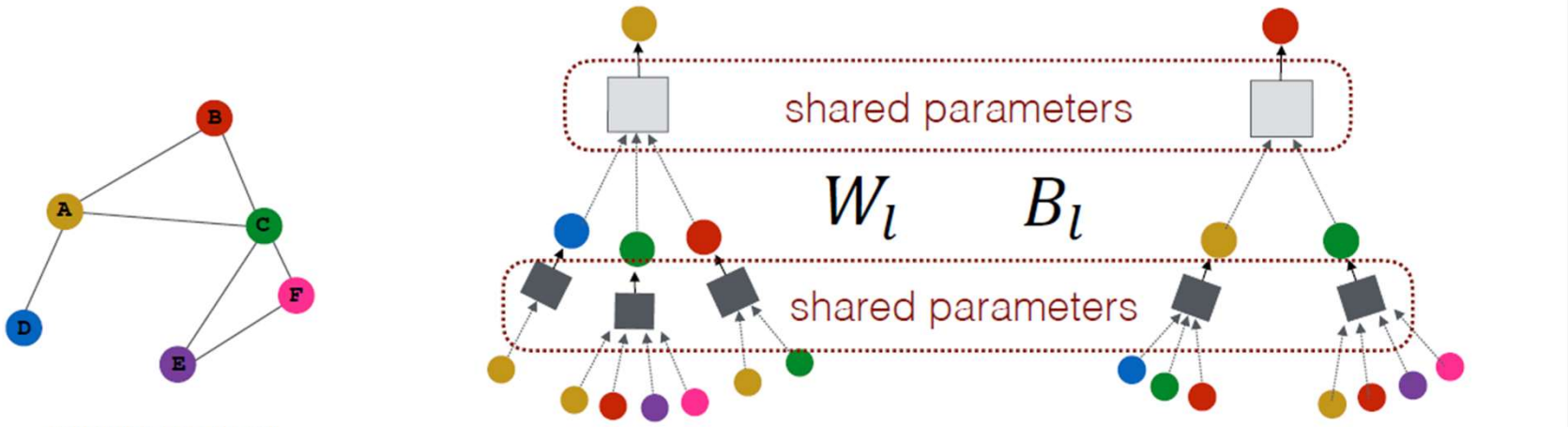


Model Design: Overview

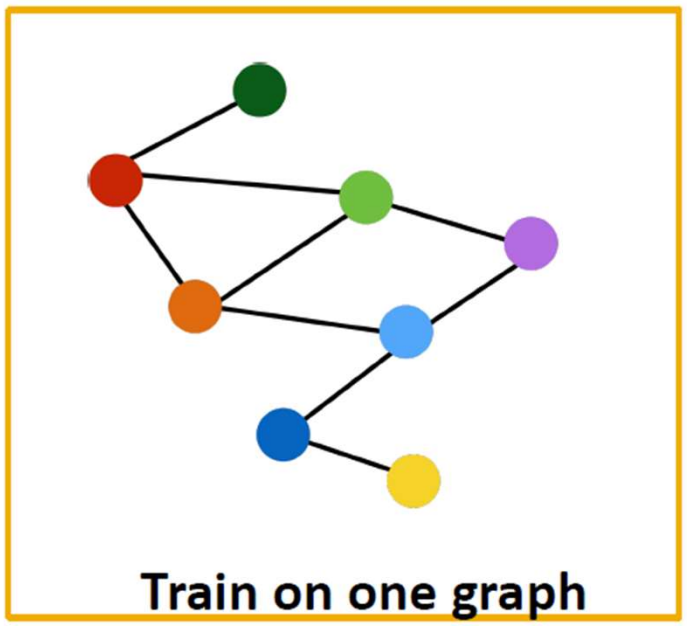


Inductive Capability

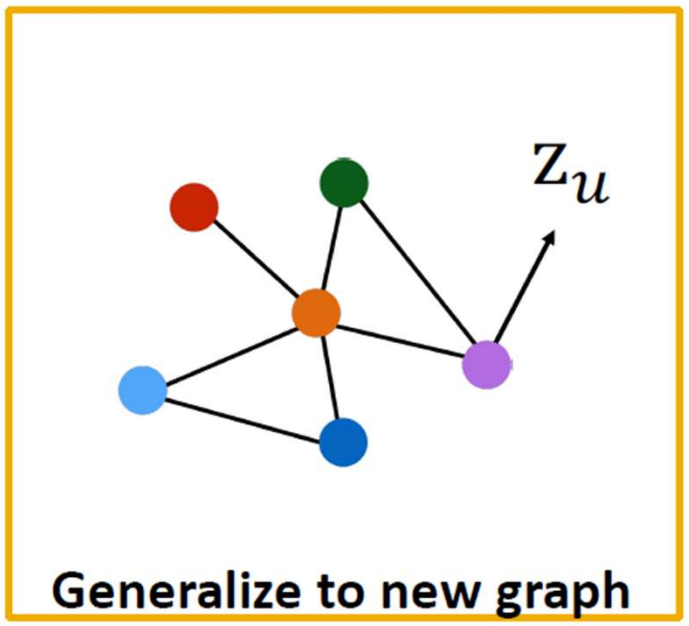
- **The same aggregation parameters are shared for all nodes:**
  - The number of model parameters is sublinear in $|V|$ and we can **generalize to unseen nodes**!



Inductive Capability: New Graphs



Train on one graph

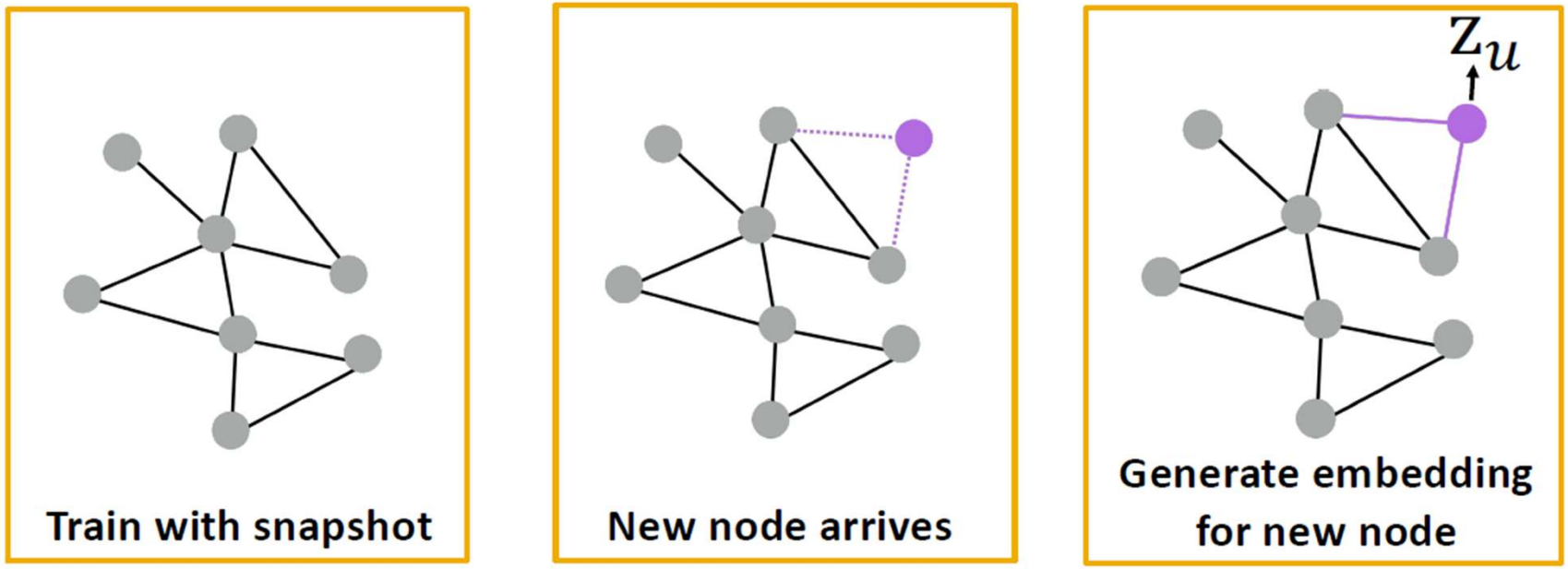


Generalize to new graph

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"

References

- Social Network Analysis Tanmoy Chakraborty. Chapter 9