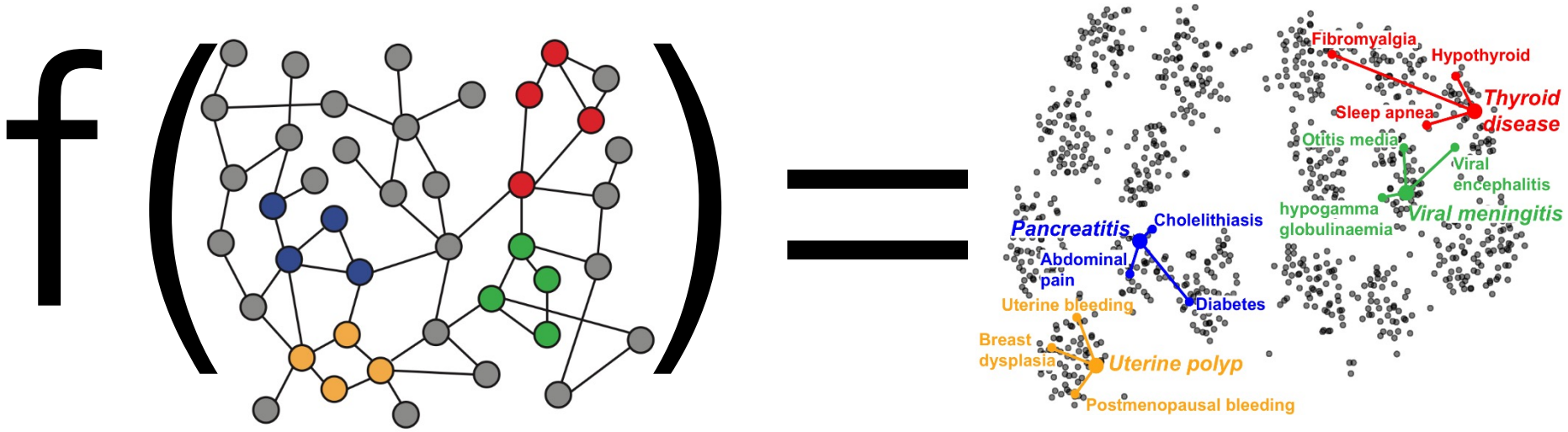
Graph Neural Networks Part I

Recap: Node Embeddings

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



Input graph

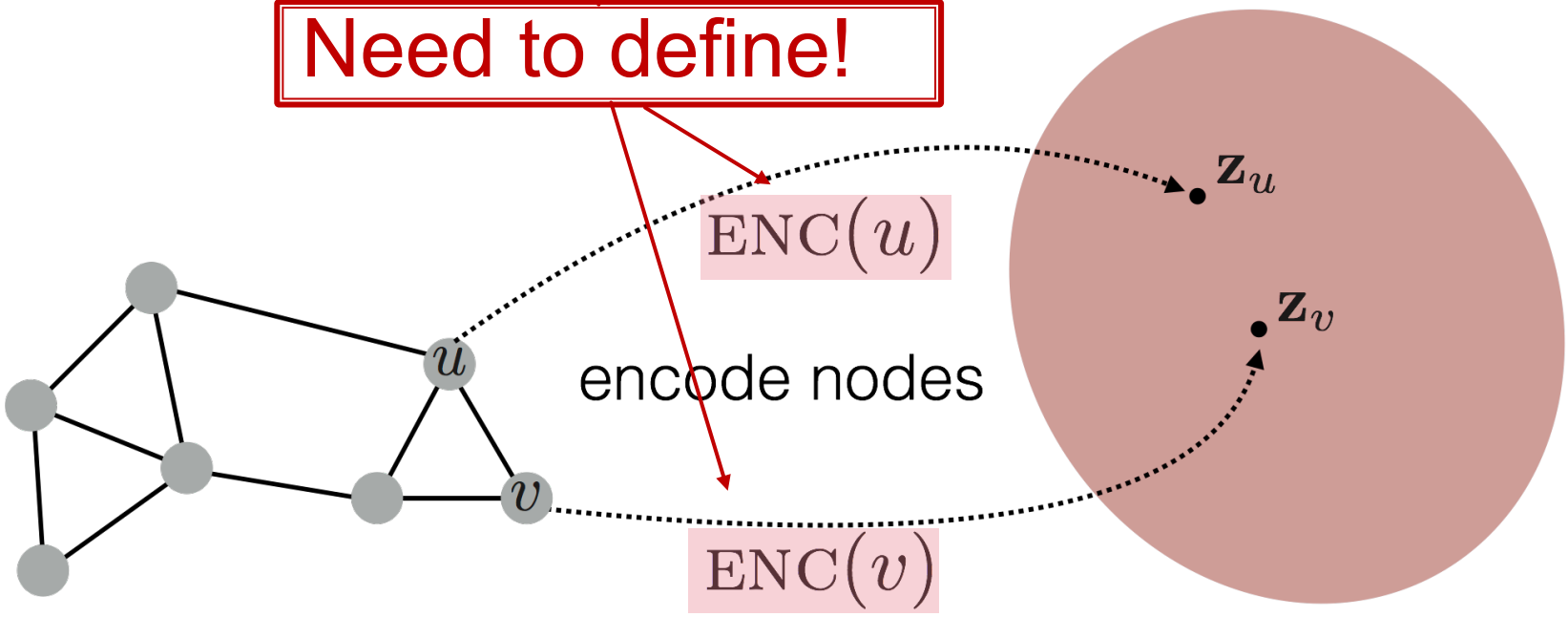
2D node embeddings

How to learn mapping function f?

Recap: Node Embeddings

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

Need to define!



Input network

d-dimensional embedding space

Recap: 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

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

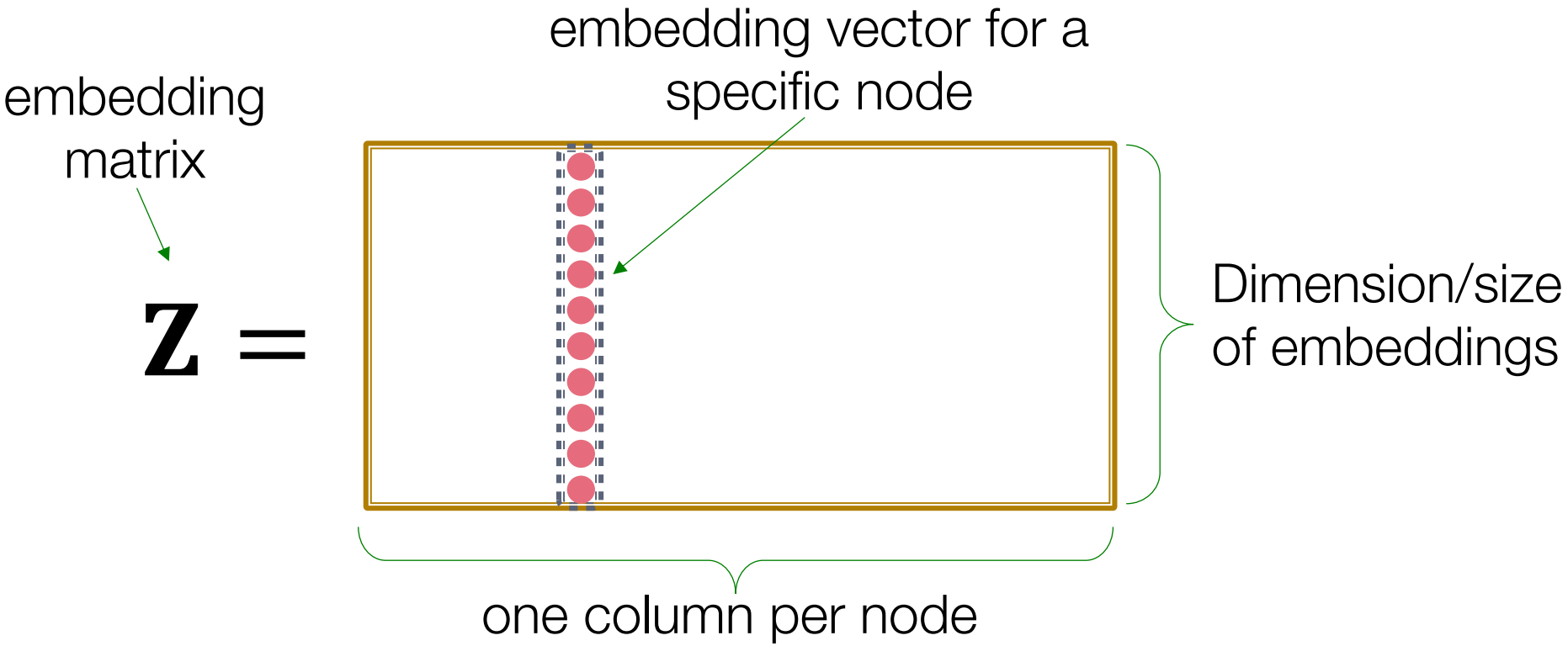
Decoder

Similarity of u and v in the original network

dot product between node embeddings

Recap: "Shallow" Encoding

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



Recap: Shallow Encoders

- **Limitations** of shallow embedding methods:
  - $O(|V|)$ **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

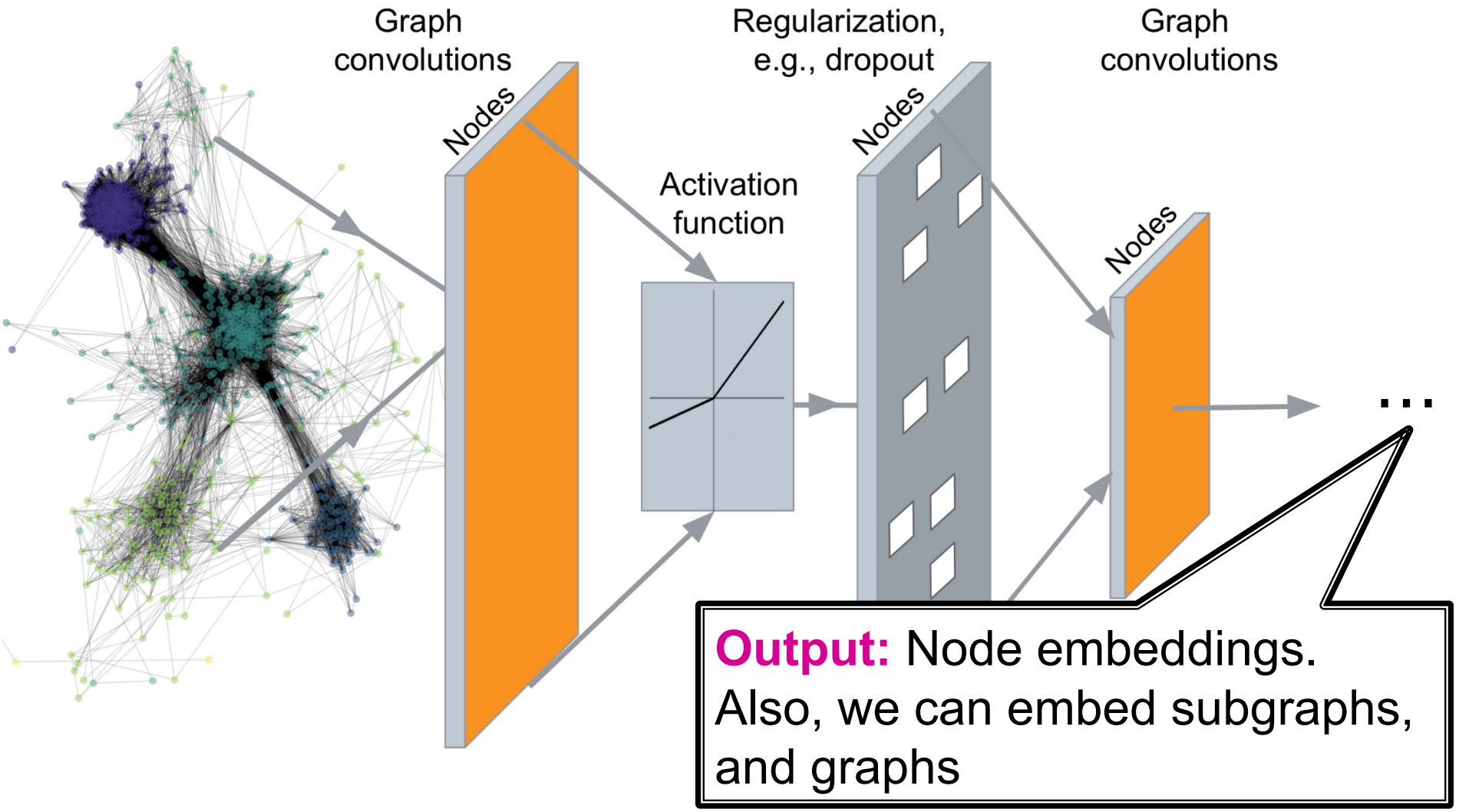
Today: Deep Graph Encoders

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

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

- **Note:** All these deep encoders can be **combined with node similarity functions** defined in the Lecture 3.

Deep Graph Encoders



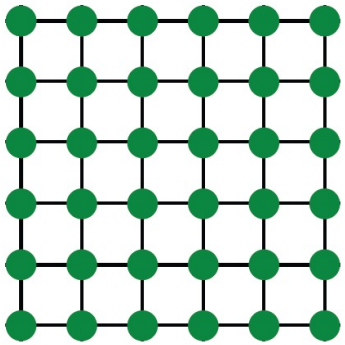
Output: Node embeddings. Also, we can embed subgraphs, and graphs

Tasks on Networks

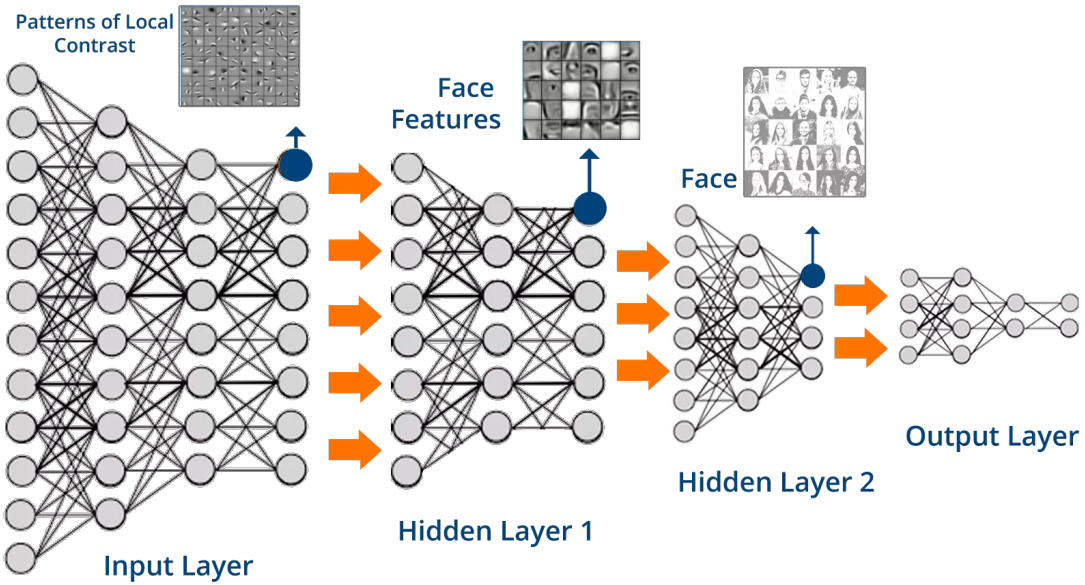
Tasks we will be able to solve:

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

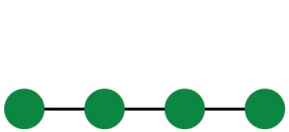
Modern ML Toolbox

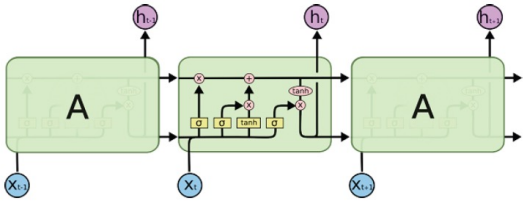


Images



Text/Speech



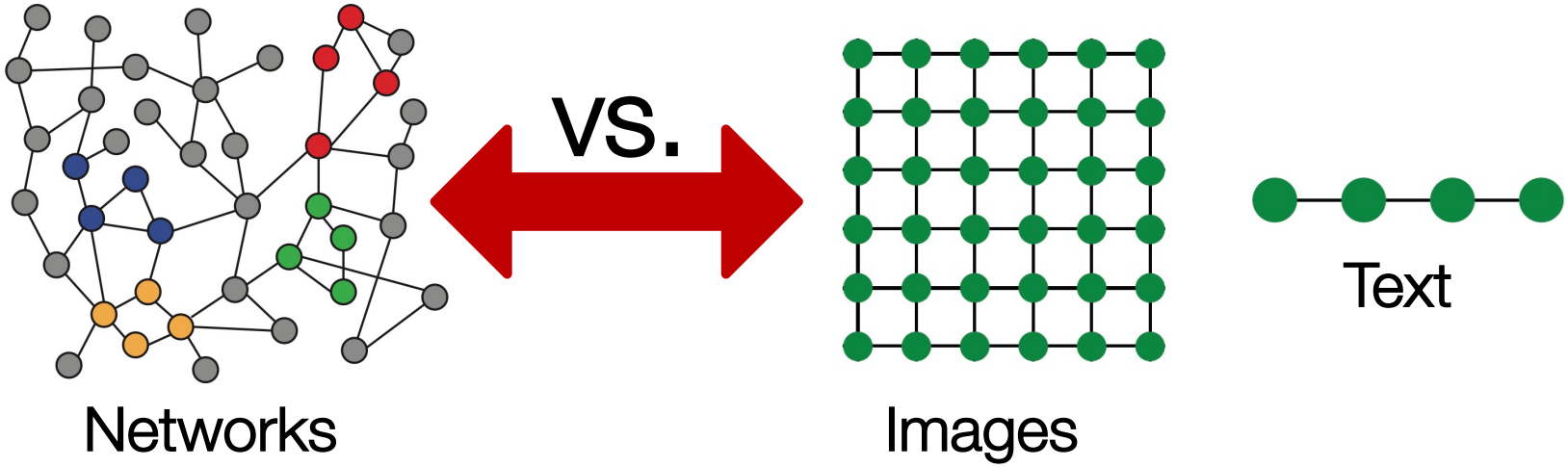


Modern deep learning toolbox is designed for simple sequences & grids

Why is it Hard?

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
- Often dynamic and have multimodal features

Outline of Today's Lecture

1. **Deep learning for graphs**

2. **Graph Convolutional Networks**

3. **GNNs subsume CNNs and Transformers**

4. **Appendix: Basics of deep learning**

Stanford CS224W: Deep Learning for Graphs

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

http://cs224w.stanford.edu

Content

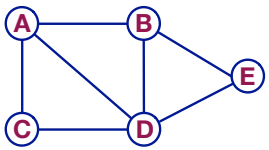
- **Local network neighborhoods:**
  - Describe aggregation strategies
  - Define computation graphs
- **Stacking multiple layers:**
  - Describe the model, parameters, training
  - How to fit the model?
  - Simple example for unsupervised and supervised training

Setup

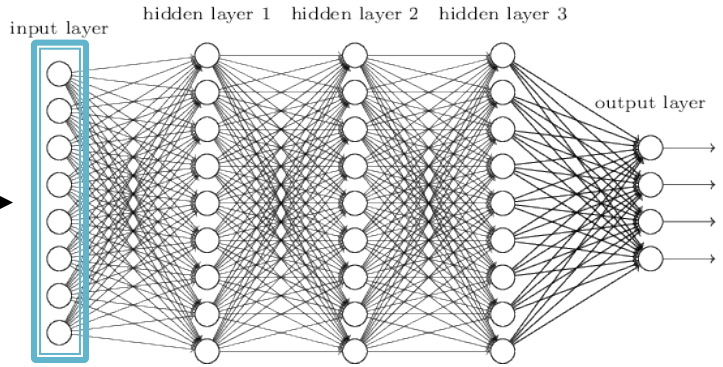
- **Assume we have a graph G:**
  - V is the **vertex set**
  - $\boldsymbol{A}$ is the **adjacency matrix** (assume binary)
  - $\boldsymbol{X} \in \mathbb{R}^{m \times |V|}$ is a matrix of **node features**
  - v: a node in V; $N(v)$: the set of neighbors of v.
  - **Node features:**
    - Social networks: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

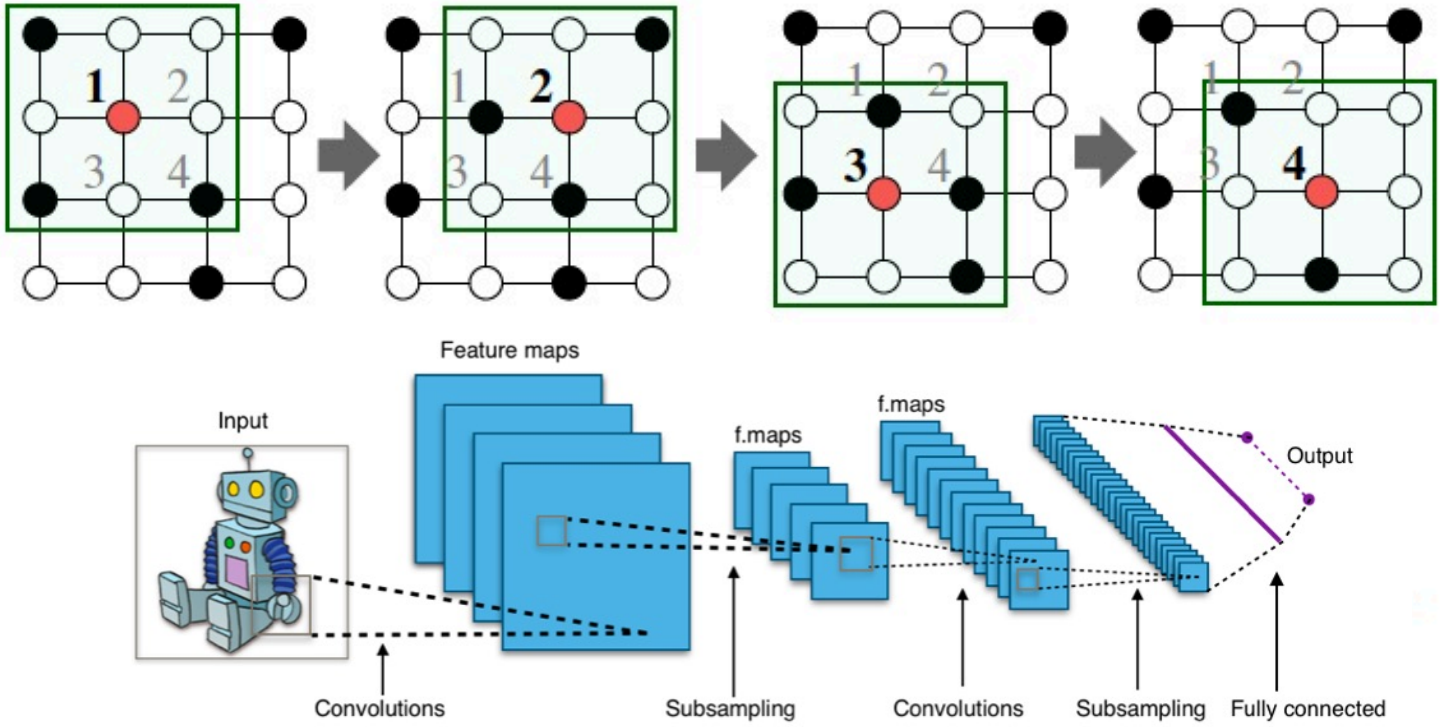


?

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

Idea: Convolutional Networks

CNN on an image:

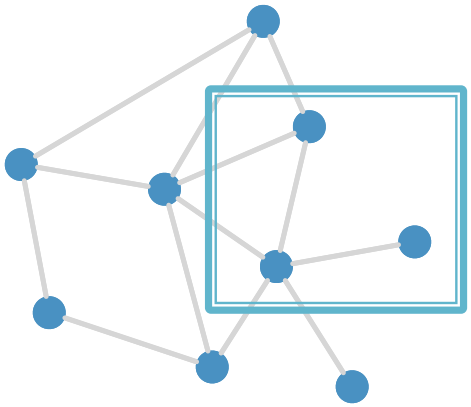


Goal is to generalize convolutions beyond simple lattices

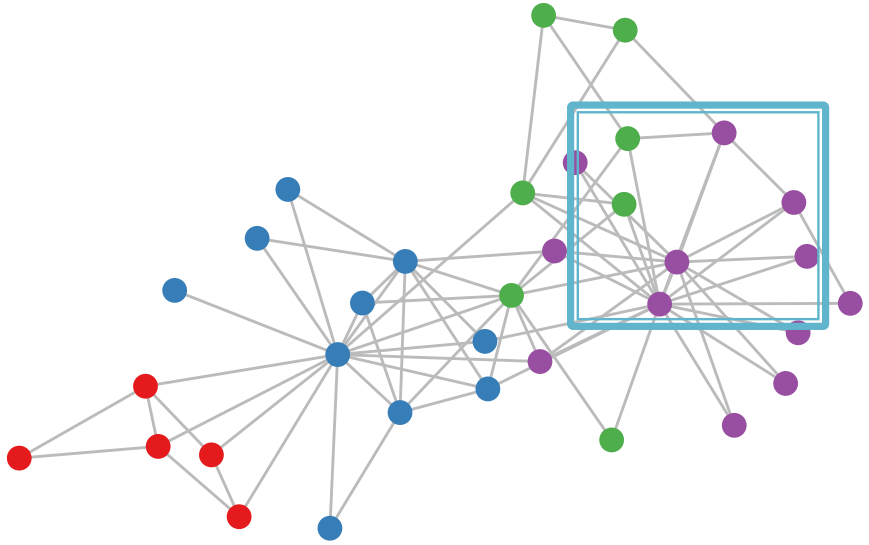
Leverage node features/attributes (e.g., text, images)

Real-World Graphs

But our graphs look like this:



or this:



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

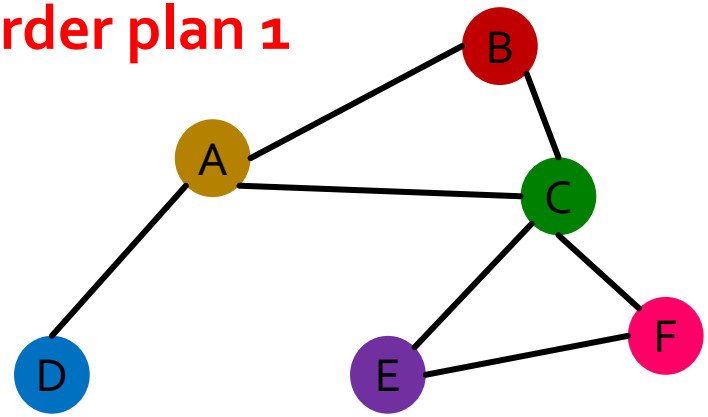
Permutation Invariance

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

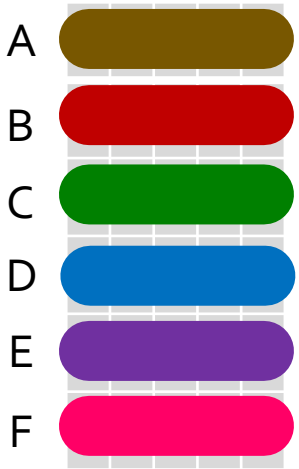
Permutation Invariance

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

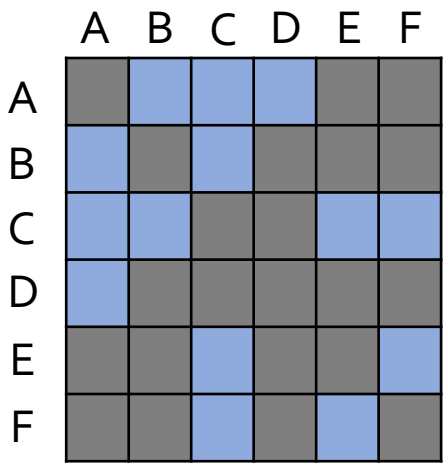
Order plan 1



Node features X_1



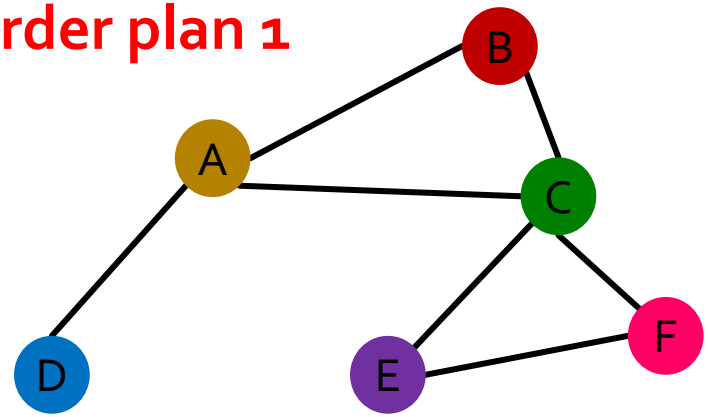
Adjacency matrix A_1



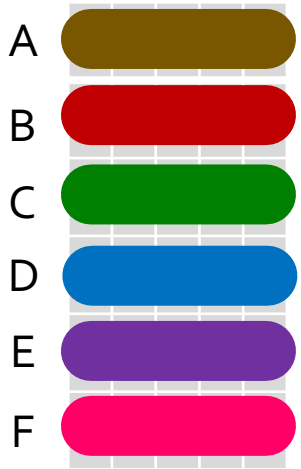
Permutation Invariance

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

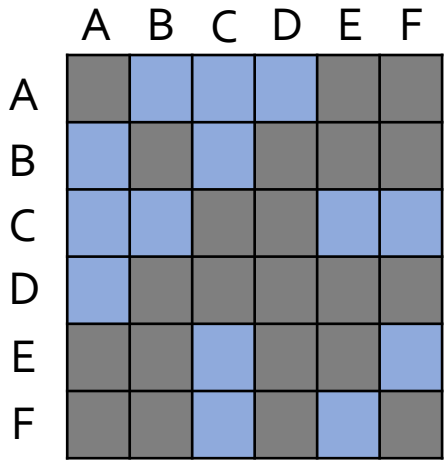
Order plan 1



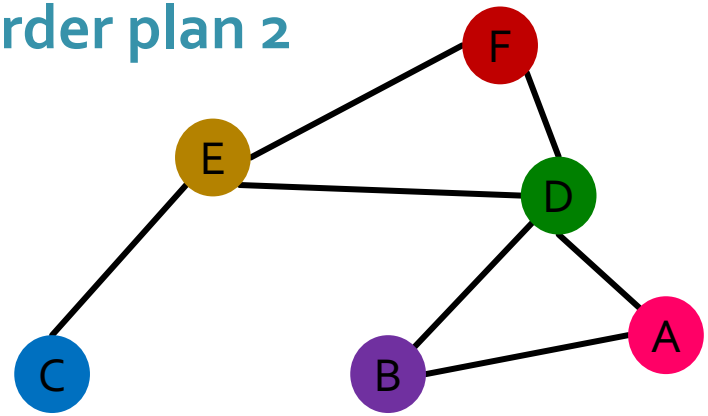
Node features X_1



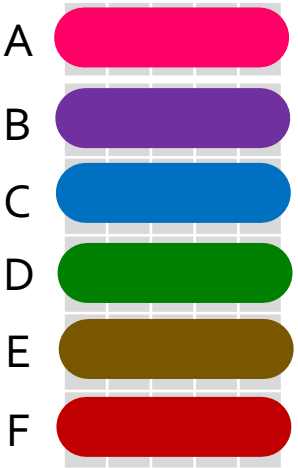
Adjacency matrix A_1



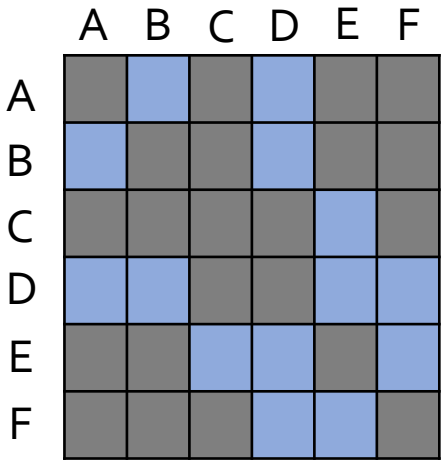
Order plan 2



Node features X_2



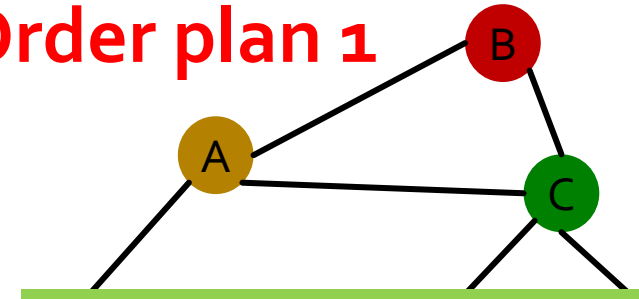
Adjacency matrix A_2



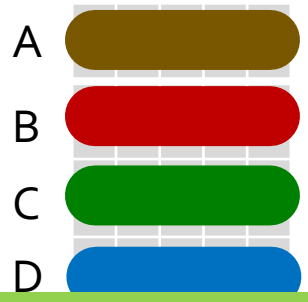
Permutation Invariance

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

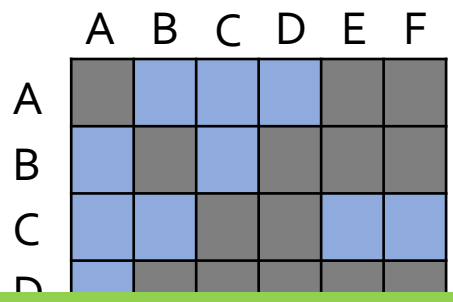
Order plan 1



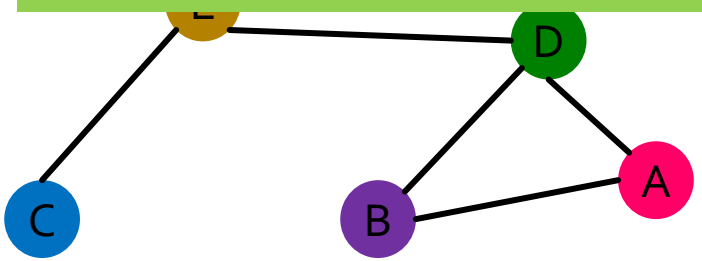
Node feature X_1

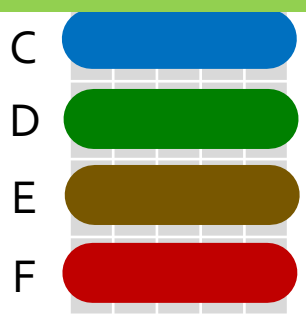


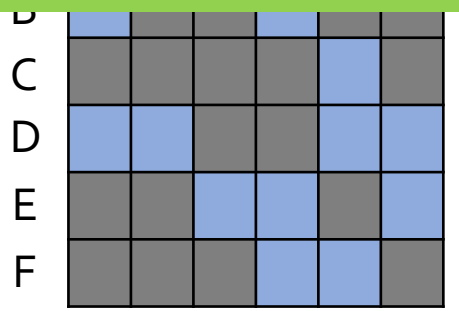
Adjacency matrix A_1



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







Permutation Invariance

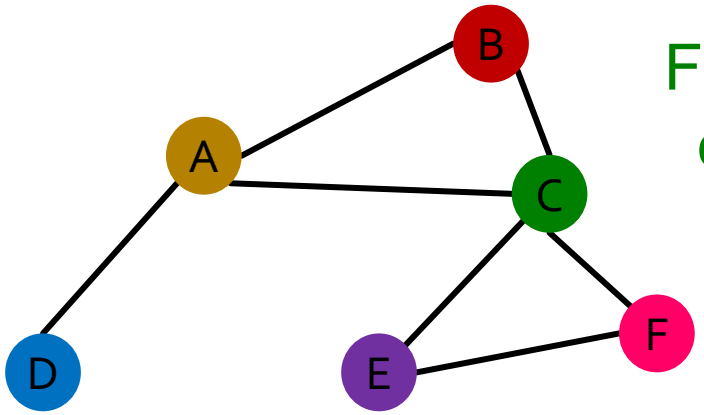
What does it mean by "graph representation is same for two order plans"?

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

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

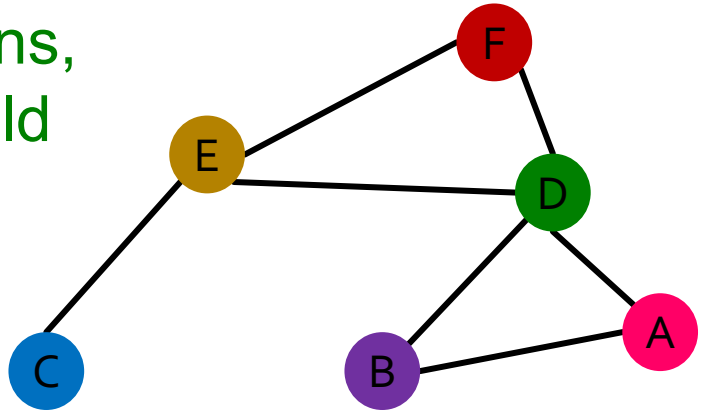
$\boldsymbol{A}$ is the adjacency matrix
$\boldsymbol{X}$ is the node feature matrix

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 $\mathbb{R}^d$.
- Then, if $f(\boldsymbol{A}_i, \boldsymbol{X}_i) = f\left(\boldsymbol{A}_j, \boldsymbol{X}_j\right)$ for any order plan i and j, we formally say f is a **permutation invariant function**.

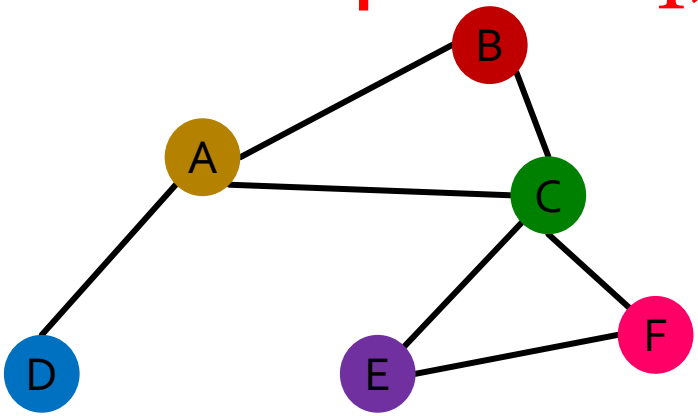
$\boldsymbol{A}$ is the adjacency matrix
$\boldsymbol{X}$ is the node feature matrix

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

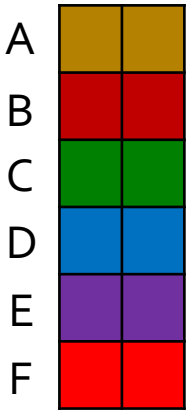
Permutation Equivariance

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

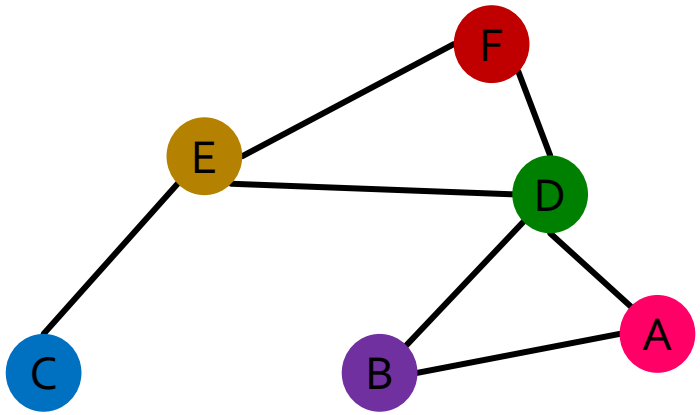
Order plan 1: A_1, X_1



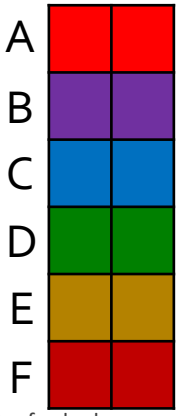
$$f(A_1, X_1) =$$



Order plan 2: A_2, X_2



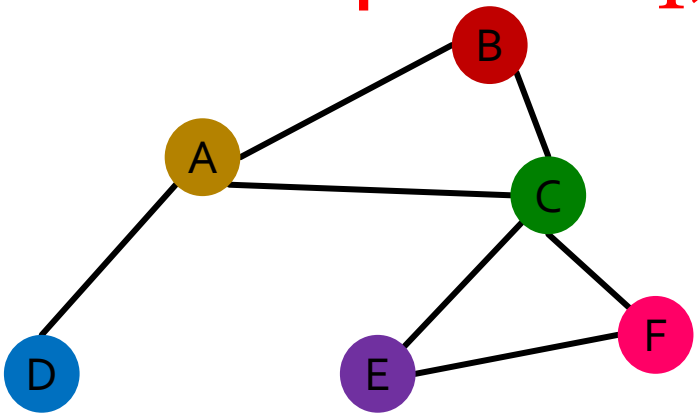
$$f(A_2, X_2) =$$



Permutation Equivariance

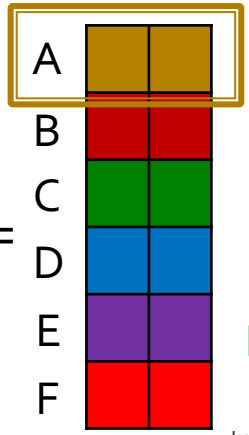
Similarly for node representation: We learn a function f that maps nodes of G to a matrix $\mathbb{R}^{m \times d}$.

Order plan 1: A_1, X_1

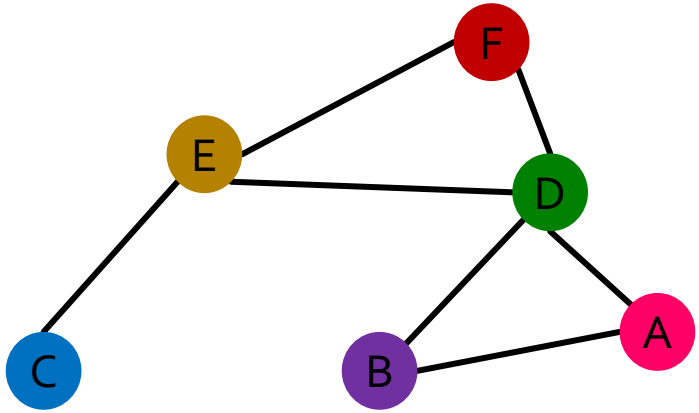


Representation vector of the brown node A

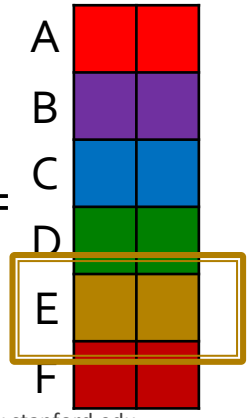
$$f(A_1, X_1) =$$



Order plan 2: A_2, X_2



$$f(A_2, X_2) =$$



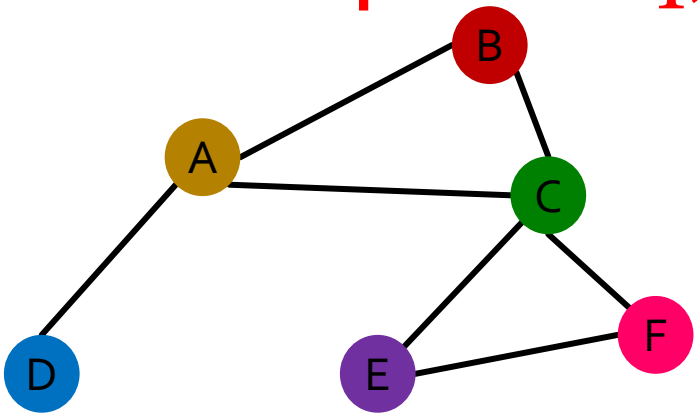
Representation vector of the brown node E

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

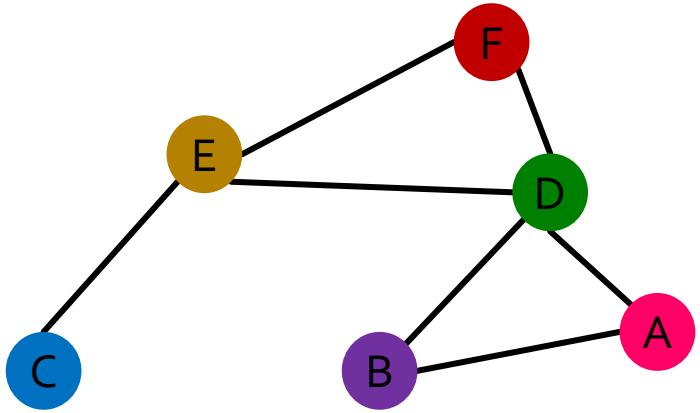
Permutation Equivariance

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

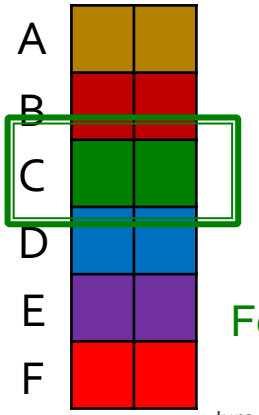
Order plan 1: A_1, X_1



Order plan 2: A_2, X_2

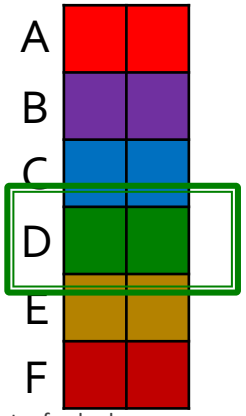


$f(A_1, X_1) =$



Representation vector of the brown node C

$f(A_2, X_2) =$



Representation vector of the brown node D

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

Permutation Equivariance

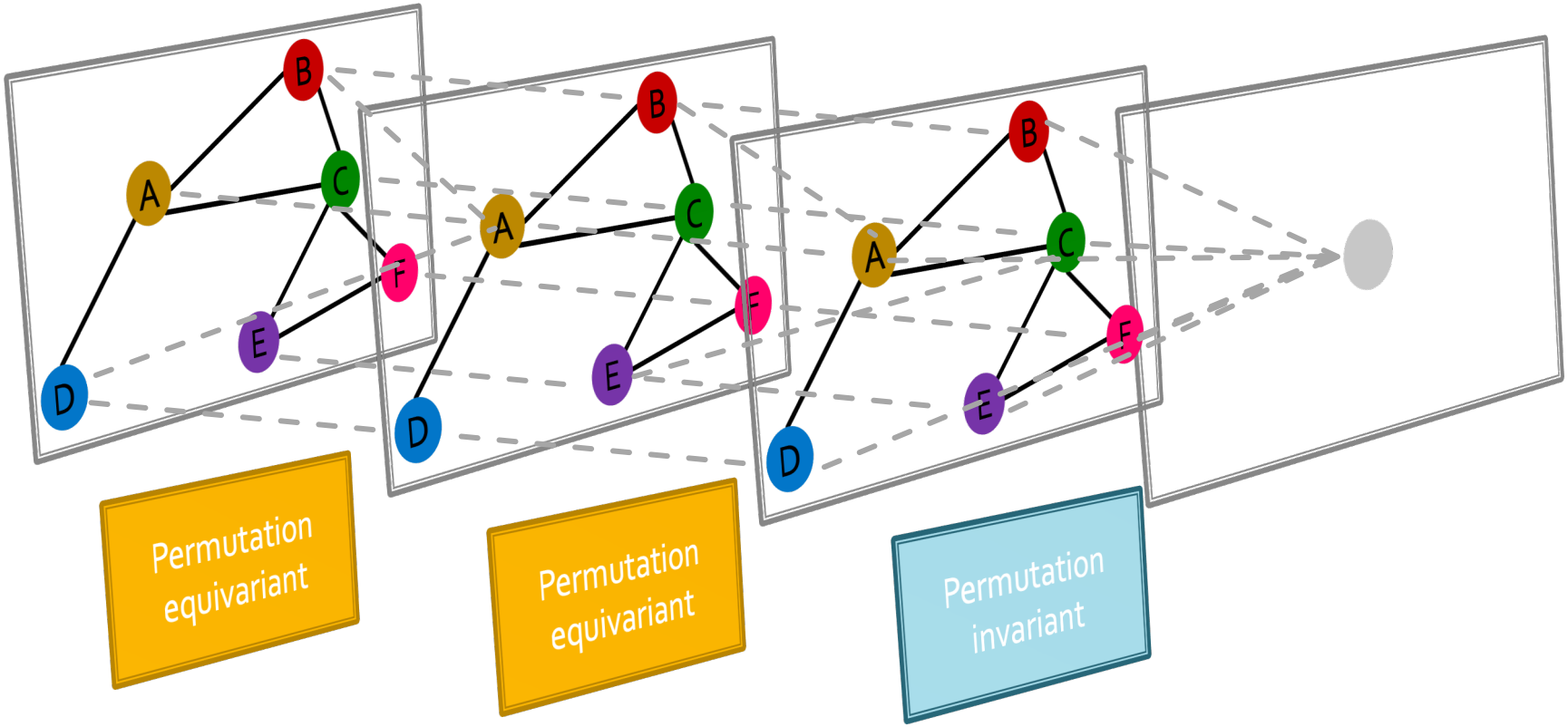
For node representation

- Consider we learn a function f that maps a graph $G = (\boldsymbol{A}, \boldsymbol{X})$ to a matrix $\mathbb{R}^{m \times d}$
  - graph has m nodes, each row is the embedding of a node.
- Similarly, if this property holds for any pair of order plan i and j, we say f is a **permutation equivariant function**.

[Bronstein, ICLR 2021 keynote]

Graph Neural Network Overview

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

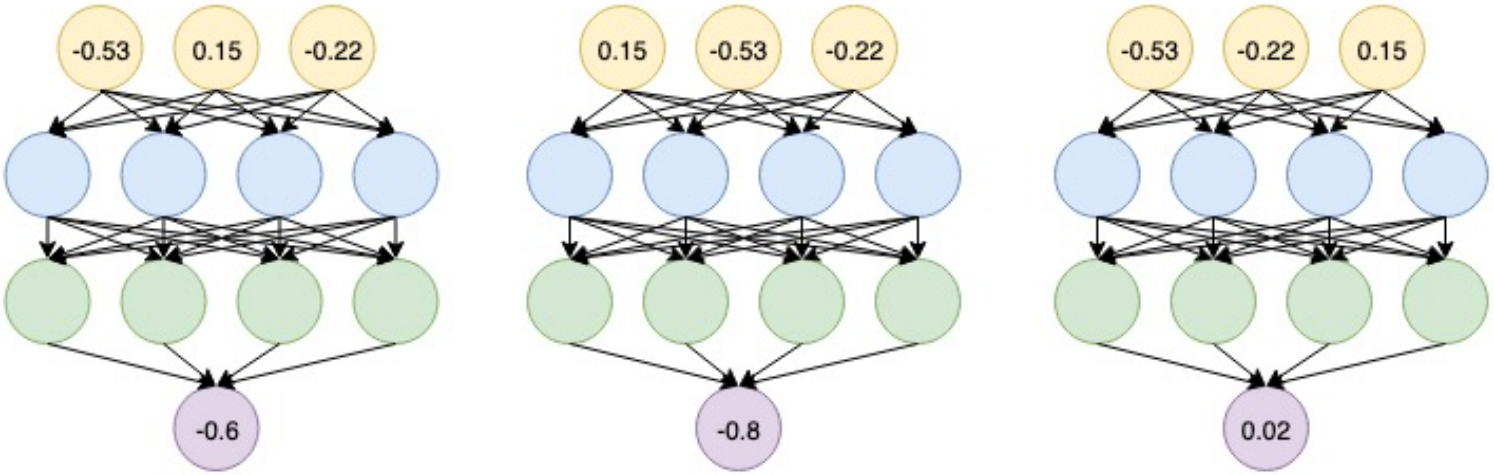


Graph Neural Network Overview

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

- **No.**

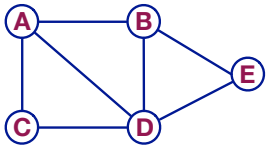
Switching the order of the input leads to different outputs!



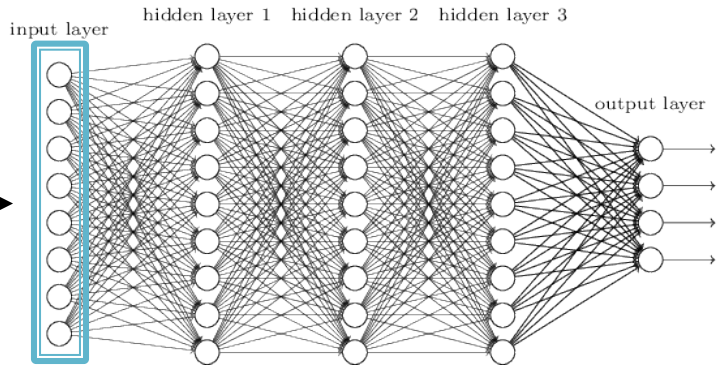
Graph Neural Network Overview

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

- **No.**



	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



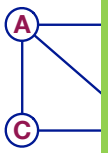
?

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

Graph Neural Network Overview

- Are any neural network architecture, e.g.,

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



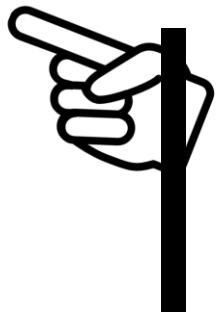
?

Discussion

Questions?

Outline of Today's Lecture

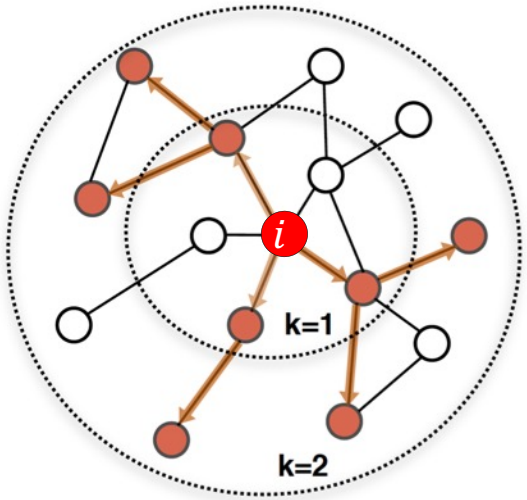
1. Deep learning for graphs ✓
2. **Graph Convolutional Networks**
3. **GNNs subsume CNNs and Transformers**
4. Appendix: Basics of deep learning



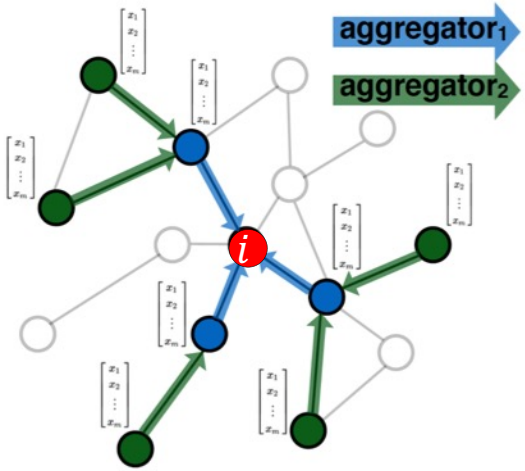
[Kipf and Welling, ICLR 2017]

Graph Convolutional Networks

Idea: Node's neighborhood defines a computation graph



Determine node computation graph



Propagate and transform information

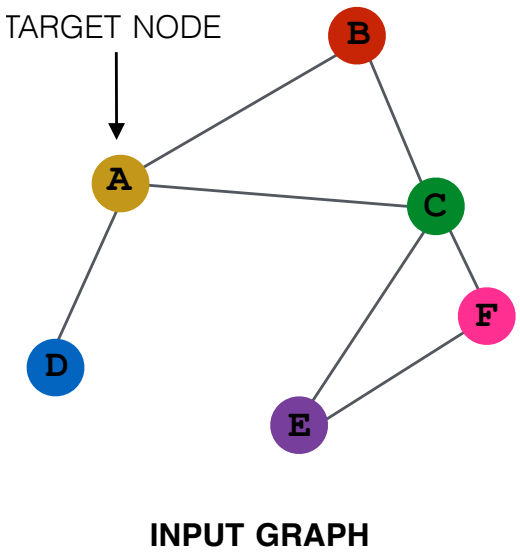
Learn how to propagate information across the graph to compute node features

Graph Convolutional Networks

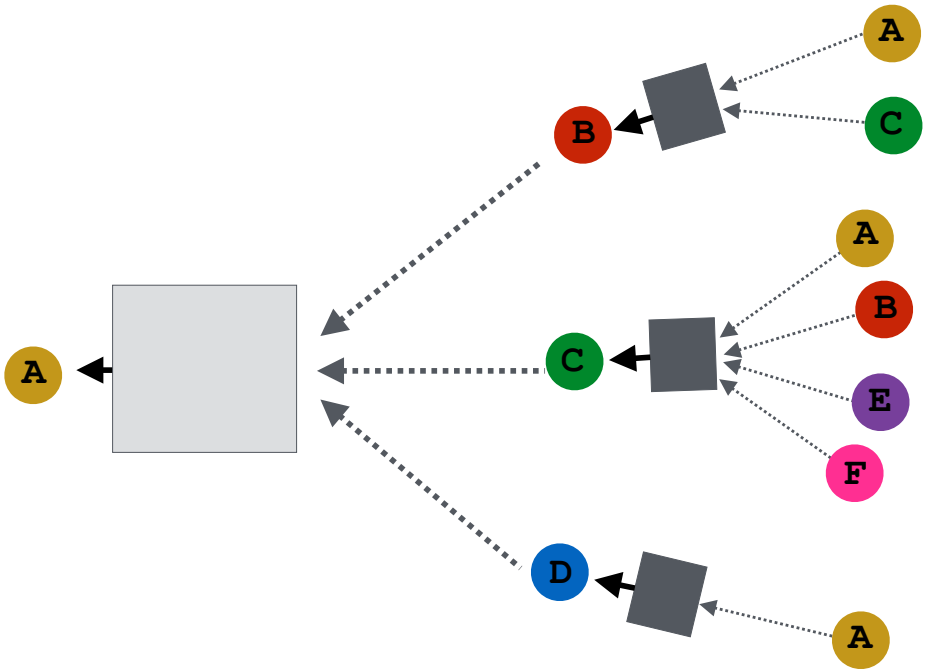
- **Analogy to CNNs**
  - The neighborhood of each node is like a custom filter for each node (instead of all being the same, such as 3x3)
  - Where a CNN slides the filter across each column and row, a GNN applies the "filter" (neighborhood computation) for every node
  - The order in which the filters are calculated does not matter in CNNs or GNNs. The "sliding" is conceptual, but all filters can be calculated in parallel, or in any order

Idea: Aggregate Neighbors

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

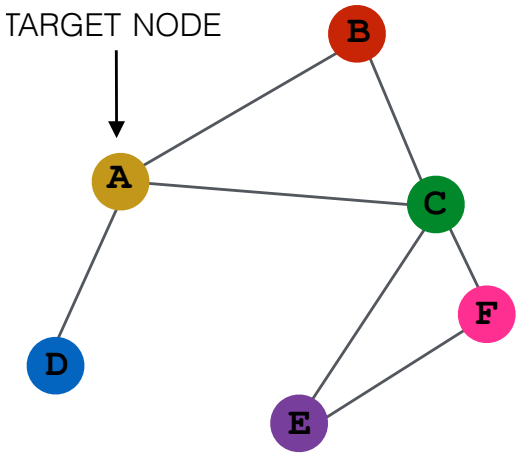


INPUT GRAPH

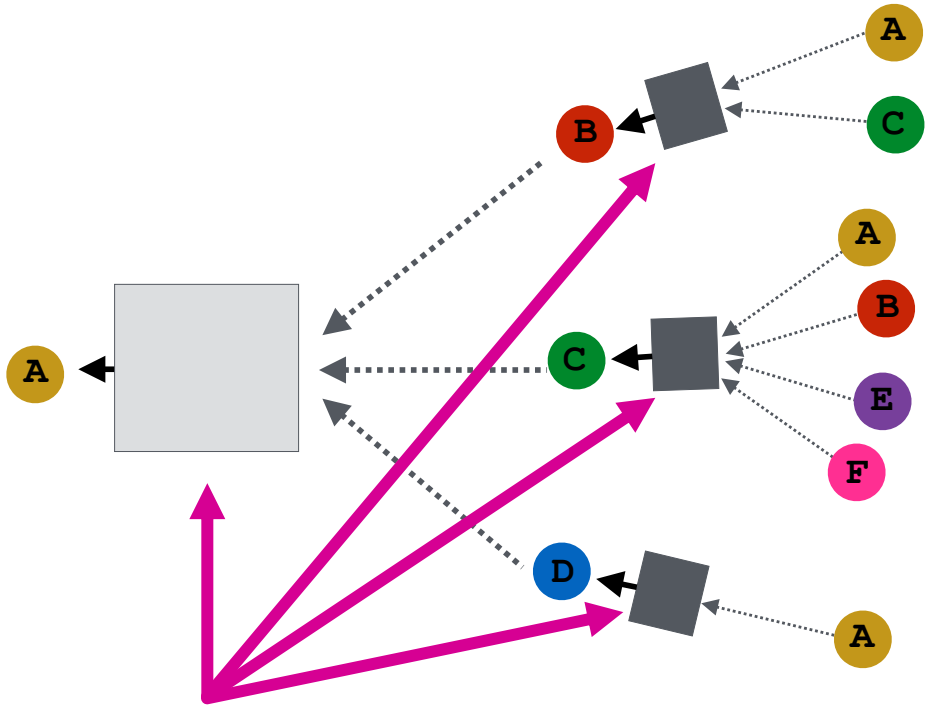


Idea: Aggregate Neighbors

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



INPUT GRAPH

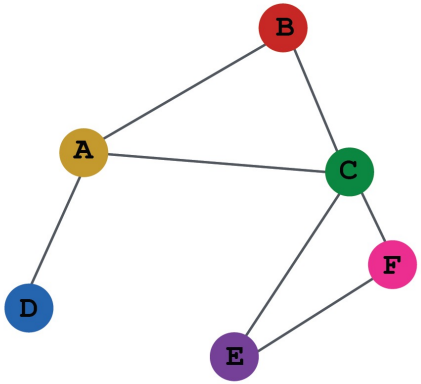


Neural networks

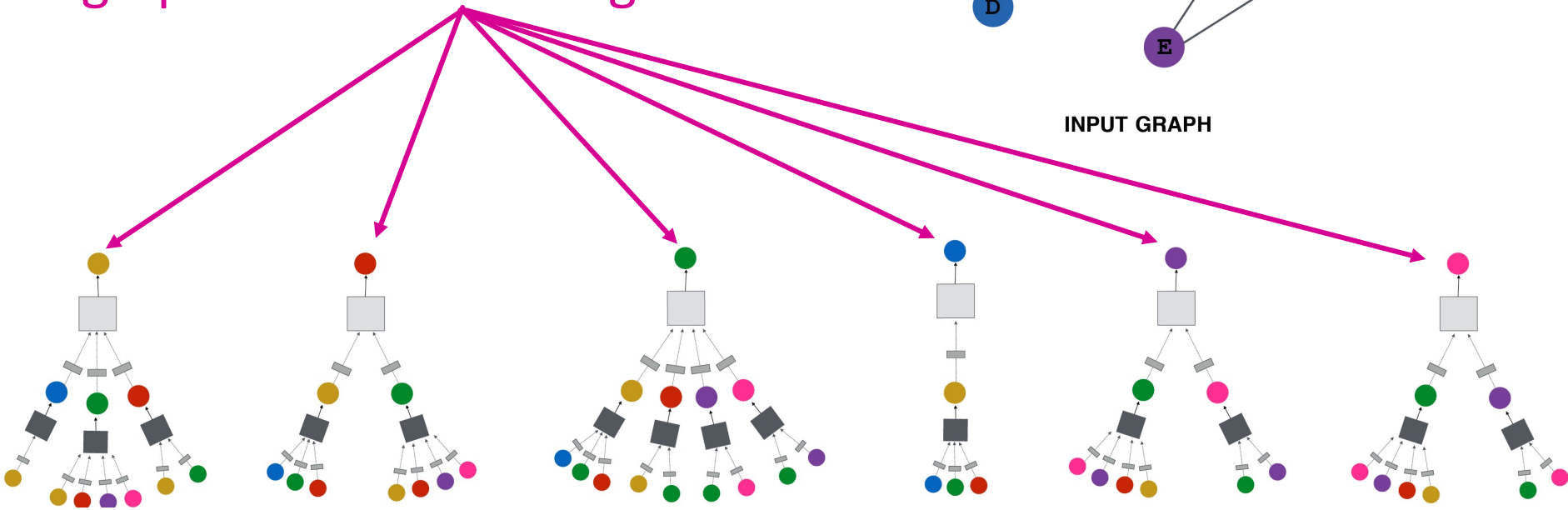
Idea: Aggregate Neighbors

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

Every node defines a computation graph based on its neighborhood!

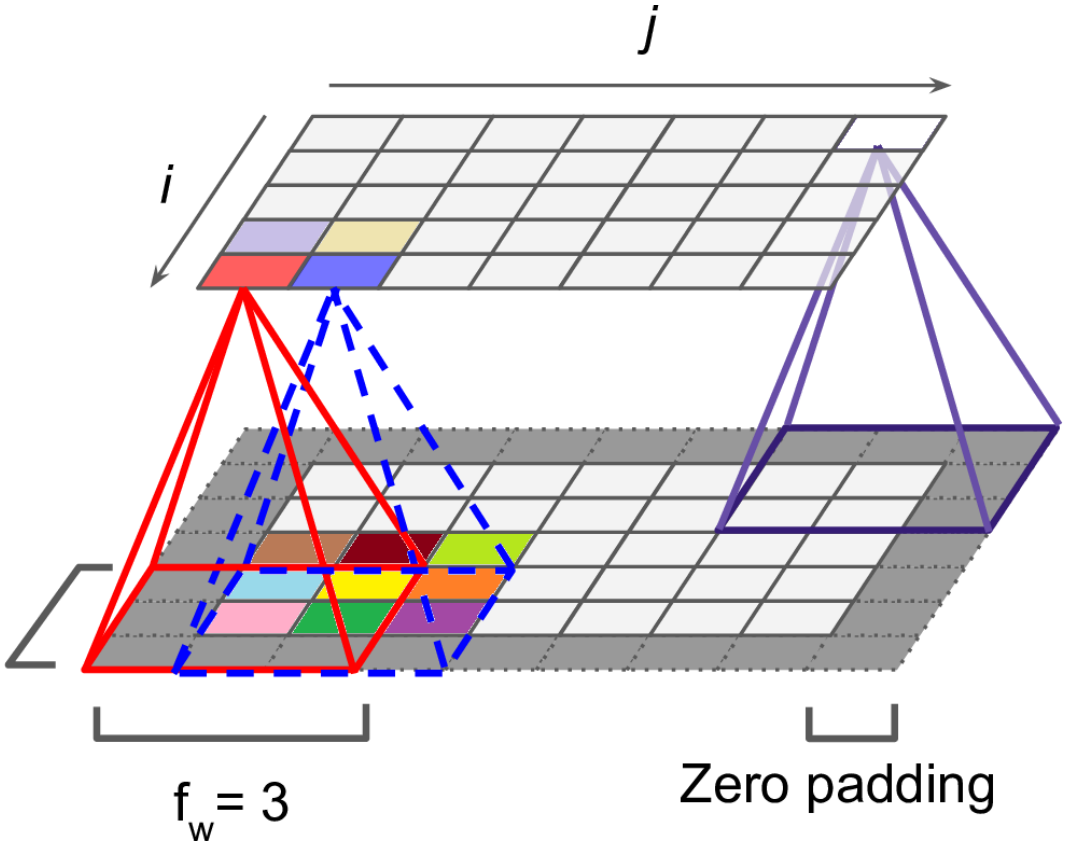


INPUT GRAPH



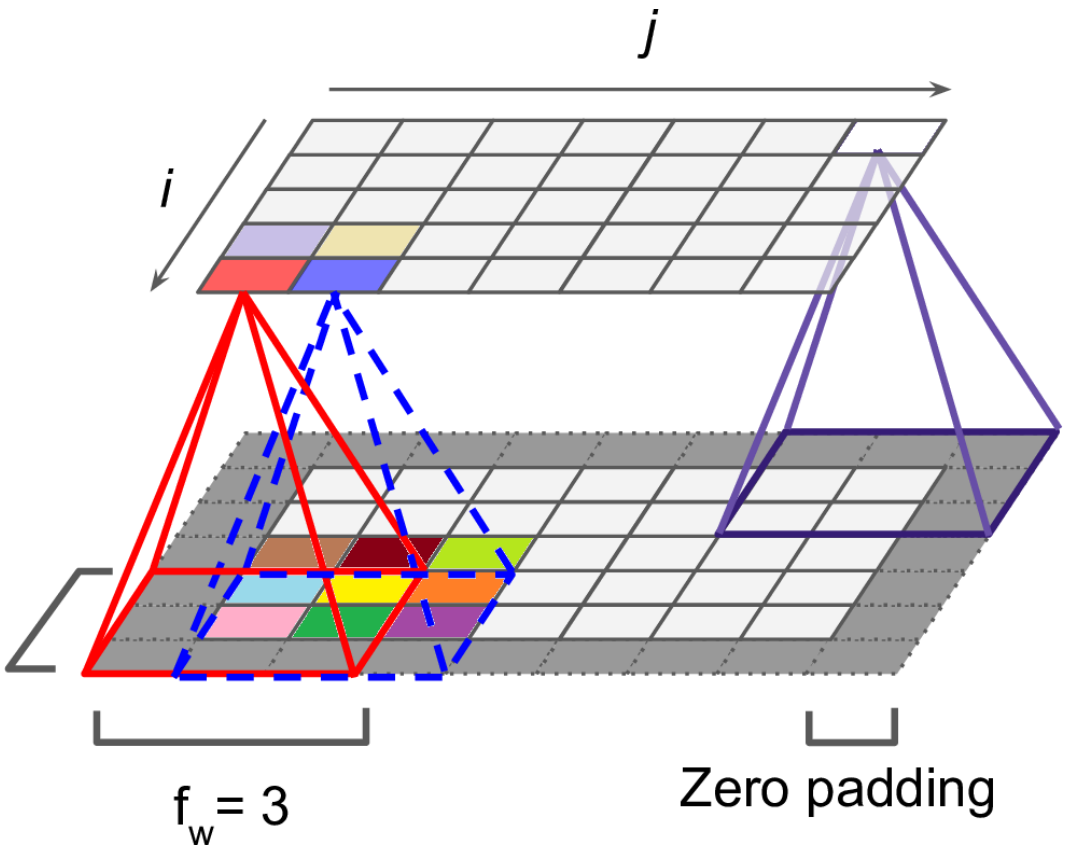
Computation Graphs

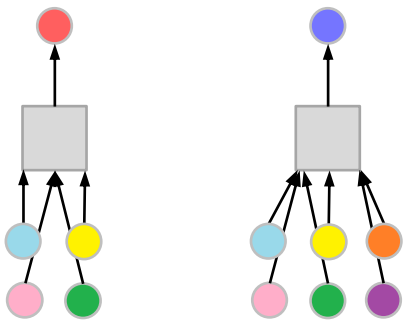
- **Analogy to CNNs**
  - In CNNs, each "pixel" in the next layer also defines a computation graph based on its "neighborhood"



Computation Graphs

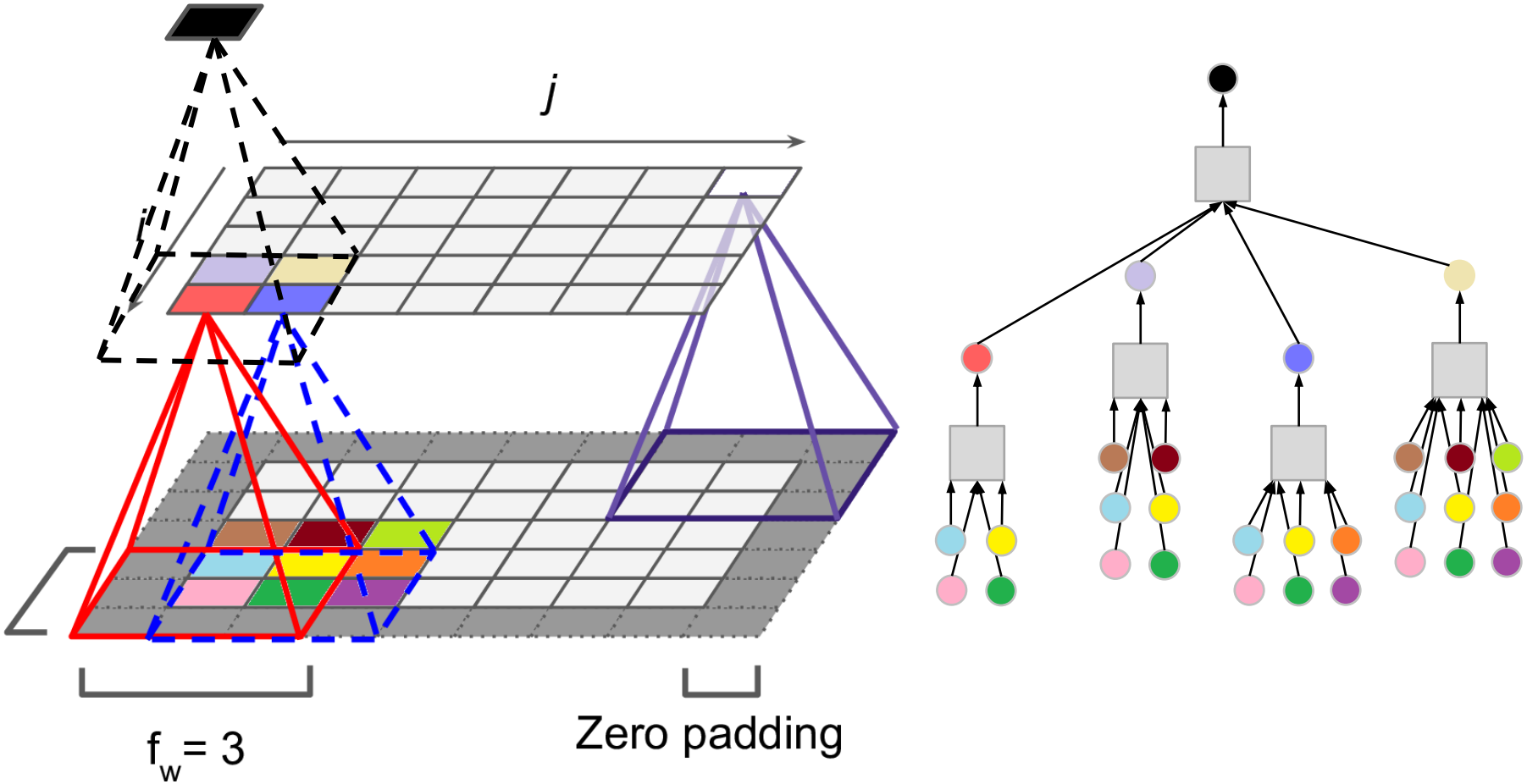
- Analogy to CNNs
  - First level computation for red and blue "pixels"





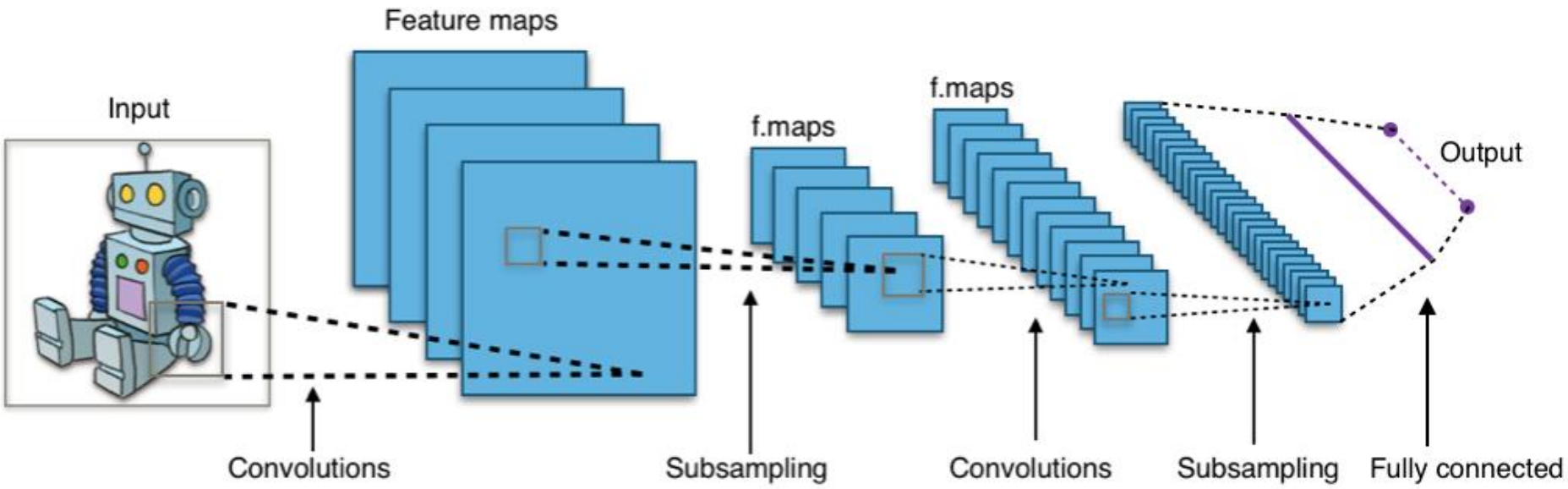
Computation Graphs

- **Analogy to CNNs**
  - Computation for the "pixel" above the red "pixel"



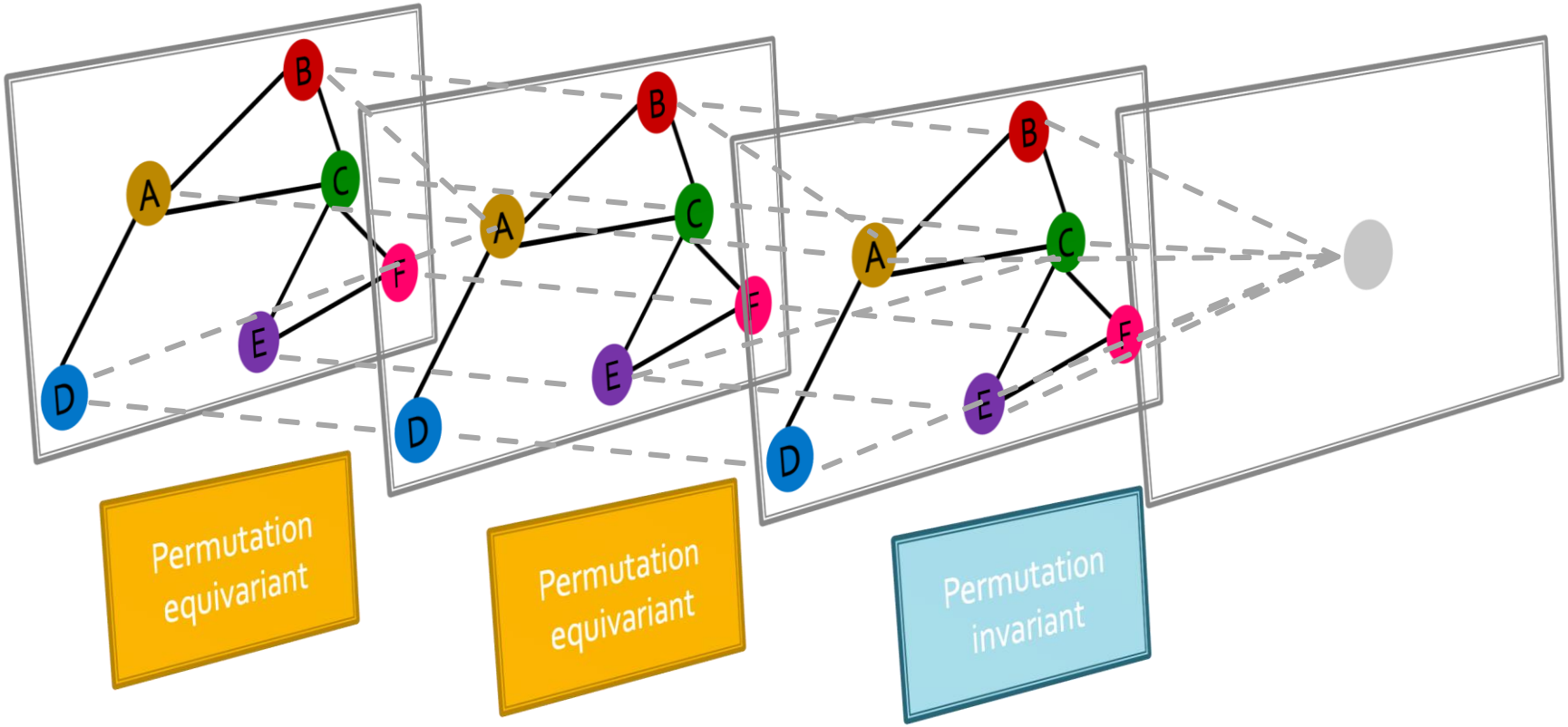
Two Intuitions

- **Analogy to CNNs**
  - We showed CNNs can be thought of as having custom computation graphs for each "pixel"
  - But we usually think of CNNs as having layers



Two Intuitions

- Graph neural networks can also be viewed from the perspective of computation graphs or as layers

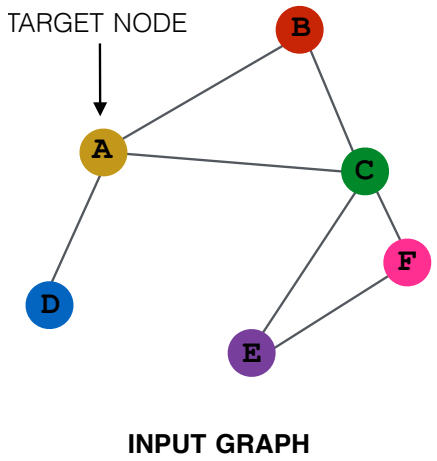


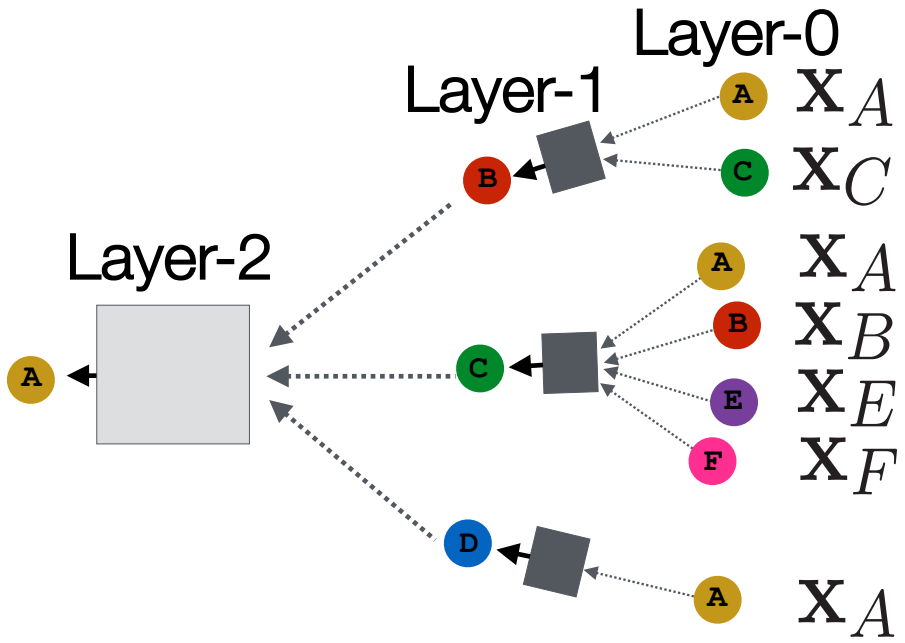
Two Intuitions

- **Analogy to CNNs**
  - Each layer in a GNN will have the same number of nodes, like a CNN with padding has the same dimensions
  - The embedding size for each layer does not have to be the same, like the number of channels for each layer in a CNN does not have to be the same

Deep Model: Many Layers

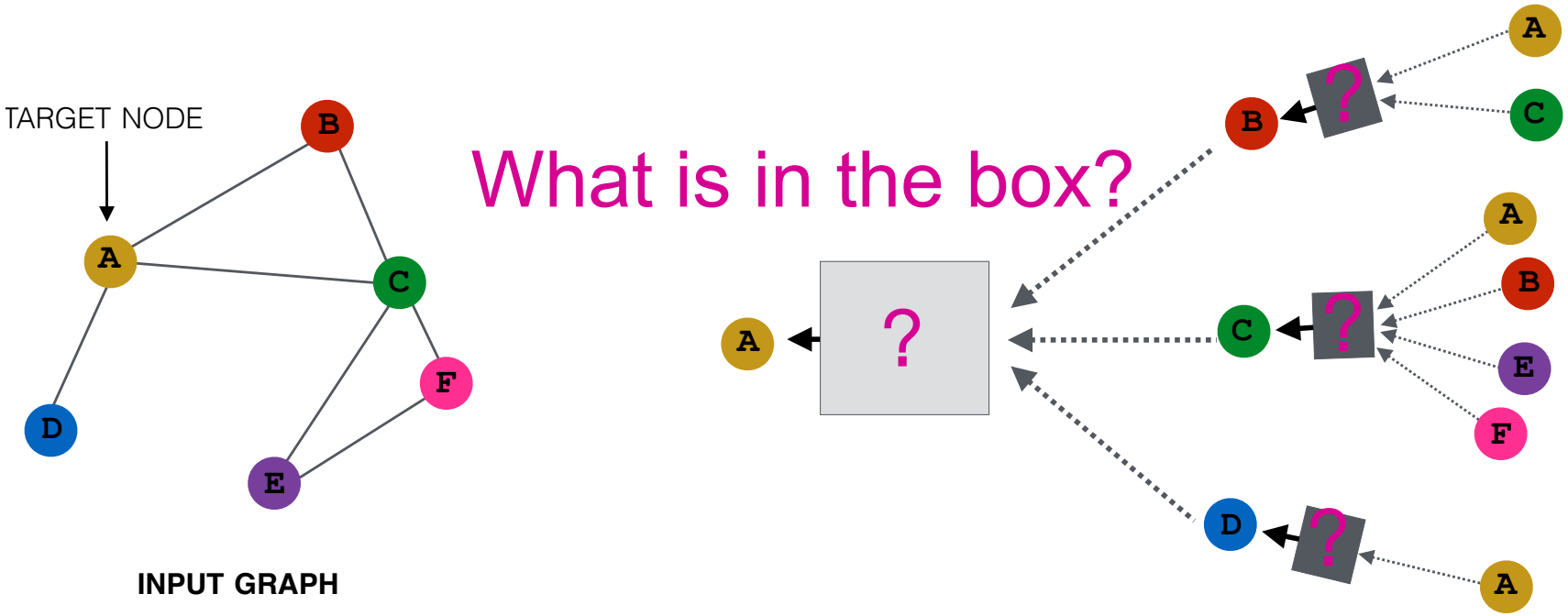
- Model can be of arbitrary depth:
  - Nodes have embeddings at each layer
  - Layer-0 embedding of node v is its input feature, x_v
  - Layer-k embedding gets information from nodes that are k hops away





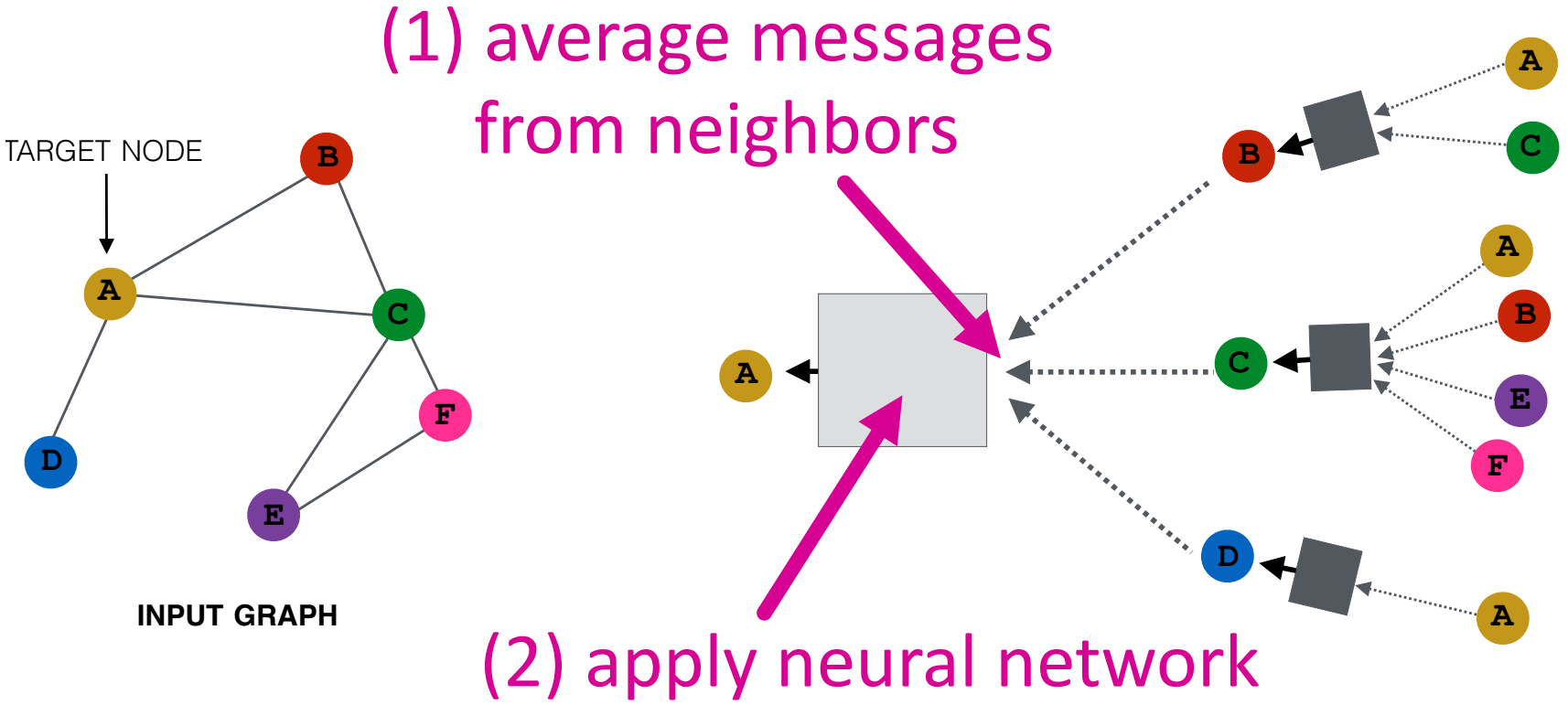
Neighborhood Aggregation

- **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



The Math: Deep Encoder

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

Initial 0-th layer embeddings are equal to node features

$$\mathrm{h}_v^0 = \mathrm{x}_v$$

The Math: Deep Encoder

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

Initial 0-th layer embeddings are equal to node features

$$\mathrm{h}_v^0 = \mathrm{x}_v$$

$$\mathrm{W}_k \sum_{u \in \mathrm{N}(v)} \frac{\mathrm{h}_u^{(k)}}{|\mathrm{N}(v)|}$$

Average of neighbor's previous layer embeddings

Notice summation is a permutation invariant pooling/aggregation.

The Math: Deep Encoder

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

Initial 0-th layer embeddings are equal to node features

$$\mathrm{h}_v^0 = \mathrm{x}_v$$

embedding of v at layer k

$$\mathrm{W}_k \sum_{u \in \mathrm{N}(v)} \frac{\mathrm{h}_u^{(k)}}{|\mathrm{N}(v)|} + \mathrm{B}_k \mathrm{h}_v^{(k)}$$

Average of neighbor's previous layer embeddings

Notice summation is a permutation invariant pooling/aggregation.

The Math: Deep Encoder

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

Initial 0-th layer embeddings are equal to node features

$$\mathrm{h}_v^0 = \mathrm{x}_v$$

embedding of v at layer k

$$\mathrm{h}_v^{(k+1)} = \sigma\left(\mathrm{W}_k \sum_{u \in \mathrm{N}(v)} \frac{\mathrm{h}_u^{(k)}}{|\mathrm{N}(v)|} + \mathrm{B}_k \mathrm{h}_v^{(k)}\right)$$

Average of neighbor's previous layer embeddings

Non-linearity (e.g., ReLU)

Notice summation is a permutation invariant pooling/aggregation.

The Math: Deep Encoder

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

Initial 0-th layer embeddings are equal to node features

$$\mathrm{h}_v^0 = \mathrm{x}_v$$

embedding of v at layer k

$$\mathrm{h}_v^{(k+1)} = \sigma\left(\mathrm{W}_k \sum_{u \in \mathrm{N}(v)} \frac{\mathrm{h}_u^{(k)}}{|\mathrm{N}(v)|} + \mathrm{B}_k \mathrm{h}_v^{(k)}\right), \forall k \in \{0, \ldots, K-1\}$$

Average of neighbor's previous layer embeddings

Total number of layers

Non-linearity (e.g., ReLU)

Notice summation is a permutation invariant pooling/aggregation.

The Math: Deep Encoder

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

Initial 0-th layer embeddings are equal to node features

$$\mathrm{h}_v^0 = \mathrm{x}_v$$

embedding of v at layer k

$$\mathrm{h}_v^{(k+1)} = \sigma\left(\mathrm{W}_k \sum_{u \in \mathrm{N}(v)} \frac{\mathrm{h}_u^{(k)}}{|\mathrm{N}(v)|} + \mathrm{B}_k \mathrm{h}_v^{(k)}\right), \forall k \in \{0, \ldots, K-1\}$$

Total number of layers

Average of neighbor's previous layer embeddings

$$\mathrm{z}_v = \mathrm{h}_v^{(K)}$$

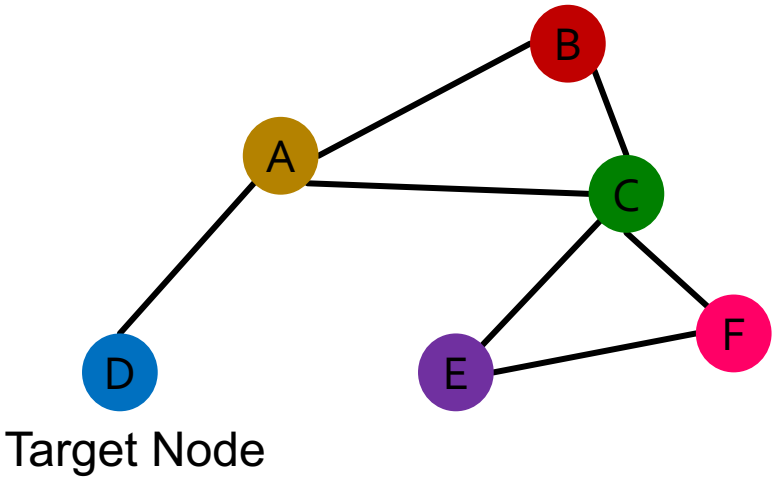
Embedding after L layers of neighborhood aggregation

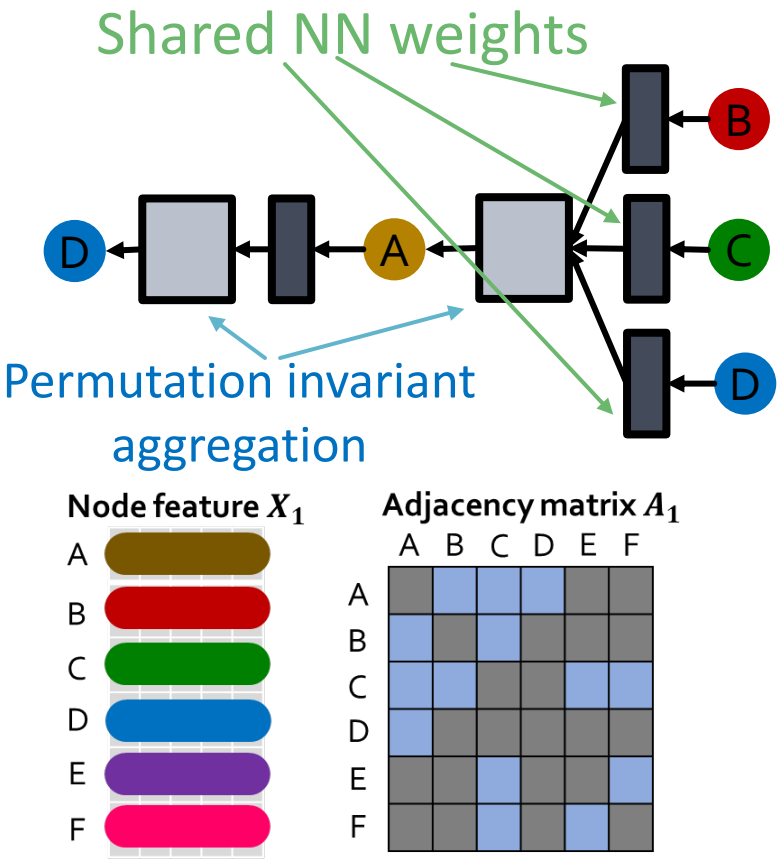
Non-linearity (e.g., ReLU)

Notice summation is a permutation invariant pooling/aggregation.

Equivariant Property

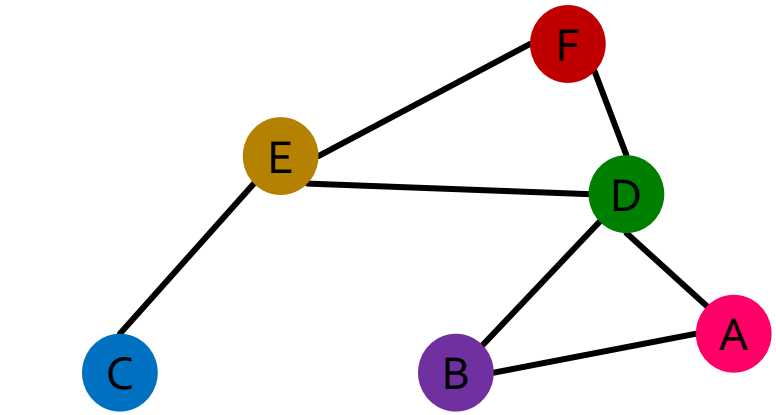
Message passing and neighbor aggregation in graph convolution networks is permutation equivariant.





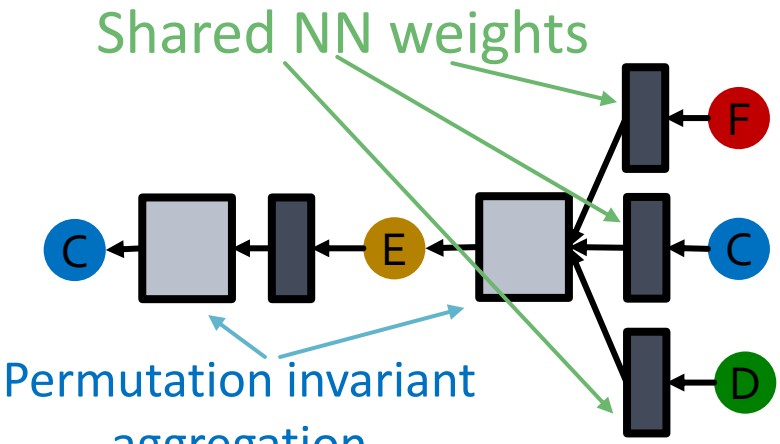
Equivariant Property

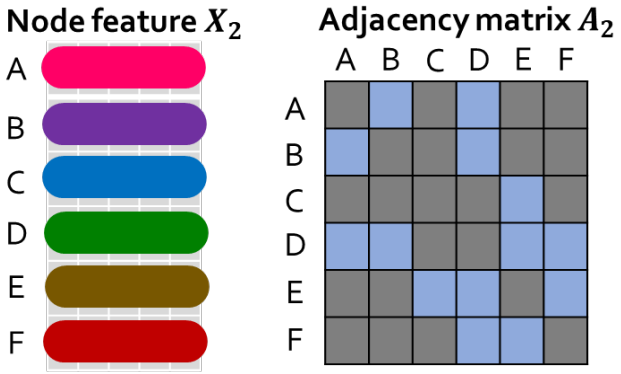
Message passing and neighbor aggregation in graph convolution networks is permutation equivariant.



Target Node

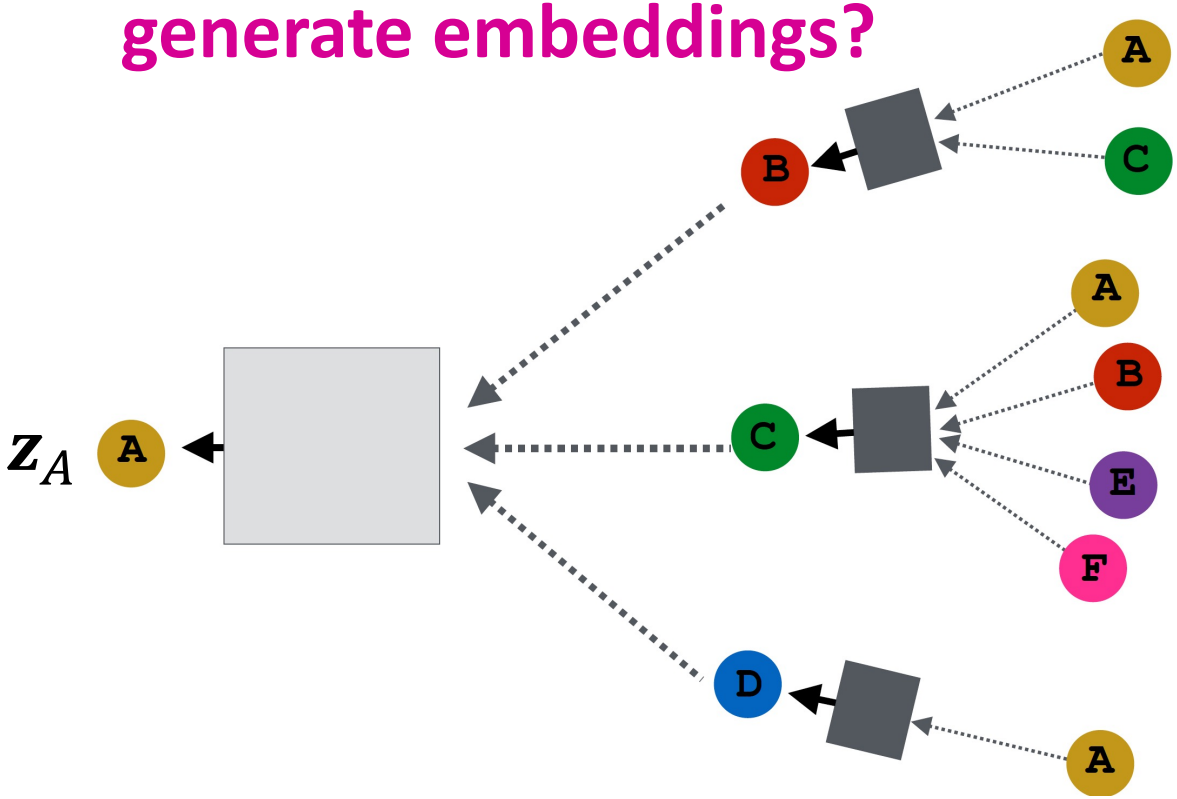
The target node (blue) has the same computation graph for different order plans





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)

$$\mathrm{h}_v^{(0)} = \mathrm{x}_v$$

$$\mathrm{h}_v^{(k+1)} = \sigma\left(\mathrm{W}_k \sum_{u \in \mathrm{N}(v)} \frac{\mathrm{h}_u^{(k)}}{|\mathrm{N}(v)|} + \mathrm{B}_k \mathrm{h}_v^{(k)}\right), \forall k \in \{0..K-1\}$$

$$\mathrm{z}_v = \mathrm{h}_v^{(K)}$$

Final node embedding

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

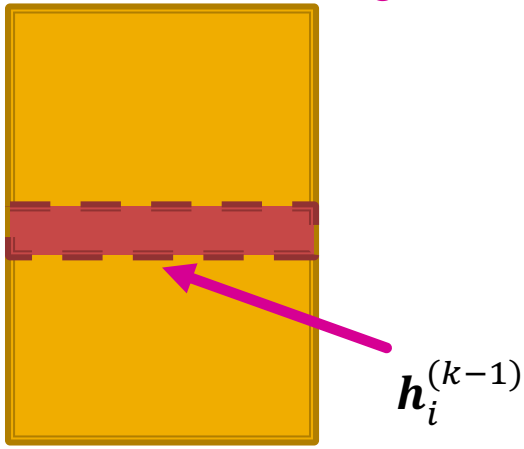
h_v^k: the hidden representation of node v at layer k

- 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,**

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



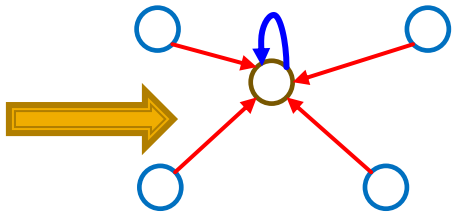
$$\sum_{u \in N(v)} \frac{h_u^{(k-1)}}{|N(v)|} \Longrightarrow H^{(k+1)} = D^{-1}AH^{(k)}$$

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**

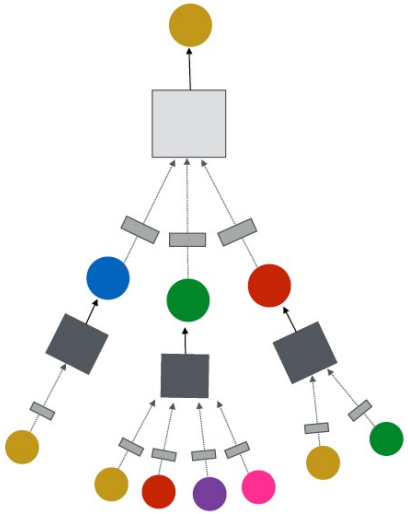
$$\mathcal{L} = \sum_{z_u, z_v} \mathrm{CE}(y_{u,v}, \mathrm{DEC}(z_u, z_v))$$

  - 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**

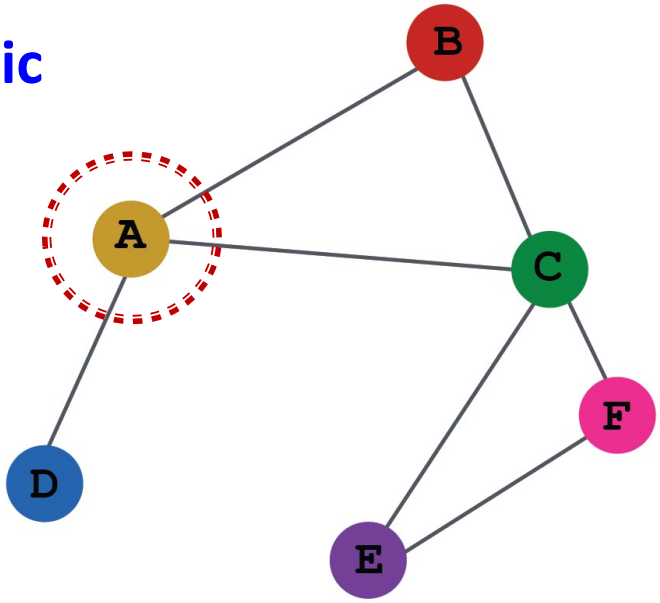
Supervised Training

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)

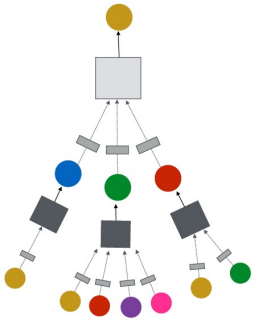
$$\mathcal{L} = \sum_{v \in V} y_v \log(\sigma(z_v^T \theta)) + (1 - y_v) \log(1 - \sigma(z_v^T \theta))$$

Encoder output: node embedding

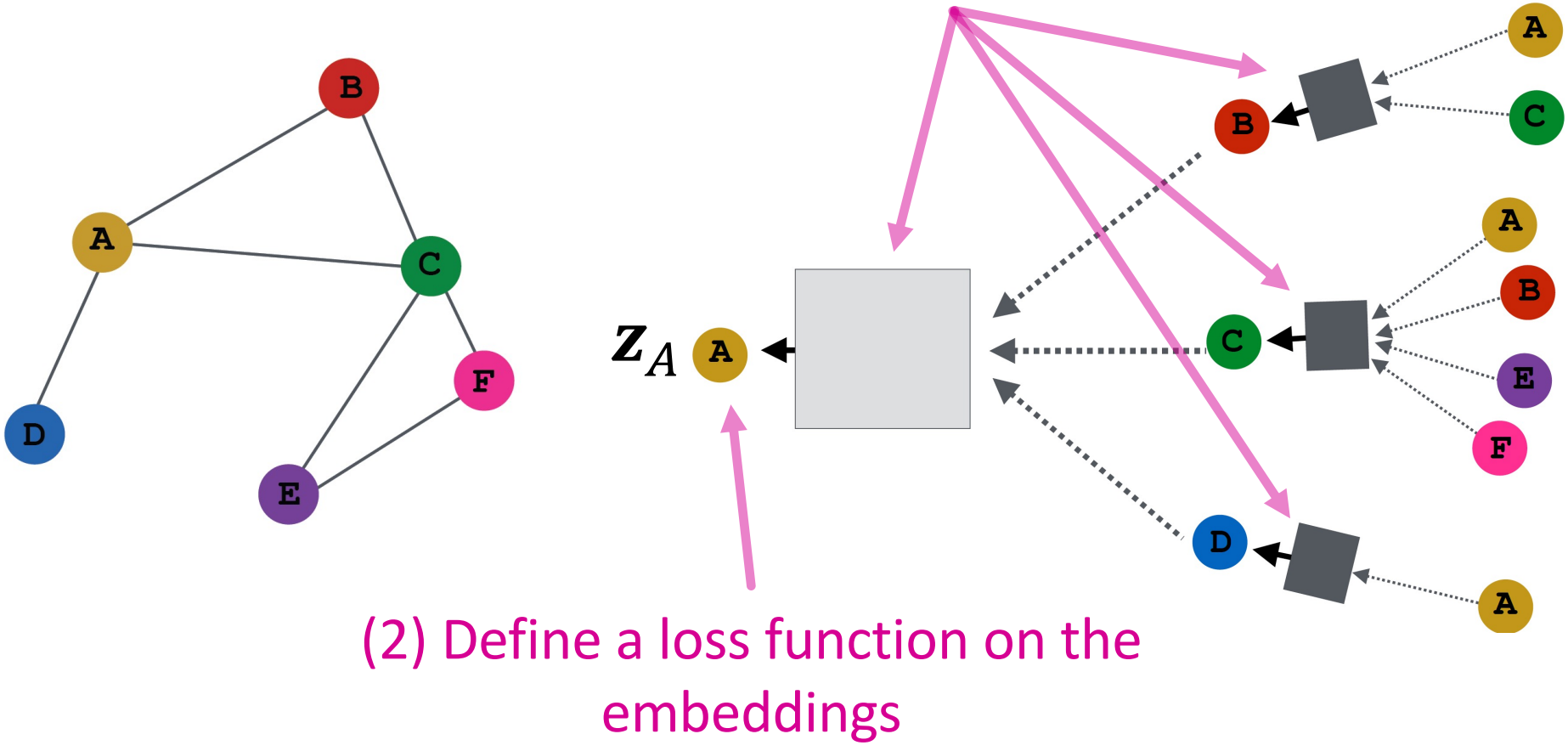
Classification weights

Node class label

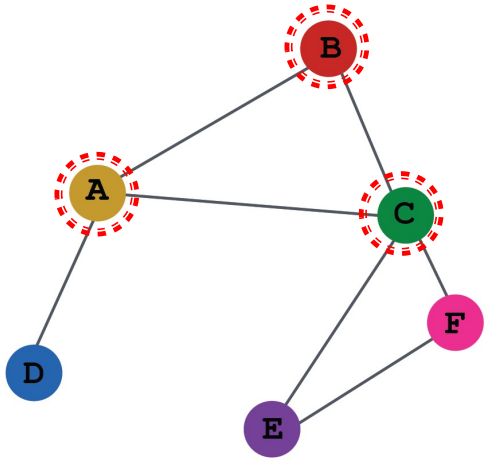
Safe or toxic drug?



Model Design: Overview

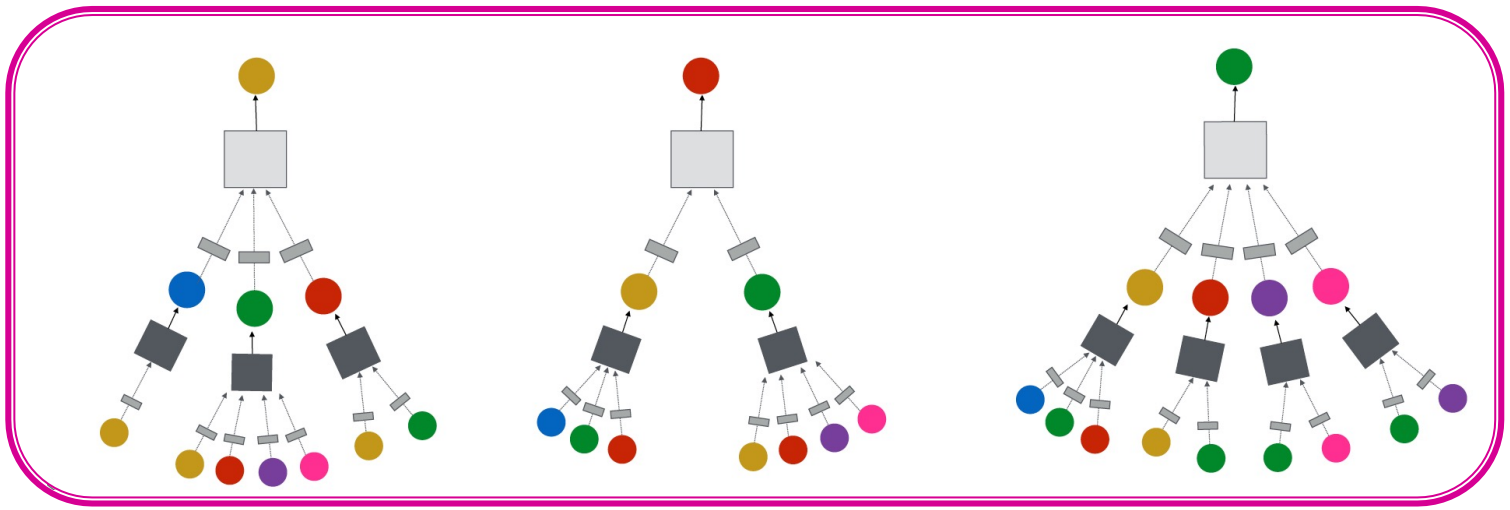


Model Design: Overview

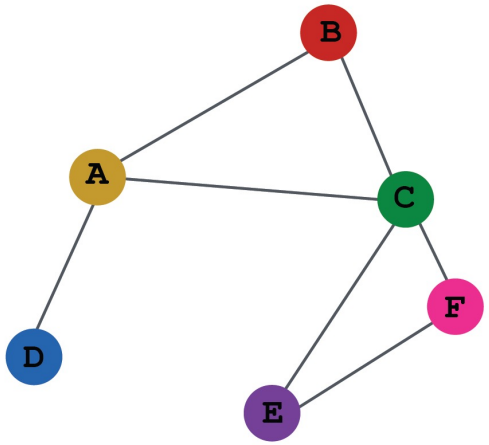


INPUT GRAPH

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



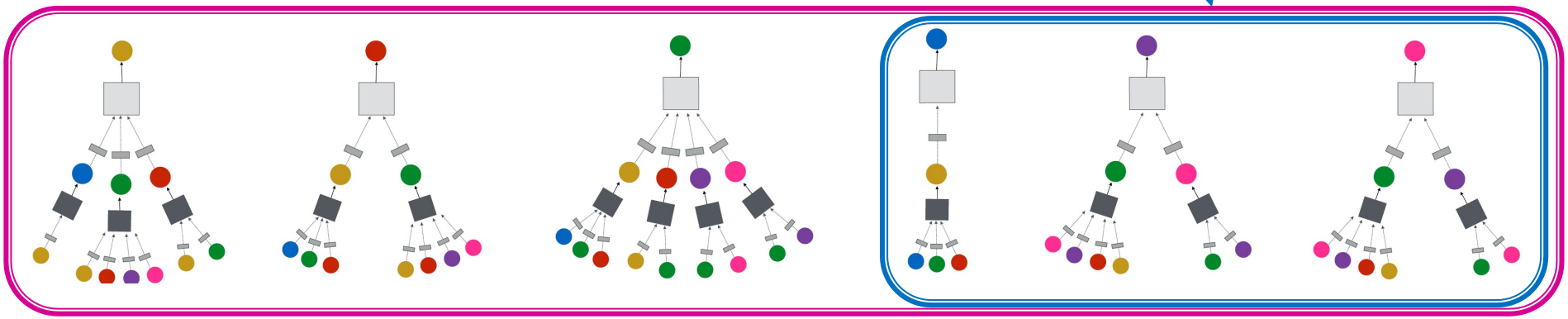
Model Design: Overview



INPUT GRAPH

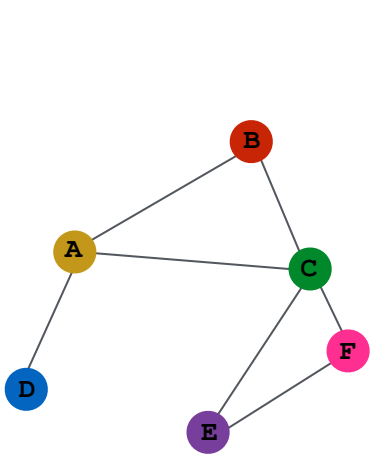
(4) Generate embeddings for nodes as needed

Even for nodes we never trained on!

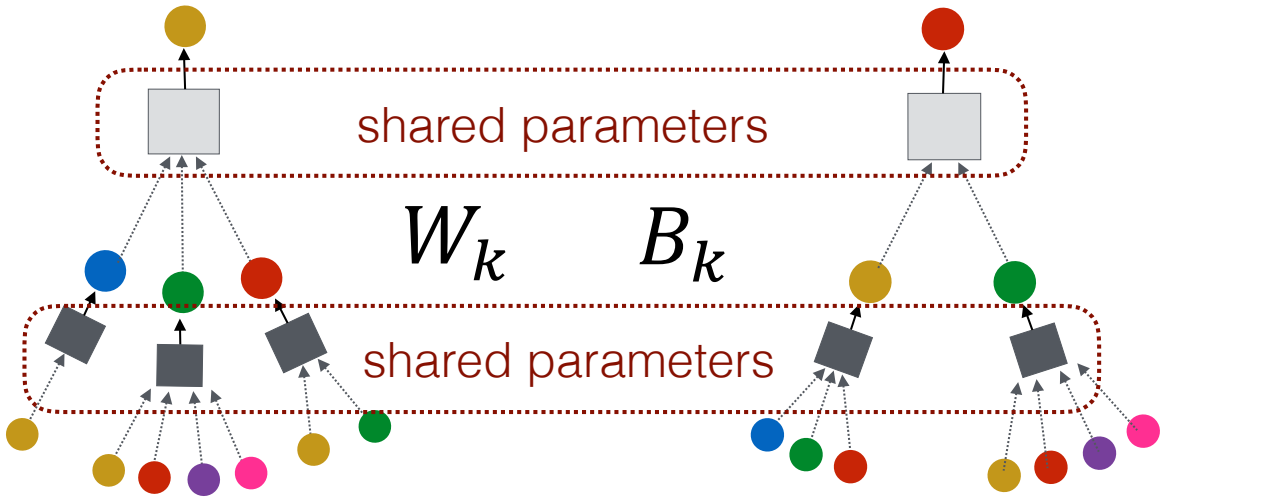


Inductive Capability

- **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**!



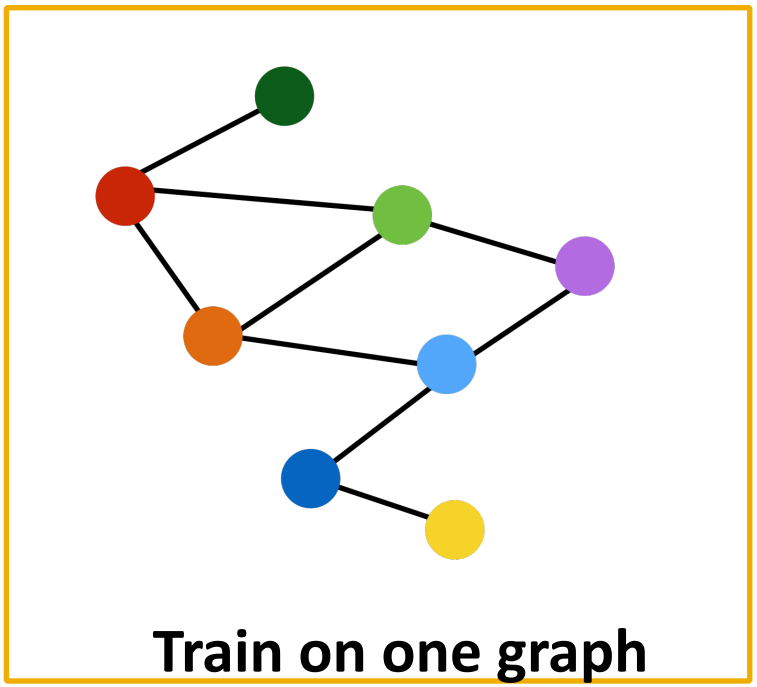
INPUT GRAPH



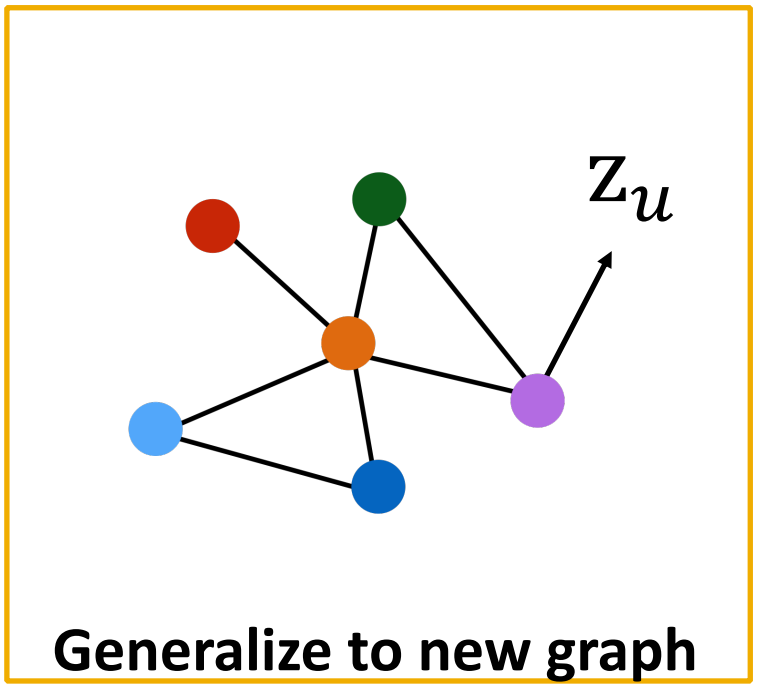
Compute graph for node A

Compute graph for node B

Inductive Capability: New Graphs



Train on one graph

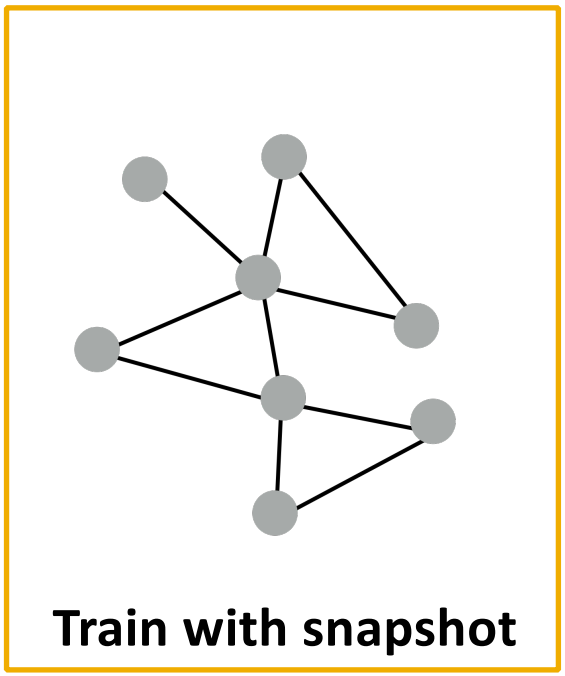


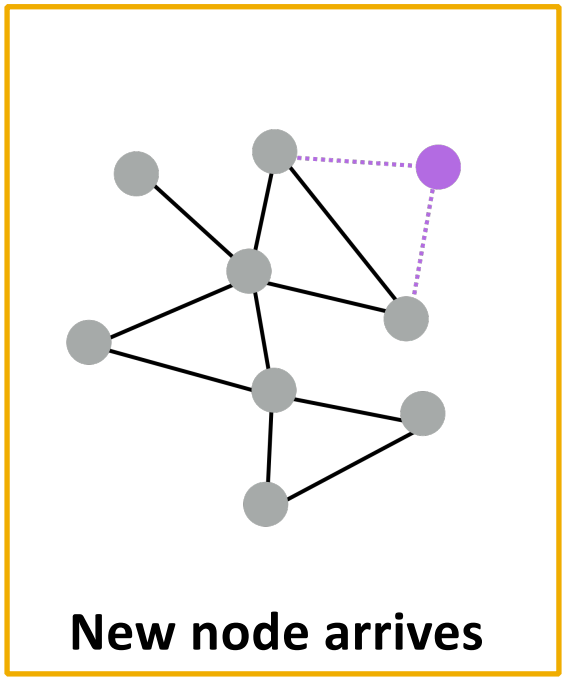
Generalize to new graph

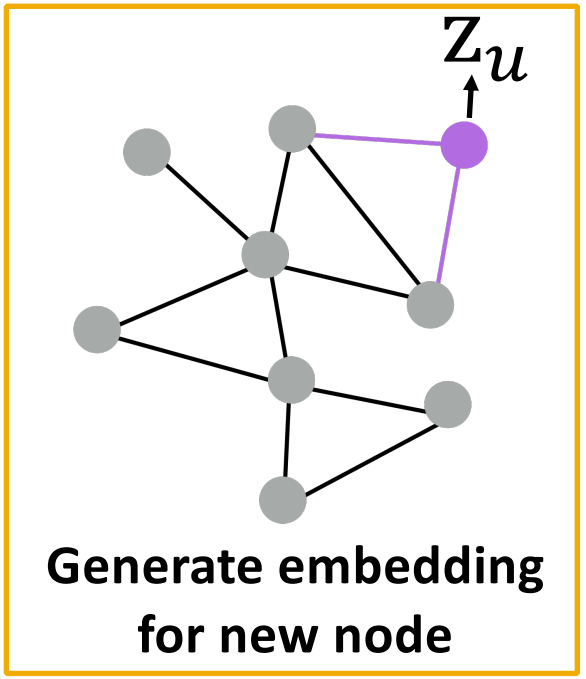
Inductive node embedding → Generalize to entirely unseen graphs

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







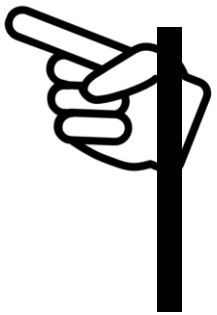
- Many application settings constantly encounter previously unseen nodes:
  - E.g., Reddit, YouTube, Google Scholar
- Need to generate new embeddings "on the fly"

Discussion

Questions?

Outline of Today's Lecture

1. Deep learning for graphs ✓
2. Graph Convolutional Networks ✓
3. **GNNs subsume CNNs and Transformers**



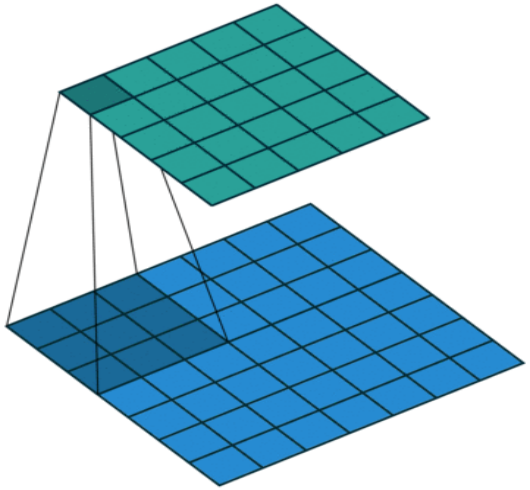
4. Appendix: Basics of deep learning

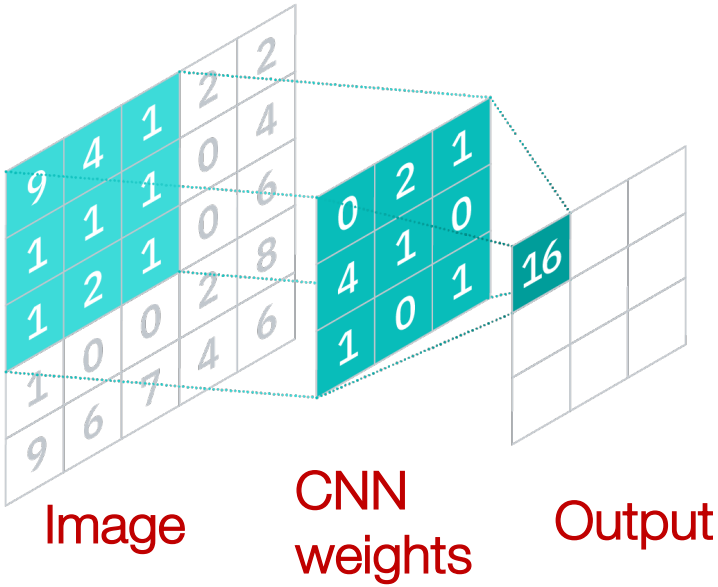
Architecture Comparison

- **How does GNNs compare to prominent architectures such as Convolutional Neural Nets, and Transformers?**

Convolutional Neural Network

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



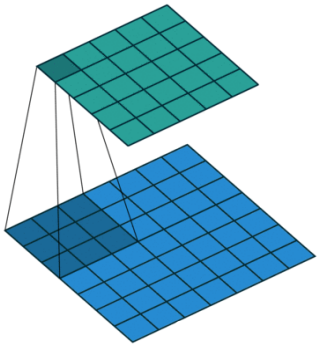


CNN formulation: $\mathrm{h}_v^{(l+1)} = \sigma(\sum_{u \in \mathrm{N}(v) \cup \{v\}} \mathrm{W}_l^u \mathrm{h}_u^{(l)}), \quad \forall l \in \{0, \ldots, L-1\}$

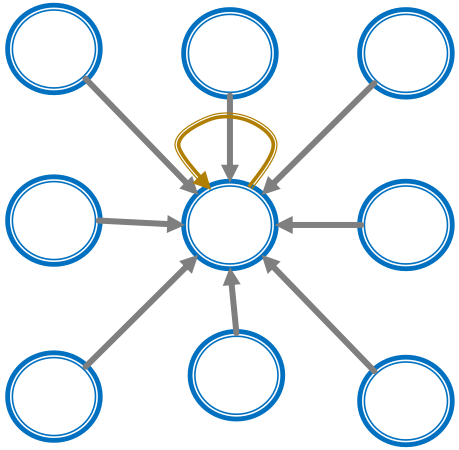
$N(v)$ represents the 8 neighbor pixels of v.

GNN vs. CNN

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



Image



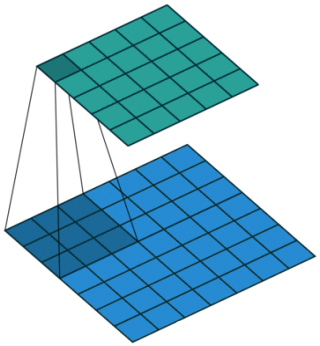
Graph

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

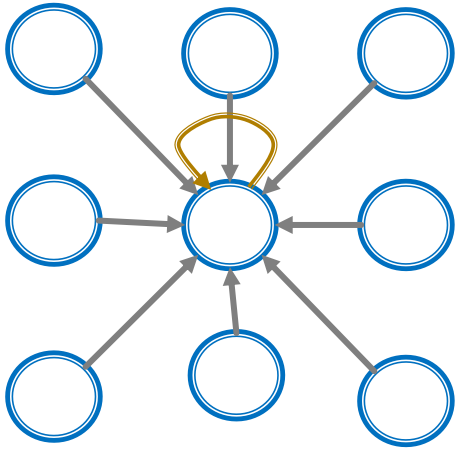
  if we rewrite: $\mathrm{h}_v^{(l+1)} = \sigma(\sum_{u \in \mathrm{N}(v)} \mathbf{W}_l^u \mathrm{h}_u^{(l)} + \mathrm{B}_l \mathrm{h}_v^{(l)}), \forall l \in \{0, \ldots, L-1\}$

GNN vs. CNN

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



Image



Graph

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

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

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

GNN vs. CNN

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

- **CNN can be seen as a special GNN with fixed neighbor size and ordering:**
  - **The size of the filter is pre-defined for a CNN.**
  - **The advantage of GNN is it processes arbitrary graphs with different degrees for each node.**

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

GNN vs. CNN

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

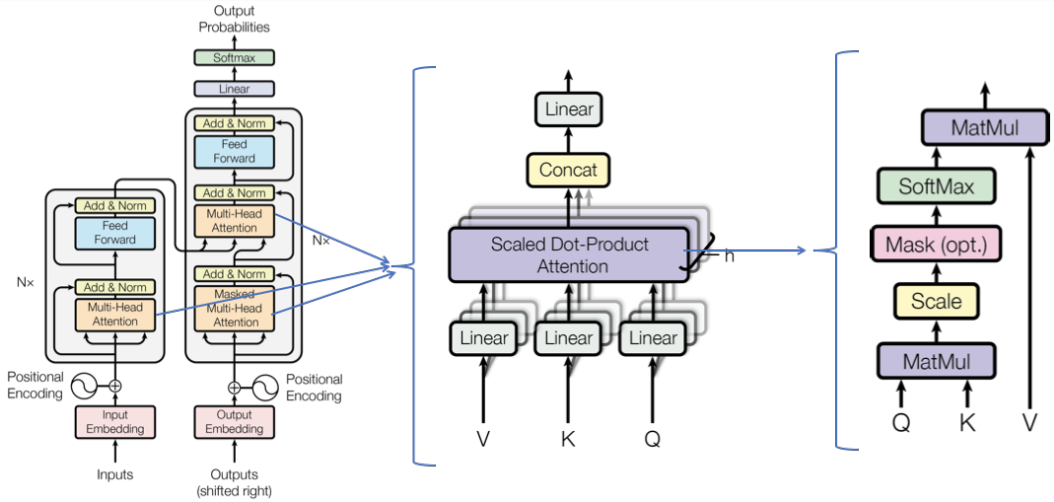
- **CNN can be seen as a special GNN with fixed neighbor size and ordering.**
- **CNN is not permutation equivariant.**
  - **Switching the order of pixels will leads to different outputs.**

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

[Attention is all you need. Vaswani et al., NeurIPS 2017]

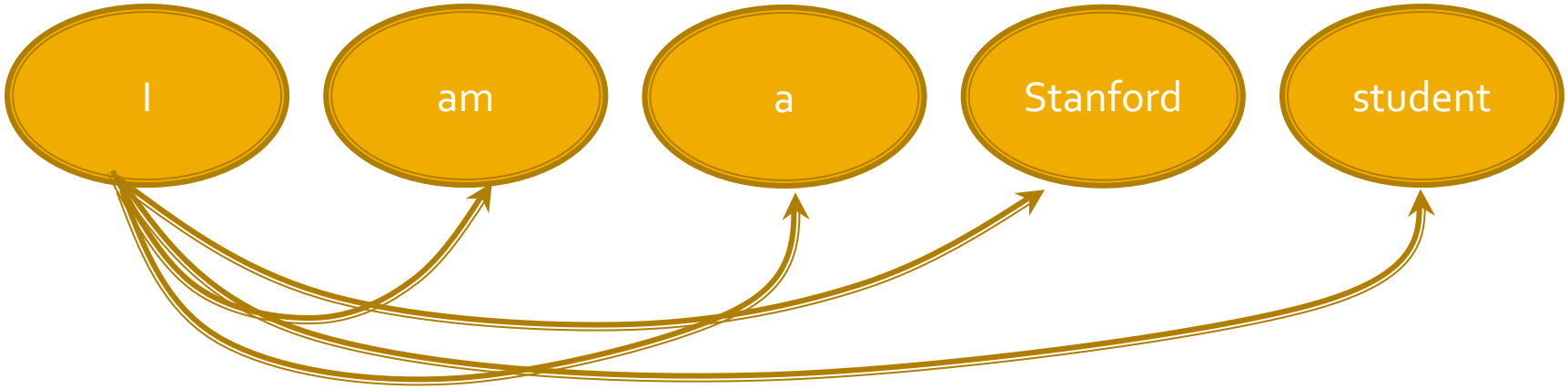
Transformer

Transformer is one of the most popular architectures that achieves great performance in many sequence modeling tasks.



Key component: self-attention

- Every token/word attends to all the other tokens/words via matrix calculation.

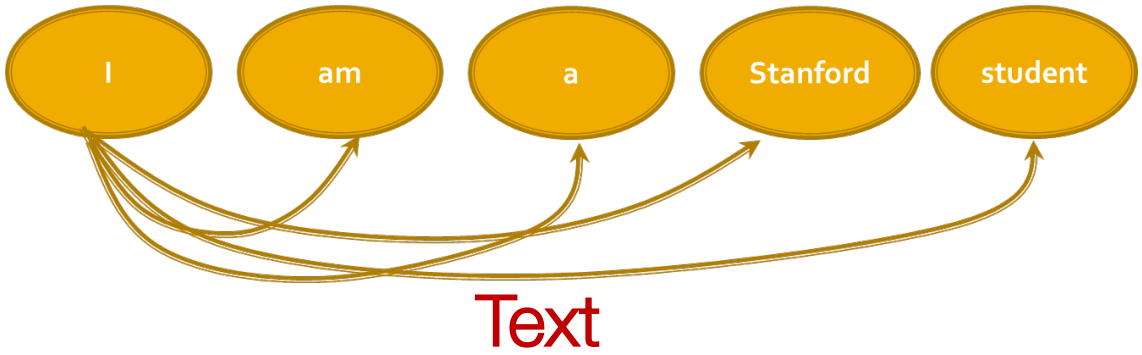


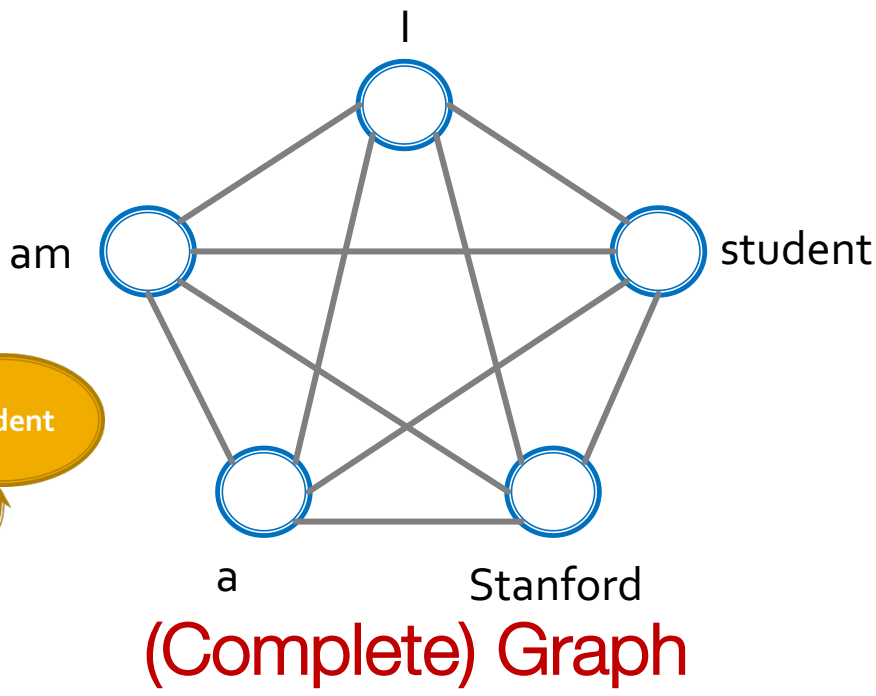
A nice blog plot for this: https://towardsdatascience.com/transformers-are-graph-neural-networks-bca9f75412aa

GNN vs. Transformer

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

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





Summary

- **In this lecture, we introduced**
  - 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 and Transformer can be viewed as a special GNN
  - Basics of neural networks
    - Loss, Optimization, Gradient, SGD, non-linearity, MLP

Summary

We also discussed:

- Computation graphs
- Basics of how to train/fit the model

What's next

Session Roadmap:

- **Today:** Intro to GNNs
  - Motivation
  - Aggregation of neighbors
  - Layers form custom computation graphs per node
- **Next:** GNN details
  - Choices for aggregation and message/update methods in a single layer
  - How to combine/stack layers
- **Then:** GNN augmentation and training

Stanford CS224W: Basics of Deep Learning

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

http://cs224w.stanford.edu

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

Loss Function Example

- One common loss for classification: **cross entropy (CE)**
- Label $\boldsymbol{y}$ is a categorical vector (one-hot encoding)
  - e.g. $\boldsymbol{y} =$ | 0 | 0 | 1 | 0 | 0 |

y is of class "3"

- $f(\boldsymbol{x}) = \text{Softmax}(g(\boldsymbol{x}))$
  - Recall from lecture 3: $f(\boldsymbol{x})_i = \frac{e^{g(x)_i}}{\sum_{j=1}^{C} e^{g(x)_j}}$

$g(x)_i$ denotes i-th coordinate of the vector output of func. $g(x)$

where C is the number of classes.

  - e.g. $f(\boldsymbol{x}) =$ | 0.1 | 0.3 | 0.4 | 0.1 | 0.1 |

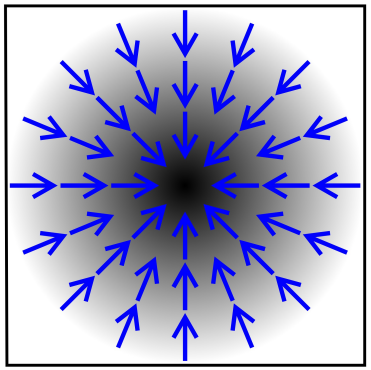
- $\text{CE}(\boldsymbol{y}, f(\boldsymbol{x})) = -\sum_{i=1}^{C}(y_i \log f(x)_i)$
  - y_i and $f(x)_i$ are the **actual** and **predicted** values of the i-th class.
  - **Intuition:** the lower the loss, the closer the prediction is to one-hot
- **Total loss over all training examples:**
  - $\mathcal{L} = \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{T}} \text{CE}(\boldsymbol{y}, f(\boldsymbol{x}))$
    - $\mathcal{T}$: training set containing all pairs of data and labels $(\boldsymbol{x}, \boldsymbol{y})$

Machine Learning as Optimization

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

Partial derivative

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



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

  - Θ_1, Θ_2 … : components of Θ
- 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) η:**
  - 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

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

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

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

- Recall **chain rule**:

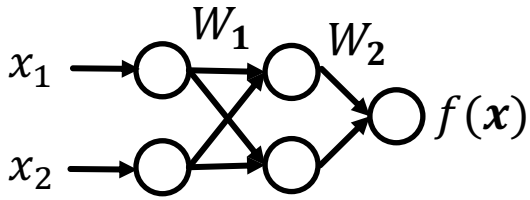
$$\frac{df}{dx} = \frac{dg}{dh} \cdot \frac{dh}{dx} \text{ or } f'(x) = g'\big(h(x)\big)h'(x)$$

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

- **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
- $f(\boldsymbol{x}) = g\big(h(x)\big) = W_2(W_1\boldsymbol{x})$



- $\mathcal{L} = \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{B}} \big|\big|\big(\boldsymbol{y}, -f(\boldsymbol{x})\big)\big|\big|_2$
  - The loss $\mathcal{L}$ sums the L2 loss in a minibatch $\mathcal{B}$.
- **Hidden layer:**
  - Intermediate representation of input $\boldsymbol{x}$
  - Here we use $h(x) = W_1\boldsymbol{x}$ to denote the hidden layer
  - $f(\boldsymbol{x}) = W_2 h(x)$

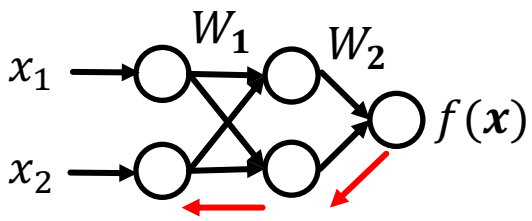
Back-propagation Example (2)

Remember:

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

$h(x) = W_1\boldsymbol{x}$

$g(z) = W_2 z$



- **Forward propagation:**

  Compute loss starting from input

  - $\boldsymbol{x} \longrightarrow h \longrightarrow g \longrightarrow \mathcal{L}$

    Multiply W_1 Multiply W_2 Loss

- **Back-propagation to compute gradient of**

$$\Theta = \{W_1, W_2\}$$

Start from loss, compute the gradient

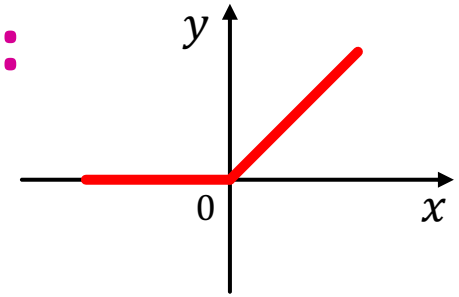
$$\frac{\partial \mathcal{L}}{\partial W_2} = \frac{\partial \mathcal{L}}{\partial f} \cdot \frac{\partial f}{\partial W_2}, \qquad \frac{\partial \mathcal{L}}{\partial W_1} = \frac{\partial \mathcal{L}}{\partial f} \cdot \frac{\partial f}{\partial W_2} \cdot \frac{\partial W_2}{\partial W_1}$$

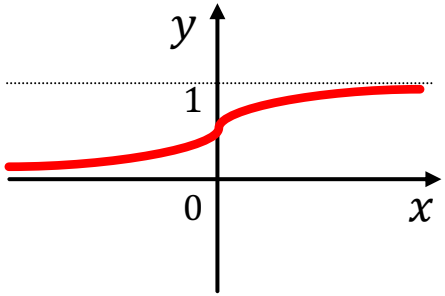
Compute backwards Compute backwards

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



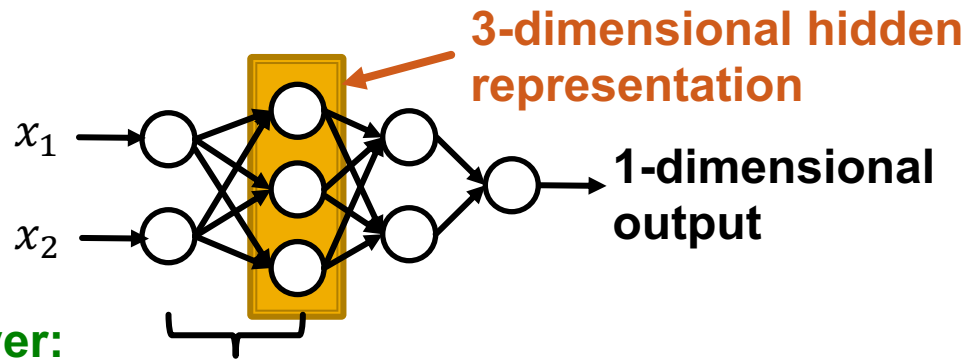


Multi-layer Perceptron (MLP)

- **Each layer of MLP combines linear transformation and non-linearity:**

$$\boldsymbol{x}^{(l+1)} = \sigma(W_l \boldsymbol{x}^{(l)} + b^l)$$

  - 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 $\boldsymbol{x}$
  - σ is non-linearity function (e.g., sigmod)
- 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.