

Note to other teachers and users of these slides: We would be delighted if you found our material useful for giving your own lectures. Feel free to use these slides verbatim, or to modify them to fit your own needs. If you make use of a significant portion of these slides in your own lecture, please include this message, or a link to our web site: <http://cs224w.Stanford.edu>

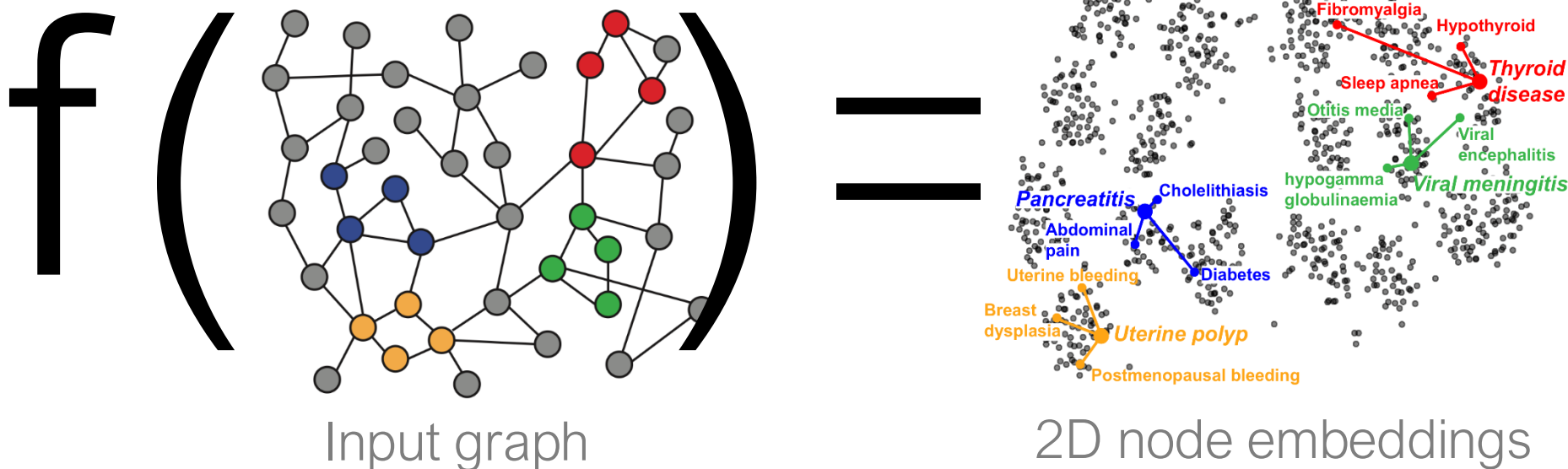
Stanford CS224W: Graph Neural Networks

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



Recap: Node Embeddings

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

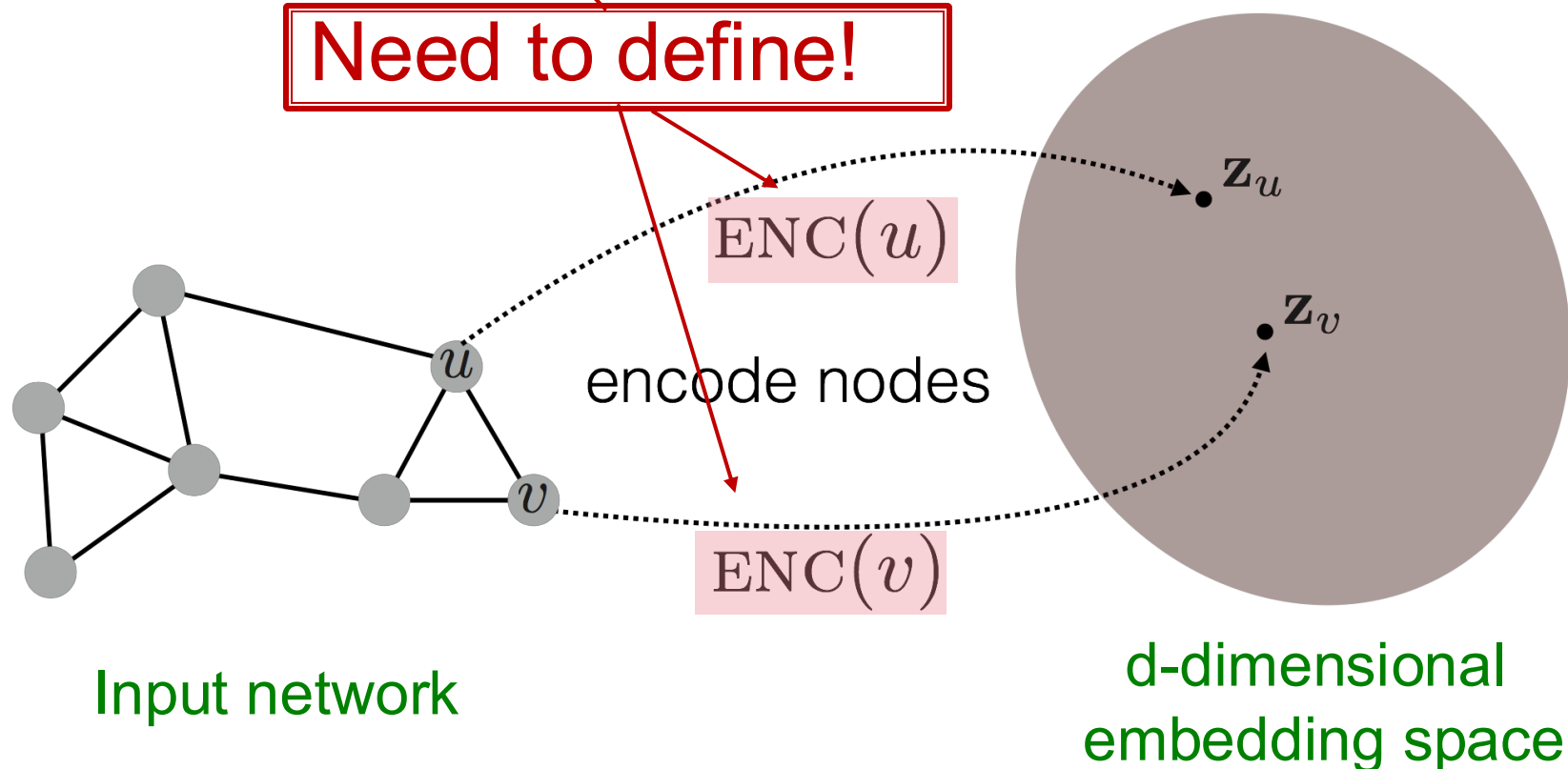


How to learn mapping function f ?

Recap: Node Embeddings

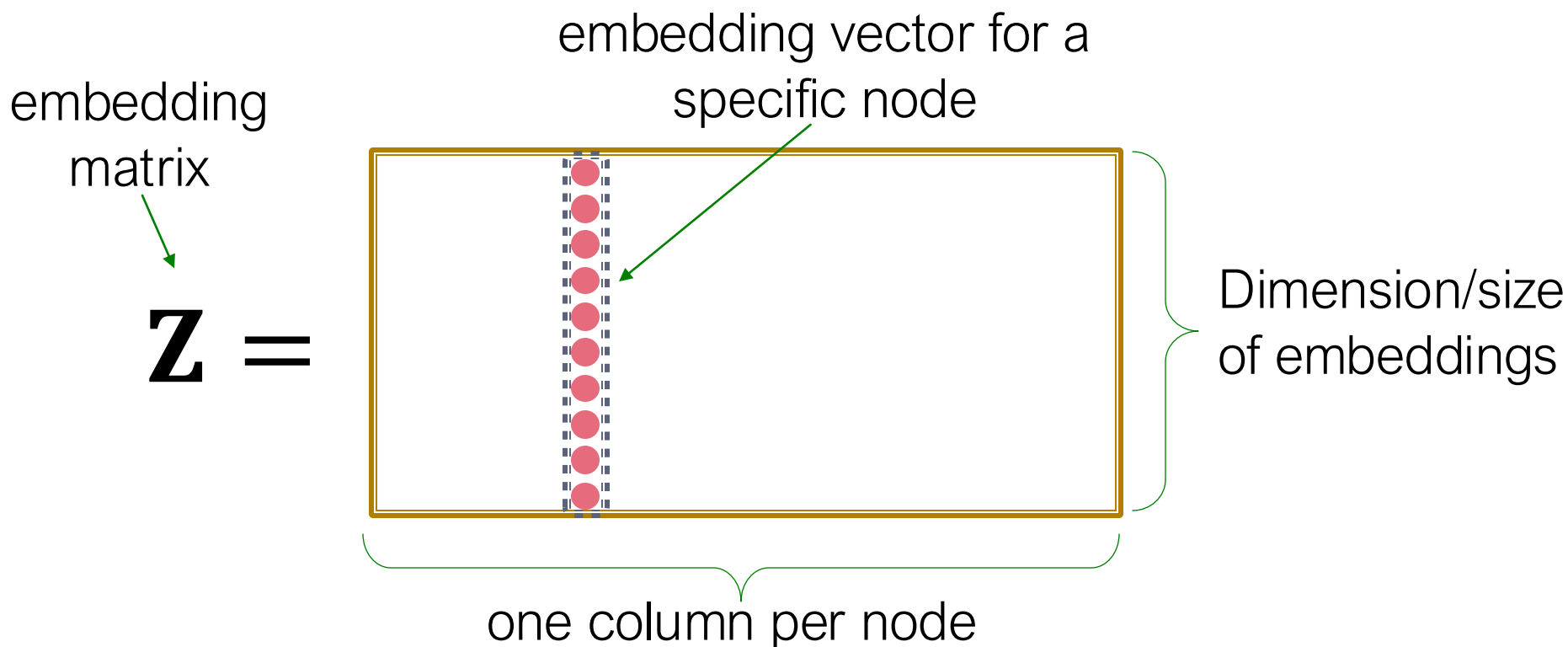
Goal: $\text{similarity}(u, v) \approx \mathbf{z}_v^T \mathbf{z}_u$

Need to define!



Recap: “Shallow” Encoding

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



Recap: Shallow Encoders

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

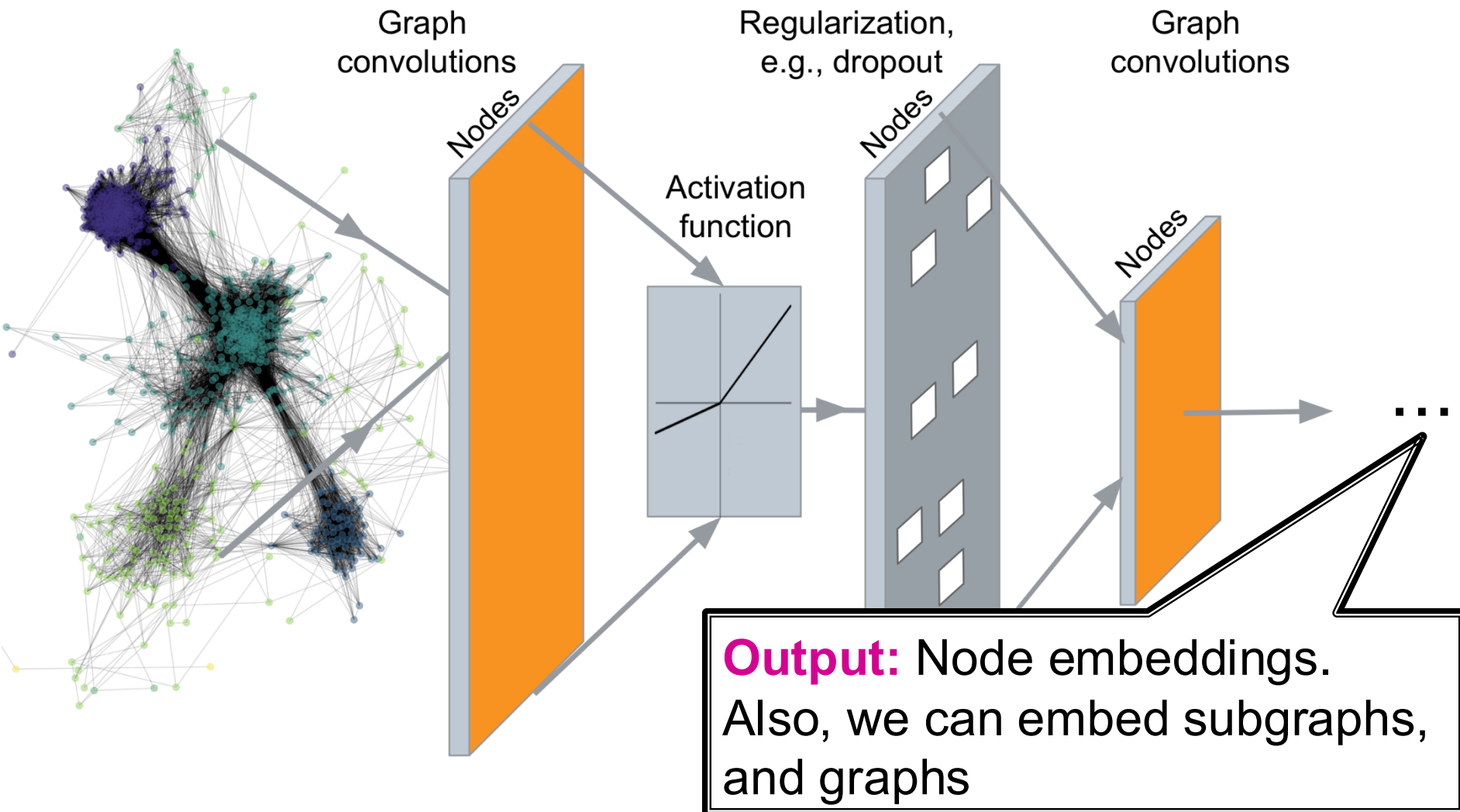
Today: Deep Graph Encoders

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

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

- **Note:** All these deep encoders can be **combined with node similarity functions** defined in the Lecture 2.

Deep Graph Encoders

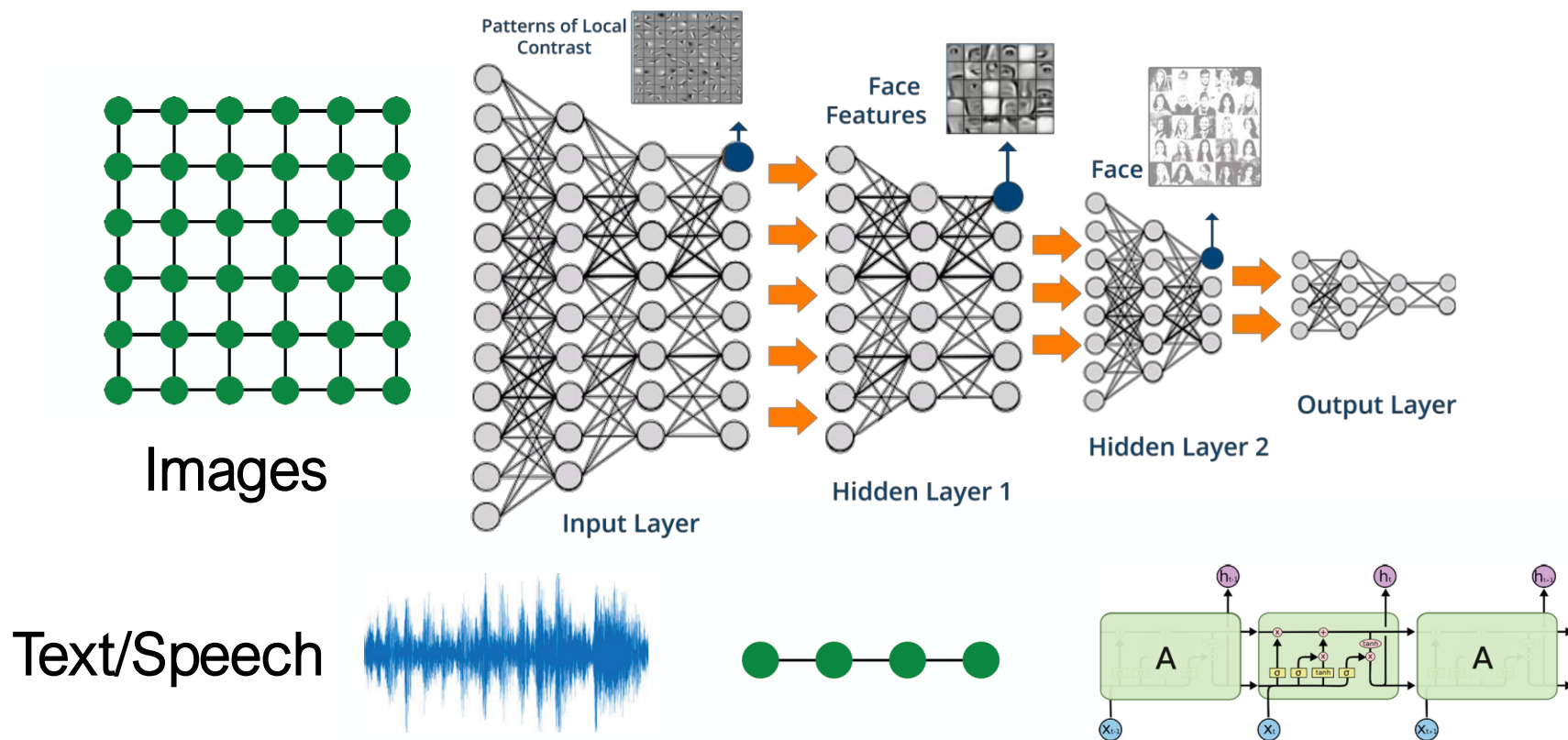


Tasks on Networks

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 two (sub)networks are

Modern ML Toolbox

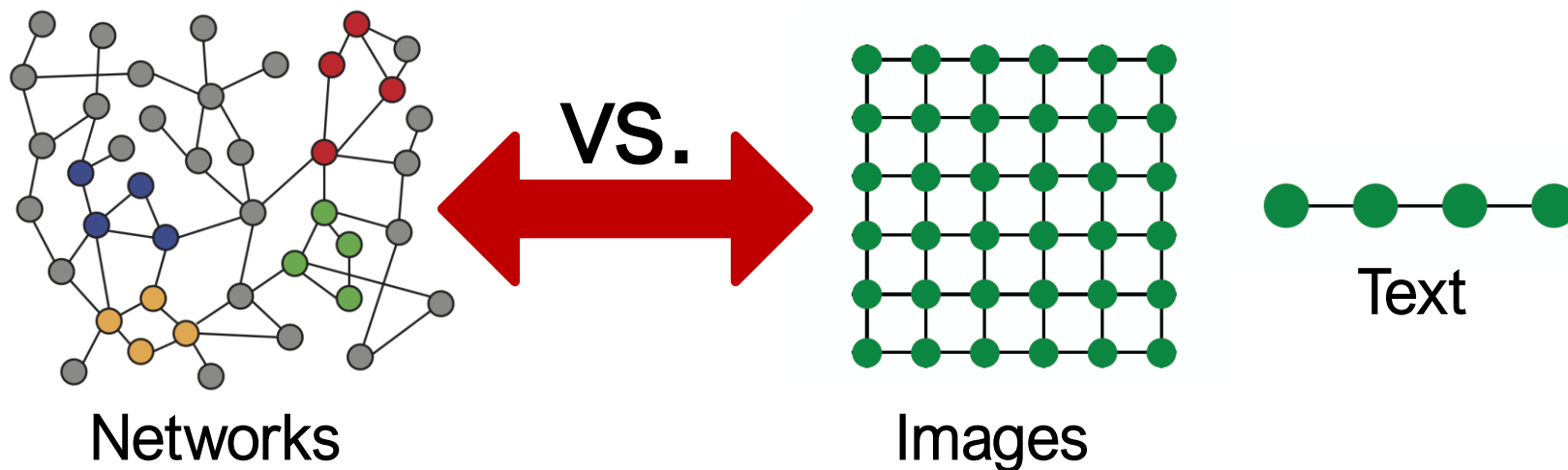


Modern deep learning toolbox is designed for simple sequences & grids

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

Background: Basics of Deep Learning

- **Loss function $\mathcal{L}$:**

$$\min_{\Theta} \mathcal{L}(\mathbf{y}, f_{\Theta}(\mathbf{x}))$$

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

Stanford CS224W: Deep Learning for Graphs

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

<http://cs224w.stanford.edu>



Content

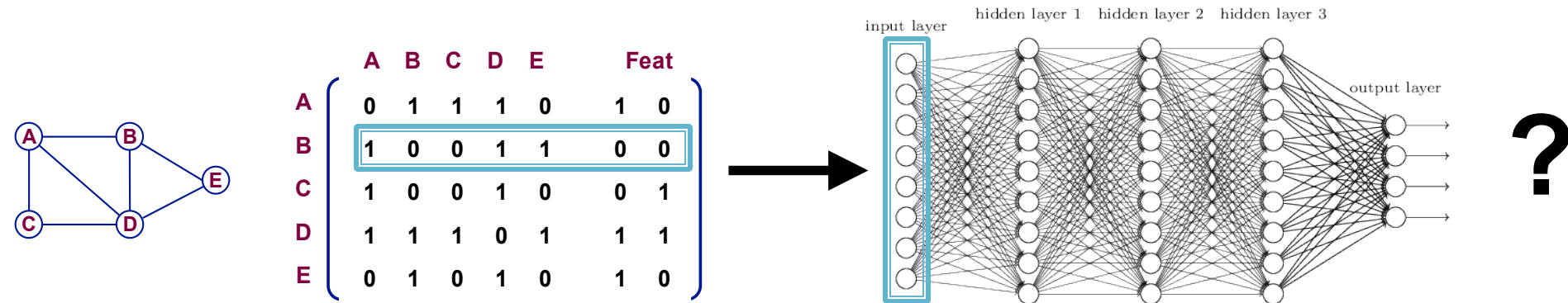
- **Local network neighborhoods:**
 - Describe aggregation strategies
 - Define computation graphs
- **Stacking multiple layers:**
 - Describe the model, parameters, training
 - How to fit the model?
 - Simple example for unsupervised and supervised training

Setup

- Assume we have a graph G :
 - V is the **vertex set**
 - A is the **adjacency matrix** (assume binary)
 - $X \in \mathbb{R}^{|V| \times m}$ 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: $[1, 1, \dots, 1]$

A Naïve Approach

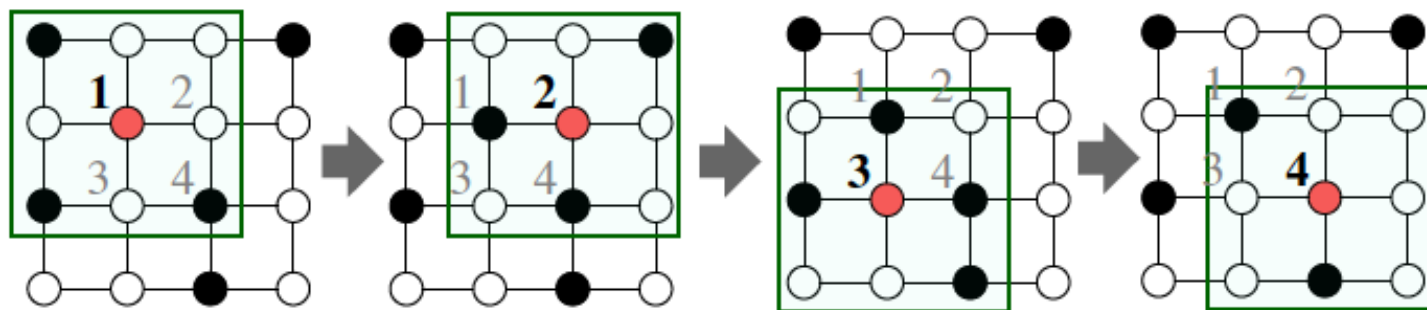
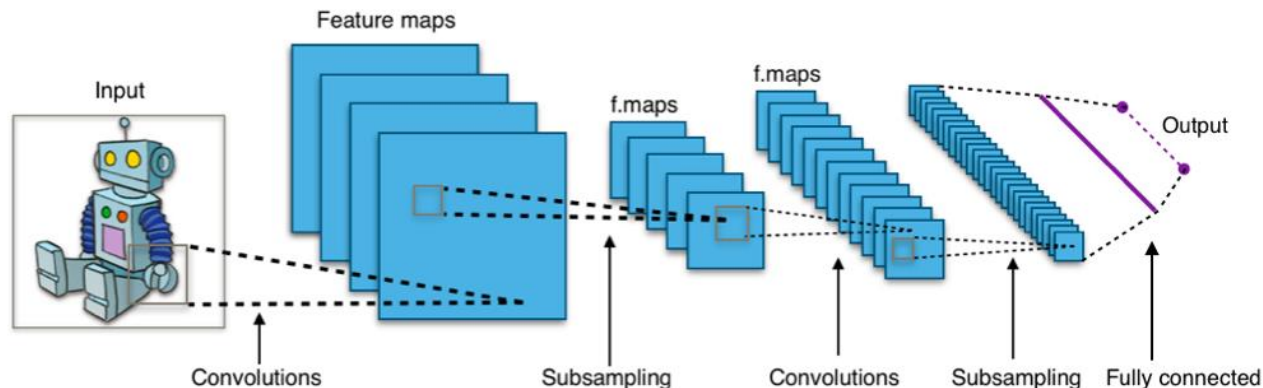
- Join adjacency matrix and features
- Feed them into a deep neural net:



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

Idea: Convolutional Networks

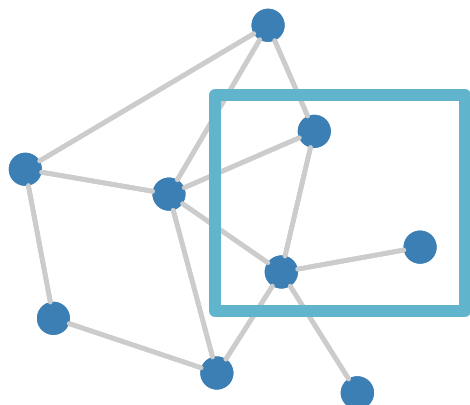
CNN on an image:



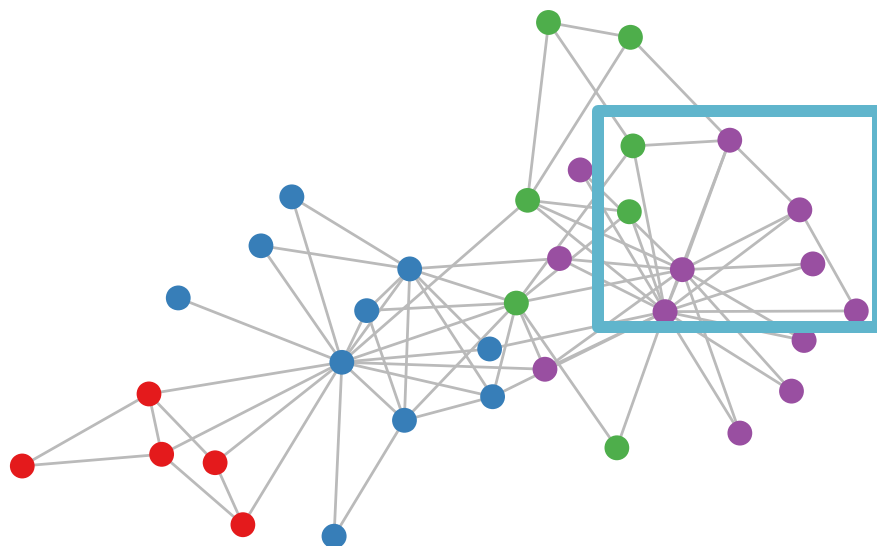
Goal is to generalize convolutions beyond simple lattices
Leverage node features/attributes (e.g., text, images)

Real-World Graphs

But our graphs look like this:



or this:



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

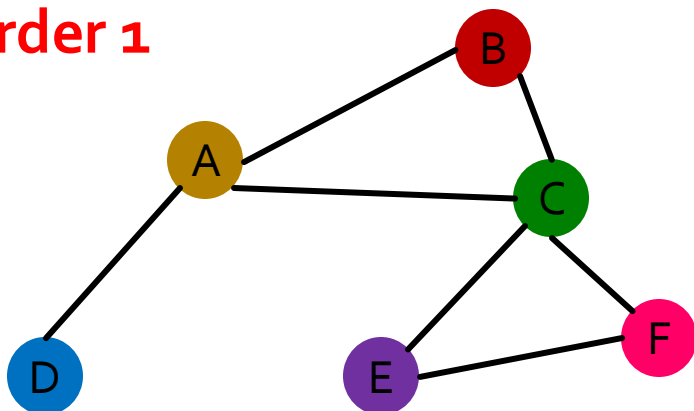
Permutation Invariance

- **Imagine we want to embed an entire graph**
- **Observation: A graph does not have a canonical ordering of its nodes**
 - We can have many different node orderings of the same graph.
- **What do we want:** If we learn an embedding function over a graph, we should get the same result (the same embedding) regardless of how the nodes are numbered.

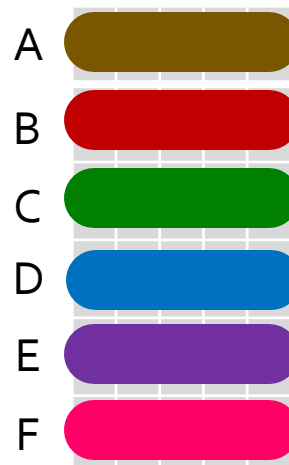
Permutation Invariance

- Graph does not have a canonical ordering of the nodes!

Order 1



Node features X_1



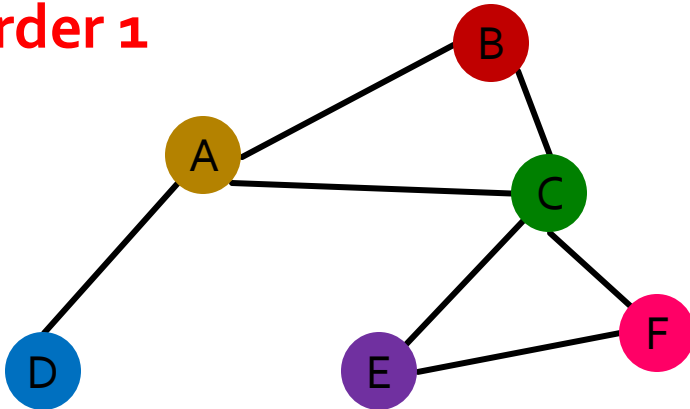
Adjacency matrix A_1

	A	B	C	D	E	F
A						
B						
C						
D						
E						
F						

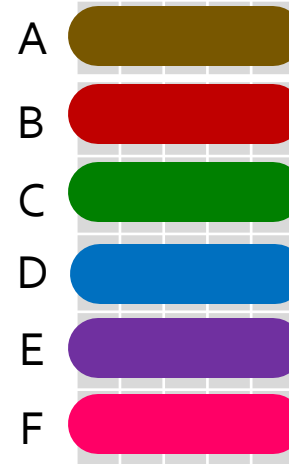
Permutation Invariance

- Graph does not have a canonical ordering of the nodes!

Order 1



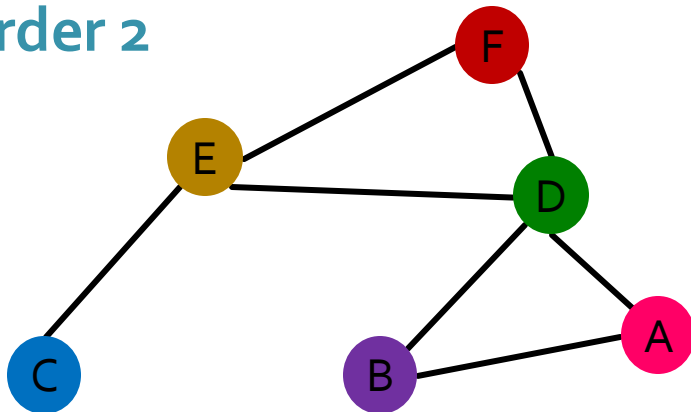
Node features X_1



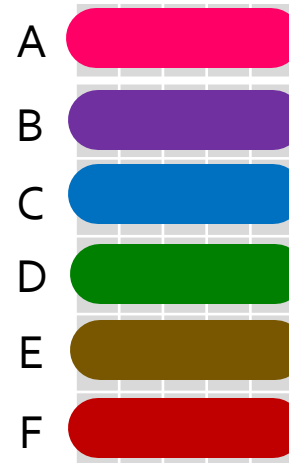
Adjacency matrix A_1

	A	B	C	D	E	F
A						
B						
C						
D						
E						
F						

Order 2



Node features X_2



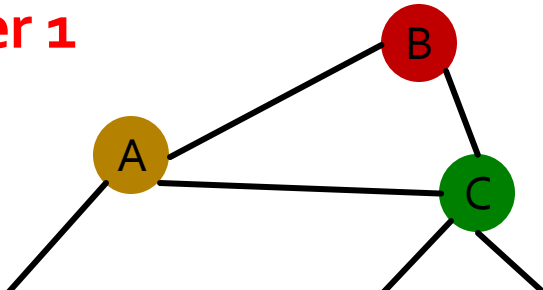
Adjacency matrix A_2

	A	B	C	D	E	F
A						
B						
C						
D						
E						
F						

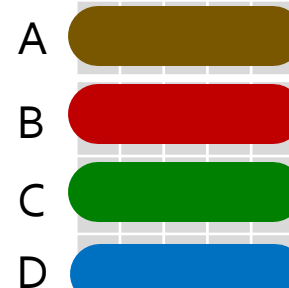
Permutation Invariance

- Graph does not have a canonical ordering of the nodes!

Order 1



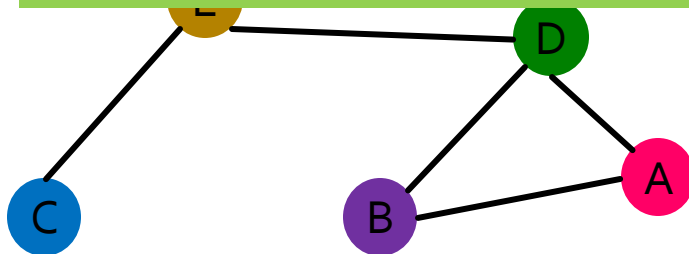
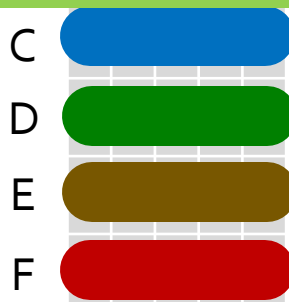
Node features X_1



Adjacency matrix A_1

	A	B	C	D	E	F
A						
B						
C						
D						

Learned graph representation
should be the same for **Order 1**
and **Order 2**



	A	B	C	D	E	F
A						
B						
C						
D						
E						
F						

Permutation Invariance

What do we mean by “graph representation is the same for two orderings”?

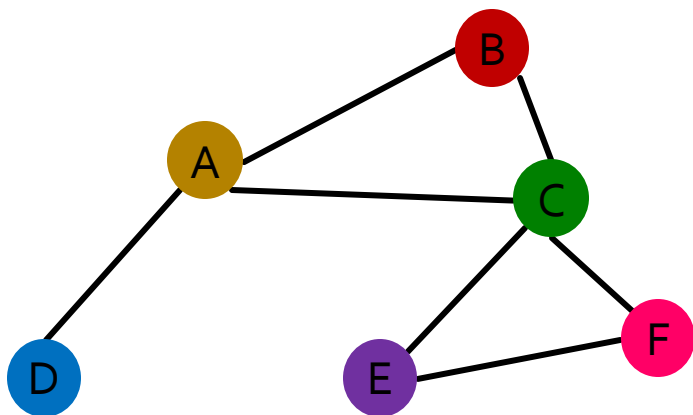
In other words, f maps a graph to a d -dim embedding

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

$$f(A_1, X_1) = f(A_2, X_2)$$

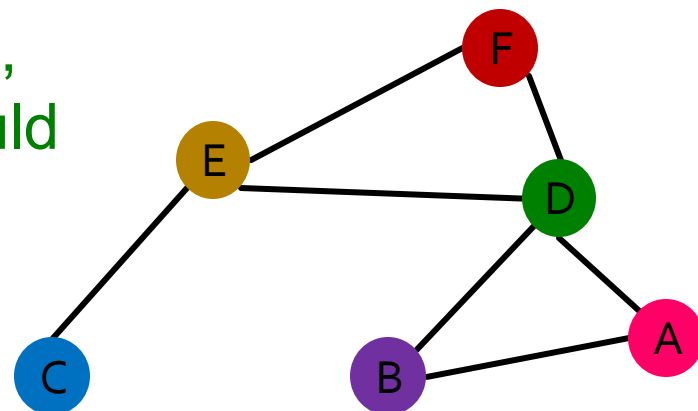
A is the adjacency matrix
 X is the node feature matrix

Order 1: A_1, X_1



For two orders,
output of f should
be the same!

Order 2: A_2, X_2



Permutation Invariance

What does it mean by “graph representation is the same for two order plans”?

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

A is the adjacency matrix
 X is the node feature matrix

- Then, if $f(A_i, X_i) = f(A_j, X_j)$ for any ordering i and j , we formally say f is a **permutation invariant function**.

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

m ... each node has a m -dim feature vector associated with it.

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

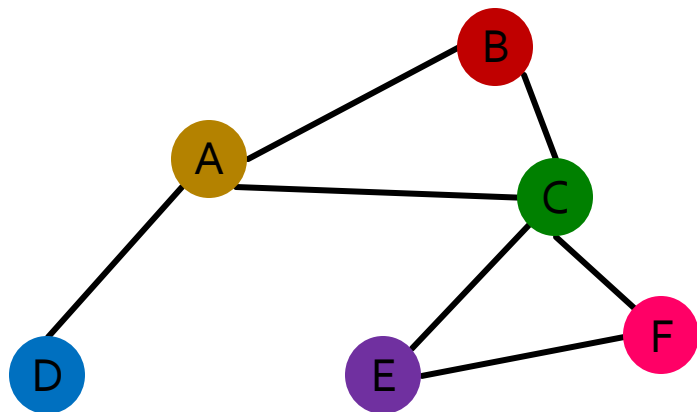
d ... output embedding dimensionality of embedding the graph $G = (A, X)$

Permutation P : a shuffle of the node order
Example: $(A, B, C) \rightarrow (B, C, A)$

Permutation Equivariance

For node representation: We learn a function f that maps nodes of G to a matrix $\mathbb{R}^{|V| \times d}$. In other words, each node in V is mapped to a d -dim embedding.

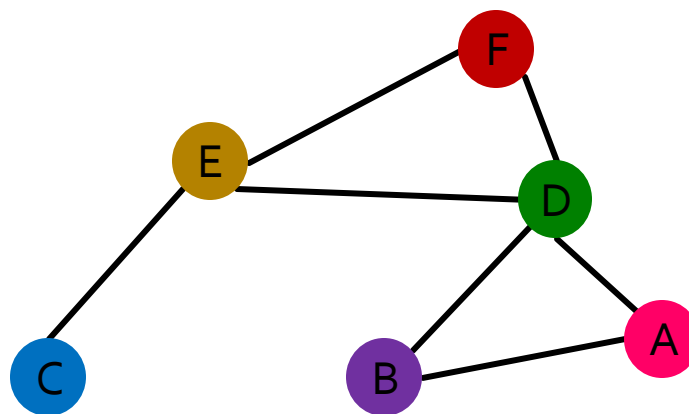
Order 1: A_1, X_1



A		
B		
C		
D		
E		
F		

$$f(A_1, X_1) =$$

Order 2: A_2, X_2



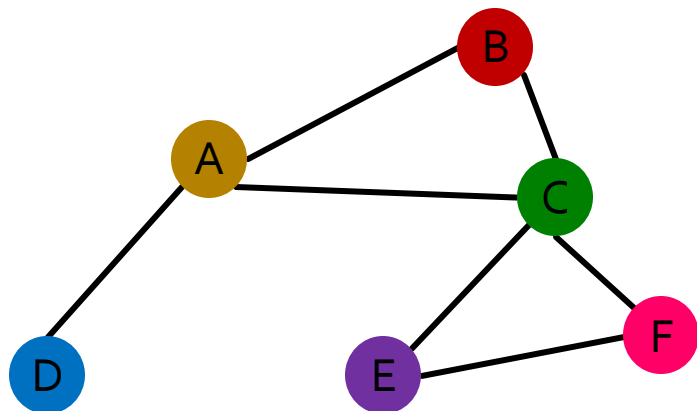
A		
B		
C		
D		
E		
F		

$$f(A_2, X_2) =$$

Permutation Equivariance

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

Order 1: A_1, X_1

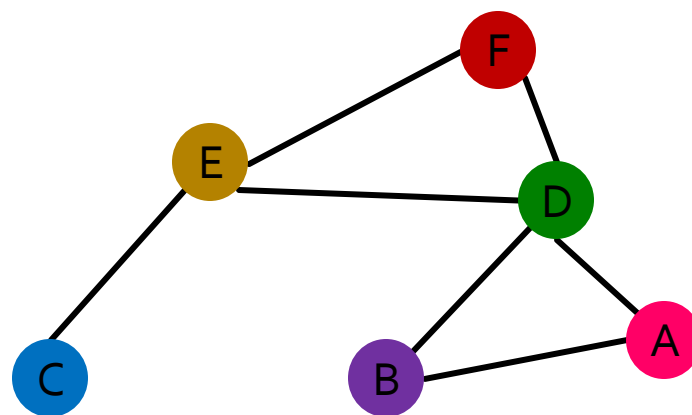


Representation vector
of the brown node A

A		
B		
C		
D		
E		
F		

$$f(A_1, X_1) =$$

Order 2: A_2, X_2



$$f(A_2, X_2) =$$

A		
B		
C		
D		
E		
F		

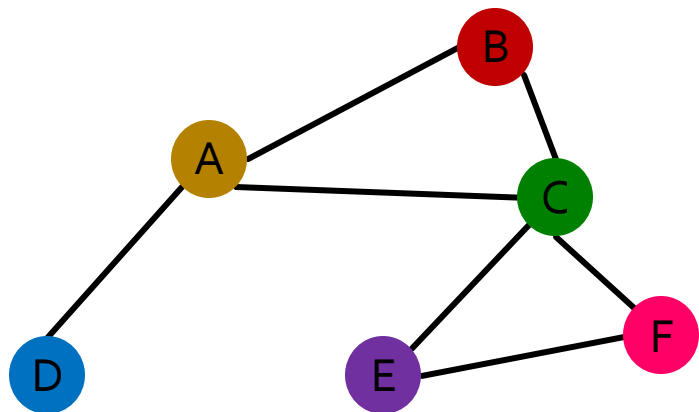
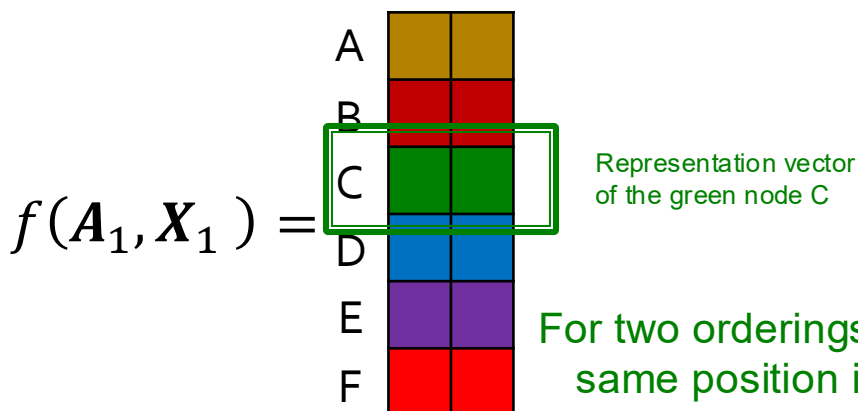
Representation vector
of the brown node E

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

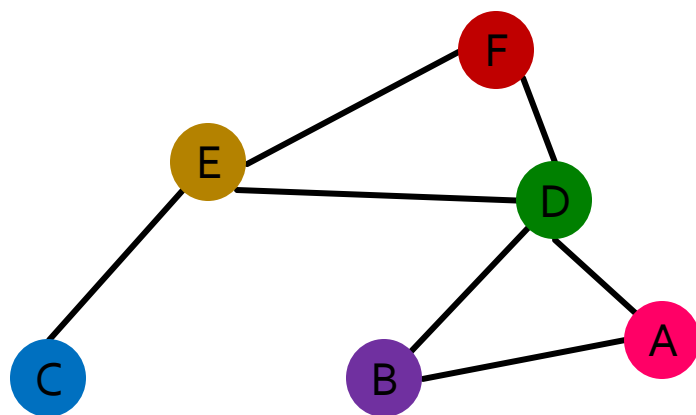
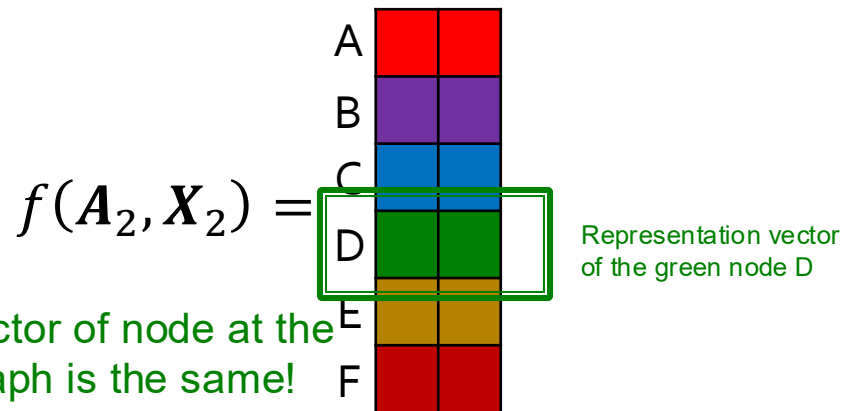
Permutation Equivariance

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

Order 1: A_1, X_1



Order 2: A_2, X_2



For two orderings, 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 $G = (A, X)$ to a matrix $\mathbb{R}^{|V| \times d}$
- If the output vector of a node at the same position in the graph remains unchanged for any ordering, we say f is **permutation equivariant**.
 $m \dots$ each node has a m -dim feature vector associated with it.
- **Definition:** For any **node** function $f: \mathbb{R}^{|V| \times |V|} \times \mathbb{R}^{|V| \times m} \rightarrow \mathbb{R}^{|V| \times d}$, f is **permutation-equivariant** if $P f(A, X) = f(P A P^T, P X)$ for any permutation P .
 f maps each node in V to a d -dim embedding.

Summary: Invariance and Equivariance

■ Permutation-invariant

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

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

■ Permutation-equivariant

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

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

■ Examples:

■ $f(A, X) = \mathbf{1}^T X$: Permutation-invariant

■ Reason: $f(PAP^T, PX) = \mathbf{1}^T PX = \mathbf{1}^T X = f(A, X)$

■ $f(A, X) = X$: Permutation-equivariant

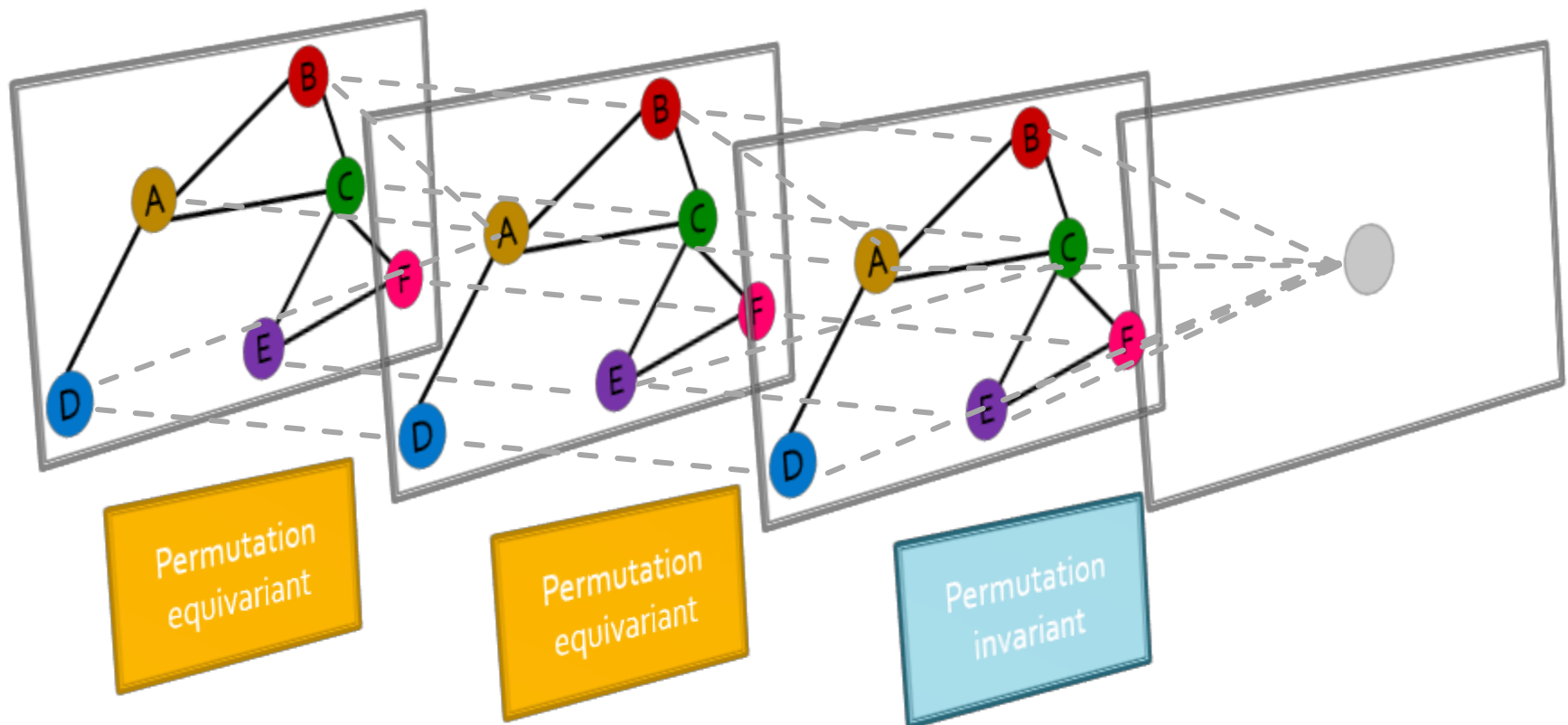
■ Reason: $f(PAP^T, PX) = PX = Pf(A, X)$

■ $f(A, X) = AX$: Permutation-equivariant

■ Reason: $f(PAP^T, PX) = PAP^T PX = PAX = Pf(A, X)$

Graph Neural Network Overview

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

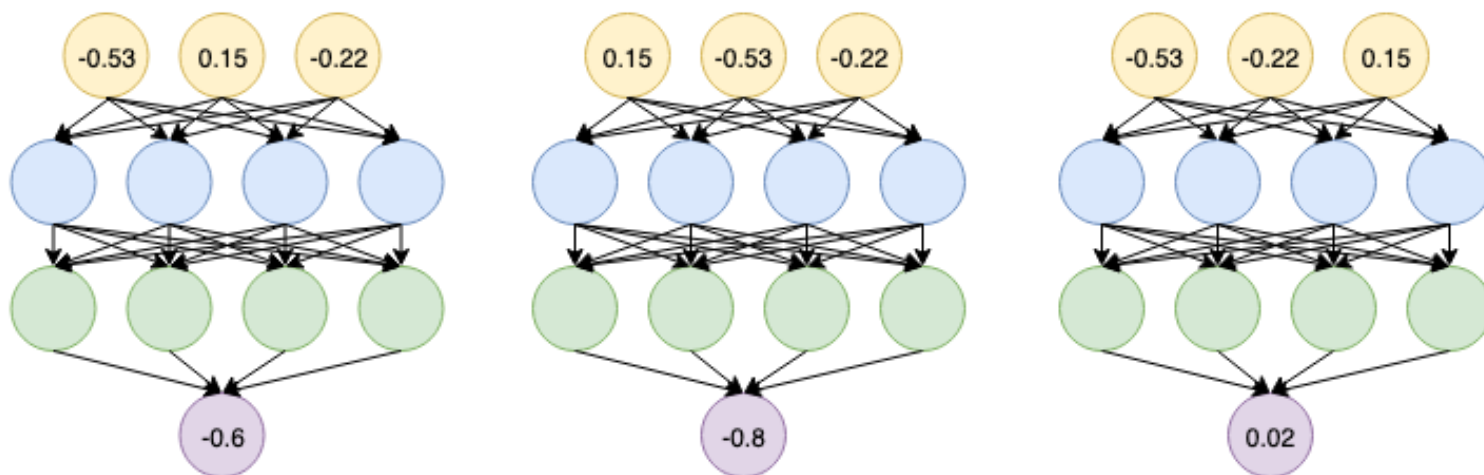


Graph Neural Network Overview

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

■ **No.**

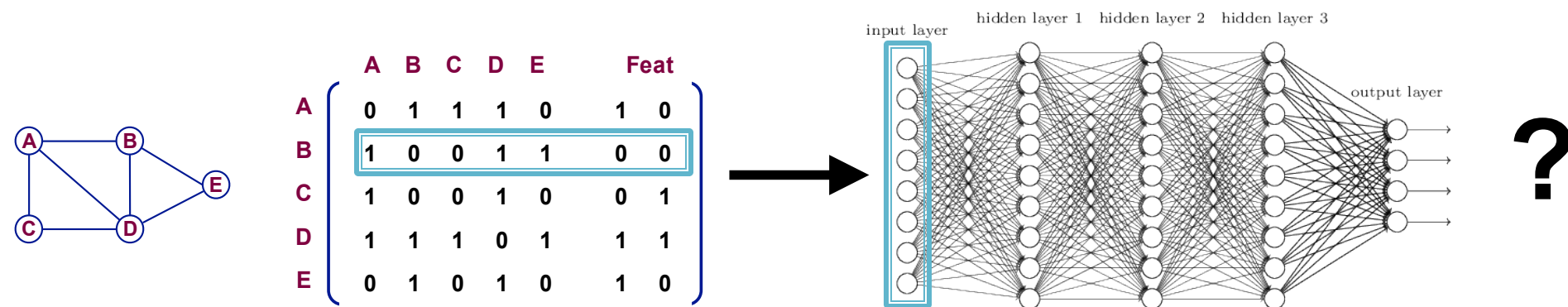
Switching the order of the input leads to different outputs!



Graph Neural Network Overview

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

■ **No.**



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

Graph Neural Network Overview

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

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

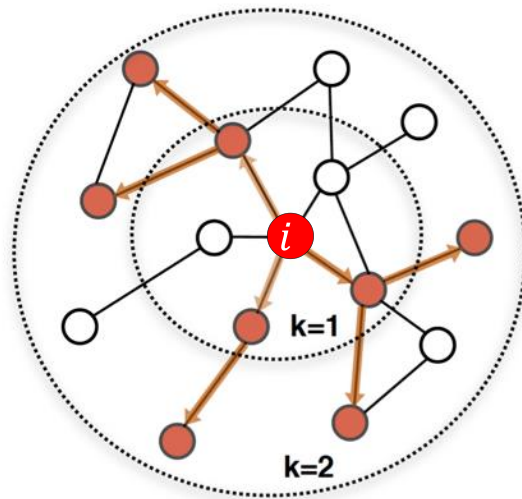
?

Outline of Today's Lecture

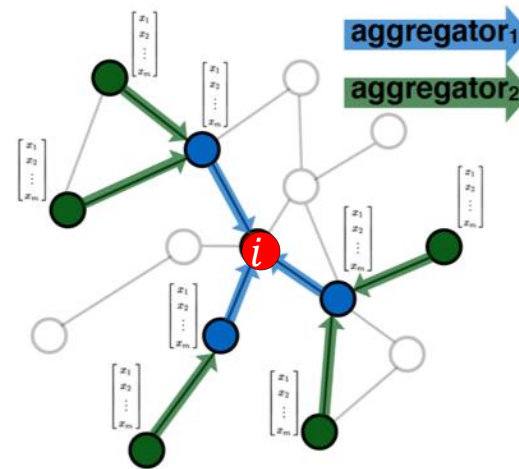
1. Basics of deep learning ✓
2. Deep learning for graphs ✓
3. Graph Convolutional Networks 
4. GNNs subsume CNNs

Graph Convolutional Networks

Idea: Node's neighborhood defines a computation graph



Determine node
computation graph

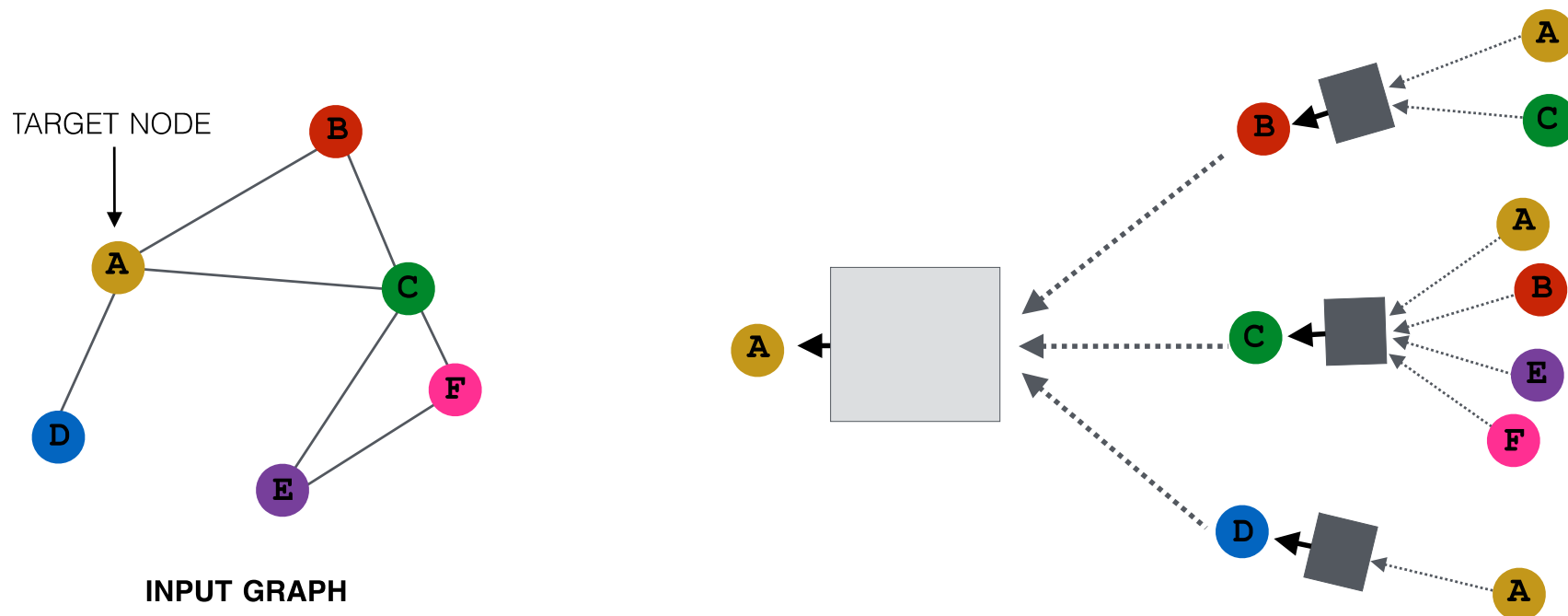


Propagate and
transform information

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

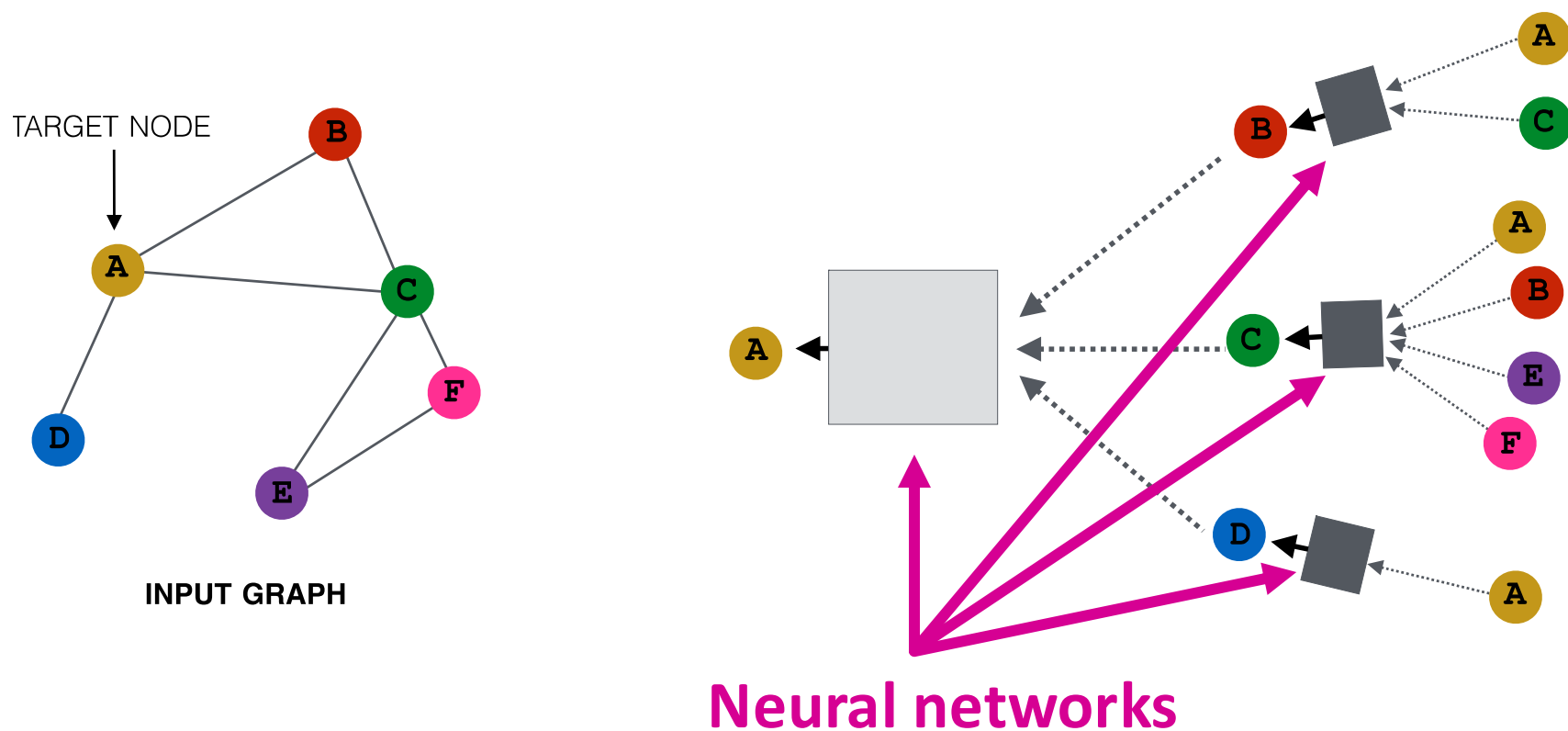
Idea: Aggregate Neighbors

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



Idea: Aggregate Neighbors

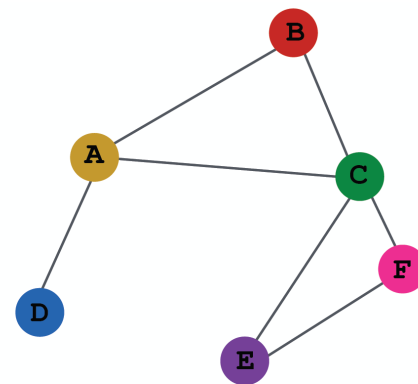
- **Intuition:** Nodes aggregate information from their neighbors using neural networks



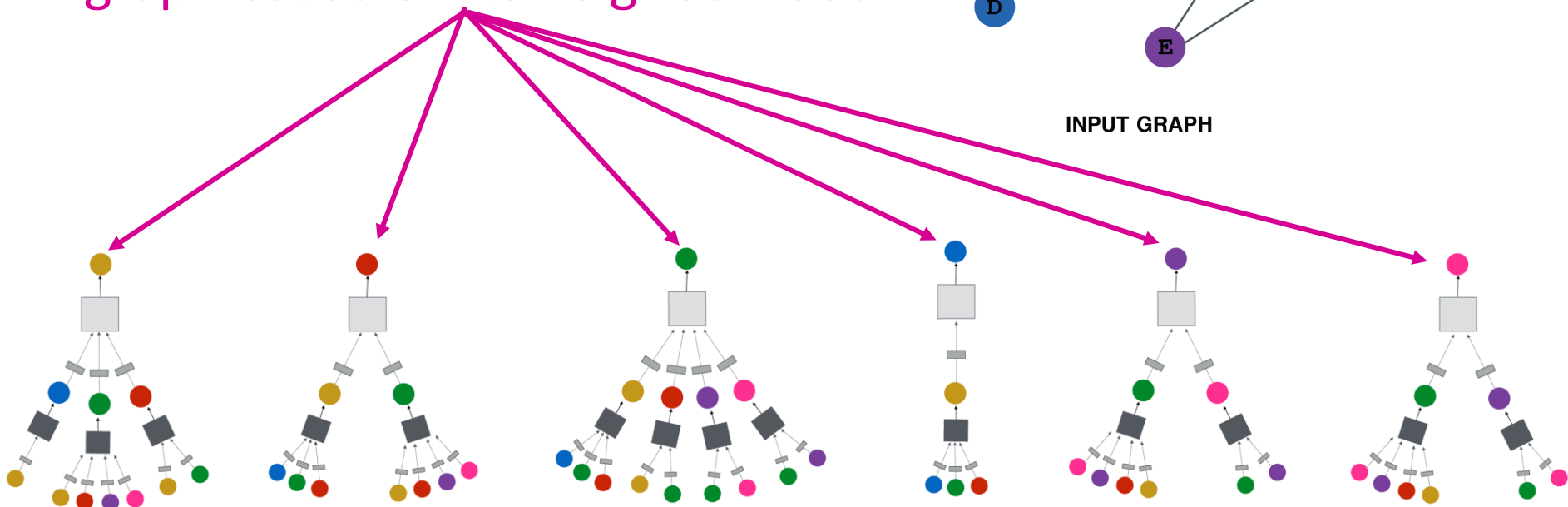
Idea: Aggregate Neighbors

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

Every node defines a computation graph based on its neighborhood!

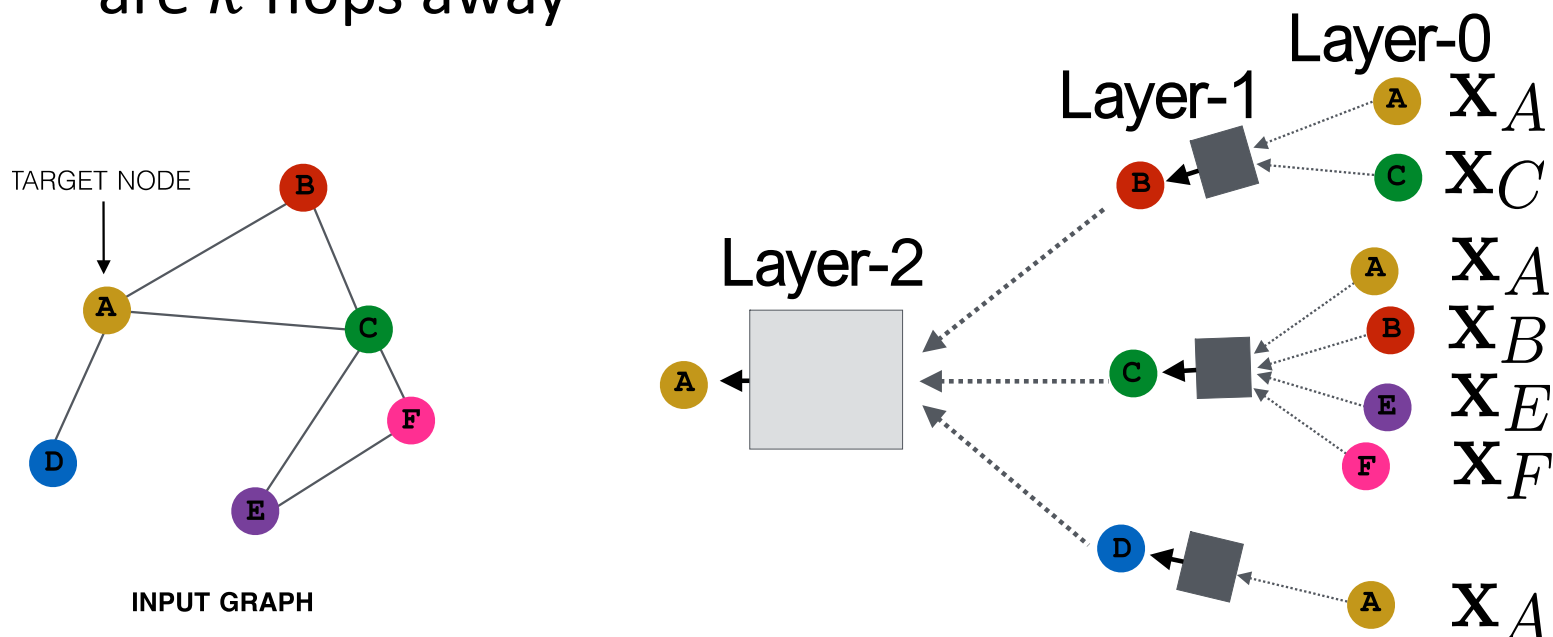


INPUT GRAPH



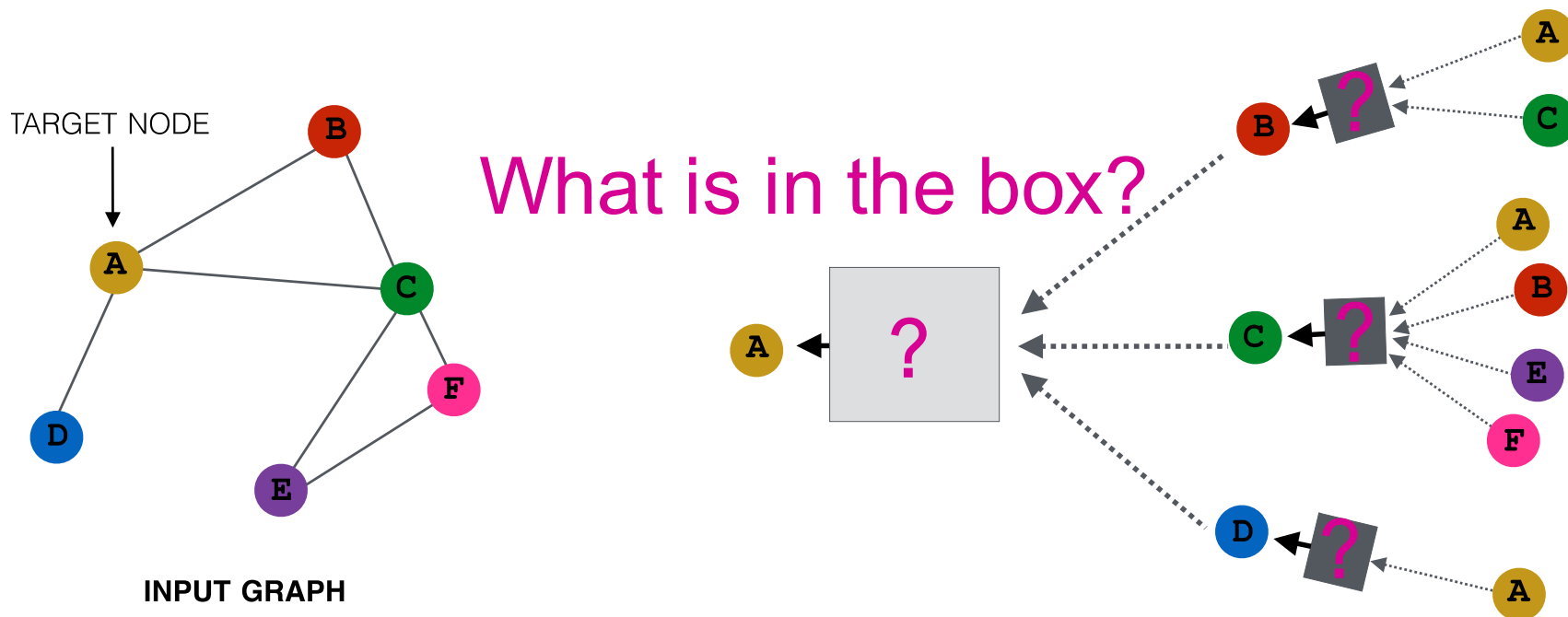
Deep Model: Many Layers

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



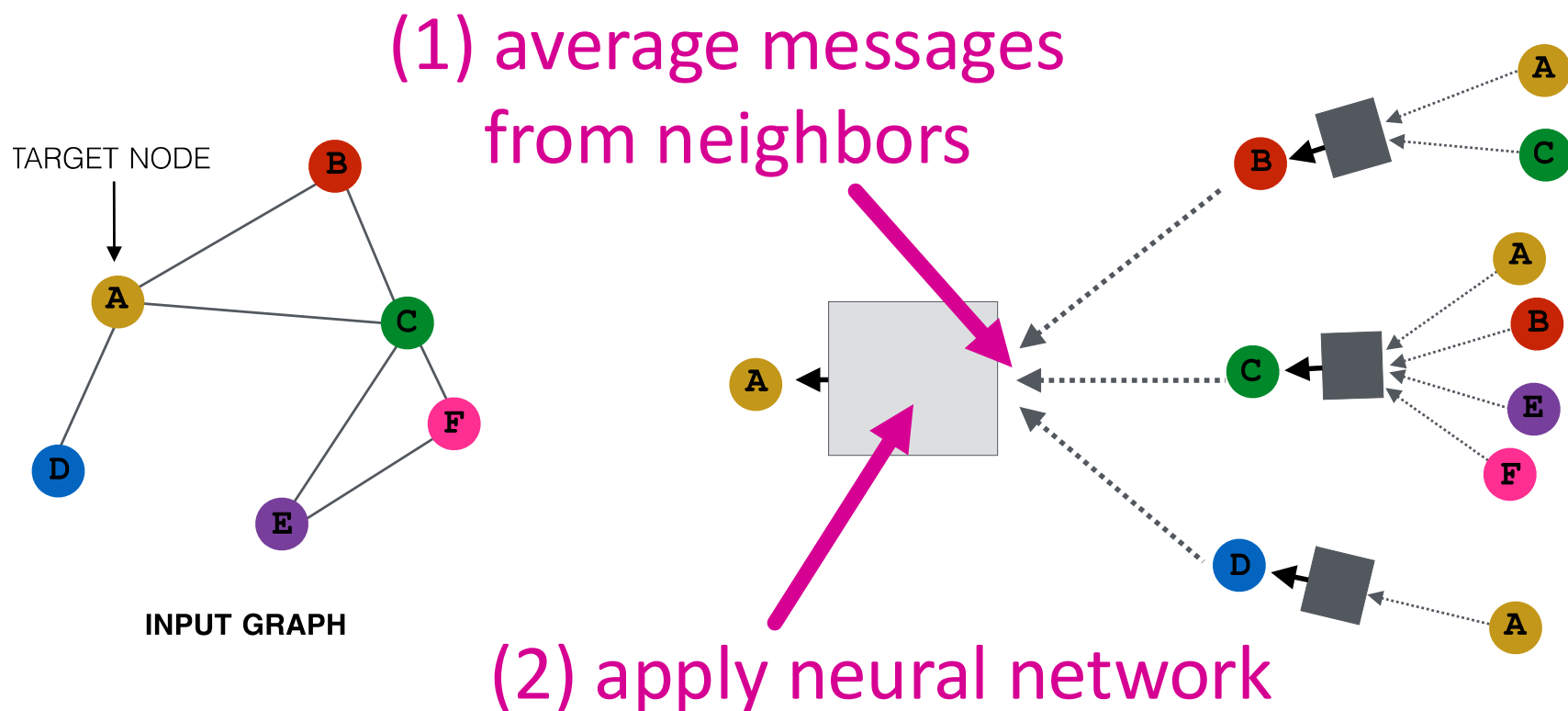
Neighborhood Aggregation

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



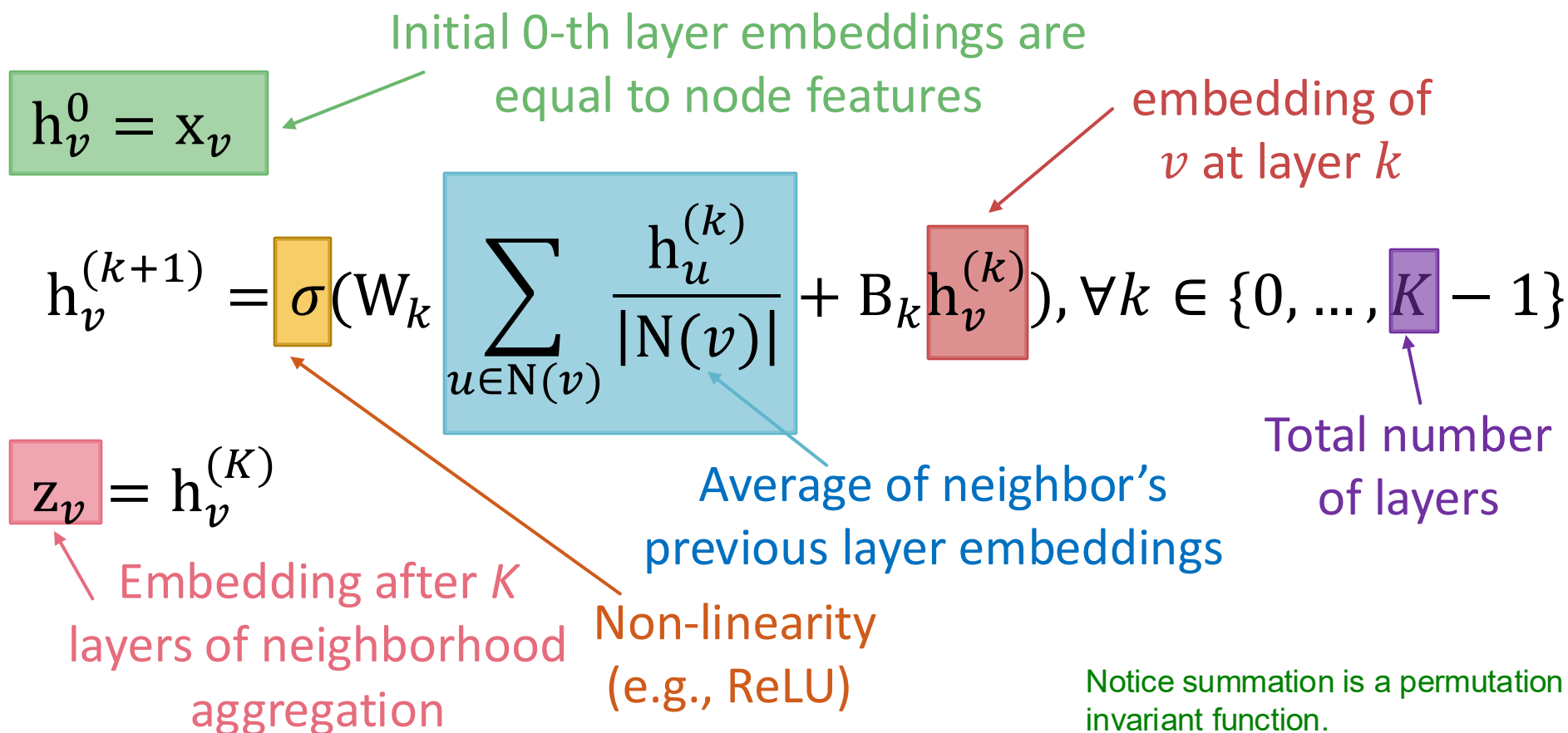
Neighborhood Aggregation

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



The Math: Deep Encoder of a GCN

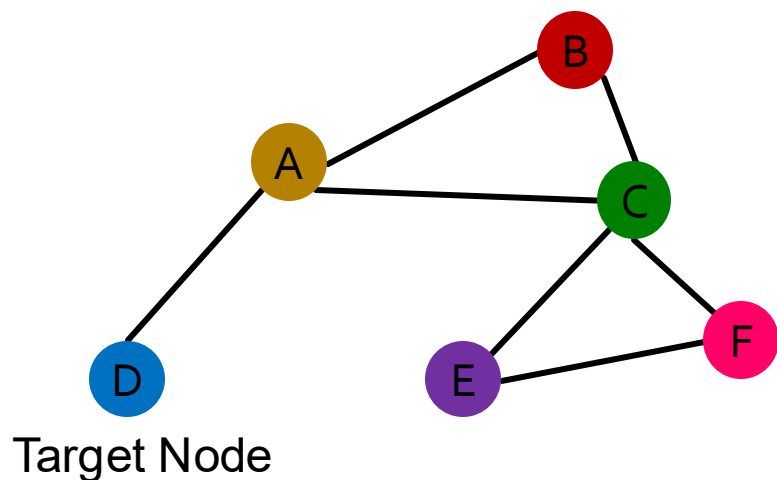
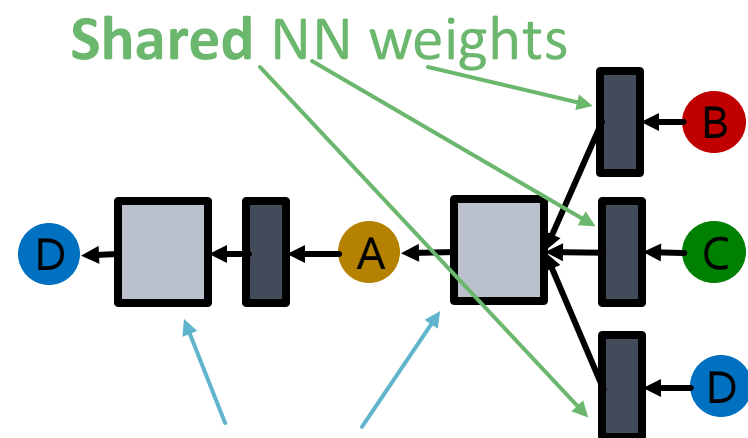
- **Basic approach:** Average neighbor messages and apply a neural network



GCN: Invariance and Equivariance

What are the **invariance** and **equivariance** properties for a GCN?

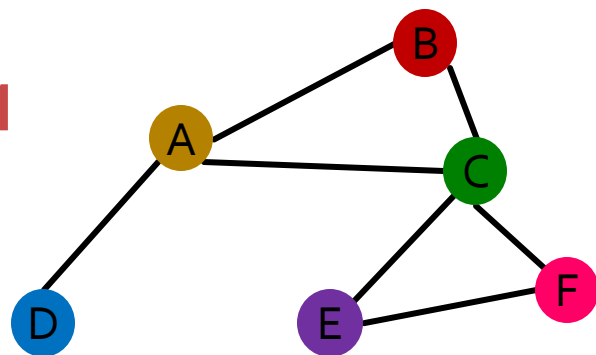
- **Given a node**, the GCN that computes its embedding is **permutation invariant**



GCN: Invariance and Equivariance

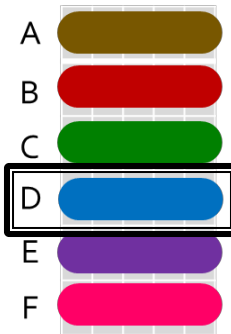
- Considering all nodes in a graph, GCN computation is **permutation equivariant**

Order 1

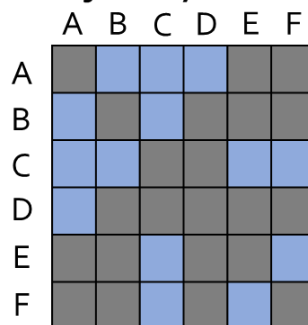


Target Node

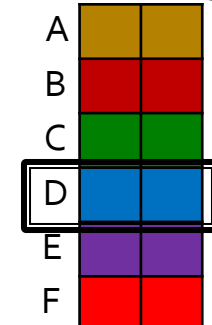
Node feature X_1



Adjacency matrix A_1

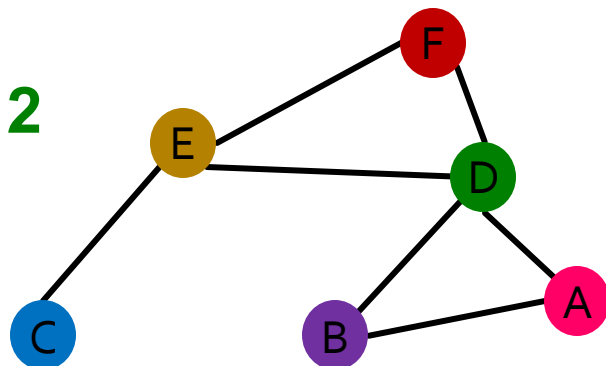


Embeddings H_1



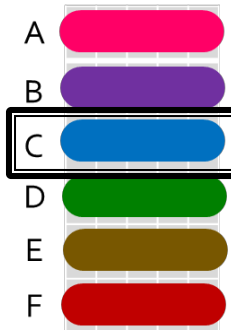
Permute the input, the output also permutes accordingly - **permutation equivariant**

Order 2

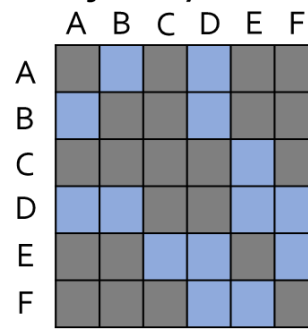


Target Node

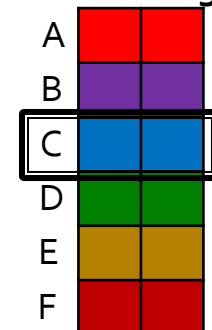
Node feature X_2



Adjacency matrix A_2



Embeddings H_2

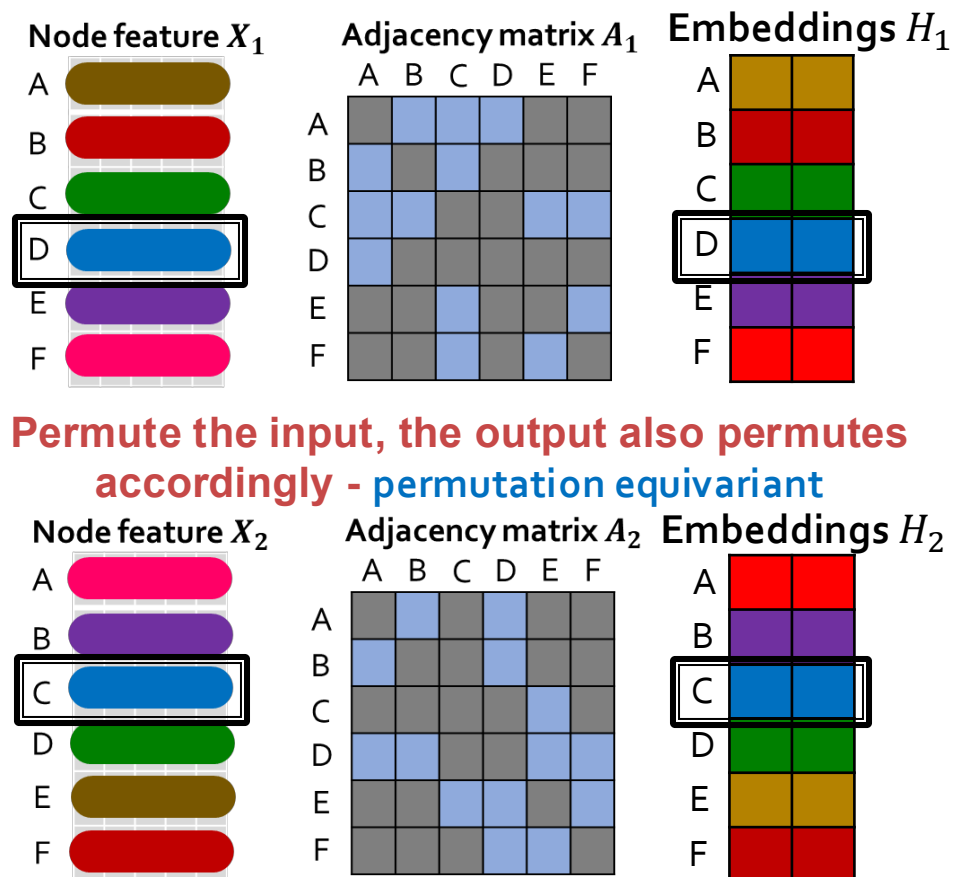


GCN: Invariance and Equivariance

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

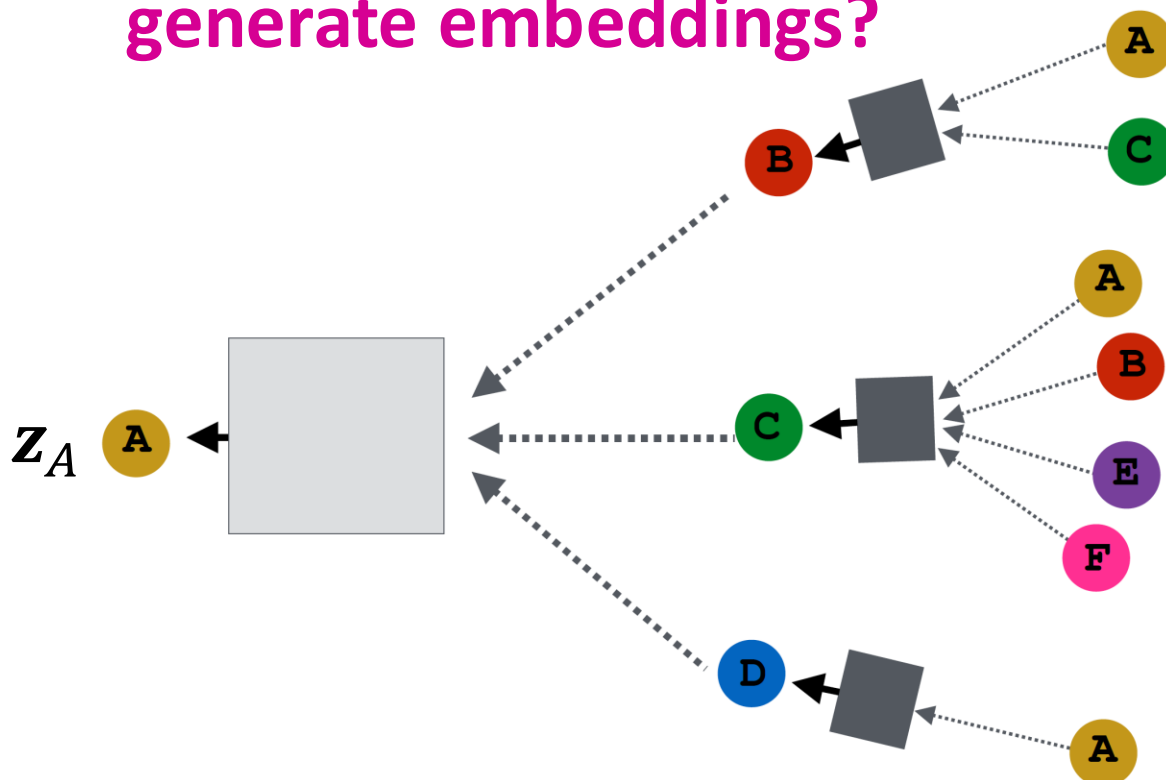
Detailed reasoning:

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



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)

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

Final node embedding

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

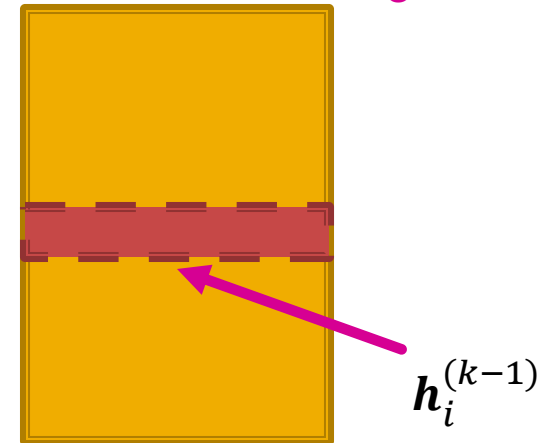
- h_v^k : the hidden representation of node v at layer k
- W_k : weight matrix for neighborhood aggregation
 - B_k : weight matrix for transforming hidden vector of self

Matrix Formulation (1)

- Many aggregations can be performed efficiently by (sparse) matrix operations

- Let $H^{(k)} = [h_1^{(k)} \dots h_{|V|}^{(k)}]^T$
- Then: $\sum_{u \in N_v} h_u^{(k)} = A_{v,:} H^{(k)}$
- Let D be diagonal matrix where $D_{v,v} = \text{Deg}(v) = |N(v)|$
 - The inverse of D : D^{-1} is also diagonal:
 $D_{v,v}^{-1} = 1/|N(v)|$
- Therefore,

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



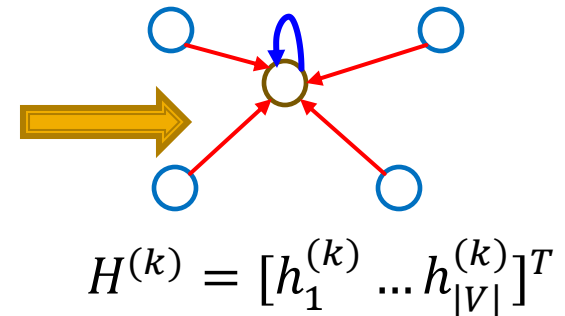
$$\sum_{u \in N(v)} \frac{h_u^{(k-1)}}{|N(v)|} \Rightarrow H^{(k+1)} = D^{-1} A H^{(k)}$$

Matrix Formulation (2)

- Re-writing update function in matrix form:

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

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



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

How to Train A GNN

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

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

- $\mathbf{y}$: node label
- $\mathcal{L}$ could be L2 if $\mathbf{y}$ is real number, or cross entropy if $\mathbf{y}$ is categorical
- **Unsupervised setting:**
 - No node label available
 - **Use the graph structure as the supervision!**

Unsupervised Training

- **One possible idea:** “Similar” nodes have similar embeddings:

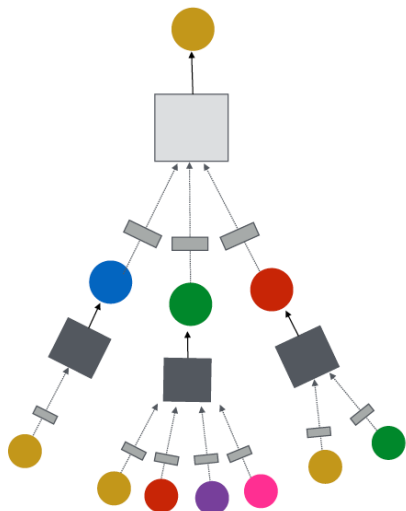
$$\min_{\Theta} \mathcal{L} = \sum_{z_u, z_v} \text{CE}(y_{u,v}, \text{DEC}(z_u, z_v))$$

- where $y_{u,v} = 1$ when node u and v are **similar**
- $z_u = f_{\Theta}(u)$ and $\text{DEC}(\cdot, \cdot)$ is the dot product
- **CE** is the cross entropy loss:
 - $\text{CE}(\mathbf{y}, f(\mathbf{x})) = -\sum_{i=1}^C (y_i \log f_{\Theta}(x)_i)$
 - y_i and $f_{\Theta}(x)_i$ are the **actual** and **predicted** values of the i -th class.
 - **Intuition:** the lower the loss, the closer the prediction is to one-hot
- **Node similarity** can be anything from Lecture 2, e.g., a loss based on:
 - **Random walks** (node2vec, DeepWalk, struc2vec)
 - **Matrix factorization**

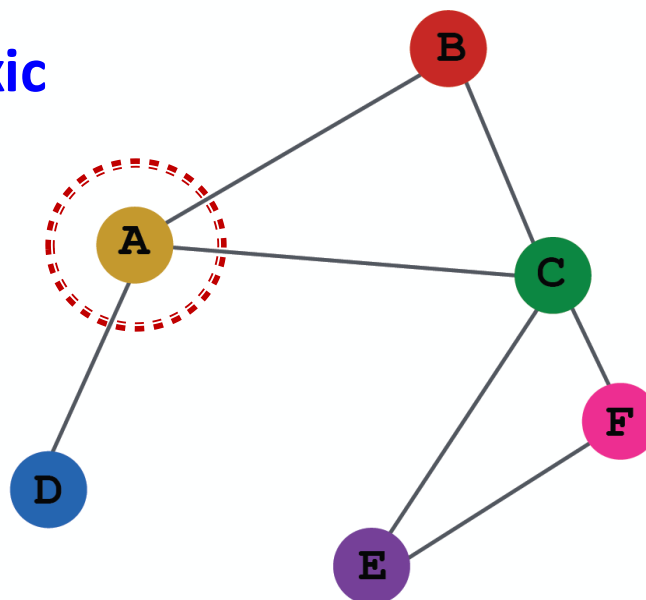
Supervised Training

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

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

Encoder output:
node embedding

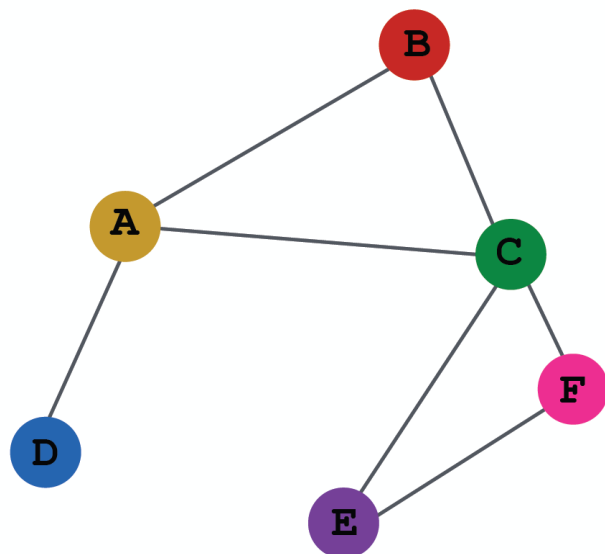
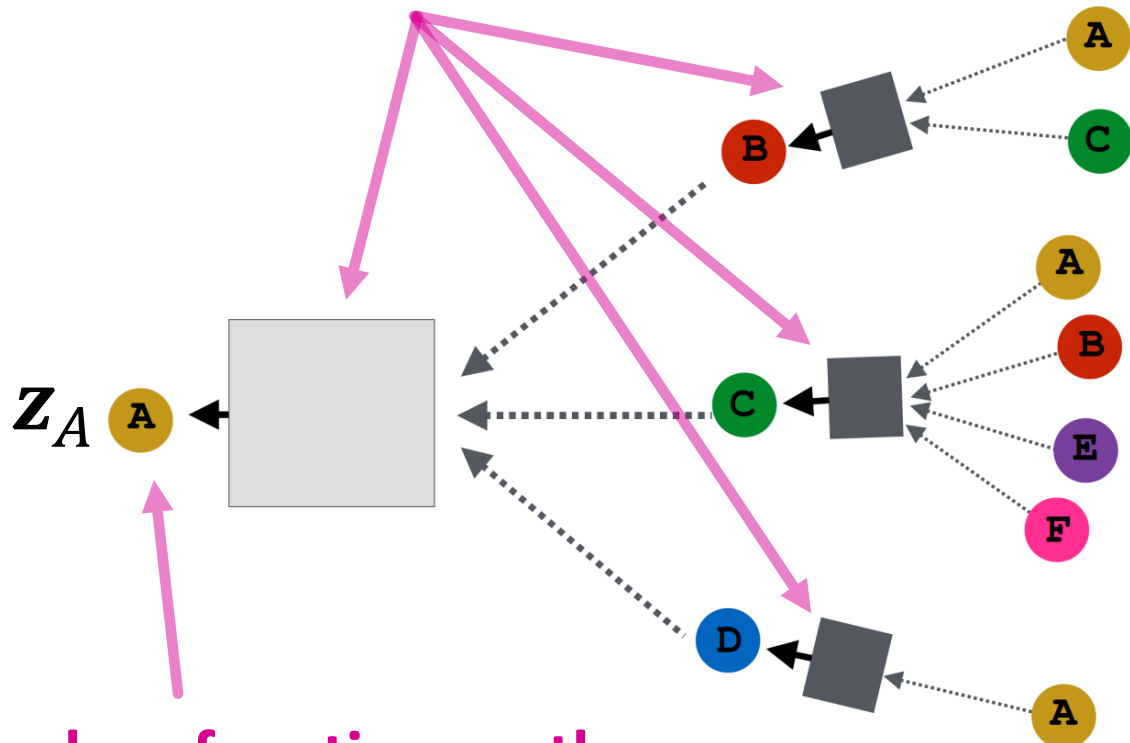
Classification weights

Node class label

Safe or toxic drug?

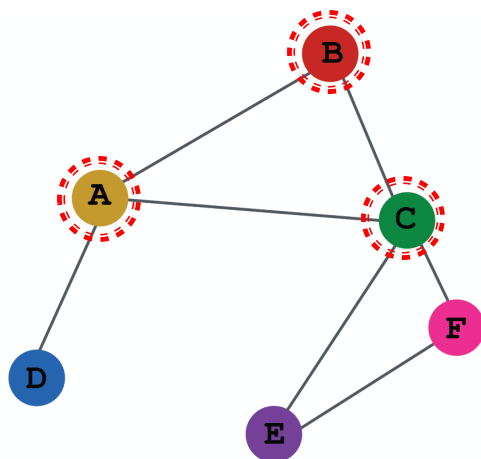
Model Design: Overview

(1) Define a neighborhood aggregation function



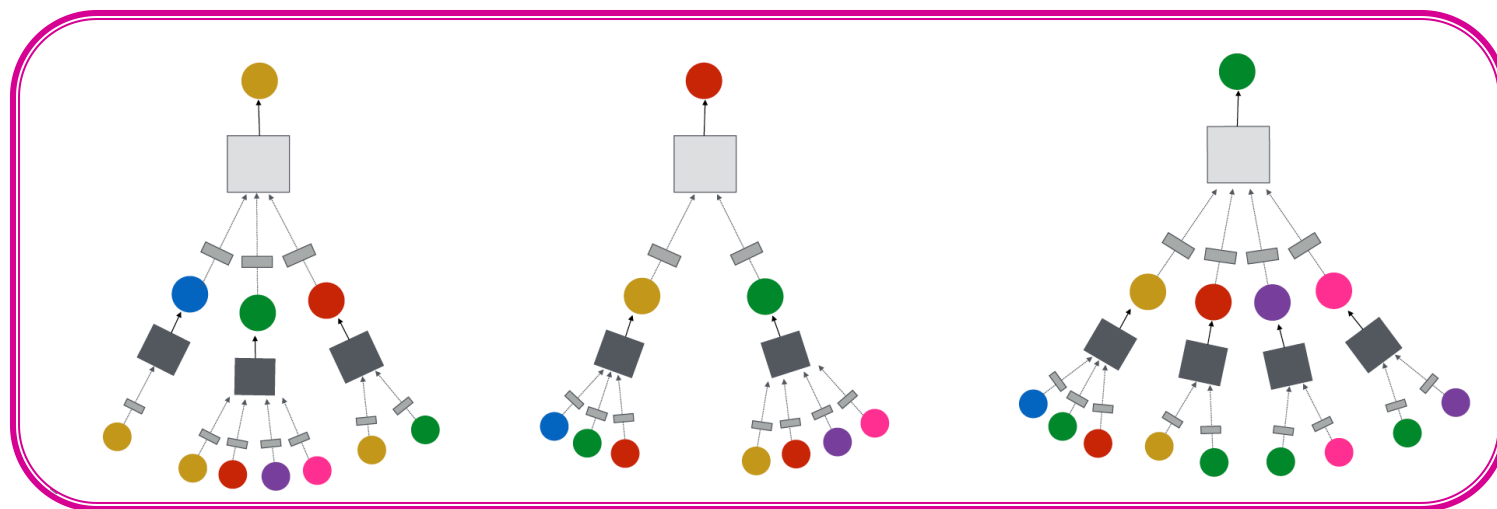
(2) Define a loss function on the embeddings

Model Design: Overview

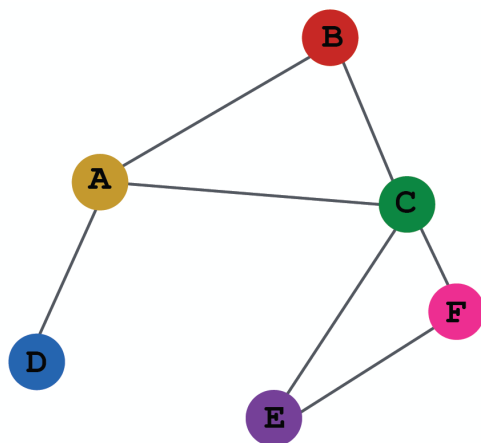


INPUT GRAPH

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



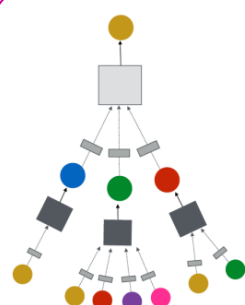
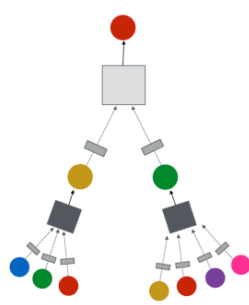
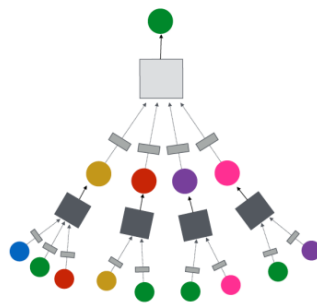
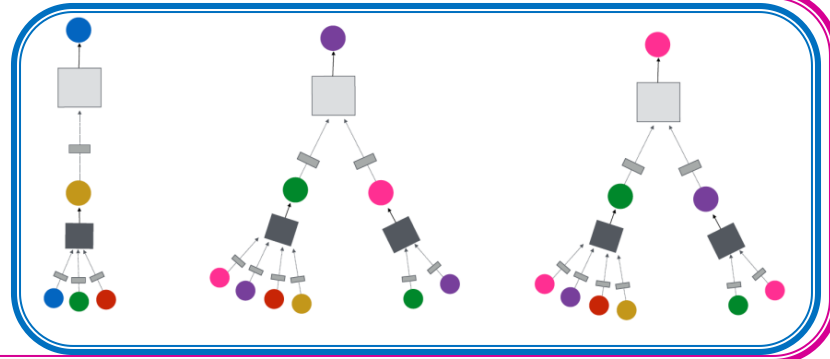
Model Design: Overview



INPUT GRAPH

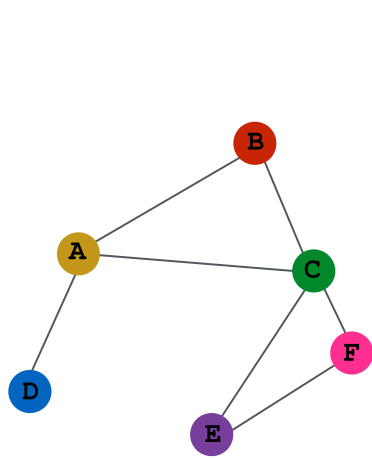
(4) Generate embeddings
for nodes as needed

Even for nodes we never
trained on!

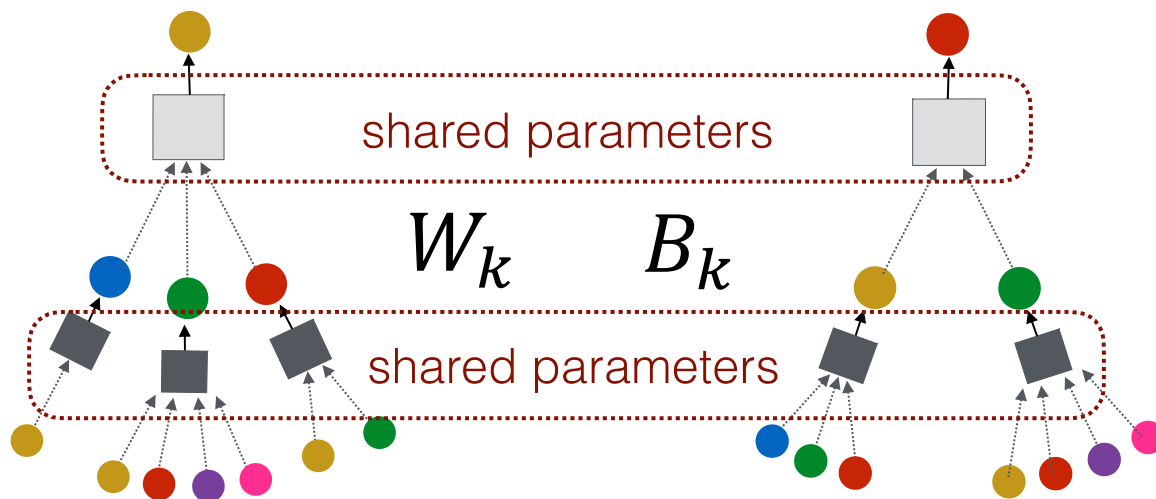


Inductive Capability

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



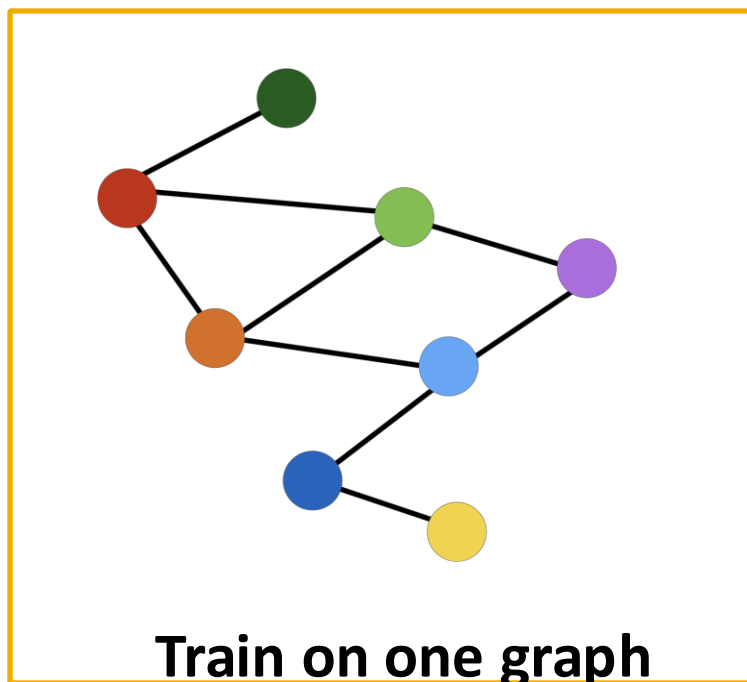
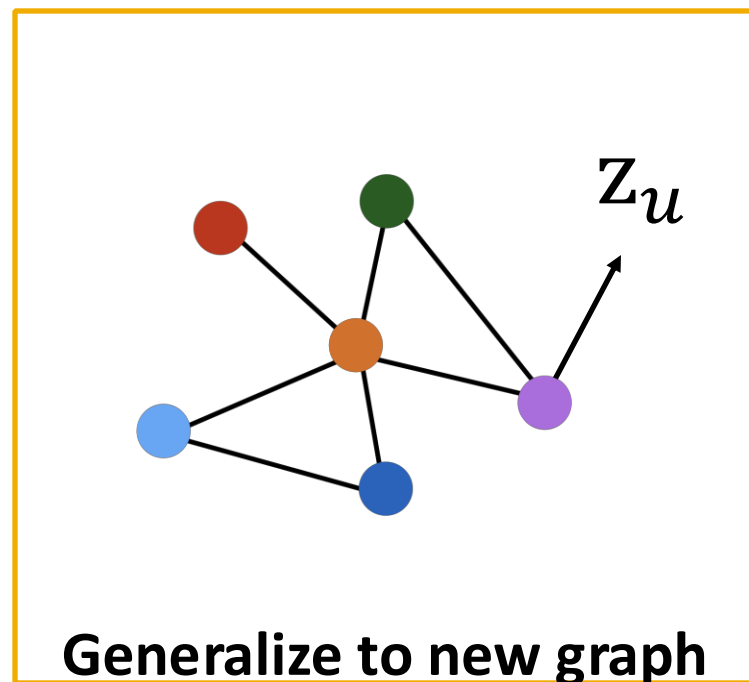
INPUT GRAPH



Compute graph for node A

Compute graph for node B

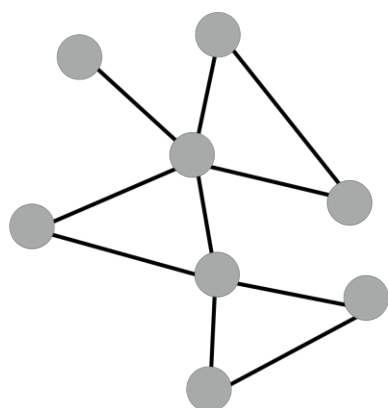
Inductive Capability: New Graphs



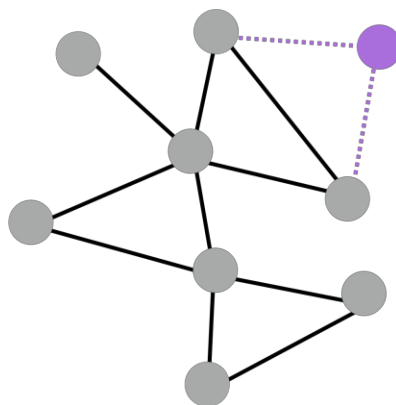
Inductive node embedding → Generalize to entirely unseen graphs

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

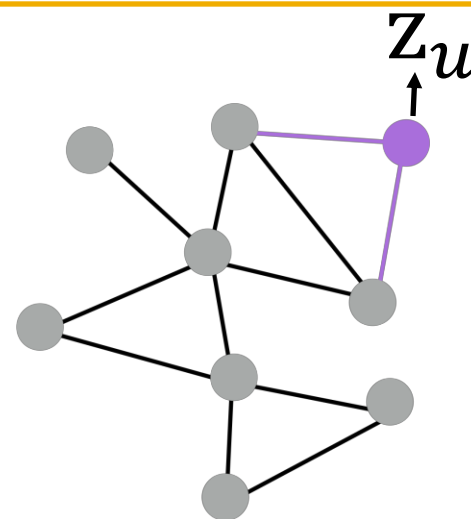
Inductive Capability: New Nodes



Train with snapshot



New node arrives

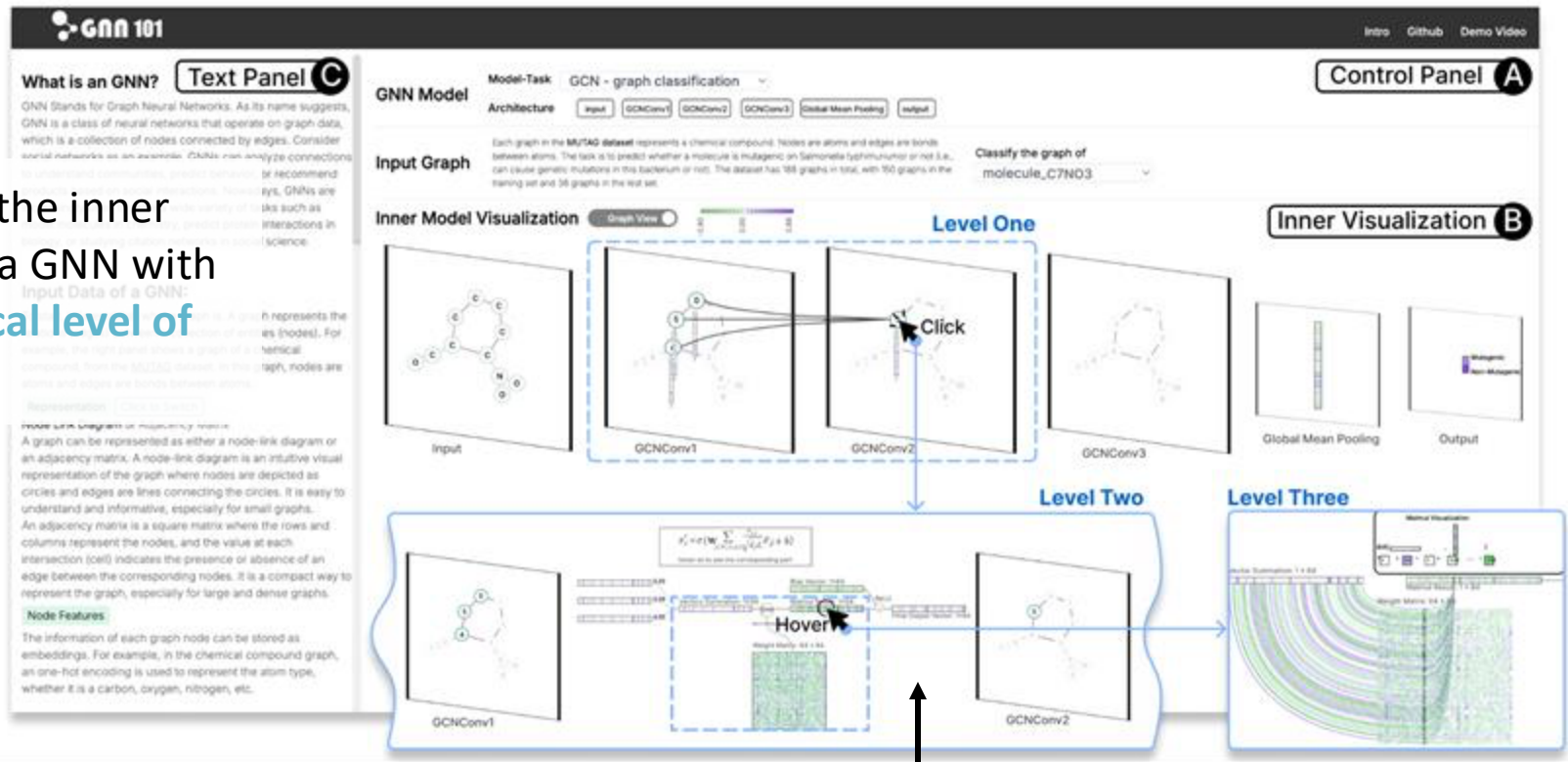


Generate embedding
for new node

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

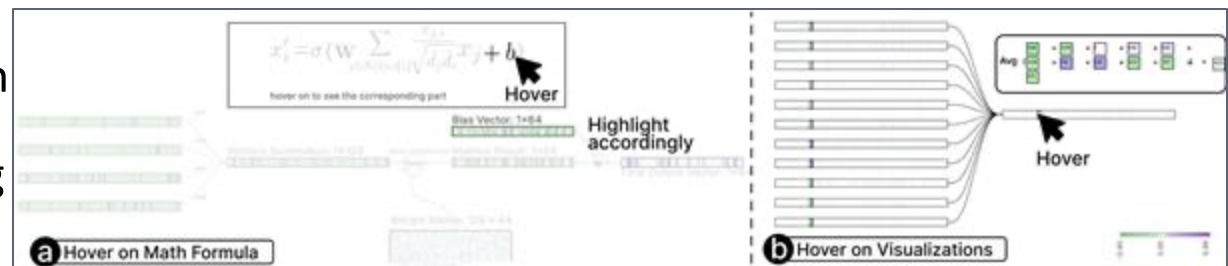
Interactive Visualization of GNNs

Visualizing the inner working of a GNN with a **hierarchical level of detail**



Parameters are shown as heatmaps and curves animations are the computation process.

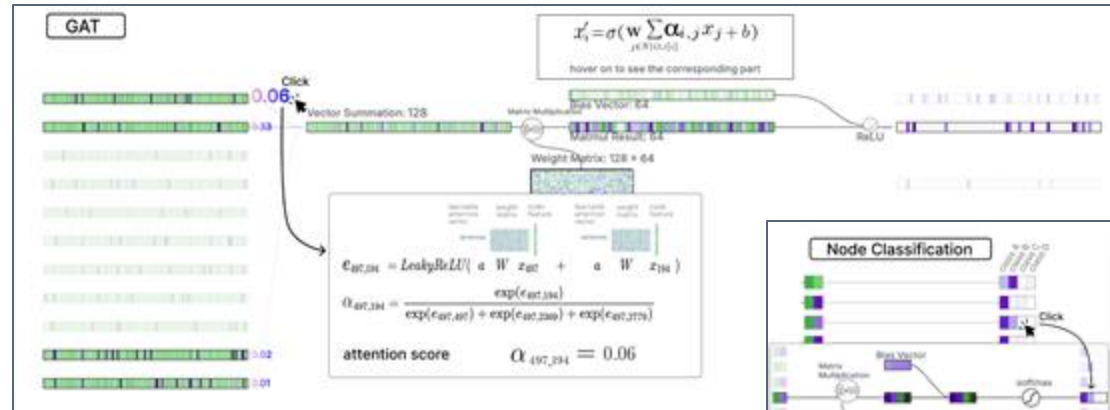
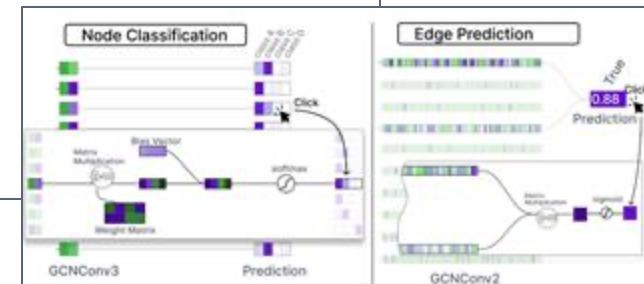
You can hover over terms in the equations, and it will highlight the corresponding visualization.



Interactive Visualization of GNNs

Other Functions

Observe different GNN variants and tasks



GNN Model

Model-Task:

Architecture:

Input Graph

The Twitch dataset is a social network of Twitch users. Nodes are Twitch users and edges are mutual follower relationships between them. The task is to predict whether two users are friends or not. The graph has 77774 edges in total, with 60052 in the training set.

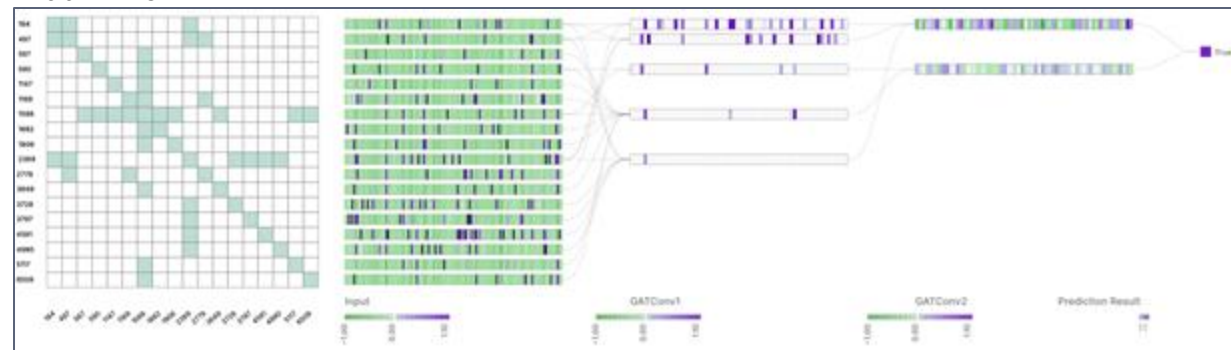
Predict whether there is a link from node to node

Inner Model Visualization

☐ Graph View ☐ Matrix View

Toggle between a node-link subgraph view and an adjacency matrix view.

Matrix View



Outline of Today's Lecture

1. Basics of deep learning ✓
2. Deep learning for graphs ✓
3. Graph Convolutional Networks ✓
4. GNNs subsume CNNs

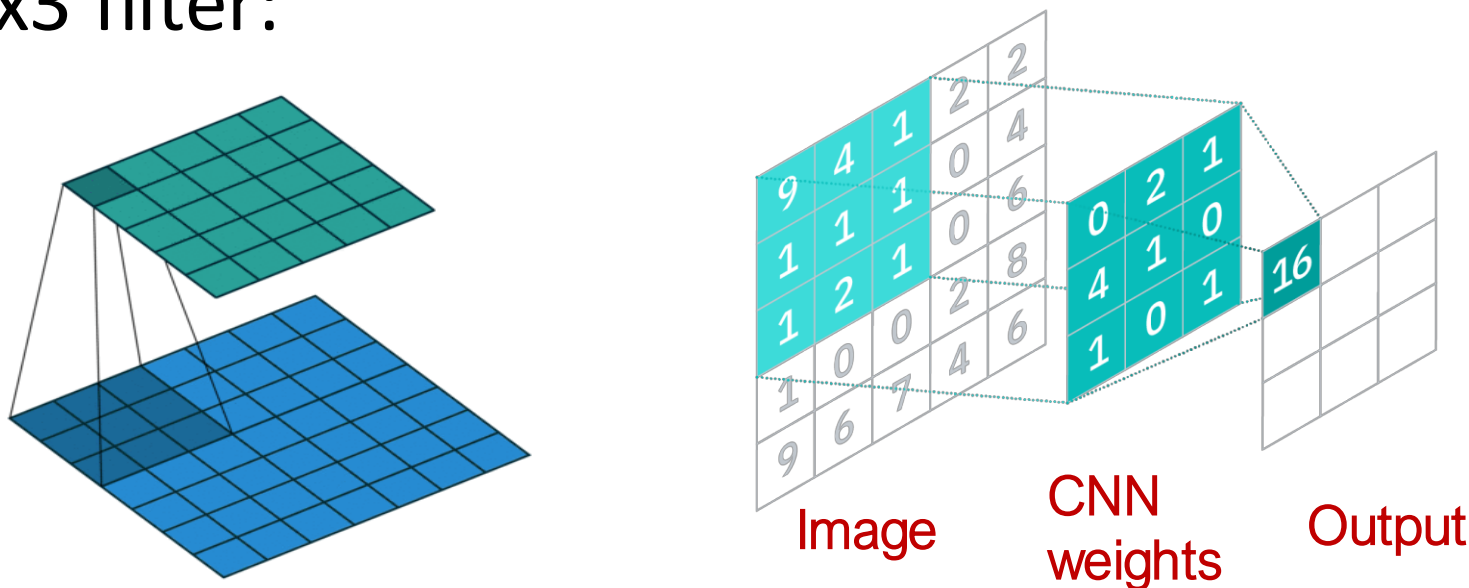


Architecture Comparison

- **How do GNNs compare to prominent architectures such as Convolutional Neural Nets?**

Convolutional Neural Network

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

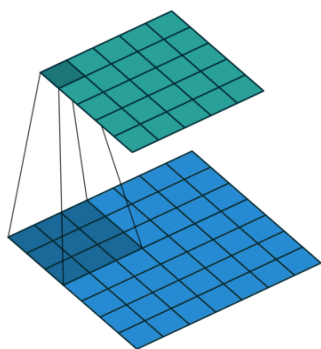


$$\text{CNN formulation: } h_v^{(l+1)} = \sigma(\sum_{u \in N(v) \cup \{v\}} W_l^u h_u^{(l)}), \quad \forall l \in \{0, \dots, L-1\}$$

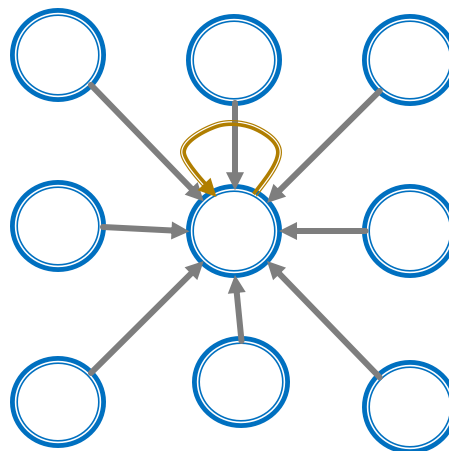
$N(v)$ represents the 8 neighbor pixels of v .

GNN vs. CNN

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



Image

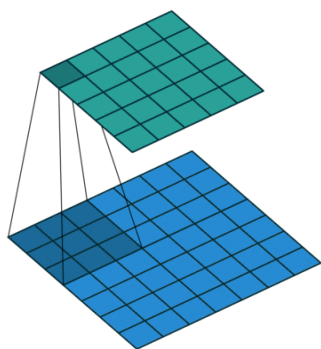


Graph

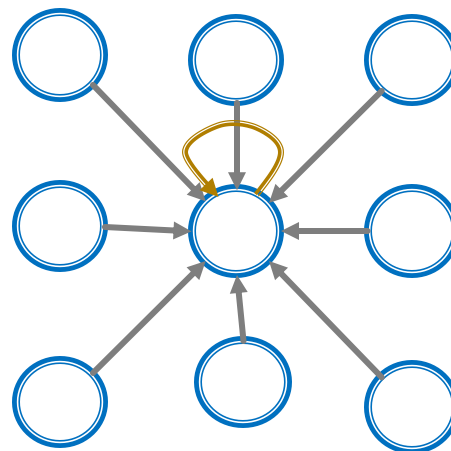
- GNN formulation: $h_v^{(l+1)} = \sigma(\mathbf{W}_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)}), \forall l \in \{0, \dots, L-1\}$
- CNN formulation: (previous slide) $h_v^{(l+1)} = \sigma(\sum_{u \in N(v) \cup \{v\}} W_l^u h_u^{(l)}), \forall l \in \{0, \dots, L-1\}$
if we rewrite:
 $h_v^{(l+1)} = \sigma(\sum_{u \in N(v)} \mathbf{W}_l^u h_u^{(l)} + B_l h_v^{(l)}), \forall l \in \{0, \dots, L-1\}$

GNN vs. CNN

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



Image



Graph

$$\text{GNN formulation: } h_v^{(l+1)} = \sigma(\mathbf{W}_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)}), \forall l \in \{0, \dots, L-1\}$$

$$\text{CNN formulation: } h_v^{(l+1)} = \sigma(\sum_{u \in N(v)} \mathbf{W}_l^u h_u^{(l)} + B_l h_v^{(l)}), \forall l \in \{0, \dots, L-1\}$$

Key difference: We can learn different $\mathbf{W}_l^u$ 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: $\{(-1, -1), (-1, 0), (-1, 1), \dots, (1, 1)\}$

GNN vs. CNN

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

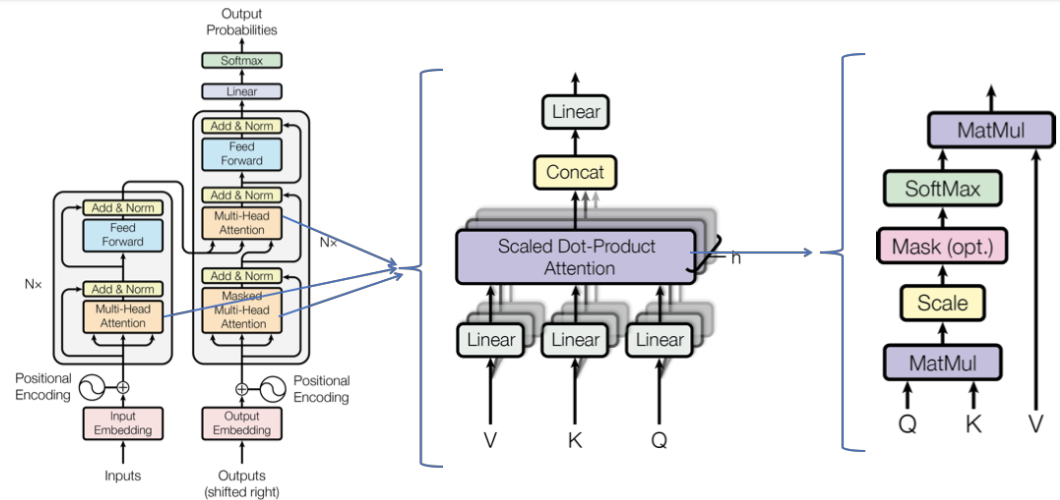


- CNN can be seen as a special GNN with fixed neighbor size and ordering:
 - The size of the filter is pre-defined for a CNN.
 - The advantage of GNN is it processes arbitrary graphs with different degrees for each node.
- CNN is not permutation invariant/equivariant.
 - Switching the order of pixels 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: $\{(-1,-1), (-1,0), (-1,1), \dots, (1,1)\}$

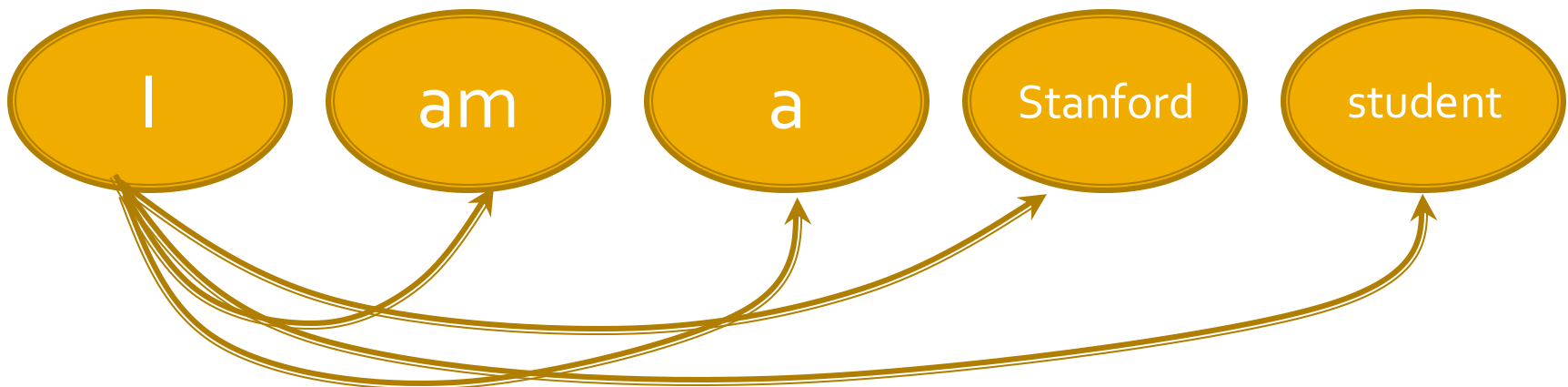
Transformer

Transformers are among the most popular architectures that achieve great performance in many sequence modeling tasks.



Key component: self-attention

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

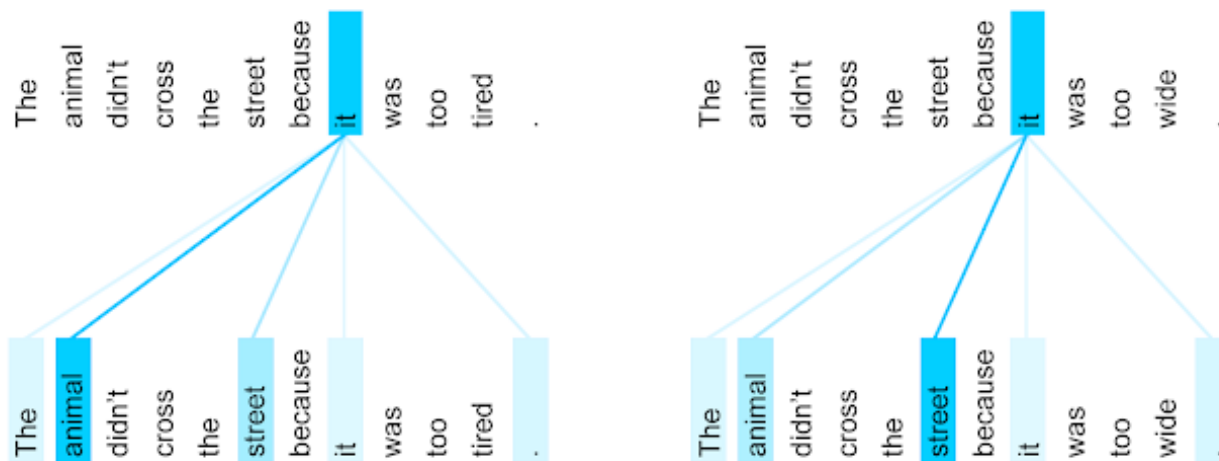


Transformer

A general definition of attention:

Given a set of vector values, and a vector query, **attention** is a technique to compute a weighted sum of the values, dependent on the query.

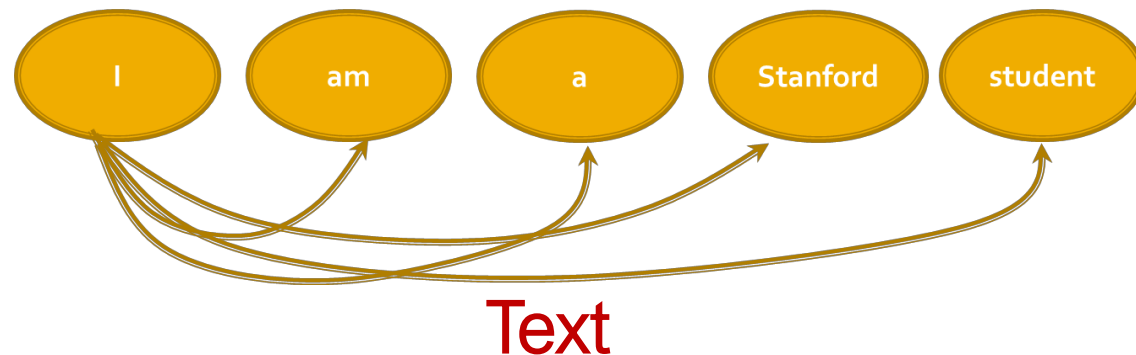
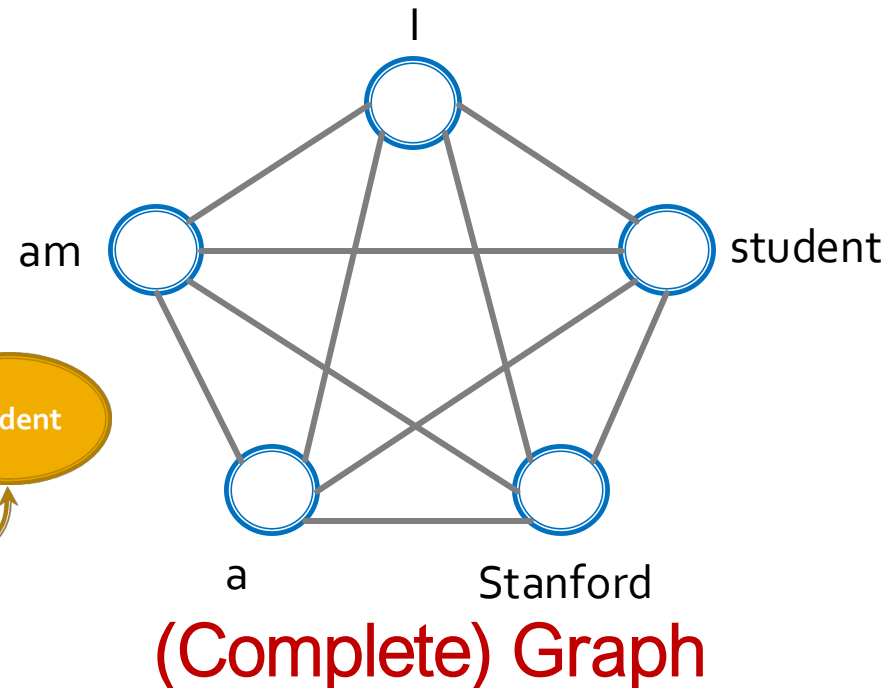
Each token/word has a **value vector** and a **query vector**. The value vector can be seen as the representation of the token/word. We use the query vector to calculate the attention score (weights in the weighted sum).



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 can be viewed as a special GNN

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 x , and the goal is to predict label 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:**

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

← **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

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

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

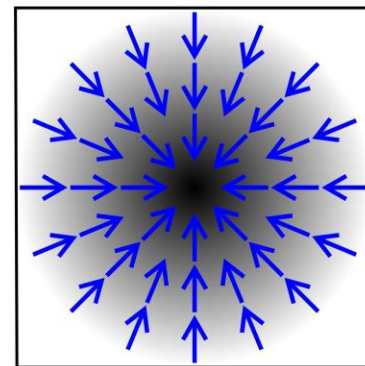
Loss Function Example

- One common loss for classification: **cross entropy (CE)**
- Label $\mathbf{y}$ is a categorical vector (**one-hot encoding**)
 - e.g. $\mathbf{y} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \end{bmatrix}$ $\mathbf{y}$ is of class “3”
- $f(\mathbf{x}) = \text{Softmax}(g(\mathbf{x}))$
 - Recall from lecture 3: $f(\mathbf{x})_i = \frac{e^{g(\mathbf{x})_i}}{\sum_{j=1}^C e^{g(\mathbf{x})_j}}$
where C is the number of classes.
 $g(\mathbf{x})_i$ denotes i -th coordinate of the vector output of func. $g(\mathbf{x})$
 - e.g. $f(\mathbf{x}) = \begin{bmatrix} 0.1 & 0.3 & 0.4 & 0.1 & 0.1 \end{bmatrix}$
- $\text{CE}(\mathbf{y}, f(\mathbf{x})) = -\sum_{i=1}^C (y_i \log f(\mathbf{x})_i)$
 - y_i and $f(\mathbf{x})_i$ 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:**
 - $\mathcal{L} = \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{T}} \text{CE}(\mathbf{y}, f(\mathbf{x}))$
 - $\mathcal{T}$: training set containing all pairs of data and labels $(\mathbf{x}, \mathbf{y})$

Machine Learning as Optimization

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

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



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

- $\Theta_1, \Theta_2 \dots$: 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 $\nabla_{\Theta} \mathcal{L}(\mathbf{y}, f(\mathbf{x}))$, where $\mathbf{x}$ 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** $\mathcal{B}$ containing a subset of the dataset, use it as input $\mathbf{x}$

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, Adadelata, RMSprop ...

Neural Network Function

- **Objective:** $\min_{\Theta} \mathcal{L}(\mathbf{y}, f(\mathbf{x}))$
- In deep learning, function f can be very complex
- **Example:**

- To start simple, consider linear function

$$f(\mathbf{x}) = \mathbf{W} \cdot \mathbf{x}, \quad \Theta = \{\mathbf{W}\}$$

- Then, if f returns a scalar, then $\mathbf{W}$ is a learnable **vector**

$$\nabla_{\mathbf{W}} f = \left(\frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}, \frac{\partial f}{\partial w_3} \dots \right)$$

- But, if f returns a vector, then $\mathbf{W}$ is the **weight matrix**

$$\nabla_{\mathbf{W}} f = \begin{bmatrix} \frac{\partial f_1}{\partial w_{11}} & \frac{\partial f_1}{\partial w_{12}} \\ \frac{\partial f_2}{\partial w_{21}} & \frac{\partial f_2}{\partial w_{22}} \end{bmatrix}$$

Jacobian
matrix of f

Intuition: Back Propagation

- **Goal:** $\min_{\Theta} \mathcal{L}(\mathbf{y}, f(\mathbf{x}))$
 - To minimize $\mathcal{L}$, we need to evaluate the gradient:
$$\nabla_{\mathbf{w}} \mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial w_1}, \frac{\partial \mathcal{L}}{\partial w_2}, \frac{\partial \mathcal{L}}{\partial w_3} \dots \right)$$
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 $\mathcal{B}$.

Back-propagation

- How about a more complex function:

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

In other words:

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

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

$$g(z) = W_2z$$

- Recall **chain rule**:

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

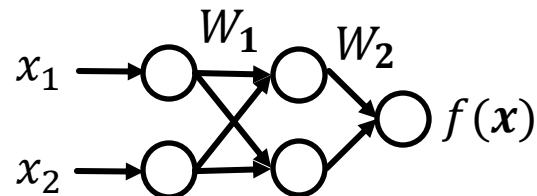
- **Example:** $\nabla_{\mathbf{x}} f = \frac{\partial f}{\partial (W_1\mathbf{x})} \cdot \frac{\partial (W_1\mathbf{x})}{\partial \mathbf{x}}$

- **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

- $f(\mathbf{x}) = g(h(\mathbf{x})) = W_2(W_1\mathbf{x})$



- $\mathcal{L} = \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{B}} \left\| (\mathbf{y}, -f(\mathbf{x})) \right\|_2$

- The loss $\mathcal{L}$ sums the L2 loss in a minibatch $\mathcal{B}$.

- **Hidden layer:**

- Intermediate representation of input $\mathbf{x}$
- Here we use $h(\mathbf{x}) = W_1\mathbf{x}$ to denote the hidden layer
- $f(\mathbf{x}) = W_2h(\mathbf{x})$

Back-propagation Example (2)

■ Forward propagation:

Compute loss starting from input

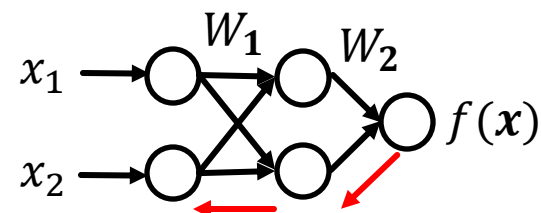


Remember:

$$f(x) = W_2(W_1x)$$

$$h(x) = W_1x$$

$$g(z) = W_2z$$



■ Back-propagation to compute gradient of

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

Start from loss, compute the gradient

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

Compute backwards

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

Compute backwards

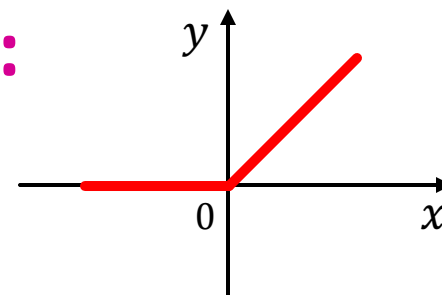
Non-linearity

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

- **We introduce non-linearity:**

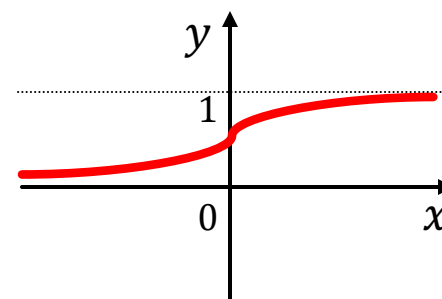
- **Rectified linear unit (ReLU)**

$$\text{ReLU}(x) = \max(x, 0)$$



- **Sigmoid**

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

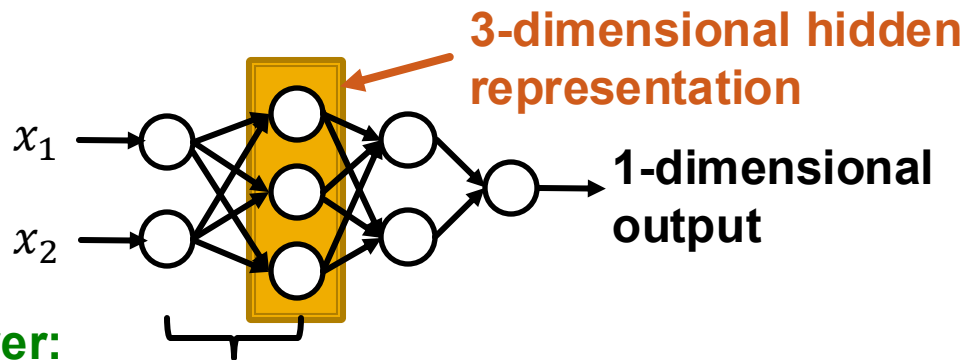


Multi-layer Perceptron (MLP)

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

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

- 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 $\mathbf{x}^{(l)}$
- σ is non-linearity function (e.g., sigmoid)
- Suppose $\mathbf{x}$ is 2-dimensional, with entries x_1 and x_2



Every layer:
Linear transformation +
non-linearity

Summary

- **Objective function:**

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

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

Outline of Today's Lecture

1. Basics of deep learning



2. Deep learning for graphs



3. Graph Convolutional Networks

4. GNNs subsume CNNs