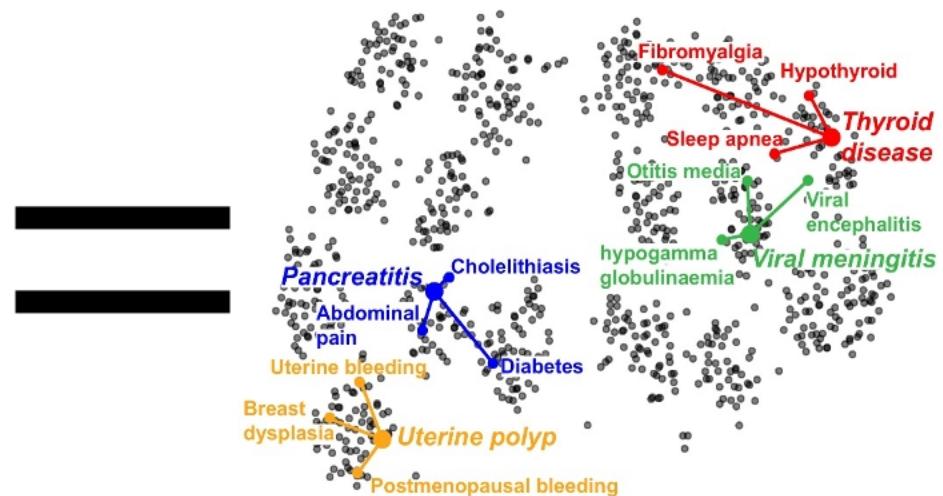
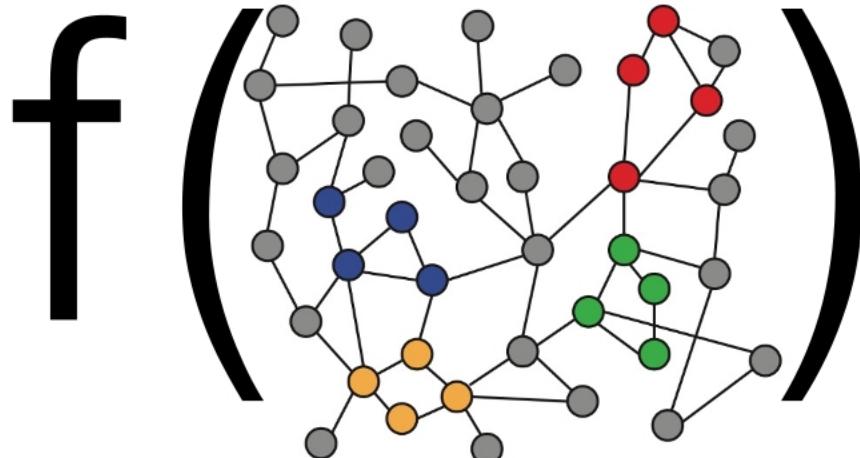


# **Graph Neural Networks**

# Recap: Node Embeddings

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



Input graph

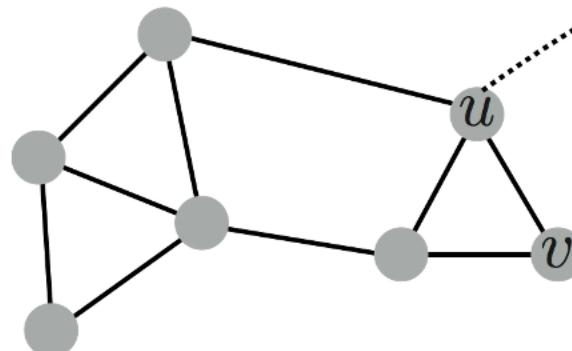
2D node embeddings

How to learn mapping function  $f$ ?

# Recap: Node Embeddings

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

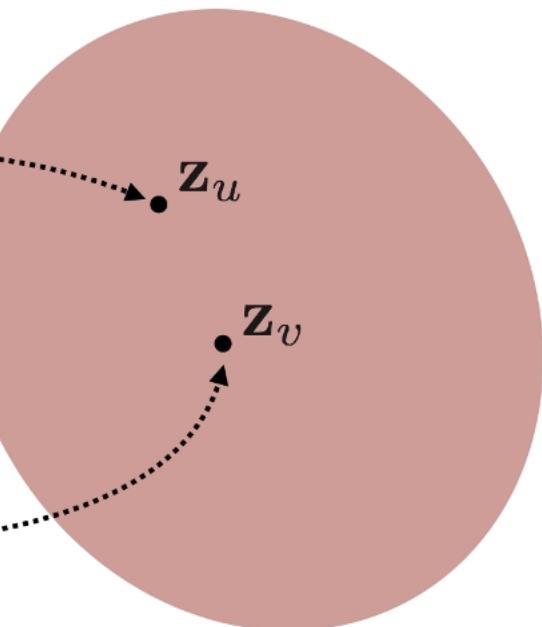
Need to define!



encode nodes

$\text{ENC}(u)$

$\text{ENC}(v)$

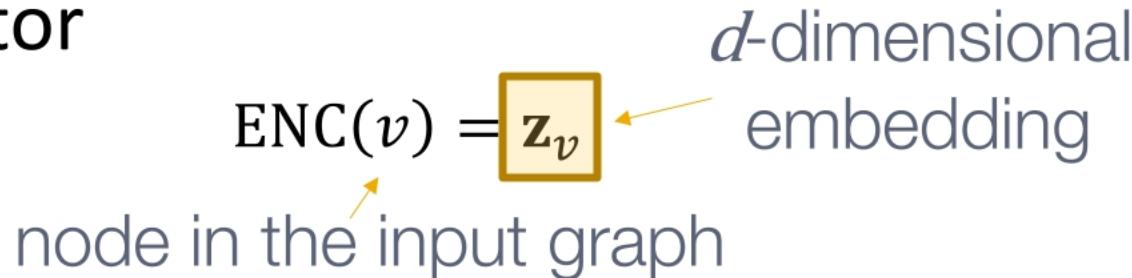


Input network

$d$ -dimensional  
embedding space

# Recap: Two Key Components

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



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

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

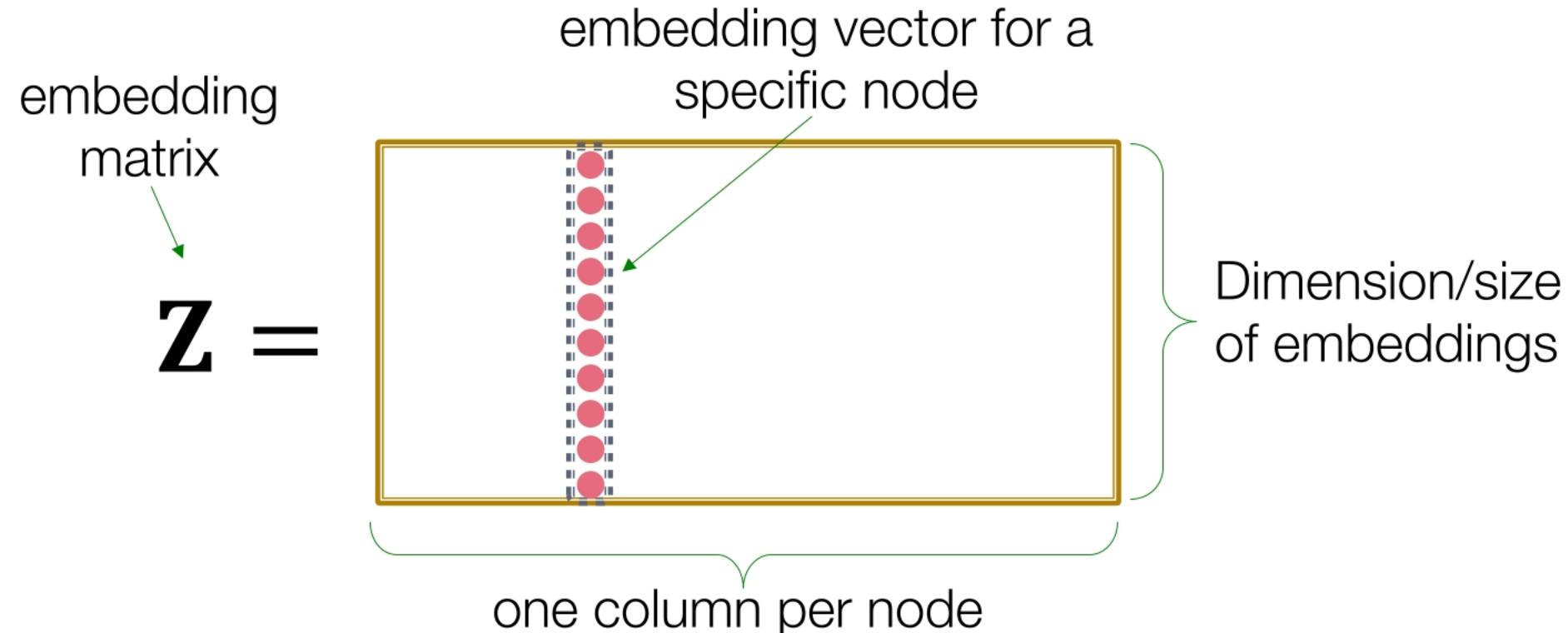
Similarity of  $u$  and  $v$  in the original network

**Decoder**

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

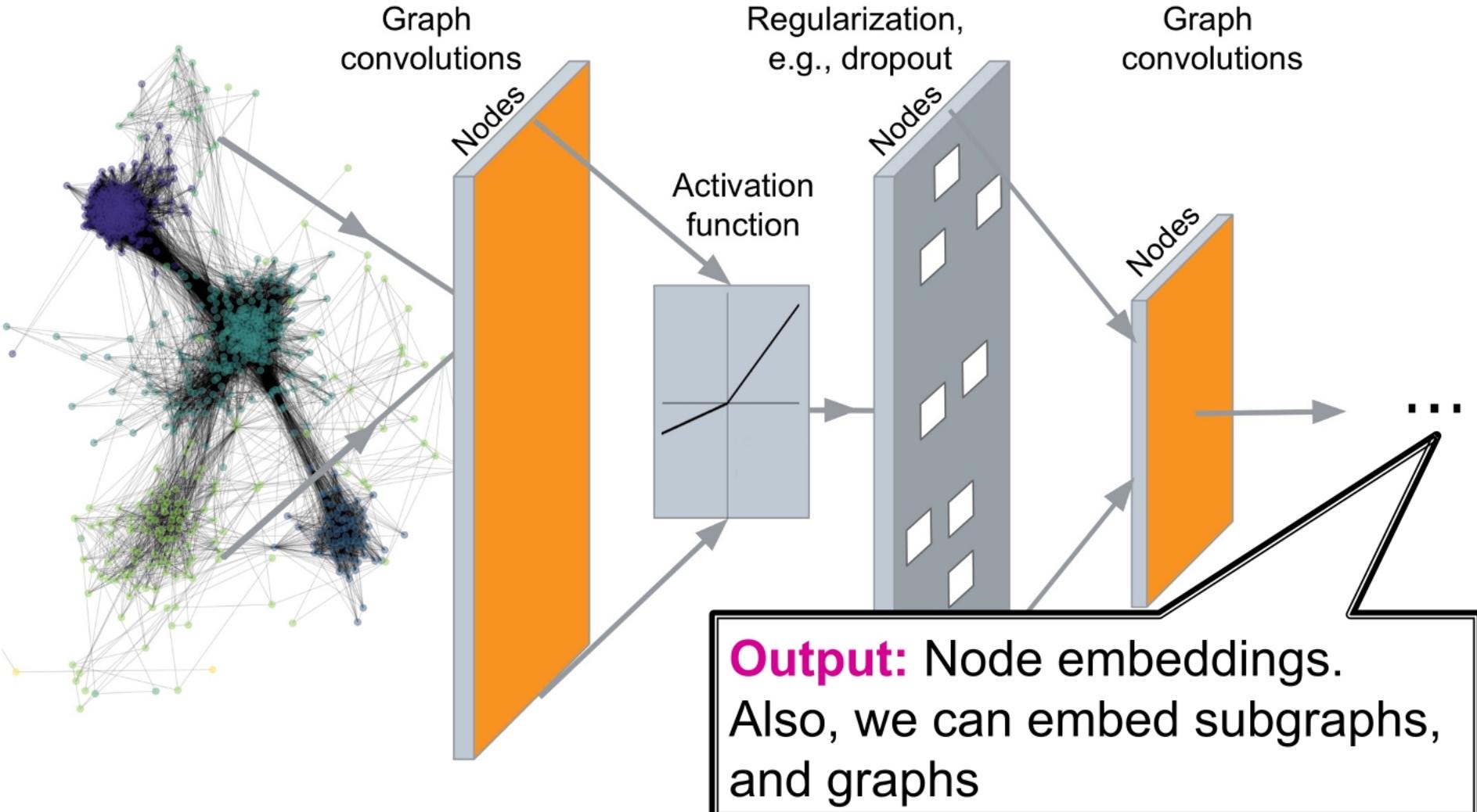
- **Today:** We will now discuss deep learning methods based on **graph neural networks (GNNs)**:

$$\text{ENC}(\nu) =$$

**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

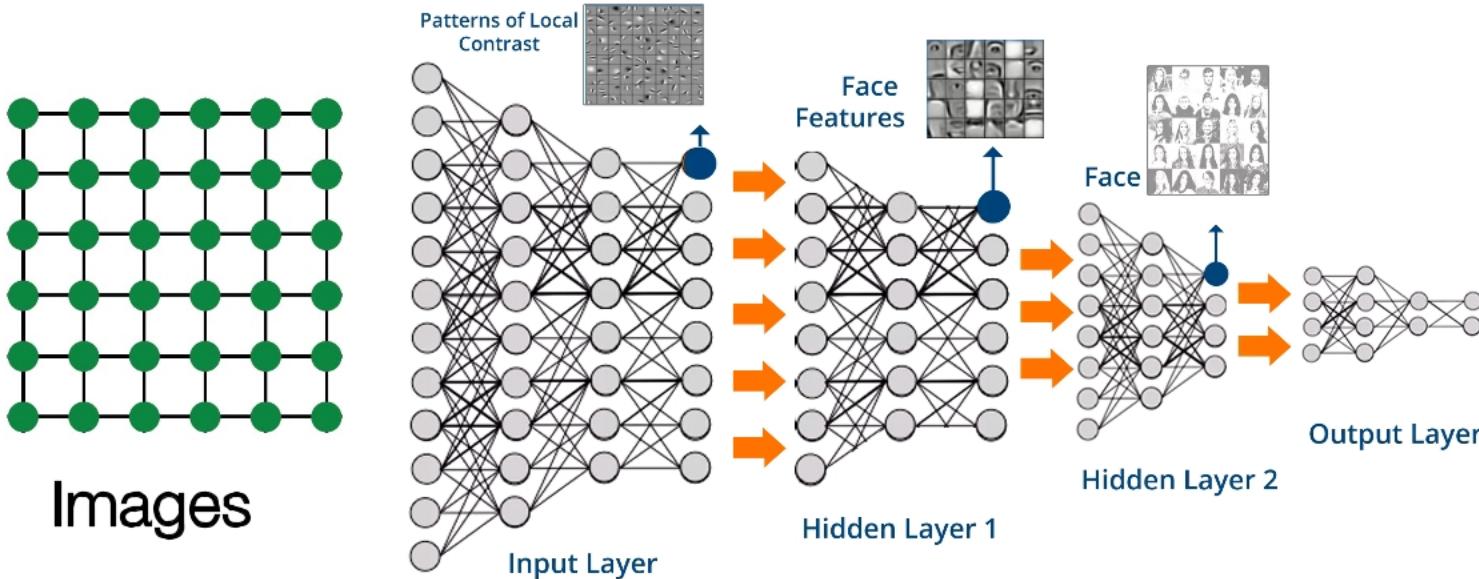


# Tasks on Networks

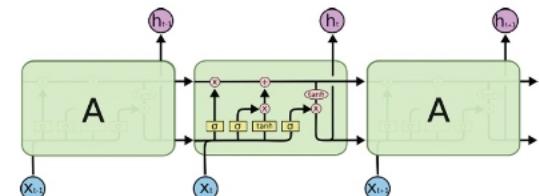
## Tasks we will be able to solve:

- 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



Text/Speech

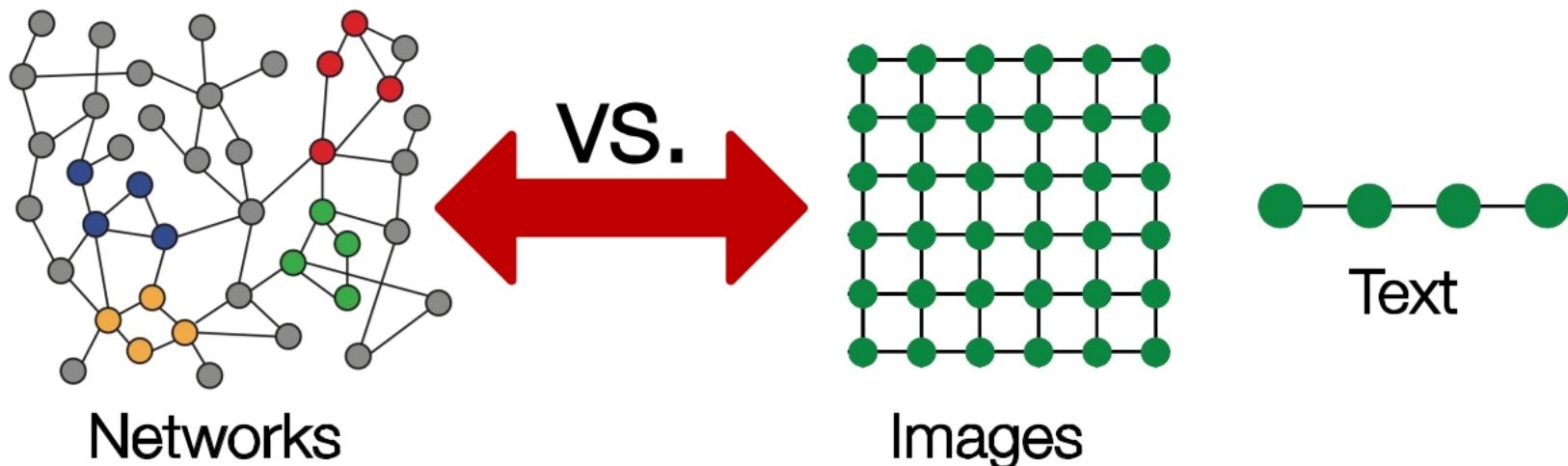


Modern deep learning toolbox is designed  
for simple sequences & grids

# 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. Basics of deep learning**



**2. Deep learning for graphs**

**3. Graph Convolutional Networks**

**4. GNNs subsume CNNs and  
Transformers**

# Basics of Deep Learning

# Machine Learning as Optimization

- **Supervised learning:** we are given input  $\mathbf{x}$ , and the goal is to predict label  $\mathbf{y}$ .
- **Input  $\mathbf{x}$  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:

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

Objective function

- $\Theta$ : 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

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

- 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

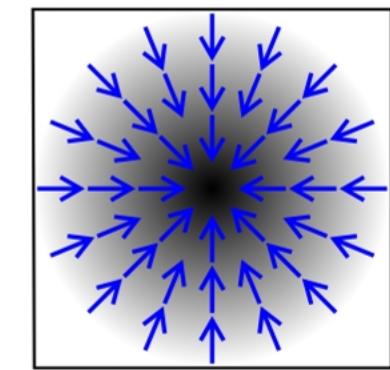
- One common loss for classification: **cross entropy (CE)**
- Label  $\mathbf{y}$  is a categorical vector (**one-hot encoding**)
  - e.g.  $\mathbf{y} = \begin{array}{c|c|c|c|c} \text{o} & \text{o} & \text{1} & \text{o} & \text{o} \end{array}$   $\mathbf{y}$  is of class "3"
- $f(\mathbf{x}) = \text{Softmax}(g(\mathbf{x}))$ 
  - Recall from lecture 3:  $f(\mathbf{x})_i = \frac{e^{g(\mathbf{x})_i}}{\sum_{j=1}^C e^{g(\mathbf{x})_j}}$   $g(\mathbf{x})_i$  denotes  $i$ -th coordinate of the vector output of func.  $g(\mathbf{x})$
  - where  $C$  is the number of classes.
  - e.g.  $f(\mathbf{x}) = \begin{array}{c|c|c|c|c} \text{0.1} & \text{0.3} & \text{0.4} & \text{0.1} & \text{0.1} \end{array}$
- $\text{CE}(\mathbf{y}, f(\mathbf{x})) = -\sum_{i=1}^C (y_i \log f(\mathbf{x})_i)$ 
  - $y_i$  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

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

- $\Theta_1, \Theta_2 \dots$  : components of  $\Theta$
- 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**.



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

# 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  $\Theta$  iteratively
  - **Iteration:** 1 step of gradient descent
- **Learning rate (LR)  $\eta$ :**
  - Hyperparameter that controls the size of gradient step
  - Can vary over the course of training (LR scheduling)
- **Ideal termination condition:** gradient =  $\mathbf{0}$ 
  - 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  $\nabla_{\Theta} \mathcal{L}(y, f(\mathbf{x}))$ , 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, Adadelta, RMSprop ...

# Neural Network Function

- **Objective:**  $\min_{\Theta} \mathcal{L}(y, f(\mathbf{x}))$
- 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**

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

Jacobian  
matrix of  $f$

# Intuition: Back Propagation

- **Goal:**  $\min_{\Theta} \mathcal{L}(y, f(\mathbf{x}))$

- To minimize  $\mathcal{L}$ , we need to evaluate the gradient:

$$\nabla_{\mathbf{W}} \mathcal{L} = \left( \frac{\partial f}{\partial \mathbf{w}_1}, \frac{\partial f}{\partial \mathbf{w}_2}, \frac{\partial f}{\partial \mathbf{w}_3} \dots \right)$$

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:

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

- Recall chain rule:

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

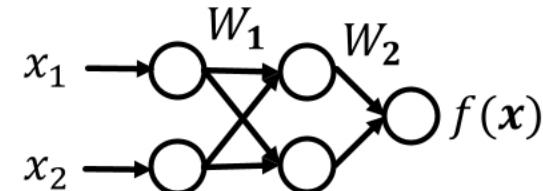
- Example:  $\nabla_{\mathbf{x}} f = \frac{\partial f}{\partial (W_1 \mathbf{x})} \cdot \frac{\partial (W_1 \mathbf{x})}{\partial \mathbf{x}}$

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

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

# Back-propagation Example (1)

- **Example:** Simple 2-layer linear network
- $f(\mathbf{x}) = g(h(x)) = W_2(W_1 \mathbf{x})$
- $\mathcal{L} = \sum_{(x,y) \in \mathcal{B}} \left\| (y, -f(x)) \right\|_2$ 
  - The loss  $\mathcal{L}$  sums the L2 loss in a minibatch  $\mathcal{B}$ .
- **Hidden layer:**
  - Intermediate representation of input  $\mathbf{x}$
  - Here we use  $h(x) = W_1 \mathbf{x}$  to denote the hidden layer
  - $f(\mathbf{x}) = W_2 h(\mathbf{x})$



# Back-propagation Example (2)

## ■ Forward propagation:

Compute loss starting from input

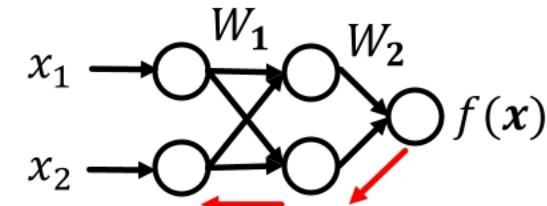


Remember:

$$f(x) = W_2(W_1 x)$$

$$h(x) = W_1 x$$

$$g(z) = W_2 z$$



## ■ Back-propagation to compute gradient of

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

Start from loss, compute the gradient

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



Compute backwards

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

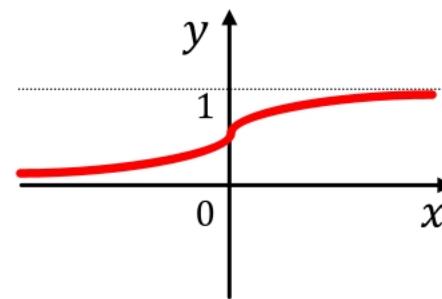
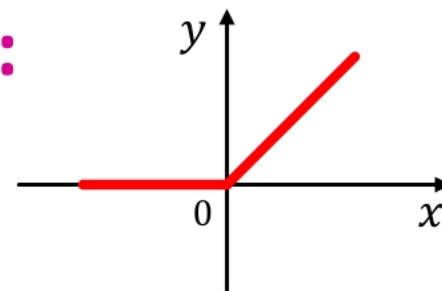


Compute backwards

# Non-linearity

- Note that in  $f(\mathbf{x}) = W_2(W_1 \mathbf{x})$ ,  $W_2 W_1$  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)  
 $ReLU(x) = \max(x, 0)$
  - Sigmoid

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

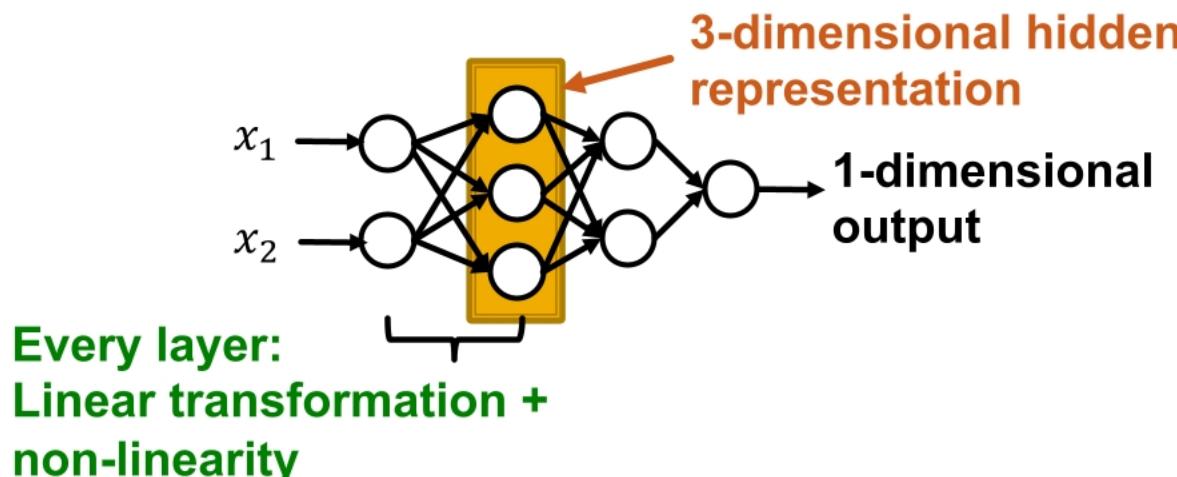


# Multi-layer Perceptron (MLP)

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



# Summary

- **Objective function:**

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

- $f$  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  $\Theta$  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 and  
Transformers

# Deep Learning for Graphs

# Content

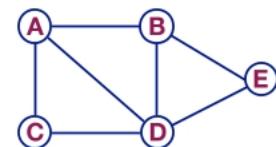
- **Local network neighborhoods:**
  - Describe aggregation strategies
  - Define computation graphs
- **Stacking multiple layers:**
  - Describe the model, parameters, training
  - How to fit the model?
  - Simple example for unsupervised and supervised training

# Setup

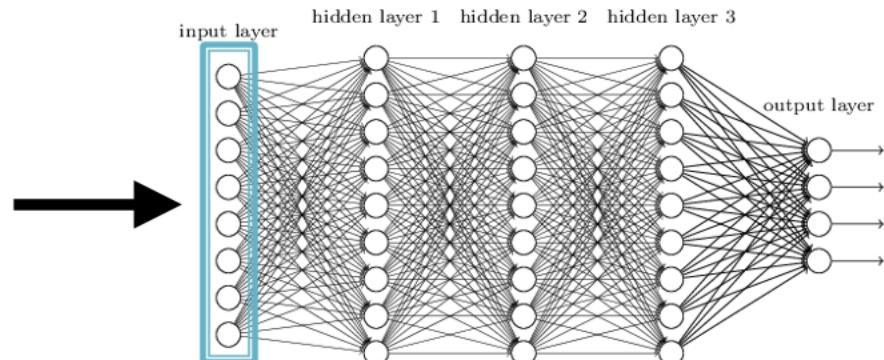
- Assume we have a graph  $G$ :
  - $V$  is the **vertex set**
  - $A$  is the **adjacency matrix** (assume binary)
  - $X \in \mathbb{R}^{m \times |V|}$  is a matrix of **node features**
  - $v$ : a node in  $V$ ;  $N(v)$ : the set of neighbors of  $v$ .
  - **Node features:**
    - Social networks: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



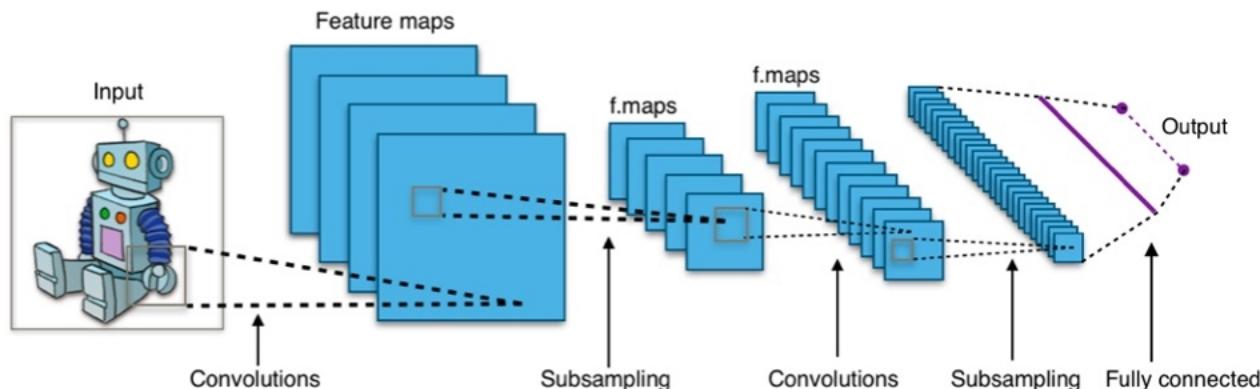
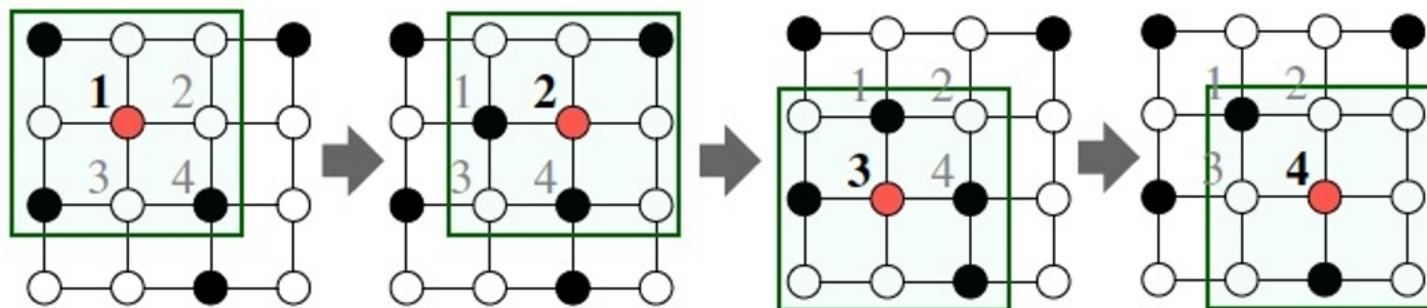
	A	B	C	D	E	Feat
A	0	1	1	1	0	1 0
B	1	0	0	1	1	0 0
C	1	0	0	1	0	0 1
D	1	1	1	0	1	1 1
E	0	1	0	1	0	1 0



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

# Idea: Convolutional Networks

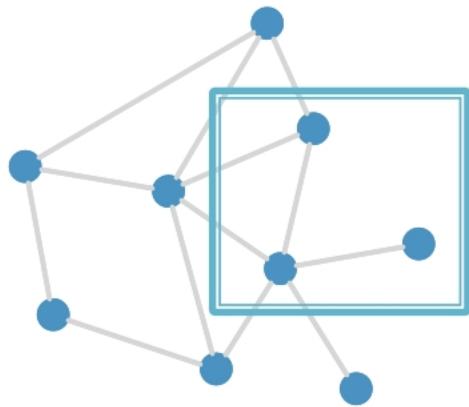
CNN on an image:



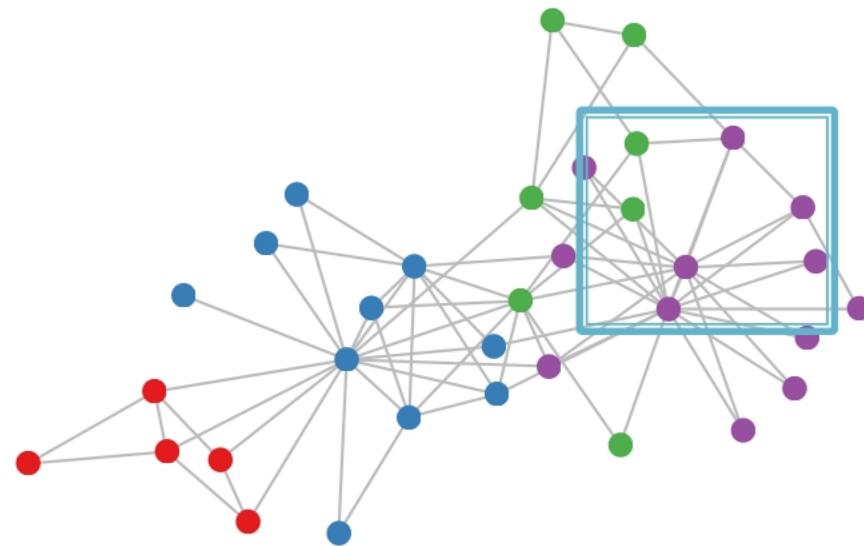
Goal is to generalize convolutions beyond simple lattices  
Leverage node features/attributes (e.g., text, images)

# Real-World Graphs

But our graphs look like this:



or this:



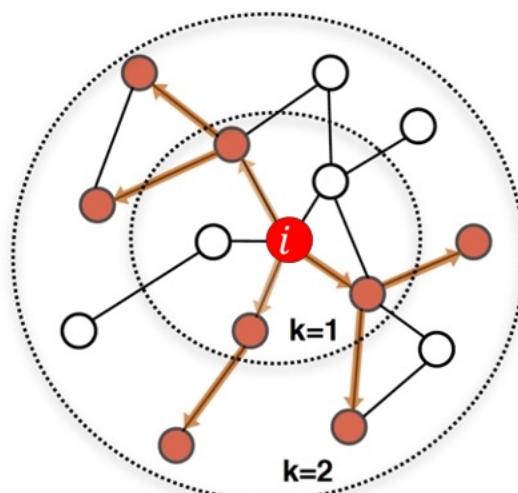
- There is no fixed notion of locality or sliding window on the graph
- Graph is permutation invariant

# Permutation Invariance

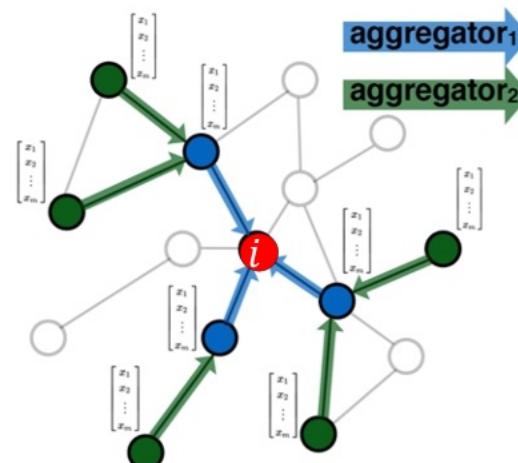
- **Graph does not have a canonical order of the nodes!**
- We can have many different order plans.

# Graph Convolutional Networks

**Idea:** Node's neighborhood defines a computation graph



Determine node computation graph

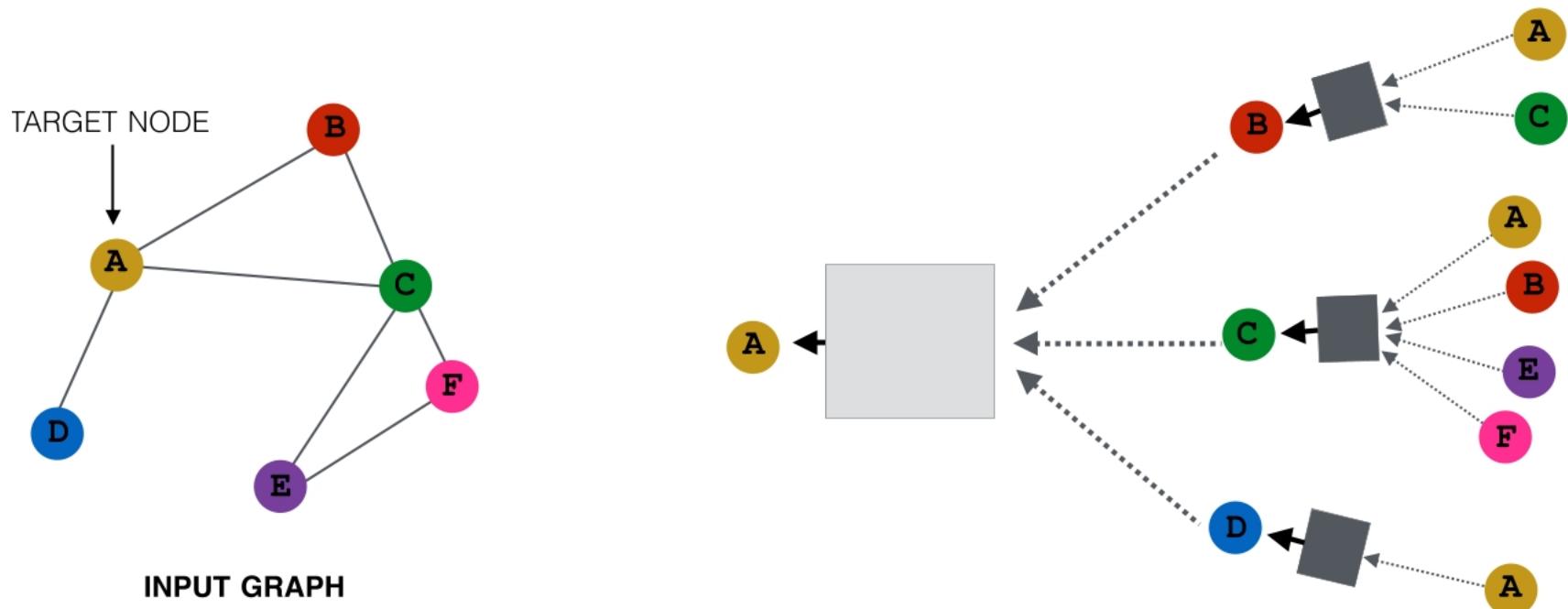


Propagate and transform information

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

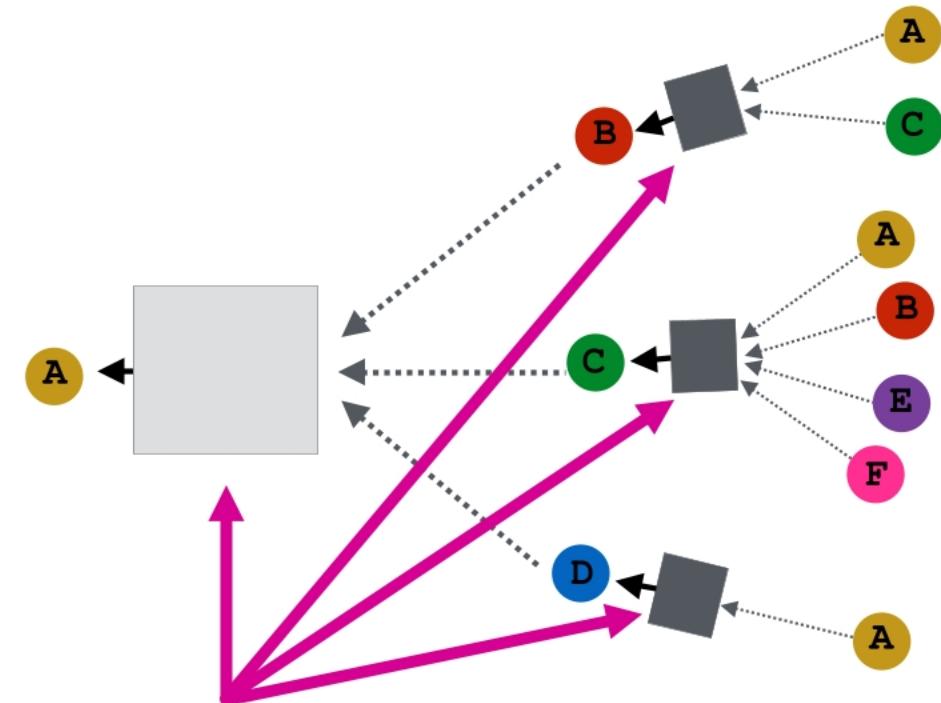
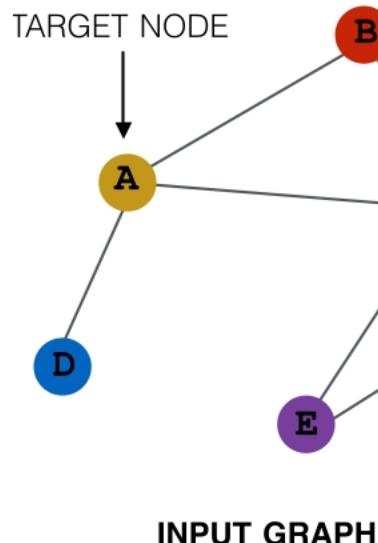
# Idea: Aggregate Neighbors

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



# Idea: Aggregate Neighbors

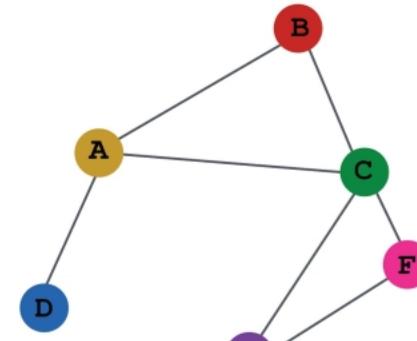
- **Intuition:** Nodes aggregate information from their neighbors using neural networks



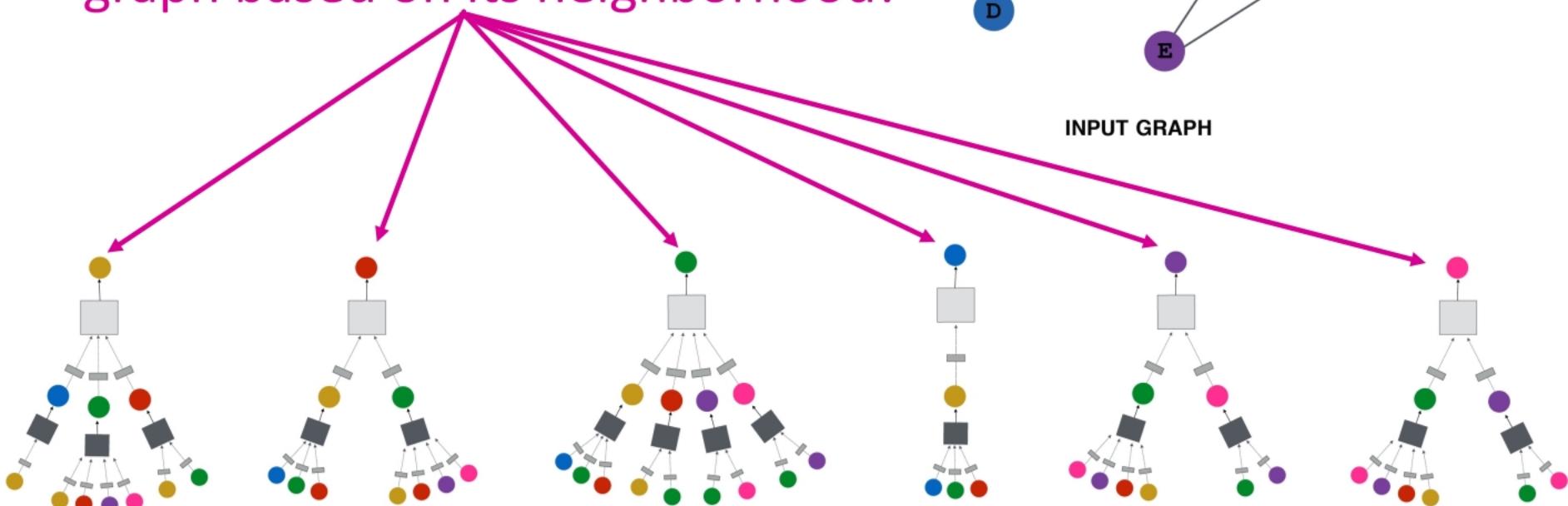
# Idea: Aggregate Neighbors

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

Every node defines a computation graph based on its neighborhood!

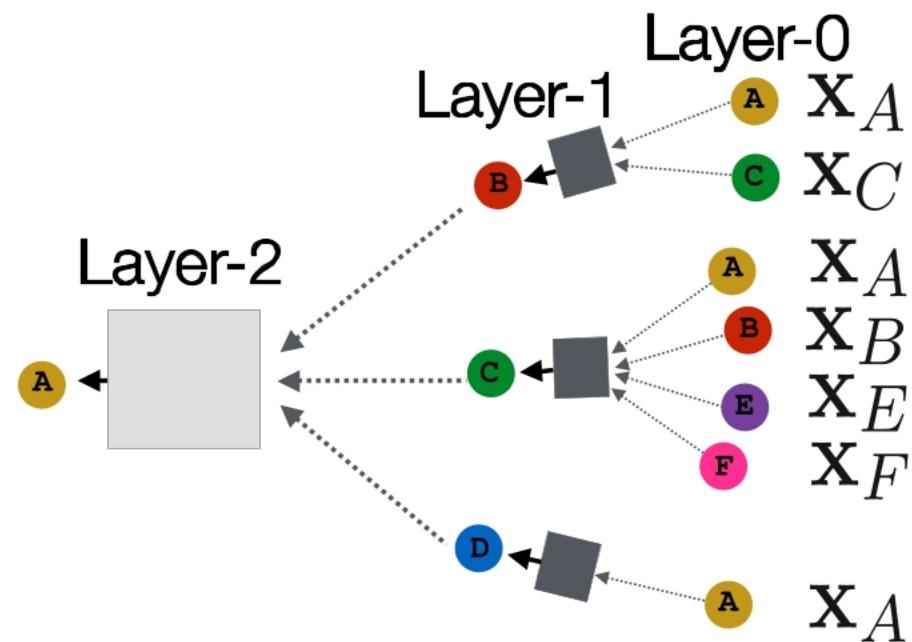
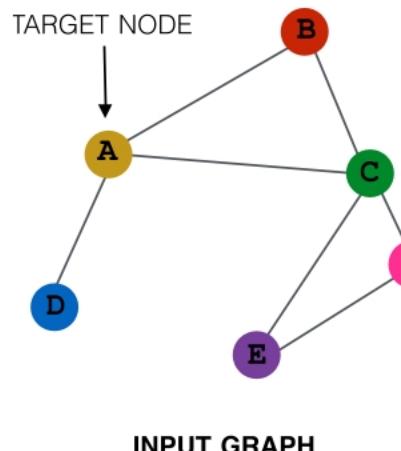


INPUT GRAPH



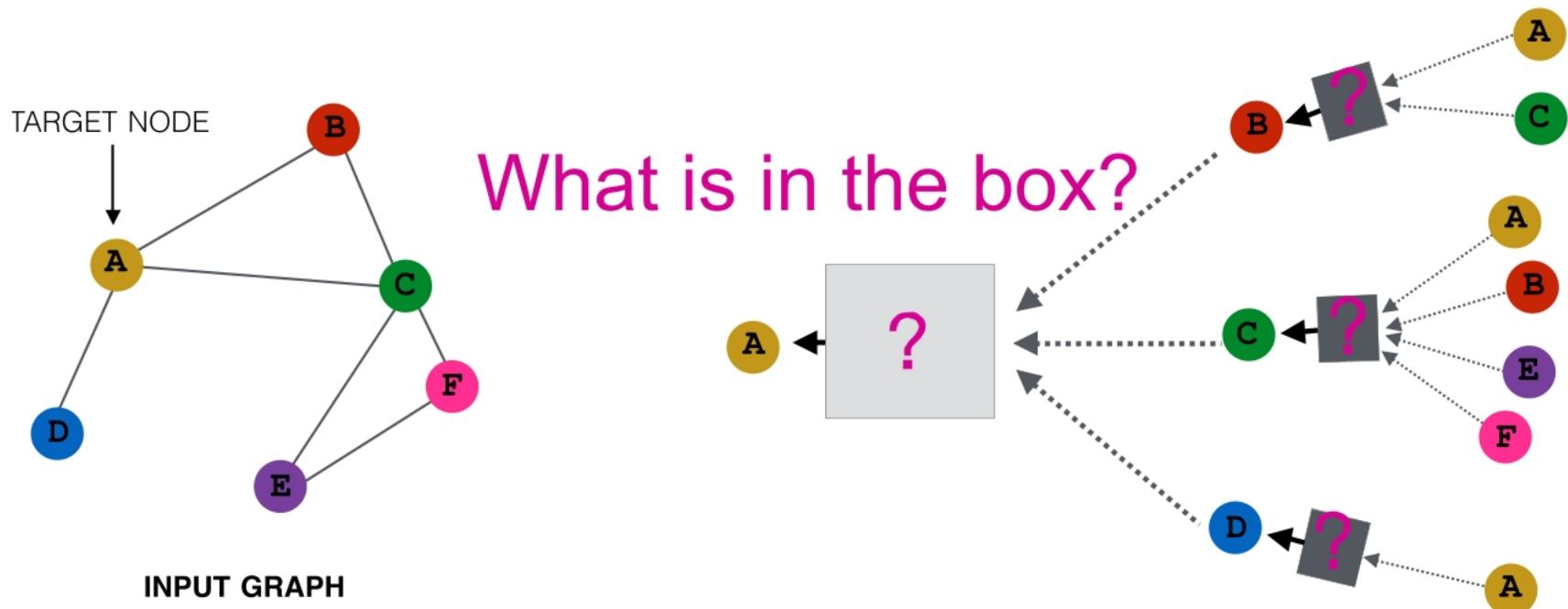
# Deep Model: Many Layers

- 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



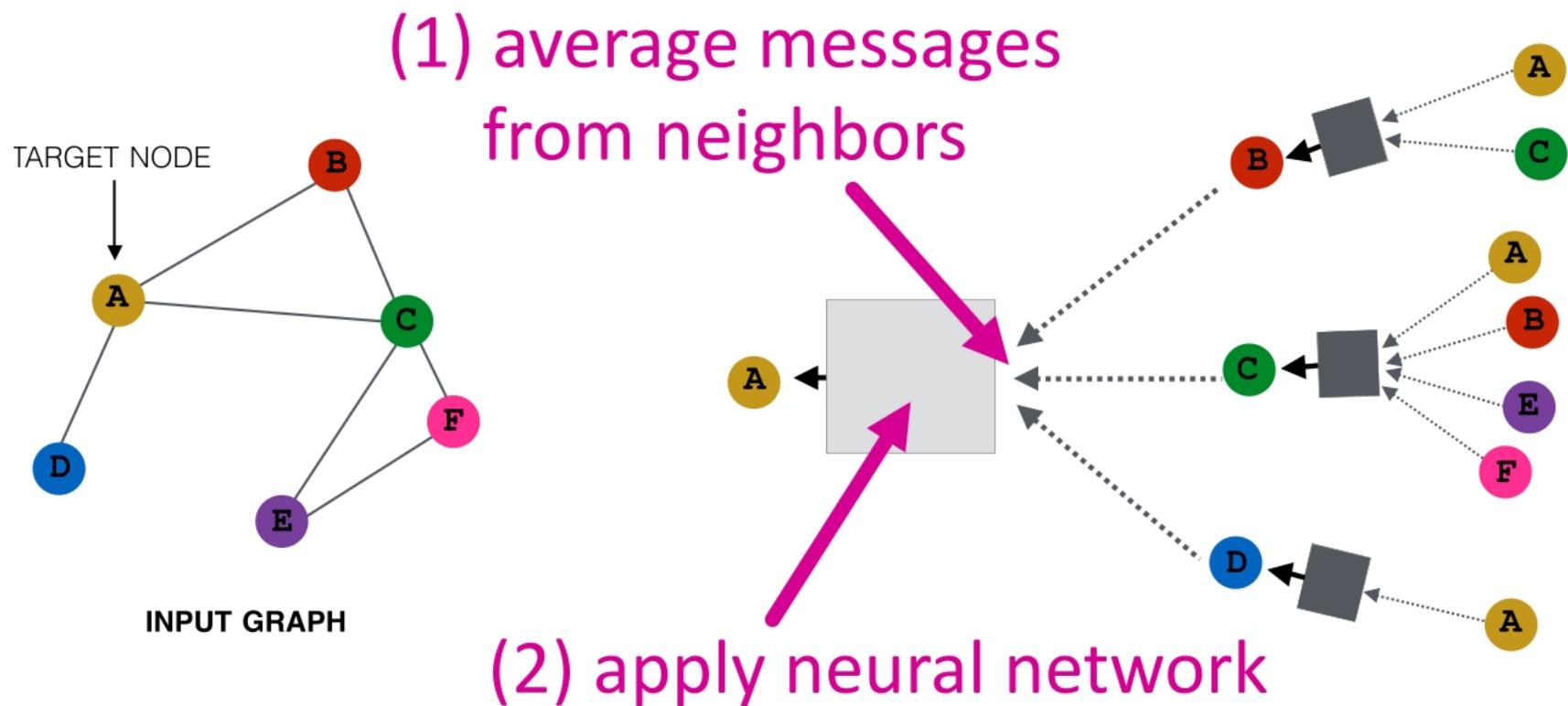
# Neighborhood Aggregation

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



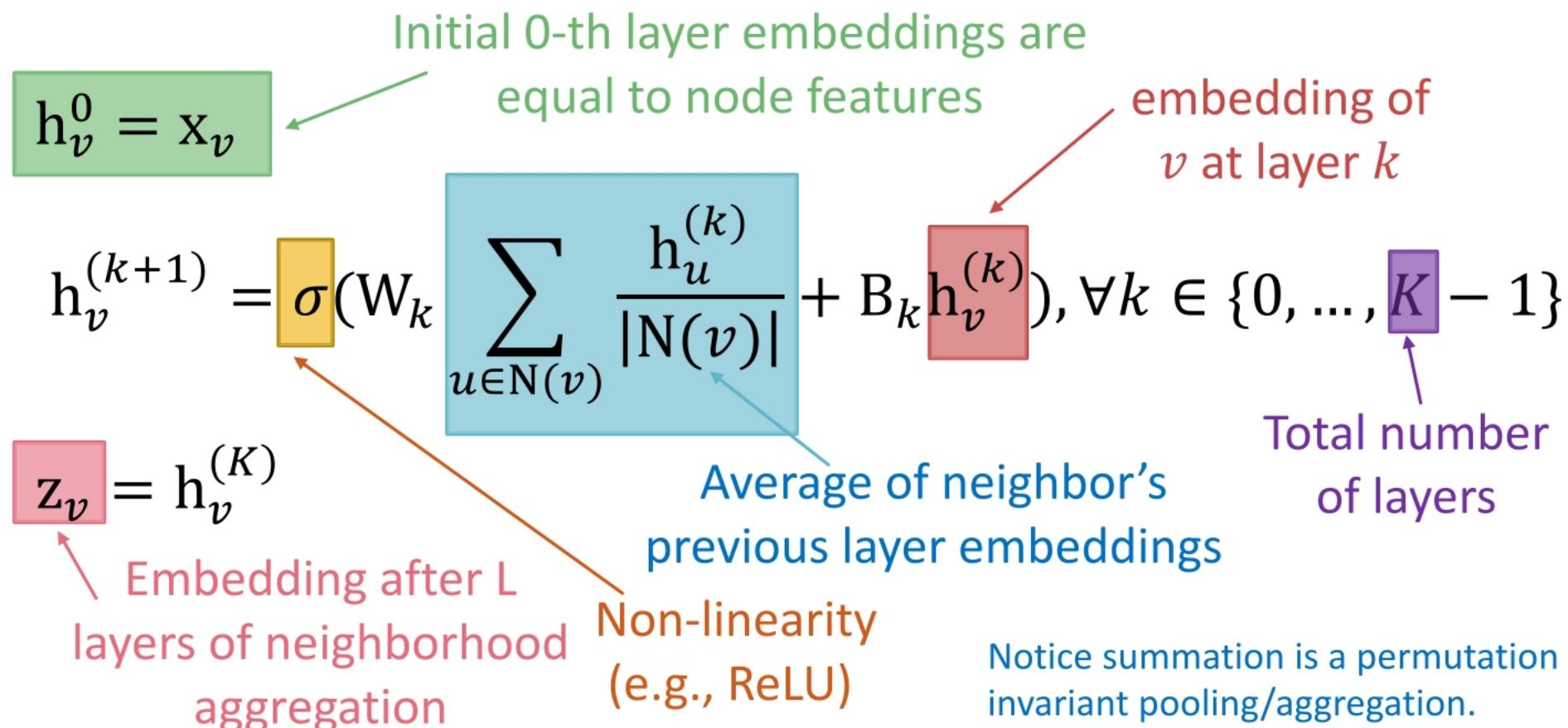
# Neighborhood Aggregation

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



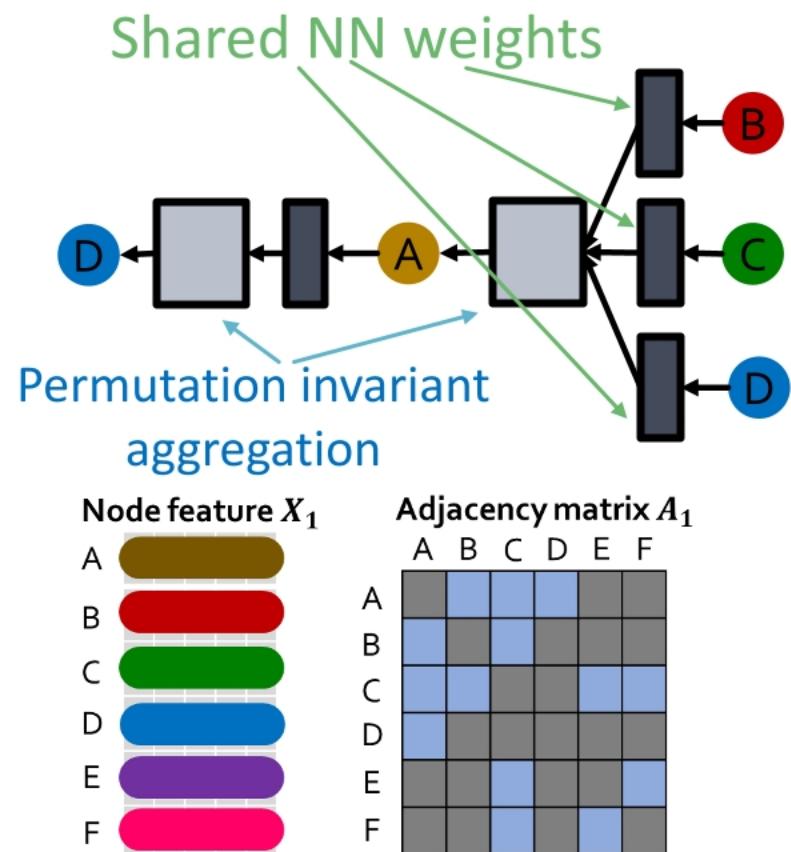
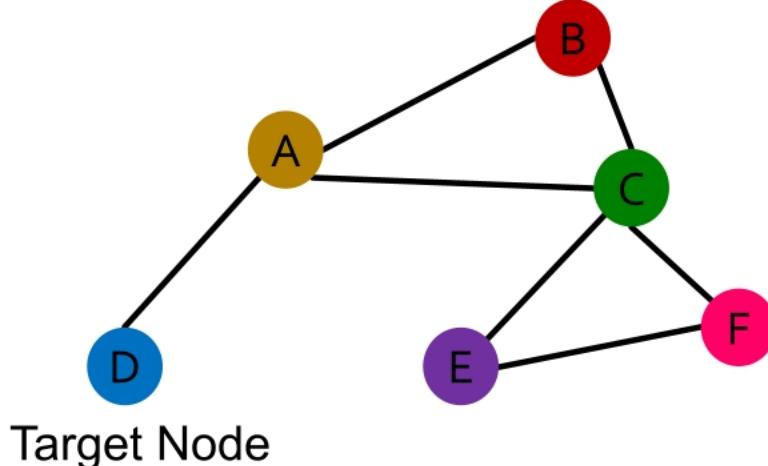
# The Math: Deep Encoder

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



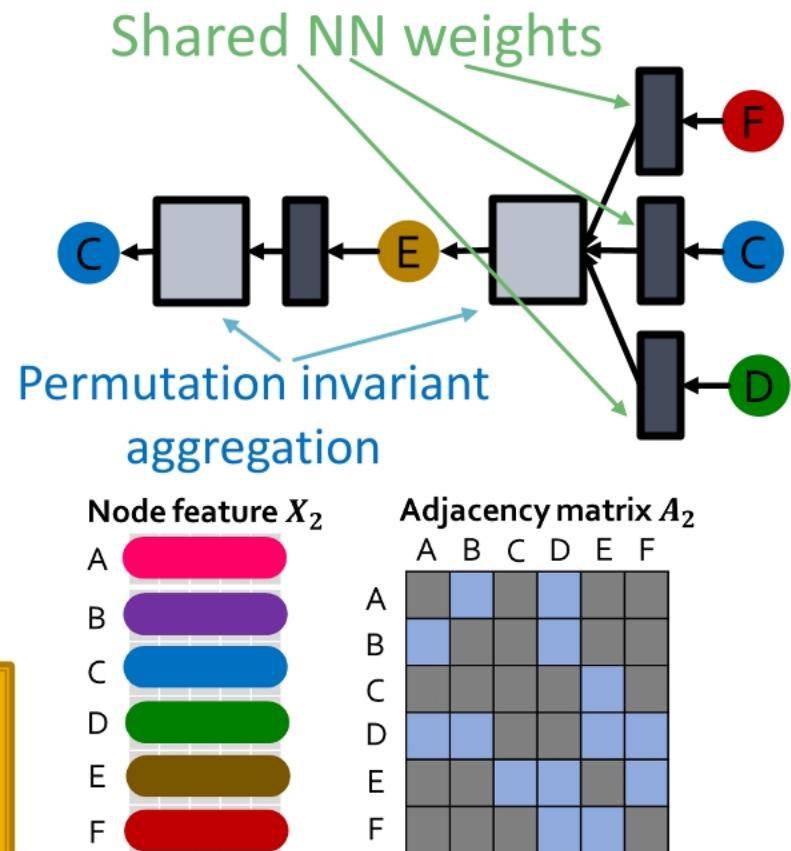
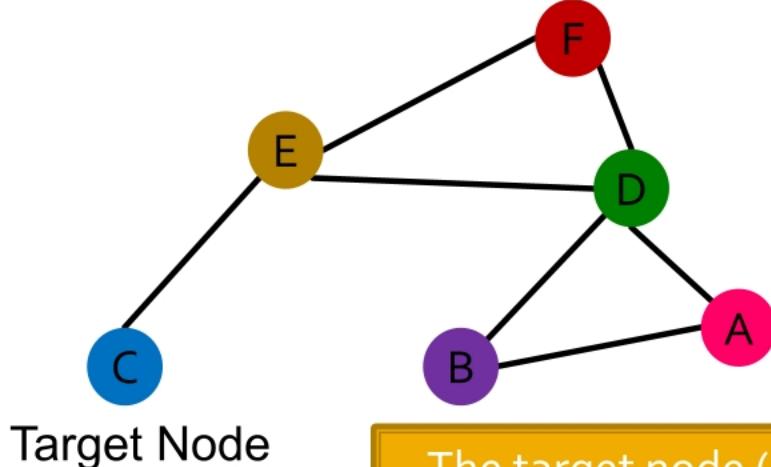
# Equivariant Property

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



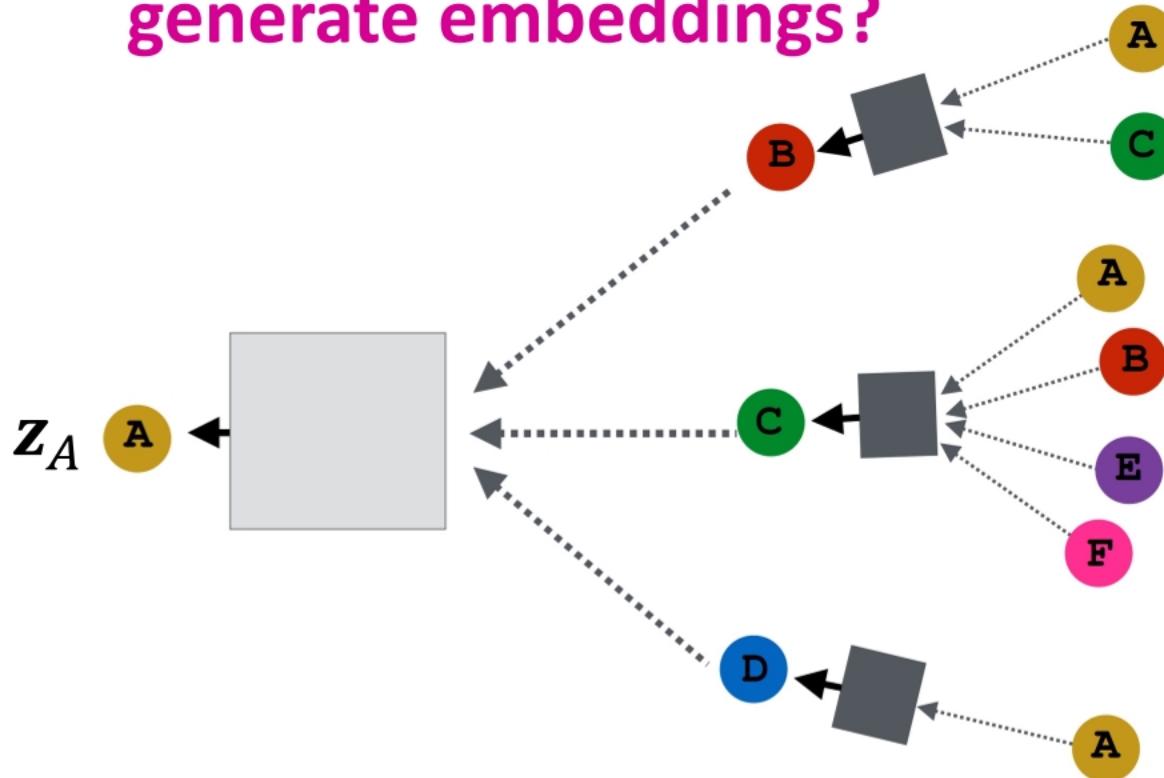
# Equivariant Property

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



# 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)

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

Final node embedding

```
graph TD; x_v[x_v] --> h0[h_v^(0)]; h0 --> hkplus1[h_v^(k+1)]; hkplus1 --> zk[z_v]; h0 --> text["Trainable weight matrices  
(i.e., what we learn)"]; hkplus1 --> Wk[W_k]; hkplus1 --> Bk[B_k]; Wk --> sum(( )); Bk --> sum; sum --> hkplus1; text --> hkplus1; sum --> zk; zk --> fnembedding[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:

$$D_{v,v}^{-1} = 1/|N(v)|$$

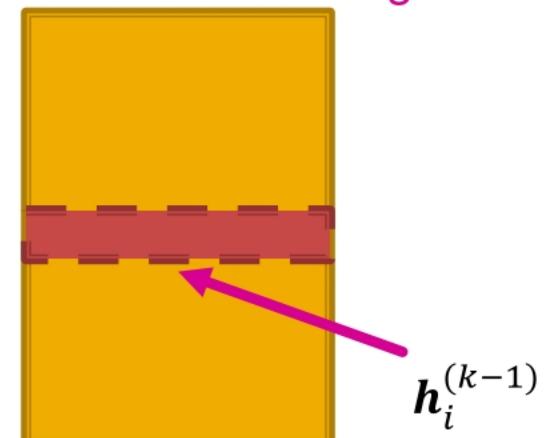
- Therefore,

$$\sum_{u \in N(v)} \frac{h_u^{(k-1)}}{|N(v)|}$$



$$H^{(k+1)} = D^{-1} A H^{(k)}$$

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

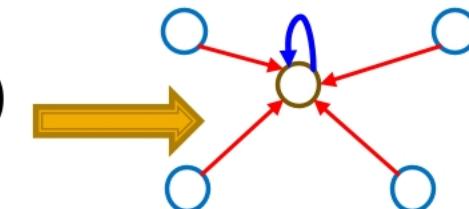


# Matrix Formulation (2)

- Re-writing update function in matrix form:

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

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

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

- $\mathbf{y}$ : 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

- “Similar” nodes have similar embeddings

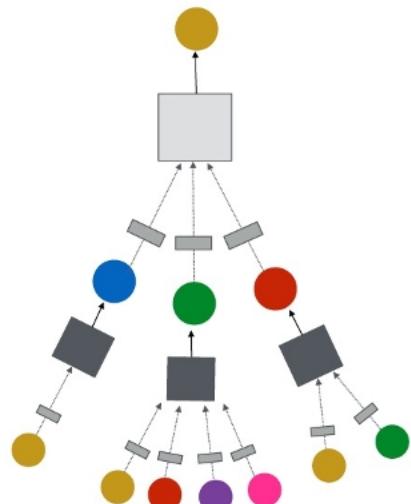
$$\mathcal{L} = \sum_{z_u, z_v} \text{CE}(y_{u,v}, \text{DEC}(z_u, z_v))$$

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

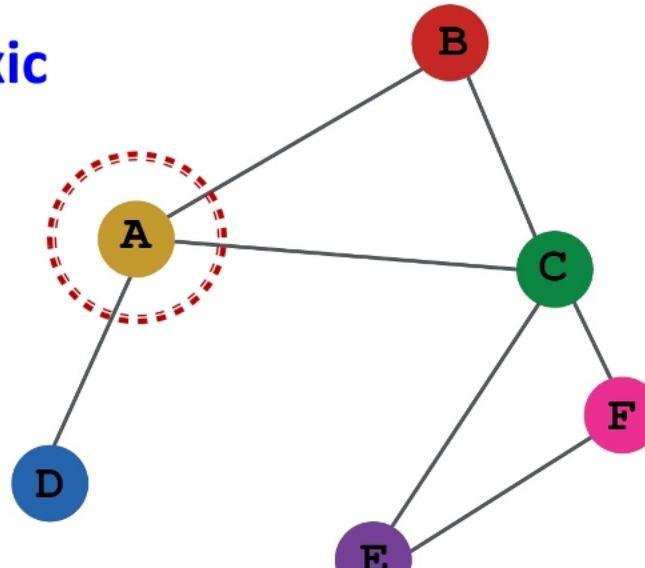
# Supervised Training

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

Safe or toxic  
drug?



Safe or toxic  
drug?

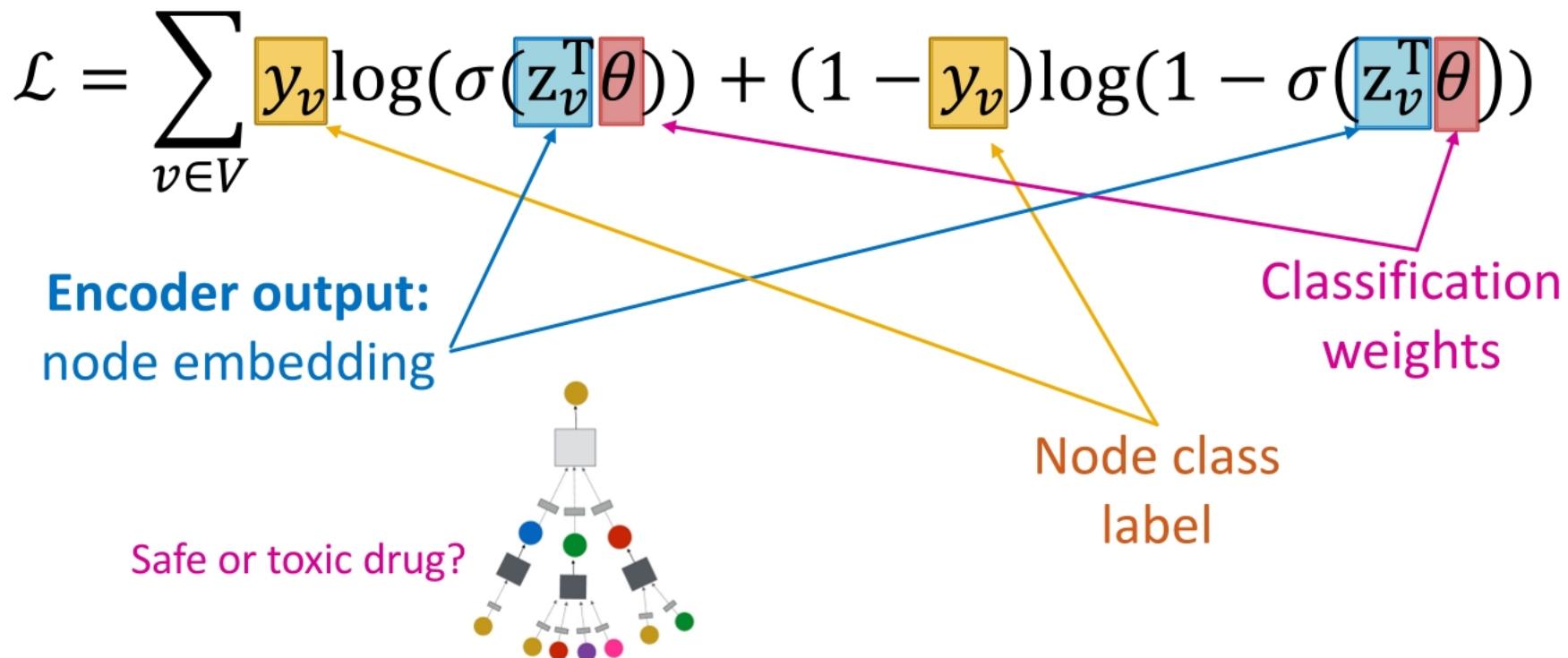


E.g., a drug-drug  
interaction network

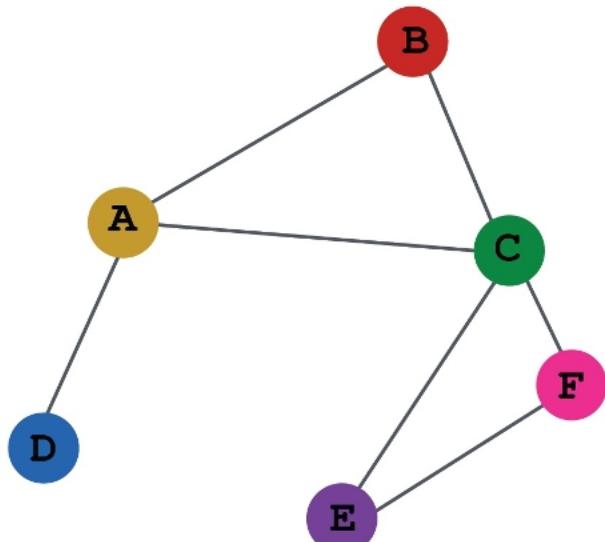
# Supervised Training

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

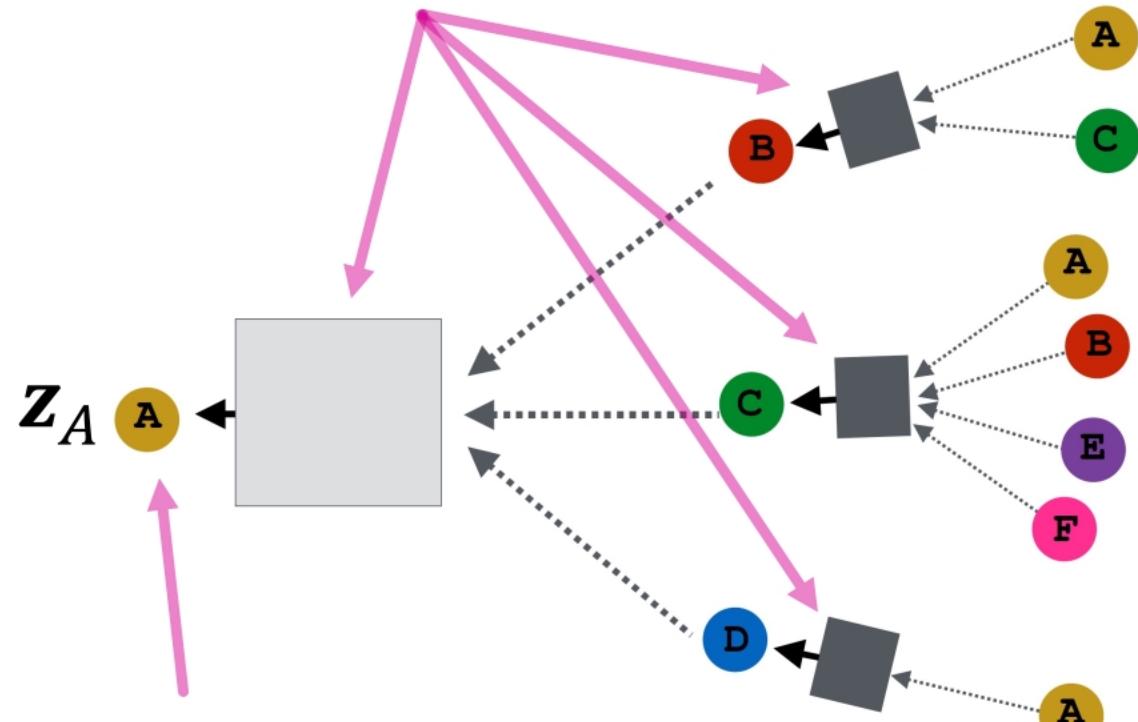
- Use cross entropy loss (Slide 16)



# Model Design: Overview

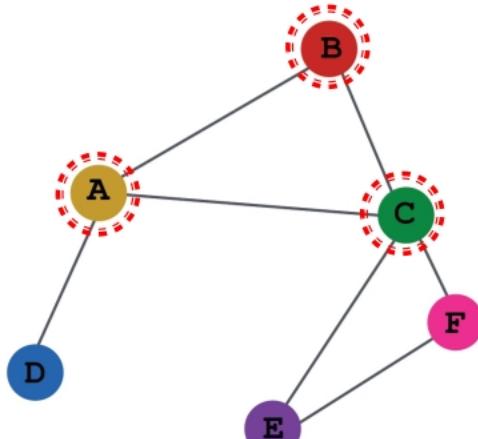


(1) Define a neighborhood aggregation function



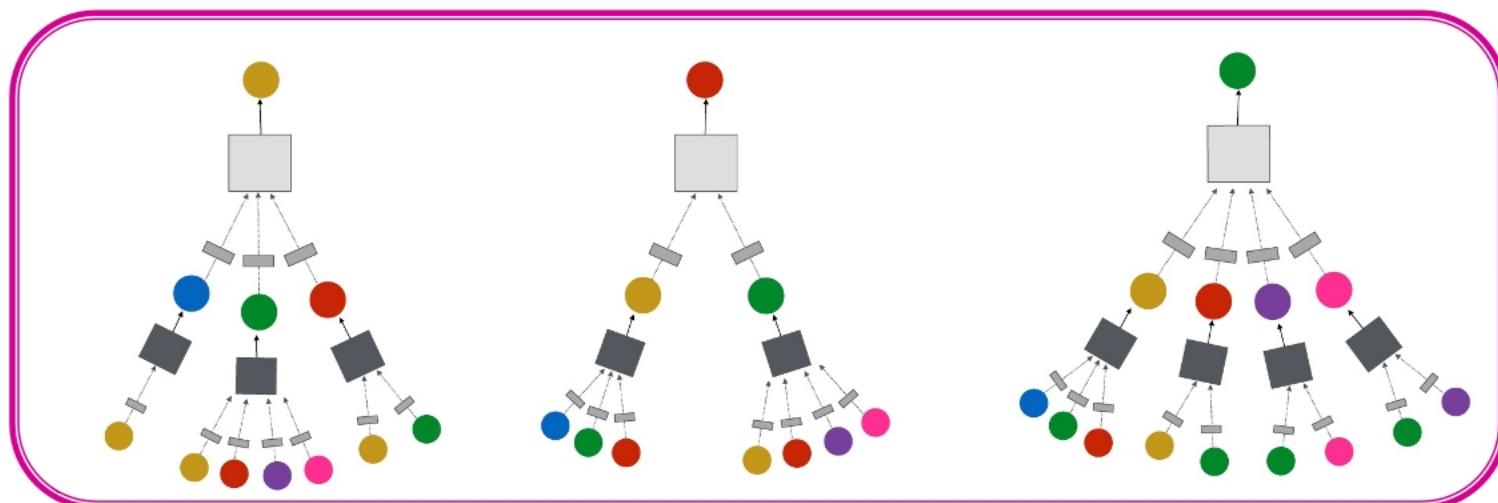
(2) Define a loss function on the embeddings

# Model Design: Overview

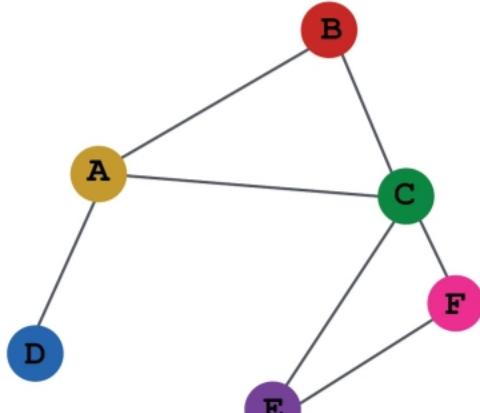


INPUT GRAPH

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



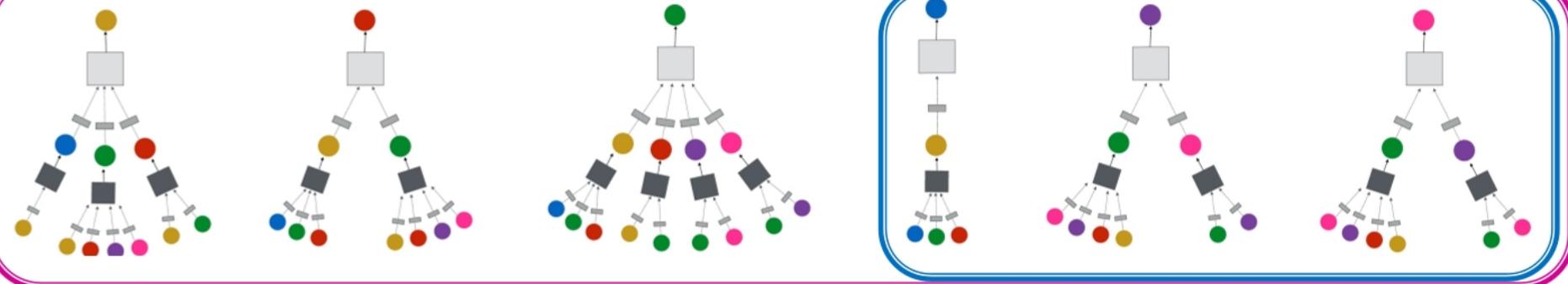
# Model Design: Overview



INPUT GRAPH

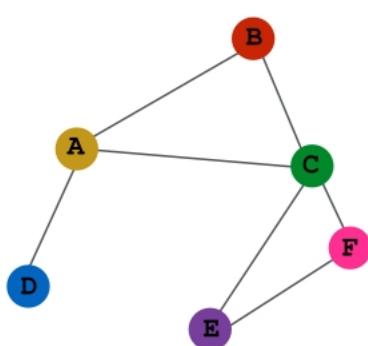
(4) Generate embeddings  
for nodes as needed

Even for nodes we never  
trained on!

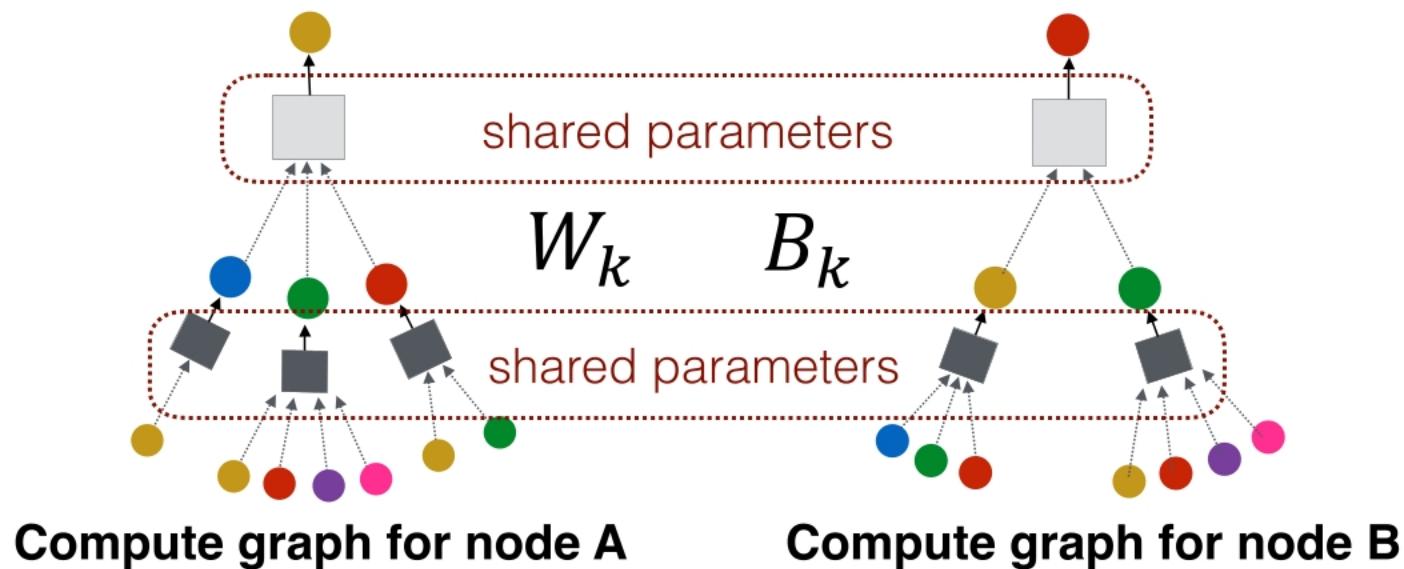


# Inductive Capability

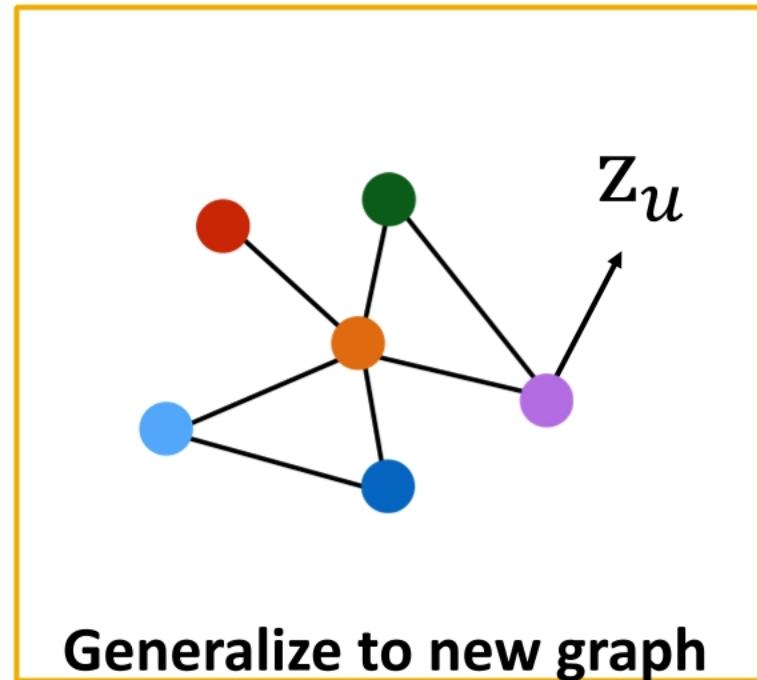
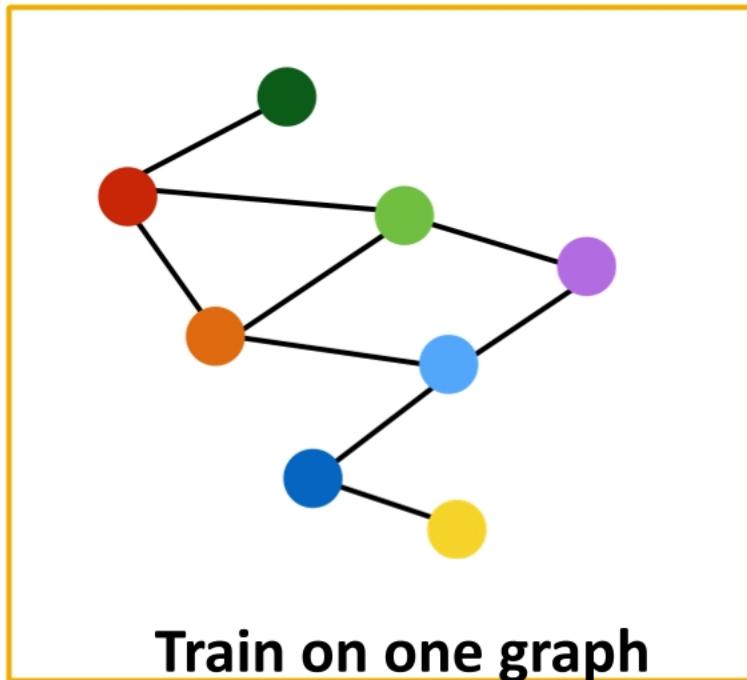
- The same aggregation parameters are shared for all nodes:
  - The number of model parameters is sublinear in  $|V|$  and we can **generalize to unseen nodes!**



INPUT GRAPH



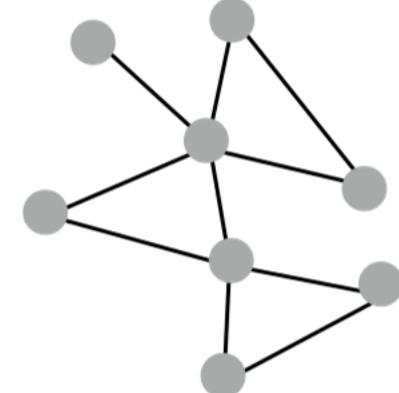
# Inductive Capability: New Graphs



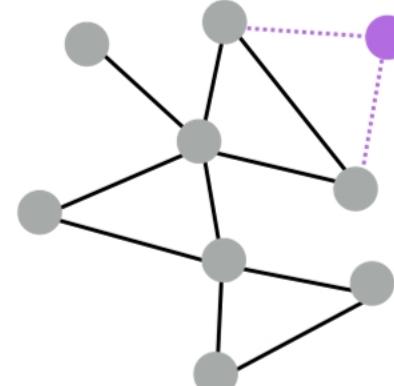
Inductive node embedding → Generalize to entirely unseen graphs

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

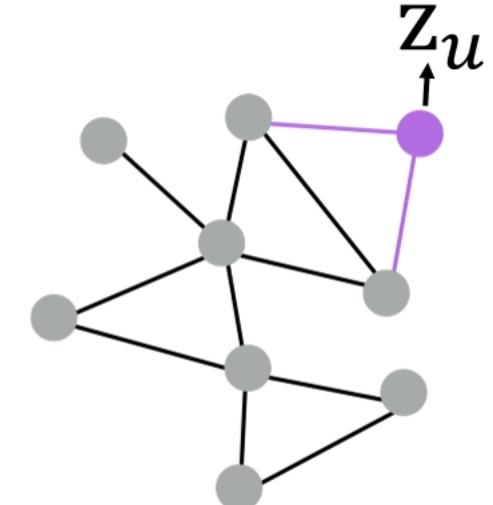
# Inductive Capability: New Nodes



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”

# Outline of Today's Lecture

1. Basics of deep learning



2. Deep learning for graphs



3. Graph Convolutional Networks



4. GNNs subsume CNNs and  
Transformers

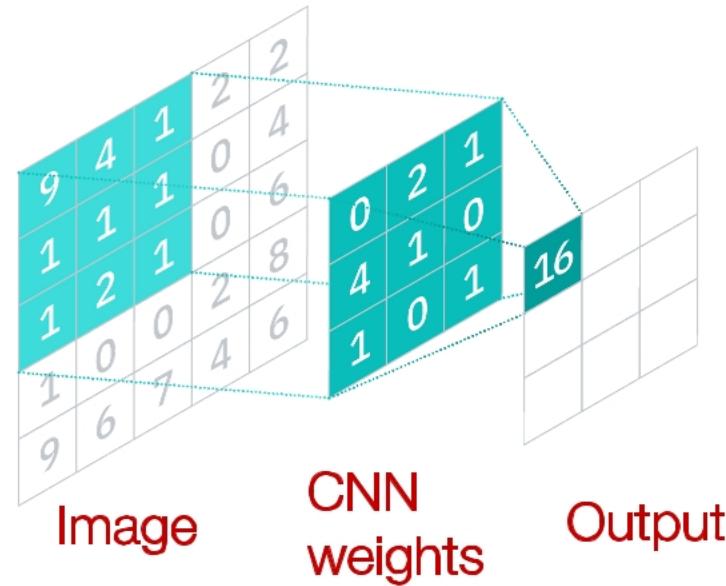
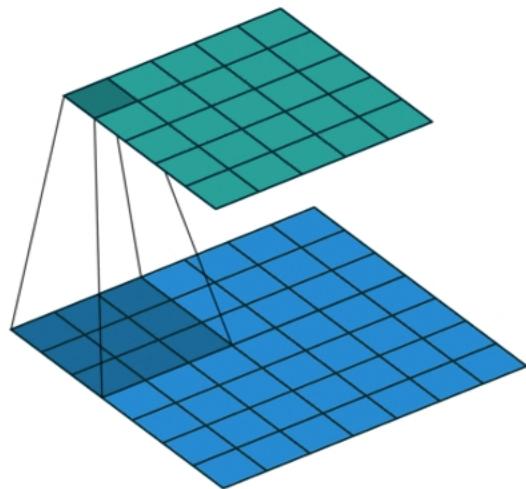


# Architecture Comparison

- 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:

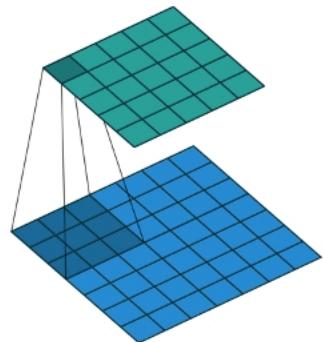


$$\text{CNN formulation: } h_v^{(l+1)} = \sigma(\sum_{u \in N(v) \cup \{v\}} W_l^u h_u^{(l)}), \quad \forall l \in \{0, \dots, L-1\}$$

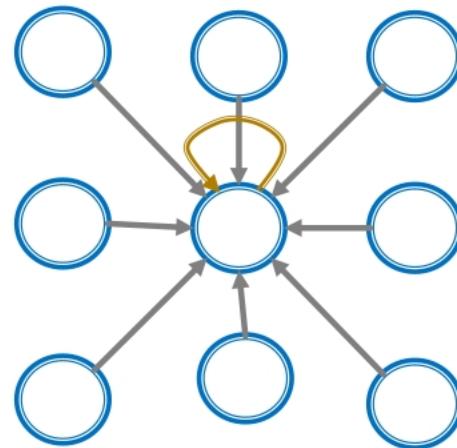
**$N(v)$  represents the 8 neighbor pixels of  $v$ .**

# GNN vs. CNN

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



Image

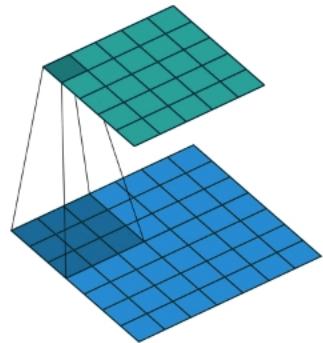


Graph

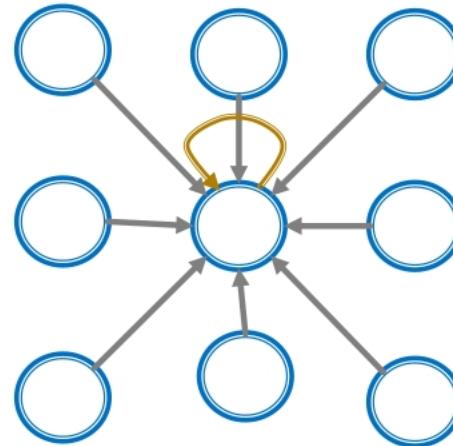
- GNN formulation (previous slide):  $h_v^{(l+1)} = \sigma(\mathbf{W}_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)})$ ,  $\forall l \in \{0, \dots, L-1\}$
- CNN formulation:
  - if we rewrite:
$$h_v^{(l+1)} = \sigma(\sum_{u \in N(v) \cup \{v\}} W_l^u h_u^{(l)})$$
$$h_v^{(l+1)} = \sigma(\sum_{u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

# 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  $W_l^u$  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.

**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:  $\{(-1, -1), (-1, 0), (-1, 1), \dots, (1, 1)\}$

# GNN vs. CNN

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

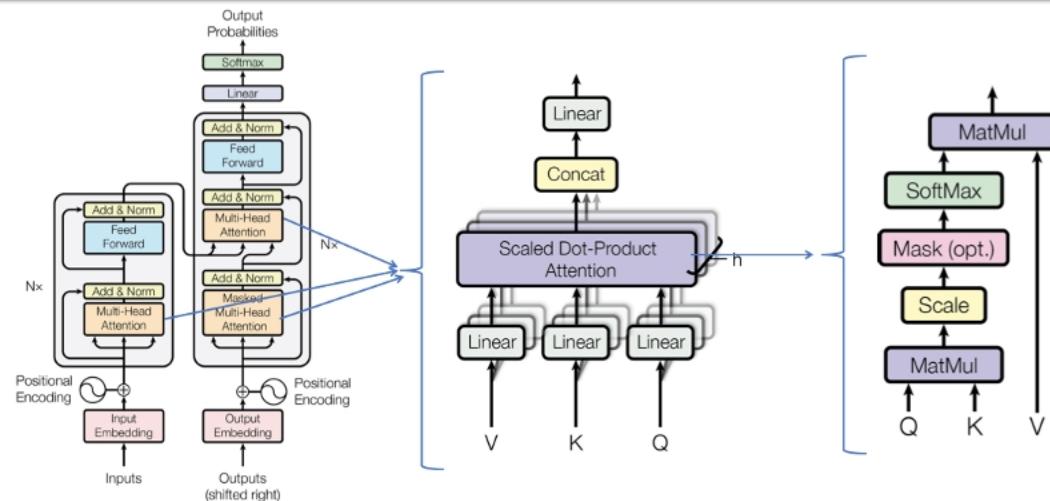


- CNN can be seen as a special GNN with fixed neighbor size and ordering.
- CNN is not permutation equivariant.
  - Switching the order of pixels will leads to different outputs.

**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:  $\{(-1, -1), (-1, 0), (-1, 1), \dots, (1, 1)\}$

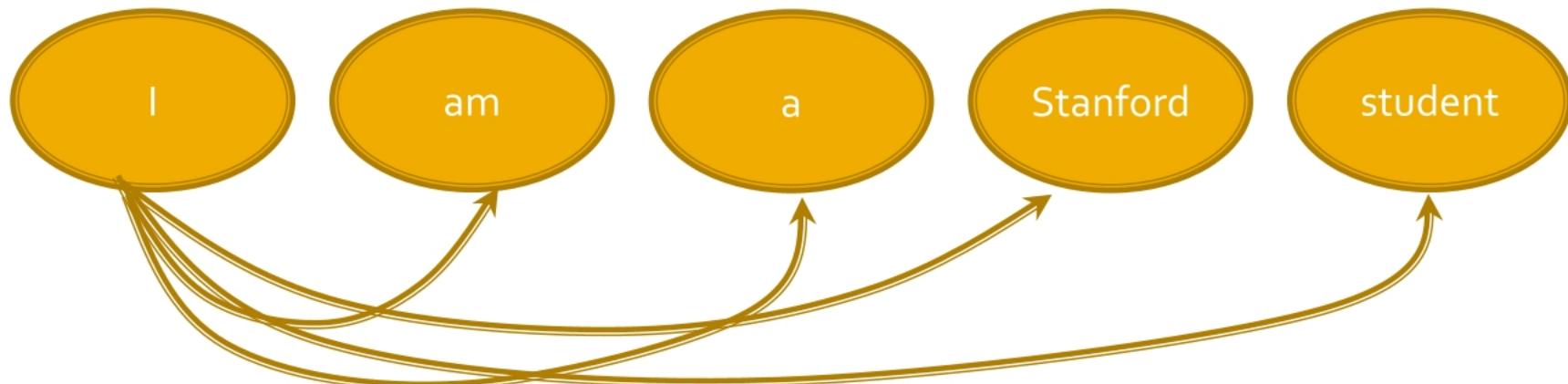
# Transformer

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.



# 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**.

