

# **Stanford CS224W:** **Graph Neural Networks**

CS224W: Machine Learning with Graphs

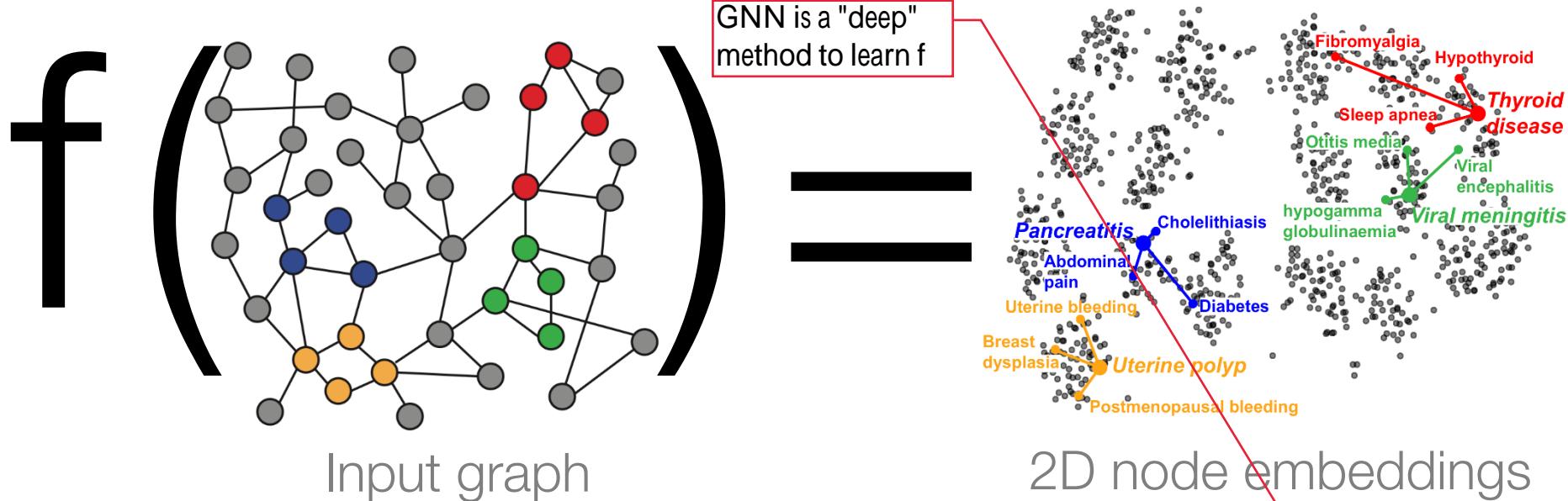
Jure Leskovec, Stanford University

<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

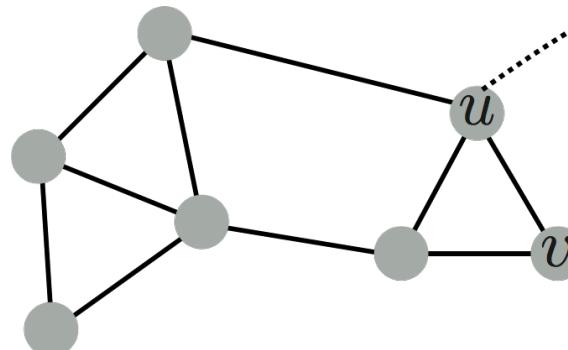


How to learn mapping function  $f$ ?

# Recap: Node Embeddings

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

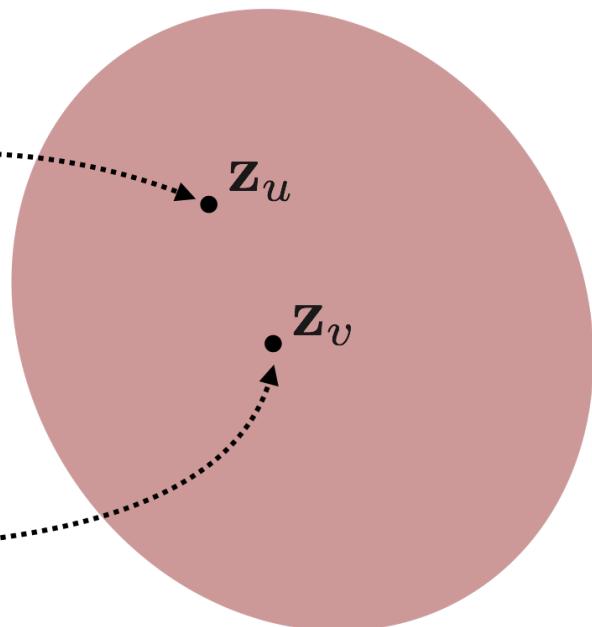
Need to define!



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

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

*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

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

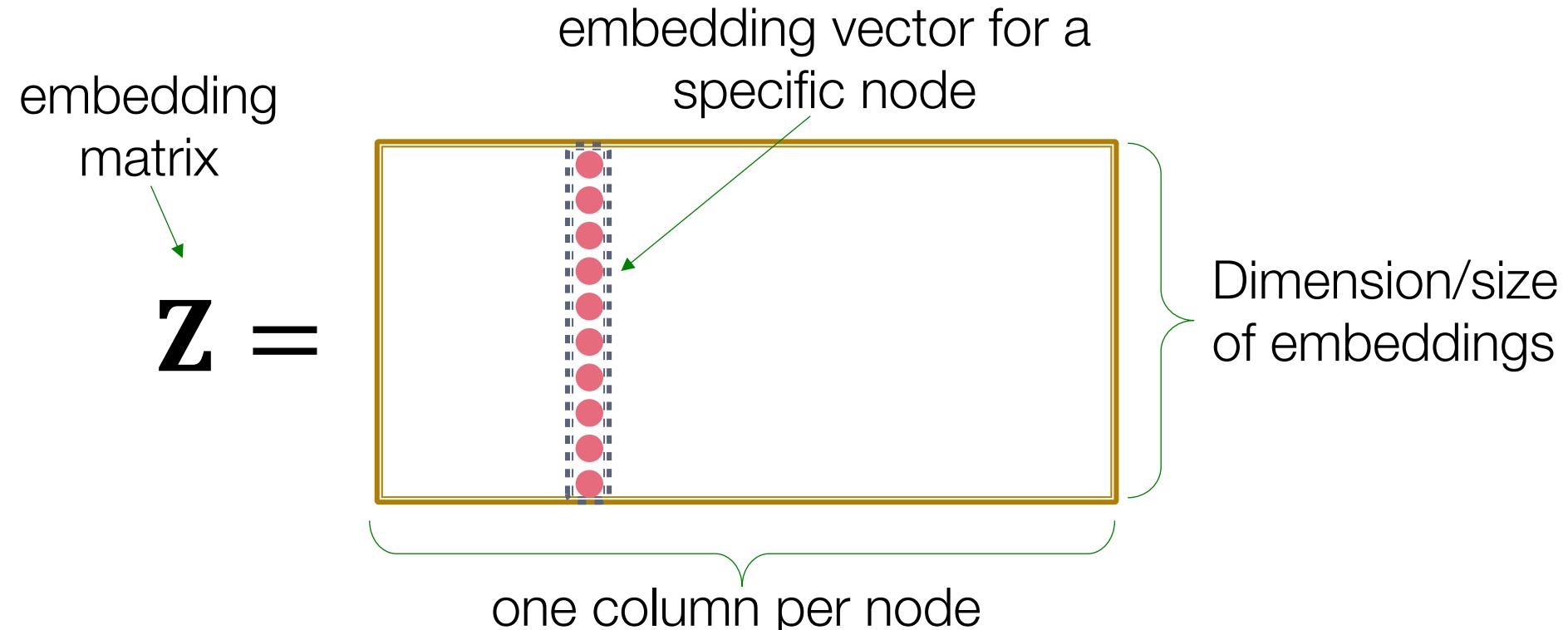
Similarity of  $u$  and  $v$  in  
the original network

**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:**
    - Many graphs have features that we can and should leverage

# Today: Deep Graph Encoders

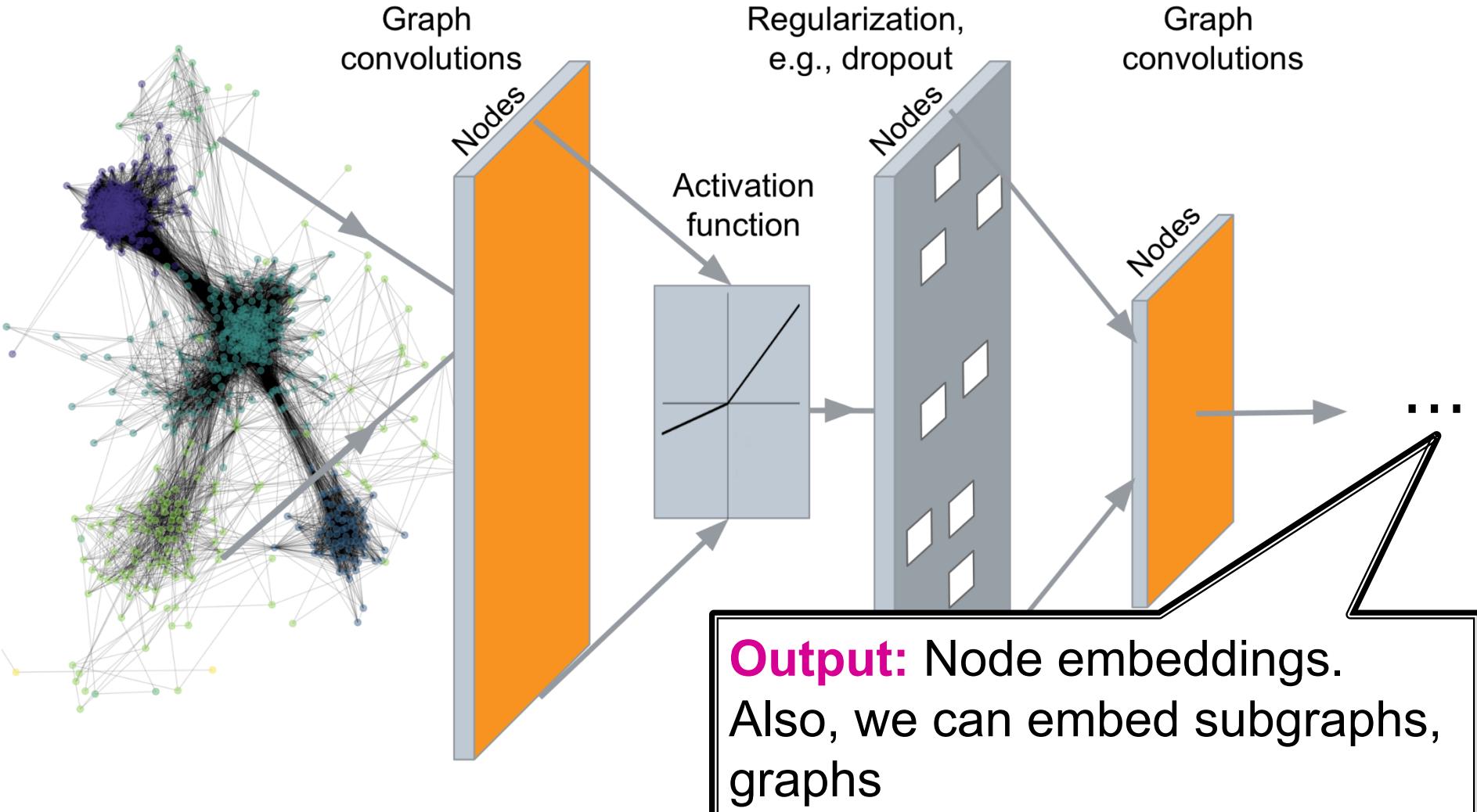
- **Today:** We will now discuss deep methods based on **graph neural networks (GNNs)**:

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

**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

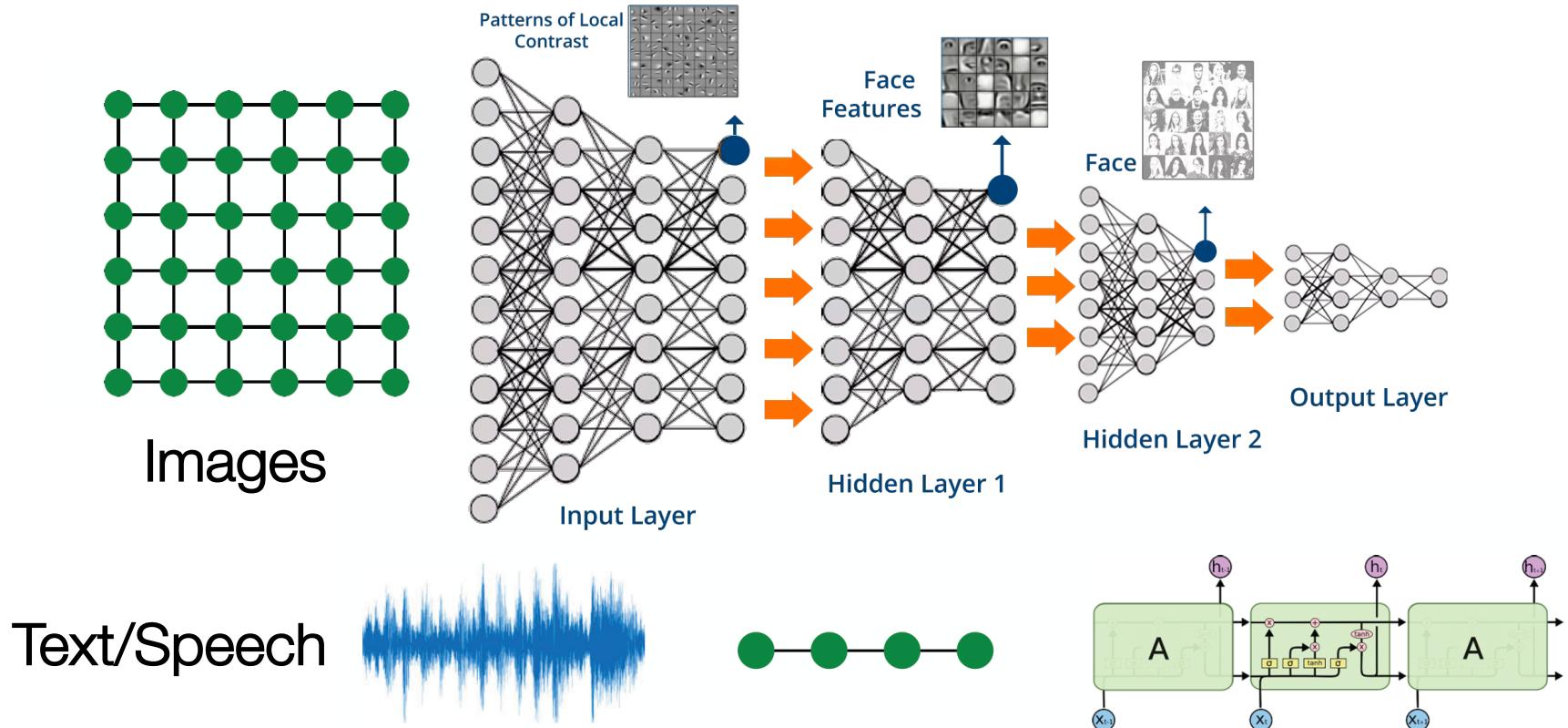


# Tasks on Networks

## 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

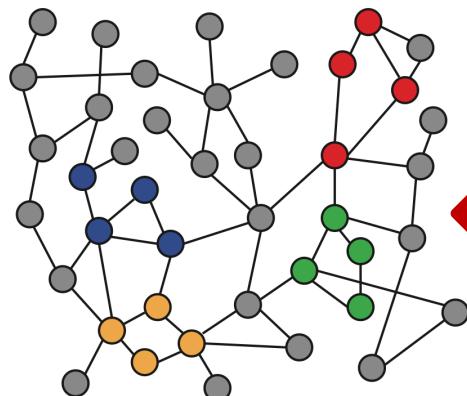


Modern deep learning toolbox is designed  
for simple sequences & grids

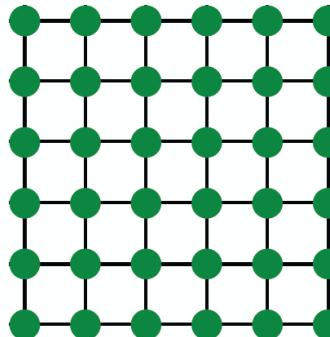
# Why is it Hard?

**But networks are far more complex!**

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



Networks



Images



Text

- 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 and  
GraphSAGE**

# 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  $\mathbf{x}$ , 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

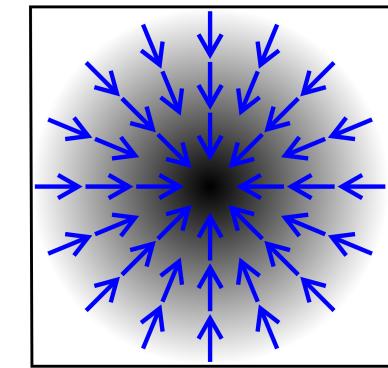
- One common loss for classification: cross entropy (CE)
- Label  $y$  is a categorical vector (**one-hot encoding**)
  - e.g.  $y = \begin{array}{c|c|c|c|c} 0 & 0 & 1 & 0 & 0 \end{array}$   $y$  is of class "3"
- $f(x) = \text{Softmax}(g(x))$ 
  - Recall from lecture 3:  $f(x)_i = \frac{e^{g(x)_i}}{\sum_{j=1}^C e^{g(x)_j}}$ ,  
where  $C$  is the number of classes.
    - e.g.  $f(x) = \begin{array}{c|c|c|c|c} 0.1 & 0.3 & 0.4 & 0.1 & 0.1 \end{array}$

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

# Machine Learning as Optimization

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

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



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

- $\Theta_1, \Theta_2 \dots$  : 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**

# 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:**  $\mathbf{0}$  gradient
  - 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, the function  $f$  can be very complex

- To start simple, consider linear function

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

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

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

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

$$\nabla_{\mathbf{W}} f = \mathbf{W}^T \quad \text{Jacobian matrix of } f$$

# Back-propagation

- How about a more complex function:

$$\textcolor{magenta}{f}(\textcolor{blue}{x}) = W_2(W_1 \textcolor{blue}{x}), \quad \Theta = \{W_1, W_2\}$$

- Recall chain rule:

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

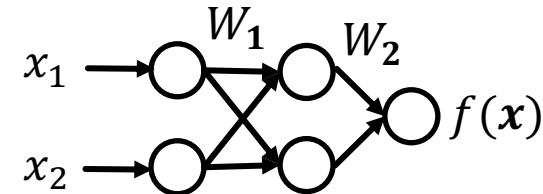
In other words:  
 $\textcolor{magenta}{f}(\textcolor{blue}{x}) = W_2(W_1 \textcolor{blue}{x})$   
 $h(x) = W_1 \textcolor{blue}{x}$   
 $g(z) = W_2 z$

- E.g.  $\nabla_{\textcolor{blue}{x}} \textcolor{magenta}{f} = \frac{\partial \textcolor{magenta}{f}}{\partial (W_1 \textcolor{blue}{x})} \cdot \frac{\partial (W_1 \textcolor{blue}{x})}{\partial \textcolor{blue}{x}}$

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

# Back-propagation Example (1)

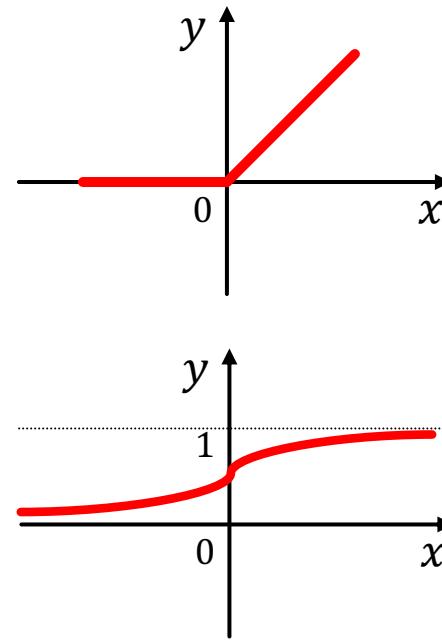
- **Example:** Simple 2-layer linear network
- $f(\mathbf{x}) = g(h(x)) = W_2(W_1 \mathbf{x})$
- $\mathcal{L} = \sum_{(x,y) \in \mathcal{B}} \left\| (y, -f(x)) \right\|_2$  sums the L2 loss in a minibatch  $\mathcal{B}$
- **Hidden layer:** intermediate representation for input  $\mathbf{x}$ 
  - Here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



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

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

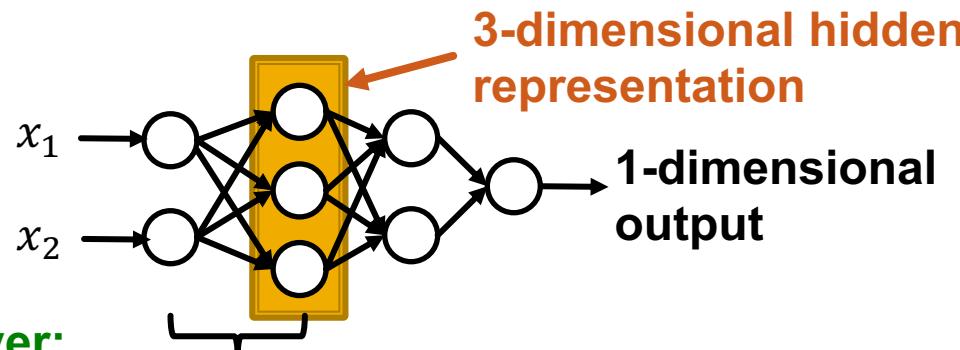


# Multi-layer Perceptron (MLP)

- 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  $x_2$



Every layer:  
Linear transformation +  
non-linearity

# 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  $\nabla_{\Theta} \mathcal{L}$  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 and GraphSAGE**

# Stanford CS224W: Deep Learning for Graphs

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

<http://cs224w.stanford.edu>



# Content

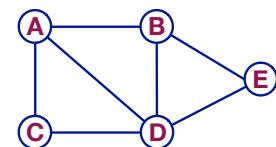
- **Local network neighborhoods:**
  - Describe aggregation strategies
  - Define computation graphs
- **Stacking multiple layers:**
  - Describe the model, parameters, training
  - How to fit the model?
  - Simple example for unsupervised and supervised training

# Setup

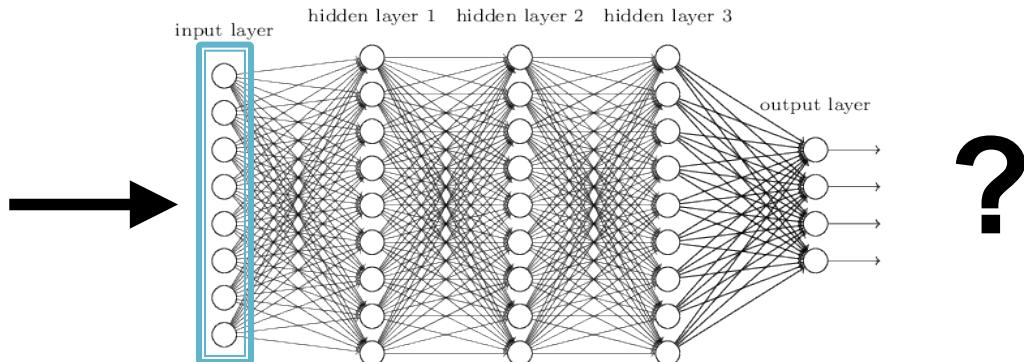
- Assume we have a graph  $G$ :
  - $V$  is the **vertex set**
  - $A$  is the **adjacency matrix** (assume binary)
  - $X \in \mathbb{R}^{m \times |V|}$  is a matrix of **node features**
  - $v$ : a node in  $V$ ;  $N(v)$ : the set of neighbors of  $v$ .
- **Node features:**
  - Social networks: User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

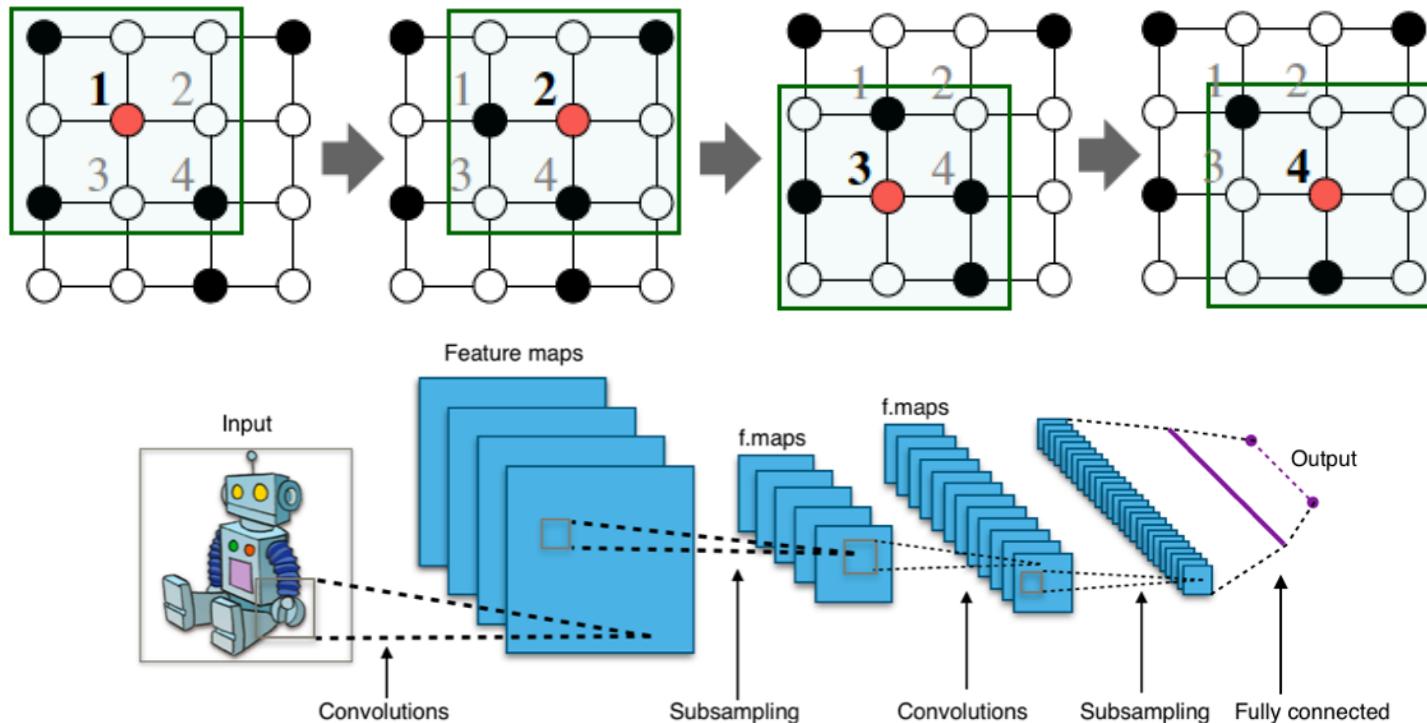


- Issues with this idea:
  - $O(|V|)$  parameters
  - Not applicable to graphs of different sizes
  - Sensitive to node ordering

If node numbers (which is A, which is B) change, the entries of adjacency matrix change, which means the inputs are changed

# Idea: Convolutional Networks

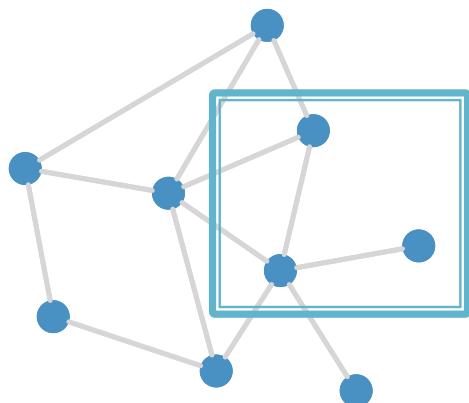
CNN on an image:



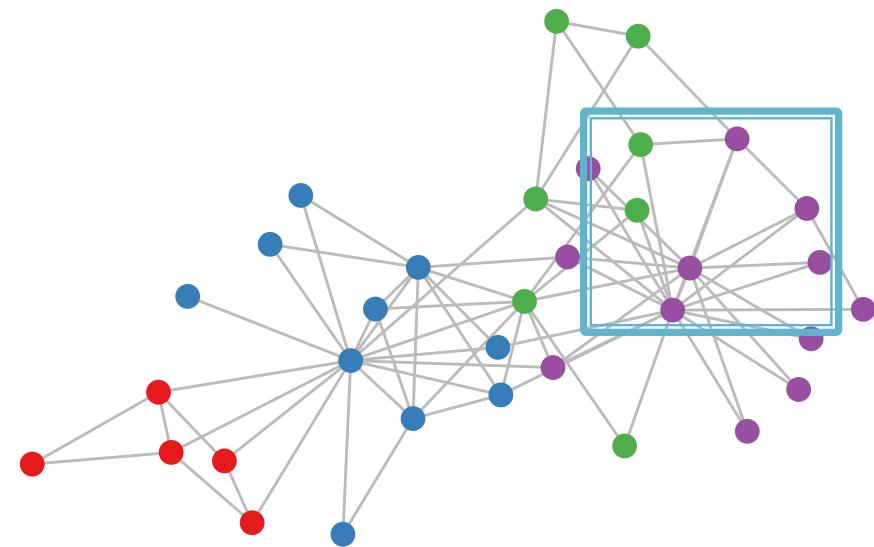
Goal is to generalize convolutions beyond simple lattices  
Leverage node features/attributes (e.g., text, images)

# Real-World Graphs

But our graphs look like this:



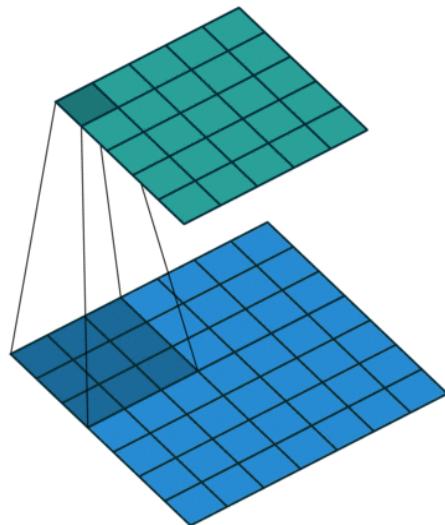
or this:



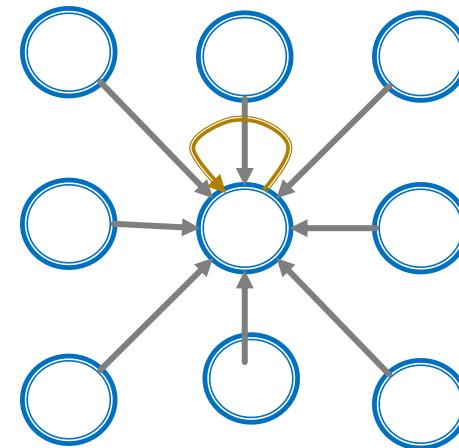
- There is no fixed notion of locality or sliding window on the graph
- Graph is permutation invariant

# From Images to Graphs

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



Image



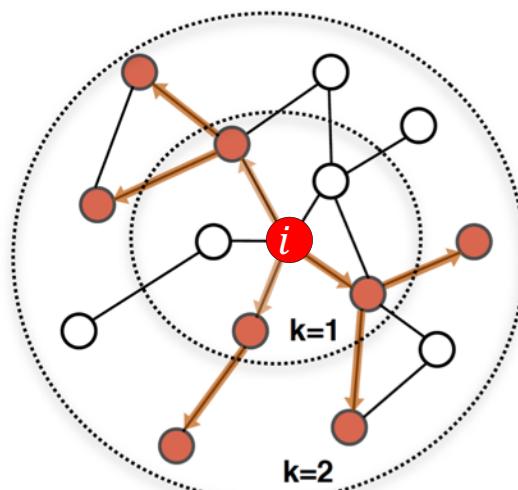
Graph

**Idea:** transform information at the neighbors and combine it:

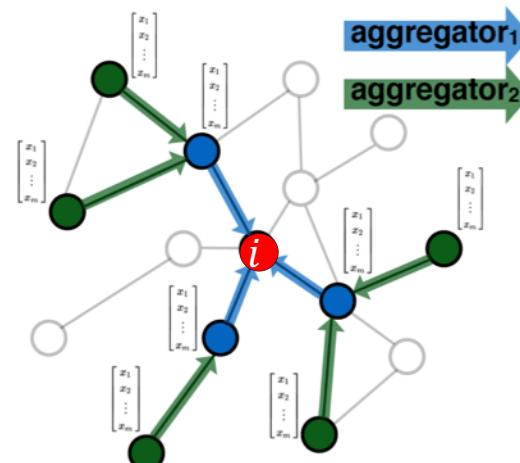
- Transform “messages”  $h_i$  from neighbors:  $W_i h_i$
- Add them up:  $\sum_i W_i h_i$

# Graph Convolutional Networks

Idea: Node's neighborhood defines a computation graph



Determine node computation graph

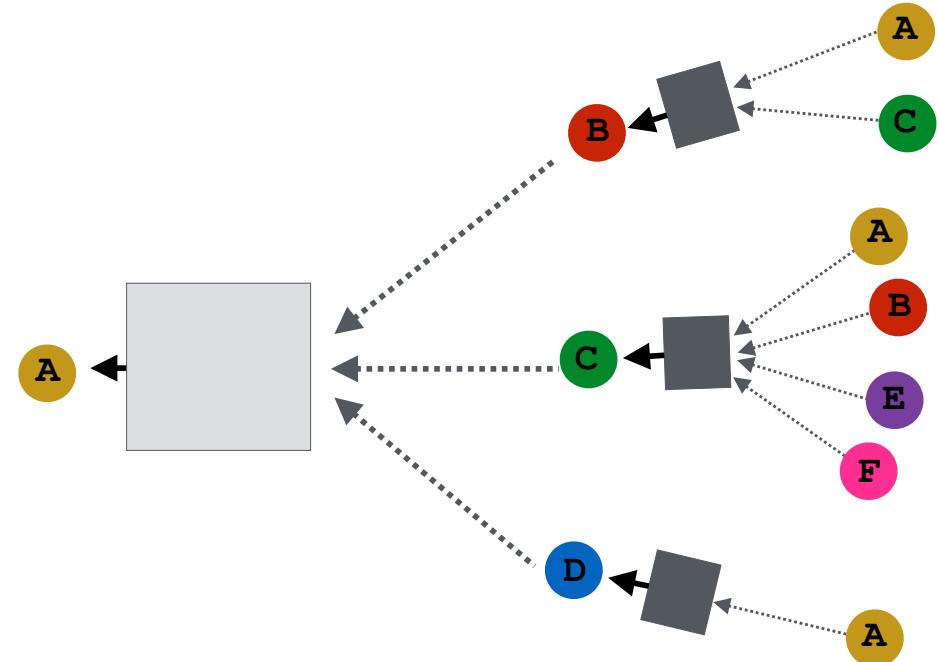
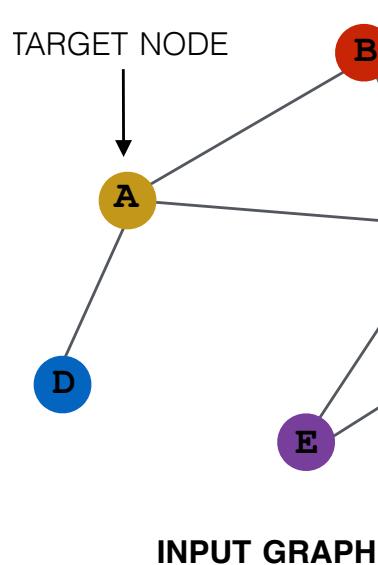


Propagate and transform information

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

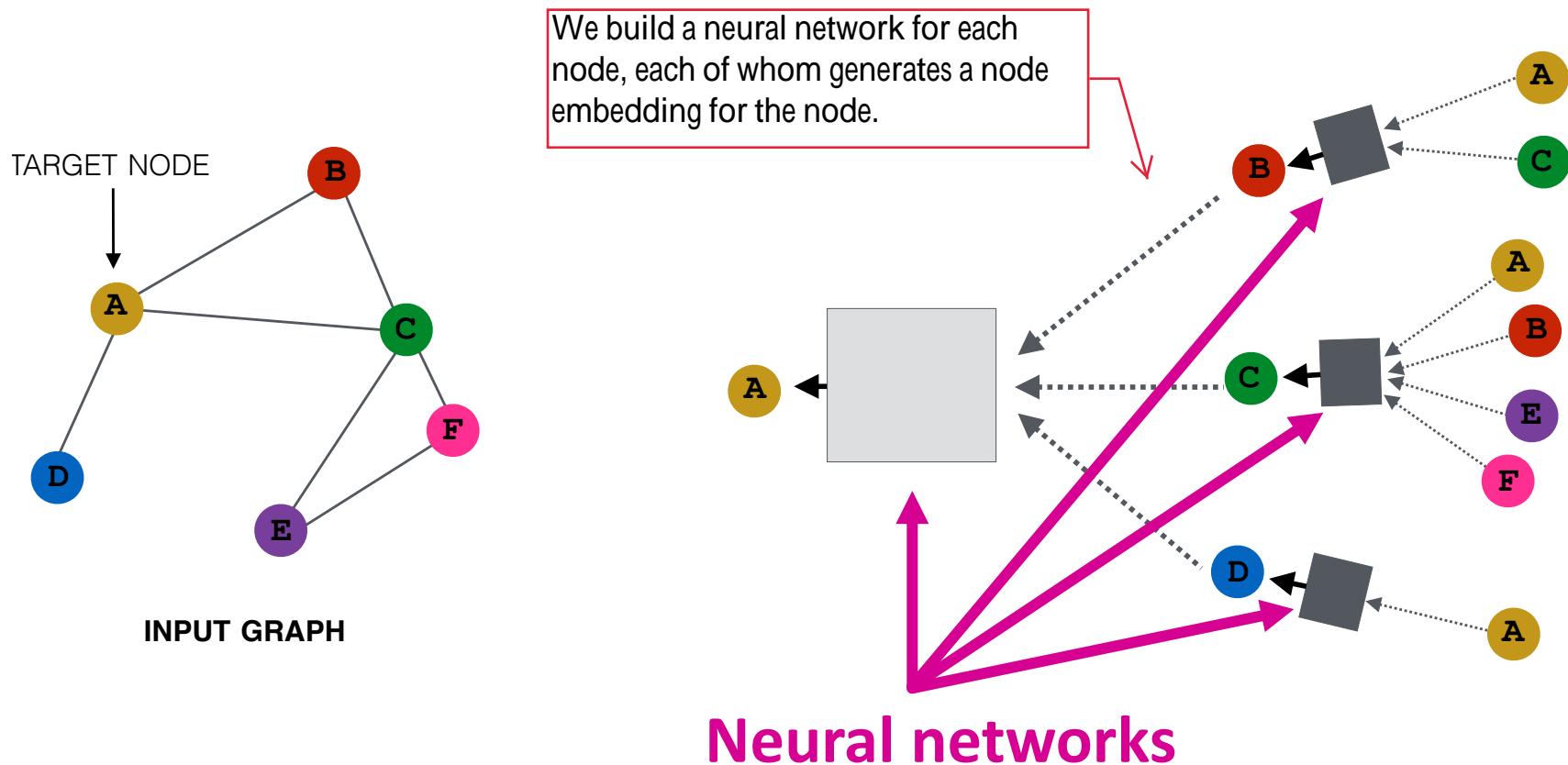
# Idea: Aggregate Neighbors

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



# Idea: Aggregate Neighbors

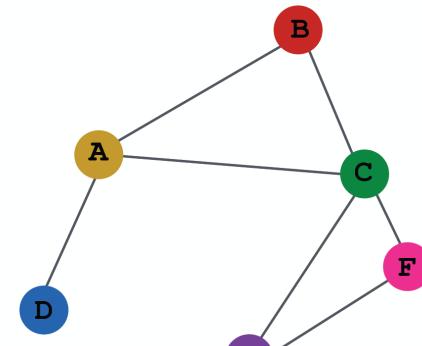
- **Intuition:** Nodes aggregate information from their neighbors using neural networks



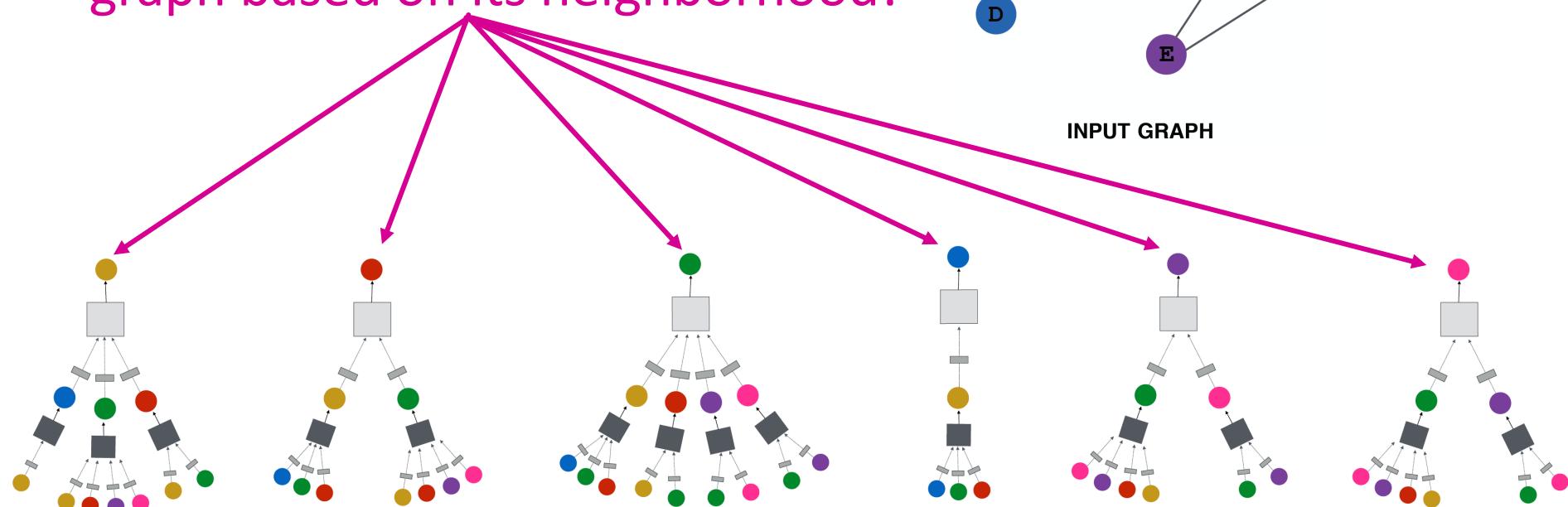
# Idea: Aggregate Neighbors

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

Every node defines a computation graph based on its neighborhood!

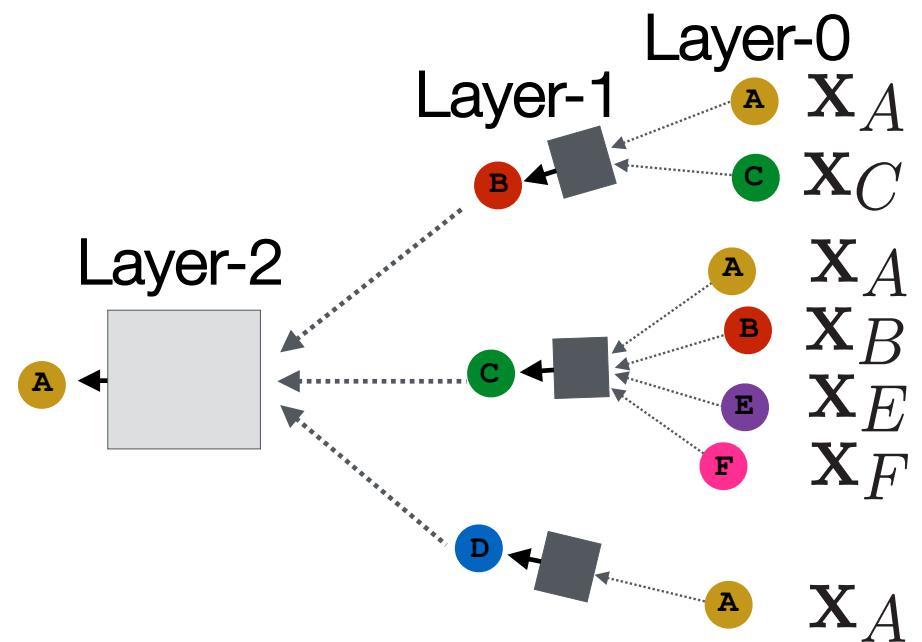
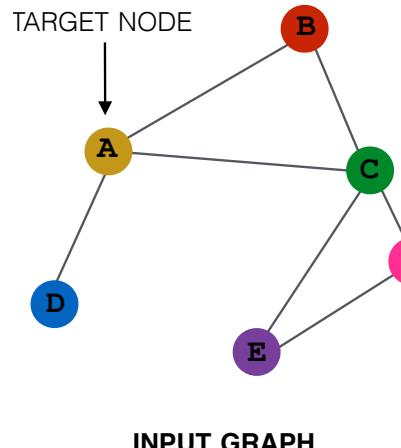


INPUT GRAPH



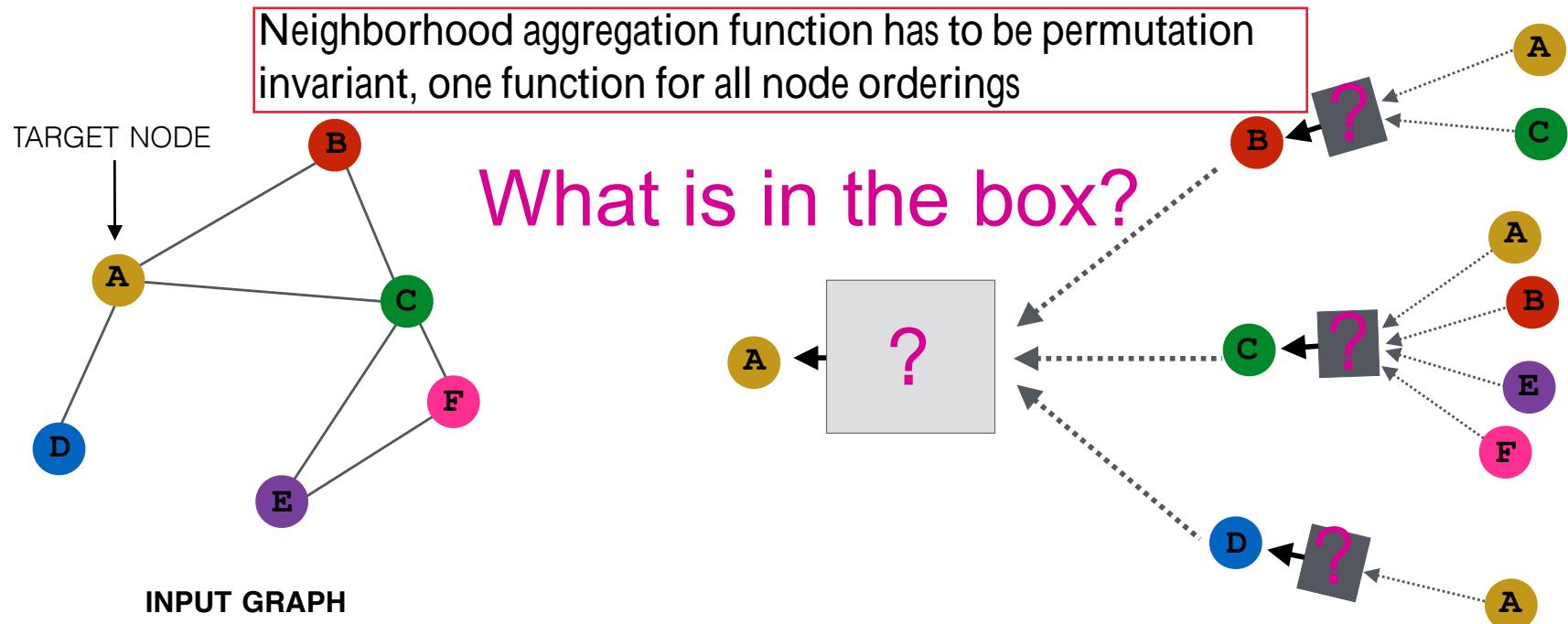
# Deep Model: Many Layers

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



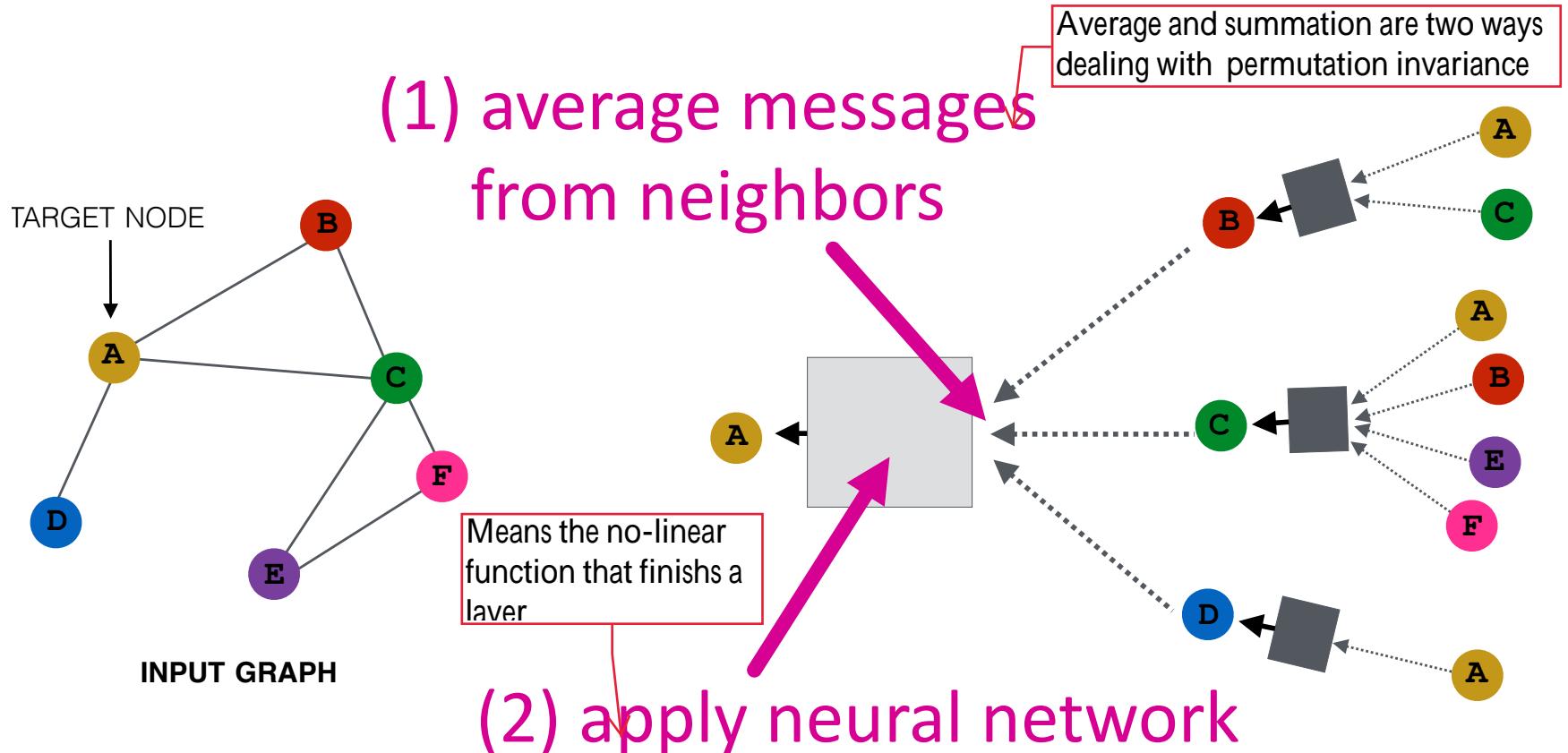
# Neighborhood Aggregation

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



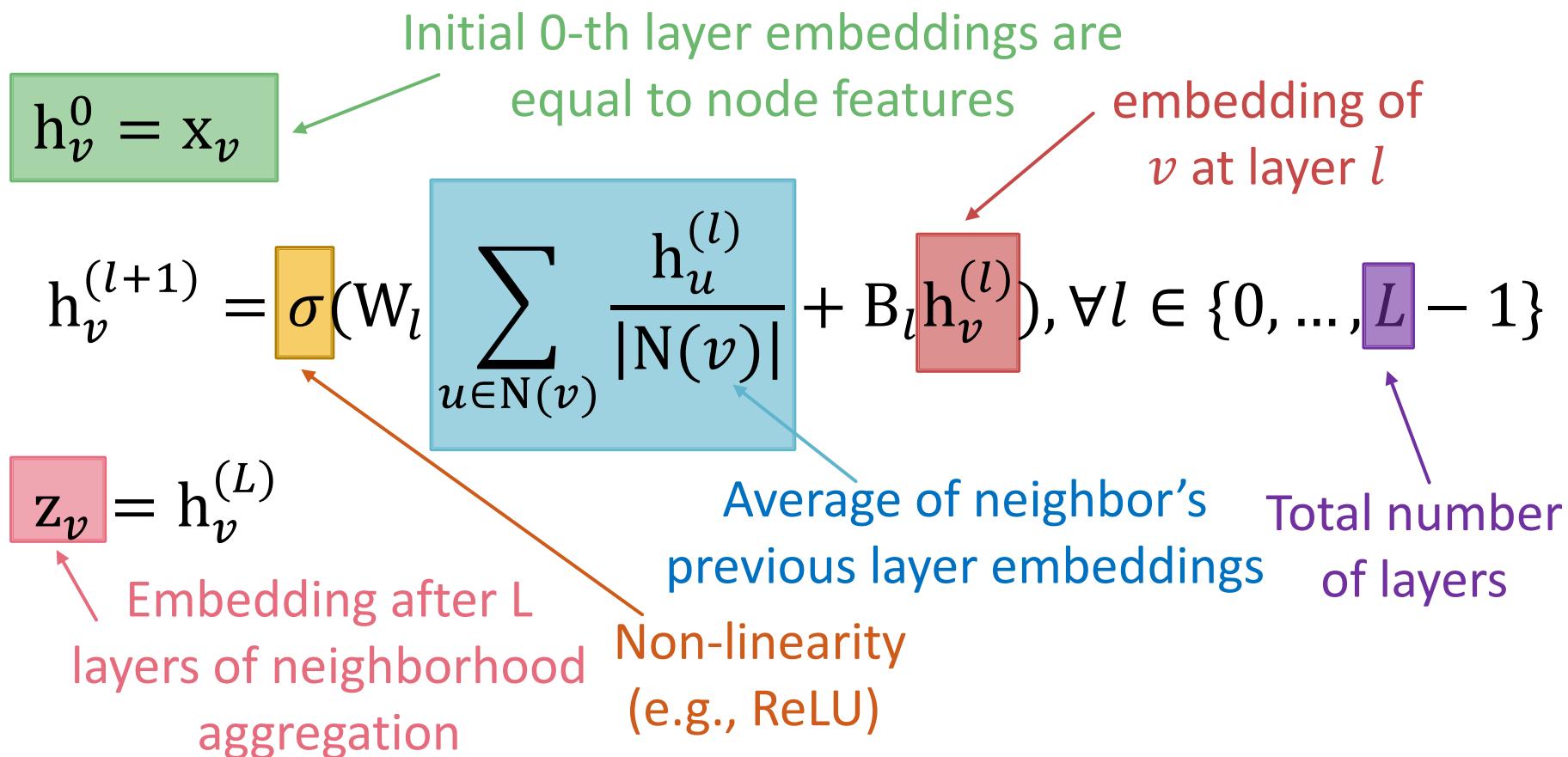
# Neighborhood Aggregation

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



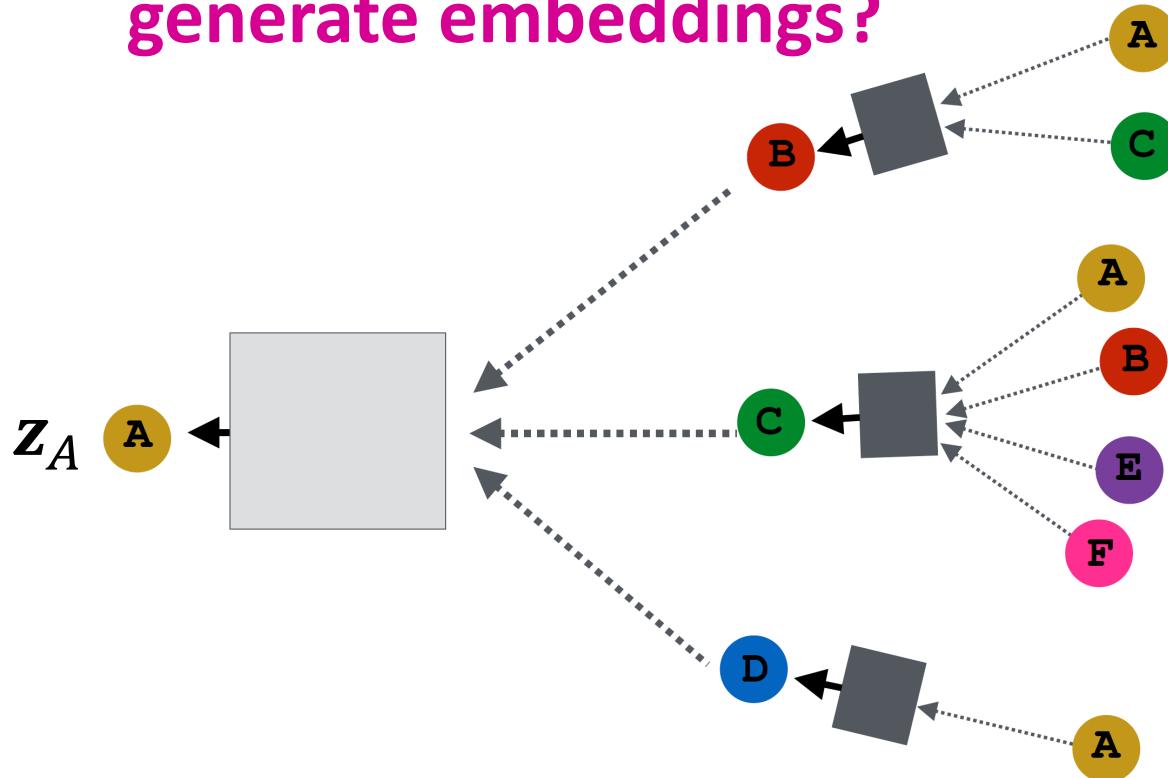
# The Math: Deep Encoder

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



# Training the Model

How do we train the model to generate embeddings?



Need to define a loss function on the embeddings

# Model Parameters

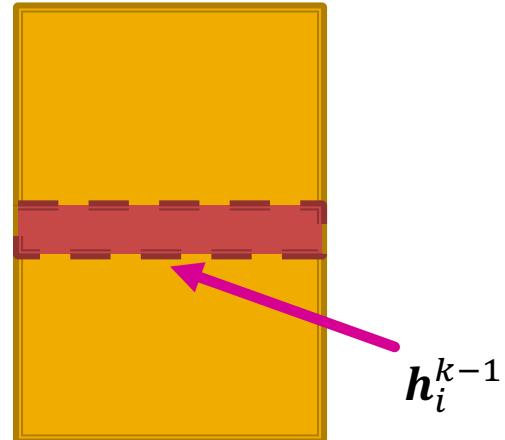
$$\begin{aligned} h_v^{(0)} &= x_v && \text{Trainable weight matrices} \\ &&& (\text{i.e., what we learn}) \\ h_v^{(l+1)} &= \sigma(W_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)}), \forall l \in \{0, \dots, L-1\} && \begin{array}{l} \text{Different layers have different W and B} \\ \text{Nodes in the same layer share the same W and B} \end{array} \\ z_v &= h_v^{(L)} && \text{Final node embedding} \end{aligned}$$

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

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

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

# Matrix Formulation (1)

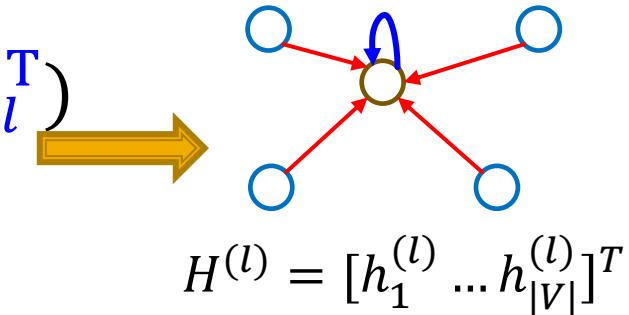
- Many aggregations can be performed efficiently by (sparse) matrix operations
  - Let  $H^{(l)} = [h_1^{(l)} \dots h_{|V|}^{(l)}]^T$
  - Then:  $\sum_{u \in N_v} h_u^{(l)} = A_{v,:} H^{(l)}$
  - Let  $D$  be diagonal matrix where  $D_{v,v} = \text{Deg}(v) = |N(v)|$ 
    - The inverse of  $D$ :  $D^{-1}$  is also diagonal:  
$$D_{v,v}^{-1} = 1/|N(v)|$$
  - Therefore,
- $A_{v,:} H^{(l)}$  means the summation of some features of the neighbors of  $v$
- $D_{v,v}^{-1}$  averages the summation
- 
- $$\sum_{u \in N(v)} \frac{h_u^{(l-1)}}{|N(v)|}$$
- 
- $$H^{(l+1)} = D^{-1} A H^{(l)}$$
- Matrix of hidden embeddings  $H^{k-1}$
- 
- $h_i^{k-1}$
- 12/6/18
- Jure Leskovec, Stanford CS224W: Machine Learning with Graphs, <http://cs224w.stanford.edu>
- 45

# Matrix Formulation (2)

- Re-writing update function in matrix form:

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

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 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

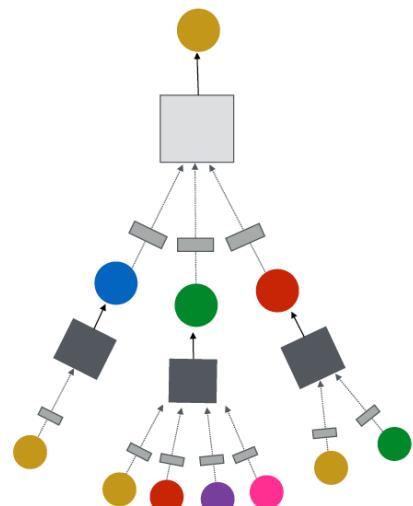
$$\mathcal{L} = \sum_{z_u, z_v} \text{CE}(y_{u,v}, \text{DEC}(z_u, z_v))$$

- 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**

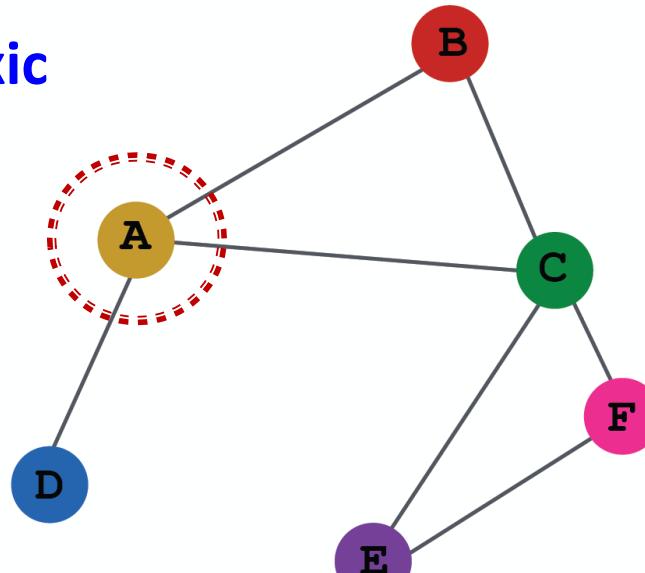
# Supervised Training

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

Safe or toxic  
drug?



Safe or toxic  
drug?



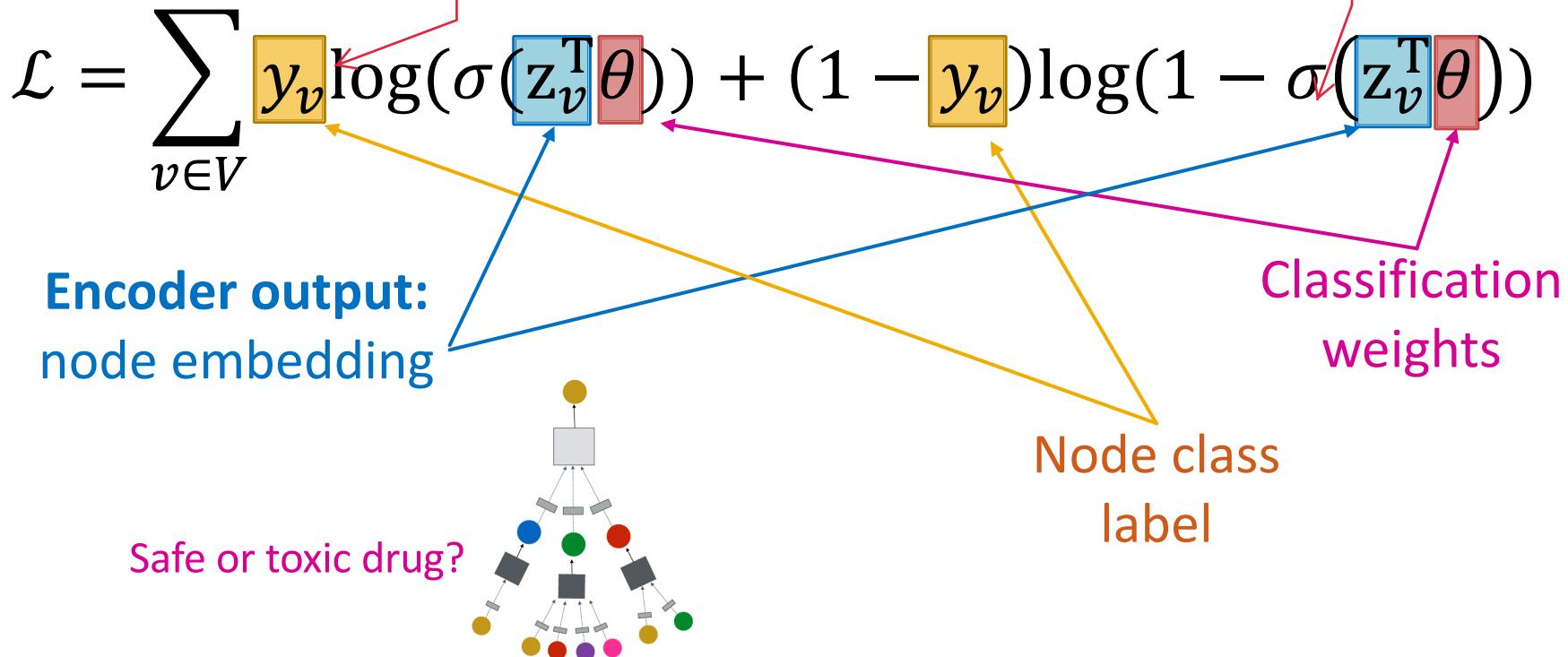
E.g., a drug-drug  
interaction network

# Supervised Training

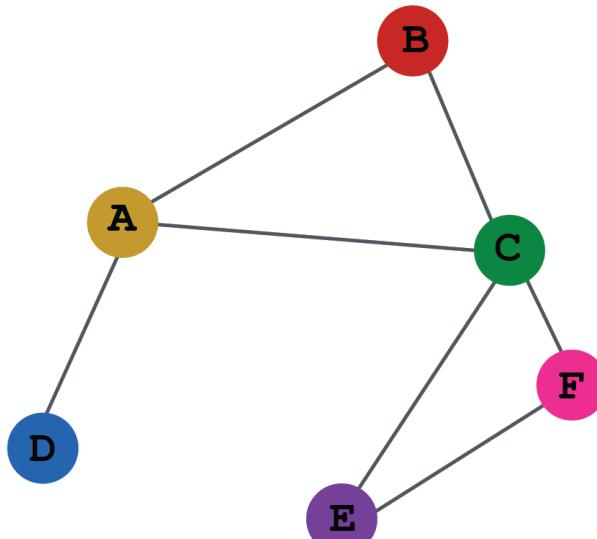
**Directly train** the model for a supervised task  
(e.g., **node classification**)

- Use cross entropy loss (slide 16)

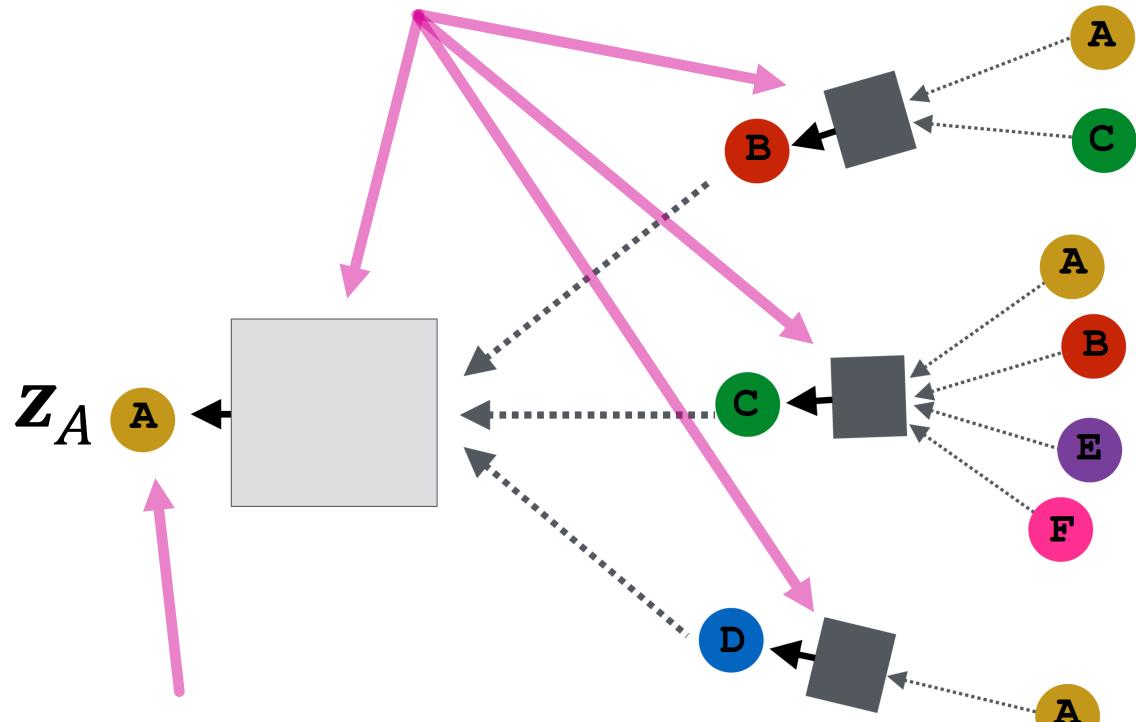
If ground truth is 1 this term survives and we want to  $Z_v^T \theta$  to be as big as possible



# Model Design: Overview

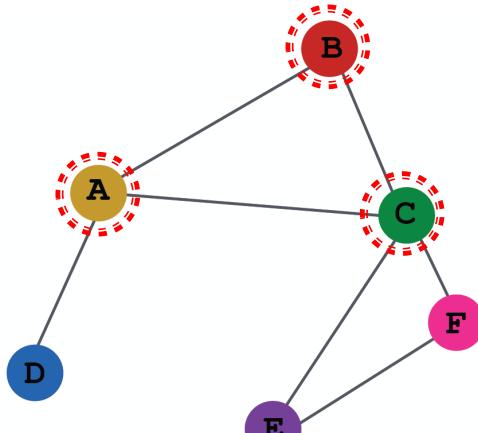


(1) Define a neighborhood aggregation function



(2) Define a loss function on the embeddings

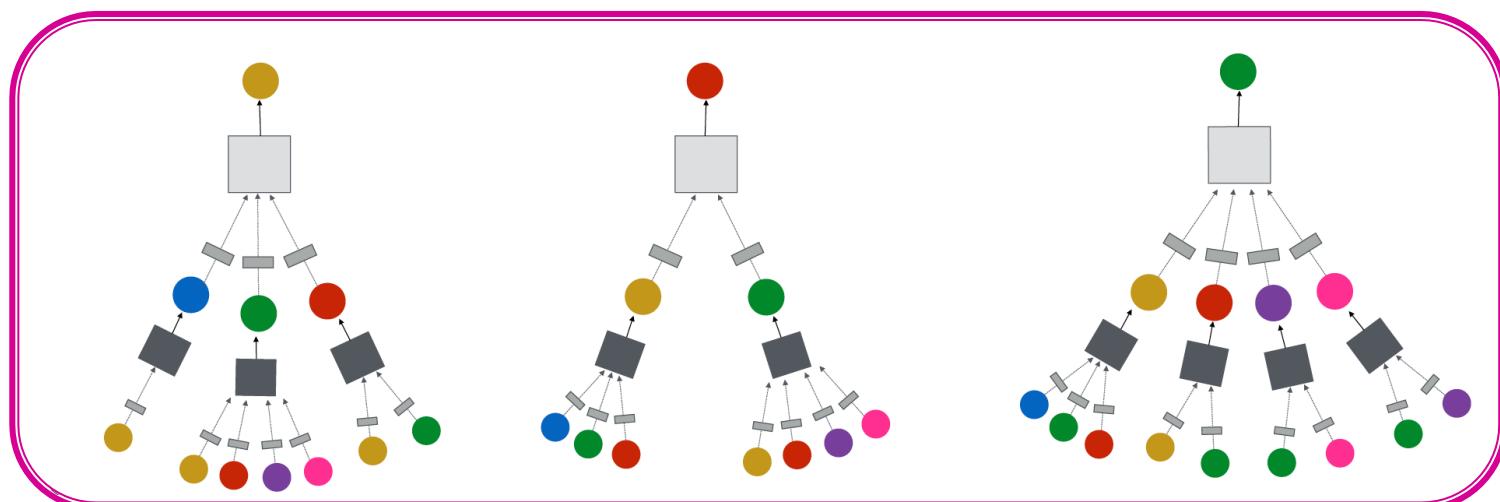
# Model Design: Overview



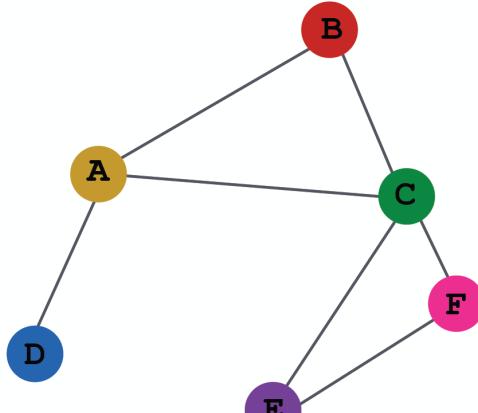
INPUT GRAPH

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

a batch of nodes



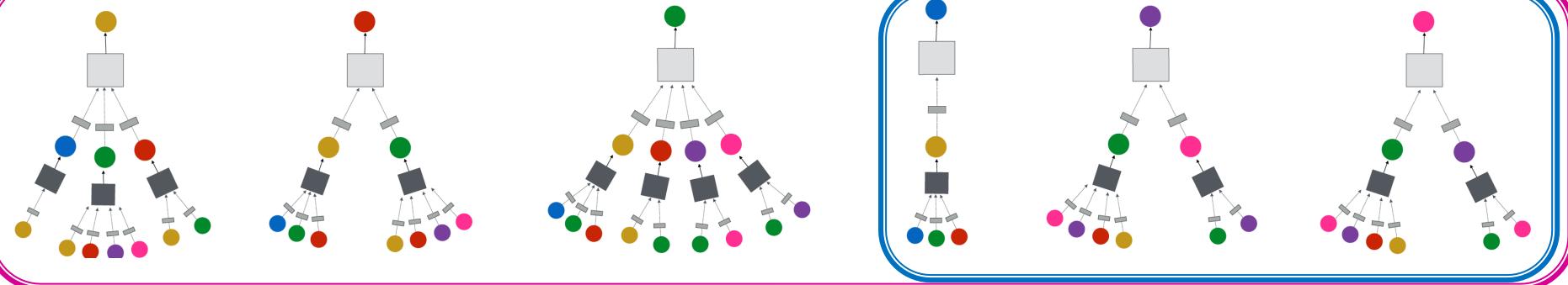
# Model Design: Overview



INPUT GRAPH

(4) Generate embeddings  
for nodes as needed

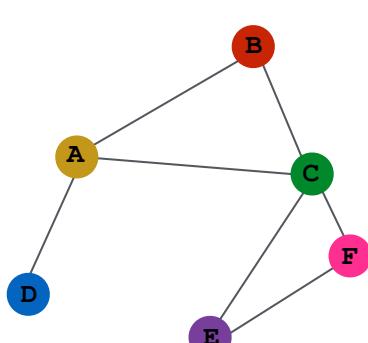
Even for nodes we never  
trained on!



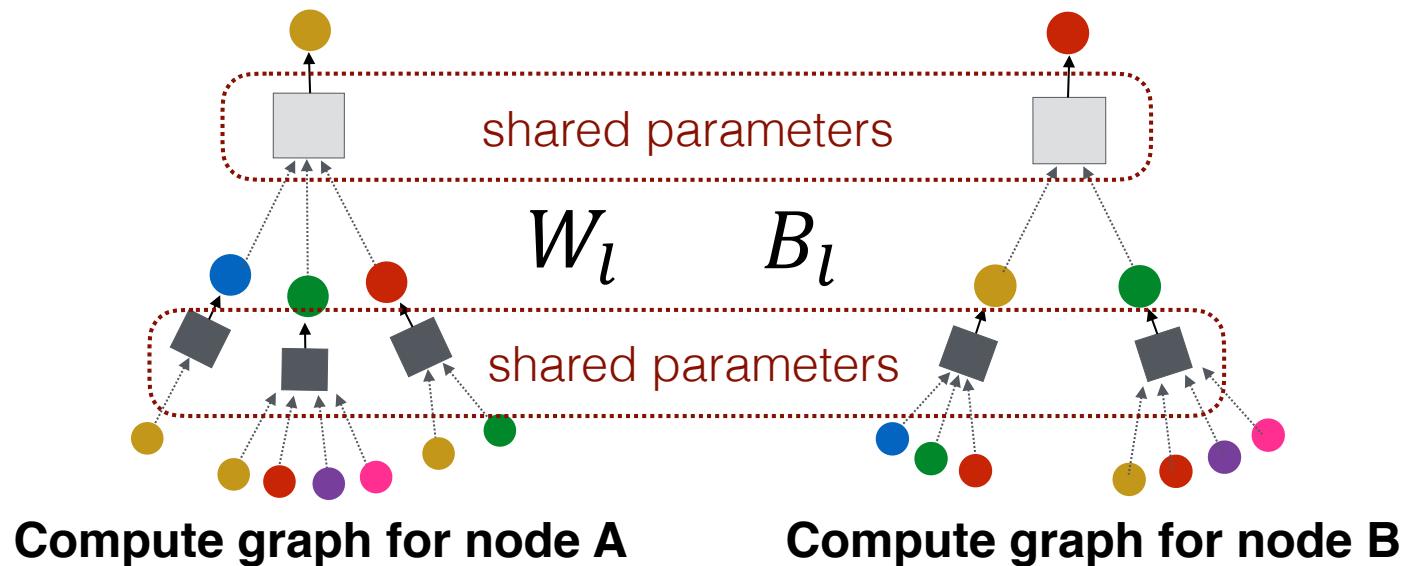
# Inductive Capability

- 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!**

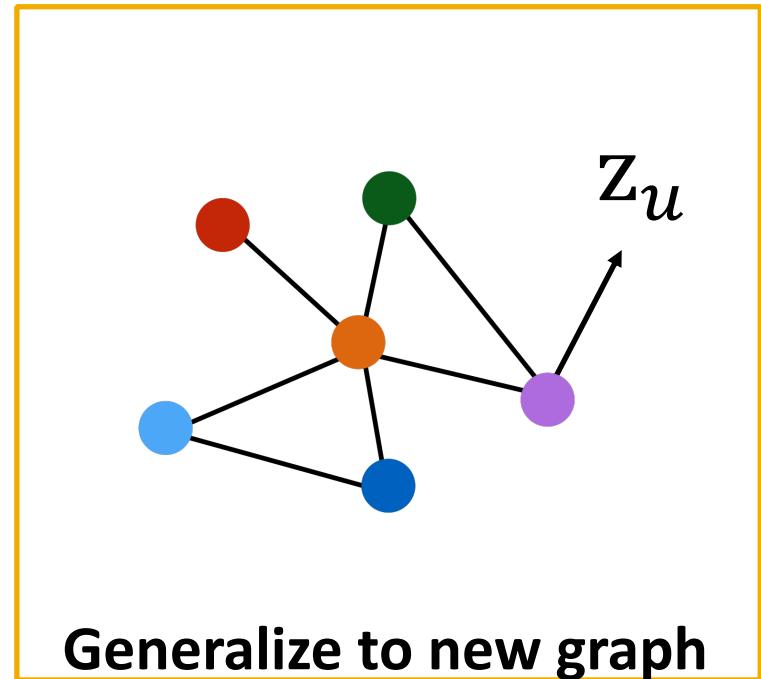
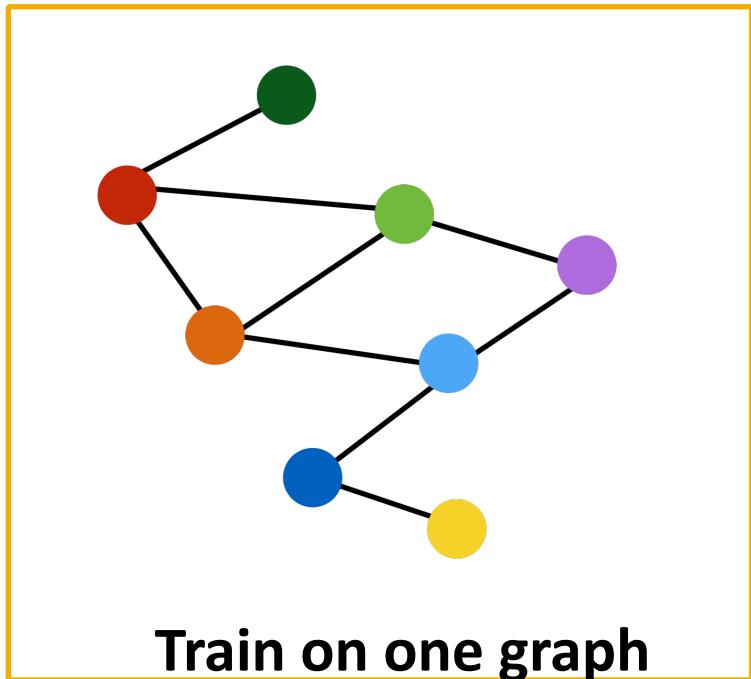
The shared parameters are actually unseen nodes.  
We could use them to generate a new graph.



INPUT GRAPH



# Inductive Capability: New Graphs

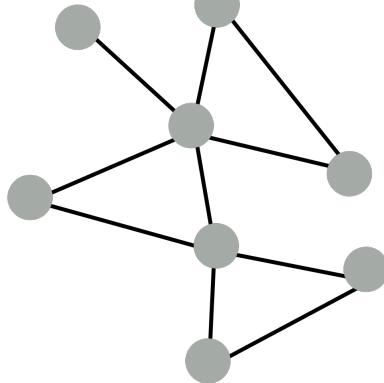


Inductive node embedding → Generalize to entirely unseen graphs

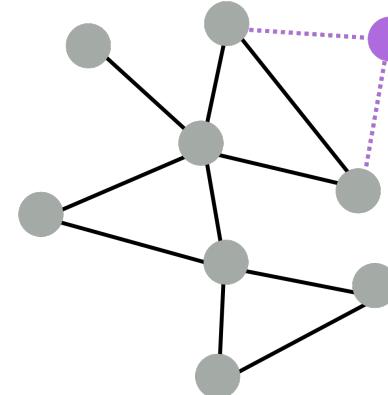
E.g., train on protein interaction graph from model organism A and generate embeddings on newly collected data about organism B

# Inductive Capability: New Nodes

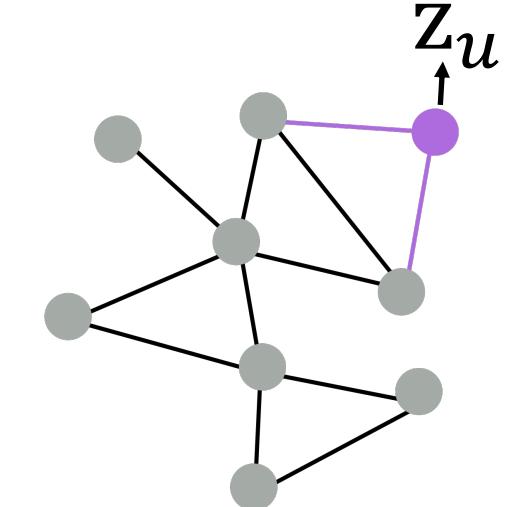
when new node arrives, you just need to do forward pass without retraining the model.



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”

# Summary

- **Recap:** Generate node embeddings by aggregating neighborhood information
  - We saw a **basic variant of this idea**
  - Key distinctions are in how different approaches aggregate information across the layers
- **Next:** Describe GraphSAGE graph neural network architecture

# Outline of Today's Lecture

1. Basics of deep learning 
2. Deep learning for graphs 
3. Graph Convolutional Networks and GraphSAGE 

# **Stanford CS224W: Graph Convolutional Networks and GraphSAGE**

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

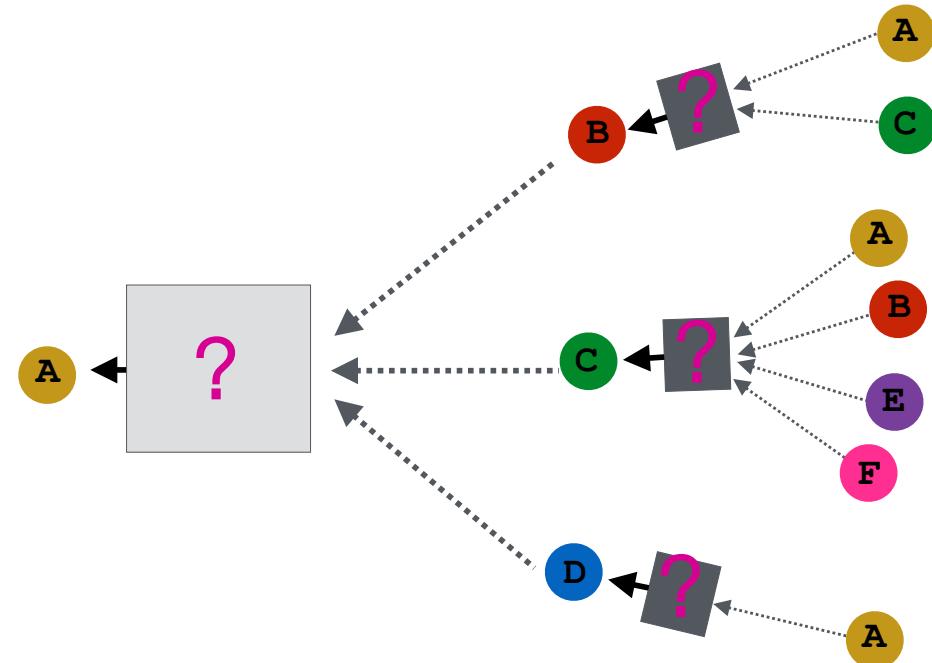
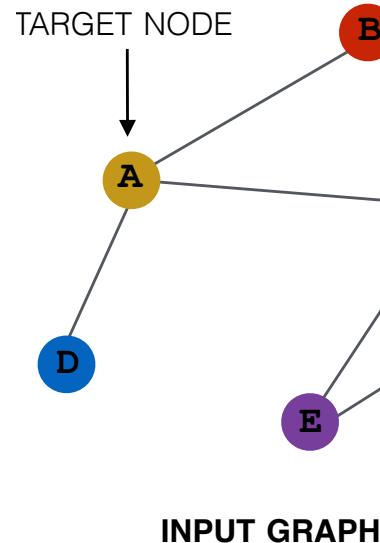
<http://cs224w.stanford.edu>



# GraphSAGE Idea

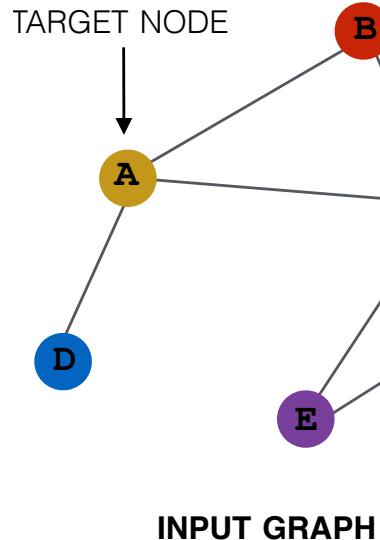
So far we have aggregated the neighbor messages by taking their (weighted) average

**Can we do better?**



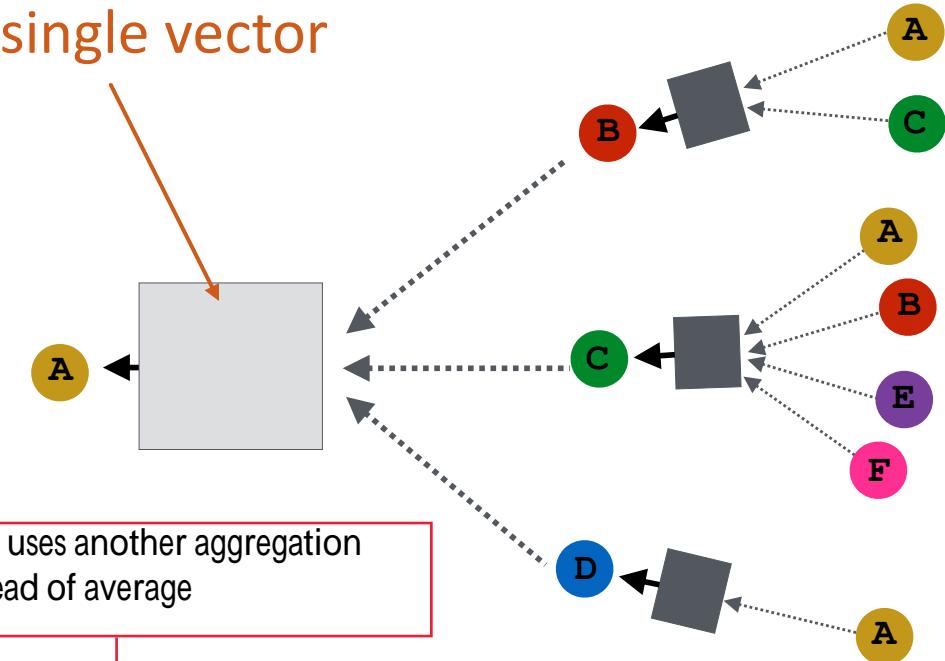
# GraphSAGE Idea (1)

Any differentiable function that  
maps set of vectors in  $N(u)$  to  
a single vector



$$h_v^{(l+1)} = \sigma([W_l \cdot \text{AGG}(\{h_u^{(l)}, \forall u \in N(v)\}), B_l h_v^{(l)}])$$

GraphSAGE uses another aggregation  
function instead of average



How does this message passing architecture differ?

# GraphSAGE Idea (2)

$$h_v^{(l+1)} = \sigma([W_l \cdot \text{AGG}(\{h_u^{(l)}, \forall u \in N(v)\}), B_l h_v^{(l)}])$$

Optional: Apply L2 normalization to  $h_v^{(l+1)}$  embedding at every layer

## ■ $\ell_2$ Normalization:

- $h_v^k \leftarrow \frac{h_v^k}{\|h_v^k\|_2}$   $\forall v \in V$  where  $\|u\|_2 = \sqrt{\sum_i u_i^2}$  ( $\ell_2$ -norm)
- Without  $\ell_2$  normalization, the embedding vectors have different scales ( $\ell_2$ -norm) for vectors
- In some cases (not always), normalization of embedding results in performance improvement
- After  $\ell_2$  normalization, all vectors will have the same  $\ell_2$ -norm

# Neighborhood Aggregation

- Simple neighborhood aggregation:

$$h_v^{(l+1)} = \sigma(W_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)})$$

- GraphSAGE:

Concatenate neighbor embedding  
and self embedding

$$h_v^{(l+1)} = \sigma([W_l \cdot \text{AGG}(\{h_u^{(l)}, \forall u \in N(v)\}), B_l h_v^{(l)}])$$

Flexible aggregation function  
instead of mean

# Neighbor Aggregation: Variants

- **Mean:** Take a weighted average of neighbors

$$\text{AGG} = \sum_{u \in N(v)} \frac{\mathbf{h}_u^{(l)}}{|N(v)|}$$

- **Pool:** Transform neighbor vectors and apply symmetric vector function

Element-wise mean/max

$$\text{AGG} = \gamma(\{\text{MLP}(\mathbf{h}_u^{(l)}), \forall u \in N(v)\})$$

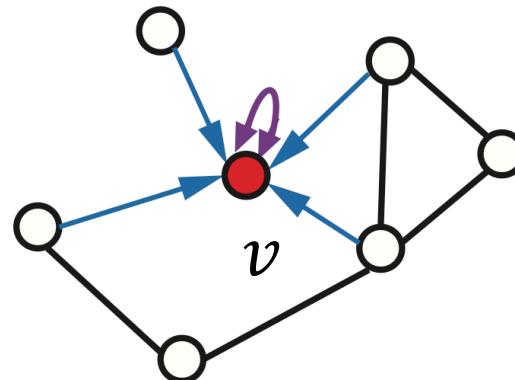
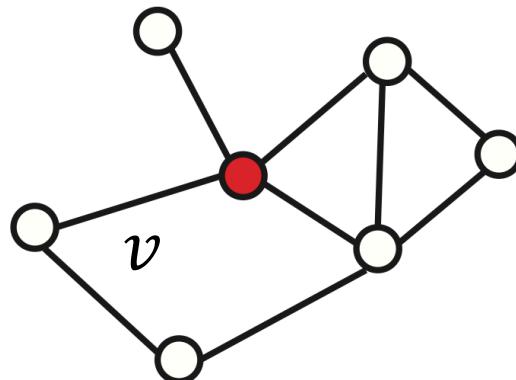
- **LSTM:** Apply LSTM to reshuffled of neighbors

$$\text{AGG} = \text{LSTM}([\mathbf{h}_u^{(l)}, \forall u \in \pi(N(v))])$$

# Recap: GCN, GraphSAGE

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

- Nodes aggregate “messages” from their neighbors using neural networks
- **Graph convolutional networks:**
  - **Basic variant:** Average neighborhood information and stack neural networks
- **GraphSAGE:**
  - Generalized neighborhood aggregation



# Summary

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