Graph Neural Networks

CPSC483: Deep Learning on Graph-Structured Data

Rex Ying

Readings

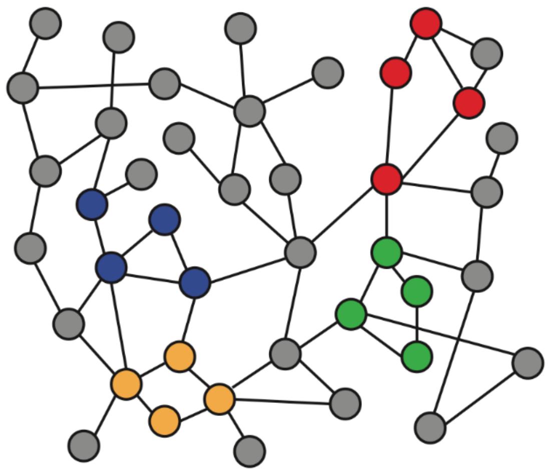
- Readings are updated on the website (syllabus page)

- **Lecture 2 readings**: PageRank and Personalized PageRank (PPR)
- **Lecture 3 readings**:
  - Graph representation learning: methods and application
  - Inductive Representation Learning for Large Graphs (GraphSAGE)

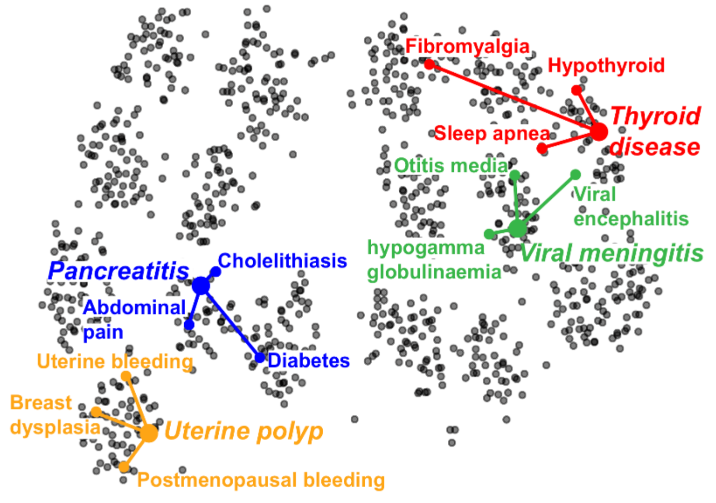
Recap: Node Embeddings

- Intuition: Map nodes to d-dimensional **embeddings** (which are “**representations**” of nodes) such that similar nodes in the graph are embedded close together

$f\Big($



$\Big) =$



Input graph

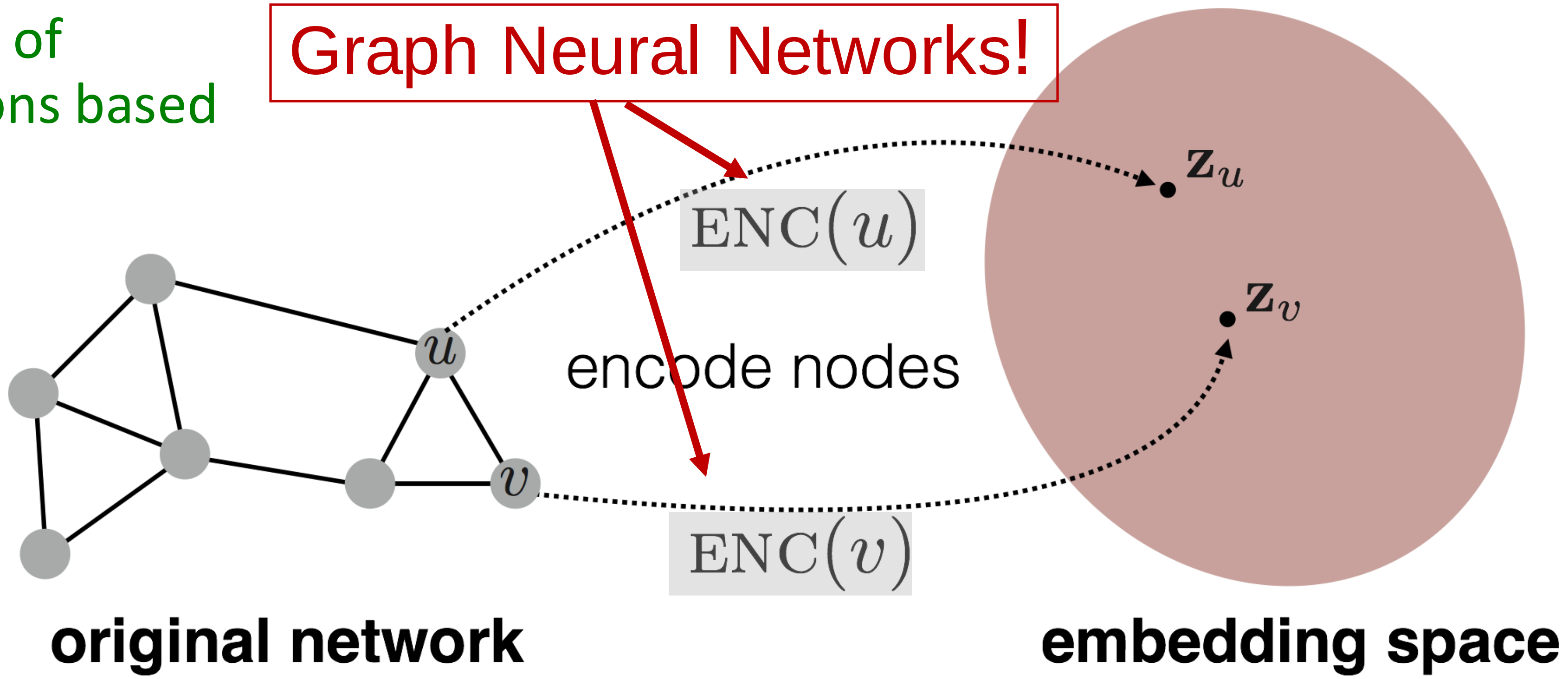
2D node embeddings

How to learn the mapping function f?

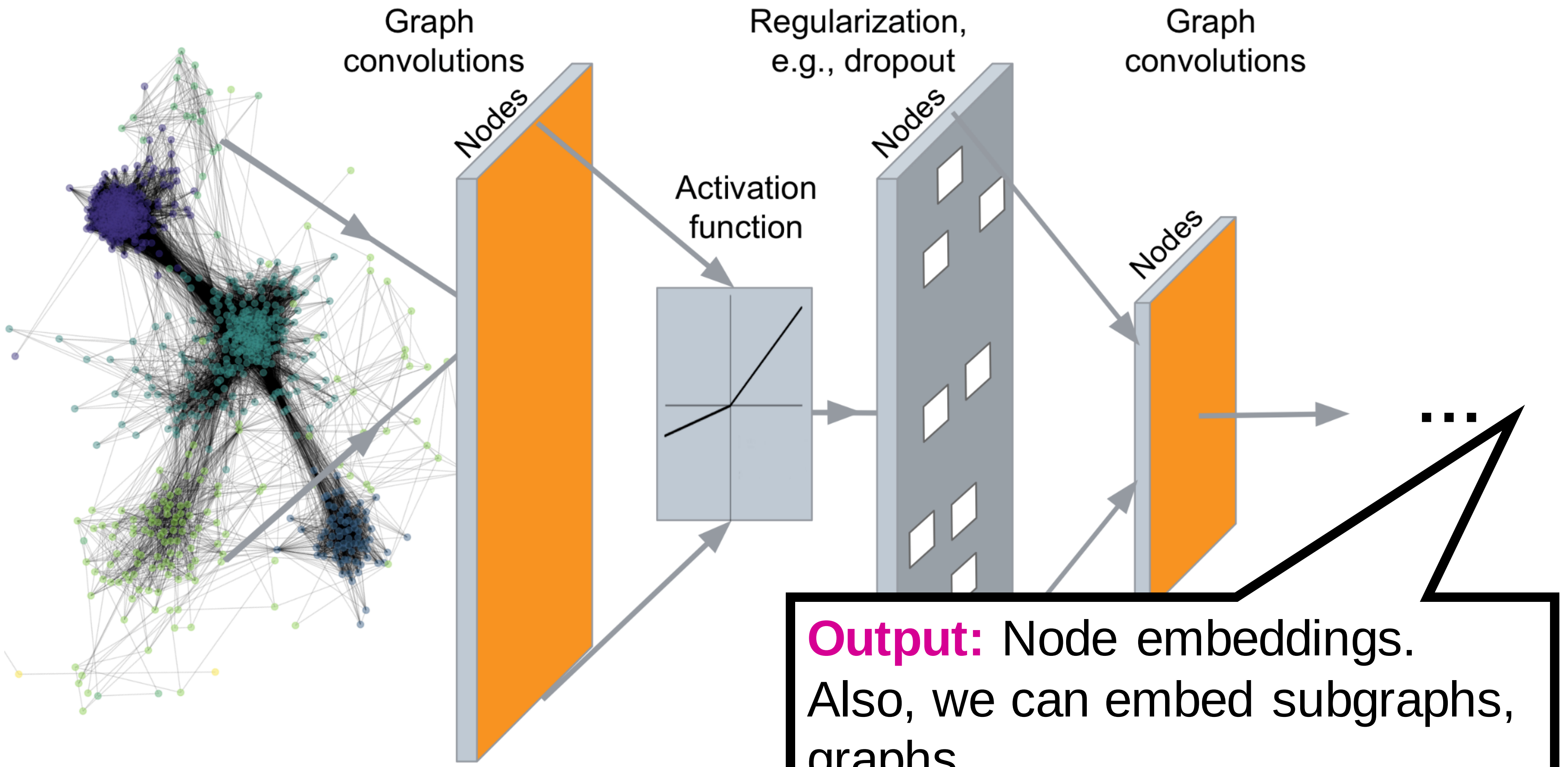
Deep Graph Encoders (1)

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

  $\text{ENC}(\cdot) =$ multiple layers of non–linear transformations based on graph structures

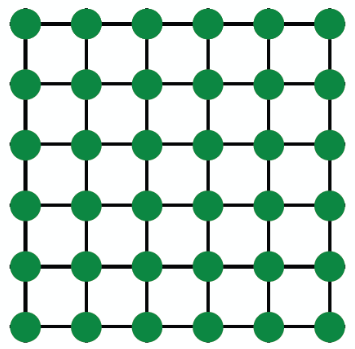


Deep Graph Encoders (2)

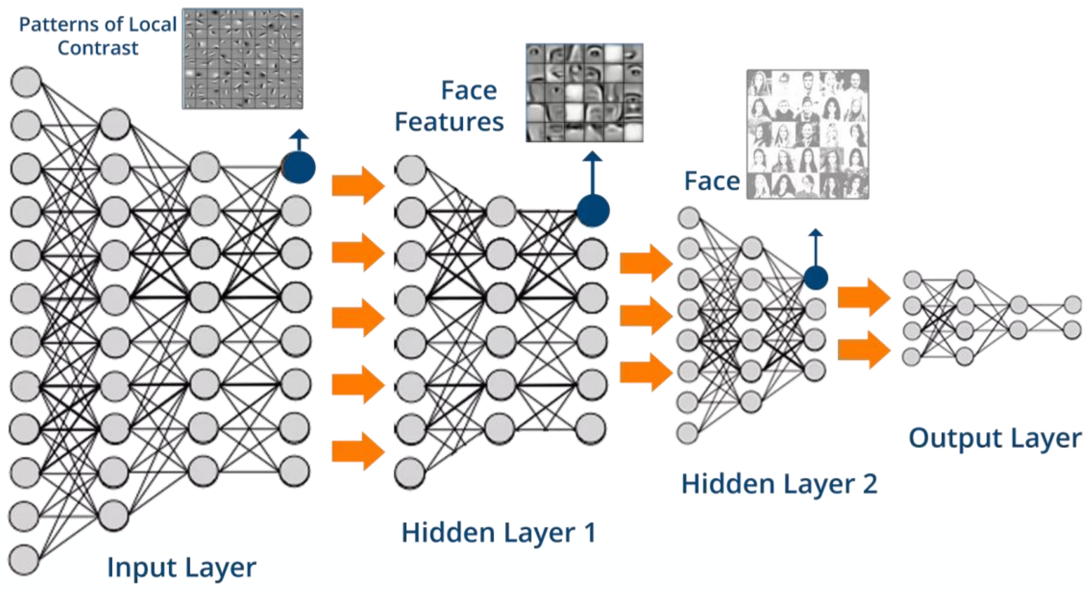


Output: Node embeddings. Also, we can embed subgraphs, graphs

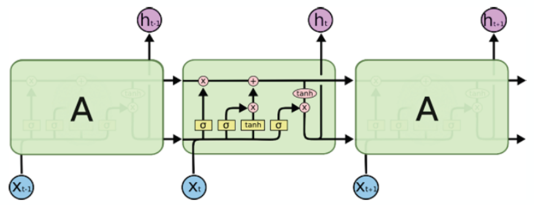
Modern ML Toolbox



Images



Text/Speech

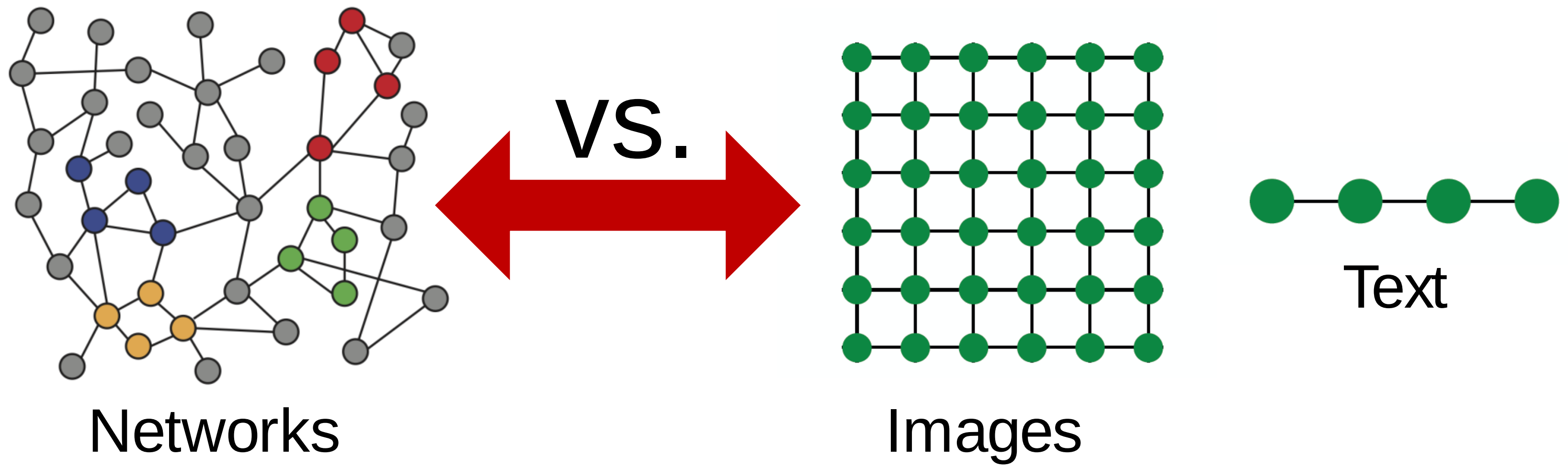


Modern deep learning toolbox is designed for simple sequences & grids

Why is it Hard?

But networks are far more complex!

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



- No fixed node ordering or reference point
- Often dynamic and have multimodal features

Outline of Today's Lecture

1. Basics of deep learning

2. Deep learning for graphs

Outline of Today's Lecture

1. Basics of deep learning

2. Deep learning for graphs

Basics of Deep Learning

Machine Learning as Optimization (1)

- **Supervised learning:** we are given input $\boldsymbol{x}$, and the goal is to predict label $\boldsymbol{y}$
- **Input $\boldsymbol{x}$ 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 (2)

- **Formulate the task as an optimization problem:**

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

Objective function

- Θ: a set of **parameters** we optimize
  - Could contain one or more scalars, vectors, matrices …
  - E.g. $\Theta = \{Z\}$ in the shallow encoder (the embedding lookup)
- $\mathcal{L}$: **loss function**. Example: L2 loss

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

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

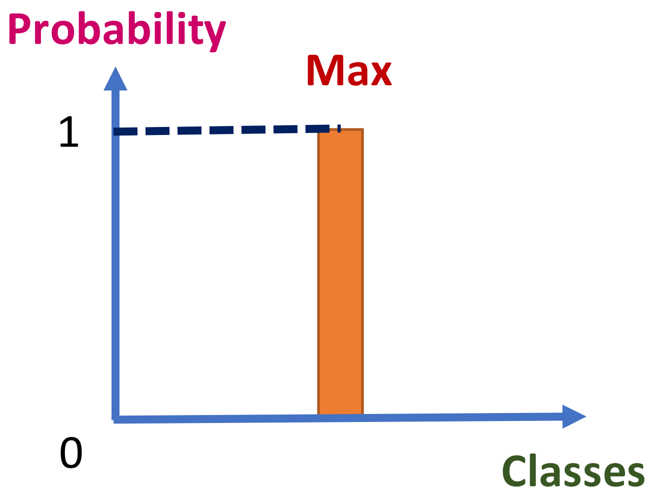
Loss Function Example: Cross Entropy (1)

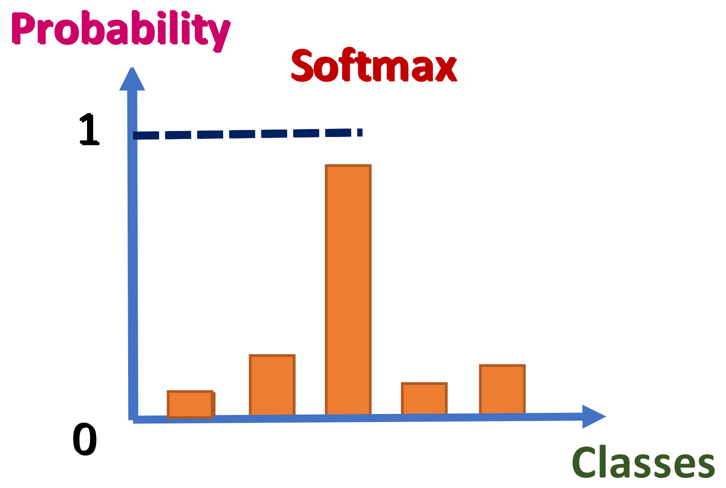
- One common loss for **classification**: cross entropy (CE). Supposed that:
- $f(\boldsymbol{x})$ is the output of a model
  - E.g. $f(\boldsymbol{x}) = [0.1, 0.1, 0.6, 0.2, 0]$
- Label $\boldsymbol{y}$ is a categorical vector (**one-hot** encoding)
  - E.g. $y = [0, 0, 1, 0, 0]^T$ $\quad y$ is of class "3"

- $\text{Softmax}(f(\boldsymbol{x}))_i = \frac{e^{f(\boldsymbol{x})_i}}{\sum_{j=1}^{C} e^{f(\boldsymbol{x})_j}}$ $\longrightarrow$ $f(\boldsymbol{x})_i$ denotes i-th coordinate of the vector $f(\boldsymbol{x})$
  - Where C is the number of classes. ($C = 5$ in this example)
  - E.g. $f(\boldsymbol{x}) = [0.1767, 0.1767, 0.2914, 0.1953, 0.1599]^T$

Softmax

- Softmax is a **differentiable** (or soft) version of the max function

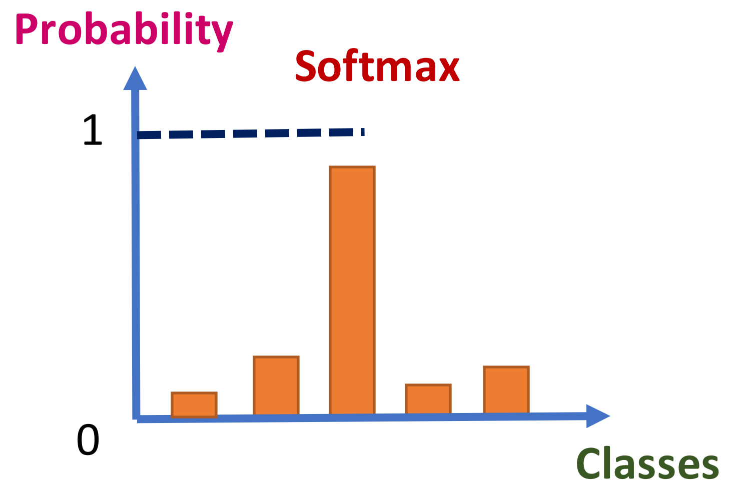




- $\text{Softmax}(f(\boldsymbol{x}))_i = \frac{e^{f(\boldsymbol{x})_i}}{\sum_{j=1}^{C} e^{f(\boldsymbol{x})_j}}$
  - Where C is the number of classes. ($C = 5$ in this example)
  - E.g. $f(\boldsymbol{x}) = [0.1767, 0.1767, 0.2914, 0.1953, 0.1599]^T$

Loss Function Example: Cross Entropy (2)

- $\mathrm{CE}\big(\boldsymbol{y}, f(\boldsymbol{x})\big) = -\sum_{i=1}^{C}(\boldsymbol{y}_i \log f(\boldsymbol{x})_i)$
  - $\boldsymbol{y}_i, f(\boldsymbol{x})_i$ are the **actual** and **predicted** value of the i-th class.
  - **Intuition:** the lower the loss, the closer the prediction is to one-hot
- In classification, $\boldsymbol{y}$ is **one-hot**, whereas $f(\boldsymbol{x})$ is the output of a softmax
  - The summation in CE only has **1 non-zero term**
- Total loss over all training examples
  - $\mathcal{L} = \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{T}} \mathrm{CE}\big(\boldsymbol{y}, f(\boldsymbol{x})\big)$
  - $\mathcal{T}$: training set containing all pairs of data and labels $(\boldsymbol{x}, \boldsymbol{y})$



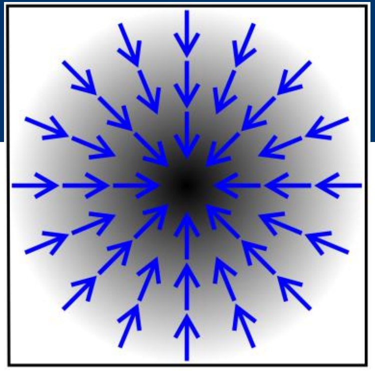
Machine Learning as Optimization (1)

- **How to optimize the objective function?**
- Non-gradient approaches
  - Bayesian optimization, Gaussian processes, Simulated annealing, Evolutionary algorithms

In deep learning, we use gradient approaches for scalability

- Therefore, we require the loss function $\mathcal{L}$ to be **differentiable**
  - There are ways to tackle optimization for non-differentiable functions:
    - Straight-through estimator (Gumbel Softmax)
    - Reinforce algorithm, or more generally, reinforcement learning

Machine Learning as Optimization (2)



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

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

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

  - $\Theta_1, \Theta_2 \ldots$: 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: 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 $\nabla_{\Theta}\mathcal{L}(\boldsymbol{y}, f(\boldsymbol{x}))$, where $\boldsymbol{x}$ is the **entire** dataset!
    - This means summing gradient contributions over all the points in the dataset
    - Modern datasets often contain billions of data points
    - Extremely expensive for every gradient descent step

- **Solution: Stochastic gradient descent (SGD)**
  - At every step, pick a different **minibatch** $\boldsymbol{\mathcal{B}}$ containing a subset of the dataset, use it as input $\boldsymbol{x}$

Minibatch SGD

- **Concepts:**
  - **Batch size**: the number of data points in a minibatch
    - E.g. number of nodes for node classification task
  - **Iteration**: 1 step of SGD on a minibatch
  - **Epoch**: one full pass over the dataset (# iterations is equal to ratio of dataset size and batch size)
- SGD is unbiased estimator of full gradient:
  - But there is no guarantee on the rate of convergence
  - In practice often requires tuning of learning rate
- Common optimizer that improves over SGD:
  - Adam, Adagrad, Adadelta, RMSprop …

Neural Network Function (1)

- **Objective:** $\min_{\Theta} \mathcal{L}(\boldsymbol{y}, f(\boldsymbol{x}))$
- In deep learning, the function f can be very complex
- To start simple, consider linear function

$$f(\boldsymbol{x}) = W \cdot \boldsymbol{x}, \qquad \Theta = \{W\}$$

- If f returns a scalar, then W is a learnable **vector**

$$\nabla_W f = (\frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}, \frac{\partial f}{\partial w_3} \dots)$$

- If f returns a vector, then W is the **weight matrix**

$$\nabla_W f = W^T$$

Neural Network Function (2)

Derivative of f w.r.t. X	Scalar	Vector	Matrix
Scalar	Scalar	Vector	Matrix
Vector	Vector	Matrix	Tensors ☹
Matrix	Matrix	Tensors ☹	Tensors ☹

Jacobian matrix of f

Back-propagation

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

$$f(\boldsymbol{x}) = a = W_2(\underbrace{W_1\boldsymbol{x}}_{\boldsymbol{z}}), \qquad \Theta = \{W_1, W_2\}$$

- Recall **chain rule**:
- E.g. $\nabla_{\boldsymbol{x}} f = \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial \boldsymbol{x}}$

We define:

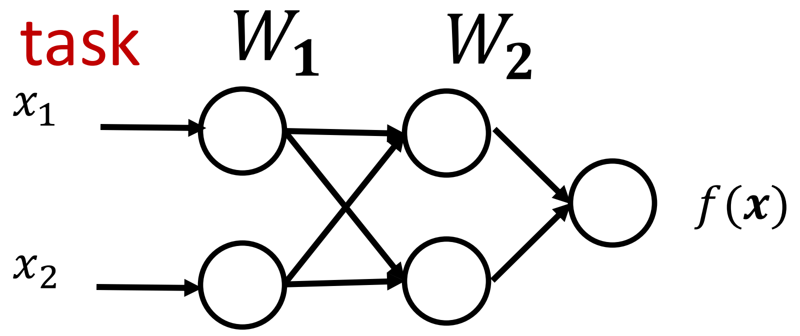
$\boldsymbol{z} = W_1\boldsymbol{x}$

$a = f(\boldsymbol{x}) = W_2\boldsymbol{z}$

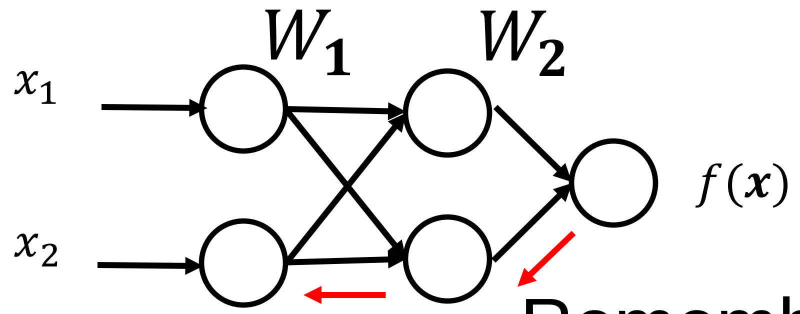
- **Back-propagation**: Use of **chain rule** to propagate gradients of intermediate steps, and finally obtain gradient of $\mathcal{L}$ w.r.t. Θ

Back-propagation Example (1)

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



Back-propagation Example (2)



- **Forward propagation:**
  Compute loss starting from input
  - $$\begin{tikzcd} \boldsymbol{x} \arrow[r, "\text{Multiply } W_1"'] & \boldsymbol{z} \arrow[r, "\text{Multiply } W_2"'] & a \arrow[r, "\text{Loss}"'] & \mathcal{L} \end{tikzcd}$$

Remember:

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

$\boldsymbol{z} = W_1\boldsymbol{x}$

$a = W_2\boldsymbol{z}$

- **Back-propagation to compute gradient of** $\Theta = \{W_1, W_2\}$

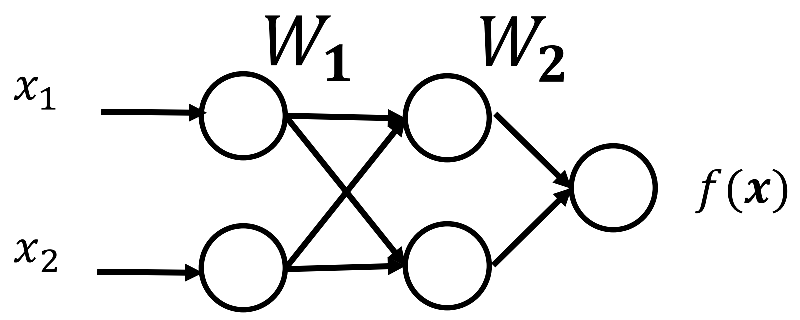
- Start from loss, compute the gradient

$$\frac{\partial\mathcal{L}}{\partial W_2} = \frac{\partial\mathcal{L}}{\partial a}\cdot\frac{\partial a}{\partial W_2}, \qquad \frac{\partial\mathcal{L}}{\partial W_1} = \frac{\partial\mathcal{L}}{\partial a}\cdot\frac{\partial a}{\partial \boldsymbol{z}}\cdot\frac{\partial \boldsymbol{z}}{\partial W_1}$$

Compute backwards **Compute backwards**

Back-propogation: Concrete Example (1)

- Suppose that (minibatch of size 1)
  - **Features**: $\boldsymbol{x} = (x_1 \quad x_2)^T = (0.8 \quad 1.1)^T$
  - **Label**: $y = 1.00$
  - **Weights**: $W_1 = \begin{pmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{pmatrix}, W_2 = (0.5 \quad 0.6)$
  - **Model output**: $a = f(\boldsymbol{x}) = W_2 W_1 \boldsymbol{x} = (0.5 \quad 0.6) \begin{pmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{pmatrix} \begin{pmatrix} 0.8 \\ 1.1 \end{pmatrix} = 0.558$
  - **Loss function**: $\mathcal{L} = \|y - a\| = (0.558 - 1)^2 = 0.1954$



Back-propogation: Concrete Example (2)

- To calculate gradients of W_2:
- Recall that
  - $\mathcal{L} = \|y - a\|_2$
  - $a = W_2 \boldsymbol{z}$
  - $\boldsymbol{z} = W_1 \boldsymbol{x}$
- Apply chain rule:

$$\frac{\partial a}{\partial \boldsymbol{z}} \cdot \frac{\partial \boldsymbol{z}}{\partial W_1}$$

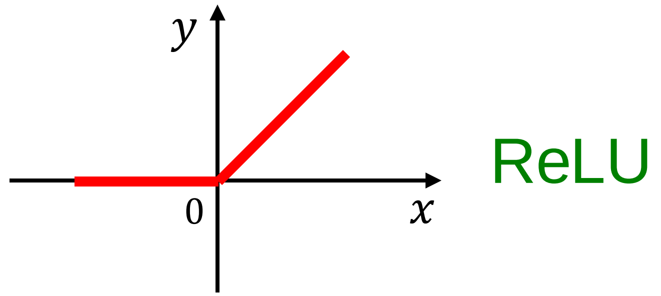
$$\frac{\partial \mathcal{L}}{\partial W_1} = \frac{\partial \mathcal{L}}{\partial a} \cdot \frac{\partial a}{\partial W_1} = -2(y - a) \cdot (xW_2)^T = \begin{pmatrix} -0.3536 & -0.4862 \\ -0.4243 & -0.5834 \end{pmatrix}$$

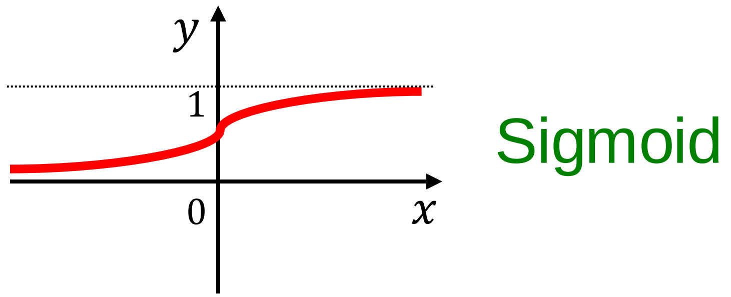
Non-linearity

- Note that in $f(\boldsymbol{x}) = W_2(W_1\boldsymbol{x})$, W_2W_1 is another matrix (or vector, if we do binary classification and output only 1 logit)
- Hence $f(\boldsymbol{x})$ is still linear w.r.t. $\boldsymbol{x}$ no matter how many weight matrices we compose
- **Introduce non-linearity:**
  - **Rectified linear unit (ReLU)**

    $ReLU(x) = \max(x, 0)$
  - **Sigmoid**

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



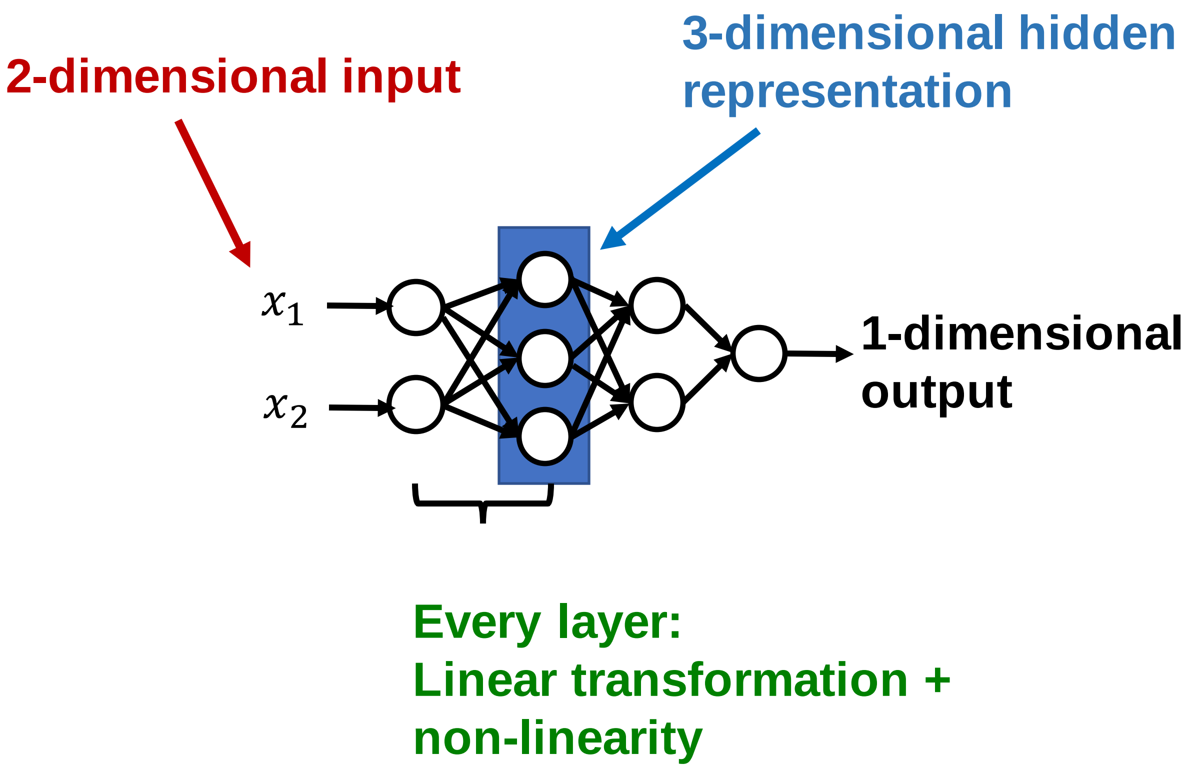


Multi-layer Perceptron (MLP)

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

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

  - where W_l is weight matrix that transforms hidden representation at layer l to layer $l+1$
  - b^l is bias at layer l, and is added to the linear transformation of $\boldsymbol{x}$
  - σ is non-linearity function (e.g., sigmoid)
- Suppose $\boldsymbol{x}$ is 2-dimensional, with entries x_1 and x_2


2-dimensional input
3-dimensional hidden representation
x_1
x_2
1-dimensional output
Every layer:
Linear transformation + non-linearity

Summary

- **Objective function:**

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

- f can be a simple linear layer, an MLP, or other neural networks (e.g., a GNN later)
- Sample a minibatch of input $\boldsymbol{x}$
- **Forward propagation:** compute $\mathcal{L}$ given $\boldsymbol{x}$
- **Back-propagation:** obtain gradient $\nabla_{\Theta}\mathcal{L}$ 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

Deep Learning for Graphs

Content

- **Local network neighborhoods:**
  - Describe aggregation strategies
  - Define computation graphs

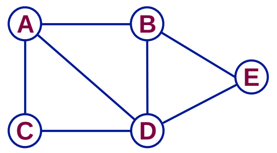
- **Stacking multiple layers:**
  - Describe the model, parameters, training
  - How to fit the model?
  - Simple example for unsupervised and supervised training

Setup

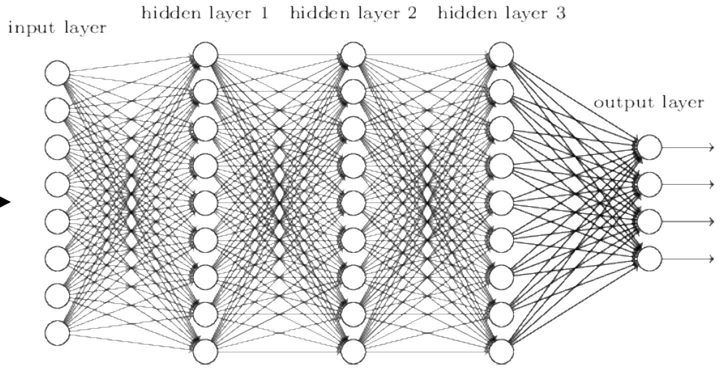
- **Assume we have a graph $\boldsymbol{G}$:**
  - V is the **vertex set**
  - $\boldsymbol{A}$ is the **adjacency matrix** (assume binary)
  - $\boldsymbol{X} \in \mathbb{R}^{d \times |V|}$ is a matrix of **node features**
  - v: a node in V; $N(v)$: the set of neighbors of v.
  - **Node features:**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



$$\begin{array}{c|ccccc|cc} & A & B & C & D & E & \multicolumn{2}{c}{\text{Feat}} \\ \hline A & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ C & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ D & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{array} \longrightarrow$$

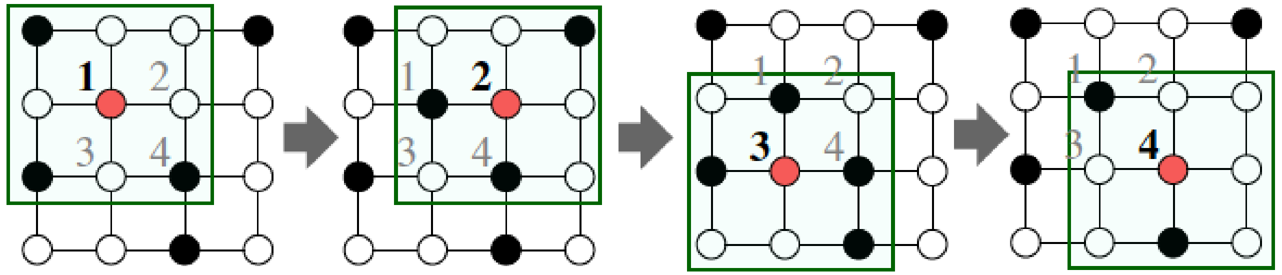


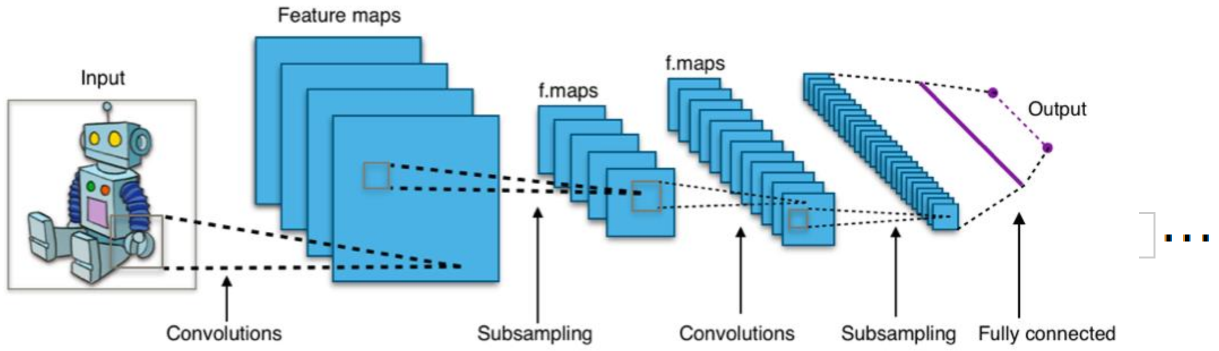
?

- 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:**

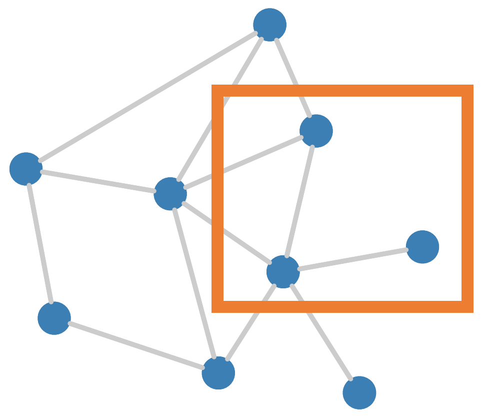




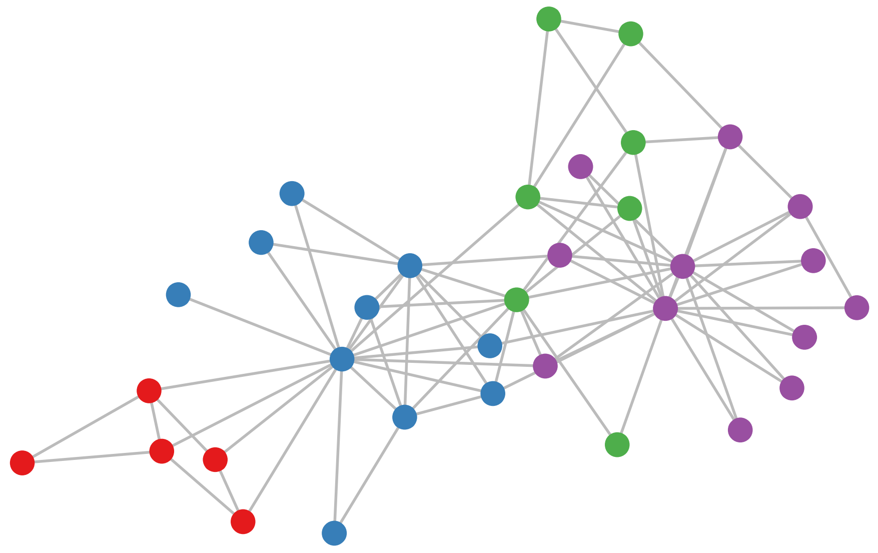
- Goal is to generalize convolutions beyond simple lattices
- Leverage node features/attributes (e.g., text, images)

Real-World Graphs

- **But our graphs look like this:**



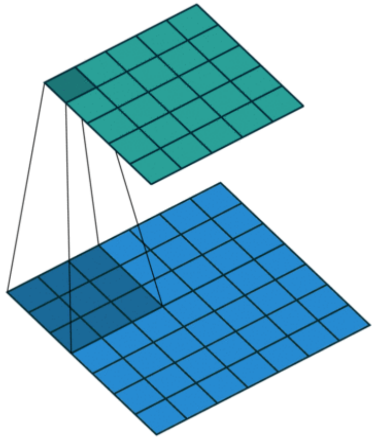
or this



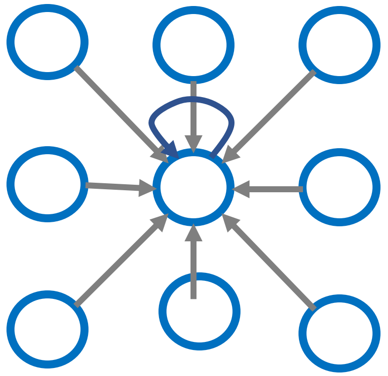
- There is no fixed notion of locality or sliding window on the graph
- Graph is permutation invariant

From Images to Graphs

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



Image



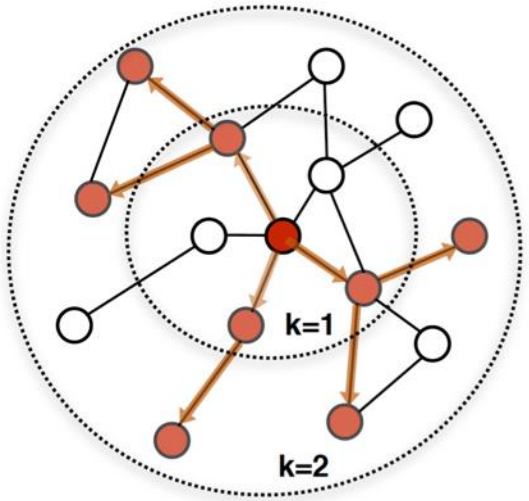
Graph

Idea: transform information at the neighbors and combine it:

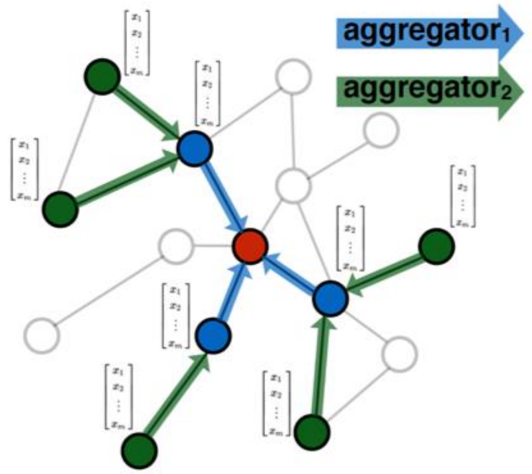
- Transform "messages" h_i from neighbors: $W_i\, h_i$
- Add them up: $\sum_i W_i\, h_i$

Graph Convolutional Networks

- Idea: Node's neighborhood defines a computation graph



Determine node computation graph

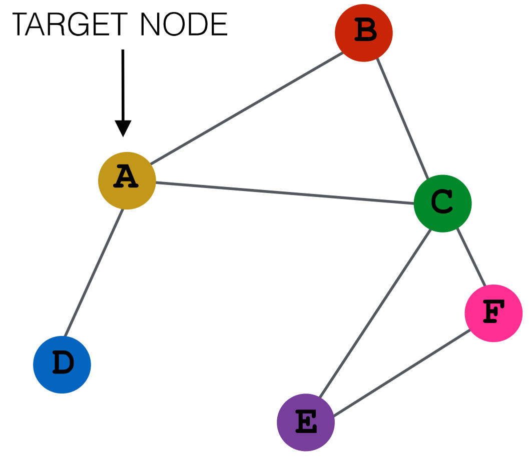


Propagate and transform information

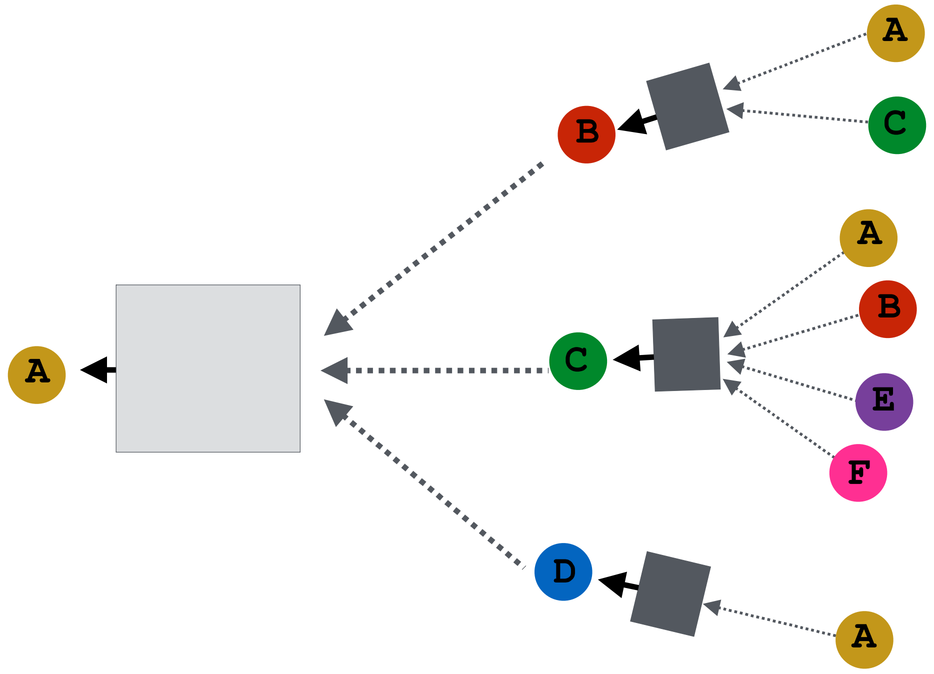
Learn how to propagate information across the graph to compute node features

Idea: Aggregate Neighbors (1)

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

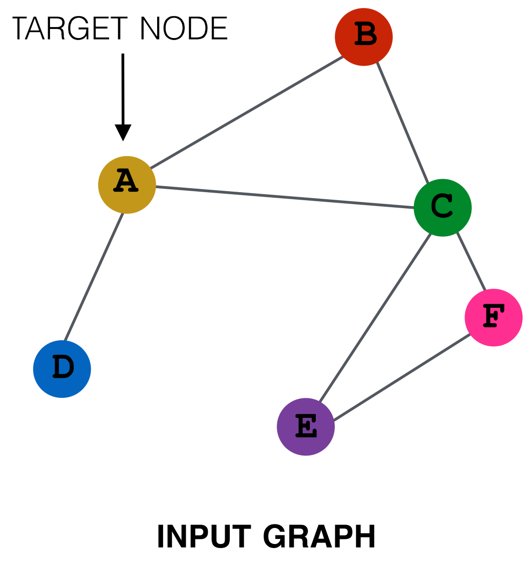


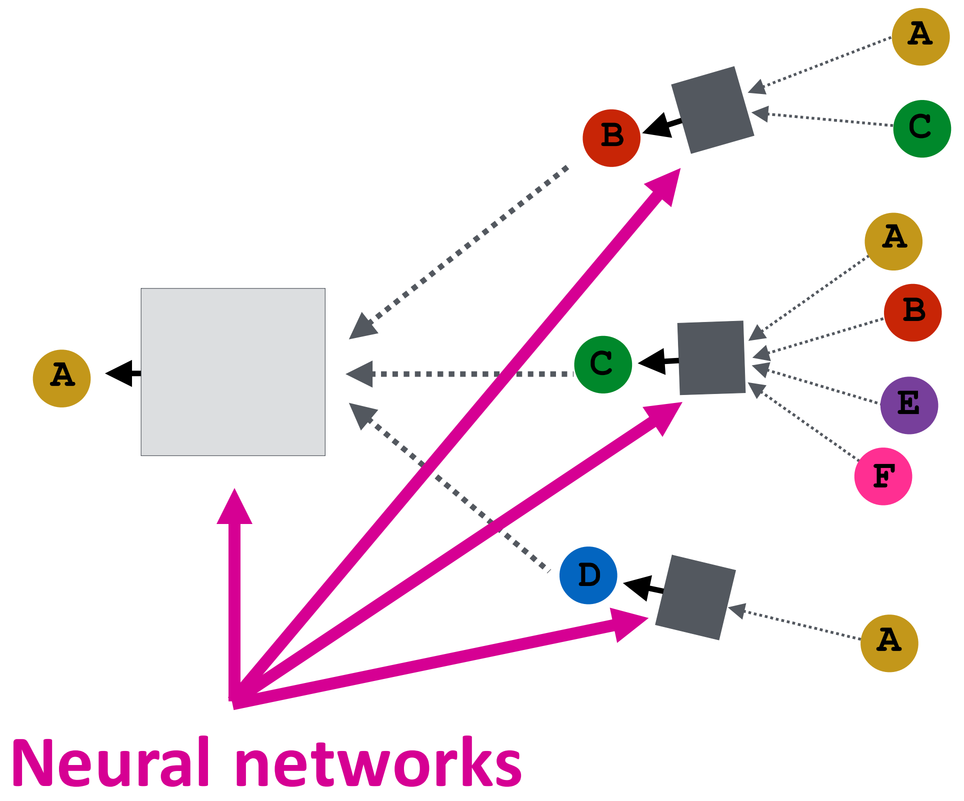
INPUT GRAPH



Idea: Aggregate Neighbors (2)

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

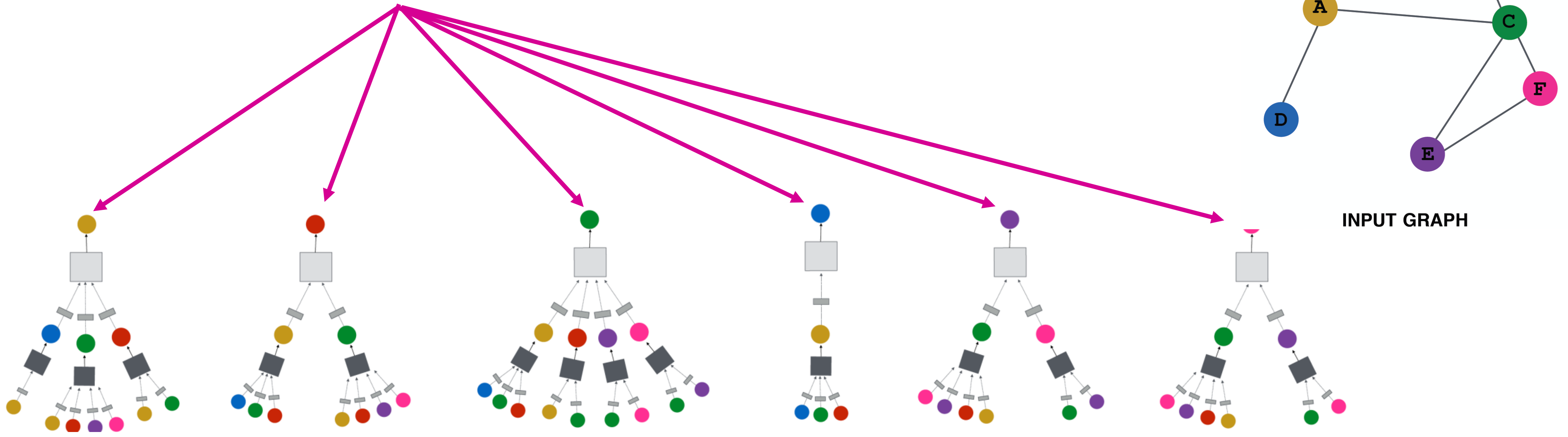




Idea: Aggregate Neighbors (3)

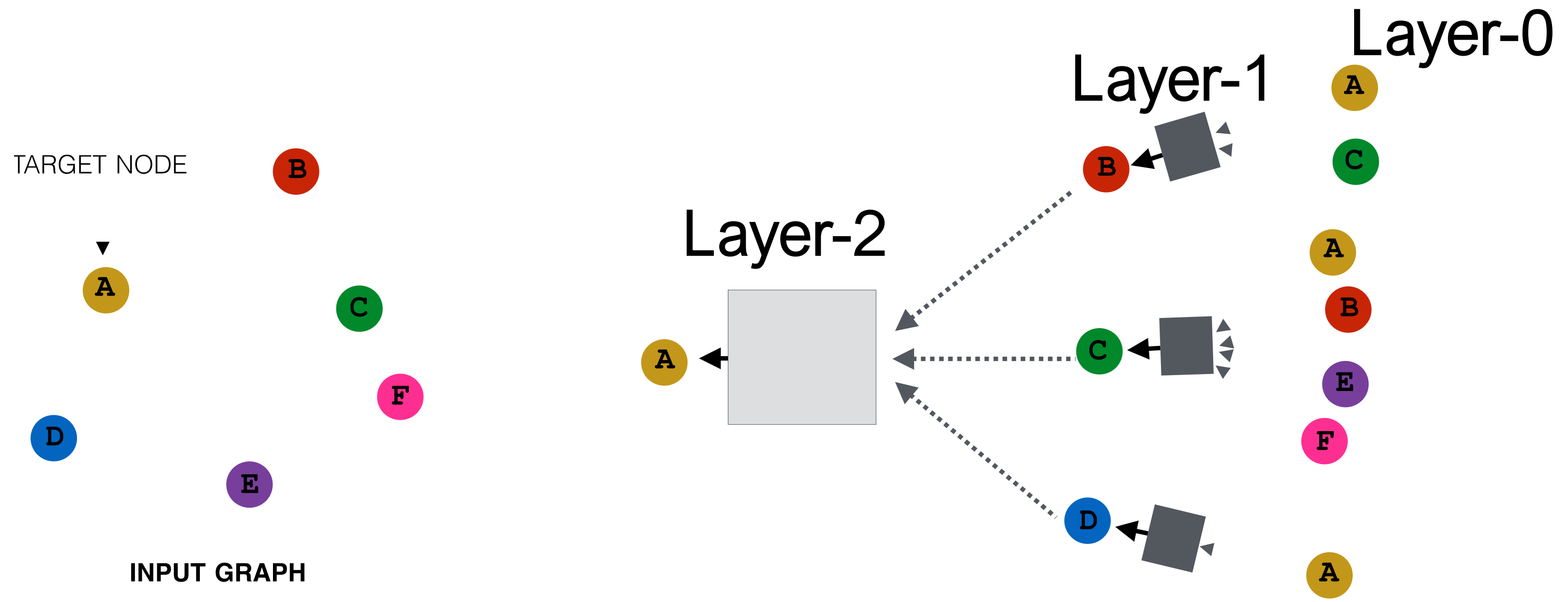
- **Intuition:** Network neighborhood defines a computation graph

Every node defines a computation graph based on its neighborhood!



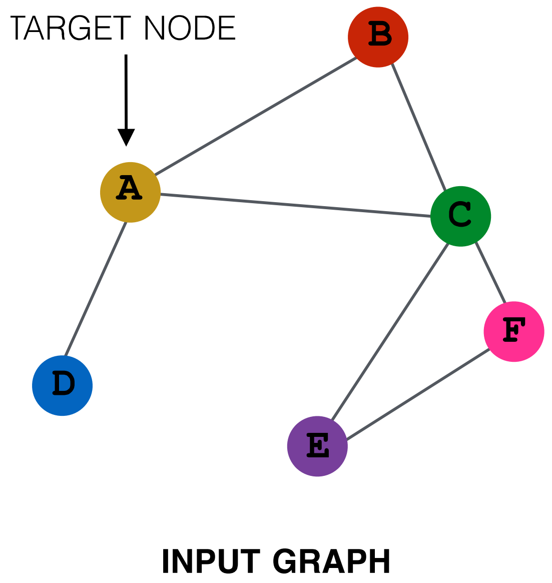
Deep Model: Many Layers

- Model can be of arbitrary depth:
  - Nodes have embeddings at each layer
  - Layer-0 embedding of node u is its input feature, $\boldsymbol{x}_u$
  - Layer-k embedding gets information from nodes that are k hops away

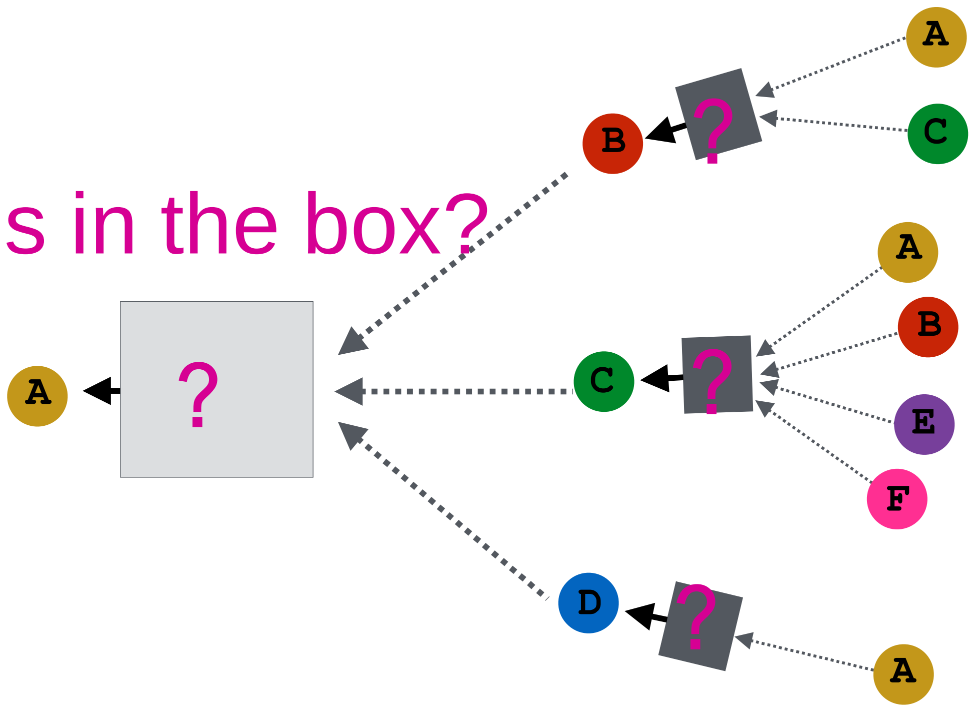


Neighborhood Aggregation (1)

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

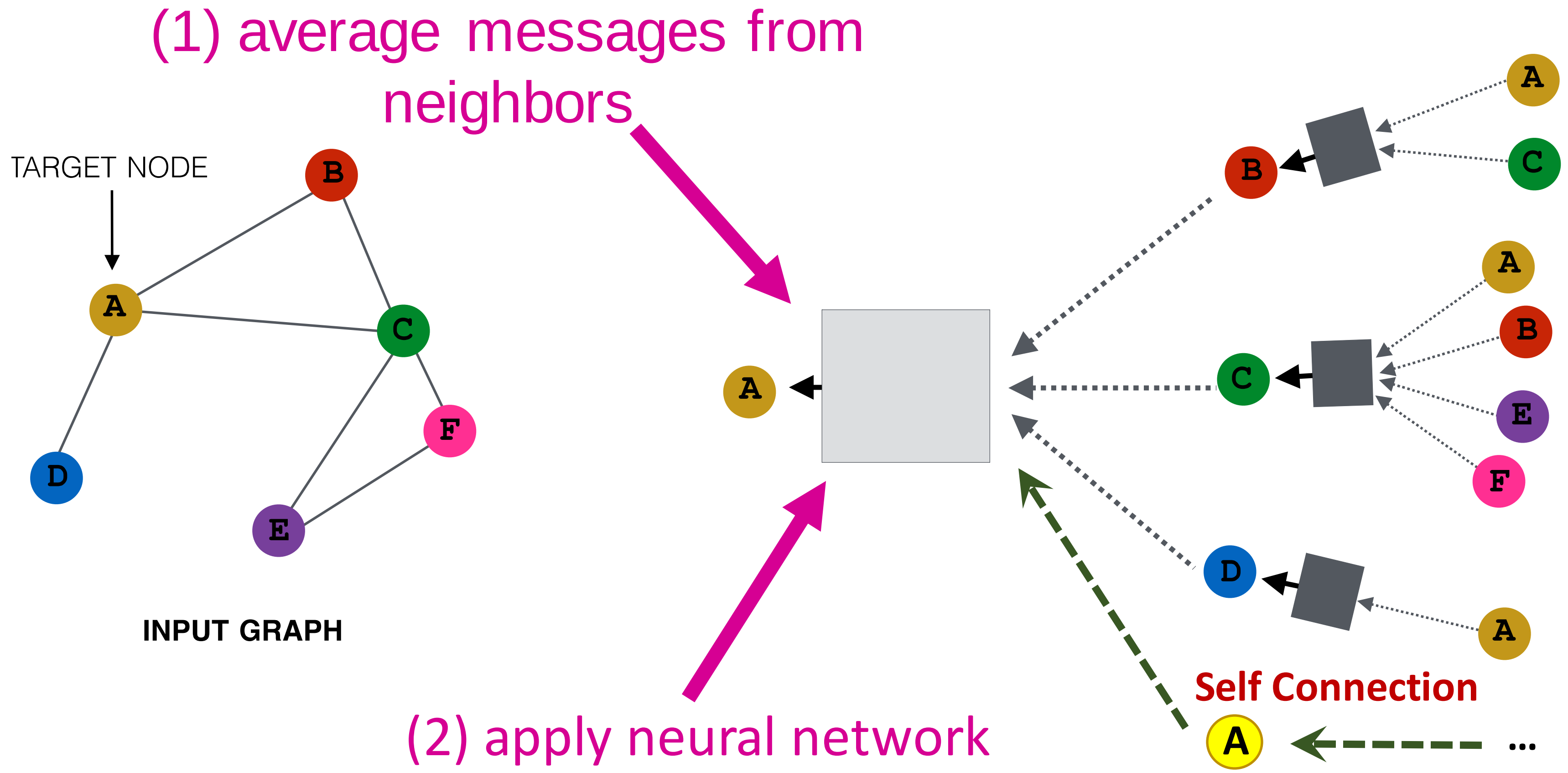


What is in the box?



Neighborhood Aggregation (2)

- **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

$$\boxed{h_v^0 = x_v}$$

Initial 0-th layer embeddings are equal to node features

embedding of v at layer l

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

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

Embedding after L layers of neighborhood aggregation

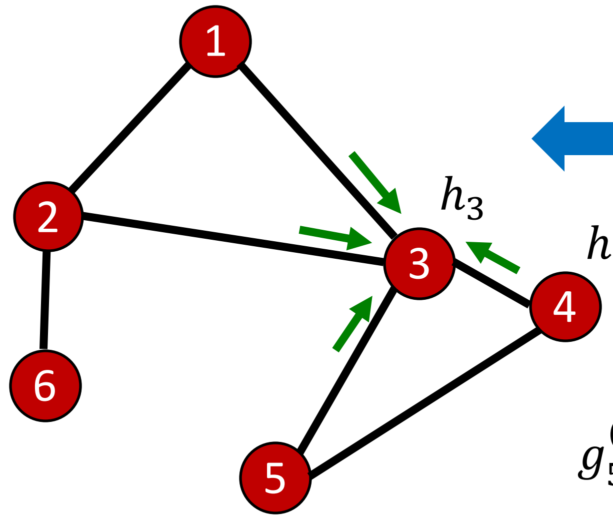
Non-linearity (e.g., ReLU)

Average of neighbor's previous layer embeddings

Total number of layers

Order Invariance (Node Permutation)

- Ordering of nodes is **not** important.

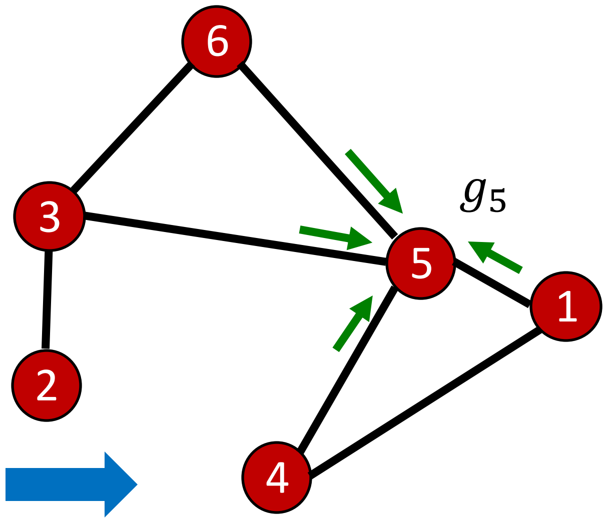


$$h_3^{(l)} = h_5^{(l)}$$

$$h_3^{(l+1)} = \sigma\left(W_l \frac{h_1^{(l)} + h_2^{(l)} + h_4^{(l)} + h_5^{(l)}}{|N(3)|} + B_l h_3^{(l)}\right)$$

$$g_5^{(l+1)} = \sigma\left(W_l \frac{g_6^{(l)} + g_3^{(l)} + g_4^{(l)} + g_1^{(l)}}{|N(5)|} + B_l g_5^{(l)}\right)$$

$$h_3^{(l+1)} = h_5^{(l+1)}$$

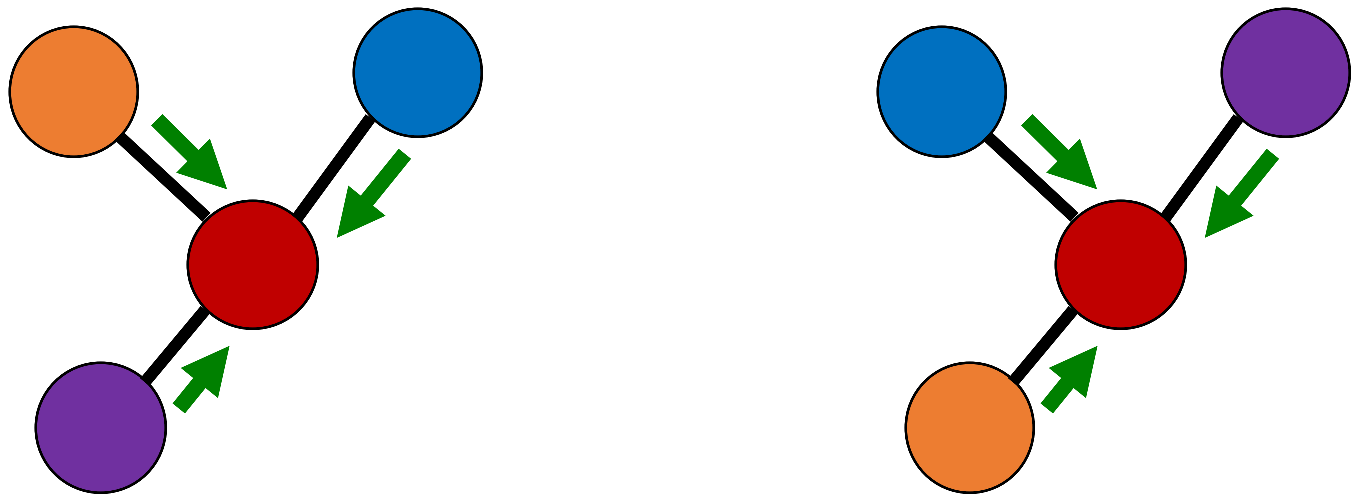


$$A = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Order Invariance (Neighbor Permutation)

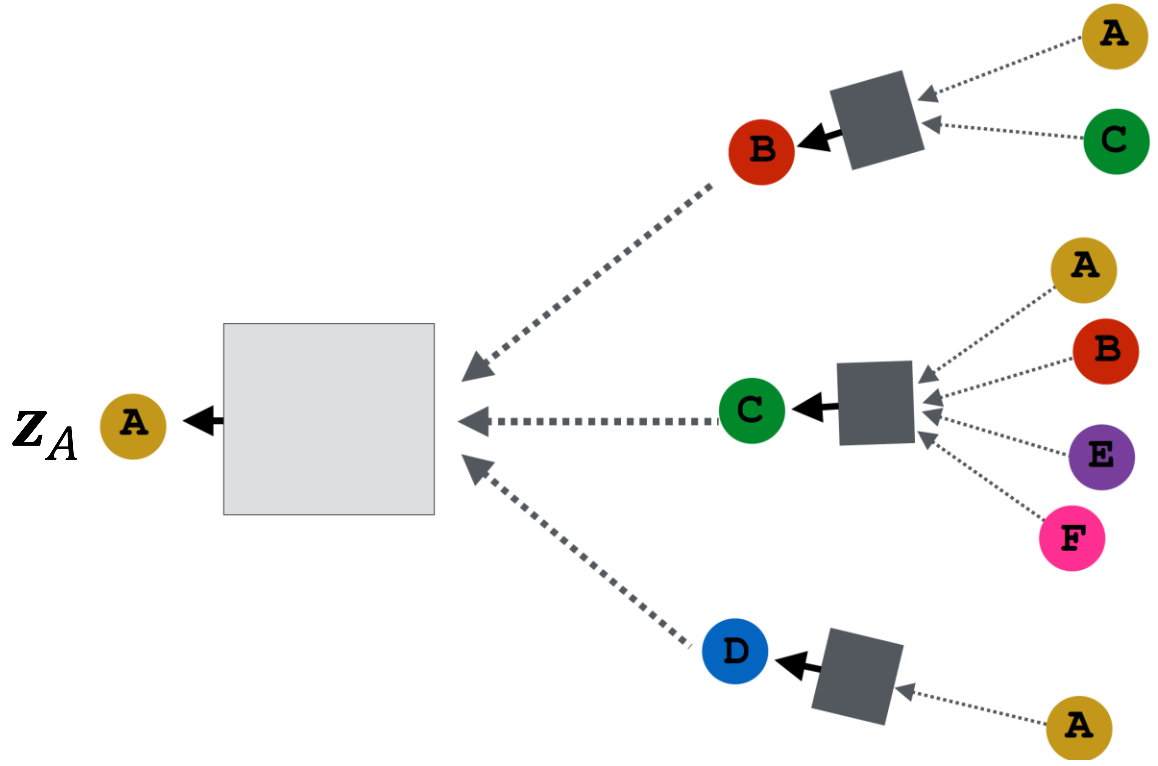
- Aggregation function has to be order-invariant.



- Aggr(● , ● , ●) ≠ Aggr(● , ● , ●) if the aggregation function is not order invariant. That's not what we want because the two graphs are actually the same graph.

Training the Model

How do we train the model to generate embeddings?



Need to define a loss function on the embeddings

Model Parameters

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

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

$$\mathrm{h}_v^{(l+1)} = \sigma\left(\boxed{\mathrm{W}_l} \sum_{u \in \mathrm{N}(v)} \frac{\mathrm{h}_u^{(l)}}{|\mathrm{N}(v)|} + \boxed{\mathrm{B}_l} \mathrm{h}_v^{(l)}\right), \forall l \in \{0, \dots, L-1\}$$

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

Final node embedding

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

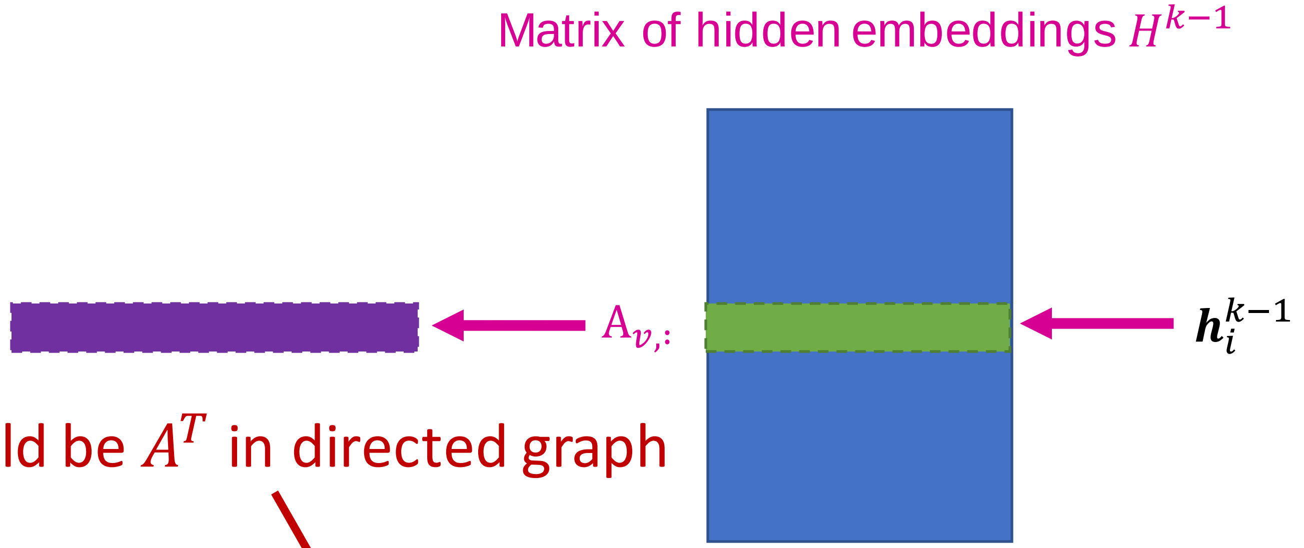
h_v^l: the hidden representation of node v at layer l

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

Matrix Formulation (1)

- **Many aggregations can be performed efficiently by (sparse) matrix operations**
- Let $H^{(l)} = [h_1^{(l)} \dots h_{|V|}^{(l)}]^{\mathrm{T}}$
- Then: $\sum_{u \in N_v} h_u^{(l)} = \mathrm{A}_{v,:} H^{(l)}$
- Let D be diagonal matrix where $D_{v,v} = \mathrm{Deg}(v) = |N(v)|$
  - The inverse of D: D^{-1} is also diagonal: $D_{v,v}^{-1} = 1/|N(v)|$
- **Therefore,** $\boxed{\sum_{u \in N(v)} \frac{h_u^{(l-1)}}{|N(v)|}} \Rightarrow H^{(l+1)} = D^{-1} A H^{(l)}$

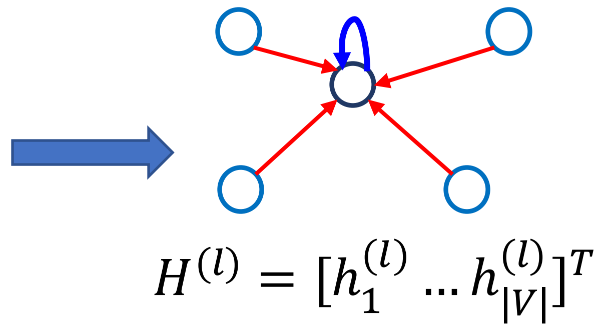
should be A^T in directed graph



Matrix Formulation (2)

- Re-writing update function in matrix form:

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



- where $\tilde{A} = D^{-1} A$
  - Red: neighborhood aggregation
  - Blue: self transformation (matrix)
- In practice, this implies that efficient sparse matrix multiplication can be used ($\tilde{A}$ is sparse)
- **Note**: not all GNNs can be expressed in matrix form, when aggregation function is complex

How to train a GNN

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

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

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

Unsupervised Training

- **"Similar" nodes have similar embeddings**

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

Recall in slides 15:

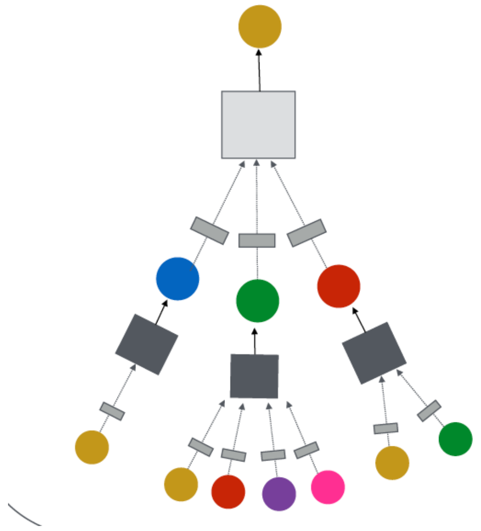
$$\mathrm{CE}(\boldsymbol{y}, f(\boldsymbol{x})) = -\sum_{i=1}^{C} (\boldsymbol{y}_i \log f(\boldsymbol{x})_i)$$

  - Where $y_{u,v} = 1$ when node u and v are **similar**
  - CE is the cross entropy (slide)
  - DEC is the decoder such as inner product
- **Node similarity** can be anything, e.g., a loss based on (we may talk in the future):
  - **Random walks** (node2vec, DeepWalk, struc2vec)
  - **Matrix factorization**
  - **Node proximity in the graph**

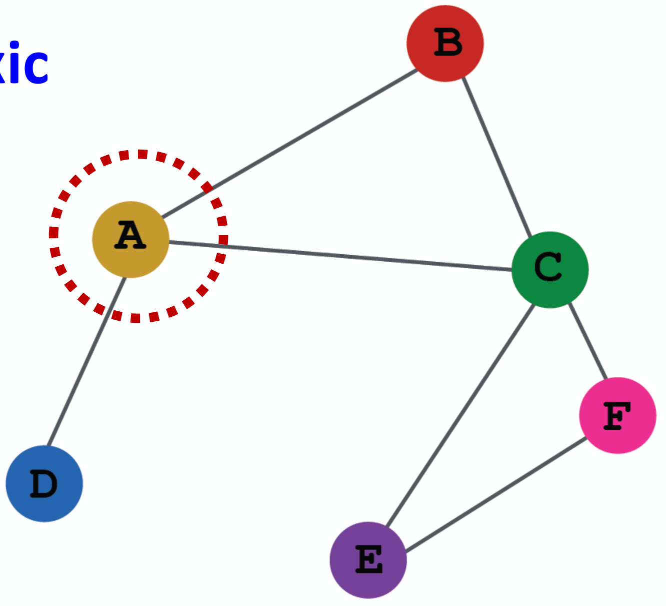
Supervised Training (1)

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

INPUT GRAPH

Supervised Training (2)

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

- Use cross entropy loss (slide)

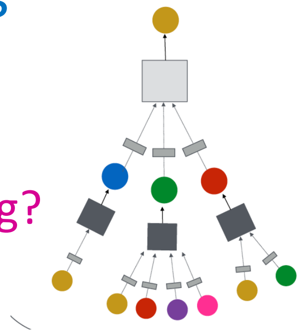
$$\mathcal{L} = \sum_{v \in V} \boxed{y_v} \log(\sigma(\boxed{z_v^{\mathrm{T}}}\boxed{\theta})) + (1 - \boxed{y_v})\log(1 - \sigma(\boxed{z_v^{\mathrm{T}}}\boxed{\theta}))$$

Encoder output: node embedding

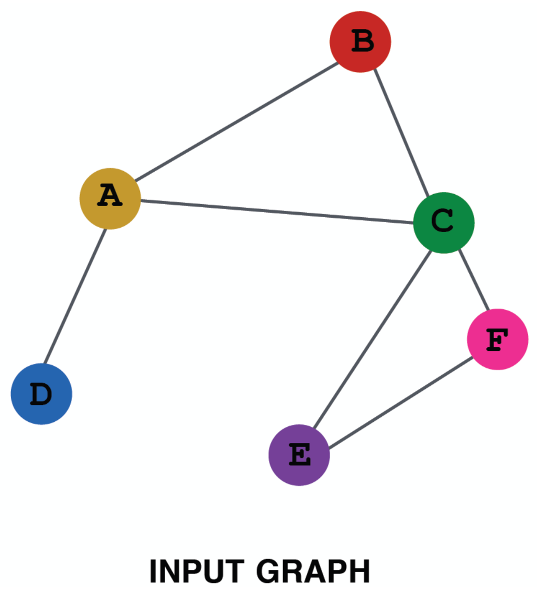
Classification weights

Node class label

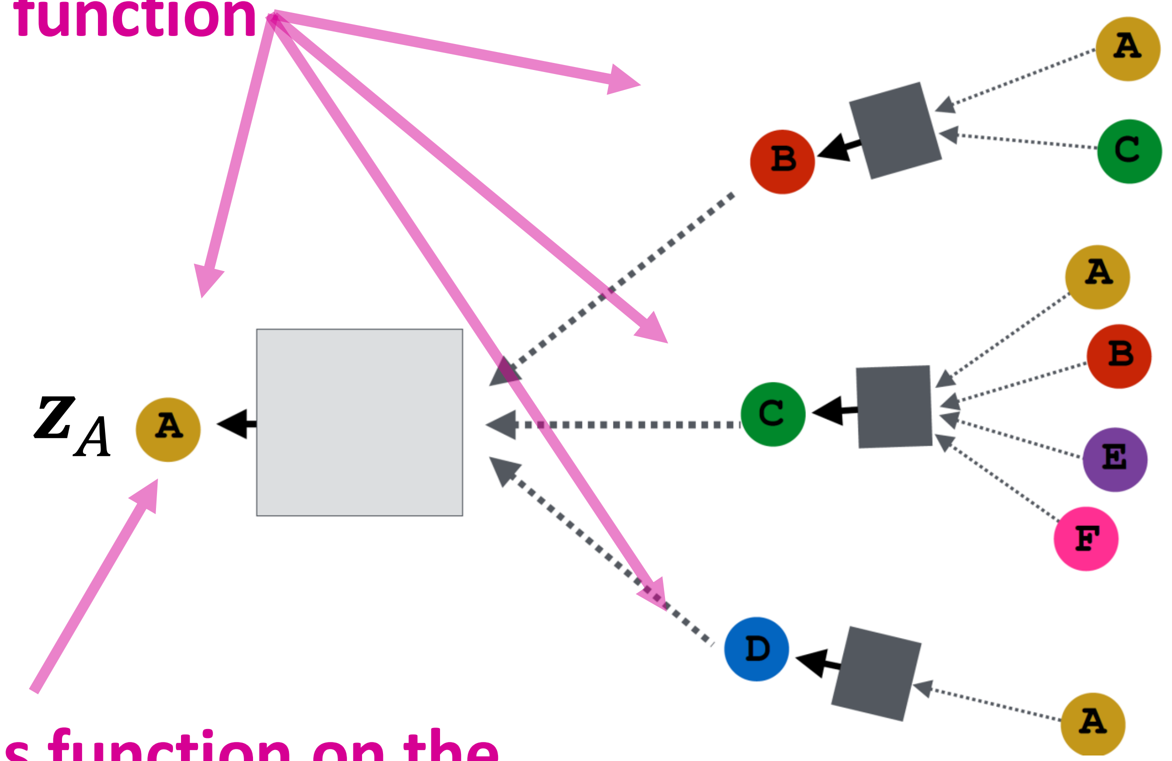
Safe or toxic drug?



Model Design: Overview (1)

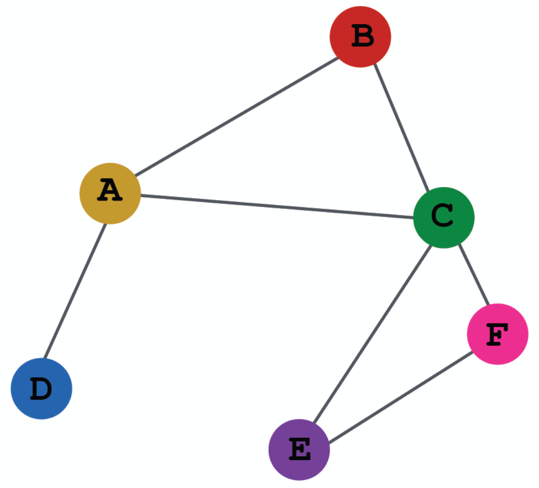


(1) Define a neighborhood aggregation function



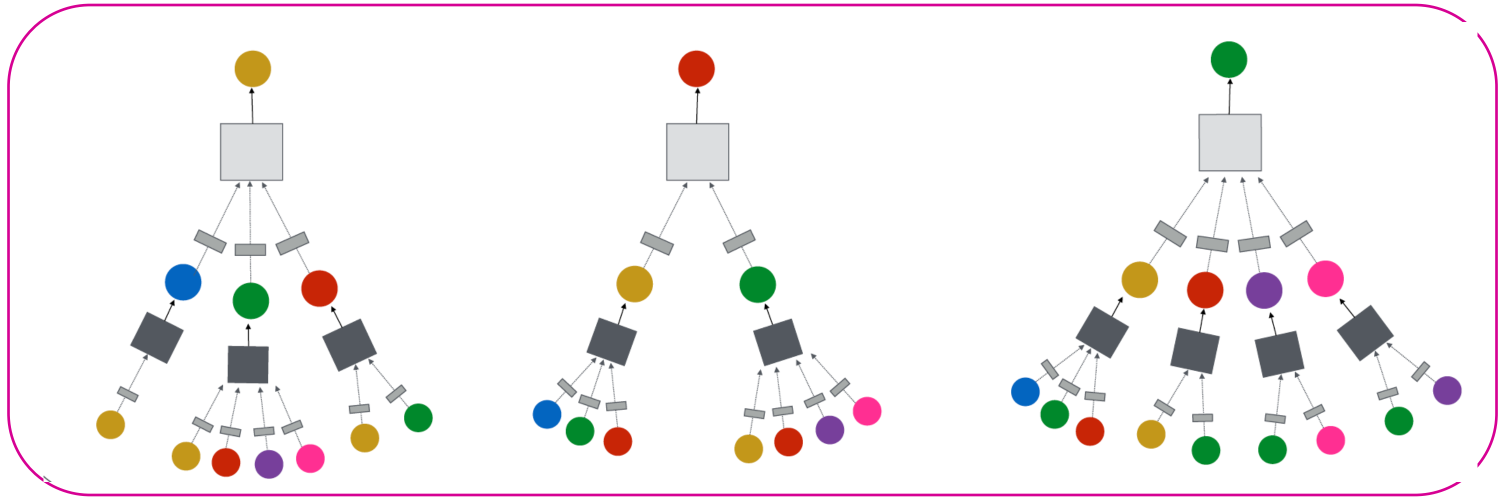
(2) Define a loss function on the embeddings

Model Design: Overview (2)

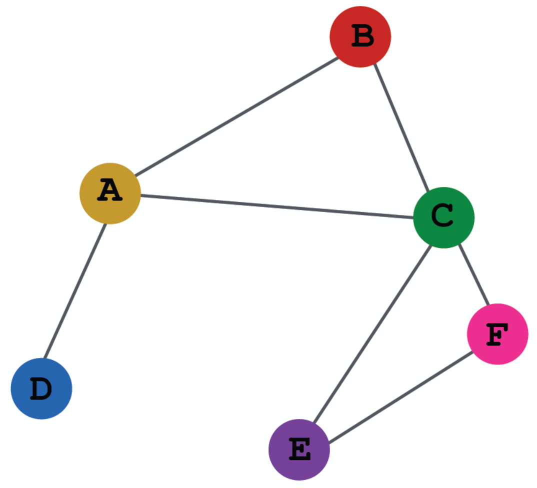


INPUT GRAPH

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



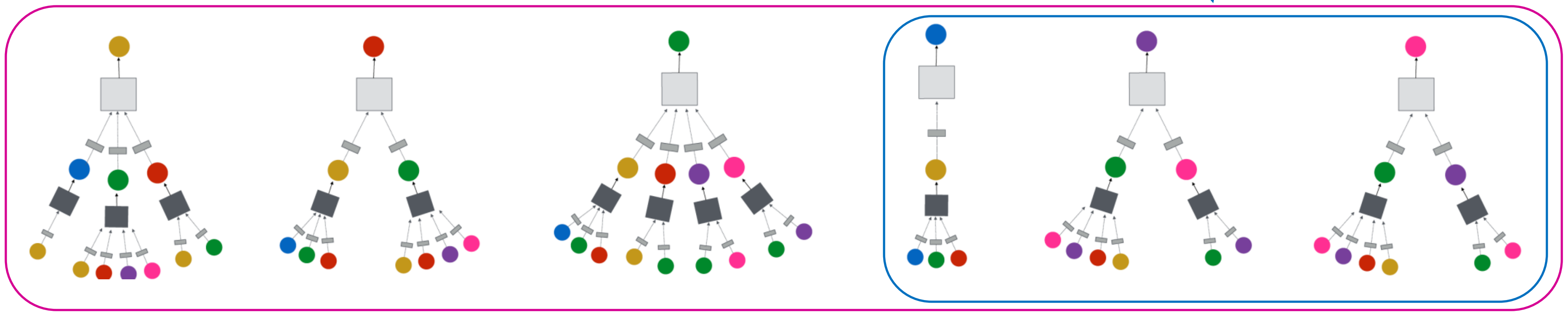
Model Design: Overview (3)



INPUT GRAPH

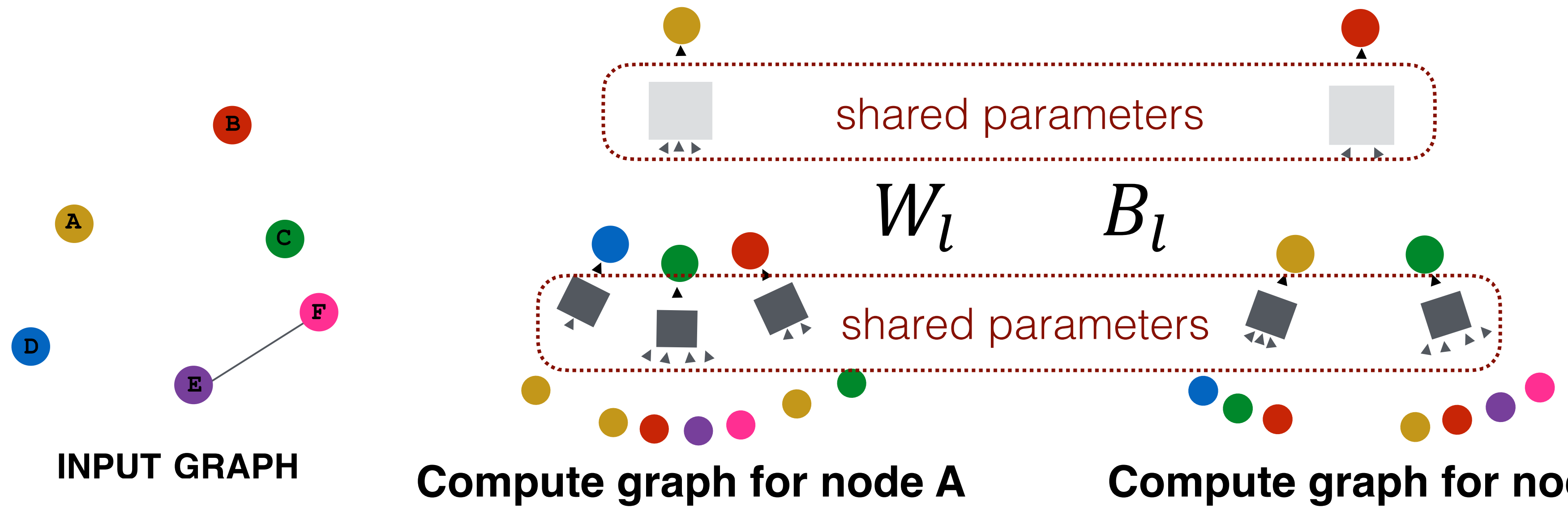
(4) Generate embeddings for nodes as needed

Even for nodes we never trained on!

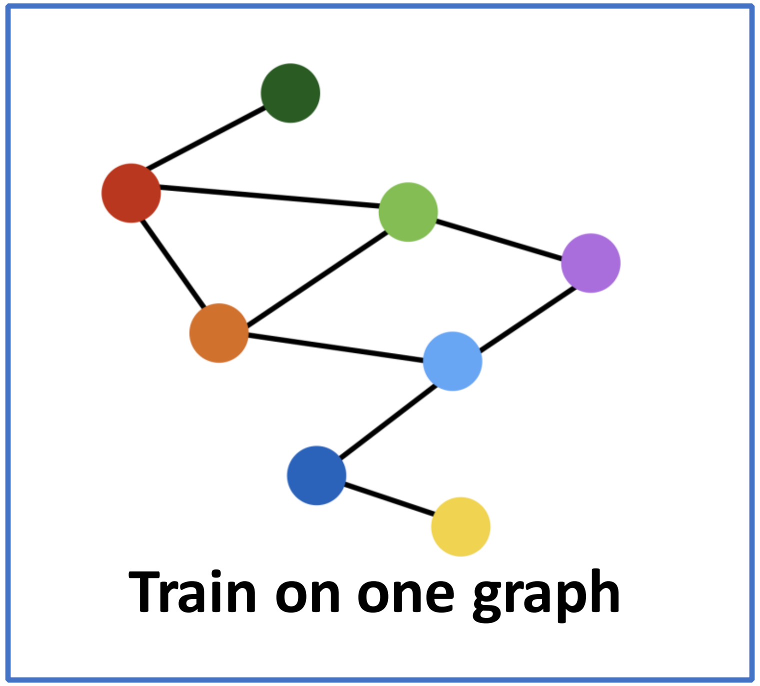


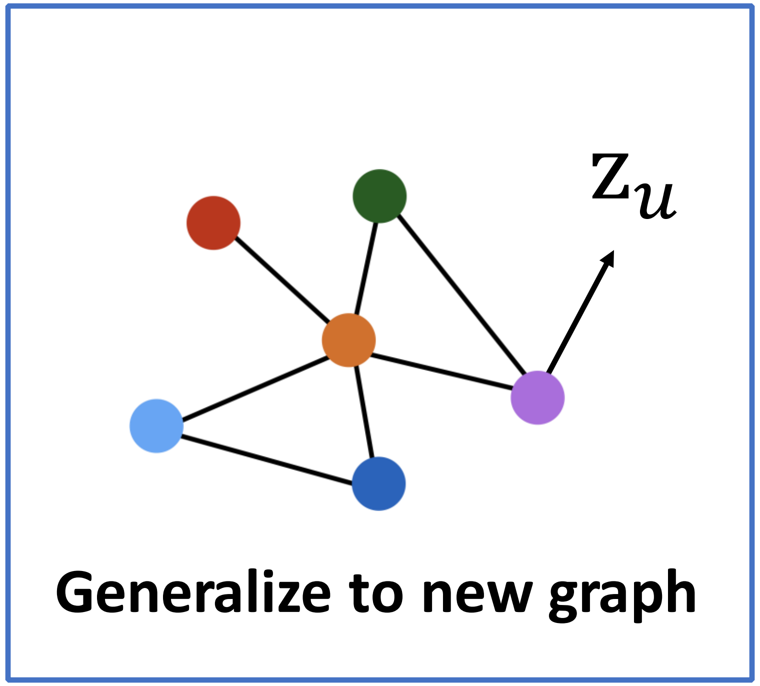
Inductive Capability

- **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**!



Inductive Capability: New Graphs

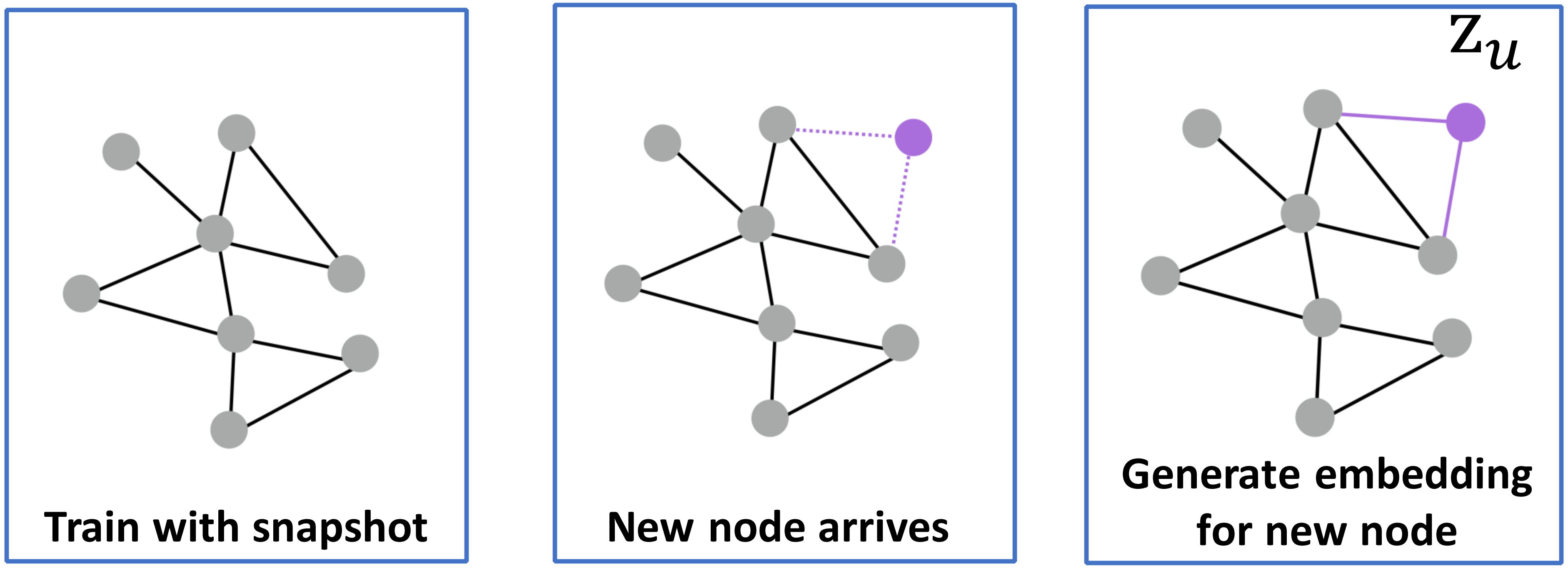




Inductive node embedding $\rightarrow$ Generalize to entirely unseen graphs

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

Inductive Capability: New Nodes



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

Summary

- **Recap: Graph Neural Networks (GNNs)** generates node embeddings by aggregating neighborhood information
  - Key distinctions between different architectures are in how they 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