Graph Neural Networks

Lecture 5

Danila Biktimirov
Applied Computer Science
Neapolis University Pafos

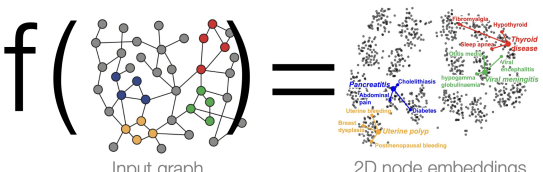
15 March 2025

Overview

1. **Previously on...**
2. **Deep Graph Encoders**
3. **Permutation Invariance**

Recap: Node Embeddings

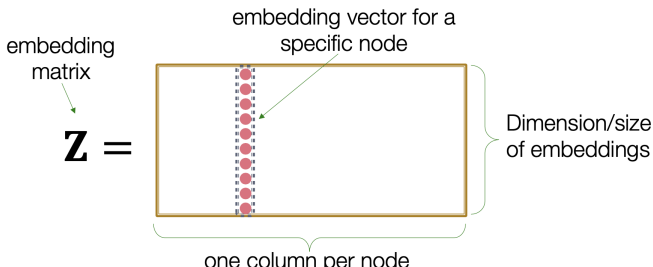
- **Intuition:** Map nodes to d-dimensional embeddings such that similar nodes in the graph are embedded close together.



How to learn mapping function f?

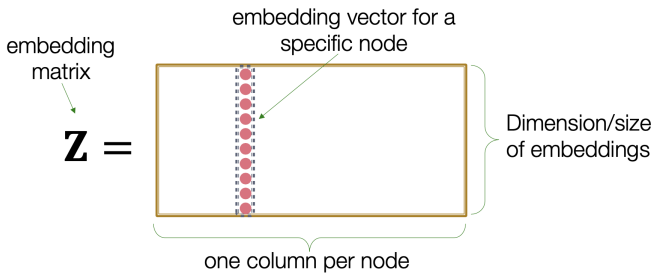
Recap: Node Embeddings

Goal: $\boxed{\textbf{similarity}(u, v)} \approx \mathbf{z}_v^\top \mathbf{z}_u$



Recap: "Shallow" Encoding

Simplest encoding approach: Encoder is just an embedding-lookup



Recap: Shallow Encoders

Limitations of shallow embedding methods:

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

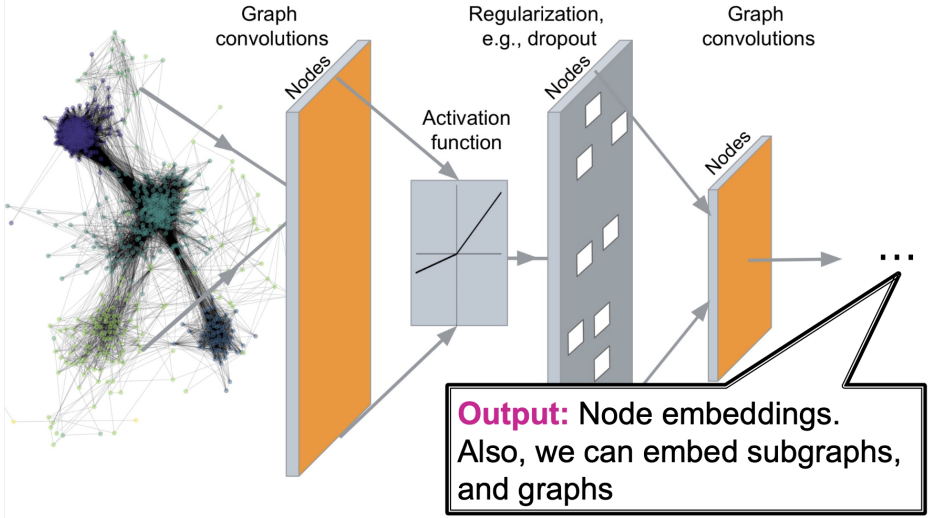
Today: Deep Graph Encoders

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

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

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

Deep Graph Encoders

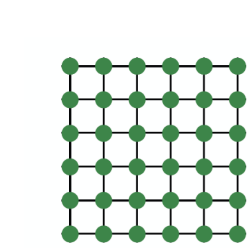


Tasks on Networks

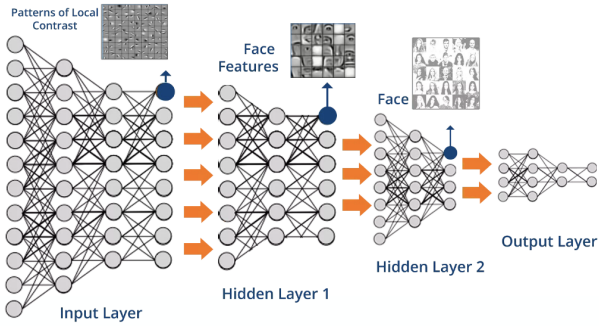
Tasks we will be able to solve:

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

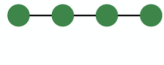
Modern ML Toolbox

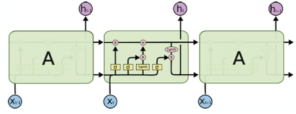


Images



Text/Speech



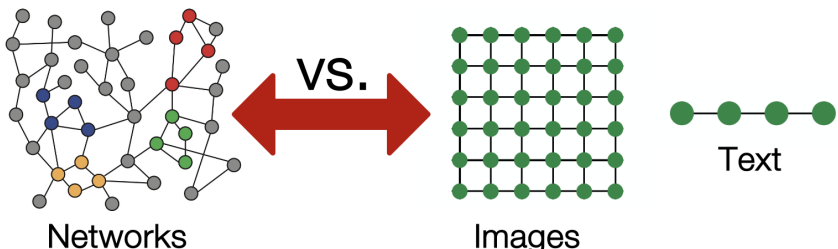


Modern deep learning toolbox is designed for simple sequences & grids

Why is it Hard?

But networks are far more complex!

- Arbitrary size and complex topological structure (i.e., no spatial locality like grids)
- No fixed node ordering or reference point
- Often dynamic and have multimodal features



Summary: Basics of Deep Learning

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

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

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

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

Content

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

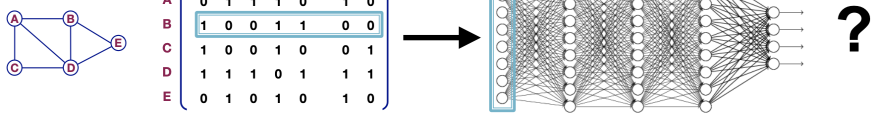
Setup

Assume we have a graph G:

- V is the **vertex set**
- A is the **adjacency matrix** (assume binary)
- $X \in \mathbb{R}^{|V| \times m}$ is a matrix of **node features**
- v: a node in V; $N(v)$: the set of neighbors of v.
- **Node features:**
  - Social networks: User profile, User image
  - Biological networks: Gene expression profiles, gene functional information
  - When there is no node feature in the graph dataset:
    - Indicator vectors (one-hot encoding of a node)
    - Vector of constant 1: [1, 1, ..., 1]

A Naive Approach

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

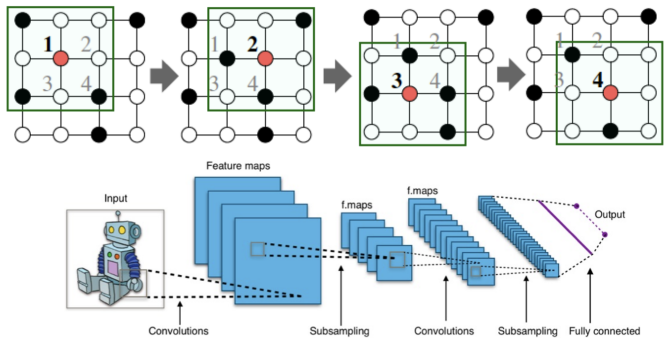


Issues with this idea:

- $O(|V|)$ parameters
- Not applicable to graphs of different sizes
- Sensitive to node ordering

Idea: Convolutional Networks

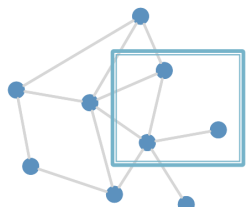
CNN on an image:



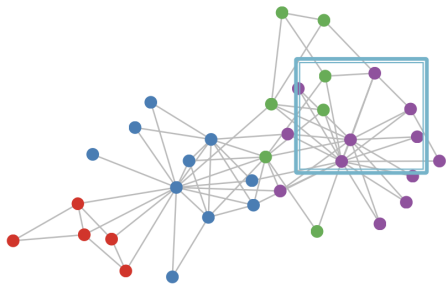
Goal is to generalize convolutions beyond simple lattices.
Leverage node features/attributes (e.g., text, images).

Real-World Graphs

But our graphs look like this:



or this:



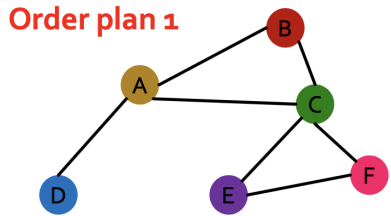
- There is no fixed notion of locality or sliding window on the graph
- Graph is **permutation invariant**

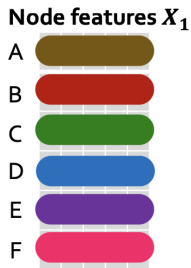
Permutation Invariance

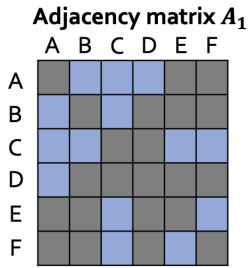
- **Consider we want to embed an entire graph**
- **Observation:** A graph does not have a canonical ordering of its nodes
  - We can have many different node orderings of the same graph.
- **What do we want:** If we learn an embedding function over a graph, we should get the same result (the same embedding)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

Permutation Invariance

Graph does not have a canonical ordering of the nodes!



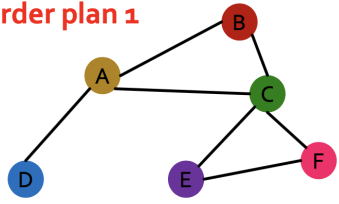




Permutation Invariance

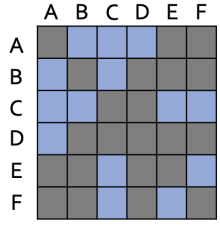
Graph does not have a canonical ordering of the nodes!

Order plan 1

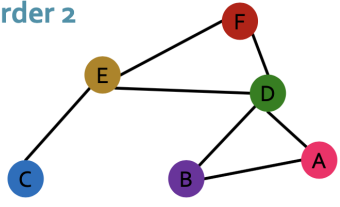


Node features X_1

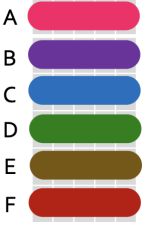
Adjacency matrix A_1



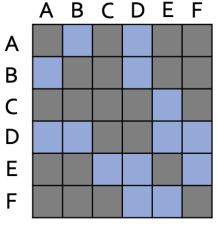
Order 2



Node features X_2



Adjacency matrix A_2



Permutation Invariance

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

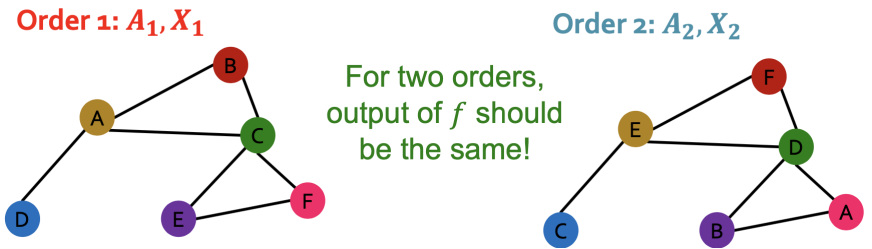
Permutation Invariance

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

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

$$f(A_1, \mathbf{X}_1) = f(A_2, \mathbf{X}_2)$$

In other words, f maps a graph to a d-dim embedding.
A is the adjacency matrix, $\mathbf{X}$ is the node feature matrix.



Permutation Invariance

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

- Consider we learn a function f that maps a graph $G = (A, \mathbf{X})$ to a vector $\mathbb{R}^d$. A is the adjacency matrix, $\mathbf{X}$ is the node feature matrix.
- Then, if

$$f(A_i, \mathbf{X}_i) = f(A_j, \mathbf{X}_j)$$

for any ordering i and j, we formally say f is a permutation invariant function. For a graph with $|V|$ nodes, there are $|V|!$ different orderings.

Definition: For any graph function $f : \mathbb{R}^{|V| \times |V|} \times \mathbb{R}^{|V| \times m} \rightarrow \mathbb{R}^d$, f is permutation invariant if

$$f(A, \mathbf{X}) = f(PAP^T, P\mathbf{X})$$

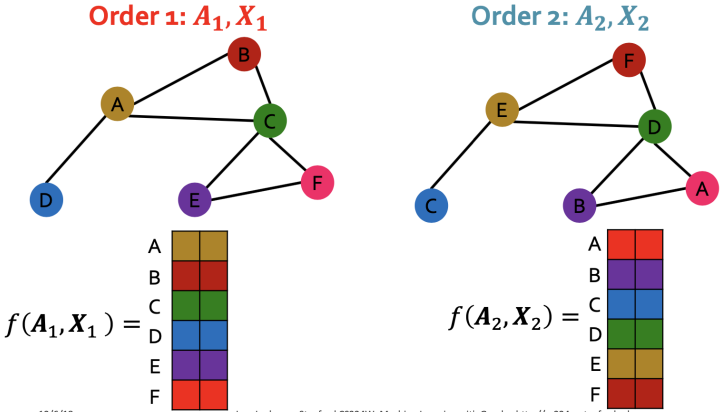
for any permutation P.

d... output embedding dimensionality of embedding the graph $G = (A, \mathbf{X})$.
Permutation P: a shuffle of the node order. Example: (A,B,C) $\rightarrow$ (B,C,A).

Permutation Equivariance

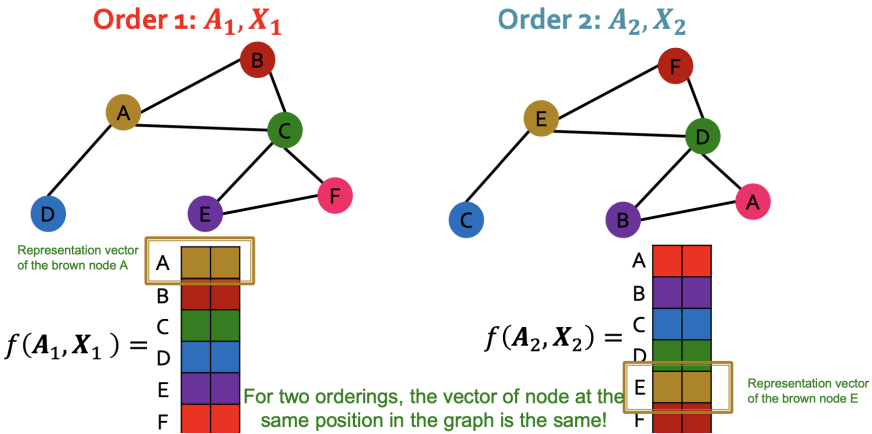
For node representation: We learn a function f that maps nodes of G to a matrix $\mathbb{R}^{|V| \times d}$.

In other words, each node in V is mapped to a d-dim embedding.



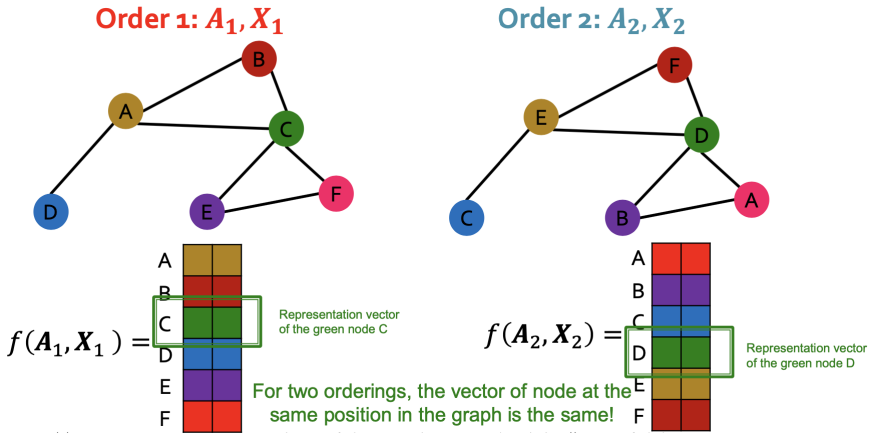
Permutation Equivariance

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



Permutation Equivariance

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



Permutation Equivariance

For node representation:

- Consider we learn a function f that maps a graph $G = (A, \mathbf{X})$ to a matrix $\mathbb{R}^{|V| \times d}$.
- If the output vector of a node at the same position in the graph remains unchanged for any ordering, we say f is permutation equivariant.

Definition: For any node function $f : \mathbb{R}^{|V| \times |V|} \times \mathbb{R}^{|V| \times m} \to \mathbb{R}^{|V| \times d}$, f is permutation-equivariant if

$$Pf(A, \mathbf{X}) = f(PAP^T, P\mathbf{X})$$

for any permutation P.
f maps each node in V to a d-dim embedding.

Summary: Invariance and Equivariance

Permutation-invariant

$$f(A, \mathbf{X}) = f(PAP^T, P\mathbf{X})$$

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

Permutation-equivariant

$$Pf(A, \mathbf{X}) = f(PAP^T, P\mathbf{X})$$

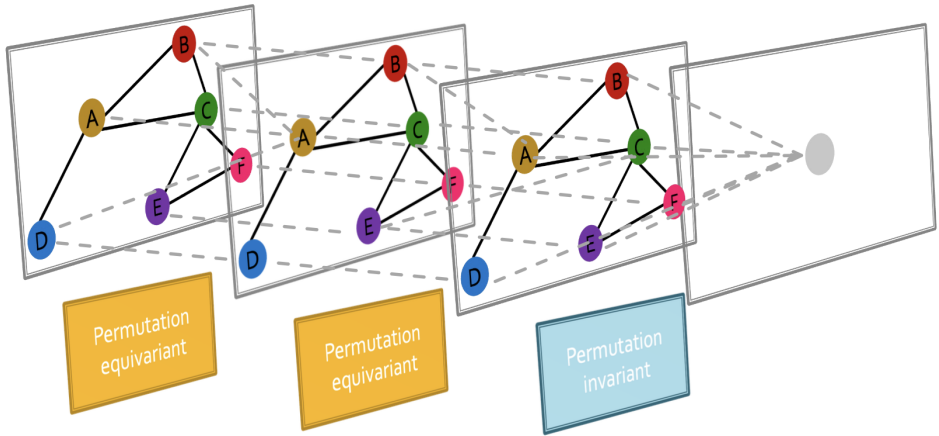
Permute the input, output also permutes accordingly. (map a graph to a matrix)

Examples:

- $f(A, \mathbf{X}) = \mathbf{1}^T\mathbf{X}$: Permutation-**invariant**
  - Reason: $f(PAP^T, P\mathbf{X}) = \mathbf{1}^T P\mathbf{X} = \mathbf{1}^T\mathbf{X} = f(A, \mathbf{X})$
- $f(A, \mathbf{X}) = \mathbf{X}$: Permutation-**equivariant**
  - Reason: $f(PAP^T, P\mathbf{X}) = P\mathbf{X} = Pf(A, \mathbf{X})$
- $f(A, \mathbf{X}) = A\mathbf{X}$: Permutation-**equivariant**
  - Reason: $f(PAP^T, P\mathbf{X}) = PAP^T P\mathbf{X} = PA\mathbf{X} = Pf(A, \mathbf{X})$

Graph Neural Network Overview

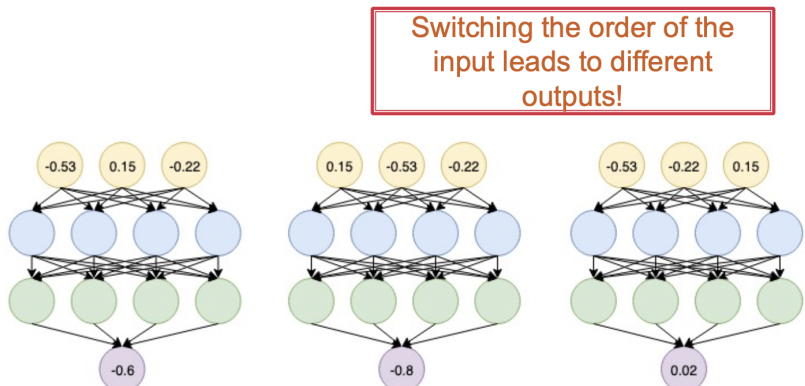
Graph neural networks consist of multiple permutation equivariant / invariant functions.



Graph Neural Network Overview

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

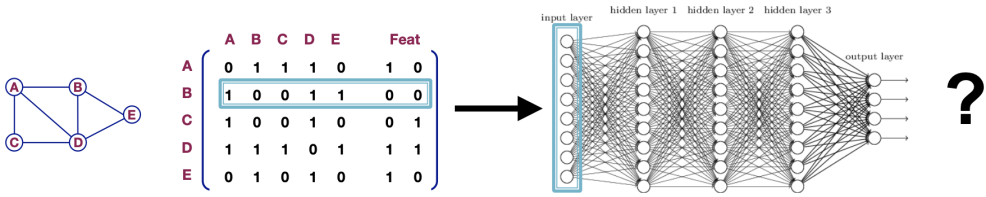
No.



Graph Neural Network Overview

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

No.



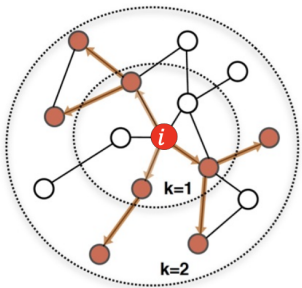
This explains why the naïve MLP approach fails for graphs!

Graph Neural Network Overview

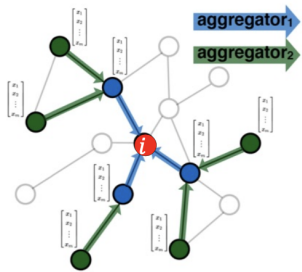
Next: Design graph neural networks that are permutation invariant / equivariant by passing and aggregating information from neighbors!

Graph Convolutional Networks

Idea: Node's neighborhood defines a computation graph



Determine node computation graph

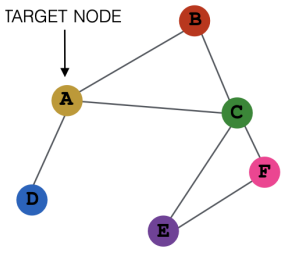


Propagate and transform information

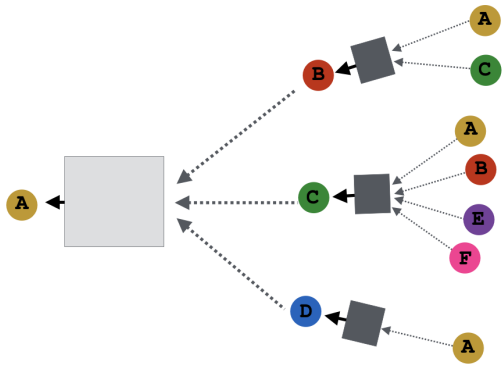
Learn how to propagate information across the graph to compute node features.

Idea: Aggregate Neighbors

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



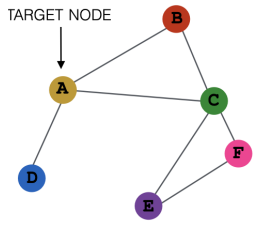
INPUT GRAPH



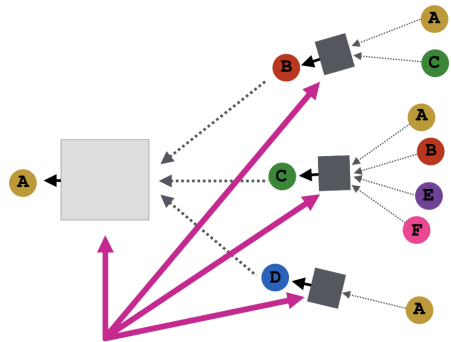
Idea: Aggregate Neighbors

Intuition: Nodes aggregate information from their neighbors using neural networks.

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



INPUT GRAPH



Neural networks

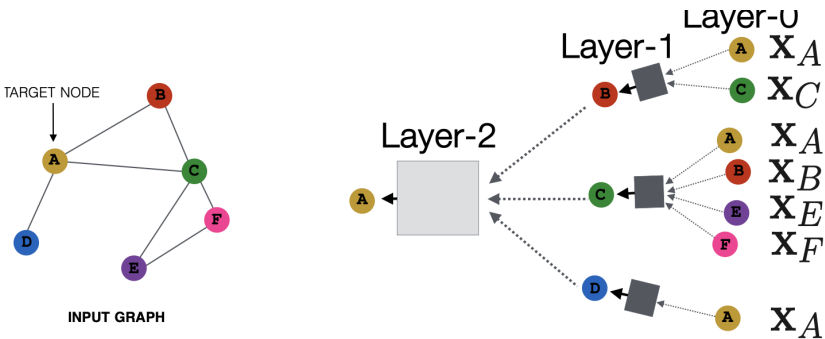
Idea: Aggregate Neighbors

Intuition: Network neighborhood defines a computation graph.
Every node defines a computation graph based on its neighborhood!

Deep Model: Many Layers

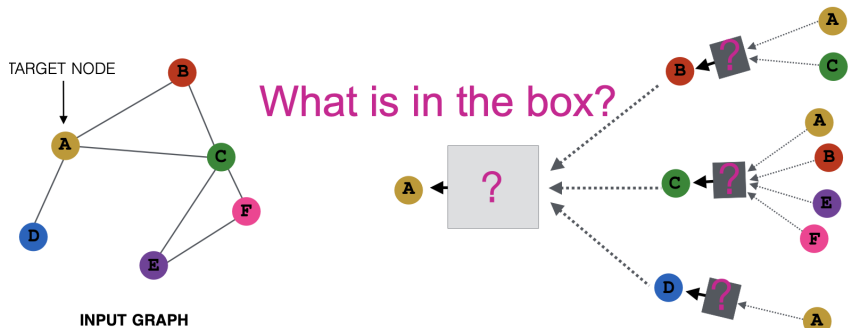
Model can be of arbitrary depth:

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



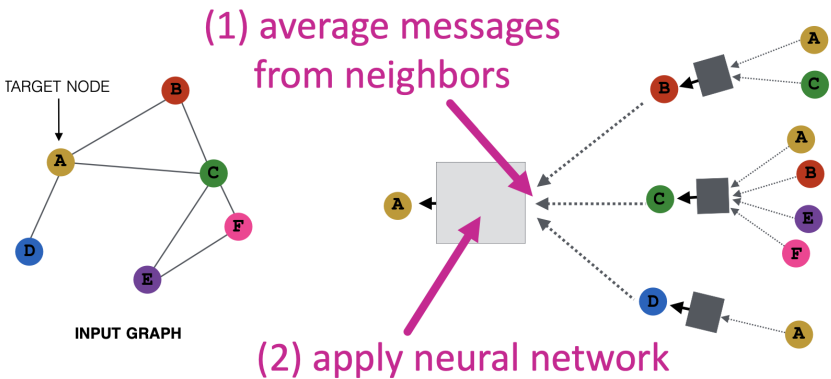
Neighborhood Aggregation

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



Neighborhood Aggregation

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



The Math: Deep Encoder of a GCN

Basic approach: Average neighbor messages and apply a neural network

- Initial 0-th layer embeddings are equal to node features: $h_v^0 = x_v$
- Update rule:

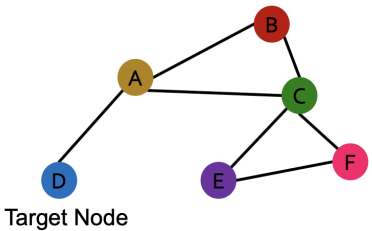
$$h_v^{(k+1)} = \sigma\left(W_k \sum_{u \in N(v)} \frac{h_u^{(k)}}{|N(v)|} + B_k h_v^{(k)}\right), \quad \forall k \in \{0, ..., K-1\}$$

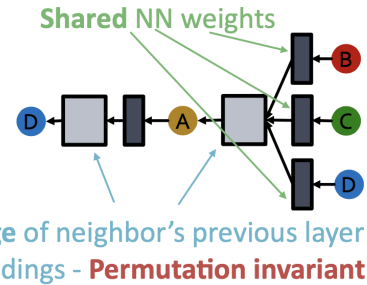
- **Final embedding:** $z_v = h_v^{(K)}$

GCN: Invariance and Equivariance

What are the invariance and equivariance properties for a GCN?

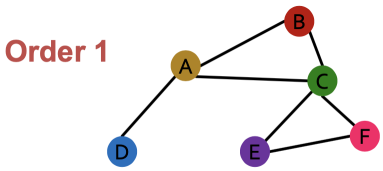
- **Given a node**, the GCN that computes its embedding is **permutation invariant**

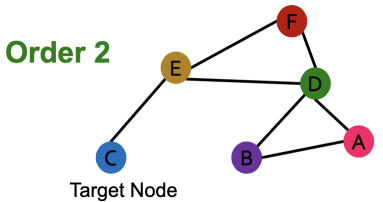


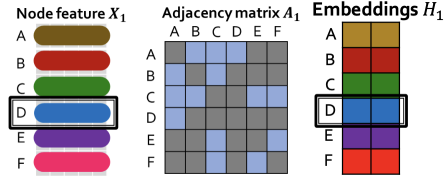


GCN: Invariance and Equivariance

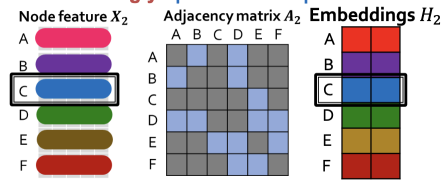
Considering all nodes in a graph, GCN computation is **permutation equivariant**.







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



GCN: Invariance and Equivariance

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

Detailed reasoning:

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

This is permutation equivariant.

Model Parameters

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

Matrix Formulation (1)

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

$$H^{(k)} = [h_1^{(k)}, \ldots, h_{|V|}^{(k)}]^T$$

$$\sum_{u \in N_v} h_u^{(k)} = A_{v,:} H^{(k)}$$

Let D be a diagonal matrix where $D_{v,v} = \text{Deg}(v) = |N(v)|$, then:

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

Matrix Formulation (2)

Re-writing update function in matrix form:

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

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

How to Train a GNN

Supervised setting: We want to minimize loss $\mathcal{L}$:

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

Unsupervised setting:

- No node label available
- **Use the graph structure as the supervision!**

Unsupervised Training

One possible idea: "Similar" nodes have similar embeddings:

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

where $y_{u,v} = 1$ when nodes u and v are similar.
CE is the cross entropy loss:

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

Supervised Training

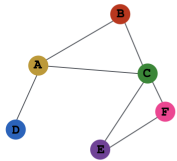
Directly train the model for a supervised task (e.g., node classification)
Use cross entropy loss:

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

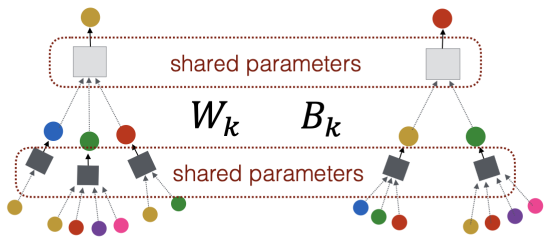
Inductive Capability

The same aggregation parameters are shared for all nodes:

- The number of model parameters is sublinear in $|V|$ and we can generalize to unseen nodes!



INPUT GRAPH



Compute graph for node A

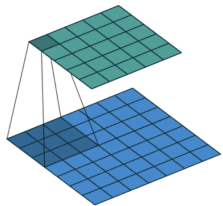
Compute graph for node B

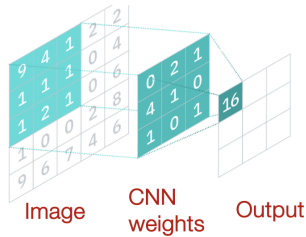
Architecture Comparison

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

Convolutional Neural Network

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





CNN formulation:

$$h_v^{(l+1)} = \sigma\left(\sum_{u \in N(v) \cup \{v\}} W_l^u h_u^{(l)}\right), \quad \forall l \in \{0, \ldots, L-1\} \tag{1}$$

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

GNN vs. CNN

GNN formulation:

$$h_v^{(l+1)} = \sigma\left(\mathbf{W_l} \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)}\right), \quad \forall l \in \{0, \ldots, L-1\} \tag{2}$$

CNN formulation (previous slide):

$$h_v^{(l+1)} = \sigma\left(\sum_{u \in N(v) \cup \{v\}} W_l^u h_u^{(l)}\right) \tag{3}$$

If we rewrite:

$$h_v^{(l+1)} = \sigma\left(\sum_{u \in N(v)} \mathbf{W_l^u} h_u^{(l)} + B_l h_v^{(l)}\right) \tag{4}$$

GNN vs. CNN

GNN formulation:

$$h_v^{(l+1)} = \sigma\left(\mathbf{W_l} \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)}\right), \quad \forall l \in \{0, \ldots, L-1\} \tag{5}$$

CNN formulation:

$$h_v^{(l+1)} = \sigma\left(\sum_{u \in N(v)} \mathbf{W_l^u} h_u^{(l)} + B_l h_v^{(l)}\right) \tag{6}$$

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

The End?