

## Contents

Overview .....	1
Stepup: vertex set, adjacency matrix, node features .....	1
Naive Approach: append node feaetures to adjacency matrix .....	1
3 problems: parameter size, graph size, node ordering .....	1
Adopt CNN .....	1
Problem .....	2
Solutions .....	2
Kipf & Welling, ICLR 2017 .....	2
Explain – transformation & aggregation .....	2
Explain – each node has a computational graph .....	2
Many layers – k-hops .....	2
Neighbourhood aggregation .....	2
Transformation choice – average embedding .....	3
Model Parameters: W, B; neighborhood aggregation, transformation .....	3
Matrix Formulation - efficient sparse matrix .....	3
Averging of neighbour embedding .....	3
Re-writing update function in matrix form .....	3
How to train GNN: supervised & unsupervised setting .....	3
Unsupervised Training .....	3
Supervised Training .....	3
Cross entropy loss .....	4
Model design .....	4
Step 1,2 - neighbour aggregation & loss function .....	4
Step 3 – train on a batch .....	4
Step 4 - enerate embeddings for nodes as needed .....	4
Inductive capcability .....	4
New graph .....	4
New nodes .....	4

## Overview

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

## Stepup: vertex set, adjacency matrix, node features

### 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:  $[1, 1, \dots, 1]$

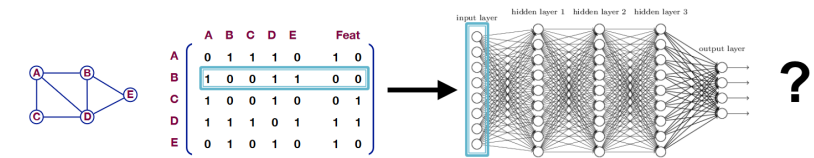
## Naive Approach: append node feaetures to adjacency matrix

### 3 problems: parameter size, graph size, node ordering

1. Parameter size
  - a. One training example per node but for each node there are  $N + X$  (node features) number of features
  - b. Training unstable / easy to overfit
2. Graph with different size
  - E.g. If 5 nodes, hard to fit in input size of 7
3. Node ordering
  - If the column order change, then the adjacency matrix changes
    - Rows & cols permuted thou the info is the same

- For images, the ordering can be top left pixel to bottom right
- but for graphs, there is no fix node ordering i.e. unclear how to sort the graph to put them as input in the matrix
- Has to be **invariant to node ordering**

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



### Issues with this idea:

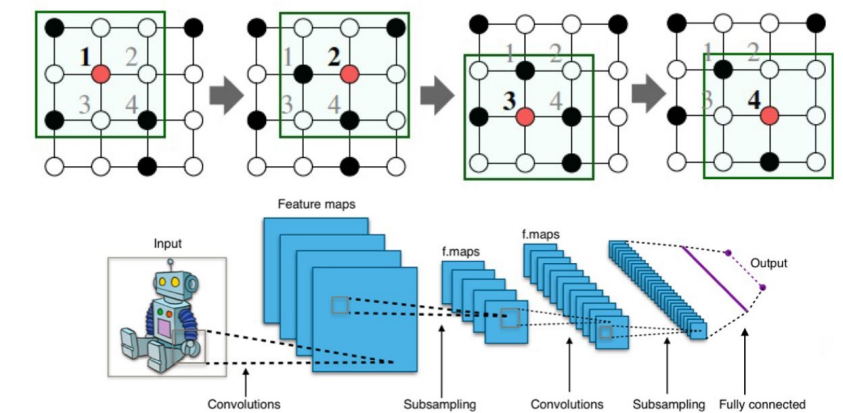
- $O(|V|)$  parameters
- Not applicable to graphs of different sizes
- Sensitive to node ordering

## Adopt CNN

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

**CNN = Sliding windows & locality.**

### CNN on an image:



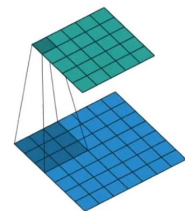
**Problem**

- no fixed notion of locality or sliding window on the graph
  - L: covers 3 nodes
  - R: covers more nodes
- Graph is permutation invariant

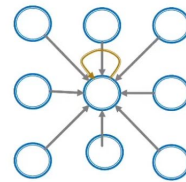
**Solutions**

- Aggregate information about a node based on its neighbouring nodes

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



Image



Graph

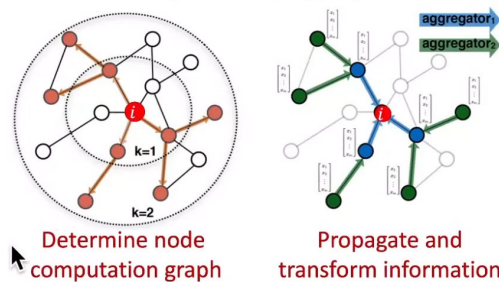
**Idea:** transform information at the neighbors and combine it:

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

**Kipf & Welling, ICLR 2017**

- Neighbour nodes takes the message from the node and propagate. Steps
  - Determine node computation graph
  - Propagate & transform information

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

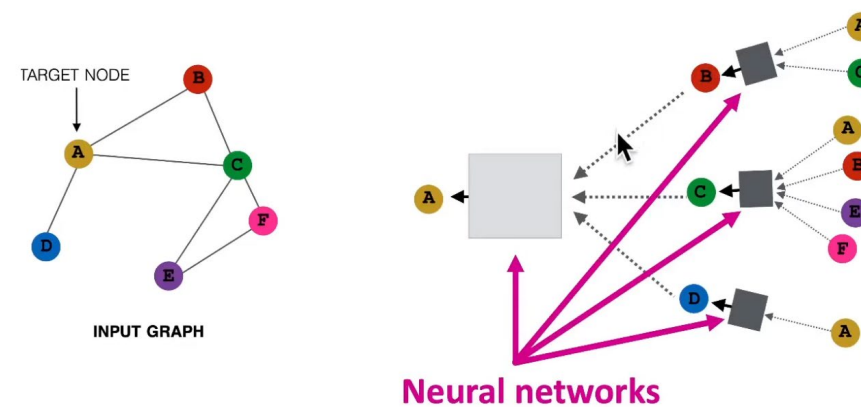


Learn how to propagate information across the graph to compute node features

Explain – transformation & aggregation

- To decide node A informatino, we collect from its neighbours {B, C, D}, in which their info are based on their neighbours.
- The message passing is then from the leaf to root
  - First, transform the info from leaf
  - Second, aggregate them in the parent node
  - Repeat

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

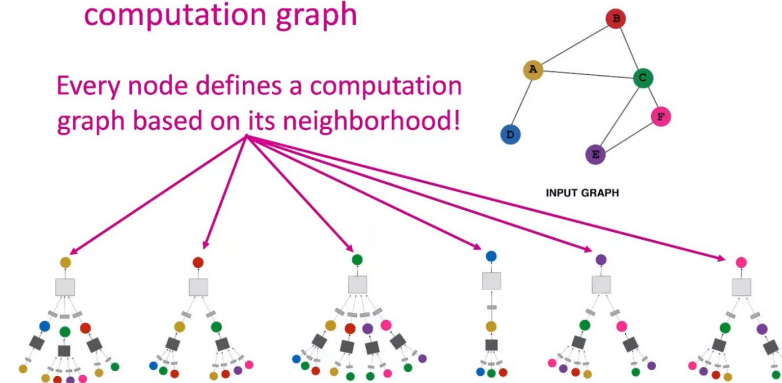


Explain – each node has a computational graph

- Every node has its own computation graph / architecture
- The structure depends on other structure
- Different to classical DL

- Intuition:** Network neighborhood defines a computation graph

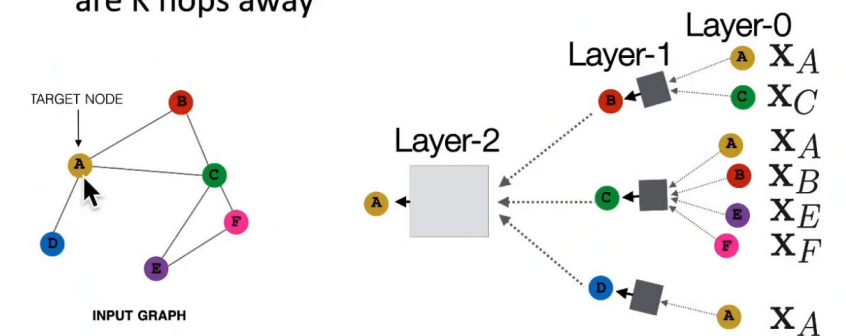
Every node defines a computation graph based on its neighborhood!



Many layers – k-hops

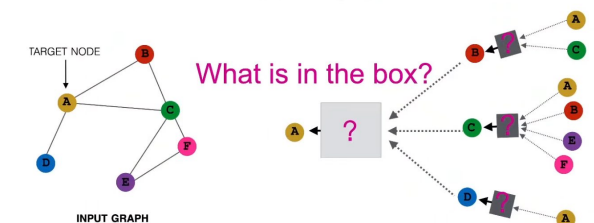
- Model can be of arbitrary depth:**

- 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

**Neighbourhood aggregation**

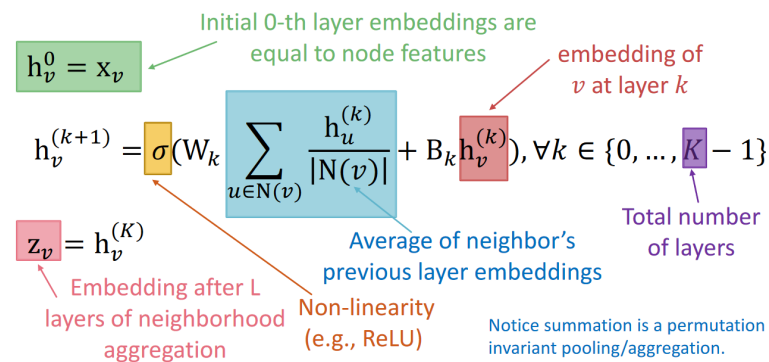
- The aggregation result is the same regardless of the ordering

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



### Transformation choice – average embedding

- Ordering invariant
- Examples
  - Average messages
  - Apply NN
    - Apply linear transformation
    - Follow by non-linearity
- Transform current layer features + aggregated previous child nodes messages



### Model Parameters: W, B; neighborhood aggregation, transformation

$$h_v^{(0)} = x_v$$

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

$$z_v = h_v^{(K)}$$

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

Final node embedding

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

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

## Matrix Formulation - efficient sparse matrix

### Averging of neighbour embedding

- Many aggregations can be performed **efficiently** by (sparse) matrix operations
- Then
  - Node embedding is the average of the neighbour embedding
  - i.e. the adjacency matrix \* embedding spaces at a layer
- in short
  - averaging of neighbour embedding
    - summing and averaging can be rewritten as
    - **matrix multiplication (dot product)**
  - the drawing is what correspond to what

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

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

### Re-writing update function in matrix form

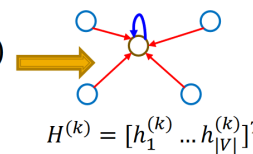
- In practice, this implies that **efficient sparse matrix** multiplication can be used ( $\tilde{A}$  is sparse)
- Note: not all GNNs can be expressed in matrix form, when aggregation function is **complex**

### Re-writing update function in matrix form:

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

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

- Red: neighborhood aggregation
- Blue: self transformation



## How to train GNN: supervised & unsupervised setting

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

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

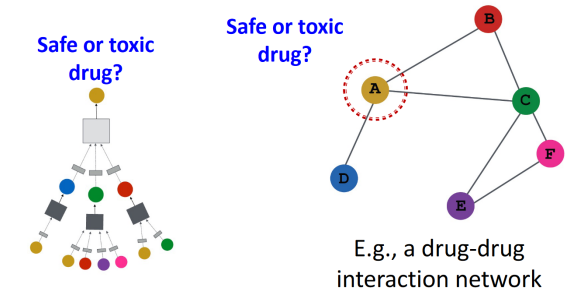
- $\mathbf{y}$ : node label
- $\mathcal{L}$  could be L2 if  $\mathbf{y}$  is real number, or cross entropy if  $\mathbf{y}$  is categorical
- **Unsupervised setting**:
  - No node label available
  - Use the graph structure as the supervision!

### Unsupervised Training

- No labels
- “Similar” nodes have similar embeddings
 
$$\mathcal{L} = \sum_{z_u, z_v} \text{CE}(\mathbf{y}_{u,v}, \text{DEC}(z_u, z_v))$$
  - Where  $\mathbf{y}_{u,v} = 1$  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)





## Cross entropy loss

- If label is 1, want output to be 1
- If 0, want 0

■ Use cross entropy loss (Slide 16)

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

Encoder output:  
node embedding

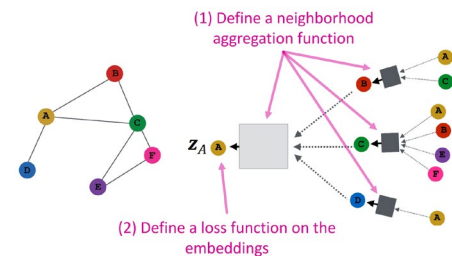
Classification  
weights

Node class  
label

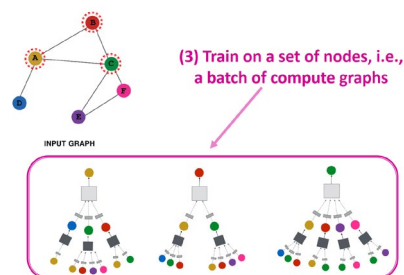
Safe or toxic drug?

## Model design

## Step 1,2 - neighbour aggregation &amp; loss function

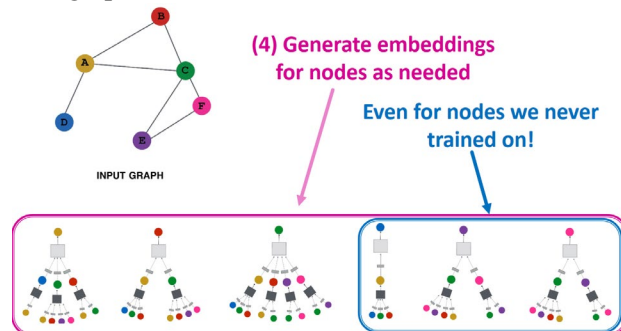


## Step 3 – train on a batch



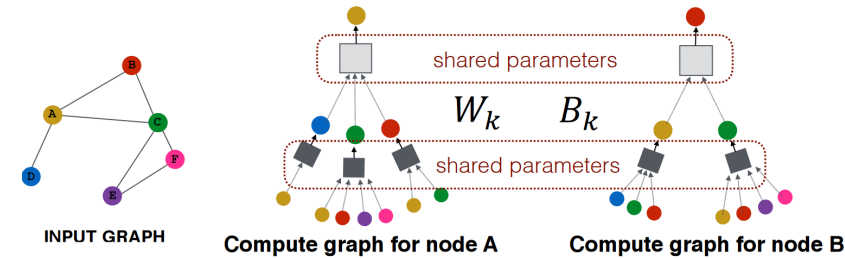
## Step 4 - enerate embeddings for nodes as needed

- **Generalisability.** Train embedding for one graph and transfer to another graph

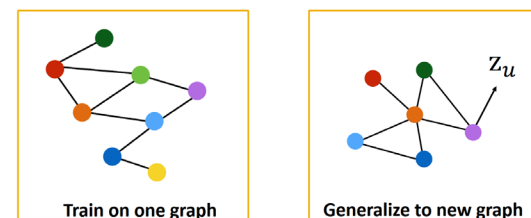


## Inductive capability

- The same aggregation parameters are **shared** for all nodes
- The number of model parameters is **sublinear** in  $|V|$
- # model parameters **W & B**, depends on
  - #features / embedding dimensionality (since shared)
  - not the size of graph (#nodes)
- Thus, generalize to **unseen** nodes



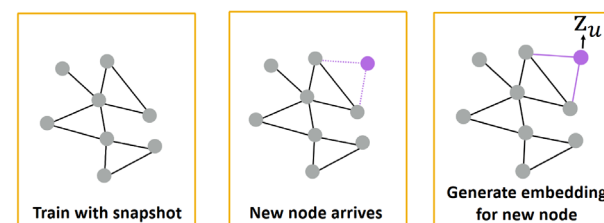
## New graph



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

## New nodes

- One forward pass can generate new embedding for the new node as the graph evolving



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