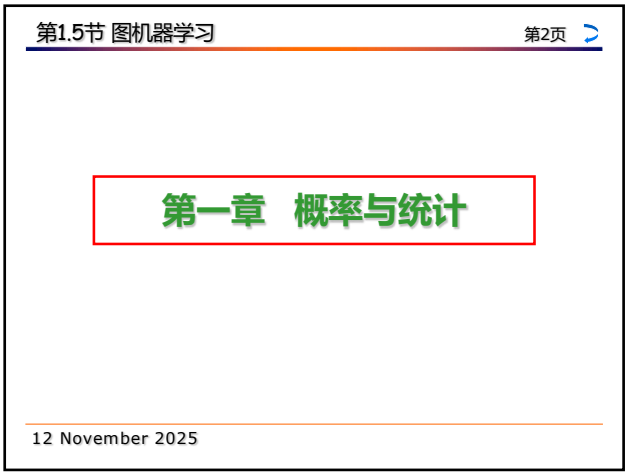
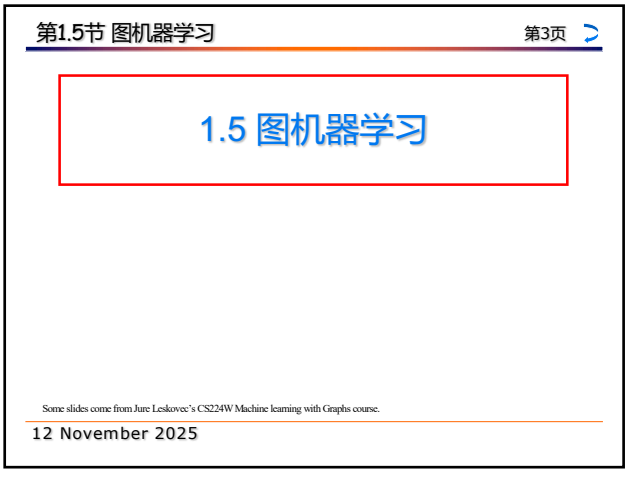

计算机应用数学
Applied Mathematics for Computer Science
主讲教师：宋骐
https://songqi1990.github.io/
中国科学技术大学
University of Science and Technology of China

1

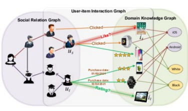

第1.5节 图机器学习
第2页
第一章 概率与统计
12 November 2025

2

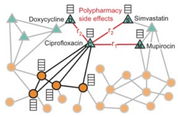

第1.5节 图机器学习
第3页
1.5 图机器学习
Some slides come from Jure Leskovec's CS224W Machine learning with Graphs course.
12 November 2025

3

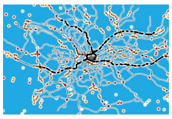
Graph Applications



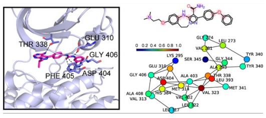
Recommendation System



Drug Side Effect



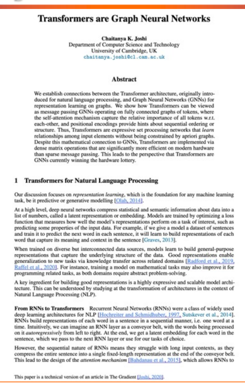
Traffic Prediction



Drug Discovery

Anomaly Detection

4



Transformers are Graph Neural Networks

Chaitanya K. Joshi
Department of Computer Science and Technology
University of Cambridge, UK
chaitanya.joshi@cl.cam.ac.uk

Abstract

We establish connections between the Transformer architecture, originally introduced for natural language processing, and Graph Neural Networks (GNNs) for representation learning on graphs. We show how Transformers can be viewed as message passing GNNs operating on fully connected graphs of tokens, where the self-attention mechanism capture the relative importance of all tokens w.r.t. each-other, and positional encodings provide hints about sequential ordering or structure. Thus, Transformers are expressive set processing networks that *learn* relationships among input elements without being constrained by apriori graphs. Despite this mathematical connection to GNNs, Transformers are implemented via dense matrix operations that are significantly more efficient on modern hardware than sparse message passing. This leads to the perspective that Transformers are GNNs currently winning the hardware lottery.

1 Transformers for Natural Language Processing

Our discussion focuses on *representation learning*, which is the foundation for any machine learning task, be it predictive or generative modelling [Olah, 2014].

At a high level, deep neural networks compress statistical and semantic information about data into a list of numbers, called a latent representation or embedding. Models are trained by optimizing a loss function that measures how well the model's representations perform on a task of interest, such as predicting some properties of the input data. For example, if we give a model a dataset of sentences and train it to predict the next word in each sentence, it will learn to build representations of each word that capture its meaning and context in the sentence [Graves, 2013].

When trained on diverse but interconnected data sources, models learn to build general-purpose representations that capture the underlying structure of the data. Good representations enable generalization to new tasks via knowledge transfer across related domains [Radford et al., 2019, Raffel et al., 2020]. For instance, training a model on mathematical tasks may also improve it for programming related tasks, as both domains require abstract problem-solving.

A key ingredient for building good representations is a highly expressive and scalable model architecture. This can be understood by studying at the transformation of architectures in the context of Natural Language Processing (NLP).

From RNNs to Transformers Recurrent Neural Networks (RNNs) were a class of widely used deep learning architectures for NLP [Hochreiter and Schmidhuber, 1997, Sutskever et al., 2014]. RNNs build representations of each word in a sentence in a sequential manner, i.e. one word at a time. Intuitively, we can imagine an RNN layer as a conveyor belt, with the words being processed on it *autoregressively* from left to right. At the end, we get a latent embedding for each word in the sentence, which we pass to the next RNN layer or use for our tasks of choice.

However, the sequential nature of RNNs means they struggle with long input contexts, as they compress the entire sentence into a single fixed-length representation at the end of the conveyor belt. This lead to the design of the *attention mechanism* [Bahdanau et al., 2015], which allows RNNs to

This paper is a technical version of an article in The Gradient [Joshi, 2020].

5

Learning with graphs

- 1. Generating Graph Representation
  - 3.1 Traditional Methods
  - 3.1 Random Walk - based Methods
- 2. Deep Learning for Graphs
- 3. Graph Neural Networks
  - 3.1 GNN Architecture
  - 3.2 Optimization and Application

6

Learning with graphs

- 1. Generating Graph Representation
  - 3.1 Traditional Methods
  - 3.1 Random Walk - based Methods
- 2. Deep Learning for Graphs
- 3. Graph Neural Networks
  - 3.1 GNN Architecture
  - 3.2 Layers of GNN
  - 3.3 Training GNNs
  - 3.4 Expressiveness of GNN

7

Graph Representation

- Some nodes may carry some information
  - age, income of a person in a social network
  - price, weight, size of a product in a product knowledge graph
- We can also generate node features - characterize the structure and position of a node in the network (Traditional methods)
  - Node degree
  - Node centrality
  - Clustering coefficient
  - Graphlets
- Random Walk – based methods
  - Node2vec, LINE

8

Node Centrality

- Node degree counts the neighboring nodes without capturing their importance.
- Node centrality c_v takes the node importance in a graph into account
- Different ways to model importance:
  - Engienvector centrality
  - Betweenness centrality
  - Closeness centrality
  - and many others…

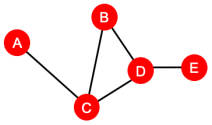
9

Betweenness centrality

- A node is important if it lies on many shortest paths between other nodes

$$c_v = \sum_{s \neq v \neq t} \frac{\#(\text{shortest paths betwen } s \text{ and } t \text{ that contain } v)}{\#(\text{shortest paths between } s \text{ and } t)}$$

- Example:



$c_A = c_B = c_E = 0$
$c_C = 3$
(A-C-B, A-C-D, A-C-D-E)

$c_D = 3$
(A-C-D-E, B-D-E, C-D-E)

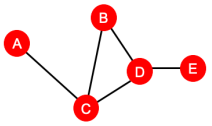
10

Closeness centrality

- A node is important if it has small shortest path lengths to all other nodes

$$c_v = \frac{1}{\sum_{u \neq v} \text{shortest path length between } u \text{ and } v}$$

- Example:



$c_A = 1/(2+1+2+3) = 1/8$
(A-C-B, A-C, A-C-D, A-C-D-E)

$c_D = 1/(2+1+1+1) = 1/5$
(D-C-A, D-B, D-C, D-E)

11

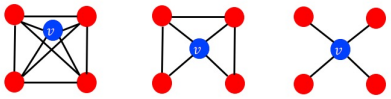
Clustering Coefficient

- Measures how connected v's neighboring nodes are:

$$e_v = \frac{\#(\text{edges among neighboring nodes})}{\binom{k_v}{2}} \in [0,1]$$

$\#(\text{node pairs among } k_v \text{ neighboring nodes})$
In our examples below the denominator is 6 (4 choose 2).

Examples:

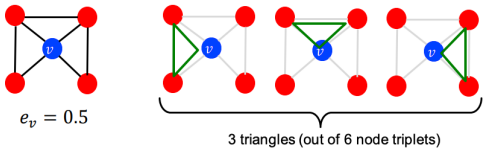


$e_v = 1$ $e_v = 0.5$ $e_v = 0$

12

Graphlets

- Clustering coefficient counts the #(triangles) in the ego-network

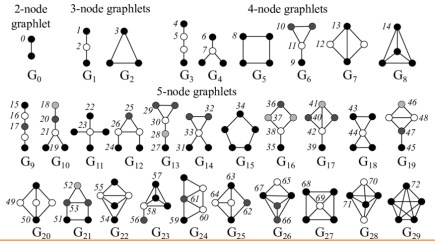


- We can generalize the above by counting #(pre-specified subgraphs, i.e., graphlets)

13

Graphlets

- Graphlets of size 2-5 nodes - a vector of 73 coordinates is a signature of a node that describes the topology of node's neighborhood

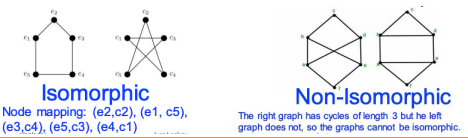


14

Graphlets

- Induced subgraph is another graph, formed from a subset of vertices and all of the edges connecting the vertices in that subset

- Graph Isomorphism: Two graphs which contain the same number of nodes connected in the same way are said to be isomorphic.

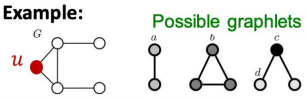


15

Graphlets - Example

- Graphlet Degree Vector (GDV): A count vector of graphlets rooted at a given node

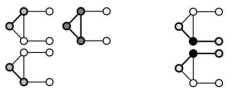
Example:



Possible graphlets up to size 3

Graphlet instances of node u:

a b c d



GDV of node u:
a, b, c, d
[2,1,0,2]

16

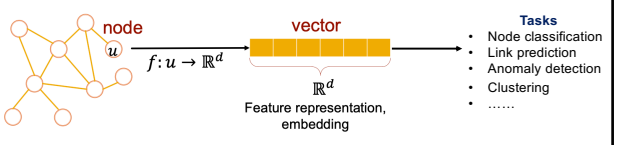
Learning with graphs

- 1. Generating Graph Representation
  - 3.1 Traditional Methods
  - 3.1 Random Walk - based Methods
- 2. Deep Learning for Graphs
- 3. Graph Neural Networks
  - 3.1 GNN Architecture
  - 3.2 Layers of GNN
  - 3.3 Training GNNs
  - 3.4 Expressiveness of GNN

17

Graph Embedding

- Task: Map nodes into an embedding space
  - Similarity of embeddings between nodes indicates their similarity in the network, e.g. both nodes are close to each other (connected by an edge)
  - Encode network information
  - Potentially used for many downstream predictions



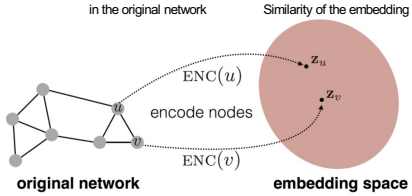
18

Embedding Nodes

- Goal is to encode nodes so that similarity in the embedding space (e.g., dot product) approximates similarity in the graph

$$similarity(u, v) \approx z_v^T z_u$$

in the original network — Similarity of the embedding



19

Learning Node Embeddings

- Encoder-decoder framework
- Encoder maps from nodes to embeddings
- Define a node similarity function (i.e., a measure of similarity in the original network)
- Decoder **DEC** maps from embeddings to the similarity score
- Optimize the parameters of the encoder so that:

$$similarity(u, v) \approx z_v^T z_u$$

in the original network — Similarity of the embedding

20

Encoder and Similarity Function

- Encoder: maps each node to a low-dimensional vector

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

$\mathbf{z}_v$: d-dimensional embedding

v: node in the input graph

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

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

Similarity of u and v in the original network

Decoder

dot product between node embeddings

21

Node2Vec

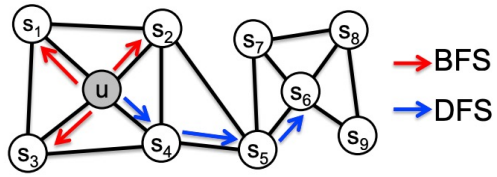
- Goal: Embed nodes with similar network neighborhoods close in the feature space.
- Frame this goal as a maximum likelihood optimization problem, independent to the downstream prediction task.
- Key observation: Flexible notion of network neighborhood $N_R(u)$ of node u leads to rich node embeddings
- Develop biased 2nd order random walk R to generate network neighborhood $N_R(u)$ of node

Grover, Aditya, and Jure Leskovec. "node2vec: Scalable feature learning for networks." KDD. 2016.

22

Node2Vec – Biased Walks

- Use flexible, biased random walks that can trade off between local and global views of the network
- Two classic strategies to define a neighborhood $N_R(u)$ of a given node u :



$N_{BFS}(u) = \{s_1, s_2, s_3\}$ $N_{DFS}(u) = \{s_4, s_5, s_6\}$

23

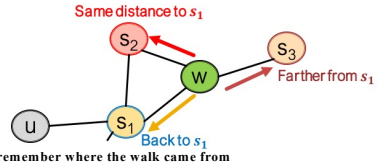
Node2Vec – Interpolating BFS and DFS

- Biased fixed-length random walk R that given a node u generates neighborhood N_R(u)
- Return parameter p: return back to the previous node
- In-out parameter q: moving outwards (DFS) vs. inwards (BFS). q is the 'ratio' of BFS vs. DFS

24

Node2Vec – Biased Random Walks

- Biased 2nd -order random walks explore network neighborhoods:
  - Random walk just traversed edge *(s1, w)* and is not at *w*
  - Insight: Neighbors of *w* can only be

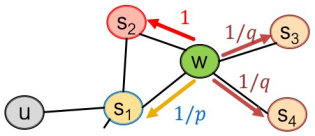


  - Idea: remember where the walk came from

25

Node2Vec – Biased Random Walks

- Walker came over edge (**s1**, **w**) and is at **w**.
- Where to go next?
  - p,q model transition probabilities
  - p: return parameter
  - q: walk away parameter

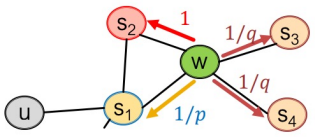


Note: unnormalized probabilities

26

Node2Vec – Biased Random Walks

- Walker came over edge (**s1**, **w**) and is at **w**.
- Where to go next?
  - BFS-like walk: low value of p
  - DFS-like walk: low value of q



w →

Target t	Prob.	Dist. (s_1, t)
s_1	$1/p$	0
s_2	1	1
s_3	$1/q$	2
s_4	$1/q$	2

27

Node2Vec – Algorithm

- 1) Compute random walk probabilities
- 2) Simulate r random walks of length l starting from each node u
- 3) Optimize the node2vec objective using Stochastic Gradient Descent

Algorithm 1 The *node2vec* algorithm.

LearnFeatures (Graph $G = (V, E, W)$, Dimensions d, Walks per node r, Walk length l, Context size k, Return p, In-out q)
π = PreprocessModifiedWeights(G, p, q)
$G' = (V, E, \pi)$
Initialize $walks$ to Empty
for $iter = 1$ **to** r **do**
  for all nodes $u \in V$ **do**
    $walk$ = node2vecWalk(G', u, l)
    Append $walk$ to $walks$
f = StochasticGradientDescent(k, d, $walks$)
return f

node2vecWalk (Graph $G' = (V, E, \pi)$, Start node u, Length l)
Inititalize $walk$ to $[u]$
for $walk\_iter = 1$ **to** l **do**
  $curr = walk[-1]$
  V_{curr} = GetNeighbors($curr$, G')
  s = AliasSample(V_{curr}, π)
  Append s to $walk$
return $walk$

28

Struc2vec

- Novel framework for node representations based on structural identity
- Structural similarity does not depend on hop distance
  - Neighbor nodes can be different, far away nodes can be similar
- Structural identity as a hierarchical concept
  - Depth of similarity varies
- Flexible four step procedure
  - Operational aspect of steps are flexible

Ribeiro, Leonardo FR, Pedro HP Saverese, and Daniel R. Figueiredo. "struc2vec: Learning node representations from structural identity." KDD. 2017.

29

Struc2vec - Structural Similarity

- Hierarchical measure for structural similarity between two nodes
- $R_k(u)$: set of nodes at distance k from u (ring)
- s(S): ordered degree sequence of set S

$s(R_0(u)) = 4$ $s(R_1(u)) = 1,3,4,4$ $s(R_2(u)) = 2,2,2,2$

$s(R_0(v)) = 3$ $s(R_1(v)) = 4,4,4$ $s(R_2(v)) = 1,2,2,2,2$

30

Struc2vec - Structural Similarity

- $g(D_1,D_2)$: distance between two ordered sequences
  - cost of pairwise alignment: max(a,b) / min(a,b) -1
  - optimal alignment by Dynamic Time Warping (DTW) in our framework

$s(R_0(u)) = 4$	$s(R_1(u)) = 1,3,4,4$	$s(R_2(u)) = 2,2,2,2$
$s(R_0(v)) = 3$	$s(R_1(v)) = 4,4,4$	$s(R_2(v)) = 1,2,2,2,2$
$g(.,.) = 0.33$	$g(.,.) = 3.33$	$g(.,.) = 1$

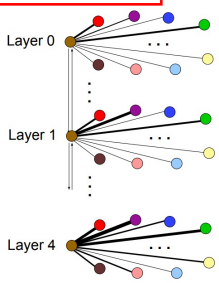
- $f_k(u,v)$: structural distance between nodes u and v considering first k rings
  - $f_k(u,v) = f_{k-1}(u,v) + g(s(R_k(u)),s(R_k(v)))$

$f_0(u,v) = 0.33$ $f_1(u,v) = 3.66$ $f_2(u,v) = 4.66$

31

Struc2vec - Multi-layer graph

- Encodes structural similarity between all node pairs
- Each layer is weighted complete graph
  - corresponds to similarity hierarchies
- Edge weights in layer k
  - $w_k(u,v) = \exp\{-f_k(u,v)\}$
- Connect corresponding nodes in adjacent layers



32

Struc2vec - Generate Context

- Context generated by biased random walk
  - walking on multi-layer graph
- Walk in current layer with probability p
  - choose neighbor according to edge weight
  - RW prefers more similar nodes
- Change layer with probability 1-p
  - choose up/down according to edge weight
  - RW prefer layer with less similar neighbors

33

Struc2vec – Learn Representation

- For each node, generate set of independent and relative short random walks
  - context for node; sentences of a language
- Train a neural network to learn latent representation for nodes
  - maximize probability of nodes within context
  - Skip-gram (Hierarchical Softmax) adopted

34

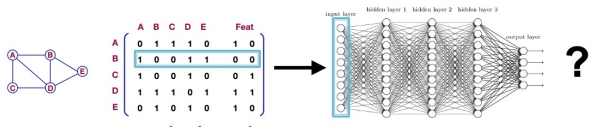
Learning with graphs

- 1. Generating Graph Representation
  - 3.1 Traditional Methods
  - 3.1 Random Walk - based Methods
- 2. Deep Learning for Graphs
- 3. Graph Neural Networks
  - 3.1 GNN Architecture
  - 3.2 Layers of GNN
  - 3.3 Training GNNs
  - 3.4 Expressiveness of GNN

35

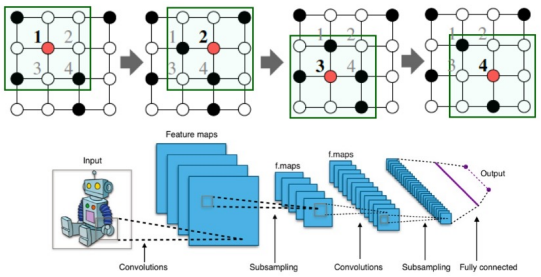
A Naïve approach

- Suppose we have a graph and a node prediction task
- We can either use node2vec to generate features and train a classification model,
- Or, we can join adjacency matrix and features, and feed them into a deep neural net:



36

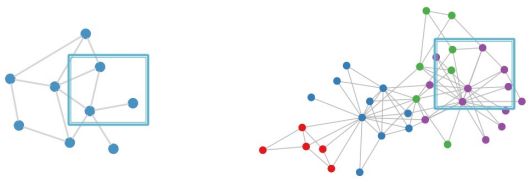
Idea: CNN



37

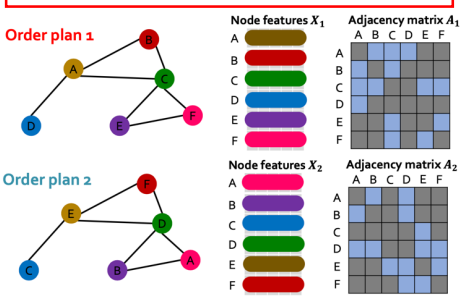
Does this work for graphs?

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



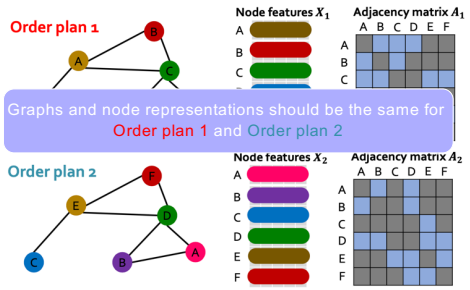
38

Permutation Invariance



39

Permutation Invariance



40

Permutation Invariance

Graphs and node representations should be the same for Order plan 1 and Order plan 2

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

$$f(A_1, X_1) = f(A_2, X_2)$$

- f maps a graph to a d-dim embedding, where A is the adjacency matrix and X is the node feature matrix.
- For a graph with $|V|$ node, there are $|V|!$ different order plans !

41

Permutation Invariance

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

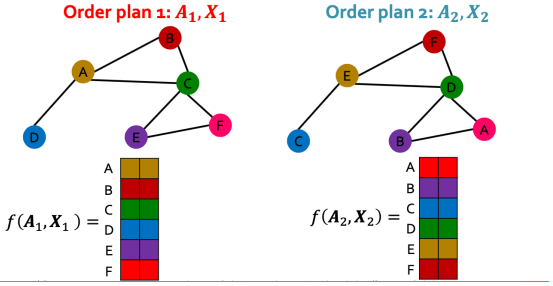
$$f(A_1, X_1) = f(A_2, X_2)$$

- If $f(A_i, X_i) = f(A_j, X_j)$ for any order plan i and j, we formally say f is a permutation invariant function.
- Definition: For any graph function $f: \mathbb{R}^{|V|\times m} \times \mathbb{R}^{|V|\times|V|} \to \mathbb{R}^d$, f is permutation-invariant if $f(A, X) = f(PAP^T, PX)$ for any permutation P.
- P – permutation, a shuffle of the node order, i.e. (A, B, C) → (B, C, A)
- m – each node has a m – dim feature vector associated with it
- d – output embedding dimensionality of embedding the graph $G = (A, X)$.

42

Permutation Equivariance

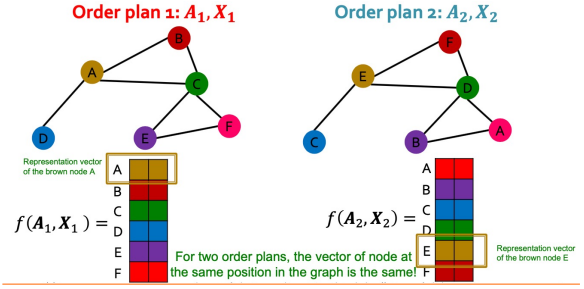
Order plan 1: A_1, X_1 Order plan 2: A_2, X_2



43

Permutation Equivariance

Order plan 1: A_1, X_1 Order plan 2: A_2, X_2

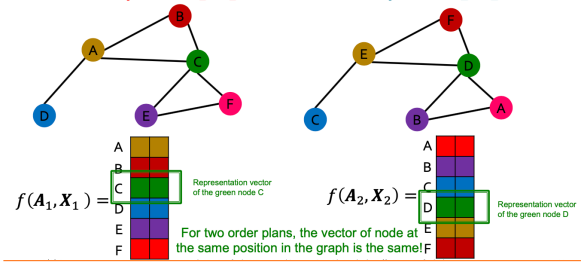


For two order plans, the vector of node at the same position in the graph is the same!

44

Permutation Equivariance

Order plan 1: A_1, X_1 Order plan 2: A_2, X_2



For two order plans, the vector of node at the same position in the graph is the same!

45

Permutation Equivariance

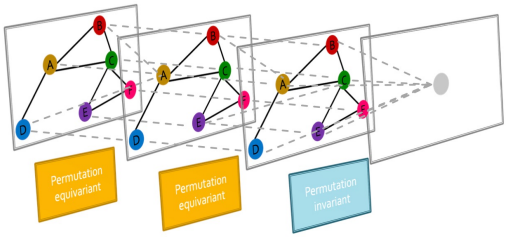
- Consider we learn a function f that maps a graph $G = (A, X)$ to a matrix $\mathbb{R}^{|V| \times d}$.
- If the output vector of a node at the same position in the graph remains unchanged for any order plan, we say f is permutation equivariant.
- Definition: For any graph function $f: \mathbb{R}^{|V| \times m} \times \mathbb{R}^{|V| \times |V|} \to \mathbb{R}^{|V| \times d}$, f is permutation-equivariant if $Pf(A, X) = f(PAP^T, PX)$ for any permutation P.

Invariance and Equivariance

- Permutation invariant: permute the input, the output stays the same (map a graph to a vector):
$$f(A, X) = f(PAP^T, PX)$$
- Permutation equivariant: Permute the input, output also permutes accordingly (map a graph to a matrix):
$$Pf(A, X) = f(PAP^T, PX)$$
- Examples:
  - $f(A, X) = 1^T X$: permutation invariant
    - Reason: $f(PAP^T, PX) = 1^T PX = 1^T X = f(A, X)$
  - $f(A, X) = AX$: permutation equivariant
    - Reason: $f(PAP^T, PX) = PAP^T PX = PAX = Pf(A, X)$

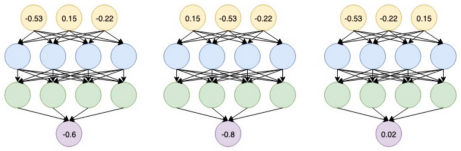
Graph Neural Network

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



Graph Neural Network

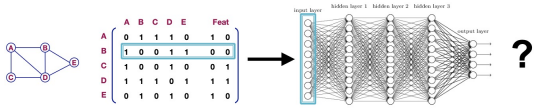
- Are other neural network architectures, e.g., MLPs, permutation invariant / equivariant?
- No! Switching the order of the input leads to different outputs!



49

Graph Neural Network

- Are other neural network architectures, e.g., MLPs, permutation invariant / equivariant?
- No! Switching the order of the input leads to different outputs!



This explains why the naïve MLP approach fails for graphs!

50

Graph Neural Network

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

51

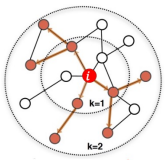
Learning with graphs

- 1. Generating Graph Representation
  - 3.1 Traditional Methods
  - 3.1 Random Walk - based Methods
- 2. Deep Learning for Graphs
- 3. Graph Neural Networks
  - 3.1 GNN Architecture
  - 3.2 Layers of GNN
  - 3.3 Training GNNs
  - 3.4 Expressiveness of GNN

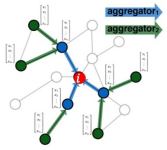
52

Graph Neural Network

- Node's neighborhood defines a computation graph
- Learn how to propagate information across the graph to compute node features



Determine node computation graph

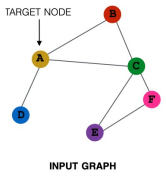


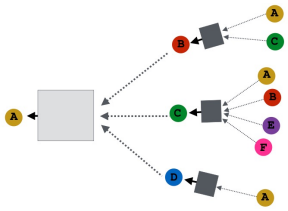
Propagate and transform information

53

Graph Neural Network

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

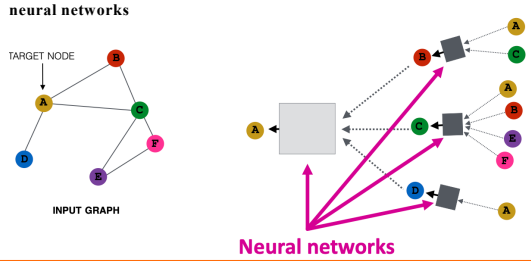




54

Graph Neural Network

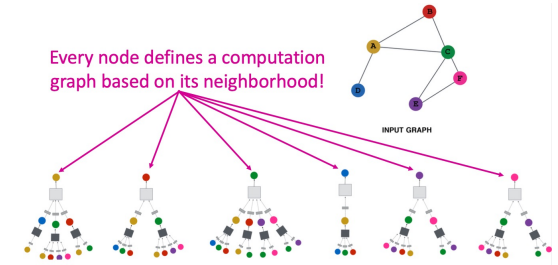
- **Intuition:** Nodes aggregate information from their neighbors using neural networks



55

Graph Neural Network

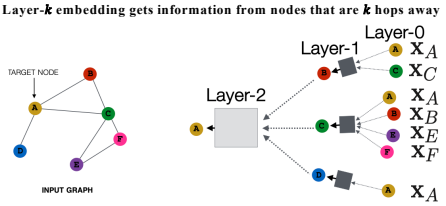
- **Intuition:** Network neighborhood defines a computation graph



56

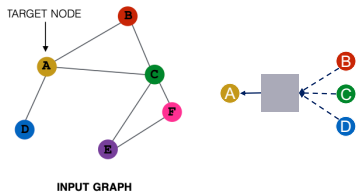
Deep Model: Many layers

- **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



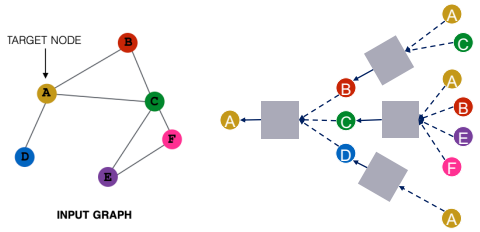
57

Graph Neural Network



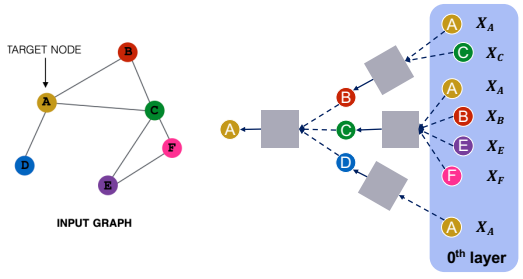
58

Graph Neural Network



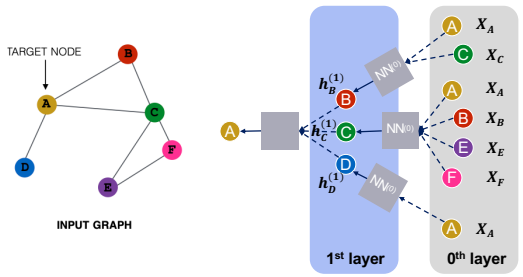
59

Graph Neural Network



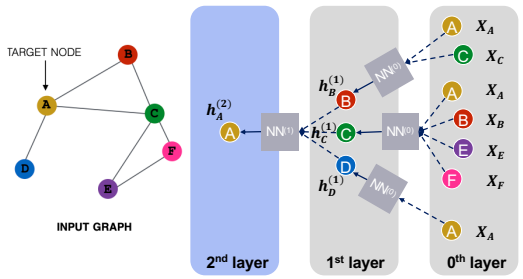
60

Graph Neural Network



61

Graph Neural Network

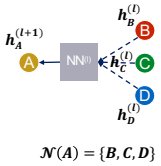


62

Aggregate and transform messages

- **Aggregate messages from neighbors**
  - $h_v^{(l)}$: node embedding of v at l-th layer
  - $\mathcal{N}(v)$: neighborhood of v
  - $f^{(l)}$: aggregation function at l-th layer
  - $m_v^{(l)}$: message vector of v at l-th layer

$$m_A^{(l)} = f^{(l)}\left(h_A^{(l)}, \left\{h_u^{(l)} : u \in \mathcal{N}(A)\right\}\right)$$
$$= f^{(l)}\left(h_A^{(l)}, h_B^{(l)}, h_C^{(l)}, h_D^{(l)}\right)$$



$$\mathcal{N}(A) = \{B, C, D\}$$

- **Transform messages**
  - $g^{(l)}$: transformation function at l-th layer

$$h_A^{(l+1)} = g^{(l)}\left(m_A^{(l)}\right)$$

63

Aggregate and transform messages

In each level l,

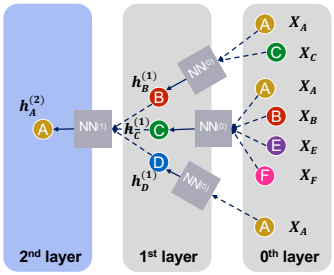
for each target node v:

- Aggregate messages

$$m_v^{(l)} = f^{(l)}\left(h_v^{(l)}, \left\{h_u^{(l)} : u \in \mathcal{N}(v)\right\}\right)$$

- Transform messages

$$h_v^{(l+1)} = g^{(l)}(m_v^{(l)})$$



64

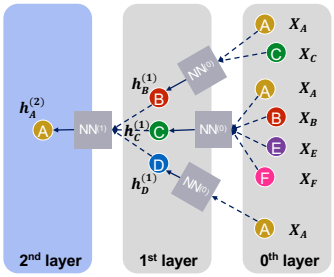
Graph Convolutional Networks

- Aggregate messages

$$m_v^{(l)} = \frac{1}{|\mathcal{N}(v)+1|} \sum_{u \in \mathcal{N}(v) \cup \{v\}} h_u^{(l)}$$

- Transform messages

$$h_v^{(l+1)} = \sigma(W^{(l)} \circ m_v^{(l)})$$

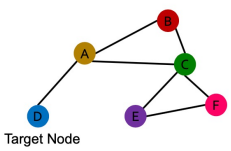


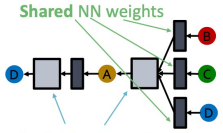
 Kipf, Thomas N., et al. "Semi-supervised classification with graph convolutional networks."

65

GCN: Invariance and Equivariance

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



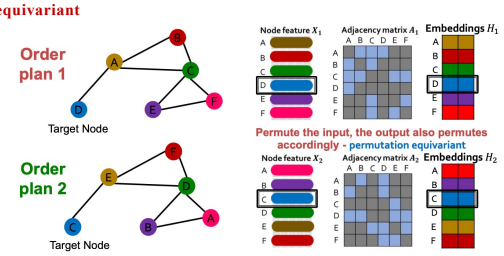


Average of neighbor's previous layer embeddings - Permutation invariant

66

GCN: Invariance and Equivariance

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



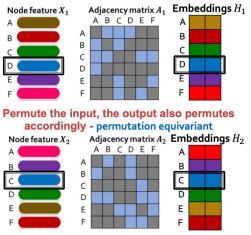
67

GCN: Invariance and Equivariance

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

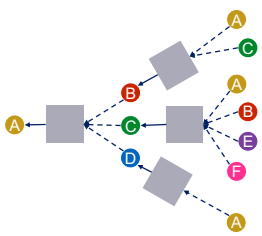
Detailed reasoning:
1. The rows of **input node features** and **output embeddings** are **aligned**
2. We know computing the embedding of **a given node** with GCN is **invariant.**
3. So, after permutation, the **location** of **a given node** in the **input node feature** matrix is changed, and the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

This is permutation equivariant



68

Training GCN



69

Model Parameters

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

$$h_v^{(l+1)} = \sigma(W_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)}, \forall l \in \{0 \dots L-1\}$$

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

- We can feed these embeddings into any loss function and run SGD to train the weight parameters
- W_l: weight matrix for neighborhood aggregation
- B_l : weight matrix for transforming hidden vector of self

70

Learning with graphs

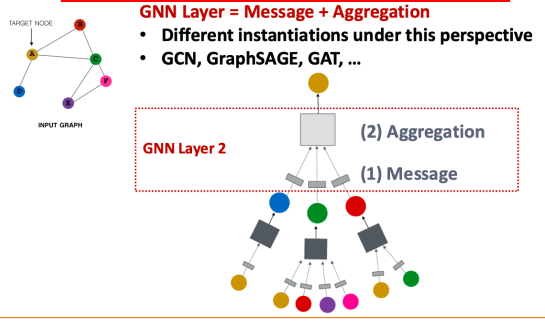
- 1. Generating Graph Representation
  - 3.1 Traditional Methods
  - 3.1 Random Walk - based Methods
- 2. Deep Learning for Graphs
- 3. Graph Neural Networks
  - 3.1 GNN Architecture
  - 3.2 Layers of GNN
  - 3.3 Training GNNs
  - 3.4 Expressiveness of GNN

71

A GNN Layer

GNN Layer = Message + Aggregation

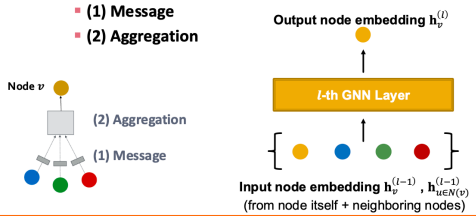
- Different instantiations under this perspective
- GCN, GraphSAGE, GAT, …



72

A Single GNN Layer

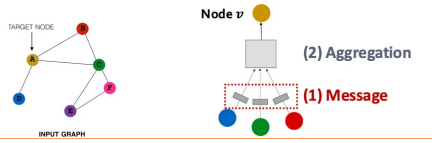
- Idea of a GNN Layer:
  - Compress a set of vectors into a single vector
  - **Two-step process:**
    - **(1) Message**
    - **(2) Aggregation**



73

Message Computation

- **Message function:** $\mathbf{m}_u^{(l)} = \mathrm{MSG}^{(l)}\left(\mathbf{h}_u^{(l-1)}\right)$
  - **Intuition:** Each node will create a message, which will be sent to other nodes later
  - **Example:** A Linear layer $\mathbf{m}_u^{(l)} = \mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}$
    - Multiply node features with weight matrix $\mathbf{W}^{(l)}$



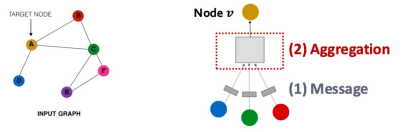
74

Message Aggregation

- **Intuition:** Node v will aggregate the messages from its neighbors u:

$$\mathbf{h}_v^{(l)} = \mathrm{AGG}^{(l)}\left(\left\{\mathbf{m}_u^{(l)}, u \in N(v)\right\}\right)$$

- **Example:** $\mathrm{Sum}(\cdot)$, $\mathrm{Mean}(\cdot)$ or $\mathrm{Max}(\cdot)$ aggregator
  - $\mathbf{h}_v^{(l)} = \mathrm{Sum}(\{\mathbf{m}_u^{(l)}, u \in N(v)\})$



75

Message Aggregation: Issue

- **Issue:** Information from node v itself **could get lost**
  - Computation of $\mathbf{h}_v^{(l)}$ does not directly depend on $\mathbf{h}_v^{(l-1)}$
- **Solution:** Include $\mathbf{h}_v^{(l-1)}$ when computing $\mathbf{h}_v^{(l)}$
  - **(1) Message: compute message from node v itself**
    - Usually, a **different message computation** will be performed

$\mathbf{m}_u^{(l)} = \mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}$ $\mathbf{m}_v^{(l)} = \mathbf{B}^{(l)}\mathbf{h}_v^{(l-1)}$

  - **(2) Aggregation:** After aggregating from neighbors, we can **aggregate the message from node v itself**
    - Via **concatenation** or **summation**

Then aggregate from node itself

$$\mathbf{h}_v^{(l)} = \text{CONCAT}\left(\text{AGG}\left(\left\{\mathbf{m}_u^{(l)}, u \in N(v)\right\}\right), \mathbf{m}_v^{(l)}\right)$$

First aggregate from neighbors

A Single GNN Layer

- **Putting things together:**
  - **(1) Message**: each node computes a message

$$\mathbf{m}_u^{(l)} = \text{MSG}^{(l)}\left(\mathbf{h}_u^{(l-1)}\right), u \in \{N(v) \cup v\}$$

  - **(2) Aggregation**: aggregate messages from neighbors

$$\mathbf{h}_v^{(l)} = \text{AGG}^{(l)}\left(\left\{\mathbf{m}_u^{(l)}, u \in N(v)\right\}, \mathbf{m}_v^{(l)}\right)$$

  - **Nonlinearity (activation):** Adds expressiveness
    - Often written as $\sigma(\cdot)$. Examples: ReLU$(\cdot)$, Sigmoid$(\cdot)$, ...
    - Can be added to **message or aggregation**

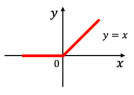
Activation (Non-linearity)

Apply activation to i-th dimension of embedding $\mathbf{x}$

- **Rectified linear unit (ReLU)**

$$\text{ReLU}(\mathbf{x}_i) = \max(\mathbf{x}_i, 0)$$

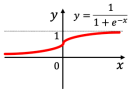
  - Most commonly used



- **Sigmoid**

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

  - Used only when you want to restrict the range of your embeddings

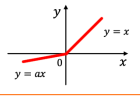


- **Parametric ReLU**

$$\text{PReLU}(\mathbf{x}_i) = \max(\mathbf{x}_i, 0) + a_i \min(\mathbf{x}_i, 0)$$

a_i is a trainable parameter

  - Empirically performs better than ReLU



Classical GNN Layers: GCN(1)

- (1) Graph Convolutional Networks (GCN)

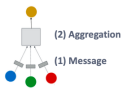
$$\mathbf{h}_v^{(l)} = \sigma\left(\mathbf{W}^{(l)} \sum_{u \in N(v)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}\right)$$

- How to write this as Message + Aggregation?

Message

$$\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}\right)$$

Aggregation

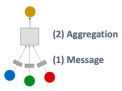


79

Classical GNN Layers: GCN(2)

- (1) Graph Convolutional Networks (GCN)

$$\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}\right)$$



- **Message:**
  - Each Neighbor: $\mathbf{m}_u^{(l)} = \frac{1}{|N(v)|}\mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}$ Normalized by node degree (In the GCN paper they use a slightly different normalization)
- **Aggregation:**
  - **Sum** over messages from neighbors, then apply activation
  - $\mathbf{h}_v^{(l)} = \sigma\left(\text{Sum}\left(\left\{\mathbf{m}_u^{(l)}, u \in N(v)\right\}\right)\right)$ In GCN the input graph is assumed to have self-edges that are included in the summation.

80

Classical GNN Layers: GraphSAGE

- (2) GraphSAGE

$$\mathbf{h}_v^{(l)} = \sigma\left(\mathbf{W}^{(l)} \cdot \text{CONCAT}\left(\mathbf{h}_v^{(l-1)}, \text{AGG}\left(\left\{\mathbf{h}_u^{(l-1)}, \forall u \in N(v)\right\}\right)\right)\right)$$

  - Two-stage aggregation
    - **Stage 1:** Aggregate from node neighbors

$$\mathbf{h}_{N(v)}^{(l)} \leftarrow \text{AGG}\left(\left\{\mathbf{h}_u^{(l-1)}, \forall u \in N(v)\right\}\right)$$

    - **Stage 2:** Further aggregate over the node itself

$$\mathbf{h}_v^{(l)} \leftarrow \sigma\left(\mathbf{W}^{(l)} \cdot \text{CONCAT}(\mathbf{h}_v^{(l-1)}, \mathbf{h}_{N(v)}^{(l)})\right)$$

  - **Message** is computed within the $\text{AGG}(\cdot)$
- How to write this as Message + Aggregation?

81

GraphSAGE Neighbor Aggregation

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

$$\text{AGG} = \sum_{u \in N(v)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}$$

Aggregation — Message computation

- **Pool:** Transform neighbor vectors and apply symmetric vector function $\text{Mean}(\cdot)$ or $\text{Max}(\cdot)$

$$\text{AGG} = \text{Mean}(\{\text{MLP}(\mathbf{h}_u^{(l-1)}), \forall u \in N(v)\})$$

Aggregation — Message computation

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

$$\text{AGG} = \text{LSTM}([\mathbf{h}_u^{(l-1)}, \forall u \in \pi(N(v))])$$

Aggregation

82

GraphSAGE: L2 Normalization

- ℓ_2 **Normalization:**
  - **Optional:** Apply ℓ_2 normalization to $\mathbf{h}_v^{(l)}$ at every layer
  - $\mathbf{h}_v^{(l)} \leftarrow \frac{\mathbf{h}_v^{(l)}}{\|\mathbf{h}_v^{(l)}\|_2} \quad \forall v \in V$ where $\|u\|_2 = \sqrt{\sum_i u_i^2}$ (ℓ_2-norm)
  - Without ℓ_2 normalization, the embedding vectors have different scales (ℓ_2-norm) for vectors
  - In some cases (not always), normalization of embedding results in performance improvement
  - After ℓ_2 normalization, all vectors will have the same ℓ_2-norm

83

Classical GNN Layers: GAT(1)

- **(3) Graph Attention Networks**

$$\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \alpha_{vu} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}\right)$$

Attention weights

- **In GCN / GraphSAGE**
  - $\alpha_{vu} = \frac{1}{|N(v)|}$ is the **weighting factor (importance)** of node u's message to node v
  - $\Longrightarrow \alpha_{vu}$ is defined **explicitly** based on the structural properties of the graph (node degree)
  - $\Longrightarrow$ All neighbors $u \in N(v)$ are equally important to node v

84

Classical GNN Layers: GAT(2)

- (3) Graph Attention Networks

$$\mathbf{h}_v^{(l)} = \sigma(\sum_{u \in N(v)} \alpha_{vu} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})$$

Attention weights

Not all node's neighbors are equally important

- **Attention** is inspired by cognitive attention.
- The **attention** α_{vu} focuses on the important parts of the input data and fades out the rest.
  - **Idea:** the NN should devote more computing power on that small but important part of the data.
  - Which part of the data is more important depends on the context and is learned through training.

85

Graph Attention Networks

Can we do better than simple neighborhood aggregation?

Can weighting factors α_{vu} be learned?

- **Goal:** Specify **arbitrary importance** to different neighbors of each node in the graph
- **Idea:** Compute embedding $\boldsymbol{h}_v^{(l)}$ of each node in the graph following an **attention strategy**:
  - Nodes attend over their neighborhoods' message
  - Implicitly specifying different weights to different nodes in a neighborhood

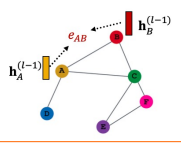
86

Attention Mechanism(1)

- Let α_{vu} be computed as a byproduct of an **attention mechanism** $\boldsymbol{a}$:
  - (1) Let a compute **attention coefficients** $\boldsymbol{e_{vu}}$ across pairs of nodes u, v based on their messages:

$$e_{vu} = a(\mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}, \mathbf{W}^{(l)} \boldsymbol{h}_v^{(l-1)})$$

  - **e_{vu} indicates the importance of u's message to node v**



$$e_{AB} = a(\mathbf{W}^{(l)} \mathbf{h}_A^{(l-1)}, \mathbf{W}^{(l)} \mathbf{h}_B^{(l-1)})$$

87

Attention Mechanism(2)

- **Normalize** e_{vu} into the **final attention weight** α_{vu}
  - Use the **softmax** function, so that $\sum_{u \in N(v)} \alpha_{vu} = 1$:

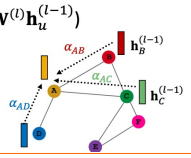
$$\alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})}$$

- **Weighted sum** based on the **final attention weight** α_{vu}:

$$\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \alpha_{vu} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}\right)$$

Weighted sum using α_{AB}, α_{AC}, α_{AD}:

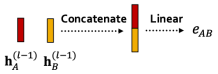
$\mathbf{h}_A^{(l)} = \sigma(\alpha_{AB}\mathbf{W}^{(l)}\mathbf{h}_B^{(l-1)} + \alpha_{AC}\mathbf{W}^{(l)}\mathbf{h}_C^{(l-1)} + \alpha_{AD}\mathbf{W}^{(l)}\mathbf{h}_D^{(l-1)})$



88

Attention Mechanism(3)

- **What is the form of attention mechanism a?**
  - The approach is agnostic to the choice of a
    - E.g., use a simple single-layer neural network
      - a have trainable parameters (weights in the Linear layer)



$$e_{AB} = a\left(\mathbf{W}^{(l)}\mathbf{h}_A^{(l-1)}, \mathbf{W}^{(l)}\mathbf{h}_B^{(l-1)}\right)$$
$$= \text{Linear}\left(\text{Concat}\left(\mathbf{W}^{(l)}\mathbf{h}_A^{(l-1)}, \mathbf{W}^{(l)}\mathbf{h}_B^{(l-1)}\right)\right)$$

  - Parameters of a are trained jointly:
    - Learn the parameters together with weight matrices (i.e., other parameter of the neural net $\mathbf{W}^{(l)}$) in an end-to-end fashion

89

Attention Mechanism(4)

- **Multi-head attention:** Stabilizes the learning process of attention mechanism
  - **Create multiple attention scores** (each replica with a different set of parameters):

$$\mathbf{h}_v^{(l)}[1] = \sigma\left(\sum_{u \in N(v)} \alpha_{vu}^1 \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}\right)$$
$$\mathbf{h}_v^{(l)}[2] = \sigma\left(\sum_{u \in N(v)} \alpha_{vu}^2 \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}\right)$$
$$\mathbf{h}_v^{(l)}[3] = \sigma\left(\sum_{u \in N(v)} \alpha_{vu}^3 \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}\right)$$

  - **Outputs are aggregated:**
    - By concatenation or summation
    - $\mathbf{h}_v^{(l)} = \text{AGG}(\mathbf{h}_v^{(l)}[1], \mathbf{h}_v^{(l)}[2], \mathbf{h}_v^{(l)}[3])$

90

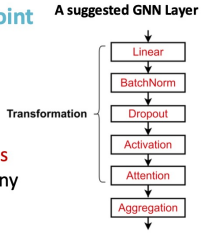
Benefits of Attention Mechanism

- **Key benefit:** Allows for (implicitly) specifying **different importance values (α_{vu}) to different neighbors**
- **Computationally efficient**:
  - Computation of attentional coefficients can be parallelized across all edges of the graph
  - Aggregation may be parallelized across all nodes
- **Storage efficient**:
  - Sparse matrix operations do not require more than $O(V+E)$ entries to be stored
  - **Fixed** number of parameters, irrespective of graph size
- **Localized**:
  - Only **attends over local network neighborhoods**
- **Inductive capability**:
  - It is a shared *edge-wise* mechanism
  - It does not depend on the global graph structure

91

GNN Layer in Practice

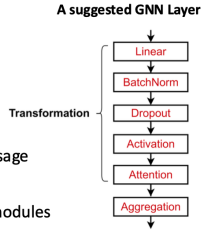
- **In practice, these classic GNN layers are a great starting point**
  - We can often get better performance by considering a general GNN layer design
  - Concretely, we can include modern deep learning modules that proved to be useful in many domains



92

GNN Layer in Practice

- **Many modern deep learning modules can be incorporated into a GNN layer**
  - **Batch Normalization:**
    - Stabilize neural network training
  - **Dropout:**
    - Prevent overfitting
  - **Attention/Gating:**
    - Control the importance of a message
  - **More:**
    - Any other useful deep learning modules



93

Batch Normalization

- **Goal**: Stabilize neural networks training
- **Idea**: Given a batch of inputs (node embeddings)
  - Re-center the node embeddings into zero mean
  - Re-scale the variance into unit variance

Input: $\mathbf{X} \in \mathbb{R}^{N \times D}$
N node embeddings

Trainable Parameters:
$\boldsymbol{\gamma}, \boldsymbol{\beta} \in \mathbb{R}^{D}$

Output: $\mathbf{Y} \in \mathbb{R}^{N \times D}$
Normalized node embeddings

Step 1:
Compute the mean and variance over N embeddings

$$\boldsymbol{\mu}_j = \frac{1}{N}\sum_{i=1}^{N} \mathbf{X}_{i,j}$$
$$\boldsymbol{\sigma}_j^2 = \frac{1}{N}\sum_{i=1}^{N} (\mathbf{X}_{i,j} - \boldsymbol{\mu}_j)^2$$

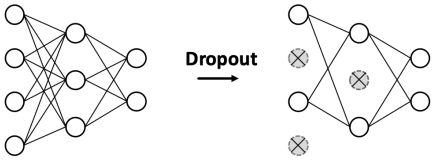
Step 2:
Normalize the feature using computed mean and variance

$$\widehat{\mathbf{X}}_{i,j} = \frac{\mathbf{X}_{i,j} - \boldsymbol{\mu}_j}{\sqrt{\boldsymbol{\sigma}_j^2 + \epsilon}}$$
$$\mathbf{Y}_{i,j} = \boldsymbol{\gamma}_j \widehat{\mathbf{X}}_{i,j} + \boldsymbol{\beta}_j$$

S. Loffe, C.Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, ICML 2015

Dropout

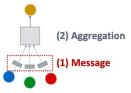
- **Goal**: Regularize a neural net to prevent overfitting.
- **Idea**:
  - **During training**: with some probability p, randomly set neurons to zero (turn off)
  - **During testing:** Use all the neurons for computation

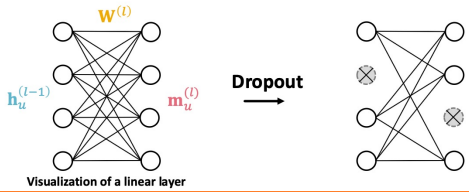


Srivastava et al. Dropout: A Simple Way to Prevent Neural Networks from Overfitting, JMLR 2014

Dropout for GNNs

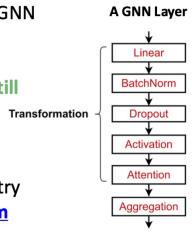
- In GNN, Dropout is applied to **the linear layer in the message function**
  - **A simple message function with linear layer:** $\mathbf{m}_u^{(l)} = \mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}$





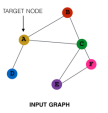
GNN Layer in Practice

- **Summary:** Modern deep learning modules can be included into a GNN layer for better performance
- **Designing novel GNN layers is still an active research frontier!**
- **Suggested resources:** You can explore diverse GNN designs or try out your own ideas in GraphGym



97

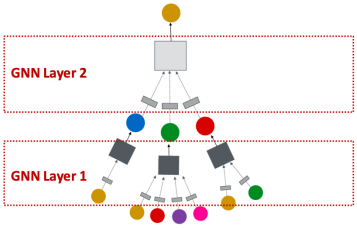
Stacking GNN Layers



How to connect GNN layers into a GNN?

- **Stack layers sequentially**
- **Ways of adding skip connections**

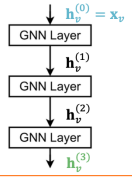
(3) Layer connectivity



98

Stacking GNN Layers

- **How to construct a Graph Neural Network?**
  - **The standard way:** Stack GNN layers sequentially
  - **Input:** Initial raw node feature $\mathbf{x}_v$
  - **Output:** Node embeddings $\mathbf{h}_v^{(L)}$ after L GNN layers



99

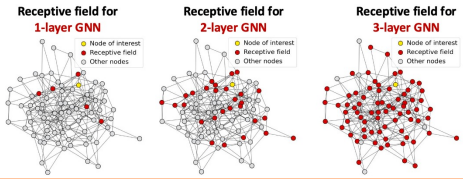
The Over-smoothing Problem

- The issue of stacking many GNN layers
  - GNN suffers from **the over-smoothing problem**
- **The over-smoothing problem: all the node embeddings converge to the same value**
  - This is bad because we **want to use node embeddings to differentiate nodes**
- **Why does the over-smoothing problem happen?**

100

Receptive Field of a GNN

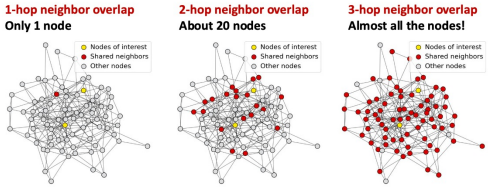
- **Receptive field:** the set of nodes that determine the embedding of a node of interest
  - **In a K-layer GNN, each node has a receptive field of K-hop neighborhood**



101

Receptive Field of a GNN

- **Receptive field overlap** for two nodes
  - **The shared neighbors quickly grow** when we increase the number of hops (num of GNN layers)



102

Receptive Field & Over-smoothing

- We can explain over-smoothing via the notion of the receptive field
  - We know **the embedding of a node is determined by its** receptive field
    - If two nodes **have highly-overlapped receptive fields, then their embeddings are highly similar**
  - Stack many GNN layers → nodes will have highly-overlapped receptive fields → node embeddings will be highly similar → **suffer from the over-smoothing problem**
- Next: how do we overcome over-smoothing problem?

103

Design GNN Layer Connectivity

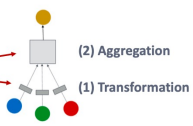
- What do we learn from the over-smoothing problem?
- **Lesson 1: Be cautious when adding GNN layers**
  - Unlike neural networks in other domains (CNN for image classification), **adding more GNN layers do not always help**
  - **Step 1: Analyze the necessary receptive field** to solve your problem. E.g., by computing the diameter of the graph
  - **Step 2:** Set number of GNN layers L to be a bit more than the receptive field we like. **Do not set L to be unnecessarily large**!
- **Question:** How to enhance the expressive power of a GNN, if the number of GNN layers is small?

104

Expressive Power for Shallow GNNs

- How to make a shallow GNN more expressive?
- **Solution 1:** Increase the expressive power **within each GNN layer**
  - In our previous examples, each transformation or aggregation function only includes one linear layer
  - We can **make aggregation / transformation become a deep neural network**!

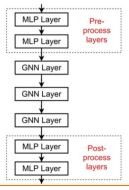
If needed, each box could include a 3-layer MLP



105

Expressive Power for Shallow GNNs

- How to make a shallow GNN more expressive?
- **Solution 2:** Add layers that do not pass messages
  - A GNN does not necessarily only contain GNN layers
    - E.g., we can add **MLP layers** (applied to each node) before and after GNN layers, as **pre-process layers** and **post-process layers**



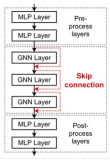
Pre-processing layers: Important when encoding node features is necessary.
E.g., when nodes represent images/text

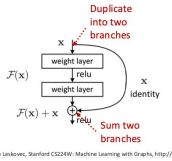
Post-processing layers: Important when reasoning / transformation over node embeddings are needed
E.g., graph classification, knowledge graphs

In practice, adding these layers works great!

Design GNN Layer Connectivity

- What if my problem still requires many GNN layers?
- **Lesson 2: Add skip connections in GNNs**
  - **Observation from over-smoothing:** Node embeddings in earlier GNN layers can sometimes better differentiate nodes
  - **Solution:** We can increase the impact of earlier layers on the final node embeddings, **by adding shortcuts in GNN**

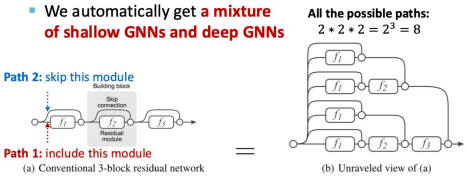




Idea of skip connections:
Before adding shortcuts:
$\boldsymbol{F}(\mathbf{x})$
After adding shortcuts:
$\boldsymbol{F}(\mathbf{x}) + \mathbf{x}$

Idea of Skip Connections

- Why do skip connections work?
  - **Intuition:** Skip connections create **a mixture of models**
  - N skip connections → 2^N possible paths
  - Each path could have up to N modules
  - We automatically get **a mixture of shallow GNNs and deep GNNs**

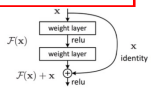


 Veit et al. Residual Networks Behave Like Ensembles of Relatively Shallow Networks, ArXiv 2016

Example: GCN with Skip Connections

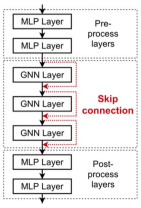
- A standard GCN layer
- $\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}\right)$

This is our $F(\mathbf{x})$



- A GCN layer with skip connection
- $\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|} + \mathbf{h}_v^{(l-1)}\right)$

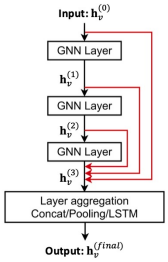
$F(\mathbf{x})$ + $\mathbf{x}$



109

Other Options of Skip Connections

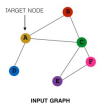
- **Other options:** Directly skip to the last layer
  - The final layer directly **aggregates from the all the node embeddings** in the previous layers



 Xu et al. Representation learning on graphs with jumping knowledge networks, ICML 2018

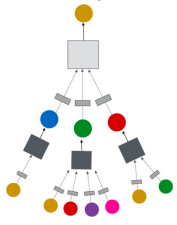
110

General GNN Framework



Idea: Raw input graph ≠ computational graph

- **Graph feature augmentation**
- **Graph structure manipulation**



(4) Graph manipulation

111

Why Manipulate Graphs

Our assumption so far has been
- Raw input graph = computational graph

Reasons for breaking this assumption
- Feature level:
  - The input graph lacks features → feature augmentation
- Structure level:
  - The graph is too sparse → inefficient message passing
  - The graph is too dense → message passing is too costly
  - The graph is too large → cannot fit the computational graph into a GPU
- It's just unlikely that the input graph happens to be the optimal computation graph for embeddings

112

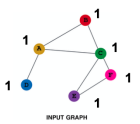
Graph Manipulation Approaches

- Graph Feature manipulation
  - The input graph **lacks features** → **feature augmentation (will cover in Lecture 7)**
- Graph Structure manipulation
  - The graph is **too sparse** → **Add virtual nodes / edges**
  - The graph is **too dense** → **Sample neighbors when doing message passing**
  - The graph is **too large** → **Sample subgraphs to compute embeddings**
    - Will cover later in lecture: Scaling up GNNs

113

Feature Augmentation on Graphs

Why do we need feature augmentation?
- (1) Input graph does not have node features
  - This is common when we only have the adj. matrix
- Standard approaches:
- a) Assign constant values to nodes

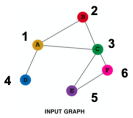


114

Feature Augmentation on Graphs

Why do we need feature augmentation?

- **(1) Input graph does not have node features**
  - This is common when we only have the adj. matrix
- **Standard approaches:**
- **b) Assign unique IDs to nodes**
  - These IDs are converted into **one-hot vectors**



One-hot vector for node with ID=5

ID = 5

[0, 0, 0, 0, 1, 0]

Total number of IDs = 6

115

Feature Augmentation on Graphs

- Feature augmentation: **constant** vs. **one-hot**

	Constant node feature 	One-hot node feature 
Expressive power	**Medium.** All the nodes are identical, but GNN can still learn from the graph structure	**High.** Each node has a unique ID, so node-specific information can be stored
Inductive learning (Generalize to unseen nodes)	**High.** Simple to generalize to new nodes: we assign constant feature to them, then apply our GNN	**Low.** Cannot generalize to new nodes: new nodes introduce new IDs, GNN doesn't know how to embed unseen IDs
Computational cost	**Low.** Only 1 dimensional feature	**High.** High dimensional feature, cannot apply to large graphs
Use cases	Any graph, inductive settings (generalize to new nodes)	Small graph, transductive settings (no new nodes)

116

Feature Augmentation on Graphs

Why do we need feature augmentation?

- **(2) Certain structures are hard to learn by GNN**
- **Example:** Cycle count feature
  - Can GNN learn the length of a cycle that v_1 resides in?
  - **Unfortunately, no**

v_1 resides in a cycle with length 3

v_1 resides in a cycle with length 4

117

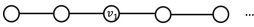
Feature Augmentation on Graphs

- v_1 cannot differentiate which graph it resides in
  - Because all the nodes in the graph have degree of 2
  - The computational graphs will be the same binary tree

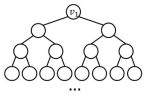
v_1 resides in a cycle with length 3

v_1 resides in a cycle with length 4

v_1 resides in a cycle with infinite length



The computational graphs for node v_1 are always the same



118

Feature Augmentation on Graphs

Why do we need feature augmentation?

- **(2) Certain structures are hard to learn by GNN**
- **Solution:**
  - We can use cycle count as augmented node features

We start from cycle with length 0

Augmented node feature for v_1: **[0, 0, 0, 1, 0, 0]**

v_1 resides in a cycle with length 3

Augmented node feature for v_1: **[0, 0, 0, 0, 1, 0]**

v_1 resides in a cycle with length 4

 J. You, J. Gomes-Selman, R. Ying, J. Leskovec. Identity-aware Graph Neural Networks, AAAI 2021

119

Add Virtual Nodes/Edges

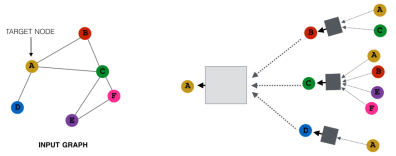
- **Motivation:** Augment sparse graphs
- **(2) Add virtual nodes**
  - The virtual node will connect to all the nodes in the graph
    - Suppose in a sparse graph, two nodes have shortest path distance of 10
    - After adding the virtual node, **all the nodes will have a distance of 2**
      - Node A – Virtual node – Node B
  - **Benefits:** Greatly **improves message passing in sparse graphs**

The virtual node

120

Node Neighborhood Sampling

- Our approach so far:
  - All the neighbors are used for message passing
- Problem: Dense/large graphs, high-degree nodes

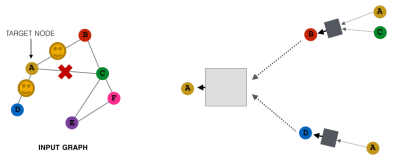


- New idea: (Randomly) determine a node's neighborhood for message passing

121

Neighborhood Sampling Example

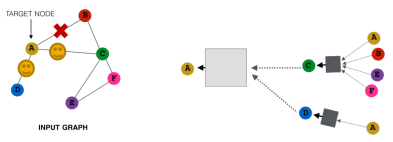
- For example, we can randomly choose 2 neighbors to pass messages
  - Only nodes B and D will pass message to A



122

Neighborhood Sampling Example

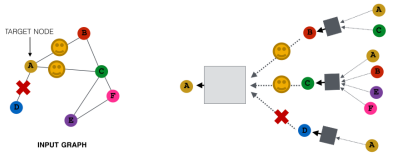
- Next time when we compute the embeddings, we can sample different neighbors
  - Only nodes C and D will pass message to A



123

Neighborhood Sampling Example

- In expectation, we can get embeddings similar to the case where all the neighbors are used
  - **Benefits:** Greatly reduce computational cost
  - And in practice it works great!



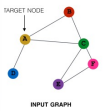
124

Learning with graphs

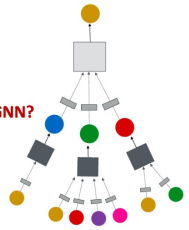
- 1. Generating Graph Representation
  - 3.1 Traditional Methods
  - 3.1 Random Walk - based Methods
- 2. Deep Learning for Graphs
- 3. Graph Neural Networks
  - 3.1 GNN Architecture
  - 3.2 Layers of GNN
  - 3.3 Training GNNs
  - 3.4 Expressiveness of GNN

125

A General GNN Framework(4)



(5) Learning objective

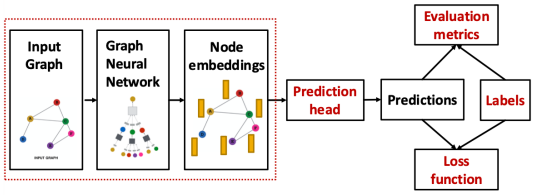


Next: How do we train a GNN?

126

GNN Training Pipeline

So far what we have covered

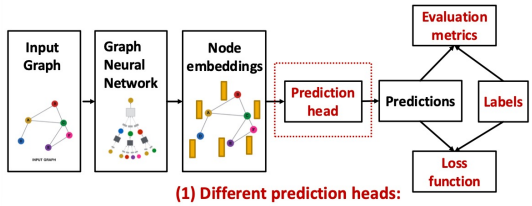


Output of a GNN: set of node embeddings

$\{\mathbf{h}_v^{(L)}, \forall v \in G\}$

127

GNN Training Pipeline(1)



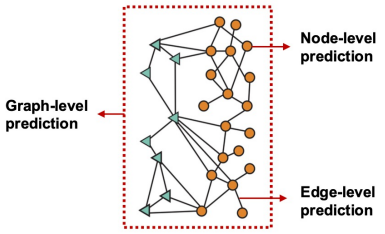
(1) Different prediction heads:

- Node-level tasks
- Edge-level tasks
- Graph-level tasks

128

GNN Prediction Heads

- **Idea:** Different task levels require different prediction heads



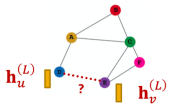
129

Prediction Heads: Node-level

- **Node-level prediction**: We can directly make prediction using node embeddings!
- After GNN computation, we have d-dim node embeddings: $\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\}$
- Suppose we want to make k-way prediction
  - Classification: classify among k categories
  - Regression: regress on k targets
- $\hat{\boldsymbol{y}}_v = \text{Head}_{\text{node}}(\mathbf{h}_v^{(L)}) = \mathbf{W}^{(H)}\mathbf{h}_v^{(L)}$
  - $\mathbf{W}^{(H)} \in \mathbb{R}^{k\times d}$: We map node embeddings from $\mathbf{h}_v^{(L)} \in \mathbb{R}^d$ to $\hat{\boldsymbol{y}}_v \in \mathbb{R}^k$ so that we can compute the loss

Prediction Heads: Edge-level

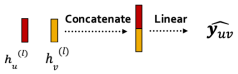
- **Edge-level prediction**: Make prediction using pairs of node embeddings
- Suppose we want to make k-way prediction
- $\hat{\boldsymbol{y}}_{uv} = \text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$



- What are the options for $\text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$?

Prediction Heads: Edge-level

- **Options for** $\text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$:
- **(1) Concatenation + Linear**
  - We have seen this in graph attention



  - $\hat{\boldsymbol{y}}_{uv} = \text{Linear}(\text{Concat}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)}))$
  - Here $\text{Linear}(\cdot)$ will map $2d$-dimensional embeddings (since we concatenated embeddings) to k-dim embeddings (k-way prediction)

Prediction Heads: Edge-level

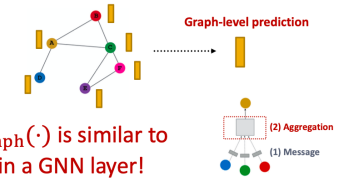
- **Options for** $\text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$:
- **(2) Dot product**
  - $\hat{\mathbf{y}}_{uv} = (\mathbf{h}_u^{(L)})^T \mathbf{h}_v^{(L)}$
  - **This approach only applies to 1-way prediction** (e.g., link prediction: predict the existence of an edge)
  - **Applying to k-way prediction:**
    - Similar to **multi-head attention**: $\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(k)}$ trainable

$$\hat{\mathbf{y}}_{uv}^{(1)} = (\mathbf{h}_u^{(L)})^T \mathbf{W}^{(1)} \mathbf{h}_v^{(L)}$$
$$\ldots$$
$$\hat{\mathbf{y}}_{uv}^{(k)} = (\mathbf{h}_u^{(L)})^T \mathbf{W}^{(k)} \mathbf{h}_v^{(L)}$$
$$\hat{\mathbf{y}}_{uv} = \text{Concat}(\hat{\mathbf{y}}_{uv}^{(1)}, \ldots, \hat{\mathbf{y}}_{uv}^{(k)}) \in \mathbb{R}^k$$

133

Prediction Heads: Graph-level

- **Graph-level prediction**: Make prediction using all the node embeddings in our graph
- Suppose we want to make k-way prediction
- $\hat{\mathbf{y}}_G = \text{Head}_{\text{graph}}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$



- $\text{Head}_{\text{graph}}(\cdot)$ is similar to $\text{AGG}(\cdot)$ in a GNN layer!

134

Prediction Heads: Graph-level

- Options for $\text{Head}_{\text{graph}}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$
- **(1) Global mean pooling**

$$\hat{\mathbf{y}}_G = \text{Mean}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$$

- **(2) Global max pooling**

$$\hat{\mathbf{y}}_G = \text{Max}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$$

- **(3) Global sum pooling**

$$\hat{\mathbf{y}}_G = \text{Sum}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$$

- These options work great for small graphs
- **Can we do better for large graphs?**

 K. Xu*, W. Hu*, J. Leskovec, S. Jegelka. How Powerful Are Graph Neural Networks, ICLR 2019

135

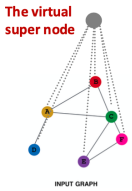
Issue of Global Pooling

- **Issue:** Global pooling over a (large) graph will lose information
- **Toy example:** we use 1-dim node embeddings
  - Node embeddings for G_1: $\{-1, -2, 0, 1, 2\}$
  - Node embeddings for G_2: $\{-10, -20, 0, 10, 20\}$
  - Clearly G_1 and G_2 have very different node embeddings
    → Their structures should be different
- **If we do global sum pooling:**
  - **Prediction for** G_1: $\hat{y}_G = \mathrm{Sum}(\{-1, -2, 0, 1, 2\}) = 0$
  - **Prediction for** G_2: $\hat{y}_G = \mathrm{Sum}(\{-10, -20, 0, 10, 20\}) = 0$
  - We cannot differentiate G_1 and G_2!

136

Add Virtual Super Node

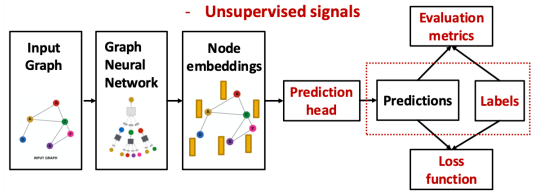
- To embed a graph add virtual super node
  - The virtual node will connect to all the nodes in the graph
  - **Benefits:** GNN learns how to encode the entire input graph



137

GNN Training Pipeline(2)

(2) Where does ground-truth come from?
- Supervised labels
- Unsupervised signals



138

Supervised vs. Unsupervised

- Supervised learning on graphs
  - Labels come from external sources
    - E.g., predict drug likeness of a molecular graph
- Unsupervised learning on graphs
  - Signals come from graphs themselves
    - E.g., link prediction: predict if two nodes are connected
- Sometimes the differences are blurry
  - We still have "supervision" in unsupervised learning
    - E.g., train a GNN to predict node clustering coefficient
  - An alternative name for "unsupervised" is "self-supervised"

139

Supervised Labels on Graphs

- Supervised labels come from the specific use cases. For example:
  - Node labels y_v: in a citation network, which subject area does a node belong to
  - Edge labels y_{uv}: in a transaction network, whether an edge is fraudulent
  - Graph labels y_G: among molecular graphs, the drug likeness of graphs
- Advice: Reduce your task to node / edge / graph labels, since they are easy to work with
  - E.g., we knew some nodes form a cluster. We can treat the cluster that a node belongs to as a node label

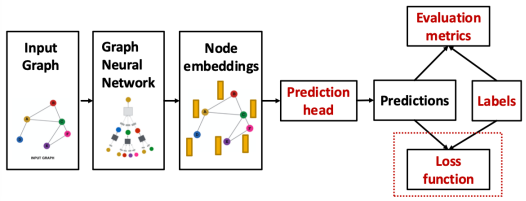
140

Unsupervised Signals on Graphs

- The problem: sometimes we only have a graph, without any external labels
- The solution: "self-supervised learning", we can find supervision signals within the graph.
  - For example, we can let GNN predict the following:
  - Node-level y_v. Node statistics: such as clustering coefficient, PageRank, ...
  - Edge-level y_{uv}. Link prediction: hide the edge between two nodes, predict if there should be a link
  - Graph-level y_G. Graph statistics: for example, predict if two graphs are isomorphic
  - These tasks do not require any external labels!

141

GNN Training Pipeline(3)



(3) How do we compute the final loss?
- Classification loss
- Regression loss

142

Setting for GNN Training

- **The setting:** We have N data points
  - Each data point can be a node/edge/graph
  - **Node-level**: prediction $\hat{\boldsymbol{y}}_v^{(i)}$, label $\boldsymbol{y}_v^{(i)}$
  - **Edge-level**: prediction $\hat{\boldsymbol{y}}_{uv}^{(i)}$, label $\boldsymbol{y}_{uv}^{(i)}$
  - **Graph-level**: prediction $\hat{\boldsymbol{y}}_G^{(i)}$, label $\boldsymbol{y}_G^{(i)}$
  - We will use prediction $\hat{\boldsymbol{y}}^{(i)}$, label $\boldsymbol{y}^{(i)}$ to refer **predictions at all levels**

143

Classification or Regression

- **Classification**: labels $\boldsymbol{y}^{(i)}$ with discrete value
  - E.g., Node classification: which category does a node belong to
- **Regression**: labels $\boldsymbol{y}^{(i)}$ with continuous value
  - E.g., predict the drug likeness of a molecular graph
- GNNs can be applied to both settings
- **Differences: loss function & evaluation metrics**

144

Classification Loss

- **Cross entropy (CE)** is a very common loss function in classification
- K-way prediction for i-th data point:

$$\mathrm{CE}(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)}) = -\sum_{j=1}^{K} \boldsymbol{y}_j^{(i)} \log(\hat{\boldsymbol{y}}_j^{(i)})$$

$\boldsymbol{y}^{(i)}$: Label; $\hat{\boldsymbol{y}}^{(i)}$: Prediction; i-th data point; j-th class

where:

E.g. | 0 | 0 | 1 | 0 | 0 |

$\boldsymbol{y}^{(i)} \in \mathbb{R}^K$ = one-hot label encoding

$\hat{\boldsymbol{y}}^{(i)} \in \mathbb{R}^K$ = prediction after $\mathrm{Softmax}(\cdot)$

E.g. | 0.1 | 0.3 | 0.4 | 0.1 | 0.1 |

- Total loss over all N training examples

$$\text{Loss} = \sum_{i=1}^{N} \mathrm{CE}(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)})$$

145

Regression Loss

- For regression tasks we often use **Mean Squared Error (MSE)** a.k.a. **L2 loss**
- K-way regression for data point (i):

$$\mathrm{MSE}(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)}) = \sum_{j=1}^{K} (\boldsymbol{y}_j^{(i)} - \hat{\boldsymbol{y}}_j^{(i)})^2$$

i-th data point; j-th target

where:

E.g. | 1.4 | 2.3 | 1.0 | 0.5 | 0.6 |

$\boldsymbol{y}^{(i)} \in \mathbb{R}^k$ = Real valued vector of targets

$\hat{\boldsymbol{y}}^{(i)} \in \mathbb{R}^k$ = Real valued vector of predictions

E.g. | 0.9 | 2.8 | 2.0 | 0.3 | 0.8 |

- Total loss over all N training examples

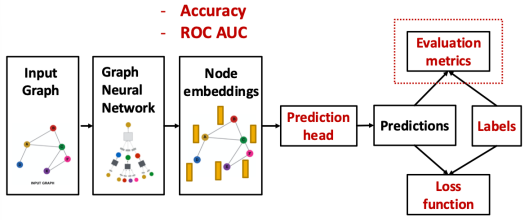
$$\text{Loss} = \sum_{i=1}^{N} \mathrm{MSE}(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)})$$

146

GNN Training Pipeline(4)

(4) How do we measure the success of a GNN?

- **Accuracy**
- **ROC AUC**



147

Evaluation Metrics: Regression

- **We use standard evaluation metrics for GNN**
  - (Content below can be found in any ML course)
  - In practice we will use sklearn for implementation
  - Suppose we make predictions for N data points
- **Evaluate regression tasks on graphs:**
  - Root mean square error (RMSE)

$$\sqrt{\sum_{i=1}^{N} \frac{(\boldsymbol{y}^{(i)} - \hat{\boldsymbol{y}}^{(i)})^2}{N}}$$

  - Mean absolute error (MAE)

$$\frac{\sum_{i=1}^{N} |\boldsymbol{y}^{(i)} - \hat{\boldsymbol{y}}^{(i)}|}{N}$$

148

Evaluation Metrics: Classification

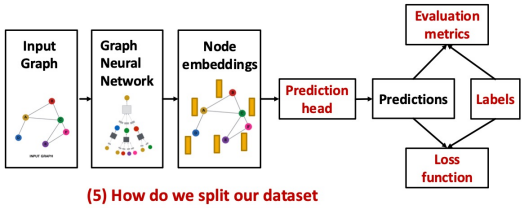
- **Evaluate classification tasks on graphs:**
- **(1) Multi-class classification**
  - We simply report the accuracy

$$\frac{1\left[\text{argmax}\left(\hat{\boldsymbol{y}}^{(i)}\right) = \boldsymbol{y}^{(i)}\right]}{N}$$

- **(2) Binary classification**
  - Metrics sensitive to classification threshold
    - Accuracy
    - Precision / Recall
    - If the range of prediction is [0,1], we will use 0.5 as threshold
  - Metric Agnostic to classification threshold
    - ROC AUC

149

GNN Training Pipeline(5)



(5) How do we split our dataset into train / validation / test set?

Dataset split

150

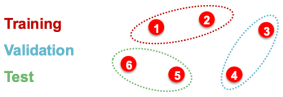
Dataset Split: Fixed/Random Split

- **Fixed split:** We will split our dataset **once**
  - **Training set**: used for optimizing GNN parameters
  - **Validation set**: develop model/hyperparameters
  - **Test set**: held out until we report final performance
- **A concern:** sometimes we cannot guarantee that the test set will really be held out
- **Random split:** we will **randomly split** our dataset into training / validation / test
  - We report **average performance over different random seeds**

151

Why Splitting Graphs is Special

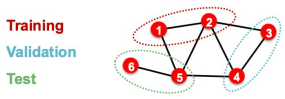
- **Suppose we want to split an image dataset**
  - **Image classification: Each data point is an image**
  - Here **data points are independent**
    - **Image 5 will not affect our prediction on image 1**



152

Why Splitting Graphs is Special

- **Splitting a graph dataset is different!**
  - **Node classification: Each data point is a node**
  - Here **data points are NOT independent**
    - **Node 5 will affect our prediction on node 1,** because it will participate in message passing → affect node 1's embedding



- **What are our options?**

153

Why Splitting Graphs is Special

- **Solution 1 (Transductive setting): The input graph can be observed in all the dataset splits (training, validation and test set).**
- **We will only split the (node) labels**
  - **At training time**, we compute embeddings **using the entire graph**, and train **using node 1&2's labels**
  - **At validation time**, we compute embeddings **using the entire graph**, and **evaluate on node 3&4's labels**

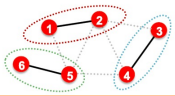
Training
Validation
Test

154

Why Splitting Graphs is Special

- **Solution 2 (Inductive setting): We break the edges between splits to get multiple graphs**
  - **Now we have 3 graphs that are independent. Node 5 will not affect our prediction on node 1 any more**
  - **At training time**, we compute embeddings **using the graph over node 1&2**, and train **using node 1&2's labels**
  - **At validation time**, we compute embeddings **using the graph over node 3&4**, and **evaluate on node 3&4's labels**

Training
Validation
Test



155

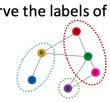
Transductive / Inductive Settings

- **Transductive setting:** training / validation / test sets are **on the same graph**
  - The **dataset consists of one graph**
  - **The entire graph can be observed in all dataset splits, we only split the labels**
  - Only applicable to **node / edge** prediction tasks
- **Inductive setting:** training / validation / test sets are **on different graphs**
  - The **dataset consists of multiple graphs**
  - Each split can **only observe the graph(s) within the split.** A successful model should **generalize to unseen graphs**
  - Applicable to **node / edge / graph** tasks

156

Example: Node Classification

- **Transductive** node classification
  - **All the splits can observe the entire graph structure**, but can only observe the labels of their respective nodes



- **Inductive** node classification
  - Suppose we have a dataset of 3 graphs
  - **Each split contains an independent graph**

157

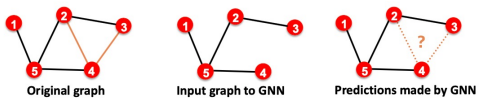
Example: Graph Classification

- Only the **inductive setting** is well defined for **graph classification**
  - Because **we have to test on unseen graphs**
  - Suppose we have a dataset of 5 graphs. Each split will contain independent graph(s).

158

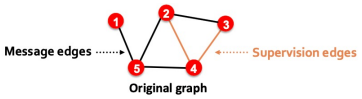
Example: Link Prediction

- **Goal of link prediction: predict missing edges**
- **Setting up link prediction is tricky:**
  - Link prediction is an unsupervised / self-supervised task. We need to **create the labels** and **dataset splits** on our own
  - Concretely, we need to **hide some edges from the GNN** and **let the GNN predict if the edges exist**



159

Setting up Link Prediction

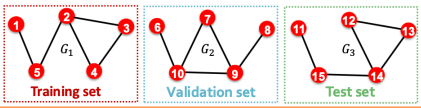


- For link prediction, we will split edges twice
- Step 1: Assign 2 types of edges in the original graph
  - Message edges: Used for GNN message passing
  - Supervision edges: Use for computing objectives
  - After step 1:
    - Only message edges will remain in the graph
    - Supervision edges are used as supervision for edge predictions made by the model, will not be fed into GNN

160

Setting up Link Prediction

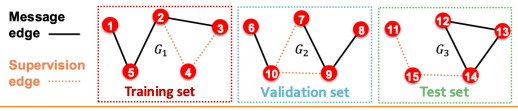
- Step 2: Split edges into train / validation / test
- Option 1: Inductive link prediction split
  - Suppose we have a dataset of 3 graphs. Each inductive split will contain an independent graph



161

Setting up Link Prediction

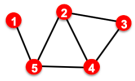
- Step 2: Split edges into train / validation / test
- Option 1: Inductive link prediction split
  - Suppose we have a dataset of 3 graphs. Each inductive split will contain an independent graph
  - In train or val or test set, each graph will have 2 types of edges: message edges + supervision edges
    - Supervision edges are not the input to GNN



162

Setting up Link Prediction

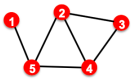
- **Option 2: Transductive link prediction split:**
  - This is the default setting when people talk about link prediction
  - Suppose we have a dataset of 1 graph



163

Setting up Link Prediction

- **Option 2: Transductive link prediction split:**
  - By definition of "transductive", the entire graph can be observed in all dataset splits
    - But since edges are both part of graph structure and the supervision, we need to hold out validation / test edges
    - To train the training set, we further need to hold out supervision edges for the training set

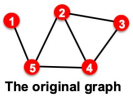


  - **Next:** we will show the exact settings

164

Setting up Link Prediction

- **Option 2: Transductive link prediction split:**



The original graph

(1) At training time:
Use training message edges to predict training supervision edges

(2) At validation time:
Use training message edges & training supervision edges to predict validation edges

(3) At test time:
Use training message edges & training supervision edges & validation edges to predict test edges

165

Setting up Link Prediction

- **Option 2: Transductive link prediction split:**

Why do we use growing number of edges?
After training, supervision edges are known to GNN.
Therefore, an ideal model should use supervision edges in message passing at validation time.
The same applies to the test time.

(1) At training time: Use **training message edges** to predict **training supervision edges**

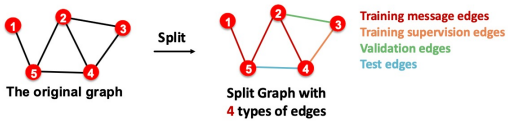
(2) At validation time: Use **training message edges** & **training supervision edges** to predict **validation edges**

(3) At test time: Use **training message edges** & **training supervision edges** & **validation edges** to predict **test edges**

166

Setting up Link Prediction

- **Summary: Transductive link prediction split:**



- **Note:** Link prediction settings are tricky and complex. You may find papers do link prediction differently.
- Luckily, we have full support in **PyG and GraphGym**

167

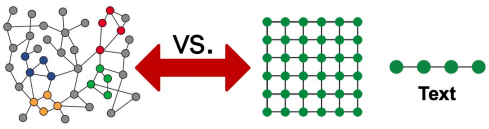
Learning with graphs

168

Why is Graph Deep Learning Hard?

Graphs are complex.

- Complex topological structure (*i.e.*, no spatial locality like grids)



- No fixed node ordering or reference point

169

Theory of GNNs

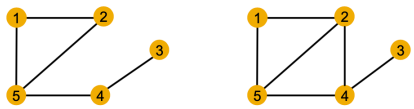
How powerful are GNNs?

- **Many GNN models have been proposed (e.g., GCN, GAT, GraphSAGE, design space).**
- **What is the expressive power (ability to distinguish different graph structures) of these GNN models?**
- **How to design a maximally expressive GNN model?**

170

Graph Isomorphism Problem

- **A classical expressivity test is graph isomorphism.**
  - **Given a pair of graphs, can we tell if they are isomorphic?**

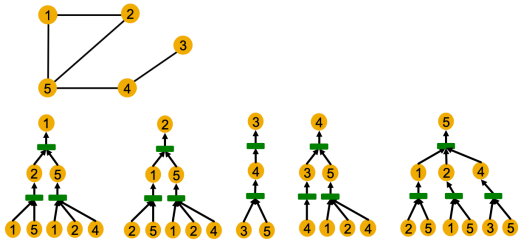


- **Open problem (a polynomial time algorithm does not exist, has not been proven NP-complete).**

171

A simpler Problem

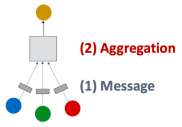
- Given a pair of nodes with different neighborhood structures. Is there a GNN that can always tell them apart?



172

Background: A Single GNN Layer

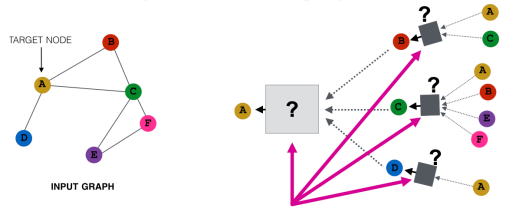
- **We focus on message passing GNNs:**
  - **(1) Message**: each node computes a message
    $$\mathbf{m}_u^{(l)} = \mathrm{MSG}^{(l)}\left(\mathbf{h}_u^{(l-1)}\right), u \in \{N(v) \cup v\}$$
  - **(2) Aggregation**: aggregate messages from neighbors
    $$\mathbf{h}_v^{(l)} = \mathrm{AGG}^{(l)}\left(\left\{\mathbf{m}_u^{(l)}, u \in N(v)\right\}, \mathbf{m}_v^{(l)}\right)$$



173

Background: Many GNN Models

- Many GNN models have been proposed:
  - GCN, GraphSAGE, GAT, Design Space etc.

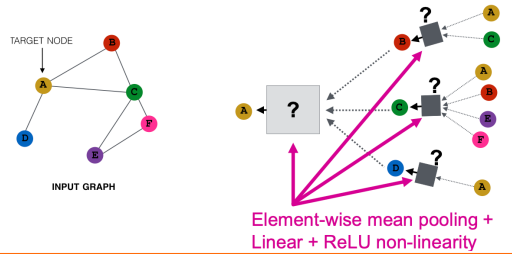


Different GNN models use different neural networks in the box

174

GNN Model Example(1)

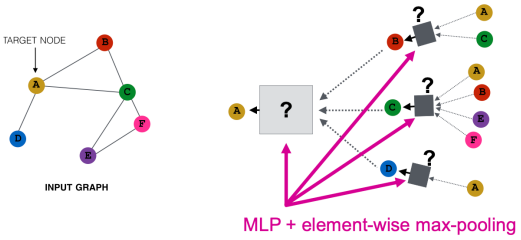
- **GCN (mean-pool)** [Kipf and Welling ICLR 2017]



175

GNN Model Example(2)

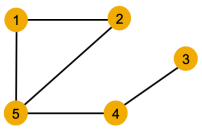
- **GraphSAGE (max-pool)** [Hamilton et al. NeurIPS 2017]



176

Note: Node Colors

- We use node same/different **colors** to represent nodes with same/different features.
  - For example, the graph below assumes all the nodes share the same feature.

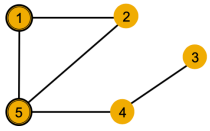


- **Key question: How well can a GNN distinguish different graph structures?**

177

Local Neighborhood Structures

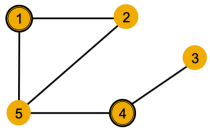
- We specifically consider **local neighborhood structures** around each node in a graph.
  - **Example:** Nodes 1 and 5 have **different** neighborhood structures because they have different node degrees.



178

Local Neighborhood Structures

- We specifically consider **local neighborhood structures** around each node in a graph.
  - **Example:** Nodes 1 and 4 both have the same node degree of 2. However, they still have **different** neighborhood structures because **their neighbors have different node degrees.**

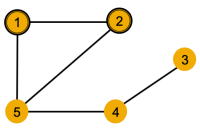


Node 1 has neighbors of degrees 2 and 3.
Node 4 has neighbors of degrees 1 and 3.

179

Local Neighborhood Structures

- We specifically consider **local neighborhood structures** around each node in a graph.
  - **Example:** Nodes 1 and 2 have the **same** neighborhood structure because **they are symmetric within the graph.**



Node 1 has neighbors of degrees 2 and 3.
Node 2 has neighbors of degrees 2 and 3.
And even if we go a step deeper to 2nd hop neighbors, both nodes have the same degrees (Node 4 of degree 2)

180

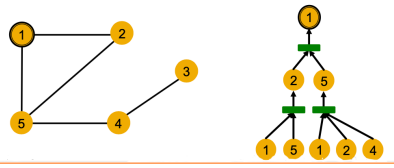
Local Neighborhood Structures

- **Key question**: **Can GNN node embeddings distinguish different node's local neighborhood structures?**
  - If so, when? If not, when will a GNN fail?
- **Next**: **We need to understand how a GNN captures local neighborhood structures.**
  - **Key concept**: **Computational graph**

181

Computational Graph (1)

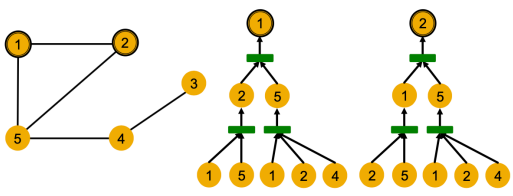
- **In each layer, a GNN aggregates neighboring node embeddings.**
- A GNN generates node embeddings through a computational graph defined by the neighborhood.
  - **Ex:** Node 1's computational graph (2-layer GNN)



182

Computational Graph (2)

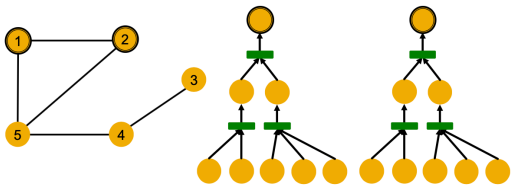
- **Ex:** Nodes 1 and 2's computational graphs.



183

Computational Graph (3)

- **Ex:** Nodes 1 and 2's computational graphs.
- **But GNN only sees node features (not IDs):**

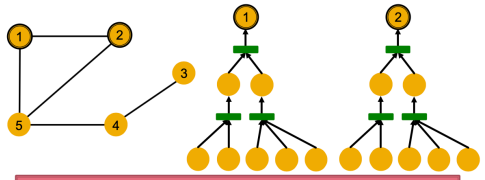


184

Computational Graph (4)

- **A GNN will generate the same embedding for nodes 1 and 2 because:**
  - Computational graphs are the same.
  - Node features (colors) are identical.

Note: GNN does not care about node ids, it just aggregates features vectors of different nodes

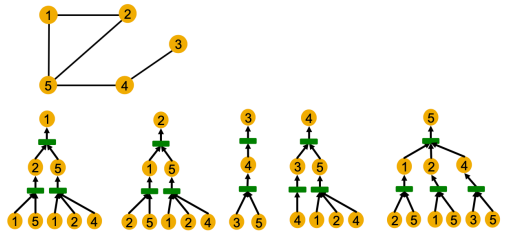


GNN won't be able to distinguish nodes 1 and 2

185

Computational Graph

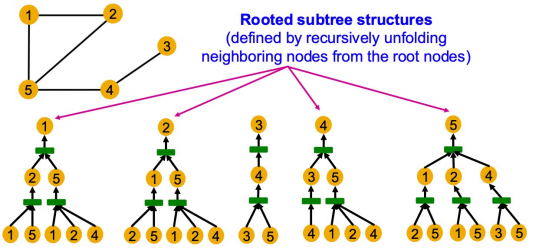
- **In general, different local neighborhoods define different computational graphs**



186

Computational Graph

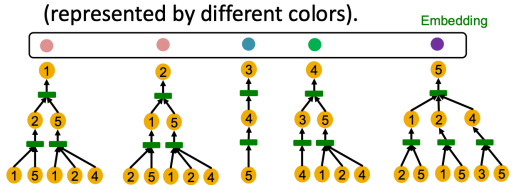
- Computational graphs are identical to **rooted subtree structures** around each node.



187

Computational Graph

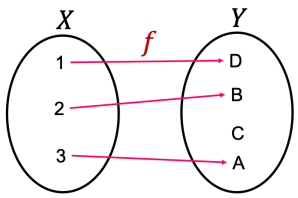
- GNN's node embeddings capture rooted subtree structures.
- Most expressive GNN maps different rooted subtrees into different node embeddings (represented by different colors).



188

Recall: Injective Function

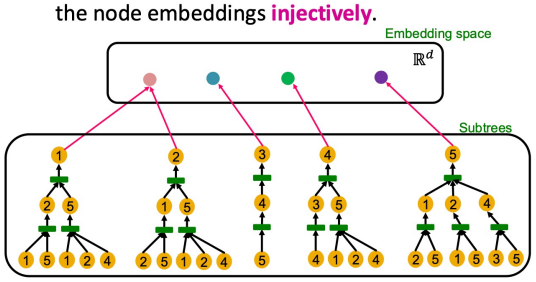
- **Function** $f: X \to Y$ is **injective** if it maps different elements into different outputs.
- **Intuition:** f retains all the information about input.



189

How Expressive is a GNN?

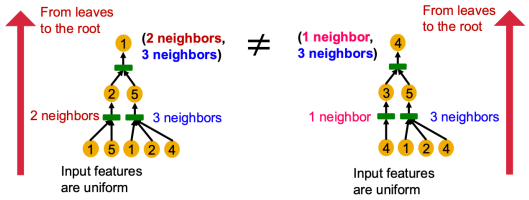
- Most expressive GNN should map subtrees to the node embeddings **injectively**.



190

How Expressive is a GNN?

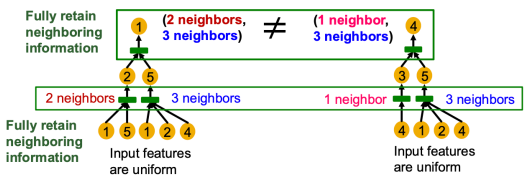
- **Key observation**: Subtrees of the same depth can be recursively characterized from the leaf nodes to the root nodes.



191

How Expressive is a GNN?

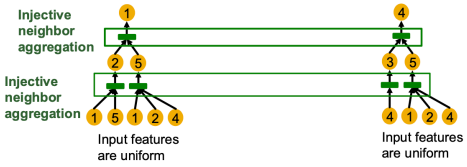
- If each step of GNN's aggregation **can fully retain the neighboring information**, the generated node embeddings can distinguish different rooted subtrees.



192

How Expressive is a GNN?

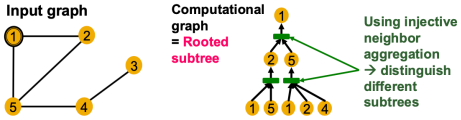
- In other words, most expressive GNN would use an **injective neighbor aggregation** function at each step.
  - Maps different neighbors to different embeddings.



193

How Expressive is a GNN?

- **Summary so far**
  - To generate a node embedding, GNNs use a computational graph corresponding to a subtree rooted around each node.

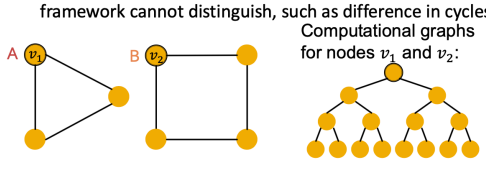


  - GNN can fully distinguish different subtree structures if **every step of its neighbor aggregation is injective.**

194

Improving GNN's Power

- **Can the expressive power of GNNs be improved?**
  - There are basic graph structures that existing GNN framework cannot distinguish, such as difference in cycles.



  - GNNs' expressive power **can be improved** to resolve the above problem. [You et al. AAAI 2021, Li et al. NeurIPS 2020, Kanatsoulis et al. ICLR 2024]

195

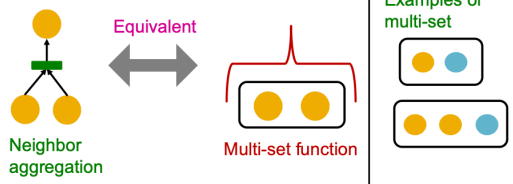
Expressive Power of GNNs

- **Key observation: Expressive power of GNNs can be characterized by that of neighbor aggregation functions they use.**
  - A more expressive aggregation function leads to a more expressive GNN.
  - Injective aggregation function leads to the most expressive GNN.
- **Next:**
  - Theoretically analyze expressive power of aggregation functions.

196

Neighbor Aggregation

- **Observation: Neighbor aggregation** can be abstracted as **a function over a multi-set** (a set with repeating elements).



197

Neighbor Aggregation

- **Next: We analyze aggregation functions of two popular GNN models**
  - **GCN** (mean-pool) [Kipf & Welling, ICLR 2017]
    - Uses **element-wise** mean pooling over neighboring node features

$$\text{Mean}(\{x_u\}_{u \in N(v)})$$

  - **GraphSAGE** (max-pool) [Hamilton et al. NeurIPS 2017]
    - Uses **element-wise** max pooling over neighboring node features

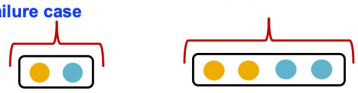
$$\text{Max}(\{x_u\}_{u \in N(v)})$$

198

Neighbor Aggregation: Case Study

- **GCN (mean-pool)** [Kipf & Welling ICLR 2017]
  - Take **element-wise mean**, followed by linear function and ReLU activation, i.e., $\max(0, x)$.
  - **Theorem** [Xu et al. ICLR 2019]
    - GCN's aggregation function **cannot distinguish different multi-sets with the same color proportion**.

Failure case



- **Why?**

199

Neighbor Aggregation

- For simplicity, we assume node features (colors) are represented by **one-hot encoding**.
  - **Example:** If there are two distinct colors:

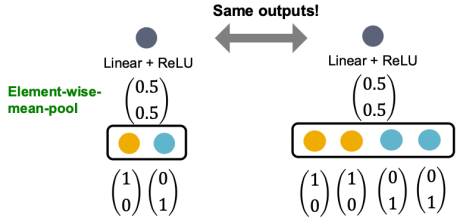
$$\text{●} = \begin{pmatrix}1\\0\end{pmatrix} \qquad \text{●} = \begin{pmatrix}0\\1\end{pmatrix}$$

  - This assumption is sufficient to illustrate how GCN fails.

200

Neighbor Aggregation: Case Study

- **GCN (mean-pool)** [Kipf & Welling ICLR 2017]
  - **Failure case illustration**

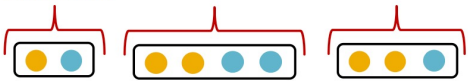


201

Neighbor Aggregation: Case Study

- **GraphSAGE (max-pool)** [Hamilton et al. NeurIPS 2017]
  - Apply an MLP, then take **element-wise max**.
  - **Theorem** [Xu et al. ICLR 2019]
    - GraphSAGE's aggregation **function cannot distinguish different multi-sets with the same set of distinct colors.**

Failure case

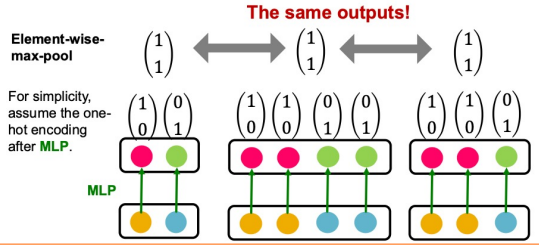


- **Why?**

202

Neighbor Aggregation: Case Study

- **GraphSAGE (max-pool)** [Hamilton et al. NeurIPS 2017]
  - **Failure case illustration**



203

Summary So Far

- We analyzed the **expressive power of GNNs.**
- **Main takeaways**:
  - Expressive power of GNNs can be characterized by that of the neighbor aggregation function.
  - Neighbor aggregation is a function over multi-sets (sets with repeating elements)
  - GCN and GraphSAGE's aggregation functions fail to distinguish some basic multi-sets; hence **not injective**.
  - Therefore, GCN and GraphSAGE are **not** maximally powerful GNNs.

204

Design Most Expressive GNNs

- **Our goal: Design maximally powerful GNNs in the class of message-passing GNNs.**
- **This can be achieved by designing injective neighbor aggregation function over multi-sets.**
- **Here, we design a neural network that can model injective multiset function.**

205

Injective Multi-Set Function

Theorem [Xu et al. ICLR 2019]

Any injective multi-set function can be expressed as:

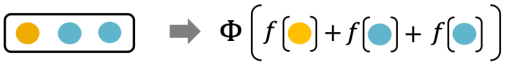
Some non-linear function

$$\Phi\left(\sum_{x \in S} f(x)\right)$$

Some non-linear function

Sum over multi-set

S : multi-set



$$\Phi\left(f(\bigcirc)+f(\bigcirc)+f(\bigcirc)\right)$$

206

Injective Multi-Set Function

Proof Intuition: [Xu et al. ICLR 2019]

f produces one-hot encodings of colors. Summation of the one-hot encodings retains all the information about the input multi-set.

$$\Phi\left(\sum_{x \in S} f(x)\right)$$

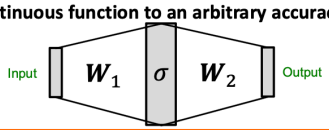
Example: $\Phi\left(f(\bigcirc)+f(\bigcirc)+f(\bigcirc)\right)$

One-hot $\begin{pmatrix}1\\0\end{pmatrix} + \begin{pmatrix}0\\1\end{pmatrix} + \begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}1\\2\end{pmatrix}$

207

Universal Approximation Theorem

- How to model Φ and f in $\Phi(\sum_{x \in S} f(x))$?
- We use a Multi-Layer Perceptron (MLP).
- **Theorem: Universal Approximation Theorem**
  [Hornik et al., 1989]
  - 1-hidden-layer MLP with sufficiently-large hidden dimensionality and appropriate non-linearity $\sigma(\cdot)$ (including ReLU and sigmoid) can **approximate any continuous function to an arbitrary accuracy**.



208

Injective Multi-Set Function

- **We have arrived at a neural network that can model any injective multiset function.**

$$\text{MLP}_\Phi\left(\sum_{x \in S} \text{MLP}_f(x)\right)$$

  - In practice, MLP hidden dimensionality of 100 to 500 is sufficient.

209

Most Expressive GNN

- **Graph Isomorphism Network (GIN)** [Xu et al. ICLR 2019]
  - Apply an MLP, element-wise **sum**, followed by another MLP.

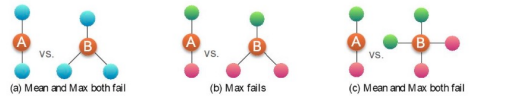
$$\text{MLP}_\Phi\left(\sum_{x \in S} \text{MLP}_f(x)\right)$$

- **Theorem** [Xu et al. ICLR 2019]
  - GIN's neighbor aggregation function is injective.
- **No failure cases!**
- **GIN is THE most expressive GNN in the class of message-passing GNNs we have introduced!**

210

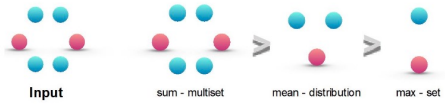
Discussion: The Power of Pooling

Failure cases for mean and max pooling:



Colors represent feature values

Ranking by discriminative power:



211

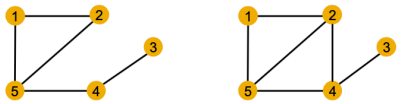
Full Model of GIN

- **So far:** We have described the neighbor aggregation part of GIN.
- We now describe the full model of GIN by relating it to **WL graph kernel** (traditional way of obtaining graph-level features).
  - We will see how GIN is a "neural network" version of the WL graph kernel.

212

Graph Isomorphism Problem

- **A classical expressivity test is Graph isomorphism.**
  - **Given a pair of graphs, can we tell if they are isomorphic?**



- **Open problem (a polynomial time algorithm does not exist, has not been proven NP-complete)**

213

Relation to WL Graph Kernel

Color refinement algorithm in WL kernel.

- **Given:** A graph G with a set of nodes V.
  - Assign an initial color $c^{(0)}(v)$ to each node v.
  - Iteratively refine node colors by

$$c^{(k+1)}(v) = \text{HASH}\left(c^{(k)}(v), \{c^{(k)}(u)\}_{u \in N(v)}\right),$$

where HASH maps different inputs to different colors.

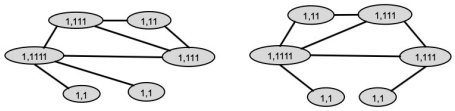
  - After K steps of color refinement, $c^{(K)}(v)$ summarizes the structure of K-hop neighborhood

Color Refinement(1)

Example of color refinement given two graphs

- Assign initial colors

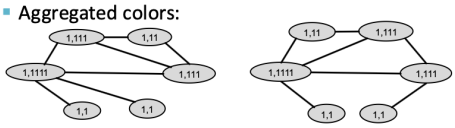
- Aggregate neighboring colors



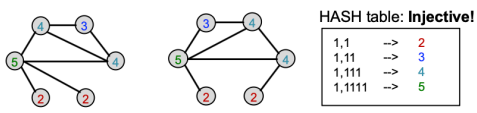
Color Refinement(2)

Example of color refinement given two graphs

- Aggregated colors:



- **Injectively** HASH the aggregated colors



HASH table: **Injective!**

1,1	-->	2
1,11	-->	3
1,111	-->	4
1,1111	-->	5

Color Refinement(3)

Example of color refinement given two graphs

- Process continues until a stable coloring is reached
- Two graphs are considered **isomorphic** if they have the same set of colors.

217

The Complete GIN Model

- **GIN uses a neural network to model the injective HASH function.**

$$c^{(k+1)}(v) = \text{HASH}\left(c^{(k)}(v), \{c^{(k)}(u)\}_{u \in N(v)}\right)$$

- Specifically, we will model the injective function over the tuple:

$$\left(c^{(k)}(v), \{c^{(k)}(u)\}_{u \in N(v)}\right)$$

Root node features

Neighboring node colors

218

The Complete GIN Model

Theorem (Xu et al. ICLR 2019)

Any injective function over the tuple

Root node feature $\left(c^{(k)}(v), \{c^{(k)}(u)\}_{u \in N(v)}\right)$ Neighboring node features

can be modeled as

$$\text{MLP}_\Phi\left((1+\epsilon) \cdot \text{MLP}_f(c^{(k)}(v)) + \sum_{u \in N(v)} \text{MLP}_f(c^{(k)}(u))\right)$$

where ϵ is a learnable scalar.

219

The Complete GIN Model

- If input feature $c^{(0)}(v)$ is represented as one-hot, **direct summation is injective**.

Example: $\Phi\left(\begin{pmatrix}1\\0\end{pmatrix}+\begin{pmatrix}0\\1\end{pmatrix}+\begin{pmatrix}0\\1\end{pmatrix}\right)=\begin{pmatrix}1\\2\end{pmatrix}$

- We only need Φ to ensure the injectivity.

$$\text{GINConv}\left(c^{(k)}(v),\{c^{(k)}(u)\}_{u\in N(v)}\right)=\text{MLP}_\Phi\left((1+\epsilon)\cdot c^{(k)}(v)+\sum_{u\in N(v)}c^{(k)}(u)\right)$$

Root node features; Neighboring node features; This MLP can provide "one-hot" input feature for the next layer.

220

The Complete GIN Model

- **GIN's node embedding updates**
- **Given:** A graph G with a set of nodes V.
  - Assign an **initial vector** $c^{(0)}(v)$ to each node v.
  - Iteratively update node vectors by

$$c^{(k+1)}(v)=\text{GINConv}\left(\left\{c^{(k)}(v),\{c^{(k)}(u)\}_{u\in N(v)}\right\}\right),$$

Differentiable color HASH function

where GINConv maps different inputs to different embeddings.

  - After K steps of GIN iterations, $c^{(K)}(v)$ summarizes the structure of K-hop neighborhood.

221

GIN and WL Graph Kernel

- **GIN can be understood as differentiable neural version of the WL graph Kernel:**

	Update target	Update function
WL Graph Kernel	Node colors (one-hot)	HASH
GIN	Node embeddings (low-dim vectors)	GINConv

- **Advantages of GIN over the WL graph kernel are:**
  - Node embeddings are **low-dimensional**; hence, they can capture the fine-grained similarity of different nodes.
  - Parameters of the update function can be **learned for the downstream tasks.**

222

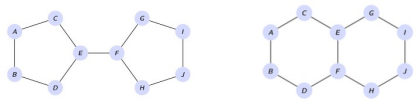
Expressive Power of GIN

- Because of the relation between GIN and the WL graph kernel, their expressive is exactly the same.
  - If two graphs can be distinguished by GIN, they can be also distinguished by the WL kernel, and vice versa.
- How powerful is this?
  - WL kernel has been both theoretically and empirically shown to distinguish most of the real-world graphs [Cai et al. 1992].
  - Hence, GIN is also powerful enough to distinguish most of the real graphs!

223

Some bad news

- The WL kernel cannot distinguish between some basic graph structures, e.g.,

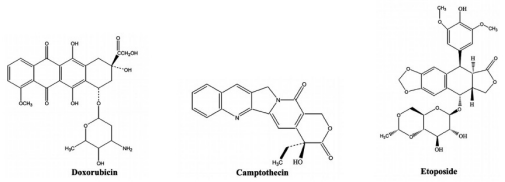


- The WL kernel cannot count important sub-structures, e.g., cycles and cliques.

224

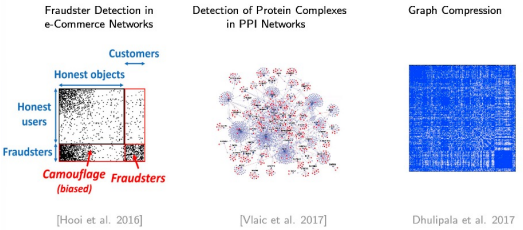
Why do we care about cycles?

- The **bonds between different atoms** in chemical compounds form **cycles**.
- The hexagon is the **most common polygon** in organic compounds ⇒ yields stable materials.



225

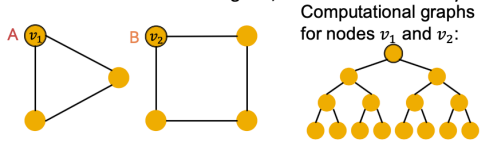
Why cliques and quasi-cliques



226

Improving GNNs' Power

- **Can the expressive power of GNNs be improved?**
  - There are basic graph structures that existing GNN framework cannot distinguish, such as difference in cycles.



Computational graphs for nodes v_1 and v_2:

  - GNNs' expressive power **can be improved** to resolve the above problem. [You et al. AAAI 2021, Li et al. NeurIPS 2020, Kanatsoulis et al. ICLR 2024]

227

Summary

- We design a neural network that can model an injective multi-set function.
- We use the neural network for neighbor aggregation function and arrive at **GIN**---the most expressive GNN model.
- The key is to use **element-wise sum pooling**, instead of mean-/max-pooling.
- GIN is closely related to the WL graph kernel.
- Both GIN and WL graph kernel can distinguish most of the real graphs!

228

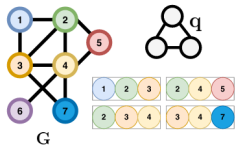
Learning with graphs

- 1. Generating Graph Representation
- 2. Deep Learning for Graphs
- 3. Graph Neural Networks
- Special topic: GNNs for Subgraph Counting

229

GNNs for Subgraph Counting

- Subgraph Counting: a fundamental problem within graph analytics
  - Input: Data graph $G = (V, E)$, Query graph $q = (V_q, E_q)$
  - Output: The number of all subgraphs in G that is isomorphic to q



 Najafi, Mohammad Matin, et al. "BEACON: A Benchmark for Efficient and Accurate Counting of Subgraphs."

230

GNNs for Subgraph Counting

- Induced vs. Non-Induced Subgraph Counting
  - Induced Subgraph: The induced subgraph of G on a subset of nodes S is the subgraph $G[S] = (S, E_S), E_S = \{(u, v) \in E : u \in S, v \in S\}$
  - Induced counting(Default) vs. Non-Induced counting

- Induced:
  $C_I(G; G^{[P_1]}) = 1, C_I(G; G^{[P_2]}) = 0$
- Non-Induced:
  $C_S(G; G^{[P_1]}) = 1, C_S(G; G^{[P_2]}) = 1$

231

Can GNNs Count Subgraph?

- Known results of WL algorithms:
  - 1-WL and 2-WL have equivalent discrimination power.
  - 2-WL can perform subgraph-count of star-shaped patterns.
  - For $k \geq 2$, (k+1)-WL is strictly more powerful than k-WL.
  - k-WL is able to perform both induced-subgraph-count and subgraph count of patterns consisting of at most k nodes.
  - Running T iterations of k-WL cannot perform induced-subgraph-count of any path pattern of $(k+1)2^T$ or more nodes.
    - Path pattern is an unattributed graph:

$$H_m = (V^{[H_m]}, E^{[H_m]}), V^{[H_m]} = [m], E^{[H_m]} = \{(i, i+1): 1 \leq i < m\} \cup \{(i+1, i): i \leq i < m\}$$

Can GNNs Count Subgraph?

- Message Passing Neural Netowrk(MPNNs):

$$m_i^{(t+1)} = \sum_{\mathcal{N}(i)} M_t(h_i^{(t)}, h_j^{(t)}, e_{i,j}), \qquad h_i^{(t+1)} = U_t(h_i^{(t)}, m_i^{(t+1)}),$$

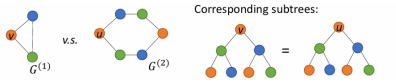
- MPNNs cannot induced-subgraph-count any connected pattern with 3 or more nodes
  - Two attributed graphs that are indistinguishable by 2-WL cannot be distinguished by any MPNN.
  - 2-WL cannot induced-subgraph-count any connected pattern with 3 or more nodes.
- MPNNs can perform subgraph-count of star-shaped patterns.

Can GNNs Count Subgraph?

- Invariant Graph Networks(IGNs): $F = m \circ h \circ L^T \circ \sigma \circ \cdots \circ \sigma \circ L^1$,
  - L^t is a linear equivariant layer
  - σ is a pointwise activation function
  - h is a linear invariant layer
  - m is an MLP
- 2-IGNs are exactly as powerful as 2-WL
  - 2-IGNs cannot perform induced-subgraph-count of any connected pattern with 3 or more nodes.
  - 2-IGNs can perform subgraph-count of star-shaped patterns.
- k-IGNs are no less powerful than k-WL
  - k-IGNs and k-WL can subgraph-count and induced subgraph-count any pattern of size k.

Limitations of MPNNs

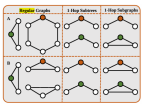
- The information about the distance between multiple nodes is lost.
  - MPNNs have limited power in capturing the position/location of a given node with respect to another node in the graph.
  - Many nodes may share similar subtrees, and thus, MPNNs produces the same representation for them although the nodes may be located at different locations in the graph.
- The information about cycles is lost.



235

What's especially hard for MPNNs?

- Regular graphs: Each vertex has the same number of neighbors



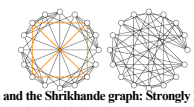
MPNNs fail to distinguish graphs A and B as they are 2-regular graphs with identical 1-hop subtrees.

- Strongly regular graphs: A SRG is a regular graph which can be defined with four parameters (v, k, λ, μ):
  - v: The number of nodes
  - k: The number of neighbors each vertex has.
  - λ: The number of common neighbors between any two adjacent nodes.
  - μ: The number of common neighbors between any two non-adjacent vertices.

236

What's especially hard for MPNNs?

- Regular graphs: Each vertex has the same number of neighbors
- Strongly regular graphs: A SRG is a regular graph which can be defined with four parameters (v, k, λ, μ):
  - v: The number of nodes
  - k: The number of neighbors each vertex has.
  - λ: The number of common neighbors between any two adjacent nodes.
  - μ: The number of common neighbors between any two non-adjacent vertices.



Rook's 4x4 graph and the Shrikhande graph: Strongly Regular non-isomorphic graphs with parameters SR(16,6,2,2), which can not be distinguished by 3-WL tests.

237

Methods to Enhance MPNNs

- Injecting Random Attributes: inject each node with a unique attribute.
  - Random Permutations: RP-GNN randomly assigns an order of nodes as their extra attributes
  - Distances to random anchor sets: PGNN uses node positional embeddings as extra attributes.
  - Random signed Laplacian eigenmap: LE-GNN uses the randomly perturbed Laplacian eigenmaps as the additional attributes.

Strictly Stronger than the original MPNNs

238

Methods to Enhance MPNNs

- Injecting Deterministic Distance Attributes: If MPNNs are paired with some distance information, then the composite model must achieve more expressive power.
  - Distance Encoding: DE-GNN defined distance encoding $\zeta(u|S)$ as an extra node attribute. $\zeta(u|S) = \sum_{v \in S} \text{MLP}(\zeta(u|v))$
  - Identity-aware GNN: ID-GNN attaches each node u with an extra binary attribute $\zeta_{ID}(u|\{v\})$ to indicate the identity of node v $\zeta_{ID}(u|\{v\}) = \begin{cases} 1 & \text{if } u = v, \\ 0 & \text{o.w.} \end{cases}$

The injection of additional attributes focuse on the message passing framework of GNNs, which leverages the sparsity of real-world graphs and thus the enhancement is not stable and strong enough.

239

Methods to Enhance MPNNs

- Higher-order Graph Neural Networks: Removes the need for sparsity and mimic higher-dimensional WL tests to achieve more expressive power.
  - k-WL induced GNNs(k-GNNs):

$$\mathbf{h}_j^{(l+1)} = \text{MLP}(\mathbf{h}_j^{(l)} \oplus \sum_{V_{j'}:N_{k-GNN}(V_j)} \mathbf{h}_{j'}^{(l)}), \quad \forall\, k\text{-sized node sets } V_j,$$

$$N_{k-GNN}(V_j) = \{V_{j'} \,|\, |V_{j'} \cap V_j| = k-1\}$$

k-GNN is at most as powerful as the k-WL test. To be more expressive than MP-GNN, k =3 is needed.

240

Methods to Enhance MPGNNs

- Higher-order Graph Neural Networks: Removes the need for sparsity and mimic higher-dimensional WL tests to achieve more expressive power.
  - Invariant and equivariant GNNs: Graphs are viewed as tensors and NNs take these tensors as input. The NNs are designed to be invariant to the order of tensor indices.

$$f_{k-inv} = g_{inv} \circ g_{equ}^{(L)} \circ \sigma \circ g_{equ}^{(L-1)} \circ \sigma \cdots \circ \sigma \circ g_{equ}^{(1)},$$

f_{k-inv} is also at most as powerful as the k-WL test, slightly weaker than k-IGNs.