CS224w

CH8: Applications of Graph Neural Networks

IE LAB
Interlligence Engineering

Presenter: Jaeyoung Cheong

1. GNN Augmentation for GNNs

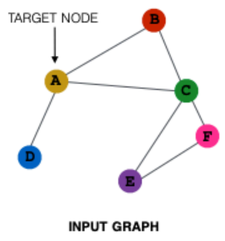
Common assumption:

Input graph ⇔ Computational graph

Proposed new idea:

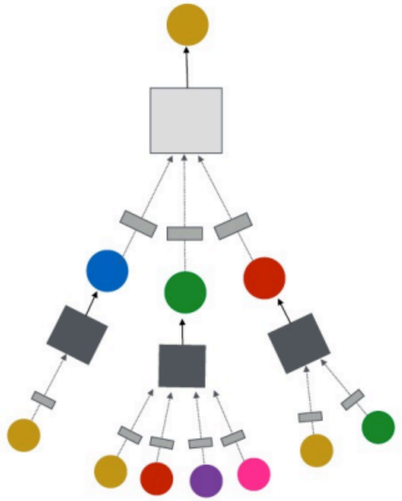
do not need 1:1 between input graph & Computational graph (GNN)

⇒ <u>Breaking the common assumption</u>



Idea: Raw input graph ≠ computational graph

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



(4) Graph augmentation

1. GNN Augmentation for GNNs

Reasons for breaking it

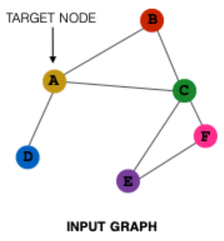
Motivation: the raw input graph is unlikely to optimal computational graph

1. Features (in the node level)

- The input graph lacks features
- Features may be hard to encode

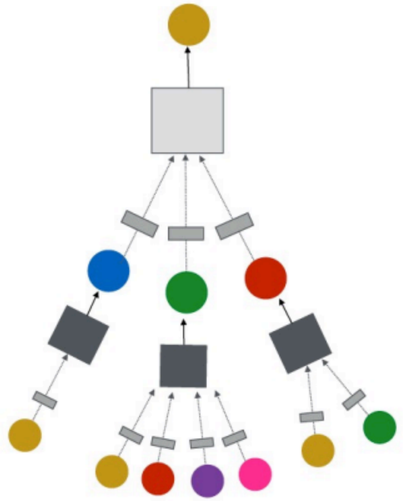
2. Structures

- input graph **too sparse** ⇒ inefficient message passing (a.k.a mp)
  ⌊ a lot of iterations
- input graph **too dense** ⇒ message passing is to costly
  ⌊ A C.Ronaldo node in the instagram has many followers, so mp from them to Ronaldo would be too expensive
- input graph **too large** ⇒ cannot fit the computational graph into a GPU



Idea: Raw input graph ≠ computational graph

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



(4) Graph augmentation

⇒ How can we get better the gnn embeddings?

Methods of the Augmentation

1. Features augmentation

- The input graph lacks features → **feature augmentation**

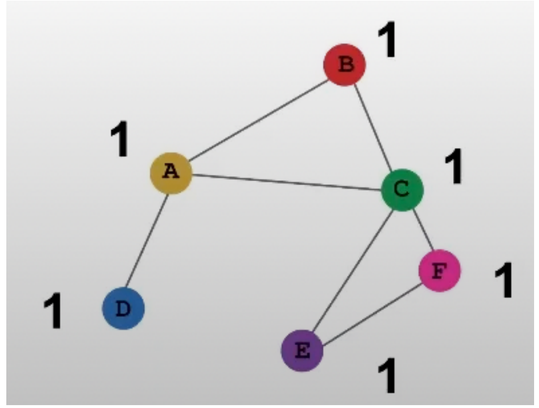
2. Structures augmentation

- input graph **too sparse** → **Add virtual nodes/edges**
  - a lot of iterations
- input graph **too dense** → **Sample neighbors when doing mp**
  - A C.Ronaldo node in the instagram has many followers, so mp from them to Ronaldo would be too expensive
- input graph **too large** → **Sample subgraph to compute embeddings**

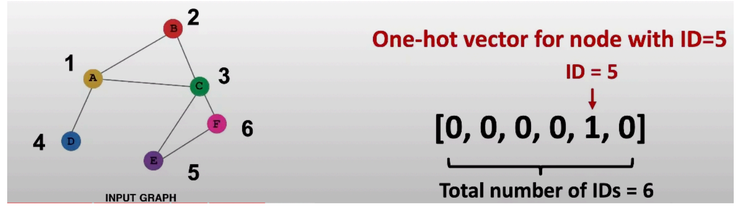
Why do we need feature augmentation?

1. Input graph does not have node features (∃only adj mat)

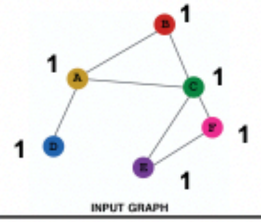
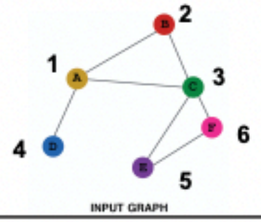
- (a) Standard approach: Assign **constant values** to nodes
  - What aggregation does? To count how many neighbors



- (b) Alternative approach: Assign **unique IDs** to nodes
  - * The IDs are converted to 1-hot vectors
  - * Ordering of the nodes is arbitrary
  - **Advantage**: to learn expressive models
    - The model knows what are the IDs of the neighbors of the node
  - **Disadvantage**:
    - hard to generalize 1-hot encoding across different graphs
    - it might be costly, for # attributes = # nodes



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

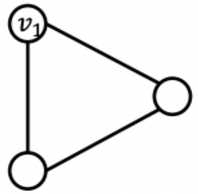
	Constant node feature	One-hot node feature
		
Expressive power	**Medium**. All the nodes are identical, but GNN can still learn from the graph structure	**High**. Each node has a unique ID, so node-specific information can be stored
Inductive learning (Generalize to unseen nodes)	**High**. Simple to 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**. $O(\lvert V \rvert)$ dimensional feature, cannot apply to large graphs
Use cases	**Any graph, inductive settings** (generalize to new nodes)	**Small graph, transductive settings** (no new nodes)

Expressiveness ↑ ⇔ Generalization ↓

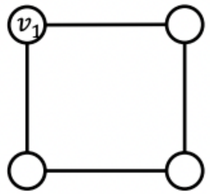
Why do we need feature augmentation?

2. Certain structures are hard to learn by GNN

v_1 resides in a cycle with length 3

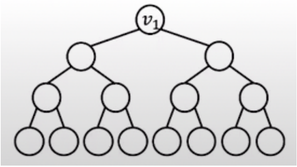


v_1 resides in a cycle with length 4



- **Example**: Cycle count feature
  - Can GNN learn the length of a cycle that v1 resides in? **No**
    - **Reason**: Since $v1$ has two neighbors and $v2$ also has two, so that <u>there are not discriminative node features btn them</u>, the GNN do not differentiate the length of a cycle.
      - ⇒ Their computational graphs will be the same binary tree
      - ⇒ Their embeddings will be the same.

The computational graphs for node v_1 are always the same



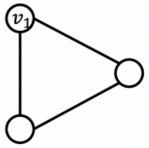
- Solution: Use **cycle count** as augmented node features

We start from cycle with length 0

Augmented node feature for v_1

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

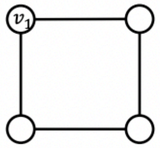
v_1 resides in a cycle with length 3



Augmented node feature for v_1

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

v_1 resides in a cycle with length 4



- Other used augmented features

: Node degree, Centrality, PageRank … studied in the Lecture2

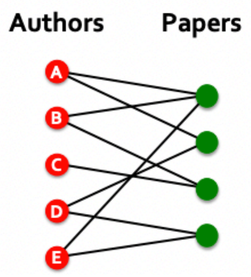
How do we Augment sparse graphs?

1. Add virtual edges

- **Common Approach**:
  [Additional information]
  Connect 2-hop neighbors via virtual edges
- **Intuition:**
  Let A be adj-mat.
  Then, after connecting 2-hop, changing A into $A + A^2$

  In the Lecture2,3, we studied the powering adj-mat counts the number of nodes that are neighbors at level 2

- **Use cases:** Bipartite graphs
  - Author-to-papers (they authored)
  - 2-hop virtual edges make an author-author collaboration graph



- **Effect**:
  For example, if A sending messages directly → paper → B, they will be able to directly exchange messages
  ⇒ # layers of gnn ↓ → training faster ↑

How do we Augment sparse graphs?

1. Add virtual nodes

- The virtual node will connect to all/some subset of the nodes in the graph

- **Example**

  Suppose that two nodes are ten hops apart in the super sparse graph

  ⌊ where one node sending a message to the another requires 10 layer gnn

  Resolution?

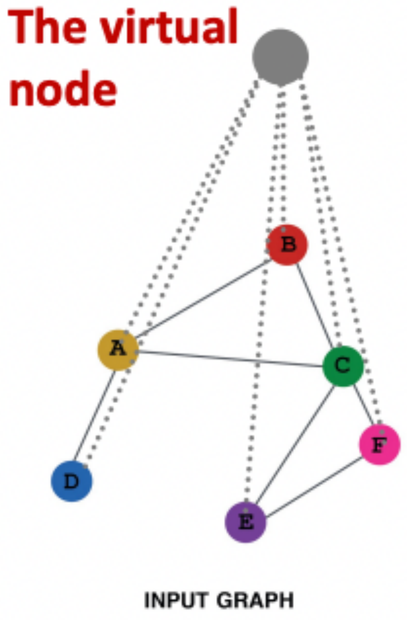
  ⌊ Create the virtual node and then connect to several nodes. (See the picture)

  ⇒ smaller distance btn nodes

  ⇒ to be able to communicate with each other much more efficiently

  ⇒ mp will be faster

  ⇒ We do not need the deeper layer gnn



How do we sample nodes from the dense graphs?

1. Node Neighborhood Sampling

- **idea:** Randomly sample a nodes' neighborhood for mp

- **Example**

Image the **C.Ronaldo** account in the instagram. He has a million of followers.
If we need to aggregate messages from million of his followers, it can be **expensive!**

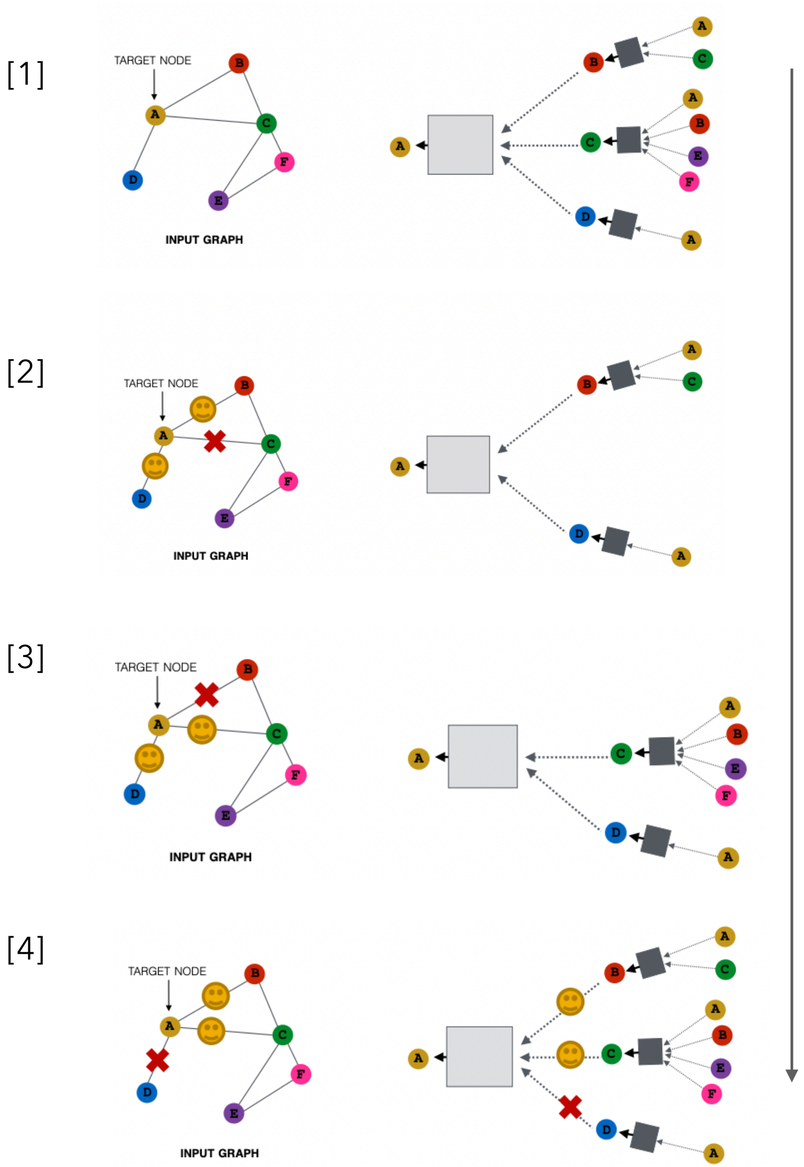
In [1], it is an original dense graph. It's too expensive :(
In [2], ignore the one edge btn A & C → Only nodes B, D will pass messages to A
└ ∃ trade-off

- Good: The computational graph is smaller
- Bad: C might have important information. (The degree of the C node is 4)

In [3], ignore the one edge btn A & B → Only nodes C, D will pass messages to A
└ For each layer or epoch of training, we can sample ignoring nodes differently

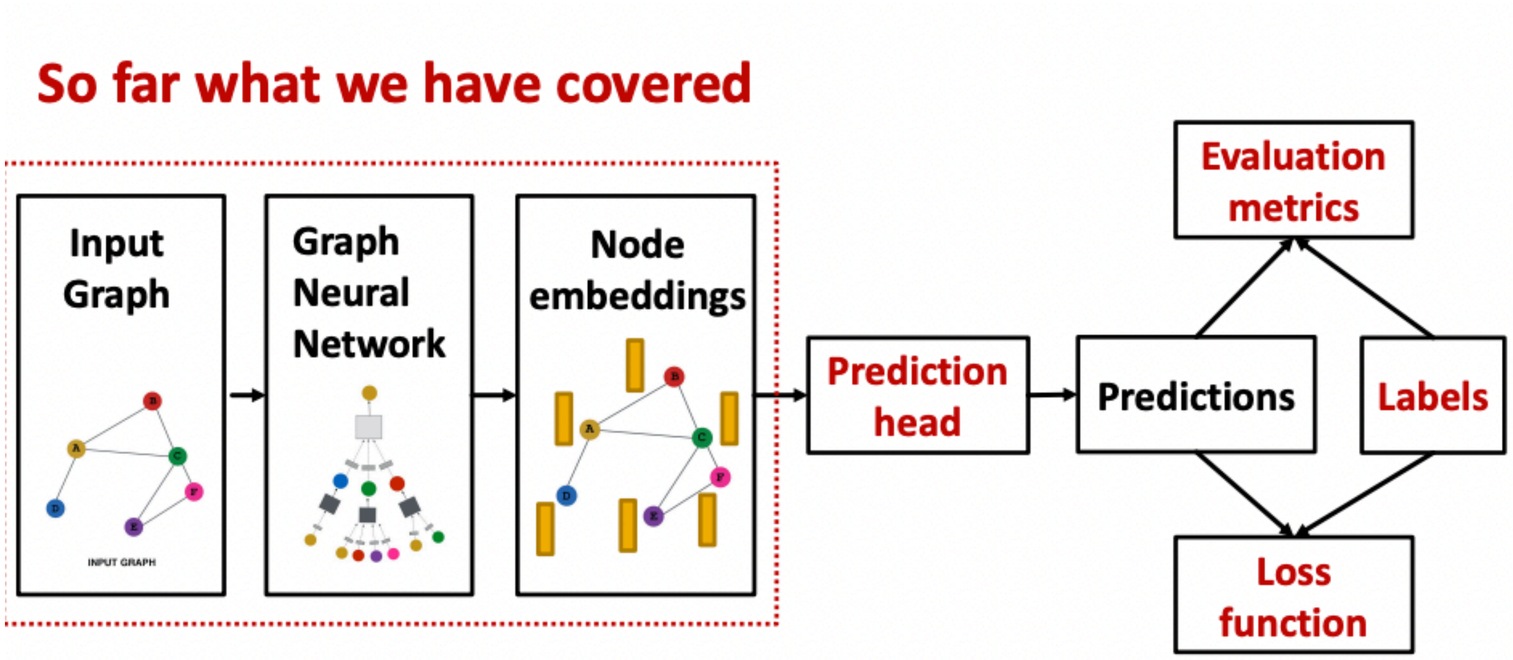
In [4], sample nodes similar to the case where all the neighbors are used
└ Sample only important nodes

⇒ The computational graph is smaller
⇒ It allows us to scale GNN to masive graphs (In practice, to scale up gnn is a good approach)



2. Training Graph Neural Network

Q. How do you get from node embeddings to the actual prediction?
Q. How do you evaluate them based against some ground truth labels?
Q. How do you compute the losses?



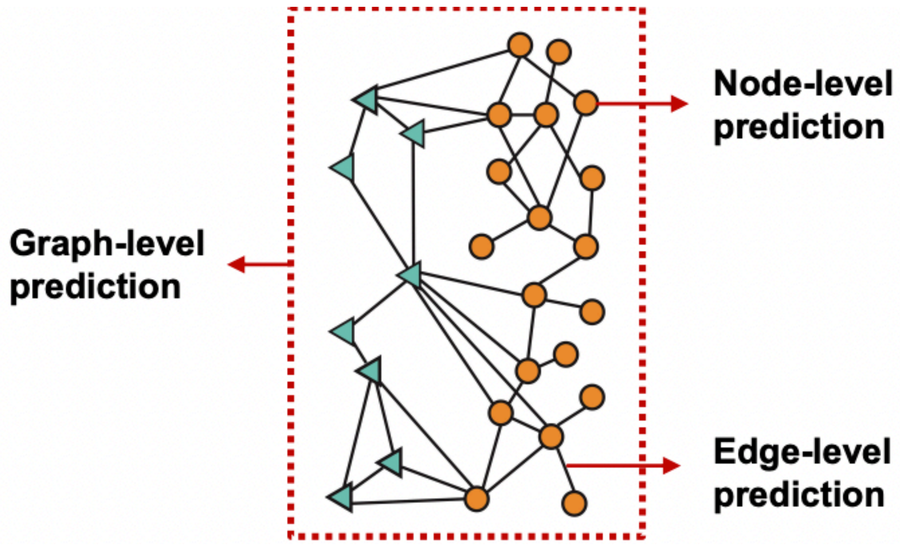
Output of a GNN: set of node embeddings

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

(1) Different prediction heads

: Node-level, Edge-level, Graph-level

⇒ It implies that different task levels require different prediciton head



(1-1) Node-level

Want to make k-way prediction.

⌊Classification: k categories, Regression, regress on k targets.

Given that d-dim embeddings: $\{h_v^{(L)} \in R^d, \forall v \in G\}$,

$$\hat{y}_v = Head_{node}(v_v^{(L)}) = W^{(H)} h_v^{(L)}$$

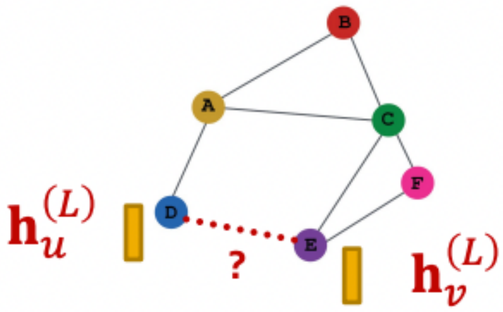
⌊$W^H \in R^{k*d}$: mapping $h_v \in R^d$ to $\hat{y}_v \in R^k$

(1-2) Edge-level

Make prediction using **pairs of node embeddings**

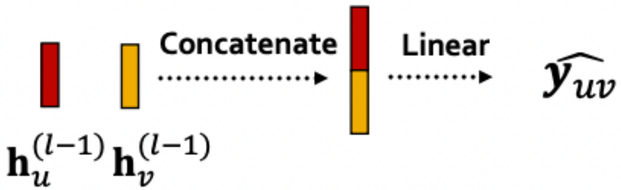
* Want to make k-way prediction

$$\hat{y_{uv}} = HEAD_{edge}(h_u^{(L)}, h_v^{(L)})$$



Q. What are the options for HEAD_edge ?

[Option1] Concat + Linear



[Option2] Dot product

$$\hat{\mathbf{y}}_{uv} = (\mathbf{h}_u^{(L)})^T \mathbf{h}_v^{(L)}$$

⇒ Single scalar output, which is the One-way prediction

Binary: Link/Not

Applying to k-way prediction, use different trainable weights (W1, … Wk)

It's like the multi-head attention

$$\hat{y}_{uv}^{(1)} = (\mathbf{h}_u^{(L)})^T \mathbf{W}^{(1)} \mathbf{h}_v^{(L)}$$

$$\cdots$$

$$\hat{y}_{uv}^{(k)} = (\mathbf{h}_u^{(L)})^T \mathbf{W}^{(k)} \mathbf{h}_v^{(L)}$$

$$\hat{\mathbf{y}}_{uv} = \mathrm{Concat}(\hat{y}_{uv}^{(1)}, \ldots, \hat{y}_{uv}^{(k)}) \in \mathbb{R}^k$$

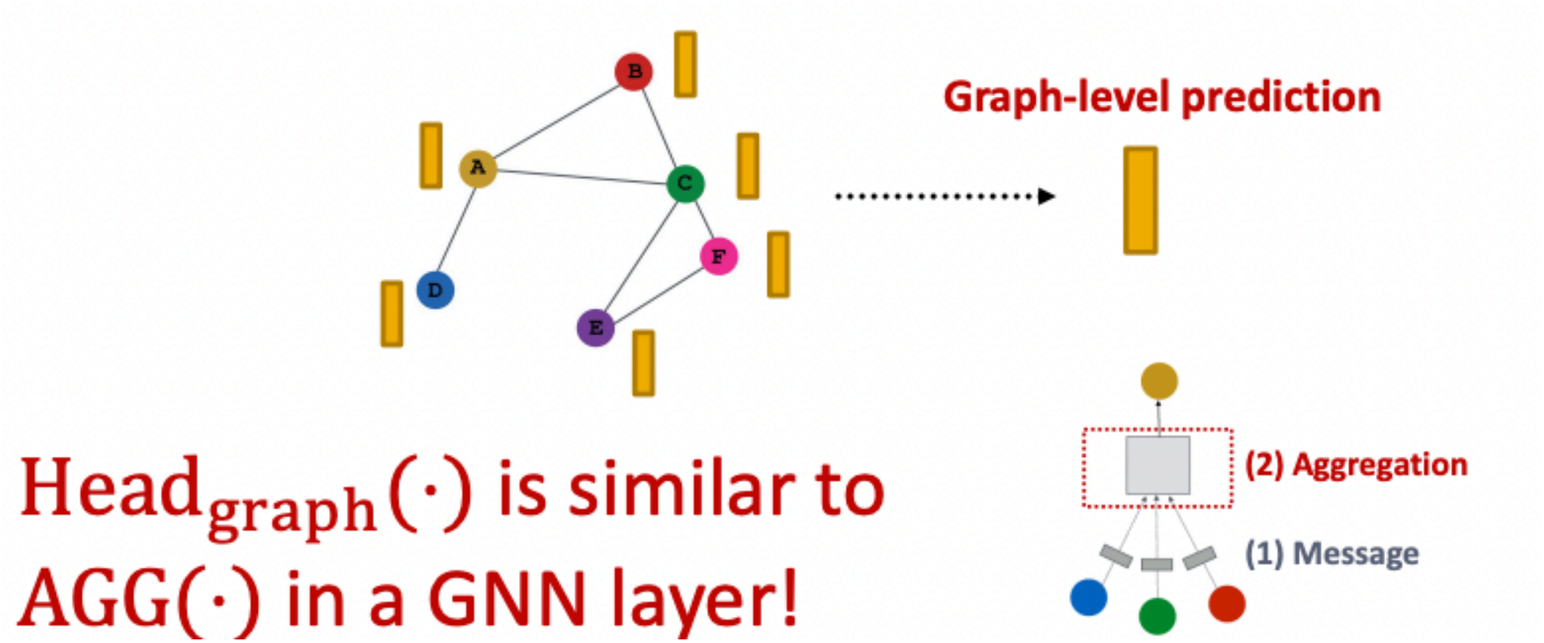
(1-3) Graph-level

Make prediction using **the node embeddings in our graph**

* Want to make k-way prediction

$$\hat{y_{uv}} = HEAD_{graph}(h_v^{(L)} \in R^d, \forall v \in G)$$

└ It means that we have to take the **individual node embeddings**



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

(1-3) Graph-level: Many options to make a head

(1) Global mean pooling

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

(2) Global max pooling

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

(3) Global sum pooling

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

These options work great for small graphs

(1-3-1) Issue of Global Pooling

Global Pooling over a (large) graph will lose information

Node embeddings for G_1: $\{-1, -2, 0, 1, 2\}$

Node embeddings for G_2: $\{-10, -20, 0, 10, 20\}$

Prediction for G_1: $\hat{y}_G = \text{Sum}(\{-1, -2, 0, 1, 2\}) = 0$

Prediction for G_2: $\hat{y}_G = \text{Sum}(\{-10, -20, 0, 10, 20\}) = 0$

Although the structures of G1,G2 are clearly different, the prediction is the same.

⇒ The model cannot differentiate G1, G2

(1-3-2) Hierarchical Global Pooling

- **Toy example:** We will aggregate via $\mathrm{ReLU}(\mathrm{Sum}(\cdot))$
  - We first separately aggregate the first 2 nodes and last 3 nodes
  - Then we aggregate again to make the final prediction
- G_1 node embeddings: $\{-1, -2, 0, 1, 2\}$
  - **Round 1:** $\hat{y}_a = \mathrm{ReLU}(\mathrm{Sum}(\{-1, -2\})) = 0$, $\hat{y}_b = \mathrm{ReLU}(\mathrm{Sum}(\{0,1,2\})) = 3$
  - **Round 2:** $\hat{y}_G = \mathrm{ReLU}(\mathrm{Sum}(\{y_a, y_b\})) = \mathbf{3}$
- G_2 node embeddings: $\{-10, -20, 0, 10, 20\}$
  - **Round 1:** $\hat{y}_a = \mathrm{ReLU}(\mathrm{Sum}(\{-10, -20\})) = 0$, $\hat{y}_b = \mathrm{ReLU}(\mathrm{Sum}(\{0,10,20\})) = 30$
  - **Round 2:** $\hat{y}_G = \mathrm{ReLU}(\mathrm{Sum}(\{y_a, y_b\})) = \mathbf{30}$

Now we can differentiate G_1 and G_2 !

(2) Supervised Vs Unsupervised

Supervised: The labels come from some external sources

* Notation

Node label y_v, Edge label y_{uv}, Graph label y_G

(3) Classification Vs Regression

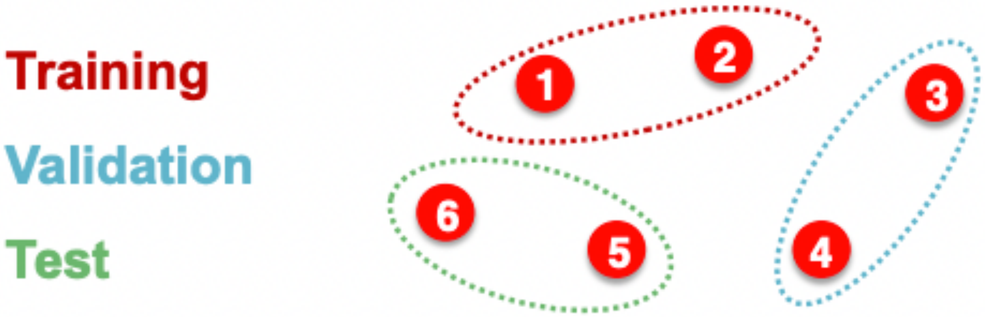
(4) Evaluation Metrics

3. Setting up GNN Prediction Tasks

(1) Why splitting Graphs is special?

Document , Image dataset → Assume that data points are **independent** from each other

Example: Image 3 will not affect our prediction on image 1



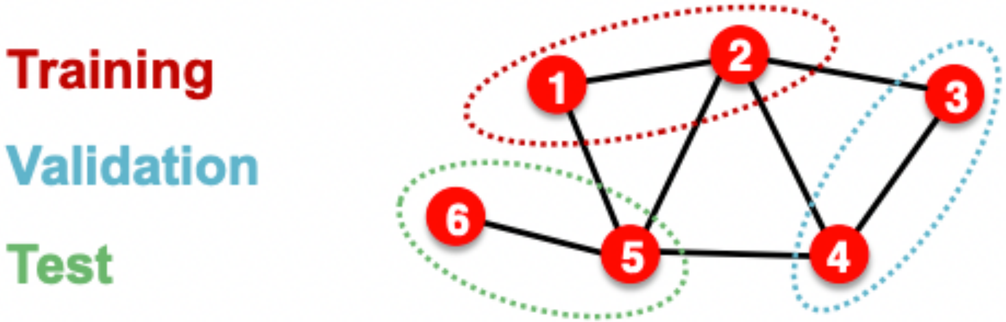
Node classification: Each data point is a node

Example

Node 5 **will affect our prediction** on node 1

└ Node 5 participate in **messagePassing** to node 1 → affect node 1

⇒ **Problematic**

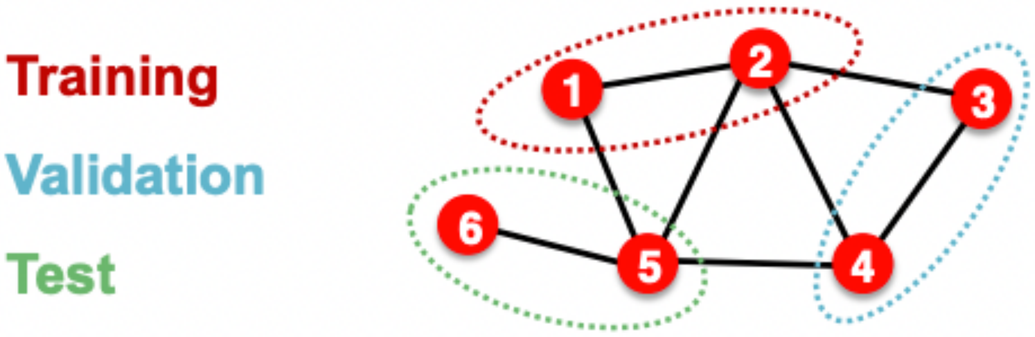


3. Setting up GNN Prediction Tasks

(2) Solution1 (Trasductive setting)

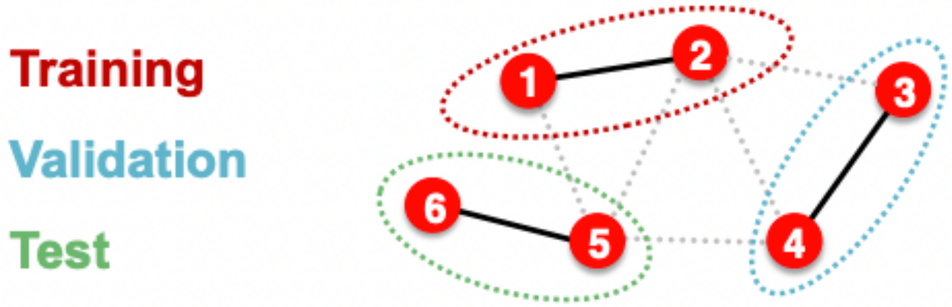
* **Split**: features | labels
* In training, compute embeddings **using entire graphs**
  * Train model using node 1&2's labels

* In validation, compute embeddings using also entire graphs
  * Evaluate on node 3&4's labels



(3) Solution2 (Inductive setting)

* **Break the edges** btn splits to get multiple graphs
* In training, compute embeddings **using the graph over node 1&2**
  * Train model using node 1&2's labels
* In validation, compute embeddings using the graph over node **3&4**
  * Evaluate on node 3&4's labels



3. Setting up GNN Prediction Tasks

(4) Settings

	Transductive Setting	**Inductive Setting**
Dataset	Consist of one graph	Consist of multiple graphs
Method	1. Use the entire graph at the training time 2. Only split the labels	1. Use a part is splitted in the entire graph at the training time 2. It allows us to test how can we generalize to unseen graph
Application	Node/Edge prediction	Node/Edge/Graph tasks