
IT5429E-1-24 (24.1A01)(Fall 2024): Graph Analytics for Big Data

Week 4: Graph Neural Nets (Part 2)

Instructor: Thanh H. Nguyen

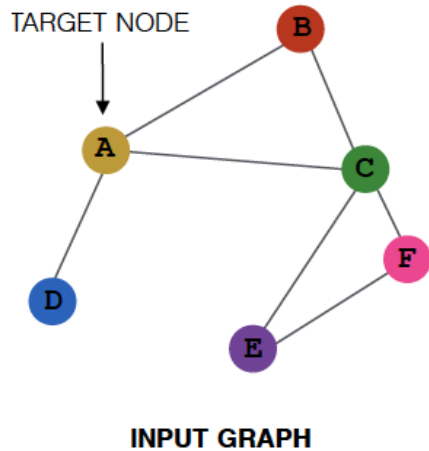
Many slides are adapted from <https://web.stanford.edu/class/cs224w/>



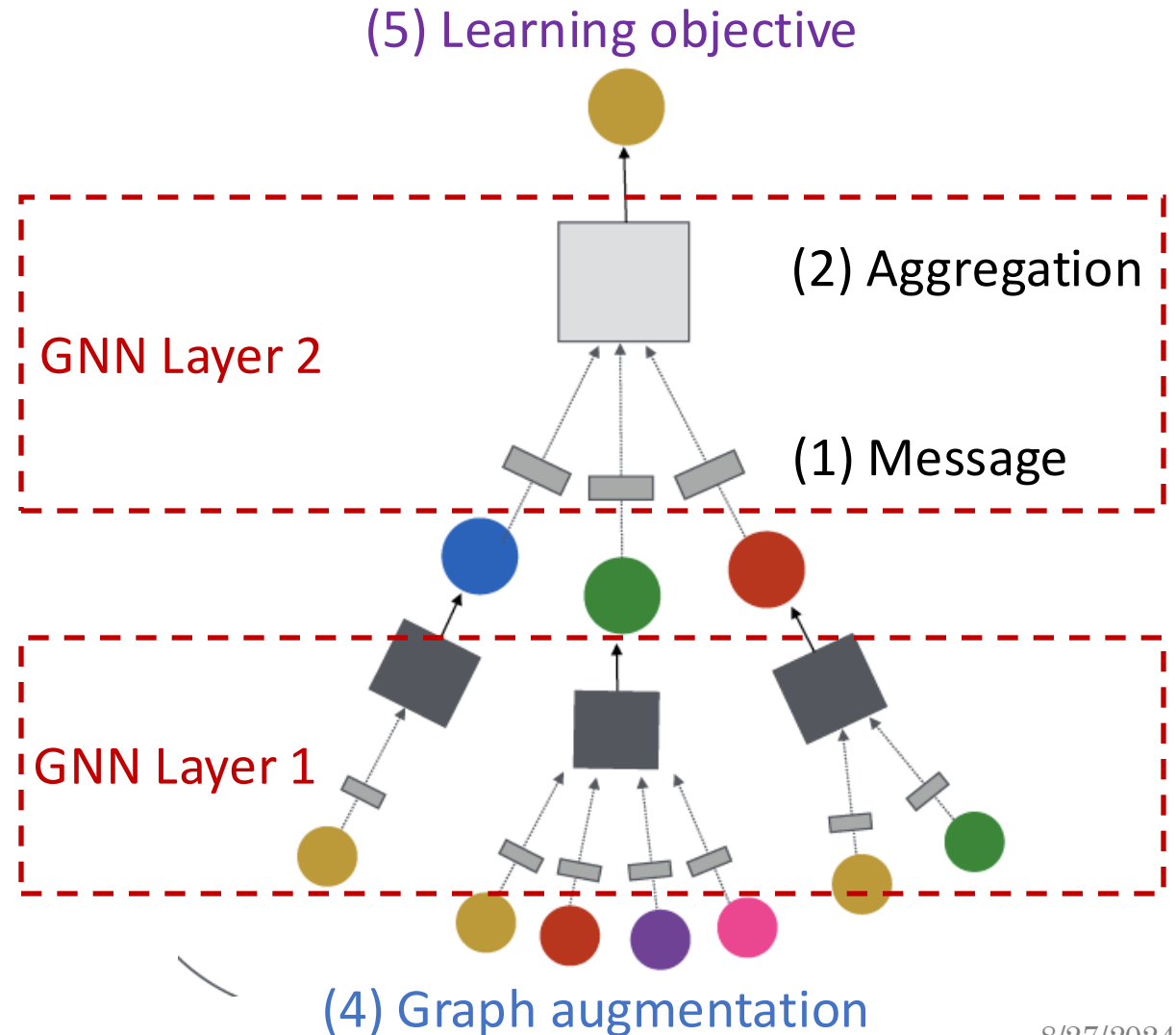
Reminder

- Project proposal
 - Deadline: The end of Week 4 (this week)

Recap: A General GNN Framework



(3) Layer connectivity



Recap: A Single GNN Layer

- Putting things together

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

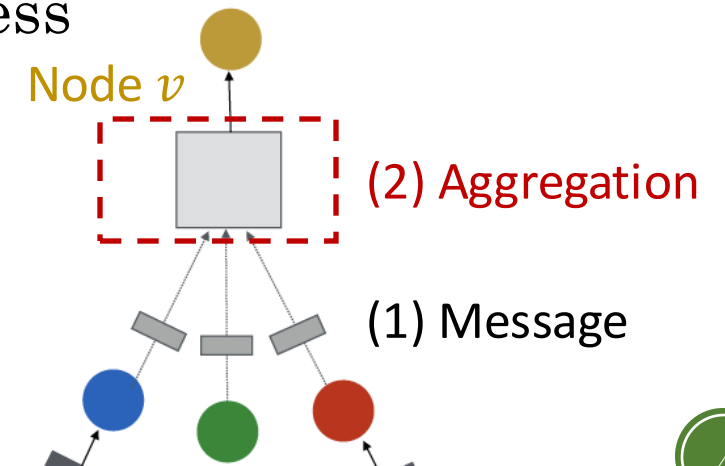
$$m_u^{(l)} = MSG^{(l)}(h_u^{(l-1)}), \forall u \in N(v) \cup \{v\}$$

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

$$h_v^{(l)} = AGG^{(l)}(\{m_u^{(l)}, u \in N(v)\}, m_v^{(l)})$$

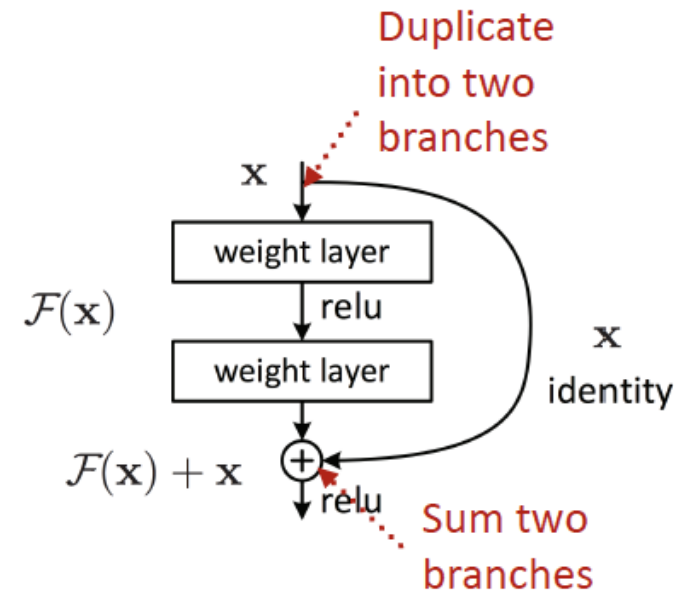
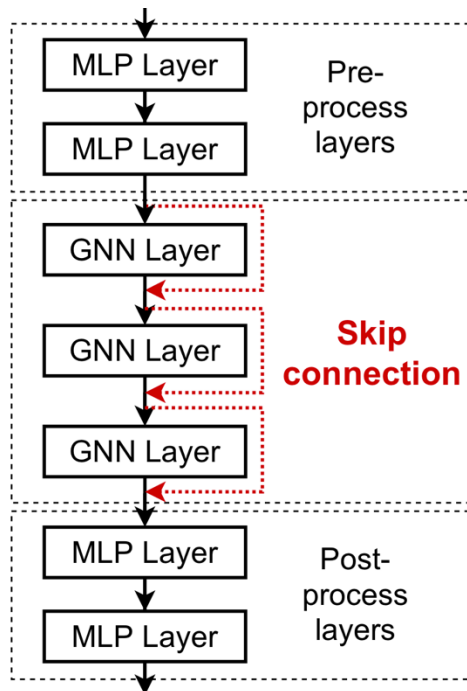
- **Non-linearity** (activation function): Add expressiveness

- Often write as $\sigma(\cdot)$. Example: ReLU(), Sigmoid(), etc
- Can be added to message or aggregation



Recap: 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

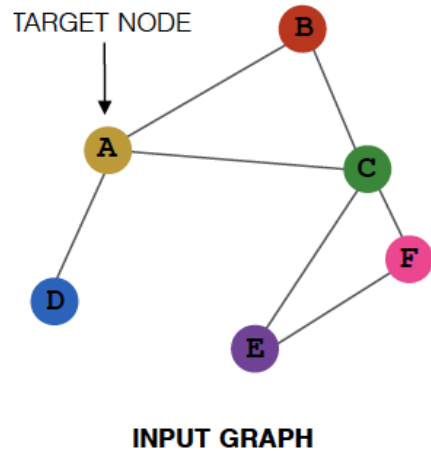


Recap: Graph Manipulation

- Graph Feature manipulation
 - The input graph lacks features → feature augmentation
- 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

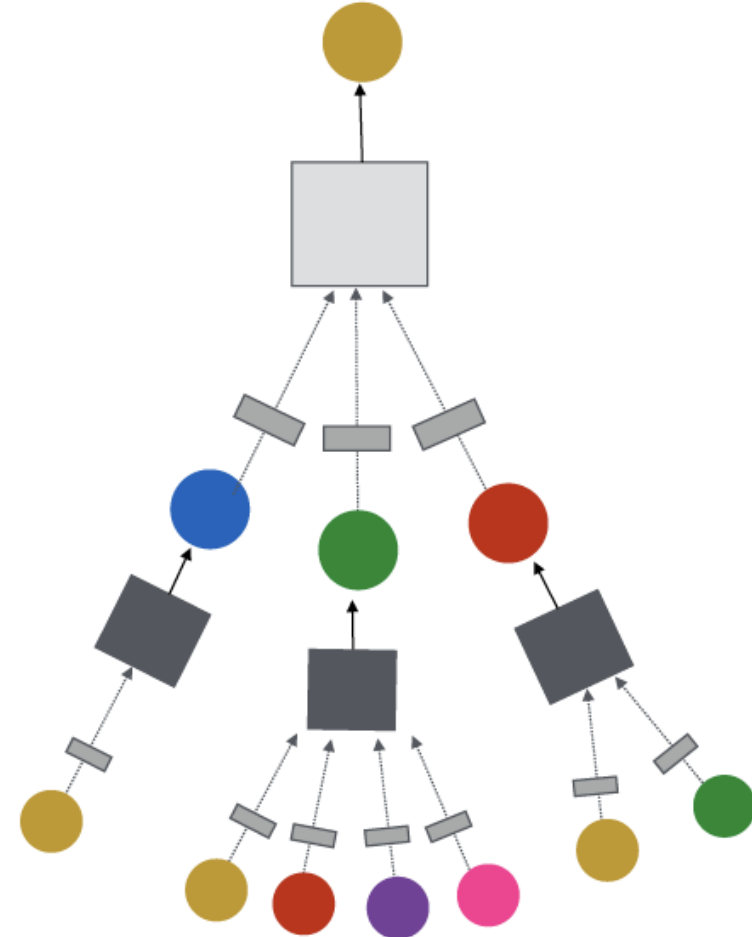
Prediction with GNNs

A General GNN Framework



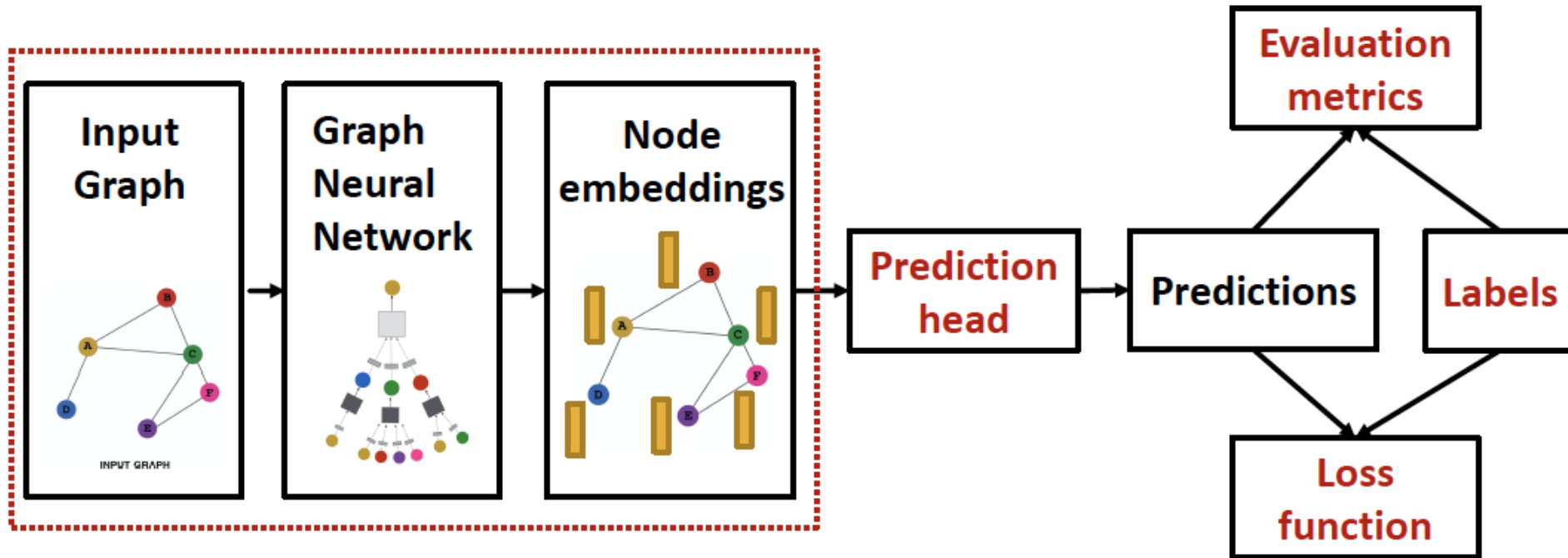
Next: How to train a GNN?

(5) Learning objective



GNN Training Pipeline

What we have covered so far

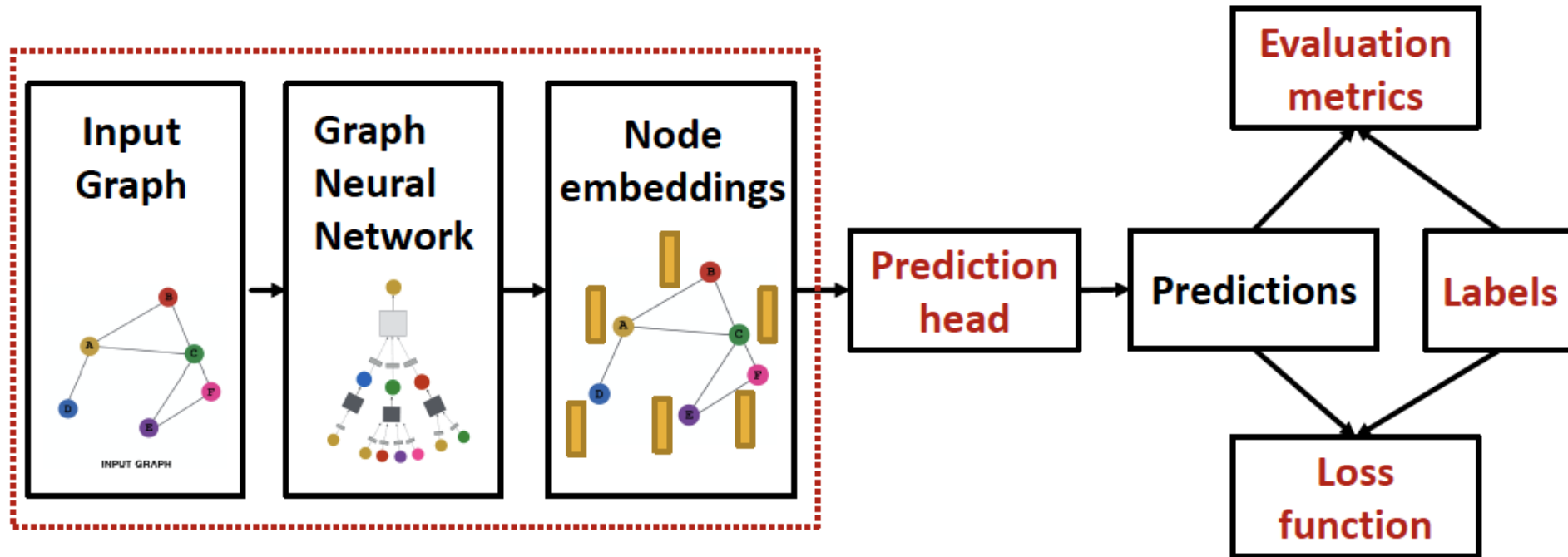


Output of a GNN: set of node embeddings

$$\{h_v^{(L)}, \forall v \in G\}$$

GNN Training Pipeline (1)

What we have covered so far



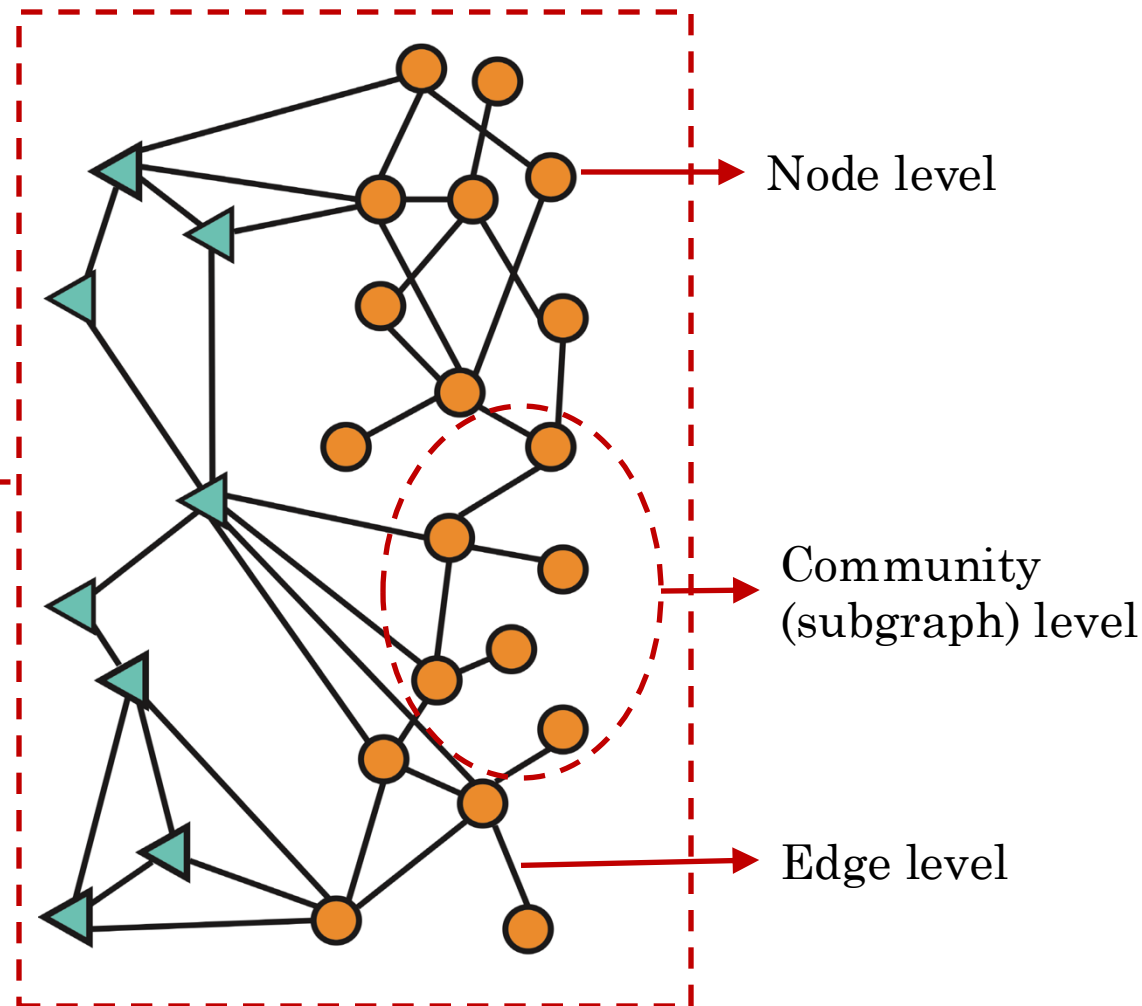
(1) Different prediction heads:

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

Different Types of Tasks

Idea: Different task levels require different prediction heads

Graph level prediction,
Graph generation

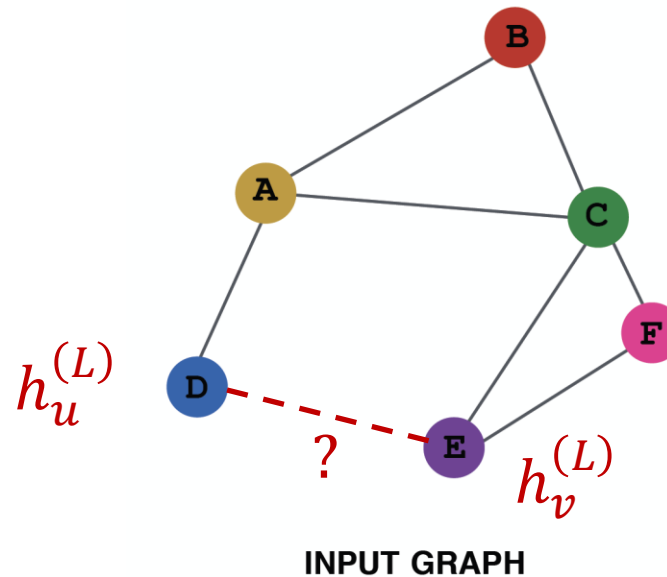


Prediction Heads: Node Level

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

Prediction Heads: Edge Level

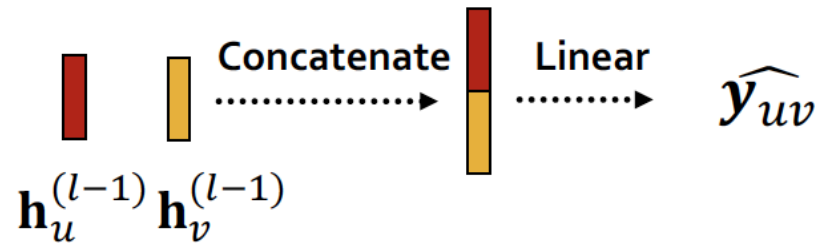
- **Edge level prediction:** Make prediction using pairs of node embeddings
- Suppose we want to make k -way prediction
- $\hat{y}_{uv} = Head_{edge} \left(h_u^{(L)}, h_v^{(L)} \right)$



- What are the options for $Head_{edge} \left(h_u^{(L)}, h_v^{(L)} \right)$?

Prediction Heads: Edge Level

- Options for $Head_{edge}(h_u^{(L)}, h_v^{(L)})$:
- (1) Concatenation + Linear:
 - We have seen this in graph attention



- $\hat{y}_{uv} = Linear\left(Concat\left(h_u^{(L)}, h_v^{(L)}\right)\right)$
- Linear(\cdot) will map $2d$ -dimensional embeddings (since we concatenated embeddings) to k -dim embeddings (k -way prediction)

Prediction Heads: Edge Level

- Options for $Head_{edge}(h_u^{(L)}, h_v^{(L)})$:
- (2) Dot product:**
 - $\hat{y}_{uv} = (h_u^{(L)})^T \cdot h_v^{(L)}$
 - 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: $W^{(1)}, W^{(2)}, \dots, W^{(k)}$ trainable

$$\hat{y}_{uv}^{(1)} = (h_u^{(L)})^T W^{(1)} h_v^{(L)}$$

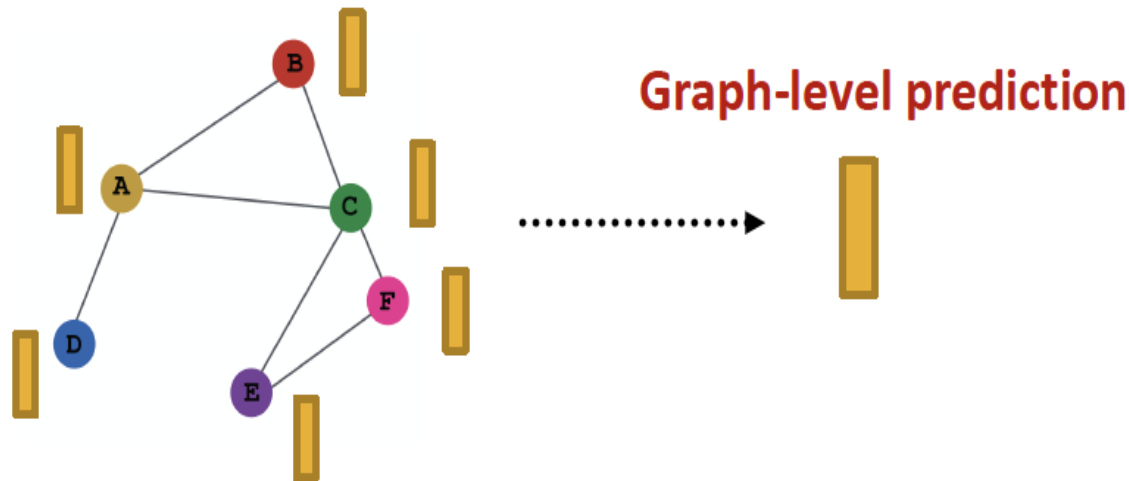
...

$$\hat{y}_{uv}^{(k)} = (h_u^{(L)})^T W^{(k)} h_v^{(L)}$$

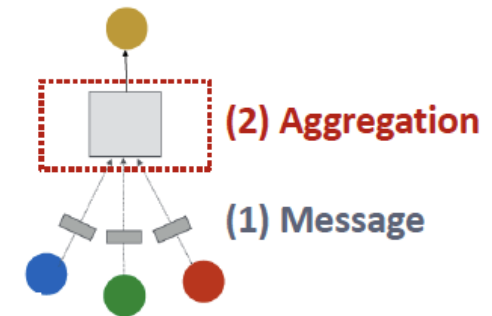
$$\hat{y}_{uv} = \text{Concat}(\hat{y}_{uv}^{(1)}, \dots, \hat{y}_{uv}^{(k)}) \in \mathbb{R}^k$$

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
- $\hat{y}_G = Head_{graph} \left(\{h_v^{(L)} \in \mathbb{R}^d, \forall v \in G\} \right)$



- $Head_{graph}(\cdot)$ is similar to AGG() in a GNN layer



Prediction Heads: Graph Level

- Options for $Head_{graph} \left(\{h_v^{(L)} \in \mathbb{R}^d, \forall v \in G\} \right)$:

- (1) Global mean pooling

$$\hat{y}_G = \text{Mean} \left(\{h_v^{(L)} \in \mathbb{R}^d, \forall v \in G\} \right)$$

- (2) Global max pooling

$$\hat{y}_G = \text{Max} \left(\{h_v^{(L)} \in \mathbb{R}^d, \forall v \in G\} \right)$$

- (3) Global sum pooling

$$\hat{y}_G = \text{Sum} \left(\{h_v^{(L)} \in \mathbb{R}^d, \forall v \in G\} \right)$$

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

Issue of Global Pooling

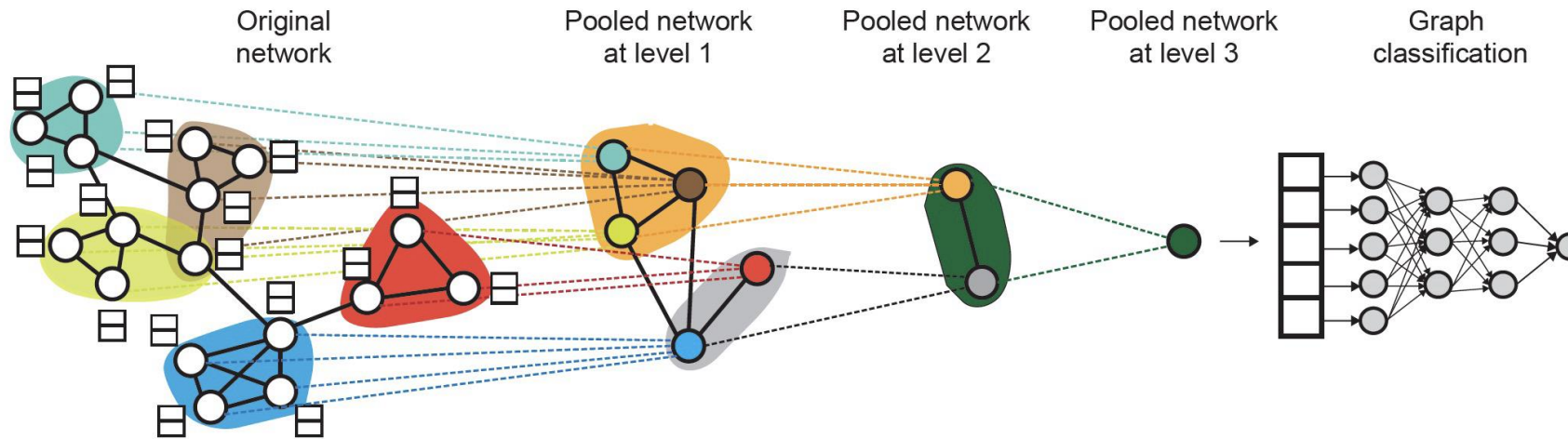
- **Issue:** Global pooling over a (large) graph will lose information
- **Toy example:** we use 1-dim node embeddings
 - Node embeddings for G_1 : $\{-1, -2, 0, 1, 2\}$
 - Node embeddings for G_2 : $\{-10, -20, 0, 10, 20\}$
 - Clearly G_1 and G_2 have very different node embeddings
 - Their structures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

Hierarchical Global Pooling

- A solution: Let's aggregate all the node embeddings *hierarchically*
 - **Toy example:** We will aggregate via $ReLU(\text{Sum}(\cdot))$
 - 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: $\hat{y}_a = ReLU(\text{Sum}(\{-1, -2\})) = 0$, $\hat{y}_b = ReLU(\text{Sum}(\{0, 1, 2\})) = 3$
 - Round 2: $\hat{y}_G = ReLU(\text{Sum}(\{\hat{y}_a, \hat{y}_b\})) = 3$
 - G_2 node embeddings: $\{-10, -20, 0, 10, 20\}$
 - Round 1: $\hat{y}_a = ReLU(\text{Sum}(\{-10, -20\})) = 0$, $\hat{y}_b = ReLU(\text{Sum}(\{0, 10, 20\})) = 30$
 - Round 2: $\hat{y}_G = ReLU(\text{Sum}(\{\hat{y}_a, \hat{y}_b\})) = 30$

Hierarchical Global Pooling in Practice

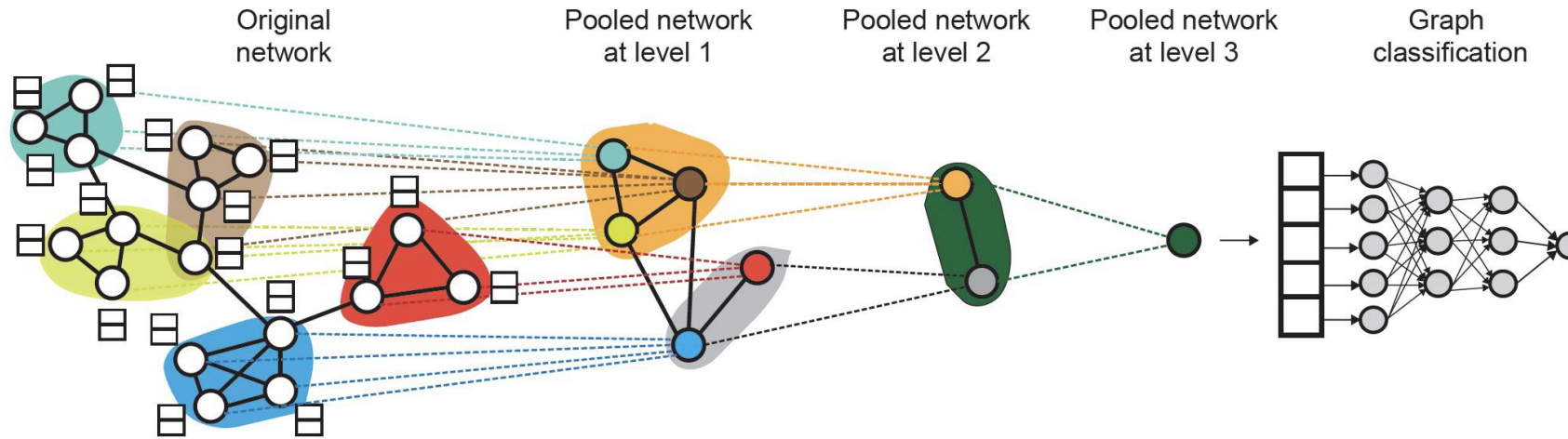
- DiffPool idea:
 - Hierarchically pool node embeddings



- Leverage 2 independent GNNs at each level
 - GNN A: Compute node embeddings
 - GNN B: Compute the cluster that a node belongs to
- GNNs A and B at each level can be executed in parallel

Hierarchical Global Pooling in Practice

- **DiffPool idea:**
 - Hierarchically pool node embeddings



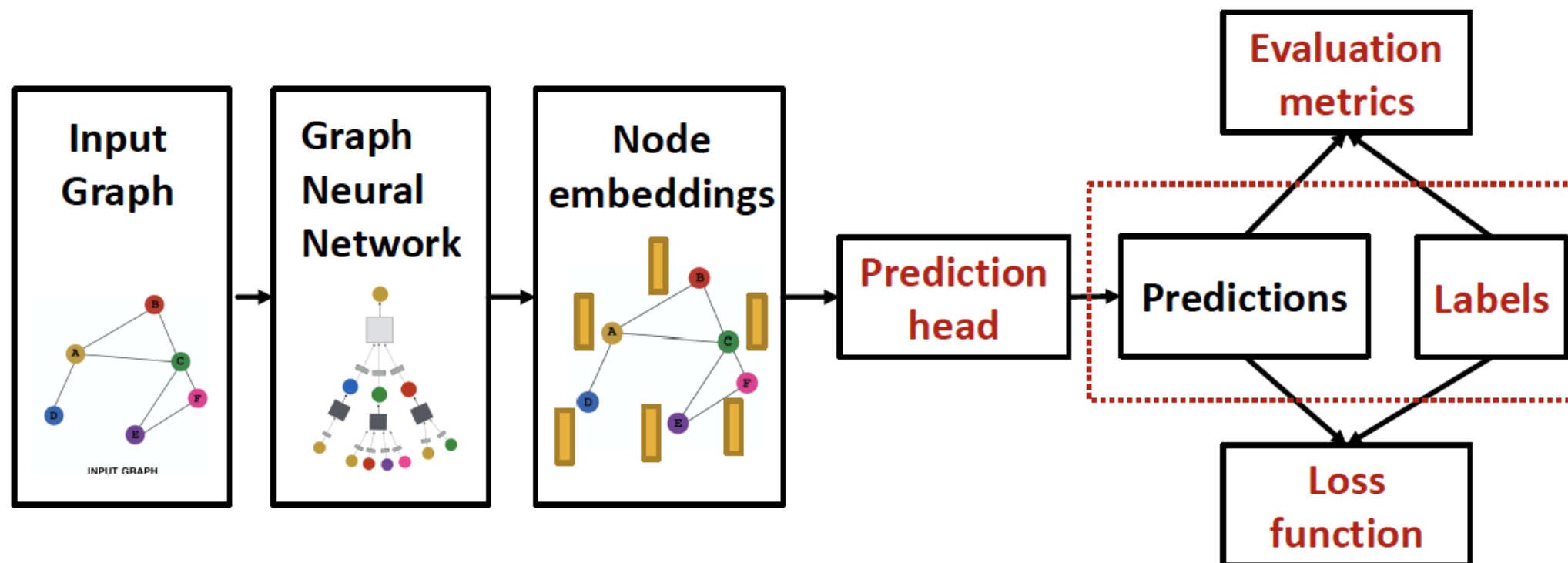
- **For each Pooling layer**
 - Use clustering assignments from GNN B to aggregate node embeddings generated by GNN A
 - Create a single new node for each cluster, maintaining edges between clusters to generated a new pooled network
- GNNs A and B at each level can be executed in parallel

Training Graph Neural Networks

GNN Training Pipeline (2)

(2) Where does ground-truth come from?

- Supervised labels
- Unsupervised signals



Supervised versus 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”

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 label 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

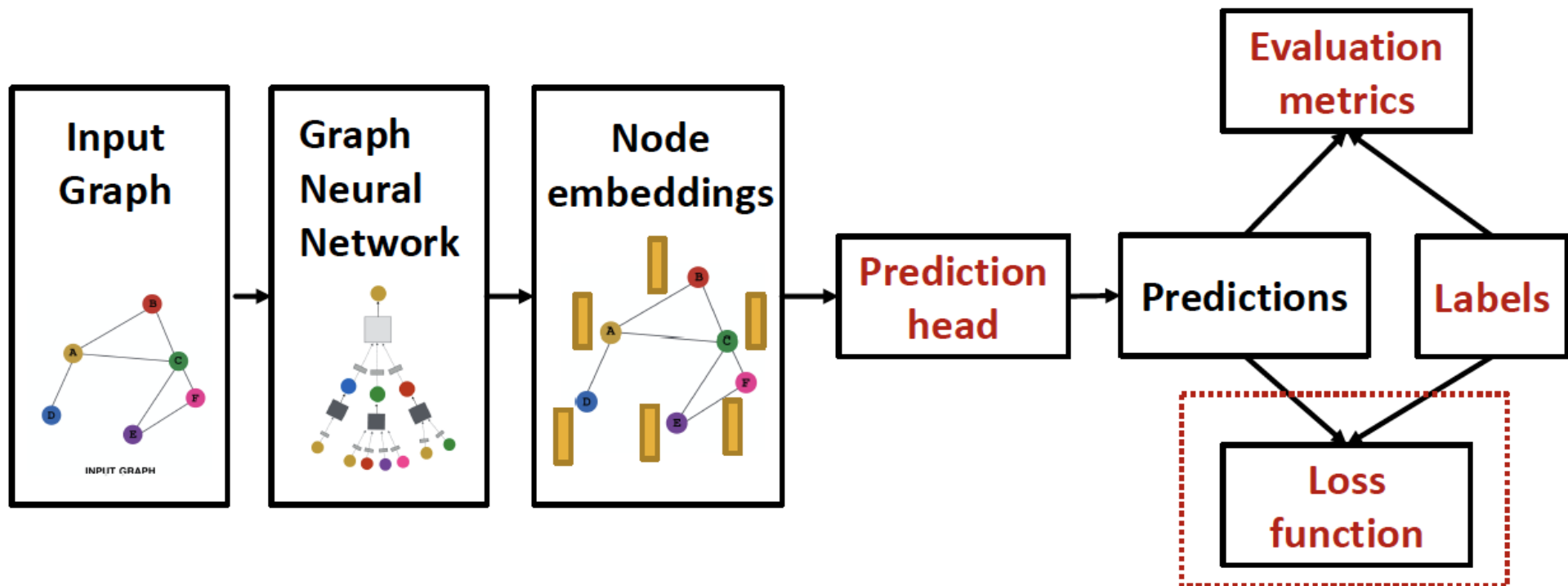
Unsupervised Signals on Graphs

- **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.
 - **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!

GNN Training Pipeline (3)

(3) How do we compute the final loss?

- Classification loss
- Regression loss



Settings for GNN Training

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

Classification or Regression

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

Classification Loss

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

$$CE(\underbrace{y^{(i)}}_{\text{label}}, \underbrace{\hat{y}^{(i)}}_{\text{prediction}}) = - \sum_{j=1}^K y_j^{(i)} \log(\underbrace{\hat{y}_j^{(i)}}_{\text{j-th class}})$$

i-th data point

- Where:

- $y^{(i)} \in \mathbb{R}^K$ = one-hot label encoding. E.g.,

0	0	1	0	0
---	---	---	---	---
- $\hat{y}^{(i)} \in \mathbb{R}^K$ = prediction after Softmax(). E.g.,

.1	.3	.4	.1	.1
----	----	----	----	----

- Total loss:

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

Regression Loss

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

$$CE(y^{(i)}, \hat{y}^{(i)}) = - \sum_{j=1}^K (y_j^{(i)} - \hat{y}_j^{(i)})^2$$

- Where:

- $y^{(i)} \in \mathbb{R}^K$ = real-valued vector of targets. E.g.,
- $\hat{y}^{(i)} \in \mathbb{R}^K$ = real-valued vector of predictions. E.g.,

1.4	2.3	1.0	0.5	0.6
0.9	2.8	2.0	0.3	0.8

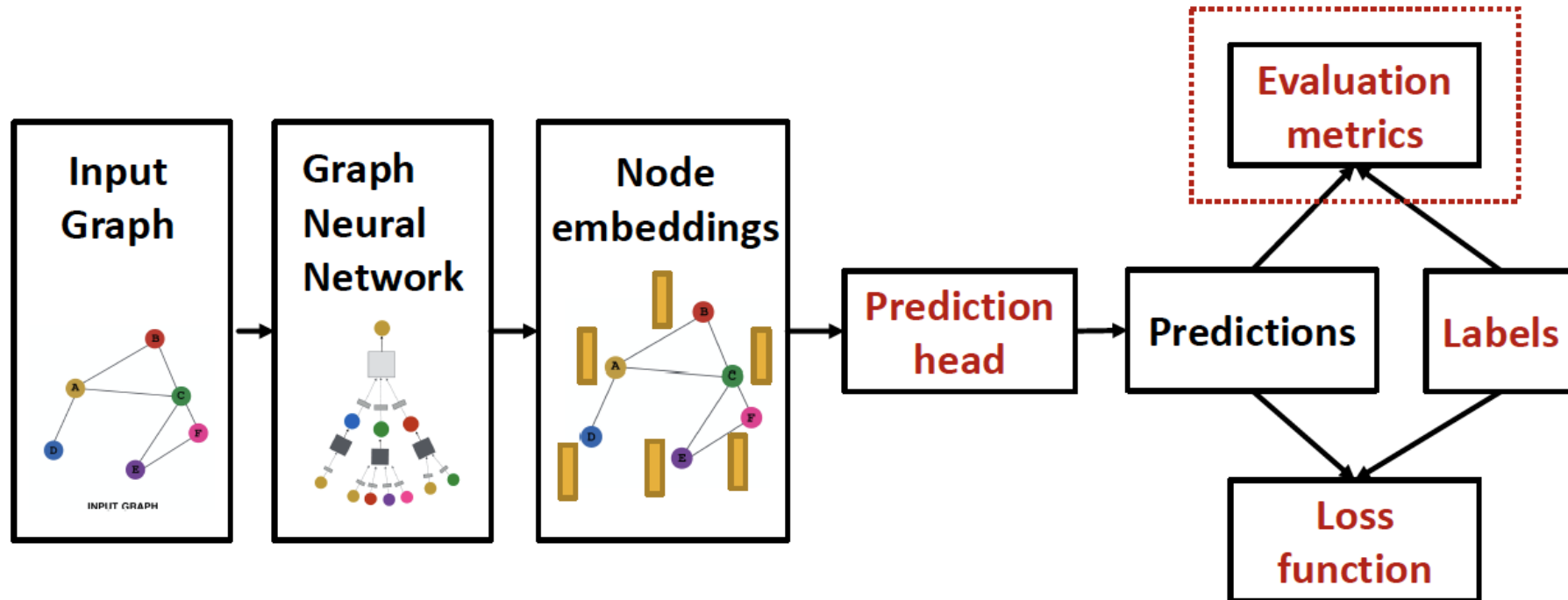
- Total loss:

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

GNN Training Pipeline (4)

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

- Accuracy
- ROC AUC



Evaluation Metrics: Regression

- We use standard evaluation metrics for GNN
 - 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)

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

- Mean absolute error (MAE)

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

Evaluation Metrics: Classification

- Evaluate classification tasks on graphs:
- (1) Multi-class classification
 - We simply report the **accuracy**:
$$\frac{1[\operatorname{argmax}(\hat{y}^{(i)}) = y^{(i)}]}{N}$$
- (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**

Metrics for Binary Classification

- Accuracy:

$$\frac{TP + TN}{TP + TN + FP + FN} = \frac{TP + TN}{|Dataset|}$$

- Precision (P)

$$\frac{TP}{TP + FP}$$

- Recall (R)

$$\frac{TP}{TP + FN}$$

- F1-Score

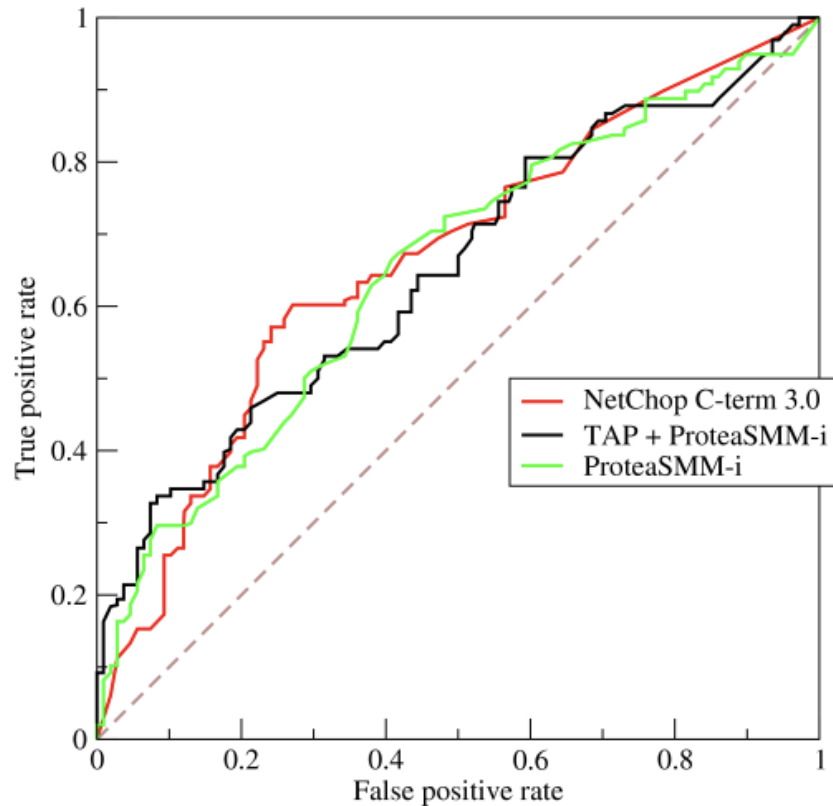
$$\frac{2P \times R}{P + R}$$

Confusion matrix

	Actually Positive (1)	Actually Negative (0)
Predicted Positive (1)	True Positives (TPs)	False Positives (FPs)
Predicted Negative (0)	False Negatives (FNs)	True Negatives (TNs)

Evaluation Metrics

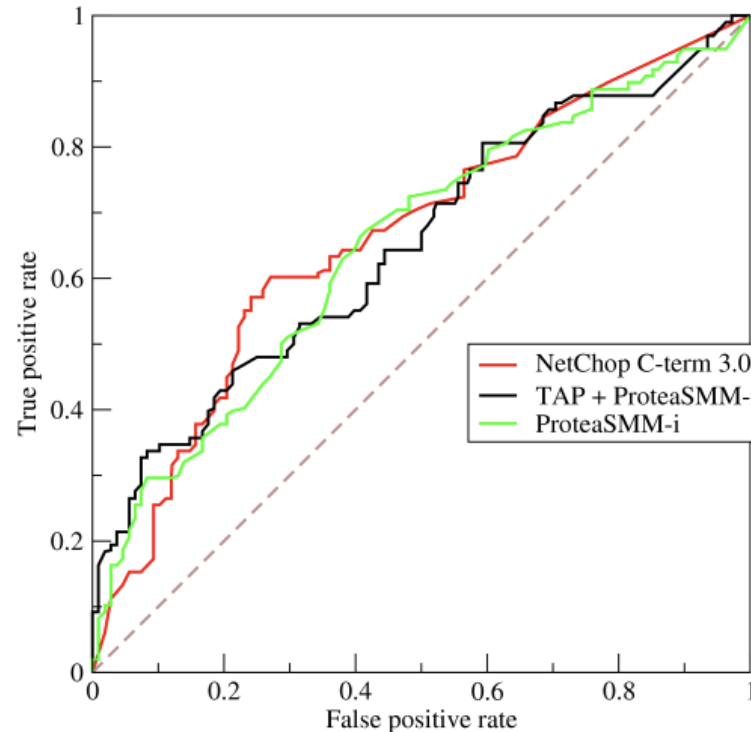
- ROC Curve: Captures the tradeoff in TPR and FPR as the classification threshold is varied for a binary classifier.



$$TPR = Recall = \frac{TP}{TP + FN}$$

$$FPR = \frac{FP}{FP + TN}$$

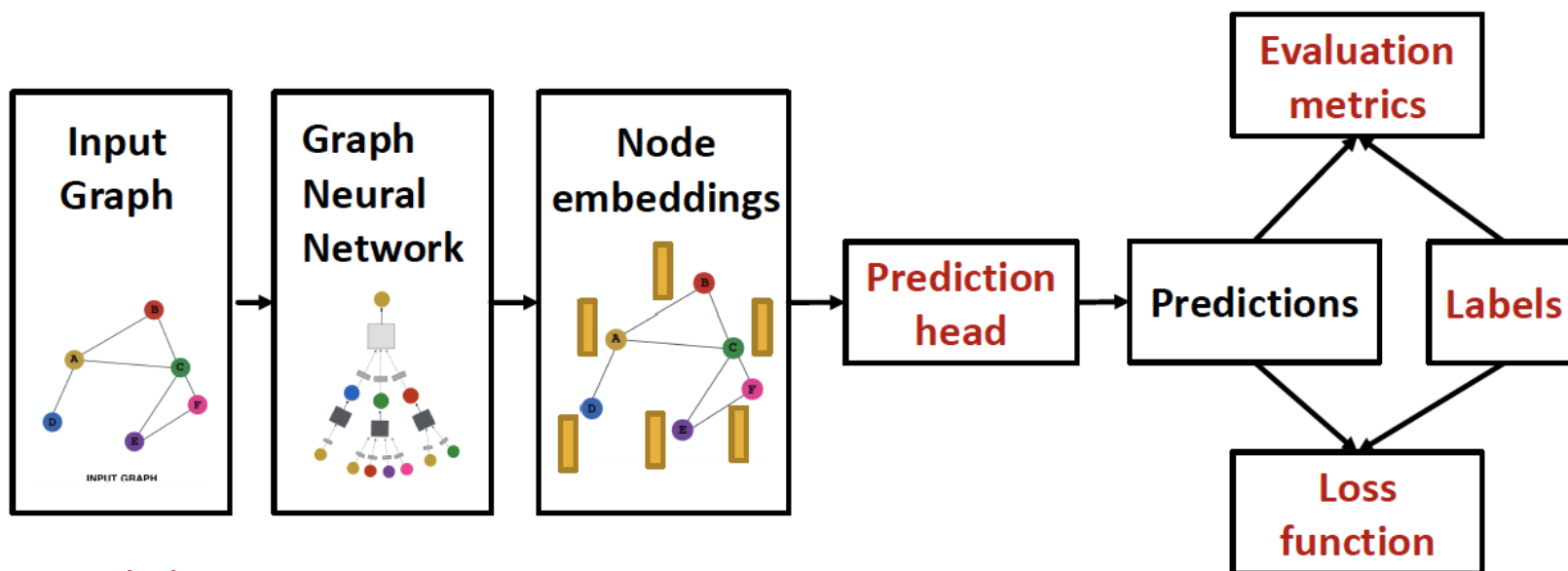
Evaluation Metrics



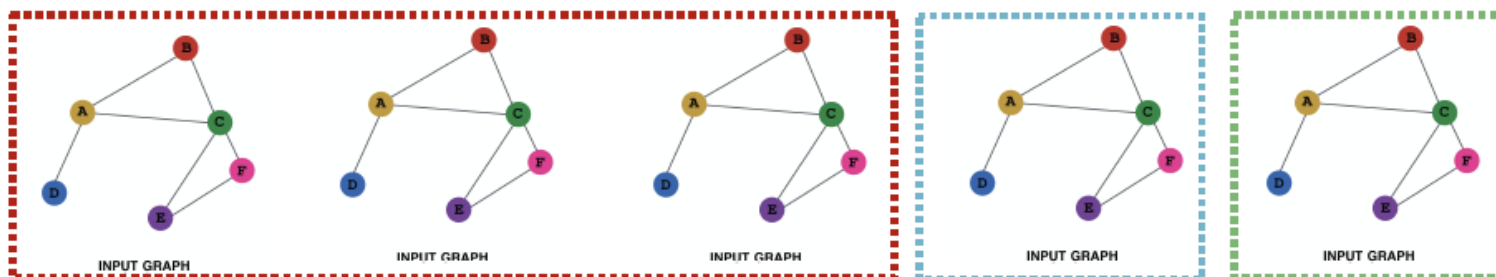
- ROC AUC: Area under the ROC Curve.
- **Intuition:** The probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one

Setting Up GNN Prediction Tasks

GNN Training Pipeline (5)



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



Dataset split

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**

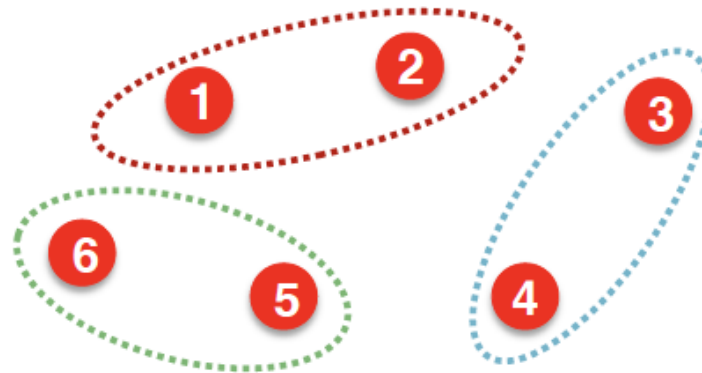
Why Splitting Graphs is Special

- 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

Training

Validation

Test



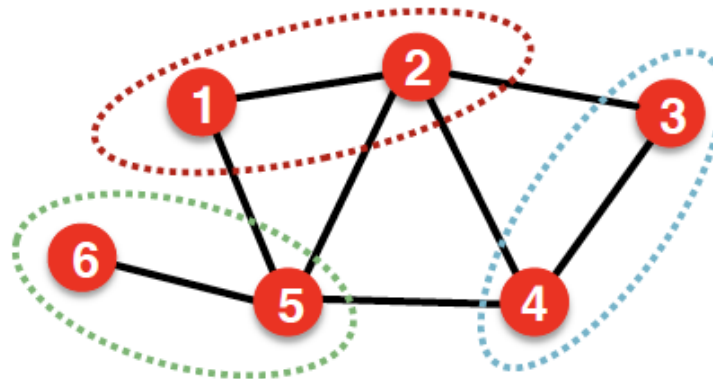
Why Splitting Graphs is Special

- 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

Training

Validation

Test



- What are our options?

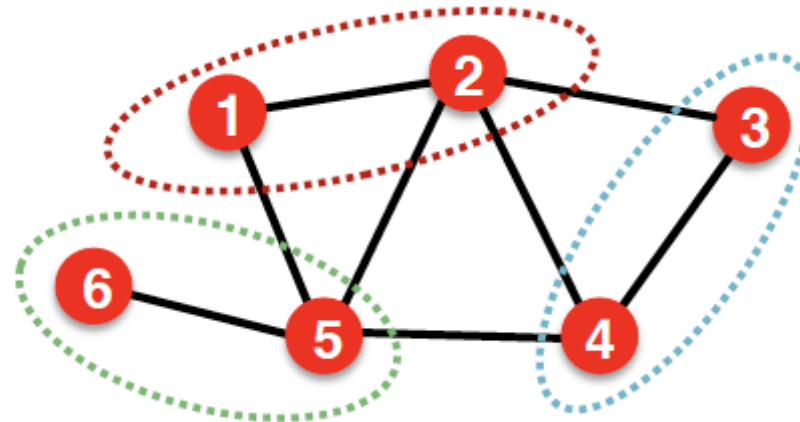
Why Splitting Graphs is Special

- **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



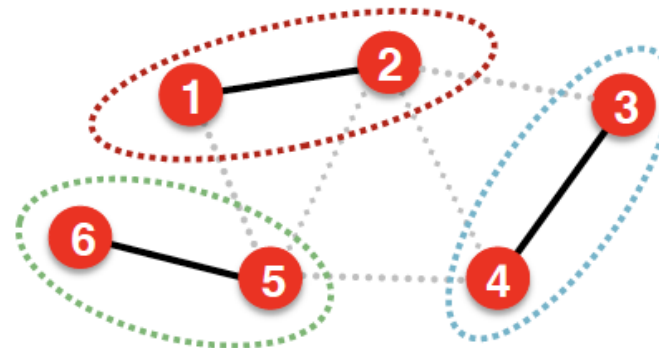
Why Splitting Graphs is Special

- **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 anymore.
 - 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



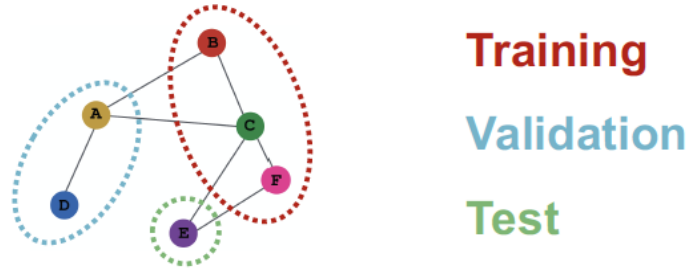
Transductive / Inductive Settings

- **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

Example: Node Classification

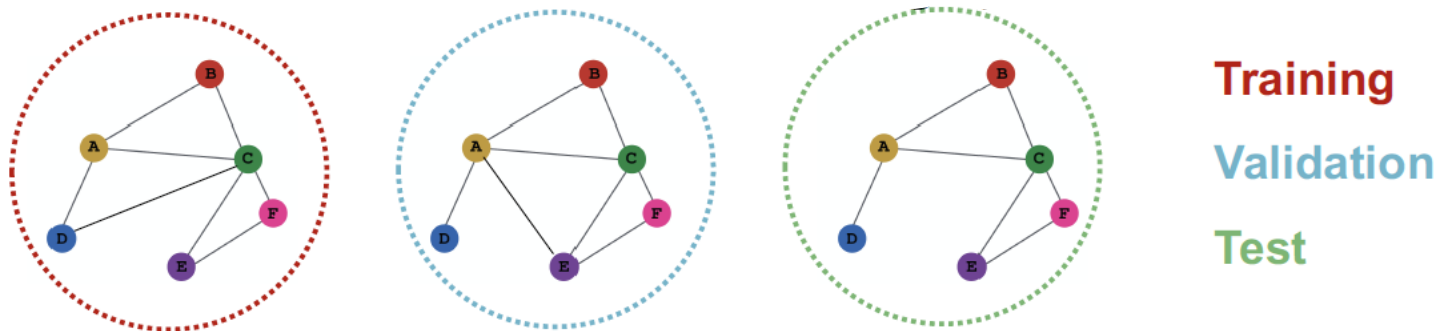
■ Transductive node classification

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



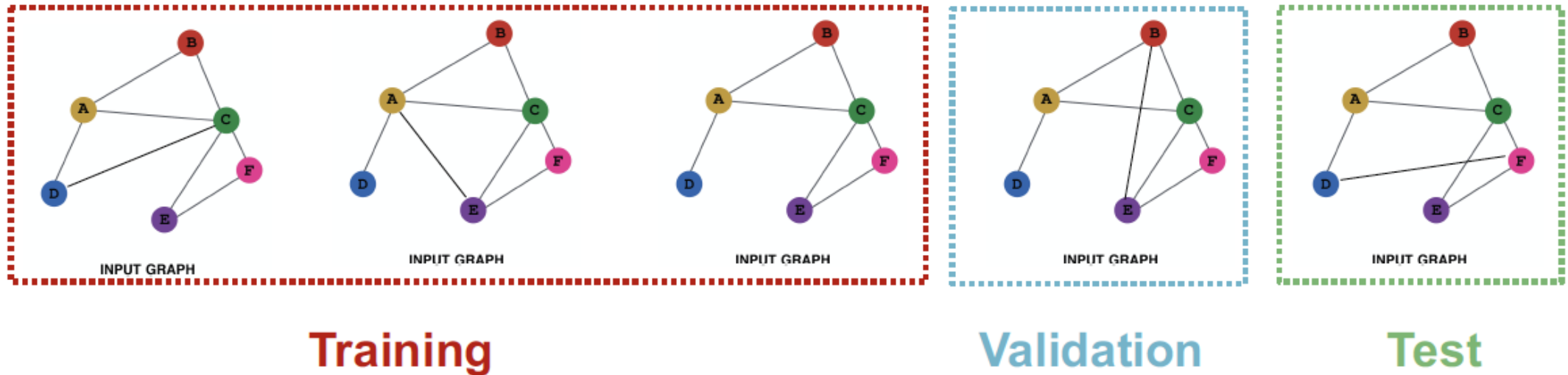
■ Inductive node classification

- Suppose we have a dataset of 3 graphs
- Each split contains an independent graph



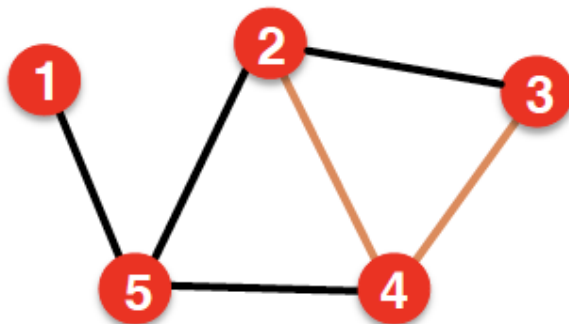
Example: Graph Classification

- 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).

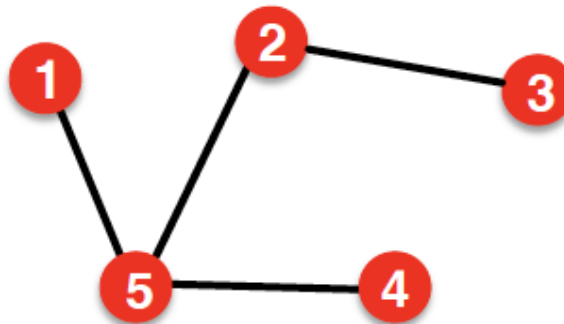


Example: Link Prediction

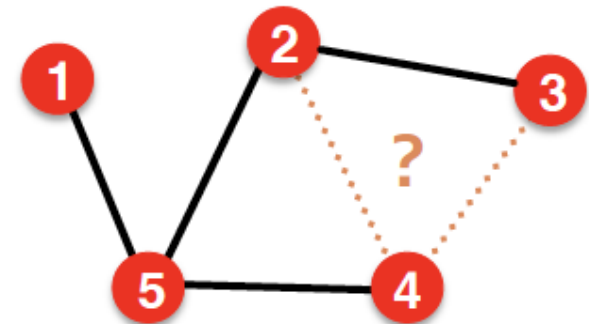
- Goal of link prediction: predict missing edges
- Setting up link prediction is tricky:
- Link prediction is an **unsupervised / self-supervised task**. We need to create the labels and dataset splits on our own
- Concretely, we need to hide some edges from the GNN and let the GNN predict if the edges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



Original graph

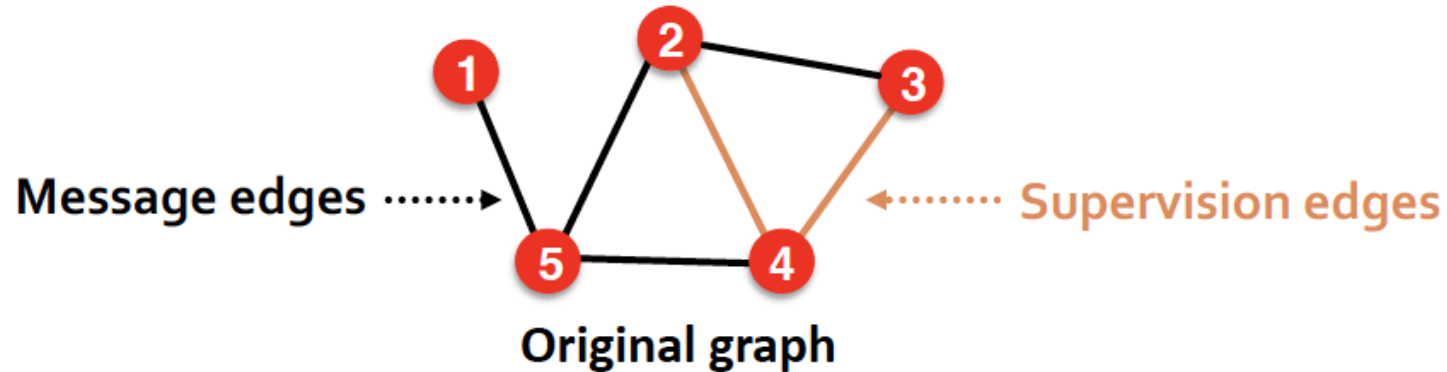


Input graph to GNN



Predictions made by GNN

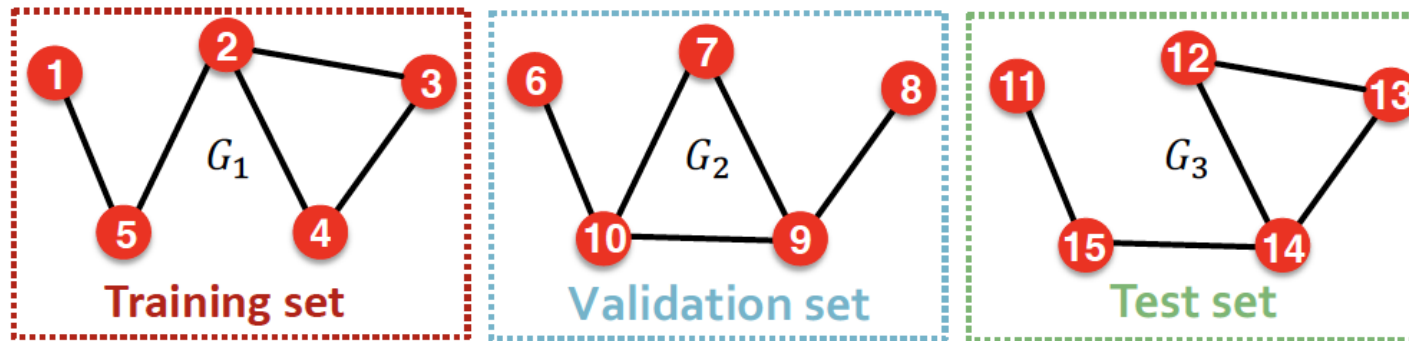
Setting up Link Prediction



- 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!

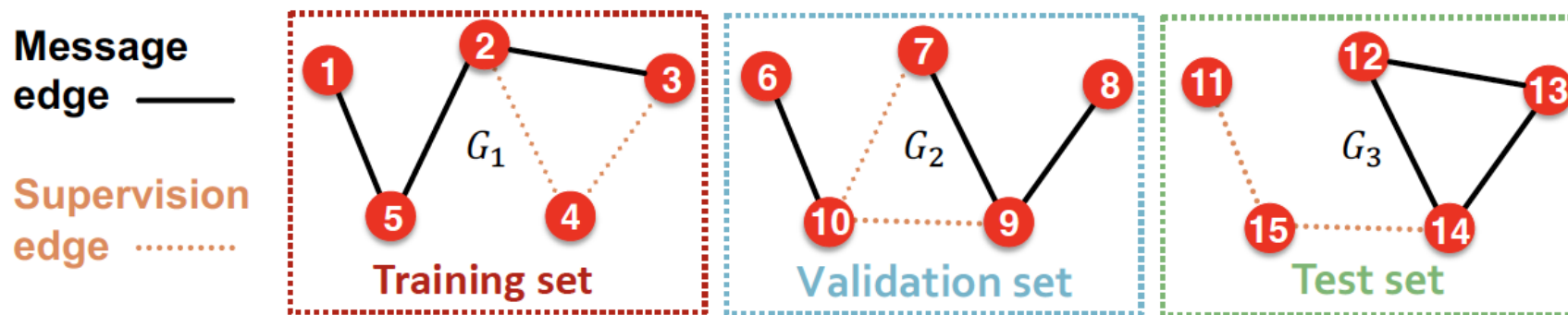
Setting up Link Prediction

- 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



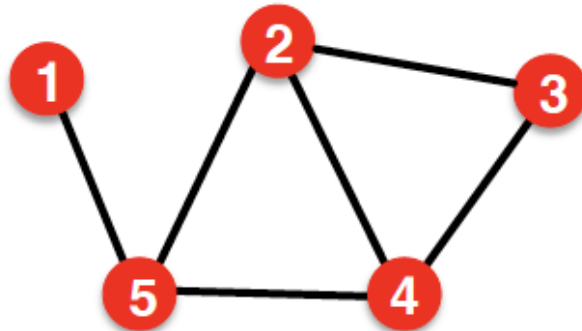
Setting up Link Prediction

- 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 validation or test set, each graph will have 2 types of edges: **message edges** + **supervision edges**
 - Supervision edges are not the input to GNN



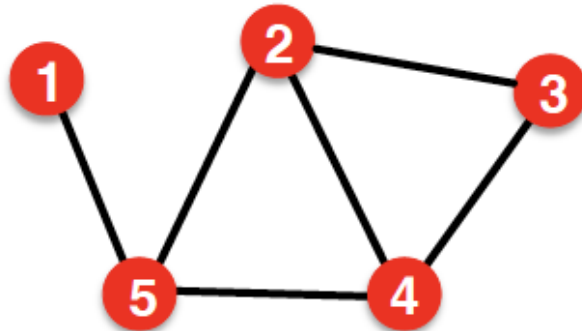
Setting up Link Prediction

- **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



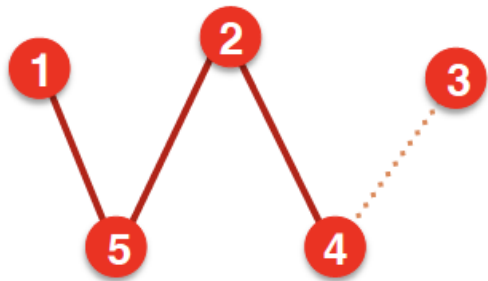
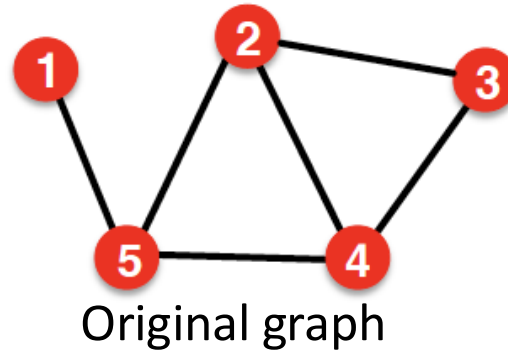
Setting up Link Prediction

- **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

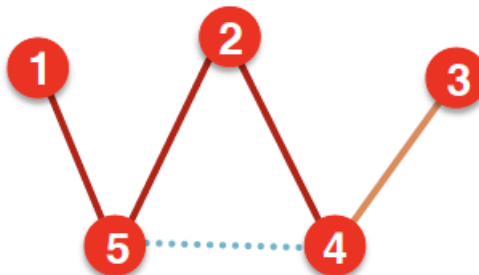


Setting Up Link Prediction

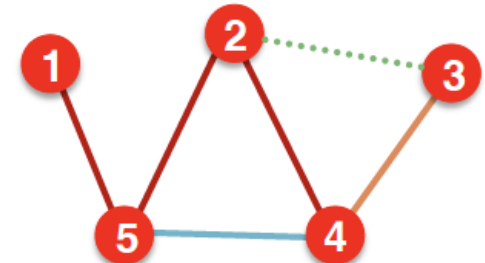
- Option 2: Transductive link prediction split:



(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

Setting Up Link Prediction

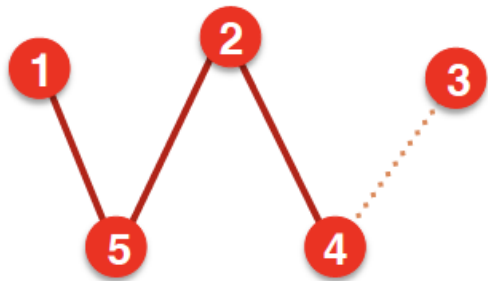
- Option 2: Transductive link prediction split:

Why do we use growing number of edges?

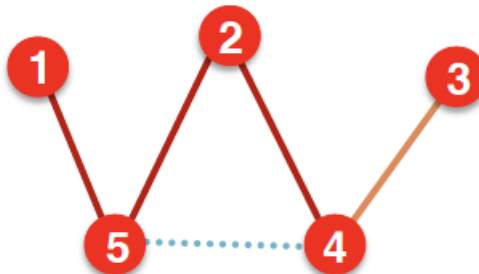
After training, supervision edges are known to GNN.

Therefore, an ideal model should use supervision edges in message passing at validation time.

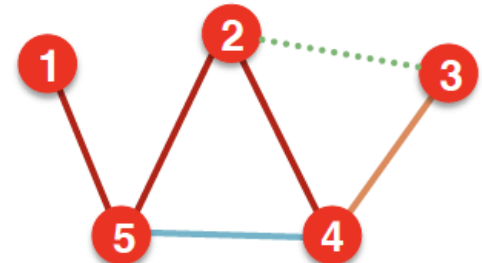
The same applies to the test time.



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



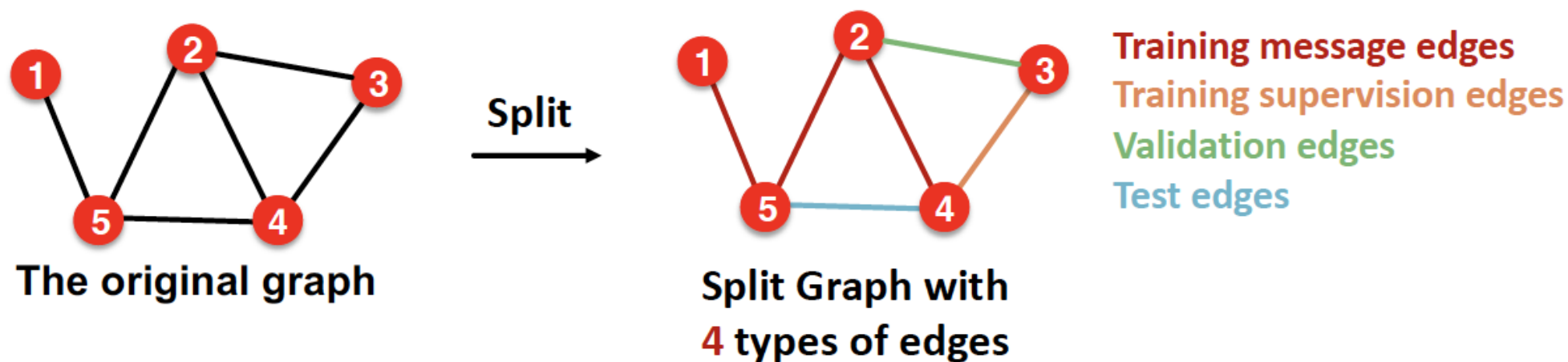
(2) At validation time: Use training message edges & training supervision edges to predict validation edges



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

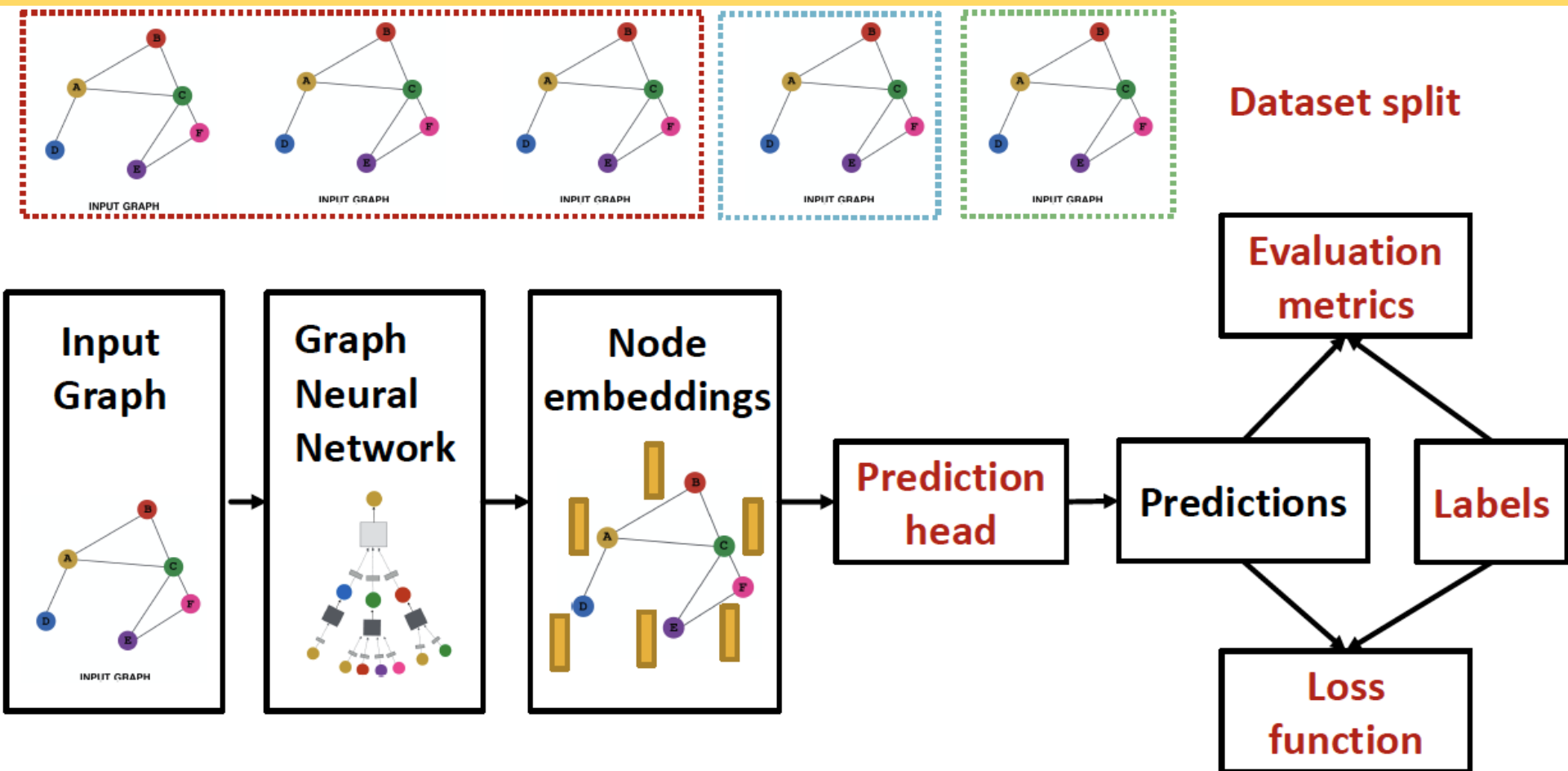
Setting Up Link Prediction

- **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

GNN Training Pipeline



- Implementation resources:
 - GraphGym further implements the full pipeline to facilitate GNN design

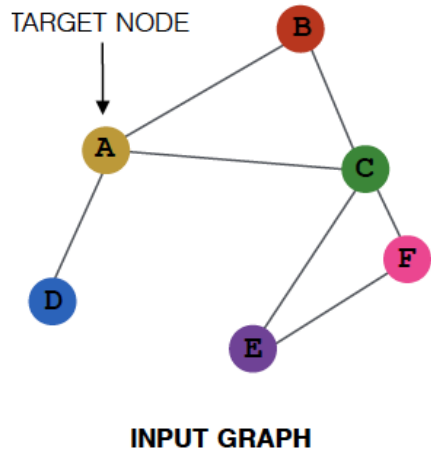
Summary

- We introduce a general GNN framework:
 - **GNN Layer:**
 - Transformation + Aggregation
 - Classic GNN layers: GCN, GraphSAGE, GAT
 - **Layer connectivity:**
 - The over-smoothing problem
 - Solution: skip connections
 - **Graph Augmentation:**
 - Feature augmentation
 - Structure augmentation
 - **Learning Objectives**
 - The full training pipeline of a GNN

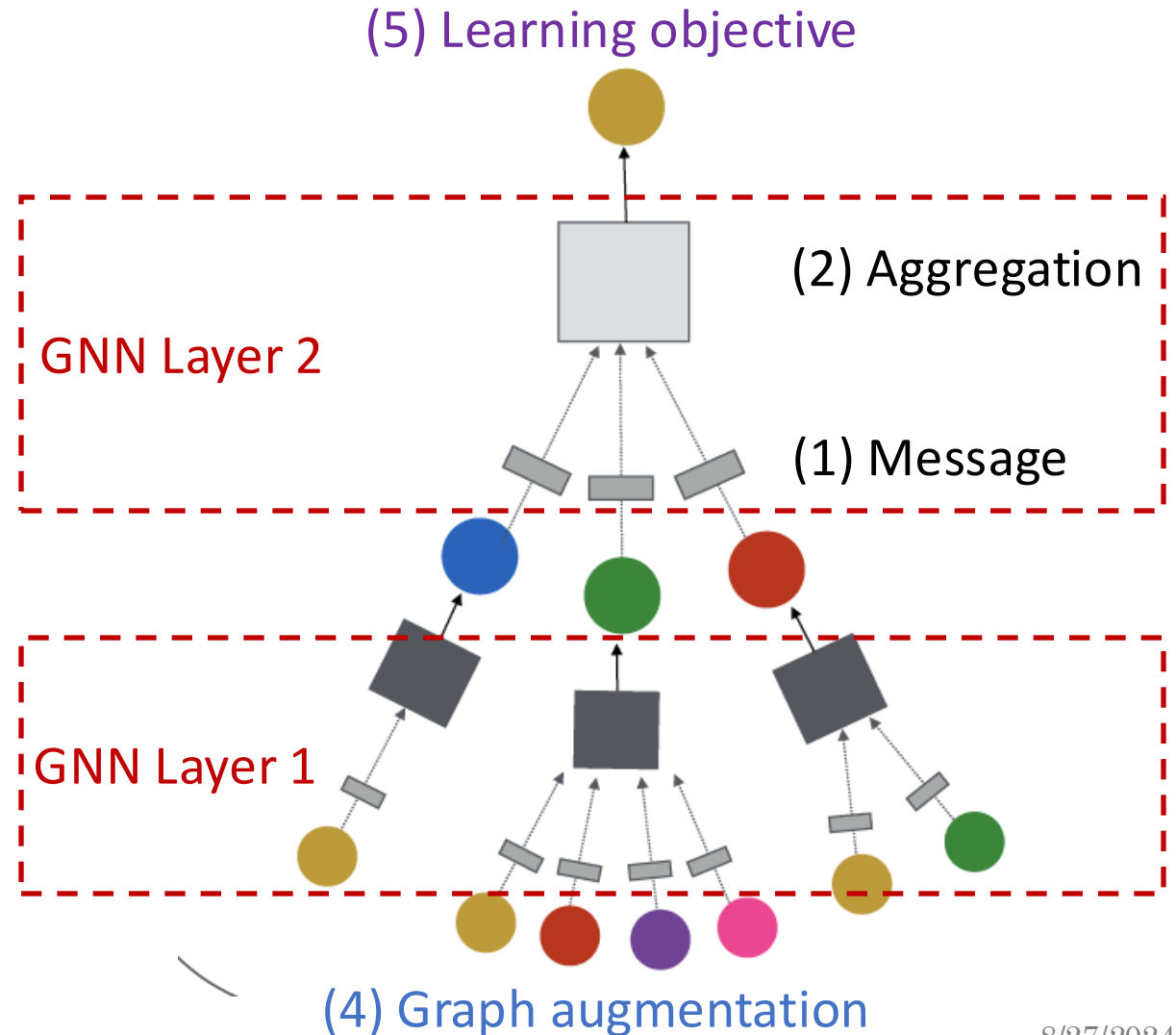
Coffee Break!

How Expressive are Graph Neural Networks?

Recap: A General GNN Framework



(3) Layer connectivity



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?

Recap: A Single GNN Layer

- Putting things together

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

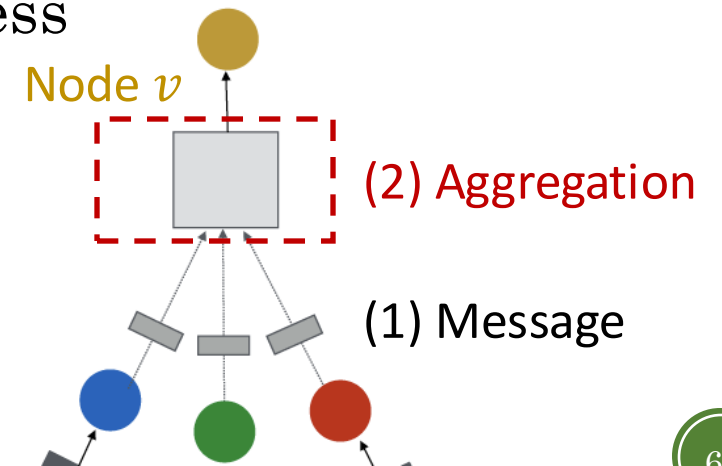
$$m_u^{(l)} = MSG^{(l)}(h_u^{(l-1)}), \forall u \in N(v) \cup \{v\}$$

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

$$h_v^{(l)} = AGG^{(l)}(\{m_u^{(l)}, u \in N(v)\}, m_v^{(l)})$$

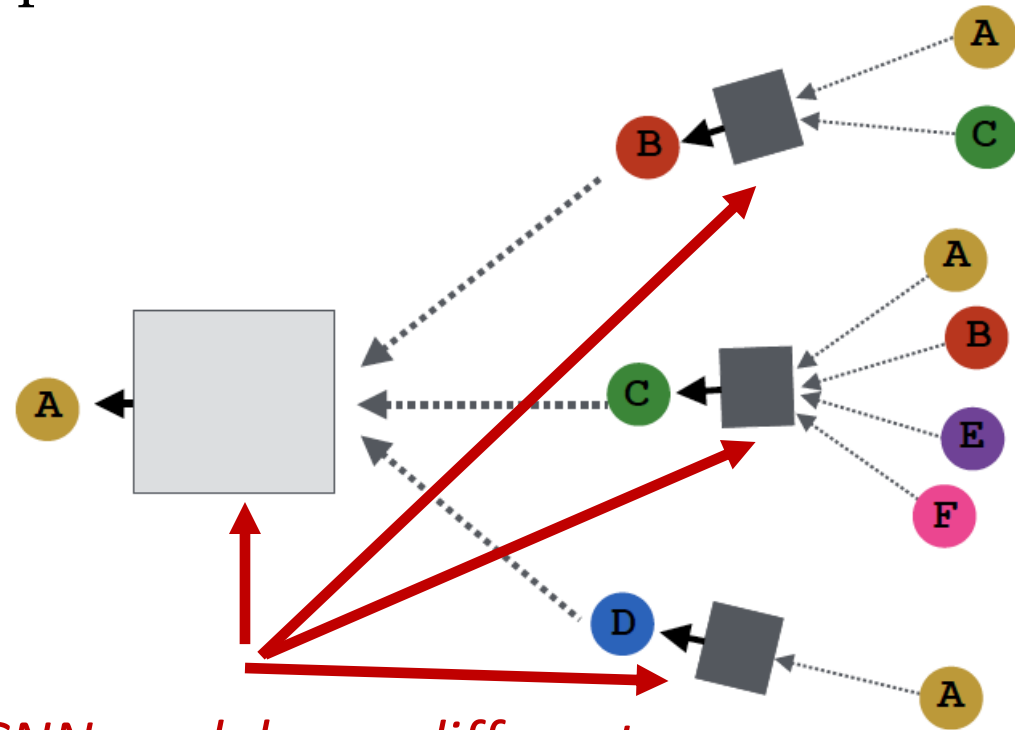
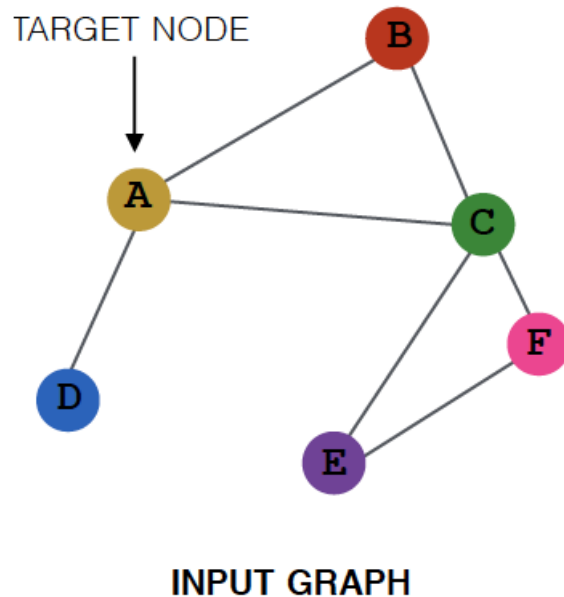
- **Non-linearity** (activation function): Add expressiveness

- Often write as $\sigma(\cdot)$. Example: ReLU(), Sigmoid(), etc
- Can be added to message or aggregation



Recap: Many GNN Layers

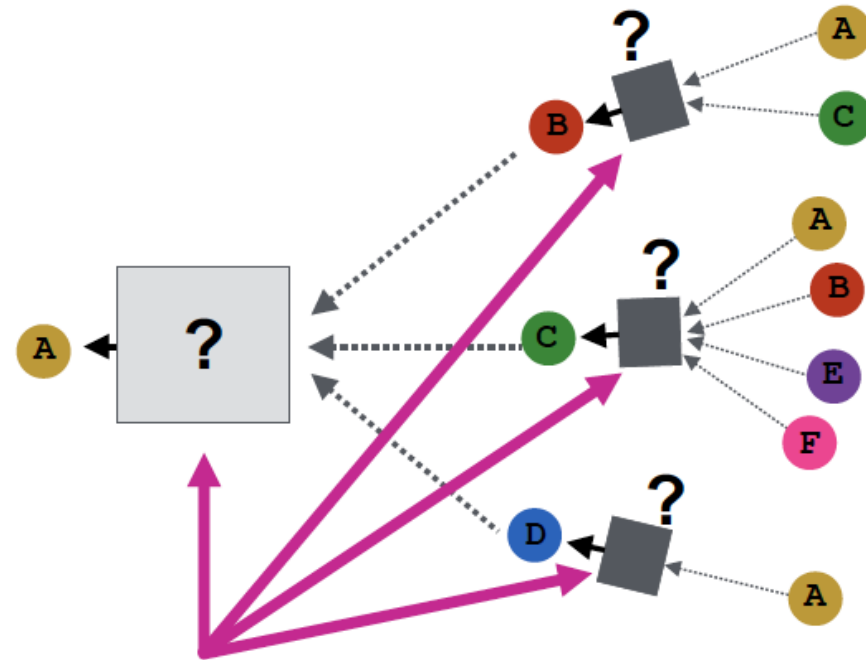
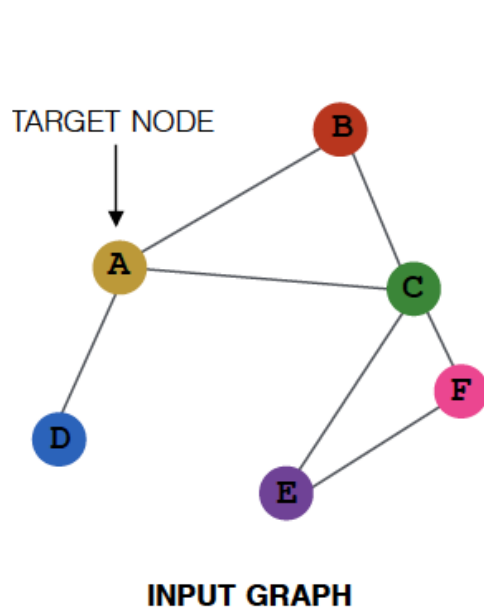
- Many GNN models have been proposed:
 - GCN, GraphSAGE, GAT, Design Space etc.



Different GNN models use different neural networks in the box

Recap: GNN Model Example (1)

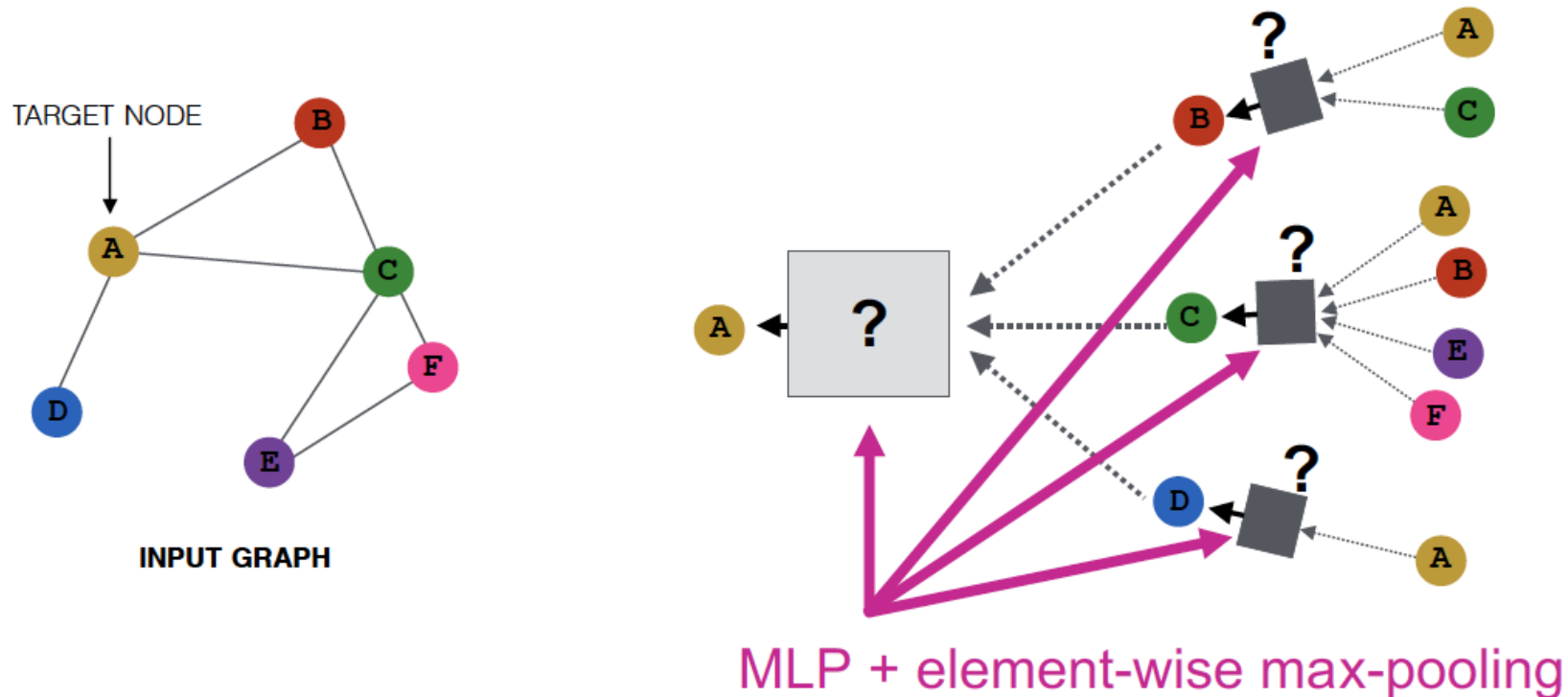
- GCN (mean-pool) ([Kipf and Welling ICLR 2017])



Element-wise mean pooling +
Linear + ReLU non-linearity

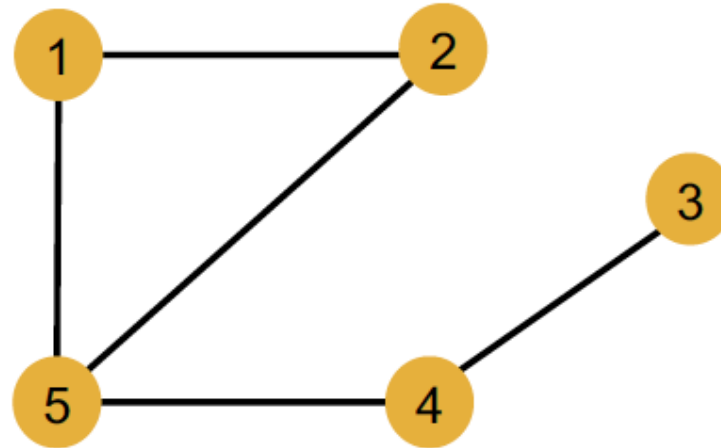
Recap: GNN Model Example (2)

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



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.

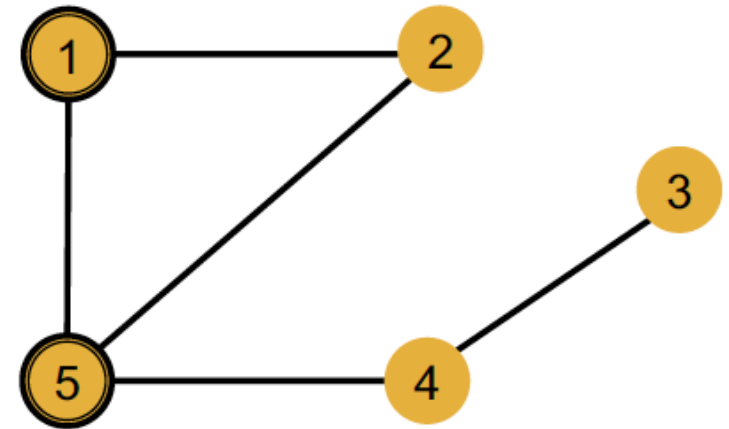


- **Key question:** How well can a GNN distinguish different graph structures?

Local Neighborhood Structures

- We specifically consider local neighborhood structures around each node in a graph.

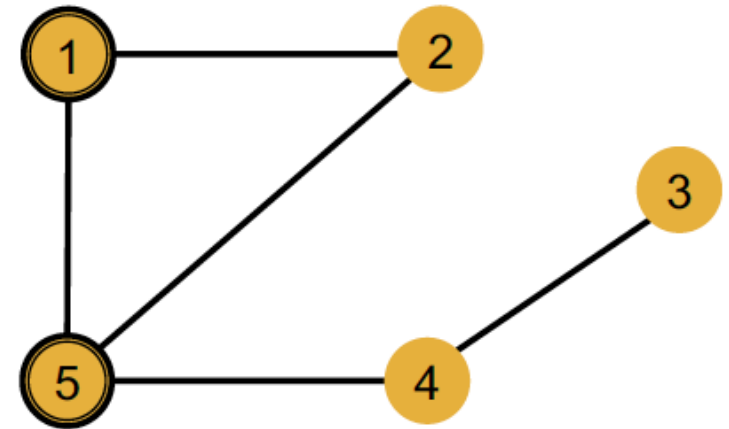
Example: Nodes 1 and 5 have **different neighborhood structures** because they have **different node degrees**.



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.

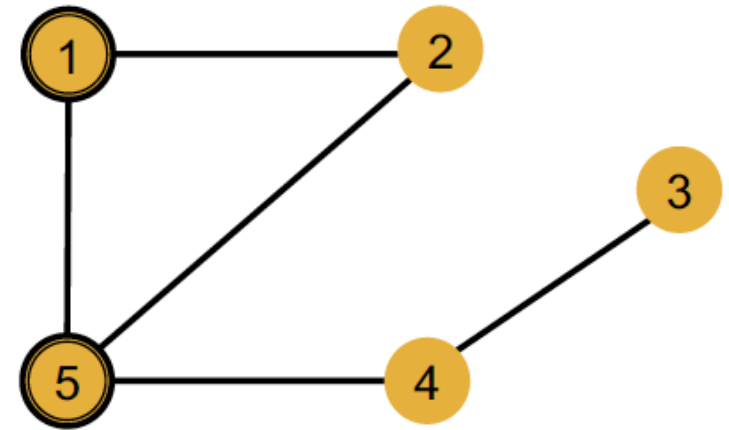


*Node 1 has neighbors of degrees 2 and 3.
Node 4 has neighbors of degrees 1 and 3.*

Local Neighborhood Structures

- We specifically consider local neighborhood structures around each node in a graph.

Example: *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.

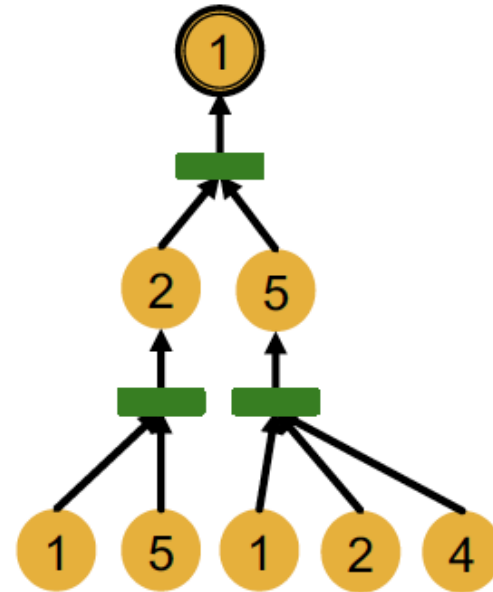
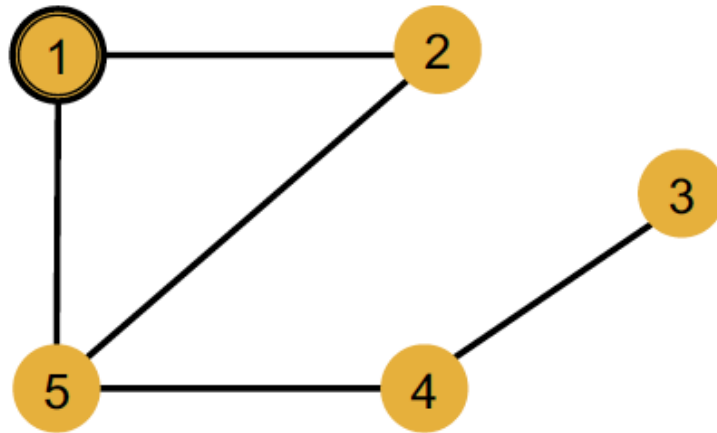
And even if we go a step deeper to 2nd hop neighbors, both nodes have the same degrees (Node 4 of degree 2)

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

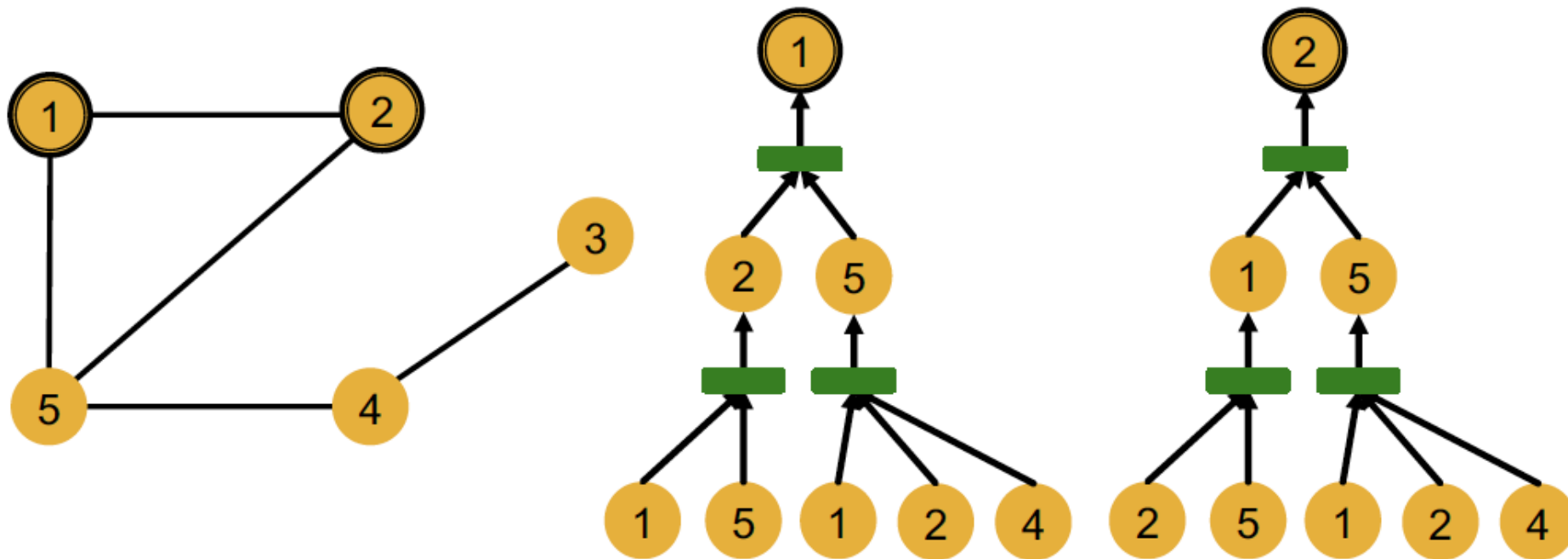
Computational Graph

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



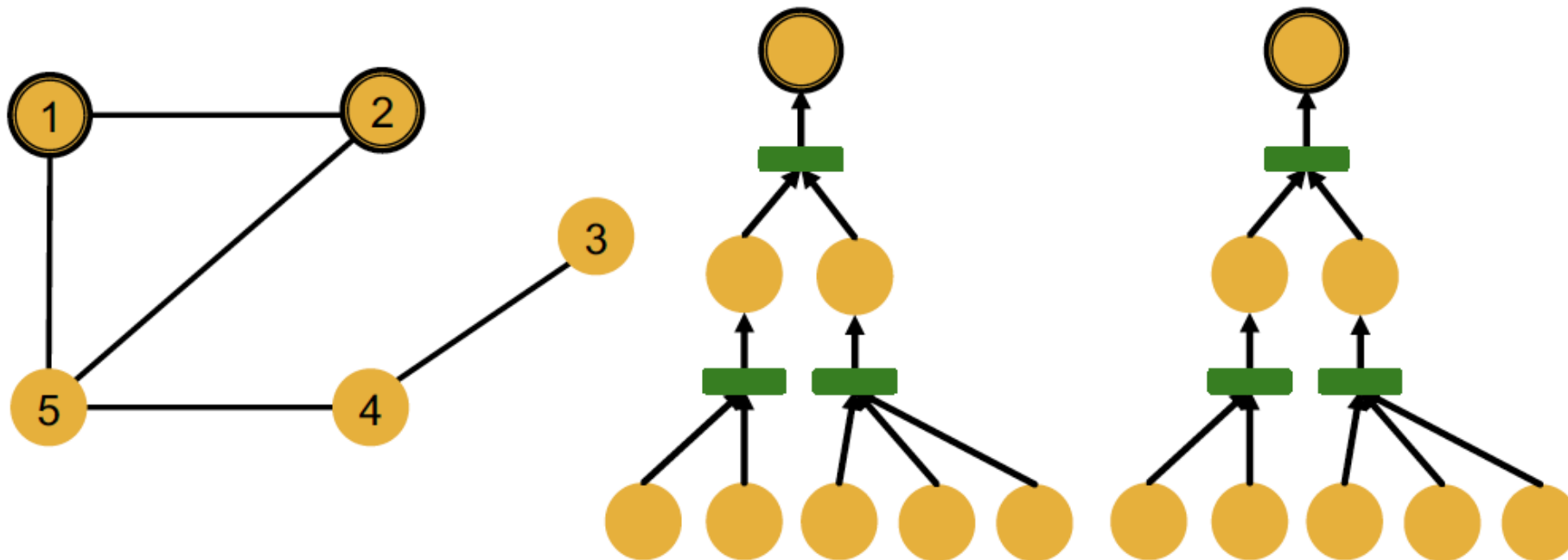
Computational Graph

- Example: Node 1 and 2's computational graphs



Computational Graph

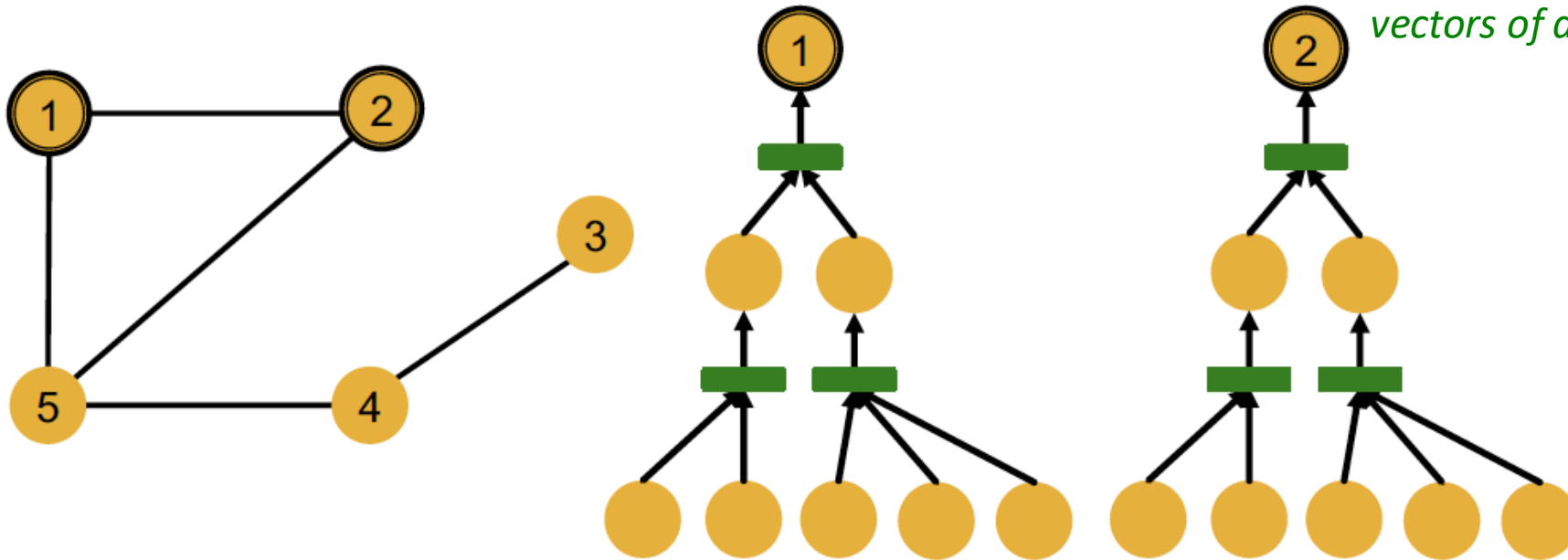
- Example: Node 1 and 2's computational graphs
- But GNN only see node features (not IDs)



Computational Graph

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

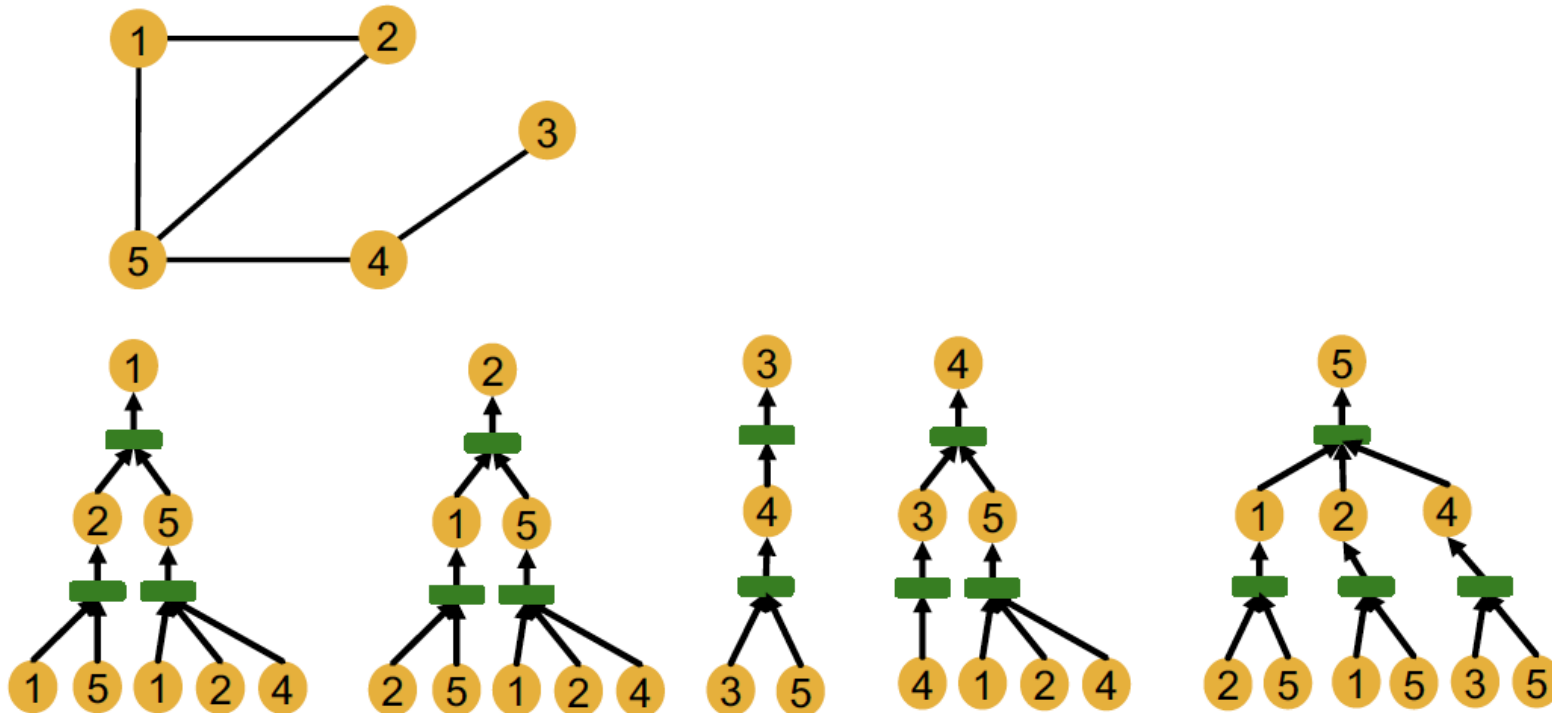
Note: GNN does not care about node ids, it just aggregates features vectors of different nodes



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

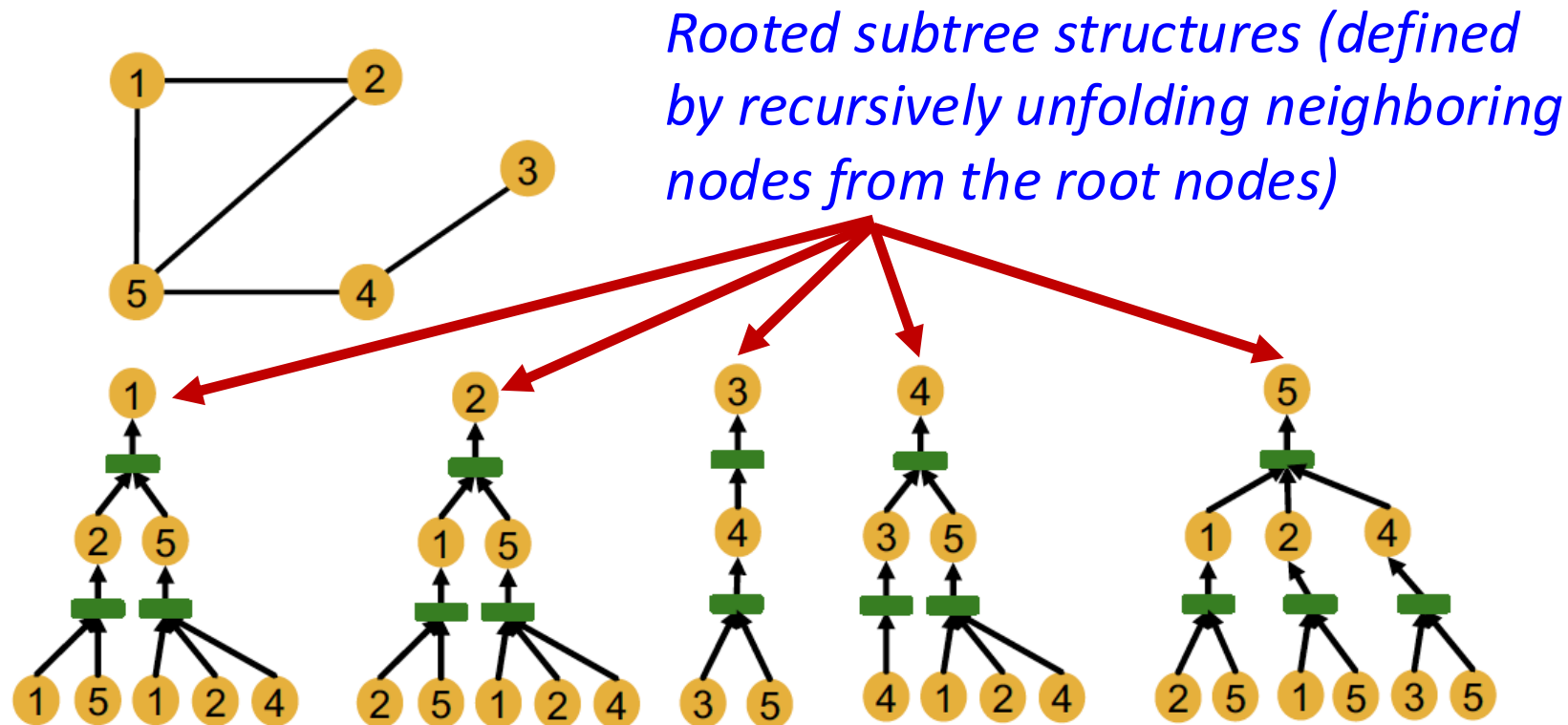
Computational Graph

- In general, different local neighborhoods define different computational graphs



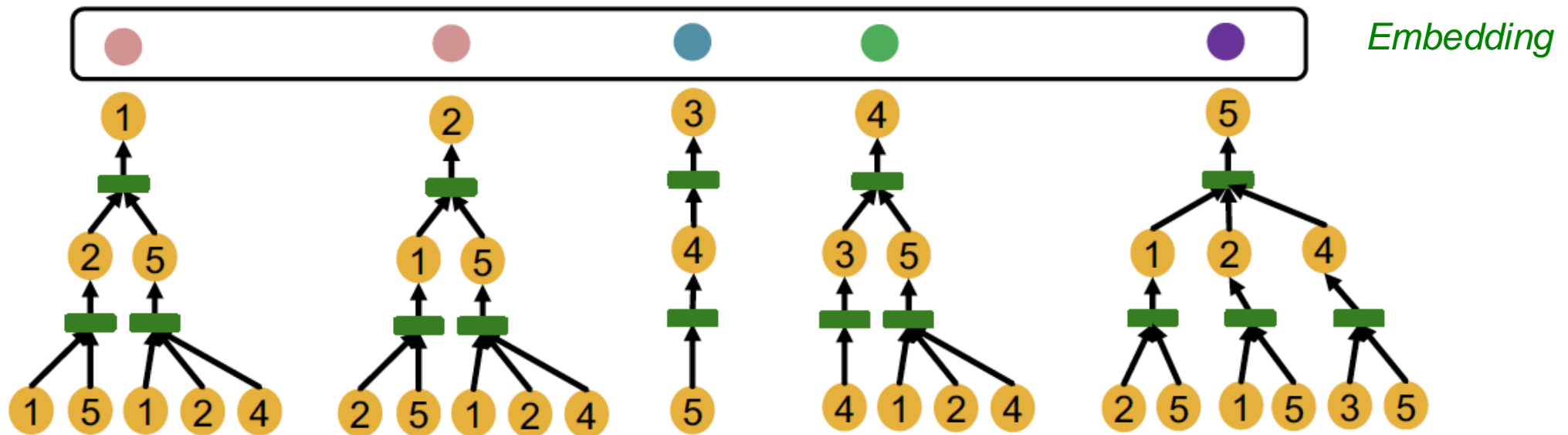
Computational Graph

- In general, different local neighborhoods define different computational graphs



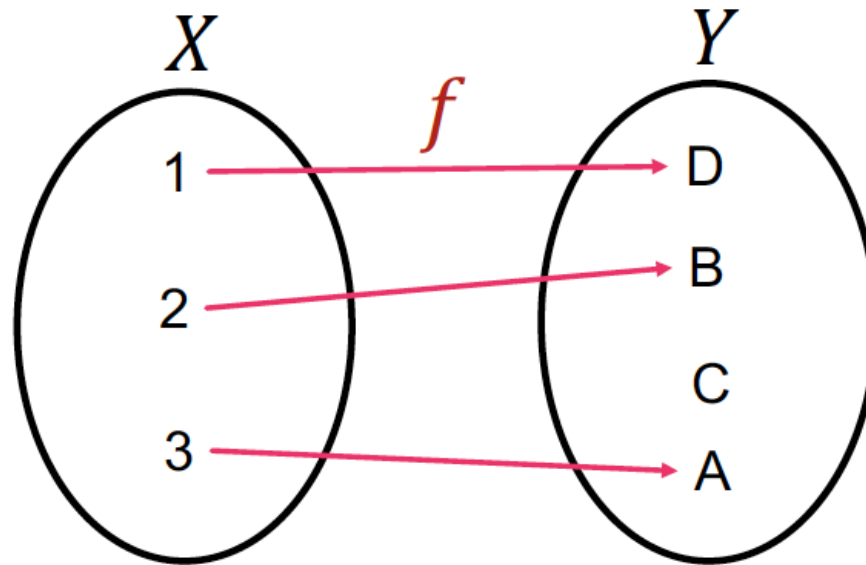
Computational Graph

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



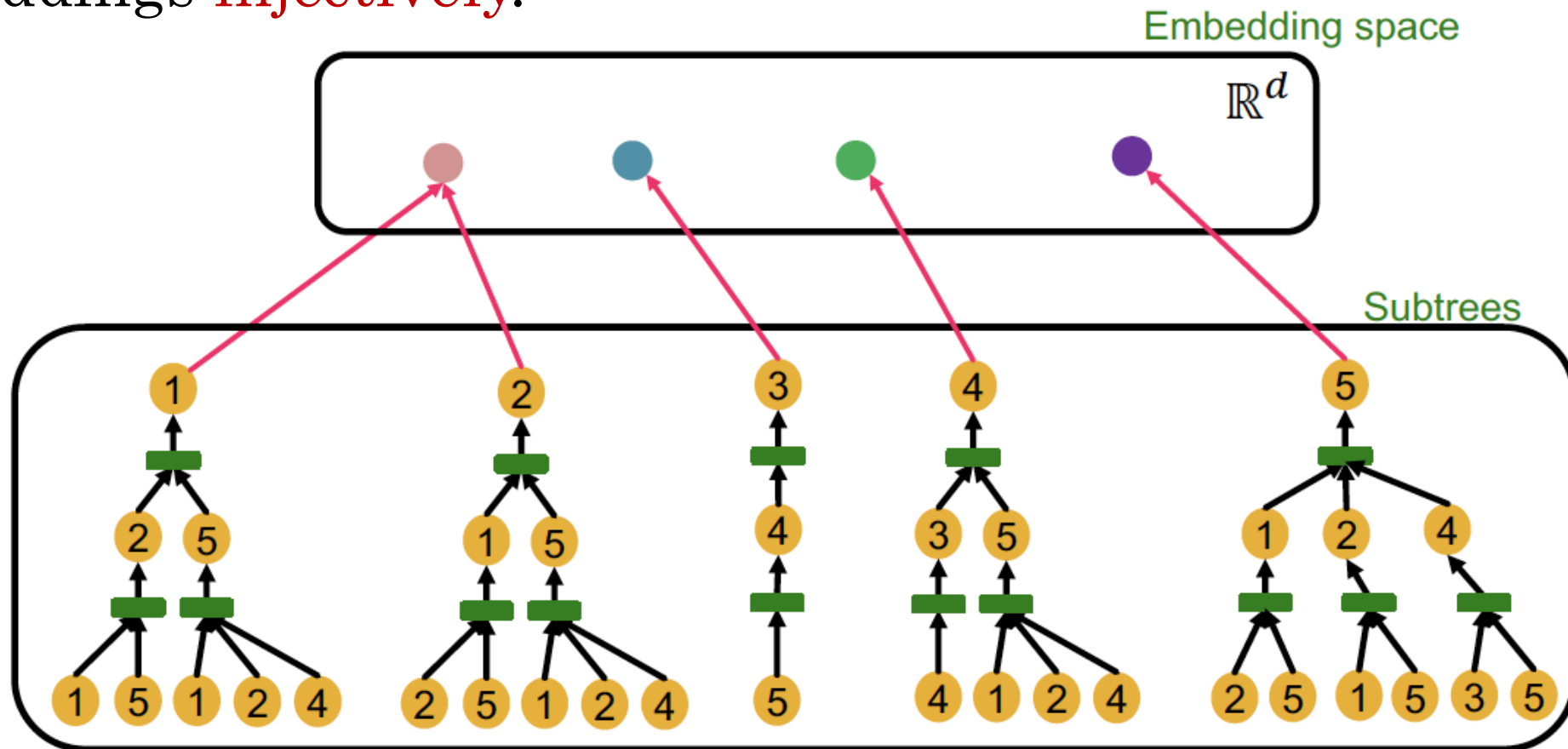
Recall: Injective Function

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



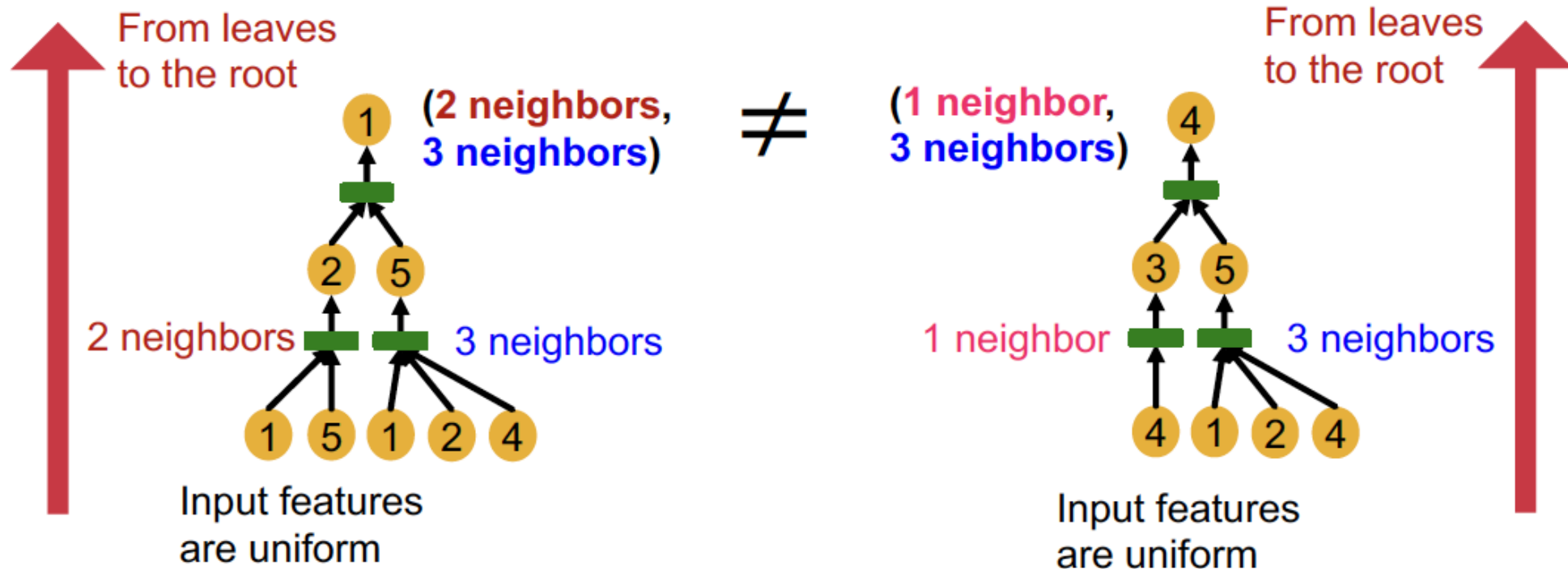
How Expressive is a GNN?

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



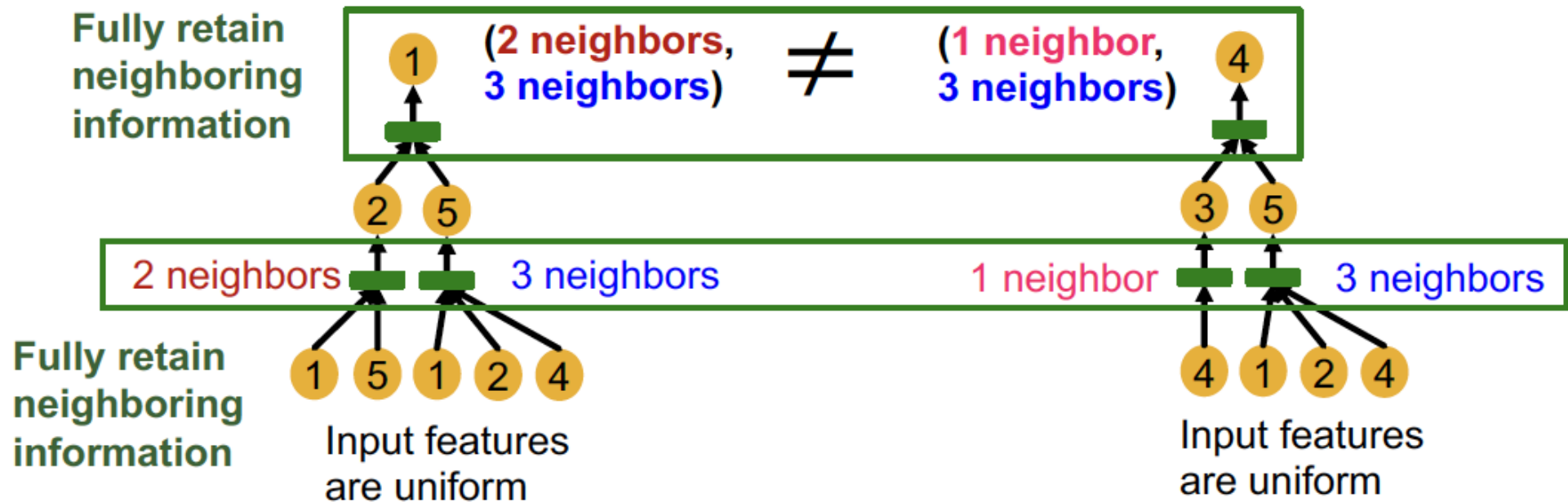
How Expressive is a GNN?

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



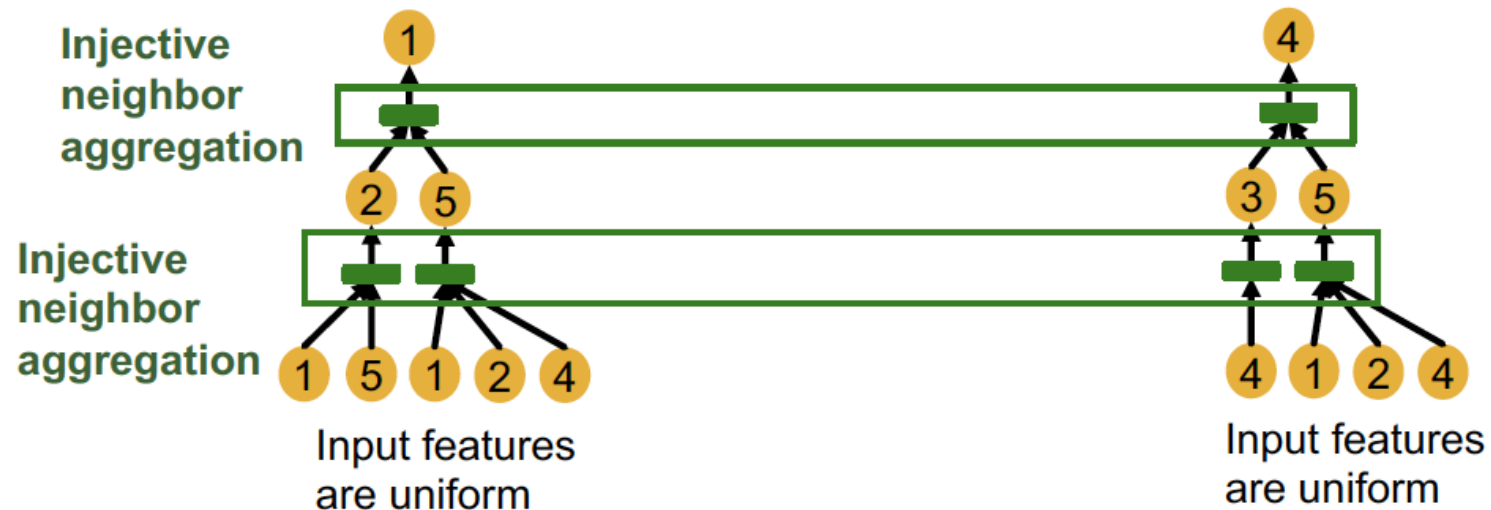
How Expressive is a GNN?

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



How Expressive is a GNN?

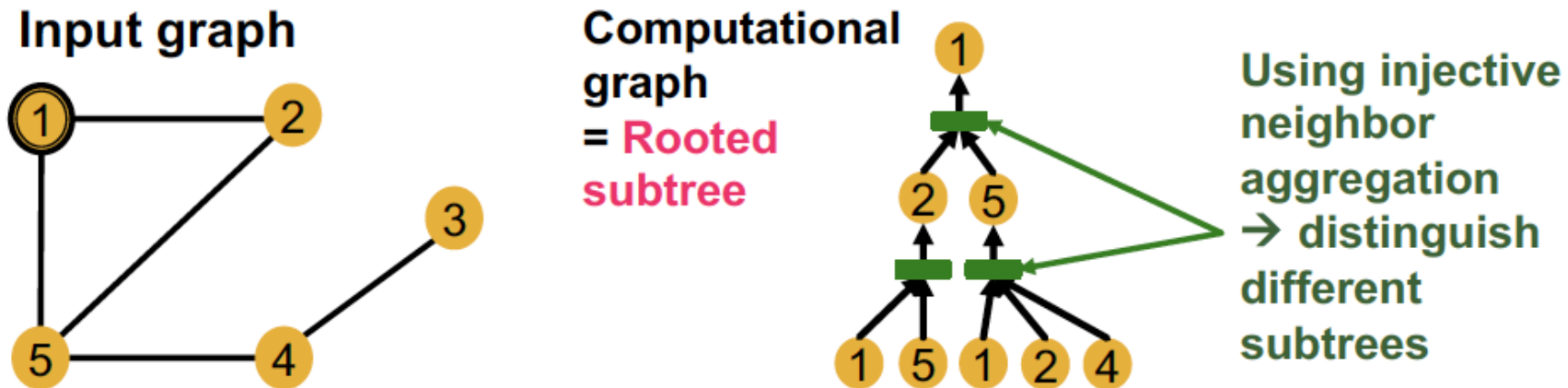
- In other words, most expressive GNN would use an injective neighbor aggregation function at each step.
 - Maps different neighbors to different embeddings.



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.

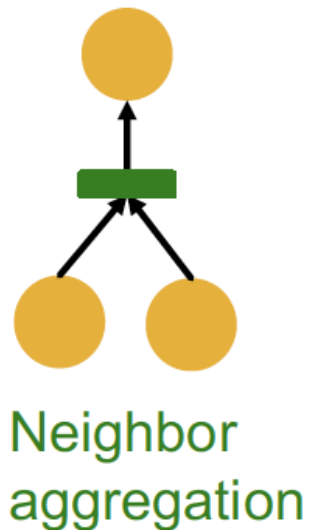
Designing the Most Powerful GNN

Expressive Power of GNNs

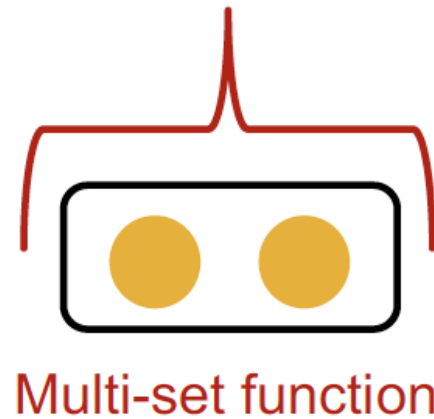
- **Key observation:** Expressive power of GNNs can be characterized by that of neighbor aggregation functions they use.
 - A more expressive aggregation function leads to a more expressive GNN.
 - Injective aggregation function leads to the most expressive GNN.
- **Next:** Theoretically analyze expressive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

Neighbor Aggregation

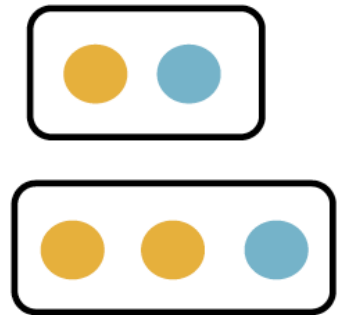
- **Observation:** Neighbor aggregation can be abstracted as a function over a multi-set (a set with repeating elements).



Equivalent



Examples of multi-set



Same color indicates the same features.

Neighbor Aggregation

- Next: We analyze aggregation functions of two popular GNN models

- GCN (mean-pool) ([Kipf and 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

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

Neighbor Aggregation: Case Study

- GCN (mean-pool) ([Kipf and 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?

Neighbor Aggregation

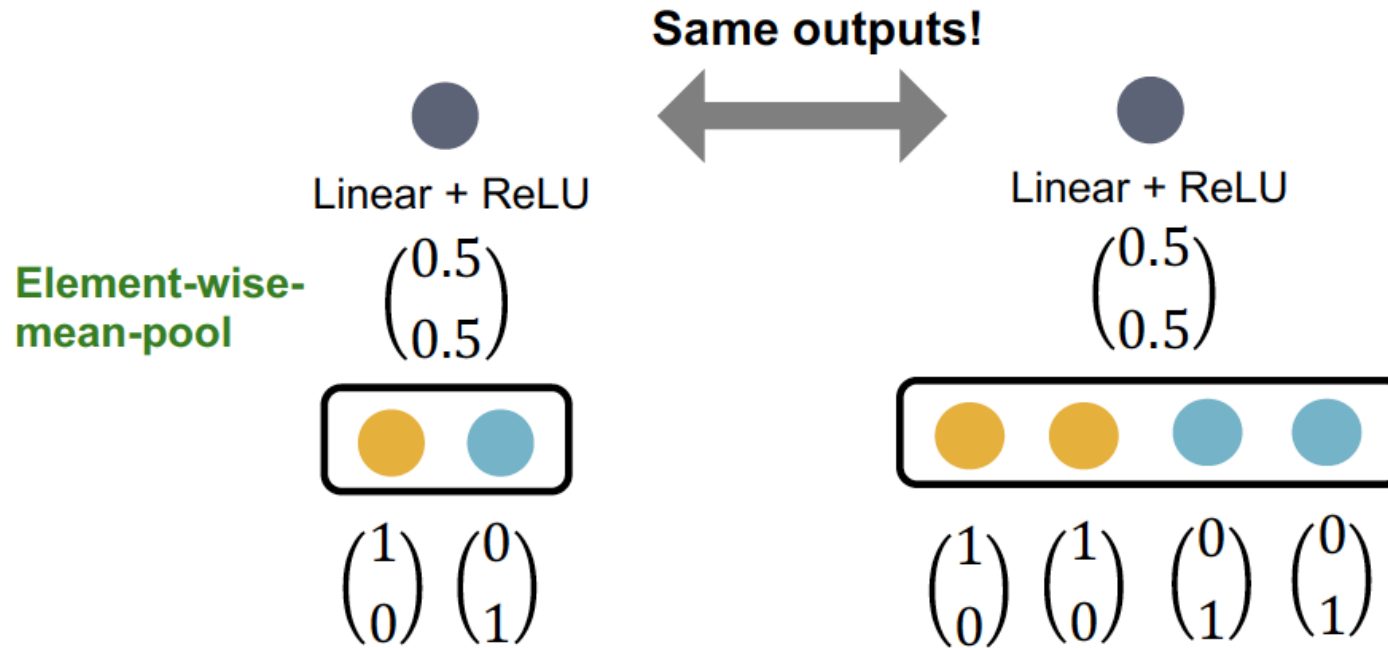
- For simplicity, we assume node features (colors) are represented by one-hot encoding.
- Example: If there are two distinct colors:

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

- This assumption is sufficient to illustrate how GCN fails.

Neighbor Aggregation: Case Study

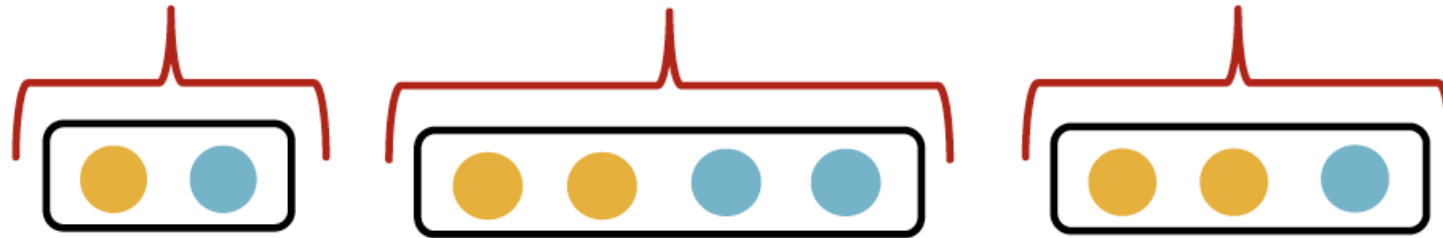
- GCN (mean-pool) ([Kipf and Welling ICLR 2017])
 - Failure case illustration



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.**

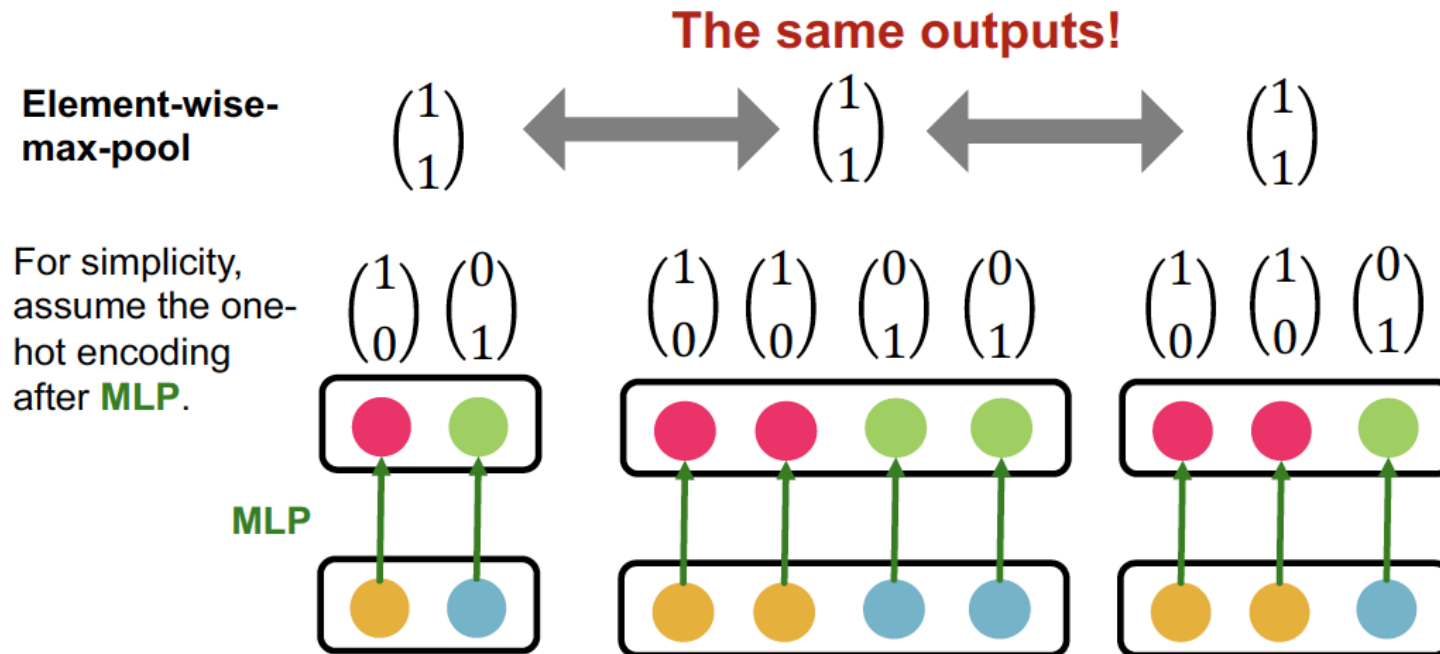
Failure case



- Why?

Neighbor Aggregation: Case Study

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



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.

Designing 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 multisets.
- Here, we design a neural network that can model injective multiset function.

Injective Multi-Set Function


- Theorem[Xu et al. ICLR 2019]: Any injective multi-set function can be expressed as:

Some non-linear function $\longrightarrow \Phi \left(\sum_{x \in S} f(x) \right)$

Some non-linear function $\longleftarrow f(x)$

Sum over multi-set $\longleftarrow \sum_{x \in S}$

S : multi-set

 $\longrightarrow \Phi \left[f(\text{yellow circle}) + f(\text{blue circle}) + f(\text{blue circle}) \right]$

Injective Multi-Set Function

- Proof Intuition:

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

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

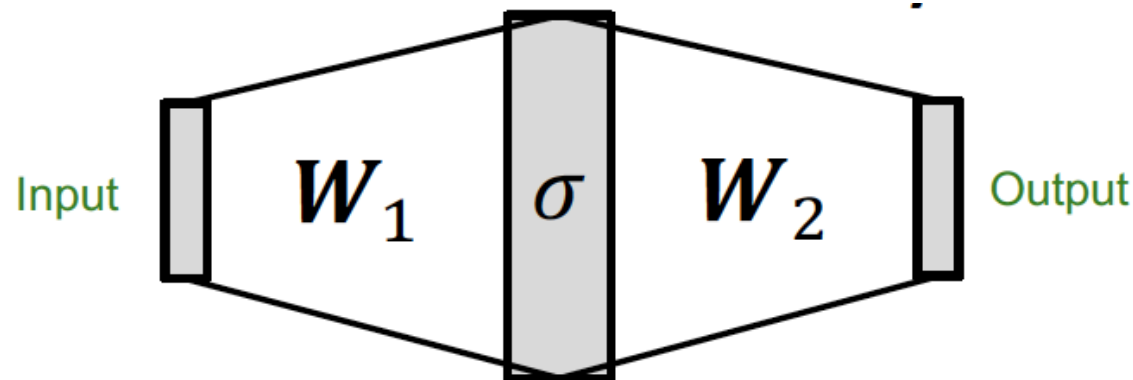
- Example:

$$\Phi\left[\underbrace{f[\text{yellow}]}_{\begin{pmatrix} 1 \\ 0 \end{pmatrix}} + \underbrace{f[\text{blue}]}_{\begin{pmatrix} 0 \\ 1 \end{pmatrix}} + \underbrace{f[\text{blue}]}_{\begin{pmatrix} 0 \\ 1 \end{pmatrix}}\right]$$

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

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.



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.

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!

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.

Relation to WL Graph Kernel

Recap: Color Refinement 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

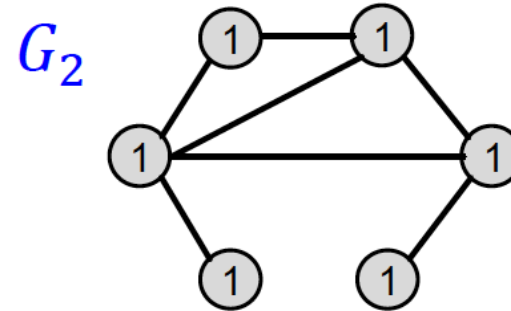
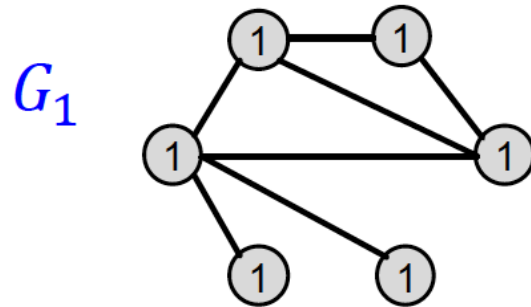
$$c^{(k+1)}(v) = \text{HASH} \left(\left\{ c^{(k)}(v), \{c^{(k)}(u)\}_{u \in N(v)} \right\} \right)$$

where HASH maps different inputs to different colors

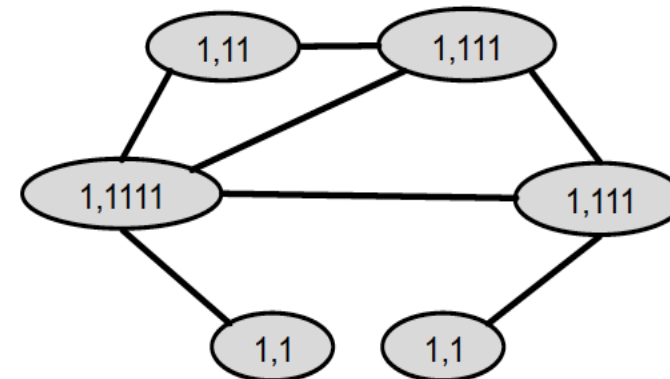
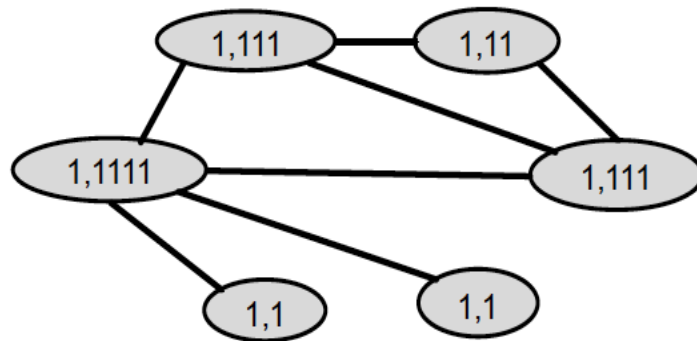
- After K steps of color refinement, $c^{(K)}(v)$ summarizes the structure of K -hop neighborhood.

Color Refinement: Example

- Assign initial colors

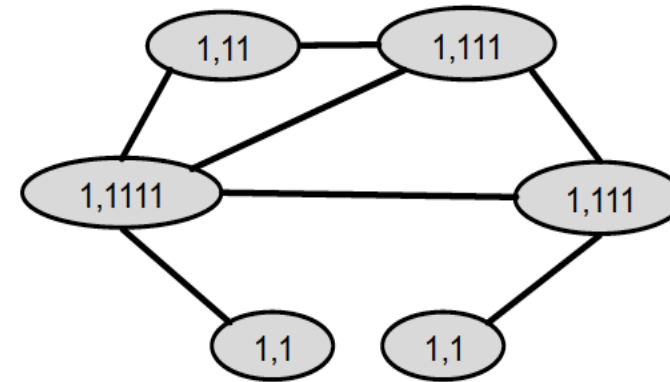
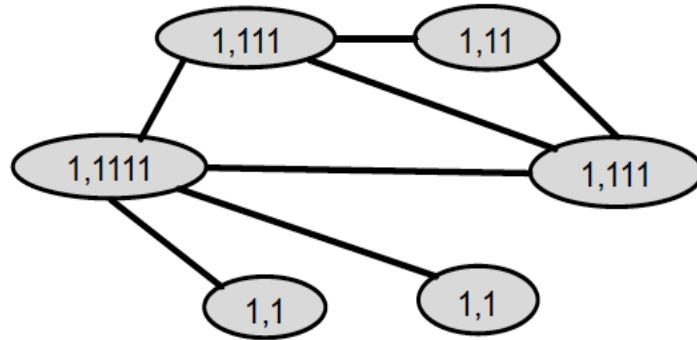


- Aggregate neighboring colors

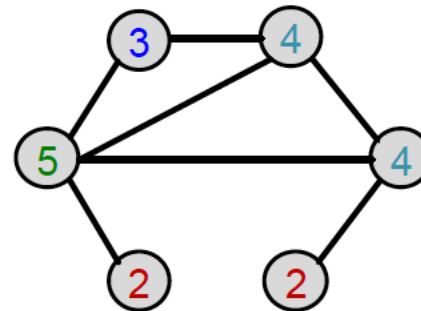
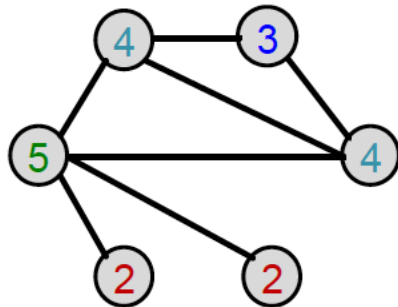


Color Refinement: Example

- Aggregate neighboring colors



- Hash aggregated colors

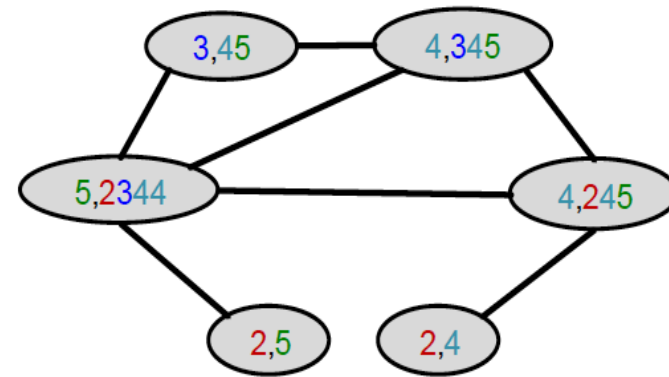
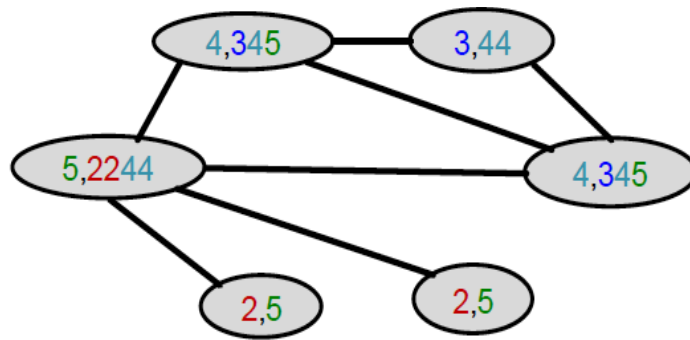


Hash table

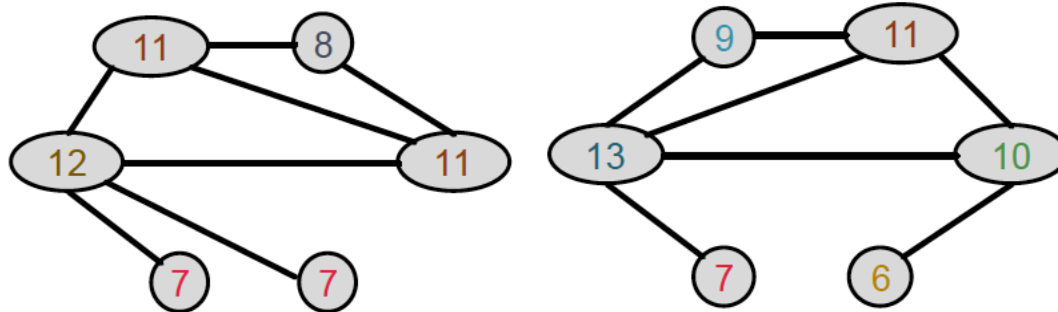
1,1	-->	2
1,11	-->	3
1,111	-->	4
1,1111	-->	5

Color Refinement: Example

- Aggregate neighboring colors



- Hash aggregated colors



Hash table

2,4	-->	6
2,5	-->	7
3,44	-->	8
3,45	-->	9
4,245	-->	10
4,345	-->	11
5,2244	-->	12
5,2344	-->	13

Weisfeiler-Lehman Graph Features

- After color refinement, WL kernel counts #nodes with a given color

$$\phi\left(\begin{array}{c} \text{Graph 1} \end{array}\right) = \begin{array}{c} \text{Colors} \\ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13 \\ \text{Counts} \\ [6, 2, 1, 2, 1, 0, 2, 1, 0, 0, 2, 1, 0] \end{array}$$

$$\phi\left(\begin{array}{c} \text{Graph 2} \end{array}\right) = \begin{array}{c} 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13 \\ [6, 2, 1, 2, 1, 1, 1, 0, 1, 1, 1, 0, 1] \end{array}$$

The Complete GIN Model

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

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

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

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

Root node
feature

Neighboring
node colors

The Complete GIN Model

- Theorem[Xu et al. ICLR 2019]: Any injective function over the tuple $\left(\left\{c^{(k)}(v), \{c^{(k)}(u)\}_{u \in N(v)}\right\}\right)$ can be modeled as:

$$MLP_{\Phi} \left((1 + \epsilon) \cdot MLP_f \left(c^{(k)}(v) \right) + \sum_{u \in N(v)} MLP_f \left(c^{(k)}(u) \right) \right)$$

- Where ϵ is a learnable scalar.

The Complete GIN Model

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

- Example:

$$\Phi \left[\underbrace{f[\text{yellow circle}]}_{\text{One-hot } \begin{pmatrix} 1 \\ 0 \end{pmatrix}} + \underbrace{f[\text{blue circle}]}_{\begin{pmatrix} 0 \\ 1 \end{pmatrix}} + \underbrace{f[\text{blue circle}]}_{\begin{pmatrix} 0 \\ 1 \end{pmatrix}} \right]$$

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

- We only need Φ to ensure injectivity

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

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

This MLP can provide “one-hot” input feature for the next layer.

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

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

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.

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

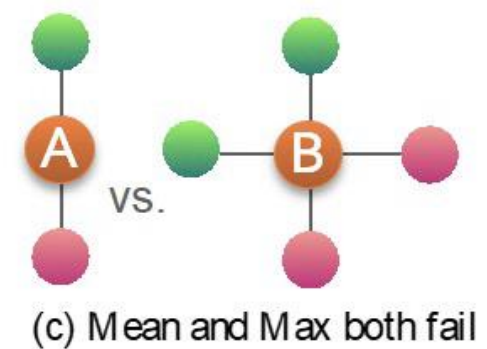
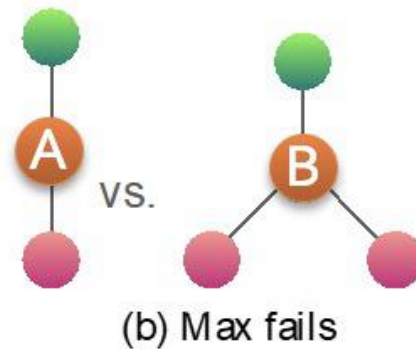
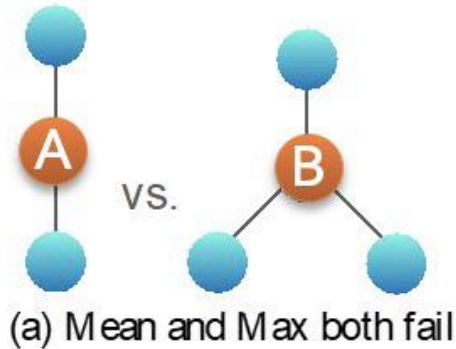
- Advantages of GIN over the WL graph kernel are:
 - Node embeddings are low-dimensional; hence, they can capture the fine-grained similarity of different nodes.
 - Parameters of the update function can be learned for the downstream tasks.

Expressive Power of GIN

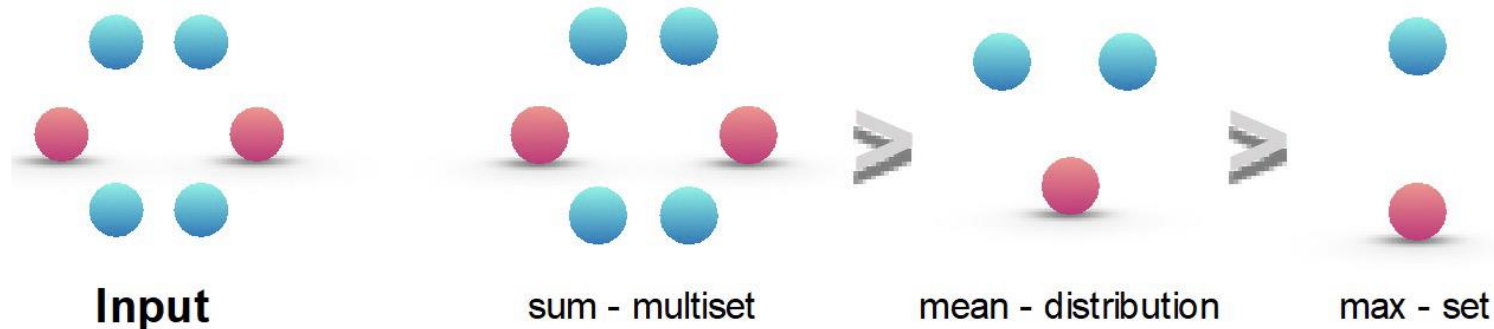
- Because of the relation between GIN and the WL graph kernel, their expressive is exactly the same.
 - If two graphs can be distinguished by GIN, they can be also distinguished by the WL kernel, and vice versa.
- How powerful is this?
 - WL kernel has been both theoretically and empirically shown to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

Discussion: The Power of Pooling

- Failure cases for mean and max pooling:

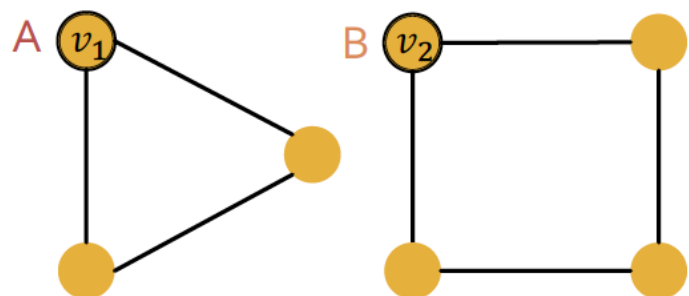


- Ranking by discriminative power:

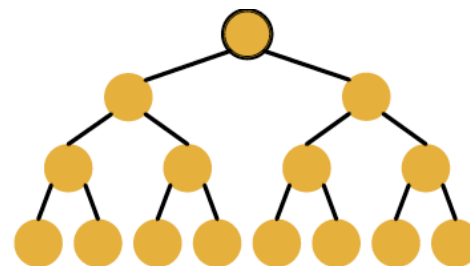


Improving GNN Power

- 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]

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!

When Things Do Not Go as Planned

General Tips

- **Data preprocessing** is important:
 - Node attributes can vary a lot! Use normalization
 - E.g. probability ranges $(0,1)$, but some inputs could have much larger range, say $(-1000, 1000)$
- **Optimizer**: ADAM is relatively robust to learning rate
- **Activation function**
 - ReLU activation function often works well
 - Other good alternatives: LeakyReLU, PReLU
 - No activation function at your output layer
 - Include bias term in every layer
- **Embedding dimensions**:
 - 32, 64 and 128 are often good starting points

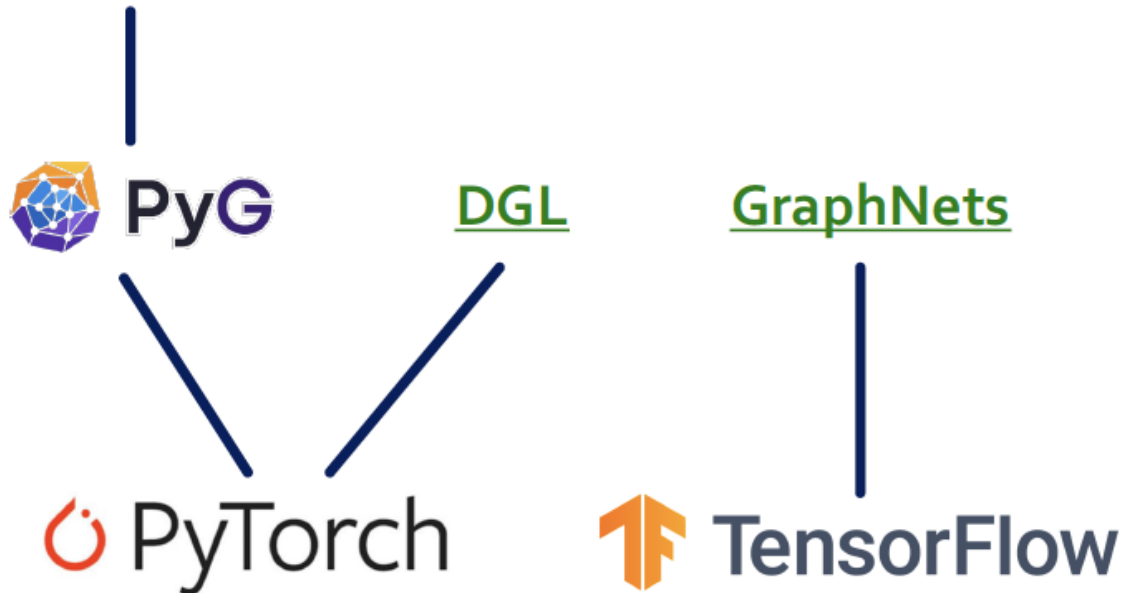
Debugging Deep Networks

- **Debug issues:** Loss/accuracy not converging during training
 - Check pipeline (e.g. in PyTorch we need `zero_grad`)
 - Adjust hyperparameters such as learning rate
 - Pay attention to weight parameter initialization
 - Scrutinize loss function!
- **Important for model development:**
 - Overfit on (part of) training data:
 - With a small training dataset, loss should be essentially close to 0, with an expressive neural network
 - Monitor the training & validation loss curve

Resources on Graph Neural Networks

GraphGym:

Easy and flexible end-to-end GNN pipeline
based on PyTorch Geometric (PyG)



GNN frameworks:

Implements a variety
of GNN architectures

Auto-differentiation frameworks

Resources on Graph Neural Networks

- Tutorials and overviews:
 - Relational inductive biases and graph networks (Battaglia et al., 2018)
 - Representation learning on graphs: Methods and applications (Hamilton et al., 2017)
- Attention-based neighborhood aggregation:
 - Graph attention networks (Hoshen, 2017; Velickovic et al., 2018; Liu et al., 2018)
- Embedding entire graphs:
 - Graph neural nets with edge embeddings (Battaglia et al., 2016; Gilmer et al., 2017)
 - Embedding entire graphs (Duvenaud et al., 2015; Dai et al., 2016; Li et al., 2018) and graph pooling (Ying et al., 2018, Zhang et al., 2018)
 - Graph generation and relational inference (You et al., 2018; Kipf et al., 2018)
 - How powerful are graph neural networks (Xu et al., 2017)

Resources on Graph Neural Networks

- Embedding nodes:
 - Varying neighborhood: Jumping knowledge networks (Xu et al., 2018), GeniePath (Liu et al., 2018)
 - Position-aware GNN (You et al. 2019)
- Spectral approaches to graph neural networks:
 - Spectral graph CNN & ChebNet (Bruna et al., 2015; Defferrard et al., 2016)
 - Geometric deep learning (Bronstein et al., 2017; Monti et al., 2017)
- Other GNN techniques:
 - Pre-training Graph Neural Networks (Hu et al., 2019)
 - GNNExplainer: Generating Explanations for Graph Neural Networks (Ying et al., 2019)