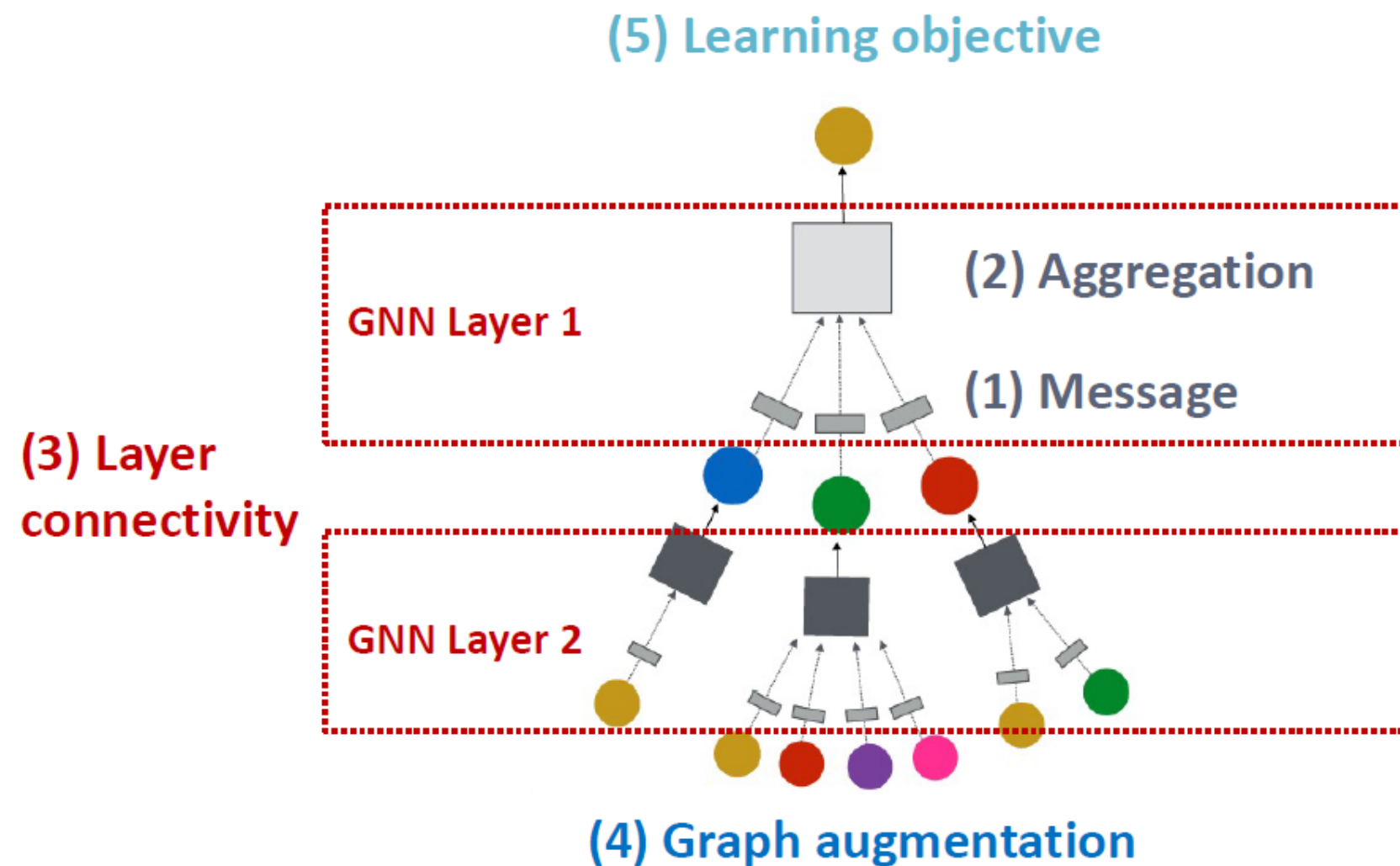




Applications of Graph Neural Networks

최지우, 최하경

A General GNN Framework



1. Message Transformation
2. Message Aggregation
3. Layer Connectivity
4. Graph Augmentation
5. Learning Objective

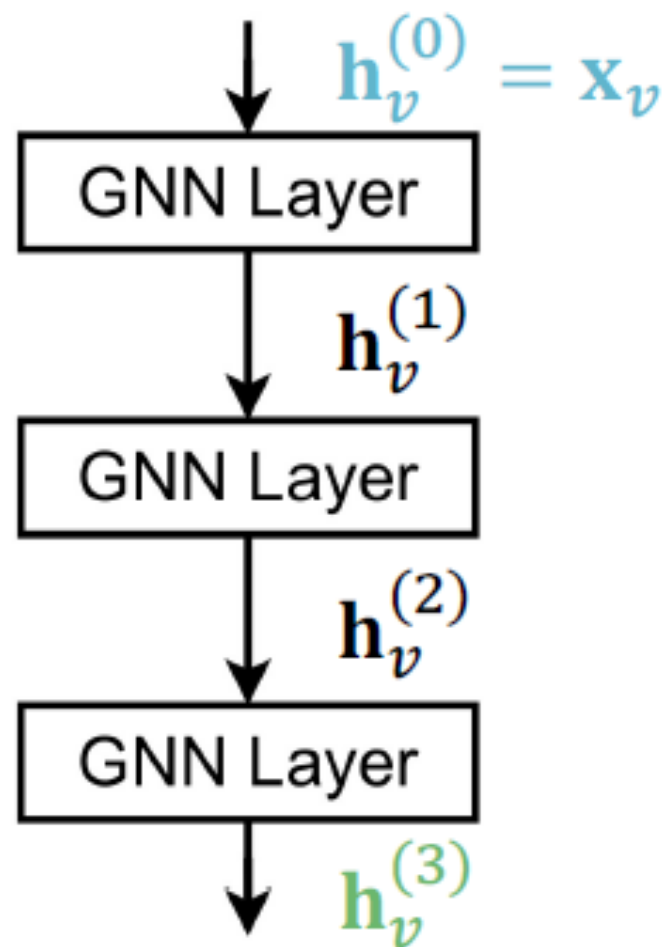
How to connect GNN layers into a GNN?

- Stack layers sequentially
- Ways of adding skip connections

A General GNN Framework

How to connect GNN layers into a GNN?

- Stack layers sequentially
- Ways of adding skip connections



Stacking GNN layers의 문제점

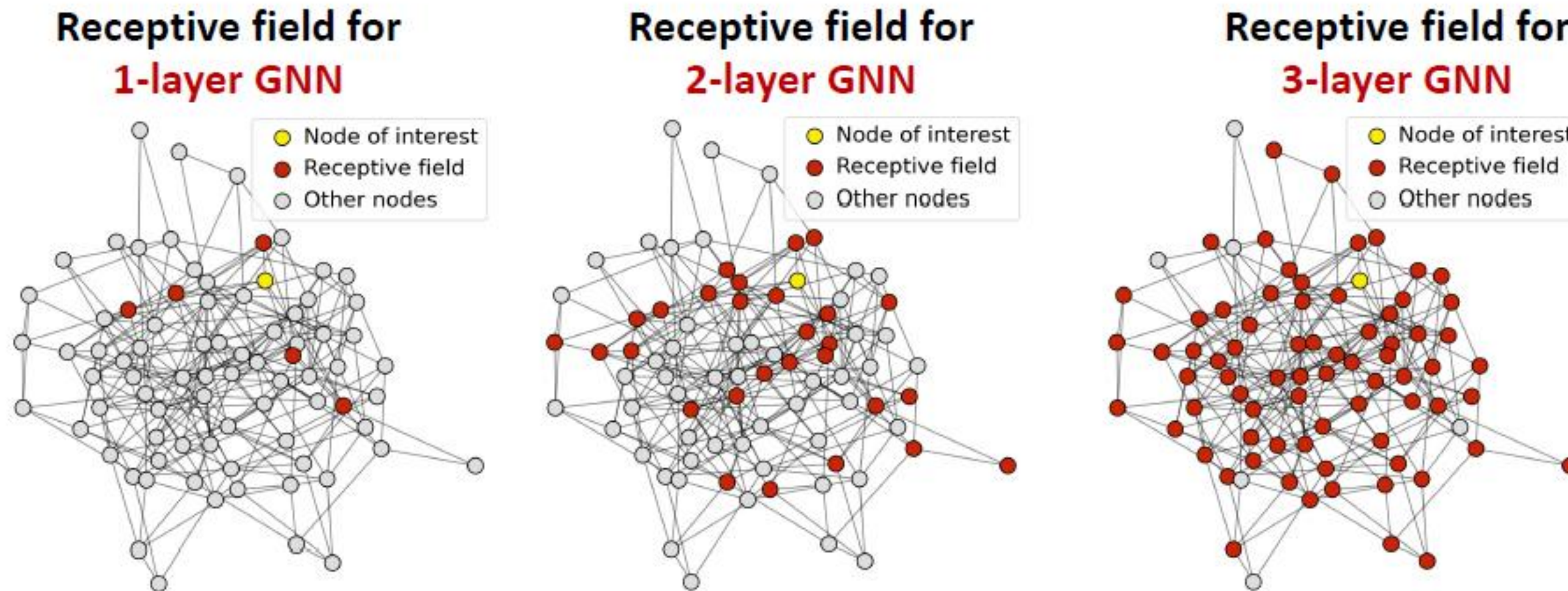
The over-smoothing problem,
모든 임베딩 노드가 같은 값으로
수렴되는 문제.

→ 노드를 구분지으려고 사용한 것이
임베딩 노드인데 사용하는 의미를
잃게 됨.

Receptive Field of a GNN

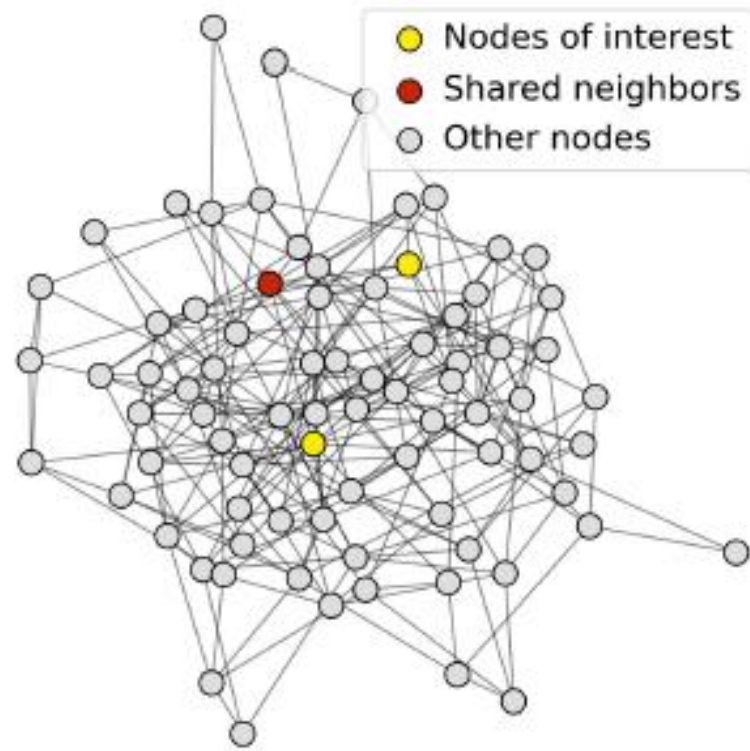
- Receptive field : 관심있는 노드의 embedding을 결정하는 노드의 set

In a K -layer GNN, each node has a receptive field of K -hop neighborhood

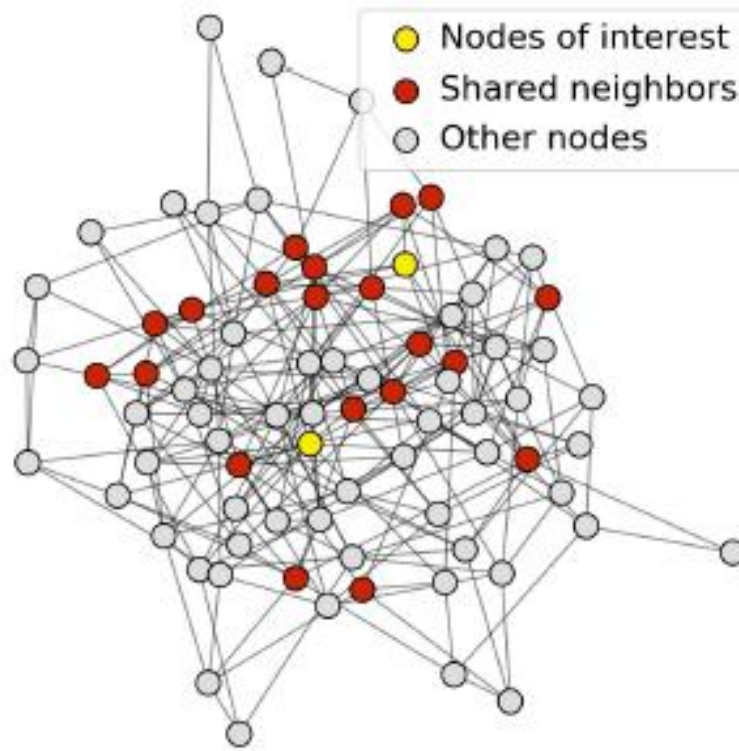


Receptive Field of a GNN

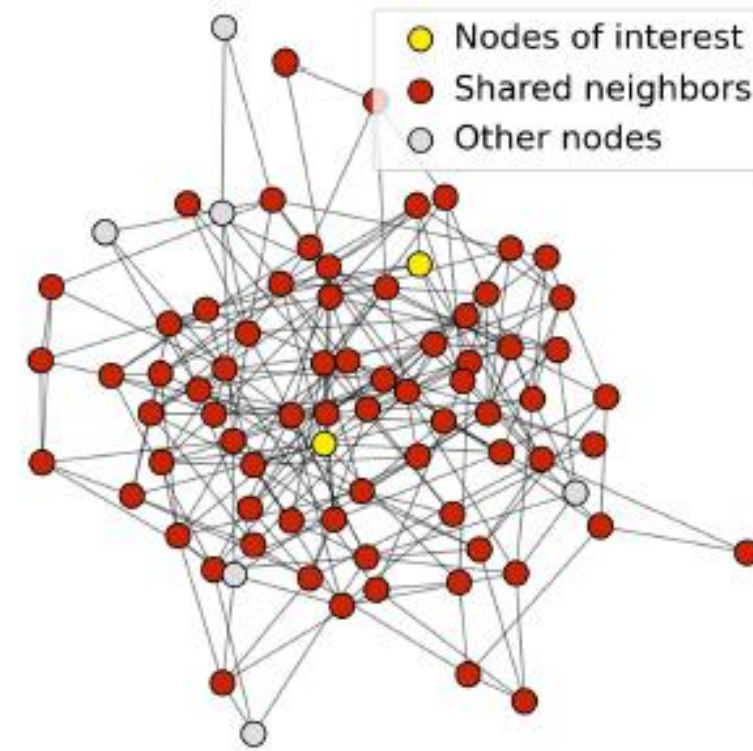
1-hop neighbor overlap
Only 1 node



2-hop neighbor overlap
About 20 nodes



3-hop neighbor overlap
Almost all the nodes!



- Receptive field을 통해 over-smoothing 문제를 발견할 수 있다.
 - Stack many GNN layers → nodes will have highly-overlapped receptive fields → node embeddings will be highly similar → suffer from the over-smoothing problem

Solutions

What do we learn from the over-smoothing problem?

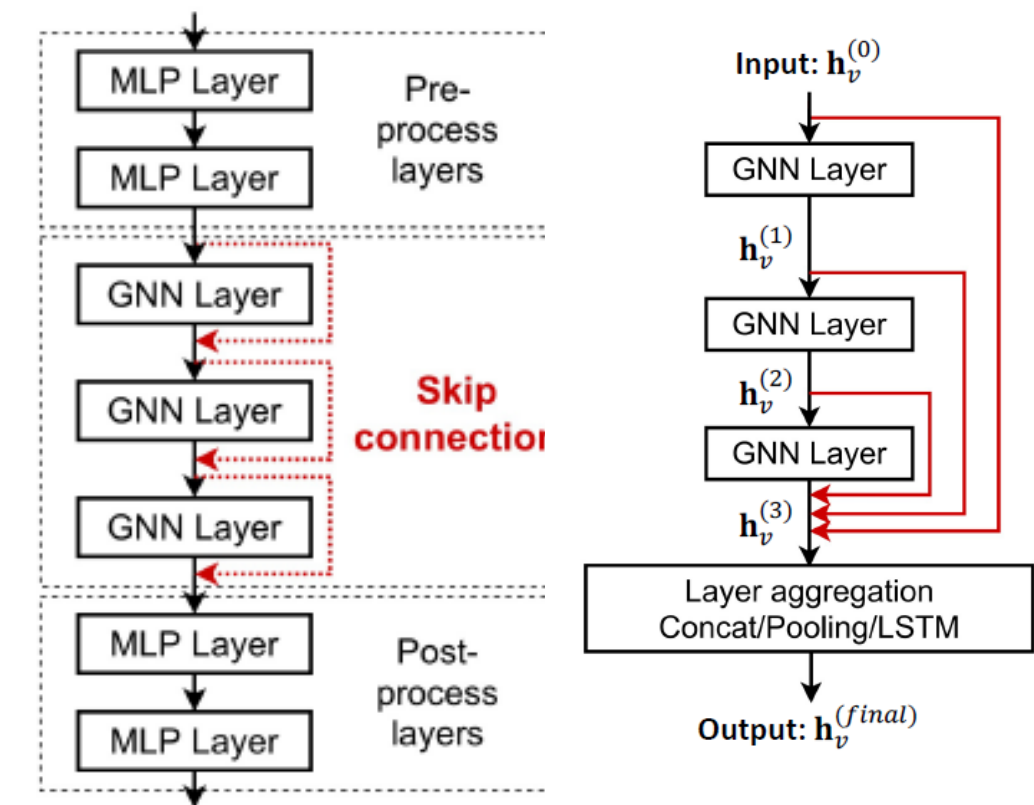
- Be cautious when adding GNN layers.

How to make a shallow GNN more expressive?

- Increase the expressive power within each GNN layer (aggregation, transformation)
- Add layers that do not pass messages

What if my problem still requires many GNN layers?

- Add skip connections in GNNs



Graph Augmentation for GNNs

왜 Graph Augmentation이 필요할까?

- **Graph structure:**

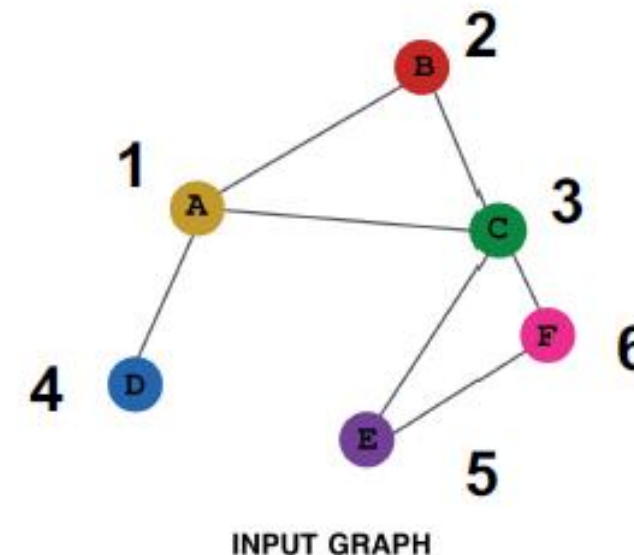
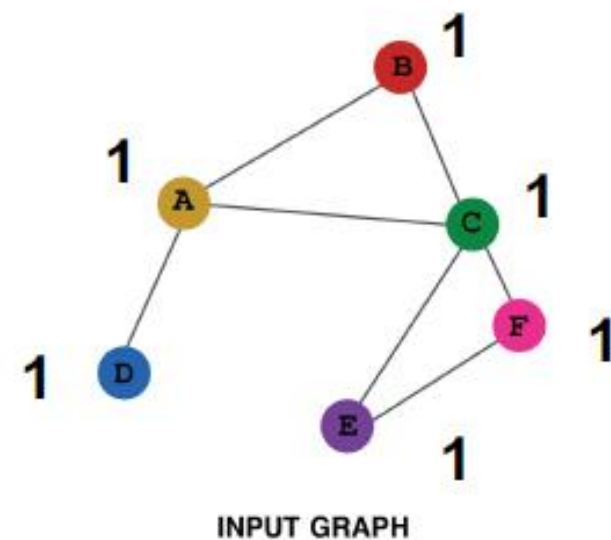
- The graph is **too sparse** → inefficient message passing
- The graph is **too dense** → message passing is too costly
- The graph is **too large** → cannot fit the computational graph into a GPU
- The graph is **too sparse** → **Add virtual nodes / edges**
- The graph is **too dense** → **Sample neighbors when doing message passing**
- The graph is **too large** → **Sample subgraphs to compute embeddings**

Graph Augmentation for GNNs_feature augmentation

왜 Graph Augmentation이 필요할까?

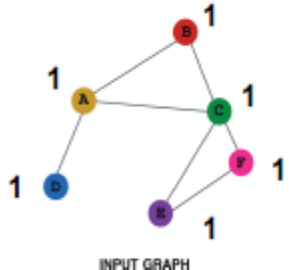
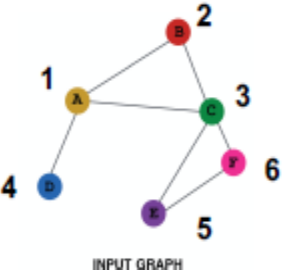
Input graph가 features가 부족한 경우 feature augmentation을 통해 data의 양을 늘릴 수 있다 → 학습에 대한 정확도를 높일 수 있고 임베딩이 효과적으로 생성될 것이다.

대부분 그래프들이 구조는 갖지만 node feature가 없다. GNN은 node feature를 입력값으로 갖기 때문에 feature augmentation이 필요하다. 이때 Constant node feature 과 One-Hot node feature 방식을 들 수 있다.

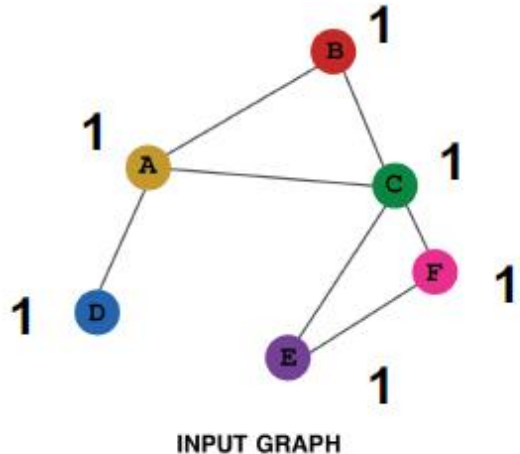


Graph Augmentation for GNNs_feature augmentation

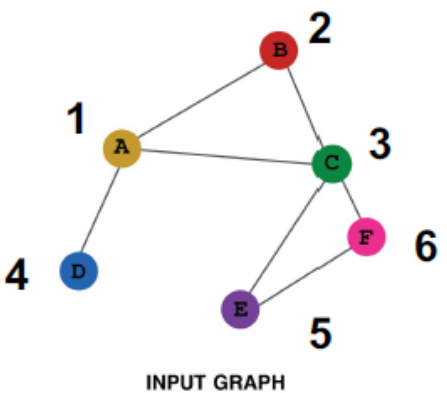
■ Feature augmentation: constant vs. one-hot

	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. $O(V)$ dimensional feature, cannot apply to large graphs
Use cases	Any graph, inductive settings (generalize to new nodes)	Small graph, transductive settings (no new nodes)

Constant



One-Hot



One-hot vector for node with ID=5

One-hot vector for node with ID=5

↓ ID = 5

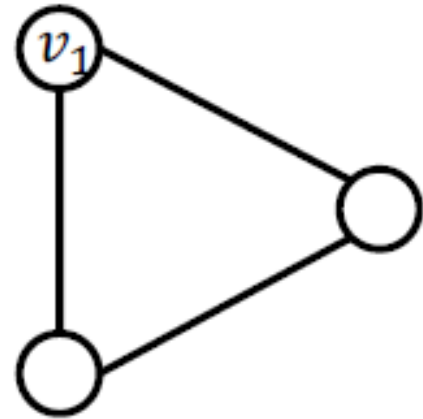
[0, 0, 0, 0, 1, 0]

⏟ Total number of IDs = 6

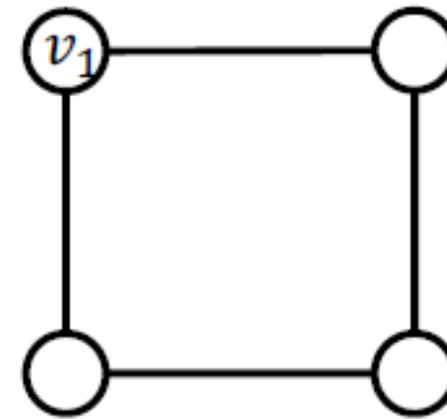
Graph Augmentation for GNNs_feature augmentation

왜 Graph Augmentation이 필요할까?

v_1 resides in a cycle with length 3



v_1 resides in a cycle with length 4



Cycle 구조를 갖고 노드의 개수가 다르지만 GNN은 두 그래프를 구별하지 못한다. GNN은 이웃 노드를 살피기 때문이다. 두 그래프 모두 degree가 2인 노드로만 이뤄져 있기 때문에 구분하지 못함.

→ Augmented node features로서 cycle count를 사용하는 것을 통해 해결한다!

Graph Augmentation for GNNs_feature augmentation

왜 Graph Augmentation이 필요할까?

Cycle 구조를 갖고 노드의 개수가 다르지만 GNN은 두 그래프를 구별하지 못한다. GNN은 이웃 노드를 살펴기 때문이다. 두 그래프 모두 degree가 2인 노드로만 이뤄져 있기 때문에 구분하지 못함.

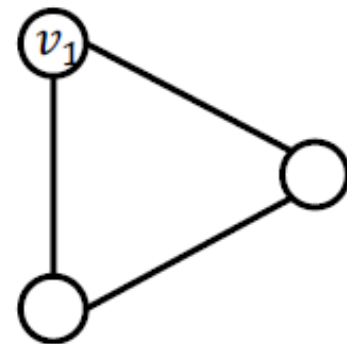
→ Augmented node features로서 cycle count를 사용하는 것을 통해 해결한다!

Augmented node feature for v_1

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



v_1 resides in a cycle with length 3

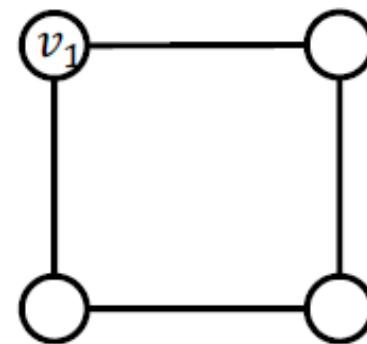


Augmented node feature for v_1

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



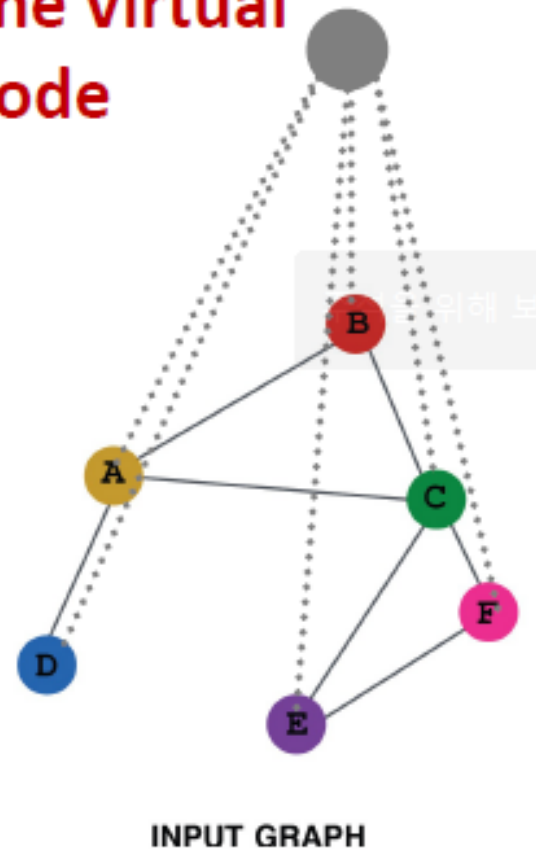
v_1 resides in a cycle with length 4



해당 노드가 속한 cycle의 degree를 one hot vector 로 표현할 수 있다. 이를 통해 GNN은 그래프가 어떤 cycle을 가지고 있는지 파악하고 이를 이용해 임베딩을 할 수 있다. Cycle의 degree 외에도 Node degree, Clustering coefficient, PageRank, Centrality 등이 augmented features로 사용될 수 있다.

Graph Augmentation for GNNs_add virtual Nodes/edges

The virtual node

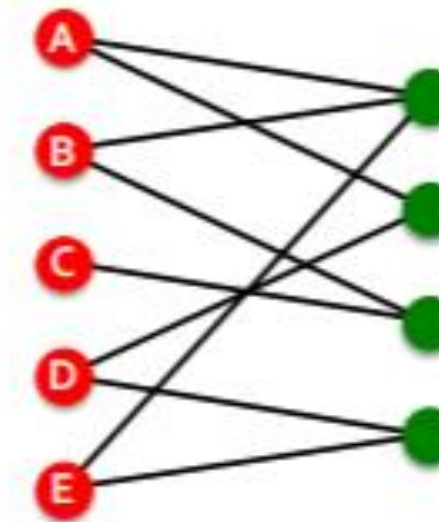


가상의 노드를 활용하는 방법

Use cases: Bipartite graphs

- Author-to-papers (they authored)
- 2-hop virtual edges make an author-author collaboration graph

Authors Papers

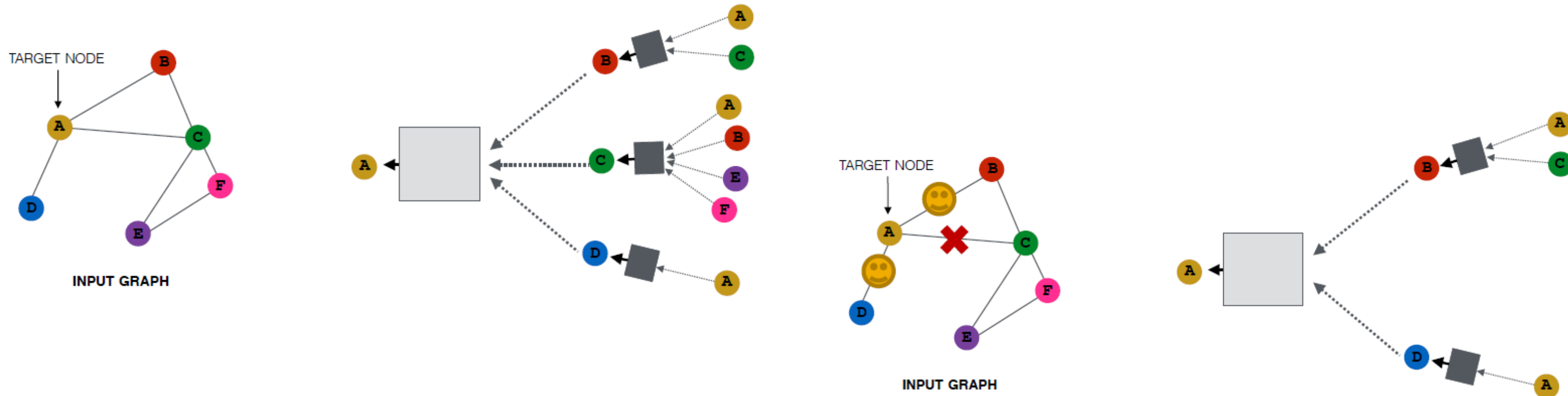


가상의 엣지를 활용하는 방법

그래프가 sparse할 경우에 Add virtual Nodes/Edges 를 통해 그래프의 모든 노드를 연결할 수 있다. 보통 2-hop neighbor를 이용하는 것이 흔한 방법이다.

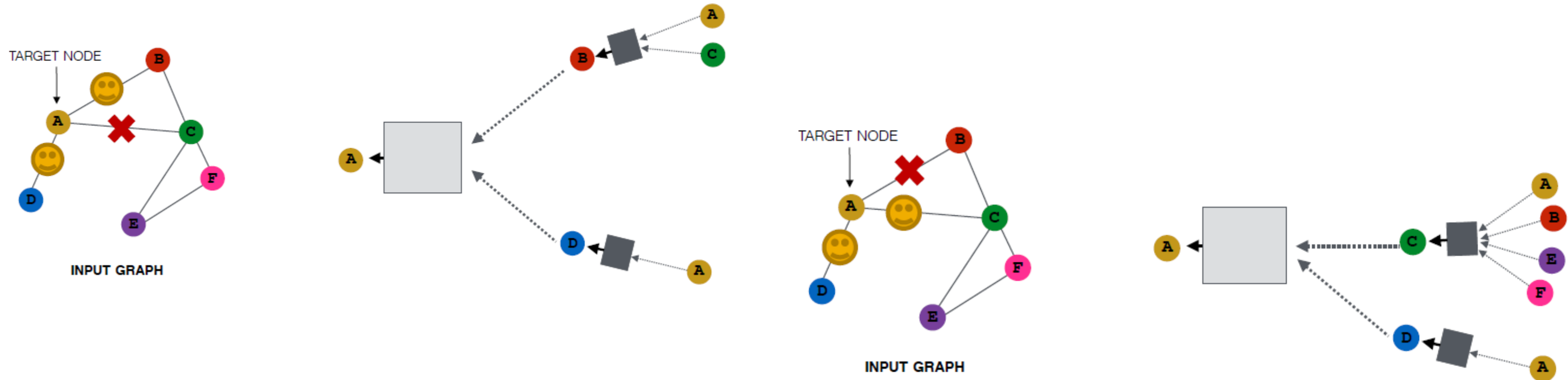
가상 노드를 이용하면 모든 노드가 2-hop으로 연결된다. 노드 끼리 거리가 멀어도 서로 관련된 message passing 구조를 가질 수 있다.

Graph Augmentation for GNNs_Node Neighborhood Sampling



그래프가 dense할 경우 연산량을 줄이기 위해 모든 노드를 사용하지 않고 sampling을 이용할 수 있다. 랜덤 sampling 해서 계산 그래프를 구성할 수 있다.

Graph Augmentation for GNNs_Node Neighborhood Sampling

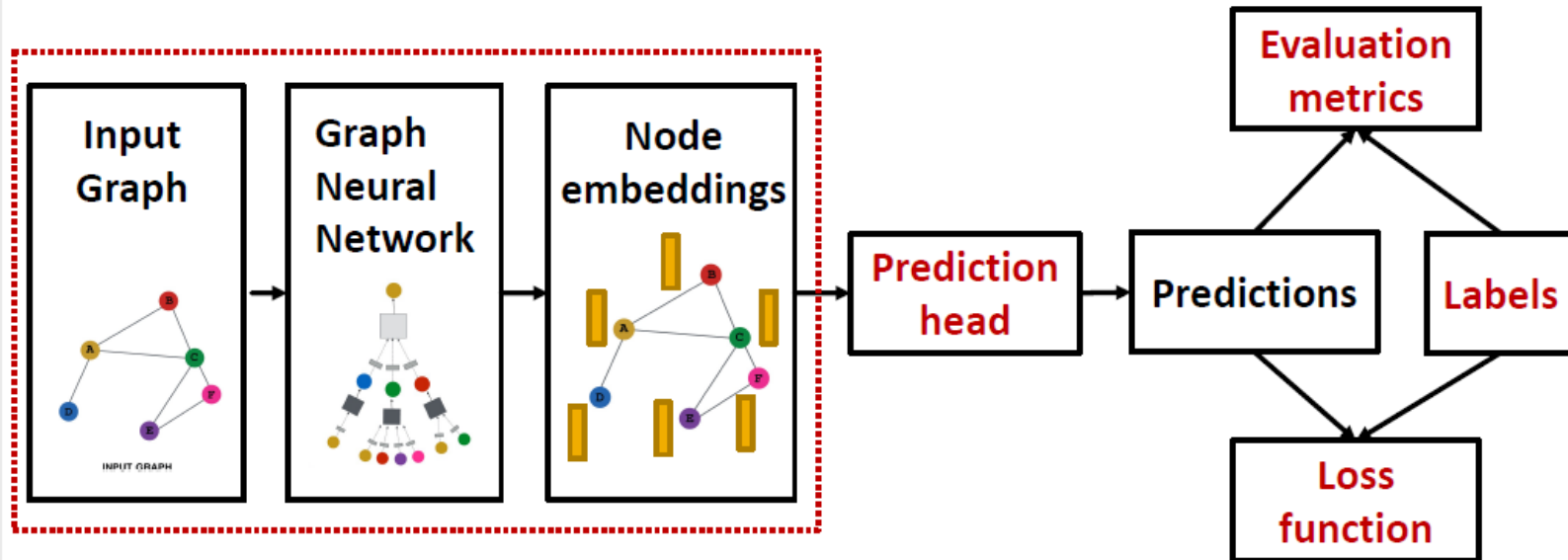


Epoch, step, layer마다 샘플링을 고정하지 않고, 다르게 반복하면 동일한 노드 임베딩을 생성하는 계산 그래프가 다르게 생성될 수 있다.

Neighborhood Sampling을 통해 계산량을 줄이고, 큰 그래프에 GNN을 적용할 수 있거나 모델의 표현력을 높힐 수 있는 장점을 가진다.

GNN Training Pipeline

So far what we have covered



Output of a GNN: set of node embeddings

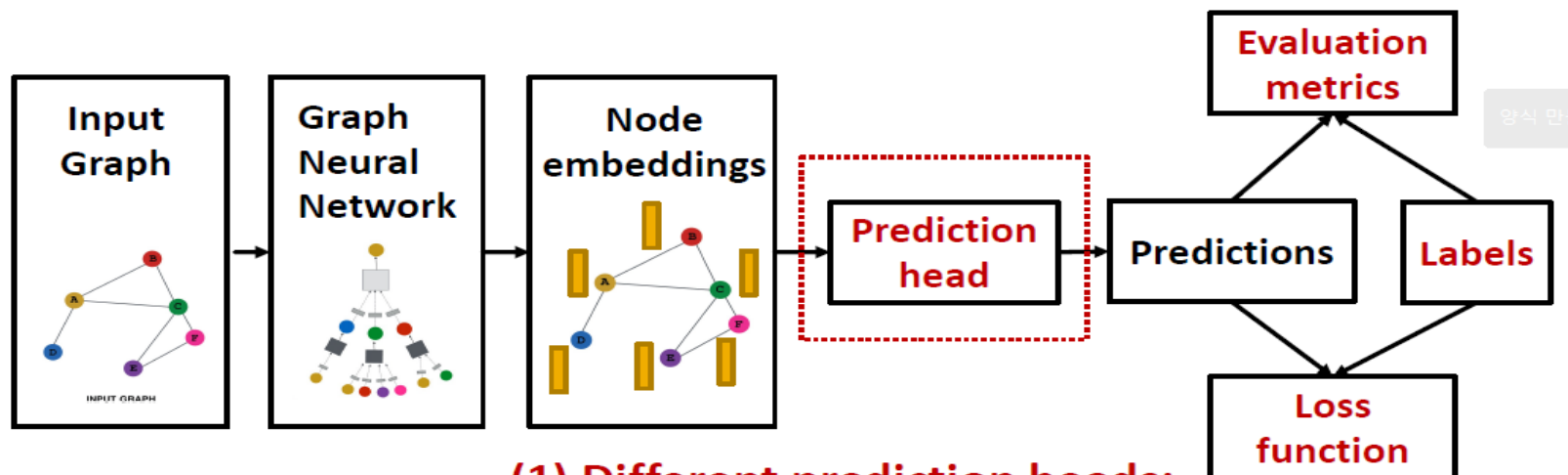
$$\{\mathbf{h}_v^{(L)}, \forall v \in G\}$$

실제 GNN의 학습은 GNN을 통과하여 생성된 노드 임베딩을 이용해 task를 수행한다. Loss를 계산하여 역전파가 이루어져야 학습이 가능하다.

GNN, Graph augmentation 등은 예측을 위해 임베딩 벡터를 생성하는 과정에 해당한다

Prediction head로 생성된 노드 임베딩 벡터를 활용해서

Node-level tasks
Edge-level-tasks
Graph-level-tasks 를 수행할 수 있다.



(1) Different prediction heads:

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

Prediction Heads : Node-level

GNN을 통해 얻은 노드 임베딩은 다음과 같이 표현할 수 있다.

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

분류와 회귀 task로 나눌 수 있고 다음과 같은 식으로 표현할 수 있다.

$$\hat{\mathbf{y}}_v = \text{Head}_{\text{node}}(\mathbf{h}_v^{(L)}) = \mathbf{W}^{(H)} \mathbf{h}_v^{(L)}$$

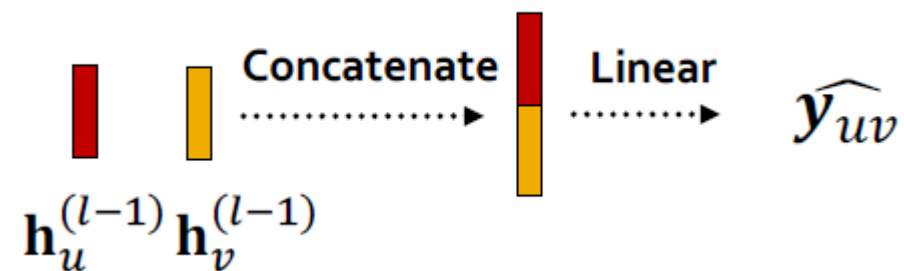
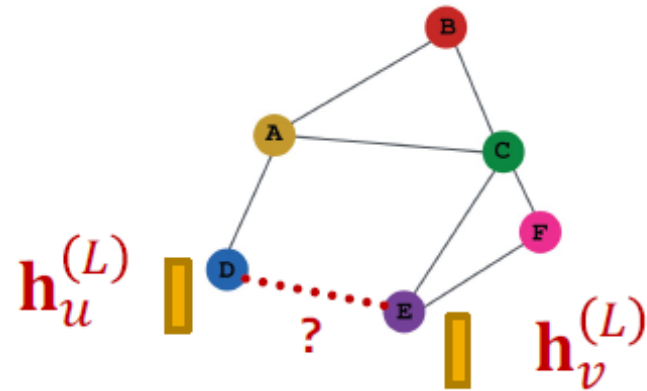
L = GNN의 layer 번호 W: loss를 계산하기 위해서 h를 y로 임베딩 노드를 매핑할 수 있다.

마지막 layer의 출력을 활용하는 것이다.

Node level task는 node 하나에 대한 예측을 하기 때문에 한 노드의 임베딩 벡터만 입력값으로 받는 것을 알 수 있다.

Prediction Heads : Edge-level

$$\hat{\mathbf{y}}_{uv} = \text{Head}_{\text{edge}}(\mathbf{h}_u^{(L)}, \mathbf{h}_v^{(L)})$$



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

...

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

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

Edge-level tasks를 노드 임베딩을 이용해 사용할 때 Head를 구성해야 한다.

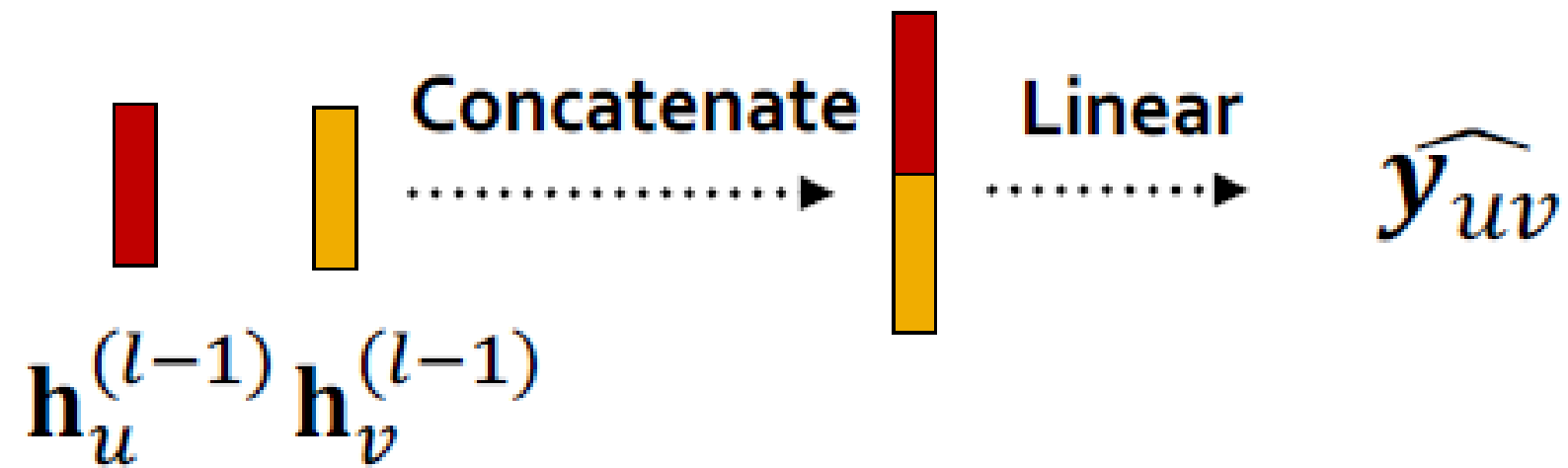
Edge를 예측하기 위해선 두 노드의 임베딩 벡터를 입력값으로 하는 Head를 구성해야 함.

-Concatenation + Linear

-Dot Product

이용할 수 있다.

Prediction Heads : Edge-level



Concatenation + Linear

가장 단순한 방법으로는 두 임베딩 벡터를 concat하고 선형 변환을 하는 것이다.
2*d 차원의 벡터를 k차원 벡터 혹은 스칼라로 맵핑하는 선형함수를 구성하는 것과 같다.

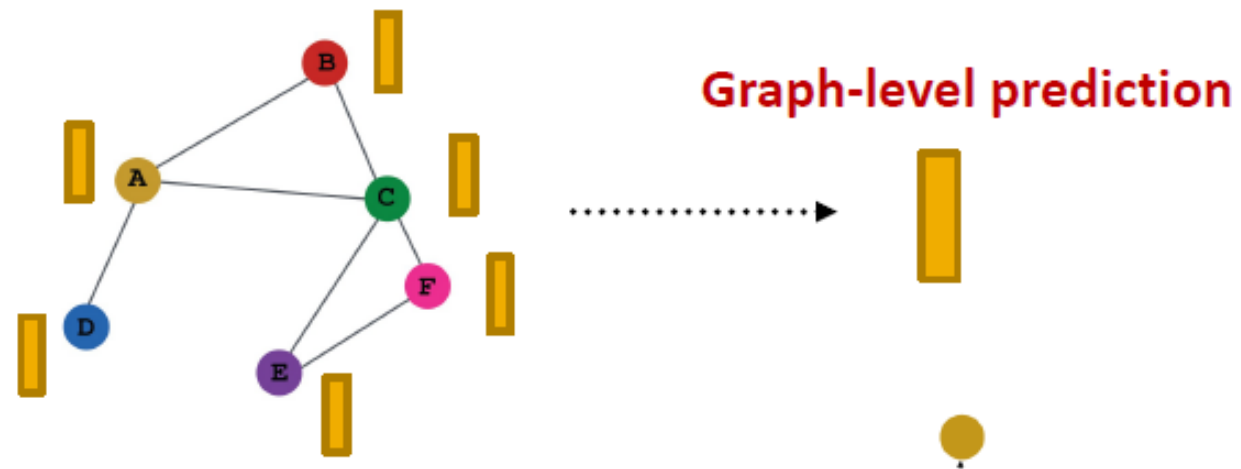
Dot product

Concat 방법은 undirected graph일 때 노드 순서에 따라 다른 값이 나오기 때문에 문제가 된다.(undirected graph은 입력 순서는 예측에 영향을 끼쳐선 안된다.)

임베딩 벡터를 내적하는 것으로 해결한다.
다중 레이블에 대한 분류일 때 선형 변환을 추가하여 구성할 수 있다.

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

Prediction Heads : Graph-level



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

(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

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

모든 노드를 이용해 예측하는 것이다.

Head는 그래프 내 노드들의 정보를 종합한다는 점에서 GNN layer의 aggregation에 사용하는 것과 similar한 역할을 한다.

상황마다 pooling을 다르게 쓰는데
Size의 차이를 무시하고 싶으면 mean pooling

노드의 개수, 그래프 구조를 알고 싶으면 sum pooling

상황에 다르게 사용한다.

Global Pooling Problems

- 작은 그래프에서는 성능이 잘 나오지만, 큰 그래프에서는 성능이 좋지 않음 -> Global Pooling의 근본적인 문제점
- 예) 1차원으로 이루어진 노드 임베딩을 갖는 그래프 2개
 - Node embeddings for $G_1: \{-1, -2, 0, 1, 2\}$
 - Node embeddings for $G_2: \{-10, -20, 0, 10, 20\}$
- 두 그래프에 대해 global pooling 진행 시
 - **Prediction for G_1 :** $\hat{y}_G = \text{Sum}(\{-1, -2, 0, 1, 2\}) = 0$
 - **Prediction for G_2 :** $\hat{y}_G = \text{Sum}(\{-10, -20, 0, 10, 20\}) = 0$
- 두 그래프를 구분할 수 없게 됨
- 노드가 많아질수록 각 노드의 특징이 전체 그래프 임베딩에서 차지하는 비중이 적어지기 때문에 발생하는 문제

Hierarchical Global Pooling

- Solution of global pooling problems

- **Toy example:** We will aggregate via $\text{ReLU}(\text{Sum}(\cdot))$
 - 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:** $\hat{y}_a = \text{ReLU}(\text{Sum}(\{-1, -2\})) = 0$, $\hat{y}_b = \text{ReLU}(\text{Sum}(\{0, 1, 2\})) = 3$
 - **Round 2:** $\hat{y}_G = \text{ReLU}(\text{Sum}(\{y_a, y_b\})) = 3$
- G_2 node embeddings: $\{-10, -20, 0, 10, 20\}$
 - **Round 1:** $\hat{y}_a = \text{ReLU}(\text{Sum}(\{-10, -20\})) = 0$, $\hat{y}_b = \text{ReLU}(\text{Sum}(\{0, 10, 20\})) = 30$
 - **Round 2:** $\hat{y}_G = \text{ReLU}(\text{Sum}(\{y_a, y_b\})) = 30$

Now we can
differentiate
 G_1 and G_2 !

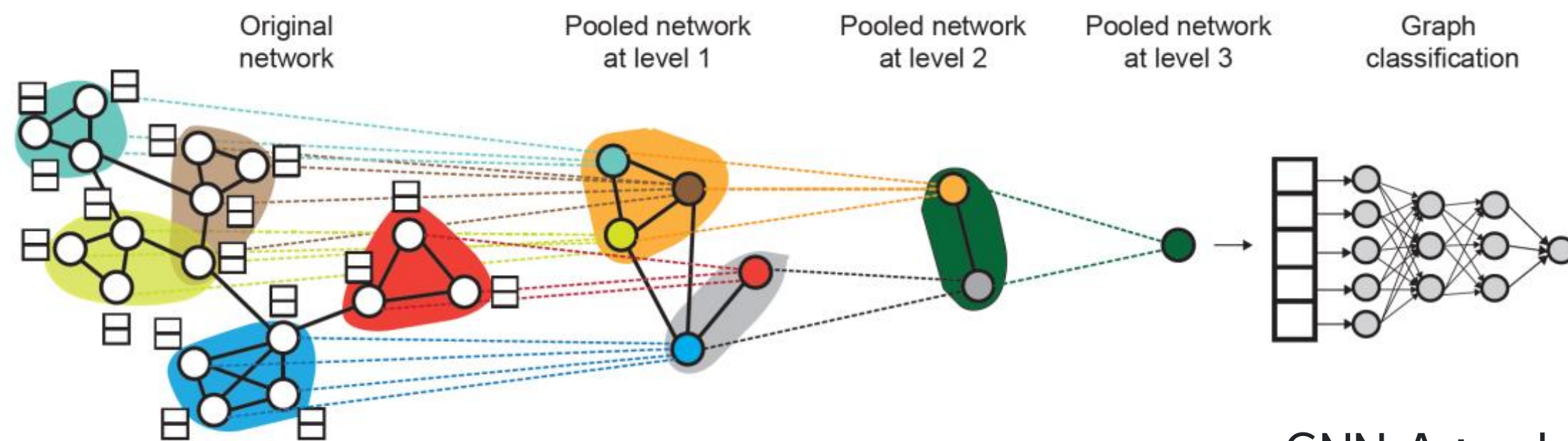
- 일부 노드를 모아 pooling > 해당 노드들에 대한 임베딩 벡터 생성
- 이 벡터들을 다시 pooling
- 이때, 비선형 함수를 추가해 global pooling과는 다른 결과를 가지도록 함

- pooling할 일부 노드를 선택하는 것 == 일종의 sub-graph에 대해 임베딩
- 지역적으로 모여져 있는 노드들을 하나의 sub graph로 간주하고 이들을 단계적으로 pooling하는 것이 효과적

Hierarchical Pooling in Practice

- DiffPool

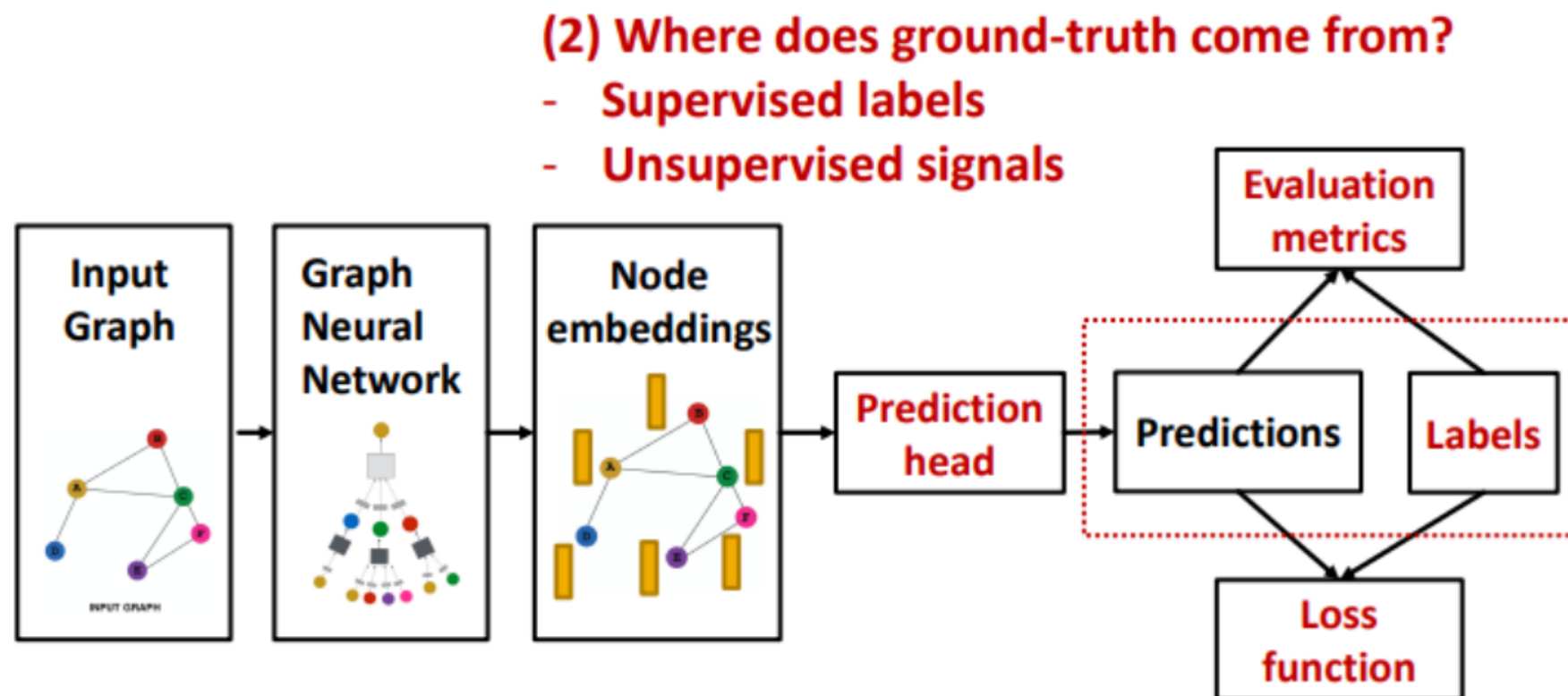
- Hierarchical pooling 시 subgraph를 어떻게 구성할 수 있을지에 대한 고민으로 나온 방법
- 두 가지 GNN을 활용해 매 pooling level마다 독립적인 두 작업을 진행



- GNN A : subgraph에 대한 임베딩 벡터 생성
- GNN B : 어떤 노드를 클러스터링할 지 계산
- 매 레벨마다 병렬적으로 진행되어, 효과적으로 임베딩하고 클러스터링할 방법을 찾음

GNN Training Pipeline (2)

- 이번에 살펴볼 것 : Prediction head를 통과한 예측값과 레이블



- 지도 학습 : 외부에 출처를 두고 있는 레이블을 가지는 경우 (ex. 분자 구조에 대한 약효 레이블)
- 비지도 학습 : 그래프 스스로 레이블을 갖고 있는 경우 (ex. 두 노드의 엣지 존재 유무)

- 더 이상 지도, 비지도 학습의 구분은 레이블의 존재에 있지 않음
- 비지도 학습이라 하더라도 데이터 내에서 레이블로 삼을 수 있는 요소를 이용해 일종의 self supervised learning을 수행하게 됨

Labels on Graph

- 지도 학습의 레이블

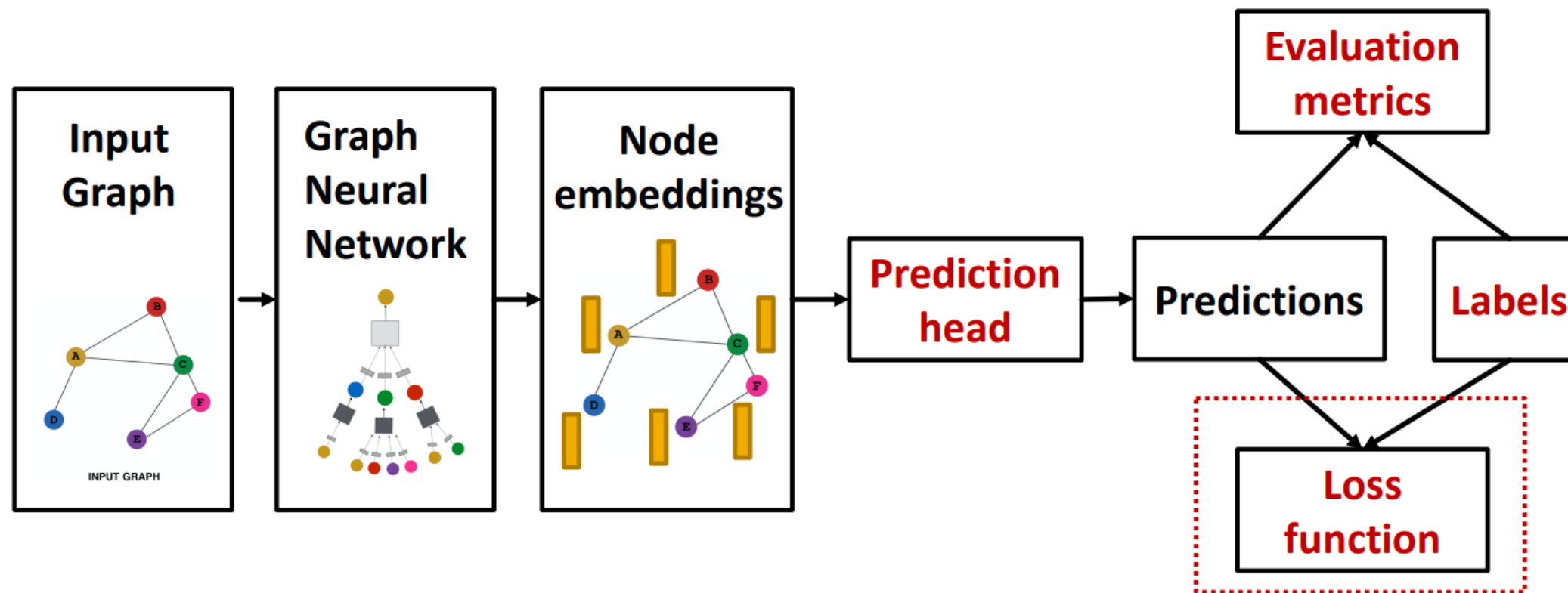
- Node labels : 논문 인용 네트워크에서 해당 논문이 갖고 있는 연구 분야
- Edge labels : 거래 내역 네트워크에서 가짜 거래 여부
- Graph labels : 분자 구조 네트워크에서 해당 네트워크의 약효
- 일반적으로 지도학습이 설계하거나 학습시키기 수월한 방법론들이 개발되었기 때문에, label이 존재한다면 지도학습을 추천

- 비지도 학습의 레이블

- 비지도 학습의 문제점 : 레이블이 없어 학습의 방법이 없음 > self supervised learning
- 그래프에서 지도학습으로 사용할 만한 요소들을 찾는 것
 - **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

GNN Training Pipeline (3&4)

- 이번에 살펴볼 것 : classification vs regression



(3) How do we compute the final loss?

- **Classification loss**
- **Regression loss**

- 차이점 : loss function & evaluation metrics

Loss Function

- Classification Loss

- As discussed in lecture 6, **cross entropy (CE)** is a very common loss function in classification

- **K -way prediction** for i -th data point:

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

i -th data point

where:

E.g.

0	0	1	0	0
---	---	---	---	---

 $\mathbf{y}^{(i)} \in \mathbb{R}^K$ = one-hot label encoding
 $\hat{\mathbf{y}}^{(i)} \in \mathbb{R}^K$ = prediction after Softmax(\cdot)
E.g.

0.1	0.3	0.4	0.1	0.1
-----	-----	-----	-----	-----

- **Total loss over all N training examples**

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

- Regression Loss

- For regression tasks we often use **Mean Squared Error (MSE)** a.k.a. **L2 loss**

- **K -way regression** for data point (i):

$$\text{MSE}(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}) = \sum_{j=1}^K (\underbrace{y_j^{(i)}}_{j\text{-th target}} - \underbrace{\hat{y}_j^{(i)}}_{j\text{-th target}})^2$$

i -th data point

where:

E.g.

1.4	2.3	1.0	0.5	0.6
-----	-----	-----	-----	-----

 $\mathbf{y}^{(i)} \in \mathbb{R}^k$ = Real valued vector of targets
 $\hat{\mathbf{y}}^{(i)} \in \mathbb{R}^k$ = Real valued vector of predictions
E.g.

0.9	2.8	2.0	0.3	0.8
-----	-----	-----	-----	-----

- **Total loss over all N training examples**

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

- 전체 노드에 대해 실제값보다는 노드의 크기 순서가 중요할 때
MSE 사용

Evaluation Metrics

- Regression

■ Root mean square error (RMSE)

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

■ Mean absolute error (MAE)

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

- Classification

- Multi-class : accuracy
- Binary classification

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

■ Accuracy:

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

■ Precision (P):

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

■ Recall (R):

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

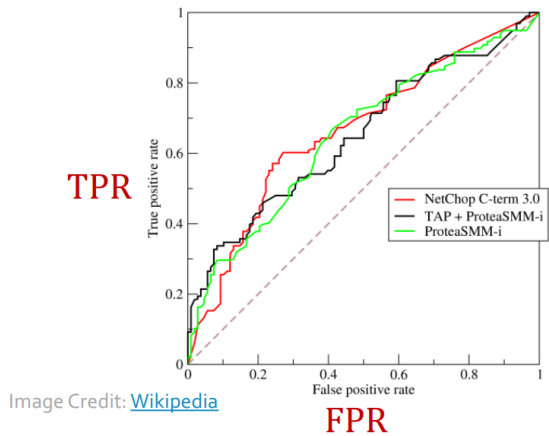
■ F1-Score:

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

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)

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



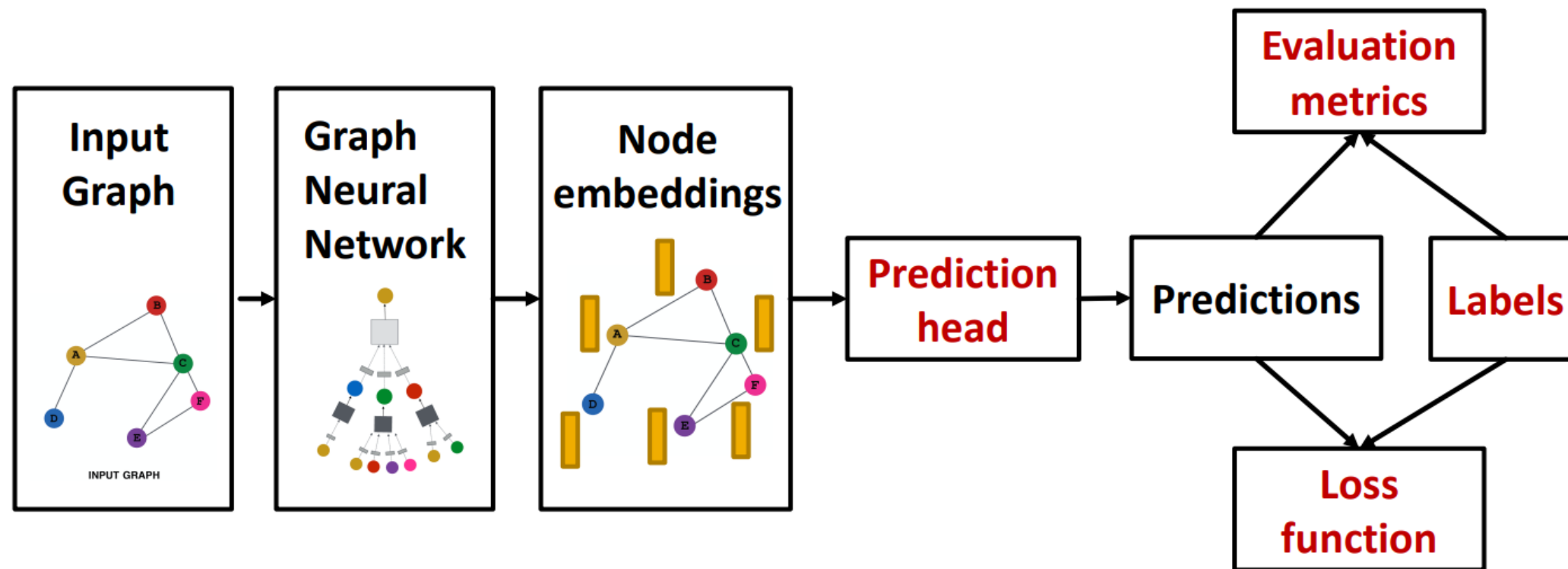
$$TPR = \text{Recall} = \frac{TP}{TP + FN}$$

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

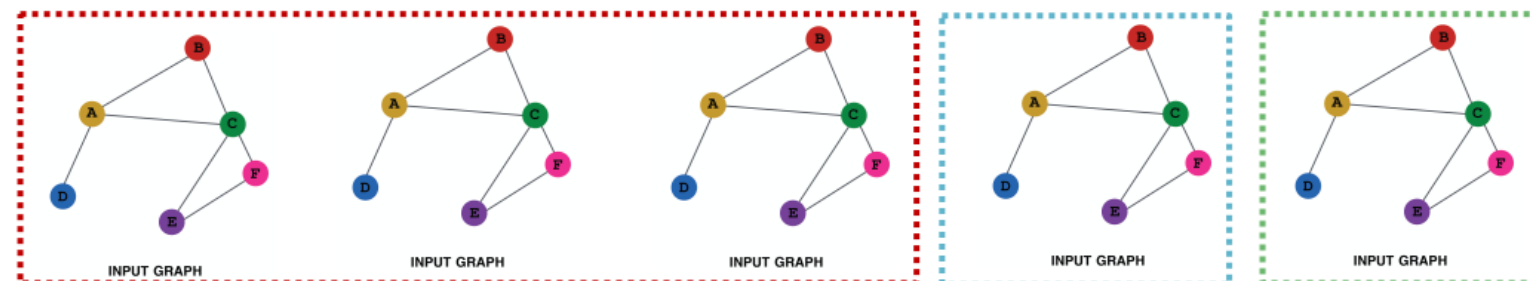
Note: the dashed line represents performance of a random classifier

GNN Training Pipeline (5)

- 이번에 살펴볼 것 : train / validation / test set 분리



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



Dataset split

- 차이점 : loss function & evaluation metrics

Data Splitting

- Fixed Split
 - 흔히 알고 있는 데이터 분리 방법
 - 하나의 데이터셋을 train, validation, test set으로 분리시키고 고정해 사용
 - 되도록 모집단의 분포와 유사하도록 층화 추출 등을 사용 but 여전히 각 데이터셋이 편향될 위험 있음
- Random Split
 - Kfold와 같이 여러 번 랜덤하게 데이터 추출
 - 동일한 데이터셋에서 시드를 바꿔가며 train, validation, test 데이터셋을 추출하고, 각 시드에 대한 모델 성능의 평균을 구해 최종 성능을 계산함

Special Point of Splitting Graph

- 문제점

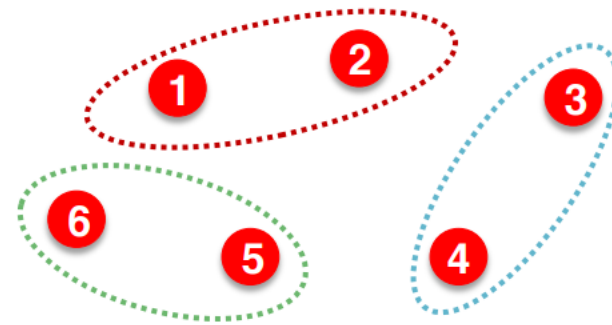
- 그래프는 각 노드들이 연결되어 있는 구조 > node classification : data points are not independent!

- **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



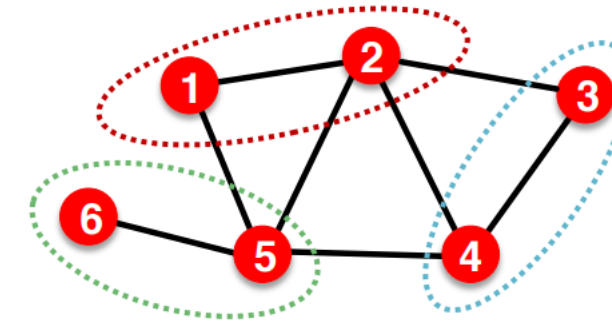
- **Splitting a graph dataset is different!**

- **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

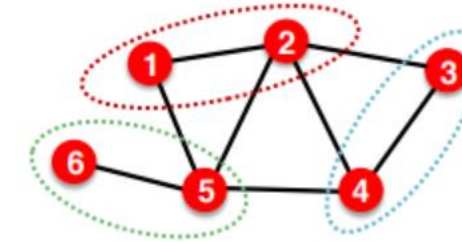


Special Point of Splitting Graph

- Solution 1 : Transductive setting

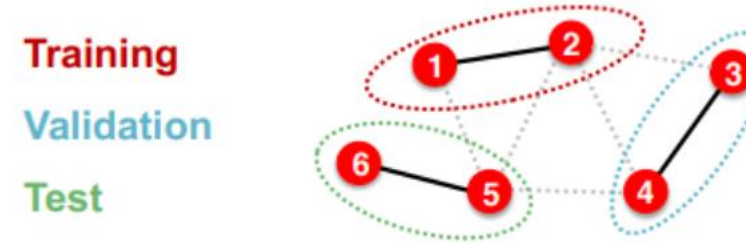
- 하나의 그래프에서 단순히 노드를 추출하는 방법 사용
- 서로 연결된 노드끼리 다른 데이터셋을 구성
- 학습 시 모든 그래프에 대해 임베딩을 생성하지만, 역전파를 위해서는 train dataset에 해당하는 노드만 사용
- Validation, test 시 동일하게 모든 그래프에 대한 임베딩을 생성하고, 해당 데이터셋에 있는 노드에 대해서만 성능을 확인함
- 그래프 구조는 각 노드가 다른 노드로부터 정보를 받아 GNN을 통과해야 완전한 학습, 추론 상황이라고 할 수 있기 때문에, 이러한 방법을 사용
- But 그래프 단위의 task에서는 사용할 수 없음

Training
Validation
Test



Special Point of Splitting Graph

- Solution 2 : Inductive Setting



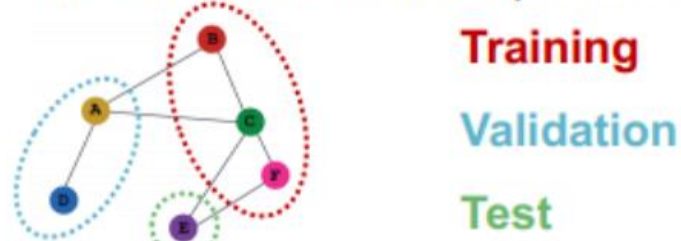
- 데이터셋 간의 독립성 강조
 - 각 데이터셋끼리의 edge는 지우고 train, validation, test 진행
 - (장점) 데이터셋을 완벽하게 분리해 독립성을 높일 수 있음
 - (단점) 학습과 추론이 실제 GNN 모델의 성능과 동일하지 않을 수 있음
 - 만약 여러 그래프를 이용해 학습을 진행하는 상황이라면, 위와 같은 우려 줄어들기도 함
- Transductive Setting vs Inductive Setting
 - transductive setting
데이터셋이 같은 graph 상에 있음. Label만 split한 것! -> node, edge classification만 가능
 - Inductive setting
데이터셋이 다른 graph 상에 있음. 성공적인 모델은 unseen graph들에 대해 generalize되어야 함
Node, edge, graph level task 모두 가능

Node & Graph Classification

- Node Classification

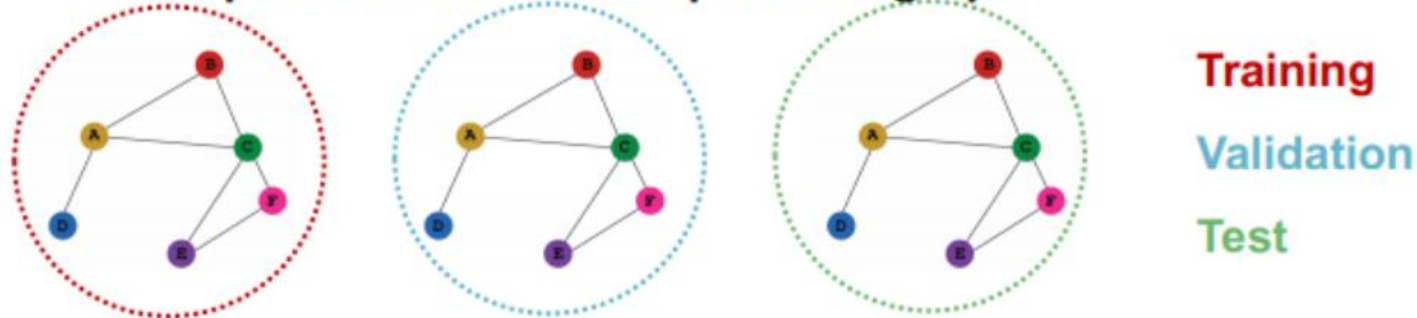
■ Transductive node classification

- 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

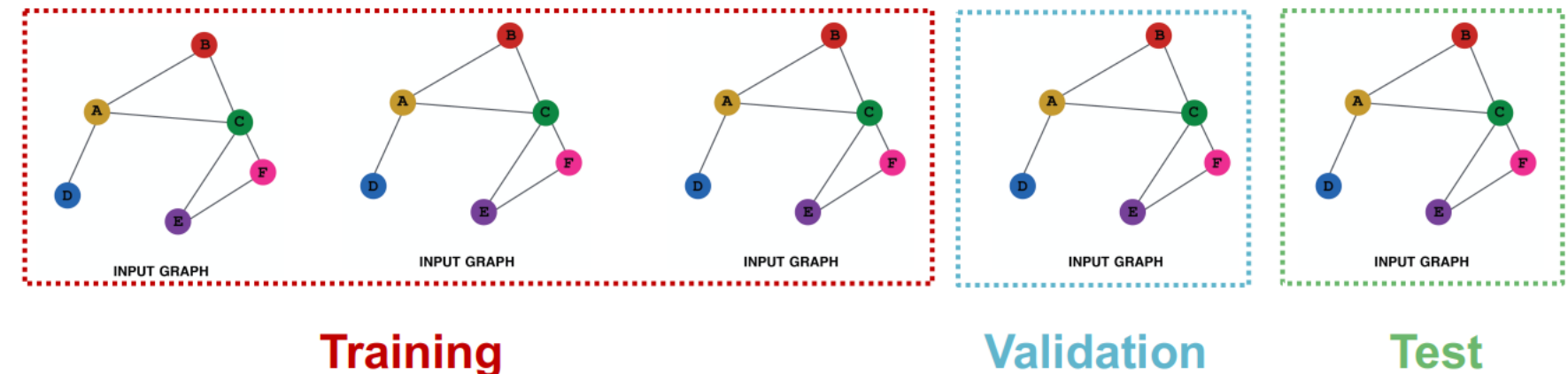


- transductive, inductive setting 모두 가능

- 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).

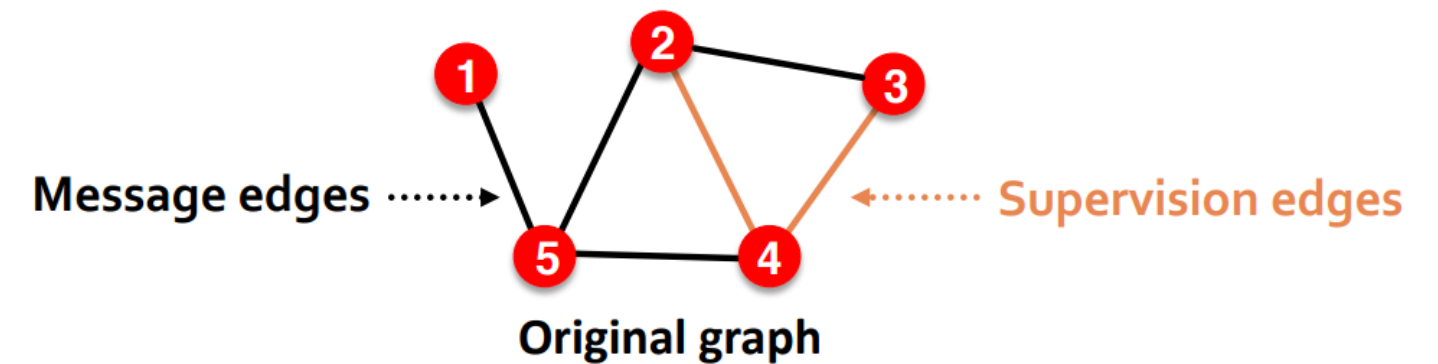


- inductive setting만 가능

- 이때, 각 그래프는 독립적이라고 가정

Link Prediction

- Goal : predict missing edges
 - Label이 없기 때문에, self supervised learning 사용해야 함 > 이미 존재하는 일부 edge들을 제거하고 이를 맞추는 식으로 학습과 평가 진행
 - 기존의 edge들을 2가지로 분리
 - Message edges : 실제 message passing 시 사용
 - Supervision edges : 예측에 사용
- > supervision edge들은 예측에 사용되기 때문에, GNN에 포함되지 않음
(그래프에 message edge들만 존재)

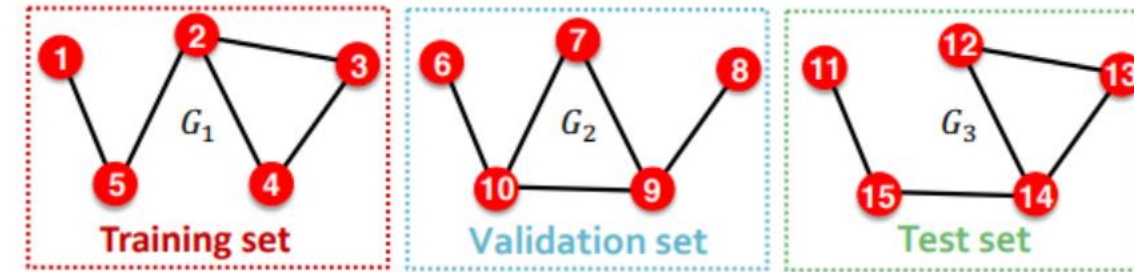


Link Prediction

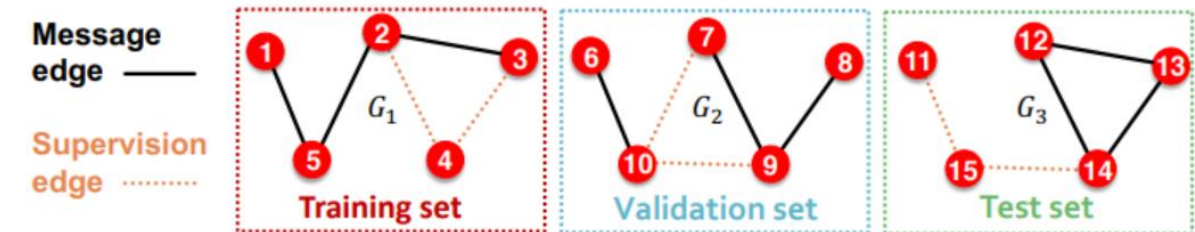
- Goal : predict missing edges
 - Train / validation / test set 분리
 - Inductive setting

개별적인 그래프를 다른 데이터셋으로 분리

모든 message edge로 모델 학습하고, supervision edge로 loss function, 역전파, 평가지표 계산



- Transductive setting
 - 그래프가 하나라면, supervision edge만 각 데이터셋에 따라 분리
 - 학습에 사용된 supervision edge가 validation과 test에서 message edge로 전환, validation에 사용된 supervision edge는 test에서 message edge로 전환되어 사용



- 시간에 따라 데이터셋을 분리한 것이라고 할 수 있음