

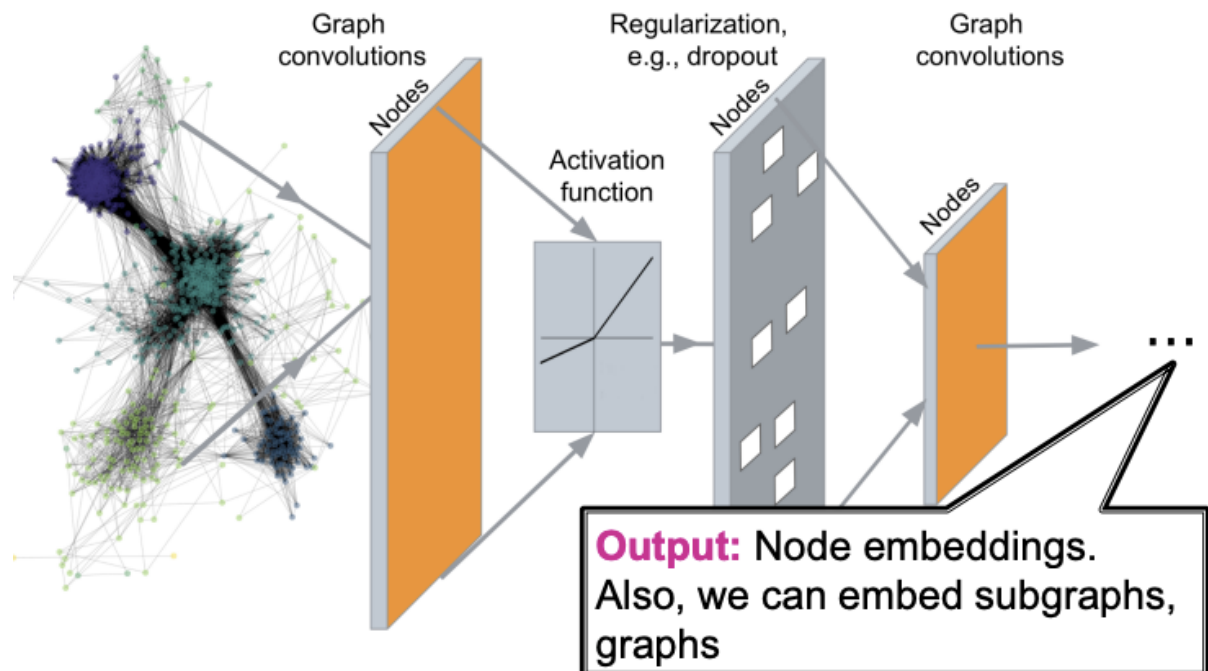
7. A General Perspective on Graph Neural Networks

Contents

1. Recap
 2. A General GNN Framework
 3. A Single Layer of a GNN
 4. Stacking Layers of a GNN
-

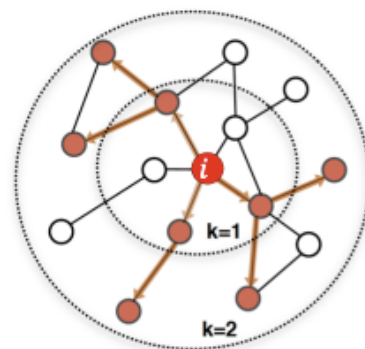
1. Recap

Recap : Deep Graph Encoders

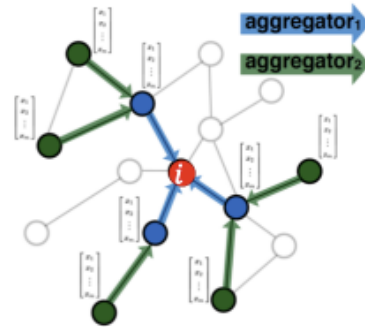


Recap : Graph Neural Networks

Idea: Node's neighborhood defines a computation graph



Determine node computation graph

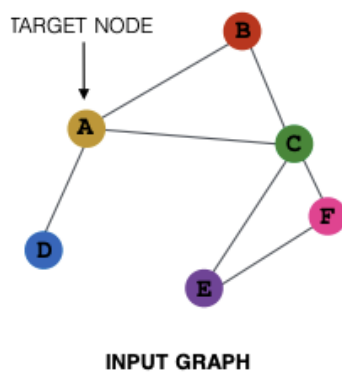


Propagate and transform information

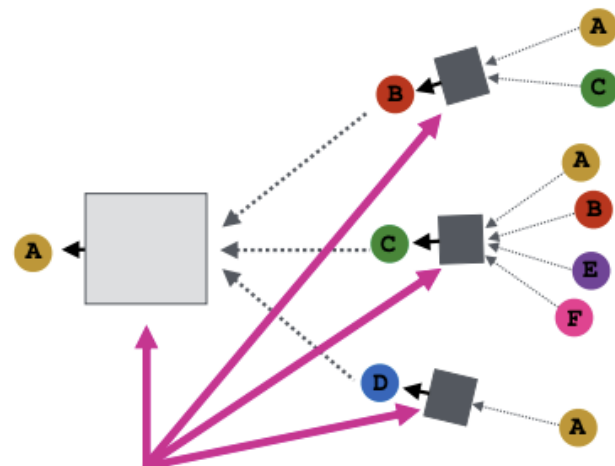
Learn how to propagate information across the graph to compute node features

Recap : Aggregate from Neighbors

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



INPUT GRAPH

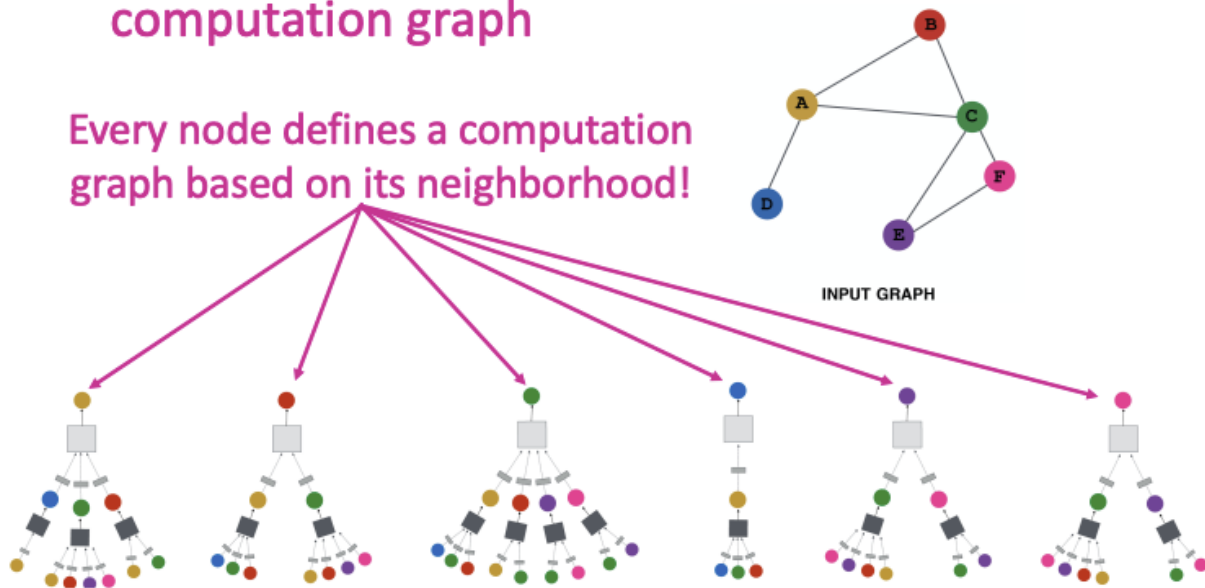


Neural networks

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

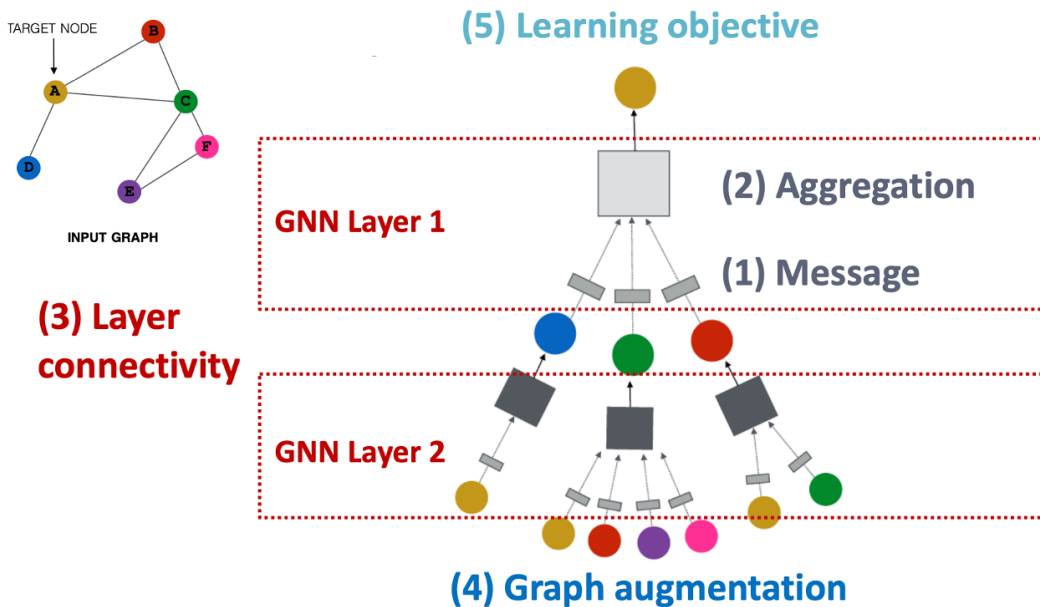
- **Intuition:** Network neighborhood defines a computation graph

Every node defines a computation graph based on its neighborhood!



2. A General GNN Framework

A General GNN Framework



위 그림과 같이 일반적인 GNN의 framework는 다음과 같고, GNN을 디자인하는 방법을 총 5가지로 나눠서 볼 수 있다. 오늘 강의에서는 이 중 3단계까지에 대해서만 다룬다. 간단하게 설명하자면

- GNN Layer = Message + Aggregation은 핵심적인 요소로 이 과정을 어떻게 하느냐에 따라서 GCN, GraphSAGE, GAT 등의 GNN 그래프 종류가 나뉜다.
- Layer Connectivity: 어떻게 multi layer를 쌓을건지를 결정하는 부분으로, 일반적으로 sequential 하게 층을 쌓고 옵션으로 skip connection도 추가할 수 있다.

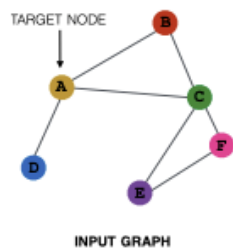
GNN Layer = Message + Aggregation

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

Connect GNN layers into a GNN

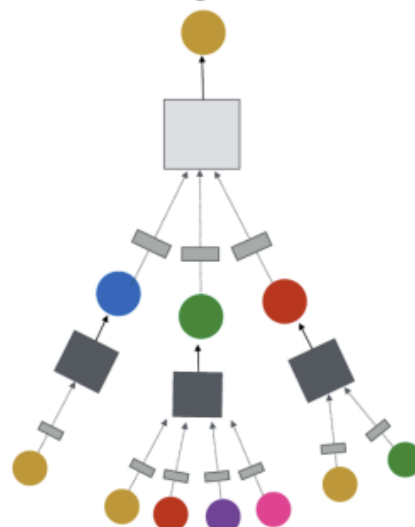
- Stack layers sequentially
- Ways of adding skip connections

A General GNN Framework(2)



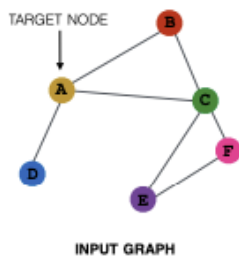
Idea: Raw input graph \neq computational graph

- Graph feature augmentation
- Graph structure augmentation

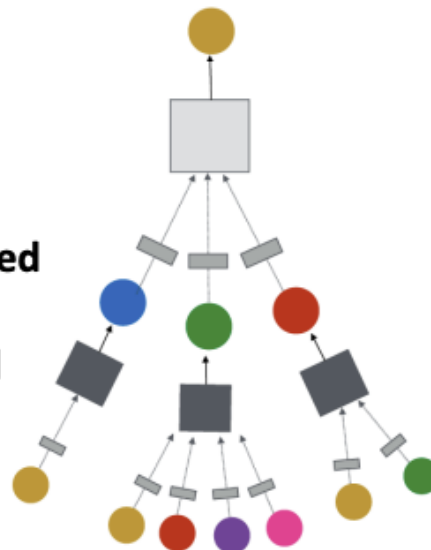


(4) Graph augmentation

A General GNN Framework(3)



(5) Learning objective

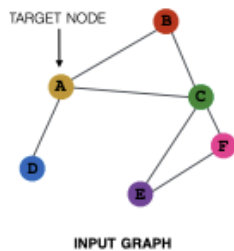


How do we train a GNN

- Supervised/Unsupervised objectives
- Node/Edge/Graph level objectives

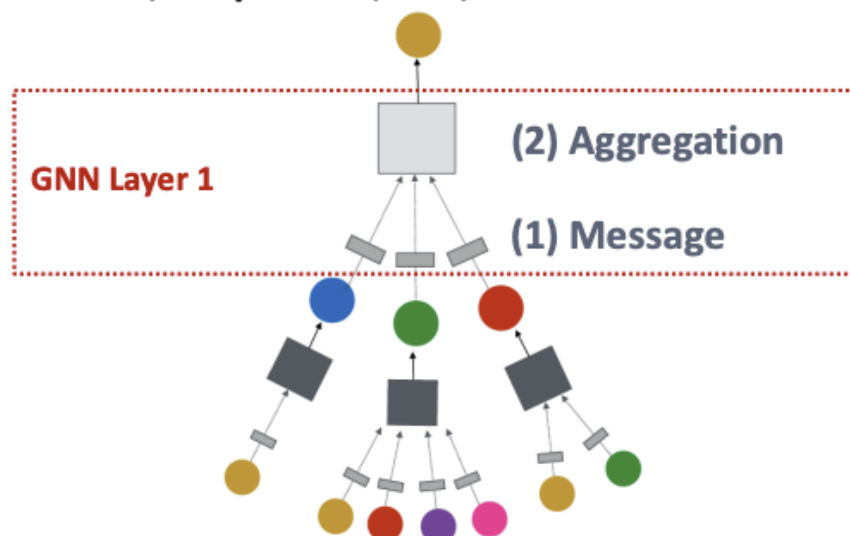
3. A Single Layer of a GNN

A GNN Layer



GNN Layer = Message + Aggregation

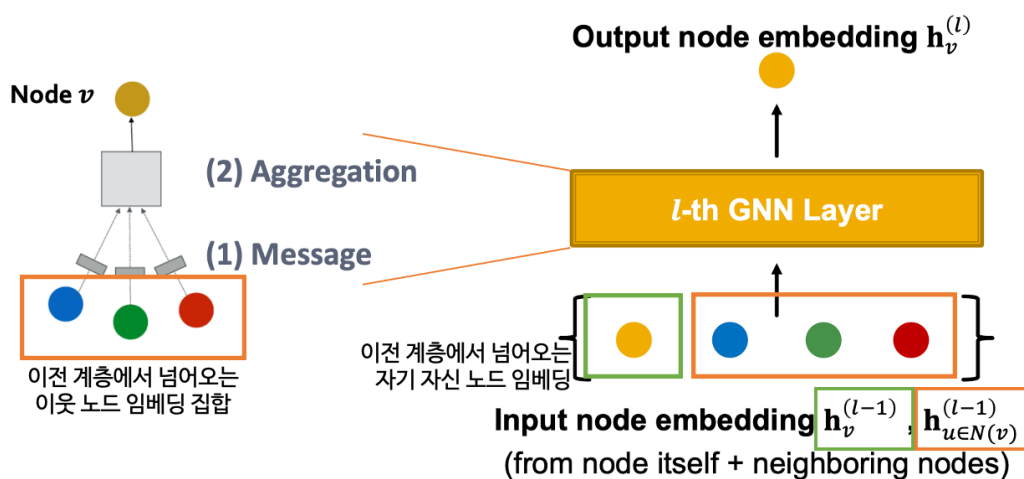
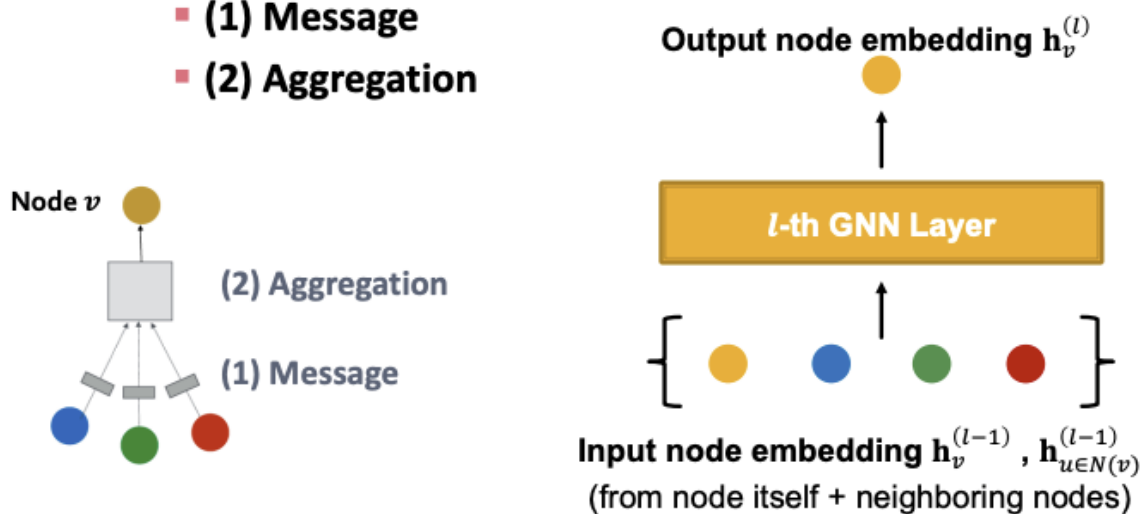
- Different instantiations under this perspective
- GCN, GraphSAGE, GAT, ...



A Single GNN Layer

■ Idea of a GNN Layer:

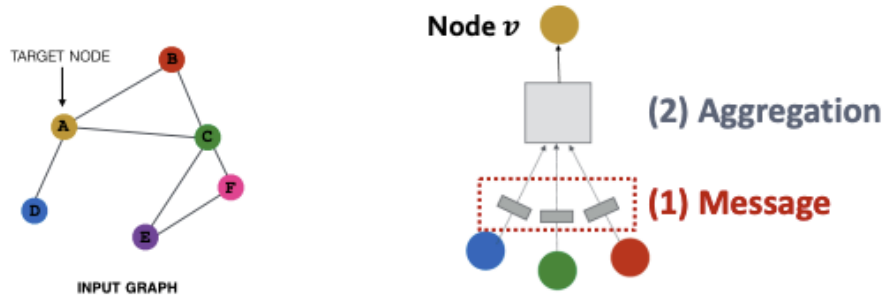
- Compress a set of vectors into a single vector
- Two step process:
 - (1) Message
 - (2) Aggregation



Message Computation

1. Message computation

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



Message Aggregation

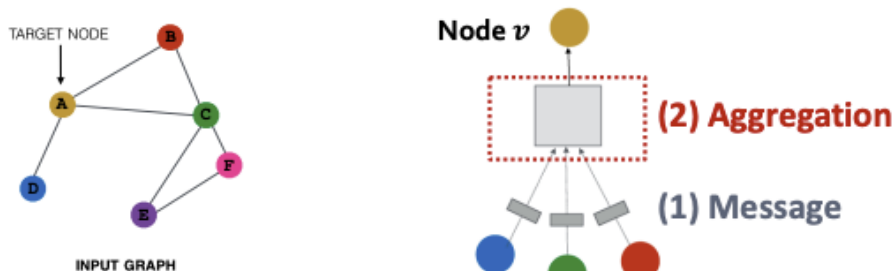
2. Aggregation

- **Intuition:** Each node will aggregate the messages from node v 's neighbors

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

- **Example:** Sum(\cdot), Mean(\cdot) or Max(\cdot) aggregator

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



Message Aggregation : Issue

- **Issue:** Information from node v itself **could get lost**

- Computation of $\mathbf{h}_v^{(l)}$ does not directly depend on $\mathbf{h}_v^{(l-1)}$

- **Solution:** Include $\mathbf{h}_v^{(l-1)}$ when computing $\mathbf{h}_v^{(l)}$

- **(1) Message:** compute message from node v itself

- Usually, a **different message computation** will be performed

$$\text{blue circle} \text{ green circle} \text{ red circle} \quad \mathbf{m}_u^{(l)} = \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)} \quad \text{yellow circle} \quad \mathbf{m}_v^{(l)} = \mathbf{B}^{(l)} \mathbf{h}_v^{(l-1)}$$

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

- Via **concatenation** or **summation**

$$\mathbf{h}_v^{(l)} = \text{CONCAT} \left(\underbrace{\text{AGG} \left(\left\{ \mathbf{m}_u^{(l)}, u \in N(v) \right\} \right)}_{\text{First aggregate from neighbors}}, \underbrace{\mathbf{m}_v^{(l)}}_{\text{Then aggregate from node itself}} \right)$$

A Single GNN Layer

- Putting things together:

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

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

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

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

- **Nonlinearity (activation)** : Adds expressiveness

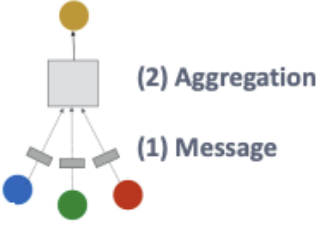
- Often written as $\sigma(\cdot)$: ReLU(\cdot), Sigmoid(\cdot),...
 - Can be added to **message** or **aggregation**

Classical GNN Layers: GCN (1)

- (1) **Graph Convolutional Networks(GCN)**

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

- How to write this as Message + Aggregation?

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


Classical GNN Layers : GCN (2)

- (1) Graph Convolutional Networks(GCN)

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

■ Message:

- Each Neighbor: $\mathbf{m}_u^{(l)} = \frac{1}{|N(v)|} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}$ **Normalized by node degree**
(In the GCN paper they use a slightly different normalization)

■ Aggregation:

- **Sum** over messages from neighbors, then apply activation
- $\mathbf{h}_v^{(l)} = \sigma \left(\text{Sum} \left(\left\{ \mathbf{m}_u^{(l)}, u \in N(v) \right\} \right) \right)$

Classical GNN Layers : GraphSAGE

- (2) GraphSAGE

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

GraphSAGE는 GCN을 기반으로 확장된 모델로, aggregation에서 차이점을 가진다.

- How to write this as Message + Aggregation?

- **Message** is computed within the **AGG(·)**

- **Two-stage aggregation**

- **Stage 1:** Aggregate from node neighbors

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

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

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

GraphSAGE Neighbor Aggregation

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

$$\text{AGG} = \underbrace{\sum_{u \in N(v)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}}_{\text{Aggregation}} \quad \text{Message computation}$$

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

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

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

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

AGG 종류에는 크게 3가지가 있다.

- Mean: GCN과 동일한 방법으로, 이웃들의 가중 평균값을 구한다.
- Pool: Message computation으로 MLP를 사용하여 non-linear하게 만들고, 이를 Mean 혹은 Max 하여 집계를 도출해낸다.
- LSTM: 이웃들로부터 오는 Message sequence model에 LSTM을 사용할 수 있다. 다만, sequence model은 order invariant하지 않으므로 훈련할 때 이웃들의 순서를 resuffle해야 한다.

GraphSAGE : L2 Normalization

■ ℓ_2 Normalization:

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

Classical GNN Layers : GAT (1)

- (3) Graph Attention Networks

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

Attention weights

- In GCN / GraphSAGE

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

기존 GCN과 GraphSAGE에서는 중요도를 $\alpha_v u = 1/|N(v)|$ 즉, node degree로 나눠서 표현했다고 볼 수 있다. 이렇게 되면 중요도는 노드 u 에 상관없이 노드 v 에만 의존하여 이웃 노드 u 는 모두 같은 중요도를 가지게 된다.

그러나 모든 이웃들은 같은 중요도를 갖고 있지 않다. 따라서 attention weight는 중요한 부분에 포커스를 맞추고 나머지는 fade out 시켜, 신경망이 중요한 곳에 컴퓨팅 파워를 쏟을 수 있게 한다. 어떤 부분이 중요한지는 다 다르기 때문에 attention weight는 훈련을 통해 학습된다.

Classical GNN Layers : GAT (2)

- (3) Graph Attention Networks

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

Attention weights

Not all node's neighbors are equally important

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

Graph Attention Networks

Can we do better than simple neighborhood aggregation?

Can we let weighting factors α_{vu} to be learned?

- **Goal** : Specify **arbitrary importance** to different neighbors of each node in the graph
- **Idea** : Compute embedding h_v^l of each node in the graph following an **attention strategy**:
 - Nodes attend over their neighborhood's message
 - Implicitly specifying different weights to different nodes in a neighborhood

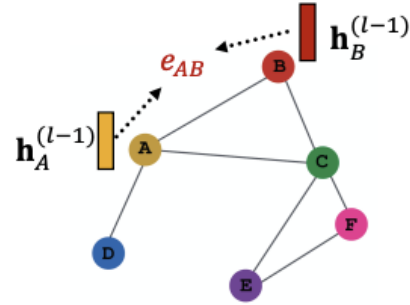
Attention Mechanism (1)

- Let α_{vu} be computed as a byproduct of an **attention mechanism** α :

- (1) Let a compute **attention coefficients** e_{vu} across pairs of nodes u, v based on their messages:

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

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



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

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

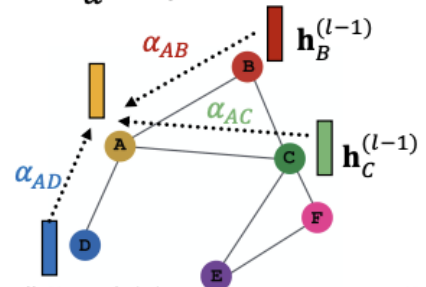
$$\alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})}$$

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

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

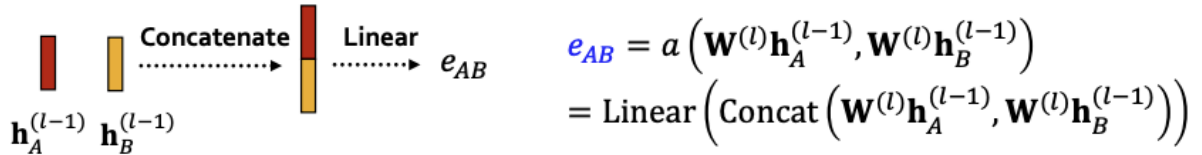
Weighted sum using $\alpha_{AB}, \alpha_{AC}, \alpha_{AD}$:

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



■ What is the form of attention mechanism α ?

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



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

■ Multi-head attention: Stabilizes the learning process of attention mechanism

- Create **multiple attention scores** (each replica with a different set of parameters):

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

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

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

- **Outputs are aggregated:**

- By concatenation or summation

$$\mathbf{h}_v^{(l)} = \text{AGG}(\mathbf{h}_v^{(l)}[1], \mathbf{h}_v^{(l)}[2], \mathbf{h}_v^{(l)}[3])$$

Multi-head attention은 각각 다른 파라미터 즉, 다른 attention coefficient로 여러 개의 $\mathbf{h}_v^{(l)}[N]$ 을 구해서 이를 aggregation해 최종 $\mathbf{h}_v^{(l)}$ 로 사용한다.

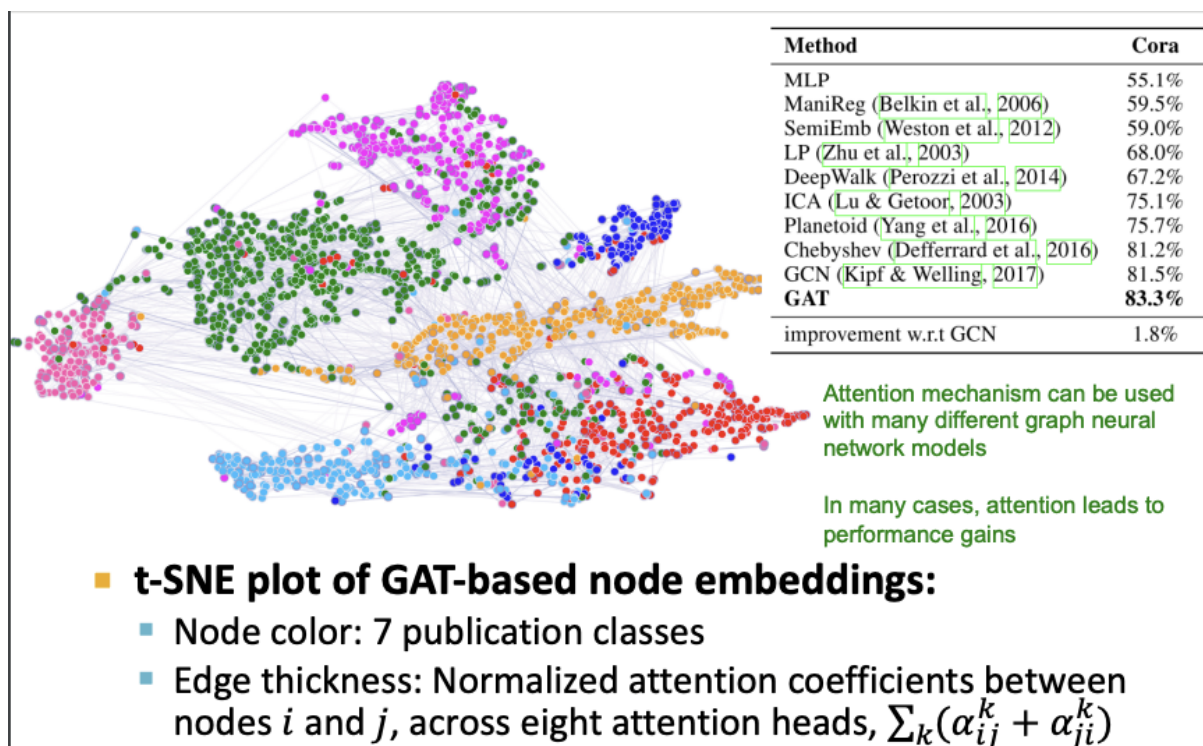
기존 attention에서 α 를 랜덤하게 초기화시키면 local minimum에 빠져 학습과 수렴에 문제가 발생할 수 있다. multi-head attention은 각각의 α 를 다 합치기 때문에 모델을 robust하고 stabilize하게 바꿔 이러

한 문제를 해결할 수 있다.

Benefits of Attention Mechanism

- **Key benefit** : Allow for (implicitly) specifying **different importance values** (α_{uv}) to **different neighbors**
- **Computationally efficient**:
 - 모든 엣지에 대해 병렬 계산이 가능하고, 모든 노드에 대해 병렬적으로 aggregation이 가능하다.
- **Storage efficient**:
 - Sparse matrix operations do not require more than $O(V + E)$ entries to be stored
 - **Fixed** number of parameters, irrespective of graph size
- **Localized**:
 - Only **attends over local network neighborhoods**
- **Inductive capability**:
 - It is a shared edge-wise mechanism
 - It does not depend on the global graph structure

GAT Example : Cora Citation Net

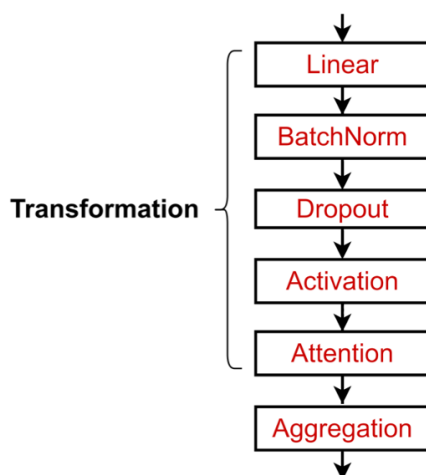


GNN Layer in Practice

<https://arxiv.org/pdf/2011.08843.pdf>

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

A suggested GNN Layer



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

Batch Normalization

<https://arxiv.org/pdf/1502.03167.pdf>

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

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

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

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

Step 1:
Compute the mean and variance over N embeddings

$$\mu_j = \frac{1}{N} \sum_{i=1}^N \mathbf{x}_{i,j}$$

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (\mathbf{x}_{i,j} - \mu_j)^2$$

Step 2:
Normalize the feature using computed mean and variance

$$\hat{\mathbf{x}}_{i,j} = \frac{\mathbf{x}_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

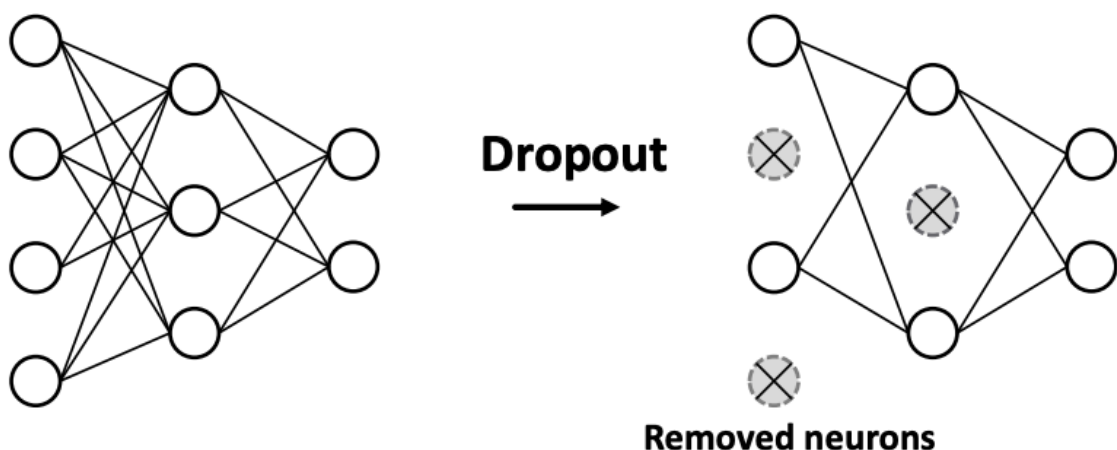
$$\mathbf{y}_{i,j} = \gamma_j \hat{\mathbf{x}}_{i,j} + \beta_j$$

Dropout

[https://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf?](https://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf?utm_campaign=buffer&utm_content=buffer79b43&utm_medium=social&utm_source=twitter.com)

[utm_campaign=buffer&utm_content=buffer79b43&utm_medium=social&utm_source=twitter.com](https://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf?utm_campaign=buffer&utm_content=buffer79b43&utm_medium=social&utm_source=twitter.com)

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



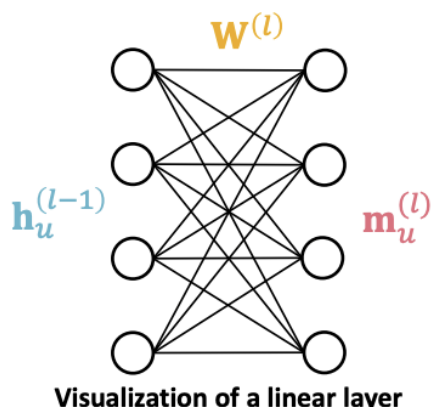
Dropout for GNNs

- In GNN, Dropout is applied **to the linear layer in the message function**

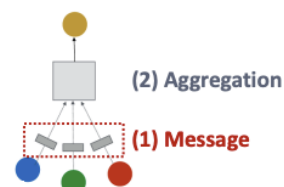
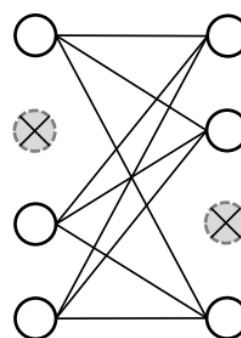
- In GNN, Dropout is applied to **the linear layer in the message function**

- A simple message function with linear

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

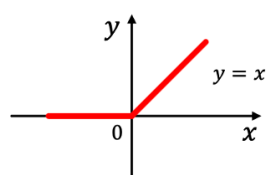


Dropout



Activation (Non-linearity)

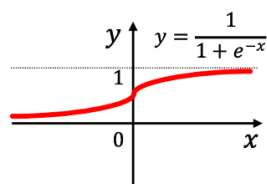
Apply activation to *i*-th dimension of embedding \mathbf{x}



- Rectified linear unit (ReLU)**

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

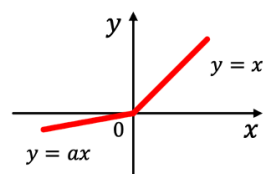
- Most commonly used



- Sigmoid**

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

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



- Parametric ReLU**

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

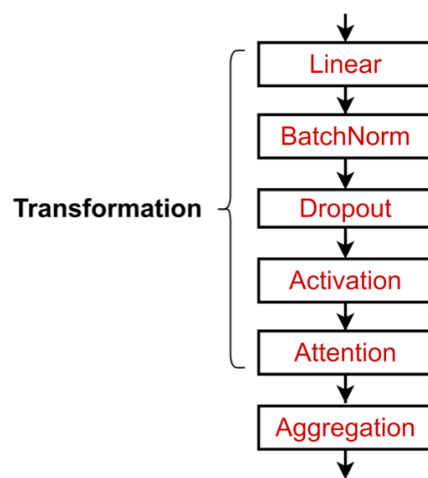
a_i is a trainable parameter

- Empirically performs better than ReLU

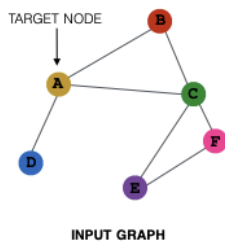
GNN Layer in Practice

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

A suggested GNN Layer



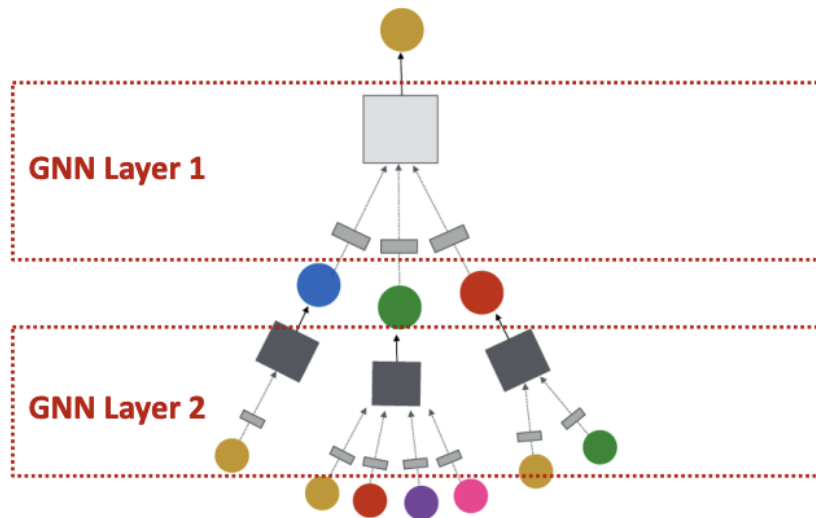
4. Stacking Layers of a GNN



How to connect GNN layers into a GNN?

- Stack layers sequentially
- Ways of adding skip connections

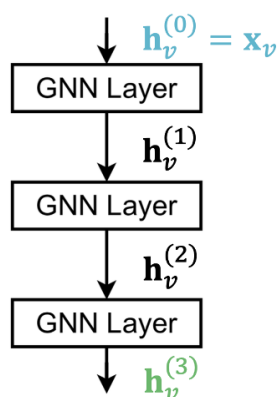
(3) Layer connectivity



Stacking GNN Layers

- How to construct a Graph Neural Networks?
 - The standard way : Stack GNN layers sequentially
 - **Input** : Initial raw node feature x_v
 - **Output** : Node embeddings h_v^L after L GNN layers

Input: Initial raw node feature \mathbf{x}_v



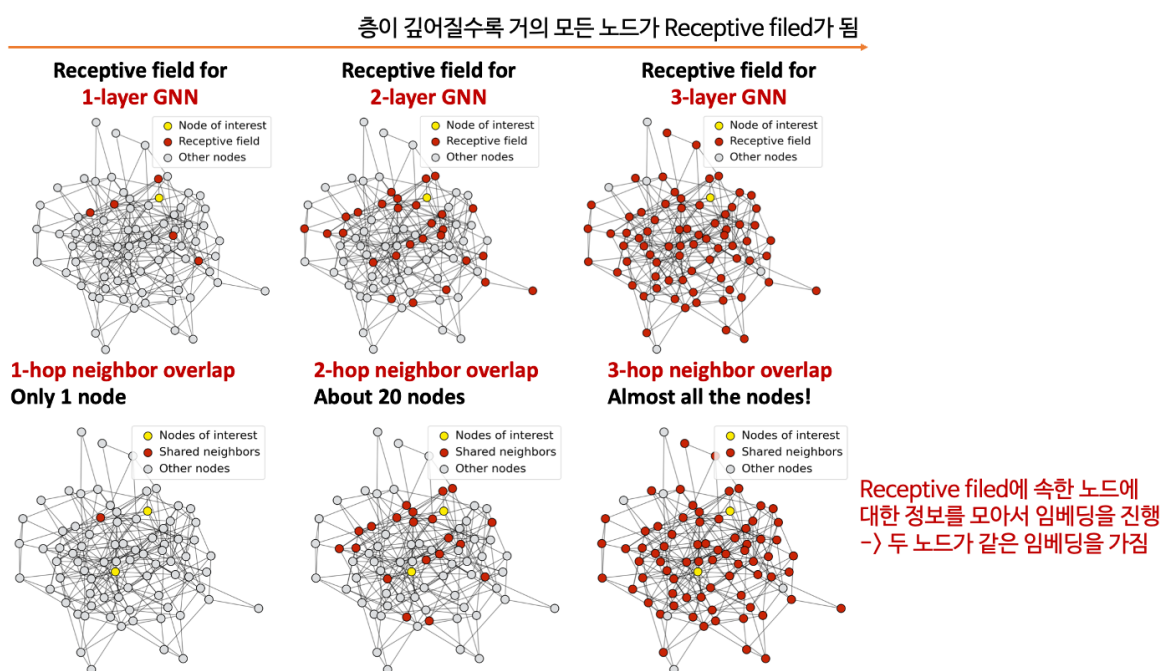
Output: Node embeddings $\mathbf{h}_v^{(L)}$ after L GNN layers

The Over-smoothing Problem

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

Receptive Field of a GNN

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



K-layer GNN의 각 노드는 K-hop neighborhood의 receptive field를 가지게 되는데 이게 깊어지면 깊어질수록 GNN 거의 대부분을 포함하게 된다.

임베딩은 receptive field에 속한 노드들에 의해 결정되기 때문에 그림과 같이 두 노드가 high-overlapped receptive field를 갖는다면 거의 동일한 임베딩 값을 갖게 된다.

Design GNN Layer Connectivity

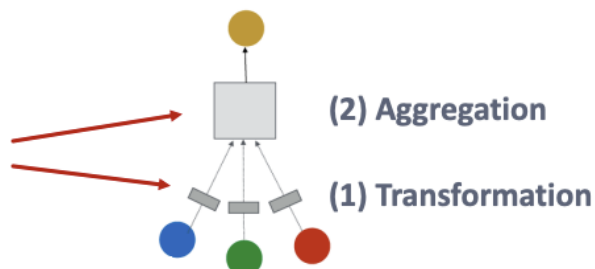
- What do we learn from the over-smoothing problem?

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

Expressive Power for Shallow GNNs

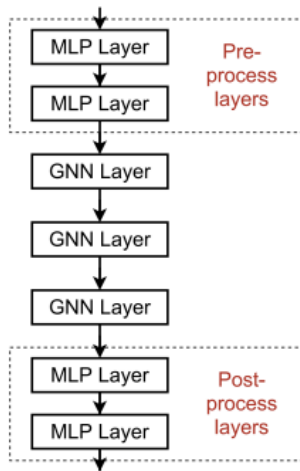
- How to make a shallow GNN more expressive?
- **Solution 1** : Increase the expressive power **within each GNN layer**
 - In our previous examples, each transformation or aggregation function only include one linear layer
 - we can **make aggregation / transformation become a deep neural network!**

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



- **Solution 2** : Add layers that do not pass messages
 - A GNN does not necessarily only contain GNN layers

- E.g., we can add **MLP layers** (applied to each node) before and after GNN layers, as **pre-process layers** and **post-process layers**



Pre-processing layers: Important when encoding node features is necessary.

E.g., when nodes represent images/text

Post-processing layers: Important when reasoning / transformation over node embeddings are needed

E.g., graph classification, knowledge graphs

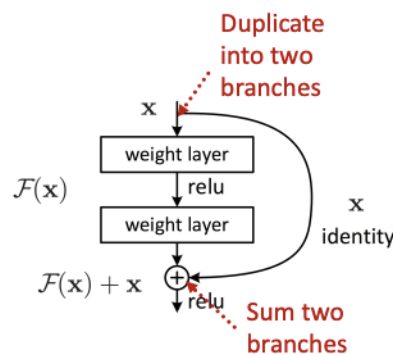
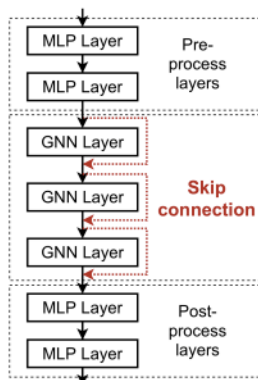
In practice, adding these layers works great!

Design GNN Layer Connectivity

- What if my problem still requires many GNN layers?

- **Lesson 2 : Add skip connections in GNNs**

- **Observation from over-smoothing:** Node embeddings in earlier GNN layers can sometimes better differentiate nodes
- **Solution:** We can increase the impact of earlier layers on the final node embeddings, **by adding shortcuts in GNN**



Idea of skip connections: Before adding shortcuts:

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

After adding shortcuts:

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

Jure Leskovec, Stanford CS224W: Machine Learning with Graphs, <http://cs224w.stanford.edu>

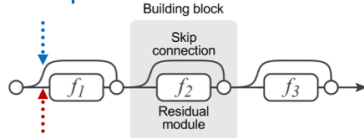
50

- We automatically get **a mixture of shallow GNNs and deep GNNs**

All the possible paths:

$$2 * 2 * 2 = 2^3 = 8$$

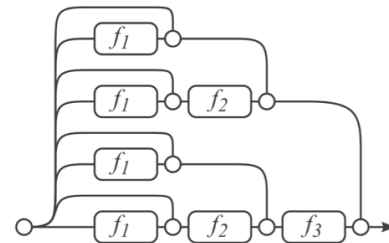
Path 2: skip this module



Path 1: include this module

(a) Conventional 3-block residual network

=



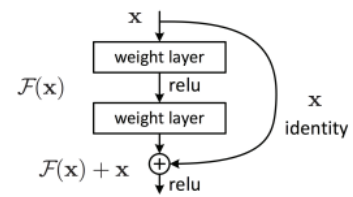
(b) Unraveled view of (a)

Example :GCN with Skip Connections

- A standard GCN layer

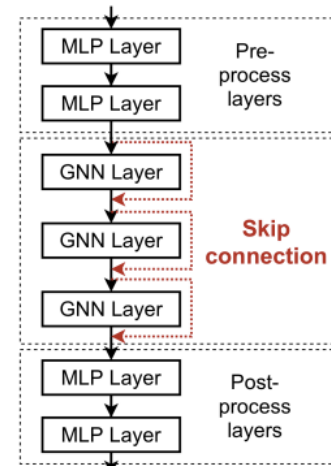
$$\mathbf{h}_v^{(l)} = \sigma \left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|} \right)$$

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



- A GCN layer with skip connection

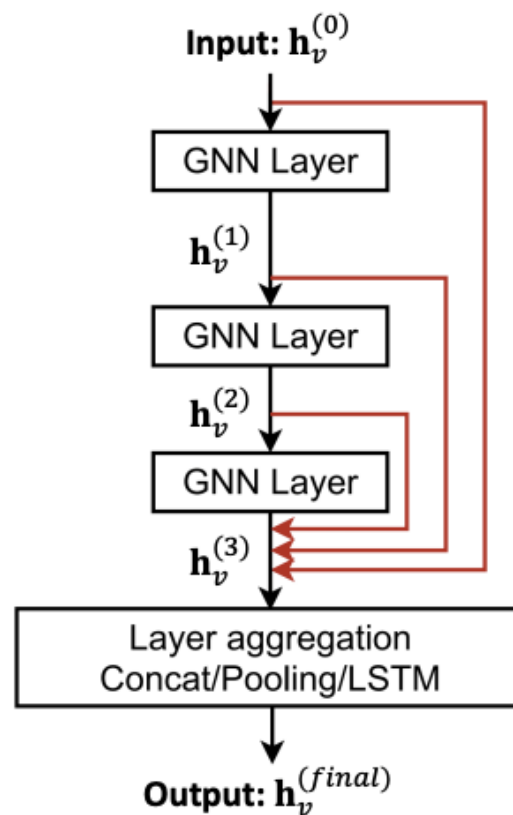
$$\mathbf{h}_v^{(l)} = \sigma \left(\underbrace{\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}}_{F(\mathbf{x})} + \underbrace{\mathbf{h}_v^{(l-1)}}_{\mathbf{x}} \right)$$



Other Options of Skip Connections

- Other options : Directly skip to the last layer

- The final layer directly **aggregates from all the node embeddings** in the previous layers



Summary

- Recap : A general perspective for GNNs
 - GNN layer:
 - Transformation(message) + Aggregation
 - Classic GNN layers : GCN, GraphSAGE, GAT
 - Layer connectivity:
 - Deciding number of layers
 - Skip connections

References

<https://velog.io/@tobigsgnn1415/7.-Graph-Neural-Networks-2>
<http://snap.stanford.edu/class/cs224w-2020/slides/07-GNN2.pdf>