6. Graph Neural Networks

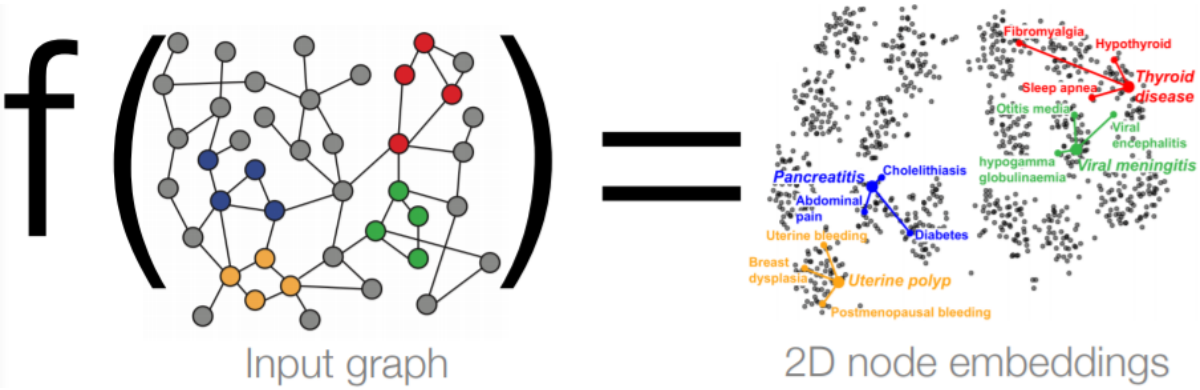
Contents

Recap : Node Embeddings

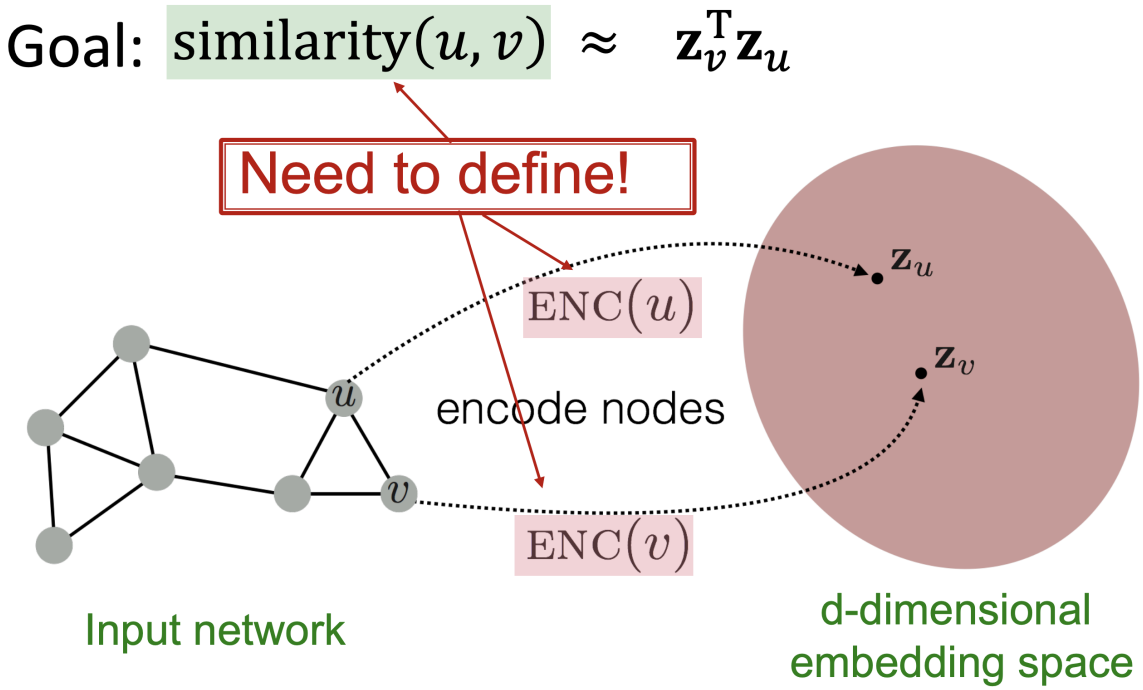
1. Basics of Deep Learning
2. Deep Learning for Graphs
3. Graph Convolutional Networks and GraphSAGE

Recab : Node Embeddings

- Intution : Map nodes to d-dimensional embeddings such that similar nodes in the graph are embedded close together



How to learn mapping function f?



Recap : Two Key Components

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

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

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

v: node in the input graph

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

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

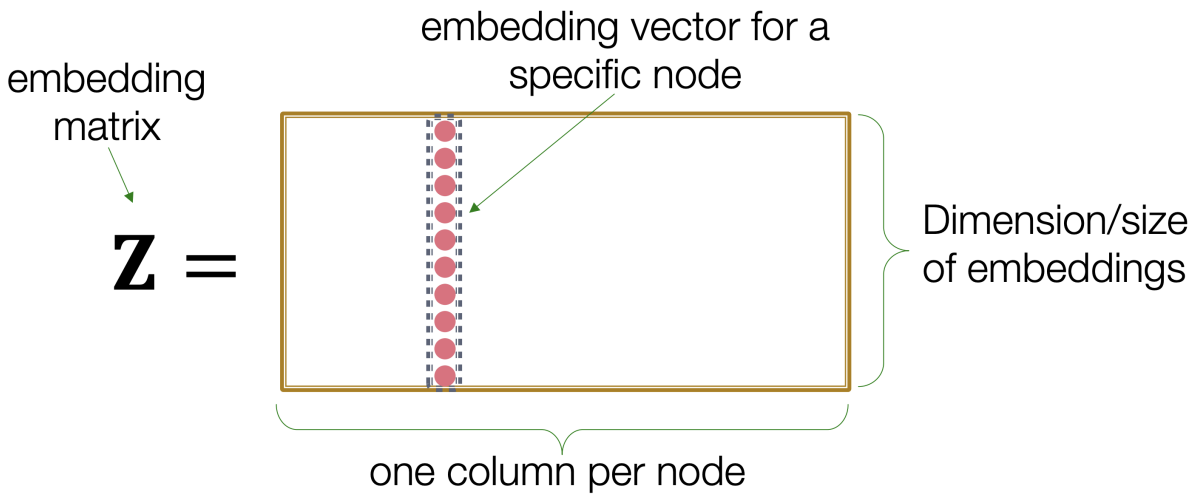
Decoder

Similarity of u and v in the original network

dot product between node embeddings

Recap : “Shallow” Encoding

Simplest encoding approach : **encoder is just an embedding-lookup**



하지만 Shallow Encoder은 몇 가지 문제점이 있다.

1. $O(|V|)$ **parameters are needed:**
   - No sharing of parameters between nodes
   - Every node has its own unique embedding
2. **Inherently “transductive”:**
   - Cannot generate embeddings for nodes that are not seen during training
3. **Do not incorporate node features:**
   - Many graphs have features that we can and should leverage

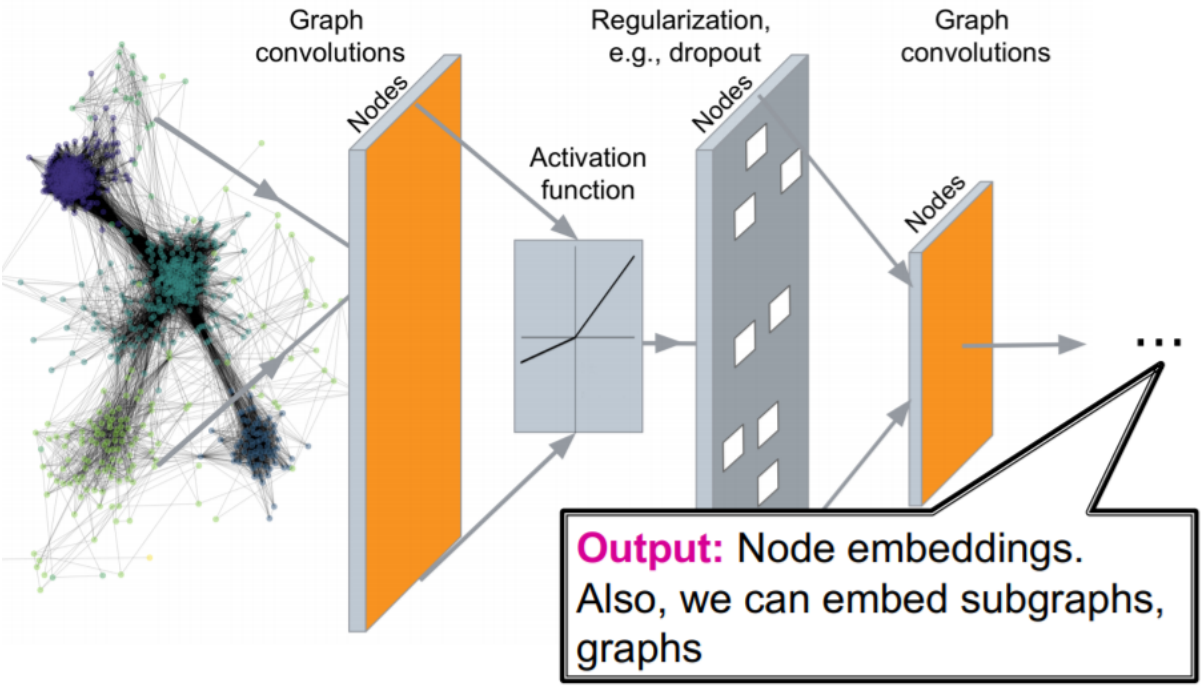
Today : Deep Graph Encoders

- Today : We will now discuss deep methods based on **graph neural networks(GNNs):**

$$\mathrm{ENC}(v) =$$ **multiple layers of non-linear transformations based on graph structure**

- Note : All these deep encoders can be **combined with node similarity functions** defined in the lecture 3

Deep Graph Encoders



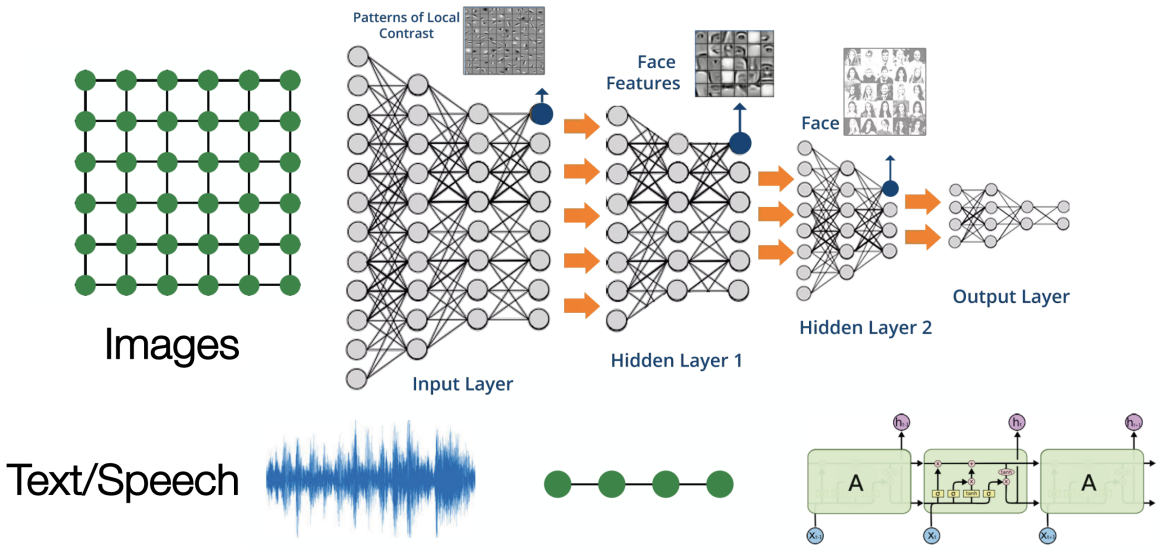
이제는 단순히 look-up만 하는 것이 아니라, cnn, rnn, dnn 등의 딥러닝 방법론들을 차용하여 인코더를 구성하게 된다. 즉, 인코더의 레이어가 깊어질 것이다.

Tasks on Networks

Tasks we will be able to solve:

- **Node classification**
  - Predict a type of a given node
- **Link prediction**
  - Predict whether two nodes are linked
- **Community detection**
  - Identify densely linked clusters of nodes
- **Network similarity**
  - How similar are two (sub)networks

Modern ML Toolbox

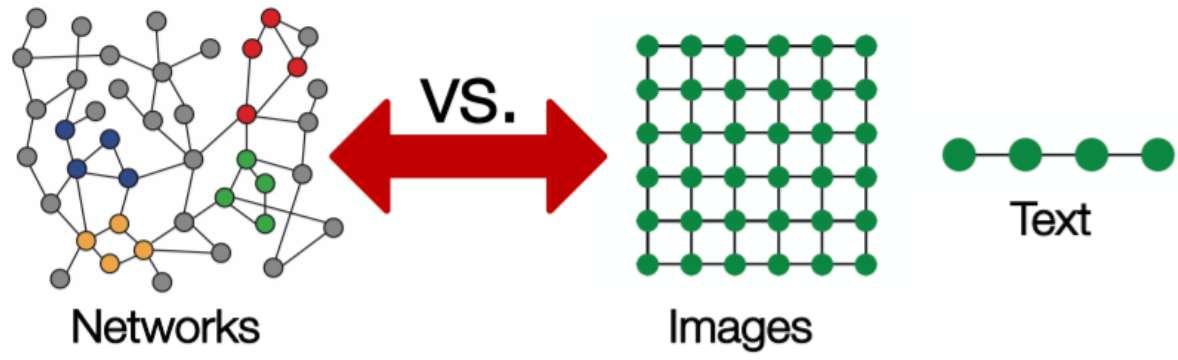


Modern deep learning toolbox is designed for simple sequences & grids

Why is it Hard?

But networks are far more complex!

- Arbitary size and complex topological structure(i.e., no spatial locality like grids)
- No fixed node ordering or reference point
- Often dynamic and have multimodal features



1. Basics of Deep Learning

Machine Learning as Optimization

- Supervised learning : we are given input x, and the goal is to predict label y
- Input x can be:
  - Vectors of real numbers
  - Sequences(natural language)
  - Matrices(images)
  - Graphs(potentially with node and edge features)
- **We formulate the task as an optimization problem**
- **Formulate the task as an optimization problem:**

$$\min_{\Theta} \mathcal{L}(y, f(x))$$

Objective function

- Θ : a set of **paramters** we optimize
  - Could contain one or more scalars, vectors, matrices …
  - E.g. Θ = {Z} in the shallow encoder (the embedding lookup)
- L : **loss function.** Example : L2 loss

$$\mathcal{L}(y, f(x)) = \|y - f(x)\|_2$$

  - Other common loss functions:
    - L1 loss, huber loss, max margin(hinge loss), cross entropy …
    - See https://pytorch.org/docs/stable/nn.html

Loss Function Example

- One common loss for classification: **cross entropy (CE)**
- Label $\boldsymbol{y}$ is a categorical vector (one-hot encoding)
  - e.g. $\boldsymbol{y} =$ | 0 | 0 | 1 | 0 | 0 |

y is of class "3"

- $f(\boldsymbol{x}) = \text{Softmax}(g(\boldsymbol{x}))$
  - Recall from lecture 3: $f(\boldsymbol{x})_i = \frac{e^{g(x)_i}}{\sum_{j=1}^{C} e^{g(x)_j}}$,

where C is the number of classes.

$g(x)_i$ denotes i-th coordinate of the vector output of func. $g(x)$

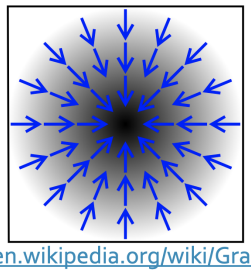
  - e.g. $f(\boldsymbol{x}) =$ | 0.1 | 0.3 | 0.4 | 0.1 | 0.1 |
- $\text{CE}(\boldsymbol{y}, f(\boldsymbol{x})) = -\sum_{i=1}^{C}(y_i \log f(x)_i)$
  - $y_i, f(x)_i$ are the **actual** and **predicted** value of the i-th class.
  - **Intuition:** the lower the loss, the closer the prediction is to one-hot
- **Total loss over all training examples:**
  - $\mathcal{L} = \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{T}} \text{CE}(\boldsymbol{y}, f(\boldsymbol{x}))$
  - $\mathcal{T}$: training set containing all pairs of data and labels $(\boldsymbol{x}, \boldsymbol{y})$

Machine Learning as Optimization

- **How to optimize the objective function?**
- **Gradient vector:** Direction and rate of fastest increase

Partial derivative

$$\nabla_{\Theta}\mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial \Theta_1}, \frac{\partial \mathcal{L}}{\partial \Theta_2}, \dots\right)$$



https://en.wikipedia.org/wiki/Gradient

  - $\Theta_1, \Theta_2 \dots$: components of Θ
- Recall **directional derivative** of a multi-variable function (e.g. $\mathcal{L}$) along a given vector represents the instantaneous rate of change of the function along the vector.
- Gradient is the directional derivative in the **direction of largest increase**

Summary

- Objective function:

$$\min_{\Theta} \mathcal{L}(y, f(x))$$

- f can be a simple linear layer, an MLP, or other neural networks (e.g., a GNN later)
- Sample a minibatch of input x
- Forward propagation : Compute L given x
- Back-propagation : Obtain gradient using a chain rule

---

2. Deep Learning for Graphs

Content

- **Local network neighborhoods:**
  - Describe aggregation strategies
  - Define computation graphs
- **Stacking multiple layers:**
  - Describe the model, parameters, training
  - How to fit the model?
  - Simple example for unsupervised and supervised training

Setup

- Assume we have a graph G :

- V is the **vertex set**
- A is the adjacency matrix (assume binary)
-

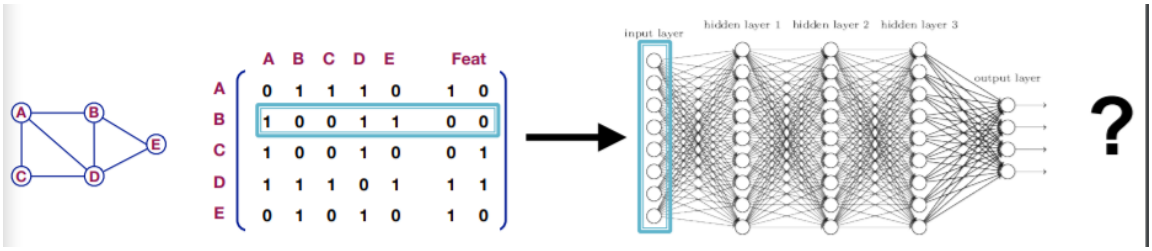
$$\boldsymbol{X} \in \mathbb{R}^{m \times |V|}$$

is a matrix of **node features**

- **Node features:**
  - Social networks : User profile, User image
  - Biological networks : Gene expression profiles, gene functional information
  - When there is no node feature in the graph dataset:
    - Indicator vectors(one-hot encoding of a node)
    - Vector of constant 1 : [1, 1, 1, …, 1]

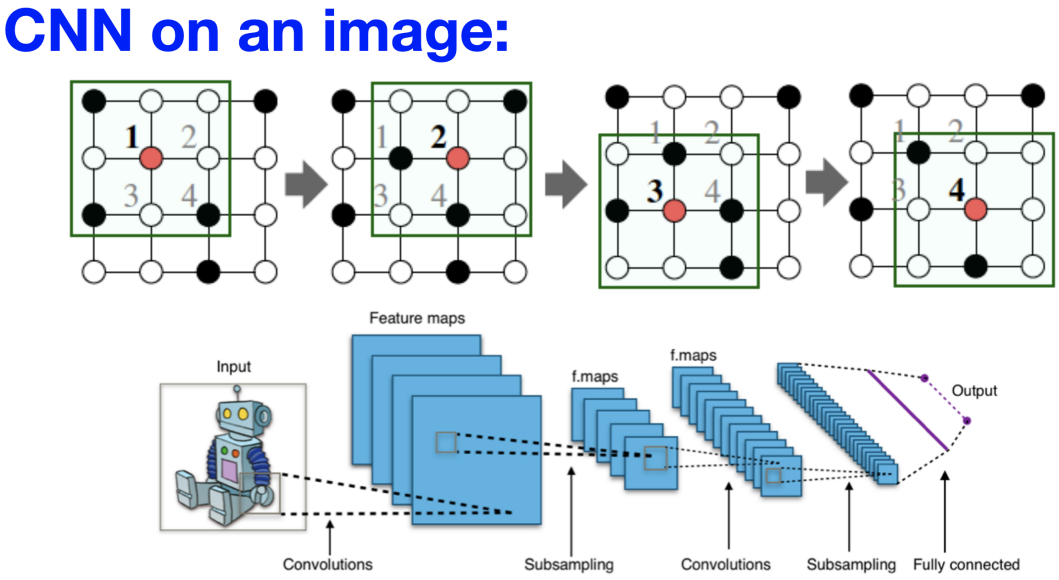
A naive Approach

- Join adjacency matrix and features
- Feed them into a deep neural net:



1. $O(|V|)$의 파라미터가 필요하다. 즉, 인접행렬의 column 수만큼은 입력값과 hidden layer를 연결하는 파라미터가 필요한데, 이러면 또 파라미터가 너무 많아져 과적합이 발생할 우려가 있다.
2. 그래프가 다른 크기를 가질 경우 적용할 수 없다. 만약 전체 노드의 수가 10개 짜리인 그래프로 학습한 모델이 있다고 하자. 이 모델은 입력값으로 크기가 10인 벡터를 받는다. 하지만 그래프의 노드 수가 20인 그래프가 있다면, 해당 모델을 통한 분석이 불가능해진다.
3. 노드 순서에 민감하다.우리는 결국 그래프를 분석하고자 하는 것인데, SGD에 따라 어떤 노드를 먼저 입력값으로 받는지가 중요한 문제가 된다 결국 먼저 입력값으로 들어온 노드에 맞추어 그래디언트를 따라 흐르고 다음 입력값을 고려하게 되기 때문이다.

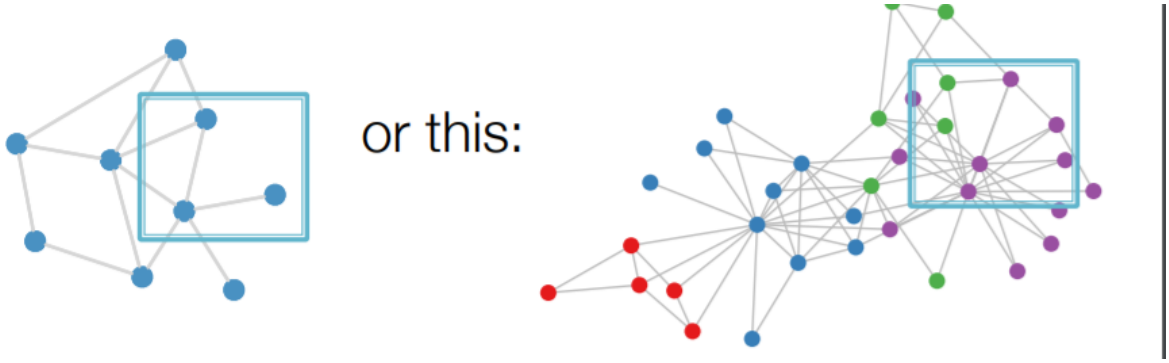
Idea : Convolutional Networks



Goal is to generalize convolutions beyond simple lattices Leverage node features/attributes(e.g., text, images)

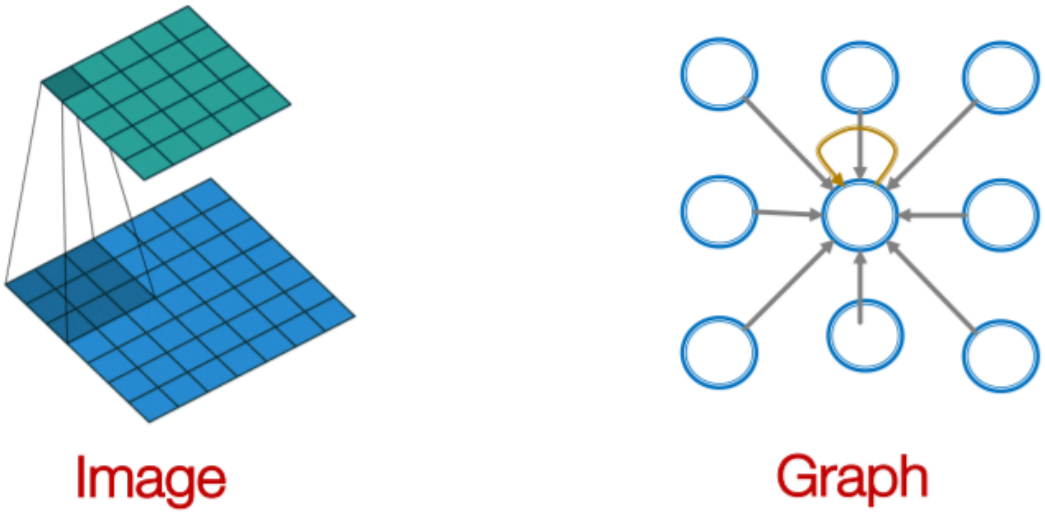
Real-World Graphs

But our graphs look like this:



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

위와 같이 똑같은 window size에 들어오는 노드의 갯수가 슬라이딩하면서 큰 차이가 나버린다. 즉, 각 슬라이딩 위도우마다 제각기 다른 이미지가 들어오게 되는 것이다. 이는 이미지는 고정되 크기임에 반해, 그래프는 고정된 크기의 그리드로 되어 있지 않아 발생하는 문제이다.



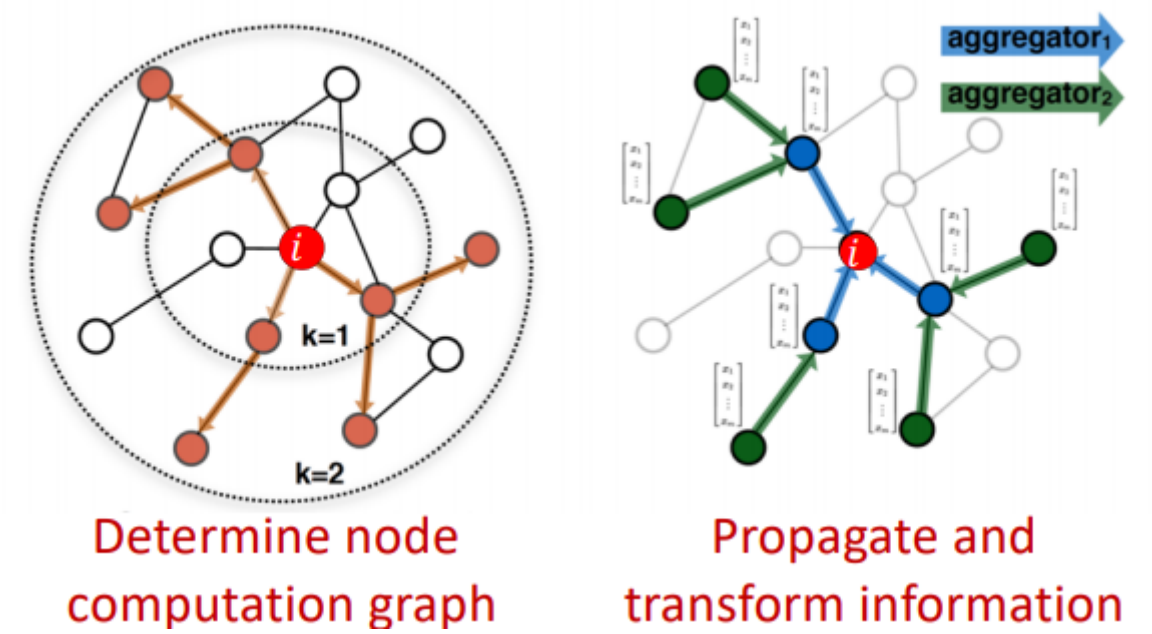
이를 개선하고자 위와 같이 생각해보자. CNN에서 한 레이어가 하는 일은, 기준이 되는 픽셀을 중심으로 상하좌우 대각선의 픽셀에서 정보를 받아 종합하고, 이를 일종의 연산을 통해 하나의 픽셀을 내뱉는다. 이는 그래프에서 한 노드를 중심으로 이웃노드에서 메세지를 받아 하나의 노드로 메세지를 종합하고, 연산하여 현재 노드의 메세지를 만드는 일과 비슷하다고 할 수 있다. 즉 이미지의 Locality를 그래프의 메세지 개념으로 연결해볼 수 있지 않을까?

Idea: transform information at the neighbors and combine it:

- Transform "messages" h_i from neighbors: $W_i\ h_i$
- Add them up: $\sum_i W_i\ h_i$

Graph Convolutional Networks

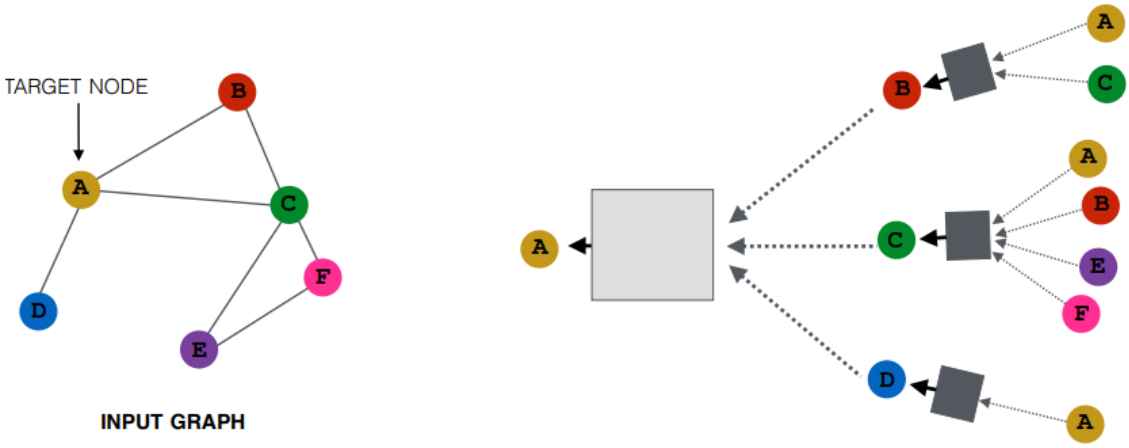
- **Idea** : Node's neighborhood defines a computation graph



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

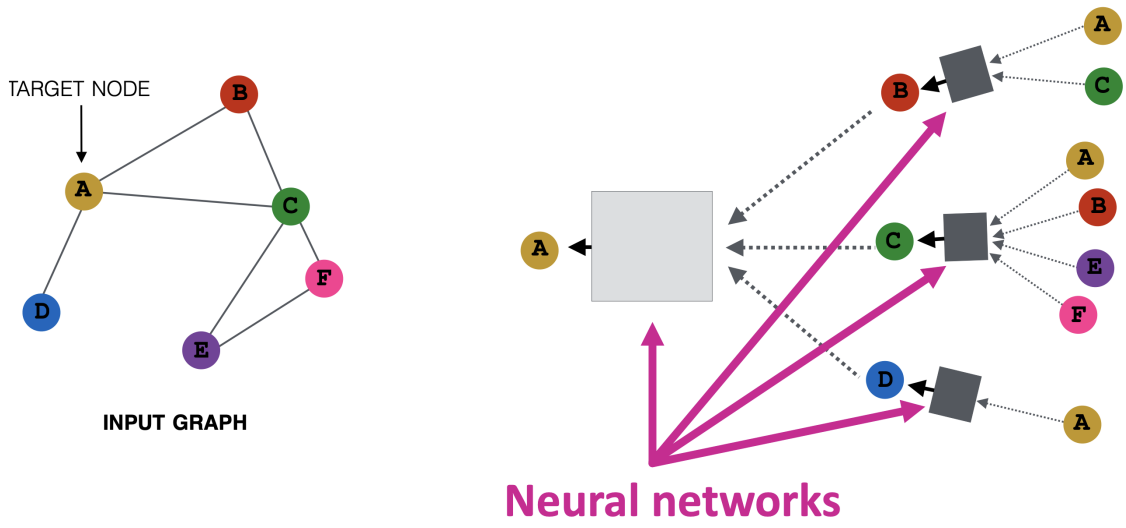
Idea : Aggregate Neighbors

- **Key idea** : Generate node embeddings based on **local network neighborhoodes**

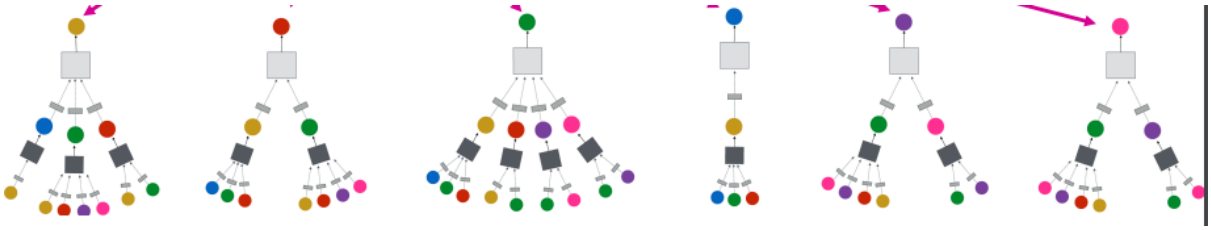


왼쪽과 같은 그래프에서 A 노드에 대해 2 hop node까지 계산 그래프를 생성하면 오른쪽 그림과 같다. 즉, A는 이웃 노드 D, B, C로부터 정보를 받고, 해당 노드들의 각자의 이웃노드에서 정보를 받는다. 이를 계산그래프로 풀어 A 노드에 대한 하나의 모델이 만들어지게 된다. 이때 정보의 흐름에 따라 순서를 매겨보면 계산 그래프에서 오른쪽에서 왼쪽으로 정보가 흘러야 한다. 오른쪽 가장 위를 예를 들어보면, A와 C 노드에서 정보가 B로 흘러가야 한다. 이때의 과정은 A와 C 노드에 해당하는 임베딩 벡터를 입력으로 하여 정보가 합쳐져서 연산이 이루어져 레이어를 통과한다. 이렇게 B를 향해 모아진 정보는 기존의 B의 임베딩 벡터와 연산을 거쳐 두번째 레이어의 B 노드에 대한 임베딩 벡터로 사용된다. 즉, 첫번째 레이어의 B 노드의 임베딩 벡터와 두번째 B 노드의 임베딩 벡터는 달라진다.

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



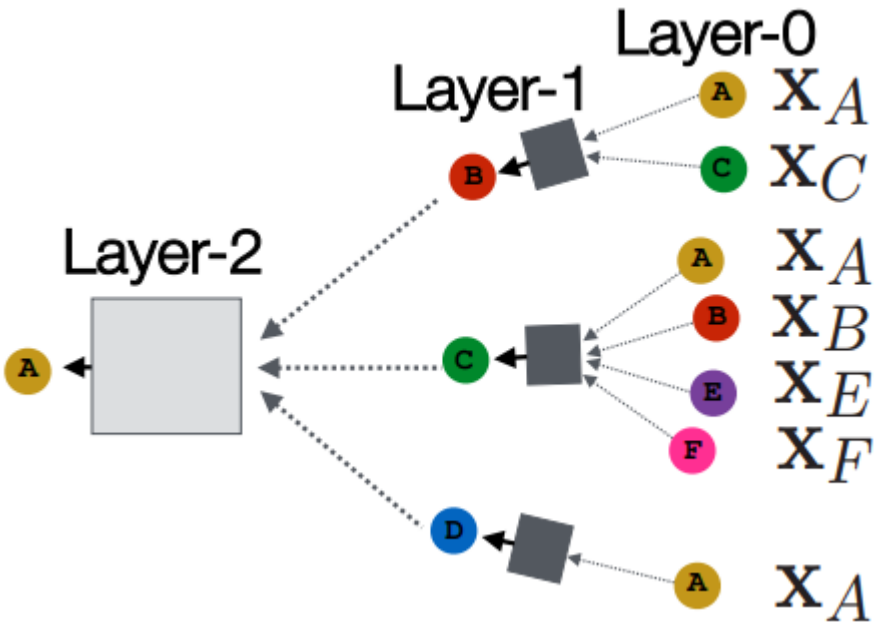
- **Intution** : Network neighborhood defines a **computation graph**



Deep Model : Many Layers

- Model can be **of arbitrary depth:**
  - Nodes have embeddings at each layer

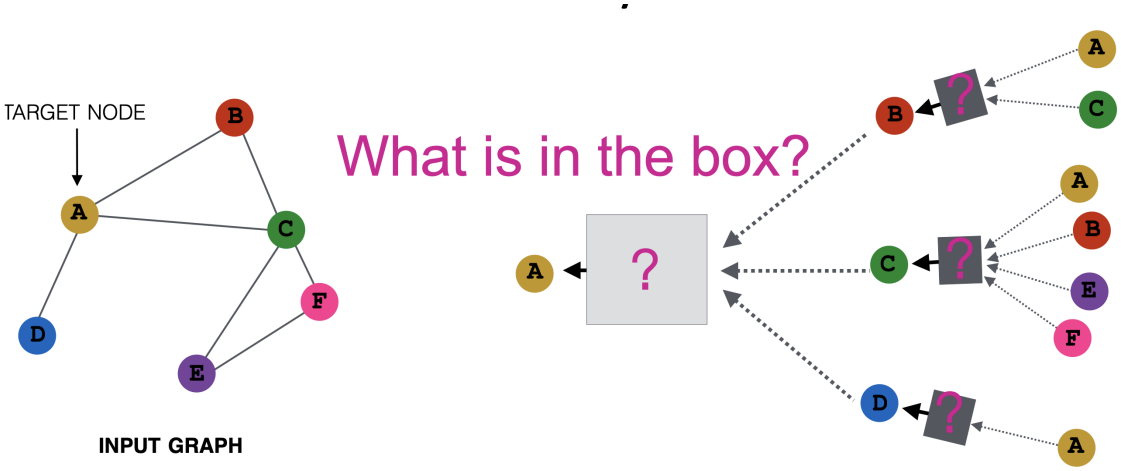
- Layer-0 embedding of node u is its input feature, x_u
- Layer-K embedding gets information from nodes that are K hops away



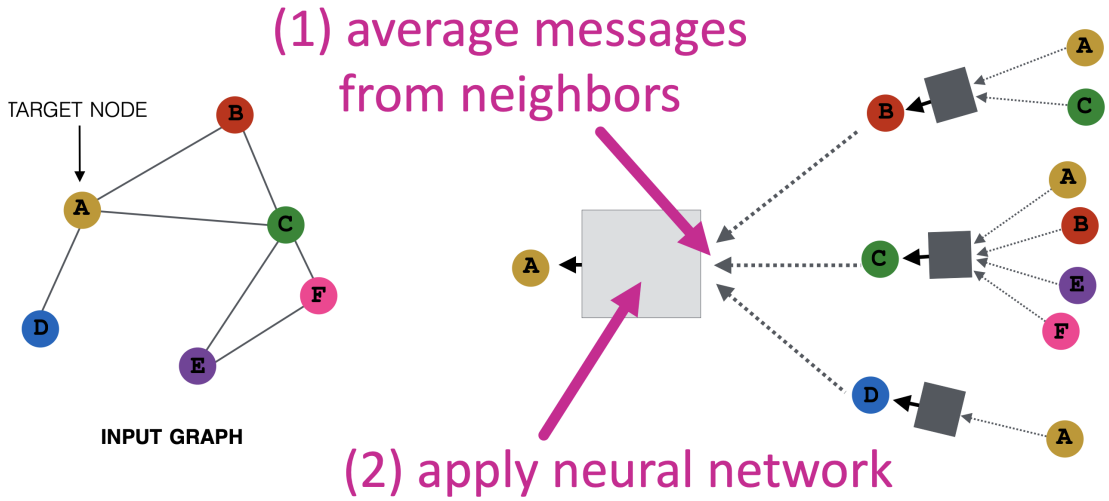
이때 레이어는 깊지 않다고 한다. 아무래도 모든 노드에 대해 각기 다른 계산 그래프가 생성되다 보니 깊어지기 힘든 것으로 보인다. 또한 k번째 레이어의 임베딩 벡터는 곧 k hop 멀리 떨어진 노드부터 정보가 모아져 온 것으로 이해할 수도 있다.

Neighborhood Aggregation

- **Neighborhood aggregation** : Key distinctions are in how different approaches aggregate information across the layers



- **Basic approach** : Average information from neighbors and apply a neural network



The Math : Deep Encoder

- **Basic approach** : Average neighbor messages and apply a neural network

Initial 0-th layer embeddings are equal to node features

$$\mathrm{h}_v^0 = \mathrm{x}_v$$

embedding of v at layer l

$$\mathrm{h}_v^{(l+1)} = \sigma\left(\mathrm{W}_l \sum_{u \in \mathrm{N}(v)} \frac{\mathrm{h}_u^{(l)}}{|\mathrm{N}(v)|} + \mathrm{B}_l \mathrm{h}_v^{(l)}\right), \forall l \in \{0, \ldots, L-1\}$$

$$\mathrm{z}_v = \mathrm{h}_v^{(L)}$$

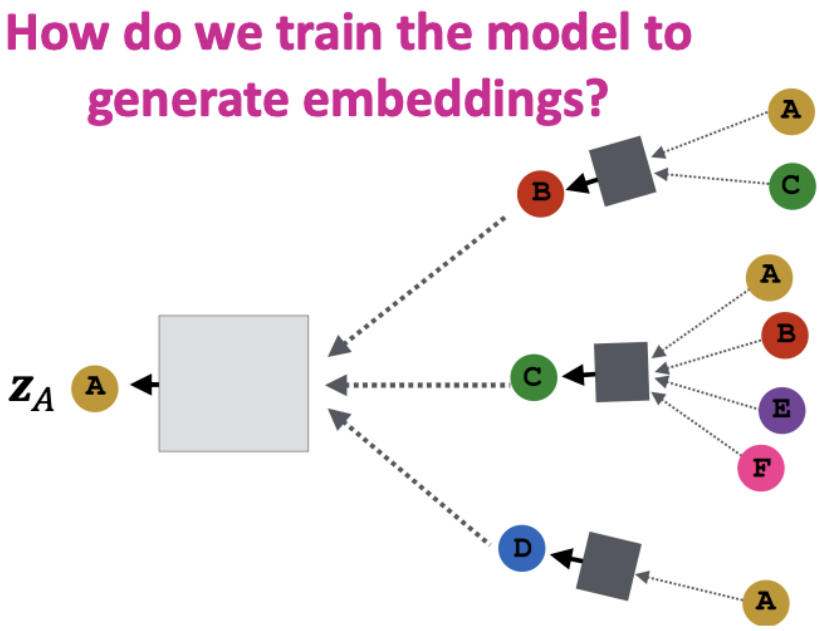
Embedding after L layers of neighborhood aggregation

Non-linearity (e.g., ReLU)

Average of neighbor's previous layer embeddings

Total number of layers

Training the Model



Need to define a loss function on the embeddings

Model Parameters

Trainable weight matrices (i.e., what we learn)

$$\mathbf{h}_v^{(0)} = \mathbf{x}_v$$

$$\mathbf{h}_v^{(l+1)} = \sigma\left(\mathbf{W}_l \sum_{u \in \mathrm{N}(v)} \frac{\mathbf{h}_u^{(l)}}{|\mathrm{N}(v)|} + \mathbf{B}_l \mathbf{h}_v^{(l)}\right), \forall l \in \{0, \ldots, L-1\}$$

$$\mathbf{z}_v = \mathbf{h}_v^{(L)}$$

Final node embedding

We can feed these **embeddings into any loss function** and run SGD to **train the weight parameters**

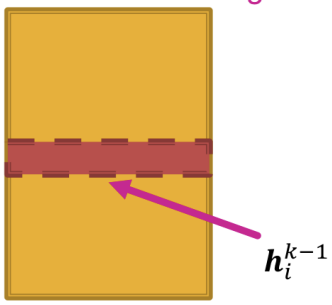
h_v^l : the hidden representation of node v at layer l

- W_k: weight matrix for neighborhood aggregation
- B_k : weight matrix for transforming hidden vector of self

Matrix Formulation (1)

- **Many aggregations can be performed efficiently by (sparse) matrix operations**
- Let $H^{(l)} = [h_1^{(l)} \dots h_{|V|}^{(l)}]^{\mathrm{T}}$
- Then: $\sum_{u \in N_v} h_u^{(l)} = \mathrm{A}_{v,:}\mathrm{H}^{(l)}$
- Let D be diagonal matrix where $D_{v,v} = \mathrm{Deg}(v) = |N(v)|$
  - The inverse of D: D^{-1} is also diagonal: $D_{v,v}^{-1} = 1/|N(v)|$
- **Therefore,**

Matrix of hidden embeddings H^{k-1}



$$\sum_{u \in N(v)} \frac{h_u^{(l-1)}}{|N(v)|} \Rightarrow H^{(l+1)} = D^{-1}AH^{(l)}$$

이웃 노드 정보를 합산(aggregate)하는 과정을 위와 같은 행렬 형태로 나타낼 수 있다.

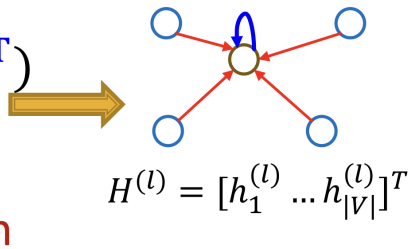
- A : 인접행렬로 연결된 이웃 노드들의 embedding을 합(summation)하는 역할
- $D^{-}1$: degree의 역수로 이루어진 대각 행렬로 이웃 노드 수로 나누는 역할

Matrix Formulation (2)

- Re-writing update function in matrix form:

$$H^{(l+1)} = \sigma(\tilde{A}H^{(l)}W_l^{\mathrm{T}} + H^{(l)}B_l^{\mathrm{T}})$$

where $\tilde{A} = D^{-1}A$



  - Red: neighborhood aggregation
  - Blue: self transformation

즉, 업데이트 과정을 위와 같이 행렬 형태로 표현할 수 있다.A 가 sparse 하므로 효율적인 sparse matrix multiplication을 사용할 수 있다.

대부분의 GNN의 aggregation 연산을 위와 같은 행렬 형태로 수행할 수 있다. 하지만 aggregation function이 복잡한 경우 불가능한 경우도 존재

How to train a GNN

- Node embedding z_v is a function of input graph
- **Supervised setting** : we want to minimize the loss L:

$$\min_{\Theta} \mathcal{L}(y, f(z_v))$$

- y : node label
- L could be L2 if y is real number, or cross entropy if y is categorical
- **Unsupervised setting:**
  - No node label available
  - **Use the graph structure as the supervision!**

Unsupervised Training

- **"Similar" nodes have similar embeddings**

$$\mathcal{L} = \sum_{z_u, z_v} \mathrm{CE}(y_{u,v}, \mathrm{DEC}(z_u, z_v))$$

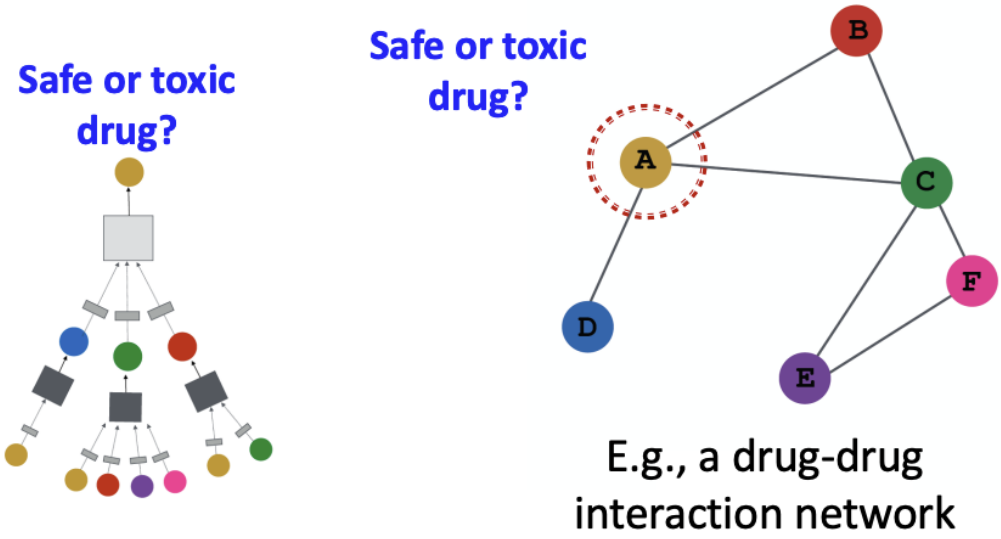
  - Where $y_{u,v} = 1$ when node u and v are **similar**
  - CE is the cross entropy (slide 16)
  - DEC is the decoder such as inner product (lecture 4)

- **Node similarity** can be anything from lecture 3, e.g., a loss based on:
  - **Random walks**(node2vec, DeepWalk, struc2vec)
  - **Matrix factorization**

- Node proximity in the graph

Supervised Training

Directly train the model for a supervised task (e.g., node classification)



- Use cross entropy loss

$$\mathcal{L} = \sum_{v \in V} y_v \log(\sigma(z_v^T \theta)) + (1 - y_v) \log(1 - \sigma(z_v^T \theta))$$

Encoder output: node embedding

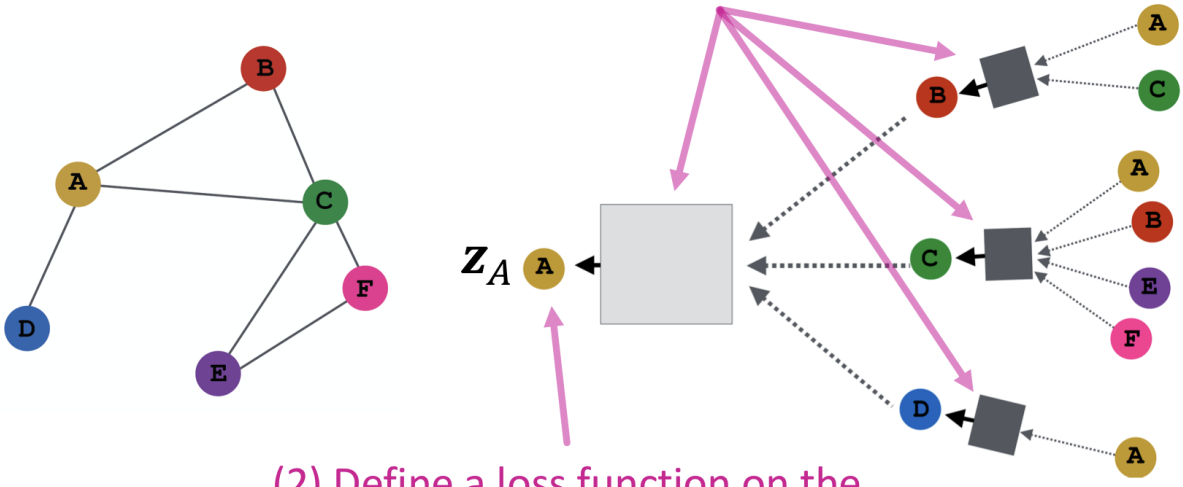
Classification weights

Node class label

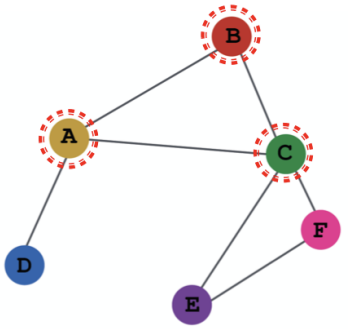
Safe or toxic drug?

Model Design : Overview

(1) Define a neighborhood aggregation function

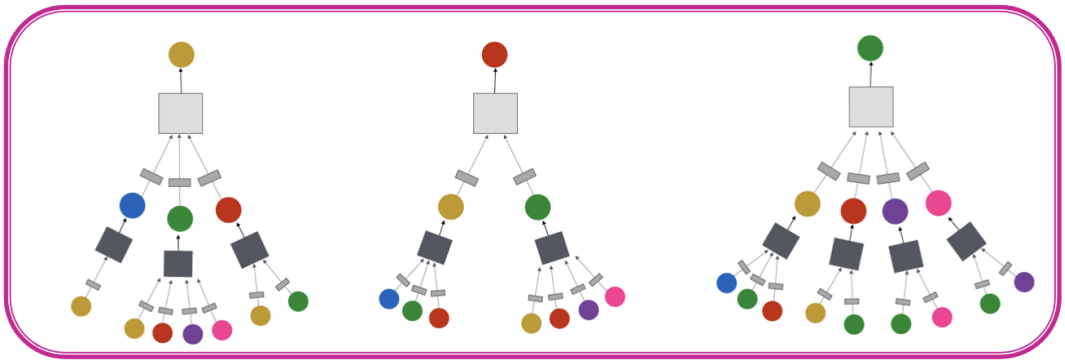


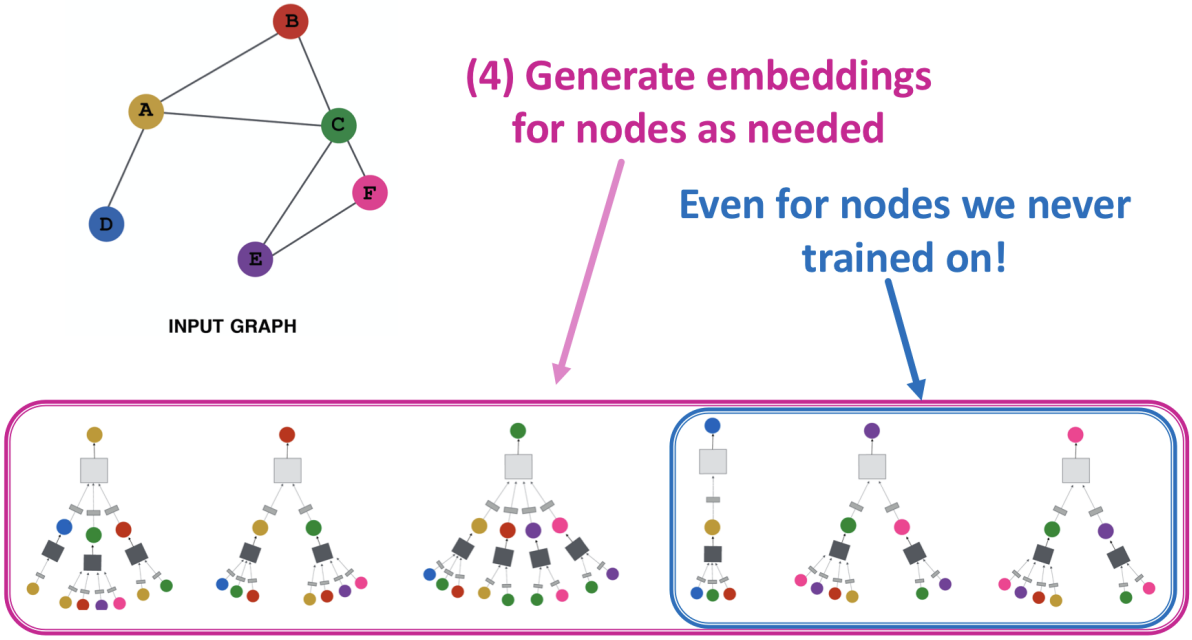
(2) Define a loss function on the embeddings



INPUT GRAPH

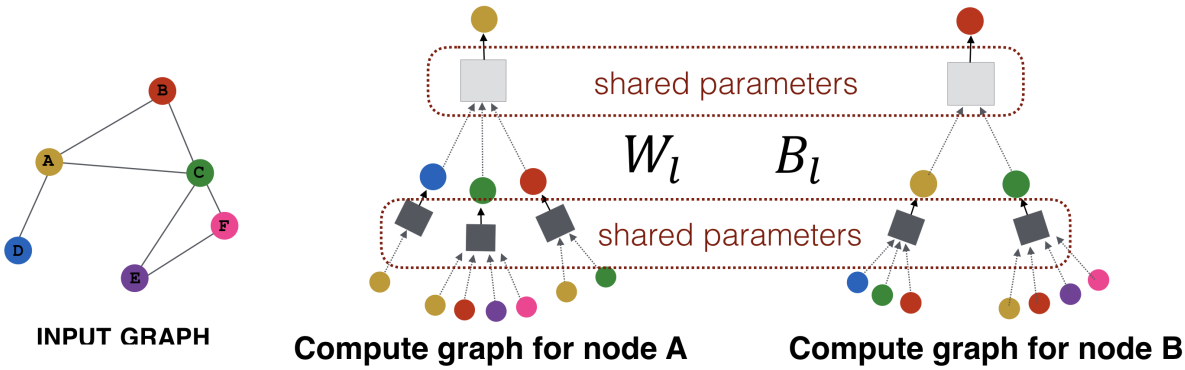
(3) Train on a set of nodes, i.e., a batch of compute graphs





Inductive Capability

- **The same aggregation parameters are shared for all nodes:**
  - The number of model parameters is sublinear in $|V|$ and we can **generalize to unseen nodes!**



Aggregation parameters들이 모든 노드들에 대해 공유된다. 같은 층(layer)의 신경망은 학습 가중치를 공유한다.

- 모델의 parameter들이 노드의 개수에 준선형(sublinear)이다.
- 보지 못한 노드(unseen node)에 일반화가 가능하다.

- 새로운 그래프에서 일반화 가능 ex) 유기체 A의 단백질 상호작용 그래프에서 학습한 모델을 통해 새롭게 취득된 데이터인 유기체 B의 embedding을 생성
- 새로 연결된 노드 embedding 가능 ex) Reddit, YouTube, Google Scholar

Summary

- **Recap** : Generate node embeddings by aggregating neighborhood information
  - We saw a **basic variant of this idea**
  - Key distinctions are in how different approaches aggregate information across the layers
- **Next** : Describe GraphSAGE graph neural network architecture

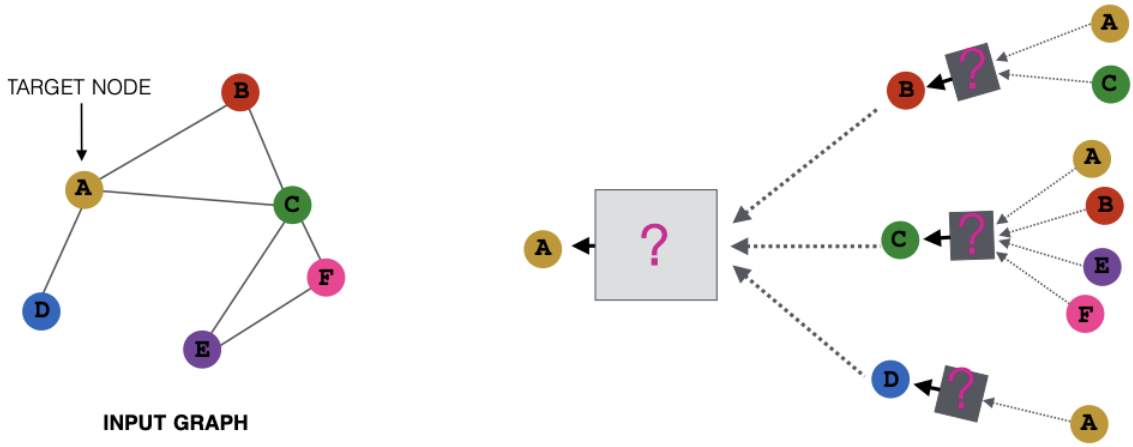
---

3. Graph Convolutional Networks and GraphSAGE

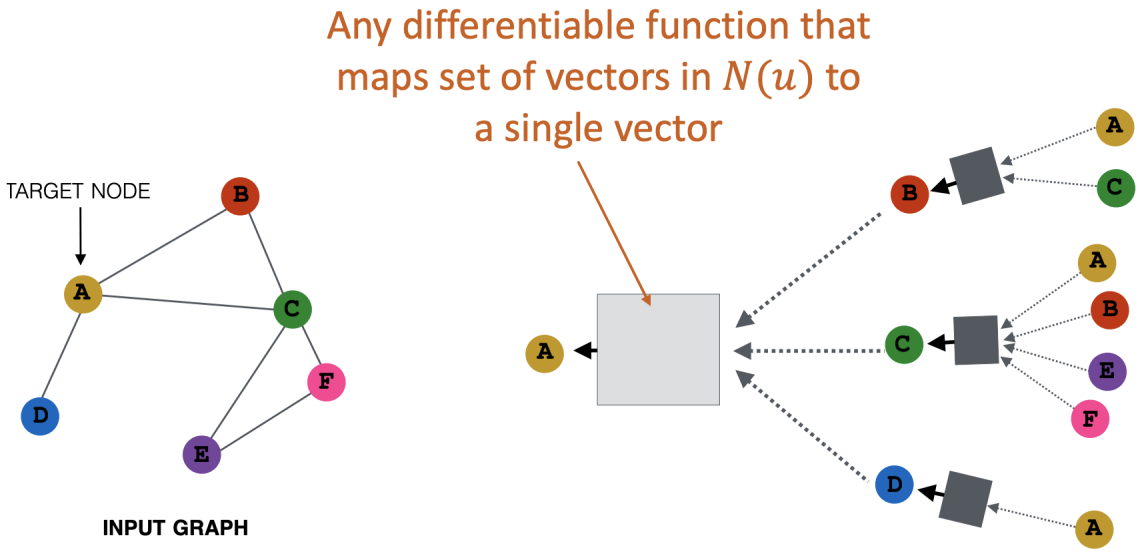
GraphSAGE Idea

So far we have aggregated the neighbor mseesages by taking their (weighted)average

Can we do better?



GraphSAGE Idea (1)



$$h_v^{(l+1)} = \sigma([W_l \cdot \text{AGG}\left(\left\{h_u^{(l)}, \forall u \in N(v)\right\}\right), B_l h_v^{(l)}])$$

How does this message passing architecture differ?

GraphSAGE Idea (2)

$$h_v^{(l+1)} = \sigma([W_l \cdot \text{AGG}\left(\left\{h_u^{(l)}, \forall u \in N(v)\right\}\right), B_l h_v^{(l)}])$$

Optional: Apply L2 normalization to $h_v^{(l+1)}$ embedding at every layer

- ℓ_2 **Normalization:**
  - $h_v^k \leftarrow \frac{h_v^k}{\|h_v^k\|_2}$ $\forall v \in V$ 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

Neighborhood Aggregation

- Simple neighborhood aggregation:

$$h_v^{(l+1)} = \sigma\left(W_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)}\right)$$

- **GraphSAGE:**

Concatenate neighbor embedding and self embedding

$$h_v^{(l+1)} = \sigma\left([W_l \cdot \mathrm{AGG}\left(\left\{h_u^{(l)}, \forall u \in N(v)\right\}\right), B_l h_v^{(l)}]\right)$$

Flexible aggregation function instead of mean

Neightbor Aggregation : Variants

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

$$\mathrm{AGG} = \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|}$$

- **Pool** : Transform neighbor vectors and apply **symmetric** vector function

Element-wise mean/max

$$\mathrm{AGG} = \gamma(\{\mathrm{MLP}(\mathrm{h}_u^{(l)}), \forall u \in N(v)\})$$

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

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

LSTM의 경우 표현력에 있어서 장점을 지니지만 본질적으로 대칭적이지 않기 때문에 permutation invariant하지 않다.

Recap : GCN, GraphSAGE

Key idea : Generate node embeddings based on **local neighborhoods**

- Nodes aggregate “message” from their neighbors using neural networks
- Graph convolutional networks:
  - Basic variant : Average neighborhood information and stack neural networks
- GraphSAGE:
  - Generalized neighborhood aggregation

Summary

- **In this lecture, we introduced**
  - Basics of neural networks
    - Loss, Optimization, Gradient, SGD, non-linearity, MLP
  - Idea for Deep Learning for Graphs
    - Multiple layers of embedding transformation
    - At every layer, use the embedding at previous layer as the input
    - Aggregation of neighbors and self embeddings

- Graph Convolutional Network
  - Mean aggregation; can be expressed in matrix form
- GraphSAGE : more flexible aggregation