Graph Neural Network

Jeong Yujin

ESC, YONSEI UNIVERSITY

February 16, 2022

Table of Contents

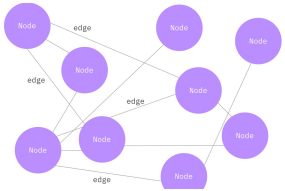
I. Graph

Definition of Graph

Definition of graph:
A graph can be denoted as

$$\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$$

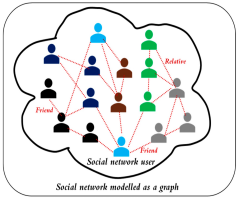
where $\mathcal{V}$=$\{V_1, V_2, ..., V_N\}$ is a set of $\mathcal{N} = |\mathcal{V}|$ nodes and $\mathcal{E} = \{e_1, ..., e_M\}$ is a set of M edges.

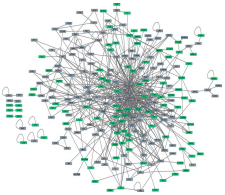


A graph is an abstract data type that can be used to represent complex, non-linear relationships between objects.

Why graph?

1. Graph provide a universal representation of data.
   Data from many systems across various area can be denoted as graphs.



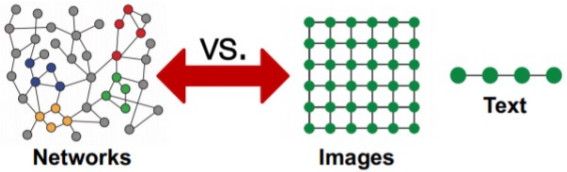


   Graphs show up every where!

2. A vast number of real-world problems can be addressed by a graph.
   - identifying relevant genes to diseases
   - suggesting medicines to patient
   - spammers or terrorists
   - recommendations

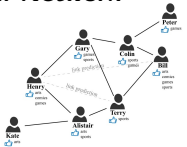
Why GNN?

- Traditional machine learning has been designed for **regular structured data** such as **images** and **sequences**, while graphs are **irregular**.
- The structural information for regular data is simple, while that for graphs is **complicated**.



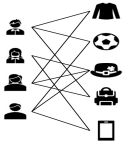
Graph 활용

- **Social Network**



In social graphs, users are viewed as nodes.

- **E-commerce site**

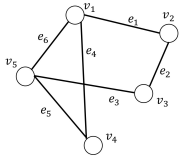


In Amazon, users can interact with items.

Definition of Graph

- **The size of a graph**:
  $N = |\mathcal{V}|$ (number of nodes)

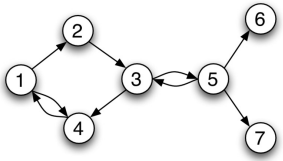
- **edges** $\mathcal{E}$:
  The set of edges describes the connection between nodes.
  An edge $e \in \mathcal{E}$ connects two nodes v_e^1 and v_e^2. Then the edge e can also be represented as $\left(v_e^1, v_e^2\right)$



- **adjacent**:
  A node v_i is adjacent to another node v_j iff there exists an edge between them.

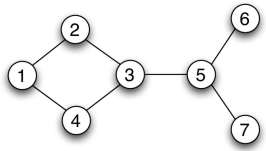
Graph 종류

- **Directed graph**



A graph in which the edges have directions.

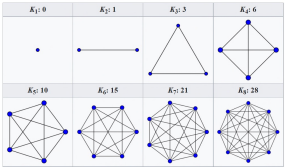
- **Undirected graph**



The edge does not have directions.
$(v_e^1, v_e^2) = (v_e^2, v_e^1)$

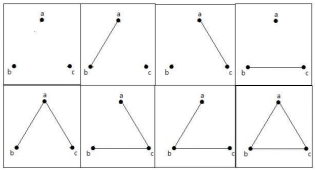
Graph 종류

- **Complete graph**



A simple undirected graph in which every pair of distinct vertices is connected by a unique edge. K_n has $\frac{n(n-1)}{2}$ edges.

- **Incomplete graph**



A graph contains at least one nonadjacent pair of vertices.

Complex Graph

Graphs in real-world applications are much more complicates.

1. **Heterogeneous Graphs**
   Model multiple types of relations between multiple types of nodes.

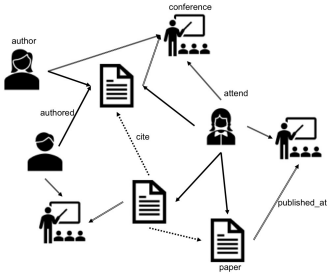
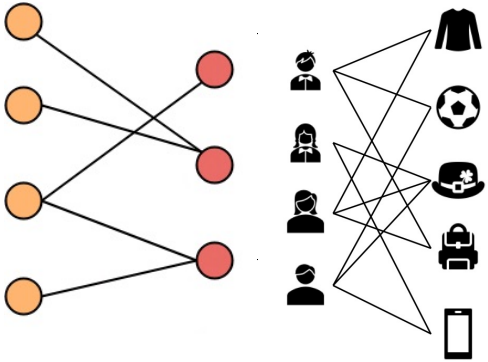


Figure 2.6 A heterogeneous academic graph

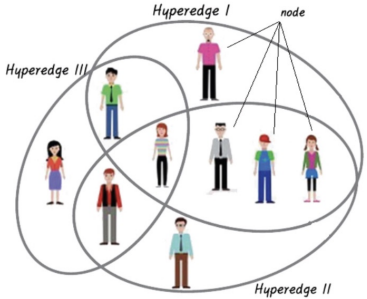
Complex Graph

2. **Bipartite Graphs**
   Bipartite graphs are widely used to capture interactions between different objects.



Complex Graph

3. **Hypergraphs**
   한 edge에 여러 node가 연결된 그래프. 이런 hypergraph 안에서 2개 이상의 node를 연결해주는 edge를 hyperedge라고 한다.



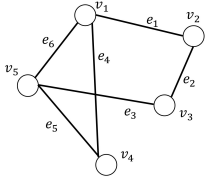
II. Measure

Graph용어

1. **Adjacency Matrix**
   For a given graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$, the corresponding adjacency matrix is denoted as $\mathcal{A} \in \{0,1\}^{N \times N}$

$$\mathcal{A}_{i,j} = \begin{cases} 1 & \text{if v}_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise} \end{cases}$$

In an undirected graph, its corresponding adjancey matrix is **symmetric**.



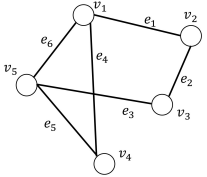
$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Graph용어

Degree

In a graph $\mathcal{G}=\{\mathcal{V},\mathcal{E}\}$, the degree of a node $v_i \in \mathcal{V}$ is the number of nodes that are adjacent to v_i.

$$d(v_i) = \sum_{v_j \in \mathcal{V}} I_{\mathcal{E}}(\{v_i, v_j\})$$



where

$$I_{\mathcal{E}}(\{v_i, v_j\}) = \begin{cases} 1 & \text{if } (\mathsf{v}_i, v_j) \in \mathcal{E} \\ 0 & \text{if } (\mathsf{v}_i, v_j) \notin \mathcal{E} \end{cases}$$

And the degree can be computed as

$$d(v_i) = \sum_{j=1}^{N} \mathbf{A}_{i,j}$$

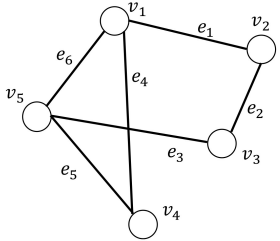
$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Graph용어

3. **Neighbors**
   For a node v_i in graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$, the set of its Neighbors $N(v_i)$ consists of all nodes that are adjacent to v_i.

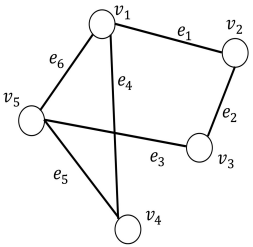
   Note that

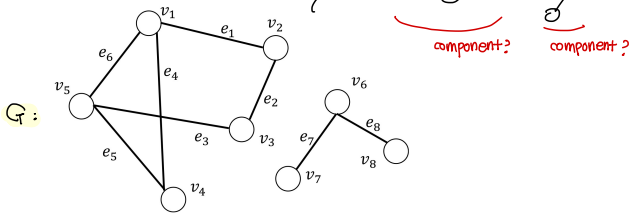
$$d(v_i) = |N(v_i)|$$



Connectivity

- **Connected Graph**:
  A graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ is said to be connected if it has exactly one component.
  Any two vertices can be joined by a path.
  - **connected component**:
    maximal connected subgraph.





Centrality

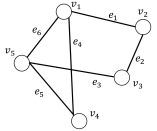
In a graph, the centrality measures the **importance of the node** in the graph.

There are different ways to measure node importance.

1. **Degree Centrality**:
   Measure the centrality of a given node based on its **degree**
   A node can be considered as important if there are many other nodes connected to it.
   - node v_i의 **degree centrality**

$$c_d(v_i) = d(v_i) = \sum_{j=1}^{N} A_{i,j}$$

   - Example:



Centrality

While the degree based centrality considers a node with many neighbors as important, it treats all the neighbors equally.
However, the neighbors can have **different importance**.

2. **Eigenvector Centrality**

   Defines the centrality score by considering the centrality scores of its neighboring nodes as:

$$c_e(v_i) = \frac{1}{\lambda} \sum_{j=1}^{N} \mathbf{A}_{i,j} \cdot c_e(v_j)$$

   which can be rewritten in a matrix form as:

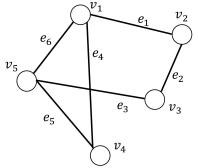
$$\mathbf{c}_e = \frac{1}{\lambda} \mathbf{A} \cdot \mathbf{c}_e$$

   where $\mathbf{c}_e \in \mathbf{R}^N$ is a vector containing the centrality scores of all nodes in the graph.

Centrality

$$\lambda \cdot \mathbf{c}_e = \mathbf{A} \cdot \mathbf{c}_e$$

$\mathbf{c}_e$ is an eigenvector of the matrix $\mathbf{A}$ with its corresponding eigenvalue λ.
A real squared matrix with positive elements has a unique largest eigenvalue and its corresponding eigenvector has all positive elements.
Thus, we can choose λ as the largest eigenvalue and its corresponding eigenvector as the **centrality score vector**.



$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

- largest eigen value: 2.48
  its corresponding eigenvector.

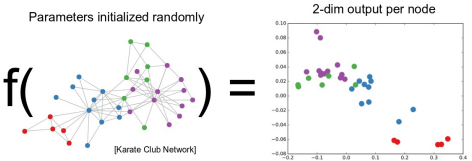
$$\begin{bmatrix} 1 \\ 0.675 \\ 0.675 \\ 0.506 \\ 1 \end{bmatrix} \begin{matrix} : Ce(v_1) \\ : Ce(v_2) \\ : Ce(v_3) \\ : Ce(v_4) \\ : Ce(v_5) \end{matrix}$$

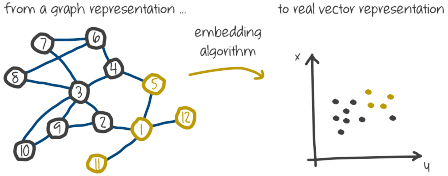
III. Node Embeddings

Node Embeddings(Graph Embeddings)

Node Embedding:
a way of representing nodes as **vectors**.

- Graph에 machine learning기법을 집적 적용하는 것은 한계가 있음.
- 공간효율이 좋음. node개수가 N개라면, adjacency matrix의 크기는 NxN. 반면에 정해진 차원에 graph를 embedding을 하게 되면 공간 절약이 가능.



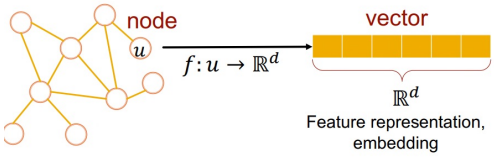


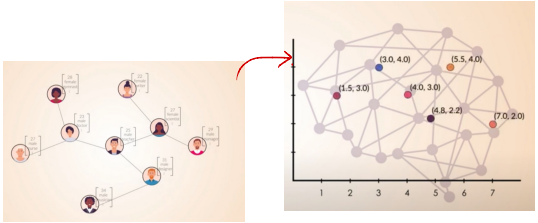
Node Embeddings

Goal of Node Embedding:

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

- similarity in the embedding space approximates similarity in the network!



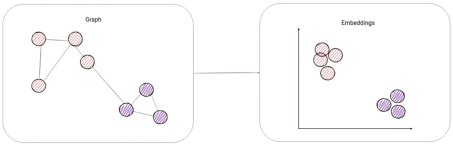


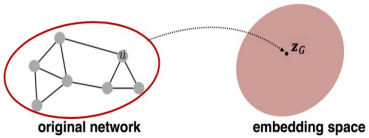
Node Embeddings

A node in a graph can be viewed from two domains.

1. **Original graph domain**
   Nodes are connected via edges.
2. **Embedding domain**
   Each node is represented as a continuous vector.

The information in the graph domain can be preserved in the embedding domain.

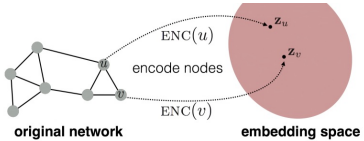




Node Embedding

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

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

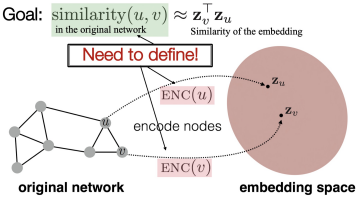


where v is node in the input graph and $\mathbf{z}_v$ is d-dimensional embedding.

- **Similarity function**:

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

Graph domain에서 node u와 v의 similarity ≈ embedding space에서 두 node의 similarity 수식을 내적, 거리로 정의함



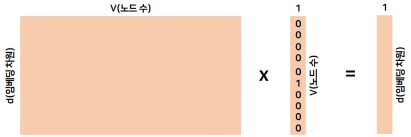
Node Embedding

- **Decoder**:
  Maps from embeddings to the similarity score.
  Optimize the parameters such that the decoded similarity corresponds as closely as possible to the underlying definition of the network similarity.

Node Embedding

Shallow Encoding:

Encoder is just an embedding-look up!

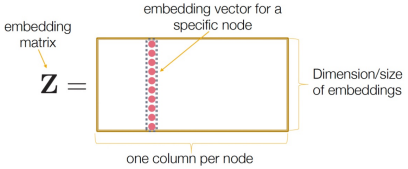


$$ENC(v) = \mathbf{z}_v = Z \cdot v$$

where

$$\mathbf{Z} \in \mathcal{R}^{d \times |\mathcal{V}|}$$

$$v \in I^{|\mathcal{V}| \times 1}$$



We have to learn **Z**!

Node Embeddings

Algorithms are vary according to the information they attempt to preserve.

- node co-occurrence
- structural role
- node status
- community structure

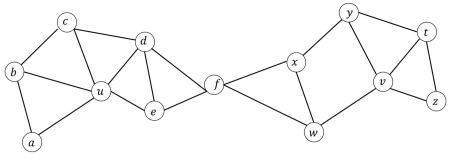
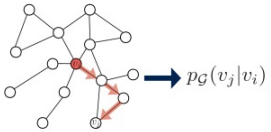


Figure 4.3 An illustration of two nodes that share similar structural role

Node co-occurrence

One of the most popular ways to extract node co-occurence in a graph is via performing **random walks**. Nodes are considered similar to each other if they tend to co-occur in **random walks**.

Given a starting node $v^{(0)}$ in a graph $\mathcal{G}$, we randomly walk to one of its neighborhood. We repeat this process from the node until T nodes are visited.



Definition of Random Walk:

$$p(v^{(t+1)}|v^{(t)}) = \begin{cases} \frac{1}{d(v^{(t)})}, & \text{if } v^{(t+1)} \in N(v^{(t)}) \\ 0, & \text{otherwise} \end{cases}$$

Node co-occurence

Random walks have been employed as a **similarity measure** in various tasks.
We detail the process of generating the set of random walks and extracting co-occurrence for them.

- Encoder: look-up
- $\mathbf{z}_u^T \mathbf{z}_v \approx P_R(v|u)$, (random walk에서 u와 v가 동시에 등장할 확률)
- $\mathbf{z}_u^T \mathbf{z}_v =$ embedding space에서의 similarity.
- $P_R(v|u) =$ graph에서의 similarity.

Node co-occurrence

- Given $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$
- Goal: to learn a mapping f: $u \rightarrow \mathcal{R}^d$
- Random walk 진행과정
  1. Run short fixed-length random walks starting from each node u in the graph using random walk strategy R
  2. For each node u, collect $N_R(u)$
  3. Optimize embedding according to:

$$\max \sum_{u \in \mathcal{V}} \log P(N_R(u) | \mathbf{z}_u)$$

Node co-occurrence

Equivalently,

$$\mathcal{L} = \sum_{u \in \mathcal{V}} \sum_{v \in N_R(u)} -\log(P(v|\mathbf{z}_u))$$

where

$$P(v|\mathbf{z}_u) = \frac{\exp(\mathbf{z}_u^T \mathbf{z}_v)}{\sum_{n \in \mathcal{V}} \exp(\mathbf{z}_u^T \mathbf{z}_n)}$$

Optimize embedding $\mathbf{z}_u$ to maximize the likelihood of random walk co-occurrences.

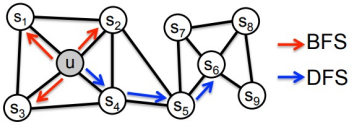
Node2Vec

지금까지 사용한 fixed-length unbiased random walks....너무 제약적인 것이 아닌가??

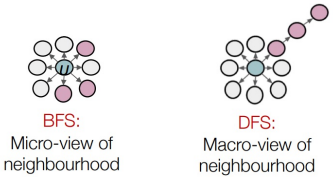
- Can we use **richer** random walk? : **Node2Vec**!
- flexible notion of neighborhoods leads to much richer node embeddings.
- We are going to develop a **second-order random walk**.

- Random walk 진행과정
  1. Run short fixed-length random walks starting from each node u in the graph using random walk strategy R
  2. For each node u, collect $N_R(u)$
  3. Optimize embedding according to:

Node2Vec



You can explore the network in different ways in node2vec. **Local & Global**

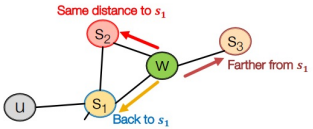


Node2Vec

A probability to sample is defined based on both $v^{(t)}$ and $v^{(t-1)}$.
An **unnormalized probability** to choose the next node is defined as follows.

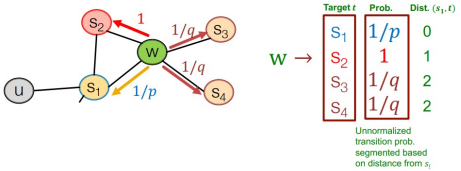
$$\alpha_{pq}\left(v^{(t+1)}|v^{(t-1)}, v^{(t)}\right) = \begin{cases} \frac{1}{p} & \text{if } dis(v^{(t-1)}, v^{(t+1)}) = 0 \\ 1 & \text{if } dis(v^{(t-1)}, v^{(t+1)}) = 1 \\ \frac{1}{q} & \text{if } dis(v^{(t-1)}, v^{(t+1)}) = 2 \end{cases}$$

- p is return parameter.
- q is in-out parameter.



Node2Vec

- If we set a low value of p, the random walk will most likely return.
- If we set a low value of q, the random walk will navigate farther away.



- $N_R(u)$ will be defined by the nodes visited by this **biased random walk!**
- The only difference between **DeepWalk and Node2vec** is how the set of neighbor nodes is defined and how the random walk is defined.

Graph Neural Network

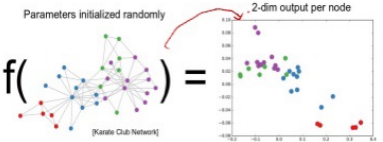
Lee Dohyoung

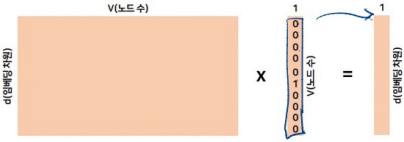
ESC, YONSEI UNIVERSITY

February 17, 2022

Review

- Graph embedding
  - shallow encoding





- GNN
  - deep encoding

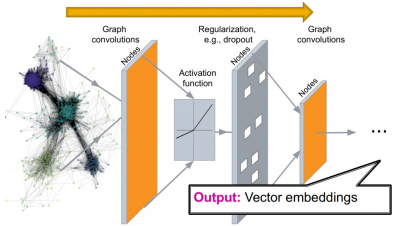


Table of Contents

I. Task

GNN : Graph Neural Network

- Apply deep neural networks to graph-structured data.
- **Node-focused tasks** : nodes as data sample.
  learning good features for each node.
- **Graph-focused tasks** : graphs as data sample.
  learn node feature - learn representative features for the entire graph.

Framework for node focused task

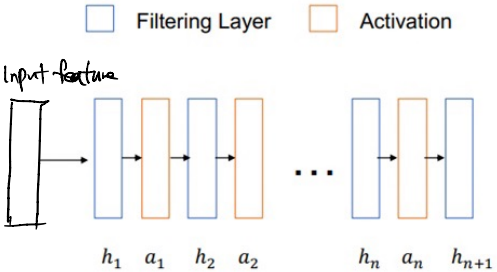


Figure 5.3 A general GNN architecture for node-focused tasks

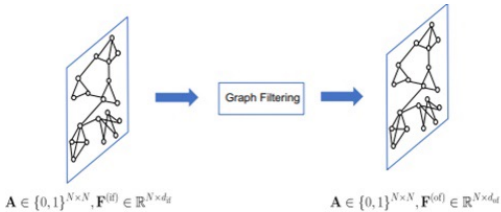


Figure 5.1 Graph filtering operation

Framework for graph focused task

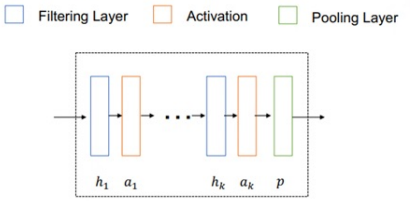


Figure 5.4 A block in GNNs for graph-focused tasks

Figure 5.5 Architectures of GNNs for graph-focused tasks

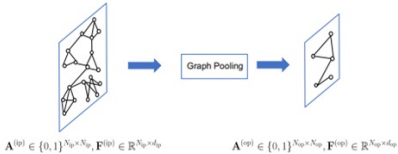


Figure 5.2 Graph pooling operation

II. Graph filtering

Graph filter

- **Spatial-based graph filter**

  explicitly use the graph structure(i.e.,the connections between the nodes) to perform the feature refining process in the graph domain.

- **Spectral-based graph filter**

  utilize spectral graph theory to design the filtering operation in the spectral domain.

Spatial-based graph filter

- The filter in the very first graph neural network. Iteratively updates features of one node by utilizing features of its neighbors.
  For node v_i, its corresponding label can be denoted as l_i.

$$\mathbf{F}_i^{'} = \sum_{v_j \in \mathbf{N}(v_i)} g(l_i, \mathbf{F}_j, l_j)$$

$g()$: parametric function, called **local transition function**, spatially localized.
F : graph feature. cf) **f**: one channel graph.
$\mathbf{F}_i$: i-th row of **F**, node $v_i^{'}$s feature.

Spatial-based graph filter

- **GraphSAGE-Filter**

$$N_S(v_i) = SAMPLE(N(v_i), S)$$
$$f^{'}_{N_S(v_i)} = AGGREGATE(\mathbf{F}_j, \forall v_j \in N_S(v_i))$$
$$\mathbf{F}^{'}_i = \sigma\left(\left[\mathbf{F}_i, \mathbf{f}^{'}_{N_S(v_i)}\right]\Theta\right)$$

where
[,] : concatenation operation
Θ : learnable parameter matrix
$\sigma()$: sigmoid function

Aggerator: mean aggregator, LSTM aggregator, pooling operator.

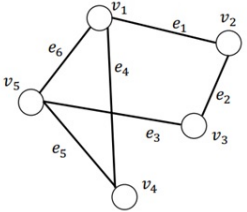
Spatially localized, use 1-hop neighbors.

Spatial-based graph filter

- **GAT-Filter** : Self-attention mechanism is introduced to build spatial graph filters in graph attention model(GAT). Graph filter in GAT. tries to differentiate the importance of the neighbors when performing the aggregation. Spatially localized, 1-hop neighbors.
- **ECC-Filter** : when edge information is available.
- **GGNN-Filter** : when edges are directed and also have different types. Original GNN filter + GRU
- **Mo-Filter** : MoNet perform convolution operations on non-Euclidean data such as graph. Graph filter in MoNet.

Before Spectral-based graph filter...

- Spectral graph theory studies the properties of a graph through analyzing the eigenvalues and eigenvectors of its Laplacian matrix.
- **Laplacian Matrix**
  $\mathbf{L} = \mathbf{D} - \mathbf{A}$ where $\mathbf{D}$=diag$(d(v_1), .., d(v_N))$



$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

$$\mathbf{h} = \mathbf{Lf} \qquad \mathbf{h}[i] = d(v_i) \cdot \mathbf{f}[i] - \sum_{j=1}^{N} \mathbf{A}_{i,j} \cdot \mathbf{f}[j]$$

$$= (\mathbf{D} - \mathbf{A})\mathbf{f} \qquad = d(v_i) \cdot \mathbf{f}[i] - \sum_{v_j \in \mathcal{N}(v_i)} \mathbf{A}_{i,j} \cdot \mathbf{f}[j]$$

$$= \mathbf{Df} - \mathbf{Af}. \qquad = \sum_{v_j \in \mathcal{N}(v_i)} (\mathbf{f}[i] - \mathbf{f}[j]).$$

$$\mathbf{f}^T\mathbf{Lf} = \sum_{v_i \in \mathcal{V}} \mathbf{f}[i] \sum_{v_j \in \mathcal{N}(v_i)} (\mathbf{f}[i] - \mathbf{f}[j])$$

$$= \sum_{v_i \in \mathcal{V}} \sum_{v_j \in \mathcal{N}(v_i)} (\mathbf{f}[i] \cdot \mathbf{f}[i] - \mathbf{f}[i] \cdot \mathbf{f}[j])$$

$$= \sum_{v_i \in \mathcal{V}} \sum_{v_j \in \mathcal{N}(v_i)} (\frac{1}{2}\mathbf{f}[i] \cdot \mathbf{f}[i] - \mathbf{f}[i] \cdot \mathbf{f}[j] + \frac{1}{2}\mathbf{f}[j] \cdot \mathbf{f}[j])$$

$$= \frac{1}{2} \sum_{v_i \in \mathcal{V}} \sum_{v_j \in \mathcal{N}(v_i)} (\mathbf{f}[i] - \mathbf{f}[j])^2.$$

L is positive semi definite.

$\lambda = \lambda \mathbf{u}^T\mathbf{u} = \mathbf{u}^T\lambda\mathbf{u} = \mathbf{u}^T L\mathbf{u} \geq 0$

Eigenvalues of **L** are non-negative.

There always exists an eigenvalue that equals to 0.

Its eigenvector: $\frac{1}{\sqrt{N}}(1, ..., 1)$

Let $0 = \lambda_1 \leq \lambda_2 \leq, ..., \leq \lambda_N$

$\mathbf{f}^T\mathbf{Lf}$: smoothness or frequency of the signal **f**

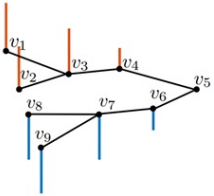
$\mathbf{f}^T\mathbf{Lf} \downarrow$: **f** smooth, frequency $\downarrow$

$\mathbf{f}^T\mathbf{Lf} \uparrow$: **f** not smooth, frequency $\uparrow$

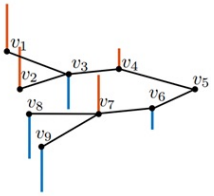
$\lambda = \mathbf{u}^T\mathbf{Lu} \downarrow$: **u** smooth, frequency $\downarrow$

$\lambda = \mathbf{u}^T\mathbf{Lu} \uparrow$: **u** not smooth, frequency $\uparrow$

ex) λ_1=0, u_1=$\frac{1}{\sqrt{N}}(1, ..., 1)$: very smooth!



$f^T Lf = 1$



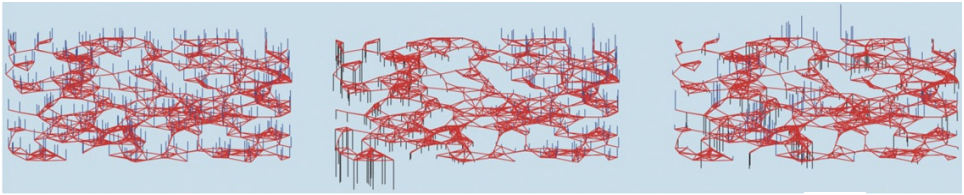
$f^T Lf = 21$

$\mathbf{f}^T\mathbf{Lf}\downarrow$: $\mathbf{f}$ smooth, frequency $\downarrow$

$\mathbf{f}^T\mathbf{Lf}\uparrow$: $\mathbf{f}$ not smooth, frequency $\uparrow$

$\lambda = \mathbf{u}^T\mathbf{Lu}\downarrow$: $\mathbf{u}$ smooth, frequency $\downarrow$

$\lambda = \mathbf{u}^T\mathbf{Lu}\uparrow$: $\mathbf{u}$ not smooth, frequency $\uparrow$



u_1 u_2 u_N

Graph Fourier transform

$\hat{\mathbf{f}} = \mathbf{U}^T\mathbf{f}$

(graph Fourier coefficient), spectral representation

$\mathbf{f} = \mathbf{U}\hat{\mathbf{f}}$

(original signal), spatial representation

$\mathbf{f} \rightarrow$(GFT) $\hat{\mathbf{f}} \rightarrow$ (IGFT) $\mathbf{f}$

Spectral-based graph filter

- Keep/amplify some of frequency components while remove/diminish the others.
- Given a graph signal f, apply GFT to obtain its graph fourier coefficients, and modulate these coefficients before reconstructing the signal in the spatial domain.

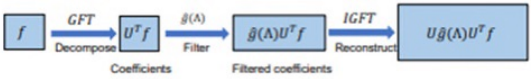


Figure 5.6 The Process of Spectral Filtering

$\mathbf{L}=\mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^{\mathbf{T}}$ (spectral decomposition)

$\hat{\mathbf{f}} = \mathbf{U}^{\mathbf{T}}\mathbf{f}$ (Fourier coefficients in spectral domain)

$\widehat{\mathbf{f}'} = \gamma(\boldsymbol{\Lambda})\,\mathbf{U}^{\mathbf{T}}\mathbf{f}$ (apply function $\gamma()$ to Fourier coefficients)

$\mathbf{f}' = \mathbf{U}\,\widehat{\mathbf{f}'} = \mathbf{U}\,\gamma(\boldsymbol{\Lambda})\,\mathbf{U}^{\mathbf{T}}\mathbf{f}$ (back to the spatial domain)

$$\Lambda = \begin{pmatrix} \lambda_1 & \cdots & 0 \\ 0 & \ddots & 0 \\ 0 & \cdots & \lambda_N \end{pmatrix}$$

Example

Want to smooth noisy graph signal $\mathbf{y}=\mathbf{f}_0+\eta$ (f_0: original signal, η : noise)

arg $\min_f \| \mathbf{f}-\mathbf{y} \|^2+c\mathbf{f}^T\mathbf{L}\mathbf{f}$,
where c: constant >0, objective: convex

$\frac{\partial obj}{\partial \mathbf{f}} = 0$
$\rightarrow 2(\mathbf{f}-\mathbf{y})+2c\mathbf{L}\mathbf{f}= 0$
$\rightarrow (\mathbf{I}+c\mathbf{L})\mathbf{f}= \mathbf{y}$
$\rightarrow (\mathbf{U}\mathbf{U}^T+c\mathbf{U}\Lambda\mathbf{U}^T)\mathbf{f}= \mathbf{y}$
$\rightarrow \mathbf{U}(\mathbf{I}+c\Lambda)\mathbf{U}^T\mathbf{f}= \mathbf{y}$
$\rightarrow \mathbf{f}^{'} = \mathbf{U}(\mathbf{I}+c\Lambda)^{-1}\mathbf{U}^T\mathbf{y}$

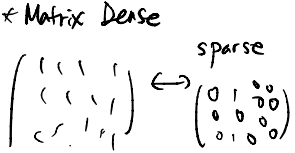
$\rightarrow \gamma(\Lambda)= (\mathbf{I}+c\Lambda)^{-1}$

$\gamma(\lambda_l) = \frac{1}{1+c\lambda_l}$: low-pass filter, smoothing

Spectral-based graph filter

- We often do not know which frequencies are more important. We can model $\gamma(\lambda)$ and learn the parameters from data.
- Natural attempt: learn N, require lots of memory, data. Dense, not spatially localized. Computational cost: spectral decomposition, matrix multiplication.

$$\gamma(\lambda_l) = \theta_l,\ \gamma(\Lambda) = \begin{pmatrix} \theta_1 & \cdots & 0 \\ 0 & \ddots & 0 \\ 0 & \cdots & \theta_N \end{pmatrix}$$



Spectral-based graph filter

- Poly-Filter(polynomial filter): learn K, K-hop neighborhood of nodes. No need spectral decomposition. Localized in the spatial domain.

$$\gamma(\lambda_l) = \sum_{k=0}^{K} \theta_k \lambda_l^k$$

$$\gamma(\mathbf{\Lambda}) = \sum_{k=0}^{K} \theta_k \mathbf{\Lambda}^k$$

$$\begin{aligned}
\mathbf{f}' &= \mathbf{U}\,\gamma(\mathbf{\Lambda})\,\mathbf{U}^{\mathrm{T}}\mathbf{f} \\
&= \mathbf{U} \sum_{k=0}^{K} \theta_k \mathbf{\Lambda}^k\ \mathbf{U}^{\mathrm{T}}\mathbf{f} \\
&= \sum_{k=0}^{K} \theta_k \mathbf{U}\,\mathbf{\Lambda}^k\ \mathbf{U}^{\mathrm{T}}\mathbf{f} \\
&= \sum_{k=0}^{K} \theta_k \mathbf{L}^k\ \mathbf{f}
\end{aligned}$$

$$\begin{aligned}
\mathbf{U}\,\mathbf{\Lambda}^k\ \mathbf{U}^{\mathrm{T}} &= \mathbf{U}\,(\mathbf{\Lambda}\,\mathbf{U}^{\mathrm{T}}\mathbf{U})^k\mathbf{U}^{\mathrm{T}} \\
&= \left(\mathbf{U}\,\mathbf{\Lambda}\ \mathbf{U}^{\mathrm{T}}\right) \ldots \left(\mathbf{U}\,\mathbf{\Lambda}\ \mathbf{U}^{\mathrm{T}}\right) \\
&= \mathbf{L}^k
\end{aligned}$$

Spectral-based graph filter

$$\mathbf{f}' = \sum_{k=0}^{K} \theta_k \mathbf{L}^k \ \mathbf{f}$$

$$\mathbf{f}'[i] = \sum_{v_j \in V} \left(\sum_{k=0}^{K} \theta_k \mathbf{L}^k_{i,j} \right) \mathbf{f}[j] \quad (general\ form)$$

$$\mathbf{f}'[i] = b_{i,i}\mathbf{f}[i] + \sum_{v_j \in \boldsymbol{N}^K(v_i)} b_{i,j}\mathbf{f}[j] \quad (spatial\ based!) \qquad \text{By } \mathbf{L}^k_{i,j} = 0 \ if \ dis(v_i, v_j) > k$$

$\boldsymbol{N}^K(v_i)$: all the nodes that are within K-hop neighborhood of the node v_i

$$b_{i,j} = \sum_{k=dis(v_i,v_j)}^{K} \theta_k \mathbf{L}^k_{i,j}$$

Lemma 5.2 *Let $\mathcal{G}$ be a graph and $\mathbf{L}$ be its Laplacian matrix. Then, the i, j-th element of the k-th power of the Laplacian matrix $\mathbf{L}^k_{i,j} = 0$ if $dis(v_i, v_j) > k$.*

Proof We prove this Lemma by induction. When $k = 1$, by the definition of the Laplacian matrix $\mathbf{L}$, we naturally have that $\mathbf{L}_{i,j} = 0$ if $dis(v_i, v_j) > 1$. Assume for $k = n$, we have that $\mathbf{L}^n_{i,j} = 0$ if $\text{dis}(v_i, v_j) > n$. We proceed to prove that for $k = n + 1$, we have $\mathbf{L}^{n+1}_{i,j} = 0$ if $\text{dis}(v_i, v_j) > n + 1$. Specifically, the element $\mathbf{L}^{n+1}_{i,j}$ can be represented using $\mathbf{L}^n$ and $\mathbf{L}$ as:

$$\mathbf{L}^{n+1}_{i,j} = \sum_{h=1}^{N} \mathbf{L}^n_{i,h}\mathbf{L}_{h,j}.$$

We next show that $\mathbf{L}^n_{i,h}\mathbf{L}_{h,j} = 0$ for $h = 1, \ldots, N$, which indicates that $\mathbf{L}^{n+1}_{i,j} = 0$.

If $\mathbf{L}_{h,j} \neq 0$, then $\text{dis}(v_h, v_j) \leq 1$, i.e., either $h = j$ or there is an edge between node v_i and node v_j. If we have $d(v_i, v_h) \leq n$, then, with $\text{dis}(v_h, v_j) \leq 1$, we have $\text{dis}(v_i, v_j) \leq n + 1$, which contradicts the assumption. Hence, $\text{dis}(v_i, v_h) > n$ must hold. Thus, we have $\mathbf{L}^n_{i,h} = 0$, which means $\mathbf{L}^n_{i,h}\mathbf{L}_{h,j} = 0$.

If $\mathbf{L}_{h,j} = 0$, then $\mathbf{L}^n_{i,h}\mathbf{L}_{h,j} = 0$ also holds. Therefore, $\mathbf{L}^{n+1}_{i,j} = 0$ if $\text{dis}(v_i, v_j) > n + 1$, which completes the proof. □

Spectral-based graph filter

- But the basis of the polynomial $(I, L, L^2, ...)$ is not orthogonal to each other. Coefficients mutually dependent. An update in one coefficient may lead to changes in other coefficients.
- Cheby-Filter : based on Chebyshev polynomial $T_k(y)$ orthogonal to each other.
  Poly filter advantage + more stable learning.

- $T_k(y) = 2yT_{k-1}(y) - T_{k-2}(y)$
- $T_k(y) = \cos(k \arccos(y))$

$$\int_{-1}^{1} \frac{T_l(y)T_m(y)}{\sqrt{1-y^2}} dy = \begin{cases} \frac{\delta_{l,m}\pi}{2} & if\ m, l > 0 \\ \pi & if\ m, l = 0 \end{cases}$$

- GCN-filter: simplified Cheby-filter(K=1, $\lambda_{\max} \approx 2$)
- Graph filter for multi-channel graph signal

III. Graph pooling

Graph pooling

- Flat graph pooling : Usually ignore the hierarchical graph structure
- Hierarchical graph pooling : preserve the hierarchical graph structural information by coarsening the graph step by step

  -Downsampling-based pooling
  -Supernode-based pooling

IV. Parameter learning

Parameter learning for node classification

- For node classification
- v_l with labels, v_u without labels.
- Goal: learn a model based on the labeled nodes v_i to predict the labels of the unlabeled nodes in v_u
- GNN model usually takes the entire graph as input to generate node representations, which are then utilized to train a node classifier

- minimize

$$minimize \quad L_{train} = \sum_{v_i \in V_I} l(f_{GNN}(\mathbf{A}, \mathbf{F}; \mathbf{\Theta})_i, y_i)$$

Parameter learning for graph classification

- For graph classification
- Each graph is treated as a sample with an associated label
- Goal: train a model on the training set D such that it can perform good predictions on unlabeled graphs
- minimize

$$minimize\ L_{train} = \sum_{G_i \in D} l(f_{GNN}(G_i; \mathbf{\Theta})_i, y_i)$$

- Simple: static graph, one type of nodes, one type of edges, simple 경우 외에도 다른 complex graph 에도 GNN 적용 가능하다
- 다양한 GNN
  RNN 활용 GNN, Graph Auto Encoder, GAN(Generative Adversarial Networks) on Graphs, GAT(Graph Attention Networks) 등등..
- HW: GNN 응용 사례 찾아보기

Reference

Ma, Y., Tang, J (2021). Deep Learning on Graphs.Cambridge University press. Ch.2,4,5
libgen link: http://library.lol/main/1BEBBF5A91CE94A5F2D5C50935BBAC9E
https://web.media.mit.edu/ xdong/talk/BDI$_G SP.pdf$