Spectral GCN and Laplacian matrix

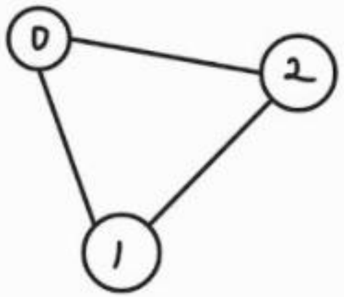
Summary

1. Spectral GCN은 Signal Processing에서 사용하는 Fourier transform을 활용하여 convolution theory와 연결하여 사용
2. Fourier transform을 위해 사용하는 것이 Laplacian operator
3. Graph Laplacian operator는 두 노드 간의 거리를 나타내는 방법이다.
4. Laplacian matrix 에 eigen decomposition 을 사용하여 eigenvalue와 eigenvector를 활용한다.
5. Fourier transform 에 사용하는 eigenvector를 Laplacian matrix 에서 구한 eigenvector로 사용한다.
6. Convolution은 Fourier transform의 곱과 같으므로 convolution 연산을 Fourier transform을 활용하여 계산 할 수 있다.

Laplacian Operator

- Graph에서 laplacian operator 를 적용하면 각 노드와 이웃 노드 간의 차이를 나타내게 된다.
- Laplacian matrix (L) = Degree matrix (D) – Adjacency matrix (A)

Graph (G)



$$A \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \quad D \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

$$\therefore L = D - A \begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix}$$

$$Lx = \begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix}$$

$$= \begin{bmatrix} 2X_1 - X_2 - X_3 \\ -X_1 + 2X_2 - X_3 \\ -X_1 - X_2 + 2X_3 \end{bmatrix} = \begin{bmatrix} (X_1 - X_2) + (X_1 - X_3) \\ (X_2 - X_1) + (X_2 - X_3) \\ (X_3 - X_1) + (X_3 - X_2) \end{bmatrix} = \begin{bmatrix} \sum (X_1 - X_i) \\ \sum (X_2 - X_1) \\ \sum (X_3 - X_i) \end{bmatrix}$$

Eigen decomposition

- L matrix 계산 후 eigen decomposition 을 통해 eigenvalue와 eigenvector를 계산 (Fourier analysis의 basis 공유하기 위해)
- $L = U\Lambda U^T$
- $U\epsilon R^{n*n}$: eigen matrix
- Λ : 대각원소가 오름차순으로 정렬된 eigenvalue 를 가진 n*n 대각행렬
- Graph에서 spectrum은 노드 간의 차이를 나타내기 때문에 $\lambda_1 < \lambda_2 < \cdots < \lambda_n$ 과 같이 spectrum의 값이 작도록 오름차순 정렬하여 사용

Fourier Transform

$$\mathscr{F}\{f\}(v) = \int_{\mathbb{R}} f(x) e^{-2j\pi vx} dx \tag{3}$$

- Fourier transform은 time domain의 신호를 frequency domain으로 변환
- $e^{j2\pi vx}$: frequency가 v인 주기함수 성분
- f(v) : 해당 주기함수 성분의 계수(coefficient) 혹은 강도(amplitude)

Convolution Theorem

- Convolution in spatial domain is equivalent to multiplication in Fourier domain

$$H(z) = \int_R f(x)\, g(z-x)\, dx$$

$$F\{f*g\}(v) = F\{h\}(v)$$

$$= \int_R h(z)\, e^{-2\pi zv}\, dz$$

$$= \int_R \int_R f(x)\, g(z-x)\, e^{-2j\pi vz}\, dx\, dz$$

$$= \int_R f(x) \left(\int_R g(z-x)\, e^{-2j\pi vz}\, dz \right) dx \qquad u = z - x$$

$$= \int_R f(x) \left(\int_R g(u)\, e^{-2\pi v(u+x)}\, du \right) dx$$

$$= \int_R f(x)\, e^{-2j\pi vx}\, dx \int_R g(u)\, e^{-2k\pi vu}\, du$$

⇒ convolution 연산은 Fourier Transform 의 multiplication으로 표현 가능.

Graph domain convolution

$$f * g = \mathcal{F}^{-1}\{\mathcal{F}\{f\} \cdot \mathcal{F}\{g\}\}$$

$$g_\theta * x = \mathcal{F}^{-1}(\mathcal{F}(x) \odot \mathcal{F}(g)) = U(U^T x \odot U^T g)$$

Laplacian.

$$\underset{\text{Signal}}{x} \longrightarrow \underset{\text{Graph Fourier Transform}}{U^T x} \longrightarrow \underset{\text{Filtering}}{(g_\theta^*)U^T x} \longrightarrow \underset{\text{Inverse Graph Fourier Transform}}{\boxed{(U) g_\theta^* U^T} x}$$

$$\underbrace{g_\theta}_{\text{Filter}} * \underbrace{x}_{\text{signal}} = U g_\theta^* U^T x$$

$$g_\theta = \begin{bmatrix} \hat{g}(\lambda_1) & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \hat{g}(\lambda_N) \end{bmatrix}$$

Graph domain 에서 Filter와 signal (node feature)의 convolution은 Laplacian matrix 의 eigen decomposition 과 signal의 곱과 같다.

Polynomial parameterization

$$g_\theta(\Lambda) = \sum_{k=0}^{K} \theta_k \Lambda^k = \theta_0 \Lambda^0 + \theta_1 \Lambda^1 + \cdots + \theta_K \Lambda^K$$

$$\begin{aligned} x * g_\theta &= U \sum_{k=0}^{K} \theta_k \Lambda^k U^T x \\ &= U(\theta_0 \Lambda^0 + \theta_1 \Lambda^1 + \cdots + \theta_K \Lambda^K) U^T x \\ &= \sum_{k=0}^{K} \theta_k L^k x \end{aligned}$$

- 기존 g_θ는 단지 1 hop만 고려 가능
- Localized feature 를 뽑기 위해 polynomial parameterization을 진행.
- *Localized : 이웃노드 (1hop), 이웃노드들의 이웃노드(2 hop) 등 k hops 를 고려하여 이웃 노드의 정보를 고려하여 feature 를 뽑겠다는 뜻

GCN Code

```python
def normalize_adj(adj):
    """
    adjacency matrix를 Symmetrically normalize
    노드의 연결성을 차원(degree)라고 부르는데, 인접 행렬 A의 차원(D)에 대한 역행렬을 곱해주면 정규화를 할 수 있다.

    """
    adj = sp.coo_matrix(adj)
    rowsum = np.array(adj.sum(1)) # D
    d_inv_sqrt = np.power(rowsum, -0.5).flatten() # D^-0.5
    d_inv_sqrt[np.isinf(d_inv_sqrt)] = 0.
    d_mat_inv_sqrt = sp.diags(d_inv_sqrt) # D^-0.5
    return adj.dot(d_mat_inv_sqrt).transpose().dot(d_mat_inv_sqrt).tocoo() # D^-0.5 A D^0.5 를 만들어 준다.


def preprocess_adj(adj):
    """
    위에 만든 normalize_adj를 이용해서 A + I 행렬을 정규화하고 tuple representation으로 변환하는 함수를 만든다.

    """
    adj_normalized = normalize_adj(adj + sp.eye(adj.shape[0]))
    return sparse_to_tuple(adj_normalized)
```