SEMI-SUPERVISED CLASSIFICATION WITH GRAPH CONVOLUTIONAL NETWORKS(GCN)

(ICLR 2017)

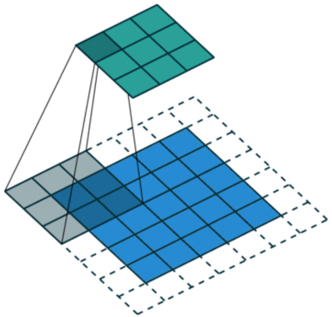
서지민

CONTENTS

BACKGROUND

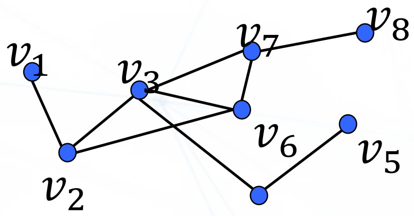
- Convolution in Graph
- Fourier Transform
- Laplacian Matrix
- Chebyshev polynomials

CONVOLUTION



Spatial Locality
Receptive field

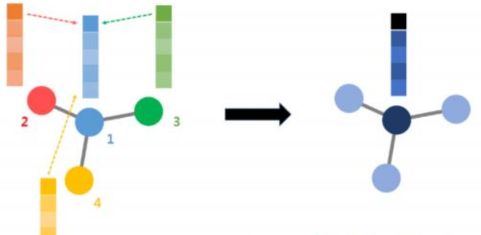
Filter



???

CONVOLUTION

Spatial Convolution



$$H_1^{(l+1)} = \sigma\left(H_1^{(l)}W^{(l)} + H_2^{(l)}W^{(l)} + H_3^{(l)}W^{(l)} + H_4^{(l)}W^{(l)} + b^{(l)}\right)$$

$$\Rightarrow H_i^{(l+1)} = \sigma\left(\sum_{j \in N(i)} H_j^{(l)}W^{(l)} + b^{(l)}\right)$$

인접한 node의 feature들의 가중치로
중심 node 표현

Spectral Convolution

한 node의 signal은 다른 node의 signal들이 혼재된 상태
:시간이 지남에 따라 signal이 흘러 간다.

F(x ⁎ g) = F(x) ⊙ F(g)

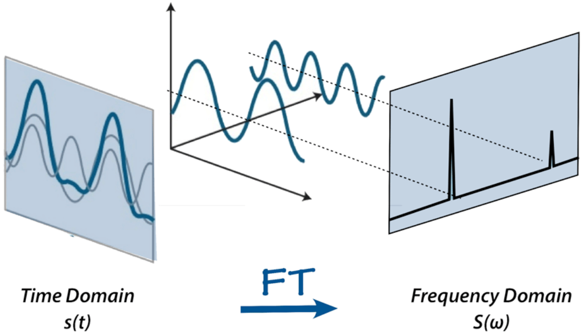
Fourier space

element-wise product on a matrix.

$$\begin{aligned} \mathbf{x} *_G \mathbf{g} &= \mathscr{F}^{-1}(\mathscr{F}(\mathbf{x}) \odot \mathscr{F}(\mathbf{g})) \\ &= \mathbf{U}(\mathbf{U}^T\mathbf{x} \odot \mathbf{U}^T\mathbf{g}), \end{aligned}$$

Graph domain에서 직접 convolution하기보다는
Fourier Transform을 거쳐 다른 domain에서 연산

FOURIER TRANSFORM



Signal을 Frequency 별로 분해하는 과정

$$f(x) = a_0 \frac{1}{2} + \sum_{n=1}^{\infty} a_n \cos nx + \sum_{n=1}^{\infty} b_n \sin nx$$

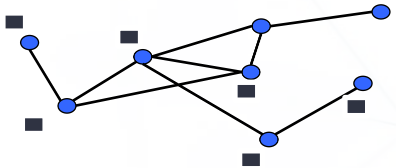
coefficients: *weights (or) amplitudes*

fourier eigen functions (or) fourier basis functions

Orthogonal한 요소(fourier basis)들의 합으로 표현

Graph에서 signal, frequency?

GRAPHS AND GRAPH SIGNALS



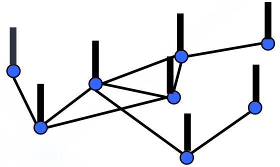
Graph Signal: $f : \mathcal{V} \rightarrow \mathbb{R}^N$

$$\mathcal{V} = \{v_1, \ldots, v_N\}$$

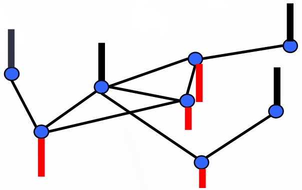
$$\mathcal{E} = \{e_1, \ldots, e_M\}$$

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

$$\mathcal{V} \longrightarrow \begin{bmatrix} f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \\ f(8) \end{bmatrix}$$



Low frequency graph signal

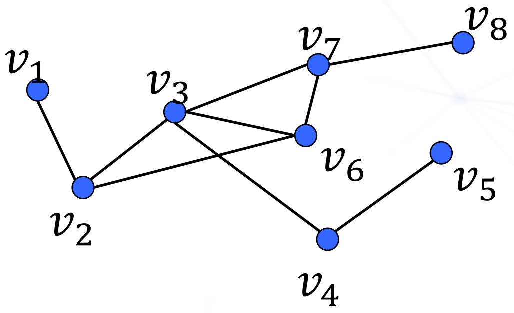


High frequency graph signal

Graph에서 signal이란 node가 가지고 있는 정보
(Social network graph에서 각 node가 사용자라면, 나이, 주소 등)

Frequency는 인접한 node들 간의 차이 정도
(인접한 node의 나이가 비슷하다면 low frequency, 급하게 변화한다면 high frequency)

MATRIX REPRESENTATIONS OF GRAPHS



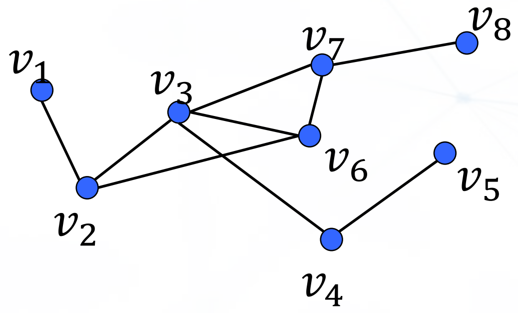
Adjacency Matrix: $A[i,j] = 1$ if v_i is adjacent to v_j
$A[i,j] = 0$, otherwise

Degree Matrix: $\mathbf{D} = \mathrm{diag}(degree(v_1), \ldots, degree(v_N))$

Degree Matrix — Adjacency Matrix = Laplacian Matrix

$$\underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}}_{\boldsymbol{D}} - \underbrace{\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}}_{\boldsymbol{A}} = \underbrace{\begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 3 & -1 & 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & 4 & -1 & 0 & -1 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & -1 & 0 & -1 & 4 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 & -1 & 3 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{pmatrix}}_{\boldsymbol{L}}$$

LAPLACIAN MATRIX IS A DIFFERENCE OPERATOR:



$$\mathbf{h} = \mathbf{L}\mathbf{f} = (\mathbf{D} - \mathbf{A})\mathbf{f} = \mathbf{D}\mathbf{f} - \mathbf{A}\mathbf{f}$$

Laplacian Matrix

$$\underbrace{\begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 3 & -1 & 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & 4 & -1 & 0 & -1 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & -1 & 0 & -1 & 4 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 & -1 & 3 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{pmatrix}}_{\boldsymbol{L}} \begin{bmatrix} f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \\ f(8) \end{bmatrix}$$

4 * f(3) – f(2) – f(6) - f(7)
= (f(3) – f(2)) + (f(3) – f(4)) + (f(3) – f(6)) + (f(3) – f(7))

-> 중심 node(v3)에 대한 이웃 노드의 차이로 해석 가능

H가 최소가 되게 하는 f = 연결된 node 간의 차이를 최소화하는 f

EIGEN-DECOMPOSITION OF LAPLACIAN MATRIX

$$\mathbf{L} = \begin{bmatrix} | & & | \\ \mathbf{u}_0 & \cdots & \mathbf{u}_{N-1} \\ | & & | \end{bmatrix} \begin{bmatrix} \lambda_0 & & 0 \\ & \ddots & \\ 0 & & \lambda_{N-1} \end{bmatrix} \begin{bmatrix} \text{—} & \mathbf{u}_0 & \text{—} \\ & \vdots & \\ \text{—} & \mathbf{u}_{N-1} & \text{—} \end{bmatrix}$$

$\boldsymbol{U}$ $\boldsymbol{\Lambda}$ $\boldsymbol{U^T}$

Eigenvalues are sorted non-decreasingly: (Eigenvalue가 크다 = high frequency)

$$0 = \lambda_0 < \lambda_1 \leq \cdots \lambda_{N-1}$$

Graph signal의 fourier transform
= Graph의 Laplacian matrix Eigen-decomposition

$$\boldsymbol{F}(\boldsymbol{f}) = \boldsymbol{U}^T \boldsymbol{f}$$

$$g_\theta \star x = U g_\theta U^\top x$$

filter signal eigenvector

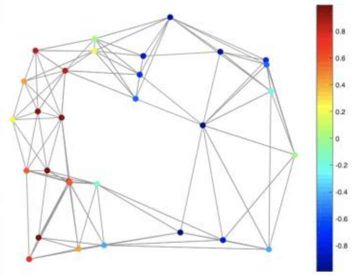
Laplacian matrix의 Eigen vector를 Fourier basis로 사용

GRAPH FOURIER TRANSFORM

A signal f can be written as graph Fourier series:

$$f = \sum_{i=0}^{N-1} \hat{f}_i \cdot u_i$$

u_i: graph Fourier mode



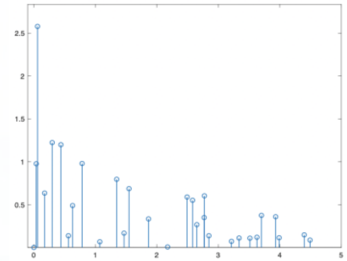
Spatial domain: $\boldsymbol{f}$

filter

$\hat{\boldsymbol{f}} = \boldsymbol{U}^T \boldsymbol{f} \longrightarrow (\boldsymbol{g_\theta})\boldsymbol{U}^T \boldsymbol{f}$

Decompose signal f

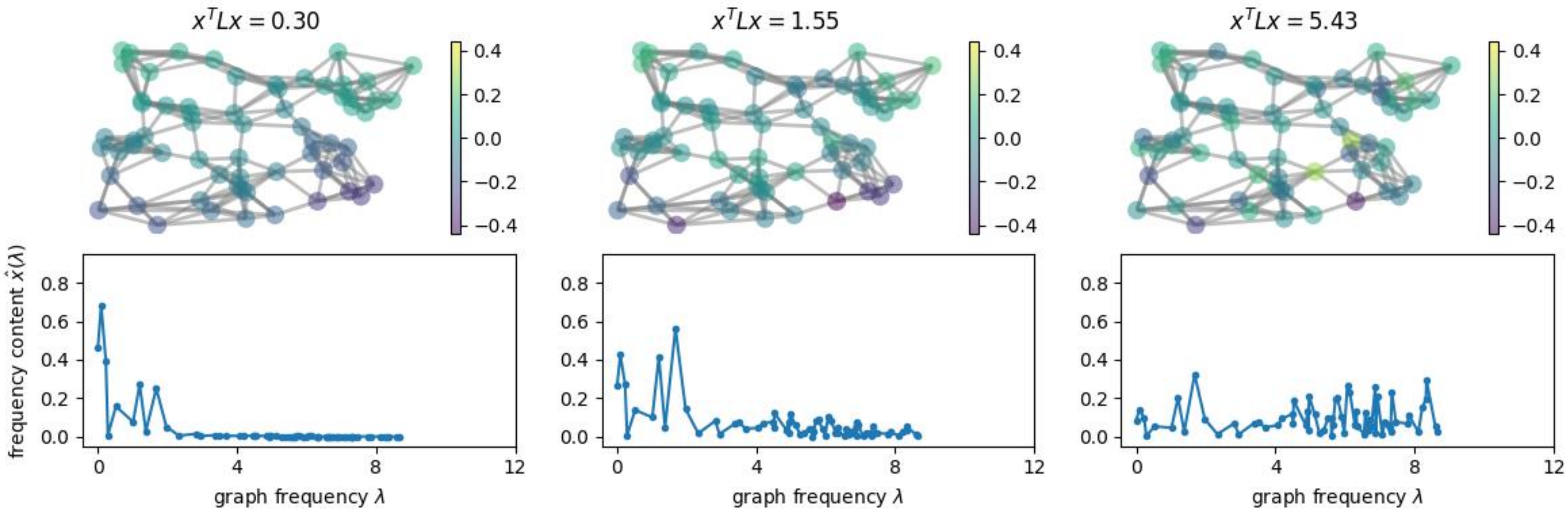
Reconstruct signal $\boldsymbol{f}$



Spectral domain: $\hat{\boldsymbol{f}}$

원하는 frequency를 filtering하여 node embedding

GRAPH FOURIER TRANSFORM



GRAPH SPECTRAL FILTERING FOR GRAPH SIGNAL

$$GFT: \hat{f} = U^T f \qquad IGFT: f = U\hat{f}$$

Filter a graph signal f:

$$f \xrightarrow[\text{Decompose}]{GFT} U^T f \xrightarrow[\text{Filter}]{\hat{g}(\Lambda)} \hat{g}(\Lambda)U^T f \xrightarrow[\text{Reconstruct}]{IGFT} U\hat{g}(\Lambda)U^T f$$

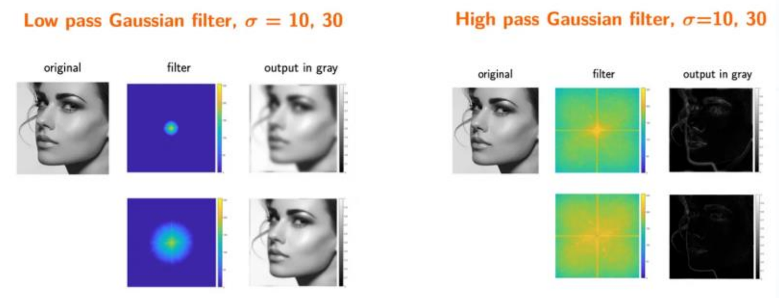
Coefficients

Filtered coefficients

$$U\hat{g}(\Lambda)U^T \Updownarrow \hat{g}(L)$$

$$\hat{g}(\Lambda) = \begin{bmatrix} \hat{g}(\lambda_0) & & 0 \\ & \ddots & \\ 0 & & \hat{g}(\lambda_{N-1}) \end{bmatrix}$$

Lsw pass filter 통과
= 비슷한 노드에서 흘러오는 signal 추출



CONVOLUTION

$g_\theta \ * x$

$= Ug_\theta U^T x$

$= \mathrm{U} \sum_{k=0}^{K} \Theta'_k T_k(\Lambda') U^T \ x$

$= \sum_{k=0}^{K} \Theta'_k T_k(U \Lambda' U^T) \ x$

$= \sum_{k=0}^{K} \Theta'_k T_k(L') \ x$

Chebyshev polynomials $\boldsymbol{T_k(x)}$

$$g_{\theta'}(\Lambda) \approx \sum_{k=0}^{K} \Theta'_k T_k(\Lambda')$$

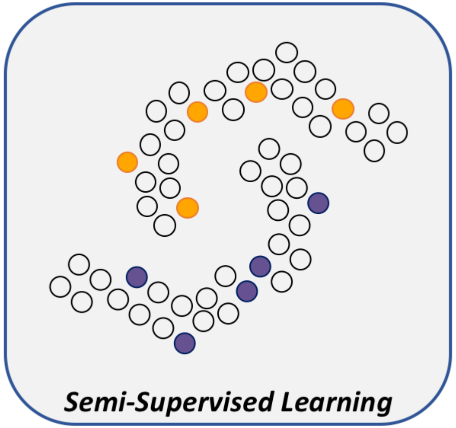
$(T_k(x) = 2T_{k-1}(x) - T_{k-2}(x),\ T_0(x) = 1, T_1(x) = \mathrm{x})$

$(\Lambda' = \frac{2\Lambda}{\Lambda\_max} - I_N,\ -1 \leqq \Lambda' \leqq 1)$

$(L' = \frac{2L}{\Lambda\_max} - I_N)$

Eigen decomposition이 필요 없어짐
-> 연산량 감소

INTRODUCTION



$$\mathcal{L} = \mathcal{L}_0 + \lambda \mathcal{L}_{\text{reg}} , \quad \text{with} \quad \mathcal{L}_{\text{reg}} = \sum_{i,j} A_{ij} \| f(X_i) - f(X_j) \|^2 = f(X)^\top \Delta f(X) .$$

Dataset 중에서 일부의 label만 존재 (semi-supervised)

L0: Supervised Loss

L_reg: connected node가 동일한 label을 가질 것이라고 추측
-> Edge가 담고 있는 similarity 외의 추가적인 정보 학습을 제한

Contribution

1. L_reg를 사용하지 않고 f(X, A)를 학습하는 layer-wise propagation rule 제안

2. 본 논문에서 제시한 모델이 빠르고 효율적인 semi-supervised classification이 가능함을 증명

METHOD

$g_\theta * x = \sum_{k=0}^{K} \Theta'_k T_k(L')\, x$

k번째 이웃 node까지 localized

K = 1로 제한
(layer를 k번 쌓으면 k번째 이웃까지 receptive)

Chebyshev polynomials

$(U\Lambda U^\top)^k = U\Lambda^k U^\top$

$$
\begin{aligned}
T_0(x) &= 1\\
T_1(x) &= x\\
T_2(x) &= 2x^2 - 1\\
T_3(x) &= 4x^3 - 3x\\
T_4(x) &= 8x^4 - 8x^2 + 1\\
T_5(x) &= 16x^5 - 20x^3 + 5x\\
T_6(x) &= 32x^6 - 48x^4 + 18x^2 - 1\\
T_7(x) &= 64x^7 - 112x^5 + 56x^3 - 7x\\
T_8(x) &= 128x^8 - 256x^6 + 160x^4 - 32x^2 + 1\\
T_9(x) &= 256x^9 - 576x^7 + 432x^5 - 120x^3 + 9x
\end{aligned}
$$

$g_\theta * x$

$\approx \Theta'_0 x + \Theta'_1 (\frac{2L}{\Lambda\_max} - I_N) x$ ($\mathbf{\Theta}'_1$은 학습되는 파라미터 -> Λ_max = 2로 근사)

$L = I_N - D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$

$= \Theta'_0 x + \Theta'_1 (L - I_N) x$

$= \Theta'_0 x - \Theta'_1 D^{-1/2} A\, D^{-1/2} x$

$= \Theta\, (I_N + D^{-1/2} A\, D^{-1/2}) x$ $(\Theta = \mathbf{\Theta}'_1 = -\mathbf{\Theta}'_2)$

$$
D^{-1/2} = \begin{pmatrix} \frac{1}{\sqrt{d(1)}} & 0 & \cdots & 0 \\ 0 & \frac{1}{\sqrt{d(2)}} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{1}{\sqrt{d(n)}} \end{pmatrix}
$$

METHOD

$g_\theta * x = \Theta\,(I_N + D^{-1/2} A\, D^{-1/2})$x

Eigen value의 범위가 [0, 2]
=> Layer를 여러 번 쌓으면 vanishing/exploding gradient 발생

$$I_N + D^{-\frac{1}{2}} A D^{-\frac{1}{2}} \rightarrow \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}}$$

$$Z = \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} X \Theta$$

$$\tilde{A} = A + I_N$$
$$\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$$

Self-loop 추가해서 normarlize (renormalization trick)

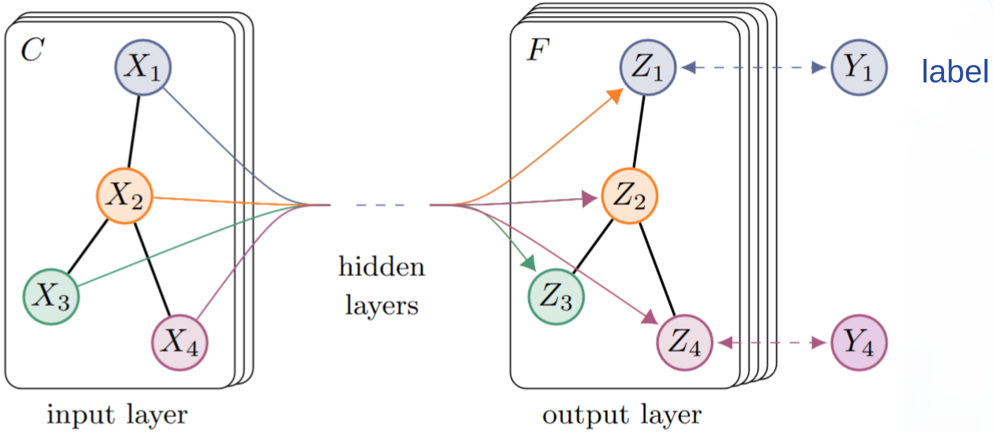
X : (N, C)
Θ : (C, F)
Z : (N, F)

EXPERIMENTS

Two-Layer GCN

$$\hat{A} = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$$

$$Z = f(X, A) = \mathrm{softmax}\left(\hat{A}\ \mathrm{ReLU}\left(\hat{A}XW^{(0)}\right)W^{(1)}\right)$$



(a) Graph Convolutional Network

$$\mathcal{L} = -\sum_{l \in \mathcal{Y}_L} \sum_{f=1}^{F} Y_{lf} \ln Z_{lf}$$

For all labeled examples

RESULTS & ABLATION

Method	Citeseer	Cora	Pubmed	NELL
ManiReg [3]	60.1	59.5	70.7	21.8
SemiEmb [28]	59.6	59.0	71.1	26.7
LP [32]	45.3	68.0	63.0	26.5
DeepWalk [22]	43.2	67.2	65.3	58.1
ICA [18]	69.1	75.1	73.9	23.1
Planetoid* [29]	64.7 (26s)	75.7 (13s)	77.2 (25s)	61.9 (185s)
GCN (this paper)	**70.3** (7s)	**81.5** (4s)	**79.0** (38s)	**66.0** (48s)
GCN (rand. splits)	67.9 ± 0.5	80.1 ± 0.5	78.9 ± 0.7	58.4 ± 1.7

Description		Propagation model	Citeseer	Cora	Pubmed
Chebyshev filter (Eq. 5)	$K = 3$	$\sum_{k=0}^{K} T_k(\tilde{L}) X \Theta_k$	69.8	79.5	74.4
	$K = 2$		69.6	81.2	73.8
1st-order model (Eq. 6)		$X\Theta_0 + D^{-\frac{1}{2}} A D^{-\frac{1}{2}} X \Theta_1$	68.3	80.0	77.5
Single parameter (Eq. 7)		$(I_N + D^{-\frac{1}{2}} A D^{-\frac{1}{2}}) X \Theta$	69.3	79.2	77.4
Renormalization trick (Eq. 8)		$\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} X \Theta$	**70.3**	**81.5**	**79.0**
1st-order term only		$D^{-\frac{1}{2}} A D^{-\frac{1}{2}} X \Theta$	68.7	80.5	77.8
Multi-layer perceptron		$X\Theta$	46.5	55.1	71.4

CONCLUSION

Renormalized propagation model의 성능이 가장 뛰어남

LIMITATION

Kth-order neighbor 계산을 위해서는 K layers가 모두 memor에 올라와야 한다.

Undirected graph에 제한된 모델

Self-connection과 neighbor node에 대한 edge의 중요도를 동일하게 처리

$$\tilde{A} = A + \lambda I_N$$

IMPLEMENTATION

```python
# GCNConv
class GCNConv(nn.Module):
    def __init__(self, input_dimension, output_dimension):
        super(GCNConv, self).__init__()
        self.weight = nn.Parameter(torch.Tensor(input_dimension, output_dimension))
        self.bias = nn.Parameter(torch.Tensor(output_dimension))

        self.reset_parameters()

    def reset_parameters(self):
        nn.init.xavier_uniform_(self.weight)
        nn.init.zeros_(self.bias)

    def forward(self, x, edge_index):
        adj_matrix = self.normalize_adjacency(edge_index, x.size(0)).to(device)
        self.weight = self.weight.to(device)
        self.bias = self.bias.to(device)
        x = x.to(device)

        x = adj_matrix.matmul(x).matmul(self.weight)
        x = x + self.bias

        return x

    def normalize_adjacency(self, edge_index, num_nodes):
        row, col = edge_index
        adj_matrix = torch.eye(num_nodes)
        adj_matrix[row, col] = 1.0
        adj_matrix[col, row] = 1.0
        deg = torch.sum(adj_matrix, dim=1)
        deg_inv_sqrt = 1 / torch.sqrt(deg.clamp(min=1))

        norm = deg_inv_sqrt.view(-1, 1) * adj_matrix * deg_inv_sqrt.view(1, -1)

        return norm
```

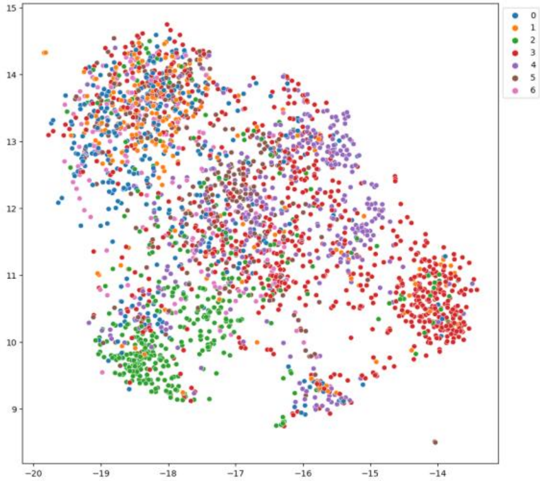
```python
# GCN 2-layer
class GCN(nn.Module):

  def __init__(self):
        super(GCN, self).__init__()
        self.conv1 = GCNConv(dataset.num_features, 16) # num_features -> 16
        self.conv2 = GCNConv(16, dataset.num_classes)       # 16 -> num_classes


  def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = F.dropout(x, training= self.training, p=0.5)
        x = self.conv2(x, edge_index)

        return F.log_softmax(x, dim=1)
```

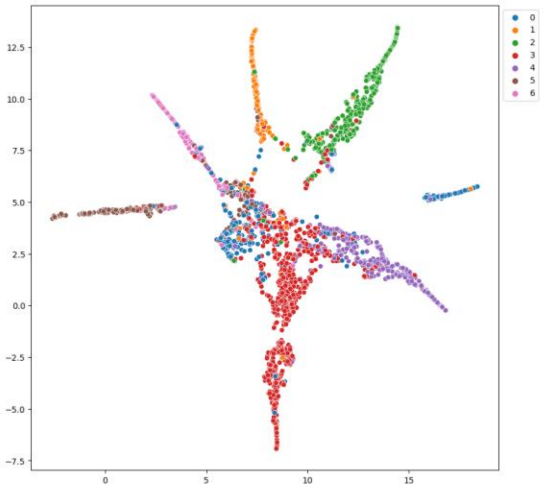
- Dataset: Cora
- Hidden dimension: 16
- Dropout: 0.5
- Epoch: 200

IMPLEMENTATION

```
Epoch 1/200, Validation Accuracy: 36.40%
Epoch 21/200, Validation Accuracy: 79.60%
Epoch 41/200, Validation Accuracy: 79.00%
Epoch 61/200, Validation Accuracy: 78.20%
Epoch 81/200, Validation Accuracy: 77.20%
Epoch 101/200, Validation Accuracy: 77.80%
Epoch 121/200, Validation Accuracy: 77.20%
Epoch 141/200, Validation Accuracy: 77.40%
Epoch 161/200, Validation Accuracy: 77.80%
Epoch 181/200, Validation Accuracy: 77.00%
```

```
Test Accuracy: 80.70%
Validation Accuracy: 77.80%
```





IMPLEMENTATION

```python
from torch_geometric.nn import GCNConv

def accuracy(y_pred, y_true):
    return torch.sum(y_pred == y_true) / len(y_true)

class GCN2(nn.Module):
    def __init__(self, in_feats, n_hidden, n_classes):
        super(GCN2, self).__init__()
        self.gcn1 = GCNConv(in_feats, n_hidden)
        self.gcn2 = GCNConv(n_hidden, n_classes)
        self.relu = nn.ReLU()

    def forward(self, x, edge_index):
        h = self.gcn1(x, edge_index)
        h = torch.relu(h)
        h = self.gcn2(h, edge_index)
        return F.log_softmax(h, dim=1)

    def fit(self, data, epochs):
        criterion = torch.nn.CrossEntropyLoss()
        optimizer = torch.optim.Adam(self.parameters(),
                                     lr=0.01,
                                     weight_decay=5e-4)

        self.train()
        for epoch in range(epochs+1):
            optimizer.zero_grad()
            out = self(data.x, data.edge_index)
            loss = criterion(out[data.train_mask], data.y[data.train_mask])
            acc = accuracy(out[data.train_mask].argmax(dim=1),
                           data.y[data.train_mask])
            loss.backward()
            optimizer.step()

            if(epoch % 20 == 0):
                val_loss = criterion(out[data.val_mask], data.y[data.val_mask])
                val_acc = accuracy(out[data.val_mask].argmax(dim=1),
                                   data.y[data.val_mask])
                print(f'Epoch {epoch:>3} | Train Loss: {loss:.3f} | Train Acc:'
                      f' {acc*100:>5.2f}% | Val Loss: {val_loss:.2f} | '
                      f'Val Acc: {val_acc*100:.2f}%')
```

```
GCN2(
  (gcn1): GCNConv(1433, 16)
  (gcn2): GCNConv(16, 7)
  (relu): ReLU()
)
Epoch   0 | Train Loss: 1.955 | Train Acc: 10.71% | Val Loss: 1.97 | Val Acc: 7.20%
Epoch  20 | Train Loss: 0.172 | Train Acc: 100.00% | Val Loss: 0.89 | Val Acc: 74.80%
Epoch  40 | Train Loss: 0.018 | Train Acc: 100.00% | Val Loss: 0.79 | Val Acc: 77.00%
Epoch  60 | Train Loss: 0.015 | Train Acc: 100.00% | Val Loss: 0.75 | Val Acc: 77.00%
Epoch  80 | Train Loss: 0.017 | Train Acc: 100.00% | Val Loss: 0.74 | Val Acc: 76.80%
Epoch 100 | Train Loss: 0.016 | Train Acc: 100.00% | Val Loss: 0.74 | Val Acc: 76.80%
Epoch 120 | Train Loss: 0.014 | Train Acc: 100.00% | Val Loss: 0.74 | Val Acc: 76.60%
Epoch 140 | Train Loss: 0.013 | Train Acc: 100.00% | Val Loss: 0.74 | Val Acc: 76.60%
Epoch 160 | Train Loss: 0.012 | Train Acc: 100.00% | Val Loss: 0.74 | Val Acc: 76.40%
Epoch 180 | Train Loss: 0.011 | Train Acc: 100.00% | Val Loss: 0.74 | Val Acc: 76.40%
Epoch 200 | Train Loss: 0.010 | Train Acc: 100.00% | Val Loss: 0.74 | Val Acc: 76.40%

GCN test accuracy: 80.30%
```

REFERENCE

https://www.nti-audio.com/en/support/know-how/fast-fourier-transform-fft
https://pygsp.readthedocs.io/en/latest/examples/fourier_transform.html
https://www.youtube.com/watch?v=BWOT57PPZ0Q
https://cse.msu.edu/~mayao4/tutorials/aaai2020/

APPENDIX

Laplacian quadratic form:

$$\mathbf{f}^T\mathbf{L}\mathbf{f} = \frac{1}{2}\sum_{i,j=1}^{N}\mathbf{A}[i,j](\mathbf{f}(i)-\mathbf{f}(j))^2$$

$$min_{f\in R^N,\,||f||_2=1}\mathbf{f}^\top\mathbf{L}\mathbf{f}$$

$$L = \mathbf{f}^\top\mathbf{L}\mathbf{f} - \lambda(\mathbf{f}^\top\mathbf{f} - 1)$$

최소화하는 f 찾기

$$\frac{\partial L}{\partial f} = 2\mathbf{L}\mathbf{f} - 2\lambda\mathbf{f} = 0$$

$$\mathbf{L}\mathbf{f} = \lambda\mathbf{f}$$

이때 f는 L의 eigen vector