SEMI-SUPERVISED CLASSIFICATION WITH GRAPH CONVOLUTIONAL NETWORKS

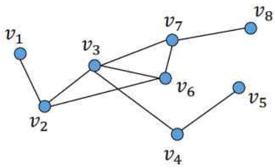
김지완

Contents

1. Background
    - Laplacian, Degree, Adjacency Matrix
    - Why Convolution ? - Convolution on Graphs
2. Introduction
    - How Do We Convolution ? - Graph Fourier Transform (GFT)
3. Fast Approximate Convolutions on Graphs
4. Semi-Supervised Node Classification
5. Experiments
6. Implementation

Background

Laplacian, Degree, Adjacency Matrix



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

Degree Matrix : D = Diag(degree(v_1, ... , v_n)

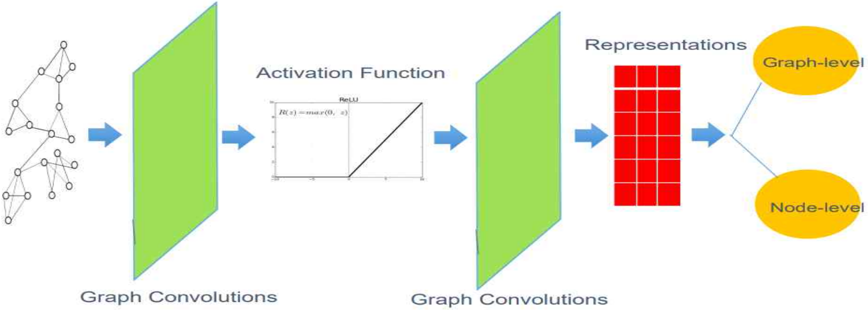
Laplacian Matrix : L = D - A

Degree Matrix   Adjacency Matrix   Laplacian Matrix

$$\begin{pmatrix}1&0&0&0&0&0&0&0\\0&3&0&0&0&0&0&0\\0&0&4&0&0&0&0&0\\0&0&0&2&0&0&0&0\\0&0&0&0&2&0&0&0\\0&0&0&0&0&4&0&0\\0&0&0&0&0&0&3&0\\0&0&0&0&0&0&0&1\end{pmatrix} - \begin{pmatrix}0&1&0&0&0&0&0&0\\1&0&1&0&0&1&0&0\\0&1&0&1&0&1&1&0\\0&0&1&0&1&0&0&0\\0&0&0&1&0&1&0&0\\0&1&1&0&1&0&1&0\\0&0&1&0&0&1&0&1\\0&0&0&0&0&0&1&0\end{pmatrix} = \begin{pmatrix}1&-1&0&0&0&0&0&0\\-1&3&-1&0&0&-1&0&0\\0&-1&4&-1&0&-1&-1&0\\0&0&-1&2&-1&0&0&0\\0&0&0&-1&2&-1&0&0\\0&-1&-1&0&-1&4&-1&0\\0&0&-1&0&0&-1&3&-1\\0&0&0&0&0&0&-1&1\end{pmatrix}$$

Background

Why Convolution ? - Layer applicable to Graph Structure



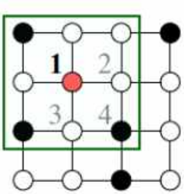
Problems in Graph Structure
1. Complex Topological Structure
2. No Fixed Node Ordering
3. Arbitrary Neighbor size

Desirable Property of Graph Layer
1. Permutation Invariant or Equivariance
2. Capturing Locality

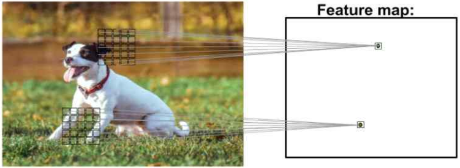
Naive Approach : MLP is not Working !

Background

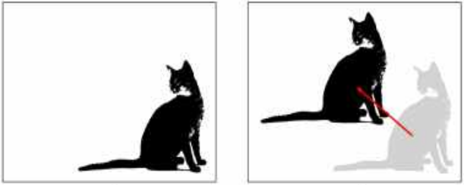
Why Convolution ? - Convolution in CNN



Locality



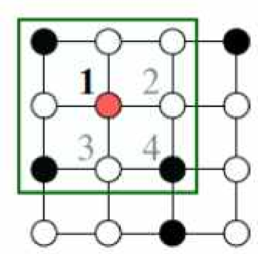
Transition Equivariance



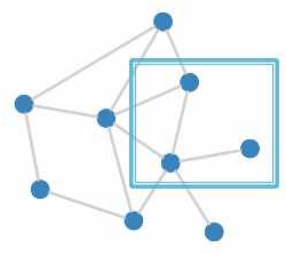
Generalize Convolution of CNN to Apply on Graph !

Background

Why Convolution ? - Convolution in Image vs Graph



Image



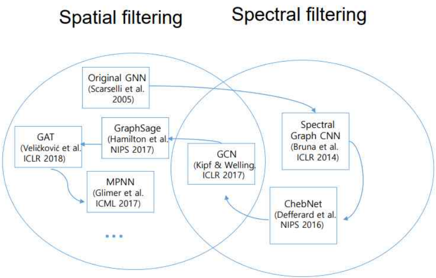
Graph

As Image is Fixed-Grid, Applying Convolution is Easy.

There are 2 ways in applying Convolution on Graph
1. Apply Directly to Graph Structure - Spatial Filtering
2. Change Graph Structure and Apply - Spectral Filtering

Background

Why Convolution ? - Spatial vs Spectral Convolution



Spatial Filtering
- Intuitively Applicable
- Calculation is Simple

- Before GCN(2017), Spatial Filtering is Unstable and Hard to Learning

Spectral Filtering
- Well-Defined Theory in Signal Processing

- Computational Cost is Expensive

Background

Why Convolution ? - What is Spectral ?

After Failing at Spatial Convolution, Try Spectral Convolution

Spectral Filtering
    Change Graph Structure to apply Convolution Easy
    -> Change Domain From Spatial to Some Other ..

Fourier Transform : Time Domain -> Frequency Domain
    -> Filtering Frequency Easy

Graph Fourier Transform : Spatial Domain -> Some Other Domain
    -> Node Similarity (Hidden Relationship) Easy

Changing Domain (Space) is related to Spectral Theory

Introduction

Semi-Supervised Learning

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

Supervised Loss With Small Labeled Dataset + Regularization Loss
    Connected Nodes by Edge is Likely to share Same Label

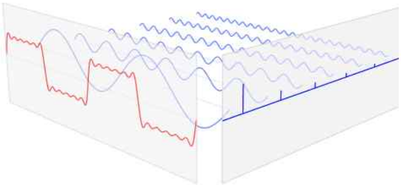
Edges Could Contain More Additional Information !

Two Contribution
1. Simple, Well-Behaved Layer-Wise Propagation Rule Can Directly Apply on Graphs
2. Fast and Scalable

-> First Order Approximation of Spectral Graph Convolution
    Stable + Fast Learning is Possible

Introduction

How Do We Convolution ?



$$
\begin{aligned}
\mathrm{F}(u) &= \langle f, e^{2j\pi ux} \rangle = \int f(t)\, e^{-2\pi jut} dt \\
\Delta(f) &= -\frac{\partial^2 f}{\partial t^2}, \textit{Laplace Operator} \\
\Delta(e^{2\pi jut}) &= -(2\pi u)^2 e^{2\pi jut} \\
\Delta f &= \lambda f
\end{aligned}
$$

Idea : Want to Know Divergence of Gradient (Diffusion) -> Laplace Operator

Laplace Operator : Operator on Function Space

$e^{2\pi jut}$ is EigenFunction(EigenVector) of Laplace Operator

Fourier Transform = Projecting Function To Other Function Space

= Representing Function With EigenFunction of Laplace Operator

Introduction

How Do We Convolution ?

In Signal Processing,
   Want to Know Divergence of Gradient (Diffusion) -> Laplace Operator

In Graph,
   Want to Know Similarity Between Nodes -> Which Operator ?

$$Laplacian\ Quadratic\ Form : f(X)^T \Delta f(X) = \sum_{i,j} A_{ij} ||f(X_i) - f(X_j)||^2$$

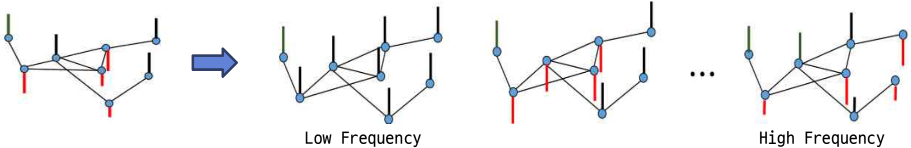
Laplacian Quadratic Form
    The Smaller, The Similar the Connected Nodes

Derivative of Laplacian Quadratic Form is Laplacian Matrix (Discrete Laplace Operator) !

Graph Fourier Transform = Projecting X (Feature) To Laplacian Matrix Space
                        = Representing X With EigenVector of Laplacian Matrix

Introduction

How Do We Convolution ?



$$L = U\Lambda U^T$$

$$GFT(X) = Representing\ X\ with\ U := [u_1, \cdots u_N]$$

$$= (U^T U)^{-1} U^T X = U^T X$$

$$IGFT(X) = UX$$

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

Introduction

How Do We Convolution ?

$$
\begin{aligned}
g_\theta * x &= IGFT(GFT(g_\theta) \cdot GFT(x)) \\
&= \quad U \quad \cdot \quad \hat{g}_\theta \quad \cdot \quad U^T x \\
&= U\hat{g}_\theta U^T x \ , \ \hat{g}_\theta \ is\ Arbitary\ Filtering\ Function \\
&= \begin{bmatrix} u_1 & u_2 & \cdots & u_N \end{bmatrix} \begin{bmatrix} \theta_{11} & \cdots & \theta_{1N} \\ \vdots & \ddots & \vdots \\ \theta_{N1} & \cdots & \theta_{NN} \end{bmatrix} \begin{bmatrix} {u_1}^T \\ {u_2}^T \\ \vdots \\ {u_N}^T \end{bmatrix} x \\
&= \begin{bmatrix} u_1 & u_2 & \cdots & u_N \end{bmatrix} \begin{bmatrix} \hat{g}_\theta(\lambda_0) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \hat{g}_\theta(\lambda_{N-1}) \end{bmatrix} \begin{bmatrix} {u_1}^T \\ {u_2}^T \\ \vdots \\ {u_N}^T \end{bmatrix} x \ , \hat{g}_\theta\, is\ Function\ of\ \Lambda
\end{aligned}
$$

Fast Approximate Convolution on Graphs

Layer-Wise Propagation Rule

$$g_\theta * x = \begin{bmatrix} u_1 & u_2 & \cdots & u_N \end{bmatrix} \begin{bmatrix} \hat{g}_\theta(\lambda_0) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \hat{g}_\theta(\lambda_{N-1}) \end{bmatrix} \begin{bmatrix} {u_1}^T \\ {u_2}^T \\ \vdots \\ {u_N}^T \end{bmatrix} x \text{ , } \hat{g}_\theta \textit{ is Function of } \Lambda$$

Problems in Computational Resource
-> Matrix Multiplication + Eigendecomposition of L is Expensive

4 Tricks For Fast Approximate

1. Approximate to ChebyShev Polynomial
2. K = 1 Set
3. Parameter Reduce
4. Renormaliation Trick

$$H^{(l+1)} = \sigma(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} H^{(l)} W^{(l)})$$

$$\tilde{A} = I_N + A \text{ , } \tilde{D}_{ii} = \sum A_{ij}$$

Fast Approximate Convolution on Graphs

Layer-Wise Propagation Rule

1. Approximate to ChebyShev Polynomial

ChebyShev Polynomial

$$\hat{g}_\theta(\Lambda) \approx \sum_{k=0}^{K} \theta_k T_k(\tilde{\Lambda}),\ \tilde{\Lambda} = \frac{2}{\lambda_{max}}\Lambda - I_N$$

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

$$\begin{aligned} g_\theta * x &= U\,\hat{g}_\theta\,U^T x \\ &= U \sum_{k=0}^{K} \theta_k{}^T T_k(\tilde{\Lambda}) U^T x \\ &= \sum_{k=0}^{K} \theta_k{}^T T_k(U\tilde{\Lambda}U^T)x,\ \tilde{L} = \frac{2}{\lambda_{max}} L - I_N \\ &= \sum_{k=0}^{K} \theta_k{}^T T_k(\tilde{L})x = \theta_0{}^T I_N x + \theta_1{}^T \tilde{L}\,x + \theta_2{}^T (2\tilde{L}^2 - I_N)x + \cdots \end{aligned}$$

K-th Polynomial Capture K-step Neighborhood

No More Need Eigen Decomposition
Only Need Spatial Data, L

Computational Still Expensive - O(KE)
Non-Linear - Hard To Stack Deep Layer

Fast Approximate Convolution on Graphs

Layer-Wise Propagation Rule 2. Setting K = 1

$$
\begin{aligned}
g_\theta * x &= \sum_{k=0}^{K} {\theta_k}^T T_k(\widetilde{L})x = {\theta_0}^T I_N x + {\theta_1}^T \widetilde{L}\, x + {\theta_2}^T (2\widetilde{L}^2 - I_N)x + \cdots \\
&= {\theta_0}^T I_N x + {\theta_1}^T \widetilde{L}\, x \\
&= {\theta_0}^T I_N x + {\theta_1}^T (\frac{2}{\lambda_{max}} L - I_N)\, x \text{ , } Assume\ \lambda_{max} \approx 2 \\
&= {\theta_0}^T I_N x - {\theta_1}^T D^{-\frac{1}{2}} A D^{-\frac{1}{2}}\, x
\end{aligned}
$$

By Setting K = 1, Convolution Capture 1-Step Neighborhood
-> More Deep Neighborhood Infomation by Stacking Layer Deep

Time Efficiency - O(E) & Linear -> Stacking Deep Layer Possible

Fast Approximate Convolution on Graphs

Layer-Wise Propagation Rule

3&4. Parameter Reduce & Renormalization Trick

$$g_\theta * x = {\theta_0}^T I_N x - {\theta_1}^T D^{-\frac{1}{2}} A D^{-\frac{1}{2}} x$$

Parameter Reduce

$$= \theta^T \left(I_N + D^{-\frac{1}{2}} A D^{-\frac{1}{2}} \right) x$$

Renormalization Trick

$$= \theta^T \left(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} \right) x \, , \tilde{A} = I_N + A : Self\ Loop$$

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

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} H^{(l)} W^{(l)}\right)$$

By Parameter Reduce, Prevent Overfitting

By Renormalization Trick, Avoid Gradient Exploding / Vanishing

Semi-Supervised Node Classification

Two Layer GCN

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

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

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

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

Experiments

Datasets

Table 1: Dataset statistics, as reported in Yang et al. (2016).

Dataset	Type	Nodes	Edges	Classes	Features	Label rate
Citeseer	Citation network	3,327	4,732	6	3,703	0.036
Cora	Citation network	2,708	5,429	7	1,433	0.052
Pubmed	Citation network	19,717	44,338	3	500	0.003
NELL	Knowledge graph	65,755	266,144	210	5,414	0.001

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

For Semi-Supervised Learning,

Train Set : 20 Samples Per Class,
Valid, Test Set : 500, 1000 Samples Each

Learing Rate : 0.01
Epoch : 200
Early Stopping : 10 Patience

Dropout : 0.1 (NELL) / 0.5 (Others)
L2 Regularization : 1e-5 (NELL) / 5e-4 (Others)
Hidden Dim : 64 (NELL) / 16 (Others)

(Upper) Mean Acc of 100 Random Node Ordering
(Lower) Mean Acc of 10 Dataset split

Graph, Feature : Row-wise Normalize

Experiments

Datasets

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)		$X\Theta_0 + D^{-\frac{1}{2}}AD^{-\frac{1}{2}}X\Theta_1$	68.3	80.0	77.5
Single parameter (Eq. 7)		$(I_N + D^{-\frac{1}{2}}AD^{-\frac{1}{2}})X\Theta$	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		$D^{-\frac{1}{2}}AD^{-\frac{1}{2}}X\Theta$	68.7	80.5	77.8
Multi-layer perceptron		$X\Theta$	46.5	55.1	71.4

Method	Eigen Value Decomposition	Propagation
Spectral GCN	$O(N^3)$	$O(N^2CF)$
+ ChebyShev Filter	-	$O(KECF)$
+ 1st-Order Model	-	$O(ECF)$
+ Single Parameter	-	$O(ECF)$
+ Renomalization Trick	-	$O(ECF)$

Limitation And Future Works

1. Memory Requirement

   Full Batch Gradient Descent

2. Directed Edges and Edges Features

   Only Applicable to Undirected Graphs

3. Limiting Assumptions

   Importance between Self-Connection and Neighborhood Edge

Implementation

Datasets

Table 1: Dataset statistics, as reported in Yang et al. (2016).

Dataset	Type	Nodes	Edges	Classes	Features	Label rate
Citeseer	Citation network	3,327	4,732	6	3,703	0.036
Cora	Citation network	2,708	5,429	7	1,433	0.052
Pubmed	Citation network	19,717	44,338	3	500	0.003
NELL	Knowledge graph	65,755	266,144	210	5,414	0.001

```
Data : Citeseer
Num Nodes : 3327
Num Edges : 9104
Feature Dim : 3703
Num Class : 6
```

```
Data : Cora
Num Nodes : 2708
Num Edges : 10556
Feature Dim : 1433
Num Class : 7
```

```
Data : PubMed
Num Nodes : 19717
Num Edges : 88648
Feature Dim : 500
Num Class : 3
```

```
Data : NELL
Num Nodes : 65755
Num Edges : 251550
Feature Dim : 61278
Num Class : 186
```

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

```
Train Finished : Citeseer
Test Set Result : After Train
loss= 1.0334
accuracy= 0.6750

Best Model Loss : 1.0238
Best Model Acc : 0.6850
```

```
Train Finished : Cora
Test Set Result : After Train
loss= 0.6913
accuracy= 0.8100

Best Model Loss : 0.6901
Best Model Acc : 0.8180
```

```
Train Finished : PubMed
Test Set Result : After Train
loss= 0.5556
accuracy= 0.7840

Best Model Loss : 0.5458
Best Model Acc : 0.7850
```

```
Train Finished : NELL
Test Set Result : After Train
loss= 2.9609
accuracy= 0.4910

Best Model Loss : 2.2798
Best Model Acc : 0.5710
```

Implementation

$$H^{(l+1)} = \sigma(\hat{A}\ H^{(l)} W^{(l)})$$

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

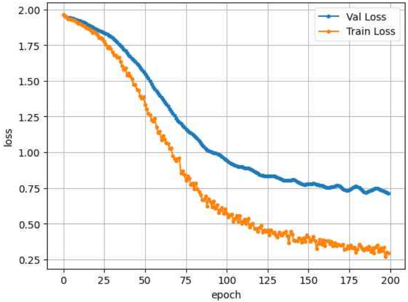
```python
# Normalize Graph
adj_graph = tgu.to_torch_coo_tensor(graph['edge_index'])

adj_graph_loop = tgu.add_self_loops(adj_graph)[0] # Add Self-Loop
D = tss.sum(adj_graph_loop, dim = 1)
indices_diag = torch.stack([D.indices(), D.indices()]).reshape(2, -1)
D_inv = torch.sparse_coo_tensor(indices = indices_diag,
                                values = 1.0 / D.values())
adj = tss.mm(D_inv, adj_graph_loop)
```

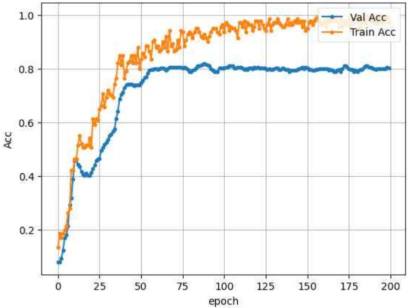
$$H^{(l+1)} = \sigma(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} H^{(l)} W^{(l)})$$

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

```python
class GraphConv(nn.Module) :

    def __init__(self, in_, out_) :
        super(GraphConv, self).__init__()

        self.in_features = in_
        self.out_features = out_

        self.weight = nn.Parameter(torch.FloatTensor(in_, out_))
        self.bias = nn.Parameter(torch.FloatTensor(out_))

        std = 1.0 / math.sqrt(self.weight.size(1))
        self.weight.data.uniform_(-std, std)
        self.bias.data.uniform_(-std, std)

    def forward(self, input, adj_graph) :

        output = tss.mm(input, self.weight)
        output = tss.mm(adj_graph, output) + self.bias

        return output
```

Implementation





In Paper, 20 Sample Per Class

Author Github Just 100 Sample Regardless of Classs - No Big Diffence

All Data 200 Epoch & Early Stop -> Citeseer / Cora / PubMed Stop Without Learning

For NELL Dataset, High Deviation in Accuracy

Implementation

$A \in R^{N\,x\,N}$ $Sparse\ Matrix\ with\ E\ elements\ nonzero$

$X \in R^{N\,x\,C}, W_1 \in R^{C\,x\,H}, W_2 \in R^{H\,x\,F}$

$One\ Layer : AXW_1 \qquad O(ECH)$

$Two\ Layer : AAXW_1W_2 \qquad O(ECHF)$

$A(AX^{(0)}W_1)W_2$, $Calculationg\ \ AX^{(0)}W_1\ is\ \ O(ECH)$

$AX^{(1)}W_2$, $Calculationg\ is\ O(EHF)$

$\rightarrow Time\ Complexity\ O(ECH + EHF)$