
Code Review

Graph Convolutional Network Tutorial

목 차

- Chapter 0 : GCN 설명
- Chapter 1 : Dataset 구성
- Chapter 2 : Data Preprocessing
- Chapter 3 : Model 구성
- Chapter 4 : Optimizer
- Chapter 5 : Model Train
- Chapter 6 : Model Evaluate
- Chapter 7 : Result in Tensorboard (Visualize)

Chapter 0

GCN 설명

Graph Convolutional Networks Paper : <https://arxiv.org/pdf/1609.02907.pdf>
Graph Convolutional Networks Code : <https://github.com/zhulf0804/GCN.PyTorch>

GCN 설명

■ 논문의 주요 내용

- 본 논문은 그래프에서 Semi-Supervised Classification에 대해 해결하고자 한다. 이때 각 label들은 전체 노드에서 아주 작은 부분에만 정의되어 있다고 가정한다.
- 이 문제는 각 레이블 정보들을 가지고 전체 그래프로 smooth시키는 일종의 그래프 기반의 semi-supervised learning 으로 볼 수 있으며, 이 smoothing에 다음과 같이 graph Laplacian regularization 식을 사용할 수 있다.

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

- \mathcal{L}_0 은 그래프에서 label되어있는 부분에 대한 supervised loss를 나타낸다.
- f 는 Neural network 등의 parameter를 가진 임의의 함수를 말한다.
- $\Delta = D - A$ 로, normalize 되지 않은 graph Laplacian 행렬을 나타낸다.
- A 는 해당 그래프의 인접행렬, D 는 해당 그래프의 차수를 $D_{ii} = \sum_j A_{ij}$ 와 같이 2차원 대각원소로 표현하고, 나머지는 값이 0인 행렬이다.

GCN 설명

■ 논문의 주요 내용

- 본 논문에서는 그래프 구조를 직접 모델 $f(X, A)$ 를 이용해 encoding을 하고, 이에 대해서 타겟 \mathcal{L}_0 대해 supervised learning으로 모든 노드에 대한 레이블을 학습한다 → 따라서 이 모델 **$f(X, A)$ 를 어떻게 만드는지**가 이 논문의 주요 내용이다.

■ 논문에서 제시한 $f(X, A)$ Fast Approximate Convolutions Model

- 그래프 기반의 Neural Network 모델 $f(X, A)$ 에 대해서 논문에서는 여러 layer에 걸친 Graph Convolutional Network를 구상하였다.

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

- $\tilde{A} = A + I_N$ 이다. 즉, 인접 행렬 A 와 항등 행렬 I_N 을 더한 행렬이다.
- $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$ 이다. 즉, \tilde{A} 에 대한 차수가 된다.
- $W^{(l)}$ 은 l 번째 레이어에 대한 학습 weight가 된다.
- σ 는 ReLU 등과 같은 activation function이다.
- $H^{(l)}$ 은 l 번째 레이어의 output이고, $H^{(0)}$ 는 input X 와 같다.

GCN 설명

■ GCN 모델의 의의

- 위의 모델 $f(X, A)$ 가 바로 논문의 핵심이고, 나머지 내용은 이를 통해서 graph convolution을 구하고, 이를 통해 semi-supervised node classification 문제를 풀어내는 과정을 소개하고 있다.

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

- 이 모델은 우리가 알고 있는 일반적인 딥러닝의 layer와 가장 다른 점이 하나 있는데 바로 각 layer의 입력마다 \hat{A} 를 곱해준다는 점이다.
- 기존의 딥러닝 layer는 각 layer의 output이 다음 layer의 input feature가 되지만, \hat{A} 는 일종의 normalize 처리가 된 인접행렬라고 볼 수 있으므로, 결국 해당 노드와 인접한 노드의 representation만 필터링하여 다음 layer를 계산한다고 볼 수 있다.
- 따라서 이를 통해 hidden layer의 output이 계속 각 노드에 대해서 이웃 노드의 정보가 더해진 representation의 역할을 수행하게 만든다.

Chapter 1

Dataset – Raw Data

Dataset

■ Citeseer

- **Node** (과학 출판물) : **3312개** , **Edge** (link) : **4732개** , **Label** : **6개** ['Agents', 'AI', 'DB', 'IR', 'ML', 'HCI']
- 각 Node는 0/1 값의 단어 벡터로 설명되는 3703개의 unique한 단어들로 구성되어 있다.

■ Cora

- **Node** (과학 출판물) : **2708개** , **Edge** (link) : **5429개** , **Label** : **7개** ['Case_Based', 'Genetic_Algorithms', 'Neural_Networks', 'Probabilistic_Methods', 'Reinforcement_Learning', 'Rule_Learning', 'Theory']
- 각 Node는 0/1 값의 단어 벡터로 설명되는 1433개의 unique한 단어들로 구성되어 있다.

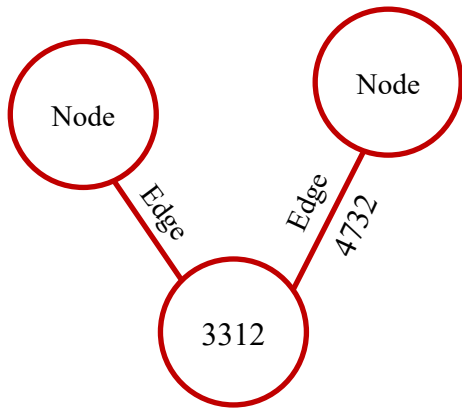
■ Pubmed

- **Node** (당뇨병과 관련된 과학 출판물) : **19717개** , **Edge** (link) : **44338개** , **Label** : **3개** ['Diabetes_Mellitus_Experimental', 'Diabetes_Mellitus_Type_1', 'Diabetes_Mellitus_Type_2']
- 각 Node는 TF-IDF 가중치 단어 벡터로 설명되는 500개의 unique 단어들로 구성되어 있다.

Dataset

■ Citeseer Dataset

x : 각각은 3703개 단어 유무 배열 , y : 각각은 6개의 class 유무 배열

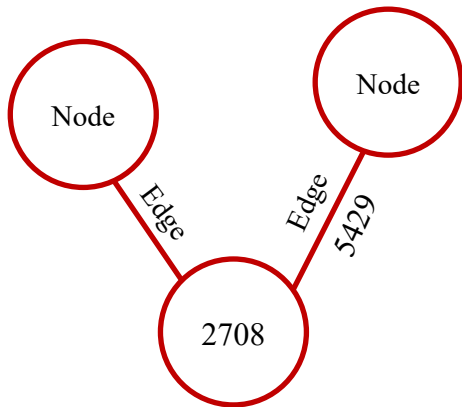
Citeseer						
	x	<div><div>3703</div><div><div>[[0 1...00]</div><div>[0 1...00]</div><div>...</div><div>[00...10]</div><div>[00...00]]</div></div><div>120</div></div>	allx	<div><div>3703</div><div><div>[[0 1...00]</div><div>[0 1...00]</div><div>...</div><div>[00...10]</div><div>[0 1...00]]</div></div><div>2312</div></div>	tx	<div><div>3703</div><div><div>[[0 1...00]</div><div>[0 1...00]</div><div>...</div><div>[00...10]</div><div>[00...00]]</div></div><div>1000</div></div>
	y	<div><div>6</div><div><div>[[000100]</div><div>[0 10000]</div><div>...</div><div>[000100]</div><div>[001000]]</div></div><div>120</div></div>	ally	<div><div>6</div><div><div>[[000100]</div><div>[0 10000]</div><div>...</div><div>[000100]</div><div>[001000]]</div></div><div>2312</div></div>	ty	<div><div>6</div><div><div>[[000100]</div><div>[0 10000]</div><div>...</div><div>[000100]</div><div>[001000]]</div></div><div>1000</div></div>

- test.index : 2312 ~ 3326인 1000개의 index data (중간에 15개의 빈 값 존재)
- graph : Dictionary type → { 0: [628], 1: [158, 2919, 2933, 1097, 486], 2: [3285], 3: [3219, 1431], 4: [467], ... }

Dataset

■ Cora Dataset

x : 각각은 1433개 단어 유무 배열 , y : 각각은 7개의 class 유무 배열

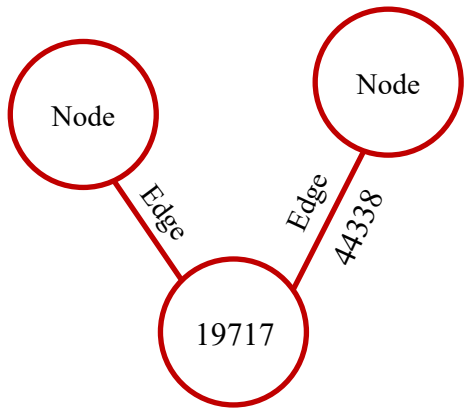
Cora						
	x	<div><div>1433</div><div>140</div><div><div><div>$[[0\ 1\ \dots\ 0\ 0]$</div><div>$[0\ 1\ \dots\ 0\ 0]$</div><div>\dots</div><div>$[0\ 0\ \dots\ 1\ 0]$</div><div>$[0\ 0\ \dots\ 0\ 0]$</div></div></div></div>	allx	<div><div>1433</div><div>1708</div><div><div><div>$[[0\ 1\ \dots\ 0\ 0]$</div><div>$[0\ 1\ \dots\ 0\ 0]$</div><div>\dots</div><div>$[0\ 0\ \dots\ 1\ 0]$</div><div>$[0\ 1\ \dots\ 0\ 0]$</div></div></div></div>	tx	<div><div>1433</div><div>1000</div><div><div><div>$[[0\ 1\ \dots\ 0\ 0]$</div><div>$[0\ 1\ \dots\ 0\ 0]$</div><div>\dots</div><div>$[0\ 0\ \dots\ 1\ 0]$</div><div>$[0\ 0\ \dots\ 0\ 0]$</div></div></div></div>
	y	<div><div>7</div><div>140</div><div><div><div>$[[0\ 0\ 0\ 1\ 0\ 0\ 0]$</div><div>$[0\ 1\ 0\ 0\ 0\ 0\ 0]$</div><div>\dots</div><div>$[0\ 0\ 0\ 1\ 0\ 0\ 0]$</div><div>$[0\ 0\ 1\ 0\ 0\ 0\ 0]$</div></div></div></div>	ally	<div><div>7</div><div>1708</div><div><div><div>$[[0\ 0\ 0\ 1\ 0\ 0\ 0]$</div><div>$[0\ 1\ 0\ 0\ 0\ 0\ 0]$</div><div>\dots</div><div>$[0\ 0\ 0\ 1\ 0\ 0\ 0]$</div><div>$[0\ 0\ 1\ 0\ 0\ 0\ 0]$</div></div></div></div>	ty	<div><div>7</div><div>1000</div><div><div><div>$[[0\ 0\ 0\ 1\ 0\ 0\ 0]$</div><div>$[0\ 1\ 0\ 0\ 0\ 0\ 0]$</div><div>\dots</div><div>$[0\ 0\ 0\ 1\ 0\ 0\ 0]$</div><div>$[0\ 0\ 1\ 0\ 0\ 0\ 0]$</div></div></div></div>

- test.index : 1708 ~ 2707인 1000개의 index data
- graph : Dictionary type $\rightarrow \{ 0: [633, 1862, 2582], 1: [2, 652, 654], 2: [1986, 332, 1666, 1, 1454], 3: [2544], \dots \}$

Dataset

■ Pubmed Dataset

x : 각각은 500개 가중치 단어 유무 배열 , y : 각각은 3개의 class 유무 배열

Pubmed						
	x	<div>500</div> <div>60</div> <div>$\begin{bmatrix} [0.104 \dots 0] \\ [0 \dots 0.163] \\ \dots \\ [0.213 \dots 0] \\ [0 \dots 0.107] \end{bmatrix}$</div>	allx	<div>500</div> <div>18717</div> <div>$\begin{bmatrix} [0.107 \dots 0] \\ [0 \dots 0.153] \\ \dots \\ [0.257 \dots 0] \\ [0 \dots 0.127] \end{bmatrix}$</div>	tx	<div>500</div> <div>1000</div> <div>$\begin{bmatrix} [0.103 \dots 0] \\ [0 \dots 0.143] \\ \dots \\ [0.223 \dots 0] \\ [0 \dots 0.307] \end{bmatrix}$</div>
	y	<div>3</div> <div>60</div> <div>$\begin{bmatrix} [0 0 0] \\ [0 1 0] \\ \dots \\ [0 0 0] \\ [0 0 1] \end{bmatrix}$</div>	ally	<div>3</div> <div>18717</div> <div>$\begin{bmatrix} [0 0 0] \\ [0 1 0] \\ \dots \\ [0 0 0] \\ [0 0 1] \end{bmatrix}$</div>	ty	<div>3</div> <div>1000</div> <div>$\begin{bmatrix} [0 0 0] \\ [0 1 0] \\ \dots \\ [0 0 0] \\ [0 0 1] \end{bmatrix}$</div>

- test.index : 18717 ~ 19716인 1000개의 index data
- graph : Dictionary type $\rightarrow \{ 0: [14442, 1378, 1544, 6092, 7636], 1: [10199, 8359, 2943], 2: [11485, 15572, 10471], \dots \}$

Chapter 2

Data Preprocessing

Data Preprocessing

- Load data

'networkx < 2.7' 로 실행시켜줘야 한다

```
## get data
data_path = 'data'
suffixs = ['x', 'y', 'allx', 'ally', 'tx', 'ty', 'graph']
objects = []
for suffix in suffixs:
    file = os.path.join(data_path, 'ind.%s.%s'%(dataset, suffix))
    objects.append(pickle.load(open(file, 'rb'), encoding='latin1'))
x, y, allx, ally, tx, ty, graph = objects
x, allx, tx = x.toarray(), allx.toarray(), tx.toarray()

# test indices
test_index_file = os.path.join(data_path, 'ind.%s.test.index'%dataset)
with open(test_index_file, 'r') as f:
    lines = f.readlines()
indices = [int(line.strip()) for line in lines]
min_index, max_index = min(indices), max(indices)
```

- min_index, max_index : test_index 사용하기 위해 저장한다

Data Preprocessing

- data extend → test index 만큼의 데이터를 확장한다
- features, labels → 확장된 dataset

```
# preprocess test indices and combine all data
tx_extend = np.zeros((max_index - min_index + 1,
tx.shape[1]))
features = np.vstack([allx, tx_extend])
features[indices] = tx
ty_extend = np.zeros((max_index - min_index + 1,
ty.shape[1]))
labels = np.vstack([ally, ty_extend])
labels[indices] = ty
```

Data extend	
<div>각각의 data shape에 맞게</div> <div>$\text{max_index} - \text{min_index} + 1$</div> <div>$\begin{bmatrix} [0\ 0\ 0\ \dots\ 0\ 0\ 0] \\ [0\ 0\ 0\ \dots\ 0\ 0\ 0] \\ \vdots \\ [0\ 0\ 0\ \dots\ 0\ 0\ 0] \\ [0\ 0\ 0\ \dots\ 0\ 0\ 0] \end{bmatrix}$</div>	
features	labels
$\begin{bmatrix} [& \text{allx} &] \\ [& &] \\ & + & \\ [& \text{tx} &] \\ [& &] \end{bmatrix}$	$\begin{bmatrix} [& \text{ally} &] \\ [& &] \\ & + & \\ [& \text{ty} &] \\ [& &] \end{bmatrix}$

Data Preprocessing

- Adjacency matrix(인접행렬)는 networkx를 활용하여 Dictionary에서 2차원 배열로 바꿔준다

```
# get adjacency matrix  
adj = nx.adjacency_matrix(nx.from_dict_of_lists(graph)).toarray()
```

ex) Citeseer

Dictionary Type

{0: [628], 1: [158, 2919, 2933, 1097, 486], 2: [3285], 3: [3219, 1431],



Adjacency matrix

3327

3327

[[0 0 0 ... 0 0 0]
[0 0 0 ... 0 0 0]
[0 0 0 ... 0 0 0]
...
[0 0 0 ... 0 0 0]
[0 0 0 ... 0 0 0]
[0 0 0 ... 0 0 0]

Data Preprocessing

- Data 마다 mask 진행해준 다음 train하기 위한 torch 형태로 변환 → train : y의 개수 만큼, val : 500개, test : test 개수만큼

```
idx_train = range(len(y))
idx_val = range(len(y), len(y) + 500)
idx_test = indices

train_mask = sample_mask(idx_train, labels.shape[0])
val_mask = sample_mask(idx_val, labels.shape[0])
test_mask = sample_mask(idx_test, labels.shape[0])
zeros = np.zeros(labels.shape)
y_train = zeros.copy()
y_val = zeros.copy()
y_test = zeros.copy()
y_train[train_mask, :] = labels[train_mask, :]
y_val[val_mask, :] = labels[val_mask, :]
y_test[test_mask, :] = labels[test_mask, :]
features = torch.from_numpy(process_features(features))
y_train, y_val, y_test, train_mask, val_mask, test_mask = \
torch.from_numpy(y_train), torch.from_numpy(y_val), torch.from_numpy(y_test), \
torch.from_numpy(train_mask), torch.from_numpy(val_mask),
torch.from_numpy(test_mask)

return adj, features, y_train, y_val, y_test, train_mask, val_mask, test_mask
```

```
def sample_mask(idx, l):
    mask = np.zeros(l)
    mask[idx] = 1
    return np.array(mask, dtype=np.bool)
```


Data Preprocessing

- features 전처리

```
def process_features(features):  
    row_sum_diag = np.sum(features, axis=1)  
    row_sum_diag_inv = np.power(row_sum_diag, -1)  
    row_sum_diag_inv[np.isinf(row_sum_diag_inv)] = 0.  
    row_sum_inv = np.diag(row_sum_diag_inv)  
    return np.dot(row_sum_inv, features)
```

1. row_sum_diag : features에서 row별(node별) 단어 개수 sum
2. row_sum_diag_inv : 각 node별 단어 개수를 분수로 만듦(-1승)
3. 만약, 분수로 만들 때 sum = 0인 값은 infinite가 되므로 이는 0으로 다시 전처리 진행
4. row_sum_inv : row_sum_diag_inv로 대각행렬 생성
5. np.dot(row_sum_inv, features) : 각 node별 node의 sum으로 표준화 진행 (! 질문 : 정확한 의미 파악 X!) – nomalize 방법 중 하나

1. array([31., 33., 25., ..., 40., 36., 26.])
2. array([0.03225806, 0.03030303, 0.04, ..., 0.025, 0.02777778,
0.03846154])
3. array([[0.03225806, 0., ..., 0., 0.,
0., 1., 0.03030303, 0., ..., 0., 0.,
0., 1., 0., ..., 0., 0.,
0., 1., 0., ..., 0., 0.,
...,
[0., 0., 0., ..., 0.025, 0.,
0., 1., 0., ..., 0., 0.02777778,
0., 1., 0., ..., 0., 0.,
[0., 0., 0., ..., 0., 0.,
0.03846154]])
- 4.
5. array([[0., 0., 0., ..., 0., 0., 0.,
[0., 0., 0., ..., 0., 0., 0.,
[0., 0., 0., ..., 0., 0., 0.,
...,
[0., 0., 0., ..., 0., 0., 0.,
[0., 0., 0., ..., 0., 0., 0.,
[0., 0., 0., ..., 0., 0., 0.]])

Data Preprocessing

■ Data preprocessing 최종 return 값

```
return adj, features, y_train, y_val, y_test, train_mask, val_mask, test_mask
```

- adj → Graph의 adjacency matrix
- features → 각 Node별 특징 (ex. 데이터 존재 유무)
- y_train → train data의 label (train_mask가 True인 data만 사용한다)
- y_val → validation data의 label (val_mask가 True인 data만 사용한다)
- y_test → test data의 label (test_mask가 True인 data만 사용한다)
- train_mask → feature data의 train에 사용하는 index (True / False로 나타낸다)
- val_mask → feature data의 validation에 사용하는 index (True / False로 나타낸다)
- test_mask → feature data의 test에 사용하는 index (True / False로 나타낸다)

예시 : Citeseer data shape

```
adj : (3327, 3327)
features : torch.Size([3327, 3703])
y_train : torch.Size([3327, 6])
y_val : torch.Size([3327, 6])
y_tset : torch.Size([3327, 6])
train_mask : torch.Size([3327])
val_mask : torch.Size([3327])
test_mask : torch.Size([3327])
```

Data Preprocessing

```
adj : (3327, 3327)
features : torch.Size([3327, 3703])
y_train : torch.Size([3327, 6])
y_val : torch.Size([3327, 6])
y_tset : torch.Size([3327, 6])
train_mask : torch.Size([3327])
val_mask : torch.Size([3327])
test_mask : torch.Size([3327])
```

■ Citeseer

- adj → Citeseer Graph의 인접행렬
- features → 각 Node별 특징 (ex. 데이터 존재 유무)
- y_train → train_mask가 True인 data만 사용
- y_val → val_mask가 True인 data만 사용
- y_test → test_mask가 True인 data만 사용
- train_mask → Train data 120개 mask : { **True [: 119] , False [120 :]** }

$$\text{tensor} (\overset{3327}{\text{[True, True, True, \dots, False, False, False]}})$$
- val_mask → Valid data 500개 mask : { **True[120 : 619] , False[: 119 , 620 :]** }

$$\text{tensor} (\overset{3327}{\text{[False, False, False, \dots, False, False, False]}})$$
- test_mask → Test data 1000개 mask : { **2312에서 3326까지 data 중 15개의 data 제외** }

$$\text{tensor} (\overset{3327}{\text{[False, False, False, \dots, True, True, True]}})$$

adj	features
$\overset{3327}{\text{[[0 1 \dots 0 0] } \\ \text{[0 1 \dots 0 0] } \\ \dots \\ \text{[0 0 \dots 1 0] } \\ \text{[0 0 \dots 0 0] }]}}$	$\overset{3703}{\text{[[0. 0. \dots 0. 0.] } \\ \text{[0. 0. \dots 0. 0.] } \\ \dots \\ \text{[0. 0. \dots 0. 0.] } \\ \text{[0. 0. \dots 0. 0.] }]}}$
	y
	$\overset{6}{\text{[[0 1 0 0 0 0] } \\ \text{[0 1 0 0 0 0] } \\ \dots \\ \text{[0 0 0 0 1 0] } \\ \text{[0 0 0 0 0 0] }]}}$

Data Preprocessing

```
adj : (2708, 2708)
features : torch.Size([2708, 1433])
y_train : torch.Size([2708, 7])
y_val : torch.Size([2708, 7])
y_tset : torch.Size([2708, 7])
train_mask : torch.Size([2708])
val_mask : torch.Size([2708])
test_mask : torch.Size([2708])
```

■ Cora

- adj → Cora Graph의 인접행렬
- features → 각 Node별 특징 (ex. 데이터 존재 유무)
- y_train → train_mask가 True인 data만 사용
- y_val → val_mask가 True인 data만 사용
- y_test → test_mask가 True인 data만 사용
- train_mask → Train data 개수 140개 mask : { **True [: 139] , False [140 :]** }

$$\text{tensor} (\overset{2708}{[\text{True}, \text{True}, \text{True}, \dots, \text{False}, \text{False}, \text{False}]})$$
- val_mask → Valid data 개수 500개 mask : { **True[140 : 639] , False[: 139 , 640 :]** }

$$\text{tensor} (\overset{2708}{[\text{False}, \text{False}, \text{False}, \dots, \text{False}, \text{False}, \text{False}]})$$
- test_mask → Test Node 개수 1000개 mask : { **True[1708 : 2707] , False[: 1707]** }

$$\text{tensor} (\overset{2708}{[\text{False}, \text{False}, \text{False}, \dots, \text{True}, \text{True}, \text{True}]})$$

adj	features
$\overset{2708}{\underset{2708}{\begin{bmatrix} [0\ 1\ \dots\ 0\ 0] \\ [0\ 1\ \dots\ 0\ 0] \\ \dots \\ [0\ 0\ \dots\ 1\ 0] \\ [0\ 0\ \dots\ 0\ 0] \end{bmatrix}}}$	$\overset{1433}{\underset{2708}{\begin{bmatrix} [0.0\ \dots\ 0.0.] \\ [0.0\ \dots\ 0.0.] \\ \dots \\ [0.0\ \dots\ 0.0.] \\ [0.0\ \dots\ 0.0.] \end{bmatrix}}}$
	y
	$\overset{7}{\underset{2708}{\begin{bmatrix} [0\ 1\ 0\ 0\ 0\ 0\ 0] \\ [0\ 1\ 0\ 0\ 0\ 0\ 0] \\ \dots \\ [0\ 0\ 0\ 0\ 0\ 1\ 0] \\ [0\ 0\ 0\ 0\ 0\ 0\ 0] \end{bmatrix}}}$

Data Preprocessing

```
adj : (19717, 19717)
features : torch.Size([19717, 500])
y_train : torch.Size([19717, 3])
y_val : torch.Size([19717, 3])
y_tset : torch.Size([19717, 3])
train_mask : torch.Size([19717])
val_mask : torch.Size([19717])
test_mask : torch.Size([19717])
```

■ Pubmed

- adj → Pubmed Graph의 인접행렬
- features → 각 Node별 특징 (ex. 데이터 존재 유무)
- y_train → train_mask가 True인 data만 사용
- y_val → val_mask가 True인 data만 사용
- y_test → test_mask가 True인 data만 사용
- train_mask → Train data 개수 60개 mask : { **True [: 59] , False [60 :]** }

$$\text{tensor} (\overset{19717}{[\text{True}, \text{True}, \text{True}, \dots, \text{False}, \text{False}, \text{False}] })$$
- val_mask → Valid data 개수 500개 mask : { **True[60 : 559] , False[: 59 , 560 :]** }

$$\text{tensor} (\overset{19717}{[\text{False}, \text{False}, \text{False}, \dots, \text{False}, \text{False}, \text{False}] })$$
- test_mask → Test Node 개수 1000개 mask : { **True[18717 : 19716] , False[: 18716]** }

$$\text{tensor} (\overset{19717}{[\text{False}, \text{False}, \text{False}, \dots, \text{True}, \text{True}, \text{True}] })$$

adj	features
$\overset{19717}{\text{[[0 1 ... 0 0]}} \\ \overset{19717}{\text{[0 1 ... 0 0]}} \\ \dots \\ \text{[0 0 ... 1 0]}} \\ \text{[0 0 ... 0 0]}}$	$\overset{500}{\text{[[0. 0. ... 0. 0.]}} \\ \overset{19717}{\text{[0. 0. ... 0. 0.]}} \\ \dots \\ \text{[0. 0. ... 0. 0.]}} \\ \text{[0. 0. ... 0. 0.]}}$
	y
	$\overset{3}{\text{[[0 1 0]}} \\ \overset{19717}{\text{[0 0 0]}} \\ \dots \\ \text{[0 1 0]}} \\ \text{[0 0 0]}}$

Chapter 2.1

Graph 표현법

Graph 표현법

■ Adjacency matrix

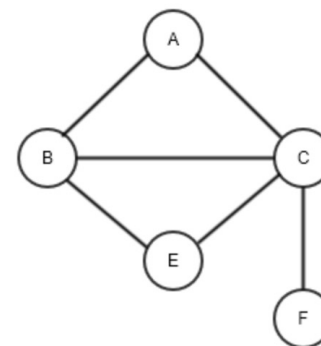
- 인접행렬은 2차원 배열로 두 Node가 Edge로 연결되어 있는지 여부에 따라 1 / 0 으로 나타낸다. Node의 수가 많다면 시간 복잡도가 크다는 단점이 존재한다

Adjacency matrix

ex) Node : 1000

1000

```
[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
```



Adjacency Matrix

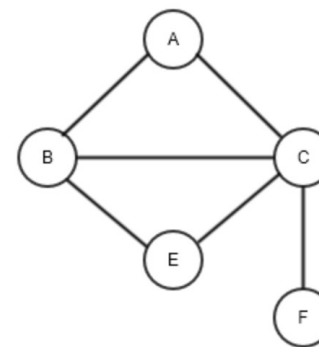
	A	B	C	E	F
A	0	1	1	0	0
B	1	0	1	1	0
C	1	1	0	1	1
E	0	1	1	0	0
F	0	0	1	0	0

■ Dictionary – linked List

- 딕셔너리는 키(key)와 값(value)으로 나타낸다. 키(key)는 그래프의 Node이고 해당 값(value)은 Edge로 연결되는 각 Node가 있는 목록으로 나타낼 수 있다.

Dictionary

ex) {0: [628], 1: [158, 2919, 2933, 1097, 486], 2: [3285], 3: [3219, 1431],



{'A': ['B', 'C'],
'B': ['A', 'C', 'E'],
'C': ['A', 'B', 'E', 'F'],
'E': ['B', 'C'],
'F': ['C']}

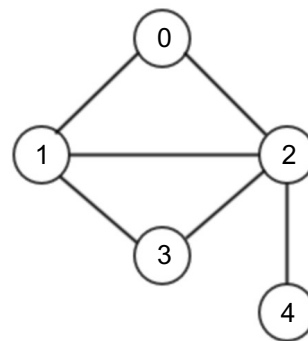
Graph 표현법

■ Edge List

- 그래프를 Edge List로 나타내는 방법이다. Edge의 수가 많다면 시간 복잡도가 크다는 단점이 존재한다

Edge List

ex) [(0, 1), (1, 2), (2, 3), (0, 2), (3, 2), (4, 5), (5, 4)]



→ [(0, 1), (0, 2), (1, 0), (1, 2), (1, 3), (2, 0), (2, 1), (2, 3), (2, 4), (3, 1), (3, 2), (4, 2)]

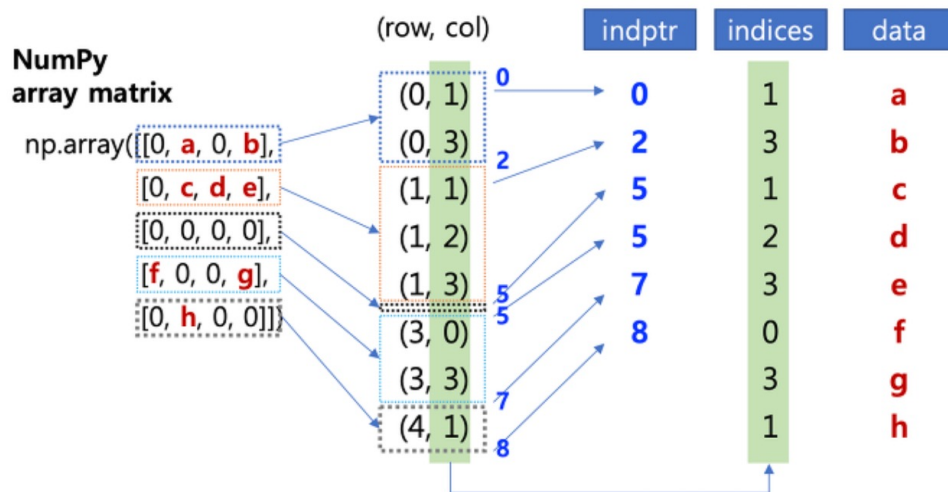
■ CSR_matrix (Compressed Sparse Row)

- 그래프를 matrix형태로 표현할 때 Edge가 별로 존재하지 않는 경우 희소행렬이 된다. 희소행렬(Sparse matrix)의 경우 대부분의 값이 0이므로 이를 그대로 사용할 경우 메모리 낭비가 심하고 시간 복잡도도 오래 걸린다는 단점이 존재한다.
- CSR(압축 희소 행)은 데이터를 행(가로)의 순서대로 정리 압축하는 방법이다

Graph 표현법

■ CSR_matrix (Compressed Sparse Row)

- **indptr**: 행렬의 '0' 이 아닌 원소의 행의 시작 위치
- **indices**: 행렬의 '0' 이 아닌 원소의 열의 위치
- **data**: 행렬의 '0' 이 아닌 원소 값



CSR_matrix

ex)

```
array([[0, 1, 0, 2],  
       [0, 3, 4, 5],  
       [0, 0, 0, 0],  
       [6, 0, 0, 7],  
       [0, 8, 0, 0]])
```

→

```
indptr: [0 2 5 5 7 8]  
indices: [1 3 1 2 3 0 3 1]  
data: [1 2 3 4 5 6 7 8]
```

python SciPy 모듈의 `csr_matrix()` 메소드를 사용하여 변환 가능하다

Chapter 3

Model 구성

Model 구성

```
class GCNLayer(nn.Module):
    def __init__(self, in_dim, out_dim, acti=True):
        super(GCNLayer, self).__init__()
        self.linear = nn.Linear(in_dim, out_dim)
        if acti:
            self.acti = nn.ReLU(inplace=True)
        else:
            self.acti = None
    def forward(self, F):
        output = self.linear(F)
        if not self.acti:
            return output
        return self.acti(output)
```

- GCN Layer 정의 클래스

데이터 F를 입력 받아서 nn에 넣어주고 output을 조건에 따라 activation function에 넣어주는 과정을 거친다.

- 활용

Activation Function 변경 가능

Model 구성

```
def preprocess_adj(A):  
    I = np.eye(A.shape[0])  
    A_hat = A + I  
    D_hat_diag = np.sum(A_hat, axis=1)  
    D_hat_diag_inv_sqrt = np.power(D_hat_diag, -0.5)  
    D_hat_diag_inv_sqrt[np.isinf(D_hat_diag_inv_sqrt)] = 0.  
    D_hat_inv_sqrt = np.diag(D_hat_diag_inv_sqrt)  
    return np.dot(np.dot(D_hat_inv_sqrt, A_hat),  
D_hat_inv_sqrt)
```

- Adj(인접행렬) 전처리 함수

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

- 인접행렬 A를 위의 GCN의 aggregate function
부분에 따라서 전처리과정을 거친다.

Model 구성

```
class GCN(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_classes, p):
        super(GCN, self).__init__()
        self.gcn_layer1 = GCNLayer(input_dim, hidden_dim)
        self.gcn_layer2 = GCNLayer(hidden_dim, num_classes, acti=False)
        self.dropout = nn.Dropout(p)
```

- GCN 모델 정의 클래스(__init__)

예제에 사용되는 GCN 모델은 다음과 같이 2개의 GCN_layer를 정의하고 있으며 1개의 dropout layer를 정의하고 있다.

- 활용

GCN Architecture는 사용할 Layer의 수에 따라 변경가능 하다.

Model 구성

```
class GCN(nn.Module):
    def forward(self, A, X):
        A = torch.from_numpy(preprocess_adj(A)).float()
        X = self.dropout(X.float())
        F = torch.mm(A, X)
        F = self.gcn_layer1(F)
        F = self.dropout(F)
        F = torch.mm(A, F)
        output = self.gcn_layer2(F)
        return output
```

torch.mm : matrix multiplication

- GCN 모델 정의 클래스 (forward)

(A = 인접행렬 , X = Feature)

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

- 전체적인 Aggregate Function의 과정을 거친다.

- 인접행렬 전처리 ($\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{\frac{1}{2}}$ 계산)
- Dropout(선택사항)
- $\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{\frac{1}{2}} H^{(l)}$
- $\sigma(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{\frac{1}{2}} H^{(l)} W^{(l)})$

이 코드에서는 위 과정을 2번 진행한다.

Code 활용

■ Code 활용

- 결과적으로 모델정의는 Aggregate Function을 구현 하는 것이므로 다른 종류의 모델 (GraphSAGE, GAT)을 적용할 때도 앞서 나온 코드의 구조에서 내용만 변경하여서 사용 가능하다.
- 또한 torch_geometric 라이브러리를 사용하면 더욱 간편하게 사용 가능하다.

https://pytorch-geometric.readthedocs.io/en/latest/modules/torch_geometric/nn/conv/gcn_conv.html#GCNConv

- 위에 링크를 확인해보면 GCNConv에 인접행렬 전처리, GCNLayer가 내장되어 있어 편리하게 사용 가능하고 parameter값만 변경해주면 더욱 다양하게 활용 가능하다.

Chapter 4

Optimizer

Optimizer

```
def build_optimizer(model, lr, weight_decay):
    gcn1, gcn2 = [], []
    for name, p in model.named_parameters():
        if 'layer1' in name:
            gcn1.append(p)
        else:
            gcn2.append(p)
    opt = optim.Adam([{'params': gcn1, 'weight_decay': weight_decay},
                      {'params': gcn2}], lr=lr)
    return opt
```

- Optimizer 정의 함수

일반적인 딥러닝에서 쓰이는 Adam을 사용하여 최적화를 진행한다.

Chapter 5

Model Train

Model Train

1. Hyper-parameter 설정

- 실험 결과에 따른 최적인 값으로 변수 설정
- Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016. 결과 참조

```
parser = argparse.ArgumentParser()
parser.add_argument('--dataset', type=str, default='citeseer', help='Dataset to train')
parser.add_argument('--init_lr', type=float, default=0.01, help='Initial learning rate')
parser.add_argument('--epoches', type=int, default=200, help='Number of traing epoches')
parser.add_argument('--hidden_dim', type=list, default=16, help='Dimensions of hidden layers')
parser.add_argument('--dropout', type=float, default=0.5, help='Dropout rate (1 - keep probability)')
parser.add_argument('--weight_decay', type=float, default=5e-4, help='Weight for l2 loss on embedding matrix')
parser.add_argument('--log_interval', type=int, default=10, help='Print iterval')
parser.add_argument('--log_dir', type=str, default='experiments', help='Train/val loss and accuracy logs')
parser.add_argument('--checkpoint_interval', type=int, default=20, help='Checkpoint saved interval')
parser.add_argument('--checkpoint_dir', type=str, default='checkpoints', help='Directory to save checkpoints')
args = parser.parse_args()
```

Model Train

2. 변수 선언

```
adj, features, y_train, y_val, y_test, train_mask, val_mask, test_mask = load_data(args.dataset)
model = GCN(features.shape[1], args.hidden_dim, y_train.shape[1], args.dropout)
optimizer = build_optimizer(model, args.init_lr, args.weight_decay)
```

- adj : 그래프 데이터
- features : 각 노드별 단어에 대한 유무 데이터 모음
- y_train : train의 라벨
- train_mask : feature 데이터의 train에 사용하는 index
- y_val : val의 라벨
- val_mask : feature 데이터의 val에 사용하는 index
- test_val : test의 라벨
- test_mask : feature 데이터의 test에 사용하는 index
- model : class GCN(객체) 생성
- optimizer : Chapter 4 에서 정의한 함수 호출 & Adam optimizer 반환 및 할당

Model Train

3. Train 시각화를 위한 데이터 저장 설정

```
from tensorboardX import SummaryWriter

log_dir = os.path.join(args.log_dir, args.dataset)
if not os.path.exists(log_dir):
    os.makedirs(log_dir)

writer = SummaryWriter(log_dir)
saved_checkpoint_dir = os.path.join(args.checkpoint_dir, args.dataset)

if not os.path.exists(saved_checkpoint_dir):
    os.makedirs(saved_checkpoint_dir)
```

- log_dir : 저장경로
- writer : 저장경로에 tensorboard로 시각화 하기위한 마커
- checkpoint_dir에 checkpoint_interval(간격) 마다 저장
ex) 전체 200epoch 반복 중에 매 20번마다 저장을 실행

Model Train

4. Train 시작

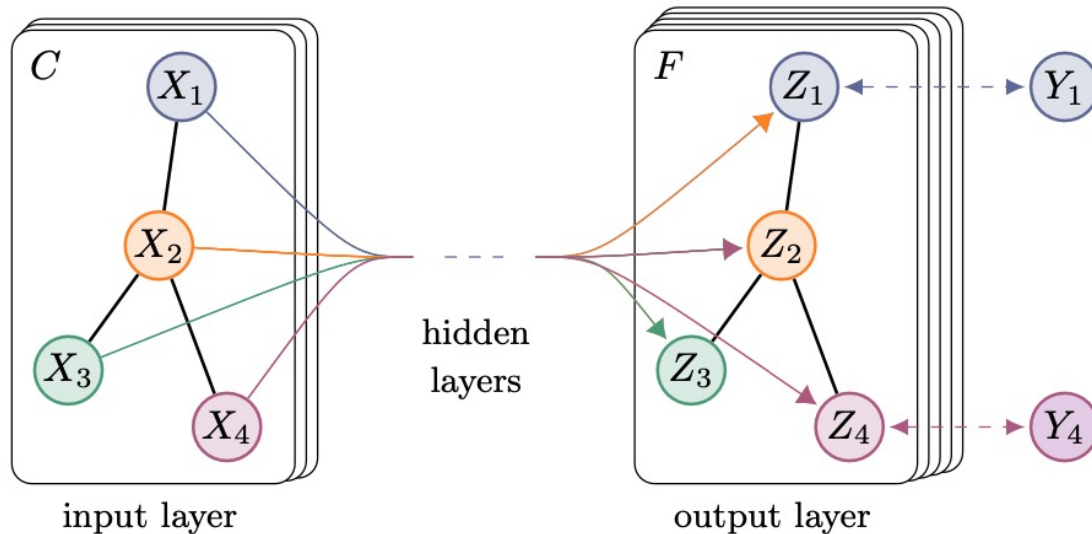
```
for epoch in range(args.epoches + 1):
    outputs = model(adj, features)
    loss = get_loss(outputs, y_train, train_mask)
    val_loss = get_loss(outputs, y_val, val_mask).detach().numpy()
    model.eval()
    outputs = model(adj, features)
    train_accuracy = get_accuracy(outputs, y_train, train_mask)
    val_accuracy = get_accuracy(outputs, y_val, val_mask)
    model.train()
    writer.add_scalars('loss', {'train_loss': loss.detach().numpy(), 'val_loss': val_loss}, epoch)
    writer.add_scalars('accuracy', {'train_ac': train_accuracy, 'val_ac': val_accuracy}, epoch)
    if epoch % args.log_interval == 0:
        print("Epoch: %d, train loss: %f, val loss: %f, train ac: %f, val ac: %f"
              %(epoch, loss.detach().numpy(), val_loss, train_accuracy, val_accuracy))
    if epoch % args.checkpoint_interval == 0:
        torch.save(model.state_dict(), os.path.join(saved_checkpoint_dir, "gcn_%d.pth"%epoch))
    optimizer.zero_grad() # Important
    loss.backward()
    optimizer.step()

writer.close()
```

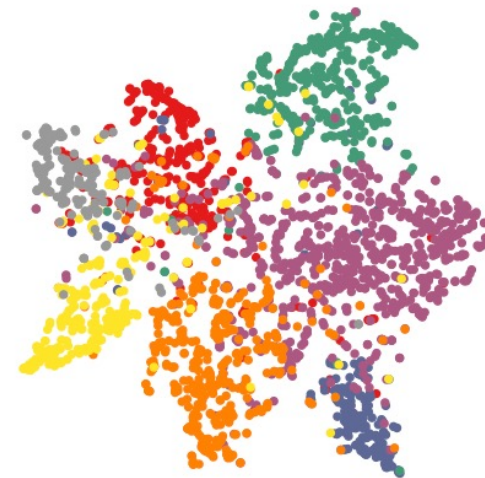
Model Train

4. Train 시작 – Step 1

```
outputs = model(adj, features)
loss = get_loss(outputs, y_train, train_mask)
val_loss = get_loss(outputs, y_val, val_mask).detach().numpy()
```



(a) Graph Convolutional Network



(b) Hidden layer activations

Model Train

4. Train 시작 – Step 1

```
outputs = model(adj, features)
loss = get_loss(outputs, y_train, train_mask)
val_loss = get_loss(outputs, y_val, val_mask).detach().numpy()
```

- model (=GCN) 객체에 adj(그래프)와 features(각 node의 feature)를 전달 시 각 node의 클래스를 반환 (=outputs)
ex) 각 node에 대해 'citeseer' : ['Agents', 'AI', 'DB', 'IR', 'ML', 'HCI'] (6개의 클래스) 중 하나라고 예측함 (Y값)

- 예측 값과 실제 y_train과 비교하여 loss값을 계산.

이 때 전체 dataset에서 x_train에 대해서만 예측을 수행했으므로 train_mask를 사용하여 해당 범위에서만 loss 계산

Model Train

4. Train 시작 – Step 1

```
class Loss(nn.Module):
    def __init__(self):
        super(Loss, self).__init__()
        self.loss =
nn.CrossEntropyLoss(reduction='none')
    def forward(self, output, labels, mask):
        labels = torch.argmax(labels, dim=1)
        loss = self.loss(output, labels)
        mask = mask.float()
        mask /= torch.mean(mask)
        loss *= mask
        return torch.mean(loss)
```

- 이때 Loss는 CrossEntropyLoss 사용
- 각 label에 속할 확률로 output이 생성됨
 - ex) [0.8, 0.5, 0.3, 0.4, 0.5, 0.7]
 - 'citeseer' : ['Agents', 'AI', 'DB', 'IR', 'ML', 'HCI']
- 이 중 최대값(속할 확률이 max)을 label로 정하여 예측 값을 정한다.
 - ex) 0.8 선택 = 해당 node는 Agents 라고 예측.

Model Train

4. Train 시작 – step 2

```
outputs = model(adj, features)
loss = get_loss(outputs, y_train, train_mask)
val_loss = get_loss(outputs, y_val, val_mask).detach().numpy()
```

1. 모든 node에 대해 예측(=outputs)
2. Train set Loss 계산.
3. Valid set Loss 계산

```
model.eval()
outputs = model(adj, features)
train_accuracy = get_accuracy(outputs, y_train, train_mask)
val_accuracy = get_accuracy(outputs, y_val, val_mask)
```

model.eval() → 성능 측정 시 해야만 함 # 모델의 파라미터를 업데이트하지 못하게 방지하는 코드

y_train 예측값(outputs)이 실제(y_train)과 몇개가 똑같은지 계산.

y_val 예측값(outputs)이 실제(y_val)과 몇개가 똑같은지 계산.

Model Train

4. Train 시작 – step3

```
model.train()
writer.add_scalars('loss', {'train_loss': loss.detach().numpy(), 'val_loss': val_loss}, epoch)
writer.add_scalars('accuracy', {'train_ac': train_accuracy, 'val_ac': val_accuracy}, epoch)

if epoch % args.log_interval == 0:
    print("Epoch: %d, train loss: %f, val loss: %f, train ac: %f, val ac: %f"
          %(epoch, loss.detach().numpy(), val_loss, train_accuracy, val_accuracy))

if epoch % args.checkpoint_interval == 0:
    torch.save(model.state_dict(), os.path.join(saved_checkpoint_dir, "gcn_%d.pth"%epoch))
```

model.train() → 모델 학습 시 해야만 함 # 모델 파라미터 변경 가능하게 만드는 코드

writer.add_scalars : 저장 경로에 , 설정한 마커로 해당 내용을 입력

if 문 : 전체 epoch 중 log_interval 마다 terminal에 로그 출력

Model Train

4. Train 시작 – step3

```
optimizer.zero_grad() # Important  
loss.backward()  
optimizer.step()
```

- `optimizer.zero_grad()` : 파라미터 초기화
 - 기존 파라미터에 gradient 만큼 다시 업데이트 하기 위해 얼마큼 업데이트할지는 초기화 해줘야 함
- `loss.backward()` : loss 값을 기준으로 gradient 계산
- `optimizer.step()` : loss값을 기준으로 계산한 gradient 를 optimizer를 사용해 파라미터 update 진행 (back propagation)

→ 해당 코드는 모델의 파라미터를 업데이트하기 위한 방법으로 세 줄이 세트로 사용됨

Model Train

5. Train 실행 결과

```
Epoch: 0, train loss: 1.805936, val loss: 1.802304, train ac: 0.166667, val ac: 0.138000
Epoch: 10, train loss: 1.749251, val loss: 1.779348, train ac: 0.350000, val ac: 0.246000
Epoch: 20, train loss: 1.693586, val loss: 1.758669, train ac: 0.716667, val ac: 0.354000
Epoch: 30, train loss: 1.597373, val loss: 1.716361, train ac: 0.850000, val ac: 0.610000
Epoch: 40, train loss: 1.475136, val loss: 1.667309, train ac: 0.883333, val ac: 0.608000
Epoch: 50, train loss: 1.353376, val loss: 1.593787, train ac: 0.908333, val ac: 0.674000
Epoch: 60, train loss: 1.185840, val loss: 1.547290, train ac: 0.916667, val ac: 0.680000
Epoch: 70, train loss: 1.084034, val loss: 1.490311, train ac: 0.916667, val ac: 0.690000
Epoch: 80, train loss: 0.935883, val loss: 1.392693, train ac: 0.941667, val ac: 0.704000
Epoch: 90, train loss: 0.829452, val loss: 1.352148, train ac: 0.941667, val ac: 0.702000
Epoch: 100, train loss: 0.785350, val loss: 1.357048, train ac: 0.950000, val ac: 0.696000
Epoch: 110, train loss: 0.727133, val loss: 1.313826, train ac: 0.950000, val ac: 0.702000
Epoch: 120, train loss: 0.680854, val loss: 1.324749, train ac: 0.941667, val ac: 0.694000
Epoch: 130, train loss: 0.619142, val loss: 1.286676, train ac: 0.958333, val ac: 0.708000
Epoch: 140, train loss: 0.601118, val loss: 1.239932, train ac: 0.958333, val ac: 0.702000
Epoch: 150, train loss: 0.579640, val loss: 1.257445, train ac: 0.958333, val ac: 0.702000
Epoch: 160, train loss: 0.528573, val loss: 1.291705, train ac: 0.966667, val ac: 0.710000
Epoch: 170, train loss: 0.521385, val loss: 1.254384, train ac: 0.975000, val ac: 0.698000
Epoch: 180, train loss: 0.547872, val loss: 1.261969, train ac: 0.975000, val ac: 0.700000
Epoch: 190, train loss: 0.494727, val loss: 1.223556, train ac: 0.975000, val ac: 0.708000
Epoch: 200, train loss: 0.472139, val loss: 1.247393, train ac: 0.975000, val ac: 0.702000
```

Chapter 6

Model Evaluate

Model Evaluate

- 성능 평가 진행

```
def evaluate(checkpoint):  
    model.load_state_dict(torch.load(checkpoint))  
    model.eval()  
    outputs = model(adj, features)  
    accuracy = get_accuracy(outputs, y_test, test_mask)  
    print("Accuracy on test set is %f" %accuracy)
```

- Dataset만 test data로 수정. 과정은 train 과 동일.

Accuracy on test set is 0.716000

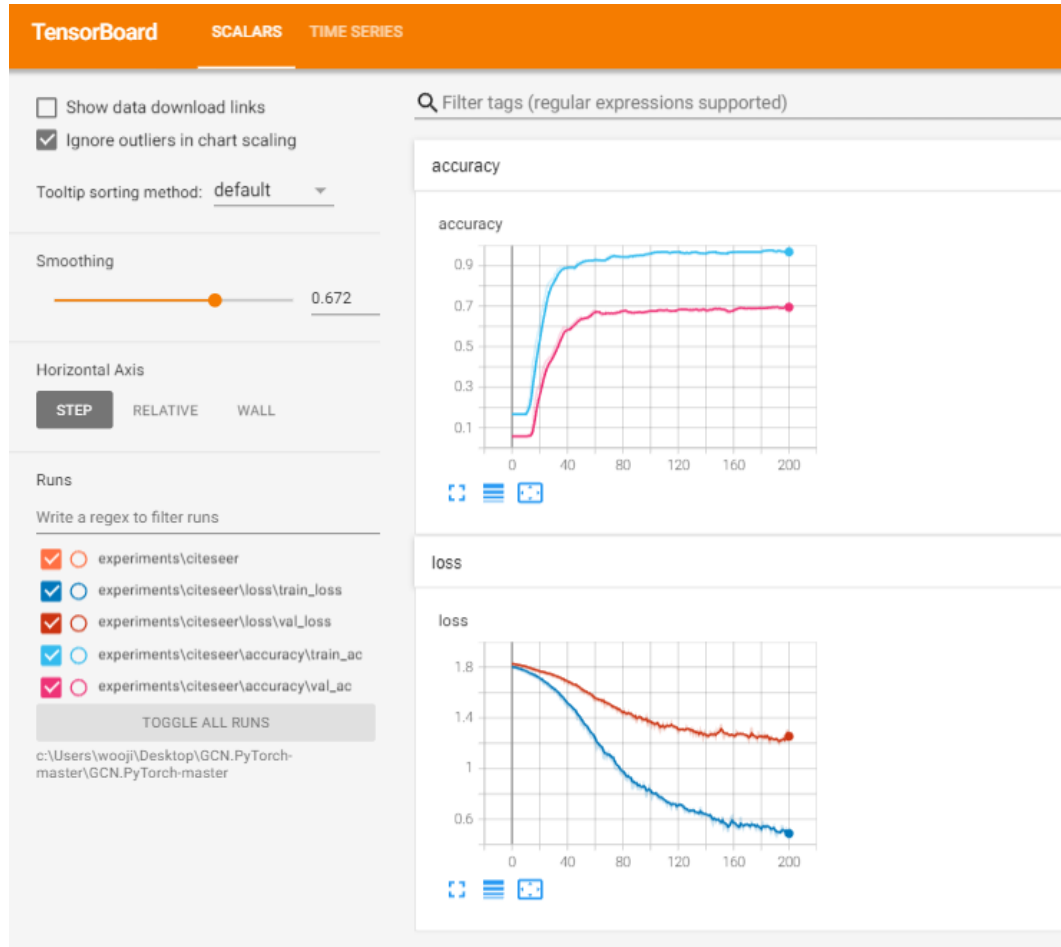
Accuracy on test set is 0.709000

- 결과는 accuracy 0.716정도 | train_size 두 배로 늘렸을 시 accuracy 0.709정도
 - train set에 대해 0.975 수준의 정확도를 가졌고 test set에 대해서는 0.716 수준의 정확도를 가진다.
 - 차이가 큰 것으로 볼 수도 있으며 과적합을 고려하여 성능개선도 가능할 것으로 볼 수 있다.

Chapter 7

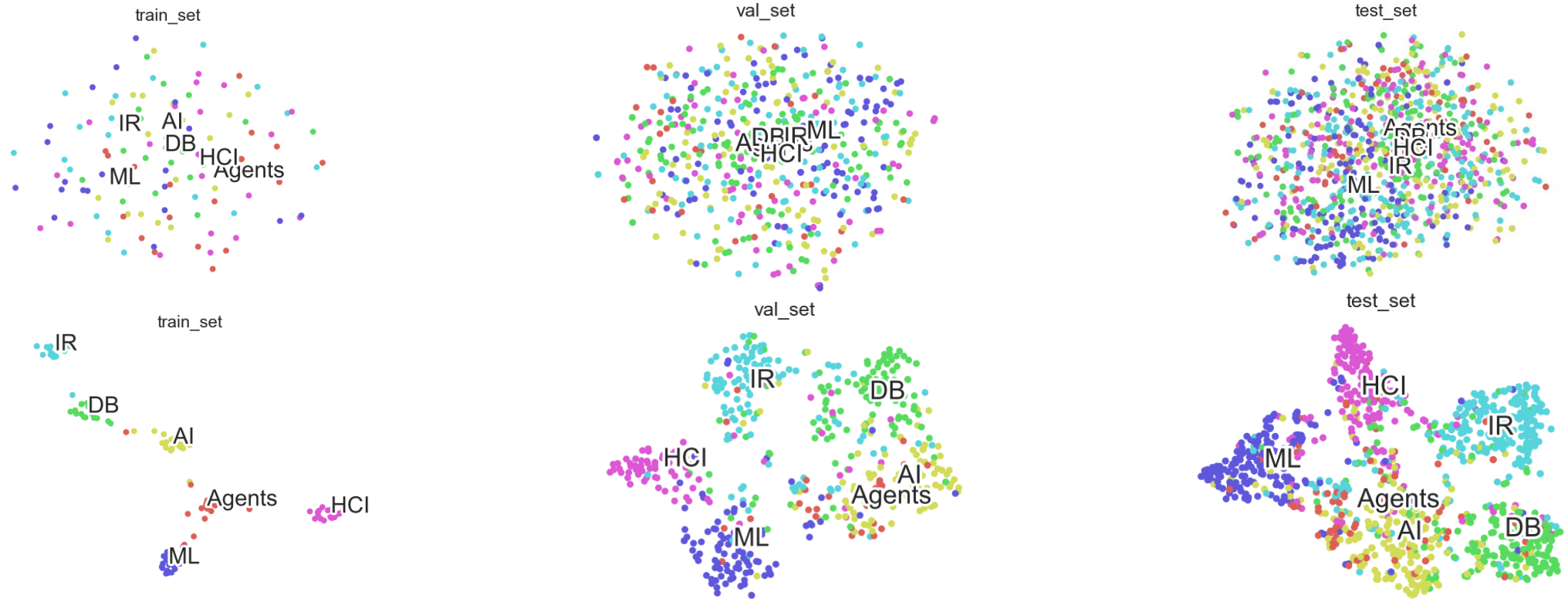
Result in Tensorboard

Visualize



- SummaryWriter 의 TensorboardX를 통해서 다음과 같이 실시간의 정확도 및 loss를 체크 가능하다.
- 이는 대부분 기계학습에 전부다 적용 가능하다.

Visualize



- Visualize 코드를 통해서 다음과 같이 모델의 노드 분류 결과를 시각화 할 수 있다.
- Checkpoint 변수에 " "을 입력하면 0 epoch 모델의 분류 결과를
- Checkpoint 변수에 "모델 경로" 를 입력하면 해당 모델의 분류 결과를 확인할 수 있다.

하지만, numpy에서 더이상 지원하지 않는 옛날방식 코드이다

GCN for link prediction

■ GCN을 활용한 link(edge) prediction

- https://github.com/quovadisss/GCN_linkprediction

Thank you