Graph Pooling: DiffPool

Pytorch Geometric – Tutorial 18

2025.02.11

윤정근
wangddaa27@chungbuk.ac.kr
BigData Lab.

목차

1. Introduction (1/3)

Pooling의 개념

- **Pooling**: 데이터를 축소하여 복잡한 구조를 단순화

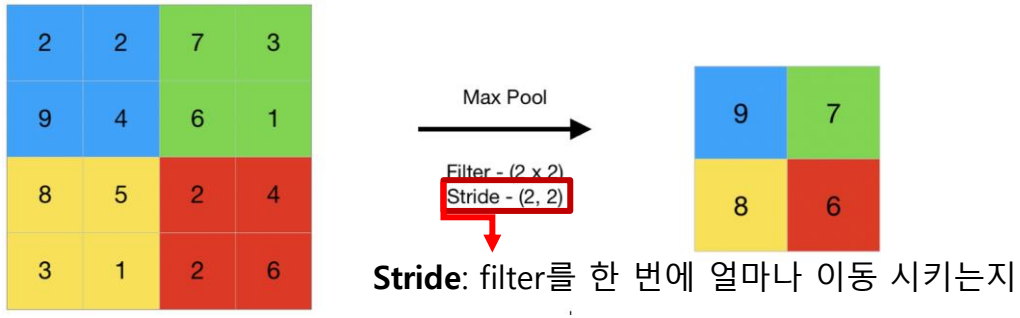
➔ 차원 감소 ▶ 계산 효율성 향상, 메모리 효율 증가

➔ 이미지 데이터를 다루는 CNN에서 공간 정보를 축소할 때 사용

- 대표적인 Pooling연산

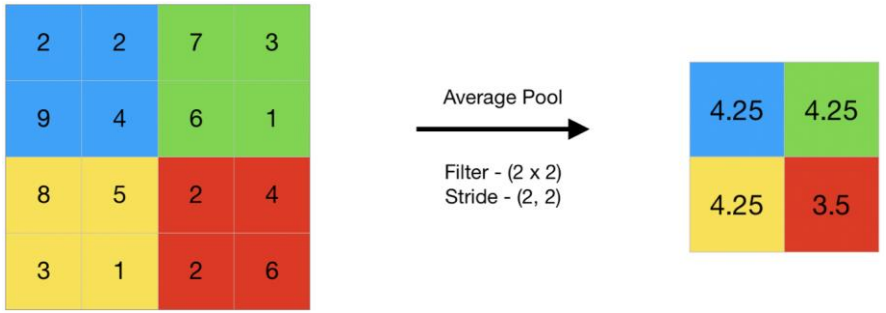
1. MAX Pooling: **지정된 영역(Filter)**에서 최댓값 출력
2. Avg Pooling: **지정된 영역(Filter)**에서 평균 값을 계산하여 출력

MAX Pooling



Stride: filter를 한 번에 얼마나 이동 시키는지

AVG Pooling



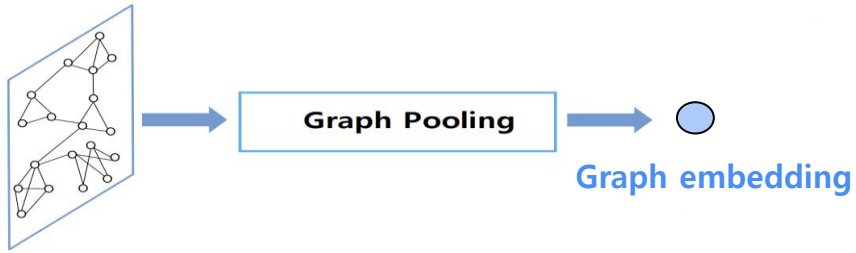
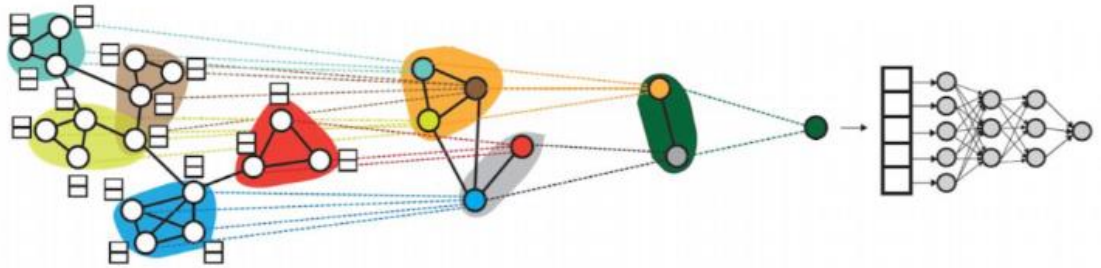
1. Introduction (2/3)

그래프에서 Pooling

- 이러한 Pooling은 그래프에서도 사용됨
- GNN에서 그래프 수준의 Task(예: graph classification)를 수행하기 위해서는 그래프 전체를 요약하는 전역 표현이 필요
- **그래프 요약** ▶ **Graph Pooling**
- **Graph Pooling:** 그래프의 각 node을 embedding하여 그래프 전체를 대표하는 Graph Embedding 생성

- 대표적인 Graph Pooling

1. **Flat Graph Pooling (2017)**: 그래프의 노드 정보를 단순히 평균 또는 합산하여 축소
   - 특별한 연산 과정이 없기 때문에 단조로운 임베딩 생성
2. **Hierarchical Graph Pooling (2019)**: 그래프의 구조적 정보를 점진적으로 축소해 나가며 임베딩 생성

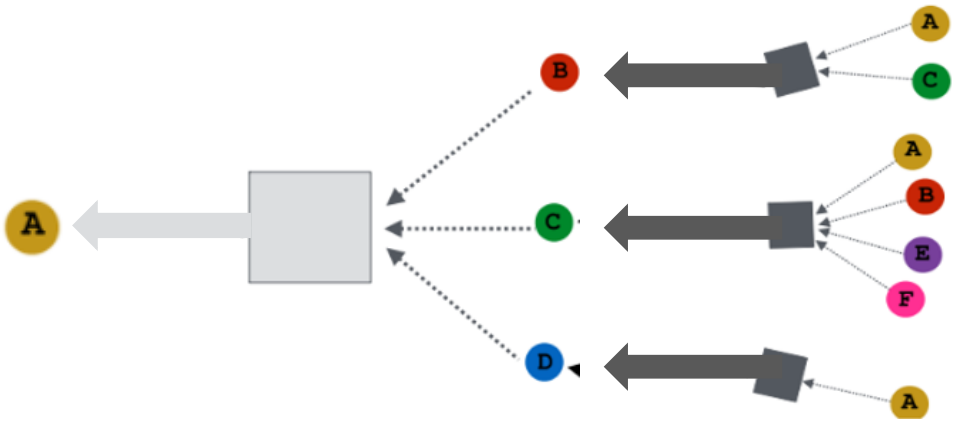
Flat Graph Pooling	Hierarchical Graph Pooling
	

1. Introduction (3/3)

그래프 표현에 적합한 Graph Pooling

- Flat Pooling은 그래프 구조를 축소하기 위해 단순히 모든 정점의 정보를 계산 ▶ GNN의 계층적인 표현 정보를 학습하지 못함
- 계산이 Flat Pooling에 비해 복잡할지라도 Hierachical Graph Pooling이 GNN의 복잡한 계층 관계를 반영
  ▶ **Hierachical Graph Pooling(DiffPool) 이 GNN에 적합**

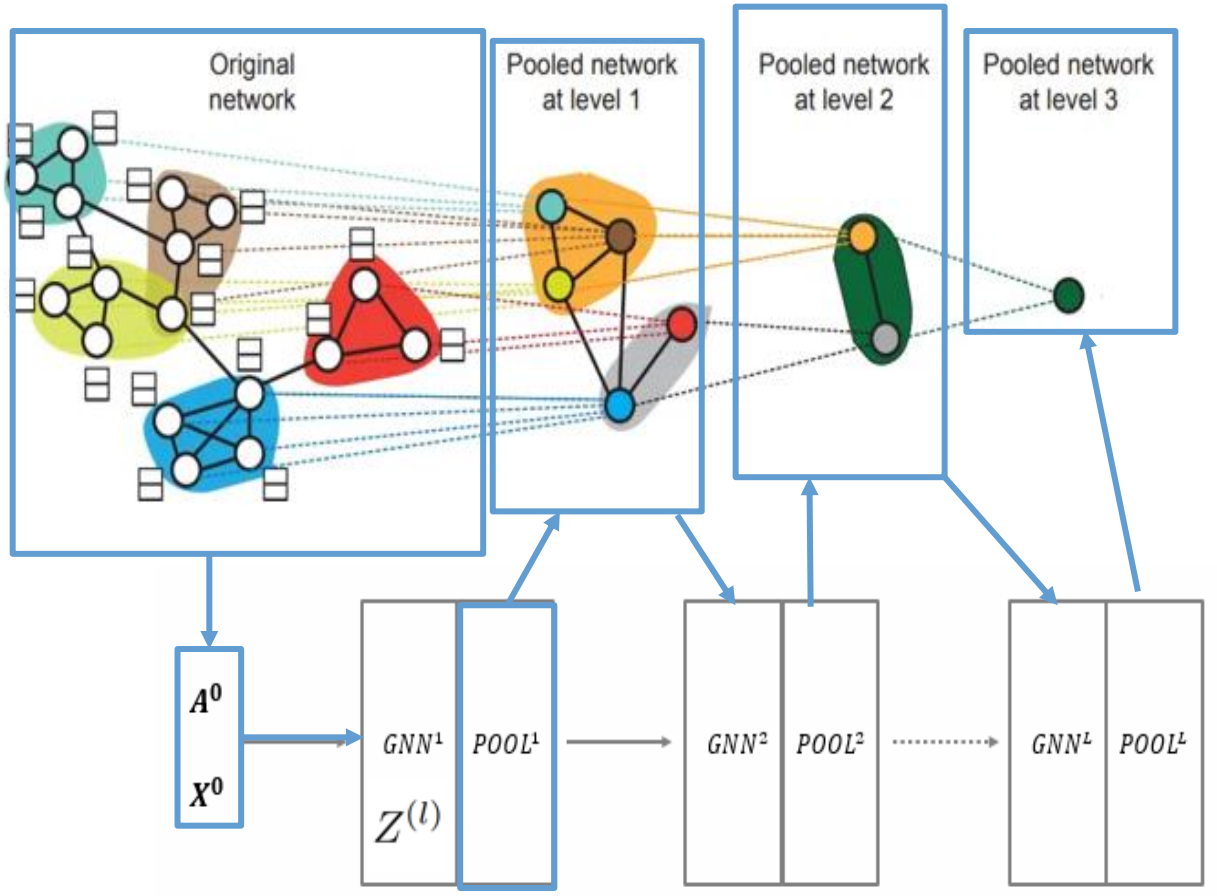
- **GNN**은 대상 node의 이웃들의 정보를 집계하는 과정을 반복적으로 수행
- Input으로 각 node vector를 사용하여 레이어마다 새로운 embedding을 얻고 이를 다시 다음 레이어의 input으로 사용
  ▶ **계층적 성질**



2. Graph Pooling – DiffPool (1/6)

2.1 전체 동작 설명

1. Original network에서 데이터 입력 **(A, X)**
2. GNN을 통한 Embedding **(Z^l)**
3. Clustering을 활용한 노드 개수 축소 ▶ Pooling
4. GNN과 Pooling 반복
5. 최종적으로 클러스터링 된 값으로 Graph Embedding 생성



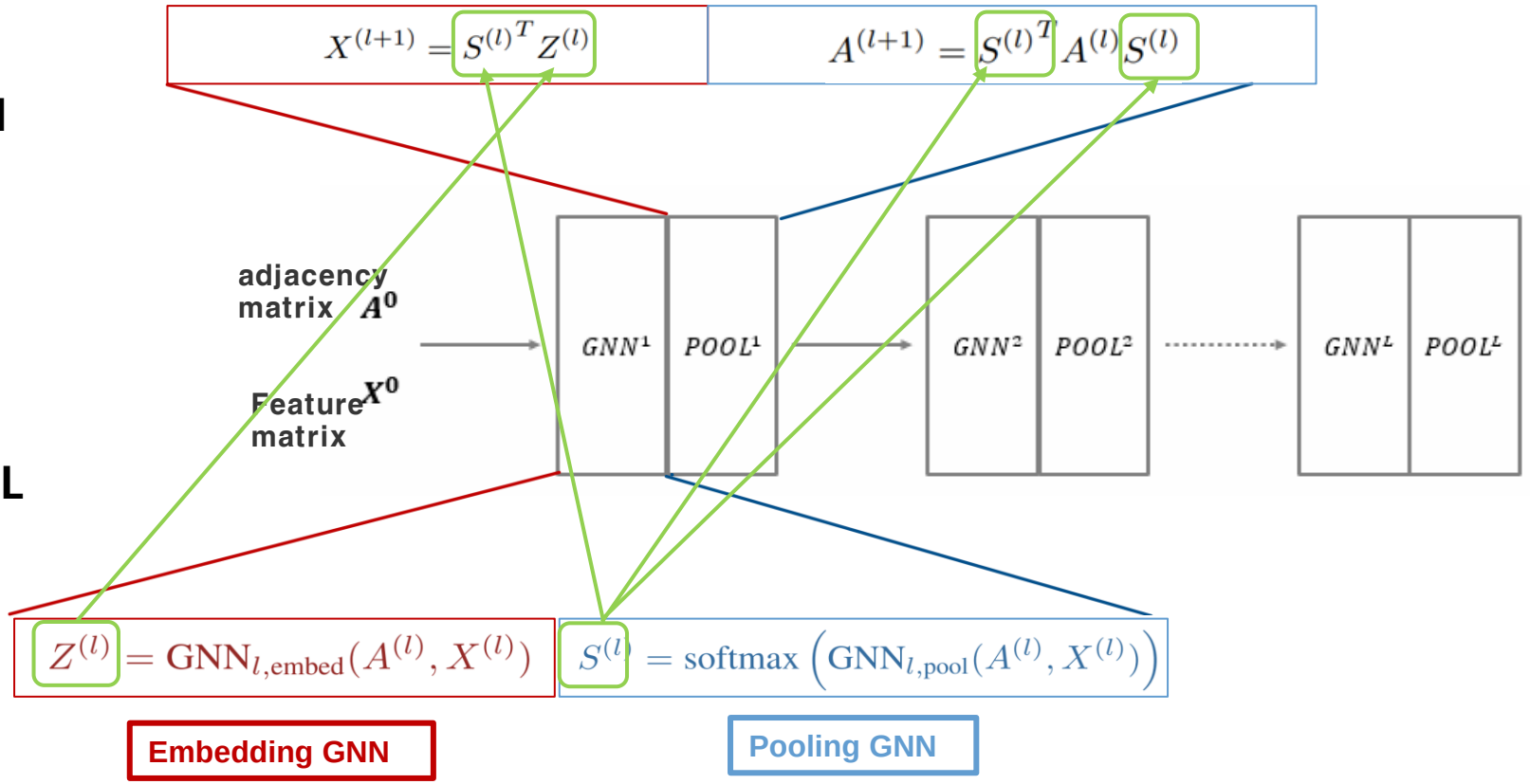
2. Graph Pooling – DiffPool (2/6)

2.2 Embedding, Pooling

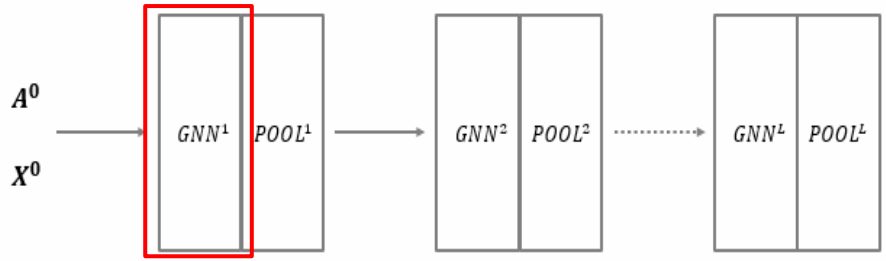
- Node feature matrix $X^{(l)}$과 Adjacency matrix $A^{(l)}$로 Embedding 한 값 ▶ **$Z^{(l)}$**
- 다음 레벨의 클러스터에 속할 확률 ▶ Cluster assignment matrix **$S^{(l)}$** (활성화 함수softmax 사용)
- $S^{(l)}$의 전치 행렬과 Z를 통해 다음 레벨의 **Cluster feature matrix ($X^{(l+1)}$)**생성
- $S^{(0)}$과 $S^{(0)}$ 전치행렬, 현재 인접 행렬로 다음 레벨의 **인접 강도** 생성
- 그 다음 $S^{(1)}$을 입력 받을 땐 인접 행렬이 아닌 인접 강도를 받음

GNN

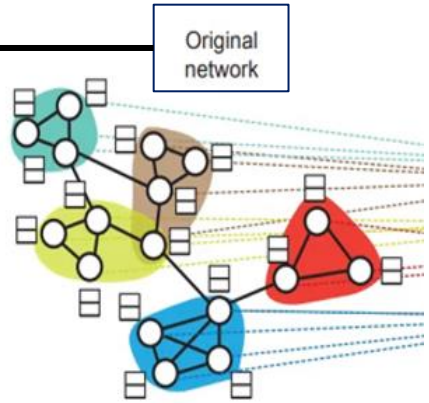
POOL



2. Graph Pooling – DiffPool (3/6)



2.3 Clustering을 활용한 Pooling (1/3)



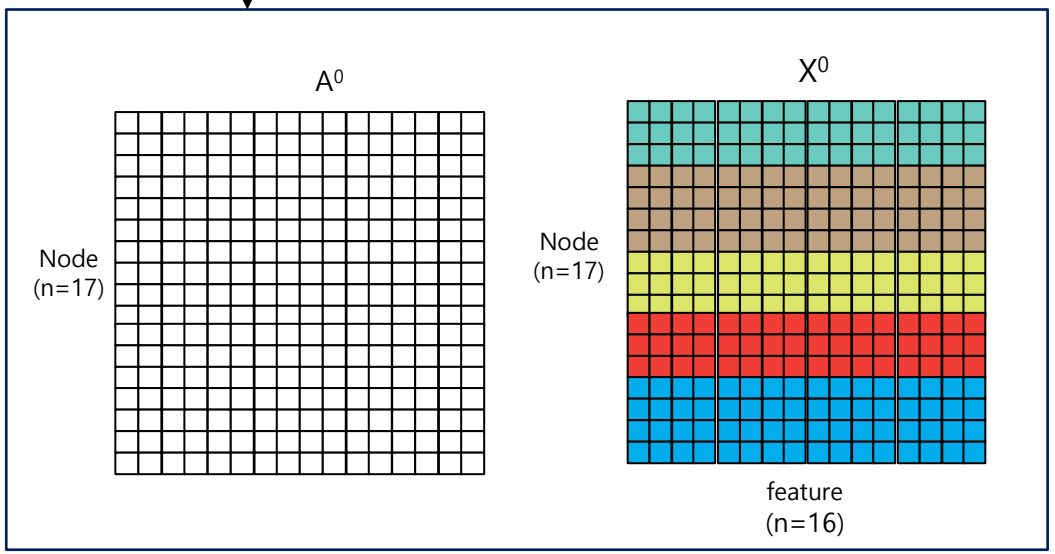
Node: 17개

Feature: 16

pooling

Node: 5개

Feature: 8



Hyper parameters 설정

1. 다음 레벨의 클러스터 개수

```
n_clusters_0 = n_nodes
n_clusters_1 = 5
```

2. 다음 레벨의 임베딩 차원 정의

```
hidden_dim = 8
```

3. **layer** 수 정의

```
n_layers = 3
```

$$Z^{(0)} = GNN_{l,embed}(A^{(0)},X^{(0)})$$

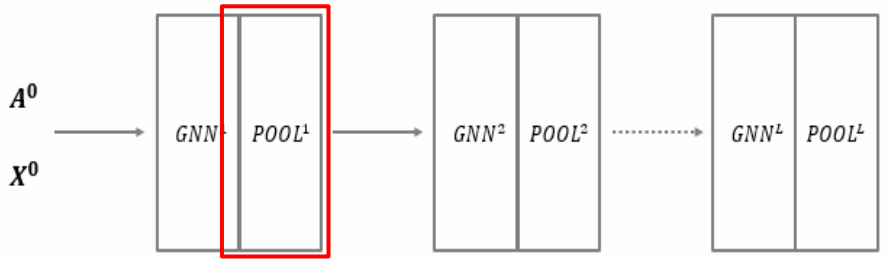
$$S^{(0)} = \textbf{softmax}(GNN_{l,pool}(A^{(0)},X^{(0)}))$$

```
print(z_0.shape)
print(s_0.shape)
```

```
torch.Size([17, 8])
torch.Size([17, 5])
```

2. Graph Pooling – DiffPool (4/6)

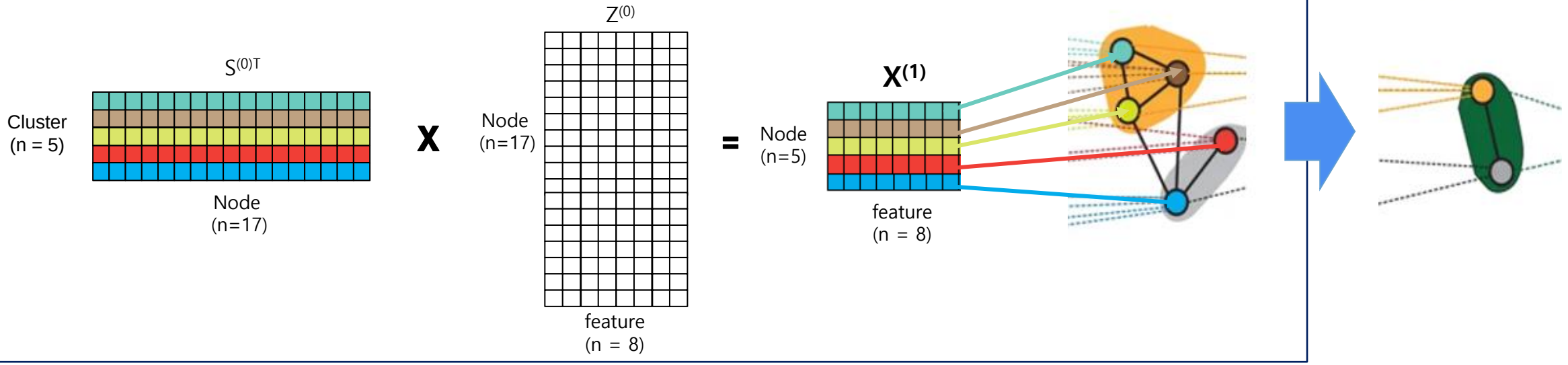
2.3 Clustering을 활용한 Pooling (2/3)



다음 레벨의 Cluster feature matrix X ⇨ $X^{(1)} = S^{(0)T} Z^{(0)}$

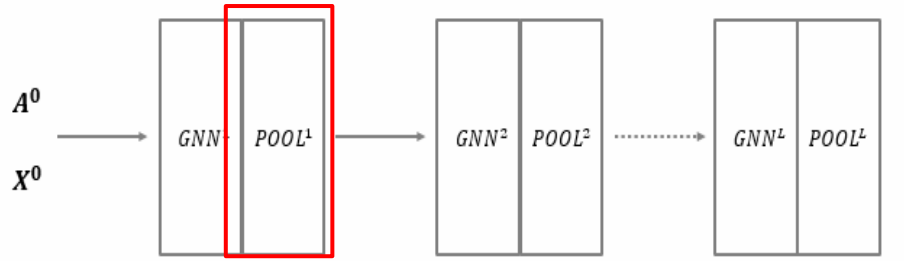
Embedding을 위해 현재 레벨의 인접 강도 필요

$S^{(0)T}$	X	임베딩된 **Feature Matrix** $Z^{(0)}$	=	Cluster feature matrix $X^{(1)}$
Cluster: 5개 **Node: 17개**		Feature: 8개 Node: 17개		Feature: 8개 **Node: 5개**



2. Graph Pooling – DiffPool (5/6)

2.3 Clustering을 활용한 Pooling (3/3)



다음 레벨의 인접 **강도 행렬**

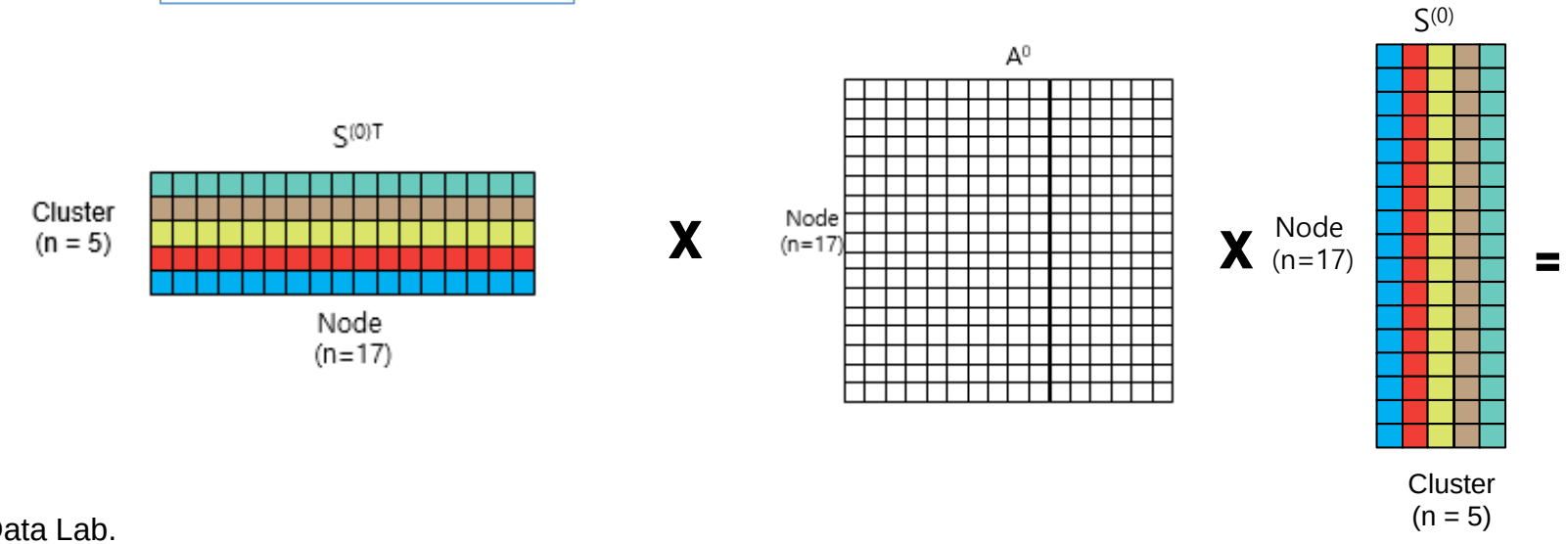
$$A'^{(1)} = S^{(0)T} A^{(0)} S^{(0)}$$

전치 행렬 $S^{(0)T}$
Cluster: 5개
Node: 17개

현재 레벨의 인접행렬
17 x 17

$S^{(0)}$
Node: 17개
Cluster: 5개

다음 레벨의 인접 강도
5x5

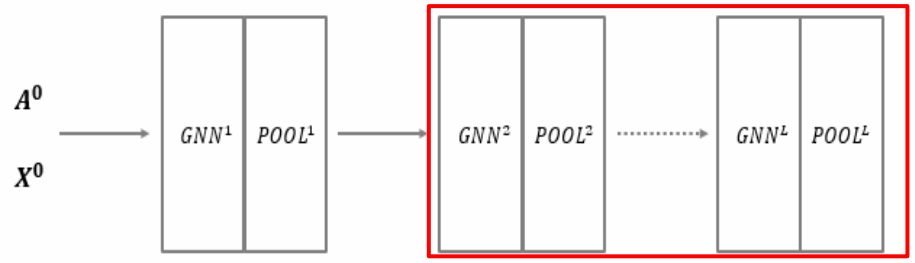


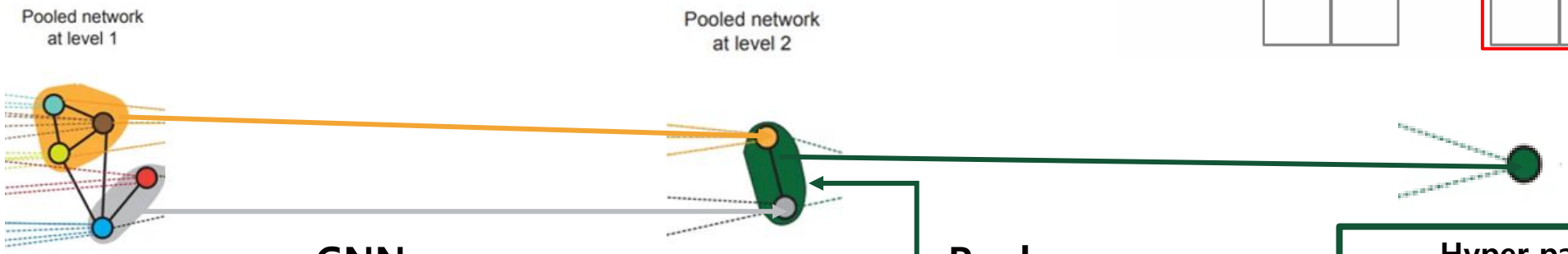

```
1 print(adj_1.shape)
```

```
torch.Size([5, 5])
```

2. Graph Pooling – DiffPool (6/6)

2.4 반복을 통한 Graph Embedding





GNN

GNN_{embed}(Node (n=5) × feature (n = 8) , A^1) → $Z^{(1)}$

GNN_{pool}(Node (n=5) × feature (n = 8) , A^1) → $S^{(1)}$

Pool

Hyper parameters 설정
Cluster: 2개
Feature: 4개

$X^{(2)} = S^{(1)T} Z^{(1)}$

feature (n = 4)

Cluster (n = 2)

$A^{(2)} = S^{(1)T} A^{(1)} S^{(1)}$

Node (n=2)

Hyper parameters 설정
Cluster: 1개
Feature: 2개

GNN → $S^{(2)}$, $Z^{(2)}$ → Pool → $X^{(3)}$, $A^{(3)}$ → $Z^{(3)}$

Graph embedding

3. Pytorch Geometric 활용

전체 과정 소개

- **목적: PROTEINS** 데이터셋을 활용하여 그래프 분류를 위해 PyTorch Geometric(PyG)를 사용 GNN과 Graph Pooling기법을 구현하고 성능을 검증

- **방법**

1. 데이터셋 로드 – PROTEINS 데이터 셋
2. 데이터 전처리
3. 모델 설계
4. 훈련 및 테스트
5. 결과 확인

Name	#graphs	#nodes	#edges	#features	#classes
MUTAG	188	~17.9	~39.6	7	2
ENZYMES	600	~32.6	~124.3	3	6
PROTEINS	1,113	~39.1	~145.6	3	2
COLLAB	5,000	~74.5	~4914.4	0	3
IMDB-BINARY	1,000	~19.8	~193.1	0	2
REDDIT-BINARY	2,000	~429.6	~995.5	0	2
...					

3. Pytorch Geometric 활용 (1/8)

1. 데이터셋 로드

```
import os.path as osp
from math import ceil

import torch
import torch.nn.functional as F
from torch_geometric.datasets import TUDataset
import torch_geometric.transforms as T
from torch_geometric.data import DenseDataLoader
from torch_geometric.nn import DenseGCNConv, dense_diff_pool
```

Name	#graphs	#nodes	#edges	#features	#classes
MUTAG	188	~17.9	~39.6	7	2
ENZYMES	600	~32.6	~124.3	3	6
PROTEINS	1,113	~39.1	~145.6	3	2
COLLAB	5,000	~74.5	~4914.4	0	3
IMDB-BINARY	1,000	~19.8	~193.1	0	2
REDDIT-BINARY	2,000	~429.6	~995.5	0	2
...					

그래프 데이터를 밀집 표현으로 변환

각 그래프 데이터는 노드와 엣지 수가 다르기 때문에 한 번에 처리하기 어려움
밀집 표현으로 변환하여 동일한 크기로 맞춤

```
max_nodes = 150
dataset = TUDataset('data', name='PROTEINS', transform=T.ToDense(max_nodes),
                    pre_filter=MyFilter())
```

목적: 단백질 구조 분석

정보

1. Graph 수: 1,113개
2. 평균 Node 수: 약 39.1개
3. 평균 Edge 수: 약 145.6개
4. Node feature 수: 3개
5. Class 수: 2개 (효소O, X)

3. Pytorch Geometric 활용 (2/8)

2. Data 전처리

```
1 batch_size = 32
```

32개의 그래프를 한 번에 처리

```
1 n = (len(dataset) + 9) // 10
2
3 test_dataset = dataset[:n]
4 val_dataset = dataset[n:2 * n]
5 train_dataset = dataset[2 * n:]
6
7 test_loader = DenseDataLoader(test_dataset, batch_size=32)
8 val_loader = DenseDataLoader(val_dataset, batch_size=32)
9 train_loader = DenseDataLoader(train_dataset, batch_size=32)
```

데이터셋을 테스트(10%), 검증(10%), 학습(80%)로 분리
Test: 112
Validation: 112
Train: 889

Dense 형식으로 변환된 그래프를 Batch 단위로 처리

32개의 그래프를 한 번에 훈련, 검증, 테스트

Batch 처리 지원

```
1 for i in train_loader: #Dataloader 객체, 데이터셋을 배치 단위로 로드
2     print(i) #train_loader가 반환하는 각 배치 데이터. 보통 다음과 같은 데이터가 포함된다.
3     break
4
```

```
DataBatch(mask=[32, 150], adj=[32, 150, 150], x=[32, 150, 3], y=[32, 1])
```

유효한 노드와 패딩된 노드를 구분. 값이 1이면 해당 노드는 실제 데이터.

3. Pytorch Geometric 활용 (3/8)

3. 모델 설계 – GNN

```
class GNN(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels,
                 normalize=False, lin=True):
        super(GNN, self).__init__()

        self.convs = torch.nn.ModuleList()
        self.convs.append(DenseGCNConv(in_channels, hidden_channels, normalize))
        self.convs.append(DenseGCNConv(hidden_channels, hidden_channels, normalize))
        self.convs.append(DenseGCNConv(hidden_channels, out_channels, normalize))

        self.dropout = torch.nn.Dropout(p=0.6)
```

DenseGCNConv: Graph Convolution 연산을 수행하는 PyG layer.
입력 노드피처와 인접행렬을 사용하여 새로운 피처 생성
밀집 형태의 입력 데이터 지원

입력차원을 은닉차원으로 변환
마지막 계층에서 출력 차원 생성

```
def forward(self, x, adj, mask=None):
    # batch_size, num_nodes, in_channels = x.size()

    for step in range(len(self.convs)):
        x = F.relu(self.convs[step](x, adj, mask))
        x = self.dropout(x)
    return x
```

그래프 데이터 입력 받고 DenseGCNConv 레이어를 순차적으로 적용

3. Pytorch Geometric 활용 (4/8)

3. 모델 설계 – DiffPool(1/2)

```
class DiffPool(torch.nn.Module):
    def __init__(self):
        super(DiffPool, self).__init__()

        num_nodes = ceil(0.25 * max_nodes)
        self.gnn1_pool = GNN(dataset.num_features, 64, num_nodes)
        self.gnn1_embed = GNN(dataset.num_features, 64, 64)

        num_nodes = ceil(0.25 * num_nodes)
        self.gnn2_pool = GNN(64, 64, num_nodes)
        self.gnn2_embed = GNN(64, 64, 64, lin=False)

        self.gnn3_embed = GNN(64, 64, 64, lin=False)

        self.lin1 = torch.nn.Linear(64, 64)
        self.lin2 = torch.nn.Linear(64, dataset.num_classes)
```

하이퍼 파라미터 정의

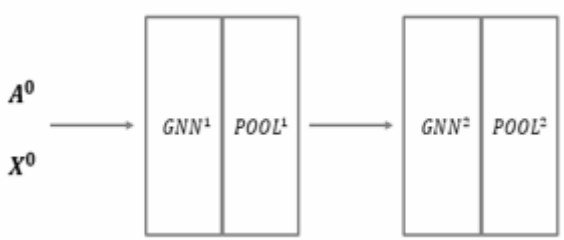
num_nodes는 0.25씩 "2번" Pooling 진행
➔ layer수: 2
➔ Layer1 클러스터 개수= 0.25 * 150 = 38
➔ Layer2 클러스터 개수= 0.25 * 38 = 10

hidden_dim = 64

그래프 데이터를 선형 layer를 통해 클래스 분류.
num_classes = 2
Self.lin1: 그래프 특징의 중간 변환
Self.lin2: 최종 출력 레이어, class 개수(2 개)로 분류

3. Pytorch Geometric 활용 (5/8)

3.2 모델 설계 – DiffPool(2/2)




```
def forward(self, x0, adj0, mask=None):
    s0 = self.gnn1_pool(x0, adj0, mask)
    z0 = self.gnn1_embed(x0, adj0, mask)

    x1, adj1, _, _ = dense_diff_pool(z0, adj0, s0, mask)
    #x_1 = s_0.t() @ z_0
    #adj_1 = s_0.t() @ adj_0 @ s_0)

    s1 = self.gnn2_pool(x1, adj1)
    z1 = self.gnn2_embed(x1, adj1)

    x2, adj2, _, _ = dense_diff_pool(z1, adj1, s1)

    z2 = self.gnn3_embed(x2, adj2)

    graph_vec = z2.mean(dim=1)
    graph_vec = F.relu(self.lin1(graph_vec))
    graph_vec = self.lin2(graph_vec)
    return F.softmax(graph_vec, dim=-1)
```

첫 번째 GNN과 Pooling
S0: 클러스터 할당 행렬
Z0: 첫 번째 GNN 계층에서 학습된 노드 임베딩
dense_diff_pool: 연산 수

두 번째 GNN, Pooling

그래프 임베딩 생성
Graph_vec: 각 그래프의 노드 특징을 평균하여 전역 그래프 특징 생성

최종 출력
그래프가 각 클래스에 속할 확률 softmax로 계산

3. Pytorch Geometric 활용 (6/8)

4. 훈련 및 테스트

```
model = DiffPool().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

```
def train(epoch):
    model.train()
    loss_all = 0

    for data in train_loader:
        data = data.to(device)
        optimizer.zero_grad()
        output, _, _ = model(data.x, data.adj, data.mask)
        loss = F.cross_entropy(output, data.y.view(-1))
        loss.backward()
        loss_all += data.y.size(0) * loss.item()
        optimizer.step()
    return loss_all / len(train_dataset)
```

Output, _, _ = model(data.x, data.daj, data.mask)
모델에 데이터를 입력하여 예측값 output 얻음

F.cross_entropy: 손실함수 계산

Loss.backward(): 역전파를 통해 그라디언트 계산

Optimizer.step(): 가중치 업데이트

에포크당 평균 손실 반환

```
@torch.no_grad()
def test(loader):
    model.eval()
    correct = 0

    for data in loader:
        data = data.to(device)
        pred = model(data.x, data.adj, data.mask)[0].max(dim=1)[1]
        correct += pred.eq(data.y.view(-1)).sum().item()
    return correct / len(loader.dataset)
```

모델에 데이터 입력 후 클래스별 점수 예측
Pred: 데이터 예측 값 계산 -> 출력 값 중 가장 높은 확률을 가진 클래스 선택 (.max(dim=1))

예측 값과 실제 값(data.y)이 같은 경우 계산

최종적으로 데이터셋 전체에 대한 정확도 반환

3. Pytorch Geometric 활용 (7/8)

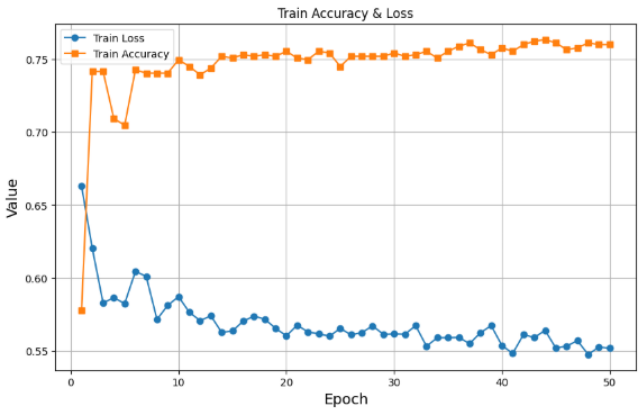
5. 결과 확인

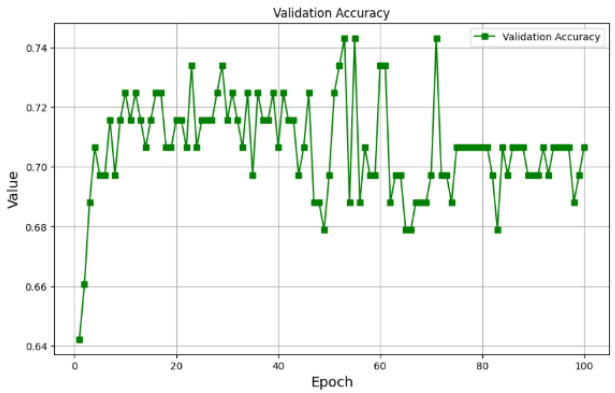
```
model.reset_parameters()
best_val_acc = test_acc = 0

for epoch in range(0, 40):
    train_loss = train(epoch)
    train_acc = test(train_loader)
    val_acc = test(val_loader)
    test_acc = test(test_loader)
    # if val_acc > best_val_acc:
    #     test_acc = test(test_loader)
    #     best_val_acc = val_acc
    print(f'Epoch: {epoch+1:03d}, Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f},
          f'Val Acc: {val_acc:.4f}, Test Acc: {test_acc:.4f}')
```

학습 및 손실을 계산하고 학습, 검증, 테스트 데이터의 정확도를 확인

Train Loss, Train Acc, val_acc, Test Acc를 추적하여 모델 성능 확인
Epoch는 40으로 설정

```
Epoch: 023, Train Loss: 0.5612, Train Acc: 0.7474, Val Acc: 0.7064, Test Acc: 0.6606
Epoch: 024, Train Loss: 0.5663, Train Acc: 0.7451, Val Acc: 0.7248, Test Acc: 0.6697
Epoch: 025, Train Loss: 0.5680, Train Acc: 0.7520, Val Acc: 0.7064, Test Acc: 0.6606
Epoch: 026, Train Loss: 0.5647, Train Acc: 0.7486, Val Acc: 0.7431, Test Acc: 0.6514
Epoch: 027, Train Loss: 0.5650, Train Acc: 0.7543, Val Acc: 0.7339, Test Acc: 0.6606
Epoch: 028, Train Loss: 0.5646, Train Acc: 0.7543, Val Acc: 0.7431, Test Acc: 0.6514
Epoch: 029, Train Loss: 0.5681, Train Acc: 0.7486, Val Acc: 0.7156, Test Acc: 0.6697
Epoch: 030, Train Loss: 0.5629, Train Acc: 0.7474, Val Acc: 0.7156, Test Acc: 0.6606
Epoch: 031, Train Loss: 0.5655, Train Acc: 0.7520, Val Acc: 0.7064, Test Acc: 0.6697
Epoch: 032, Train Loss: 0.5648, Train Acc: 0.7532, Val Acc: 0.7064, Test Acc: 0.6606
Epoch: 033, Train Loss: 0.5643, Train Acc: 0.7416, Val Acc: 0.6972, Test Acc: 0.6606
Epoch: 034, Train Loss: 0.5597, Train Acc: 0.7532, Val Acc: 0.7156, Test Acc: 0.6514
Epoch: 035, Train Loss: 0.5634, Train Acc: 0.7474, Val Acc: 0.7064, Test Acc: 0.6606
Epoch: 036, Train Loss: 0.5607, Train Acc: 0.7451, Val Acc: 0.7248, Test Acc: 0.6697
Epoch: 037, Train Loss: 0.5592, Train Acc: 0.7509, Val Acc: 0.7064, Test Acc: 0.6514
Epoch: 038, Train Loss: 0.5635, Train Acc: 0.7520, Val Acc: 0.7248, Test Acc: 0.6606
Epoch: 039, Train Loss: 0.5588, Train Acc: 0.7474, Val Acc: 0.7064, Test Acc: 0.6606
Epoch: 040, Train Loss: 0.5576, Train Acc: 0.7509, Val Acc: 0.7248, Test Acc: 0.6697
```





3. Pytorch Geometric 활용 (8/8)

실험 결과

Table 1: Classification accuracies in percent. The far-right column gives the relative increase in accuracy compared to the baseline GRAPHSAGE approach.

	Method	Data Set					
		ENZYMES	D&D	REDDIT-MULTI-12K	COLLAB	PROTEINS	Gain
Kernel	GRAPHLET	41.03	74.85	21.73	64.66	72.91	
Kernel	SHORTEST-PATH	42.32	78.86	36.93	59.10	76.43	
Kernel	1-WL	53.43	74.02	39.03	78.61	73.76	
Kernel	WL-OA	60.13	79.04	44.38	80.74	75.26	
GNN	PATCHYSAN	–	76.27	41.32	72.60	75.00	4.17
GNN	GRAPHSAGE	54.25	75.42	42.24	68.25	70.48	–
GNN	ECC	53.50	74.10	41.73	67.79	72.65	0.11
GNN	SET2SET	60.15	78.12	43.49	71.75	74.29	3.32
GNN	SORTPOOL	57.12	79.37	41.82	73.76	75.54	3.39
GNN	DIFFPOOL-DET	58.33	75.47	46.18	**82.13**	75.62	5.42
GNN	DIFFPOOL-NOLP	61.95	79.98	46.65	75.58	76.22	5.95
GNN	DIFFPOOL	**62.53**	**80.64**	**47.08**	75.48	**76.25**	**6.27**

다양한 데이터 셋으로 그래프 분류 방법 성능 비교
DiffPool은 5개의 데이터 세트 중 4 부분에서 최고 성능 기록

Table 2: Accuracy results of applying DIFFPOOL to S2V.

Data Set	Method		
	S2V	S2V WITH 1 DIFFPOOL	S2V WITH 2 DIFFPOOL
ENZYMES	61.10	62.86	**63.33**
D&D	78.92	80.75	**82.07**

DiffPool을 Structure2Vec(S2V)에 적용하여 계층적 구조 학습 성능 평가

DiffPool 적용 시 ENZYMES와 D&D 데이터 세트에서 성능 향상

다양한 GNN 아키텍처에 DiffPool 적용 시 성능 향상 가능

Conclusion

- **DiffPool은 그래프 데이터를 효과적으로 축소하여 분류 성능을 향상시키는 기법**
- 높은 정확도로 인해 높은 그래프 분류 성능

Q&A

Appendix

Differentiable? 왜 미분 가능한가?

1. 모든 수식이 행렬로 이루어져 있음
- 행렬 연산은 기본적으로 미분 가능.

$$A'^{(l+1)} = S^{(l)T} A^{(l)} S^{(l)} \qquad X^{(l+1)} = S^{(l)T} Z^{(l)}$$

2. Softmax 함수를 쓰기 때문에 이산값이 아닌 연속적인 실수가 쓰임
- DiffPool은 클러스터링(그래프 풀링)과정에서 활성화 함수인 softmax함수를 사용하여 각 노드를 여러 클러스터에 확률적으로 할당
- Softmax 함수는 연속적이기 때문에 모든 값에서 미분 가능