Introduction & Implementation of Graph neural networks with Pytorch Geometric

Reference:

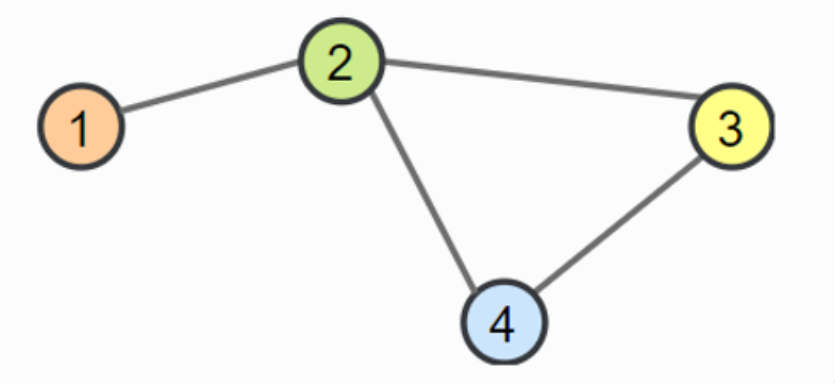
1. THOMAS KIPF, 'GRAPH CONVOLUTIONAL NETWORKS', 30 SEPTEMBER 2016, https://tkipf.github.io/graph-convolutional-networks/ (https://tkipf.github.io/graph-convolutional-networks/)
2. Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, Yoshua Bengio, 'Graph Attention Networks', ICLR 30 Oct 2017, p1-12
3. PYG DOCUMENTATION, 2022, https://pytorch-geometric.readthedocs.io/en/latest/ (https://pytorch-geometric.readthedocs.io/en/latest/)
4. University of Amsterdam, 2021,https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial7/GNN_overview.html (https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial7/GNN_overview.html)

목차

1. Graph Theory에서 그래프의 정의 - 그래프의 개념과 특성
2. Graph Convolution Networks와 Message Passing
3. Graph Attention
4. PyTorch Geometric
5. Node-level tasks : Semi-supervised node classification
6. Graph-level tasks : Graph classification
7. 다른 dataset(Amazon)에 적용

1. Graph의 개념과 특성

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



$$V = \{1, 2, 3, 4\} \qquad E = \{(1, 2), (2, 3), (2, 4), (3, 4)\}$$

1. 수학에서 그래프는 nodes/vertices 와 egdes/links 의 집합의 tuple 로 정의한다.

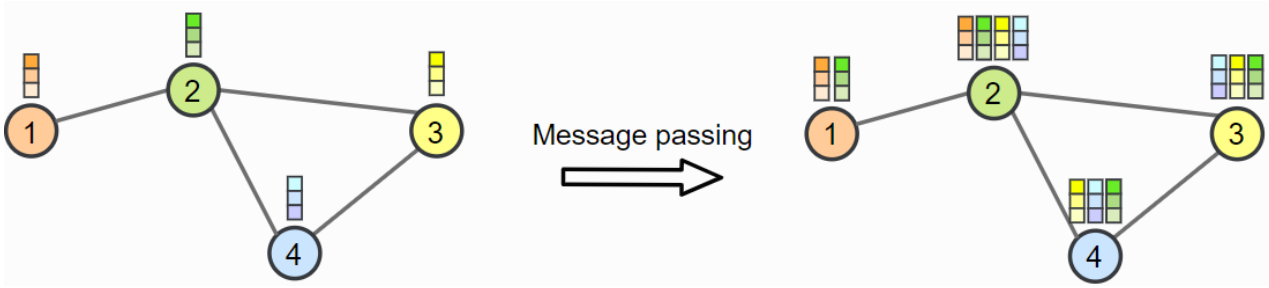
$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

2. 그래프의 노드 사이 연결점을 나타낸 행렬을 Adjacency matrix 라고 한다.

2. Graph Convolution Networks 와 Message Passing

1. Graph Convolutional Networks 는 2016년 암스테르담 대학의 Kipf 등이 소개함.
2. GCN 은 Message Passing 이라는 방법을 통해 작동한다.

Messag Passing



3. Message Passing 은 그래프의 vertices 가 이웃하는 nodes 끼리 정보를 교환하는 메커니즘을 말한다.

   여기서 Message 는 vertices 가 가지는 정보(feature vector)를,

   Passing 은 이웃하는 노드에 그 정보를 전달하는 것을 의미한다.

4. 여기서 각각의 노드에 전달되는 Message 를 어떻게 취합하는 방법에 따라

   다양한 종류의 GNN layer가 파생된다.

GCN layer

5. GCN layer 는 아래와 같이 정의한다.

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

1) H_l 은 vertices 의 input feature, W_l 은 weight parameters 이다.

2) A_hat 은 그래프의 adjacency matrix A에 단위 행렬을 더 한 것이다 : A_hat = A + I

3) D_hat 은 D(i,i) 성분이 node i 와 연결성을 가지는 node 의 개수로 이루어진 diagonal matrix 이다.

4) Sigmoid 는 임의의 activation function 을 의미한다(GNN 에서는 주로 ReLU-based).

5) H_l+1 은 GCN 을 통한 Message Passing operation 이 이루어진 output feature 이다.

6) GCN layer 에서 dropout 을 적용하는 경우 W_l 에서 dropout 이 이루어진다.

7) weight parameters W_l 가 그래프의 모든 노드에 대한 연산에서 공유되기 때문에

이미지의 Convolution 의 filter parameters 와 유사점을 가진다.

- torch_geometric 에서의 GCNConv

The graph convolutional operator from the "Semi-supervised Classification with Graph Convolutional Networks" paper

$$\mathbf{X}' = \hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{X}\mathbf{\Theta},$$

where $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ denotes the adjacency matrix with inserted self-loops and $\hat{D}_{ii} = \sum_{j=0} \hat{A}_{ij}$ its diagonal degree matrix. The adjacency matrix can include other values than `1` representing edge weights via the optional `edge_weight` tensor.

Its node-wise formulation is given by:

$$\mathbf{x}'_i = \mathbf{\Theta}^\top \sum_{j \in \mathcal{N}(v) \cup \{i\}} \frac{e_{j,i}}{\sqrt{\hat{d}_j \hat{d}_i}} \mathbf{x}_j$$

- edge 의 attridute 가 있는 경우 adjacency matrix 에 1 외 edge 에 상응하는 값을 가질 수 있다.

Pytorch implementation

In [1]:
```
## Standard libraries
import os
import json
import math
import numpy as np
import time

## Imports for plotting
import matplotlib.pyplot as plt
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg', 'pdf') # For export
from matplotlib.colors import to_rgb
import matplotlib
matplotlib.rcParams['lines.linewidth'] = 2.0
import seaborn as sns
sns.reset_orig()
sns.set()

## Progress bar
from tqdm.notebook import tqdm

## PyTorch
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data as data
import torch.optim as optim
# Torchvision
import torchvision
from torchvision.datasets import CIFAR10
from torchvision import transforms
# PyTorch Lightning
try:
    import pytorch_lightning as pl
except ModuleNotFoundError: # Google Colab does not have PyTorch Lightning installed by default. Hence, we
    !pip install --quiet pytorch-lightning>=1.4
    import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint

# Path to the folder where the datasets are/should be downloaded (e.g. CIFAR10)
DATASET_PATH = "../data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "../saved_models/tutorial7"

# Setting the seed
pl.seed_everything(42)

# Ensure that all operations are deterministic on GPU (if used) for reproducibility
torch.backends.cudnn.determinstic = True
torch.backends.cudnn.benchmark = False

device = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")
print(device)
```

```
C:WAnaconda3WlibWsite-packagesWtorchvisionWioWimage.py:11: UserWarning: Failed to load image Python extens
ion: Could not find module 'C:WAnaconda3WLibWsite-packagesWtorchvisionWimage.pyd' (or one of its dependenc
ies). Try using the full path with constructor syntax.
  warn(f"Failed to load image Python extension: {e}")
Global seed set to 42

cuda:0
```

In [2]:
```
import urllib.request
from urllib.error import HTTPError
# Github URL where saved models are stored for this tutorial
base_url = "https://raw.githubusercontent.com/phlippe/saved_models/main/tutorial7/"
# Files to download
pretrained_files = ["NodeLevelMLP.ckpt", "NodeLevelGNN.ckpt", "GraphLevelGraphConv.ckpt"]

# Create checkpoint path if it doesn't exist yet
os.makedirs(CHECKPOINT_PATH, exist_ok=True)

# For each file, check whether it already exists. If not, try downloading it.
for file_name in pretrained_files:
    file_path = os.path.join(CHECKPOINT_PATH, file_name)
    if "/" in file_name:
        os.makedirs(file_path.rsplit("/",1)[0], exist_ok=True)
    if not os.path.isfile(file_path):
        file_url = base_url + file_name
        print(f"Downloading {file_url}...")
        try:
            urllib.request.urlretrieve(file_url, file_path)
        except HTTPError as e:
            print("Something went wrong. Please try to download the file from the GDrive folder, or contact
```

GCNLayer with pytorch

In [3]:
```
class GCNLayer(nn.Module):

    def __init__(self, c_in, c_out):
        super().__init__()
        self.projection = nn.Linear(c_in, c_out)

    def forward(self, node_feats, adj_matrix):
        """
        Inputs:
            node_feats - Tensor with node features of shape [batch_size, num_nodes, c_in]
            adj_matrix - Batch of adjacency matrices of the graph. If there is an edge from i to j, adj_mat
                         Supports directed edges by non-symmetric matrices. Assumes to already have added tl
                         Shape: [batch_size, num_nodes, num_nodes]
        """
        # Num neighbours = number of incoming edges
        num_neighbours = adj_matrix.sum(dim=-1, keepdims=True)
        node_feats = self.projection(node_feats)
        node_feats = torch.bmm(adj_matrix, node_feats)
        node_feats = node_feats / num_neighbours
        return node_feats
```

1. weight matrix 를 linear layer 로 사용했기 때문에 bias 추가할 수도 있다.
2. node_feats 와 adj_matrix 의 shape 은 Pytorch 특성상 batch_size 가 추가된다.
3. 코드에서는 D_hat 을 정의하는 대신 Message 를 더하고, 이웃노드 개수로 나누는 연산으로 Message Passing 을 수행했다.

   H_l+1 = (A_hat * (H_l * W_l)) / num_neighours

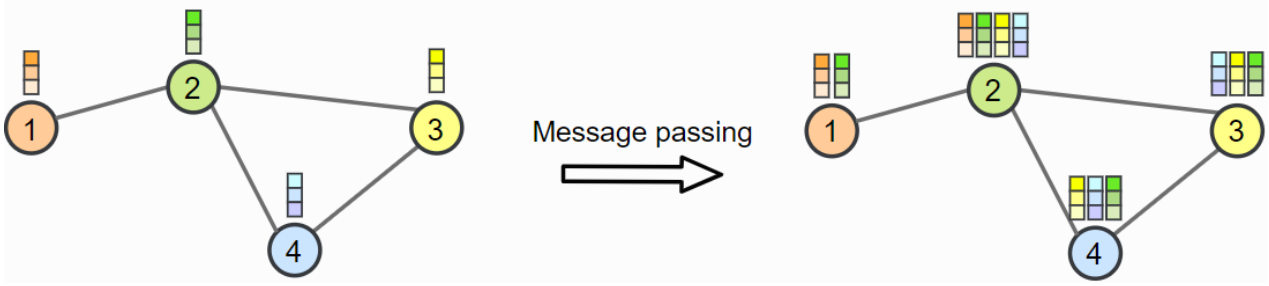
In [4]:
```
node_feats = torch.arange(8, dtype=torch.float32).view(1, 4, 2)
adj_matrix = torch.Tensor([[[1, 1, 0, 0],
                            [1, 1, 1, 1],
                            [0, 1, 1, 1],
                            [0, 1, 1, 1]]])

print("Node features:Wn", node_feats)
print("WnAdjacency matrix:Wn", adj_matrix)
```

```
Node features:
 tensor([[[0., 1.],
          [2., 3.],
          [4., 5.],
          [6., 7.]]])

Adjacency matrix:
 tensor([[[1., 1., 0., 0.],
          [1., 1., 1., 1.],
          [0., 1., 1., 1.],
          [0., 1., 1., 1.]]])
```

In [5]:
```
layer = GCNLayer(c_in=2, c_out=2)
layer.projection.weight.data = torch.Tensor([[1., 0.], [0., 1.]])
layer.projection.bias.data = torch.Tensor([0., 0.])

with torch.no_grad():
    out_feats = layer(node_feats, adj_matrix)

print("Adjacency matrix", adj_matrix)
print("Input features", node_feats)
print("Output features", out_feats)
```

```
Adjacency matrix tensor([[[1., 1., 0., 0.],
          [1., 1., 1., 1.],
          [0., 1., 1., 1.],
          [0., 1., 1., 1.]]])
Input features tensor([[[0., 1.],
          [2., 3.],
          [4., 5.],
          [6., 7.]]])
Output features tensor([[[1., 2.],
          [3., 4.],
          [4., 5.],
          [4., 5.]]])
```

결과적으로 아래 그림과 같은 neighbors nodes 사이에 feature exchange 를 달성할 수 있게 된다



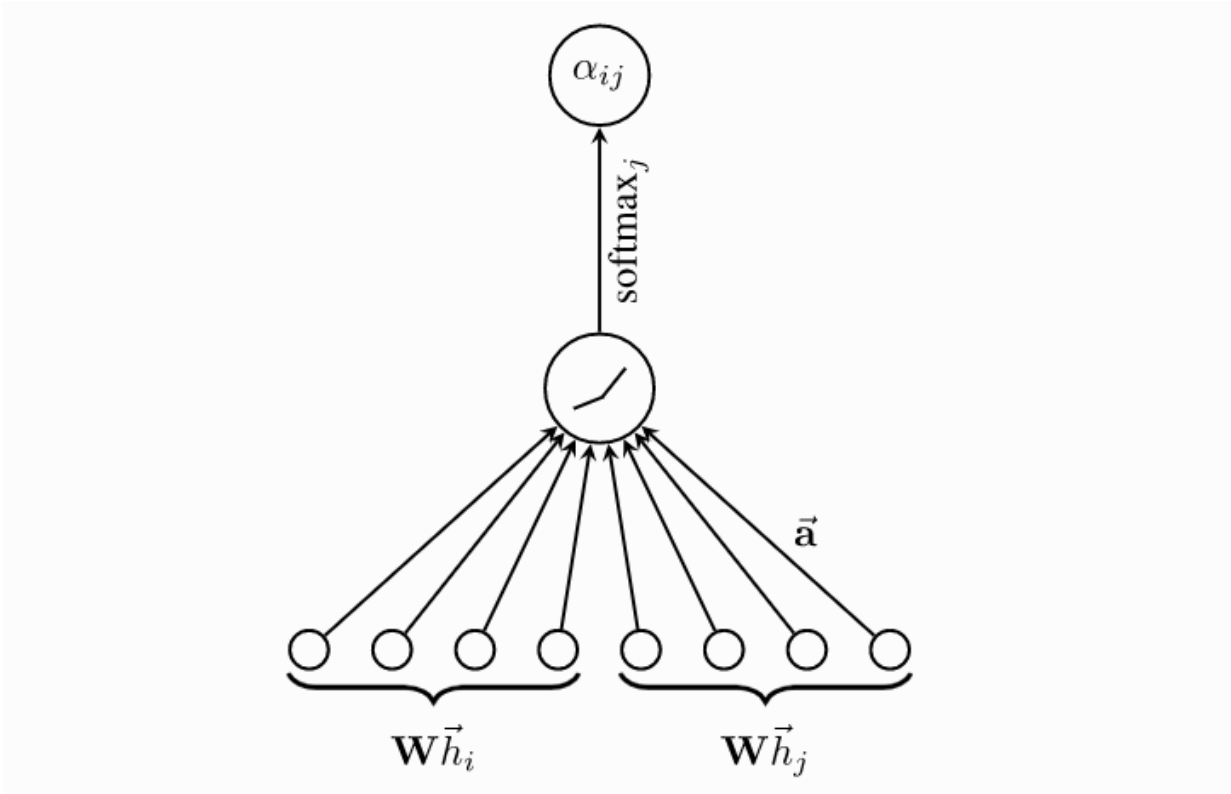
문제점

- 위 예시에서 node3과 node4의 output feature가 같음
- 이것은 messages passing operation에서 평균값을 사용할 경우

  node-specific information을 손실할 수 있음을 의미한다.
- 이에 대한 대안으로 Message passing 방식에 attention을 적용할 수 있다.

3. Graph Attention

1. 'Graph Attention Network' 은 Message Passing 방식에 Attention 을 적용한 것이다.
2. Graph Attention Network 는 2017년 Velickovic 등이 제안했고, GAT 로 지칭한다.



$$\alpha_{ij} = \frac{\exp\left(\text{LeakyReLU}\left(\mathbf{a}\left[\mathbf{W}h_i || \mathbf{W}h_j\right]\right)\right)}{\sum_{k\in\mathcal{N}_i}\exp\left(\text{LeakyReLU}\left(\mathbf{a}\left[\mathbf{W}h_i || \mathbf{W}h_k\right]\right)\right)}$$

1. h_i, h_j 는 node i and j 의 feature vector
2. W 는 weight matrix
3. the operator || - concatenation
4. a 는 weight matrix
5. N_i - node i 와 이웃하는 node 의 index
6. LeakyReLU - apply a non-linearity : 노드의 original input 의 정보을 살리는 중요한 역할을 한다
7. alpha_ij - attention coefficients

if the structure is linear

$$
\begin{aligned}
\alpha_{ij} &= \frac{\exp\left(\mathbf{a}\left[\mathbf{W}h_i || \mathbf{W}h_j\right]\right)}{\sum_{k\in\mathcal{N}_i}\exp\left(\mathbf{a}\left[\mathbf{W}h_i || \mathbf{W}h_k\right]\right)} \\
&= \frac{\exp\left(\mathbf{a}_{:,:d/2}\mathbf{W}h_i + \mathbf{a}_{:,d/2:}\mathbf{W}h_j\right)}{\sum_{k\in\mathcal{N}_i}\exp\left(\mathbf{a}_{:,:d/2}\mathbf{W}h_i + \mathbf{a}_{:,d/2:}\mathbf{W}h_k\right)} \\
&= \frac{\exp\left(\mathbf{a}_{:,:d/2}\mathbf{W}h_i\right)\cdot\exp\left(\mathbf{a}_{:,d/2:}\mathbf{W}h_j\right)}{\sum_{k\in\mathcal{N}_i}\exp\left(\mathbf{a}_{:,:d/2}\mathbf{W}h_i\right)\cdot\exp\left(\mathbf{a}_{:,d/2:}\mathbf{W}h_k\right)} \\
&= \frac{\exp\left(\mathbf{a}_{:,d/2:}\mathbf{W}h_j\right)}{\sum_{k\in\mathcal{N}_i}\exp\left(\mathbf{a}_{:,d/2:}\mathbf{W}h_k\right)}
\end{aligned}
$$

- 이는 node-specific information 이는 h_i 가 없어지는 것으로

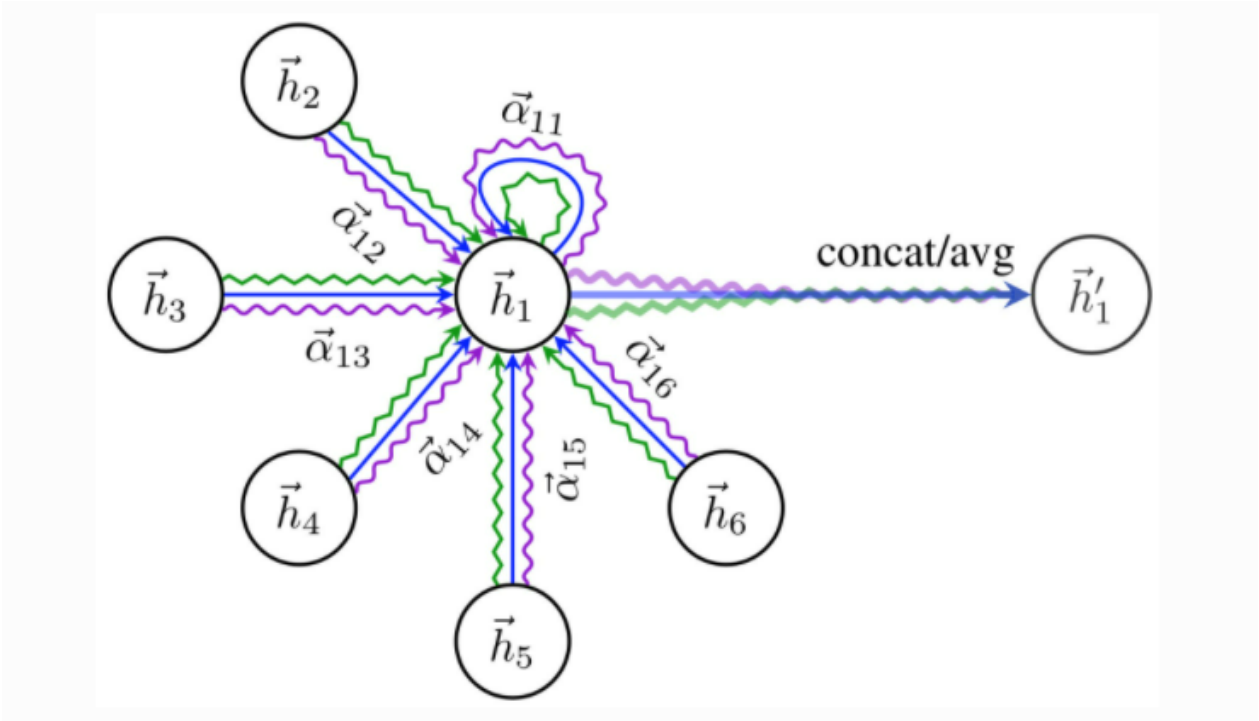
  이웃하는 노드가 동일한 노드들이 있는 경우

  node-specific information을 손실하는 문제 발생하게 된다.

$$
h_i' = \sigma\left(\sum_{j\in\mathcal{N}_i}\alpha_{ij}\mathbf{W}h_j\right)
$$

8. 최종적으로 위와 같이 attention 을 적용한 message passing 연산을 수행하여 h_i' 을 계산한다
9. sigmoid 는 임의의 non-linearity

Multi-Head Attention 을 적용하는 경우



1. Multi-Head Attention 은 N attention 을 병렬적으로 수행한다.
2. N attention 의 결과물은 concatenate / avg 방식으로 최종 output feature 를 도출한다

In [6]:
```
class GATLayer(nn.Module):

    def __init__(self, c_in, c_out, num_heads=1, concat_heads=True, alpha=0.2):
        """
        Inputs:
            c_in - Dimensionality of input features
            c_out - Dimensionality of output features
            num_heads - Number of heads, i.e. attention mechanisms to apply in parallel. The
                        output features are equally split up over the heads if concat_heads=True.
            concat_heads - If True, the output of the different heads is concatenated instead of averaged.
            alpha - Negative slope of the LeakyReLU activation.
        """
        super().__init__()
        self.num_heads = num_heads
        self.concat_heads = concat_heads
        if self.concat_heads:
            assert c_out % num_heads == 0, "Number of output features must be a multiple of the count of h
            c_out = c_out // num_heads
            # assert - 가정 설명문: assert 가정조건, asserterror시 출력문 - '방어적 프로그래밍'

        # Sub-modules and parameters needed in the layer
        self.projection = nn.Linear(c_in, c_out * num_heads)
        self.a = nn.Parameter(torch.Tensor(num_heads, 2 * c_out)) # One per head
        self.leakyrelu = nn.LeakyReLU(alpha)

        # Initialization from the original implementation
        nn.init.xavier_uniform_(self.projection.weight.data, gain=1.414)
        nn.init.xavier_uniform_(self.a.data, gain=1.414)

    def forward(self, node_feats, adj_matrix, print_attn_probs=False):
        """
        Inputs:
            node_feats - Input features of the node. Shape: [batch_size, c_in]
            adj_matrix - Adjacency matrix including self-connections. Shape: [batch_size, num_nodes, num_no
            print_attn_probs - If True, the attention weights are printed during the forward pass (for debu
        """
        batch_size, num_nodes = node_feats.size(0), node_feats.size(1)

        # Apply linear layer and sort nodes by head
        node_feats = self.projection(node_feats)
        node_feats = node_feats.view(batch_size, num_nodes, self.num_heads, -1)

        # We need to calculate the attention logits for every edge in the adjacency matrix
        # Doing this on all possible combinations of nodes is very expensive
        # => Create a tensor of [W*h_i||W*h_j] with i and j being the indices of all edges
        edges = adj_matrix.nonzero(as_tuple=False) # Returns indices where the adjacency matrix is not 0 =>
        node_feats_flat = node_feats.view(batch_size * num_nodes, self.num_heads, -1)
        edge_indices_row = edges[:,0] * num_nodes + edges[:,1]
        edge_indices_col = edges[:,0] * num_nodes + edges[:,2]
        a_input = torch.cat([
            torch.index_select(input=node_feats_flat, index=edge_indices_row, dim=0),
            torch.index_select(input=node_feats_flat, index=edge_indices_col, dim=0)
        ], dim=-1) # Index select returns a tensor with node_feats_flat being indexed at the desired positi

        # Calculate attention MLP output (independent for each head)
        attn_logits = torch.einsum('bhc,hc->bh', a_input, self.a)
        # torch.einsum: Einstein Summation Convention
        attn_logits = self.leakyrelu(attn_logits)

        # Map list of attention values back into a matrix
        attn_matrix = attn_logits.new_zeros(adj_matrix.shape+(self.num_heads,)).fill_(-9e15)
        attn_matrix[adj_matrix[...,None].repeat(1,1,1,self.num_heads) == 1] = attn_logits.reshape(-1)

        # Weighted average of attention
        attn_probs = F.softmax(attn_matrix, dim=2)
        if print_attn_probs:
            print("Attention probsWn", attn_probs.permute(0, 3, 1, 2))
        node_feats = torch.einsum('bijh,bjhc->bihc', attn_probs, node_feats) # h_i'

        # If heads should be concatenated, we can do this by reshaping. Otherwise, take mean
        if self.concat_heads:
```

```
            node_feats = node_feats.reshape(batch_size, num_nodes, -1)
        else:
            node_feats = node_feats.mean(dim=2)

        return node_feats
```

In [7]:
```
layer = GATLayer(2, 2, num_heads=2)
layer.projection.weight.data = torch.Tensor([[1., 0.], [0., 1.]])
layer.projection.bias.data = torch.Tensor([0., 0.])
layer.a.data = torch.Tensor([[-0.2, 0.3], [0.1, -0.1]])

with torch.no_grad():
    out_feats = layer(node_feats, adj_matrix, print_attn_probs=True)

print("Adjacency matrix", adj_matrix)
print("Input features", node_feats)
print("Output features", out_feats)
```

```
Attention probs
 tensor([[[[0.3543, 0.6457, 0.0000, 0.0000],
           [0.1096, 0.1450, 0.2642, 0.4813],
           [0.0000, 0.1858, 0.2885, 0.5257],
           [0.0000, 0.2391, 0.2696, 0.4913]],

          [[0.5100, 0.4900, 0.0000, 0.0000],
           [0.2975, 0.2436, 0.2340, 0.2249],
           [0.0000, 0.3838, 0.3142, 0.3019],
           [0.0000, 0.4018, 0.3289, 0.2693]]]])
Adjacency matrix tensor([[[1., 1., 0., 0.],
         [1., 1., 1., 1.],
         [0., 1., 1., 1.],
         [0., 1., 1., 1.]]])
Input features tensor([[[0., 1.],
         [2., 3.],
         [4., 5.],
         [6., 7.]]])
Output features tensor([[[1.2913, 1.9800],
         [4.2344, 3.7725],
         [4.6798, 4.8362],
         [4.5043, 4.7351]]])
```

- attention message passing 결과 node3 과 node4 는 서로 다른 output feature를 가지게 된다.
- 이는 node-specific information 의 손실을 방지한 것이라고 할 수 있다.

4. PyTorh_Geometric

- PyTorch_Geometric package 설치 - documentation 참조

In [48]:
```
!pip install torch-scatter torch-sparse torch-cluster torch-spline-conv torch-geometric -f https://data.py
```

```
Looking in links: https://data.pyg.org/whl/torch-1.10.0+cu102.html (https://data.pyg.org/whl/torch-1.10.0+cu102.html)
Requirement already satisfied: torch-scatter in c:Wanaconda3WlibWsite-packages (2.0.9)
Requirement already satisfied: torch-sparse in c:Wanaconda3WlibWsite-packages (0.6.12)
Requirement already satisfied: torch-cluster in c:Wanaconda3WlibWsite-packages (1.5.9)
Requirement already satisfied: torch-spline-conv in c:Wanaconda3WlibWsite-packages (1.2.1)
Requirement already satisfied: torch-geometric in c:Wanaconda3WlibWsite-packages (2.0.3)
Requirement already satisfied: scipy in c:Wanaconda3WlibWsite-packages (from torch-sparse) (1.6.2)
Requirement already satisfied: numpy in c:Wanaconda3WlibWsite-packages (from torch-geometric) (1.20.1)
Requirement already satisfied: scikit-learn in c:Wanaconda3WlibWsite-packages (from torch-geometric)
 (0.24.1)
Requirement already satisfied: rdflib in c:Wanaconda3WlibWsite-packages (from torch-geometric) (6.1.1)
Requirement already satisfied: yacs in c:Wanaconda3WlibWsite-packages (from torch-geometric) (0.1.8)
Requirement already satisfied: PyYAML in c:Wanaconda3WlibWsite-packages (from torch-geometric) (5.4.1)
Requirement already satisfied: networkx in c:Wanaconda3WlibWsite-packages (from torch-geometric) (2.5)
Requirement already satisfied: requests in c:Wanaconda3WlibWsite-packages (from torch-geometric) (2.25.
1)
Requirement already satisfied: pyparsing in c:Wanaconda3WlibWsite-packages (from torch-geometric) (2.4.
7)
```

In [47]:
```
!pip install --user --upgrade pip
```

```
Requirement already satisfied: pip in c:Wanaconda3WlibWsite-packages (22.0.3)
```

In [9]:
```
import torch_geometric
import torch_geometric.nn as geom_nn
import torch_geometric.data as geom_data
```

PyTorch geometric 에서의 MessagePassing Class

- PyTorch geometric 은 자동으로 message propagation 수행하여 message passing graph neural networks 생성을 지원하는 MessagePassing base class 를 제공한다

  : torch_geometric.nn.MessagePassing

Generalizing the convolution operator to irregular domains is typically expressed as a *neighborhood aggregation* or *message passing* scheme. With $\mathbf{x}_i^{(k-1)} \in \mathbb{R}^F$ denoting node features of node i in layer $(k-1)$ and $\mathbf{e}_{j,i} \in \mathbb{R}^D$ denoting (optional) edge features from node j to node i, message passing graph neural networks can be described as

$$\mathbf{x}_i^{(k)} = \gamma^{(k)} \left(\mathbf{x}_i^{(k-1)}, \square_{j \in \mathcal{N}(i)} \, \phi^{(k)} \left(\mathbf{x}_i^{(k-1)}, \mathbf{x}_j^{(k-1)}, \mathbf{e}_{j,i} \right) \right),$$

where $\square$ denotes a differentiable, permutation invariant function, *e.g.*, sum, mean or max, and γ and ϕ denote differentiable functions such as MLPs (Multi Layer Perceptrons).

1. 위 사진은 Message Passing graph neural networks 를 일반화아여 도식화한 것이다.
2. X_i(k-1) 은 (k-1) layer 에서 node i 의 node features 를 나타낸다.
3. e_j,i 는 node j 에서 i 로 연결되는 edge 의 features 를 나타낸다.
4. ㅁ 는 sum, mean, max 등과 같이 미분가능한 permutation invariant function 을 나타낸다.
5. gamma 와 phi 는 MLP 과 같이 미분가능한 함수를 나타낸다.

- PyTorch geometric 에서 모든 gnn layer class 는 nn.MessagePassing class 를 상속한다.
- 따라서 모든 gnn layer class 는 forward 연산에 feature matrix 와 함께 edge_index 를 input 으로 받는다.

geom_nn.GCNConv

The graph convolutional operator from the "Semi-supervised Classification with Graph Convolutional Networks" paper

$$\mathbf{X}' = \hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{X}\mathbf{\Theta},$$

where $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ denotes the adjacency matrix with inserted self-loops and $\hat{D}_{ii} = \sum_{j=0} \hat{A}_{ij}$ its diagonal degree matrix. The adjacency matrix can include other values than `1` representing edge weights via the optional `edge_weight` tensor.

Its node-wise formulation is given by:

$$\mathbf{x}_i' = \mathbf{\Theta}^\top \sum_{j \in \mathcal{N}(v) \cup \{i\}} \frac{e_{j,i}}{\sqrt{\hat{d}_j \hat{d}_i}} \mathbf{x}_j$$

geom_nn.GATConv

The graph attentional operator from the "Graph Attention Networks" paper

$$\mathbf{x}_i' = \alpha_{i,i}\mathbf{\Theta}\mathbf{x}_i + \sum_{j \in \mathcal{N}(i)} \alpha_{i,j}\mathbf{\Theta}\mathbf{x}_j,$$

where the attention coefficients $\alpha_{i,j}$ are computed as

$$\alpha_{i,j} = \frac{\exp\left(\mathrm{LeakyReLU}\left(\mathbf{a}^\top [\mathbf{\Theta}\mathbf{x}_i \,\|\, \mathbf{\Theta}\mathbf{x}_j]\right)\right)}{\sum_{k \in \mathcal{N}(i) \cup \{i\}} \exp\left(\mathrm{LeakyReLU}\left(\mathbf{a}^\top [\mathbf{\Theta}\mathbf{x}_i \,\|\, \mathbf{\Theta}\mathbf{x}_k]\right)\right)}.$$

If the graph has multi-dimensional edge features $\mathbf{e}_{i,j}$, the attention coefficients $\alpha_{i,j}$ are computed as

$$\alpha_{i,j} = \frac{\exp\left(\mathrm{LeakyReLU}\left(\mathbf{a}^\top [\mathbf{\Theta}\mathbf{x}_i \,\|\, \mathbf{\Theta}\mathbf{x}_j \,\|\, \mathbf{\Theta}_e\mathbf{e}_{i,j}]\right)\right)}{\sum_{k \in \mathcal{N}(i) \cup \{i\}} \exp\left(\mathrm{LeakyReLU}\left(\mathbf{a}^\top [\mathbf{\Theta}\mathbf{x}_i \,\|\, \mathbf{\Theta}\mathbf{x}_k \,\|\, \mathbf{\Theta}_e\mathbf{e}_{i,k}]\right)\right)}.$$

geom_nn.GraphConv

The graph neural network operator from the "Weisfeiler and Leman Go Neural: Higher-order Graph Neural Networks" paper

$$\mathbf{x}_i' = \mathbf{\Theta}_1\mathbf{x}_i + \mathbf{\Theta}_2 \sum_{j \in \mathcal{N}(i)} e_{j,i} \cdot \mathbf{x}_j$$

where $e_{j,i}$ denotes the edge weight from source node `j` to target node `i` (default: `1`)

1. GraphConv 은 GCN 에서 self-connections 의 weight matrix 를 차별화한 network 이다.
2. 위 공식에서 neighbor's messages 는 평균을 내지 않고 더하기를 한다
3. e_j,i 는 edge weight

Graph-structured data 의 Task 종류

1. 그래프 구조 데이터에 관한 Task는 3가지로 나눌 수 있다.

   1) node-level

   2) edge-level

   3) graph-level

2. 이러한 분류는 어떤 level 에서 classification/regression 을 수행할 것인지에 따른 구분이다.

- 여러개의 graph layer 를 사용하기 위해 dictionary 에 해당 layer 의 string 을 저정한다

In [10]:
```
gnn_layer_by_name = {
    "GCN": geom_nn.GCNConv,
    "GAT": geom_nn.GATConv,
    "GraphConv": geom_nn.GraphConv
}
```

5. Node-level tasks : Semi-supervised node classification

1. node-level tasks 는 node 를 classify 하는 목적을 가진다
2. 보통 1000개 이상의 nodes 로 구성된 하나의 큰 그래프가 주어지고,

   일부 특정 nodes 만 labeled 되있다
3. 훈련 과정에서 labeled 된 샘플을 학습하여 unlabeled nodes 를 추론할 것이다

Cora dataset

1. 논문의 citation network
2. 2708개의 과학 출판물(nodes)이 있고, 출판물 사이의 인용 관계(edges)를 포함한다
3. task 는 각 출판물(nodes)을 7개 classes 로 분류하는 것이다
4. 각 출판물은 단어 주머니(bag-of-words) vector 를 feature 로 가진다. vector의 dimension 은 1433
5. vector 원소의 1값은 해당 단어가 출판물에 있다는 것을 의미한다.

- load dataset

In [11]:
```
cora_dataset = torch_geometric.datasets.Planetoid(root=DATASET_PATH, name="Cora")
```

In [12]:
```
g = cora_dataset[0]
```

In [13]:
```
g
```

Out[13]:
```
Data(x=[2708, 1433], edge_index=[2, 10556], y=[2708], train_mask=[2708], val_mask=[2708], test_mask=[270
8])
```

PyTorch geometric 은 graph data 를 어떤 방식으로 나타내는가 -

1. graph는 'Data' 객체로 표현된다
2. edge_index - edges 의 list 를 포함하는 tensor, mirrored version edge 포함한다
3. train_mask, val_mask, test_mask - boolean masks로 training, validation, testing에 어떤 노드를 사용

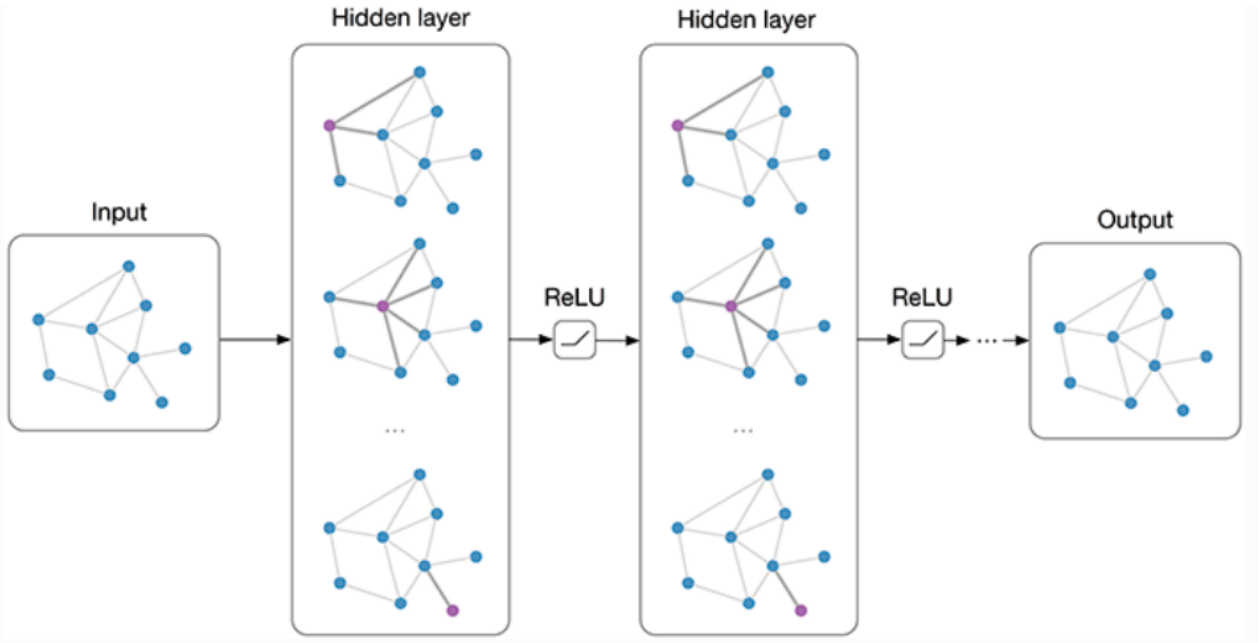
해야 하는지 나타내는 것이다 -> boolean masks sum() 결과값은 true 값의 개수를 의미한다

4. x tensor - feature tensor of 2708 publications

5. y tensor - the labels for all nodes

```
In [14]: g.x
Out[14]: tensor([[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.]])

In [15]: g.y
Out[15]: tensor([3, 4, 4,  ..., 3, 3, 3])

In [16]: g.train_mask.sum()
Out[16]: tensor(140)

In [17]: g.val_mask.sum()
Out[17]: tensor(500)

In [18]: g.test_mask.sum()
Out[18]: tensor(1000)
```

specific implementation



- GNN layer 는 여러 개 layers 로 구성할 수 있다.
- GNN layer 는 neighbors nodes 사이에 feature exchange 를 달성하는 것을 목적으로 한다.

In [19]:
```
class GNNModel(nn.Module):

    def __init__(self, c_in, c_hidden, c_out, num_layers=2, layer_name="GCN", dp_rate=0.1, **kwargs):
        """
        Inputs:
            c_in - Dimension of input features
            c_hidden - Dimension of hidden features
            c_out - Dimension of the output features. Usually number of classes in classification
            num_layers - Number of "hidden" graph layers
            layer_name - String of the graph layer to use
            dp_rate - Dropout rate to apply throughout the network
            kwargs - Additional arguments for the graph layer (e.g. number of heads for GAT)
        """
        super().__init__()
        gnn_layer = gnn_layer_by_name[layer_name]

        layers = []
        in_channels, out_channels = c_in, c_hidden
        for l_idx in range(num_layers-1):  # range(0) : (0,0) - len : 0
            layers += [
                gnn_layer(in_channels=in_channels,
                          out_channels=out_channels,
                          **kwargs),
                nn.ReLU(inplace=True),
                nn.Dropout(dp_rate)
            ]
            in_channels = c_hidden
        layers += [gnn_layer(in_channels=in_channels,
                             out_channels=c_out,
                             **kwargs)]
        self.layers = nn.ModuleList(layers)  # ModuleList : holds submodules in a list

    def forward(self, x, edge_index):
        """
        Inputs:
            x - Input features per node
            edge_index - List of vertex index pairs representing the edges in the graph (PyTorch geometric
        """
        for l in self.layers:
            # For graph layers, we need to add the "edge_index" tensor as additional input
            # All PyTorch Geometric graph layer inherit the class "MessagePassing", hence
            # we can simply check the class type.
            if isinstance(l, geom_nn.MessagePassing):  # isinstance - l 의 type 확인 함수
                x = l(x, edge_index)
            else:
                x = l(x)
        return x
```

1. GNN 에서의 dropout - message function의 linear layer 에 적용되고, message computation 이 시작될 때 같이 진행 된다

1) GCN - linear layer

2) GAT - Dropout probability of the normalized attention coefficients which

exposes each node to a stochastically sampled neighborhood during training.

3) GraphConv - linear layer 가 2개

GNN layer model 과 MLP model 성능 차이 비교 실험

- node-level tasks에서 좋은 실험은 각 node에 독립적으로 적용되는 MLP 기준을 만드는 것이다
- 이를 통해 graph information이 model prediction 성능을 향상시켰는지 아닌지를 검증할 수 있다

- feature가 node의 class를 명확하게 표현하데 충분한 경우를 생각해 볼 수도 있다
- 이를 확인하기 위해 MLP model을 구현해본다

In [20]:
```
class MLPModel(nn.Module):

    def __init__(self, c_in, c_hidden, c_out, num_layers=2, dp_rate=0.1):
        """
        Inputs:
            c_in - Dimension of input features
            c_hidden - Dimension of hidden features
            c_out - Dimension of the output features. Usually number of classes in classification
            num_layers - Number of hidden layers
            dp_rate - Dropout rate to apply throughout the network
        """
        super().__init__()
        layers = []
        in_channels, out_channels = c_in, c_hidden
        for l_idx in range(num_layers-1):
            layers += [
                nn.Linear(in_channels, out_channels),
                nn.ReLU(inplace=True),
                nn.Dropout(dp_rate)
            ]
            in_channels = c_hidden
        layers += [nn.Linear(in_channels, c_out)]
        self.layers = nn.Sequential(*layers)

    def forward(self, x, *args, **kwargs):
        """
        Inputs:
            x - Input features per node
        """
        return self.layers(x)
```

- PyTorch Lightning module로 위 두 모델을 합쳐서 training, validation, testing을 할 수 있다

In [21]:
```
class NodeLevelGNN(pl.LightningModule):

    def __init__(self, model_name, **model_kwargs):
        super().__init__()
        # Saving hyperparameters
        self.save_hyperparameters()

        if model_name == "MLP":
            self.model = MLPModel(**model_kwargs)
        else:
            self.model = GNNModel(**model_kwargs)
        self.loss_module = nn.CrossEntropyLoss()

    def forward(self, data, mode="train"):
        x, edge_index = data.x, data.edge_index
        x = self.model(x, edge_index)

        # Only calculate the loss on the nodes corresponding to the mask
        if mode == "train":
            mask = data.train_mask
        elif mode == "val":
            mask = data.val_mask
        elif mode == "test":
            mask = data.test_mask
        else:
            assert False, f"Unknown forward mode: {mode}"

        loss = self.loss_module(x[mask], data.y[mask])
        acc = (x[mask].argmax(dim=-1) == data.y[mask]).sum().float() / mask.sum()
        return loss, acc

    def configure_optimizers(self):
        # We use SGD here, but Adam works as well
        optimizer = optim.SGD(self.parameters(), lr=0.1, momentum=0.9, weight_decay=2e-3)
        # weight_decay (float, optional) - weight decay (L2 penalty) (default: 0) - preventing overfitting
        return optimizer

    def training_step(self, batch, batch_idx):
        loss, acc = self.forward(batch, mode="train")
        self.log('train_loss', loss)
        self.log('train_acc', acc)
        return loss

    def validation_step(self, batch, batch_idx):
        _, acc = self.forward(batch, mode="val")
        self.log('val_acc', acc)

    def test_step(self, batch, batch_idx):
        _, acc = self.forward(batch, mode="test")
        self.log('test_acc', acc)
```

- 추가적으로 Lightning module로 training function을 아래와 같이 정의할 수 있다
- data로 사용하는 graph는 1개이기 때문에 bathc size는 1로 사용한다
- train, validation, test set은 같은 data loader를 공유한다
- argument인 'progress_bar_refresh_rate'은 epoch에 따른 진행과정을 보여주는 argument로써 0으로 설정했는데, 이는 batch가 1개인 것에 따라 한 번의 epoch이 이루어지기 때문이다.

In [22]:
```
def train_node_classifier(model_name, dataset, **model_kwargs):
    pl.seed_everything(42) # sets seed for pseudo-random number generators
    node_data_loader = geom_data.DataLoader(dataset, batch_size=1)

    # Create a PyTorch Lightning trainer with the generation callback
    root_dir = os.path.join(CHECKPOINT_PATH, "NodeLevel" + model_name)
    os.makedirs(root_dir, exist_ok=True)
    trainer = pl.Trainer(default_root_dir=root_dir,
                         callbacks=[ModelCheckpoint(save_weights_only=True, mode="max", monitor="val_acc")
                         gpus=1 if str(device).startswith("cuda") else 0,
                         max_epochs=200,
                         progress_bar_refresh_rate=0) # 0 because epoch size is 1
    trainer.logger._default_hp_metric = None # Optional logging argument that we don't need

    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, f"NodeLevel{model_name}.ckpt")
    if os.path.isfile(pretrained_filename):
        print("Found pretrained model, loading...")
        model = NodeLevelGNN.load_from_checkpoint(pretrained_filename)
    else:
        pl.seed_everything()
        model = NodeLevelGNN(model_name=model_name, c_in=dataset.num_node_features, c_out=dataset.num_class
        trainer.fit(model, node_data_loader, node_data_loader)
        model = NodeLevelGNN.load_from_checkpoint(trainer.checkpoint_callback.best_model_path)

    # Test best model on the test set
    test_result = trainer.test(model, test_dataloaders=node_data_loader, verbose=False)
    batch = next(iter(node_data_loader))
    batch = batch.to(model.device)
    _, train_acc = model.forward(batch, mode="train")
    _, val_acc = model.forward(batch, mode="val")
    result = {"train": train_acc,
              "val": val_acc,
              "test": test_result[0]['test_acc']}
    return model, result
```

- 간단한 print_results 함수를 통해 결과 확인

In [23]:
```
# Small function for printing the test scores
def print_results(result_dict):
    if "train" in result_dict:
        print(f"Train accuracy: {(100.0*result_dict['train']):4.2f}%")
    if "val" in result_dict:
        print(f"Val accuracy:   {(100.0*result_dict['val']):4.2f}%")
    print(f"Test accuracy:  {(100.0*result_dict['test']):4.2f}%")
```

MLP model 성능 확인

In [24]:
```
node_mlp_model, node_mlp_result = train_node_classifier(model_name="MLP",
                                                        dataset=cora_dataset,
                                                        c_hidden=16,
                                                        num_layers=2,
                                                        dp_rate=0.1)

print_results(node_mlp_result)
```

```
Global seed set to 42
C:WAnaconda3WlibWsite-packagesWtorch_geometricWdeprecation.py:13: UserWarning: 'data.DataLoader' is dep
recated, use 'loader.DataLoader' instead
  warnings.warn(out)
C:WAnaconda3WlibWsite-packagesWpytorch_lightningWtrainerWconnectorsWcallback_connector.py:90: Lightning
DeprecationWarning: Setting `Trainer(progress_bar_refresh_rate=0)` is deprecated in v1.5 and will be re
moved in v1.7. Please pass `pytorch_lightning.callbacks.progress.TQDMProgressBar` with `refresh_rate` d
irectly to the Trainer's `callbacks` argument instead. Or, to disable the progress bar pass `enable_pro
gress_bar = False` to the Trainer.
  rank_zero_deprecation(
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
C:WAnaconda3WlibWsite-packagesWpytorch_lightningWtrainerWtrainer.py:902: LightningDeprecationWarning: `
trainer.test(test_dataloaders)` is deprecated in v1.4 and will be removed in v1.6. Use `trainer.test(da
taloaders)` instead.
  rank_zero_deprecation(
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

Found pretrained model, loading...
```

- MLP model의 경우 imput features의 dimension이 높아 training accuracy는 높지만 성능은 좋지 않았다 - overfitting

GCN, GraphConv, GAT model 성능 확인

- pre-trained GCN

In [25]:
```
node_gnn_model, node_gnn_result = train_node_classifier(model_name="GNN",
                                                        layer_name="GCN",
                                                        dataset=cora_dataset,
                                                        c_hidden=16,
                                                        num_layers=2,
                                                        dp_rate=0.1)
print_results(node_gnn_result)
```

```
Global seed set to 42
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

Found pretrained model, loading...
Train accuracy: 100.00%
Val accuracy:   78.60%
Test accuracy:  82.40%
```

- non-pre-trained GCN

In [26]:
```
node_gnn_model, node_gnn_result = train_node_classifier(model_name="non",
                                                        layer_name="GCN",
                                                        dataset=cora_dataset,
                                                        c_hidden=16,
                                                        num_layers=4,
                                                        dp_rate=0.1)
print_results(node_gnn_result)
```

```
Global seed set to 42
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
Global seed set to 42
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

  | Name        | Type             | Params
-----------------------------------------------
0 | model       | GNNModel         | 23.6 K
1 | loss_module | CrossEntropyLoss | 0
-----------------------------------------------
23.6 K    Trainable params
0         Non-trainable params
23.6 K    Total params
0.094     Total estimated model params size (MB)
C:WAnaconda3WlibWsite-packagesWpytorch_lightningWtrainerWdata_loading.py:116: UserWarning: The dataloader, val_dataloader 0, does not have many workers which may be a bottleneck. Consider increasing the value of the `num_workers` argument` (try 8 which is the number of cpus on this machine) in the `DataLoader` init to improve performance.
  rank_zero_warn(
Global seed set to 42
C:WAnaconda3WlibWsite-packagesWpytorch_lightningWtrainerWdata_loading.py:116: UserWarning: The dataloader, train_dataloader, does not have many workers which may be a bottleneck. Consider increasing the value of the `num_workers` argument` (try 8 which is the number of cpus on this machine) in the `DataLoader` init to improve performance.
  rank_zero_warn(
C:WAnaconda3WlibWsite-packagesWpytorch_lightningWtrainerWdata_loading.py:412: UserWarning: The number of training samples (1) is smaller than the logging interval Trainer(log_every_n_steps=50). Set a lower value for log_every_n_steps if you want to see logs for the training epoch.
  rank_zero_warn(
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

Train accuracy: 97.86%
Val accuracy:   75.80%
Test accuracy:  78.50%
```

- non-pre-trained GraphConv

In [27]:
```
node_gnn_model, node_gnn_result = train_node_classifier(model_name="non",
                                                         layer_name="GraphConv",
                                                         dataset=cora_dataset,
                                                         c_hidden=16,
                                                         num_layers=2,
                                                         dp_rate=0.1)
print_results(node_gnn_result)
```

```
Global seed set to 42
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
Global seed set to 42
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

  | Name        | Type             | Params
-----------------------------------------------
0 | model       | GNNModel         | 46.1 K
1 | loss_module | CrossEntropyLoss | 0
-----------------------------------------------
46.1 K    Trainable params
0         Non-trainable params
46.1 K    Total params
0.184     Total estimated model params size (MB)
Global seed set to 42
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

Train accuracy: 95.00%
Val accuracy:   56.40%
Test accuracy:  63.60%
```

- non-pre-trained GAT

In [28]:
```
node_gnn_model, node_gnn_result = train_node_classifier(model_name="non",
                                                         layer_name="GAT",
                                                         dataset=cora_dataset,
                                                         c_hidden=16,
                                                         num_layers=2,
                                                         dp_rate=0.1)
print_results(node_gnn_result)
```

```
Global seed set to 42
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
Global seed set to 42
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

  | Name        | Type             | Params
-----------------------------------------------
0 | model       | GNNModel         | 23.1 K
1 | loss_module | CrossEntropyLoss | 0
-----------------------------------------------
23.1 K    Trainable params
0         Non-trainable params
23.1 K    Total params
0.092     Total estimated model params size (MB)
Global seed set to 42
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

Train accuracy: 100.00%
Val accuracy:   75.00%
Test accuracy:  78.10%
```

- GNN model(GraphConv 제외)이 MLP model보다 높은 성능을 보였다.
- 이는 graph information을 사용하는 것이 정말로 prediction 성능을 향상시키는 것을 확인할 수 있게 한다

- 위 모델의 hyperparameters은 상대적으로 작은 network를 형성하도록 선택되었다

- 이는 1433개의 input dimension을 가지는 layer의 경우 큰 그래프에 대해 높은 연산 비용을 가질 수 있기 때문이다.
- 이러한 비용 때문에 GNN은 작은 hidden size를 설정하거나 특별한 batching 전략을 사용한다
- batching 전략에는 큰 규모의 original graph에서 connected subgraph를 추출하는 방법이 있다

6. Graph-level tasks : Graph classification

dataset : MUTAG

1. 188개 그래프
2. 그래프 당 평균 18개 nodes, 20개 edges
3. node는 7개의 labels(atom types)
4. the binary graph labels - 'their mutagenic effect on a specific gram negative bacterium'
5. 사용할 dataset은 TUDatasets의 일부로 torch_geometric.datasets.TUDataset으로 접근할 수 있다.

In [29]:
```
tu_dataset = torch_geometric.datasets.TUDataset(root=DATASET_PATH, name="MUTAG")
```

In [30]:
```
print("Data object:", tu_dataset.data)
print("Length:", len(tu_dataset))
print(f"Average label: {tu_dataset.data.y.float().mean().item():4.2f}")
```

```
Data object: Data(x=[3371, 7], edge_index=[2, 7442], edge_attr=[7442, 4], y=[188])
Length: 188
Average label: 0.66
```

- dataset 형태

1. dataset은 각 그래프의 nodes, edges, labels가 하나의 텐서로 concatenated 되있다.
2. dataset은 텐서의 어느 부분을 split 해야 하는지 indices를 내포하고 있다.
3. dataset의 length는 그래프의 개수이다.
4. average label은 label 1인 그래프의 비율을 의미한다.

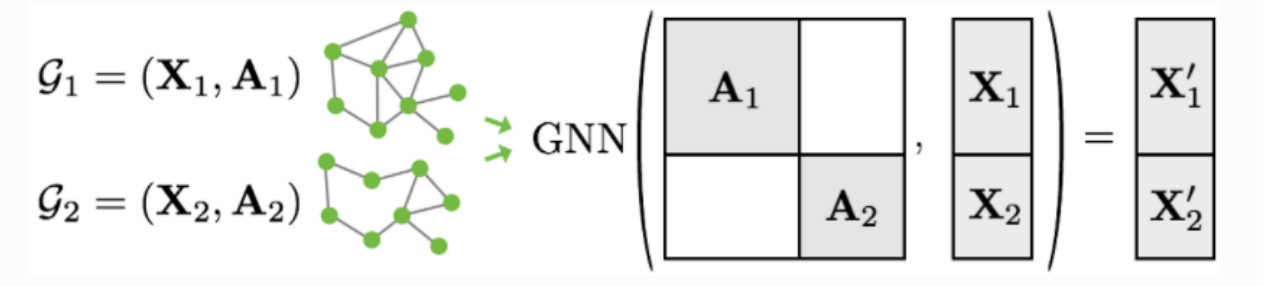
   checking the class balance is always a good thing to do...

- Split dataset into a training and test part

1. 이번 task에서는 dataset의 규모가 작기 때문에 validation set을 사용하지 않는다

In [31]:
```
torch.manual_seed(42)
tu_dataset.shuffle()
train_dataset = tu_dataset[:150]
test_dataset = tu_dataset[150:]
```

Batching strategy in graph-level task



1. graph를 batching 할 때 batch 안의 각각의 graph는 서로 다른 nodes와 edges 개수를 가질 수 있다.
2. Torch geometric은 batch 안에 있는 N 개의 graphs를 각각의 nodes와 edges를 concatenate하여 하나의 큰 그래프로 접근한다.

3. 서로 다른 graph 사이에는 edge가 없으므로 large graph를 GNN에 적용하는 것은 각각의 그래프를 따로 GNN에 적용하는 것과 같은 output을 나타낸다.
4. 서로 다른 그래프 사이의 adjacency matrix는 0 값을 가지고, 개별 그래프에 상응하는 adjacency matrix 만 각각에 해당하는 값을 가진다.
5. 이러한 batching srtategy는 torch geometric에 이미 적용되있기 때문에 해당하는 dataloader를 사용하여 활용한다

In [32]:
```
graph_train_loader = geom_data.DataLoader(train_dataset, batch_size=64, shuffle=True)
graph_val_loader = geom_data.DataLoader(test_dataset, batch_size=64) # Additional loader if you want to cha
graph_test_loader = geom_data.DataLoader(test_dataset, batch_size=64)
```

- 이번 task에서 validation set을 사용하지 않지만 일반화된 코드를 사용하므로

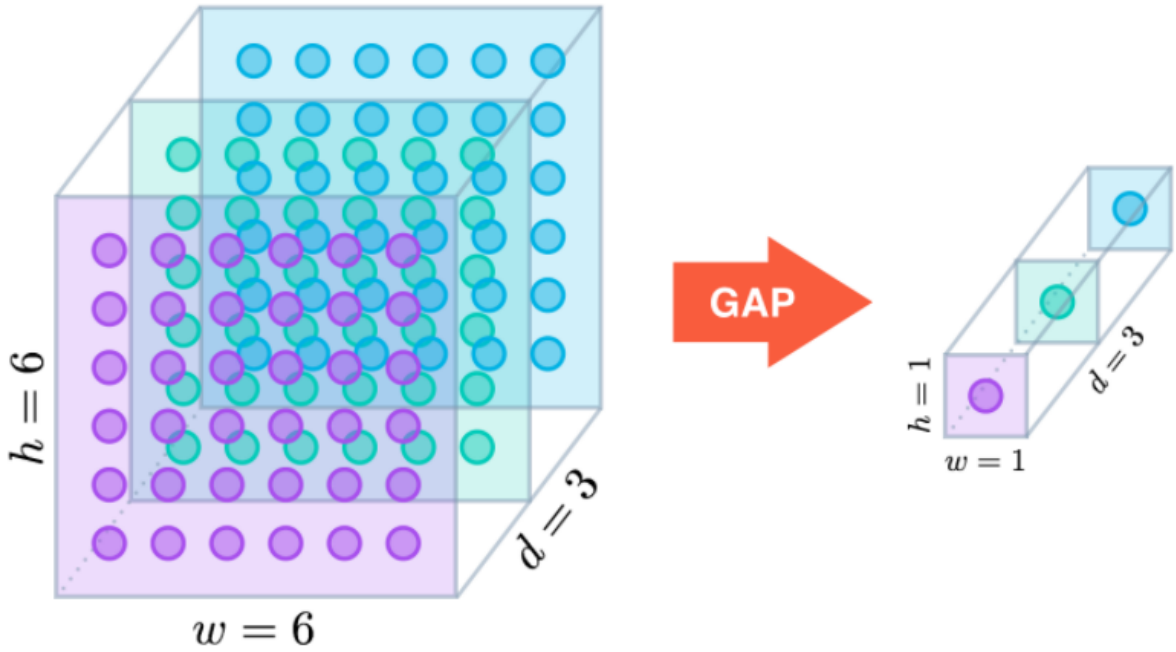
  validation set과 test set이 하나의 test set인 것처럼 인식하여 학습할 것이다

In [33]:
```
batch = next(iter(graph_test_loader))
print("Batch:", batch)
print("Labels:", batch.y[:10])
print("Batch indices:", batch.batch[:40])
```

```
Batch: DataBatch(edge_index=[2, 1512], x=[687, 7], edge_attr=[1512, 4], y=[38], batch=[687], ptr=[39])
Labels: tensor([1, 1, 1, 0, 0, 0, 1, 1, 1, 0])
Batch indices: tensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2])
```

1. 위 test dataset에는 38개의 그래프가 쌓여있다.
2. Batch indice의 앞의 12개 nodes는 0번째 graph에 속한 것이다
3. 다음 22개 nodes는 1번째 graph에 속한 것이다
4. 이 indices는 마지막 예측단계에서 중요하게 사용된다

graph-level task를 위한 GNN model 생성과 geom_nn.global_mean_pool 활용



1. 모든 그래프에 대한 prediction을 진행할 때, GNN model을 적용한 후 모든 노드에 대하여

   pooling operation을 실행할 것이다. 이 경우 geom_nn.global_mean_pool 을 사용할 것이다

2. 이러한 이유로 global average pooling에 포함할 nodes의 indices를 알아야 한다

3. Model 은 이전 node-level task에서 사용한 GNNModel에 geom_nn.global_mean_pool 과 그래프 예측을 위한 single linear layer를 추가하여 모델을 생성할 것이다

In [34]:
```
class GraphGNNModel(nn.Module):

    def __init__(self, c_in, c_hidden, c_out, dp_rate_linear=0.5, **kwargs):
        """
        Inputs:
            c_in - Dimension of input features
            c_hidden - Dimension of hidden features
            c_out - Dimension of output features (usually number of classes)
            dp_rate_linear - Dropout rate before the linear layer (usually much higher than inside the GNN)
            kwargs - Additional arguments for the GNNModel object
        """
        super().__init__()
        self.GNN = GNNModel(c_in=c_in,
                            c_hidden=c_hidden,
                            c_out=c_hidden, # Not our prediction output yet!
                            **kwargs)
        self.head = nn.Sequential(
            nn.Dropout(dp_rate_linear),
            nn.Linear(c_hidden, c_out)
        )

    def forward(self, x, edge_index, batch_idx):
        """
        Inputs:
            x - Input features per node
            edge_index - List of vertex index pairs representing the edges in the graph (PyTorch geometric
            batch_idx - Index of batch element for each node
        """
        x = self.GNN(x, edge_index)
        x = geom_nn.global_mean_pool(x, batch_idx) # Average pooling
        x = self.head(x)
        return x
```

training을 위한 Pytorch Lighning module

위에서 사용했던 모델과 유사하고 binary classification task를 수행하기 때문에

Binary Cross Entropy Loss를 사용

In [35]:
```
class GraphLevelGNN(pl.LightningModule):

    def __init__(self, **model_kwargs):
        super().__init__()
        # Saving hyperparameters - 생성자 안의 argument를 Lighning의 checkpoint에 저장하고,
        # self.hparams. 를 통해 argument에 접근할 수 있다.
        self.save_hyperparameters()

        self.model = GraphGNNModel(**model_kwargs)
        self.loss_module = nn.BCEWithLogitsLoss() if self.hparams.c_out == 1 else nn.CrossEntropyLoss()

    def forward(self, data, mode="train"):
        x, edge_index, batch_idx = data.x, data.edge_index, data.batch
        x = self.model(x, edge_index, batch_idx)
        x = x.squeeze(dim=-1)

        if self.hparams.c_out == 1:
            preds = (x > 0).float()
            data.y = data.y.float()
        else:
            preds = x.argmax(dim=-1)
        loss = self.loss_module(x, data.y)
        acc = (preds == data.y).sum().float() / preds.shape[0]
        return loss, acc

    def configure_optimizers(self):
        optimizer = optim.AdamW(self.parameters(), lr=1e-2, weight_decay=0.0) # High lr because of small da
        return optimizer

    def training_step(self, batch, batch_idx):
        loss, acc = self.forward(batch, mode="train")
        self.log('train_loss', loss)
        self.log('train_acc', acc)
        return loss

    def validation_step(self, batch, batch_idx):
        _, acc = self.forward(batch, mode="val")
        self.log('val_acc', acc)

    def test_step(self, batch, batch_idx):
        _, acc = self.forward(batch, mode="test")
        self.log('test_acc', acc)
```

- nn.BCEWithLogitsLoss()

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log \sigma(x_n) + (1 - y_n) \cdot \log(1 - \sigma(x_n)) \right],$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{‘mean’;} \\ \text{sum}(L), & \text{if reduction} = \text{‘sum’.} \end{cases}$$

1. Sigmoid layer와 BCELoss를 하나의 class로 합친 것
2. plain Sigmoid followed by a BCELoss을 사용하는 것보다 계수적(numerically)으로 안정적이라고 한다

In [36]:
```
def train_graph_classifier(model_name, **model_kwargs):
    pl.seed_everything(42)

    # Create a PyTorch Lightning trainer with the generation callback
    root_dir = os.path.join(CHECKPOINT_PATH, "GraphLevel" + model_name)
    os.makedirs(root_dir, exist_ok=True)
    trainer = pl.Trainer(default_root_dir=root_dir,
                         callbacks=[ModelCheckpoint(save_weights_only=True, mode="max", monitor="val_acc")
                         gpus=1 if str(device).startswith("cuda") else 0,
                         max_epochs=500,
                         progress_bar_refresh_rate=0)
    trainer.logger._default_hp_metric = None # Optional logging argument that we don't need

    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, f"GraphLevel{model_name}.ckpt")
    if os.path.isfile(pretrained_filename):
        print("Found pretrained model, loading...")
        model = GraphLevelGNN.load_from_checkpoint(pretrained_filename)
    else:
        pl.seed_everything(42)
        model = GraphLevelGNN(c_in=tu_dataset.num_node_features,
                              c_out=1 if tu_dataset.num_classes==2 else tu_dataset.num_classes,
                              **model_kwargs)
        trainer.fit(model, graph_train_loader, graph_val_loader)
        model = GraphLevelGNN.load_from_checkpoint(trainer.checkpoint_callback.best_model_path)
    # Test best model on validation and test set
    train_result = trainer.test(model, test_dataloaders=graph_train_loader, verbose=False)
    test_result = trainer.test(model, test_dataloaders=graph_test_loader, verbose=False)
    result = {"test": test_result[0]['test_acc'], "train": train_result[0]['test_acc']}
    return model, result
```

- pre-trained GraphConv model

In [37]:
```
model, result = train_graph_classifier(model_name="GraphConv",
                                       c_hidden=256,
                                       layer_name="GraphConv",
                                       num_layers=3,
                                       dp_rate_linear=0.5,
                                       dp_rate=0.0)
```

```
Global seed set to 42
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

Found pretrained model, loading...

C:WAnaconda3WlibWsite-packagesWpytorch_lightningWtrainerWdata_loading.py:633: UserWarning: Your `test_data
loader` has `shuffle=True`, it is strongly recommended that you turn this off for val/test/predict dataloa
ders.
  rank_zero_warn(
C:WAnaconda3WlibWsite-packagesWpytorch_lightningWutilitiesWdata.py:59: UserWarning: Trying to infer the `b
atch_size` from an ambiguous collection. The batch size we found is 10. To avoid any miscalculations, use
`self.log(..., batch_size=batch_size)`.
  warning_cache.warn(
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

In [38]:
```
print(f"Train performance: {100.0*result['train']:4.2f}%")
print(f"Test performance:  {100.0*result['test']:4.2f}%")
```

```
Train performance: 93.28%
Test performance:  92.11%
```

- Non-pre-trained GraphConv model

In [39]:
```
model, result = train_graph_classifier(model_name="non",
                                       c_hidden=256,
                                       layer_name="GraphConv",
                                       num_layers=3,
                                       dp_rate_linear=0.3,
                                       dp_rate=0.0)
```

```
Global seed set to 42
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
Global seed set to 42
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

  | Name        | Type              | Params
-------------------------------------------------
0 | model       | GraphGNNModel     | 266 K
1 | loss_module | BCEWithLogitsLoss | 0
-------------------------------------------------
266 K     Trainable params
0         Non-trainable params
266 K     Total params
1.067     Total estimated model params size (MB)
Global seed set to 42
C:WAnaconda3WlibWsite-packagesWpytorch_lightningWtrainerWdata_loading.py:412: UserWarning: The number of t
raining samples (3) is smaller than the logging interval Trainer(log_every_n_steps=50). Set a lower value
for log_every_n_steps if you want to see logs for the training epoch.
  rank_zero_warn(
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

In [40]:
```
print(f"Train performance: {100.0*result['train']:4.2f}%")
print(f"Test performance:  {100.0*result['test']:4.2f}%")
```

```
Train performance: 93.80%
Test performance:  89.47%
```

- Non-pre-trained GCN, GAT

In [41]:
```
model, result = train_graph_classifier(model_name="non",
                                       c_hidden=256,
                                       layer_name="GCN",
                                       num_layers=3,
                                       dp_rate_linear=0.2,
                                       dp_rate=0.0)
```

```
Global seed set to 42
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
Global seed set to 42
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

  | Name        | Type              | Params
-------------------------------------------------
0 | model       | GraphGNNModel     | 133 K
1 | loss_module | BCEWithLogitsLoss | 0
-------------------------------------------------
133 K     Trainable params
0         Non-trainable params
133 K     Total params
0.536     Total estimated model params size (MB)
Global seed set to 42
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

In [42]:
```
print(f"Train performance: {100.0*result['train']:4.2f}%")
print(f"Test performance:  {100.0*result['test']:4.2f}%")
```

```
Train performance: 81.49%
Test performance:  84.21%
```

In [43]:
```
model, result = train_graph_classifier(model_name="non",
                                       c_hidden=256,
                                       layer_name="GAT",
                                       num_layers=3,
                                       dp_rate_linear=0.2,
                                       dp_rate=0.0)
```

```
Global seed set to 42
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
Global seed set to 42
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

  | Name        | Type              | Params
-------------------------------------------------
0 | model       | GraphGNNModel     | 135 K
1 | loss_module | BCEWithLogitsLoss | 0
-------------------------------------------------
135 K     Trainable params
0         Non-trainable params
135 K     Total params
0.542     Total estimated model params size (MB)
Global seed set to 42
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

In [44]:
```
print(f"Train performance: {100.0*result['train']:4.2f}%")
print(f"Test performance:  {100.0*result['test']:4.2f}%")
```

```
Train performance: 74.76%
Test performance:  78.95%
```

Conclusion

1. validation set 이 없었기에 약간의 overfitted 된 결과일 것이다.
2. 그럼에도 불구하고 위 실험 결과는 GNN 이 graph 의 특성을 예측하는데

   강력한 효과를 내는 것을 확인할 수 있다.
3. Prediction 에 graph information 을 활용하면 높은 성능을 기대할 수 있다.

7. 다른 dataset(Amazon)에 적용

표1 : MLP 모델로 Layer 2개 구성하여 COMPUTER, PHOTO dataset 으로 node-level task 수행함.

Red: the best model, Violet: good models

* Amazon Computers and Amazon Photo Dataset: Amazon co-purchase graph 일부분. Photo 는 Amazon photo network data 라는 언급 뿐.
* Nodes: goods / edges: 자주 같이 산 물건 / node features: product reviews 의 bag-of-words / class labels: product category.
* Train: 200(160-photo) / Validation: 500 / Test: 1000

Global seed : 42, Max Epochs : 200			NODE CLASSIFICATION					
MLP			COMPUTER			PHOTO		
L	c_hidden	dp_rate	Param	Test Acc	Train Acc	Param	Test Acc	Train Acc
2	16	0.1	12,500	86.20	83.50	12,100	95.10	92.50
		0.2		85.70	78.50		95.30	91.88
		0.3		84.40	74.50		95.70	86.25
		0.4		83.60	68.00		93.80	71.88
	32	0.1	24,900	87.30	87.50	24,100	96.70	93.12
		0.2		85.80	86.00		96.60	93.12
		0.3		85.40	76.00		94.70	89.38
		0.4		83.70	78.50		95.80	85.62
	64	0.1	49,800	86.60	88.00	49,300	96.80	96.88
		0.2		86.60	84.00		96.30	97.50
		0.3		85.50	87.00		95.00	94.38
		0.4		84.40	81.00		94.80	94.38
	128	0.1	99,600	88.30	94.00	96,500	97.20	95.00
		0.2		87.50	94.50		97.10	93.75
		0.3		88.70	81.50		96.60	95.00
		0.4		87.10	87.00		95.60	92.50
	256	0.1	199,000	87.90	92.00	193,000	96.90	94.38
		0.2		87.20	90.50		96.50	93.75
		0.3		85.00	88.00		95.70	93.12
		0.4		84.00	87.50		96.40	91.25
	512	0.1	398,000	86.90	88.50	398,000	97.80	98.75
		0.2		87.70	91.00		96.40	96.25
		0.3		88.50	88.00		96.60	96.88
		0.4		86.20	87.00		96.50	93.75

표2: 연산 시간 문제로 이후 모든 Task 는 Dataset: Computer / Layer - 2, 4, 8, 16 / c_hidden - 16, 32, 64 / dp_rate - 0.1, 0.2, 0.3, 0.4 분류 기준으로 학습 결과를 종합함.

Global seed : 42, Max Epochs : 200			NODE CLASSIFICATION		
MLP			COMPUTER		
L	c_hidden	dp_rate	Param	Test Acc	Train Acc
4	16	0.1	13,000	75.80	70.00
		0.2		64.80	55.50
		0.3		56.40	40.00
		0.4		52.20	28.50
	32	0.1	27,000	78.40	73.50
		0.2		80.30	71.00
		0.3		76.80	55.00
		0.4		76.80	43.50
	64	0.1	58,100	80.50	79.50
		0.2		84.70	84.00
		0.3		80.60	69.50
		0.4		80.50	57.00
8	16	0.1	13,500	36.30	10.00
		0.2		41.40	34.50
		0.3		42.30	17.00
		0.4		29.30	18.50
	32	0.1	29,100	42.40	18.00
		0.2		43.00	17.00
		0.3		43.00	18.50
		0.4		40.20	20.00
	64	0.1	66,400	54.60	41.00
		0.2		45.90	17.50
		0.3		42.00	20.00
		0.4		41.40	18.50

Global seed : 42, Max Epochs : 200			NODE CLASSIFICATION		
MLP			COMPUTER		
L	c_hidden	dp_rate	Param	Test Acc	Train Acc
16	16	0.1	13,000	14.30	12.00
		0.2		39.40	11.50
		0.3		14.30	7.00
		0.4		14.30	11.50
	32	0.1	27,000	15.20	8.50
		0.2		14.80	5.00
		0.3		38.60	7.50
		0.4		38.60	8.50
	64	0.1	58,100	25.10	8.50
		0.2		26.10	8.00
		0.3		30.40	9.50
		0.4		26.50	11.00

표3: GCNConv

Global seed : 42, Max Epochs : 200 GCN			NODE CLASSIFICATION COMPUTER		
L	c_hidden	dp_rate	Param	Test Acc	Train Acc
2	16	0.1	12,500	88.60	91.00
		0.2		86.90	91.50
		0.3		88.10	88.00
		0.4		87.20	85.50
	32	0.1	24,900	87.30	90.50
		0.2		88.30	88.50
		0.3		87.00	88.50
		0.4		88.50	89.00
	64	0.1	49,800	87.70	89.50
		0.2		86.10	86.00
		0.3		87.60	85.00
		0.4		87.20	87.50
4	16	0.1	13,000	45.60	43.00
		0.2		51.40	49.50
		0.3		58.90	45.50
		0.4		77.50	77.50
	32	0.1	27,000	39.80	42.50
		0.2		58.50	65.50
		0.3		61.90	65.50
		0.4		**76.20**	79.00
	64	0.1	58,100	76.60	82.00
		0.2		81.50	81.00
		0.3		79.30	81.50
		0.4		70.60	72.00

Global seed : 42, Max Epochs : 200 GCN			NODE CLASSIFICATION COMPUTER		
L	c_hidden	dp_rate	Param	Test Acc	Train Acc
8	16	0.1	13,500	35.50	8.50
		0.2		38.60	9.50
		0.3		29.30	24.50
		0.4		30.40	12.00
	32	0.1	29,100	31.30	11.00
		0.2		43.90	31.00
		0.3		43.20	29.50
		0.4		58.40	39.50
	64	0.1	66,400	38.20	10.00
		0.2		42.30	46.00
		0.3		39.60	18.00
		0.4		38.00	19.50
16	16	0.1	14,100	17.40	15.00
		0.2		39.60	9.50
		0.3		25.80	11.50
		0.4		38.90	10.00
	32	0.1	31,200	34.00	17.50
		0.2		35.40	10.00
		0.3		29.00	20.00
		0.4		32.90	9.50
	64	0.1	74,800	36.50	10.00
		0.2		40.50	13.50
		0.3		40.00	9.50
		0.4		39.50	9.50

표4: GAT

Global seed : 42, Max Epochs : 200 GAT			NODE CLASSIFICATION COMPUTER		
L	c_hidden	dp_rate	Param	Test Acc	Train Acc
2	16	0.1	12,500	88.80	89.50
		0.2		86.00	89.00
		0.3		75.80	75.00
		0.4		87.30	81.00
	32	0.1	25,000	89.60	90.50
		0.2		89.00	92.50
		0.3		89.30	91.50
		0.4		88.50	92.00
	64	0.1	50,000	91.30	94.50
		0.2		91.90	94.00
		0.3		89.60	93.50
		0.4		90.90	95.50
4	16	0.1	13,100	81.80	88.50
		0.2		86.50	86.00
		0.3		82.30	85.00
		0.4		83.50	87.00
	32	0.1	27,200	85.80	90.00
		0.2		87.80	89.50
		0.3		85.50	88.50
		0.4		85.70	87.00
	64	0.1	58,500	81.90	85.00
		0.2		**87.70**	87.00
		0.3		84.20	87.50
		0.4		86.70	85.00

Global seed : 42, Max Epochs : 200 GAT			NODE CLASSIFICATION COMPUTER		
L	c_hidden	dp_rate	Param	Test Acc	Train Acc
8	16	0.1	13,700	49.30	24.00
		0.2		80.60	77.00
		0.3		76.30	77.50
		0.4		66.50	60.00
	32	0.1	29,500	42.90	33.00
		0.2		43.80	24.00
		0.3		77.90	79.50
		0.4		43.40	27.00
	64	0.1	67,100	55.70	47.50
		0.2		82.60	84.00
		0.3		59.00	30.00
		0.4		53.10	28.50
16	16	0.1	14,300	41.20	19.00
		0.2		35.60	21.00
		0.3		63.60	49.50
		0.4		58.10	53.00
	32	0.1	31,700	66.10	69.50
		0.2		65.60	69.00
		0.3		39.20	10.50
		0.4		19.80	19.50
	64	0.1	75,700	44.80	26.50
		0.2		46.00	36.00
		0.3		42.50	22.50
		0.4		55.30	27.50

표5: GraphConv

Global seed : 42, Max Epochs : 200 GraphConv			NODE CLASSIFICATION COMPUTER		
L	c_hidden	dp_rate	Param	Test Acc	Train Acc
2	16	0.1	24,900	17.10	10.50
		0.2		17.20	10.50
		0.3		36.80	10.00
		0.4		17.60	10.00
	32	0.1	49,800	36.10	09.50
		0.2		35.80	09.50
		0.3		36.20	10.00
		0.4		36.10	10.00
	64	0.1	99,500	40.50	10.00
		0.2		40.50	10.00
		0.3		39.20	09.50
		0.4		40.50	10.00
4	16	0.1	25,900	37.80	10.00
		0.2		37.80	10.00
		0.3		37.80	10.00
		0.4		37.80	10.00
	32	0.1	53,900	37.80	10.00
		0.2		37.80	10.00
		0.3		37.80	10.00
		0.4		36.90	10.00
	64	0.1	116,000	36.90	10.00
		0.2		36.90	10.00
		0.3		36.90	10.00
		0.4		36.90	10.00

Global seed : 42, Max Epochs : 200 GraphConv			NODE CLASSIFICATION COMPUTER		
L	c_hidden	dp_rate	Param	Test Acc	Train Acc
8	16	0.1	27,000	4.00	10.00
		0.2		4.00	10.00
		0.3		4.00	10.00
		0.4		4.00	10.00
	32	0.1	58,100	2.80	10.00
		0.2		2.80	10.00
		0.3		2.80	10.00
		0.4		2.80	10.00
	64	0.1	132,000	2.80	10.00
		0.2		2.80	10.00
		0.3		2.80	10.00
		0.4		2.80	10.00
16	16	0.1	28,100	2.50	10.00
		0.2		2.50	10.00
		0.3		2.50	10.00
		0.4		2.50	10.00
	32	0.1	62,200	3.70	10.00
		0.2		3.70	10.00
		0.3		3.70	10.00
		0.4		3.70	10.00
	64	0.1	149,000	2.40	10.00
		0.2		2.40	10.00
		0.3		2.40	10.00
		0.4		2.40	10.00

표6: GINConv

Global seed : 42, Max Epochs : 200 GINConv			NODE CLASSIFICATION COMPUTER		
L	c_hidden	dp_rate	Param	Test Acc	Train Acc
2	16	0.1	13,100	37.20	10.00
		0.2		37.20	10.00
		0.3		37.20	10.00
		0.4		37.20	10.00
	32	0.1	27,700	36.70	10.00
		0.2		36.70	10.00
		0.3		36.70	10.00
		0.4		36.70	10.00
	64	0.1	61,600	37.50	10.00
		0.2		37.50	10.00
		0.3		37.50	10.00
		0.4		37.50	10.00
4	16	0.1	14,200	3.20	10.00
		0.2		3.20	10.00
		0.3		3.20	10.00
		0.4		3.20	10.00
	32	0.1	32,000	37.80	10.00
		0.2		37.80	10.00
		0.3		37.80	10.00
		0.4		37.80	10.00
	64	0.1	78,300	3.30	10.00
		0.2		3.30	10.00
		0.3		3.30	10.00
		0.4		3.30	10.00

Global seed : 42, Max Epochs : 200 GINConv			NODE CLASSIFICATION COMPUTER		
L	c_hidden	dp_rate	Param	Test Acc	Train Acc
8	16	0.1	15,300	4.30	10.00
		0.2		4.30	10.00
		0.3		4.30	10.00
		0.4		4.30	10.00
	32	0.1	36,200	2.70	10.00
		0.2		2.70	10.00
		0.3		2.70	10.00
		0.4		2.70	10.00
	64	0.1	94,900	2.50	10.00
		0.2		2.50	10.00
		0.3		2.50	10.00
		0.4		2.50	10.00
16	16	0.1	16,400	3.00	10.00
		0.2		3.00	10.00
		0.3		3.00	10.00
		0.4		3.00	10.00
	32	0.1	40,400	3.00	10.00
		0.2		3.00	10.00
		0.3		3.00	10.00
		0.4		3.00	10.00
	64	0.1	111,000	3.80	10.00
		0.2		3.80	10.00
		0.3		3.80	10.00
		0.4		3.80	10.00

표7: Best model 비교

Global seed : 42 Max Epochs : 200		NODE CLASSIFICATION COMPUTER				
Model	*L*	Param	c_hidden	dp_rate	Test Acc	Train Acc
MLP	2	99,600	128	0.3	88.70	81.50
	4	58,100	64	0.2	84.70	84.00
	8	66,400	64	0.1	54.60	41.00
	16	13,000	16	0.2	39.40	11.50
GCN	2	12,500	16	0.1	88.60	91.00
	4	58,100	64	0.2	81.50	81.00
	8	29,100	32	0.4	58.40	39.50
	16	74,800	64	0.2	40.50	13.50
GAT	2	50,000	64	0.2	91.90	94.00
	4	27,200	32	0.2	87.80	89.50
	8	67,100	64	0.2	82.60	84.00
	16	31,700	32	0.1	66.10	69.50
GraphConv	2	99,500	64	0.1, 0.2, 0.4	40.50	10.00
	4	25,900	16, 32	0.1, 0.2, 0.3	37.80	10.00
	8	27,000	16	0.1, 0.2, 0.3, 0.4	4.00	10.00
	16	62,200	32	0.1, 0.2, 0.3, 0.4	3.70	10.00
GinConv	2	61,600	64	0.1, 0.2, 0.3, 0.4	37.50	10.00
	4	32,000	32	0.1, 0.2, 0.3, 0.4	37.80	10.00
	8	15,300	16	0.1, 0.2, 0.3, 0.4	4.30	10.00
	16	111,000	64	0.1, 0.2, 0.3, 0.4	3.80	10.00