ENS Lyon
2024/2025 **Titouan Vayer**

TD N° 2 : Graph neural network

- Exercise 1: Nodes classification with GNN on Cora -

The goal of this exercise is to implement a GNN with `PyTorch` for the problem of nodes classification. Precisely, we will have a graph G of n nodes and each node i will have a label y_i. Also there will be features $\mathbf{x}_i \in \mathbb{R}^d$ at every node. For this, we will use the Cora dataset which consists in $n = 2708$ scientific publications classified into seven classes (different categories of article). In addition each publication (node) is described by a 0/1 word vector $\mathbf{x}_i$ indicating the presence of absence of the corresponding word in a dictionary, which has $d = 1433$ unique words.

Before doing anything make sure that you have the `PyTorch Geometric` installed (https://pytorch-geometric.readthedocs.io/en/latest/install/installation.html). We will need to import the following libraries

```
import numpy as np
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import torch_geometric as pyg
import networkx as nx
import sklearn
```

To load the dataset and put the graph into `networkx` we will use the following lines.

```
from torch_geometric.datasets import Planetoid
dataset = Planetoid(root='./Cora', name='Cora')
# Print information
print(dataset)
print('------------')
print(f'Number of graphs: {len(dataset)}')
print(f'Number of features: {dataset.num_features}')
print(f'Number of classes: {dataset.num_classes}')
classes = dataset.y
nx_g = pyg.utils.to_networkx(dataset.data, to_undirected=True)
# Adj matrix of the graph
A = np.array(nx.adjacency_matrix(nx_g).todense())
```

(i) Do a quick inspection of the classes: how many element are there in each class ?

(ii) First we will re-implement from scratch a GNN. Given the $n \times n$ adjacency matrix $\mathbf{A}$ of the graph, and K layers we will implement a GNN given by

$$
\begin{aligned}
&\mathbf{Z}^{(0)} \in \mathbb{R}^{n \times d_{\text{in}}} \\
&\forall k \in \{1, \cdots, K-1\}, \mathbf{Z}^{(k)} = \sigma(G[\mathbf{A}]\mathbf{Z}^{(k-1)}\mathbf{W}^{(k)} + \mathbf{1}_n {\mathbf{b}^{(k)}}^\top) \\
&\text{where } \mathbf{W}^{(1)} \in \mathbb{R}^{d_{\text{in}} \times d_{\text{inter}}}, \forall k > 1, \mathbf{W}^{(k)} \in \mathbb{R}^{d_{\text{inter}} \times d_{\text{inter}}}, \mathbf{b}^{(k)} \in \mathbb{R}^{d_{\text{inter}}} \\
&\mathbf{Z}^{(K)} = G[\mathbf{A}]\mathbf{Z}^{(K-1)}\mathbf{W}^{(K)} + \mathbf{1}_n {\mathbf{b}^{(K)}}^\top \text{ where } \mathbf{W}^{(K)} \in \mathbb{R}^{d_{\text{inter}} \times d_{\text{out}}}, \mathbf{b}^{(K)} \in \mathbb{R}^{d_{\text{out}}}
\end{aligned}
\tag{1}
$$

In this GNN, $\mathbf{Z}^{(0)}$ are the initial features, G is a permutation equivariant function, σ is the ReLU activation function and $(\mathbf{W}^{(k)}), (\mathbf{b}^{(k)})$ are the weight, bias matrices. We also add a constraint: when $K = 1$ the GNN is simply the linear layer We will implement this GNN in pure `PyTorch`. Complete the following code:

```python
class SimpleGCN(nn.Module):
    def __init__(self, d_in, d_inter, d_out, n_layers=1):
        super(SimpleGCN, self).__init__()
        self.n_layers = n_layers
        self.d_in = d_in
        self.d_out = d_out
        self.d_inter = d_inter
        layers = []
        if self.n_layers == 1:
            layers.append(#to complete)
        if n_layers > 1:
            layers.append(#to complete)
            layers.append(#to complete)
            for _ in range(self.n_layers - 1):
                layers.append(#to complete)
                layers.append(#to complete)
            layers.append(#to complete)
        self.neural_net = nn.Sequential(#to complete)

    def forward(self, X, G):
        #Here G stands for the n times n G[A] matrix
        #to complete
        return
```

(iii) Implement the function $G[\mathbf{A}]$ of your choice (for example normalized Laplacian function). It must takes as input a $n \times n$ tensor and return a $n \times n$ tensor and be permutation equivariant.

(iv) We will train the GNN on the Cora dataset. First we do a simple train/test split of the nodes

```python
from sklearn.model_selection import train_test_split
N = len(classes)
index_nodes_train, index_nodes_test = train_test_split(
    np.arange(N), test_size=0.33, random_state=33)
```

To train the model take inspiration from the following code. You must make a choice for: the initial features $\mathbf{Z}^{(0)}$, the dimensions, learning-rate, and loss function. At first take $\mathbf{Z}^{(0)}$ defined by the degree of each node.

```python
n_epochs = #to define
lr = #to define
loss_fn = #to define
gcn = SimpleGCN(d_in=#to define,
              d_inter=#to define,
              d_out=#to define,
              n_layers=#to define)
optimizer = torch.optim.Adam(gcn.parameters(), lr=lr)
for i in range(n_epochs):
    pred = gcn(#to define, #to define)
    pred_on_train = #to define
    loss = loss_fn(input=pred_on_train, target=y_train)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

(v) Compute the accuracy on train/test during the epochs. Compare with three baselines: the `DummyClassifier` from `scikit-learn`, a GNN with $K = 1$ and a very simple linear model with no diffusion $\mathbf{Z} = \sigma(\mathbf{Z}^{(0)}\mathbf{W} + \mathbf{1}_n\mathbf{b}^\top)$. What can you say about the results ? Compare the results by taking $\mathbf{Z}^{(0)}$ to be the features of each node given in the dataset.

(vi) To put everyone on the same page we will use the train/test split directly given by the dataset:

```
index_nodes_train = np.arange(N)[dataset.train_mask]
index_nodes_test = np.arange(N)[dataset.test_mask]
```

Change the different parameters so as to have the best performance (on the test of course) !

(vii) What are the problems in our implementation of the GNN ? Compare with the one of `PyTorch Geometric` (for example with the GCN `torch_geometric.nn.models.GCN`)