ADVANCED LEARNING FOR TEXT AND GRAPH DATA

Lab session 6: Deep Learning for Graphs (2/2)

Lecture: Prof. Michalis Vazirgiannis
Lab: Giannis Nikolentzos & Johannes Lutzeyer

Tuesday, November 21, 2023

This handout includes theoretical introductions, coding tasks and questions. Before the deadline, you should submit here a **.zip** file (max 10MB in size) named `Lab<x>_lastname_firstname.pdf` containing a `/code/` folder (itself containing your scripts with the gaps filled) and an answer sheet, following the template available here, and containing your answers to the questions. Your answers should be well constructed and well justified. They should not repeat the question or generalities in the handout. When relevant, you are welcome to include figures, equations and tables derived from your own computations, theoretical proofs or qualitative explanations. **One submission is required for each student. The deadline for this lab is November 28, 2023 11:59 PM**. No extension will be granted. Late policy is as follows: $]0, 24]$ hours late $\rightarrow$ -5 pts; $]24, 48]$ hours late $\rightarrow$ -10 pts; > 48 hours late $\rightarrow$ not graded (zero).

---

1 Learning objective

In this lab, you will learn about neural networks that operate on graphs. These models can be employed for addressing various tasks such as node classification, graph classification and link prediction. The lab is divided into two parts. In the first part, you will implement a graph neural network (GNN) that operates at the graph level, and you will evaluate it in a graph classification task. In the second part, you will investigate the expressive power of standard graph neural networks, i.e., their ability to distinguish non-isomorphic graphs. We will use Python, and the following two libraries: (1) PyTorch (`https://pytorch.org/`), and (2) NetworkX (`http://networkx.github.io/`).

2 Graph Neural Networks for Graph-Level Tasks

In the second part of the lab, we will focus on graph neural networks that operate at the graph level (e.g., each sample is a graph and not a node). Common tasks for these models include graph classification and graph regression. Graph classification arises in the context of a number of classical domains such as chemical data, biological data, and the web.

2.1 Dataset Generation

We will first create a very simple graph classification dataset. The dataset will contain two types of graphs: (1) instances of the $G(n, 0.2)$ Erdős-Rényi random model, and (2) instances of the $G(n, 0.4)$ Erdős-Rényi random model. In both cases n will take values between 10 and 20. In the $G(n, p)$ model,

a graph is constructed by connecting nodes randomly. Each edge is included in the graph with probability p independent from every other edge. Therefore, the density of the graphs of the second class is very likely to be higher than that of the graphs of the first class. Use the `fast_gnp_random_graph()` function of NetworkX to generate 50 graphs using the $G(n, 0.2)$ model and 50 graphs using the $G(n, 0.4)$ model. Store the 100 graphs in a list and their class labels in another list.

Question 1 (5 points)
What is the expected number of degree of a node in Erdős-Rényi random graphs with $n = 25$ nodes and edge probabilities $p = 0.2$ and $p = 0.4$?

Task 1
Fill in the body of the `create_dataset()` function in the `utils.py` file to generate the dataset as described above.

Before applying the graph neural network, it is necessary to split the dataset into a training and a test set. We can use the `train_test_split()` function of scikit-learn as follows:

```
from sklearn.model_selection import train_test_split

G_train, G_test, y_train, y_test = train_test_split(Gs, y, test_size=0.1)
```

2.2 Implementation of Graph Neural Network

You will next implement a GNN model that consists of two message passing layers followed by a sum readout function and then, by two fully-connected layers. The first layer of the model is a message passing layer:

$$\mathbf{Z}^0 = \text{ReLU}(\tilde{\mathbf{A}}\,\mathbf{X}\,\mathbf{W}^0)$$

where $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ and $\mathbf{W}^0$ is a trainable matrix (we omit bias for clarity). The second layer of the model is again a message passing layer:

$$\mathbf{Z}^1 = \text{ReLU}(\tilde{\mathbf{A}}\,\mathbf{Z}^0\,\mathbf{W}^1)$$

where $\mathbf{W}^1$ is another trainable matrix (once again, we omit bias for clarity). The two message passing layers are followed by a readout layer which uses the sum operator to produce a vector representation of the entire graph:

$$\mathbf{z}_G = \text{READOUT}(\mathbf{Z}^1)$$

where READOUT is the readout function (i.e., the sum function). The readout layer is followed by two fully-connected layers which produce the output (i.e., a vector with one element per class):

$$\mathbf{y} = \sigma\big(\text{ReLU}(\mathbf{z}_G\,\mathbf{W}^2)\,\mathbf{W}^3\big)$$

where $\mathbf{W}^2, \mathbf{W}^3$ are matrices of trainable weights (biases are omitted for clarity) and $\sigma(\cdot)$ denotes the softmax function.

Task 2
Implement the architecture presented above in the `models.py` file. More specifically, add the following layers:
– a message passing layer with h_1 hidden units (i.e., $\mathbf{W}^0 \in \mathbb{R}^{d \times h_1}$) followed by a ReLU activation function
– a message passing layer with h_2 hidden units (i.e., $\mathbf{W}^1 \in \mathbb{R}^{h_1 \times h_2}$) followed by a ReLU activation function
– a readout function that computes the sum of the node representations
– a fully-connected layer with h_3 hidden units (i.e., $\mathbf{W}^2 \in \mathbb{R}^{h_2 \times h_3}$) followed by a ReLU activation function
– a fully-connected layer with n_{class} hidden units (i.e., $\mathbf{W}^3 \in \mathbb{R}^{h_3 \times n_{class}}$) where n_{class} is the number of different classes
(Hint: use the `Linear()` module of PyTorch to define a fully-connected layer. You can perform a matrix-matrix multiplication using the `torch.mm()` function).

2.3 Graph Classification

We next discuss some implementation details of the GNN. Since different graphs may have different number of nodes from each other, to create mini-batches, possibly the best approach is to concatenate the respective feature matrices and build a (sparse) block-diagonal matrix where each block corresponds to the adjacency matrix of one graph. This is illustrated in Figure 1 (Left) for three graphs G_1, G_2 and G_3. If n_1, n_2 and n_3 denote the number of nodes of the three graphs, and $n = n_1 + n_2 + n_3$, then we have that $\mathbf{A} \in \mathbb{R}^{n \times n}$ and $\mathbf{X} \in \mathbb{R}^{n \times d}$.

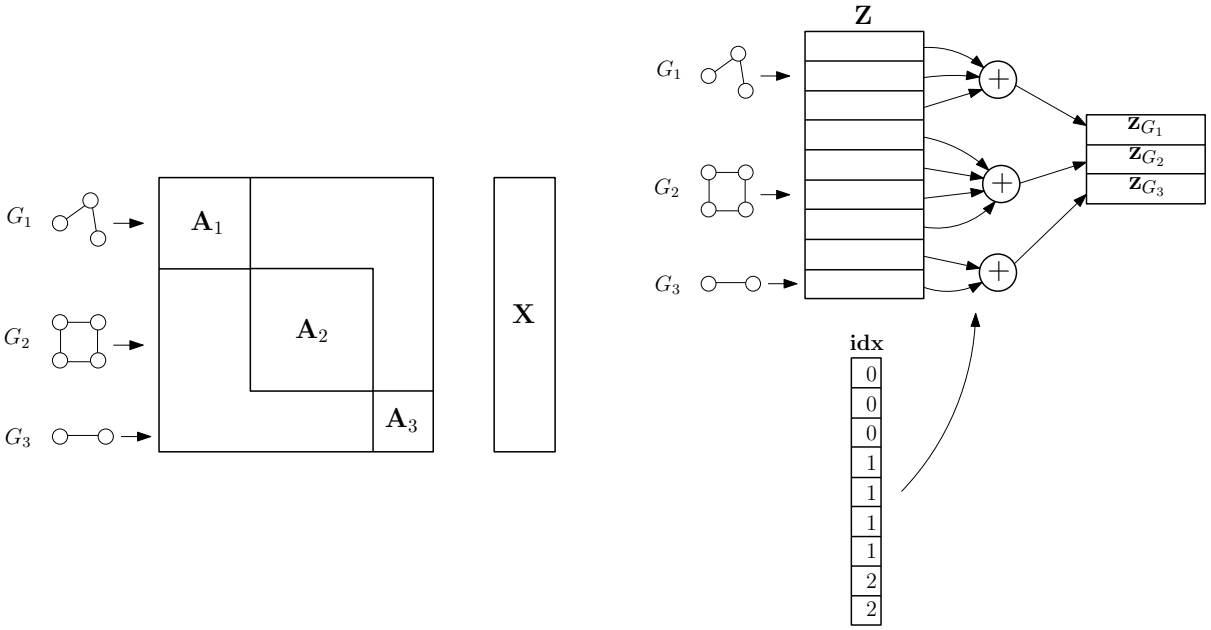

Figure 1: Implementation details of the graph neural network.

In the readout phase, for each graph, we need to apply the readout function to the rows of the feature matrix that correspond to the nodes of that graph. To achieve that, we can create a membership indicator vector which for each node indicates the graph to which this nodes belongs, and then to use an aggregation function such as the `scatter_add_()` function of PyTorch. This idea is illustrated in Figure 1 (Right).

Question 2 (5 points)
Could you briefly comment on why trainable linear layers, i.e., fully connected layers, are not commonly used as readout functions instead of the sum or mean operation in graph level GNNs? You can use the scenario depicted in Figure 1 with G_1, G_2 and G_3 to exemplify your answer.

We will next capitalize on the above scheme and iterate over the different batches to train the model.

Task 3
– In the `gnn.py` script, for each batch of indices (both for training and evaluation), create:
- a sparse block diagonal matrix that contains the adjacency matrices of all graphs of the batch (use the `block_diag()` function of Scipy)
- a matrix that contains the features of the nodes of all the graphs of the batch. Since the nodes are not annotated with any attributes, set the features of all nodes to the same value (e.g., a value equal to 1)
- a vector that indicates the graph to which each node belongs
- a vector that contains the class labels of the graphs

– Convert the above NumPy/Scipy arrays into PyTorch tensors (to covert a sparse matrix into a sparse PyTorch tensor, use the `sparse_mx_to_torch_sparse_tensor()` function)
– Execute the code to train and evaluate the model

3 How Powerful are Graph Neural Networks?

The GNNs that belong to the general framework presented in Section 2 are known as standard GNNs. It has been shown that if a model GNN: $\mathcal{G} \rightarrow \mathbb{R}^d$ maps two graphs G_1 and G_2 to different embeddings, then the Weisfeiler-Lehman (WL) graph isomorphism test also decides that G_1 and G_2 are not isomorphic [5, 3]. Therefore, standard GNNs are at most as powerful as the WL test of isomorphism. In fact, several well-established models such as GCN [2], GraphSAGE [1], and GAT [4] are less powerful than the WL test. The expressive power of a GNN depends a lot on the type of function it employs to aggregate the representations of the neighbors of a node and also on the type of the readout function. We will next try to shed some light on the weaknesses of GNNs.

We will first create a very small dataset which will consist of only a single type of graphs. Specifically, we will create 10 cycle graphs of different lengths. A cycle graph C_n is a graph on n nodes containing a single cycle through all nodes.

Task 4
Use the `cycle_graph()` function of NetworkX to generate 10 cycle graphs of size $n = 10, \ldots, 19$.

We assume that no node features are available. Therefore, we can either annotate all nodes with the same feature vector or alternatively, we can annotate nodes with their degrees. We will next produce all the data that the GNN will take as input.

Task 5

– In the `expressive_power.py` script, create:

- a sparse block diagonal matrix that contains the adjacency matrices of the 10 cycle graphs (use the `block_diag()` function of Scipy)
- a matrix that contains the features of the nodes of all the graphs of the batch. Since the nodes are not annotated with any attributes, set the features of all nodes to the same value (e.g., a value equal to 1)
- a vector that indicates the graph to which each node belongs

– Convert the above NumPy/Scipy arrays into PyTorch tensors (to covert a sparse matrix into a sparse PyTorch tensor, use the `sparse_mx_to_torch_sparse_tensor()` function)

We will next define a message passing layer which uses either the sum or the mean operator to aggregate the representations of the neighbors of each node. These two layers are defined as follows:

$$\mathbf{Z}_{sum} = \mathbf{X}\ \mathbf{W}_1 + \mathbf{A}\ \mathbf{X}\ \mathbf{W}_0$$
$$\mathbf{Z}_{mean} = \mathbf{X}\ \mathbf{W}_1 + \mathbf{D}^{-1}\ \mathbf{A}\ \mathbf{X}\ \mathbf{W}_0$$

where $\mathbf{A}$ is the adjacency matrix of the graph, $\mathbf{X}$ is a matrix that contains the representations of the nodes, $\mathbf{D}$ is a diagonal matrix such that $\mathbf{D}_{ii} = \sum_j \mathbf{A}_{ij}$, and $\mathbf{W}_0, \mathbf{W}_1$ are trainable matrices (we omit biases for clarity).

Task 6

Implement the message passing layer presented above in the `models.py` file. More specifically, add the following layers:

– a fully-connected layer with h_1 hidden units (i.e., $\mathbf{W}_0 \in \mathbb{R}^{d \times h_1}$)

– a second fully-connected layer with h_1 hidden units (i.e., $\mathbf{W}_1 \in \mathbb{R}^{d \times h_1}$)

– a sum or mean neighborhood aggregation layer.

You will next implement a GNN model that consists of two message passing layers followed by either a sum or a mean readout function and then, by a fully-connected layer. The first layer of the model is a message passing layer:

$$\mathbf{Z}^0 = \text{ReLU}\big(\text{MP}(\mathbf{A}, \mathbf{X})\big)$$

where MP is the message passing layer defined above. The second layer of the model is again a message passing layer:

$$\mathbf{Z}^1 = \text{ReLU}\big(\text{MP}(\mathbf{A}, \mathbf{Z}^0)\big)$$

The two message passing layers are followed by a readout layer which uses either the sum or the mean operator to produce a vector representation of the entire graph.

$$\mathbf{z}_G^0 = \text{READOUT}(\mathbf{Z}^1)$$

where READOUT is the readout function (i.e., either sum or mean of node representations). The readout layer is finally followed by a fully-connected layer which produces the final graph representation as follows:

$$\mathbf{z}_G^1 = \mathbf{z}_G^0\ \mathbf{W} + \mathbf{b}$$

where $\mathbf{W}$ is a matrix of trainable weights and $\mathbf{b}$ is a bias vector.

Attention!! Note that we have not defined any classification or regression task so far. Furthermore, the output of the model is a vector representation for each input graph (and not for instance a prob-

ability distribution among classes). In fact, the model will not be trained on any dataset, but we will only randomly initialize its parameters (i.e., weight matrices and biases), and we will use the model to produce vector representations of the nodes.

Task 7
Implement the architecture presented above in the `models.py` file. More specifically, add the following layers:
– an instance of the message passing layer described above with h_1 hidden units followed by a ReLU activation function
– a dropout layer with with p_d ratio of dropped outputs
– a second instance of the message passing layer described above with h_2 hidden units followed by a ReLU activation function
– a readout function that computes either the sum or the mean of the node representations
– a fully-connected layer with h_3 hidden units (i.e., $\mathbf{W} \in \mathbb{R}^{h_2 \times h_3}$) (Hint: use the `Linear()` module of PyTorch to define a fully-connected layer).

Next, we will use the model that we implemented to produce vector representations of the 10 cycle graphs.

Task 8
In the `expressive_power.py` script, initialize a GNN which uses the mean operator both for neighborhood aggregation and for the readout function. Perform a feedforward pass to compute the representations of the 10 graphs and print the 10 emerging vectors. Initialize new GNNs which use different combinations of the sum and mean operators for neighborhood aggregation and for readout, and again print the emerging vector representations of the graphs.

Question 3 (5 points)
Please comment in a few sentences what you observe in the output produced in Task 8. How can the obtained results be explained?

We next study if the standard GNN architecture can distinguishing the two graphs G_1 and G_2 shown in Figure 2. Clearly, the two graphs are not isomorphic to each other. For instance, G_2 is connected, while G_1 consists of two connected components. Let's first create the two graphs.

Task 9
Use the NetworkX library to create the two graphs G_1 and G_2 shown in Figure 2.

We assume that either all nodes or nodes with the same degree are annotated with the same feature vector. We will next produce all the data that the GNN needs to produce graph representations.

Task 10
– In the `expressive_power.py` script, for each batch of indices (both for training and evaluation), create:

- a sparse block diagonal matrix that contains the adjacency matrices of the two graphs (use the `block_diag()` function of Scipy)
- a matrix that contains the features of the nodes of all the graphs of the batch. Since the nodes are not annotated with any attributes, set the features of all nodes to the same value (e.g., a value equal to 1)
- a vector that indicates the graph to which each node belongs

– Convert the above NumPy/Scipy arrays into PyTorch tensors (to covert a sparse matrix into a sparse PyTorch tensor, use the `sparse_mx_to_torch_sparse_tensor()` function)

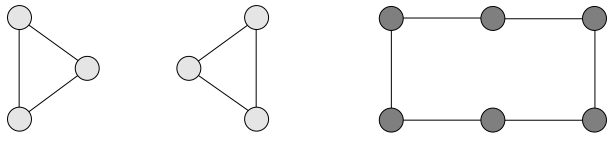

Figure 2: Implementation details of the graph neural network.

Finally, we will initialize an instance of the GNN we defined, and we will use the model to generate vector representations of the two graphs.

Task 11
Initialize a GNN which uses the sum operator both for neighborhood aggregation and as the model's readout function. Perform a feedforward pass to compute the representations of the two graphs and print those vector representations.

What do you observe?

Question 4 (5 points)
Give an example of two non-isomorphic graphs G_1 and G_2 (different from the ones shown in Figure 2 which can never be distinguished by the GNN model which uses the sum operator both for neighborhood aggregation and as the model's readout function.

References

[1] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st Conference on Neural Information Processing System*, pages 1025–1035, 2017.

[2] Thomas N Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations*, 2017.

[3] Christopher Morris, Martin Ritzert, Matthias Fey, William L Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and Leman Go Neural: Higher-Order Graph Neural Networks. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*, pages 4602–4609, 2019.

[4] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph Attention Networks. In *6th International Conference on Learning Representations*, 2018.

[5] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How Powerful Are Graph Neural Networks? In *7th International Conference on Learning Representations*, 2019.