Advanced Learning for Text and Graph Data

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

Lecture: Prof. Michalis Vazirgiannis
Lab: Giannis Nikolentzos & Michalis Chatzianastasis

Tuesday, November 29, 2022

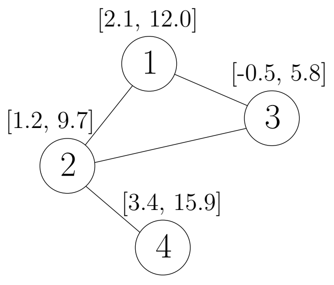
This handout includes theoretical introductions, coding tasks and questions. Before the deadline, you should submit on moodle **or** here a **.zip** file named `Lab<x>_lastname_firstname.pdf` containing a `/code/` folder (itself containing your scripts with the gaps filled) and an answer sheet, following 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 December 13, 2022 11:59 PM**. No extension will be granted. Late policy is as follows: $]0, 24]$ hours late → -5 pts; $]24, 48]$ hours late → -10 pts; > 48 hours late → 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 which belongs to the family of message passing models. The message passing layers of the model that will be implemented employs a self-attention mechanim that allows nodes to attend to their most important neighbors. The model will be evaluated in a graph classification task. In the second part, you will implement a graph neural network that operates at the graph level, and the model will be evaluated in a graph classification task. The two models will be implemented in Python, and we will use the following two libraries: (1) PyTorch (`https://pytorch.org/`), and (2) NetworkX (`http://networkx.github.io/`).

2 Graph Attention Network for Node-Level Tasks

In the first part of the lab, we will implement a graph neural network (GNN), and use it to classify the nodes of a small benchmark graph. Let $\mathbf{A}$ be the adjacency matrix of a graph $G = (V, E)$ and $\mathbf{X}$ its feature matrix. For attributed graphs, these features may be set equal to the attribute vectors of the nodes. For instance, in biology, proteins are represented as graphs where nodes correspond to secondary structure elements and the feature vector of each secondary structure element contains its physical properties. For graphs without node labels and node attributes, these vectors can either be



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

$$\mathbf{X} = \begin{bmatrix} 2.1 & 12.0 \\ 1.2 & 9.7 \\ -0.5 & 5.8 \\ 3.4 & 15.9 \end{bmatrix}$$

Figure 1: An example of a graph and of its associated matrices $\mathbf{A}$ and $\mathbf{X}$.

random or can be initialized with a collection of local vertex features that are invariant to vertex renumbering (e.g., degree). An example of a graph and of its associated matrices $\mathbf{A}$ and $\mathbf{X}$ is given in Figure 1. A standard GNN model takes the matrices $\mathbf{A}$ and $\mathbf{X}$ as input and learns a hidden representation for each node.

Most GNN models update the representation of each node by aggregating the feature vectors of its neighbors. This update procedure can be viewed as a form of message passing algorithm. Different types of message passing layers have been proposed in the past years. Recently, a lot of interest has been devoted to layers that compute the hidden representation of each node in the graph by attending over its neighbors [2, 1]. This allows a node to selectively attend to specific neighbors while ignoring others. We will next implement such a message passing layer which follows a self-attention strategy to update the representations of the nodes.

2.1 Implementation of Graph Attention Layer

You will implement a GNN model that consists of two message passing layers that use attention, and which are followed by a fully-connected layer. As discussed above, messages from some neighbors may be more important than messages from others, thus one could apply self-attention on the nodes to capture message importance. Formally, let $\mathcal{N}(v_i)$ denote the set of neighbors of node v_i. For nodes $v_j \in \mathcal{N}(v_i)$, the attention layer that we will implement computes attention coefficients that indicate the importance of node v_j's features to node v_i as follows:

$$\alpha_{ij}^{(t+1)} = \frac{\exp\Big(\text{LeakyReLU}\big(\mathbf{a}^{(t+1)\top}[\mathbf{W}^{(t+1)}\mathbf{z}_i^{(t)}||\mathbf{W}^{(t+1)}\mathbf{z}_j^{(t)}]\big)\Big)}{\sum_{k\in\mathcal{N}_i}\exp\Big(\text{LeakyReLU}\big(\mathbf{a}^{(t+1)\top}[\mathbf{W}^{(t+1)}\mathbf{z}_i^{(t)}||\mathbf{W}^{(t+1)}\mathbf{z}_k^{(t)}]\big)\Big)}$$

where $[\cdot||\cdot]$ denotes concatenation of two vectors, $\mathbf{z}_i^{(t)}$ and $\mathbf{z}_j^{(t)}$ are the hidden representations of nodes v_i and v_j, and $\mathbf{W}^{(t+1)}$, $\mathbf{a}^{(t+1)}$ denote a trainable matrix and a trainable vector of the $(t+1)$-th message passing layer. Then the representations of the nodes are updated as follows:

$$\mathbf{z}_i^{(t+1)} = \sum_{j\in\mathcal{N}_i} \alpha_{ij}^{(t+1)}\mathbf{W}^{(t+1)}\mathbf{z}_j^{(t)}$$

In matrix form, the above is equivalent to:

$$\mathbf{Z}^{(t+1)} = \big(\mathbf{A}\odot\mathbf{T}^{(t+1)}\big)\mathbf{Z}^{(t)}\mathbf{W}^{(t+1)}$$

where $\odot$ denotes elementwise product and $\mathbf{T}^{(t)}$ is a matrix such that $\mathbf{T}_{ij}^{(t)} = \alpha_{ij}^{(t)}$.

Task 1
Implement the message passing layer presented above (fill in the body of the `forward()` function of the `GATLayer` class in the `models.py` file). More specifically, you only need to
– compute the output of LeakyReLU$(\mathbf{a}^\top[\mathbf{W}\mathbf{z}_i||\mathbf{W}\mathbf{z}_j])$ for all pairs of nodes v_i, v_j that are connected by an edge (Hint: use the following to obtain a tensor (of dimension $2 \times m$) that contains all pairs of nodes that are connected by an edge: `adj.coalesce().indices()`. You can concatenate two tensors using the `torch.cat()` function of PyTorch).
– perform the message passing by multiplying matrix $(\mathbf{A} \odot \mathbf{T})$ by the matrix that stores the node features (Hint: you can perform a matrix-matrix multiplication using the `torch.mm()` function).

Question 1 (5 points)
Let $G = (V, E)$ be a graph where $V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$ is the set of nodes. Let also $\mathcal{N}(v_1) = \{v_2, v_3\}$ and $\mathcal{N}(v_4) = \{v_2, v_3, v_5, v_6\}$ denote the neighborhoods of nodes v_1 and v_4, respectively. Suppose the following hold for the representations of the nodes that emerge at the output of some message passing layer:

$$\mathbf{z}_1^{(1)} = \mathbf{z}_4^{(1)}$$
$$\mathbf{z}_2^{(1)} = \mathbf{z}_6^{(1)}$$
$$\mathbf{z}_3^{(1)} = \mathbf{z}_5^{(1)}$$

What is the relationship between $\mathbf{z}_1^{(2)}$ and $\mathbf{z}_4^{(2)}$ and why?

2.2 Implementation of Graph Neural Network

You will next implement the GNN model. Let $\mathbf{A}$ be the adjacency matrix of the graph, and $\mathbf{X}$ a matrix whose i^{th} row contains the feature vector of node v_i. The first layer of the model is a message passing layer that employs the attention mechanism presented above, and is defined as follows:

$$\mathbf{Z}^{(1)} = f\Big((\mathbf{A} \odot \mathbf{T}^{(1)})\mathbf{X}\mathbf{W}^{(1)}\Big)$$

where $\mathbf{W}^{(1)}$ is a matrix of trainable weights, f is an activation function (e.g., ReLU, sigmoid, tanh), and $\mathbf{T}^{(1)}$ is a matrix that contains the attention coefficients. The second layer of the model is again a message passing layer that employs the attention mechanism presented above, and is defined as follows:

$$\mathbf{Z}^{(2)} = f\Big((\mathbf{A} \odot \mathbf{T}^{(2)})\mathbf{Z}^{(1)}\mathbf{W}^{(2)}\Big)$$

where $\mathbf{W}^{(2)}$ is a second matrix of trainable weights, f is an activation function, and $\mathbf{T}^{(2)}$ is a matrix that contains the attention coefficients of the second message passing layer. The two message passing layers are followed by a fully-connected layer which makes use of the softmax function to produce a probability distribution over the class labels:

$$\hat{\mathbf{Y}} = \text{softmax}(\mathbf{Z}^{(2)}\ \mathbf{W}^{(3)})$$

where $\mathbf{W}^{(3)}$ is a third matrix of trainable weights. Note that for clarity of presentation we have omitted biases.

Task 2
Fill in the body of the `forward()` function of the `GNN` class in the `models.py` file to implement the architecture presented above. More specifically, add the following layers:
– a message passing layer with h_1 hidden units (i.e., $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h_1}$) followed by a ReLU activation function.
– a dropout layer with with p_d ratio of dropped outputs.
– a message passing layer with h_2 hidden units (i.e., $\mathbf{W}^{(2)} \in \mathbb{R}^{h_1 \times h_2}$) followed by a ReLU activation function.
– a fully-connected layer with n_{class} units (i.e., $\mathbf{W}^{(3)} \in \mathbb{R}^{h_2 \times n_{class}}$) followed by the softmax activation function.
– (Hint: the message passing layer is already implemented, thus you only need to feed the node features and the adjacency matrix to instances of that layer).

2.3 Node Classification

We will evaluate the GNN you implemented in a node classification task. The experiments for the first part will be performed on a small dataset which is known as the *karate* network, and which has been used as a benchmark mainly for community detection algorithms. The karate dataset is a friendship social network between 34 members of a karate club at a US university in the 1970s. Due to some conflict between the club administrator and the club main instructor, the members were split into two different groups which correspond to the two classes of the dataset.

Task 3
Compute the adjacency matrix of the karate network. Initialize the features of the nodes to values drawn from the standard normal distribution (Hint: use the `randn()` function of NumPy). Then, run the `gnn_karate.py` script to train the model, make predictions, and compute the classification accuracy.

Question 2 (5 points)
Suppose that instead of random features, nodes are annotated with identical features (i.e., the rows of matrix $\mathbf{X}$ are identical to each other). Will the model still achieve a high classification accuracy and why?

2.4 Visualization of Attention Scores

Finally, we will visualize the learned attention coefficients. Note that matrix $\mathbf{T}$ that stores the attention weights is not generally symetric, i.e., $\alpha_{ij} = \alpha_{ji}$ does not necessarily hold. We will visualize attention coefficients as follows. We will first create a directed graph that has the same structure as the karate network (i.e., each edge will be replaced with two edges of opposite directions). Then, we will set the width of each edge equal to the strength of the relationship between the source and the sink node. The width of an edge will thus capture how much the source node attends to the sink node.

We will re-train the model that we have already implemented, and then extract the attention scores that are produced from the second message passing layer of the model.

Task 4
Modify the code in the `models.py` file such that the model also returns the attention scores produced in the second message passing layer. Then, in the `gnn_karate.py` script, transform the Torch tensor that contains the attention scores into a NumPy vector (Hint: for a tensor `T`, you can do this using `T.detach().cpu().numpy()`).

3 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) [4, 3]. 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.

3.1 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.

Task 5
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.

3.2 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:

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

where $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ and $\mathbf{W}^{(1)}$ is a trainable matrix (we omit bias for clarity). The second layer of the model is again a message passing layer:

$$\mathbf{Z}^{(2)} = \tilde{\mathbf{A}}\,\mathbf{Z}^{(1)}\,\mathbf{W}^{(2)}$$

where $\mathbf{W}^{(2)}$ 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:

$$\mathbf{z}_G = \text{READOUT}(\mathbf{Z}^{(2)})$$

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):

$$\mathbf{y} = \sigma\big(\text{ReLU}(\mathbf{z}_G\,\mathbf{W}^{(3)})\,\mathbf{W}^{(4)}\big)$$

where $\mathbf{W}^{(3)}, \mathbf{W}^{(4)}$ are matrices of trainable weights (biases are omitted for clarity) and $\sigma(\cdot)$ denotes the softmax function.

Task 6
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., $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h_1}$) followed by a ReLU activation function
– a message passing layer with h_2 hidden units (i.e., $\mathbf{W}^{(2)} \in \mathbb{R}^{h_1 \times h_2}$)
– a readout function that computes the sum of the node representations
– a fully-connected layer with h_3 hidden units (i.e., $\mathbf{W}^{(3)} \in \mathbb{R}^{h_2 \times h_3}$) followed by a ReLU activation function
– a fully-connected layer with n_{class} hidden units (i.e., $\mathbf{W}^{(4)} \in \mathbb{R}^{h_3 \times n_{class}}$) 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).

3.3 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 2 (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 $\mathbf{X} \in \mathbb{R}^{n \times d}$.

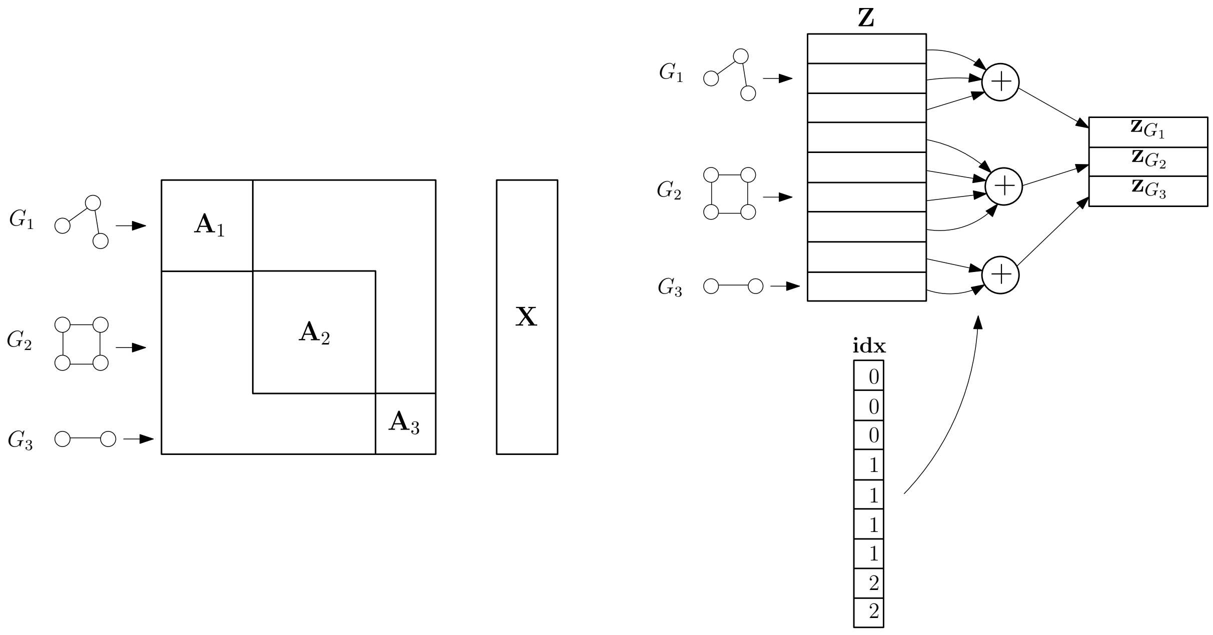


Figure 2: Implementation details of the graph neural network.

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 2 (Right).

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

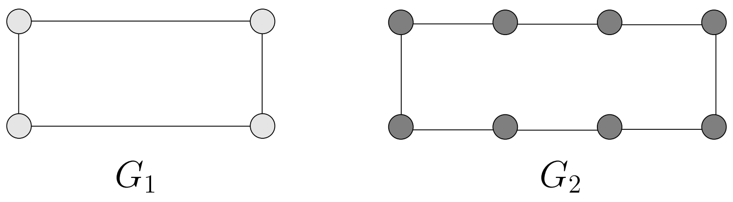


Figure 3: The C_4 and C_8 graphs where C_n denotes a cycle consisting of n nodes.

Task 7
– 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

Question 3 (5 points)
Assume that for a set of three graphs we receive the following matrix $\mathbf{Z}$

$$\mathbf{Z} = \begin{bmatrix} 2.2 & -0.6 & 1.4 \\ 0.2 & 1.8 & 1.5 \\ 0.5 & 1.1 & -1.0 \\ 0.7 & 0.1 & 1.3 \\ 1.2 & -0.9 & 0.3 \\ 2.2 & 0.9 & 1.2 \\ -0.7 & 1.8 & 1.5 \\ -0.4 & 1.8 & 0.1 \\ 2.2 & -0.6 & 1.5 \end{bmatrix}$$

where rows 1, 2 and 3 correspond to nodes in the first graph G_1, rows 4, 5, 6 and 7 correspond to nodes in G_2 and the remaining two rows correspond to nodes in G_3. Compute the representations of the three graphs (i.e., $\mathbf{z}_{G_1}, \mathbf{z}_{G_2}$ and $\mathbf{z}_{G_3}$) for each one of the following three readout functions: (i) sum, (ii) mean, and (iii) max. Which of these functions is able to distinguish these graphs best?

Question 4 (5 points)
Suppose we feed the two graphs G_1 and G_2 shown in Figure 3 to the above model (we initialize the features of all nodes to 1). What will be the relationship between the emerging representations $\mathbf{z}_{G_1}$ and $\mathbf{z}_{G_2}$ and why?

References

[1] Shaked Brody, Uri Alon, and Eran Yahav. How Attentive are Graph Attention Networks? In *10th International Conference on Learning Representations*, 2022.

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

[3] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

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