ADVANCED LEARNING FOR TEXT AND GRAPH DATA

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

Lecture: Prof. Michalis Vazirgiannis
Lab: Giannis Nikolentzos, George Panagopoulos

Tuesday, January 12, 2021

This handout includes theoretical introductions, coding tasks and questions. Before the deadline, you should submit here a **.zip** file (max 10MB in size) containing a `/code/` folder (itself containing your scripts with the gaps filled) and an answer sheet named `firstname_lastname.pdf`, 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 January 19, 2021 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 autoencoder, i.e., a model that generates node representations in an unsupervised manner. These embeddings can then be fed to conventional machine learning algorithms for solving any downstream task. In the second part, you will implement a graph neural network that operates at the graph level, and you will evaluate it in a graph classification task. We will use Python 3.6, and the following two libraries: (1) PyTorch (`https://pytorch.org/`), and (2) NetworkX (`http://networkx.github.io/`).

2 Graph Autoencoders

In this part of the lab, we will implement a graph autoencoder, and use it to visualize some data and cluster the nodes of a small benchmark graph. In the case of autoencoders for graph-structured data, the input to the model is a graph and the objective is to learn a low-dimensional representation for each node. 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 be initialized with a collection of local vertex features

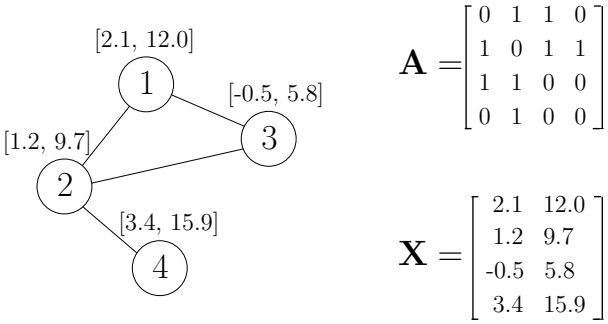

Figure 1: An example of a graph and of its associated matrices $\mathbf{A}$ and $\mathbf{X}$.

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 graph autoencoder model takes the matrices $\mathbf{A}$ and $\mathbf{X}$ as input and reconstructs the adjacency matrix $\mathbf{A}$ which contains information about the graph structure. A high-level illustration of such an autoencoder is given in Figure 2. In this first part of the lab, we will implement an autoencoder

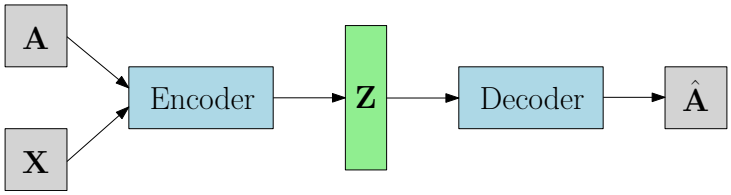

Figure 2: High-level illustration of a graph autoencoder.

similar to the one illustrated in Figure 2.

2.1 Implementation of Graph Autoencoder

Given the adjacency matrix $\mathbf{A}$ of a graph, we will first normalize it as follows:

$$\bar{\mathbf{A}} = \tilde{\mathbf{D}}^{-1}\,\tilde{\mathbf{A}}$$

where $\tilde{\mathbf{A}} = \mathbf{A}+\mathbf{I}$, and $\tilde{\mathbf{D}}$ is a diagonal matrix such that $\tilde{\mathbf{D}}_{ii} = \sum_j \tilde{\mathbf{A}}_{ij}$. The above formula adds self-loops to the graph, and normalizes each row of the emerging matrix such that the sum of its elements is equal to 1. This normalization trick addresses numerical instabilities which may lead to exploding/vanishing gradients when used in a deep neural network model.

> **Task 1**
> Fill in the body of the `normalize_adjacency()` function in the `utils.py` file that applies the above normalization trick. Note that the adjacency matrix is stored as a sparse matrix. Use operations of the NumPy and SciPy libraries (e.g., `identity()` function of SciPy) to also produce a sparse normalized matrix.

We will next implement the graph autoencoder. The encoder of the model will correspond to a graph neural network (GNN). Thus, the encoder takes the adjacency matrix $\bar{\mathbf{A}}$ and the feature matrix $\mathbf{X}$ as inputs, performs a series of neighborhood aggregation iterations and generates a vector representation for each node:

$$\mathbf{Z} = \text{GNN}(\bar{\mathbf{A}}, \mathbf{X})$$

Note that the i^{th} row of matrix $\mathbf{Z}$ contains the representation of the i^{th} node of the graph. In our implementation, the encoder will consist of two message passing layers. The first layer of the encoder is defined as:

$$\mathbf{H} = \text{ReLU}(\bar{\mathbf{A}}\ \mathbf{X}\ \mathbf{W}^0)$$

where $\mathbf{X}$ is a matrix that contains the representations of the nodes, and $\mathbf{W}^0$ is a trainable matrix (we omit bias for clarity). The second layer of the encoder is again a message passing layer:

$$\mathbf{Z} = \bar{\mathbf{A}}\ \mathbf{H}\ \mathbf{W}^1$$

The decoder takes as input the matrix of low-dimensional node representations $\mathbf{Z}$ and generates a reconstructed adjacency matrix. In our implementation, the decoder will correspond just to an inner product between the node representations. More specifically, the decoder is defined as:

$$\hat{\mathbf{A}} = \sigma(\mathbf{Z}\,\mathbf{Z}^\top)$$

where $\sigma(\cdot)$ is the sigmoid 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 dropout layer with with p_d ratio of dropped outputs
– 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
(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).

Note that the elements of matrix $\hat{\mathbf{A}}$ take values between 0 and 1, and ideally, we would like these values to be as close as possible to the correponding values contained in $\tilde{\mathbf{A}}$. To measure how well the model reconstructs the input data, i.e., the adjacency matrix (with self-loops), we will employ the mean squared error as follows:

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{n}(\tilde{\mathbf{A}}_{ij} - \hat{\mathbf{A}}_{ij})^2$$

For very sparse adjacency matrices $\tilde{\mathbf{A}}$, it can be beneficial to re-weight the nonzero terms or alternatively sub-sample pairs of nodes for which $\tilde{\mathbf{A}}_{ij} = 0$. We choose the latter strategy for our implementation.

Task 3
Fill in the body of the `loss_function()` function in the `utils.py` file which computes the above loss function. The loss function will take into account all the edges (i.e., elements for which $\tilde{\mathbf{A}}_{ij} = 1$), and an equal number of randomly sampled negative pairs of nodes (Hint: use the `adj._indices()` function of PyTorch to obtain the indices of all the nonzero terms of the adjacency matrix).

Question 1 (5 points)
For unweighted graphs, could you treat the problem as a classification one and how would you modify the architecture?

2.2 Visualization of Node Representations

The experiments for this part will be performed on a small dataset which is known as the *karate* network, 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.

We will train the model that you have already implemented, and then extract the nodes' hidden representations that are produced from the second message passing layer of the model. The representations of the nodes will be projected into the two-dimensional space and visualized. Since the karate network does not contain node attributes, we annotate the nodes with randomly-generated vectors (10-dimensional vectors).

Task 4
In the `graph_autoencoder.py` script, transform the Torch tensor that contains the low-dimensional node representations into a NumPy matrix (Hint: for a tensor `T`, you can do this using `T.detach().cpu().numpy()`). Then, project these representations to two dimensions using PCA (Hint: use scikit-learn's implementation).

2.3 Clustering of Nodes

We will next evaluate the graph autoencoder you implemented in a node clustering task.

Task 5
Use the k-means clustering algorithm to partition the network into 2 clusters and compute the homogeneity of the clustering result (Hint: to perform k-means, you can use scikit-learn's implementation of the algorithm).

Question 2 (5 points)
The graph autoencoder that we implemented above reconstructs the adjacency matrix $\mathbf{A}$ of the input graph. For graphs that contain node attributes, could we also reconstruct the feature matrix $\mathbf{X}$ and how?

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). 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 6
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)
```

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:

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

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:

$$\mathbf{Z}^1 = \tilde{\mathbf{A}}\,\mathbf{Z}^0\,\mathbf{W}^1$$

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):

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

where $\mathbf{W}^2, \mathbf{W}^3$ are matrices of trainable weights (biases are omitted for clarity).

Task 7
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., $\mathbf{W}^1 \in \mathbb{R}^{h_1 \times h_2}$)
– 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).

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

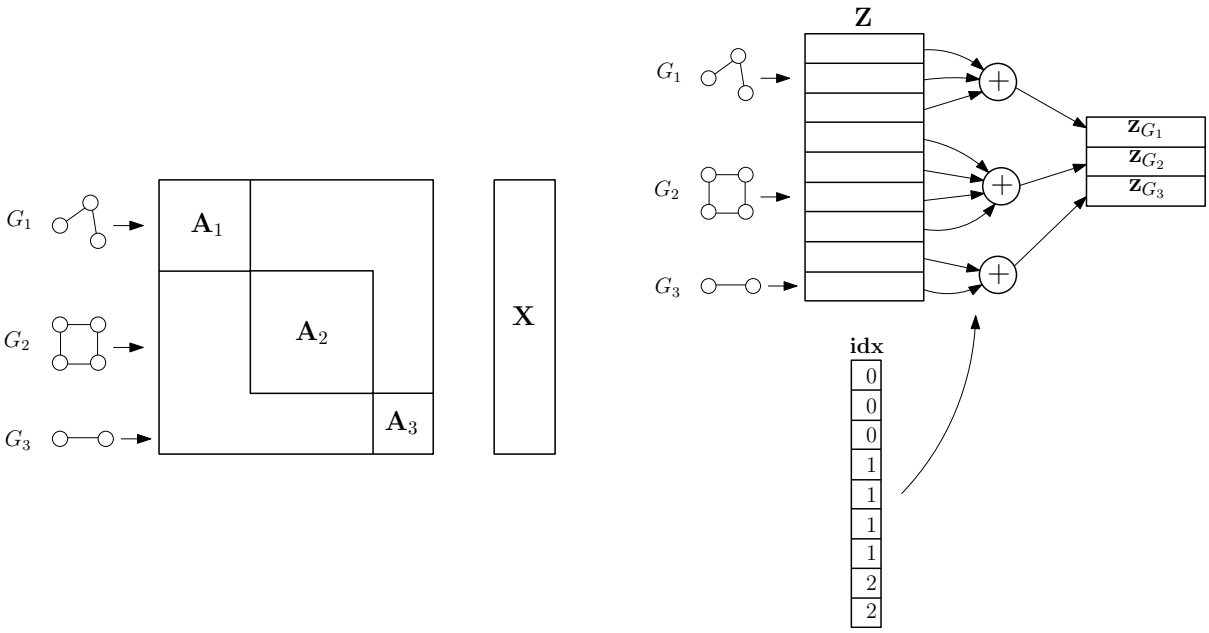

Figure 3: Implementation details of the graph neural network.

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 3 (Right).

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

Task 8

– 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

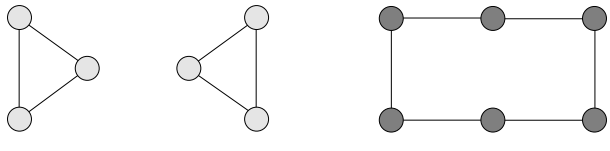

Figure 4: Two 2-regular graphs consisting of 6 nodes.

Question 3 (5 points)
Let the matrix $\mathbf{Z}$ shown in Figure 3 be defined as follows:

$$\mathbf{Z} = \begin{bmatrix} 0.77 & -1.26 & -0.63 \\ 1.15 & -1.90 & -0.94 \\ 0.77 & -1.26 & -0.63 \\ 1.15 & -1.90 & -0.94 \\ 1.15 & -1.90 & -0.94 \\ 1.15 & -1.90 & -0.94 \\ 1.15 & -1.90 & -0.94 \\ 0.77 & -1.26 & -0.63 \\ 0.77 & -1.26 & -0.63 \end{bmatrix}$$

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.

Question 4 (5 points)
Suppose we feed the two graphs G_1 and G_2 shown in Figure 4 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] Thomas N Kipf and Max Welling. Variational Graph Auto-Encoders. *arXiv preprint arXiv:1611.07308*, 2016.

[2] Guillaume Salha, Stratis Limnios, Romain Hennequin, Viet-Anh Tran, and Michalis Vazirgiannis. Gravity-Inspired Graph Autoencodersfor Directed Link Prediction. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 589–598, 2019.

[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.