ADVANCED LEARNING FOR TEXT AND GRAPH DATA

Lab session 5: Deep Learning for Graphs (1/2)

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

Tuesday, December 15, 2020

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 December 22, 2020 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 model that generates node embeddings 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 and you will evaluate it in a node classification task. We will use Python 3.6, and the following three libraries: (1) Gensim (`https://radimrehurek.com/gensim/`), (2) PyTorch (`https://pytorch.org/`), and (3) NetworkX (`http://networkx.github.io/`).

2 Node Embeddings

In this part of the lab, we will implement the *DeepWalk* algorithm, and use it to visualize some data and to perform node classification.

2.1 Implementation of DeepWalk

DeepWalk is a recently-proposed algorithm for generating node embeddings [2]. The algorithm capitalizes on recent advances in unsupervised feature learning which have proven very successful in natural language processing. DeepWalk learns representations of a graph's vertices by running short random

walks. These representations capture neighborhood similarity and community membership. The employed model is analogous to Skipgram: given a vertex v_i, it estimates the likelihood of observing the previous and the following vertices visited in the random walk. Specifically, DeepWalk solves the following optimization problem:

$$\underset{\phi}{\text{minimize}} \quad -\log \prod_{\substack{j=i-w \\ j\neq i}}^{i+w} P(v_j|\phi(v_i))$$

where $\phi : V \to \mathbb{R}^d$ is a function that maps vertices to their embeddings, and w is the size of the sliding window. Hence, given the embedding of a vertex, DeepWalk maximizes the probability of its neighbors in the walk. To make learning efficient both in terms of running time and memory, DeepWalk uses the Hierarchical Softmax to approximate the probability distribution.

Given a graph $G = (V, E)$ and a starting vertex v_i, a random walk of length t is a stochastic process with random variables $w_{v_i}^{(1)}, w_{v_i}^{(2)}, \ldots, w_{v_i}^{(t)}$ such that $w_{v_i}^{(1)} = v_i$ and $w_{v_i}^{(j)}$ is a vertex chosen uniformly at random from the neighbors of vertex $w_{v_i}^{(j-1)}$. In other words, a random walk is a sequence of vertices: we first select a neighbor of v_i at random, and move to this neighbor. Then, we select a neighbor of this new vertex at random, and move to it, etc.

Task 1
Fill in the body of the `random_walk()` function in the `deepwalk.py` file which, given a graph and a starting vertex, performs a random walk of specific length, and returns a list of the visited vertices (Hint: note that G is a NexworkX graph. Therefore, you can use the function `list(G.neighbors(v))` to obtain the neighbors of a vertex v.)

The DeepWalk algorithm does not start only a single walk from each node, but several of them. Then, these walks can be thought of as short sentences in some special language, and can be passed on to the Skipgram model.

Task 2
Fill in the body of the `generate_walks()` function in the `deepwalk.py` file which runs a number of random walks from each node of a graph, and returns a list of all the simulated random walks (i.e., list of lists) (Hint: use the function `list(G.nodes())` to get the list of nodes of graph G).

2.2 Visualization of the French Web

We will now apply the DeepWalk algorithm to a sample of the French web graph. The nodes of the graph represent web pages and directed edges represent hyperlinks between them. The graph is stored in the `french_web_sample.edgelist` file as an edge list:

```
http://www.adsrvr.org   https://www.engel-gregory-avocat.fr
http://www.adsrvr.org   https://www.dominique-abraham.com
https://www.agence-biomedecine.fr   https://www.dondespermatozoides.fr
https://www.agence-biomedecine.fr   https://www.dondemoelleosseuse.fr
https://www.bpi.fr   http://www.brgm.fr
https://www.bpi.fr   https://www.editionspixnlove.com
...
```

For the purposes of this lab, we will ignore edge directions. The network contains $33,226$ nodes and $354,529$ undirected edges in total.

Task 3
In the `visualization.py` file, use the `deepwalk()` function that you have already implemented to embed the nodes of the web graph into the 128-dimensional space. Start 10 walks from each node and set the length of each walk equal to 20.

To investigate if the learned representations are meaningful, we will focus only on a small number of web pages. We will project the embeddings of these webpages into the two-dimensional space and visualize them.

Task 4
Fill in the body of the `visualize()` function in the `visualization.py` file to visualize the representations of the 100 nodes that appear most frequently in the generated walks (Hint: you can obtain these nodes from the the `model.wv.index2entity` list). Then, store the representations of these 100 nodes in a 100×128 matrix (Hint: you can obtain the embedding of a node with `model.wv[node]`).

Question 1 (5 points)
Let G be a graph that consists of 100 nodes and 2 connected components. Each connected component is an instance of a $G_{n,p}$ Erdős-Rényi random graph where $n = 50$ and $p = 0.1$. Suppose you start 10 walks of length 20 from each node, and you use the DeepWalk algorithm to directly embed the 100 nodes in the 2-dimensional space. How do you expect the distances of the embeddings of the nodes within connected components and the embeddings of the nodes in different connected components to compare?

2.3 Node Classification

The experiments for this 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 5
Use the `draw_networkx()` function of NetworkX to visualize the karate network. Assign some color to the nodes that belong to the first class, and another color to nodes that belong to the second class.

Then, we will use again the DeepWalk algorithm to embed the nodes of the karate network in some vector space.

Task 6
Use the `deepwalk()` function that you have already implemented to generate 128-dimensional node representations. Start 10 walks from each node and set the length of each walk equal to 20.

We will next evaluate the quality of the generated embeddings in the node classification task. The data is split into a training and a test set ($80\% - 20\%$ split). Note that this is a typical supervised learning setting where we will use the training data to learn a model, and we will then evaluate the model on the test data.

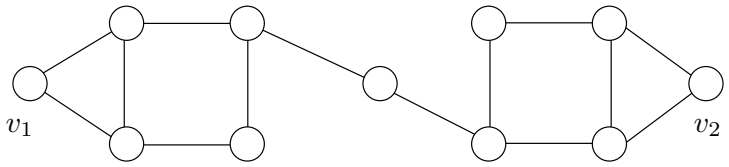

Figure 1: A synthetic graph. Nodes v_1 and v_2 are structurally identical.

Task 7
Train a logistic regression classifier on the training data. Then, use the classifier to make predictions for the test data (Hint: use the scikit-learn's implementation of the logistic regression classifier). Compute the classification accuracy (Hint: use the `accuracy_sore()` function of scikit-learn).

We will also compare the embeddings generated by the DeepWalk algorithm against those of a baseline method which utilizes the eigenvectors of the Laplacian matrix associated with the smallest eigenvalues.

Task 8
Compare (in terms of the classification accuracy) the DeepWalk embeddings against those that emerge from the spectral decomposition of the symmetric normalized graph Laplacian (Hint: use the `SpectralEmbedding` class of scikit-learn to produce these embeddings. Note that you can use the `to_numpy_matrix()` function of NetworkX to obtain the affinity/adjacency matrix).

Question 2 (5 points)
Nodes v_1 and v_2 of the graph that is shown in Figure 1 are structurally identical. Will the DeepWalk algorithm map these two nodes close to each other in the embedding space and why?

3 Graph Neural Networks

In the second part of the lab, we will focus on the problem of supervised node classification. We will implement a graph neural network (GNN) for learning node representations and performing node classification [1, 4, 3]. GNNs follow a recursive neighborhood aggregation (or message passing) scheme, where each node aggregates feature vectors of its neighbors to compute its new feature vector. After k iterations of aggregation, a node is represented by its transformed feature vector, which captures the structural information within the node's k-hop neighborhood.

3.1 Implementation of Graph Neural Network

In node classification, we are given the class labels of some nodes, and the goal is to predict the class labels of the nodes of the test set using information from both the graph structure and the attributes of the nodes. You will write your code in the `gnn.py` python script.

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

$$\hat{\mathbf{A}} = \tilde{\mathbf{D}}^{-\frac{1}{2}} \, \tilde{\mathbf{A}} \, \tilde{\mathbf{D}}^{-\frac{1}{2}}$$

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 9
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.

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

$$\mathbf{Z}^0 = f(\hat{\mathbf{A}}\ \mathbf{X}\ \mathbf{W}^0)$$

where $\mathbf{W}^0$ is a matrix of trainable weights and f is an activation function (e.g., ReLU, sigmoid, tanh). Clearly, the new feature vector of each node is the sum of the feature vectors of its neighbors. The second layer of the model is again a message passing layer:

$$\mathbf{Z}^1 = f(\hat{\mathbf{A}}\ \mathbf{Z}^0\ \mathbf{W}^1)$$

where $\mathbf{W}^1$ is a second matrix of trainable weights and f is an activation function. 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:

$$\hat{\mathbf{Y}} = \text{softmax}(\mathbf{Z}^1\ \mathbf{W}^2)$$

where $\mathbf{W}^2$ is a third matrix of trainable weights. Note that for clarity of presentation we have omitted biases.

We next discuss some practical implementation details. Let $\mathbf{H} = f(\hat{\mathbf{A}}\ \mathbf{Z}\ \mathbf{W}^0)$ be a message passing layer. Clearly, to compute $\mathbf{H}$, we need to perform two matrix-matrix multiplications. Since $\mathbf{W}$ corresponds to a matrix of trainable parameters, the first matrix-matrix multiplication (between $\mathbf{Z}$ and $\mathbf{W}$) can be defined as a fully-connected layer of the network. Then, we can multiply $\hat{\mathbf{A}}$ with the output of the above operation and apply the nonlinear function to compute $\mathbf{H}$.

Task 10
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
– a fully-connected layer with n_{class} units (i.e., $\mathbf{W}^2 \in \mathbb{R}^{h_2 \times n_{class}}$) followed by the softmax 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).

Question 3 (5 points)
Let G be the C_4 cycle, i.e., a graph on 4 nodes containing a single cycle through all nodes. Let also $\mathbf{X}$ be the 4×1 all-ones matrix, while $\mathbf{W}^1$ and $\mathbf{W}^2$ are defined as follows:

$$\mathbf{W}^1 = \begin{bmatrix} 0.8 & -0.5 \end{bmatrix} \quad \mathbf{W}^2 = \begin{bmatrix} 0.9 & -1.2 & 0.4 \\ 1.4 & -0.3 & -0.2 \end{bmatrix}$$

Assume that there are no biases and that f is the ReLU function. Compute (showing your calculations) matrix $\mathbf{Z}^1$. The rows of this matrix correspond to the representations of the 4 nodes. What do you observe?

3.2 Node Classification

We will first evaluate the GNN you implemented in a node classification task. Specifically, we will again experiment with the karate dataset. To make it possible to compare the performance of the GNN against that of the DeepWalk algorithm, we use the exact same training/test split as before.

Task 11
Run the `gnn_karate.py` script to train the model, make predictions, and compute the classification accuracy.

Note that the matrix of features is the $n \times n$ identity matrix (line 52 in the `gnn_karate.py` script). Therefore, each node is assigned a unique feature vector.

Task 12
Modify the features of the nodes. Initialize all of them to the same value (e.g., you can set them all equal to 1). Re-train and re-evaluate the neural network.

Question 4 (5 points)
Is the performance comparable to the previous one? Why?

3.3 Visualization of Node Representations

For this experiment, we will use the *Cora dataset*[1], a well-known citation network dataset. The dataset contains a number of Machine Learning papers divided into one of 7 classes (e.g., Genetic Algorithms, Neural Networks). The dataset is represented as a node-attributed graph. Nodes of the graph represent papers, and there is a directed edge from node i to node j if paper i cites paper j. There are $2,708$ nodes and $5,429$ edges in total. Each node is assigned a feature vector that corresponds to the bag-of-words representation of its textual content. The size of the vocabulary is $1,433$. Hence, each node is represented by a vector of dimensionality $1,433$.

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 that correspond to nodes of the test set will be projected into the two-dimensional space and visualized.

[1] `https://relational.fit.cvut.cz/dataset/CORA`

Task 13

Modify the code in the `models.py` file such that the model also returns the output of the second message passing layer. Then, in the `gnn_cora.py` script, use slicing to retrieve the representations of only the nodes of the test set. Transform the Torch tensor into a NumPy matrix (Hint: for a tensor `T`, you can do this using `T.detach().cpu().numpy()`). Project these representations to two dimensions using t-SNE (Hint: use scikit-learn's implementation).

References

[1] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning*, pages 1263–1272, 2017.

[2] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 701–710, 2014.

[3] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[4] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. In *In Proceedings of the 32nd AAAI Conference on Artificial Intelligence*, pages 4438–4445, 2018.