ADVANCED LEARNING FOR TEXT AND GRAPH DATA

Lab session 7: Deep Learning on graphs

Lecture: Prof. Michalis Vazirgiannis
Lab: Giannis Nikolentzos and Stratis Limnios

January 8, 2019

1 Introduction

The goal of this lab is to help you become familiar with neural networks that operate on graphs and which can be employed for addressing various tasks such as node classification, graph classification and link prediction. The lab will also allow you to work with graph data using the NexworkX library of Python (`http://networkx.github.io/`), a very popular library for the analysis and manipulation of graphs. We will first implement a neural network that generates node embeddings. These embeddings are unsupervised, in the sense that they can be fed to conventional machine learning algorithms for solving any downstream task. Next, we will implement a graph neural network. Both architectures will be evaluated in node classification.

2 Node Embeddings

In this part of the lab, we will generate embeddings for the nodes of a graph using a simple baseline method (e.g., spectral embeddings) and a more sophisticated method that is inspired by recent advances in natural language processing and which utilizes a probabilistic neural network. We will then evaluate the generated embeddings on a *multi-label classification* task.

In contrast to binary and multi-class classification, in the multi-label classification setting, there is no constraint on how many of the classes each instance can be assigned to. Hence, given a finite set of classes $\mathcal{C}$, every node can be assigned to one or more classes of the set $\mathcal{C}$. Multi-label classification is a challenging task especially if the set of classes is large.

Our dataset is a protein-protein (PPI) interaction network, that is, a graph that represents the presence or absence of physical interactions between proteins. More specifically, the dataset we will utilize is a subgraph of the PPI network for Homo Sapiens. The subgraph corresponds to the graph induced by nodes for which labels from the hallmark gene sets were available and represent biological states. The network contains $3,890$ nodes, $76,584$ edges, and 50 different labels. The graph is stored in the `Homo_sapiens.mat` file[1].

1. We will first implement a very popular and simple baseline method for generating node embeddings. The method is based on matrix factorization. Specifically, we will use the eigenvectors corresponding to the d smallest eigenvalues of the Laplacian or the normalized Laplacian

[1]The graph can be downloaded from the following link: https://snap.stanford.edu/node2vec/Homo_sapiens.mat.

matrix of the graph to generate feature vector representations for its nodes. Algorithm 1 illustrates the pseudocode for generating such spectral embeddings.

Algorithm 1 Spectral Embeddings

Require: Graph $G = (V, E)$ and parameter d

Ensure: Matrix $\mathbf{U} \in \mathbb{R}^{n \times d}$ (i.e., embedding of each node of the graph)

1: Let $\mathbf{A}$ be the adjacency matrix of the graph

2: Compute the Laplacian matrix $\mathbf{L} = \mathbf{D} - \mathbf{A}$ or the normalized Laplacian $\mathbf{L}_{sym} = \mathbf{I} - \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$. Matrix $\mathbf{D}$ corresponds to the diagonal degree matrix of graph G (i.e., degree of each node in the main diagonal), while matrix $\mathbf{I}$ is the identity matrix.

3: Apply eigenvalue decomposition to the Laplacian matrix $\mathbf{L}$ or the normalized Laplacian $\mathbf{L}_{sym}$ and compute the eigenvectors that correspond to the d smallest eigenvalues. Let $\mathbf{U} = [\mathbf{u}_1|\mathbf{u}_2|\ldots|\mathbf{u}_d] \in \mathbb{R}^{n \times d}$ be the matrix containing these eigenvectors as columns. The i-th row of $\mathbf{U}$ corresponds to the embedding of node v_i.

Next, we will use the above method to generate embeddings for the nodes of the PPI network. Fill in the body of the function `generate_spectral_embeddings()` which can be found in the `spectral_embedding.py` python script as shown below.

```
def generate_spectral_embeddings(A, d):

  # Add the body of the function here

  return U
```

Run the `spectral_embedding.py` python script to generate node embeddings and to create a file that contains these embeddings.

2. We will next implement *DeepWalk*, a more sophisticated algorithm for generating node embeddings [Perozzi et al., 2014]. DeepWalk builds 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:

$$\underset{\phi}{\text{minimize}} \quad -\log \prod_{\substack{j=i-w \\ j \neq i}}^{i+w} P(v_j|\phi(v_i)) \tag{1}$$

where $\phi : V \rightarrow \mathbb{R}^d$ 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.

You will implement a function that given a graph and a starting vertex performs a random walk of specific length, and returns a list of the visited vertices. Fill in the body of the function `random_walk()` which can be found in the `deepwalk.py` python script as shown below. Note that G is a NexworkX graph. Use the function `G.neighbors(`v`)` to get the neighbors of a vertex v.

```
def random_walk(G, node, walk_length):

  # Add the body of the function here

  walk = [str(node) for node in walk]
  return walk
```

Next, you will implement a function that given a graph, it runs a number of random walks from each node of the graph, and returns a list of all random walks. Fill in the body of the function `generate_walks()` which can be found in the `deepwalk.py` python script as shown below. Note that the function `G.nodes()` returns a list containing all the nodes of graph G.

```
def generate_walks(G, num_walks, walk_length):

  # Add the body of the function here

  return walks
```

After implementing the two functions, run the `deepwalk.py` python script to learn representations of the vertices of the PPI graph, and to create a file that contains these embeddings.

3. To evaluate the two methods, we will perform multi-label classification using the `evaluate.py` python script that you are given. The script randomly samples 90% of the nodes, and uses them as training data, while the rest of the nodes are used as test data. This process is repeated 5 times, and the micro and macro-average F1-scores of each iteration are displayed. Use the `evaluate.py` script to evaluate the embeddings generated by both methods. What do you observe? Regenerate node embeddings using different values for the hyperparameters (e.g., dimensionality of embeddings, walk length, window size, number of walks per node). Which of these hyperparameters have a large impact on the obtained performance?

3 Graph Neural Networks for Node Classification

In the second part of the lab, we will focus on the problem of node classification. We will implement a graph neural network (GNN) for learning node representations and performing node classification. 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.

For our experiments, we will use the *Cora dataset*[2], 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

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

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 $1,433$.

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.

1. 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. Apply the above normalization trick using operations of the NumPy library.

2. We next present the GNN model that we will implement. Let $\hat{\mathbf{A}}$ be the normalized adjacency matrix of the Cora graph, and $\mathbf{X}$ a matrix whose i^{th} row contains the feature vector of node i. We will implement a three-layer model. 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 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:

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

   where $\mathbf{W}^2$ is is a matrix of trainable weights.

   To learn the parameter weights, the GNN minimizes the following loss function:

$$\mathcal{L} = -\sum_{i \in I} \sum_{j=1}^{|\mathcal{C}|} \mathbf{Y}_{ij} \log \hat{\mathbf{Y}}_{ij}$$

   where I are the indices of the nodes of the training set, and $\mathcal{C}$ is the set of class labels.

   We next discuss some practical implementation details. To implement the model that is presented above, we will use the functional API[3] of Keras. The layers in the model are connected pairwise. Hence, each layer takes as input either the input data or the output of another layer. The functional API requires defining a standalone Input layer that specifies the shape of input data (already in the `gnn.py` file.). Implement the rest of the architecture in the `gnn.py` file. More specifically, add the following layers:

[3] https://keras.io/getting-started/functional-api-guide/

- a dropout layer with with 50% ratio of dropped outputs.
- a message passing layer with 32 hidden units (i.e., $\mathbf{W}^0 \in \mathbb{R}^{1,433 \times 32}$) followed by a ReLU activation function
- a dropout layer with with 50% ratio of dropped outputs.
- a message passing layer with 16 hidden units (i.e., $\mathbf{W}^1 \in \mathbb{R}^{32 \times 16}$) followed by a ReLU activation function
- a fully-connected layer with 7 units (i.e., $\mathbf{W}^2 \in \mathbb{R}^{16 \times 7}$) followed by the softmax activation function.

Note that a message passing layer is defined as shown below.

```
H = MessagePassing(units, activation)([H_prev]+G)
```

After defining all the layers and connecting them together, create the model using the Model class of Keras. Specifically, it requires that you only specify the input and output layers as follows:

```
model = Model(X_in, Y)
```

After creating the model, you can compile it and specify the loss function (categorical crossentropy) and the optimization algorithm (Adam with initial learning rate 0.01) as shown below.

```
model.compile(loss='categorical_crossentropy', optimizer=Adam(lr=0.01))
```

Run the code to train the model and to make predictions for the nodes of the test set. What do you observe? Is the performance high?

3. We will next visualize the hidden representations of the nodes of the test set. Create a new model that has the same input as the previous one, but its output is the second message passing layer. Use the `predict` function of the model to obtain the 16-dimensional representations of all the nodes and use slicing to retrieve the representations of only the nodes of the test set. Project these representations to two dimensions using t-SNE (scikit-learn provides an implementation of the algorithm). Run the code to visualize the representations. What do you observe?

References

Shaosheng Cao, Wei Lu, and Qiongkai Xu. GraRep: Learning Graph Representations with Global Structural Information. In *Proceedings of the 24th International on Conference on Information and Knowledge Management*, pages 891–900, 2015.

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.

Aditya Grover and Jure Leskovec. node2vec: Scalable Feature Learning for Networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 855–864, 2016.

Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems*, pages 1025–1035, 2017.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

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.

Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. LINE: Large-scale Information Network Embedding. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1067–1077, 2015.

Daixin Wang, Peng Cui, and Wenwu Zhu. Structural Deep Network Embedding. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1225–1234, 2016.

Zhilin Yang, William W Cohen, and Ruslan Salakhutdinov. Revisiting Semi-Supervised Learning with Graph Embeddings. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning*, pages 40–48, 2016.