清華大學

Tsinghua University

Department of Computer Science and Technology

Machine Learning

Homework 2

Sahand Sabour

2020280401

1 DeepWalk

1.1 Motivation

Sparsity of a network representation is considered as a significant disadvantage when using statistical models; yet, coping with sparsity is a necessity when it comes to real-life machine learning applications. Hence, one can utilize sparsity to create efficient discrete algorithms. Due to this fact and the huge success of deep learning in other machine learning fields, especially natural language processing, the authors were motivated to utilize those techniques and implement them in a network analysis setting by producing DeepWalk.

1.2 Methodology

The aim of DeepWalk is to take a graph as an input and discover its latent representations through a number of short truncated random walks. This algorithm consists of two main components: random walk generator and update step.
The generator takes a graph G as input and samples a vector v_i as the root of walk W_{vi} uniformly. Accordingly, each walk samples uniformly from the neighbors of the last visited vertex until an arbitrary fixed amount of walks t is reached. In other words, the algorithm for this task consists of two loops: the inner and outer loops. Once a vertex is selected, the outer loop indicates how many times a walk should start at this vertex. Consequently, the inner loop, all of the vertices of the graph are visited, a random walk is generated for each vertex, and then this walk is used to update the representations using the SkipGram language model.

1.3 Experiments

This model is evaluated on three data-sets: BlogCatalog, Flickr, and YouTube. The authors demonstrate five of previous approaches as their baselines to highlight the improvements achieved by DeepWalk: SpectralClustering, Modularity, EdgeCluster, wvRN, and Majority. For evaluation, they randomly sample a number of nodes for training and use the rest for testing; this process was repeated ten times. Accordingly, their obtained results outperforms all the identified baselines for the task of multi-label classification through all the studied datasets.

1.4 Personal Opinion

DeepWalk builds upon a simple yet clever idea. The paper does a fantastic job for explaining the terminology to the point that it becomes completely comprehensible by beginners in the field. The methodology is explained fairly well and the performance of this algorithm is quite significant.

2 Graph Convolutional Networks (GCN)

2.1 Motivation

The formulation of graph regularization in previous work was based on the assumption that connected nodes within the graph share the same label. In those methods, the total loss was calculated as the sum of the supervised loss ($\mathcal{L}_0$) and the graph-based regularization loss ($\mathcal{L}_{reg}$). However, the authors believed that by using this assumption, the modeling capacity would be restricted as nodes could contain much more useful information that encoded similarity between nodes. Hence, using this statement as their motivation, the authors aimed to create a graph structure encoding based on neural networks that excludes the graph-based regularization loss ($\mathcal{L}_{reg}$) from the total loss ($\mathcal{L}$).

2.2 Methodology

This paper presents utilizes fast approximate convolutions for their graph-based neural network model. Initially, the multi-layer GCN's layer-wise propagation rule was assumed as follows:

$$H^{(l+1)} = \sigma(D^{\frac{-1}{2}}AD^{\frac{-1}{2}}H^lW^l) \tag{1}$$

where A is the adjacency matrix, D is the sum of all adjacency matrices, W is the trainable weight, and σ is the activation function. Accordingly, they used scalar graph convolutions and stacked a number of these convolutional layers on top of each other, which has the benefit of efficient convolutional filter functions, yet dies not limit explicit parameterization. Accordingly, the created convolutional neural network is used to perform the task of node classification.

2.3 Experiments

GCN is also evaluated on three datasets: Citeseer, Cora and Pubmed. There are six baselines for the experiment result comparison: ManiReg, SemiEmb, LP, DeepWalk, ICA, and Planetoid. For evaluation, they used a two-layer GCN and provide the obtained accuracy for 1000 labeled test samples. Additionally, the the model is compared to its baselines based on the training time per epoch while the proposed propagation model is also evaluated. According to the provided results, GCN was able to outperform all of the mentioned baselines for the task of node classification across all the studied datasets.

2.4 Personal Opinion

The paper demonstrates a vague and complex explanation of their model. Hence, although creative, I personally did not enjoy nor understand the majority of this

paper after the first time reading it. The problem formulation could be simplified, more explanations could be provided, and the structure of the paper could also be improved; yet, there is no denying that the proposed method is novel and achieves outstanding results.

2.5 Comparison with DeepWalk

Similar to DeepWalk, GCN's motivation also derives from the issue of sparsity for labeled nodes within a vast network. One of the main differences between the two approaches is that DeepWalk is fully unsupervised while GCN is semi-supervised. DeepWalk builds itself on simple concepts such as random walks while GCN is built upon rather complex mathematical modifications and convolutional neural networks. Hence, although GCN may give more accurate results in the task of node classification, its complexity in formulation and implementation would be a huge disadvantage when compared to DeepWalk.

3 CogDL

3.1 Testing CogDL Models

In this assignment, we are asked to run two models on two different datasets by running the provided scripts. The first script is as follows: python scripts/train.py –task unsupervised_node_classification – dataset wikipedia –model deepwalk; by running this script, we would be training and testing the DeepWalk model on the Wikipedia dataset for the task of node classification. The results of this process is provided in the figure below (Figure 1).

```
(cogdl) ➜ cogdl git:(master) python scripts/train.py --task unsupervised_node_classification --dataset wikipedia --model deepwalk
Failed to import Deep Graph Library (DGL)
Namespace(cpu=False, dataset=['wikipedia'], device_id=[0], enhance=None, hidden_size=128, iteration=10, lr=0.01, max_epoch=500, model=
['deepwalk'], num_shuffle=5, patience=100, save_dir='.', seed=[1], task='unsupervised_node_classification', walk_length=80, walk_num=4
0, weight_decay=0.0005, window_size=5, worker=10)
Downloading http://snap.stanford.edu/node2vec/POS.mat
Processing...
Done!
node number: 4777
generating random walks...
100%|                                        | 40/40 [00:59<00:00,  1.48s/it]
training word2vec...
| Variant                | Micro-F1 0.1   | Micro-F1 0.3   | Micro-F1 0.5   | Micro-F1 0.7   | Micro-F1 0.9   |
|------------------------|----------------|----------------|----------------|----------------|----------------|
| ('wikipedia', 'deepwalk') | 0.4253±0.0000  | 0.4720±0.0000  | 0.4951±0.0000  | 0.5056±0.0000  | 0.5214±0.0000  |
```

Figure 1: Result of running the first script

Accordingly, the second script (python scripts/train.py –task node_classification –dataset citeseer – model gcn) analyzes the same task but for training and testing

the GCN model on the citeseer dataset. The obtained results are demonstrated in the below figure (Figure 2).

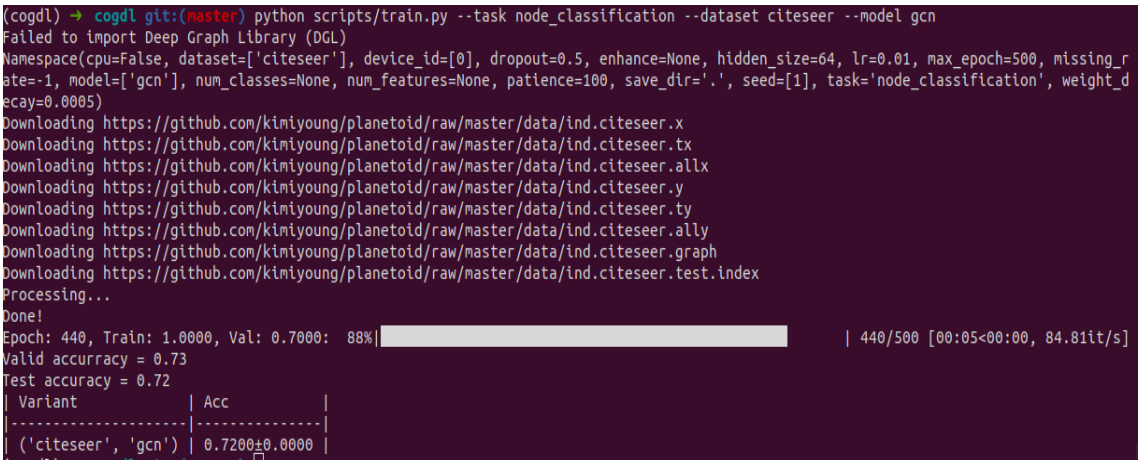

Figure 2: Result of running the second script

3.2 API Design

CogDL is a graph representation learning toolkit, based on PyTorch. Similar to what is demonstrated in the two scripts of the previous section, the main entry point of the model training is train.py located in the scripts folder. Initially, the arguments provided by the user are evaluated: first, it checks whether the provided arguments are valid; second, it checks whether the provided model matches the input task and dataset (based on match.yml file); third, the model implementation itself checks whether user provides all the required arguments.After the validity of the provided arguments is evaluated, CogDL checks whether the requested dataset is available offline. If this dataset has yet to be downloaded, the download script is called to download the corresponding datasets.

CogDL's API is fairly easy to use. First, setting the hyper-parameters for a specific model can be achieved by modifying the keys of 'args' variable, which is globally used in machine learning frameworks. For instance, for the task of graph classification, on the proteins dataset with the hgpsl model, we would have

$$\text{args} = \text{default\_args}()$$
$$\text{args.task} = \text{'graph\_classification'}$$
$$\text{args.dataset} = \text{'proteins'}$$
$$\text{args.model} = \text{'hgpsl'}$$

In addition, the model and its dataset can be built via the following lines:

$$\text{dataset} = \text{build\_dataset(args)}$$
$$\text{model} = \text{build\_model(args)}$$

Consequently, we can create the task using the model and dataset and train it via

task = build_task(args, dataset=dataset, model=model)
ret = task.train()

Each of the implemented models reside in the cogdl/nn folder. The structure of the models are somewhat similar: a sub-class of base model which has an 'add_args' function to initialize its arguments, 'build_model_from_args' function that builds the model based on those arguments, an init function to initialize and define the different layers and variables of the model structure, a forward function to demonstrate the feed forward operation of the model and its relative calculations, and occasionally, a loss and a predict function whose roles are believed to be self-explanatory. There are also different tasks and datasets within the cogdl folder. The task scripts are used to train, test, and evaluate models relative to the task while the dataset scripts download and process the corresponding dataset.
Each model also has an example of its usage, similar to what was previously explained, in the examples folder. Accordingly, the unit tests for both the models and datasets are provided in tests folder. The model unit tests dedicate two functions to declaring the required args for the model and evaluating the model respectively. Accordingly, the unit tests are tested via Travis once they are added to GitHub repository.

3.3 GNN Model Implementation

In this assignment, I implemented the Hierarchical Graph Pooling with Structure Learning model [1]. The pull request id for this implementation is #80 and it has been merged at the time of writing this report.

3.4 Suggestions

There are a number of suggestions that I believe would improve the CogDL experience, both as a user and as a contributor:

1. Create a unified template for all the models for a certain tasks; this allows users to clearly differentiate between different models and provides contributors with a clear path to implement their models.

2. Include more information on how the evaluation results in the README.md file have been obtained. For instance, for tasks such as graph classification, there aren't any mentions of using k-folds or other arguments to make the models work on certain dataset; yet, these arguments are highly necessary.

3. Make a list of content for the README. The current version of the README file looks rather complex, long, and hard to look through. Providing an interactive list of content would allow the users to focus on the topics of each section and find the section they are looking for much better and faster.

3.5 Contributions

As my contribution to this toolkit, I have made an interactive CLI to make the necessary files for a new model (check figure below).

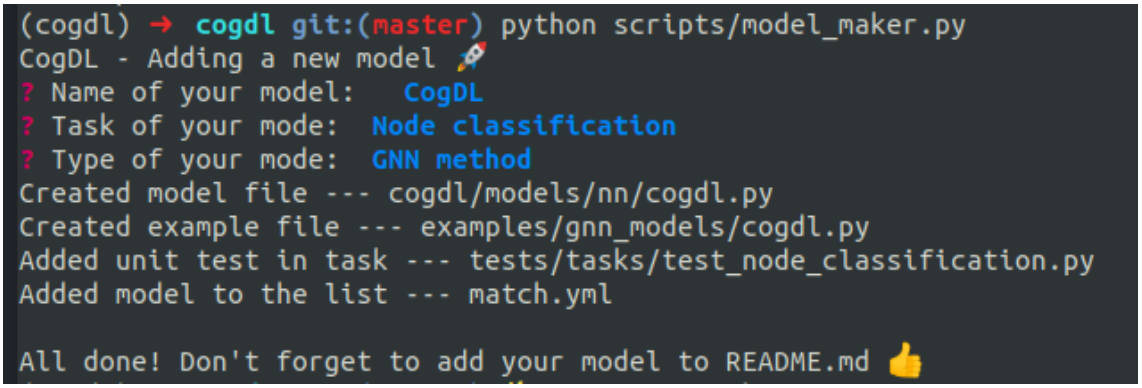

Figure 3: Model maker CLI

As shown in the figure, the script can be accessed by running 'python scripts/model_maker.py'. The script then asks the user several questions to have the necessary information for creating a new file. Accordingly, the script does the following tasks based on user inputs:

1. Create a model file in the nn folder based on a predefined template.
2. Create an example file in the corresponding folder of the model's type, according to a predefined template.
3. Add the unit test for this model to the corresponding task.
4. Add the model to the list of available models in math.yml.
5. Remind the user to add their own model to README.md since the CLI is yet unavailable to detect the correct location to add the model. In addition, since the model is yet to be defined, its evaluation is not possible.

The pull request id for this contribution is #86 and it has been merged at the time of writing this report.

References

[1] Zhen Zhang et al. "Hierarchical graph pooling with structure learning". In: arXiv(2019) arXiv:1911.05954