ML Homework #2 - CogDL

Yoke Kai Wen, 2020280598

November 29, 2020

1 DeepWalk paper summary

DeepWalk is a classic graph embedding algorithm. It uses random walks and borrows ideas from language modelling to learn vector representations of each graph node, which can then be exploited by statistical models for tasks such as node classification. Given input parameters: `window size w, embedding size d, walks per node` γ`, walk length t`, DeepWalk first generates a corpus of γ random walks per node, each walk of length `t`, sampled in a uniform manner. The corpus is analogous to a text corpus, with each random walk sequence akin to a sentence, and each node akin to a word in the text vocabulary. Then, a vector representation of each node v_i of dimension `d` is learnt by observing the nodes that occur within `w` steps to the left and right of v_i in all the random walk sequences of the corpus through solving an optimisation problem via the SkipGram neural network. Using a combination of truncated random walks and neural language models in DeepWalk offers several advantages: (1) low-dimensional representations can be generated (2) in a continuous vector space; (3) nodes with similar neighbourhoods acquire similar vector representations; (4) easy adaptation to changing network topology by updating learnt representations only in local neighbourhoods affected by the change; (5) parallelisability as several random walkers can simultaneously explore different parts of the graph. Experiments show that DeepWalk performs particularly well when labelled data is sparse.

2 GCN paper summary

The Graph Convolutional Network (GCN) is a convolutional architecture that employs a layer-wise propagation rule which operates directly on the graph, and is based on a first-order approximation of spectral graph convolutions. It can be directly used in an end-to-end manner for the task of node classification with the graph node features $X \in \mathbb{R}^{\mathrm{NxD}}$ and structure in the form of adjacency matrix $A \in \mathbb{R}^{\mathrm{NxN}}$ as input. The layer-wise propagation rule is defined as: $H^{(l+1)} = \sigma(\tilde{D}^{-0.5}\tilde{A}\tilde{D}^{-0.5}H^{(l)}W^{(l)})$, with $\tilde{A}$ the adjacency matrix with added self-connections, $\tilde{D}$ the diagonal node degree matrix corresponding to $\tilde{A}$, $W^{(l)}$ a layer-specific trainable weight matrix, $\sigma(.)$ an activation function, $H^{(l)} \in \mathbb{R}^{\mathrm{NxD}}$ the output activation map of each layer, and $H^{(0)} = X$. Each layer effectively only involves the convolution of the central node with neighbours that are 1 step away from it. By stacking K layers each followed by a non-linear activation function, a neural network model $f(X, A)$ that involves graph convolutions within the K^{th} order neighbourhood is formed. This allows feature information to be propagated from neighbouring nodes in every layer thus improving classification performance. This is in contrast with spectral graph convolutions which merely conducts one convolution across the entire graph. Experiments show that GCN is able to encode both graph structure and node features in a useful way for node classification, and is also computationally efficient. Furthermore, GCN is flexible because it allows us to control the size of the convolution neighbourhoods by adjusting the depth of the model.

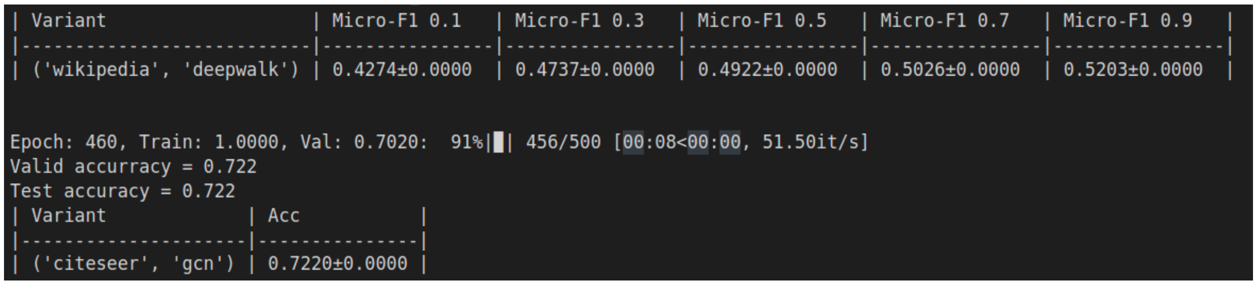
2.1 Comparison between DeepWalk and GCN

The key difference is that DeepWalk only learns the graph embedding, and has to be combined with other statistical models in a multi-step pipeline to do node classification, whereas GCN takes in the raw graph features and structure directly thus combining both graph embedding and node classification in an end-to-end manner. This makes parameter tuning more convenient in GCN, while in DeepWalk, the graph embedding step has to be optimised separately from the classification training step.

3 CogDL demo results

My fork of CogDL is here: https://github.com/kwyoke/cogdl. Figure 1 shows the results of running the following commands:

1. `python scripts/train.py --task unsupervised_node_classification --dataset wikipedia --model deepwalk`
2. `python scripts/train.py --task node_classification --dataset citeseer --model gcn --cpu`



```
| Variant                    | Micro-F1 0.1    | Micro-F1 0.3    | Micro-F1 0.5    | Micro-F1 0.7    | Micro-F1 0.9    |
|----------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| ('wikipedia', 'deepwalk')  | 0.4274±0.0000   | 0.4737±0.0000   | 0.4922±0.0000   | 0.5026±0.0000   | 0.5203±0.0000   |


Epoch: 460, Train: 1.0000, Val: 0.7020:  91%|█| 456/500 [00:08<00:00, 51.50it/s]
Valid accurracy = 0.722
Test accuracy = 0.722
| Variant              | Acc           |
|----------------------|---------------|
| ('citeseer', 'gcn')  | 0.7220±0.0000 |
```

Figure 1: Demo results of (top) (1) and (bottom) (2)

4 Summary about organisation and API design of code

The CogDL toolkit comprises six main modules: `data, datasets, layers, models, tasks, trainers`, and two helper scripts `options, utils`.

1. `cogdl.data` contains scripts for custom data downloading, extraction and formatting;
2. `cogdl.datasets` contains scripts for downloading several specific supported datasets;
3. `cogdl.layers` contains scripts for defining several GNN layer types;
4. `cogdl.models` contains two submodules: `emb, nn`, with `cogdl.models.emb` containing various models for graph embedding such as `cogdl.models.emb.deepwalk`, and `cogdl.models.nn` containing various types of GNN models such as `cogdl.models.nn.gcn`;

5. `cogdl.tasks` contains scripts for executing various graph related tasks such as `cogdl.tasks.node_-classification`;

6. `cogdl.trainers` contains scripts for graph model training.

`cogdl.options` contains functions for adding default arguments to parsers, and `cogdl.utils` contains graph-related helper functions that are commonly used such as adding self loops to adjacency matrix and normalisation.

5 Implementation of GDC-GCN model in CogDL

I implemented the Diffusion Improves Graph Learning paper, and my implementation was successfully merged into the official repository as `cogdl.models.nn.gdc_gcn`. My implementation of Graph Diffusion Convolution (GDC) allows users to choose between two filter variants used in conjunction with GCN: Personalised PageRank ('ppr') and the heat kernel ('heat'), or no filters at all ('none'), which would be equivalent to vanilla GCN. I based my implementation code on the code published by the original authors in their Github repository https://github.com/klicperajo/gdc.

5.1 Comparison of results with original results in paper

The paper found the inclusion of GDC improves node classification performance compared to just vanilla GCN. It was not clear what were the exact hyperparameters and architecture the paper used for its GCN, but it reports 2% increment in performance for the CORA dataset, and roughly 1% increment in performance for the CITESEER dataset, both when using either 'ppr' or 'heat', as seen in Figure 2.

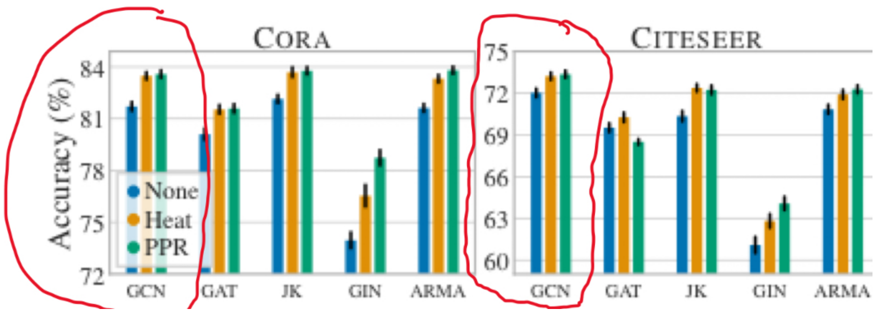


Figure 2: Original results in paper reported improvement in performance on CORA and CITESEER datasets when including 'ppr' or 'heat' GDC on vanilla GCN

Similarly, I tested my implementation on the CORA and CITESEER datasets to check if I am able to achieve similar improvement. The parameters I used were: `hidden_size=16, num_layers=2, alpha=0.05, t=5.0, k=128, eps=0.01('ppr')/ 0.0001('heat')`, with unspecified parameters being the default in `cogdl.options.get_parser`. My implementation results are shown in Table 1.

	Test accuracy		
	'none'	*'ppr'*	*'heat'*
CORA	0.821	0.821	0.816
CITESEER	0.715	0.715	0.725

Table 1: Table showing test accuracies of my implementation on CORA and CITESEER datasets

For the CORA dataset, my implementation does not see significant improvement with the 'ppr' variant, and in fact there was a decrease in accuracy with the 'heat' variant, and this is quite different from the original results reported by the paper. The results for the CITESEER dataset is more promising, with 'ppr' variant showing no significant improvement and 'heat' variant showing a 1% improvement. I suspect the difference in results could be due to my choice of hyperparameters, but in general it seems like adding GDC variants will generally improve or at least result in on-par performance with the vanilla version. Nevertheless, since GDC adds on a preprocessing step, it does incur more computational cost and is only worth doing if it has a significant improvement over GCN.

5.2 #ID and link of pull request

Pull request ID is #69, link is https://github.com/THUDM/cogdl/pull/69.

6 Suggestions on CogDL toolkit

6.1 Installation

During installation, I found it difficult to figure out what versions of pytorch and other libraries to use. Initially I thought I had to use a GPU (on a remote server) to run the codes efficiently, and it caused me lot of pain and trouble to figure out what CUDA version the remote server had, and consequently what were the respective versions I had to install. Particularly, my CUDA version was 10.0 and then somehow this option was absent in the pytorch installation page as it only had 9.2, 10.1 and 10.2 available. Eventually, I gave up on GPU and took the cpu version to run locally on my own laptop, and I realised CogDL can still run decently on cpu alone as long as the graph dataset is not too large. Therefore, I suggest a comprehensive tutorial on installation for various CUDA versions, and also a note that the cpu version is actually quite workable for the smaller datasets.

6.2 Adding own implementation and pull request

I found it particularly difficult to figure out how and where to add my own model implementation, handle all the different model arguments, and figure out what are all the scripts I need to modify and add in order to pass the pull request. Again, I suggest a more comprehensive tutorial to cover this.

6.3 Differentiation from pytorch-geometric

Many of the `utils` functions and also some of the models such as GCN already have respective implementations in pytorch-geometric, but CogDL reimplements them again, with some of their implementations also importing pytorch-geometric (although I see that many implementations avoid using pytorch-geometric). I do not understand the rationale behind this double work. If CogDL is trying to be different then shouldn't it not use pytorch-geometric at all?

7 Contribution to CogDL in other ways

I wrote a brief tutorial in the README.md on 'A brief guide to having a successful pull request (unit test)'.
Pull request ID is #71, link is https://github.com/THUDM/cogdl/pull/71.