Advanced Learning for Text and Graph Data: Data Challenge Report
Team name: Apex Legends

Ulrich GOUE[1,2*] , **Gabriel ROMON**[1,2] , **Mohamed Mehdi Loutfi**[1,2]
[1]MVA, Ens Cachan
[2]Ensae-ParisTech
ulrich.goue@ens-paris-saclay.fr, gabriel.romon@ensae.fr, mohamed.loutfi@ensae.fr

Abstract

In this challenge we were asked to predict continuous values associated with a graph. We were required to adopt an NLP perspective with a self-attention mechanism (Hierarchical Attention Networks or HAN). Indeed a graph can be seen like a document by sampling many random walks out of it. We proceeded in two steps: we first enrich our node embeddings or word embeddings using a simplified multi-layer graph convolutional network allowing us to have both node (word) and graph (doc) embeddings. Second we reduce our embeddings with classical PCA. At last we merge these node embeddings with the original ones and modify the initial HAN architecture to account for our learned documents embeddings.

1 Introduction

To tackle this challenge we drew inspiration from transfer learning which has been applied successfully in NLP and computer vision (with CNNs). For instance for image classification or object recognition, we usually use pre-trained models and tune their last layers to adapt for our specific task. This approach is also used in [1] [2] combined with SVD to enforce speedy calculations. The same idea holds in NLP, since pre-trained word embeddings are used as input to other models. So in our case we decide to enrich the node embeddings by training a simplified multi-layer graph convolutional network (MLGCN, hereafter), and then use it as input of the HAN architecture. Building on what was noticed in [1], matrix factorization is not harmful for the overall learning process, we also notice that the same holds for dimension reduction so that the use of simple PCA was able to improve our final classifier performance. The rest of this article is organized as follow: section 2 tackles the technical details of our MLGCN, section 3 outlines our model, section 4 describes all the data processing, section 5 discusses the results and section 6 concludes.

*Contact Author

2 Simplified MLGCN

2.1 Message Passing Step

Like the true GCN presented in [3], our simplified MLGCN is also a *Message Passing Neural Network*. Basically, these networks contain two parts: a message passing step and a readout step. That is for a graph g, if the message runs for T steps, for a given node u, with neighborhood $\mathcal{N}(u)$, with actual representation h_u^t, we have the following dynamics:

- **Step 1**: updates are performed by using message functions M_t, U_t.

$$m_u^{t+1} = \sum_{v \in \mathcal{N}(u)} M_t(h_u^t, h_v^t, e_{vu})$$
$$h_u^{t+1} = U_t(h_u^t, m_u^t)$$

- **Step 2**: we compute a feature vector for the whole graph with a readout function R:

$$\hat{y}_g = R\left(\left\{h_u^t \mid t \leq T, u \in g\right\}\right)$$

We called our MLGCN *simple* because in the choice of our functions U_t we overlook node degrees and our readout function R is fixed and then non trainable as in the true GCN. We used the dynamics

$$m_u^{t+1} = \sum_{v \in \mathcal{N}(u)} h_v^t$$
$$h_u^{t+1} = f_t(A_t m_u^t + B_t h_u^t,)$$

where f_t is the activation and $\hat{y}_g$ is the concatenation of information learned step-wise, $\hat{y}_g = [h_g^t \mid t \leq T]$ where $h_g^t = \sum_{u \in g} h_u^t$.

2.2 Tensorization

In this challenge, it was vital to rewrite the inputs of this graph as tensors, given that we dealt with relatively small graphs (the maximum number of nodes is 29). The loss can then be minimized by minibatch more easily[1].
We consider a training set of G graphs labeled by $g \in [\![1, G]\!]$. The graph g has n_g nodes, and let $n_G = \max_g(n_g)$. We store

[1]It allows us to avoid `for` loops in `python` for the MLGCN implementation and then save time during training.

the adjacency matrix of g in $A_g \in \mathbf{R}^{n_G \times n_G}$, and its node embeddings $H_g^0 \in \mathbf{R}^{n_G \times d_0}$. Of course for a node not belonging to g, we set to zero its corresponding line and column in A_g, and its corresponding line in H_g^0. Therefore we can rewrite the dynamics like keeping now the messages, and actual representations at graph level in matrices M_g^t and H_g^t :

$$M_g^{t+1} = A_g H_g^t$$
$$H_g^{t+1} = f_t(A_t M_g^t + B_t H_g^t,)$$

For convenience, $M_g^t \in \mathbf{R}^{n_G \times d_{t-1}}, H_g^t \in \mathbf{R}^{n_G \times d_t}$, where d_t is the dimension of the node embeddings of step t. Now at sample level adjacency matrices A_g, embeddings H_g^t, messages M_g^t are respectively stored in 3D tensors A, H^t and M^t of size $(G \times n_G \times n_G), (G \times n_G \times d_t)$ and $(G \times n_G \times d_{t-1})$. To conclude this part we can write the dynamics once for all like below :

$$M^{t+1} = H^t \ddagger A_g \tag{1}$$
$$H_g^{t+1} = f_t(A_t \dagger M_g^t + B_t \dagger H_g^t,) \tag{2}$$

where $(X \ddagger Y)_{ijk} = \sum_l X_{ilk} Y_{ilj}$
and $(X \dagger Y)_{ijk} = \sum_l X_{ijl} Y_{kj}$[2].

2.3 Optimization

Like in CNN we add an output layer on top of graph embeddings with the identity as activation since we are dealing with a regression problem. We parameters are learned using basic stochastic gradient descent. We wrote a class with Tensorflow in the file `mlgcn.py` which implements MLGCN.

3 Model

From now till the end we will do the parallel between graph and document and between node and word. Basically the baseline HAN architecture progressively encodes sentence and then document or a graph. And the encoded documents are again preprocessed through fully connected dense layers before yielding the final predictions. Then one might think that HAN performance can be improved if we can both enrich node/word embeddings and document/graph embeddings. That's what we have done through MLGCN. That is before reading MLGCN embeds nodes and graphs afterwards. So in our final architecture, the enriched node embeddings learned through MLGCN are combined to the initial one. Second we merge another layer to the HAN that also preprocesses the new document embeddings. So the final outcome of our modified HAN is also enriched by the graph representations previously learned by MLGCN. If our HAN takes advantage of the enriched embeddings learned by MLGCN, these representations are transformed before they are fed in. Indeed the last component performs multiple least-squares regressions. Yet ordinary least-squares does struggle in presence of multicollinearity. Like in the CNN, the last layer usually ends up with high dimensional vectors. So one solution could be dimension reduction. In this case, it is enough to substantially mitigate this bottleneck. Indeed even when applying simple PCA, we turned these embeddings into non-correlated vectors while keeping almost all the information.

[2]Notice that these operations can be done easily in `python` with the built-in functions `einsum` and `tensordot`.

4 Data and Sampling strategy

4.1 Document generating

Concerning documents generation from graph, we keep exactly the same choices as in the baseline . We simulate 5 random walks per node of length 10 for any graph. For larger graphs (i.e with more than 15 nodes) we limit ourselves to 70 sentences, while we pad the pseudo-documents of shorter graphs.

4.2 Graph tensors

Once the documents are generated, we also need to retrieve graph tensors, that is the embeddings and adjacency matrices. Since the bigger graph in the whole dataset only had 29 nodes, we use $n_G = 29$ throughout all the upcoming experiments. Overall we have 93719 graphs split in train and test dataset with respectively 74975 and 18744 graphs. Since the original embeddings are encoded in $\mathbf{R}^{11}$. We store the embeddings and adjacency tensors into tensors of shape $(74975, 29, 11), (74975, 29, 29)$ for the train dataset and $(18744, 29, 11), (18744, 29, 29)$ for the test dataset.

4.3 Enriching embeddings

The last point is the generation of new embeddings. To do so, we train a MLGCN with three hidden layers with respectively 100, 50 and 30 neurons. Therefore the new generated embeddings lie in $\mathbf{R}^{180}$. The MLGCN is trained over 3000 epochs with a decaying learning rate over the first 2500. First we we successively train over 500 epochs with learning equal to 0.025, 0.01, 0.005 and 0.001. We finally train over 500 epochs with a learning rate set to 0.025. At last we applied dimension reduction with PCA. They respectively account for 99.69% and 97.84% of explained variance for graph and node embeddings.

5 Experiments

Before dashing into our experiments, the first thing to notice was the activation function set by the baseline HAN on the last layer. The 4 targets to predict seem to be symmetric variable variables with amplitude greater than 2. Thus sigmoid is then a poor choice, we systematically replaced it by `linear` that is choosing the identity function as activation. We refer to this model as Han in Table 1 which reports results on the private leaderboard. Here we present also the RMSE of some ablations of our model to have a glance on their impact of the final outcome. First we only enrich node embeddings of HAN, the Han+node model in the table. the benefit of this new node embeddings is spectacular, since it reduces the RMSE from 0.35 to 0.15. In a similar fashion we add the branch of the processed graph embeddings, here the Han+Node+Graph model. We witness a slight but striking decline in RMSE. The latter result thus emphasizes the importance of the MLGCN graph embeddings. As discussed

earlier, we cannot fully benefit from the contribution of these new embeddings because of multicollinearity. So we conduct the same investigations but this time with PCA-enhanced embeddings. In every case, it leads to better results.

Table 1: The performances of our experiments

	RMSE		
Model	Han	Han+node	Han+Node+Graph
RMSE	0.352	0.164	**0.132**
RMSE	–	0.129	**0.109**

6 Conclusion

In this challenge, we developed at some extend an hybrid HAN models. First we enrich node embeddings and second we graft a layer of processed documents embeddings to the core HAN architecture. Those new embeddings were trained by the so-called MLGCN model introduced above. At the end we placed 4/46. Yet the present version of the work can be improved in many ways. First we can use a trainable readout function for the GCN. Second we can use a better autoencoder rather than PCA. When we glanced at the data, we noticed that the targets seem to be normal distributed while target 3 is the most affected by outliers. So in this setup, we think that the use of batch normalization can improve our results.

References

[1] Ross. Girshick, "Fast R-CNN", *IEEE International Conference on Computer Vision (ICCV*, 2015.

[2] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks", *Neural Information Processing Systems (NIPS)*, 2016

[3] Duvenaud et al., "Convolutional Networks on Graphs for Learning Molecular Fingerprints", *Neural Information Processing Systems (NIPS)*, 2015

Appendix

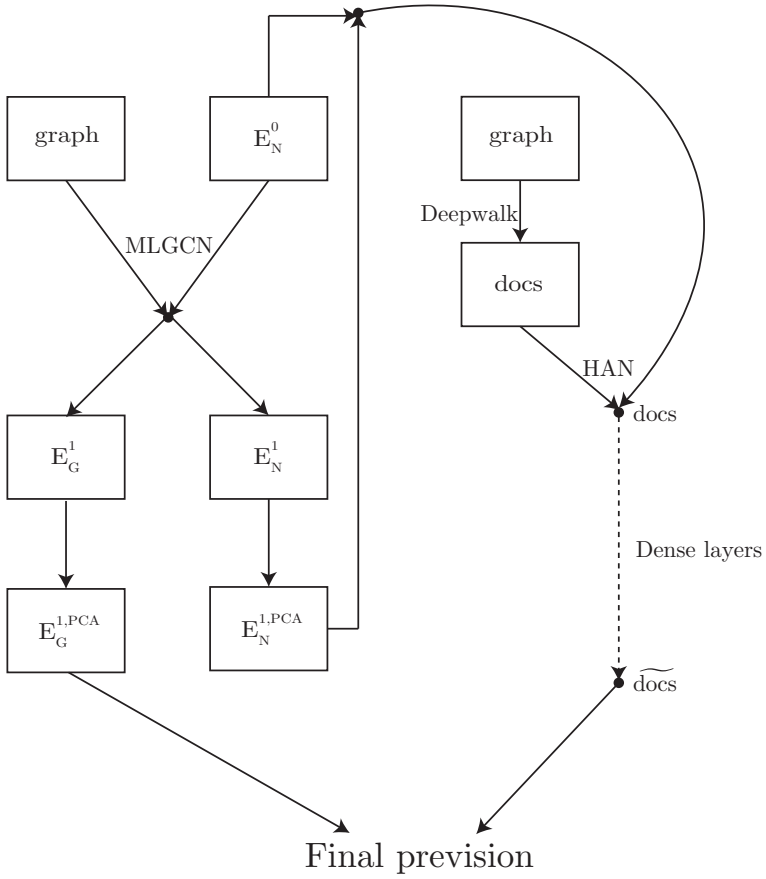

Figure 1: Architecture