Advanced Machine Learning for Graphs and Text Data Challenge 2018-19

Clément Hardy[1] , **Guillaume Fradet**[1] , **Félix Larrouy**[1]
[1]École Polytechnique, Master Data Science
{clement.hardy, guillaume.fradet, felix.larrouy}@polytechnique.edu
Kaggle Team: Larrouy - Hardy - Fradet

Abstract

We propose an improvement of the Hierarchical Attention Network (HAN) [Yang *et al.*, 2016] in the context of a very common NLP task, which is multi-target *graph regression*. We also focus on the documents extraction from the graph data, which is crucial for the HAN architecture to yield good performances.

1 Introduction

A graph can be represented as a document whose words are the nodes of the network and whose sentences are random walks sampled from the network. Our dataset is made of 93,719 undirected, unweighted graphs, where each graph is associated with a continuous value for each of 4 target variables.

We are provided a preprocessing baseline for generating the documents, where for each graph we generate random walks (i.e. sentences) with a fixed sized. Each document has a maximum number of sentences and each sentence has a maximum number of words.

We also have a model baseline, which is a very simple Hierarchical Attention Network.

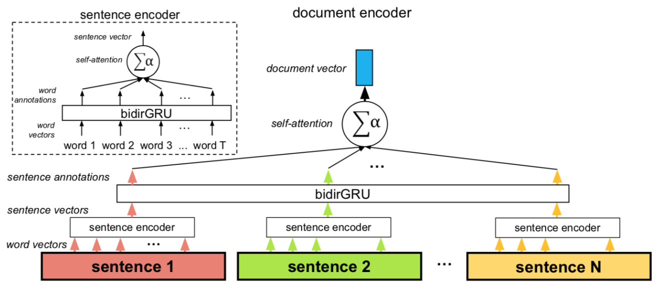

Figure 1: Hierarchical Attention Network

The goal of this model is to first determine which words are important in each sentence, and then, which sentences are important in the document. From this importance vector we can predict a continuous value for each of the 4 targets. In the next section, we will introduce several possible improvements for both the preprocessing part and the HAN architecture.

2 Our approach

2.1 Preprocessing

The goal of this part is to generate documents from the given graphs. These documents are essentials because both the learning and the prediction rely on them, as they are the inputs of our network. That being said, it is not surprising that a better way to generate the documents results in a higher accuracy of the model, as it enables the network to learn on more representative documents.

To generate our documents, we decided to replace the simple random walks through the graphs by more advanced and recent techniques: *DeepWalk* [Perozzi *et al.*, 2014] and *node2vec* [Grover and Leskovec, 2016]. Both of these techniques aim to extract meaningful features from graphs, by using random walks to quantify similarity between nodes in a graph. Said differently, that is learning feature vectors from graphs.

DeepWalk

With *DeepWalk*, we imagine random walks as “sentences” and we learn to estimate the probability that a vertex appears in a random walk, given the vertices before it. This algorithm needs three parameters to run. The first one is the walk length (t), which determines the number of nodes in each sentence. Then comes the number of walks (γ), that is the number of random walks to be generated from each node in the graph. Finally, we give the embedding dimensions (d).

node2vec

This more recent method is based on the latter technique but introduces two new hyperparameters: p and q. p is the ”return” parameter. It controls the probability of going back on the previous node. If p is low, we increase the likelihood of going back on a previous node, so we emphasize the local structure of the graph. q is the ”inout” parameter. It controls the way on which the graph is explored. $q > 1$ favors a Breadth-first search (BFS) exploration, while $q < 1$ favors a Depth-first search (DFS) exploration. The *DeepWalk* algorithm is a specific case of *node2vec* where $p = q = 1$, so implementing the latter method was enough to cover both of them.

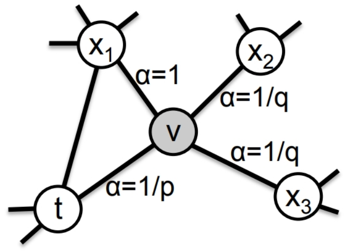

Figure 2: Illustration of the random walk procedure in node2vec. The walk just transitioned from t to v and is now evaluating its next step out of node v. (illustration from the paper)

The choice of p and q is very specific to each dataset and task. In the *node2vec* paper [Grover and Leskovec, 2016], the authors learn these hyperparemeters using 10-fold cross-validation on 10% labeled data with a grid-search over $p, q \in \{0.25, 0.50, 1, 2, 4\}$. To gain some time, we only generated 9 documents with the following grid-search: $p, q \in \{0.25, 1, 4\}$ and we found out that $p = 4$, $q = 4$ was the best couple of hyperparemeters for our task, as it was giving us the best results.

We used the very nice Python 3 library of *eliorc*[1] to implement *node2vec* and generate our documents.

2.2 Model

Self Attentive Sentence and Document Embedding

In the baseline model, sentences and documents are encoded in a single vector. This vector representation usually focuses on a specific component of the sentence, or a specific part of the document. Here h_i is the Bidirectional GRU hidden state of a token or a sentence and c is a context vector.[2]

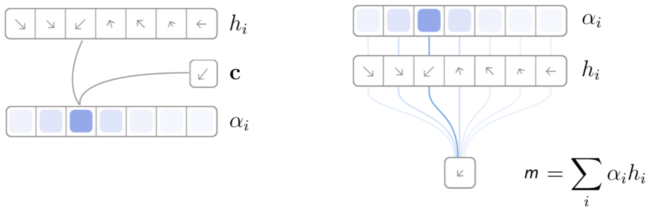

Figure 3: Attention mechanism with a single representation vector

However, there can be multiple components in a sentence/document that together form the overall semantics of the whole sentence/document, especially for long sentences/documents. Here is a very simple example for the purpose of illustration. Imagine we want to encode the following sentence: " I like to play football but I don't like tennis ". A single vector is not sufficient to capture the overall structure of this sentence as they are two information separated by the word "but". Thus, to represent the overall semantics of the sentence/document, we need multiple representation vectors [Lin *et al.*, 2017].

[1]https://github.com/eliorc/node2vec

[2]https://github.com/m2dsupsdlclass/lectures-labs/blob/master/slides/07_deep_nlp_2/index.html

Therefore, instead of computing a vector α, we will now compute a matrix A of shape $r \times n$ where r is the number of hops attention, i.e. number of parts of the sentence or document to focus on (r is a hyper-parameter to tune), and n is the number of tokens in a sentence, or the number of sentences in a document.

The embedding vector m then becomes an $r2 \times u$ embedding matrix M such that $M = A \times H$, where $H = (h_1, h_2, ..., h_n)$. Details on how to compute the matrix A are provided in the original paper [Lin *et al.*, 2017].

However, the embedding matrix M can suffer from redundancy problems if the attention mechanism always provides similar weights for all the r hops. Thus we need a penalization term to encourage the diversity of these weight vectors across different hops of attention. To do that, the goal is that each individual row of the matrix A focuses on a single aspect of semantics. A new penalization term suited for this specific task is introduced, defined as follows:

$$P = ||(AA^T - I)||_F^2$$

where $|| \cdot ||_F$ stands for the Frobenius norm of a matrix. This penalization term will then be multiplied by a coefficient λ and minimized with the original loss.

Hierarchical Convolutional Attention Network

In this model, instead of looking at each word separately, we would like to implement a method that uses the information contained within the neighborhood of the word. Moreover, we think that using the interactions between words could help.
This model was inspired by the paper, *Hierachical attentional hybrid neural network for document classification* [Abreu *et al.*, 2019]. Even if classification was not our task, we found this architecture interesting and transposable to our regression task.
Before the first recurrent neural network, we place a convolutional network. Doing so, a first issue appears because we have to use a fixed size windows with the convolutions. Indeed, the most important information in the neighborhood of a word could be contained in the adjacent word, like it could be a few words away. To avoid this, several layers with different window sizes are used. Thus different features are created, representing the information of different lengths of n-grams. Then, only the k most important features extracted by those layers are kept. A specific layer has been implemented to do such a thing, it is a k-max pooling layer. Finally, all the features extracted are concatenated. For the next layers, we keep the same architecture as the one in the baseline.

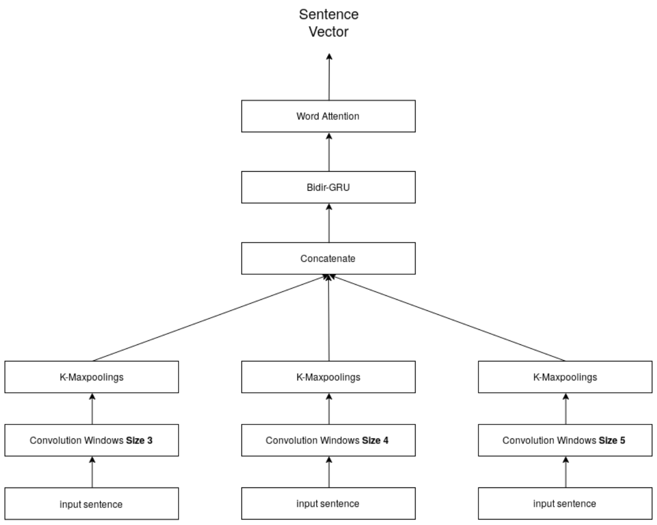

Figure 4: Sentence level

We used three different windows sizes which are 3, 4 and 5. We chose to keep these window size values presented in the paper.

Multi-Head Attention

The baseline model has one head attention. The model that we describe here uses the same type of attention but runs several attention processes in parallel. Using a multi-head attention could help by the fact that each head could focus on a different part of the sentence. We will now explain how this attention works.

The heads are used to project the input embedding into a different representation subspace, i.e. each head project the input into a different one. Doing so, we end up with different matrices (e.g. two heads give two matrices) however the model could only manage one resulting matrix. So the solution to this problem is to concatenate the matrices and then multiply the resulting concatenation by another matrix, to be projected again in a subspace with a dimension that can be managed by the model. The resulting matrix has the same size than the matrix we would have had if we were using a one-head attention, but it has the advantage to contain the information captured from the different heads.

In the paper *Attention Is All You Need* [Vaswani *et al.*, 2017] presenting this method, a positional embedding layer is used to represent the order of the words in the sentence. However, we did not use this layer as this was done by our recurrent network.
Also, we made the choice to use this attention only at the sentences level in our architecture. The documents level persists with a simple one head attention, implemented in the baseline model.

After testing with many numbers of heads, it turns out that the best results were when we used only two heads. Above this value, the model was quickly over-fitting and the generalization was therefore very bad.

Ensemble Method

We noticed that the baseline model was not so difficult and long to train. It takes only a few hours with a good GPU. Furthermore, as we said before, the type of sentences (i.e. walks) generated by *node2vec* can make a real impact on the results. Indeed, for some targets a certain value for p and q was the best, while it was another configuration that were more accurate for other targets.

By looking carefully at the predictions, we can easily see that the predictions are not always similar with two models trained on different documents.

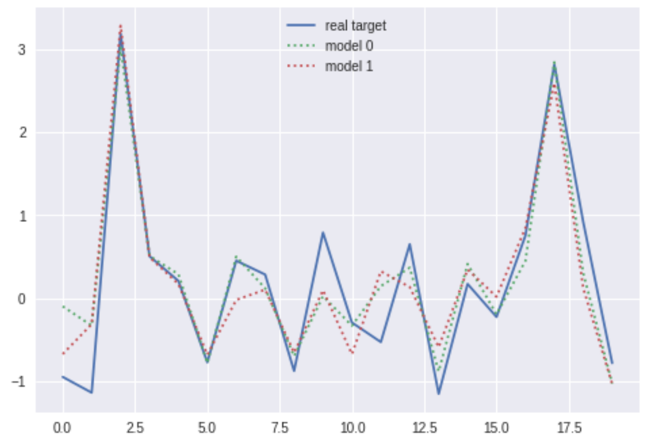

Figure 5: Prediction of 20 samples of the validation set (target 0)

That gave us the idea of using several models trained on different sentences types, so that we can then make the average of the prediction of each model. In other words, an ensemble of HAN. As the HAN baseline model was quick to converge compare to other architectures (Self Attentive, convolution attention, ...) with good results, we decided to use the baseline architecture in the ensemble, with tuned parameters.

Each model was taking documents as input, generated with *node2vec* but with different parameters (p, q), documents sizes and sentence lengths. The goal was to make each model unique, the precision of each model was not the priority, as the goal was to obtain good predictions by taking the average. So in some way, two predictors which give a really good average is better than two better predictors but with a worse average.

We keep in our ensemble five models, it seems to be a good compromise between performance and computation time.

$$\begin{tikzcd}
\boxed{\text{Model 1}} \arrow[d] & \boxed{\text{Model 2}} \arrow[d] & \boxed{\text{Model 3}} \arrow[d] & \boxed{\text{Model 4}} \arrow[d] & \boxed{\text{Model 5}} \arrow[d] \\
\hat{y}_1 \arrow[drr] & \hat{y}_2 \arrow[dr] & \hat{y}_3 \arrow[d] & \hat{y}_4 \arrow[dl] & \hat{y}_5 \arrow[dll] \\
 & & \hat{y} = \frac{1}{5}\sum_{i=1}^{5}\hat{y}_i & &
\end{tikzcd}$$

We will see in the next section, this ensemble give quite good performances. That is why we decided to keep this idea for our final submission. We also tried other types of ensemble's models (more complex ones).
For example, in the following graph (which is taken from the target 2), we can remark that some points are really badly predicted (i.e. predictions clearly above or below the real value).

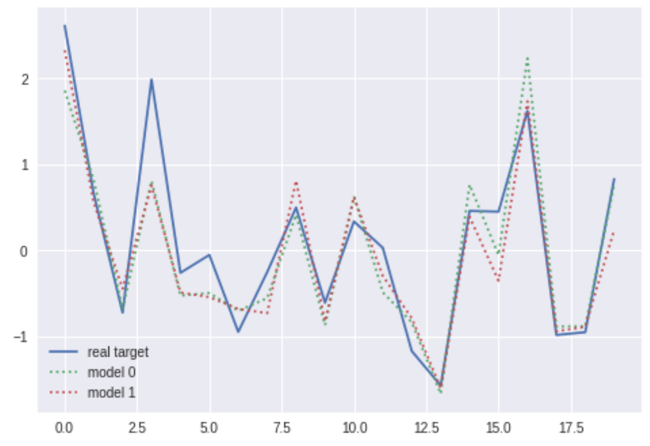

Figure 6: Prediction of 20 samples of the validation set (target 2)

We had the idea to use several models but instead of making them learn the same thing, we were thinking about using different loss functions. One of this function should penalize more an error above the real value, while another should penalize more a prediction below the real value. For instance, the following functions:

$$l_1(y, y_{pred}) = \frac{1}{n}\sum_{i=1}^{n} 1.5 * \mathbb{1}_{(}y > y_{pred})(y_{pred} - y)^2$$
$$\mathbb{1}_{(}y > y_{pred})(y_{pred} - y)^2$$

$$l_2(y, y_{pred}) = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}_{(}y > y_{pred})(y_{pred} - y)^2$$
$$1.5 * \mathbb{1}_{(}y > y_{pred})(y_{pred} - y)^2$$

By using two models, each with a different loss function, we were hoping, one will pay more attention to the "big" values and the other to the "small" values. Doing an average, the predictions should be good. Unfortunately, even if these loss functions help to predict some points where the prediction was a "disaster" without these functions. In general the prediction was clearly worse. In the end, the use of these loss functions was completely inefficient.

Other ideas

Activation function: one critical problem we found in the baseline model is the activation function for the last dense layer, the one that predicts the output for a target. We can look at the distribution of the values for each target.

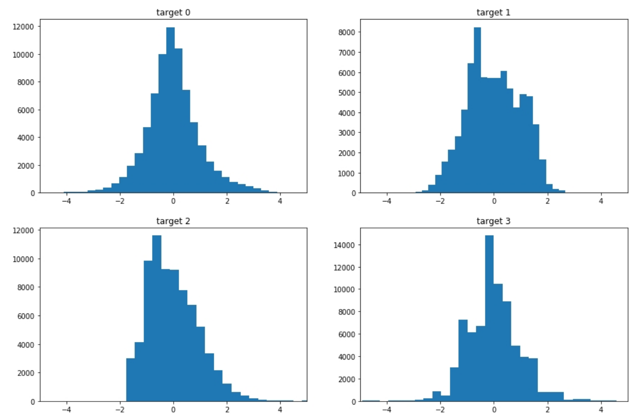

Figure 7: Output distribution for each target

The baseline activation function, sigmoid, is not suited for this problem as the sigmoid function outputs values between 0 and 1. Using sigmoid we can not predict approximately half of the true values. First, we replaced the sigmoid by tanh function, which outputs values between -1 and 1. But we are still missing some values. To overcome this problem, we tried several custom activation function that we implemented ourselves, e.g. $2 \times tanh$ which outputs values between -2 and 2. Finaly, the best results we could get was using a very simple linear activation (i.e. $a(x) = x$).

Different parameters for each target: after running our model on the 4 targets, we noticed that the performance were about as good for 3 of the 4 targets, but not satisfactory for one of them, which is the target "2". Therefore, we focused on the "bad" target to tune its hyper-parameters, separately from the 3 others. We played with the number of units of the Bidirectional GRU (maybe we need more dimensions in the hidden state to capture the sementics), the learning rate (maybe the loss landscape is very different for this target), the batch size, and the Dropout rate (maybe we are under-fitting and thus we don't need to include Dropout). According to the original paper of HAN [Yang *et al.*, 2016], batch size can accelerate training by three times (see section 3.3, Model configuration and training), but we were not able to observe this phenomenon on our task, maybe because in the original paper they are doing document classification.

We also implemented a learning rate decay scheduler, to adjust the learning rate during training by reducing it. Common learning rate schedules include time-based decay, step decay and exponential decay. We implemented a step decay schedule, which drops the learning rate by a factor (drop_rate) every few epochs (epochs_drop).

$$lr = lr_0 \times \text{drop\_rate}^{\lfloor \text{epoch / epochs\_drop} \rfloor}$$

Modifying the GRU layer: as we mentionned in the paragraph above, we might need more (or less) units than in the baseline to capture the semantics of a sentence or a document. The number of hidden units controls the dimension of the hidden states.

Layer Normalization: it is known that this procedure can improve the performance of RNN [Lei Ba *et al.*, 2016], where batch normalization is not well suited. So we tried to place this layer just after the GRU layers of the sentences level and documents level. Unfortunately, it did not behaved like expected. Instead of reducing our training time, it made our network much more difficult to optimize.

3 Results

We evaluate our model on 4 targets. We use 80% of the data for training and the remaining 20% for the validation.
For each target, we trained 5 models as part of an ensemble method. The main difference between each model is the document is takes as input. We used different combinations of the values of the parameters p and q for each of the 5 models. We also changed the values of the parameters *walk_length and num_walks* for one document.

	p	q	num_walks	walk_length
model 0	4	4	5	10
model 1	6	4	10	5
model 2	0.25	4	5	10
model 3	1	1	5	10
model 4	1	4	5	10

It took approximately 21 hours to train on a NVIDIA TITAN Xp GPU with 3840 cores, 11.4 Gbps of memory speed and 12 GB GDDR5 of memory.
For the HAN architecture, we used 100 units for the GRU layer, with no activation function (we noticed that the GRU layer does not use any activation function when we run it on a GPU). We used a dropout rate of 0.1, a batch size of 32, and the Adam optimizer with a learning rate of 0.01 and then 0.001 after 10 epochs. The number of epochs varies for each target, an can also vary for each model for a specific target.

	target 0	target 1	target 2	target 3
model 0	0.135	0.072	0.344	0.123
model 1	0.143	0.072	0.343	0.035
model 2	0.164	0.084	0.352	0.092
model 3	0.165	0.082	0.332	0.103
model 4	0.152	0.081	0.351	**0.009**
ensemble	**0.118**	**0.063**	**0.318**	0.050

This ensemble method allows us to improve our score from 0.16794 with a single model, to 0.13871, which is our best score.

To see how much the ensemble help to improve our performance, let's plot the validation loss improvement during training on the first target.

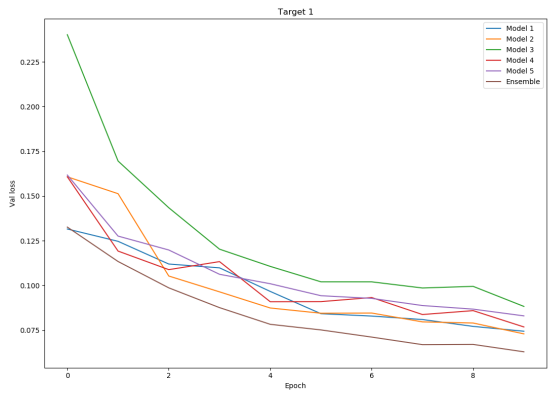

Figure 8: Val loss on the 1 target

We can easily notice that the ensemble validation loss is always below the others. This behaviour is the same for every target. The ensemble is always doing better than every single model.

4 Conclusion

As part of this challenge, our goal was to improve a provided baseline architecture, including both the preprocessing part and the predictive model. We have oriented our research in two main axes: generating better documents from the graphs that were provided, and improving the Hierarchical Attention Network architecture.

For the preprocessing part, *node2vec* gave us the best results. We could have tried other techniques such as relabeling procedures, e.g. the Weisfeiler-Lehman could be used. Concerning the HAN architecture, we explored several possible improvement (see section 2.2). All of them were not improving our predictions, but we never got worse results than the provided baseline model. Changing the activation function for the layer predicting the output value was crucial to improve the results. Adding more units to the Bidirectional GRU and using ensemble methods helped in the improvement of the model.

References

[Abreu *et al.*, 2019] Jader Abreu, Luis Fred, David Macêdo, and Cleber Zanchettin. Hierarchical attentional hybrid neural networks for document classification. *CoRR*, abs/1901.06610, 2019.

[Grover and Leskovec, 2016] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. *CoRR*, abs/1607.00653, 2016.

[Lei Ba *et al.*, 2016] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer Normalization. *arXiv e-prints*, page arXiv:1607.06450, Jul 2016.

[Lin *et al.*, 2017] Zhouhan Lin, Minwei Feng, Cícero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. A structured self-attentive sentence embedding. *CoRR*, abs/1703.03130, 2017.

[Perozzi *et al.*, 2014] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. *CoRR*, abs/1403.6652, 2014.

[Vaswani *et al.*, 2017] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[Yang *et al.*, 2016] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. Hierarchical attention networks for document classification. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1480–1489, San Diego, California, June 2016. Association for Computational Linguistics.