ALTEGRAD - Data Challenge 2019
Predicting continuous values associated with graphs

Cyril Equilbec[1] , **Vincent Jacob**[1]
[1] M2 Data Sciences - Université Paris-Saclay
{first.last}@polytechnique.edu

Abstract

This report presents our solution for the ALTEGRAD data challenge 2018-19. The goal of the project was to solve a 4-target *graph regression*, using a Hierarchical Attention Network (HAN) architecture for generating high-level graph representations.

1 Introduction

In this challenge, we are given $93,719$ undirected and unweighted graphs, each containing on average approximately 18 nodes and 19 edges. Each node comes with 13 pre-computed features, that characterize its latent attributes, independently from its position within its containing graph. Even though it is specified that each node is unique among all graphs, some nodes may be similar in this 13-dimensional embedding space. What we aim to do is to find an efficient way to train 4 regression models that make use of the HAN architecture as a graph encoder. We can therefore break the task into the following steps:

1. Build a *pseudo-document* representation for each graph, a pseudo-document being defined as a *sequence of walks in a given graph*. This permits us to draw a parallel between graphs and documents for NLP. Here each graph will correspond to a *document*, each walk to a *sentence* in this document, and each walk's node to a *word* in the corresponding sentence.

2. Use these computed *pseudo-documents* as inputs for training a HAN model, in order to construct high-level representations of those pseudo-documents, and by extension of the graphs.

3. Perform the final regression(s) on the graphs representations.

In the following, we will cover the methods we used to perform each of these steps, and some of the important results we obtained. Finally, we will get back to the overall final performance of our model(s), before emitting some ideas that might be worth exploring for further improving this overall performance.

2 Sampling strategies

In this section we present the various methods we considered and used for building *pseudo-documents* representations for the raw graph data.

2.1 Baseline method

All the methods we tested, including the baseline method, are based on the following pseudo-documents generation procedure:

```
foreach graph do
    Result: walks of the document for that graph
    repeat
        foreach node do
            random_walk(node, walk_length)
        end
    until $r$ times;
end
```

Algorithm 1: Documents extraction

Hence, for generating the pseudo-document representation of a given graph, we skim through all of its nodes r times (possibly in different orders), and generate for each node a random walk (*i.e.* path), that can be seen as a *sentence starting from the node's associated word*. The pseudo-document representation for that graph is then simply the sequence of all sentences that were generated in this process.

Using this method offers a lot of flexibility in performing this transformation from graph to document, some concerning the *quantitative* aspect of the document, like the maximum number of sentences by document, the number of words in each sentence, the number of sentences starting from a given node (r), and so on; but also concerning their *qualitative* aspect, by being able to tell what *strategy* to follow in generating the paths in the graphs.

The baseline method handles all this in a quite simple, yet not so inefficient way: the sentences length is fixed for every document, as for the number of sentences starting from each node. Hence, the number of sentences in a given document is $\#nodes \times r$. To make sure the inputs of the HAN model all have the same length, the number of sentences of all documents are further fixed to a given number: the documents that contain too much sentences are truncated, and the ones that do not contain enough are padded with sentences full a of an *empty* token word. A similar strategy will be used when we will consider sentences of variable lengths.

We did find some room for improvement in changing some of these cardinalities, including adding more variability in the sentences lengths. But the main aspect that we focused on was to change the *way* sentences were generated, not using a uniformly random walk like in the baseline method, but adding some *bias* in the randomness, so that walks follow a given *sampling strategy*, much like what was used in designing the **node2vec** algorithm.

2.2 Node2Vec sampling strategies

Given a graph $G = (V, E)$ (set of vertices and egdes), the node2vec algorithm aims to learn the best feature extraction function $f^* : V \to \mathbb{R}^d$ so that, given the features $f^*(u)$ of a node $u \in V$, the likelihood of observing its defined *neighborhood* $N_S(u) \subset V$ is on average maximized. In the algorithm, this comes down to the following optimization problem:

$$f^* = \arg\max_f \sum_{u \in V} \log \mathbb{P}(N_S(u) \mid f(u))$$

What really makes the node2vec algorithm stand out is the flexibility it offers in defining what the "neighborhood" of a given node u is, according to what *sampling strategy* S we chose to form $N_S(u)$ from u. Being able to tune this sampling strategy is very interesting since it permits to express the **connectivity patterns** that we ultimately care about within the graphs, here in the context of designing features for their nodes.

In general, there are two main types of connectivity patterns on which we might want to focus on when building similarities within a subset of nodes in a graph. The first one is a relationship known as **homophily**, that is satisfied when the set of nodes form together a compact cluster in the graph. The other is **structural equivalence**, satisfied if the set of nodes play the same *role* in the graph's structure, even if those nodes might be located far apart. In the context of node2vec, looking at either of these patterns (or sometimes at both combined) helps define the nodes for which we want the features to be similar. In our context, we might want to generate sentences that take into account these patterns in performing our 4 regressions:

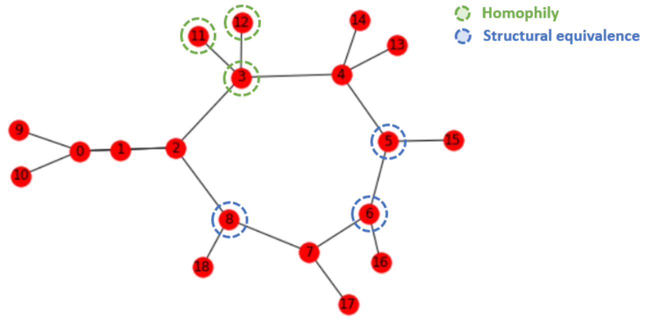

Figure 1: Example of nodes relationships in the first graph

The idea of the node2vec algorithm is then to come up with a *biased* random walk strategy to form neighborhoods, or here *sentences*, that aim to reflect either or both of these types of patterns within the graphs:

$$\mathbb{P}(c_i = x \mid c_{i-1} = v) = \begin{cases} \frac{\pi_{vx}}{Z} & \text{if } (v, x) \in E \\ 0 & \text{otherwise} \end{cases}$$

with:

$$\pi_{vx} = \alpha_{pq}(t, x) = \begin{cases} \frac{1}{p} & \text{if } d_{tx} = 0 \\ 1 & \text{if } d_{tx} = 1 \\ \frac{1}{q} & \text{if } d_{tx} = 2 \end{cases}$$

π_{vx} being the unnormalized transition probability between node v and node x, and Z a normalizing constant. We see that π_{vx} is in fact defined in terms of another node t (and the minimum distance between t and x d_{tx}), which is the node preceding v in the current walk. π_{vx} is also parameterized by two real numbers p and q:

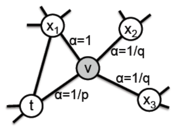

Figure 2: Random walk illustration from the node2vec paper

The implicit assumption here is that the nodes of a given random path form a 2^{nd} order Markov chain, and the *hyper-parameters* p and q are what permits us to adapt our exploration strategy, starting from the initial node of the path.

The most common exploration strategies in graph theory are the **Breadth-First Search** (BFS) algorithm, which consists in forming the explored nodes by first sampling the immediate neighbors of the closest node from the start, and the **Depth-First Search** (DFS) algorithm, which samples first the immediate neighbors of the deepest nodes in the currently sampled space (depth being defined here as the distance from the first sampled node).

We can see that these two extreme strategies correspond in fact to very different graph representations. With BFS, we want to build a *microscopic* map of the graphs, well characterizing local patterns, which enables us to discover some similarities between the structures of very distinct areas, *i.e* structural equivalence patterns. DFS, on the other hand, aims to build a more precise *macroscopic* view of the graphs, to be able to more easily spot clusters within their overall structure, *i.e* homophily patterns. In our pseudo-document generation context, using BFS would tackle the question *"are the sentences constructed in the same way, even though they might be far apart in the document?"*, whereas using DFS would try to answer *"are certain words always closely together, even within different sentences?"*.

Defining a 2^{nd} order transition probability distribution that depends on p and q instead of a plain hard BFS or DFS strategy to construct samples permits us, in addition to gain in computational efficiency, to smoothly interpolate between the two, simply by tuning p and q.

In the node2vec terminology, p is called the **return parameter**. Like we can see from the transition probability expression, the higher is p, the less likely we are to revisit a node immediately in the walk (that is, going back to t after having been from t to v).

The hyperparameter q is defined as the **in-out parameter**, and is fundamentally controlling whether we want to explore the search space in an *inward* or *outward* fashion. The higher the parameter q, the less likely we are to select a node x at a distance 2 of the node t, and therefore the more we encourage local searches (closely to the BFS strategy). Alternately, setting q at a very low value will tend to make us produce sentences that are more often spread out in the graphs (kind of like we would have obtained using DFS).

2.3 Usage and additional improvements

What is really interesting with the flexible sampling strategy presented above is that it offers, even in our multi-target regression setting, the ability to **easily tune the graphs representations crafting in the most relevant way in order to predict separately each target**. Indeed, some targets might be more receptive to sentences representing local patterns in a given graph, some others to more global views, or even some to a combination of both.

We chose to stick to the originally provided 13-dimensional feature space for the graphs' nodes, but we definitely found some added value in using the node2vec sampling strategies for creating the documents' sentences, as we will see below.

Once we know how to tune the sampling strategies, another thing that we might want to focus on is adding some more *variability* in the produced pseudo-documents. HANs were initially developed for NLP tasks, we could therefore expect better performance if we manage to make our documents look close to the ones we might find in natural language: sentences with different number of words, documents with different number of sentences, and so on.

The way we handled the lengths of sentences was quite simple. Each time we wanted to produce a sentence (by performing a biased random walk) we defined its length to be either uniformly drawn at random between two extreme values, or sampled according to a normal distribution of given mean and standard deviation.

We also considered a simple technique that presented the advantages of randomizing the number of sentences starting from each word, of complicating the sentences in documents, but especially of inducing **high improvement of the computational performance for generating pseudo-documents** (though computational efficiency was not technically required for the challenge). Instead of simply generating sentences of random size l, we tried to generate *walks* of fixed length l, from which we further extract $l - k + 1$ sentences of size $k < l$, all being shifted by 1 node (k is also random, but fixed for a given l-walk). Unfortunately, we did not found so much score improvement handling sentences this way, and therefore stuck to the random l-sentences generation for our final submission.

2.4 Experiments and results

The general method that we used for our experiments was first to select an HAN architecture that we slightly adapted from the baseline model and fix it, then run it during approximately 15 epochs, before evaluating the **Mean Squared Error** (MSE) for each target separately for a given hyperparameters sequence. From this, we finally extracted the best results we got, like in the following sub-table:

smaxl	swords	rate	p	q	t0	t1	t2	t3
-	-	-	-	-	0.19	0.10	0.40	0.22
100	uni{4;13}	5	1	2	0.18	0.10	0.39	0.06
80	uni{8;11}	5	0.5	2	0.19	0.09	0.42	0.15
80	uni{8;11}	5	2	0.5	0.19	0.09	0.40	0.16
160	uni{8;11}	8	2	2	0.16	-	0.40	-

Note: The first line is using the baseline sampling strategy. *smaxl* is the sentence maximum length (in number of words) that we allow for a given document. *swords* is the sampling strategy for the number of words in a given sentence (*uni* is for uniform (that turned out better), *norm* for normal). The *rate* is the number of sentences starting from each node, supposed fixed here.

From what we tried, we could therefore see that, unfortunately, we overall did no better than the baseline sampling strategy for some targets. For t_0 and especially t_3 however, results were promising. So we compared the performance between baseline sampling and the corresponding adapted strategy with a more advanced model and more epochs:

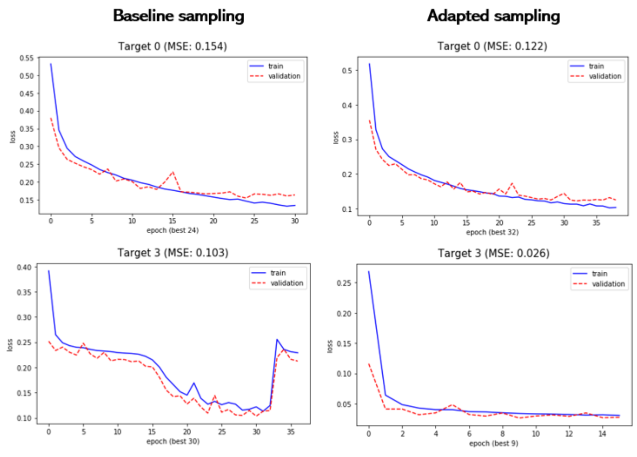

Figure 3: Baseline (left) vs. adapted (right) sampling losses

We can see that we managed to improve target t_0's MSE by a bit, but this real drastic change came, as expected from the table, with t_3. Indeed, we found that graphs' representations being random sentences between 4 and 13 words, and describing local patterns in the graph ($q = 2 > 1$) was really game changing in predicted this target, leading to a **reduction of its MSE by a factor of** 5, while presenting a **much faster convergence** than with the baseline sampling.

3 Model architectures

In this section we present the general idea behind the HAN architecture, and the various modifications we performed and tested to improve the regression results.

3.1 General HAN architecture

The Hierarchical Attention Network (HAN) has originally been designed to capture two basic insights about the structure of NLP documents. First, by observing that these documents have a hierarchical structure (words form sentences, sentences form a document), their final representation aims to be created by first building representations of sentences, before aggregating those into documents.

Second, it is observed that different words and sentences in a document are differently informative. Moreover, the importance of words and sentences are highly context dependent, *i.e.* the same word or sentence may be differently important in different contexts. To take this into account, the model includes two levels of attention mechanisms: one at the word level and one at the sentence level.

The architecture is made of several parts, the first one is the embedding layer which converts 1-hot encoded words to vectors in $\mathbb{R}^{13}$ using an already provided embedding matrix. Then, a bidirectional GRU layer is used to get annotations from words by summarizing their contextual information from both directions in the sentence they are in. By default, the resulting annotation is the concatenation of the forward GRU and the backward GRU. An attention mechanism is then introduced to extract words that are most similar to a trainable *word context vector*, at the end correlated with their relevance in the sentence. Finally, the *sentence vector* encoding a given sentence is the sum of the annotations of its words, weighted by their similarity with this word context.

More formally, if we consider the i^{th} sentence from the document formed by the words $w_{i1}, w_{i2}, ..., w_{iT}$, for a given word w_{it} its embedding is given by $x_{it} = W_e w_{it}$ where W_e is the embedding matrix, and its annotation is given by h_{it}, which is the concatenation of its forward hidden state, obtained by the forward GRU applied on x_{it}, and its backward hidden state, obtained by the backward GRU also applied on x_{it}.

That is, the annotation h_{it} is first fed through a fully-connected layer with trainable weights W_w, bias b_w and $tanh$ activation to get u_{it} as a hidden representation of h_{it}, then the importance of the word is measured as the similarity of u_{it} with the randomly initialized and trainable word level context vector u_w (using an unnormalized dot product), to finally get a normalized importance weight α_{it} through a softmax function. The resulting high-level sentence vector s_i is computed as a weighted sum of the word annotations based on the weights:

$$\begin{aligned} u_{it} &= tanh(W_w h_{it} + b_w) \\ similarity &= u_{it}^T u_w \\ \alpha_i &= softmax(similarity) \\ s_i &= \sum_t \alpha_{it} h_{it} \end{aligned} \tag{1}$$

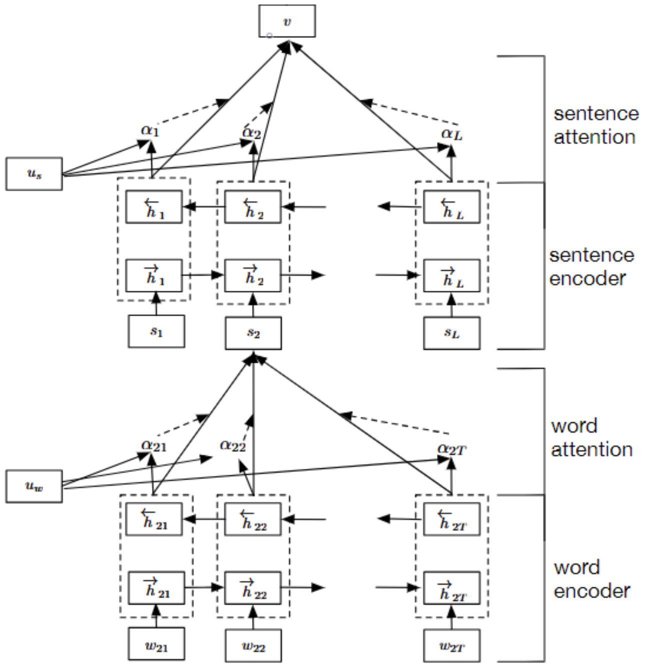

Figure 4: HAN architecture

The exact same idea is then applied at the sentence level to get v, which is the document vector that summarizes all the information of the sentences in a document. In the baseline code, this high-level representation of the document is then fed to a dense layer with a $sigmoid$ activation to perform a regression task with respect to the Mean Squared Error (MSE) loss.

Note: This architecture applied to all 4 targets yields an overall MSE of 0.77510. We managed to get this error reduced to 0.13 by tweaking this architecture for each target.

3.2 Experiments and training

In this section, we will present the experiments we conducted in order to improve the provided baseline architecture. This includes experiments on hyper-parameters (batch size, learning rate strategies, number of epochs, choice of the optimizer, number of units etc.) and experiments on the model's architecture itself.

Our very first modification was to switch the activation function of the very last dense layer from $sigmoid$ to $linear$, because the targets' values were not bounded and the $sigmoid$ activation was squashing the output between 0 and 1. This simple modification based on observations of the raw data improved our score from 0.7 to 0.2 MSE.

To improve further the scores, we made the network $deeper$ by adding fully-connected $dense$ layers with $linear$ activation (we didn't want to constraint the output) after the GRU layers. This allowed the network to be more expressive and hence better capture the data, and ended up helping us to reduce the overall MSE. We also played with the number of units in the GRU layers and the dropout rate, and found out that in our case the optimal number of units and dropout were respectively 60 and 0.1 for every targets but the target 2, and respectively 45 and 0.6 for the target 2 (with an additional embedding dropout of 0.1). We also tried to change the merge mode of the bidirectional GRU layers, we tried *average*, *concatenation* and *sum*. We found that sum and average yielded similar results as concatenation while dividing the number of parameters by 2, and therefore used those modes. With the same idea of limiting the number of parameters (mostly for computation time and over-fitting issues), we tried different attention mechanism refinements such as changing the output activation in the MLP computing annotations' hidden states. We tried $tanh$, $sigmoid$ and $linear$ activations, as well as not using any MLP at all but instead use the raw annotations h_{it} to compute the similarity score (either with the unormalized dot product, or with cosine similarity) with the context vector u_w. Those modifications did not induce a drastic change, so we decided to stick to the baseline implementation for all targets but target 2, for which not using an MLP and using the cosine similarity helped a bit the training as it involved far less parameters.

We also quickly noticed that this architecture was overfitting on the target 2, this is most likely due to the particular distribution of its values: every targets are approximately normally distributed around 0, except for the target 2 where most of the values are 0 and few that are positive and large. We therefore figured that the network was correctly predicting the almost 0 values, while struggling to predict the large ones.

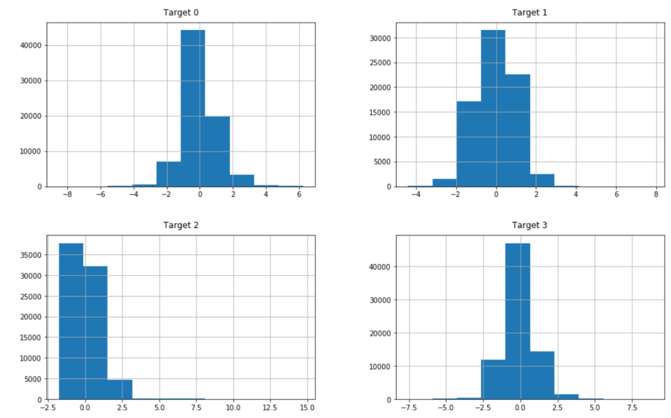

Figure 5: Targets distribution

We first reduced the number of hidden units in the GRU layers up to 45 units to reduce the complexity of the model, but still kept a high enough number to hinder the underfit. This helped a bit but was still not sufficient. We also changed the optimizer to $Nadam$ [1] which stands for $Adam$ with $Nesterov$ momentum. We thought that this could help escaping plateaus and therefore reaching better regions of the objective function. This proved to be helpful, but not as much as we expected. Moreover we tried different combinations of dropout layers and rates, the optimal one was to set a 0.1 dropout layer right after the embedding, along with a 0.6 dropout after the GRU layers. We finally added $BatchNormalization$ layers, which can act as regularizers, being able to help preventing over-fitting, or at least speeding up the training. We also set a high number of epochs with a large patience value, in order to better pass low gradient regions. Everything put together allowed us to reach a 0.31 MSE for target 2 (instead of the 0.39 with the baseline model). This tiny improvements made us gain 10 places in the final public ranking.

4 Conclusion and perspectives

As a conclusion, combining all of what we discussed, we finally managed to get the following MSE scores for each target:

t0	**t1**	**t2**	**t3**
0.12	0.06	0.31	0.02

Which aggregate to a **final average MSE around** 0.13, along with a 10^{th} ranking on the public leaderboard.

We are quite satisfied of the models and adapted preprocessing we produced overall, but also aware that these scores, especially the MSE for the target t_2, could be further greatly improved. As you could see in this report, we did try a lot of different methods, but it might have been worth exploring others to do so. From the pseudo-document sampling point of view, a perspective for improvement might be to consider a random amount of sentences among documents, as well as trying to further enrich the embeddings that were initially provided to us for the nodes (either using node2vec or methods like *deep walk*). Within the architecture, we could have also tried out using multiple context vectors, both at the word and sentence levels. It might also have been worth considering encoding sentences s_i's by introducing some dependencies between them, instead of encoding each sentence in isolation. Among the paths that we could have explored are also the addition of "skip-connections", which were proven useful when training deep neural nets. We could also have tried to use $LSTM$ cells instead of GRU's, which, as being more complex, could hypothetically capture more specificity in the data, even if the training might have been harder. We could have finally tried to add metadata about the graphs directly to their encoded vector v before performing the final regressions, as well as trying methods more complex than a 1-layer MLP to perform them.

[1]http://cs229.stanford.edu/proj2015/054_report.pdf

References

[1] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, Eduard Hovy (2016), Hierarchical Attention Networks for Document Classification. https://www.cs.cmu.edu/~diyiy/docs/naacl16.pdf

[2] Aditya Grover, Jure Leskovec (2016), node2vec: Scalable Feature Learning for Networks. https://cs.stanford.edu/people/jure/pubs/node2vec-kdd16.pdf