Predict continuous values associated with graphs

Team: Activators (Wang SUN , Bolong ZHANG)
Paris-Saclay University, M2 Data Sciences
kevin.wang.sun@gmail.com, zhang.bolong@outlook.com

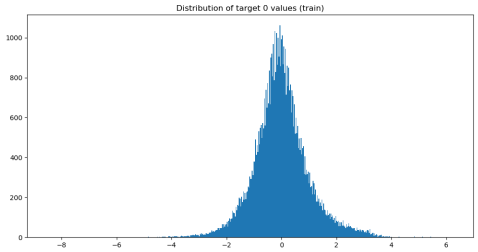
Abstract

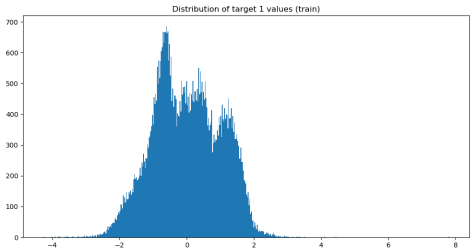
Our project focuses on the Hierarchical Attention Networks (HAN) architecture proposed by [Zichao Yang and Hovy, 2016]. We did experiments mainly on node embeddings and architecture modifications. Our final score is 0.15962 for private and 0.15509 for public.

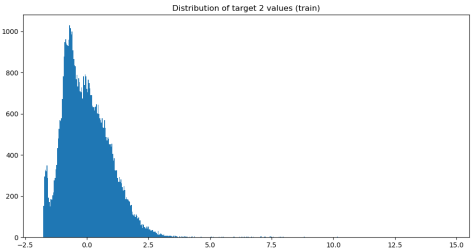
1 Target exploration

We did some explorations on 4 targets. This is a regression problem for all 4 targets. Their value ranges are not constraint in [0, 1]. We drew value distribution of training and validation set for all 4 targets. Considering the similarity of distributions between training and validation set, we show here only 4 training distributions.

dataset	min	mean	max
target 0 train	-8.5229	-0.0022	6.248
target 0 val	-6.5438	0.0081	5.63349
target 1 train	-4.2678	0.0055	7.8074
target 1 val	-4.4004	-0.0142	4.8213
target 2 train	-1.7682	-0.0051	14.6996
target 2 val	-1.7682	0.0110	12.5632
target 3 train	-7.5642	-0.0078	8.8621
target 3 val	-6.5831	0.0023	8.2821







2 Preprocessing

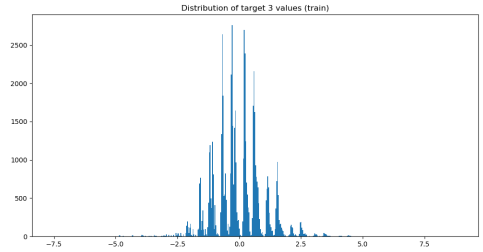
2.1 Sampling

node2vec

We use $node2vec$ to simulate biased random walks. It provides a way of balancing the exploration-exploitation tradeoff that in turn leads to representations obeying a spectrum of equivalences from homophily to structural equivalence. We have tried different values for p and q, and finally take $p = 2$ and $q = 0.5$, which encourages random walks to 'go further' instead of 'going back'. This may help explore global structure of the graph.

Other hyperparameters

We also tested different number of walks per node, length of walk and doc size. Obviously, the doc size depends on the number of walks per node. The smaller number of walks is, the faster our training will be. However, we still need to have enough walks to train the model, meanwhile it seems that too many walks or docs would not help. So we find the tradeoff and take 5 as number of walks per node, 50 as maximum doc size and 10 as walk length.



By the way, we have also tried to set the walk length as random value in a range, just defining minimum walk length and maximum walk length, as a result, it seems a fix length is slightly better.

2.2 Enriching node attribute vectors

Node embeddings

We still use here $node2vec$ to learn low-dimensional representations for nodes in a graph by optimizing a neighborhood preserving objective. The parameters is the same as mentioned above in sampling. For embedding dimension, we have tested 4, 8 and 16. As the size of each graph is not big, 8 is enough to represent the node.

Weisfeiler-Lehman relabeling

Weisfeiler-Lehman relabeling from [Nino Shervashidze and Borgwardt, 2011] modifies the label of each node in graphs. We take the degree here as the label of node. Firstly, it replaces the label of each node with a multiset label consisting of the original label of the node and the sorted set of labels of its neighbors. Secondly, the resultant multiset is compressed into a new, short label. This relabeling procedure is then repeated for 10 iterations in our works. We have implemented this relabeling procedure and applied it as a node attribute.

New features

Moreover, we have added new attributes for node representation. These new features include degree centrality, closeness centrality, betweenness centrality, graph density and graph size. Betweenness and closeness for current flow are also added. Inspired by the course, we also take k-cores and PageRank of graphs into account. Intuitively, the more information we have in dataset, the more precise model prediction will be.

Note that in the final step of preprocessing, we have scaled our new embedding matrix to make each column in the same order. Of course, the last line is always zero for padding.

As mentioned above, we have done so many experiments for different hyperparameters and combinations that there is no enough space to show it all here. So we selected some typical results to demonstrate our approach.

You can see below the validation loss for target 2 with different preprocessing approach. We test on different choice of p, q, max and min walk length (max/min), max doc size (doc), scale or not $(scale)$, add new features or not (fea). All the other hyperparameters are the same. The architecture is the same as baseline, just removing the $sigmoid$ activation function in the last dense layer.

p	q	min	max	doc	scale	fea	loss
baseline preprocessing							0.847
2	0.5	6	12	90	no	no	0.596
2	0.5	6	12	90	yes	no	0.512
2	0.5	10		50	yes	no	0.462
1	1	10		50	yes	no	0.486
2	0.5	10		50	yes	yes	0.433

By the way, adding new features gives an impressive improvement on model 3.

3 Training

Because of so many experiments on sampling, node embeddings and architecture, we want a fast and suitable training process to see the tendency of performance and then fine-tune for a better model. So we set training parameters as follows. Dropout is set 0.3 after experiments on several different architectures.

parameters	value
batch size	128/256
epochs	100
patience	4
optimizer	adam
dropout	0.3

4 Architecture

4.1 Last layers

The baseline uses $sigmoid$ as activation function in the last layer which outputs only prediction from 0 to 1. This is not suitable for these four cases. Since it's a regression problem, we remove the $sigmoid$ activation function (or use linear function) in the last layer and keep 1 as unit number. As we know, deeper network may be more expressive. We have tested different combinations of activation function, unit number, layer number in last dense layers, using baseline architecture, the same embeddings and documents.

After comparing $linear$, $sigmoid$, $tanh$, $ReLU$ and $LeakyReLU$, here are some typical results (validation loss) tested on target 2 in 10 epochs.

last layers	loss
sigmoid (1 unit)	0.773
linear (1 unit)	0.532
sigmoid (4 units) + linear (1 unit)	0.539
relu (4 units) + linear (1 unit)	0.538
tanh (4 units) + linear (1 unit)	0.504
tanh (8 units) + linear (1 unit)	0.507
1 LeakyReLU layer + linear (1 unit)	0.505
2 LeakyReLU layers + linear (1 unit)	0.485

So in our final version, we add 2 $LeakyReLU$ layers $(alpha = 0.01)$ before the last dense layer to make our model more expressive. In the following sections, we will change to test our architecture trials on target 0.

4.2 Skip Connection

We have tested the possibility of adding *Skip Connection* in HAN. As shown in Figure 1, for sentence encoder, the input vectors of bidirectional GRU layer are averaged and then added to the output vector of $AttentionWithContext$ layer as the sentence vector. It works in a similar way for document encoder. To realize this function, we wrote a custom layer called $SkipConnection$. According to the results on target 0, model with $SkipConnection$ on both sentence and document levels gives the best performance. Hence, we will apply $SkipConnection$ on both two levels on models after. Three executive files here are

$modif\_baseline\_0.py$,
$sentskip\_baseline\_0.py$,
$sentdocskip\_baseline\_0.py$.

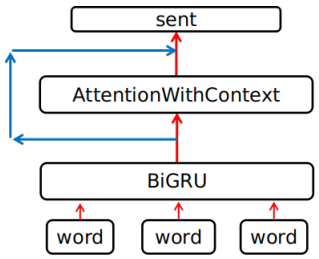


Figure 1: $SkipConnection$ in sentence encoder.

SkipConnection	target 0 train loss	target 0 val loss
none	0.1584	0.159
only on sent	0.1828	0.1779
sent and doc	**0.133**	**0.1497**

4.3 Self Attention

The original HAN uses self-attention in both sentence encoder and document encoder. The vector u to compute the coefficient of each word/sentence vectors for both sentence and document encoder is randomly initialized and then learned while training. Thus, one of our research directions is to use different u in both sentence encoder and document encoder.

Self vectors

We sought to replace randomly initialized u with the self vectors. Originally, the output vectors of bidirectional GRU layer compute coefficients with u directly. We replaced u with the average of these output vectors to compute the coefficients and then get the weighted sum as final embedding. We wrote a layer called $SelfAttention$ and applied it to both sentence and document encoder. The result on target 0 shows that this method does not give a better performance. Three executive files here are

$sentdocskip\_baseline\_0.py$,
$sentSelfAtt\_0.py$,
$sentdocSelfAtt\_0.py$.

Self vectors	target 0 train loss	target 0 val loss
none	**0.133**	**0.1497**
only on sent	0.1574	0.1631
sent and doc	0.1468	0.1572

4.4 Replace u in document encoder

Another idea is to use other sentence encoders to compute u. In this case, only u in document encoder can be replaced. As shown in Figure 2, we replace $sent$ with different sentence encoders. If we want to replace u in this way in sentence encoder, it should use different word encoder. We haven't tried it here.

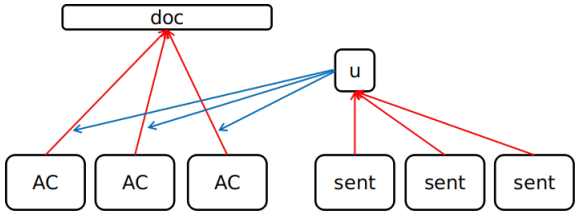


Figure 2: Document encoder with u replaced. AC: AC sentence encoder in main structure, it can be replaced with SSA; sent: sentence encoder for u, it can be $BiLSTM$, AC or SSA.

BiLSTM encoder

We first try $BiLSTM$ encoder as $sent$ and keep AC as sentence encoder in main structure. The executive file here is $ac\_lstm\_0.py$.

AttentionWithContext (AC) encoder

We also attempted the same AC sentence encoder for u but initialized and trained separately from those AC in main structure. The executive file here is $ac\_ac\_0.py$.

StructuredSelfAttentive (SSA) encoder

Inspired by structured sentence embedding proposed by [Zhouhan Lin and Bengio, 2017], we wrote by ourselves a SSA layer as a new sentence encoder and then applied on $sent$. AC is kept in main structure. The executive file here is $ac\_ssa\_0.py$.

Moreover, We also replaced AC in main structure with SSA as a try. The executive file here is $ssa\_ssa\_0.py$.

target 0	epochs	train loss	val loss
sentdocskip_baseline	46	0.133	0.1497
ac_lstm	26	0.164	0.1796
ac_ac	13	0.2159	0.2269
ac_ssa	9	0.232	0.2554
ssa_ssa	8	0.2439	0.2368

Models above used similar parameters. We find that these document encoders with replaced u reach a bad local optimal quickly under current training strategy. So we fine-tune them for further convergence.

4.5 Replace u in sentence encoder

In section 3.3, methods only replaces u in document encoder. Naturally, another research direction falls in replacing u in sentence encoder. In previous architectures, sentence encoder concentrates only on its own words without taking its neighbor sentences into account. Intuitively, sentence encoder considering contextual information will give a different view.

Documents preparation

After several attempts of implementation, we finally constructed another documents' set because of limitation of wrapper $TimeDistributed$ in Keras on multi and shared input. Documents' file that we used for architectures before has shape (93719, 50, 10), which means that there are 93719 documents, each document has 50 sentences and each sentence has 10 words (including padding). For a window with size 6, the reconstructed documents' file should have shape (93719, 50, 7, 10), which means that each sentence will attached with its 6 neighbor sentences in order. The executive file $prepare\_contextual\_docs.py$ serves for this reconstruction procedure.

Contextual sentence encoder

After document reconstruction, contextual sentence encoder can be shown as Figure 3. For the current sentence, its words are encoded by $BiGRU$ as before while u is replaced with the combination of neighbor sentence vectors. Neighbor sentence can be encoded with AC, CNN or SSA.

Note that, according to our experiments, $SkipConnection$ in AC for neighbor sentence will cause gradient vanishing or explosion problem which is indicated by NaN of loss when training. $SkipConnection$ in the document encoder will also cause bad performance. So we keep only $SkipConnection$ for the main structure of current sentence encoder, which brings an augmentation of performance.

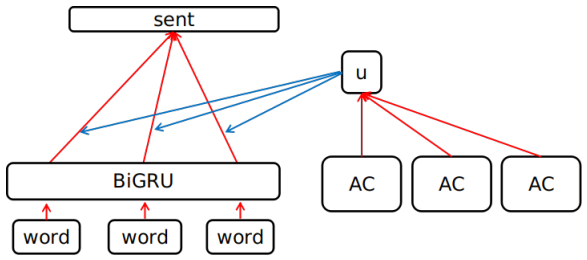


Figure 3: Contextual sentence encoder with u replaced. AC: AC sentence encoder for neighbor sentences of current sentence, can be replaced with CNN or SSA.

The most common combination of neighbor sentence vectors is average. With respectively AC, CNN, and SSA as neighbor sentence encoder, models' scores on target 0 are shown below. Three executive files here are

$$sentACcontext\_docAC\_0.py,$$
$$sentCNNcontext\_docAC\_0.py,$$
$$sentSScontext\_docAC\_0.py.$$

In contextual sentence encoder, we also attempted to use a matrix rather than a vector for neighbor sentence encoder to represent multi-context information, inspired by [Zhouhan Lin and Bengio, 2017]. However, we didn't succeed because of limitation of wrapper $TimeDistributed$ on treating 4D tensor. The executive file is $sentSSAmulticontext\_docAC\_0.py$.

Moreover, we sought the possibility of CNN as combination for neighbor sentence encoders. Because of NaN of loss when training, optimizer here is changed to SGD. Three executive files here are

$$sentACcontextCNN\_docAC\_0.py,$$
$$sentCNNcontextCNN\_docAC\_0.py,$$
$$sentSScontextCNN\_docAC\_0.py.$$

target 0	epochs	train loss	val loss
sentdocskip_baseline	46	0.133	0.1497
sentACcontext_docAC	11	0.2134	0.2206
sentCNNcontext˜	24	0.1752	0.1734
sentSScontext˜	9	0.2212	0.2245
sentACcontextCNN˜	5	0.3218	0.3003
sentCNNcontextCNN˜	15	0.2717	0.2554
sentSScontextCNN˜	5	0.3218	0.3006

We find that under current parameters and training strategy, contextual sentence encoder does not bring a satisfying performance. However, the application of CNN is obviously potential. Modification of last layer and fine-tuning on layer number, unit number, filter size and filter number can probably give a good performance.

4.6 Replace u in document & sentence encoder

Naturally, after section 3.4 and 3.5, another idea is to combine both two methods: replace u in both document and sentence encoder. Considering the complexity of architecture and the large number of parameters, we didn't test too much. We just attempt two architectures on target 2. They do not have better performance, which indicates that complex architectures are not always better than simple ones.

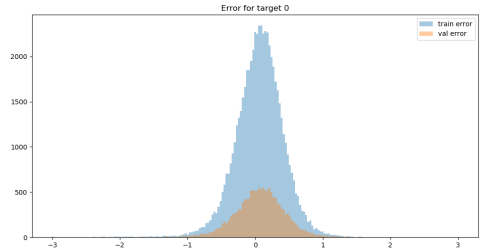
5 Fine-tuning

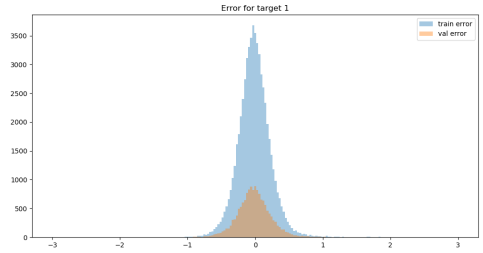
According to above tests on target 0, we first chose ac_ssa model to do fine-tuning. Similarly, it removes the $sigmoid$ activation function in the last dense layer and adds 2 $LeakyReLU$ layer with $alpha = 0.01$ before the last dense layer. $SkipConnection$ is only added to sentence encoder and sentence encoder for u but not to document encoder. Sentence encoder is a AC encoder with 2 $BiGRU$ layers. Each layer has 60 units. Sentence encoder for u is a SSA encoder with 2 $BiLSTM$ layers. Each layer has 100 units. Its other two parameters $da = 15$ and $r = 10$. Document encoder is a AC encoder with 2 $BiGRU$ layers. Each layer has 60 units.

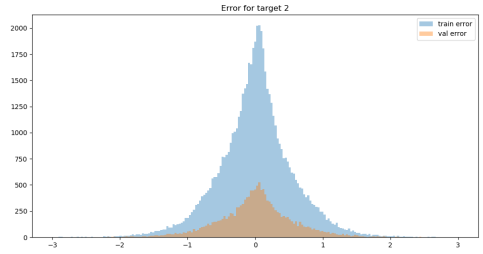
ac_ssa	train loss	val loss
target 0	0.158	0.1607
target 1	0.0764	0.07343
target 2	0.3658	0.37866
target 3	0.0312	0.00829

We used this model's predictions as our final submission on Kaggle and it has scores 0.15962 for private and 0.15509 for

public. Additionally, we drew error distribution of training and validation set for all 4 targets.





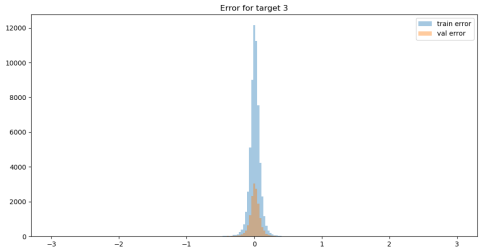


Then we chose a simpler model $sentdocskip\_baseline$ to do fine-tuning. This model is based on the baseline architecture. It removes the $sigmoid$ activation function in the last dense layer and adds 2 $LeakyReLU$ layers with $alpha = 0.01$ before the last dense layer. $SkipConnection$ is added on both sentence and document encoder. Sentence encoder has 80 units in its single $BiGRU$ layer while document encoder has 100 units in its single $BiGRU$ layer.

We can see that this simpler model gives better performance than ac_ssa model. But without finishing the training on target 3 before the deadline, we cannot use it as our submission.

6 Conclusion

Our works focused on mainly two parts: preprocessing and architecture. In preprocessing, we made trials on adding node attributes and create node embeddings. We did hyperparameters tuning while creating node embeddings. Along with $node2vec$, we did also sampling of walks as documents. According to [Abu-El-Haija *et al.*, 2018], these hyperparameters can be set trainable and earned via back propagation. This may be a good direction because their proposition can produce node embeddings that best-preserve the graph structure and also generalize to unseen information. Attention mechanism is also used in their model.



$sentdocskip\_baseline$	train loss	val loss
target 0	0.111	0.1371
target 1	0.0499	0.0667
target 2	0.2665	0.3326
target 3	0.0151	0.0095

Besides, architecture is quite important in this project but not the most complex architecture will give the best performance. Our ideas are interesting but still need time to tune details for better performance.

References

[Abu-El-Haija *et al.*, 2018] Sami Abu-El-Haija, Bryan Perozzi, Rami Al-Rfou, and Alex Alemi. Watch your step: Learning node embeddings via graph attention. *Neural Information Processing Systems*, 2018.

[Nino Shervashidze and Borgwardt, 2011] Erik Jan van Van Leeuwen Kurt Mehlhorn Nino Shervashidze, Pascal Schweitzer and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12:2539–2561, 2011.

[Zhouhan Lin and Bengio, 2017] Cicero Nogueira dos Santos Mo Yu Bing Xiang Bowen Zhou Zhouhan Lin, Minwei Feng and Yoshua Bengio. A structured self-attentive sentence embedding. *ICLR*, 2017.

[Zichao Yang and Hovy, 2016] Chris Dyer Xiaodong He Alex Smola Zichao Yang, Diyi Yang 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*, 2016.