R228 Deep Learning for NLP Practical Exercises Report

Andi Zhang [1]

Abstract

In these practical exercises, we work around a program for predicting dictionary head words given in a definition. The evaluation metric refers to the rank of the true head word in the list of predicted head words. The size of the validation set used in this work is 200, whereas the evaluation score is the median rank of such 200 cases. If we use the default hyper-parameters of the given code, the evaluation score comes to around 5000. Then after some hyper-parameter adjustment, the evaluation score is decreased to 500. At last, we did some more experiments on the structure of deep neural network in this work, with the best score being less than 230, which is much better than the baseline.

Introduction

This is a report relating to R228 Deep Learning for Natural Language Processing's practical exercises. This report comprises of three parts: evaluation function, parameter adjustment and investigation on structure. In the first part, we implemented an evaluation function introduced in practical instructions. Then in the second part, we adjusted several hyper-parameters to enable the model to perform much better on efficiency as well as our evaluation metric. Then in the third section, we changed the structure of the encoder to obtain further improvements.

Problem

In this work, our task is to predict the head given a gloss (definition). An example is:

```
<fawn, a young deer>
```

In this example, "a young deer" is the gloss and "fawn" is the head.

[1]University of Cambridge, Cambridge, United Kingdom. Correspondence to: Andi Zhang <az381@cam.ac.uk>.

Assessment of R228 Deep Learning for Natural Language Processing, University of Cambridge, United Kingdom, 2017.

Dataset

The training dataset contains 367,347 pairs of glosses and heads. The given code preprocess the pairs:

- Build a 100,000 sized vocabulary (cut off low frequency words).
- Transfer the glosses and heads to ids (0 for padding, 1 for unknown words).
- Pad the glosses to have a length of 20.

The aforementioned example after preprocess is:

```
<11440, 2 415 2534 0 0 0 ...  0>
```

Similarly, the validation dataset (dev dataset) comprises of 200 pairs of glosses and heads, which are also preprocessed.

It is noteworthy that only 71 of the 367,347 head words in the training data set are out of the vocabulary, which is small enough to be ignored. All the head words within the dev dataset are in the vocabulary.

Pre-trained Word Embedding

This work makes use of pretrained word embedding as well. The given code uses pretrained Google Word2Vec (Mikolov et al., 2013) as its word embedding; here, we follow this idea.

For our 100,000 sized vocabulary, 53,738 words have the pretrained word embedding, while nearly half of the words do not. However, this might not be a very big problem since all the head words do have the embedding (except the 71 out of vocabulary words). With regard to the 200 head words of validation dataset, 23 of them do not have pretrained word embedding, which will not be a major influence on the median rank - our evaluation metric.

The Given model

Figure 1 shows the default model of the given code, which use a recurrent neural network (RNN) as the encoder and cosine distance as the loss function. The cosine distance is between the predicted vector and the word embedding of the true head (out_emb).

By choosing these hyper parameters, we can also use cross-entropy as the loss function and bag-of-words (BOW) as the encoder, which will be discussed in part II of this report.

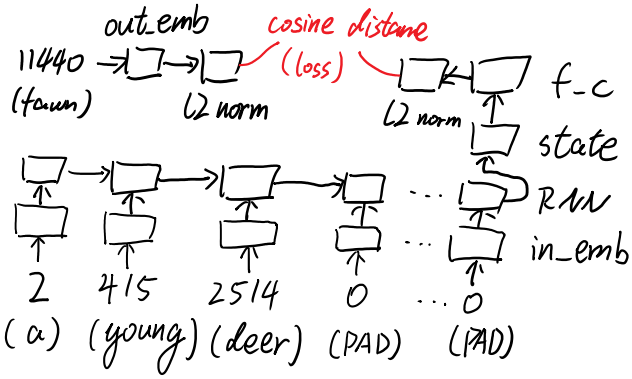

Figure 1. The given model (RNN encoder, cosine loss)

The Experiment Environment

We claim the kind of machine used within this work because we are using "training time" as a metric in the remaining parts. All the experiments of this work run on Azure's Standard_NC6 virtual machine, which features 6 v-CPU and 1 GPU. The GPU is one-half K80 card.

Part I: Evaluation Function

The first part of this practice is to implement an evaluation function for the given code.

The process of evaluation can be described as follows:

- Get the predictions of the model: if the loss is cosine distance, get the output of the fully connection layer (a 300-dim vector); if the loss is softmax, get the output of softmax layer (a 100,000-dim vector).
- Sort the predictions: if the loss is cosine distance, calculate the cosine distance between all the words' embedding and the predicted vector, then sort them from the nearest to the farthest; if the loss is softmax, merely sorting the probabilities will suffice.
- Get rank of the true head: As in practice, we are using the `argsort`, which is a straight forward process to obtain the position of the true head in the sorted list.

The code of this implementation is enlisted in the appendix. It is also been uploaded to Github, the url is given in the end of this report.

We train the given model with default hyper parameters (except `num_epochs`, which is chosen to be 11), then evaluate it by our implemented evaluation function.

Figure 2 shows the output of the evaluation function. The list containing 200 integers which are the ranks of each head within the validation set. The number in the last line is the median number of these 200 ranks, showing that the evaluation score of the validation set is 4914.5.

```
[10151 54213  1010  2150  2726 33700  3128   812 54139   828 58004 12245
  1393 28556 15797 51346 16806    48   336   776 46292   536  7657 16073
 11111 55632  5044 11008 55456 52936 58139   462   239 42549  2327 10808
  3997 55765     2     9   214  6493   379  1225   127 31185     1   343
 44591   659 29317   186   917   473 54141 25196   130  5248  1979 37867
 22319  3133  3133  4691    10 16290   815   130    70  4492 73223     3
 25796 41927  2164 20409  6984 15542  3366 55099  3735  1227 48965  9859
 10875   173 42609   545 13023  1415   276   125   121    26  2184 21213
  6055    81   996   201   502 10764  5089  1621  2068 51371  5899 23398
    90    14  6608 10602 50596   903  8960   123  8071 30521    16 44040
 10394   192 55898 10535 12663 20711 56245 55420 18116 11867 17185   157
  1937 39109  2970   681  1724  2415  9278  4102 58223   881 17617 10643
 13710  1083  5454 57470  9421    38  1488     3    85 12980 39229   411
 13240 12694  7390  1632     9  5490 18863 33386   219  6122 50149   544
  3035 22762  4588  2430 39502 20886   538   180  2032 55759  4785 54881
 54108  4320 54414   794  3355  3180  1870 20969  1696 47775  9904  2790
    13  8195  6725  1052  1827   757    93   497]
4914.5
```

Figure 2. The output of the evaluation function

From the result, we can see that in some cases the model has very good behaviour. Figure 3 shows a positive case: the gloss is "to make something better" and the true head is "improve", which is ranked fourth by our model.

```
Type a definition: to make something better
Number of candidates: 5
Top 5 baseline candidates:
1: better
2: something
3: prefer
4: deserved
5: lacking

 Top 5 candidates from the model:
1: reassess
2: standardise
3: augment
4: improve
5: strengthen
```

Figure 3. A positive example.

Figure 4 illustrates a negative example: the true head "grow" ranked more than 50,000 by our model.

```
Type a definition: when a living thing gets bigger naturally
Number of candidates: 5
Top 5 baseline candidates:
1: naturally
2: a
3: vegan
4: summertime
5: motto

 Top 5 candidates from the model:
1: mycotoxin
2: fungal
3: keratitis
4: carotenoid
5: differentially
```

Figure 4. A negative example

In order to find out the reason why these two words has such a big difference in ranking, we look back at the training data. As is shown in figure 5, there are several glosses of "improve" in the training dataset. In fact, no gloss in the

training set is exactly same as the "to make something better" in the validation set. However, there is a gloss in the training dataset containing the words "to make something better", which is highlighted in figure 5. On the other hand, the word "grow" is not in the training set as a head word.

```
improve to raise to a more desirable or more excellent quality or condition make better
improve to increase the productivity or value of land or property
improve to put to good use use profitably
improve to become better
improve to make beneficial additions or changes improve on the translation
improve to make something better to increase the value or productivity of something
improve to become better
improve to grow better to advance or make progress in what is desirable to make or show improvement
improve to advance or progress in bad qualities to grow worse
improve to increase to be enhanced to rise in value
```

Figure 5. "improve" in training data

Hence, we can deduce that the model is good at "fuzzy memorizing" but not understanding. This discussion will be continued in the "Final Model" part of this report.

Part II: Parameter Adjustment

In this part, we have focused on increasing the evaluation score (implemented in part I) of our model. As there are too many hyper-parameters, we cannot use grid search to enumerate all the choices. Hence, our greedy strategy here is to adjust them one by one: Firstly, find the parameter with the greatest influence at the moment. Then find the best value for that particular parameter and fix it. Repeat the steps until all the parameters are adjusted.

Number of Epochs

The first hyper-parameter we have to decide on is the number of epochs. It is impossible to decide a fixed number of epochs because we cannot predict the point that our model converges to. Hence, we are using a dynamic number of epochs here (as always used in practice): we calculated the evaluation score defined in part I after every epoch, then we recorded the best dev evaluate score, if the dev evaluate score has not been better for n epochs, we terminate the training. In this work, n is chosen to be 6.

Shuffle Data

Bengio indicated that the data needs to be shuffled before every epoch (2012). The given code has a function `gen_epoch`, which invokes `gen_batch` to generate the batches. We notice that the code does not shuffle the data at all, which is why we add the shuffling to the code. The comparison between our shuffled version and the original version is shown in figure 6, the shuffled version converges faster and the best dev evaluation score is also shown to be better.

Classification vs Regression

The argument `--pretrained_target` decides the output form: cosine or softmax. If the output form is cosine, this is a 300-dimension regression model; otherwise this is a classification model with 100,000 classes. By intuition, the 100,000-classes classification is intractable, however, we still have the experiment and the result is shown in figure 7. Both train evaluation score and dev evaluation score of the classification model is poor, only a little bit better than the randomly initialized model. Hence we will always use the regression model in the remainder of this work.

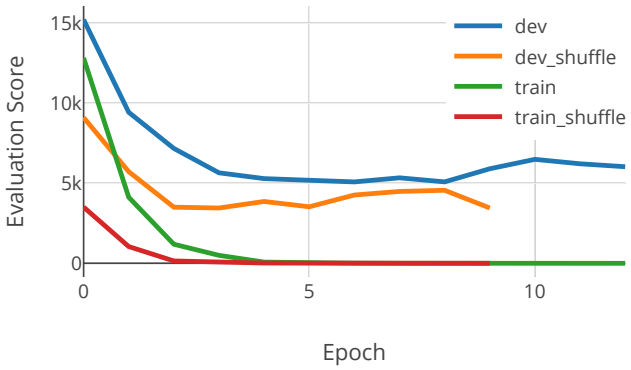

Figure 6. Evaluation scores before and after shuffle

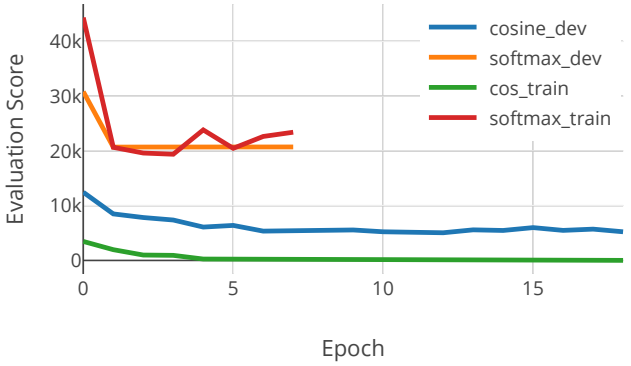

Figure 7. Classification (softmax) vs Regression (cosine)

Word Embedding

We use "in embedding" to express the word embedding of gloss words and "out embedding" to express the word embedding for head words (corresponding to `in_emb` and `out_emb` in figure 1). In the given code, we can only calculate the cosine distance between the output of the fully connected layer and the out embedding if we want to do regression, which is hard coded. Besides, the out embedding is not trainable since we are using the numpy matrix `pre_embs` in our evaluation and query functions. To let this interface become more flexible to add trainable out embedding in our experiment, we make the following modifications:

- Separate the in embedding and out embedding if the arguments "pretrained_input" and "pretrained_output" are both selected.
- Let the pre-trained word embedding be trainable.
- Use random initialized word embedding if pretrained is not ticked.
- At this time, the out embedding is dynamic during the training process, so we have to load the out embedding from Tensorflow but do not use the static numpy matrix in evaluation and query functions.

As these modifications will change the original code structure as well as our solution to Part I, we apply them on a copy of the original version.

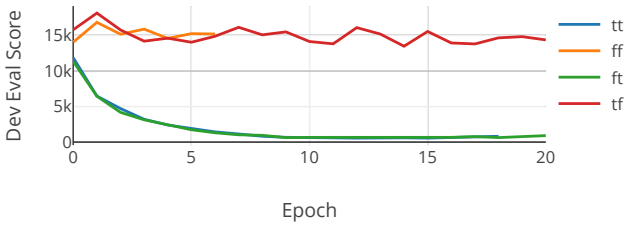

Figure 8. Evaluation scores of trainable word embeddings

Figure 8 shows the performance of trainable word embeddings. "tt" stands for both in embedding and out embedding are pre-trained, "tf" means that the in embedding is pre-trained but out embedding is not. "ft" and "ff" are similar. From the figure, we know that if the out embedding is not pre-trained, the model will behave poorly. In contrast, the in embedding does not have a major effect.

In conclusion, the trainable out embedding increases the best evaluated score of our model from 3000 to 500, which is a very big increase.

Dropout

In this work, we add dropout after the fully connected layer. The implementation of dropout in our work is a bit more than switching the argument: we have to declare a new place holder `dropout_keep_prob` and then feed the keep probability into `feed_dict` of `sess.run()` during the training process; for the purpose of predicting, we can just feed 1.0 to that placeholder.

Figure 9 illustrates the performance of dropout. We do not find a very significant difference between 50% keep probability and 75% keep probability, while the model without dropout behaves worse. Hence, we think that 75% keep probability is a good choice for our model.

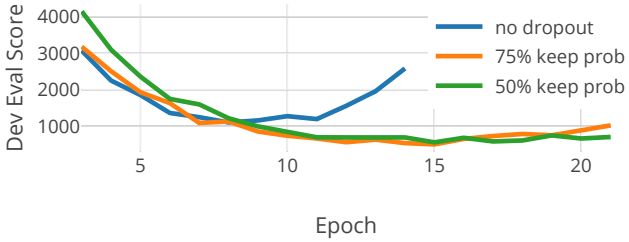

Figure 9. Eval scores by different dropout keep probabilities

Activation Function

The given code uses activation function after the fully-connected layer. Since this is a regression model, we think that the linear activation might behave well. Hence, here we have tried three activation functions: Linear, Relu and Tanh. Figure 10 shows the result: Relu is shown to be the fastest with best score 607.0; Linear is the slowest with a best score of 619.5; the speed of Tanh lies in the middle, but its best score is 511.0. Hence, Tanh behaves best in this experiment and we will keep using it in the remainder of this work.

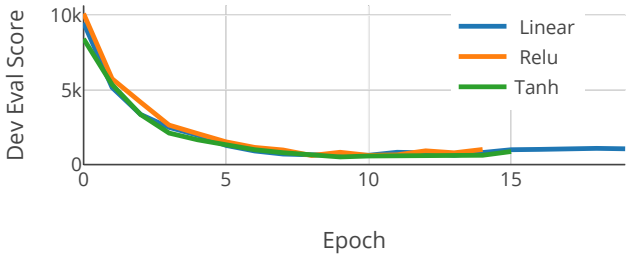

Figure 10. Evaluation scores by different activations

Optimizer

In this experiment, we try three popular optimizers: stochastic gradient descent (SGD), Root Mean Square Propagation (RMSProp) and Adaptive Moment Estimation (Adam). For SGD, the scores are worse than randomly initialized model, hence it was terminated after epoch 5, which is depicted as a short green line in figure 11. The model converges when using RMSProp and Adam. It can also be seen in figure 11 that Adam does better in both speed and performance. (The best dev score of Adam is 580.5, while the RMSProp's is 1760.0; each epoch running by Adam takes 4 minutes while RMSProp takes 3 minutes)

Batch size

In fact, during these experiments, there is no evidence shows that the batch size influences the performance (evaluation score) of the model. However, it does influence the training time - smaller batch takes more time. The detail is shown in table 1.

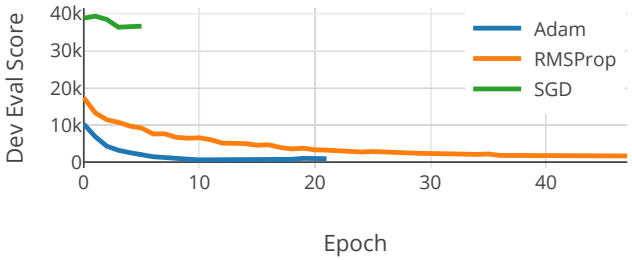

Figure 11. Dev Evaluation scores when using different optimizers

Table 1. Comparason between different batch sizes

SIZE	TIME / EPOCH	NUM EPOCH	BEST EVAL
32	723 S	8	638.0
64	406 S	9	511.0
128	251 S	12	558.0
256	176 S	16	496.5

As illustrated in table 1, the batch size of 256 is nearly four times faster than the batch size of 32 on a epoch. This gives us a tip: when implementing some new structures of model in the following experiments, we could use a larger batch size initially to see if the model actually works.

Part III: Investigation on Structure

In this part, we will carry out some more experiments on modifying the structure of the neural network. As is shown in figure 12, there are seven structures participated in the comparison (including two given model: RNN and BOW).

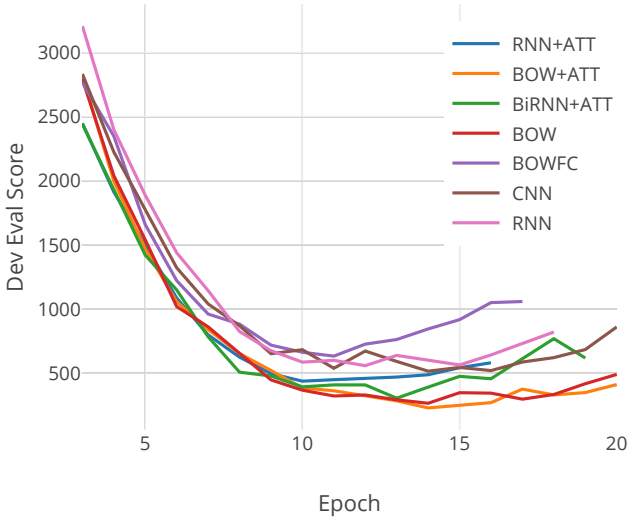

Figure 12. Dev Eval Scores of the models discussed in part III

Bag of Words (BOW)

The bag of words (BOW) structure is shown in figure 13. Instead of using the recurrent neural network, this encoder only averages the input embeddings, then feed the mean into the full connected layer. This is a very simple model in the given code, while it has a very good performance in this task.

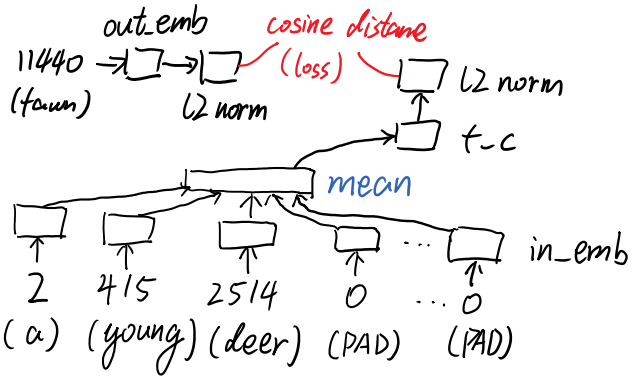

Figure 13. Bag of words

Bag of Words - Fully Connected (BOWFC)

Considering the fact that BOW encoder is too simple and the average operation will cause a loss in information. A straight forward idea is to use concatenate instead of mean (as is shown in figure 14). We call this modification "bag of words - fully connected" because it connect all the word embeddings to the fully connected layer. However, in the experiment this model is found to easily get overfitted because the connections are too dense.

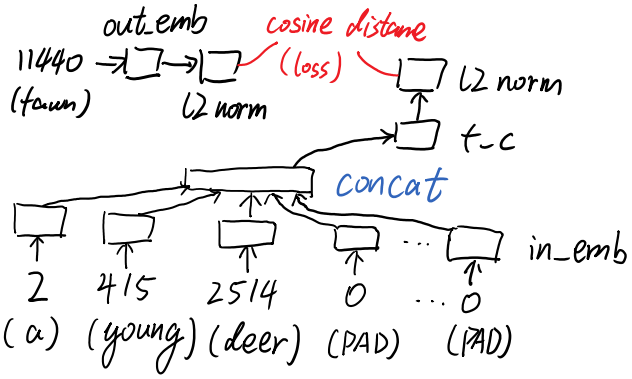

Figure 14. Bag of words - fully connected

Text-CNN

In order to make a sparser connection than BOWFC, we try the Text-CNN (Kim, 2014) on our task, which has two convolutional layers, with the window size 1 and 300 filters

in each layer. The window size is chosen to be 1 because of the text is not longer than 20, so we tried 1,2 and 3 as the window size and found 1 is best for this task. The number of filters is chosen to be 300 because this is a regression model, and every dimension should be predicted, we do not want the dimension of the hidden layer to be less than the dimension of the output. As shown in figure 12, the performance of Text-CNN turns out to be better than BOWFC and RNN.

RNN, Bidirectional RNN with Attention

Inspired by sequence to sequence model (Sutskever et al., 2014) and attention mechanism (Bahdanau et al., 2014), here we implement the attention on the top of RNN, as shown in figure 15. We train an attention mechanism and then use the attention probabilities to build a context vector, which is feed into the fully connected layer. Moreover, we also try the bidirectional RNN: feed the average of forward output and backward output to the attention mechanism, which has a better result, as shown in figure 12.

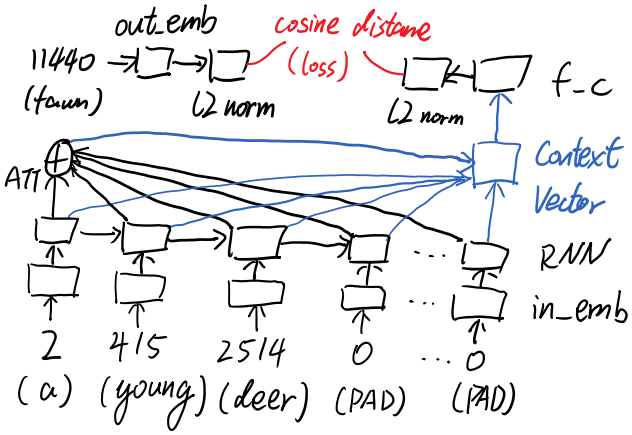

Figure 15. RNN with Attention

"Attention is all you Need" (BOW + ATT)

As the problem is about working on short text, we find out that the order of the words are not very important by observing the cases. Hence, it is acceptable to get rid of the recurrent neural network and apply the attention on the in embedding directly, as shown in figure 16. Surprisingly, this model performs best in the evaluation score.

Final Model

Our final model uses 0.0001 as its learning rate, the in embedding and out embedding are both trainable, the dropout keep probability is fixed at 0.75, and the encoder is bag of the words with attention. The optimizer is Adam and the activation function is Tanh.

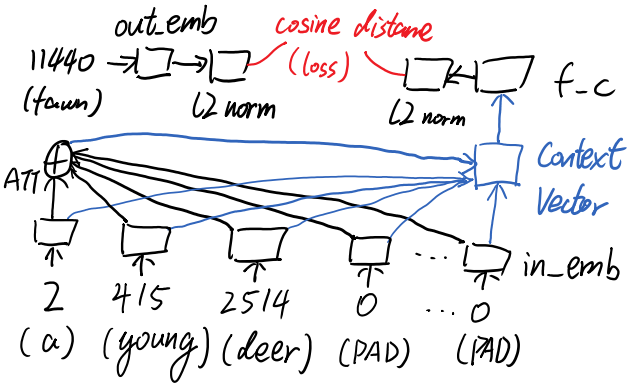

Figure 16. Bag of words with attention

The result of our final model is illustrated in figure 17. As compared to figure 1, our final model is much better in many cases. However, there is still some cases have a very large rank. Then we find that the cases that still ranked larger than 50,000 does not occurred in the training dataset as a head word at all. For these cases it is reasonable that our model does not have a good prediction. In all, our final model behaves well, with an evaluate score of 228.0.

```
[ 1630 53791     5   382     8   544   108   151 55306   276 60235   165
  1300  1119   280 58783   285     1  8578    51   195    22  1445    66
  2200 56200    27    40 56392 54879 60050     5   275   195     3   562
  2954 54734     2     2  1048   181     7     3    19   256     1   271
 42419     7   277 10682    34   108 53787     2    29   146     1   436
   186  1069     3     2  1100 14528   100  3161     3    16 73331     0
    14   533  1648 15925    54    14    22 56035    37   608  2421   170
   191   622  3094    50    12   215   171    48   129     4    41  4038
     1     9   439   777    23   193    20    25   890 57567  2031  8046
     8   154     1   150  4165   623    17   508     8   607    10  7661
  1382  5887 55191  1164     7  1393 54938 55744 10980    84    47     0
 42437  1523   580     2  3286   381     0     8 55634     2   238   116
   964   166  1159 54129  1708 25810     3     1     4  6447    35   719
    92    49  1285    30   115    84   115 12459    25  2372 54007   218
  1063  4173   791  2524  1663   164  7873   396 15870 59541    13 55790
 56049   209 53749     5   112     1 19319   419     2  3130   309   127
    57 35326     2   330     3    75    13     3]
228.0
```

Figure 17. The evaluation of our final model

Another interesting fact is that the heads of most of these positive cases occur 10 times in the training dataset, while the heads of low ranked cases occurs fewer times. For example, the word "week" is ranked 42,437 and it only occurs 2 times in the training set, while the word "improve" is ranked 0 and it occurs 10 times as a head word in the training set. This phenomenon shows that the data balance might also be a factor in influencing the evaluation score.

Conclusion

In a word, this is a short text regression problem and the order of the words are not very important. Hence the bag of word with attention has the best evaluation score in the comparison.

The pretrained word embedding plays a crucial rule in this work. As we learnt in part II, with out pretrained out embedding, the model will not be converged. Hence our

model is highly depend on the pretrained word embedding - but not at all, because if we set the embedding be trainable, the performance will improve considerably.

Let us explain this phenomenon, the pretrained word embeddings (Word2Vec, GloVe and so on) have an important property - word analogue. For example, king + woman - man = queen.

In essence, the bag of word with attention is the weighted sum of the word embeddings. Hence if the word embeddings have a good word analogue property, the performance of the model should be better. That is why using pretrained word embedding is better.

The pretrained word embeddings are trained by selected corpus. There exists difference between the corpus and the context of our training dataset. Therefore, if the word embeddings are trainable, it will be adjusted to our task better. That's why our final model have a good performance.

Future Work

Here is some ideas come out from this work, which might increase the performance of our model further.

Data Augment As we discussed before, the head words with fewer glosses in the training dataset are not well trained. Hence the label shuffle should be tried in this work.

Use GloVe As mentioned, this work is based on the word analogue property of the word embedding. In fact the Glove's word analogue property is better than Word2Vec. Hence trying Glove might be a better choice.

Replication

All the code of this work is uploaded to Github:

`https://github.com/andiac/R228DLNLP`

To replicate the experiments, please follow the instructions of the repository.

Acknowledgement

I would like to thank Dr. Stephen Clark for his advice and support in this work.

References

Bahdanau, Dzmitry, Cho, Kyunghyun, and Bengio, Yoshua. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Bengio, Yoshua. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pp. 437–478. Springer, 2012.

Kim, Yoon. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

Mikolov, Tomas, Chen, Kai, Corrado, Greg, and Dean, Jeffrey. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

Sutskever, Ilya, Vinyals, Oriol, and Le, Quoc V. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pp. 3104–3112, 2014.