NLP HW2

Tiago Antunes

April 10, 2021

1 Gradients Calculation

The cross entropy cost function is given by

$$J = -\sum_{i=1}^{W} y_i log(\hat{y_i})$$

which, given the softmax function, we can extend it to

$$J = -\sum_{i=1}^{W} y_i \log(\frac{exp(u_i^T v_c)}{\sum_{w=1}^{W} exp(u_w^T v_c)})$$

$$J = -\sum_{i=1}^{W} y_i [u_i^T v_c - log(\sum_{w=1}^{W} exp(u_w^T v_c))]$$

Since the skip-gram is using a one-hot encoding, only one element will have value 1, at index k. Both y and $\hat{y}$ have $|W|$ x 1 shape and so the final formula will be

$$J = -y_k [u_k^T v_c - log(\sum_{w=1}^{W} exp(u_w^T v_c))]$$

where the term y_k equals 1 and so can be omitted. Then we can calculate its gradient

$$\frac{\delta J}{\delta v_c} = -u_k + \frac{1}{\sum_{w=1}^{W} exp(u_w^T v_c)} \sum_{w=1}^{W} exp(u_w^T v_c) u_w$$

which can be written taking into account the relationship between the sum of division and the division of sums to have the format

$$\frac{\delta J}{\delta v_c} = -u_k + \sum_{w=1}^{W} (\frac{exp(u_w^T v_c)}{\sum_{x=1}^{W} exp(u_x^T v_c)} u_w)$$

thus yielding the final result

$$\frac{\delta J}{\delta v_c} = -u_k + \sum_{w=1}^{W} Pr(word_w | v_c, u) u_w$$

2 Word2Vec Implementation

2.1 Important note

For this part I implemented a SkipGram model with Negative Sampling. **Due to hardware limitations, I couldn't run the data on my local computer**. Google Colab, which was the tool I used to train the model, has a limit on the available RAM and on the disk space that exists. This overall limited my ability to train the model and was actually very time consuming to do.

The results below only represent training on 1/15 of the Wikipedia dataset, which after pre-processing resulted on having around 100M individual words to be trained. With a window size of 2 to each side, this resulted in having 400M test cases per each epoch. Each embedding size trained for 50 epochs, **taking around 10h of total training time on google colab**. The results are located in Table 1.

2.2 Code explanation

The code is located in the *word2vec/* folder, and is divided into multiple files: *model.py* contains the skipgram model implementation; *dataloader.py* includes two available dataloaders, both lazy and eager evaluation depending on the machine that is running; *train.py* includes the main logic for training and the negative sampler implementation; *parse.py, util.py, generate_frequency.py* are all for pre-processing the dataset.

The negative sampler also performs eager preparation. For performance improvement, there is a trade-off with memory, in which a contiguous very long array is created and given a number, depending on the number of elements that exist. Generating random numbers according to a distribution is very expensive so this approach would only require generating a random number with a uniform distribution, which is much faster and can be performed very quickly for all values. For correctness purposes, every time the given negative example is the same as the target word in the example, it just picks the following word.

The formula used for the training of the model was

$$y = -log(\sigma(\boldsymbol{u}_x \cdot \boldsymbol{v}_c)) - \sum_{i=1}^{N} log(\sigma(-\boldsymbol{u}_x \cdot \boldsymbol{v}_k))$$

2.3 Result analysis

Embedding size	Spearman correlation value
100	0.4931
200	0.4676
300	0.4894

Table 1: Embedding evaluation results

The results are displayed in Table 1. The results differ a lot from the original paper and I presume it to the inability to use the full dataset. This is noticeable as the word embedding with size 100 performed the best out of all of them. Because sizes 200 and 300 require more data, they have not performed as well as expected. The evaluation was done using cosine similarity between each word vector, and spearman correlation between the cosine similarity and human result (transformed into the [-1,1] range).

Another thing to be noticed is present in Figure 1, where we can see that all the embeddings look to be performing similarly when the words have a similarity or are orthogonal, but fails miserably when the words are opposing each other. This is possibly due to the SkipGram model not actually learning opposing words, but only which words show up on the context and affects the performance of it. The lack of training data is expected to be affecting the words which don't have a high enough correlation as the human score. This is noticeable where the embeddings of size 300 have a much lower magnitude but have the same pattern present as the embeddings of size 100 and the human embeddings.

To further look on this, I decided to do some word analogy comparison and check if the model was indeed learning. For that I did

$$vec('tokyo') - vec('japan') = vec('paris') - \beta$$

to which the model replied **China**, which is incorrect. The cosine similarity was of 0.26453, which is low and its value correspond to what's observed in Figure 1. After it it also suggested other countries like Britain, Sudan, Bulgaria, so it indeed managed to capture the capital-country relationship (or the basics of it). I also decided to check other values to see what's their similarity:

Word	Cosine Similarity
washington	0.02062
dog	-0.02913
France	0.20117
Portugal	0.20851

Table 2: Word analogy of some words for the above example

I conclude then that the module is indeed learning, just not enough, and thus I can say it's due to the lack of data. Some things that I would try if I had more time / hardware are:

1. Use more data
2. Perhaps use the same negative embeddings all the time (they changed every epoch as they were generated on the moment)
3. Use Stochastic Gradient Descent instead of Adam Optimizer. Multiple sources used it, but I didn't manage to train once using it instead

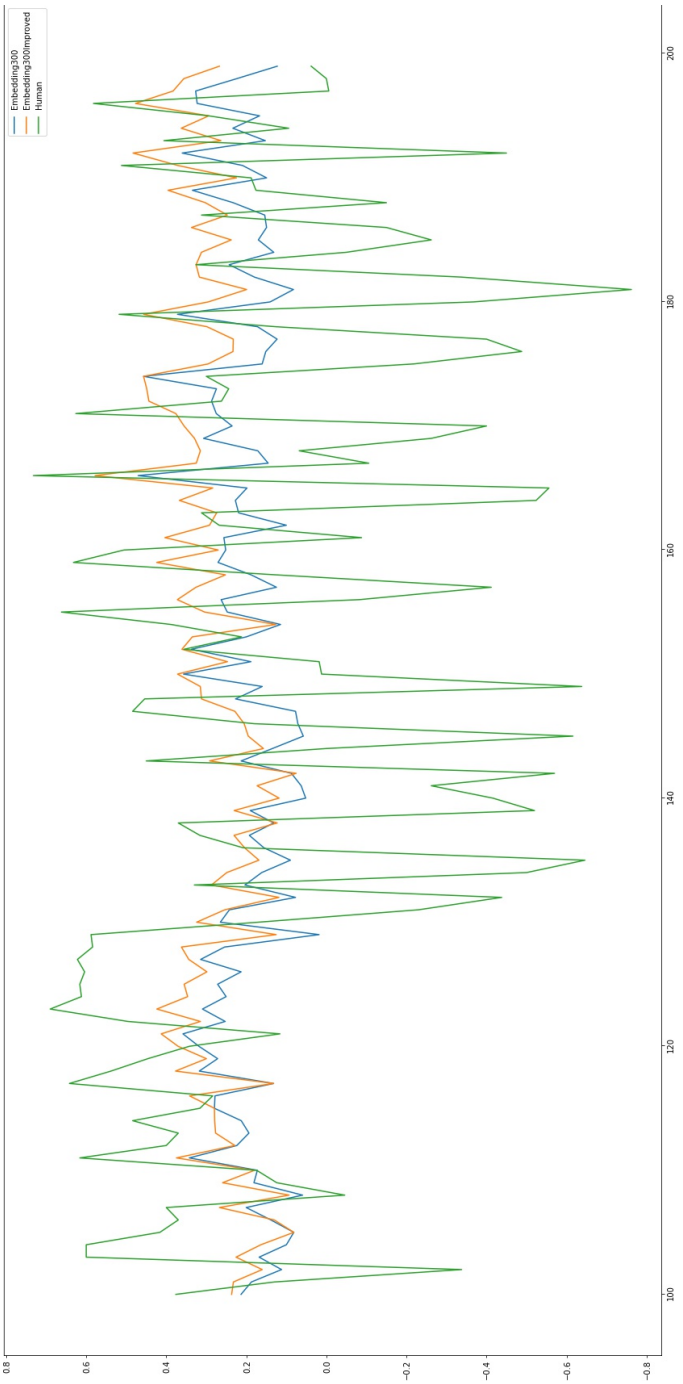

Figure 1: Comparison between different embedding sizes' similarity and human performance

3 Word2Vec improvement

As an improvement, I implemented [1] which is a post-processing improvement that can be applied on any pre-trained embedding. The code is in the file *retrofitting.py*. It consists of minimizing the function

$$\psi = \sum_{i=1}^{n} [\alpha_i ||q_i - \hat{q}_i||^2 + \sum_{(i,j)\in E} \beta_{ij} ||q_i - q_j||^2]$$

which is concave and can be easily done via an iterative updating method. It can then be equated to 0 to give the formula

$$q_i = \frac{\sum_{j:(i,j)\in E} \beta_{ij} q_j + \alpha_i \hat{q}_i}{\sum_{j:(i,j)\in E} \beta_{ij} + \alpha_i}$$

What the formula is doing is approximating the embeddings that were obtained from training and approximating them from their synonyms while also maintaining the properties obtained before. This will essentially become as a translation of the vector and **its result highly depends on the quality of the pre-trained embeddings**. Of course, in my case, as my previous results lacked more data to be successful, resulted in bad results. The translation is also noticeable in Figure 2 as we see that the similarity of the vectors has translated. This translation depends on the embedding of the synonyms which, if wrong, will result in a bad translation as it's the case here.

Similarly to Table 2.3 I have performed some word analogy to analyze the performance. The result was interesting: *nippon* was the highest one, with 0.42539. *Nippon* is the English name of Japan. This is to be expected given that the incorrect embeddings were being used and it just approximated the 'Tokyo' and 'Japan' to the Japanese knowledge rather than modelling the capital-country relationship, and it's a side effect of the given improvement.

References

[1] Faruqui, Manaal, et al. "Retrofitting word vectors to semantic lexicons." arXiv preprint arXiv:1411.4166 (2014).

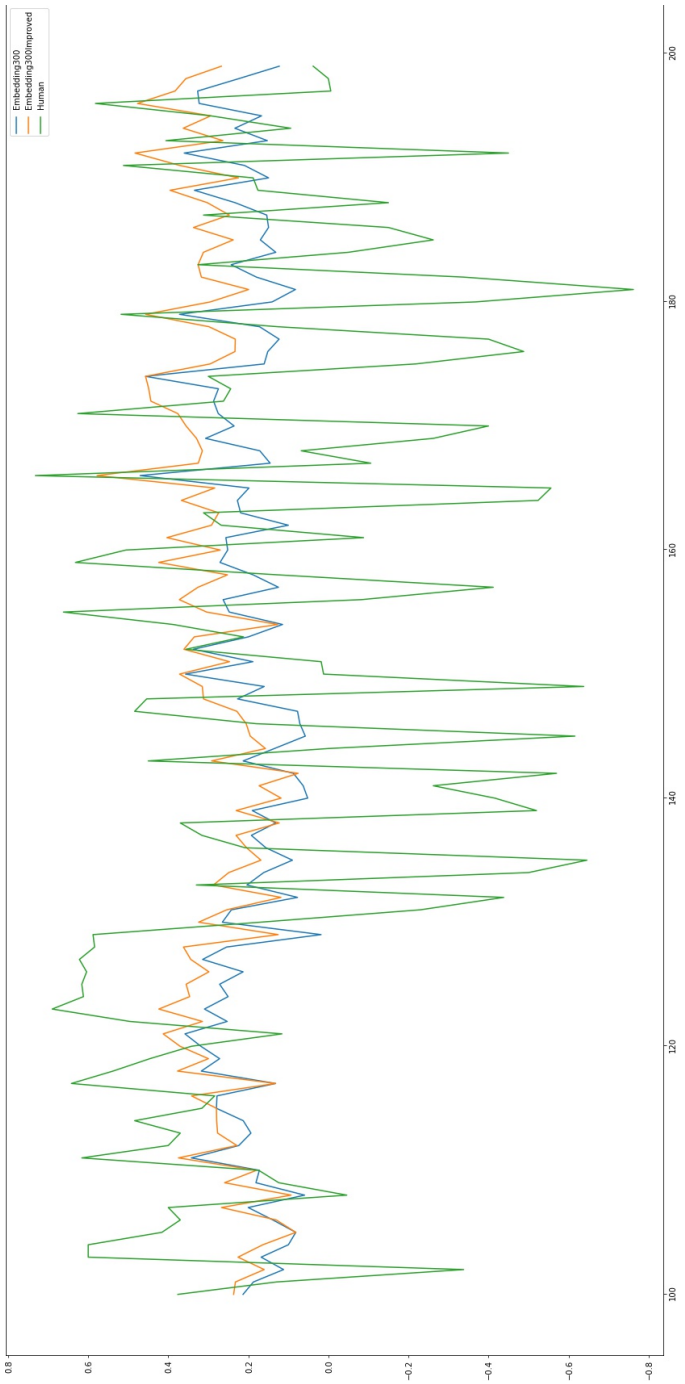

Figure 2: Improvement similarity comparison with embedding size 300 and human performance. Notice the vertical translation