GPU accelerated Implementation of Doc2VecC

B08902100 資工一江昱勳

June 23, 2020

1 Introduction

Finding machine-understandable representation that captures the semantics of text documents is a challenge for text understanding. Doc2VecC [1] provided some efficient ways to represent words and documents in continuous vector space by training a neural network model. However, the Doc2VecC algorithms do not provide an efficient way on GPU architecture. In this work, I tried to build up an efficient algorithm based on NVIDIA GPU architecture that could beat the CPU one.

Doc2VecC is mainly based on Word2Vec [2]. It represents each document as a simple average of the word embedding of selected words in the document. Therefore, it provides an efficient and simple way to represent documents without losing meaning behind the document.

2 Preliminary

2.1 Model Architecture

Doc2VecC consists of one input layer, one hidden layer, and one output layer (as Figure 1). Thus, two weighted matrices are learned by using gradient descent. Both sizes of input matrices W_i and output matrices W_o are all $|V| \times w$ where V is the set of words in the corpus, and w is a configurable parameter which controls the embedding size, typically between 100 and 1000.

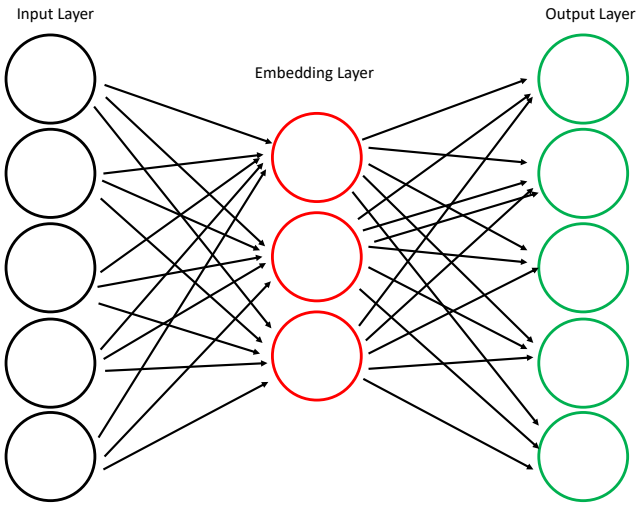

Figure 1: A Possible Neural Network

There are two well-known models used for learning the whole network, referred as Continuous Bag-of-Word (CBoW) and Skip-gram models. Both these models are based on the n-gram language model, they aim to maximize the likelihood between a word and its near-by word. CBoW maximizes

the probability of observing the target word at each position of the document by given the average of the embedding of the near-by word, while Skip-gram maximizes the similarity of the average of the embedding of the near-by word by given the target word at each position of the document.

In addition to the state-of-model CBoW model and Skip-gram model, Doc2VecC introduce a unbiased *mask-out/drop-out* corruption, which will randomly overwrites each dimension of the original document x with probability q. That is

$$\tilde{x}_d = \begin{cases} 0, & \text{with probability } q \\ \dfrac{x_d}{1-q}, & \text{otherwise} \end{cases} \tag{1}$$

Doc2VecC then defines the probability of observing a target word w^t given n its local context c^t as well as the global context $\tilde{x}$ as

$$P(w^t|c^t, \tilde{x}) = \frac{\exp(v_{w^t}^{\mathrm{T}}(U_{c^t} + \frac{1}{T}U_{\tilde{x}}))}{\sum_{w' \in V} \exp(v_{w'}^{\mathrm{T}}(U_{c^t} + \frac{1}{T}U_{\tilde{x}}))} \tag{2}$$

Here V is the corpus, T is the length of the document, U_y is the "input" vector of word y, and v_z^{T} is the "output" vector of word z. Figure 2 illustrates the model architecture.

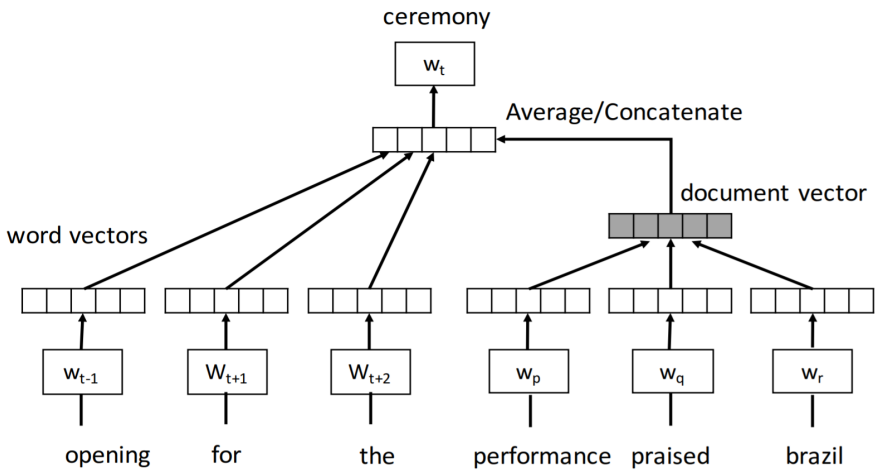

Figure 2: The Doc2VecC architecture

2.2 Hierarchical Softmax and Negative Sampling

To reduce the amount of weight matrix update in the training procedure, Word2Vec proposed two methods called *hierarchical softmax* and *Negative Sampling*.

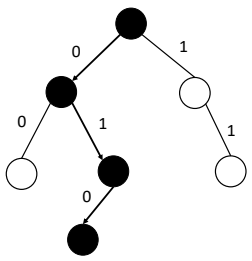

Figure 3: An example Huffman Tree

Hierarchical Softmax breaks down vocabulary into a Huffman Tree (as Figure 3) where the number of existence in the documents become the weight of Huffman Tree. As the model is trained, it will maximize the probability to walk from root to the target leaf representing the word. Due to the property of Huffman Tree, in each training step, it will only update $O(\log V)$ times on average which is significantly smaller than V.

Negative Sampling is much simpler than Hierarchical Softmax. It will sample some words as negatively labeled while the real target word is positively labeled with their number of existence in the document considered. During the training process, the positively labeled words will provide positive gradient while the negative ones provide negative gradients. This method not only maximizes the similarity between target words and their given input but also minimizes the irrelevant words.

3 Implementation

For each document being sampled, I use as much thread as the length of the document while the block size is 1024. Also, to compute the length of sampled documents, I use the parallel reduction technique to sum it up. Also, I've used the **cuRAND** library for getting random numbers inside GPU kernel code.

The forward propagation stage and the backward propagation stage of Skip-gram, CBoW, and Doc2VecC are almost the same. I tried to run a kernel with block size equal to the embedding size and each thread could simply do a single multiplication or division and then add these values to its destination like input matrix.

For negative sampling and skip-gram, I would run a kernel with block size equal to the embedding size and grid size equal to the number of training steps of one target word for each word. After that, to sum over the probability, I've tried the parallel reduction technique. Also, I've precomputed the output of the probability function and store them inside the constant memory to speed up the process by avoiding calculating complicate functions.

4 Experiment

The CPUs of testing machine are all *Intel(R) Xeon(R) CPU @ 2.20GHz* with 4 CPU installed here, and the GPU is *NVIDIA Tesla T4*. The code is available on my GitHub, and the experiment can be reproduced by running `test.sh`. For the original CPU implementation of Doc2VecC on GPU, it takes `17m54.094s` to train 5 iterations with CBoW-Negative-Sampling method. However, the GPU implementation implemented by me takes `89m52.141s` to train the model. The result is quite disappointing to me.

After investigating more with some performance analysis tools like **NVIDIA Nsight Systems**, I think that the main bottleneck is the cost of calling GPU kernel. Launching GPU kernels with different block sizes and grid sizes is quite slow in comparison to calling a function inside a CPU. Therefore, maybe I should migrate kernel codes into one kernel to avoid the overhead of calling device function.

5 Conclusion

With the experiment, I've learned that naively transplanting CPU code to GPU kernel not always gain improvement on performance even though we might have a better complexity of the whole algorithm. Also, there are too many factors that might affect the performance with CUDA, we should consider every factor carefully, not only the complexity of the algorithm. Additionally, we could check some implementations by other people with similar topics like the implementation of Word2Vec on GPU [3] first.

Though implementing models like Word2Vec on GPU with high performance is not an easy topic, and many famous projects like **keras** also fails on speeding it up on GPU [4], I think it is still a

good topic to be discussed. Not only because it is a challenging problem of how to apply efficient techniques to improve performance but also it is practical to improve the performance of the ability of machines to realize text documents. In the future, I hope to continue developing this project to improve the performance of Doc2VecC and similar models.

References

[1] Chen, Minmin, "Efficient Vector Representation for Documents Through Corruption," 5th International Conference on Learning Representations, 2017.

[2] Mikolov, T., Chen, K., Corrado, G. and Dean, J., "Efficient estimation of word representations in vector space," ICLR Workshop, (Scottsdale, AZ, USA, 2013).

[3] T. M. Simonton and G. Alaghband, "Efficient and accurate Word2Vec implementations in GPU and shared-memory multicore architectures," 2017 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, 2017, pp. 1-7, doi: 10.1109/HPEC.2017.8091076.

[4] Gensim word2vec on CPU faster than Word2veckeras on GPU, `https://rare-technologies.com/gensim-word2vec-on-cpu-faster-than-word2veckeras-on-gpu-incubator-student-blog/`