Acceleration of Word2vec Using GPUs

Seulki Bae and Youngmin Yi(✉)

School of Electrical and Computer Engineering, University of Seoul,
Seoul, Republic of Korea
{zpero422,ymyi}@uos.ac.kr

Abstract. Word2vec is a widely used word embedding toolkit which generates word vectors by training input corpus. Since word vector can represent an exponential number of word cluster and enables reasoning of words with simple algebraic operations, it has become a widely used representation for the subsequent NLP tasks. In this paper, we present an efficient parallelization of word2vec using GPUs that preserves the accuracy. With two K20 GPUs, the proposed acceleration technique achieves 1.7M words/sec, which corresponds to about 20× of speedup compared to a single-threaded CPU execution.

Keywords: Machine learning · Natural language processing · Neural network · Word2vec · Word embedding · CUDA

1 Introduction

Word embedding is mapping of words onto a continuous vector space. It is a distributed representation of words in a real value vector space, where the similar words in terms of semantics or syntax are close together. For example, 'learn' and 'studied' will be located nearby 'study'. Word embedding has become very successful since word embedding can capture various latent features of a word with a low-dimensional vector and can represent an exponential number of word clusters, which is not possible with the traditional approaches such as N-gram where words are treated as atomic units. Also, as similar words are located closely, it is natural and easy to reason words or phrases with this representation. For example, vector("smallest") can be obtained by simple algebraic operation of vector("biggest") – vector("big") + vector("small"), and it is also true with semantic reasoning: vector("France") – vector("Paris") + vector ("Korea") = vector("Seoul"). It can also reason the word in a sentence since, for example, Score(I, am, a, researcher) would be higher than Score(I, am, a, eat) or Score (I, am, a, him). Hence, the word vectors obtained by word embedding are widely used as a representational basis for subsequent NLP tasks such as named entity recognition, part-of-speech tagging, syntactic parsing, semantic role labeling, and so forth.

Bengio et al. first used a neural network to project words onto continuous vector space [3] since neural networks can learn better ways to represent the data

This work was supported by ICT R&D program of MSIP/IITP. [R0101-15-0054, WiseKB: Big data based self-evolving knowledge base and reasoning platform].

A. Hirose et al. (Eds.): ICONIP 2016, Part II, LNCS 9948, pp. 269–279, 2016.
DOI: 10.1007/978-3-319-46672-9_31

automatically. Mikolov et al. proposed efficient neural network model architectures called Continuous Bag-of-Words (CBoW) and Skip-gram [1, 2] for faster training, which do not have a hidden layer but have only two layers. Many efforts have continued to improve word embedding [7–9].

The quality of word vectors depends on many factors such as the amount of the training data, size of the vectors, and the training algorithm. In particular, it is much affected by the amount of training data. Thus, faster training implies larger amount of data in a given time, which in turn could result in higher accuracy. In fact, *word2vec*, a widely used word embedding toolkit provided by Mikolov et al. exploits multiple CPUs to process the training algorithm in parallel. Still, the training speed is 660 kWords/sec.

In this paper, we present an efficient parallelization of word2vec using a GPU, achieving an order of magnitude speedup compared to the sequential C implementation. Moreover, we present multi-GPU acceleration of word2vec, which scales well with an efficient model synchronization. This multi-GPU implementation also achieves an order of magnitude speedup against the multiple-CPU (i.e., multi-threaded) implementation of the original word2vec. Since CBoW and Skip-gram, the neural network model architectures in word2vec, do not have any hidden layer, they do not have the time-consuming matrix multiplication, which other DNNs usually have and are accelerated with cuDNN included in many popular deep learning toolkits. Thus, the proposed parallelization and acceleration strategy of word2vec using GPUs involves an elaborate design from scratch with comprehensive understanding of the algorithm. In contrast to other approaches which also attempted GPU acceleration of word2vec, we carefully considered the algorithmic dependency when parallelizing the model architectures, and was able to maintain the accuracy without any loss.

The rest of the paper is organized as follows: In Sect. 2, we review the related work to accelerate word2vec either using a GPU or using a cluster with multiple nodes. In Sect. 3, we presents the model architecture and the training algorithm in word2vec, and the proposed parallelization strategy is explained in detail in Sect. 4. Section 5 presents the experimental results, followed by a conclusion in Sect. 6.

2 Related Work

cuDNN is de facto standard GPU accelerated Deep Learning library, where optimized implementation of forward and backward convolution, pooling, normalization, and neuron activation are provided. As mentioned in the previous section, the neural network models in word2vec is not actually deep, even without any hidden layer. Thus, utilizing cuDNN for accelerating word2vec using GPUs would not work well. The recent seemingly ongoing work [5] leverages cuDNN in Keras and Theano to accelerate the word2vec implemented in Gensim, but shows even slower training time than the CPU version.

Another open source project named word2vec_cbow [4] accelerates CBoW model in word2vec using GPUs. However, it employs word-level parallelization without any synchronization, which results in significant accuracy drops. The original word2vec toolkit provides two input corpora, one with new line delimiter and the other without it.

It also provides a test set with about 20,000 questions and the answers to measure the accuracy obtained from the training. The accuracy obtained by the original word2vec with a corpus that distinguishes each sentence is 60 %, whereas the one with the other corpus that does not distinguish sentences is 44 %. Although word2vec_cbow achieves 20 times speedup against the original word2vec sequential version, the accuracy by word2vec_cbow drops to 26 % for the corpus without new line delimiter. For the corpus with new line delimiter, as their kernel is launched only per sentence, it is even slower than the CPU sequential version by 1.8 times, also showing huge accuracy drop to 7.2 %.

As word2vec is so popular that it has been implemented in MLlib for Spark [6], which is a widely used distributed framework. Word2vec in MLlib scales well as the node in the Spark increases but we found that the accuracy drops significantly when the training iteration, or epoch, increases in a distributed environment. More detailed results are shown in Sect. 5.

As a short summary, the proposed GPU-accelerated word2vec in this paper, to the best of our knowledge, is the first implementation that achieves an order of magnitude speedup without any loss of accuracy.

3 Background

Word2vec has is a word embedding toolkit that has released in 2013, which takes text corpus as input and generates word vectors. Word2vec provides two model architectures: CBoW and Skip-gram. The former predicts the current word based on the context, or the neighboring words, and the latter predicts the neighboring words based on the current word [1, 2].

Figure 1 illustrates CBoW model architecture and the pseudocode below presents the training algorithm for CBoW with negative sampling. As shown in Fig. 1 the input layer consists of the neighboring words in a sentence. Instead of having hidden layers, word2vec introduces a projection layer where vector values of each word in the context are averaged, which is denoted as *neu1* in line 8 in pseudocode. Then, forward propagation is done in line 14 obtaining *f*. The current word is used as the positive sample (line 11) while randomly chosen words are used as negative samples (line 12–13). Gradient, *g*, is calculated with the supervised label and *f*, which are back propagated to the projection layer (line 15–16), updating the weight (line 17) for each

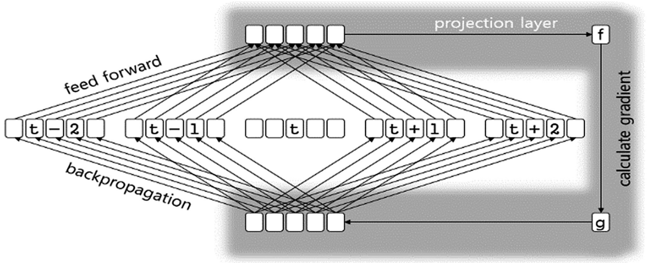

Fig. 1. The forward projection and backward projection in word2vec when w = 2.

```
Input: train_data /* the input train data */
       window /* window size */
       vDim /* dimensionality of the word vector */
       nSample /* samples that selected randomly */
Output: updated syn0 /* word vectors */
01 repeat until train_data run out
02    // read a sentence from train data
03    sen[] = ReadWords(train_data);
04    foreach tWord ∈ sen
05       // feed forward
06       foreach wWord ∈ sideWords(tWord, window)
07          for d=0 to vDim
08             neu1[d] = neu1[d] + syn0[wWord][d];
09       // negative sampling
10       foreach sample ∈ (nSample || tWord)
11          if sample == tWord then label = true
12          else // sample ∈ negative samples
13             label = false
14          for d=0 to vDim do f += neu1[d] * syn1[sample][d];
15          g = getGradient(f, label);
16          for d=0 to vDim do neu1e[d] += g * syn1[sample][d];
17          for d=0 to vDim do syn1[sample][d] += g * neu1[d];
18       // backpropagation
19       foreach wWord ∈ sideWords(tWord, window)
20          for d=0 to vDim do syn0[wWord][d] += neu1e[d];
```

Fig. 2. Pseudocode for word2vec with CBOW & negative sampling

sample. Once back propagations to the projection layer for all samples have been completed, the back propagation to the input layer is done by updating *syn0* using *neu1e* (line 19–20).

4 The Proposed Parallelization

Word2vec mainly consists of two parts: reading sentences and training. As for CBoW and Skip-gram models, the training part takes more than 90 % of the total execution time when the original C sequential implementation with default parameters was executed. Thus, it is obvious that the training part should be accelerated using GPUs. Also, the good side-effect of implementing the training part in a GPU kernel is that the time for reading sentences can be completely hidden since a kernel is asynchronously executed with a CPU and the training kernel output does not need to be transferred to the CPU during the training loop, allowing the CPU to read ahead the sentences for the next kernel launch.

Since training algorithm exhibits massive level of fine grained data-parallelism as shown in Fig. 2, it fits well to the GPU acceleration. However, a parallelization strategy without meeting the required dependency would only result in huge accuracy drop. Thus, the dependency in the algorithm is explained first, followed by the proposed mapping and synchronization in the GPU implementation, which incurs no accuracy drop. Then, further issue on the synchronization is discussed to accelerate the training algorithm using multiple GPUs.

4.1 Dependency in the Training Algorithm

Word2vec reads a sentence from the input text and trains the vectors for each word in the sentence *sequentially*. The context of a word is defined as $2w$ neighboring words, which are the left w and the right w words from the word. For the current word t, the vectors of the neighboring words from word $t - w$ to word $t + w$ are updated. Likewise, when the next word $t + 1$ is trained, the vectors of the words from word $t + 1 - w$ to word $t + 1 + w$ is updated. In other words, the window of which the size is $2w + 1$ including the current word slides from the left to the right in the sentence. Since the window of the previous word and the one of the current word overlap, the window must move sequentially (i.e., train the words sequentially) in order not to discard the training result of the previous words. The vector of a word in the sentence can be updated up to $2w$ times since a word can belong to a window, or a context, as the rightmost word, and also to another window as the leftmost word. If we parallelize the training at the word-level, as many updates are done independently, failing to capture the latent information in the sentence: it can only capture the information in the context, or the window. Such parallelization would result in a huge accuracy drop (Fig. 3).

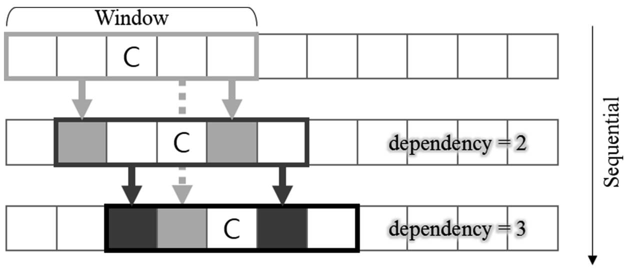

Fig. 3. The window whose size is 5 (w = 2) updates the vector of the words in the window except the current word (represented as bold rectangle), and slides to the next word. The node represents the vector of a word, and it gets darker as it is updated. The arrow indicates the dependency.

4.2 The Sentence-Wise Mapping

As discussed in the previous subsection, word-wise parallelization is not acceptable, and words in a sentence must be executed sequentially. However, the vector values in a word is independent of one another as shown in Fig. 1 (line 7, 16, 17, 20) and they are usually hundreds, typical value being 200. Thus, we map each dimension in the vector to each CUDA thread, and a sentence to a CUDA block.

A sentence that is mapped on a CUDA block runs each word in the sentence sequentially, but updating the vector of the word in parallel, as each dimension in the vector is mapped to a CUDA thread in the block. This corresponds to executing line 7, 14, 16, 17, and 20 in parallel. Note that, however, sum reduction is required among the threads for line 14. Three different well known reduction methods among threads in the block is used and compared to find the most efficient one in our context: atomic operation, parallel reduction using shared memory, and using shuffle instructions.

Mapping a sentence to a block means that as many sentences as the number of CUDA blocks in the kernel run concurrently. Since a typical input corpus usually has tens of millions of sentences, there are enough number of sentences to fully utilize the GPU in the kernel. In fact, a single kernel may not train all the sentences at once as the number of CUDA blocks in a kernel is limited to about 2 billion. In addition, the sentence buffer size would be several GBs, which could be a limiting factor, as most GPUs has less than 10 GBs of memory. Therefore, we divide the input corpus to the chunks of sentences, of which the size is determined empirically to achieve the highest throughput of the device.

The number of words in each sentence can vary drastically, which results in a different completion time of a block. However, as a CUDA block can run independently of one another, the variation in completion time does not matter as long as sufficient number of sentences are provided in the kernel. Since each sentence size is different, indexes that mark the end of each sentence are maintained.

Although the sentence-wise mapping has the same issue as in the word-wise mapping the same words in different sentences are much less dependent than the ones in different contexts in the same sentence. As the number of sentences running concurrently in a kernel gets larger, it is more likely that those sentences contain many same words, yet the accuracy still remains almost the same when the chunk size is less than a threshold, which will be confirmed in Sect. 5.

4.3 Using Multiple GPUs

If the kernel execution time is larger than the time for reading sentences, word2vec can be accelerated further by using multiple GPUs. The same model, or vectors, are duplicated to each GPU memory, and each GPU device needs to train only half of the corpus. However, the trained result, or the updated weights in the vectors, must be used in training the other words in order to maintain the accuracy. This means the models that reside in different GPU memories should be synchronized. As the total number of words in the model is in the order of hundreds of millions and the dimension size is in the order of hundreds, the model size is in the order of hundreds of MBs. Thus, the synchronization time is costly, especially if we synchronize the model at every kernel launch. In order to reduce the synchronization overhead, the model should be synchronized at less frequency yet can prevent the accuracy drop.

Note that CUDA provides a peer-to-peer data transfer API which does not go through host memory. Each GPU sends its own model to the other using this API, while at the same time receiving the model of the other to the temporary buffer.

5 Experimental Result

Table 1 shows the target platform information and word2vec parameter values used in the experiments. Gcc 4.4.6 was used for the CPU implementation and CUDA 7.0 for the GPU implementation. As an input corpus, we used Google News dataset, which contains 1.5 million sentences with 300 million words, with each sentence separated by a new line character. The accuracy in Figs. 4 and 6 was measured with about 12,000 questions in the test set, which cover 30k words appeared frequently. But Tables 3 and 4 was measured with every questions in the test set.

Table 1. Specification of the target platform and parameters

CPU	Intel Xeon E5-2630@2.30 GHz × 2	Vector dimension	200
GPU	Tesla K20 × 2	Window size	8
System memory	64 GB 1333 MHz	Negative sampling	25
		Iterations	15

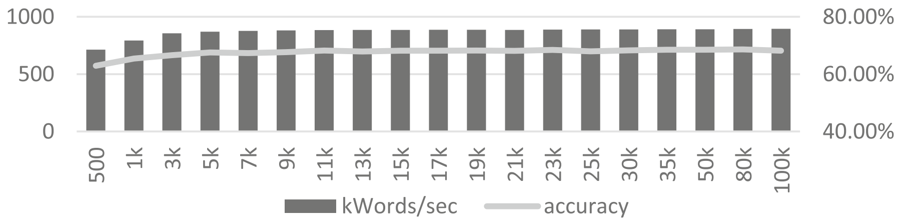

Fig. 4. Training throughput and accuracy when number of sentences per kernel varies.

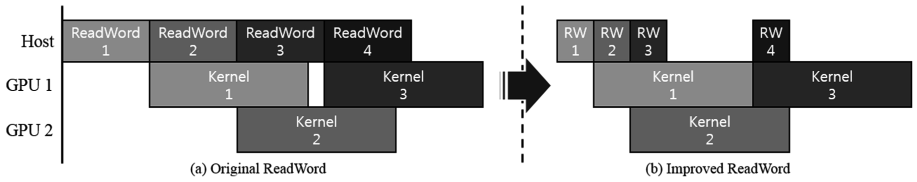

Fig. 5. (a) 2-GPU execution with the original ReadWord and (b) with the improved ReadWord

As discussed in Sect. 4, the number of sentences that are executed in a single kernel affects both the throughput and the accuracy. Figure 4 shows the overall throughput of word2vec when varying the number of sentences per kernel. The throughput saturates at 5k, reaching the training speed of about 880kWords/sec for the entire corpus. Since the accuracy saturates at 15k, we chose the number to be 15k.

We also applied multiple streams, expecting that the throughput may increase in case a GPU is not fully utilized. However, only 1.2 % of performance gain was achieved, which implies that a GPU is fully utilized. In fact, the variation in the completion time of CUDA blocks due to different sentence size mapped to each block

Table 2. Accelerated throughput using GPUs and multiple streams for the CBoW model with negative sampling

Version	kWords/sec
1-GPU CUDA (1-stream)	885.5
1-GPU CUDA (2-stream)	896.1
2-GPU CUDA (2-stream)	1502.9
2-GPU CUDA (2-stream + improved ReadWord)	1704.7

can result in load imbalance. However, since 15k sentences are provided to the kernel, there is no SM that is idle, executing the next sentence as soon as it completes the current one. It is only at the final phase of the kernel execution when some SMs become idle since there is no more input sentences to train. With two streams, this idleness can be avoided but it is already too little fraction.

Then, we accelerated the training further with two GPUs, achieving 1.68 time of speedup compared to 1-GPU execution (896kWords/sec → 1,503kWords/sec). The reason that 1.68× of speedup is achieved instead of near 2.0 times of speedup was that it was ReadWord that is the performance bottleneck. As shown in Fig. 5(a), ReadWord reads 15k sentences for a kernel, then launches the first kernel. At the same time it reads the next 15k sentence for the second kernel. Note that ReadWord can be processed in separate threads concurrently but the result is the same since ReadWord is bound to disk access time. As a result, the third ReadWord ends after the first kernel ends, making the GPU wait for the input to be read. We refactored ReadWord such that it reads a line as a whole and tokenize each word in the buffer, instead of reading a sentence one character by one. The revised ReadWord takes only 40 % of the original ReadWord implementation, removing the idle time in the GPU, as illustrated in Fig. 5(b). As a result, it achieves the Table 2 summaries the throughput of GPU each version.

Figure 6 shows the throughput and the accuracy of the training with two GPUs, varying the synchronization frequency. To distinguish the effect of multiple iteration on the accuracy from the one of the synchronization frequency, the results with iteration 1 and 5 are shown. We could confirm that the accuracy with iteration 1 clearly drops as the synchronization frequency increases. However, the accuracy increases with multiple iterations, recovering to the level similar to the one with 1 GPU. On the other

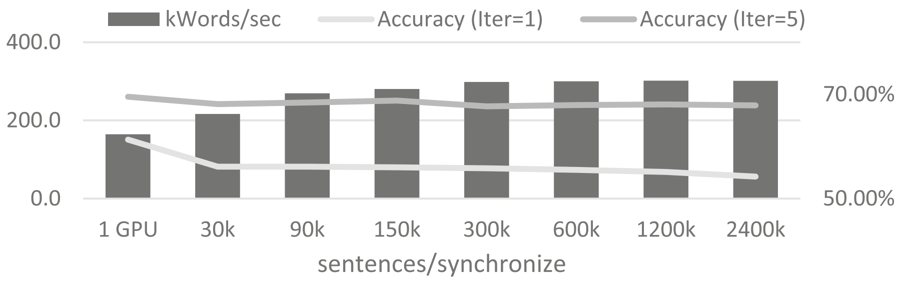

Fig. 6. Training throughput and accuracy when the frequency of the synchronization for 2 GPUs varies

hand, the throughput saturates at 300k. Thus, we perform synchronization between two GPUs at 300k sentences (i.e., at every 20 kernel launches).

Tables 3 and 4 summaries the increased throughput of the four different models by the proposed acceleration technique, as well as the Top-1 accuracy. The speedups ranging from 19.8× to 38.2× were achieved, while accuracy drops only by 0.2 %p, except for Skip-gram with hierarchical softmax algorithm where 2.1 %p of accuracy drop was observed. Note that, for some GPU implementations, the accuracy increases on the contrary.

Table 3. Acceleration results for CBoW model with negative sampling and softmax

Negative sampling	Throughtput [kWords/sec]	Acceleration over 1-thread CPU	Top1 accuracy [%]
1-thread CPU	86.2	1.0×	58.2 %
12-thread CPU	660.2	7.7×	60.5 %
1-GPU	897.7	10.4×	56.7 %
2-GPU	1704.7	19.8×	60.3 %
Hierarchical softmax	Throughtput [kWords/sec]	Acceleration over 1-thread CPU	Top1 accuracy [%]
1-thread CPU	112.4	1.0×	43.8 %
12-thread CPU	825.2	7.3×	45.4 %
1-GPU	2117.1	18.8×	41.8 %
2-GPU	3169.3	28.2×	45.3 %

Table 4. Acceleration results for Skip-gram model with negative sampling and softmax

Negative sampling	Throughtput [kWords/sec]	Acceleration over 1-thread CPU	Top1 accuracy [%]
1-thread CPU	16.4	1.0×	61.4 %
12-thread CPU	108.8	6.6×	61.8 %
1-GPU	177.5	10.8×	61.8 %
2-GPU	349.6	21.3×	59.9 %
Hierarchical softmax	Throughtput [kWords/sec]	Acceleration over 1-thread CPU	Top1 accuracy [%]
1-thread CPU	26.8	1.0×	55.0 %
12-thread CPU	174.2	6.5×	55.6 %
1-GPU	535.3	20.0×	54.1 %
2-GPU	1021.7	38.2×	54.4 %

Finally, we compared our results with the distributed word2vec version in MLlib of Spark. The train data and parameters were set identically as in Table 1, but the iteration count was set 1 since word2vec in MLlib seems to have a critical limitation of significant accuracy drop when the iteration is set larger.

Each node in the cluster has 12 CPU cores as specified in Table 1, thus 12 partitions are assigned to a node. And, the nodes are connected to one another through ConnectX-3 InfiniBand. Figure 7 shows the reduced throughput and the accuracy when multiple nodes are used in Spark. As the number of nodes increases, the throughput increases almost proportionally, but the accuracy also drops significantly (29 %p). Using four nodes (48 partitions) achieves the throughput similar to one with one K20 GPU.

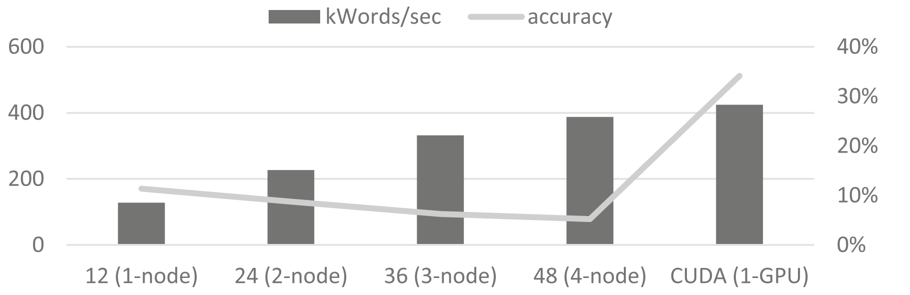

Fig. 7. Throughput and accuracy of word2vec in MLlib of Spark, and the comparison with ours (1-GPU)

6 Conclusion

In this paper, we proposed an efficient parallelization of word2vec using GPUs, which preserves almost the same accuracy. By carefully considering the algorithm dependency when mapping the algorithm concurrency to the GPU threads, we could maintain the accuracy while accelerating word2vec by an order of magnitude. Compared to a single threaded CPU implementation, the proposed technique achieved up to 38.2 times of speedup using two GPUs, while showing only 0.2 %p decrease in the accuracy except one model that shows 1.2 %p drop. This contrasts sharply with the currently available GPU implementations or a Spark implementation, where the accuracy drops drastically (29 %p – 59 %p).

We leave it as a future work to utilize more GPUs in a machine, and to utilize both GPUs and Spark on a GPU-cluster, for further acceleration of word2vec.

References

1. Mikolov, T., Sutskever, I., Chen, K., Corrado, G., Dean, J.: Distributed representations of words and phrases and their compositionality. NIPS (2013)
2. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. In: ICLR Workshop (2013)
3. Bengio, Y., Ducharme, R., Vincent, P., Jauvin, C.: A neural probabilistic language model. J. Mach. Learn. Res. **3**, 1137–1155 (2003)

4. word2vec_cbow. https://github.com/ChenglongChen/word2vec_cbow
5. word2vec-keras-in-gensim. https://github.com/niitsuma/word2vec-keras-in-gensim
6. Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., et al.: MLlib: machine learning in apache spark. arXiv preprint arXiv:1505.06807 (2015)
7. Huang, E., Socher, R., Manning, C., Ng, A.: Improving word representations via global context and multiple word prototypes. In: Association for Computational Linguistics, pp. 873–882 (2012)
8. Collobert, R., Weston, J.: A unified architecture for natural language processing: deep neural networks with multitask learning. In: International Conference on Machine Learning (2008)
9. Mnih, A., Hinton, G.: Three new graphical models for statistical language modelling. In: International Conference on Machine Learning (2007)