Scaling Word2Vec with Uniform Word Sampling

(Unrefereed Workshop Manuscript)

Bofang Li[1,2,a)] Aleksandr Drozd[1,3,b)] Satoshi Matsuoka[1,3,4,,c)]

Abstract: Word2Vec is a popular model for learning word embeddings, an important component of many natural language processing applications. Original Word2Vec algorithm's training time is proportional to the size of the corpus, which makes it prohibitive for training on terabyte-scale datasets. Another issue with the original approach is that embeddings corresponding to frequent words are updated more frequently. This results in lower quality embeddings for low-frequency words. These two issues also cause sub-optimal GPU utilization when using batch training. This work addresses these issues by proposing an alternative training scheme in which word-context pairs are sampled from the corpus uniformly, regardless of their frequencies. This allows us to use a larger batch size (equal to the vocabulary size) and better utilize GPU resources. Using the ChainerMN framework, our model can scale to up to 16 GPUs without accuracy drop. Our model also obtains lower loss for low-frequency words compared to the reference implementation.

1. Introduction

In recent years, there is a growing interest in word embedding models, where words are embedded into low-dimensional (dense) real-valued vectors. Semantically similar words tend to be close in this vector space. The trained word embeddings can be directly used for solving intrinsic tasks like word similarity and word analogy [10, 11, 14]. Word embeddings have also become the building blocks for extrinsic tasks, such as part-of-speech tagging, chunking, named entity recognition [7] and text classification [19, 34].

Multiple models of training word embeddings have been proposed in recent years [4, 5, 8, 21, 26, 27, 29, 30, 31, 33]. Probably one of the most popular model among them is Word2Vec [29]. It achieves state-of-the-art results on a range of linguistic tasks, and scales to corpora with billions of words.

The original Word2Vec implementation is based on a single node CPU. Recent works have tried to accelerate the training speed using multiple nodes with CPUs and GPUs [3, 12, 16, 28, 35]. Since Word2Vec is inherently a sequential algorithm, directly parallelizing the model is unfeasible. Scaling often results in the low-quality of learned word embeddings or sub-optimal GPU utilization. To the best of our knowledge, by using 8 GPUs, the fastest implementation [12] is only around 9 times faster than the original Word2Vec on one 8-threaded CPU.

In this paper, we address two issues in the original Word2Vec algorithm. The first is online learning scheme. Original Word2Vec scans through the corpus and optimizes the cosine distance between word-context pairs. The training time is thus proportional to the size of the corpus. This makes the algorithm hard to train on terabyte-scale datasets such as CommonCrawl [*1]. The second issue is the insensitivity to low-frequency words. The number of times a word is updated is equal to the word's frequency. This causes high loss for low-frequency words. These two issues also limit the use of the large batch size, and make the GPU utilization sub-optimal.

We propose an alternative training scheme called W2V-UWS (Word2Vec with Uniform Word Sampling). W2V-UWS samples word-context pairs from the corpus uniformly. Each word is assigned with an equal number of context samples. In this way, the training time depends only on the vocabulary size and the context size. During training, W2V-UWS scans through all sampled word-context pairs, and optimize their embedding's distance. The number of times that high-frequency and low-frequency words are updated is equal. Moreover, we are able to use a larger batch size (equal to the vocabulary size) and better utilize GPU resources.

Experiments show that on single GPU, our model is around 2 times faster than the Chainer baseline without accuracy loss. Due to the large batch size used in this model, W2V-UWS can be scaled on multiple nodes. We achieve 5.5 times speedup using 8 GPUs, and around 8 times speed up using 2 nodes with total 16 GPUs.

Since W2V-UWS only requires the pre-assignment of word-context pairs, it can be directly integrated into deep learning frameworks. We demonstrate the flexibility of W2V-UWS by implementing a subword-level word embedding model [24] on

[1] Tokyo Institute of Technology, Meguro-ku, Tokyo 152-8550, Japan
[2] Renmin University of China, Haidian, Beijing 100872, P.R. China
[3] AIST- Tokyo Tech Real World Big-Data Computation Open Innovation Laboratory, Meguro-ku, Tokyo 152-8550, Japan
[4] RIKEN Center for Computational Science, Chuo-ku, Kobe, Hyogo 650-0047 Japan
[a)] libofang@ruc.edu.cn
[b)] alex@blackbird.pw
[c)] matsu@is.titech.ac.jp

[*1] http://commoncrawl.org/

Table 1: Illustration of word-context pairs collection P for sentence "i like apple juice" (window size is 2).

word	contextual word	word	contextual word
i	like	apple	i
i	apple	apple	like
like	i	apple	juice
like	apple	juice	like
like	juice	juice	apple

Table 2: Examples of high-frequency (left) and low-frequency (right) words.

Word	Frequency	Word	Frequency
the	1061396	diluted	50
of	593677	bored	50
and	416629	salaries	50
one	411764	jp	50
in	372201	clearer	50
a	325873	ridiculous	50
to	316376	trailer	50
zero	264975	bitmap	50
nine	250430	originals	50
two	192644	sensible	50

top of it. Experiments show similar trends of scaling, and faster training speed compared to the original Chainer implementation.

2. Word2Vec

Due to the popularity and state-of-the-art performance, we choose the Skip-Gram model with negative sampling in Word2Vec [29] as our word embedding baseline. For simplicity, we directly refer it as Word2Vec in the rest of this paper.

Given a training corpus $Text$ consists of n words $w_1, w_2, w_3, ..., w_n$, Word2Vec first extracts the word-context pair collection P. Each element $(w, c) \in P$ represents a word and its corresponding contextual word. There are multiple ways of defining the context [21, 25, 37, 38]. In this work, we consider the words that precede and follow the target word within some fixed distance as context. An example of collection P is shown in Table 1.

The objective function of Word2Vec is defined as:

$$\sum_{(w,c)\in P} \log p\left(c \middle| \vec{w}\right) \tag{1}$$

where $\vec{w}$ is the contextual vector of word w. With Negative Sampling technique, the log probability in above equation can be estimated as:

$$\sum_{(w,c)\in P}\left[\log \sigma\left(\vec{w}\cdot\vec{c}\right) - \sum_{k=1}^{K}\log \sigma\left(\vec{w}\cdot\vec{c_{N_k}}\right)\right] \tag{2}$$

where σ is the sigmoid function, K is the negative sampling size, $\vec{w}$ the vector for word w. The negatively sampled word c_{N_k} is randomly selected on the basis of its unigram distribution $(\frac{\#(w)}{\sum_w \#(w)})^{ds}$, where $\#(w)$ is the number of times that word w appears in the corpus, and ds is the distribution smoothing hyper-parameter which is usually defined as 0.75.

2.1 Issues with the Online Learning Scheme

Original Word2Vec is trained in an online fashion. It scans through the corpus to obtain word-context pairs, and optimize the distance between them based on the objective function. The larger the corpus is, the longer training time is needed. This online learning scheme also fixes the order of training word-context pairs. Pairs that appear at the head of the corpus are always optimized first. We argue that it is possible to re-arrange the order of training for better quality of learned word embeddings.

2.2 Issues with Low-frequency Words

Aside from the online training issue, there is also a possible problem with how Word2Vec treat low-frequency words. The number of times a vector updates depends on the number the corresponding word appears in collection P, which in terms depend on the word's frequency.

For example, in the Text8 corpus (described in Section 6.1), word "predicting" appears only 63 times while word "the" appears 1061396 times. This indicates that the vector of word "the" will update 16847 times more often than the vector of word "predicting".

It is reasonable to treat high-frequency words more carefully since they are often also more frequent in downstream tasks. However, intuitively, millions of updates may be too much for learning a word embedding. It may also be problematic that the vectors of low-frequency words update too few times and could not be properly learned.

3. Chainer's Implementation of Word2Vec

The original W2V-C (Word2Vec Toolkit [*2]) is implemented using C on CPU. It moves over each word in the corpus and repeats the training step in an online fashion. In order to take the advantage of GPU, recent deep learning frameworks implement and optimize Word2Vec using mini-batch training.

We choose Chainer deep learning framework [36] in this study. It is a Python-based deep learning framework, and supports the use of CUDA/cuDNN for building and training neural networks. For example, Fig. 6 shows how Chainer framework [*3] implements the Word2Vec algorithm. We refer to this implementation as W2V-Chainer in the rest of this paper.

Compared to W2V-C, W2V-Chainer is more flexible. For example, one can easyly modify this architecture to subword-level word embeddings [24]. The resulting word embeddings can also be directly read as input for neural networks on downstream tasks.

Note that W2V-C can be regarded as a special case of this implementation where the batch size is set to 1.

3.1 Profiling Training Speed on GPUs

In our experiments, we use Tesla K80 GPU and Xeon(R) CPU E5-2630 v4. The memory bandwidth of CPU and GPU are 480 GB/s and 68.3 GB/s respectively. In Word2Vec training the dot product between rows of two lookup tables is the dominant operation. For each memory access, only one multiplication operation is performed. In theory, the speed of Word2Vec is bound by the memory bandwidth, and GPU should be around 7 times faster than CPU.

To saturate GPU utilization, sufficient number of

[*2] https://github.com/tmikolov/word2vec
[*3] https://github.com/chainer/chainer/tree/master/examples/word2vec

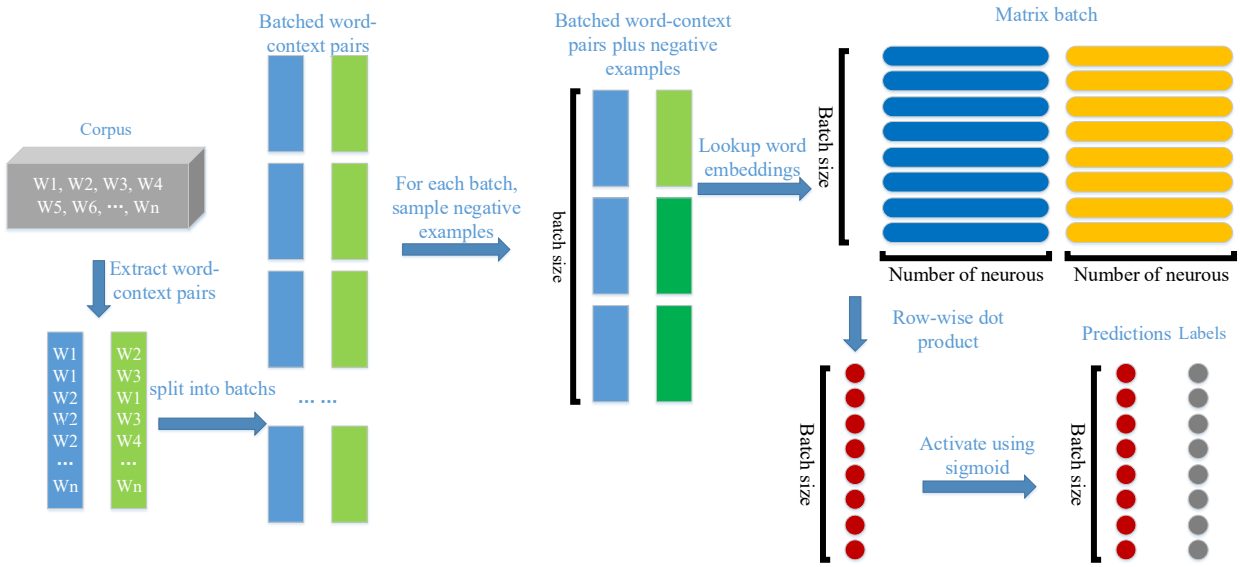


Fig. 1: Illustration of Skip-Gram with Negative Sampling in Chainer Framework

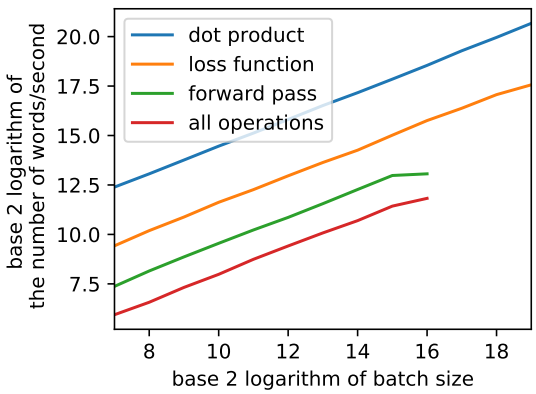


Fig. 2: Number of words/second with different batch sizes. **dot product**: calculate the row-wise dot product of two input matrices directly using CuPy. **loss function**: calculate the loss of two input matrices directly using CuPy. **forward pass**: calculate the loss of a batch of word pairs using Chainer. **all operations**: train the full model using Chainer.

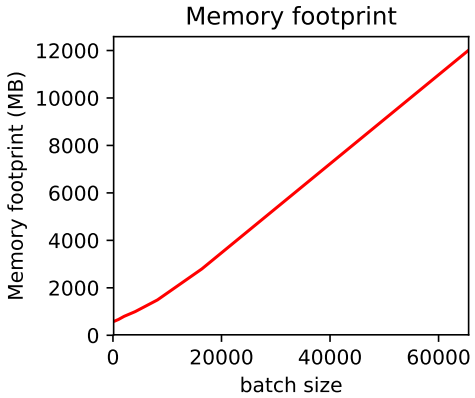


Fig. 3: Memory footprint with different batch sizes

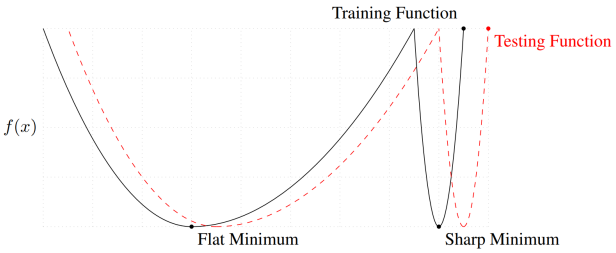


Fig. 4: Illustration of flat and sharp minimum [17].

threads/operations have to be executed per batch update. Moreover, as each batch needs to be transferred from host memory to GPU, increased computation per batch alleviates communication overhead. Since the dimension of embeddings is fixed, the only way to increase the amount of computation per batch size is to have more samples per batch. Finally, having a larger batch size improves scalability on multiple GPUs or multiple GPU nodes, as the amount of data to be sent through the network does not depend on the batch size, but only in the size of the model.

In our experiments, we have observed GPU performance to be inferior to what can be estimated for just the dot product of embedding vectors. This can be explained by multiple layers of overhead imposed by the deep learning framework: computation history need to be recorded for tape-based differentiation etc. Yet for all the operations involved we observe similar performance dynamics with respect to the batch size. The bottom line is that larger batch sizes are always beneficial performance wise, as measured empirically and illustrated in Fig. 2. The number of words/second increases almost linearly with the batch size.

3.2 Issues with the Large Batch Size

It seems promising to use large batch size for training word embeddings. However, we find that there are mainly two problems with large batch size that limits Word2Vec achieve the ideal

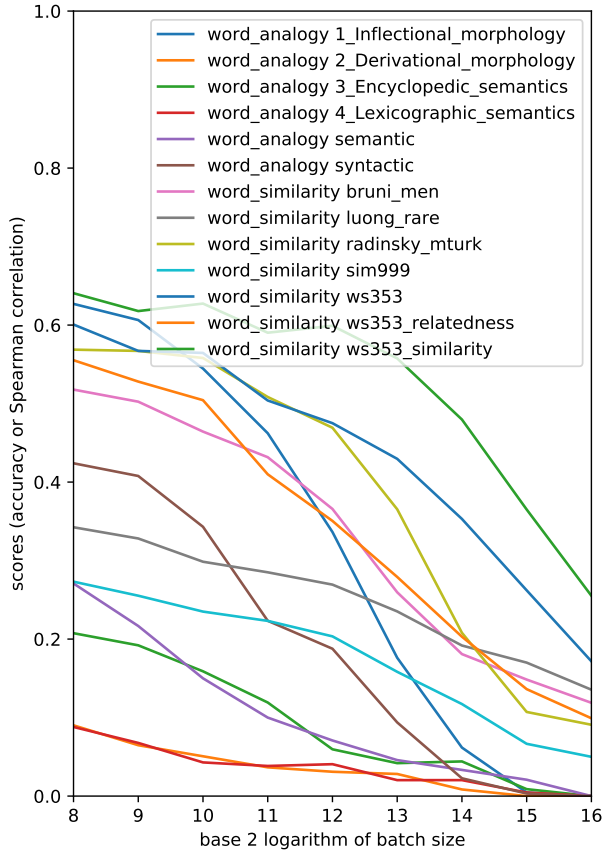


Fig. 5: Word similarity scores with different batch sizes.

speed on GPU.

The first and most obvious problem with large batch sizes is the the memory boundedness of the algorithm. It is especially problematical when we training neural networks, as per-neuron gradients have to be stored to perform back prorogation. As shown in Fig. 3, the memory footprint increases linearly with the batch size. The Tesla K80 GPU has 12GB memory, thus $2^{16} = 65536$ is the maximum number of batch size that used to train the full Word2Vec algorithm.

Another more dramatic problem with large batch size is the low-quality of learned word embeddings. Previous researches [15] show that there can be flat and sharp minimums in training functions. Models with large batch size tend to converge to the sharp minimum, and result in bad performance on a range of computer vision tasks [17]. We test the impact of batch size on word similarity and word analogy task as shown in Fig. 5. Probably due to the same reason, for all the tasks and datasets that we tested, the scores drop when we increase the batch size. In the rest of this paper, we experiment only on the batch size $2^{10} = 1024$, which is also the default setting in Chainer's implementation.

4. Word2Vec with Uniform Word Sampling

This study tries to use large batch sizes by addressing the above issues of Word2Vec. We propose an alternative training scheme called W2V-UWS (Word2Vec with Uniform Word Sampling). The overall architecture of W2V-UWS is shown in Fig. 6.

Given a training corpus $Text$ and its vocabulary V, the first step

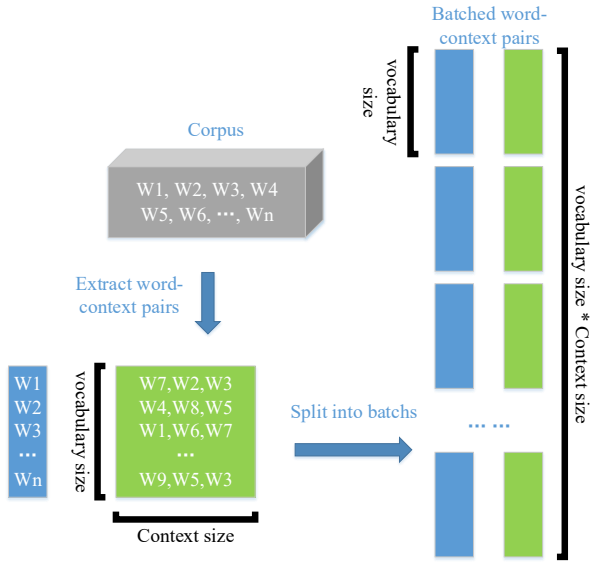


Fig. 6: Illustration of W2V-UWS

of W2V-UWS is to sample word-context pairs uniformly from $Text$. Each word in the vocabulary is assigned with l context samples. This pre-processing step creates a word-contexts matrix M with size $(|V|, l)$. Element $m_{i,j} \in M$ represents the j_{th} contextual words of the i_{th} word in the vocabulary.

More precisely, we scan through the corpus before training, and extract the contextual words for all the words in the vocabulary. For high-frequency words that have contextual words more than l, we randomly discard the abundant contextual words. For low-frequency words that have fewer contextual words than l, we randomly duplicate the existing contextual words.

This pre-processing step is done on CPU using single thread. Empirically, it takes around 400 seconds to process 100M corpus. Note that word-contexts matrix M can be stored and re-used. Due to this season, we do not count the time of this pre-processing step in the rest of our experiments.

The objective function of W2V-UWS is similar to the original Word2Vec. However, instead of scanning through the corpus, W2V-UWS scans through the word-contexts matrix M:

$$\sum_{j=0}^{l} \sum_{i=0}^{|V|} \left[\log \sigma\left(\vec{w_i} \cdot \vec{m_{i,j}}\right) - \sum_{k=1}^{K} \log \sigma\left(\vec{w_i} \cdot \vec{c_{N_k}}\right) \right] \quad (3)$$

where w_i is the i_{th} word in the vocabulary.

For each iteration, the batch size is set to $|V|$. As shown in Fig. 6, words appear once and only once in the left word list. However, in the right word list, high-frequency words still appear much more times than low-frequency words. In order to ensure the equivalence of updating, the left word list looks up the word vectors and context vectors iteratively. We only update the left side of the matrix, and directly ignore the loss on the right side.

Note that $|V|$ is usually ranging from 50000 to 500000, which is much larger than the 1024 batch size used in W2V-Chainer.

4.1 Scaling W2V-UWS using Multiple GPUs and Nodes

We also explore the possibility of using multiple nodes and multiple GPUs to scale our W2V-UWS model. Due to the large

batch size used in W2V-UWS, we choose model parallelism in this study. The larger batch size we have, the more data we can send to each GPU. In this way, the number of iterations is reduced, and thus speeds up the training.

More specifically, we first split each batch into multiple small batches. Suppose we have N workers, we randomly assign each worker with B/N word-context pairs, where B is the batch size. For each iteration, workers communicate to obtain the averaged gradient over gradients of all workers. Then, the aggregated gradient is used to improve the model in the optimization step. For fair comparison, we also implement the parallelized W2V-Chainer using the same idea.

5. Related Work

This paper takes the Chainer framework as an example of scaling Word2Vec. There are also several other deep learning frameworks have implemented Word2Vec on GPU, such as Gensim with Keras [*4] and Theano [*5]. However, those frameworks share the same issues with Chainer, and performs worse than original Word2Vec in terms of speed [12]. TensorFlow [*6] implements Word2Vec on CPU and calls C++ functions, which achieved comparable speed as the original Word2Vec.

Another promising direction for accelerating Word2Vec is directly parallelizing it regardless of the conflicts. In fact, the original Word2Vec implementation utilizes multiple threads to accelerate the training speed. It seems promising to train the same model with multiple nodes. However, as shown in Spark MLLib [28] and Deeplearning4j [35], the accuracy drops significantly when more nodes are employed. To the best of our knowledge, the most promising work on CPU is to convert the dot product to matrix multiplication by carefully aligning the order of word-context pairs. It speeds up the original Word2Vec 2-3 times on Intel BDW and Intel KNL [16] without accuracy drop.

There are also works that tries to accelerate Word2Vec on GPUs using CUDA [3, 12]. To alleviate the conflict issue, these two methods assign each CUDA block with a whole sentence instead of word pairs. This ensures the good quality of learned word embeddings. Using 2 GPUs, 21.3 times speedup over 1-threaded CPU are achieved in [3]. Similarly, using 8 GPUs, around 9x speedup over an 8-threaded CPU are achieved in [12].

6. Experiments

6.1 Implementation Details

We implement W2V-UWS using Chainer deep learning framework [36]. For scaling to multiple GPUs and nodes, we use the ChainerMN framework [2], which is an additional package for Chainer. We also choose a relatively small TEXT8 [*7] corpus, which contains the first 10^9 bytes of the English Wikipedia dump from Mar. 3, 2006. The word embedding size N is set to 500. The negative sampling size is set to 5, and the window size is set to 2. Following Chainer's original word2vec implementation, we use Adam [20] as the optimization function, which performs much better than SGD [18] in our pilot experiments. Words which appear fewer than 50 times are directly discarded, which results in vocabulary size of 18498. All models are trained for 1 epochs on the Nvidia Tesla K80 GPU(s).

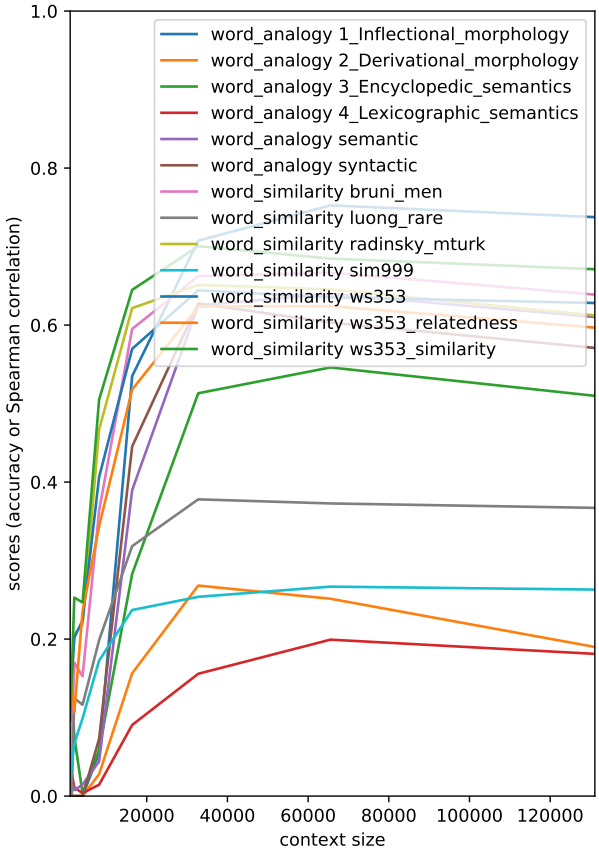


Fig. 7: Word similarity and word analogy scores with different context size.

6.2 Datasets

In order to measure the quality of learned word embeddings, we choose word similarity and word analogy tasks as benchmarks.

Word similarity task aims at producing semantic similarity scores of word pairs, which are compared with the human scores using Spearman's correlation. The cosine distance is used for generating similarity scores between two word vectors. In our experiments, we use WordSim353 (WS) [10] dataset, divided into similarity (sem.) and relatedness (rel.) categories [1,39], Sim 999 dataset [14], MEN dataset [6], and Mech Turk dataset [32], Rare Words dataset [26].

The word analogy task aims at answering questions generalize as "a is to a' as b is to __ ?", such as "London is to Britain as Tokyo is to Japan". We use the LRCos method [9] for solving word analogies, which significantly improves on the traditional vector offset method. We use Google analogy dataset [29] along with a much bigger and balanced BATS analogy dataset [11].

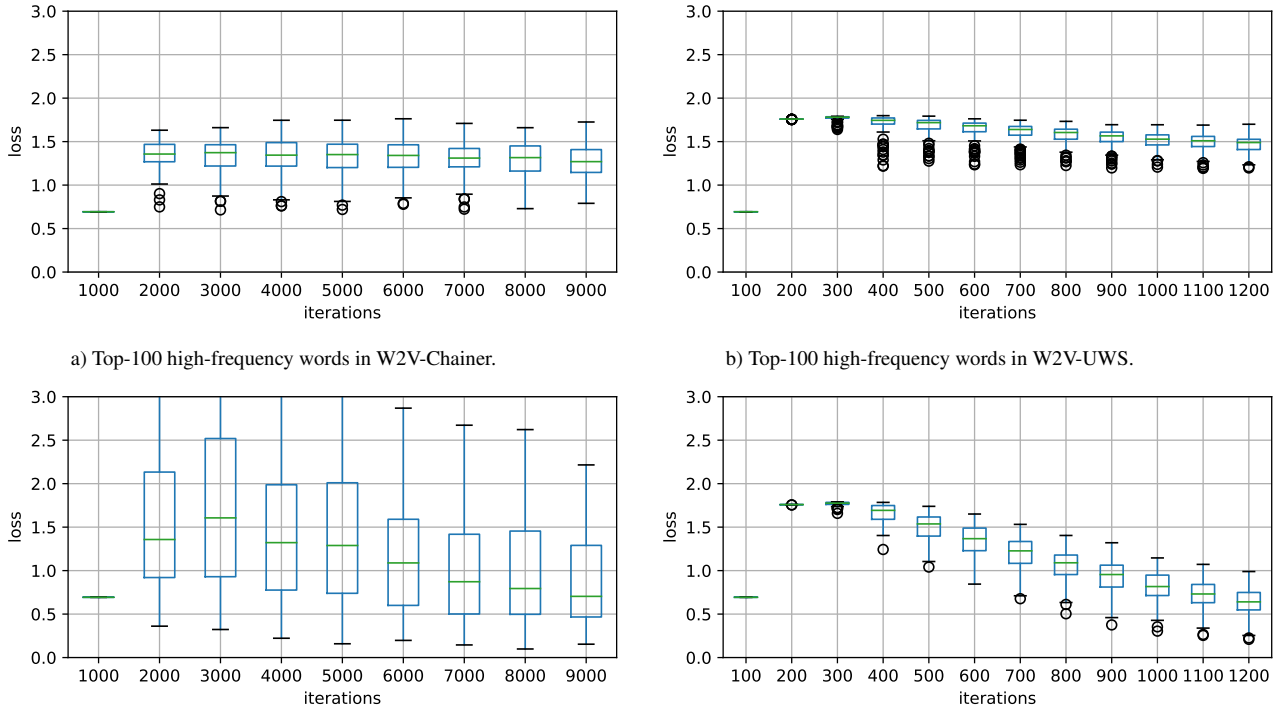
6.3 Impact of Context Size *l*

The most important hyper-parameter in our W2V-UWS model

[*4] https://github.com/niitsuma/word2vec-keras-in-gensim
[*5] https://github.com/Ignotus/theano-word2vec
[*6] https://www.tensorflow.org/tutorials/word2vec
[*7] http://mattmahoney.net/dc/textdata.html

Table 3: Different model's results on word similarity and word analogy datasets. The best results in each datasets are marked **BOLD**.

Model	Training time (seconds)	Word Similarity						Word Analogy					
		Rare Words	WS sem.	WS rel.	Sim 999	MEN	Mech Turk	BATS				Google	
								inf.	der.	enc.	lex.	sem.	syn.
Word2Vec (1 threaded CPU)	138	**.379**	.580	.475	**.241**	.474	.462	.470	.078	.211	.054	.147	.381
W2V-Chainer (1 GPU)	3061	.351	.606	.496	.232	.531	.562	**.583**	.123	.201	.065	.287	.397
W2V-UWS (16 GPUs)	193	.314	**.655**	**.526**	.232	**.594**	**.623**	.490	**.151**	**.332**	**.098**	**.390**	**.405**



a) Top-100 high-frequency words in W2V-Chainer.

b) Top-100 high-frequency words in W2V-UWS.

c) Top-100 low-frequency words in W2V-Chainer.

d) Top-100 low-frequency words in W2V-UWS.

Fig. 8: The loss of high-frquency and low-frequency words on W2V-Chainer and W2V-UWS.

is context size l. Since the vocabulary size is fixed for a given corpus, context size l directly defines how big the word-contexts matrix M is. Consequently, it affects the quality of learned embeddings, as well as the training speed.

As shown in Fig. 7, we test the effect of different context size on word similarity and word analogy tasks. It is clear that larger context size yields higher scores. However, the scores stop growing when the context size reaches around $2^{16} = 65536$. We thus set context size to this number in the following experiments.

For context size 65536, the word-contexts matrix M has $|V| \cdot l = 18498 \cdot 65536 = 12$ billion elements. This indicates that we have to do 12 billion dot product operations in total in one epoch of training. In the case of W2V-Chainer, the number of dot product operations directly depends on the corpus size. There are 0.17 billion words in total in the vocabulary. For each word, we have to calculate 4 contextual words and 5 negative samples. The total number of operations is $0.17 \cdot 4 = 0.68$ billion in W2V-Chainer, which is around 17.6 times less than W2V-UWS. This suggests that compared to W2V-Chainer, our W2V-USW has a lot of redundant computation, which may need further investigation in the future. Nonetheless, due to the large batch size used in W2V-USW, it is still faster than W2V-Chainer as shown in Section 6.5.

6.4 The Quality of Learned Word Embeddings

In this section, we evaluate the quality of learned word embeddings. As shown in Table 3, all three models perform comparable on these tasks and datasets. This may first look counter-intuitive since those models have different training strategies. For example, the Chainer implementation of Word2Vec uses mini-batch training with Adam optimizer, while the C implementation uses Stochastic Gradient Descent. Our W2V-UWS even differs from other two models in the objective function. However, all those models are actually based on the Distributional Hypothesis [13]. No matter how those models are trained, the goal is still to make "words that occur in similar contexts to have similar embeddings". As also shown in previous works [22, 23], when the hyper-parameters are set to the same values, it is hard to find a consistent advantage to one model over another.

6.5 Analyzing the Loss of Individual Words

In this section, we analyze the loss of high-frequency words and low-frequency words. In every iteration, we save the losses for each individual words and draw the loss curve in Fig. 8. The total training time is similar in all the sub-figures. On a single GPU, W2V-UWS takes 1594 seconds to train for 1024 iterations. W2V-Chainer takes 1573 seconds to train for 8000 iterations.

For both models, the losses of high-frequency words show

the same trends (Fig. 8-a and Fig. 8-b). They all gradually decrease as the number of iterations increases. However, for the low-frequency words, the losses actually increase or stay high in W2V-Chainer (Fig. 8-c). The updating times of low-frequency words in W2V-Chainer is low. As we discussed in Section 2.2, it is hard to learn meaningful word embeddings with few updates.

Note that for W2V-UWS, the losses of high-frequency words are much higher than the losses of low-frequency words. The high-frequency words are related to much more contextual words than low-frequency words, and is harder to optimize.

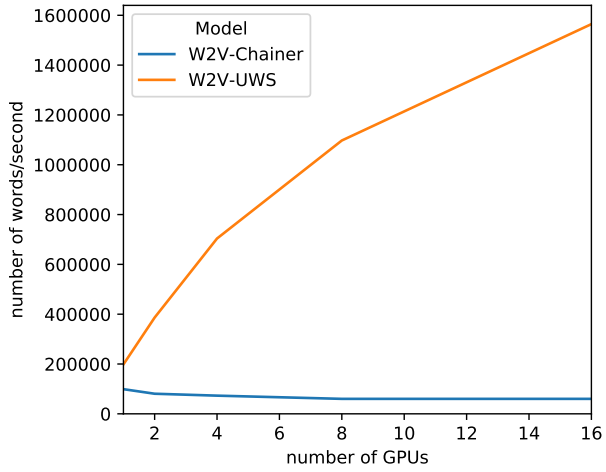


Fig. 9: Training speed of word-level word embeddings with different number of GPUs.

6.6 Scaling with Multiple Nodes

Fig. 9 shows the scaling results with multiple nodes. Our W2V-UWS scales quite linearly. Using 8 GPUs on a single nodes, our model achieves around 11 million words per second, which is 5.5 times faster than single GPU implementation. It also achieves 7.5 time speed up using 2 nodes with total 16 GPUs. Note that W2V-Chainer actually gets lower speed when using multiple GPUs. This is mainly due to the small batch size.

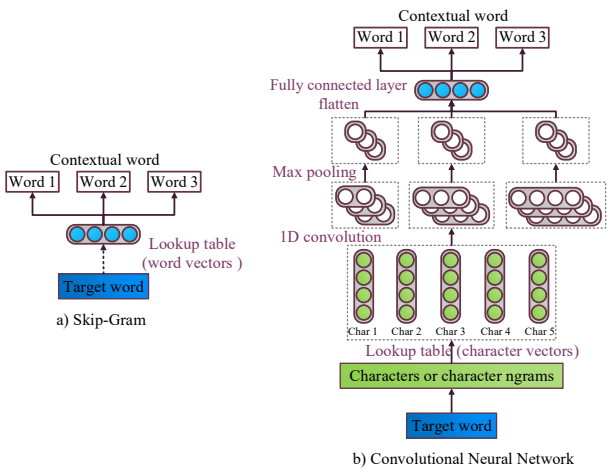


Fig. 10: Illustration of word-level and subword-level word embedding models.

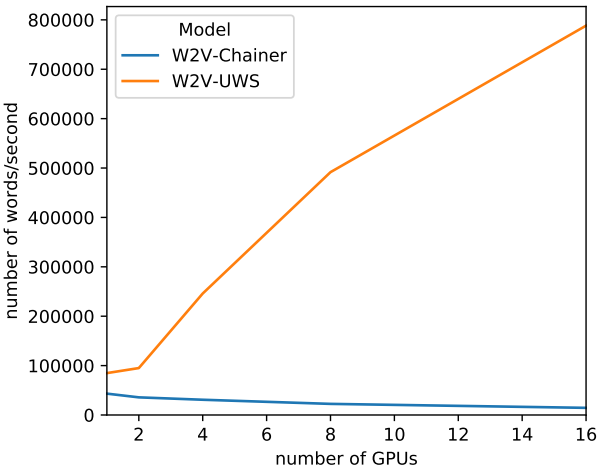


Fig. 11: Training speed of subword-level word embeddings with different number of GPUs.

6.7 Training Subword-level Word Embeddings

In this section, we explore the flexibility of our model by implementing the subword-level word embeddings model [24] on top of W2V-UWS.

As shown in Fig. 10, the only difference between training word-level word embeddings and subword-level word embeddings is the way of obtaining word vector. For word-level word embeddings, each word is directly assigned with a vector. For subword-level word embeddings, the vector of a word is composed by its character vectors. More precisely, subword-level word embedding model first extracts the characters in the word, and assign vectors for the characters. All those character vectors are then fed into a Convolutional Neural Network (CNN). Finally, the convolution results are flattened, and fed into a fully connected layer to form the word vector.

Implementing a CNN from scratch is not easy, especially for both inference and training. However, since our W2V-UWS is implemented using Chainer deep learning framework, we can directly call chainer's CNN functions with few lines of code. This simplicity and flexibility are the main reason that we choose to use this deep learning framework instead of CUDA.

Fig. 11 shows the scaling results of subword-level word embedding models. The trends of subword-level word embeddings are similar to that on word-level word embeddings. On single GPU, our W2V-UWS model is around 2 times faster than W2V-Chainer. Compared to single GPU performances, we are able to speed up the training around 9 times with 16 GPUs.

Note that compared to the training speed of word-level models in Fig. 9, the speed of subword-level W2V-UWS is around 2 times slower. This is reasonable since CNNs in the subword-level model need a lot of computation.

7. Conclusion

We propose W2V-UWS, an algorithm to train word embeddings with uniform word sampling. Our algorithm is invariant to the corpus size, and treats high-frequency and low-frequency words equally. Compared to the reference Chainer implemen-

tation, W2V-UWS is able to train using large batch size without accuracy loss, and is around 2 times faster. Moreover, W2V-UWS is able to scale on multiple GPUs and nodes. We show 5.5 times speed up using 8 GPUs, and 7.5 times speed up using 2 nodes with total 16 GPUs.

W2V-UWS is flexible. We implement a subword-level word embedding model on top of it with little effort. Our subword-level model is around 2 times faster than the original implementation, and scales well on multiple GPUs and nodes.

Acknowledgments This work was partially supported by JSPS KAKENHI Grant Number JP17K12739 and JST CREST Grant Numbers JPMJCR1303 and JPMJCR1687.

References

[1] Agirre, E., Alfonseca, E., Hall, K., Kravalova, J., Paşca, M. and Soroa, A.: A study on similarity and relatedness using distributional and wordnet-based approaches, *NAACL*, Association for Computational Linguistics, pp. 19–27 (2009).

[2] Akiba, T., Fukuda, K. and Suzuki, S.: ChainerMN: Scalable Distributed Deep Learning Framework, *Proceedings of Workshop on ML Systems in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, (online), available from ⟨http://learningsys.org/nips17/assets/papers/paper$_2$5.*pdf*⟩ (2017).

[3] Bae, S. and Yi, Y.: Acceleration of Word2vec Using GPUs, *International Conference on Neural Information Processing*, Springer, pp. 269–279 (2016).

[4] Bengio, Y., Ducharme, R., Vincent, P. and Janvin, C.: A neural probabilistic language model, *The Journal of Machine Learning Research*, Vol. 3, pp. 1137–1155 (2003).

[5] Bojanowski, P., Grave, E., Joulin, A. and Mikolov, T.: Enriching Word Vectors with Subword Information, *Transactions of the Association for Computational Linguistics*, Vol. 5, pp. 135–146 (2017).

[6] Bruni, E., Boleda, G., Baroni, M. and Tran, N.-K.: Distributional semantics in technicolor, *ACL*, Association for Computational Linguistics, pp. 136–145 (2012).

[7] Collobert, R. and Weston, J.: A unified architecture for natural language processing: Deep neural networks with multitask learning, *ICML*, ACM, pp. 160–167 (2008).

[8] Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K. and Kuksa, P.: Natural language processing (almost) from scratch, *The Journal of Machine Learning Research*, Vol. 12, pp. 2493–2537 (2011).

[9] Drozd, A., Gladkova, A. and Matsuoka, S.: Word Embeddings, Analogies, and Machine Learning: Beyond king - man + woman = queen, *COLING* (2016).

[10] Finkelstein, L., Gabrilovich, E., Matias, Y., Rivlin, E., Solan, Z., Wolfman, G. and Ruppin, E.: Placing search in context: The concept revisited, *WWW*, ACM, pp. 406–414 (2001).

[11] Gladkova, A., Drozd, A. and Matsuoka, S.: Analogy-based Detection of Morphological and Semantic Relations With Word Embeddings: What Works and What Doesn't., *NAACL-HLT*, pp. 8–15 (2016).

[12] Gupta, S. and Khare, V.: BlazingText: Scaling and Accelerating Word2Vec using Multiple GPUs, *Proceedings of the Machine Learning on HPC Environments*, ACM, p. 6 (2017).

[13] Harris, Z.: Distributional structure, *Word*, Vol. 10(23), pp. 146–162 (1954).

[14] Hill, F., Reichart, R. and Korhonen, A.: Simlex-999: Evaluating semantic models with (genuine) similarity estimation, *Computational Linguistics* (2016).

[15] Hochreiter, S. and Schmidhuber, J.: Flat minima, *Neural Computation*, Vol. 9, No. 1, pp. 1–42 (1997).

[16] Ji, S., Satish, N., Li, S. and Dubey, P.: Parallelizing word2vec in multi-core and many-core architectures, *arXiv preprint arXiv:1611.06172* (2016).

[17] Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M. and Tang, P. T. P.: On large-batch training for deep learning: Generalization gap and sharp minima, *arXiv preprint arXiv:1609.04836* (2016).

[18] Kiefer, J. and Wolfowitz, J.: Stochastic estimation of the maximum of a regression function, *The Annals of Mathematical Statistics*, pp. 462–466 (1952).

[19] Kim, Y.: Convolutional Neural Networks for Sentence Classification, *EMNLP*, pp. 1746–1751 (2014).

[20] Kingma, D. P. and Ba, J.: Adam: A method for stochastic optimization, *arXiv preprint arXiv:1412.6980* (2014).

[21] Levy, O. and Goldberg, Y.: Dependency-Based Word Embeddings., *ACL*, pp. 302–308 (2014).

[22] Levy, O. and Goldberg, Y.: Neural Word Embedding as Implicit Matrix Factorization, *NIPS*, pp. 2177–2185 (2014).

[23] Levy, O., Goldberg, Y. and Dagan, I.: Improving Distributional Similarity with Lessons Learned from Word Embeddings, *TACL*, Vol. 3, pp. 211–225 (2015).

[24] Li, B., Drozd, A., Liu, T. and Du, X.: Subword-level Composition Functions for Learning Word Embeddings, *Proceedings of the Second Workshop on Subword/Character LEvel Models*, pp. 38–48 (2018).

[25] Li, B., Liu, T., Zhao, Z., Tang, B., Drozd, A., Rogers, A. and Du, X.: Investigating Different Syntactic Context Types and Context Representations for Learning Word Embeddings, *EMNLP*, pp. 2411–2421 (2017).

[26] Luong, T., Socher, R. and Manning, C. D.: Better Word Representations with Recursive Neural Networks for Morphology., *CoNLL*, pp. 104–113 (2013).

[27] Melamud, O., Goldberger, J. and Dagan, I.: context2vec: Learning generic context embedding with bidirectional lstm, *Proceedings of The 20th SIGNLL Conference on Computational Natural Language Learning*, pp. 51–61 (2016).

[28] 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, *The Journal of Machine Learning Research*, Vol. 17, No. 1, pp. 1235–1241 (2016).

[29] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S. and Dean, J.: Distributed Representations of Words and Phrases and their Compositionality, *NIPS*, pp. 3111–3119 (2013).

[30] Mikolov, T., Yih, W.-t. and Zweig, G.: Linguistic Regularities in Continuous Space Word Representations., *HLT-NAACL*, Vol. 13, pp. 746–751 (2013).

[31] Pennington, J., Socher, R. and Manning, C. D.: Glove: Global Vectors for Word Representation, *EMNLP*, pp. 1532–1543 (2014).

[32] Radinsky, K., Agichtein, E., Gabrilovich, E. and Markovitch, S.: A word at a time: computing word relatedness using temporal semantic analysis, *WWW*, ACM, pp. 337–346 (2011).

[33] Salle, A., Idiart, M. and Villavicencio, A.: Matrix Factorization using Window Sampling and Negative Sampling for Improved Word Representations, *The 54th Annual Meeting of the Association for Computational Linguistics*, p. 419 (2016).

[34] Socher, R., Perelygin, A., Wu, J. Y., Chuang, J., Manning, C. D., Ng, A. Y. and Potts, C.: Recursive deep models for semantic compositionality over a sentiment treebank, *EMNLP*, Vol. 1631, Citeseer, p. 1642 (2013).

[35] Team, D.: Deeplearning4j: Open-source distributed deep learning for the jvm, *Apache Software Foundation License*, Vol. 2 (2016).

[36] Tokui, S., Oono, K., Hido, S. and Clayton, J.: Chainer: a next-generation open source framework for deep learning, *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, Vol. 5 (2015).

[37] Vulic, I. and Korhonen, A.: Is "Universal Syntax" Universally Useful for Learning Distributed Word Representations?, *ACL*, p. 518 (2016).

[38] Yatbaz, M. A., Sert, E. and Yuret, D.: Learning Syntactic Categories Using Paradigmatic Representations of Word Context, *EMNLP-CoNLL*, pp. 940–951 (2012).

[39] Zesch, T., Müller, C. and Gurevych, I.: Using Wiktionary for Computing Semantic Relatedness., *AAAI*, Vol. 8, pp. 861–866 (2008).