Distributed Asynchronous Word2Vec Training

Johannes Schneider
Hasso Plattner Institute,
University of Potsdam
johannes.schneider@student.hpi.de

Julian Weise
Hasso Plattner Institute,
University of Potsdam
julian.weise@student.hpi.de

Thorsten Papenbrock
Hasso Plattner Institute,
University of Potsdam
thorsten.papenbrock@hpi.de

ABSTRACT

The Word2Vec skip-gram approach has gained a lot of popularity since its initial release in 2013. Although modern tweaks and improvements have led to a outstanding model performance, established implementation still have difficulties processing huge text corpora. Various researchers proposed different approaches for distributing training of word embeddings, often limited by the need of model synchronization. We present a leaderless implementation based on the Akka toolkit, theoretically capable of coping with corpora of any size.

Employing the actor programming concept allows us to encapsulate model partitions, thus promoting synchronization-less model training through message-only communication.

KEYWORDS

distributed, model training, asynchronous, actor, akka, word2vec, unsupervised machine learning

1 INTRODUCTION

As human-computer interaction shifts more towards direct interaction through natural language, understanding and interpretation of unstructured data becomes increasingly more important in today's world. First approaches to tackle this problem have been made as early as in the 1950s by automatically translating a Russian text into English [5].

Although involved researchers expected those days computer systems to be capable of understanding natural language within the next three years, it took fifty more years to produce promising results: In 2013 Mikolov et al. proposed a novel technique to capture semantic relations between words in a text [9]. Representing terms as vectors within an n-dimensional space enables the identification of word similarities through spatial proximity. Over the years the initial implementation of Mikolov et al. has been adapted by many researchers and software engineers to improve usability and performance.

The – probably – most often used adaption of that initial implementation is called gensim [11]. By combining the simplicity of Python's programming language and underlying C translations, gensim achieves great usability while still outperforming many other implementations. However, gensim is tied to a single machine only. This becomes first and foremost a problem in the context of training based on extensive vocabularies: The entire vocabulary has to be present in main-memory to complete training within an acceptable amount of time. Thus, the available amount of memory of one machine limits the size of the text corpus to train embeddings on. The demand for more precise word embeddings has forced researchers to use larger text corpora for training, resulting in vocabularies outgrowing the main memory of a commodity machines.

In this work, we propose a distributed implementation of Mikolov's Word2Vec Skip-Gram algorithm [9] that scales well with vocabularies of any size. By distributing the corpus and the vocabulary over many commodity nodes, we solve the issue of limited main-memory, while still achieving acceptable training performance.

The remaining work is structured as follows: Section 2 gives a short overview of similar approaches aiming to solve the problem of distributing word embedding training. Section 3 describes our concept in an overview fashion, identifying three conceptual steps necessary for performing training in a distributed way. Sections 4, 5, and 6 explain each one of the beforementioned steps in detail. Section 7 examines some benchmarks, whereas section 8 concludes our work.

2 RELATED WORK

Unsupervised machine learning is applied in many disciplines to solve various problems. Examples are speech recognition [1], image classification [8], and breast cancer detection [4]. All of these examples have in common that, besides other optimizations, increasing the number of training examples dramatically improves accuracy. Consequently, researchers started to improve model training in order to be capable of dealing with larger amounts of data. Building model training systems in a way so that they can be scaled vertically, seems to be the most promising approach, as proven by almost any modern big data application.

Recent research has yield two major concepts for performing distributed model training: Either split the model over multiple nodes or distribute the training data itself. The first approach is called model-parallelism whereas the second approach is known as data-parallelism. Dean et al. [2] have implemented and evaluated both approaches within their framework DistBelief. Their work does not focus on a specific downstream task but supports any model training targeting stochastical gradient descent based optimization patterns. Although this approach yields competitive learning results, it still relies on a single point of weight synchronization (the so-called parameter server). This contradicts the idea of true model parallelism as all the weights are synchronized on just a single machine. Additionally, data throughput and overall system reliability are highly dependent on that parameter server.

Ji et al. [6] focus solely on data parallelism across up to 32 nodes within a cluster. By introducing mini-batches of weight updates they are able to achieve remarkable throughput leveraging modern hardware capabilities. However, in a distributed setup, model synchronization becomes necessary. This is implemented by broadcasting updated model parts to every other node, which is bound by the available network bandwidth.

Distributed model training, especially Word2Vec, has also been implemented using state-of-the-art batch-processing frameworks such as Spark. Moritz et al. [10] have shown the advantages of

employing such frameworks: Ease of setup and use, great scalability to a potentially unlimited number of nodes, and compatibility with existing systems. Besides these advantages, batching yields synchronization efforts between each chunk of training samples. Additionally, batching-based training can be seen as only pseudo-distribution, because workers are only utilized for a short amount of time in each training interval.

All these data-parallelized implementations have the downside of running into memory issues if the shared model exceeds a node's memory. Therefore, Stergiou et al. [12] proposed a system realizing true model-parallelism based upon a self-implemented graph processing framework. Following this concept, the authors are able to train on vocabularies containing more than 1.4 billion words. Optimizing various aspects of the original Word2Vec algorithm, the authors decrease inter-node communication and memory consumption. Although the optimizations are based on assumptions that theoretically decrease the model's accuracy, the authors have shown that in practice the performance of the resulting embedding is not affected remarkably. Furthermore, the reduced network traffic in combination with the model-parallelism leads to a tremendous speedup in training.

This work explores the applicability of the concepts and ideas presented by Stergiou et al. using an established toolset for reliable inter-machine communication.

3 DISTRIBUTION OF WORD2VEC MODEL TRAINING

Following the theory of model-parallelism, as described by Stergiou et al., allows refraining from any synchronization between machines during the training phase. This enables the system to work in a **fully asynchronous** fashion, embracing a **leaderless communication** protocol. Consequently, it becomes possible to utilize **heterogeneous cluster** setups, where the throughput of individual nodes is not bound by the least performing one.

In order to support the above-mentioned advantages of the model-parallel architecture, we decided to base our implementation[1] on the Akka toolkit. Realizing the inter-node communication using the actor programming model allows for easy decoupling of different nodes, inherently asynchronous execution (through fire-and-forget message exchange), and fault-aware protocol design [3]. Furthermore, deployment on heterogenous machines is a breeze since Akka runs natively on the Java Virtual Machine (JVM).

Our process of training word embeddings using the Word2Vec algorithm consists of four phases:

(1) Preprocessing
(2) Creating the distributed vocabulary
(3) Propagating training data and updating model weights
(4) Merging the distributed model partitions

According to the concept of model-parallelism, each node within the cluster is responsible for a specific part of the overall model. This means implicitly, that each node takes ownership of a defined subset of the training vocabulary. To create a distributed vocabulary, as described in section 5, it is necessary to, first of all, perform preprocessing, which is also done in a distributed manner. Details about the preprocessing are given in section 4 and following. Afterward, the actual training is performed by generating skip-grams on each node (see section 6). Lastly, the distributed partitions of the model have to be merged.

[1] https://github.com/Johannes-Schneider/RDSE/tree/master/implementation

4 PREPROCESSING

Before the actual training can be started, a few preprocessing steps are required. These are performed utilizing the cluster as much as possible. Therefore, we have to split apart the entire corpus file – which is only present on one node initially – and distribute the resulting partitions to other workers.

The corpus transfer is done in a streaming fashion, using Akka-Streams, which yields the advantage of applying back-pressure in case the receiver is getting flooded. Each receiving worker stores it's corpus partition in a local file for re-use later on. Parallelly, the worker already starts counting the number of unique words in the partition. This is a necessity for frequency-based sub-sampling, which is done when creating the final vocabulary (see section 5).

Simultaneously, the responsibilities for sub-sets of the vocabulary are defined by assigning each worker to a certain range of words on a hash-ring (consistent hashing). Since the used hash function works deterministically, introducing the workers to each other is enough to establish a common knowledge of responsibilities for words.

Each node reports the results of the local word counting to the responsible participant. This procedure leads to a state where every machine holds the total number of occurrences (in the entire training corpus) for each word it is responsible for.

5 CREATION OF THE DISTRIBUTED VOCABULARY

As soon as a node has received the all the word counting results from every other node, it can start determining the relevance of each word. This process is known as *Subsampling* and is meant to increase the statistical significance of the resulting embedding. Therefore, both words that occur too rarely and too often should be excluded from the further process.

a) Subsampling rare words: In the first step we remove all words that do not appear at least a minimum number of times within the corpus. By doing so, we ensure not to have words embedded that are very infrequent, misspelled or irrelevant for downstream tasks. As a practical side-effect, this step reduces the overall vocabulary tremendously and thus decreases the workload for training.

b) Subsampling overused words: The second step aims to remove words that occur too often. Typical examples of such words are 'a', 'the', or 'is'. These words appear in many different contexts, loosing any distinguishable semantic. Thus, embedding frequent words does not yield valuable information.
Therefore, we calculate $P(w)$ – the probability to keep a specific word in the vocabulary – for each word that is not *rare*. Formula 1 takes into account the word's count c_{word} and a threshold count t. t is the sum of all occurrences of *non-rare* words, multiplied by a frequency-cut-off value (typically $1e^{-3}$) [9].

$$P(w) = (\sqrt{\frac{c_{word}}{t}} + 1) * \frac{t}{c_{word}} \quad (1)$$

Subsampling, as described above, is done on a per-node basis, having the advantage of paralyzing this step and separating the duties per node to the vocabulary they are responsible for. However, this approach also yields a disadvantage: Subsampling a word locally on a node prevents other nodes from being aware of this decision. Thus, other nodes may consider an already subsampled word for training. Checking each time the relevance of a word is not an option, as it increases inter-node network communication dramatically.

In order to make the removal of certain words transparent to all other peers within the cluster, we are using Bloom Filters. Bloom Filters are a lightweight technique to determine whether an element is definitely not contained within a set [7]. Due to the efficient use of hashing and bit encoding, Bloom Filters are used to make statements about huge collections while consuming itself a comparably small amount of memory. Consequently, Bloom Filters are well suitable for the use case of communicating subsampled words to all other nodes. Creating and distributing these is the last step of the vocabulary creation phase. Each node receives a Bloom Filter that contains the corresponding part of the vocabulary from every other node. These are used in Section 7 to perform an upfront check, whether a certain word is excluded, thus avoiding unnecessary network traffic.

6 MODEL TRAINING

Word2Vec embeddings, in contrast to other concepts (such as bag-of-words representation [13]), aim to represent semantics of words. Semantic representation is achieved by looking at the surrounding words (neighborhood) of a target word within the training corpus. The general assumption is that words frequently occurring in the same context have a similar meaning and thus have to be represented correspondingly. Aligning such words happens iteratively during training. To identify words that appear in the same context, Word2Vec builds so-called *skip-grams*. Each skip-gram comprises two words from the training corpus: A *target word*, also referred to as *output word*, and a co-occurring *input word* from the target word's direct neighborhood.

In the unsupervised Word2Vec learning algorithm, skip-grams are used as training samples. The input word serves as a trigger, whereas the output portion represents the expected answer of the neuronal network. During training, the model is continuously adapted in order to minimize the error between actual and expected output. The neuronal network consists of an input and an output neuron for each word within the vocabulary. Triggering the neuronal network, therefore, works by stimulating the corresponding neuron of the input word. The resulting signal is forwarded through the hidden layer to trigger output neurons. This signal transfer is possible because all neurons of a layer are connected to all neurons of the following layer. Each of these connections is weighted, resulting in a set of weights for all outgoing or incoming edges per neuron. These weights are called *input weights* ($\check{w}$) for neurons

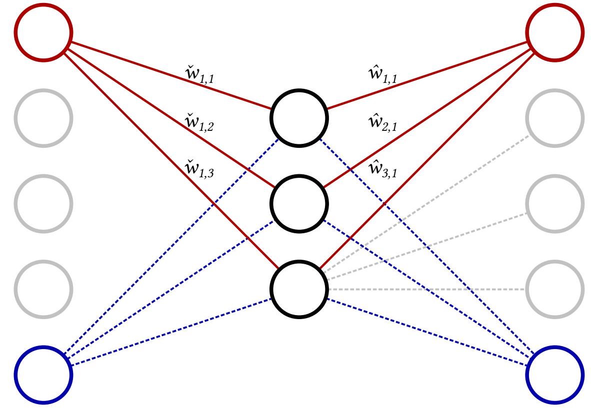

Figure 1: Illustration of an one-layer neuronal network

on the input layer and *output weights* ($\hat{w}$) for neurons on the output layer, respectively. An exemplary version of such a one-layer neuronal network is depicted in figure 1.

Following the concept of model parallelism, we distribute the vocabulary over all available nodes, rendering each node responsible for a subset of the vocabulary. Consequently, the input and output neurons and the corresponding input and output weights of the neuronal network are also distributed over all nodes. When looking at the exemplary model, depicted in Figure 1, we see the responsibilities of two different nodes in the cluster: The first node (painted in red) is responsible for the first word of the vocabulary. Therefore, it stores the corresponding input and output weights ($\check{w}_{1,j}$ and $\hat{w}_{j,1}$ with $j \in \{1...n\}$ ($n = 3$; the dimension of the hidden-layer / the embeddings). The same goes for the blue node, just for a different word. The neurons of the hidden layer (middle layer) are copied all together on every node, maintaining only the edges leading to available input and output neurons on the respective node.

Training the model is performed in parallel on all nodes. Therefore, each skip-gram is sent to the node responsible for the input portion first. The corresponding input neuron gets triggered and the resulting signals for the hidden layer (concrete: $\check{w}_i$) are forwarded to the node that is responsible for the output word of the skip gram. Applying these signals to the duplicated hidden layer on the output word's node yields the output signals (concrete: $\hat{w}_o$). The error is calculated by comparing these signals against the expected output (word of the skip-gram). Using the resulting error, we are able to calculate $\Delta_{\check{w}}$ and $\Delta_{\hat{w}}$. Applying Δ to the respective weights ensures reducing the prediction error and thus improves semantic representation of the embedding. While this can be done locally for $\Delta_{\hat{w}}$, we have to send $\Delta_{\check{w}}$ back to the node responsible for the input word.

Besides ensuring the stimulation of the expected neuron, it is also necessary to make sure all other neurons are less active, as they should have not been triggered according to the prediction. This can be achieved by updating the corresponding weights negatively. However, Mikolov et al. have shown that it is sufficient to only use a small fraction of neurons for negative updates. This concept is referred to as *Negative Sampling*. In addition to that, Stergiou

et al. have pointed out that drawing negative samples from local weights only, does not impact the embedding's accuracy significantly. Therefore, we select negative samples only on the node we calculate Δ on, saving inter-node communication overhead.

7 PROPAGATING TRAINING DATA

As described above, the distributed neuronal network consumes skip-grams for learning semantic relations between words. Skip-grams comprise words co-occurring within the training corpus, representing the idea of semantic relatedness when occurring in the same context.

As our approach distributes the training corpus over all participating nodes within the cluster, skip-gram creation happens on every node independently. Therefore, each node (we call this node a *producer*) parses its corpus partition word-by-word. Each word is checked against the Bloom Filter, as discussed in section 5. In case the Bloom Filter indicates that a word has been subsampled, it is ignored. Otherwise, the node generates skip-grams for all neighboring words. Words are considered as a neighbor if they are located within a defined *window size*, pre- or succeeding the target word. Similarly, as for the target word, neighboring words not satisfying the Bloom Filter check, are ignored. An example of the skip-gram creation procedure is the following:

Example 7.1. Given an exemplary sentence from the corpus:

"The quick brown **fox** jumps over the lazy dog."

The current target word is printed bold. Assuming that the *window size* is set to two, we would end up with the following skip-grams:

- (quick, fox)
- (brown, fox)
- (jumps, fox)
- (over, fox)

Each of the words within the neighborhood forms an independent skip-gram together with the target word. Doing so, the neighboring word takes the position of the *input word*, whereas the target word becomes the *output word*.

Since model training requires to stimulate input neurons first in order to obtain signals for the hidden layer ($\check{w}_i$), we first have to send a skip-gram to the node being responsible for the input word. We name this process skip-gram *encoding* because the responsible node translates the input word of the skip-gram into its corresponding input weights. A node responsible for a particular word and thus, having the ability to encode it, is referred to as *resolver*. To sum it all up, we are left with the following situation: All nodes within the cluster own a portion of the overall training corpus. By parsing it word-by-word, these *producers* generate skip-grams, which consist of two words. Each word has exactly one corresponding *resolver*, being able to *encode* it.

Following the approach of Stergiou et al., we want to perform training on the node responsible for *resolving* the target portion of the skip-gram – making that *resolver* node a *consumer*. The *consumer* is able to resolve the target word on its own; therefore, it is enough just to send the actual string. Depending on which node is responsible for the input word, we have three different scenarios of how the input weight for this word is obtained:

1.) The *consumer* is also the *resolver*: Consuming and resolving on the same node is the best case because the producer only has to send over both string representations of the skip-gram. The receiver (*consumer*) is able to perform a local lookup in order to obtain the needed weights. Transferring plain strings on the network is comparably cheap, which is beneficial for the overall throughput of the system.

2.) The *producer* is also the *resolver*: The second case requires more bandwidth than the first one, but still, it is not the worst-case scenario. Since the *producer* is responsible for resolving the word, this process can be done locally, avoiding inter-node communication. The *producer* finally sends over the string presentation of the target word and the input weight of the input word, which – even for low dimensional embeddings – requires multiple orders of magnitude more bandwidth than just the plain string.

3.) The *resolver* is a third node: The last case is also the worst-case scenario because the skip-gram has to be sent on a route requiring a detour. Hence, resulting in more network traffic. This case is handled by first sending the plain strings of the skip-gram to the *resolver*, which is then able to encode the input word and forward the result to the actual *consumer*. Although transferring the string representations to the *resolver* does not use up too much bandwidth, it is still an additional message that needs to be processed.

Since every node produces skip-grams from its local corpus partition, we run into the risk of flooding the network. As a consequence, heartbeat-messages between nodes are not processed regularly, leading to a connection loss. To overcome this issue, we utilize the concept of *work-polling* in our implementation.

Therefore, each producer initially sends out a batch of skip-grams to every available consumer, including itself. A dedicated *end-of-batch indicator* concludes each batch of skip-grams. All nodes apply a heuristic to ensure that in predominant cases the indicator is received last: To do so, they remember the last resolver they sent a skip-gram to that has not been the producer or consumer (third case as described above). At the end of the batch, the indicator is sent via the resolver to the consumer. By doing so, we ensure the indicator takes the longest route and thus, is processed after all skip-grams of the corresponding batch. The consumer uses the indicator as an opportunity to request the next chunk from the producer. This procedure leads to an even utilization of the available network and prevents overloads.

As described in section 7, a producer generates skip-grams targeting many consumers, depending on the responsibility for the words contained in each skip-gram. Therefore, the corpus file is read on demand because keeping it entirely in memory could exceed available memory on the node. However, it is not sufficient to naively continue parsing the file for each work request received.

Doing so would produce skip-grams for approximately all nodes within the cluster as words and their responsibilities are expected to be distributed evenly. We rather need skip-grams that are consumed by the node requesting more work. Our approach solves this problem by storing a separate file pointer for every consumer. If a producer receives a new work request from a consumer, it continues reading its local corpus partition file starting at the associated file pointer. The resulting batch of words is parsed, taking into account solely words that target the requesting consumer. Consequently, we generate only skip-grams that are relevant to the consumer.

Each producer generates multiple skip-grams from the same words pairs by parsing the corpus file repeatedly. Taking into account skip-grams multiple times leads to more fine-grained training

results, as the model deepens its knowledge about semantics. The number of iterations can be controlled by the *epoch* parameter, trading of training time against model accuracy. With respect to generating multiple skip-grams, each producer also stores the current epoch for every consumer. Consequently, producers send the current epoch for the corresponding producer alongside the actual skip-gram. Upon receiving a skip-gram, the consumer adjusts the learning rate according to the current epoch.

8 MERGING DISTRIBUTED MODEL PARTITIONS

Once the training is finished, the distributed model partitions need to be merged and persisted for further use. However, realizing that training is finished for a particular node, is not a trivial task. This is, because the producer does not share any progression information (besides the epoch for each skip-gram) with the consumer. Thus, the consumer is not able to track how many more skip-grams need to be processed.

We solve this issue by signaling the end of training to individual consumers, per producer. This signal is emitted once a consumer requests more work, but neither the file pointer nor the current epoch for that consumer can be further increased. By storing all the producers that have already sent this message, each consumer is able to determine whether more skip-grams are expected to arrive.

As soon as all producers have declared that no more skip-grams can be produced, the consumer initializes the model partition transfer. The model partitions are sent to the node, which initially contained the entire training corpus. Using Akka-Streaming, we send over all the input weights for every word stored on the consumer node, which are then written to disk. Although both the input and output weights are updated during the training, we are only interested in the input weights since these represent words in the trained vector space.

After a node has transferred its results, it has to remain active because other participants within the cluster may still rely on it as a resolver. To allow a graceful cluster shutdown, a consumer which finished its partition transfer also unsubscribes from all other nodes. As soon as a node has no more subscribers and is done with its training, it is save to leave the cluster.

9 EVALUATION

For evaluating our implementation, we measure the utilization of the cluster (CPU, main memory, and network), the overall training time, and the resulting model‘s performance. Our tests have been performed on a heterogeneous cluster consisting of twelve nodes. Each of the nodes is equipped with an Intel Xeon CPU E5-2630 @ 2.2 GHz operating 20 cores each. Four of the twelve nodes have 32 Gigabytes of main memory, whereas the other eight nodes own 64 Gigabytes. The entire cluster is interconnected using a Gigabit Ethernet connection. All nodes run Ubuntu 18.04 LTS as their operating system. Our implementation is executed on a Java Virtual Machine (JVM) limited to 25 Gigabytes of RAM. The proposed implementation is built, deployed, and executed using a self-developed python toolset[2].

[2]https://github.com/Johannes-Schneider/RDSE/tree/master/scripts

Training is performed on a cleaned English Wikipedia corpus resulting from a dump of the second quarter of 2019. The corpus contains roughly 2.5 Billion words. We limited the training to `1` epoch, using a learning rate of `0.025`, building skip-grams in a window of `5`, and reducing the dimensionality to `100`. Additionally, each node is allowed to spawn up to `16` training workers.

Utilization statistics have been observed using established bash tools on all running nodes. During the training phase, we found an average CPU utilization of `1400%` (out of a maximum of `1600%`). Interestingly, increasing the number of allowed workers does not increase CPU utilization. We initiated multiple runs with various subsampling configurations, leading different vocabulary sizes. Depending on the vocabulary size, the memory consumption ranges from `8` up to `22 Gigabytes` per node.

Network traffic has been measured employing *bmon*, a reliable network bandwidth monitor. As expected, *bmon* showed maxed-out network utilization of, on average, `100 Megabyte` of data per second received and sent per node. Furthermore, we noticed growing heartbeat intervals and extremely large message buffers (up to `10` Million buffered messages) on single nodes. With respect to these observations, it comes as no surprise that increasing the worker limit does not increase CPU utilization.

We observed that training time highly depends on chosen hyperparameters like the number of epochs or the window size. Executing training, parameterized as described above, takes approximately `18 hours` to complete. In comparison to the gensim implementation, which runs this training setup in about `9 hours`, our results can not be seen as competitive.

The resulting model‘s performance is assessed with the *Google analogy test set*. Therefore, we make use of gensim‘s evaluation capabilities. Unfortunately, our model hardly solved any analogies. The tests revealed that round about `50%` of the analogies could not be executed due to missing words in the vocabulary. In contrast, a gensim-trained model with the same hyperparameters achieved `24.4%` accuracy with only `0.3%` out-of-vocabulary failures.

10 CONCLUSION

The findings described in Section 9 do not meet our expectations. In the following, we aim to discuss potential causes that might be responsible for these deviations.

First of all, we observed a heavy overutilizing of the network during the training phase. In that phase, skip-grams are the predominant information that is distributed. Network exhaustion is therefore caused by sending too many skip-grams at a time. Thus, we suspect our employed work-polling-heuristic to be the principal cause. While the heuristic takes care of not flooding a specific consumer with skip-grams for training, it does not take resolving nodes into account. In particular, fully occupied resolvers implicitly receive even more work when idle nodes request more skip-grams. This effect amplifies itself, leading to growing message queues, massive delays in training, and unbalance workload within the cluster.

Looking at an unusually high number of out-of-vocabulary errors within the evaluation outcome, we assume the applied subsampling strategy to be faulty. Even after consulting the work of Mikolov

et al. [9], Stergiou et al. [12], and the gensim implementation[3], we were unable to identify a specific implementation error. More surprisingly, we discovered that the vocabulary size is comparable to a model trained by gensim.

Besides the out-of-vocabulary issues, there are a few other possible causes for the bad analogy performance. Firstly, the gradient calculation might be erroneous. Secondly, the selected negative samples are potentially unevenly distributed. Lastly, applying gradients to model weights possibly introduces inaccuracies. For all of these causes, we reiterated our implementation multiple times without finding the root cause of poor accuracy.

Nevertheless, the approach seems promising as handling huge vocabularies is only possible in a distributed environment. We have shown that splitting the vocabulary amongst several nodes supports the approach of model parallelism and thus makes training on unlimited corpora feasible. In particular, using the Akka Toolkit for cluster management, reliable inter-node communication, and encapsulating model partitions has shown great potential. The actor programming concept is especially useful because communication is reduced to only gradients and skip-grams, while the internal state and behavior of actors (model partitions) do not need to be exposed.

REFERENCES

[1] G. E. Dahl, D. Yu, L. Deng, and A. Acero. 2012. Context-Dependent Pre-Trained Deep Neural Networks for Large-Vocabulary Speech Recognition. *IEEE Transactions on Audio, Speech, and Language Processing* 20 (2012), 30–42.

[2] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *Proceedings of the International Conference on Neural Information Processing Systems (NIPS'12)*, Vol. 1. Curran Associates Inc., 1223–1231.

[3] Munish Gupta. 2012. *Akka essentials.* Packt Publishing Ltd.

[4] Hugh Harvey, Andreas Heindl, Galvin Khara, Dimitrios Korkinof, Michael O'Neill, Joseph Yearsley, Edith Karpati, Tobias Rijken, Peter Kecskemethy, and Gabor Forrai. 2019. *Deep Learning in Breast Cancer Screening.* Springer International Publishing, 187–215.

[5] W John Hutchins. 2004. The Georgetown-IBM experiment demonstrated in January 1954. In *Conference of the Association for Machine Translation in the Americas*. Springer, 102–114.

[6] S. Ji, N. Satish, S. Li, and P. K. Dubey. 2019. Parallelizing Word2Vec in Shared and Distributed Memory. *IEEE Transactions on Parallel and Distributed Systems* 30, 9 (2019), 2090–2100.

[7] Steven Glen Jorgensen. 2015. Bloom filter with memory element. US Patent 9,171,153.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the International Conference on Neural Information Processing Systems*, Vol. 1. Curran Associates Inc., 1097–1105.

[9] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *Workshop Proceedings of the Internation Conference on Learning Representations.*

[10] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I Jordan. 2015. SparkNet: Training Deep Networks in Spark. *arXiv preprint arXiv:1511.06051* (2015).

[11] Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks.* European Language Resource Association, 45–50.

[12] Stergios Stergiou, Zygimantas Straznickas, Rolina Wu, and Kostas Tsioutsiouliklis. 2017. Distributed Negative Sampling for Word Embeddings. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI'17).* AAAI Press, 2569–2575.

[13] Yin Zhang, Rong Jin, and Zhi-Hua Zhou. 2010. Understanding Bag-of-Words model: A Statistical Framework. *International Journal of Machine Learning and Cybernetics* 1, 1-4 (2010), 43–52.

[3] https://github.com/RaRe-Technologies/gensim