Project

Network-efficient distributed Word2Vec for large vocabularies

Markus Gabriel
01326657

March 2019

The complete source code is available at
`https://github.com/MGabr/evaluate-glint-word2vec`
`https://github.com/MGabr/glint-word2vec`
`https://github.com/MGabr/glint`

1 Introduction

With rising amounts of data there is a need for machine learning on large data sets which may not fit in the memory of a single machine and which should be processed fast. While this can be handled with data-parallel approaches which distribute data partitions to different machines performing computations in parallel, there is also a need to handle machine learning problems with large models which may themselves not fit in the memory of a single machine. To handle these problems model-parallel approaches can be applied. These approaches partition the model and store the model partitions on different machines.

One well known machine learning problem is the training of word2vec [1] word vectors. The traditional natural language processing use cases of training word vectors on text corpora have vocabularies of several hundred thousand to several million words and therefore usually don't require model-parallelism. There are however alternative use cases for word2vec with vocabularies of several hundred million words. Researchers at Yahoo! [2] came up with such a use case for Sponsored Search and developed a system for training word2vec data- and model-parallel. Their main contribution was training word2vec in a network-efficient way significantly reducing the required network bandwith in comparison to existing data- and model-parallel word2vec systems.

The aim of this project was to implement the proposed system, for which to the best of my knowledge there is no open-source reference implementation, as Spark ML package providing both the RDD-based and DataFrame-based API. The API is mostly the same as the existing word2vec implementation on Spark [3]. This existing implementation has to broadcast the word vectors and therefore the size of the word vectors is limited to 8GB because of Sparks broadcast size limit [4]. My implementation has no

such limits and allows training very large models which do not fit in the memory of a single machine. It also supports word to vector transformations and synonym computations using the parameter servers instead of just training word vectors which is an aspect not described in the original paper. Further, it is possible to perform the setup of the parameter servers automatically and transparent to the user. The implemented system was then also trained on the german wikipedia and a large english text corpus and evaluated. Observations are presented in this article.

2 Background

A widely used concept for model-parallelism is the parameter server architecture. A parameter server is a high performance, distributed, in-memory key-value store which supports simple `get` and `put` operations to either get or change model weights [2]. There are a couple of open-source parameter servers like PS-Lite [5], Multiverso [6] and Glint [7]. I used a heavily adapted version of the Glint parameter server which is written in Scala, uses the Akka framework [8] and promises easy interoperability with Spark.

The approach of Ordentlich at al. [2] is however not based on a traditional parameter server but a parameter server offering specialized operations for word2vec training. To understand the need for these operations it is important to know about the limitations of existing word2vec systems using the traditional parameter server architecture like Microsofts Distributed Word Embeddings on top of their DMTK parameter server [9]. These systems distribute the input and output vectors to parameter server shards with each shard getting a distinct subset of the vocabulary. The client nodes then process the words of the corpus in minibatches, find the corresponding context words and random negative examples, make `get` requests for the relevant vectors from the parameter server shards, compute the gradients and then update the vectors with `put` requests. The required network bandwidth for each minibatch word is

$$r(w, n, d) = (2 + w * n) * d * 4 \quad [2]$$

bytes with w being the average window size, n the number of negative examples and d the word vector dimension. This assumes negligible repetition of words and negative examples in the minibatch and transportation of the gradients as arrays of 4 byte single precision numbers. Ordentlich et al. [2] explained that for parameters within the ranges recommended in the original word2vec paper [1] these bandwidth requirements are too impractical for training on large amounts of data. Their approach to solve this issue is based on the following new ideas.

- The parameter server shards are partitioned by columns. So each shard contains all words but only a part of each words input and output vector.
- The partial dot products are computed on the shards themselves and then added up on the client to form the complete dot product. This is possible since each shard contains partial vectors for each word but also means that the client has to make a request to each parameter server shard.
- Random negative examples are also generated on the shards themselves using a seed broadcasted from the client.

- Finally, these previous ideas are exploited to prevent the transmission of word vectors or vector gradients across the network. Instead it is sufficient to only request the partial dot products, add them up on the client to form the complete dot products, compute the weights for the gradient updates using the gradients descent steps with the sigmoid function applied to the dot products and finally broadcast these weights to the shards which compute the actual vector gradients and update the words vectors by them.

This reduces to required network bandwith to

$$r(w, n, S) = (w * (n + 1)) * S * 4 \quad [2]$$

bytes with S being the number of parameter server shards. So their approach reduces the bandwidth to about S/d of the previously required bandwidth. For 10 parameter server shards and a word vector dimension of 300 this would be 1/30.
To support this efficient way of training word vectors a column partitioned parameter server with two specialized operations is needed. `dotprod` takes as parameters an array of input words and a 2D array of their corresponding context words as well as a seed. It produces an array tuple consisting of the array of dot products between the input words and each of their context words as well as the array of dot products between input words and the random negative examples produced using the supplied seed. `adjust` takes the same parameters and additionally two arrays with weights for the positive and negative word updates. These weights are then used to update the vectors of the input, output and negative example words according to mini-batch gradient descent rules. More detailed information and pseudo-code can be found in [2].

3 Implementation

The implementation of this project is split into three repositories. The first one contains a fork of `Glint` adapted to support the specialized word2vec operations. The second repository `glint-word2vec` is the actual ML package which uses the Glint fork. Finally, `evaluate-glint-word2vec` contains scripts for evaluating the ML package and comparing it to the existing Spark word2vec implementation.

3.1 Glint

Most of the functionality can be found in the Glint fork. Many adaptions had to be made. The following paragraphs describe the main points.

Merging existing feature branches: The last commit to the Glint project is from July 2017 and the second-last commit, which has actual functionality changes, is from October 2016. Besides requiring the outdated sbt 0.13 release to build the project many versions were outdated. There existed a branch with an update of the Akka version which uses the low latency transport protocol Aeron [10] and a branch which added the possibility of starting a Glint cluster directly on Spark instead of having to start Glint parameter servers manually on machines. I merged these branches, fixed occurring issues and made some additional adaptions like version updates.

Column-partitioning: Glint only supports parameter servers which are partitioned by rows so additional support for column partitioning had to be added.

Spark Integration: There is already some initial support for starting a Glint cluster on Spark but with limitations. A partition is created for each executor and the `foreachPartition` method is used to start a parameter server once on each executor JVM. This approach does not allow creating a fixed number of parameter servers and does not work if there are multiple executor cores. To solve this issue a `PartitionMaster` Akka actor which holds the available partitions / shards is created. The `foreachPartition` method is executed with enough partitions (number of executors times executor cores) so that all executors will run this method. It then checks if there is already a parameter server started on the executor JVM and if not tries to claim a partition from the `PartitionMaster` using Akka actor communication as in other parts of Glint. If there is an available partition a parameter server is started. To avoid confusion, note that the partitions of the `PartitionMaster` refer to parameter server shards while the partitions of `foreachPartition` refer to partitions for Spark tasks. Listing 1 shows pseudo code for starting the parameter servers on Spark.

```
// start partition master
val nrOfPartitions = getNumExecutors(sc) * getExecutorCores(sc)
sc.range(0, nrOfPartitions, numSlices = nrOfPartitions).foreachPartition {
  case _ =>
    lock.acquire()
    if (!started) {
        started = true
        (partitionMaster ? AcquirePartition()).flatMap {
            case Some(p: Partition) =>
                // start parameter server, responsible for partition p
        }
    }
    lock.release()
}
```

Listing 1: Pseudo code for starting parameter servers on Spark

The downside of these integrated parameter servers is that it is now necessary to request enough memory so that each executor can hold its data partitions as well as the model partition even though only executors actually holding the model partition would require it. Further, the executors holding parameter server shards have to perform their normal computations as well as the parameter server computations. If these issues are not worth the easier usage it is possible to run the parameter servers in a separate Spark application. This application starts the parameter servers in the same way but performs no other non-parameter server computations. The Word2Vec implementation accepts the IP of the node on which the parameter server master runs as argument and then connects to it. The parameter server Spark application can be terminated automatically when the word2vec Spark application finishes.

Saving to and loading from HDFS: The simple option for storing the word vectors would be to load all the word vectors to the driver and then storing them as parquet using the common Spark methods. In this case the driver would either have to have enough memory to store all word vectors or the word vectors would have to be requested from the parameter servers in batches. Further, initializing parameter server matrices with saved data would be difficult since Akka has a maximum message size. To avoid the word vectors passing through the driver altogether I implemented a `save` method for matrices on the parameter servers which stores the data of their partition directly to HDFS as well as a `load` method allowing matrices to load their initial, previously saved data directly from HDFS.

Word2vec matrix: Glint provides distributed vectors and matrices of type int, long, float and double supporting `pullRows`, `pull` and `push` operations. Efficient word2vec training, however, requires a float matrix with custom `dotprod` and `adjust` operations. I therefore implemented a word2vec matrix as a special version of the existing float matrix. The implementation of the `dotprod` and `adjust` operations follows the pseudo code from [2] but there are a couple noteworthy technical details.
To generate the negative examples a unigram table has to be built which represents the distribution of the words in the corpus. Therefore each matrix partition has to know the occurrence count of each vocabulary word. The array of occurrence counts can be quite large and there are limits on the maximum size of Akka messages. Therefore, when running the parameter servers in the same Spark application, the array is broadcasted as a Spark broadcast variable and the Akka actors for each matrix partition are created directly in the `foreachPartition` method when the Glint parameter servers are started on Spark and in this way deployed locally on the executor instead of having their initialization data sent over the network using Akka remote deployment. If the system runs connected to parameter servers in separate Spark applications the array is instead saved temporarily to HDFS and loaded from there on each matrix partition actor.
Secondly, Glint matrices are adjusted to store the data as 1D array instead of as 2D array. This enables the usage of efficient BLAS matrix operations on the partition data.
Another important aspect is the support for multithreading on parameter servers. A fundamental guarantee of Akka is that messages to an actor are not processed concurrently. Therefore only at most one thread handles incoming messages. In our use case this would result in the time to compute the `dotprod` and `adjust` methods becoming the bottleneck of the entire system. Therefore, as in the original approach [2], a thread pool is used for processing `dotprod` and `adjust` messages asynchronously. This means that the matrix partition input and output weights are accessed as shared mutable state and gradient updates can overwrite each other. According to previous research [16] this can, however, still achieve a nearly optimal rate of convergence when the optimization problem is sparse.
Finally, as also described in the original paper [2], the `adjust` method only applies the gradient updates at the end. With naive temporary allocation of arrays holding the gradient updates there would be excessive garbage collection overhead. Besides significant performance degradation this also caused Aeron timeouts. As solution arrays are taken from and returned to a pool of zero-valued arrays of the same size. This array pool is thread local to remove the need for synchronization.

Additional word2vec operations: One goal of this implementation was to also use the parameter servers for the actual use of the word vectors since those might not fit into the memory of a single machine. While the existing `pull` or `pullRows` operations would be enough to transform words to vectors three additional operations were required to support the existing methods provided by the ML- and MLlib-API of the Spark implementation.
First, the API provides a method for finding synonyms of a word or vector. To compute the synonyms / closest words of a word the cosine distance of the word vector to all other word vectors has to be computed. The words whose vectors have the least cosine distance to the query word vector are the synonyms. For a query vector $\vec{v}$ and a word vector matrix W the cosine distance of the query vector to the i-th word vector is
$cos\ \theta_i = \frac{\vec{v}\ W_{i*}}{\|\vec{v}\|\ \|W_{i*}\|}$ which can be rewritten as $cos\ \theta_i = \frac{(W\frac{\vec{v}}{\|\vec{v}\|})_i}{\|W_{i*}\|}$.

One can observe that this can be efficiently computed if we have a method for multiplying the word vector matrix with a given vector and a method for getting all word vector norms. Therefore the additional operations `multiply` and `norms` were added. `multiply` simply performs the multiplication of the matrix partition with the corresponding partition of the query vector and then sums up the multiplication results of all partitions on the client. `norms` computes the dot product of each partial word vector with itself, sums up those partial dot products and then takes the square root of the sum. Both operations allow getting only the results for a range of word vectors to allow splitting up the requests into smaller requests which are not bigger than the Akka maximum message size.
The third operation that was added is `pullAverage`. This works like `pullRows` but instead of accepting an array of row indices it accepts an array of arrays of row indices. The result for each array of row indices is the element-wise average of all those rows. This method improves the performance of the Spark ML transformation which returns the average vector for a list of words and prevents having to send all rows to average to the client.

Integration tests on Spark: With the added better Spark integration and HDFS storage there is more interaction with the Spark framework which can not be easily tested with unit tests or integration tests in Spark local mode. To nevertheless have reliable test coverage for the added functionality I added an integration test setup which relies on a tiny 4 node Spark standalone cluster in a docker container presented in a Spark Summit talk [11]. This docker image was extended to also provide HDFS. Both the existing tests and the integration tests are integrated in a Continuous Integration pipeline running on Travis CI [12].

3.2 Glint-word2vec

The `glint-word2vec` ML package reuses large parts of the code of the existing Spark word2vec.
Regarding the fit stage, the preprocessing stayed the same and only the core training was changed to make `dotprod` calls to the parameter servers, sum up the received partial dot products, compute the gradients and then make `adjust` calls with the computed gradients.

Looking at the transform stage, more changes were necessary. In the existing Spark Word2Vec implementation the map of all words to their word vectors is broadcasted to all executors which can then transform the words in their partitions using a user defined function applied to the input column. Since this does not work for large models my system uses the parameter servers for transformation also. Parameter servers can be started and initialized with saved word vector data. Given the index of a word the corresponding word vector can then be retrieved from the parameter servers. This means that only a map of words to their indices has to be broadcast to all executors. To get an impression on the scalability limits of this approach consider a vocabulary of 80 million words with an average word length of 10. Java strings have an overhead of about 40 bytes and store 2 bytes per character. So a 10-character string consumes about 60 bytes [13]. A java `HashMap` has an overhead of 36 byte per entry and an integer takes up 4 bytes [14]. The map of words to indices would therefore take 80 million * (60 + 4 + 36) bytes which is exactly 8GB and not possible anymore with this implementation because of Sparks broadcast limit of 8GB [4]. This scalability limit could, however, be removed easily by using a distributed map implemented on top of Glint as parameter server-like system in which each parameter server stores partitions of the word to index map. A request for the vector of a given word would then involve first requesting the index from the word index map and then request the vector from the original word2vec matrix.
Besides this solveable scalability issue, the use of parameter servers also introduces asynchronous operations and makes per-word transformations impossible since this would mean blocking server requests for each word. Instead batches of 10.000 word indices at a time are transformed. This is enabled through the use of the experimental `foreachPartition` method on `DataFrame` which allows iteration over the whole partition. `Row` instances with an additional output column containing the word vector are returned so that the result of `foreachPartition` is again a `DataFrame` with a correctly transformed schema. Regarding the MLlib version of my system, there is an additional transform implementation accepting word iterators as arguments to allow a similar prevention of blocking on each word. In this case the user himself would, however, have to call this method inside `foreachPartition`.
Apart from that `glint-word2vec` passes most of the work on to `glint` and most adjustments are only to fit the system to the Spark ML and MLlib API. Further, python bindings are provided which are also based on the python bindings of the existing Spark word2vec implementation. There are also Spark integration tests for this package which use a small test data set consisting of wikipedia articles of some countries and capital cities.

3.3 Evaluate-glint-word2vec

This repository contains the python scripts used to perform the evaluation. The kinds of evaluations performed are described in the next section. The README in the git repository contains further information on how to use them.

4 Observations and Evaluation

4.1 German Wikipedia

As initial evaluation the system was trained on a dump of the german wikipedia and evaluated on the country capital analogies from the original word2vec paper as well as the SimLex-999 and the WordSim-353 evaluation sets. The wikipedia dump consists of more than 940 million words and a vocabulary of more than 2 million words. The country capital analogies were translated to german and as german versions of the SimLex-999 and the WordSim-353 evaluation sets I used translated and re-scored versions from [17]. The scores are the number of correct analogies computed by the word vectors and the Spearman correlation coefficient between the cosine similarity computed by the word vectors and the similarity specified in the evaluation set. The analogies were computed by subtracting the first country vector (in this case *China*) from the first capital vector (in this case *Peking*) and adding it to each country vector. The nearest vector is then the predicted capital / analogy.

A first observation was that a batch size of 50 and a learning rate of 0.01875 in combination with a sufficiently large amount of partitions like 150 resulted in exploding gradients and finally NaN values for all word vectors. This issue can be solved by reducing the batch size to 10. However, when further increasing the partitions this issue still persisted and could only be tackled by also decreasing the learning rate. For 300 partitions a batch size of 10 and a learning rate of 0.01 prevented exploding gradients. Regarding this problem there is some research showing that for asynchronous SGD low batch sizes and low learning rates lead to the best performance and that halving the learning rate and the batch size when the number of workers is doubled can be a good heuristic [20]. This article did, however, only talk about performance and did not mention exploding gradients specifically. The distributed word2vec from Ordentlich et al. [2] did also not adapt the batch size or learning rate for a large number of workers. Further, a lower batch size would lead to a longer runtime and a lower learning rate could also result in more iterations being required to attain the same word vector quality, therefore again resulting in a longer runtime. Because of this the solution of adapting the batch size and learning rate seems impractical. Looking into the underlying reasons for the exploding gradients it seems that frequently occurring words cause this problem. For example, the german words *der*, *und* and *die* occur in most sentences. These words will therefore be updated concurrently by many workers which do not know about updates already performed by other workers. This results in them taking on large values and propagating them on to other words. One approach to mitigate this issue is to use subsampling and remove frequent words from many sentences. Ordentlich et al. [2] used quite strong subsampling with a subsampling rate of 1e-6 for their evaluation on a large english text corpus. I also applied subsampling with this rate but this did not solve the problem since frequent words are still often used as random negative examples and therefore updated concurrently by many workers. I am unclear as to why this problem did not seem to occur in the original implementation or is at least not described. Surprisingly, even adjusting the generation of random negative examples to take into account the applied subsampling did not help. Finally, simply removing certain frequent words altogether solved the issue. I used the Spark `StopWordsRemover` with the default list of german stop words available at [21]. In some constellations I still experienced exploding gradients and had to

therefore add a few custom stop words. Table 1 shows the top 60 most frequently occuring words with gray words being in the custom stop word list, bold words being accepted words and all other words being in the default stop word list.
Because of these observations my recommendation for using Glint-word2vec would be to remove stop words before training the model. In most cases this should not pose a problem since these words do not carry that much meaning. This stop word removal should, however, be taken into account when choosing the window size. In most cases a smaller window size will have to be chosen to get the same effect.

word	frequency	word	frequency	word	frequency	word	frequency
der	34,853,996	wurde	5,811,940	sie	3,263,048	the	1,670,540
und	25,634,957	auf	5,714,842	zum	2,771,207	de	1,613,166
die	25,054,361	ist	5,557,569	einer	2,637,739	of	1,594,814
in	21,134,518	zu	5,458,162	es	2,500,150	unter	1,522,059
von	13,589,248	ein	5,356,591	durch	2,335,903	wurden	1,506,236
im	10,250,512	eine	5,064,983	zur	2,174,360	weblinks	1,422,277
des	9,364,129	an	4,500,140	nicht	2,058,049	**jahr**	1,404,475
den	9,184,344	sich	4,310,527	einem	2,019,172	vor	1,393,012
kategorie	8,845,047	nach	4,121,118	über	1,978,015	dass	1,388,129
mit	8,401,218	war	3,995,021	werden	1,967,057	vom	1,372,371
das	7,795,940	am	3,977,131	sind	1,944,406	wie	1,316,136
er	7,387,015	aus	3,643,768	wird	1,917,685	seine	1,265,241
dem	6,167,530	bei	3,607,932	um	1,820,541	seit	1,222,314
als	5,949,245	auch	3,591,179	oder	1,812,309	ab	1,179,403
für	5,898,101	bis	3,485,115	einen	1,703,041	**deutscher**	1,171,524

Table 1: Frequencies of the top 60 words in the german wikipedia dump. Gray words were added to a custom stop word list, bold words were accepted and all other words were already in the default german stop word list of Spark [21]

To get insights of the scaling behavior of the distributed Word2Vec implementation I trained word vectors with different numbers of executors and evaluated them on the german wikipedia on the scores mentioned previously. Table 2 shows the evaluation results. The gensim type refers to a single-machine implementation of word2vec using the Gensim library [22], the ml type refers to the standard Spark word2vec implementation and the glint type to my implementation. The postfixes aeron and tcp specify the underlying transport protocol used by Akka and therefore the communication with the parameter servers. For all glint types 5 parameter servers were used and a plus indicates that they were run as separate Spark application instead of being integrated in the same Spark application. A forward slash separates the number of cores requested for the main application and the parameter server application. All types have a vector dimension of 100, a window size of 5, a learning rate of 0.01875 and the removal of stop words in common. The gensim and glint types use subsampling with a rate of 1e-6 and the glint types also have a batch size of 50. As executor and driver memory I used 30GB for all evaluations. The SimLex coverage is 968 of 999 and the WordSim coverage is 344 of 353.

type	executors	cores	train time	eval time	analogies	SimLex	WordSim
gensim			166 mins	1 min	2/11	0.255	0.536
ml	1	1	382 mins	9 mins	4/11	0.264	0.494
ml	5	1	82 mins	9 mins	2/11	0.206	0.413
ml	5	30	16 mins	9 mins	3/11	0.233	0.454
ml	10	30	11 mins	9 mins	2/11	0.173	0.346
glint aeron	5	30	116 mins	5 mins	7/11	0.286	0.522
glint aeron	10	30	71 mins	4 mins	8/11	0.279	0.541
glint aeron	15	30	69 mins	4 mins	7/11	0.287	0.533
glint tcp	5	30	45 mins	2 mins	7/11	0.278	0.526
glint tcp	10	30	36 mins	2 mins	5/11	0.277	0.531
glint tcp	15	30	32 mins	2 mins	7/11	0.282	0.535
glint tcp	17	30	25 mins	2 mins	6/11	0.276	0.531
glint tcp	5 + 5	30	32 mins	2 mins	7/11	0.28	0.528
glint tcp	10 + 5	30	29 mins	2 mins	7/11	0.283	0.543
glint tcp	12 + 5	30	17 mins	3 mins	6/11	0.275	0.542
glint tcp	12 + 5	40 / 30	15 mins	2 mins	7/11	0.277	0.529
glint tcp	12 + 5	40 / 40	38 mins	2 mins	6/11	0.281	0.529
glint tcp	12 + 5	40 / 20	26 mins	2 mins	5/11	0.283	0.519

Table 2: Evaluation of Word2Vec implementations with different numbers of executors and executor cores

Looking at the evaluation scores in table 2, one can see that Glint-word2vec surprisingly always produces better word vectors concerning the evaluation metrics except for WordSim. Regarding the difference to the default Spark implementation, one reason for this could be that Glint-word2vec uses negative sampling instead of hierarchical softmax and also applies subsampling. Both negative sampling and subsampling decrease the runtime, with subsampling resulting in a speedup of 2 to 10 times. According to [1] negative sampling may even improve word vector quality at some times especially for frequent words while subsampling results in significantly better word vectors for uncommon words. More interesting are, however, the better scores of my implementation when compared to the single-machine gensim implementation with the same settings. I am unclear about why my implementation performed even better. Maybe the use of mini-batch gradient descent instead of stochastic gradient descent contributed to the improved performance.

Regarding scalability, we see that the training time of the original Spark implementation decreases well with an increasing number of executors or executor cores. The quality of the word vectors however strongly decreases when using a large amount of total cores. The glint type implementation seems to not suffer from quality decrease when increasing the total executor cores. The scores vary a bit but there are no clear trends visible. It, however, has a significantly higher training time even though it uses strong subsampling. The training time also decreases when increasing the number of executors but some strange decreases can be observed. For glint tcp with separate parameter servers doubling the number of executors from 5 to 10 decreases the training time by only about 10% while increasing it by 2 more to 12 executors nearly halves the training time. Overall my implementation seems to scale not perfectly but good enough.

For completeness, table 2 also shows the evaluation runtime. Here the standard Spark implementation takes significantly longer. The probable reason for this is that it has to broadcast the whole vectors to the executors while in Glint-word2vec each parameter server shard only needs its partition and also loads it directly from HDFS.
Another interesting aspect are the different Glint-word2vec types. First I evaluated my implementation using Aeron as transport protocol. Aeron is a protocol for efficient reliable UDP unicast optimized for low latency and high throughput and also supported by Akka as default protocol [10]. While it appears like a good fit and is also used in the DL4J deep learning library [23] it lead to poor performance in my case and was even worse than the TCP protocol. Coupled with this were errors because of insufficient storage in the underlying `/dev/shm` directory when the number of executors was increased to 15. While this could be mitigated by decreasing `aeron.term.buffer.length` from 2^{24} to 2^{20} the performance improvements gained from more workers were minuscule because of that. There might be some configuration fine tuning or other approaches which could make my system faster with Aeron than with TCP but in my case with mostly default settings and only some experimentation I could not achieve this. One noteworthy detail is that after changing the protocol from Aeron to TCP I again experienced exploding gradients and had to add the mentioned custom list of stop words. I am not completely clear about why this happened but my guess would be that the speedup through TCP lead to more updates being applied between a specific `dotprod` and corresponding `adjust` request and therefore the values received through `dotprod` being even more out of date.
The other evaluations compare the two modes in which the developed system can be run. As discussed earlier, it can be run with parameter servers conveniently started in the same Spark application or with a separate parameter server Spark application being started beforehand. The problem with the first approach is that the normal executor tasks compete for the resources with the parameter server shard running on the same executor. Since all executors have to communicate with the parameter servers the parameter server throughput is very important for the system efficiency. The effect of this can be observed in the training time curve which shows that separate parameter servers allow for better scalability.
The final experiments were varying the number of cores available to the parameter servers. The aim for these experiments was to see if the parameter servers are maybe already working at maximum capacity and if the system could be speeded up significantly by increasing the number of parameter server cores. Surprisingly both decreasing and increasing the number of cores lead to performance decreases. Increasing the number of cores even more than doubled the training time. The only plausible explanation I could come up with is that the cluster may have set the number of virtual cores higher than the actual number of available virtual cores and that some overhead caused by the larger fixed size thread pool caused this performance degradation. In this case the actual number of cores might be somewhere around 30 for the observations to make sense. Unfortunately this gives us no further insights into the capacity of the parameter servers.

Figures 1 and 2 present visualizations of the word vector quality in relation to country capital analogies. They consist of two-dimensional PCA projections of the corresponding word vectors similar to figure 2 from [1]. The impressions of these images match the analogy scores from the evaluations. For Glint-word2vec, countries and capitals are separated more clearly into capitals on the left and countries on the right and the analogy vectors more often have the same direction. Additionally, the structure of the word vectors appears more stable and the value ranges stay nearly the same. In contrast, the visualization for the word vectors trained with the standard Spark implementation change their overall structure and also have an increased value range when trained with more total executor cores.

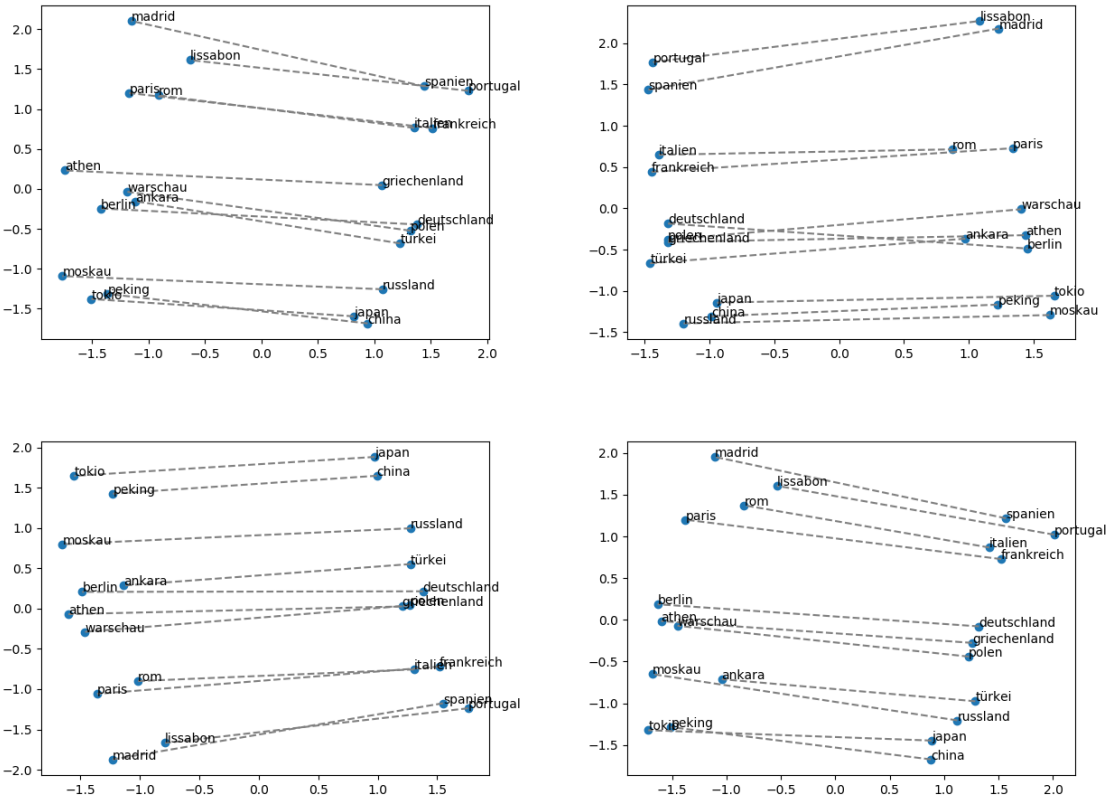

Figure 1: PCA of country and capital vectors of glint word vectors with 5 separate parameter servers and with 5 (left top) and 10 (right top), 12 (left bottom) executors with all 30 cores, except once 12 executors with 40 cores (right bottom)

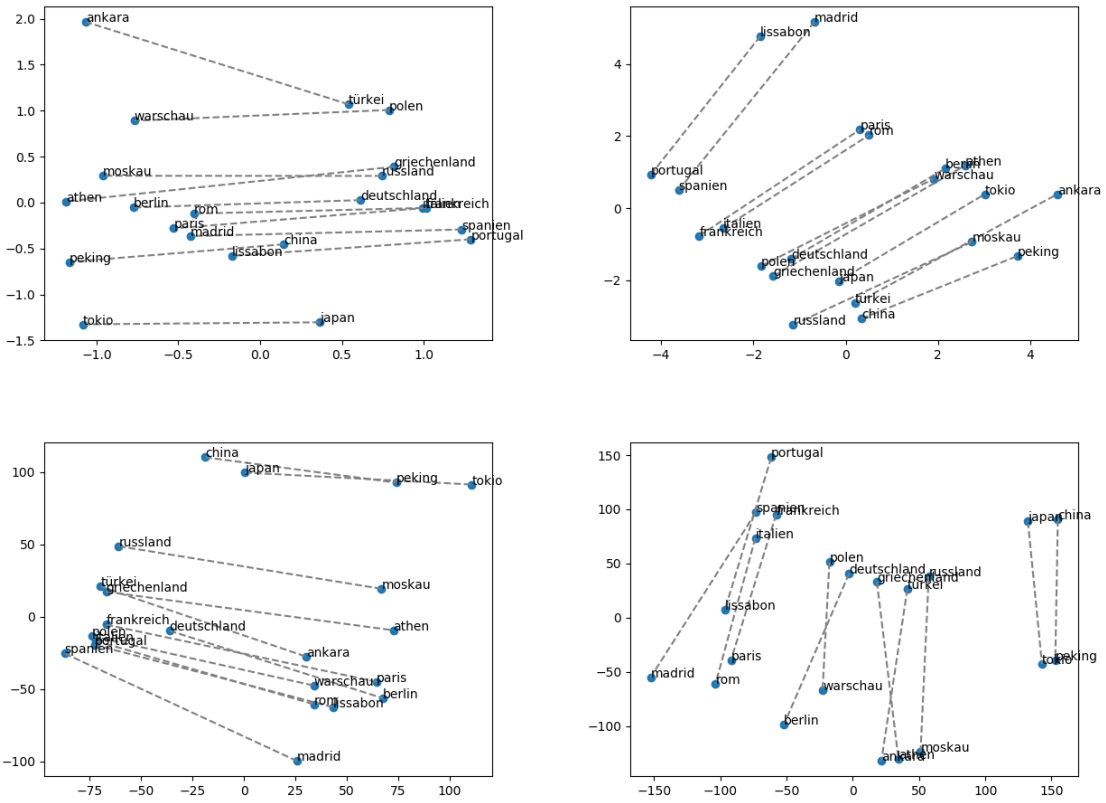

Figure 2: PCA of country and capital vectors of ml word vectors with 1 executor and 1 core (left top), 5 executors and 1 (right top) or 30 cores (left bottom) and 10 executors and 30 cores (right bottom)

The previous evaluations were made with a vector size of 100 and were primarily for determining the scaling behavior. My implementation is, however, mainly intended for large models which will also often have a larger vector size. I therefore also evaluated word vectors with larger vector sizes. The settings were the same as in the previous experiments unless specified otherwise. As driver and executor memory I used 50GB. Table 3 presents the findings. One can observe that Glint-word2vec can handle increasing vector sizes better than the default Spark implementation. Using a vector size of 1000 I ran into problems with the default Spark implementation with timeouts occurring and the application not finishing even after hours. I did not investigate these issues further and only listed the training time up to the point were problems occurred. A final interesting point to mention is the very long evaluation time of the default implementation. This is probably again because of the broadcasting of the now even larger word vectors.

type	executors	cores	vec size	train time	eval time	analogies	SimLex	WordSim
ml	5	30	500	40 mins	33 mins	2/11	0.271	0.528
ml	5	30	1000	80+ mins				
glint tcp	12 + 5	40 / 30	500	24 mins	3 mins	3/11	0.269	0.539
glint tcp	12 + 5	40 / 30	1000	32 mins	3 mins	1/11	0.264	0.526

Table 3: Evaluation of Word2Vec implementations with different vector sizes

4.2 Combined english data sets with phrases

As a final step, I evaluated my implementation on a large combination of english data sets created by the script `demo-train-big-model-v1-compute-only.sh` from [1] which was also used to evaluate the original distributed word2vec implementation. This script fetches a couple of data sets and adds often co-occurring words as phrases to the final data set. The resulting data set has a vocabulary of 8.7 million words and consists of more than 9 billion words. For this data set it was again necessary to remove stop words. Table 4 shows the 60 most frequent words, with custom stop words not occurring in the default stop word list marked. Besides the words in the table about 20 further stop words among the most frequent words had to be removed.

word	frequency	word	frequency	word	frequency	word	frequency
the	489,807,128	for	73,834,375	or	29,761,732	we	16,928,676
,	453,086,609	s	62,381,639	an	28,202,869	who	15,537,805
.	445,696,625	on	59,557,762	i	27,227,756	all	15,356,559
	357,354,700	as	55,557,227	have	26,913,723	also	15,099,185
of	246,086,709	was	54,477,701	not	24,707,307	had	14,843,339
and	212,386,936	it	51,338,805	his	23,763,915	can	14,453,875
to	194,985,549	”	49,306,774	but	22,500,334	**utc**	13,845,655
in	179,909,057	with	48,375,201	which	20,757,222	there	13,771,405
a	165,269,920	by	44,438,208	they	20,071,542	–	13,417,299
-	155,602,535	be	38,919,020	you	19,839,344	if	13,277,274
’	135,487,956	this	36,225,828	will	18,346,649	would	12,992,798
)	94,604,971	are	35,520,683	were	17,648,970	its	12,816,245
(	93,812,296	from	34,611,309	has	17,545,976	other	12,692,067
is	82,200,733	at	34,279,500	one	17,447,824	said	12,669,800
that	73,913,697	he	32,546,278	their	17,259,533	when	12,026,410

Table 4: Frequencies of the top 60 words in the demo-big dump. Gray words were added to a custom stop word list, bold words were accepted and all other words were already in the default english stop word list of Spark [24]

For evaluation I used mostly the same word2vec parameters as Ordentlich et al. [2] in their original paper. This means a vector size of 500, a learning rate of 0.01875, a subsample ratio of 1e-6, 5 negative examples and a mini-batch size of 50. I only performed one instead of three iterations and set the window size to 5 instead of 10 since my implementation has to have stop words removed beforehand and a first try with a window size of 10 yielded very bad scores. The SimLex coverage was 992 of 999 and the WordSim coverage was 353 of 353. Table 5 shows the evaluation scores. The attained WordSim score of 0.696 is close to the scores of 0.69 and 0.67 reported in the original paper.

type	executors	cores	train time	eval time	analogies	SimLex	WordSim
glint tcp	12 + 5	30	648 mins	10 mins	3/11	0.41	0.696

Table 5: Evaluation of my word2vec implementation on a large combination of english data sets

5 Conclusion and future work

To sum up, an existing approach for network-efficient word2vec training of large vocabularies was implemented as Spark ML package that also supports transforming words to vectors using parameter servers. In an evaluation on the german wikipedia the implementation surprisingly yielded better scores than both the default Spark word2vec implementation and a single-machine word2vec implementation. The training takes considerably longer but scaled without a decrease in word vector quality. Another evaluation similar to the one performed for the existing approach yielded similar results.
Regarding future work, there are a couple of things which may be analysed. First, garbage collection still takes up a considerable time despite measures to prevent excessive garbage collection. On the german wikipedia for a combined executor task runtime of 14.7 hours garbage collection takes up between 57 minutes and 2,4 hours on executors with parameter servers running while other executor only have 1.6 to 2.1 minutes of garbage collection. Further, the throughput of the parameter servers as well as the scaling behavior with considerably more executors could be analysed to discover potential scalability limits or assure that the fixed amount of parameter server executors won't become a bottleneck. Finally, the fault tolerance of this implementation is still untested and could benefit from some experiments and potential adjustments.

References

[1] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado and Jeff Dean. Distributed Representations of Words and Phrases and their Compositionality. *In NIPS 2013, Pages 3111-3119, source code at* `https://code.google.com/p/word2vec/`

[2] Erik Ordentlich, Lee Yang, Andy Feng, Peter Cnudde, Mihajlo Grbovic, Nemanja Djuric, Vladan Radosavljevic and Gavin Owens. Network-Efficient Distributed Word2vec Training System for Large Vocabularies. *In CIKM, 2016, Pages 1139-1148*

[3] `https://github.com/apache/spark/blob/master/mllib/src/main/scala/org/apache/spark/mllib/feature/Word2Vec.scala`

[4] `https://github.com/apache/spark/blob/master/sql/core/src/main/scala/org/apache/spark/sql/execution/exchange/BroadcastExchangeExec.scala#L101`

[5] `https://github.com/dmlc/ps-lite`

[6] https://github.com/Microsoft/Multiverso

[7] Rolf Jagerman, Carsten Eickhoff and Maarten de Rijke. Computing Web-scale Topic Models using an Asynchronous Parameter Server. *In ACM SIGIR, 2017, Pages 1337-1340, source code at* https://github.com/rjagerman/glint

[8] https://akka.io/

[9] http://www.dmtk.io/word2vec.html*, source code at* https://github.com/Microsoft/multiverso/tree/master/Applications/WordEmbedding

[10] https://github.com/real-logic/aeron

[11] https://databricks.com/session/continuous-integration-for-spark-apps

[12] https://travis-ci.org/

[13] https://spark.apache.org/docs/latest/tuning.html

[14] http://java-performance.info/memory-consumption-of-java-data-types-2/

[15] https://databricks.com/session/scaling-machine-learning-to-billions-of-parameters

[16] Feng Niu, Benjamin Recht, Christopher Re and Stephen J. Wright. HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. *In NIPS, 2011, Pages 693-701*

[17] Ira Leviant and Roi Reichart. Separated by an Un-common Language: Towards Judgment Language Informed Vector Space Modeling. *In arXiv preprint arXiv:1508.00106*

[18] Lev Finkelstein, Evgeniy Gabrilovich, Yossi Matias, Ehud Rivlin, Zach Solan, Gadi Wolfman, and Eytan Ruppin. Placing Search in Context: The Concept Revisited. *In ACM Transactions on Information Systems, 2002, Pages 116-131*

[19] Felix Hill, Roi Reichart and Anna Korhonen. SimLex-999: Evaluating Semantic Models with (Genuine) Similarity Estimation. *In Computational Linguistics, 2015, Pages 665-695*

[20] Anand Srinivasan, Ajay Jain and Parnian Barekatain. An Analysis of the Delayed Gradients Problem in Asynchronous SGD. https://openreview.net/forum?id=BJLSGcywG*, 2018*

[21] https://github.com/apache/spark/blob/master/mllib/src/main/resources/org/apache/spark/ml/feature/stopwords/german.txt

[22] RadimRehurek and Petr Sojka. Software framework for topic modelling with large corpora. *LREC Workshop on New Challenges for NLP Frameworks, 2010, Pages 46–50, project website at* https://radimrehurek.com/gensim/

[23] https://deeplearning4j.org/docs/latest/deeplearning4j-scaleout-intro

[24] https://github.com/apache/spark/blob/master/mllib/src/main/resources/org/apache/spark/ml/feature/stopwords/english.txt