Word Embeddings as Matrix Factorisation

May 13, 2020

Introduction

In this project I will examine the popular word embedding model skip-gram with negative sampling (SGNS)[3] by comparing different implementations, from the original shallow Neural Network (NN) approach to two more recent matrix-factorisation based methods. I will detail the similarities and differences of each method, before comparing performance and speed on an example corpus.

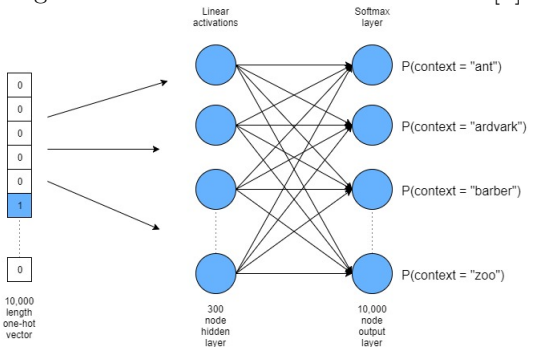
1 Background

1.1 Word Embeddings

Word embeddings describe a group of methods used to extract dense low-dimensional vector representations for individual words. These embedded vectors can be used to represent the meaning of a word; words that are located close to each other in the embedded vector space typically have similar roles in the original corpus and thus have similar meanings. Methods for finding effective word embeddings make up some of the most active areas of research in modern natural language processing.

In 2013 Google patented *word2vec*[2], a neural word embedding (NWE) that has since become the most widely used word embedding tool for NLP researchers. The model consists of a shallow, two layer network that maps a corpus of text to a vector space with embedded vector representations for each unique word in the corpus by extracting the second hidden, fully connected layer inside the network. The method uses one of two different architectures: continuous bag of words or continuous skip-gram. In this project I will focus on the skip-gram model architecture. I will briefly discuss the traditional shallow NN approach, *word2vec*, before comparing it to two different matrix-factorisation based approaches.

Figure 1: word2vec Network Architecture[**?**]



1.2 Skip-Gram with Negative Sampling

Skip-gram models consider a corpus of words, $w \in V_w$, and their contexts, $c \in V_c$, such that for any given word, w_i, its context is the set of words contained in the L-sized surrounding it, $\{w_{i-L}, \cdots, w_{i+L}\}/\{w_i\}$ (NB V_w and V_c are virtually identical but may differ slightly due to windows near the edge of a sentence). We call this window an n-gram: in this project I consider pentagrams, i.e. the case where L=2, as is common in the literature. A skip-gram model aims to predict words likely to be in the context of a given word.

Let us consider the word-embedding matrix, $W \in \mathbb{R}^{|V_w| \times d}$, whose rows, W_i, are the embedded vectors for word w_i, and equivalently the matrix $C \in \mathbb{R}^{|V_c| \times d}$ whose rows, C_i, are the embedded vectors for context c_i. The objective of the skip-gram model is to identify the likelihood of a word-context pair, (w_i, c_j), occurring given our data, i.e. the training corpus. Mathematically we can model the probability of (w_i, c_j) co-occurring as:

$$\sigma(\overrightarrow{w_i} \cdot \overrightarrow{c_j}) \quad = \quad \frac{1}{1 + e^{-\overrightarrow{w_i} \cdot \overrightarrow{c_i}}} \tag{1}$$

where $\overrightarrow{w_i}$ and $\overrightarrow{c_j}$ are the true embedded vectors. Thus the objective of skip-gram is to find approximations, $W_i \in W$ and $C_j \in C$, such that this probability is maximised for positive samples. In *word2vec* this constitutes an architecture consisting of one-hot encoded inputs, length V_w, connected to a dense hidden layer with weights $W \in \mathbb{R}^{|V_w| \times d}$, connected to a dense output layer with softmax, length V_c. This is illustrated in figure 1. Training this architecture is very computationally strenuous: the softmax activation requires calculating the product of the hidden layer output and the final layer weights for every output node, which, given V_c is usually many tens of thousands of words, can results in millions of weight updates at each training step.

One proposed solution is to use negative samples. For a given word w_i, k negative samples are drawn from the empirical unigram distribution, $\{c_{-1}, ..., c_{-k};\ c_{-j} \sim U_{V_c}\}$. The model aims to maximise their joint *negative* log likelihood:

$$\sum_{j=1}^{k} \mathbb{E}_{U_{V_c}}[\log \sigma(-\overrightarrow{w_i} \cdot \overrightarrow{c_{-j}})] \tag{2}$$

Combining (1) and (2) gives us the objective function for the SGNS model:

$$l(W_i, C_j) \quad = \quad \log \sigma(W_i \cdot C_i) \quad + \quad \sum_{j=1}^{k} \log \sigma(-W_i \cdot C_{-j}) \tag{3}$$

This ultimately involves updating only the weights of the negative sample, reducing computation by several orders of magnitude. For example, in an embedding with $d = 300$, $V_c = 100,000$ and $k = 15$, there are $(15 + 1) \times 300 = 4,800$ updated weights per step with negative sampling, compared to $100,000 \times 300 = 30,000,000$ without. Using negative sampling means that whilst observed word-context pairs have similar embeddings, unobserved pairs are scattered. I will briefly introduce another important NLP concept before moving on to discuss methods for solving the SGNS model.

1.3 Pointwise Mutual Information

Pointwise Mutual Information (PMI) is a pairwise similarity metric between two discrete outcomes, x and y, measuring the log ratio of the joint probability of x and y occurring and their marginal probabilities:

$$PMI(x, y) = \log \frac{P(x, y)}{P(x)P(y)}$$

In this project I will work with the PMI matrix, $M \in \mathbb{R}^{|V_w| \times |V_c|}$, whose entries $M_{i,j}$ are the PMI for a word-context pair, $PMI(w_i, c_j)$. We can compute an empirical estimate of this quantity using the observed co-occurrences in the training corpus:

$$PMI(w_i, c_j) = \log \frac{\#(w_i, c_j) \cdot N}{\#(w_i) \cdot \#(c_j)} \tag{4}$$

where $\#(w_i), \#(c_j), \#(w_i, c_j)$ refer to the number of independent occurrences of w_i and c_j and their co-occurrences respectively, and N to the number word-context pairs in the corpus. There are issues that arise from its application to NLP tasks however. For any unobserved word-context pair, where $\#(w_i, c_j) = 0$, the PMI is equal to $-\infty$. The matrix is also dense with potentially large dimensions $|V_w| \times |V_c|$.

Adaptations to overcome these issues include the shifted-PMI matrix (SPMI) and the shifted-Positive PMI matrix (SPPMI), whose entries are defined as:

$$SPMI_k(w_i, c_j) = PMI(w_i, c_c) - \log k \tag{5}$$

$$SPPMI_k(w_i, c_j) = \max(PMI(w_i, c_c) - \log k, 0) \tag{6}$$

where k is a constant. This SSPMI metric was introduced by Levy & Goldberg[1], and is used as the basis for their SGNS matrix-factorisation solution.

2 Methods

In this section I will give an overview of the three SGNS solvers covered, explaining the similarities and differences.

2.1 word2vec

As mentioned above, word2vec is the most widely used NWE implementation and is available through many different software packages, including PySparkML. The method is most often described as a shallow NN (though some researchers have labelled this as misleading[1]), and attempts to maximise the likelihood (3) by using online SGD updates over the observed word-context pairs D:

$$l = \sum_{(w_i,c_j)\in D} \#(w_i, c_j) \left(\log \sigma(W_i \cdot C_j) + k \cdot \mathbb{E}_{U_{V_c}}[\log \sigma(-W_i \cdot C_j)]\right) \tag{7}$$

to obtain W_i and C_j, estimates of $\overrightarrow{w_i}$ and $\overrightarrow{c_j}$ respectively. This method is known to be reasonably fast and can scale to large corpora, and requires training over every single observed word-context pair and thus is linear in N.

2.2 Explicit Factorisation via Singular Value Decomposition

Levy & Goldberg[1] showed that SGNS is in fact an implicit matrix factorisation. The embedded vectors, $\overrightarrow{w_i}$ and $\overrightarrow{c_j}$, make up word and context matrices, W and C, that are to be factorised from some matrix M. They found M to be the SPMI_k matrix with k corresponding to the number of negative samples used. The authors were not able to derive the appropriate loss function that allows exact reconstruction of the SGNS solution and instead approximate it by performing SVD on the the SPPMI to obtain estimates for W and C:

$$M^{SPPMI_k} \;=\; U\Sigma V^T \;= U\sqrt{\Sigma}\sqrt{\Sigma}V^T \;=\; WC \tag{8}$$

Levy & Goldberg found this to yield an approximation close to the optimal solution in the vector space, and had comparable performance on various NLP tasks. This method requires only the PMI matrix and thus can be used with count aggregations of a corpus (it does not require training over every word-context pair like the word2vec solution), and is thus sub-linear in N if provided precomputed counts. I will implement this approach as an SVD using PySpark MLlib.

2.3 Implicit Factorisation via Logistic Loss Minimisation

Later work by Kenyon-Dean[10] shows that it is possible to obtain an exact solution to the original SGNS problem using matrix factorisation. The author argues that using SPPMI has little theoretical justification and does in fact disregard much of the data, and, that by using a different loss function to the Frobenius norm used in SVD, one can exactly frame the SGNS problem as SPMI factorisation. By accumulating all the repeated terms in the object function one can rewrite it as:

$$l \;=\; \sum_{(w_i,c_j)\in D} \left(\#(w_i, c_j) \log \sigma(W_i \cdot C_j) + \frac{k \cdot \#(w_i) \cdot \#(c_j)}{N} \log \sigma(-W_i \cdot C_j)\right) \tag{9}$$

This function uses the logistic sigmoid and thus naturally attenuates for $-\infty$, we are thus able to use the SPMI without issue. As with the SVD implementation, this method can be run using only corpus aggregated counts and can be sub-linear in N. In practice this procedure takes a very different form to the SVD approach as we don't directly use the PMI matrix but an algebraic decomposition of it. I will implement this approach in TensorFlow.

3 Implementations

Preparation of the corpus and aggregated counts

For this project I used a partial wikipedia dump containing text from a random collection of articles taken in April 2020. The raw XML files were converted to text files using *wikiextractor*[17] before being stored in Hadoop FS. Two corpora of different sizes were prepared, the first a smaller subset containing V_w=V_c=31,986 unique words and the second containing 75,854 unique words.

The data was tokenized and converted to numeric codes, before all word-context pairs were extracted from the sentence pentagrams, resulting in N=136 billion word-context pairs in the smaller corpus and 108 trillion in the larger (amounting to 2.3Gb when stored as plain text). To convert the words to unique numeric codes, a simple ordered in-memory list was used. If the corpus was of a size that prevented the vocabulary being held in-memory one could use a distributed index-word mapping, though this would have to be a very large vocabulary (32-bit python can hold lists of up to 530,000,000 entries). The resulting word-context RDD is the input to the *word2vec* model.

In order to compute the two matrix factorisation approaches, the data must first be aggregated by count. This is done using spark DataFrame operations, where word-context pairs were grouped to find counts before the marginal word/context counts were found by summing over windows partitioned by word/context.

Finally, the implicit matrix factorisation is implemented using a block computation approach. It operates over batches of word-context-count triplets grouped by word-count such that the members of a batch represent the entries of a single block of the matrix M. A word-context-count triplet RDD is prepared such that each partition represents a matrix block. This RDD is written to CSV and stored in the google filesystem. The marginalised counts are also written and stored as separate single column ordered CSV files.

3.1 word2vec

Implementations of *word2vec* are widely available, with different versions available in a number of software packages. There are several fundamental challenges for distributed word embedding training systems to overcome; using gradient descent means most implementations require storage of embedding matrices on all executors. These matrices can easily grow to unmanageable sizes with large vocabularies, requiring in excess of hundreds of gigabytes storage for large applications[5]. Recent work has proposed a new distributed training framework leveraging state of the art distributed graph analytics systems[4], achieving significant improvements in computation time.

PySpark MLlib's *word2vec* implementation is not based on negative sampling, rather hierarchical softmax[12]. Hierarchical softmax reduces complexity from O(V_w) to O(log(V_w)). The methods can provide quite different results and training times, though for the purposes of demonstration this hierarchical softmax approach will be used. If required, and care is taken to sample correctly, an implementation of SGNS through TensorFlow and Keras is possible, though not explored in this project.

3.2 SVD

Levy and Goldberg's SVD approach is implemented using PySpark MLlib's matrix operations. First, an RDD of SPPMI values is calculated using DataFrame operations, before being converted to a distributed CoordinateMatrix and then to a distributed RowMatrix. Whilst the CoordinateMatrix is a better representation given the sparsity of our SPPMI matrix, RowMatrix is currently the only distributed matrix in the PySpark toolbox that supports SVD. If one were able to implement a factorisation approach that makes use of sparsity there might be significant computational performance gains.

Spark's implementation for SVD of an approximately square matrix roughly works as follows: "we compute M^TM in a distributive way and send it to ARPACK to compute M^TMs top eigenvalues and eigenvectors on the driver node. This requires O(k) passes, O(n) storage on each executor, and O(nk) storage on the driver." ARPACK[16] is a linear algebra library for large scale eigen-decompositions. It is worth noting that, whilst there is a parallel ARPACK package, the Spark implementation is *not* fully distributed and thus there is scope for computational performance gains. ARPACK is optimised for sparse matrices.

Once the factorised matrices are returned, the word embedding is extracted by calculating $W = U\sqrt{\Sigma}$ using RDD row transformations.

3.3 Implicit Factorisation

As the implicit factorisation approach is essentially an optimisation problem, TensorFlow is used in the implementation. For small corpora, this method can best be set up as finding W and C whose product M minimises the logistic, SGNS loss with the aggregated count matrix. This method does not scale well to large corpora however as it requires storing very large matrices W and C, and their potentially massive product, M, leading to out of memory errors. In order to alleviate this strain and allow the potential for parallelisation, a block computational model is introduced whereby the loss (9) is decomposed into the summation of the loss for each block:

$$l = \sum_{B_k \in \{B\}} \sum_{(w_i, c_j) \in B_k} \left(\#(w_i, c_j) \log \sigma(W_i \cdot C_j) + \frac{k \cdot \#(w_i) \cdot \#(c_j)}{N} \log \sigma(-W_i \cdot C_j) \right) \tag{10}$$

In this computational model, W and C are optimised block by block. By taking n^2 blocks, one needs only work with the relevant $1/n$ rows $W_{\{B_k\}} \in W$ and $C_{\{B_k\}} \in C$ at any point, and can calculate the loss of its product $W_{\{B_k\}} \cdot C_{\{B_k\}} = M_{B_k}$ against the corresponding word-context pairs, $(w_i, c_j) \in \{B_k\}$. This means one needs only store M_{B_k} and $(w_i, c_j) \in \{B_k\}$ in memory at any given time.

Distributing this approach through TensorFlow is non-trivial. In an ideal scenario, TensorFlow's distributed parameter-server computation model would be applied over the cluster through a distributed dataset using *MultiWorkerMirroredStrategy*. This is difficult to implement however, as TensorFlow requires constant length batches and, even though the blocks are already split across files, thus makes it hard to keep blocks separate in training. One workaround to this issue would be to pad the block entries such that batch is the same length. This adds a fair amount of computation and is not explored here. Instead, we will revert to serial computation.

It is also important to consider how best to train the blockwise model. In this implementation an epoch is defined as the process of training over a specified number of iterations for each block sequentially. The model is then trained for a specified number of epochs. This approach balances reasonably frequent alternation between blocks without changing the data stored on the driver node too often. Once training is complete, the embedding matrix W is extracted.

4 Results

The algorithms were run on a Google Cloud Dataproc cluster consisting of a 6 n1-standard-4 (4 vCPUs, 15 GB memory) nodes (1 master and 5 workers). When necessary, data was stored in Hadoop FS and Google FS. Due to restrictive computational expense results are only reported for the smaller corpus.

Computation times were recorded for the three approaches on the smaller corpus, displayed in table 1. Both factorisation approaches operate over aggregated statistics so the time taken to compute these was also recorded, though let the numbers in the table serve as a lower bound (perhaps by up to around a few minutes) as a some processing operations were not executed until saving. The computation time for the explicit matrix factorisation was recorded. As the implicit approach is a numerical optimisation, the computation time is dependent on the number of training iterations and epochs. The time displayed in table 1 is the time for 100 iterations per block, per epoch for 50 epochs for a model consisting of 64 blocks computed serially. Figure 2 shows that the algorithm loss converged after around 10 epochs, albeit to a seemingly erroneous level, and thus we might optimistically hope for around 4-5x faster training time for the implicit loss in the future (even before parallelisation is applied).

Figure 2: Training Loss for Implicit Matrix Factorisation

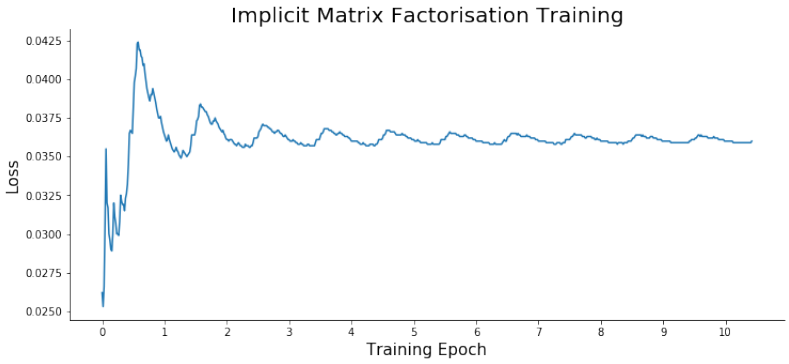


Table 1: Computation Times

		word2vec	explicit	implicit
Data	aggregate pairs	-	691.4	
	write to HDFS	-	15,661.4*	
	calculate SPPMI	-	935.9	-
	convert to matrix	-	2485.0	-
Training		2302.4	1307.9	35,129.0
TOTAL		2302.4	5420.2	35,820.4

*not included in total

On the smaller corpus, *word2vec* is the fastest. This is in large part due to the time spent calculating aggregate statistics for the factorisation methods, a process which could probably be optimised further. It is also worth noting that aggregate statistics are a "one-off" cost, in the sense that once computed for a corpus, many factorisation models can be trained using them. In a larger corpus we would expect to see the computational advantages of matrix factorisation manifest themselves more clearly: as mentioned previously, *word2vec* is linear in N whereas the factorisation methods are sublinear. With a well implemented blockwise algorithm we might expect the implicit factorisation to solve fastest on large corpora.

Table 2: Word Similarities

	word2vec	**explicit**	**implicit**
football	baseball rugby stadium	stadium beach baseball	happy sandal lemon
seven	eight nine week	cafe nine spain	computer paint walk
king	queen prince lord	court royal prince	center district maple
manchester	london leeds arsenal	london north europe	recent declaration address

For a selection of test words, the three vectors in each embedding matrix with the highest cosine similarity are returned (see table 2). In terms of accuracy of implementation, the explicit matrix factorisation provides a reasonable approximation to the *word2vec* word similarities. The implicit factorisation minimisation converges to a relatively high loss (figure 2), likely due to inaccuracies in the implementation, and the resulting word comparisons are nonsensical. If a more detailed evaluation of the word embeddings was desired, there are a number of established word similarity linguistic challenges, for example *WordSim353*[13] and *MEN*[14]. Theoretically we would expected *word2vec* and the implicit approach to tend to the same embedding.

5 Conclusion

In conclusion, the explicit matrix factorisation has proved to be a reasonable alternative to Spark's NN implementation for our small corpus, giving reasonable results on word similarities and computing in a similar time. It would be interesting to compare speed over a larger corpus and, with a more robust implementation, the blockwise implicit approach would quite possibly outperform the others over massive corpora, especially if an efficient aggregation procedure was discovered. It is worth noting there is a significant research aimed at improving the training procedure for NWEs and that Spark's implementation of *word2vec* is far from state of the art. A full investigation into the comparative speed of a state of the art distributed NN approach versus a well implemented implicit factorisation would prove quite interesting.

References

[1] Yoav Goldberg and Omer Levy. Neural Word Embeddings as Implicit Matrix Factorization. *Advances in Neural Information Processing Systems 27 (NIPS)*, 2014.

[2] Mikolov et al. Efficient Estimation of Word Representations in Vector Space. *https://arxiv.org/abs/1301.3781*, 2013.

[3] Mikolov et al. Distributed Representations of Words and Phrases and their Compositionality. *Advances in Neural Information Processing Systems 26 (NIPS)*, 2013.

[4] Gill, Dathathri, Maleki, Musuvathi, Mytkowicz, Saarikivi "Distributed Training of Embeddings using Graph Analytics", *https://arxiv.org/abs/1909.03359 [cs.LG]*, 2019.

[5] Ordentlich, Yang, Feng, Cnudde, Grbovic, Djuric, Radosavljevic, Owens "Network-Efficient Distributed Word2vec Training System for Large Vocabularies", *arXiv:1606.08495 [cs.CL]*, 2016.

[6] Gittens et al. Matrix Factorizations at Scale: a Comparison of Scientific Data Analytics in Spark and C+MPI Using Three Case Studies. *arXiv:1607.01335v3 [cs.DC]*, 2016.

[7] Bhavanaa, Kumarb, Padmanabhan Block based Singular Value Decomposition approach to matrix factorization for recommender systems. *arXiv:1907.07410v1 [cs.LG]*, 2019.

[8] Zadeh et al. Matrix Computations and Optimization in Apache Spark. *arXiv:1509.02256v3 [cs.DC]*, 2016.

[9] Kenyon-Dean. Word Embedding Algorithms as Generalized Low Rank Models and their Canonical Form. *https://arxiv.org/abs/1911.02639*, 2019.

[10] Kenyon-Dean. Unifying Word Embeddings and Matrix Factorisation. *Blog series on Medium.com*, 2019.

[11] Yoav Goldberg and Omer Levy. word2vec explained: deriving Mikolov et al.s negative-sampling word- embedding method. *arXiv preprint arXiv:1402.3722*, 2014.

[12] Mnih, Hinton. A Scalable Hierarchical Distributed Language Model *Advances in Neural Information Processing Systems 21 (NIPS 2008)*, 2008.

[13] Finkelstein, Gabrilovich, Matias, Rivlin, Solan, Wolfman, & Ruppin. "Placing search in context: The concept revisited", *ACM TOIS*, 2002.

[14] Bruni, Boleda, Baroni, & Khanh Tran, "Distributional semantics in technicolor", *ACL*, 2012.

[15] Keras word2vec tutorial *https://adventuresinmachinelearning.com/word2vec-keras-tutorial/*

[16] https://www.caam.rice.edu/software/ARPACK/

[17] wikiextractor. *https://github.com/attardi/wikiextractor*