Interspeech 2018
2-6 September 2018, Hyderabad, India

Structured Word Embedding for Low Memory Neural Network Language Model

Kaiyu Shi and Kai Yu

Key Lab of Shanghai Education Commission for Intelligent Interaction and Cognitive Engineering
SpeechLab, Department of Computer Science and Engineering
Shanghai Jiao Tong University, Shanghai, China
skyisno.1@gmail.com, kai.yu@sjtu.edu.cn

Abstract

Neural network language model (NN LM), such as long short term memory (LSTM) LM, has been increasingly popular due to its promising performance. However, the model size of an uncompressed NN LM is still too large to be used in embedded or portable devices. The dominant part of memory consumption of NN LM is the word embedding matrix. Directly compressing the word embedding matrix usually leads to performance degradation. In this paper, a product quantization based structured embedding approach is proposed to significantly reduce memory consumption of word embeddings without hurting LM performance. Here, each word embedding vector is cut into partial embedding vectors which are then quantized separately. Word embedding matrix can then be represented by an index vector and a code-book tensor of the quantized partial embedding vectors. Experiments show that the proposed approach can achieve 10 to 20 times embedding parameter reduction rate with negligible performance loss.

Index Terms: Language Model, word embedding, vector quantization, product quantization, model compression

1. Introduction

In automatic speech recognition (ASR) , language model (LM) is the core component that incorporates syntactic and semantic constraints of a given language. Although the conventional N-gram back-off language model with smoothing has been widely used in ASR, it suffers from limited context length and the huge memory requirements for large vocabulary. Recently, neural network based language model (NN LM) [1] has attracted great interest due to its effective encoding of word context history and memory efficiency. In neural network based language models, both the word context and the target word are projected into a continuous space. The projection, represented by the transformation matrix, is learned during training. The projected continuous word vectors are also referred to as *word embeddings*. With the effective word context encoding, feed-forward neural network language model (FNNLM) achieves better PPL and word error rate (WER) for ASR. Following FNNLM, recurrent neural network (RNN) and long-short term memory [2] (LSTM) LM are proposed to handle long context history in sentences. They have achieved state-of-the-art results on various datasets [3, 4, 5, 6].

The corresponding author is Kai Yu. This work has been supported by the National Key Research and Development Program of China under Grant No.2017YFB1002102, the China NSFC projects (No. 61573241) and the Shanghai International Science and Technology Cooperation Fund (No. 16550720300). Experiments have been carried out on the PI supercomputer at Shanghai Jiao Tong University.

NN LMs, including FNNLM, RNNLM and other variants, share the same embedding map from words to continuous vector space. Such approach requires a large number of parameters for word embeddings. This is unfavored in many scenarios. First, memory consumption becomes a major concern when NN LM are deployed in resource restricted systems [7]. Second, as each word is assigned a unique embedding vector, NN LM are unlikely to learn meaningful embeddings for infrequent words due to data sparsity [8]. Notably, [9] incorporates sub-word features into word embeddings in RNNLM and outperforms straightforward word embeddings, but the memory cost is increased for additional neural network structures.

Model compression for NN LM has attracted much research interest in recent years. There are two basic components to compress in NN LM, the recurrent layer and the word embeddings. In most cases, the majority of parameters in NN LM lies in word embeddings. [10] explores the independence of neurons in the recurrent layers of LSTM and achieves state-of-the-art results while obtaining $2.5\times$ compression rate in LSTM layer. Neither input nor output embeddings is studied in the paper, so the memory cost of the whole model remains large. LightRNN [11] addresses the problem by decomposing word embeddings into row embeddings and column embeddings. The embeddings are shared among fixed number of words, leading to huge memory reduction. However, significant performance degradation may be observed for relatively small vocabulary. In [8], the embeddings of infrequent words are represented with the embeddings of frequent words by a sparse linear combination. It solves both of the problems above, but the memory reduction rate is insignificant under tiny size vocabulary (20% when $|V| = 10K$). In addition, the architecture is complicated because it invokes an additional layer in the output layer.

In this paper, an effective product quantization based structured word embedding framework [12] is proposed to save NN LM memory. Word embedding is split into sub-embeddings, each of which is quantized and represented by code-book and indices. Therefore, each word *partially* shares embeddings with other words in both output and input word embeddings. The sharing relations are decided automatically by the syntactic and semantic similarities between words. Experiments show that significant memory reduction rates can be obtained without hurting NN LM performance.

The rest of the paper is organized as follows. In section 2, a detailed analysis of standard LSTM LM is given. In section 3, the proposed structured embedding is introduced and compared with other methods. Experimental results and analysis are presented in section 4. Finally, section 5 concludes the paper and discusses the future work.

10.21437/Interspeech.2018-1057

2. Memory Consumption of LSTM LM

In this section, the architecture of LSTM LM is reviewed. Then we discuss the memory problem in conventional LSTM LM.

2.1. LSTM LM Review

LSTM LM consists of three parts: *input embedding*, *LSTM encoder* and *output embedding*. In the following part of this section, $\boldsymbol{x_t}$ denotes the $\boldsymbol{x}$ at time t. $\boldsymbol{w}_n^\top$ denotes the transpose of n-th row of $\boldsymbol{W}$, where $\boldsymbol{W}$ is the weight matrix. V is the vocabulary and $|V|$ is the vocabulary size. e is the embedding dimension, and h is the hidden size of LSTM.

The input embedding is a lookup table denoted by $\boldsymbol{W}^{(\texttt{in})} \in \mathbb{R}^{|V| \times e}$ which maps word index m to word embedding $\boldsymbol{x}$.

$$\boldsymbol{x} = \boldsymbol{w}_m^{(\texttt{in})} \tag{1}$$

Concretely, one step of the LSTM takes $\boldsymbol{x}_t, \boldsymbol{h}_{t-1}, \boldsymbol{c}_{t-1}$ as inputs, and produces $\boldsymbol{h}_t, \boldsymbol{c}_t$. Computation details are omitted in this paper.

The output embedding is a projection layer denoted by $\boldsymbol{W}^{(\texttt{out})} \in \mathbb{R}^{|V| \times h}$, followed by a softmax operation. It converts the hidden state of LSTM $\boldsymbol{h}$ to the word probability distribution $\boldsymbol{P}$.

$$P_n = \frac{\exp(\boldsymbol{w}_n^{(\texttt{out})\top} \boldsymbol{h})}{\sum_{k=1}^{v} \exp(\boldsymbol{w}_k^{(\texttt{out})\top} \boldsymbol{h})} \tag{2}$$

where P_n is the probability of the n-th word.

Each row in $\boldsymbol{W}^{(\texttt{in})}$ or $\boldsymbol{W}^{(\texttt{out})}$ can be viewed as a continuous vector representation of the corresponding word, i.e. the *word embedding*.

LSTM LM can be trained using the *back propagation through time* (BPTT) algorithm [13]. Since the probability is normalized among V, the most computational cost is induced by propagating on $\boldsymbol{W}^{(\texttt{out})}$.

2.2. Memory Problem In LSTM LM

The memory consumption of LSTM LM has become a serious problem recently caused by the rapidly rising size of datasets. Ignoring the biases, the parameters in LSTM LM θ can be divided into two parts: parameters in the embeddings $\theta_e = \{\boldsymbol{W}^{(\texttt{in})}, \boldsymbol{W}^{(\texttt{out})}\}$ and parameters in the LSTM layer $\theta_{lstm} = \{\boldsymbol{W}_f, \boldsymbol{W}_i, \boldsymbol{W}_o, \boldsymbol{W}_c\}$. The total number of parameters can be easily computed given the corresponding vocabulary V, embedding size e and LSTM cell size h. Here we use $|\theta|$ to denote the total number of parameters in θ.

$$\begin{aligned} |\theta| &= |\theta_e| + |\theta_{lstm}| = |V|e + |V|h + 4h\,(h+e) \\ &= (|V| + 4h)\,(h+e) \end{aligned} \tag{3}$$

A widely used dataset in language model is OneBillionWord (OBW)[6] with $|V| = 793K$. To model such number of words, the embedding part will cost nearly 1.2GB memory on small embedding size like $e = h = 200$ on OBW, which is usually too big for portable devices to hold. At the same time, one LSTM layer has only $\sim 1M$ parameters with the same configuration.With larger vocabulary e.g. $|V| = 100M$ in ClueWeb09 [14], even GPUs or workstations can not hold such models in native memory.

The number of parameters increases linearly with respect to vocabulary size, embedding size and LSTM size. We will discuss the memory problem under the assumption that $|V| \geq 10,000$ and $h, e \leq 1,000$, which is reasonable in most large vocabulary language models. In fact, smaller e or h can greatly reduce the parameters as the coefficient $|V|$ is large enough. But the performance degrades sharply as e or h goes down, due to poor **representation ability**.

The major cause of memory problem in conventional embeddings is the lack of structure which exploits the similarity relations among words. The current embedding framework treats each word separately as a row in embedding matrix $\boldsymbol{W}$, thus all the embeddings are totally independent. Once a new word w is added into vocabulary, a complete row vector $\boldsymbol{x}_w$ will be appended to the embedding matrix $\boldsymbol{W}$.

Low-rank decomposition is usually used to reduce the parameters in matrix. A full rank matrix $\boldsymbol{W}$ is decomposed by two matrices $\boldsymbol{U}, \boldsymbol{V}$ with lower rank. The compression rate of this method is controllable by the rank value. But the performance will decrease sharply at high compression rate, indicating naive low-rank can not fully utilize the underlying structure in embeddings. Vector quantization can also be employed. It compresses vectors by exploiting the global structure of these points and has been successfully used in speech recognition[15][16], computer vision[17]. However, naive vector quantization method requires a global structure in high dimension space for good performance [18], which is seldom satisfied in real-world occasions.

There are also other promising methods proposed recently to exploit the similarity among words. They explicitly define the sharing principles for word embeddings, so these methods out-perform traditional ones in language model tasks.

LightRNN [11] assumes a word w can be represented by row embedding $\boldsymbol{x}_w^r$ and column embedding $\boldsymbol{x}_w^c$ rather than one single embedding $\boldsymbol{x}_w$. To allocate all the words into a square table, there exists another strong assumption in lightRNN that there are exactly $\sqrt{|V|}$ row and column embeddings and each row or column embedding is shared among exactly $\sqrt{|V|}$ words. Under these assumptions, lightRNN compresses $|V|$ embeddings into $2\sqrt{|V|}$ embeddings. The drawbacks of lightRNN lie under the second assumption, which is merely satisfied for relatively small vocabulary. As is shown later in table 4, though the compression rate is the highest, the performance is not acceptable in real applications. What's more, the compression rate of lightRNN is fixed given the vocabulary.

Paper [8] explores a different approach to structured embeddings. It's assumed that word can be represented by other words in vocabulary. This approach is an ad-hoc method because the threshold separating rare and frequent words are specified by intuition. [19, 20] utilize more complicated methods for compositional code learning, but these methods are only applied on input embedding.

Most importantly, [12] utilizes product quantization embeddings and achieves 8 times embedding parameters reduction on text classification. The major contribution of this paper is that we utilized a different configurations of product quantization and achieved higher compression ratio (20X) without any other compression methods.

3. Structured Embeddings with Product Quantization

The introduction of product structured embedding (PSE) aims to reduce the memory consumption of language model via sharing partial embeddings among similar words, with the assumption that one word shares different underlying properties with different words. In this structure, rather than rows in weight

matrix $\boldsymbol{W}$, word embeddings are composed of the partial embedding candidates from the compressed embedding structure.

As is described in section 2.2, naive VQ usually hurts the performance. To alleviate this problem, product quantization [21] explores the redundancy in vector space by decomposing the space into a Cartesian product of low dimensional subspaces and quantizing each subspace separately. Using product quantization, [18, 22] achieve high compression rate in CNN for image tasks with little performance degrade.

To fully utilize the partial similarity among words, we use product quantization [21] to compress the embeddings. As illustrated in Figure 1, product quantization consists of two basic steps: decomposing the embedding matrix into several subspaces (also called groups in this paper) and quantizing the vectors in each sub-space. Decomposing into sub-spaces ensure the representation ability, while quantization drastically reduces the parameters and memory cost.

In our model, we first train a vanilla language model with conventional embeddings. Then the input and output embedding matrices are individually compressed by product quantization(PQ), after which the whole model is fine-tuned or totally re-trained for optimal performance.

3.1. Compress via Product Quantization

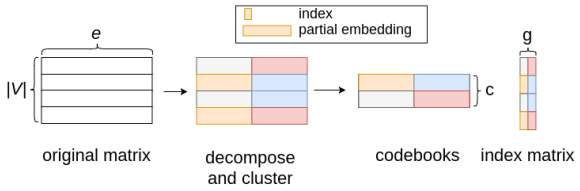

Figure 1: *A simple illustration of product quantization method, partial embeddings with the same color are compressed into one centroid*

At compressing phase, the input embedding and output embedding are compressed independently. The operations for two embeddings are identical, so we take input embedding for example in this section. Product quantization needs two hyper-parameters, the number of clusters c and the number of groups g. At compressing phase, product quantization is used to compress the matrix $\boldsymbol{W} \in \mathbb{R}^{|V|\times e}$ to the *index* matrix $\boldsymbol{Q} \in \mathbb{N}^{|V|\times g}$ and the *codebook* tensor $\boldsymbol{C} \in \mathbb{R}^{g\times c\times \frac{e}{g}}$, where e is the embedding size and $|V|$ is the vocabulary size. It should be noted that in our current work, the matrix is equally divided into g groups, so e is divisible by g, which is unnecessary in general.

In decomposition step, the original matrix $\boldsymbol{W}$ is simply chunked into g groups along the second dimension:

$$\boldsymbol{W} = \left[\boldsymbol{W}^1, \boldsymbol{W}^2, \cdots, \boldsymbol{W}^g\right] \tag{4}$$

Afterwards, the sub-matrix $\boldsymbol{W}^i$ is quantized one by one. In this paper the row vectors in sub-matrix $\boldsymbol{W}^i$ is clustered via K-means [23] with $K = c$. K-means algorithm uses centroids,namely *codebook*, to represent all the vectors. And it maintains a vectors-centroids mapping, namely *index*, indicating which centroid is closest to each vector.

So each sub-matrix $\boldsymbol{W}^i$ is compressed to an index vector $\boldsymbol{q}^i \in \mathbb{N}^{|V|}$ and a codebook $\boldsymbol{C}^i \in \mathbb{R}^{c\times \frac{e}{g}}$. Consequently the original matrix $\boldsymbol{W}$ is compressed to an index matrix $\boldsymbol{Q} = \left[\boldsymbol{q}^1, \boldsymbol{q}^2, \cdots, \boldsymbol{q}^g\right] \in \mathbb{N}^{|V|\times g}$ and the codebook tensor $\boldsymbol{C} = \left[\boldsymbol{C}^1, \boldsymbol{C}^2, \cdots, \boldsymbol{C}^g\right] \in \mathbb{R}^{g\times c\times \frac{e}{g}}$

The parameters in PSE, θ_{PSE}, consist of parameters in the codebooks $\theta_{\boldsymbol{C}}$ and the index matrix$\theta_{\boldsymbol{Q}}$. According to the matrix size, total number of parameters is simply $|\theta_{PSE}| = |\theta_{\boldsymbol{C}}| + |\theta_{\boldsymbol{Q}}| = ec + |V|g$. A typical setup is $|V| = 10K, e = 200, g = 8, c = 400$, thus the compression rate is computed by $\frac{\theta_e}{\theta_{PSE}} = \frac{e|V|}{ec+|V|g} = \frac{2000K}{80K+80K} = 12.5$. Note that the index is non-negative integer, so we can get even higher compression rate by using only needed bits.

3.2. Interpretation as Low-rank Decomposition

Following [24], we can regard product quantization as a special low-rank decomposition $\boldsymbol{W} = \boldsymbol{U}\boldsymbol{V}$ where the value of $\boldsymbol{U}$ is fixed and only $\boldsymbol{V}$ is updated during training. [24] argues that fixing $\boldsymbol{U}$ helps to reduce the redundancy in naive low-rank decomposition.

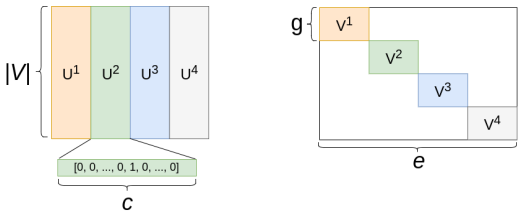

Figure 2: $\boldsymbol{U}$ *(left) and* $\boldsymbol{V}$*(right) in low-rank decomposition, different colors for different sub-spaces*

As is illustrated in Fig 2, rows in $\boldsymbol{U}$ is obtained from the rows in $\boldsymbol{Q}$, with $\boldsymbol{U}^k_{j,\boldsymbol{Q}^k_j} = 1$ for $1 \le j \le g$. $\boldsymbol{V}$ is a block diagonal matrix where the i-th block is the codebook $\boldsymbol{V}^i = \boldsymbol{C}^i$.

Furthermore, our model may be considered as adding an intermediate layer where one word is represented by multiple one-hot vectors rather than one single one-hot vectors.

4. Experiments

The PSE model is evaluated on both PTB and SWB datasets, containing 10K and 30K words in vocabulary respectively. LSTM LM is used in our experiments on both datasets. On PTB, we set the embedding size to 200 and use a 2-layer LSTM with hidden size of 200. On SWB, we build a larger model by increasing the embedding size to 300, followed by a single-layer LSTM with hidden size of 300. Models are trained and tested on GTX 1080, Intel Xeon E5-2620 V4, with CUDA-8.0 and Pytorch framework.

The embedding matrix is sequentially chunked into g chunks along the second dimension, followed by a simple K-means from sklearn[25] in terms of vector quantization. The k-means algorithm is initialized with "k-means++" [26] method and runs 10 times to get one best results.

4.1. Performance and Representation Ability

Our method is firstly investigated on PTB. We fix the hyper-parameters $g^{(in)} = g^{(out)} = 8, c^{(in)} = c^{(out)} = 400$ and test the performance of various initialization methods on PTB, the results are shown in Table 1. *pre-train (W)* indicates that a pre-trained embedding matrix is used to do product quantiza-

Table 1: *Performance of different initialization methods of product structured embeddings on PTB.*

Model	Initialization			PPL
	pre-train	tied	codebook	
Baseline	-	-	-	97
PSE-R	-	-	-	165
PSE-W	✓	-	-	103
PSE-WT	✓	✓	-	100
PSE-W+	✓	-	✓	102
PSE-WT+	✓	✓	✓	**98**

Table 2: *Grid search results of PSE-WT+ on PTB, the size should double if both input and output are counted.*

$g \backslash c$	200	400	1000	Index size
4	117	105	97	40K
8	104	98	95	80K
10	100	96	**94**	100K
Codebook size	40K	80K	200K	-

tion, otherwise a random initialized matrix will be used. Tied embeddings [27] is also used as a trick to improve embeddings quality, denoted as *tied (T)*. Note that the structured input and output embeddings in our model are not tied regardless of the initialization method. And we also explored the situation when we only know the similarities among words, that is, codebook $\boldsymbol{C}$ are randomly initialized while index $\boldsymbol{Q}$ is initialized via some prior knowledge (here simply pre-trained embeddings), this is denoted as *codebook (+)*.

The model initialized randomly (PSE-R) lacks prior knowledge of word similarity. The partial embeddings are shared among randomly picked words, leading to poor performance (165) after compressing. Meanwhile, PSE-W utilizes such prior by clustering the pre-trained embedding weight matrix and obtains acceptable performance (103). We also tried to do product quantization based on the tied embeddings. It gives us the best results (98) when combined with codebook initialization (PSE-WT+). It concludes that tying weights produces better embeddings for word similarities in LSTM LM, which is consistency with [27]. On the best performing model PSE-WT+, we achieve $12.5\times$ parameters reduction in embeddings with almost no performance loss.

The effect of different PQ configurations on g and c is also explored. As shown in table 2, the PPL decreases as g and c increase. Larger g helps to discovery local similarities and c helps to distinguish different properties. But when these numbers are big enough, the performance gains are not obvious while index size and codebook size grow linearly. For $g = 10, c = 1000$, We get even better PPL (94) than baseline (97) while still achieving 6.7 $\times$ compression in embeddings. It proves that sharing partial embeddings does not hurt the representation ability of word embeddings.

Table 3: *Performance on SWB*

Model	PPL	WER	#Parameters	
			Embeddings	Overall
Baseline	**53.2**	**20.1**	18M	18.7M
PSE-W	55.4	20.2	**0.9M**	**1.6M**

Table 4: *Comparison of various embeddings compression methods on PTB.*

Compression method	PPL	Compression rate	
		Embeddings	Overall
Baseline	97	1	1
Naive low-rank	112	9.8	4.5
Vector quantization[1]	130	12.5	4.8
LightRNN [2]	223	**40**	**6.2**
PSE-WT+	**98**	12.5	4.8
sPSE-WT+[3]	206	**40**	**6.2**

We have also tested our method on switchboard rescoring task. The language model is trained on Fisher dataset, vocabulary consists of words with more than 3 occurrences, about 30K words in total. By experience, we set $c^{(in)} = c^{(out)} = 1000$ and $g^{(in)} = 4, g^{(out)} = 6$. In this task, we only initialized the index matrix $\boldsymbol{Q}$ with pre-trained LSTM LM baseline (PSE-W). As is shown in table 3, the proposed model gives nearly the same WER (20.2%) as baseline (20.1%) while achieving $20\times$ compression rate in the embeddings, $11.7\times$ compression rate in the whole model.

4.2. Comparison with Other Methods

Performances of different compression methods are compared in table 4. In order to get similar compression rate, the rank is set to 20 in naive low-rank decomposition, and the number of clusters is set to 400 in Vector quantization.

LightRNN has the highest compression rate but worst performance. Moreover, the compression rate of lightRNN is fixed. To compare with lightRNN we build a smaller model named sPSE-WT+ with equal compression rate. The sPSE-WT+ has even lower PPL than lightRNN. Compared with naive low-rank and vector quantization, the proposed model PSE-WT+ achieves the best PPL at minimal memory cost because it utilizes deeper structure in word embeddings.

5. Conclusion and Future Work

This paper proposes structured word embedding framework based on product quantization to reduce memory consumption of NN LM. Both input and output embeddings are replaced by product structured embeddings. The performance gap between original model and compressed model is negligible in terms of both PPL and WER, while compressed model has $10\times \sim 20\times$ fewer parameters in input and output embeddings. Future work will focus on the training efficiency of the proposed approach as well as combination with other compression techniques.

[1] The vector quantization is equal to the PSE-WT+ when $g = 1$.

[2] We follow the example configurations on `https://github.com/Microsoft/CNTK/tree/master/Examples/Text/LightRNN`, except for that the embedding size and hidden size are both set to 200

[3] $g = 4, c = 50$

6. References

[1] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," *Journal of machine learning research*, vol. 3, no. Feb, pp. 1137–1155, 2003.

[2] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[3] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur, "Recurrent neural network based language model." in *Interspeech*, vol. 2, 2010, p. 3.

[4] M. Sundermeyer, I. Oparin, J.-L. Gauvain, B. Freiberg, R. Schlüter, and H. Ney, "Comparison of feedforward and recurrent neural network language models," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013, pp. 8430–8434.

[5] M. Sundermeyer, R. Schlüter, and H. Ney, "Lstm neural networks for language modeling," in *Thirteenth Annual Conference of the International Speech Communication Association*, 2012.

[6] C. Chelba, T. Mikolov, M. Schuster, Q. Ge, T. Brants, P. Koehn, and T. Robinson, "One billion word benchmark for measuring progress in statistical language modeling," *arXiv preprint arXiv:1312.3005*, 2013.

[7] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[8] Y. Chen, L. Mou, Y. Xu, G. Li, and Z. Jin, "Compressing neural language models by sparse word representations," *arXiv preprint arXiv:1610.03950*, 2016.

[9] T. He, X. Xiang, Y. Qian, and K. Yu, "Recurrent neural network language model with structured word embeddings for speech recognition," in *In proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing(ICASSP)*, Brisbane, Australia, April 2015, pp. 5396–5400.

[10] O. Kuchaiev and B. Ginsburg, "Factorization tricks for lstm networks," *arXiv preprint arXiv:1703.10722*, 2017.

[11] X. Li, T. Qin, J. Yang, X. Hu, and T. Liu, "Lightrnn: Memory and computation-efficient recurrent neural networks," in *Advances in Neural Information Processing Systems*, 2016, pp. 4385–4393.

[12] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, and T. Mikolov, "Fasttext. zip: Compressing text classification models," *arXiv preprint arXiv:1612.03651*, 2016.

[13] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[14] J. Pomikálek, M. Jakubícek, and P. Rychlỳ, "Building a 70 billion word corpus of english from clueweb." in *LREC*, 2012, pp. 502–506.

[15] L. Bahl, P. De Souza, P. Gopalakrishnan, and M. Picheny, "Context dependent vector quantization for continuous speech recognition," in *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on*, vol. 2. IEEE, 1993, pp. 632–635.

[16] P. Renevey, R. Vetter, and J. Krauss, "Robust speech recognition using missing feature theory and vector quantization." in *INTERSPEECH*, 2001, pp. 1107–1110.

[17] N. M. Nasrabadi and Y. Feng, "Vector quantization of images based upon the kohonen self-organizing feature maps," in *Proc. IEEE Int. Conf. Neural Networks*, vol. 1, 1988, pp. 101–105.

[18] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing deep convolutional networks using vector quantization," *arXiv preprint arXiv:1412.6115*, 2014.

[19] R. Shu and H. Nakayama, "Compressing word embeddings via deep compositional code learning," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=BJRZzFlRb

[20] T. Chen, M. R. Min, and Y. Sun, "Learning k-way d-dimensional discrete code for compact embedding representations," *arXiv preprint arXiv:1711.03067*, 2017.

[21] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2011.

[22] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[23] S. Lloyd, "Least squares quantization in pcm," *IEEE transactions on information theory*, vol. 28, no. 2, pp. 129–137, 1982.

[24] M. Denil, B. Shakibi, L. Dinh, N. de Freitas *et al.*, "Predicting parameters in deep learning," in *Advances in Neural Information Processing Systems*, 2013, pp. 2148–2156.

[25] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[26] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035.

[27] O. Press and L. Wolf, "Using the output embedding to improve language models," *arXiv preprint arXiv:1608.05859*, 2016.