Adaptive Compression of Word Embeddings

Yeachan Kim[1] **Kang-Min Kim**[2] **SangKeun Lee**[2,3]
[1]Artificial Intelligence Research Institute, Republic of Korea
[2]Department of Computer Science and Engineering [3]Department of Artificial Intelligence
Korea University, Seoul, Republic of Korea
yeachan@airi.kr, {kangmin89, yalphy}@korea.ac.kr

Abstract

Distributed representations of words have been an indispensable component for natural language processing (NLP) tasks. However, the large memory footprint of word embeddings makes it challenging to deploy NLP models to memory-constrained devices (e.g., self-driving cars, mobile devices). In this paper, we propose a novel method to adaptively compress word embeddings. We fundamentally follow a code-book approach that represents words as discrete codes such as (8, 5, 2, 4). However, unlike prior works that assign the same length of codes to all words, we adaptively assign different lengths of codes to each word by learning downstream tasks. The proposed method works in two steps. First, each word directly learns to select its code length in an end-to-end manner by applying the Gumbel-softmax tricks. After selecting the code length, each word learns discrete codes through a neural network with a binary constraint. To showcase the general applicability of the proposed method, we evaluate the performance on four different downstream tasks. Comprehensive evaluation results clearly show that our method is effective and makes the highly compressed word embeddings without hurting the task accuracy. Moreover, we show that our model assigns word to each code-book by considering the significance of tasks.

1 Introduction

Deep neural networks have greatly improved the performance in various tasks, such as image classification (Huang et al., 2017), text classification (Liu and Lapata, 2018), and machine translation (Edunov et al., 2018). This break-through performance facilitates the demand to deploy such models to embedded systems (e.g., self-driving cars, mobile devices). However, the neural models typically require a large storage or memory footprint, which is a significant concern when deploying neural models to memory-constrained devices (Hinton et al., 2015). To alleviate this limitation, several works have proposed methods that compress the neural models while minimizing loss of accuracy as much as possible (Han et al., 2015, 2016; Liu and Zhu, 2018).

However, deploying models for natural language processing (NLP) tasks is challenging. Unlike other domains, NLP models have an embedding layer which maps words and phrases to real-valued vectors. The problem is that these embeddings usually take more parameters than the remaining networks. In practice, for a neural translation model in OpenNMT (Klein et al., 2017), the word embedding parameters accout for 80% of the total parameters. Therefore, it is significant to reduce the parameters of the embedding layer for deploying NLP models to memory-constrained devices.

To compress word embeddings, several works proposed code-book based approaches (Shu and Nakayama, 2018; Tissier et al., 2019), which represent each word as few discrete and shared codes. For example, the word *dog* and *dogs* could be represented as (3, 5, 2, 1) and (3, 5, 2, 7), respectively. This sharing scheme and discrete codes make the embeddings have smaller parameters and interpretability to some extent. However, these methods assign the same length of codes to each word without considering the significance of downstream tasks. It means that, for a sentiment analysis, *excellent* and *the* require the same amount of memory. This observation makes room for improvement in compressing word embeddings.

In this paper, we attempt to further compress word embeddings by adaptively assigning different lengths of codes to each word in an end-to-end manner. We propose *AdaComp* that adaptively learns to compress word embeddings by considering downstream tasks. The proposed compression works

Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pages 3950–3959
July 5 - 10, 2020. ©2020 Association for Computational Linguistics

in two steps. First, each word in pre-trained word embeddings learns to select its code length in an end-to-end manner by applying Gumbel-softmax tricks (Jang et al., 2016). After selecting its code length, each word learns discrete codes through a binary-constraint encoder and decoder network. To instill task-specific features to the selection process, we compress each word embedding by learning a downstream task. This allows us to learn the task-specific features naturally. Compared to prior works, *AdaComp* could give each word more options to represent their meaning since the proposed model utilizes a number of different code-books.

To showcase the general applicability of AdaComp, we conduct four different NLP tasks, which are sentiment classification, chunking, natural language inference, and language modeling. Comprehensive evaluation results not only show that our method could compress original word embeddings quite well without hurting task accuracy but also demonstrate that AdaComp assigns each word to different code-books by considering the significance of a task. *AdaComp* could be applied to most existing NLP systems with minor modifications since the proposed model is a network-agnostic, in-place architecture. We thus believe that existing NLP systems could benefit from our work.

We organize the remainder of this paper as follows. In Section 2, we discuss related work. In Section 3, we describe the proposed method. We report our performance evaluation results and analyze our methodology in detail in Section 4 and 5, respectively. Finally, we conclude this paper in Section 6.

2 Related Work

In this section, we review several studies that attempt to compress neural models, including an embedding layer.

2.1 Neural Networks Compression

The majority of works for compression is to compress neural networks itself (e.g., convolutional neural network, recurrent neural network), and most of them focus on compressing neural models in the field of computer vision. These approaches usually include pruning, quantization, and low precision representation methods. For pruning, several works (Han et al., 2015; Li et al., 2017; Lee et al., 2019) focus on how each connection (i.e., weights) affects to tasks, and they remove redundant or unimportant connections from the networks. Some works (Han et al., 2016; Chen et al., 2016; Louizos et al., 2019) quantize the connections into several bins to enforce weight sharing. These approaches represent each connection as some representative values, and such values are selected by clustering (centroids) or hashing (hash buckets) techniques. Representing each connection with low precision (i.e., few bits or binary) is also appealing for compressing neural networks (Anwar et al., 2015; Courbariaux et al., 2015; Hubara et al., 2016). In particular, Courbariaux et al. (2015) and Hubara et al. (2016) show that binary constraint is sufficiently effective in network learning without largely affecting the task accuracy.

2.2 Word Embeddings Compression

Several studies have proposed compressing methods for word embeddings because the majority of parameters in NLP models lies in an embedding layer. For example, Ling et al. (2016) reduces the memory requirement of word embeddings by quantizing each dimension of embeddings into significantly fewer bits than the standard 64 bits. It shows that 4 or 8 bit is enough to represent each word embedding. Instead of reducing the parameters of each word embedding, Chen et al. (2016) reduces the number of words in vocabulary by filtering out uncommon words. For the removed words, they reconstruct these embeddings by combining several frequent words. Recently, several methods (Shu and Nakayama, 2018; Shi and Yu, 2018; Tissier et al., 2019) decompose each word into a few numbers of codes and learn corresponding code vectors to represent the original embeddings. Shu and Nakayama (2018) uses a deep code-book approach to represent each word. To automatically learn discrete codes, they utilize reparameterization tricks in an encoder and decoder architecture. Similarly, Tissier et al. (2019) utilizes an auto-encoder with a binary constraint to represent words. Compared to the aforementioned methods, AdaComp is the first work that represents each word differently in terms of length of codes. Furthermore, we learn task-specific features directly by learning a downstream task at the same time.

3 Adaptive Compression

In this section, we describe the proposed method, which is denoted as *AdaComp*, in detail. The primary strategy of AdaComp is straightforward and

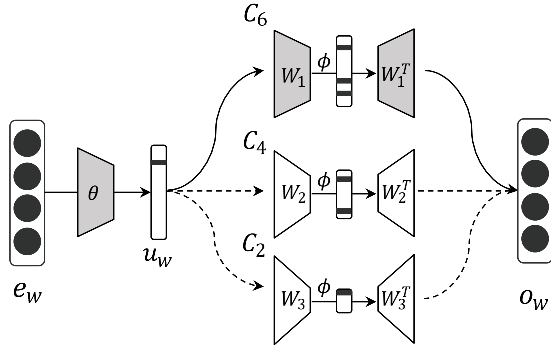


Figure 1: Main strategy of our compression model (AdaComp). Solid line indicates the selected code-book.

is shown in Figure 1. We start with the pre-trained word embeddings (e.g., GloVe (Pennington et al., 2014), word2vec (Mikolov et al., 2013)), and the compression method works in two steps. Given an input embedding, AdaComp learns to adaptively select its code length in an end-to-end manner by applying Gumbel-softmax tricks (Jang et al., 2016) (Section 3.1). After selecting a code length, each word learns its discrete codes through an encoder and decoder, which has a binary latent space (Section 3.2).

3.1 Adaptive Code-book Selection

To represent each word as discrete codes, several code-book approaches build a single code-book C_k where *k* is the length of codes. Instead of assigning the same length of codes, we adaptively assign different lengths of codes to each word. To this end, we have a set of different code-books $C = \{C_{k_1}, C_{k_2}, ..., C_{k_n}\}$. The objective for the first phase is to select a single code-book from the set of code-books in an end-to-end manner.

Given an input embedding e_w, we first compute an encoding vector α_w by feeding it to neural networks.

$$\alpha_w = \sigma_1(\theta^T \sigma_2(\theta^{'T} e_w + b^{'}) + b) \tag{1}$$

where $\theta \in \mathbb{R}^{d \times |C|}$, $\theta^{'} \in \mathbb{R}^{d \times d}$ and b, $b^{'}$ are trainable weight matrices and biases of the networks, respectively, where d is the dimension of the original embeddings. The functions $\sigma_1(\cdot)$, $\sigma_2(\cdot)$ are the *softplus* and *tanh* function, respectively. Then, we could select a single code-book by applying an *argmax* or a *sign* function into the resultant encoding. However, deriving discrete values (i.e., the index of the code-books) in the neural networks is not trivial since the aforementioned functions are not differentiable.

To handle such problem, several methods proposed to deal with discrete values in a neural network naturally. In our work, we use the Gumbel softmax tricks since we need a one-hot vector to represent the discrete index of the set of code-books. The Gumbel softmax allows the neural networks to naturally have a k-dimensional one-hot vector in the intermediate of the networks. Let u_w be the one-hot vector for a word w, the i-th element of the vector is computed as follows:

$$\begin{aligned} u_w^i &= softmax_\tau(\log \alpha_w^i + g_i)) \\ &= \frac{exp((\log \alpha_w^i + g_i)/\tau)}{\sum_{j=1}^{|C|} exp((\log \alpha_w^j + g_j)/\tau)} \end{aligned} \tag{2}$$

where $g_i, ..., g_{|C|}$ are i.i.d noise samples drawn from Gumbel distribution[1] and τ is the relaxation factor of the Gumbel softmax. Similarly, (Shu and Nakayama, 2018) utilized Gumbel softmax for compression. However, they used it to derive discrete codes of each word, not the index of the set of code-books as in AdaComp.

3.2 Binarized Codes Learning

After selecting a specific code-book from the set C, AdaComp learns the discrete codes in the selected code-book. To this end, we use a binary constraint encoder and decoder, which has a binary latent space. When the training converges, the binary latent vector of each word is used as the discrete code, and the decoder is used as the code vectors in each code-book.

Again, we start from the original word embedding. To produce discrete codes, we feed the embeddings to the binary constraint networks. Let w be the word in an input text and n be the code length of the selected code-book, and the code learning works as follows:

$$e_w^{'} = W\phi(W^T e_w + b) + b^{'} \tag{3}$$

where $W \in \mathbb{R}^{d \times n}$ and b, $b^{'}$ are trainable weight matrices and biases in the encoder and decoder, respectively. As can be seen from the equation, we use the same weights at the encoding and decoding phase. This is because such tied weights enable

[1]Gumbel distribution can be sampled using inverse transform sampling by drawing $u \sim$ Uniform[0, 1] and computing $g = -\log(-\log(u))$

faster training and have a greater regularization effect than individual weights (Alain and Bengio, 2014; Gulordava et al., 2018). The function ϕ is the binary constraint function. We use the following threshold function[2]:

$$\phi(x_i) = ReLU(Sign(x_i)) = \begin{cases} +1 & x_i \geq 0, \\ 0 & \text{otherwise,} \end{cases}$$

This function produces the binary codes, which consist of 1 and 0. However, we face the same problem with the previous section. The derivative of the *sign* function is zero almost everywhere, making it incompatible with back-propagation. To naturally learn the binary codes in an end-to-end manner, we apply the straight-through estimator (Hinton, 2013) to the threshold function. This estimator allows gradients to skip the threshold function. In our work, we use a different version of the straight-through estimator to take into account a saturation effect. Let the gradients above the threshold function as $\frac{\partial L}{\partial N}$, we obtain gradients of the threshold function as follows:

$$\frac{\partial L}{\partial \phi} = \frac{\partial L}{\partial N} 1_{|g| \leq 1} \tag{4}$$

where g is the value of the gradients above the threshold function. This function allows us to naturally learn binary codes by preserving the information of the gradients and canceling the gradient when g is too large, which could mess up the training.

Thus far, we adaptively select the code-book from the set, which has a different length of code-books, and produce the binary codes of each word. To jointly learn the above two phases in an end-to-end manner, we relate them as follows:

$$o_w = E_w^T u_w \tag{5}$$

where $E_w^T \in \mathbb{R}^{|C| \times d}$ is the reconstructed embeddings of w for all code lengths. By multiplying the selection vector (i.e., u_w) by the reconstructed embeddings, AdaComp learns two phases in an end-to-end manner. We feed the reconstructed embedding o_w to task-specific networks for learning a downstream task.

[2]We also experimented with only applying sign function which results in -1 and +1. We empirically found that the two functions produced nearly identical results. We thus use ReLU with Sign function for a decoding efficiency.

3.3 Orthogonality Constraint

To cover a large number of words in the vocabulary, reducing the redundancy of representations for each code vector is significant. We thus put the orthogonality constraint into code vectors, which penalizes redundant latent representations and encourages each code vector to encode different aspects of the original word embeddings.

$$P = \|W^T W - I\|_F^2 \tag{6}$$

where W is the parameters of the code vectors (i.e., decoder), I is an identity matrix. $\|\cdot\|_F$ stands for Frobenius norm of a matrix. We add this term to our objective function.

3.4 Optimization

Since AdaComp learns compression by learning a downstream task, the objective function depends on each task. For example, if the task is sentiment classification, the objective function could be negative log-likelihood over sentiments. Let the objective function be L_{task}, the total objective function is as follows:

$$L = L_{task} + \lambda \cdot P \tag{7}$$

where λ is the control factor of orthogonality, and we set this to 0.01.

We empirically found that pretraining AdaComp significantly increases the performance for several tasks (detailed in Section 5.1). We thus pretrain our model using an auto-encoder loss, which is as follows:

$$L_{pre} = \sum_{w \in V} \|o_w - e_w\|_2^2 \tag{8}$$

When the loss of pretraining converges, we attach the pre-trained AdaComp to an embedding layer of task-specific networks and learn a downstream task using Eq.7.

4 Experiments

In this section, we show the performance evaluation of the proposed model. To showcase the general applicability of AdaComp, we conduct four different tasks, which are sentence classification, chunking, natural language inference, and language model. Through the above tasks, we validate the efficacy of AdaComp on the settings of many-to-one (sentiment classification), many-to-many (chunking, language modeling), and multiple inputs (natural language inference).

Task	Vocabulary size	Memory size (MB)
SST-5	17,080	20.5
CoNLL-2000	19,072	22.9
SNLI	34,045	40.9
PTB	9,809	11.8

Table 1: Memory size of the original word embeddings (i.e., GloVe) and the number of words in each task. Each embedding is 300 dimensional vectors and is represented by 32 bit floating point.

Experimental settings

The proposed compressing model starts from pre-trained word embeddings. In the experiments, we use the publicly available GloVe[3] (Pennington et al., 2014) with 300 dimension for 400k words. For hyper-parameter settings, we use Adam (Kingma and Ba, 2014) optimizer with 0.001 learning rate and the batch size is 64. We choose the above parameters by validating both sentiment classification and natural language inference tasks.

Model Comparison

In this paper, we examine the following methods which use different kind of compressing methods:

- ***QWE*** (Ling et al., 2016): This model quantizes the weights from floating-point to few bits of precision less than standard 64 bits. We evaluate two settings, which are 4 and 8-bit representations.

- ***Pruning*** (Han et al., 2015): This model prunes redundant weights from the networks. We prune the weights of word embeddings until this technique removes 80% or 90% neurons from the embeddings.

- ***NC*** (Shu and Nakayama, 2018): This model compresses the pre-trained embeddings using a single code-book using a deep neural network. We compare two different settings, which are the moderate size (16x16 codebook) and the large size (32x16 code-book).

- ***Bin*** (Tissier et al., 2019): This model compresses word embeddings through an auto-encoder which has binary constraint on a latent space. Among their two methods, we choose **rec** since it performs better with deep neural networks (i.e., LSTM, CNN). We compare two settings that have 64 and 128 binary codes.

- ***AdaComp*** (Ours): This is the proposed model in this paper. We use four different code-books since we found that using four code-books leads to the most effective performance with a memory requirement (detailed in Section 5.2). We use three different settings on the four code-books which have (**128**, 64, 32, 16), (**64**, 32, 16, 8) and (**32**, 16, 8, 4) length of code-books. On the tables and figures, we use the max length of codes to denote each model.

The aforementioned methods do not learn task-specific features since they learn to compress embeddings using the auto-encoder loss. To fairly compare with our method, we apply the strategy in (Shu and Nakayama, 2018) to each model. In short, we first fine-tune the original embeddings to tasks and then compress the learned embeddings through the above methods.[4]

Evaluation metrics

We report both a task performance and a total memory size. The total memory size is estimated from all parameters which are used to represent all words in tasks. Note that it does not contain the size of task-specific networks. For our method, we report memory size and performance when we deploy our model to other systems. It contains the parameters of multiple code-books and binary codes about each word. The memory size of the original embeddings about each task is listed in Table 1.

4.1 Experimental Results

Table 2 shows the overall results on four tasks. We describe each task and the task-specific networks as below.

Sentiment classification

Sentence classification is the task of classifying a sentence into pre-defined classes. We use the stanford sentiment treebank (SST) dataset as a representative dataset. The SST has 5 classes about sentiment (very negative, negative, neutral, positive, very positive). The performance is measured

[3]https://nlp.stanford.edu/projects/glove/

[4]We have also applied an end-to-end compression learning to each model. However, we confirmed that this training was only significant in AdaComp and, for the other methods, produced nearly identical results with the strategy in (Shu and Nakayama, 2018).

Model	SST-5		CoNLL-2000		SNLI		PTB	
	accuracy	ratio	F1	ratio	accuracy	ratio	ppl	ratio
GloVe	42.1	x1	93.1	x1	79	x1	100.3	x1
QWE (4-bit) (Ling et al., 2016)	41.8	x8	93.1	x8	77.9	x8	113.8	x8
QWE (8-bit) (Ling et al., 2016)	41.9	x4	93.3	x4	78.6	x4	109.1	x4
Pruning 90% (Han et al., 2015)	35.4	x10	90.4	x10	78	x10	113.2	x10
Pruning 80% (Han et al., 2015)	41.6	x5	91.7	x5	78.2	x5	124.7	x5
NC (16×16) (Shu and Nakayama, 2018)	37.2	x46	91.8	x50	77.8	x71	119.2	x30
NC (32×16) (Shu and Nakayama, 2018)	40.9	x23	92.4	x25	78.5	x35	112.4	x15
Bin (64) (Tissier et al., 2019)	36.8	x95	91.5	x100	77.3	x116	116	x74
Bin(128) (Tissier et al., 2019)	39.1	x48	92.7	x49	77.6	x59	110.1	x37
AdaComp (32)	42.0	x171	92.1	x173	77.6	x232	110.8	x105
AdaComp (64)	43.2	x84	93	x89	78.4	x119	106	x52
AdaComp (128)	42.9	x45	93.1	x44	78.7	x60	108.9	x26

Table 2: Comparison results on four tasks. The ratio on the table is calculated by dividing the size of the original embeddings by that of comparison models. Higher performance means better model except for PTB (perplexity).

by the accuracy on test set. For text classification model, we reproduce the LSTM model used in (Zhang et al., 2015) as a baseline. It feeds word embeddings in sequence, and averages hidden states of the last layer to represent an input sentence for classification. In this model, we set the hidden states to 450 dimension and use two-stacked LSTMs.

As can be seen from the table, code-book approaches (i.e., NC, Bin, AdaComp) basically show better results than others in both performance and memory size. Among them, AdaComp makes more highly compressed embeddings than others with better performance. For example, *AdaComp (32)* achieve as much as 11% improvement on test accuracy compared to other code-book approaches which use the same number of codes with the longest codes in ours. Furthermore, our model requires nearly 2x less memory sizes compared to others.

Chunking

Chunking is the task of dividing a sentence into syntactically correlated parts of words. The CoNLL 2000 shared task (Tjong Kim Sang and Buchholz, 2000) is a benchmark dataset for text chunking. It has 24 tags about each word with its start and end symbols. The performance is measured by F1 score. For the chunking model, we use an LSTM-based sequence tagger which was proposed by (Huang et al., 2015). We set the hidden states to 300 dimensions and use two-stacked LSTMs.

The results are shown in the same table. Even though the quantization method (i.e., QWE 8-bit) achieves the best performance when they restrict the values into 8-bits, the compression ratio is quite lower than other methods, and the performance starts to degrade as they use smaller bits to represent words. Compared to the other code-book methods, AdaComp achieves strong performance with highly compressed embeddings. For example, AdaComp (128) does not hurt the accuracy of the original embeddings with approximately 44x compressed embeddings.

Textual entailment

Textual entailment is the task of determining whether a hypothesis is true, given a premise. The Stanford Natural Language Inference (SNLI) (Bowman et al., 2015) dataset is a benchmark for this task. This dataset contains approximately 550K hypothesis/premise pairs with entailment, contradiction, and neutral labels. For this task, we use an LSTM-based encoder model which was proposed by (Bowman et al., 2016). It uses two different LSTMs with 300-dimensional hidden states to encode each information (i.e., premise and hypothesis). The concatenated vectors for two sentences are classified into the three labels.

Even though the performance of our method, including others, is lower than the elementary embeddings, AdaComp yields strong performance with a high compression ratio in this task. Compared to other methods that use the largest memory, the proposed model (i.e., AdaComp (128)) requires the

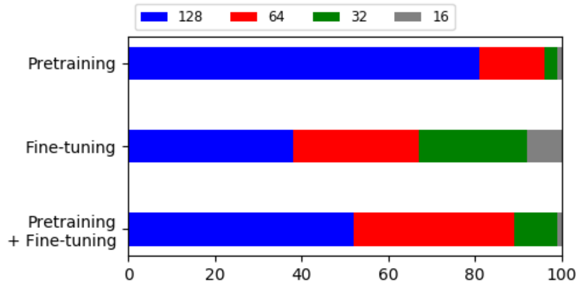


Figure 2: Ratio of code-book assignment on each setting. Best viewed in color.

least memory size while resulting in the closest performance with the original embeddings.

Language modeling

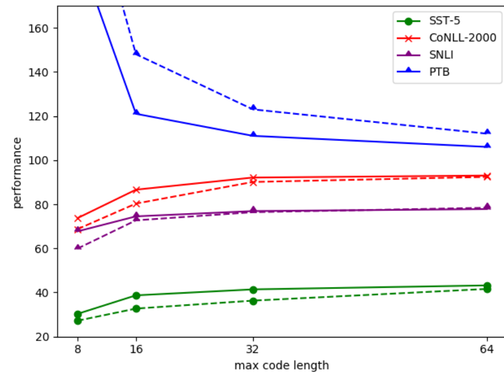
Language modeling is the task of scoring a sentence whether it is natural or not comparing to the training dataset. This task has been widely used in several mobile applications by recommending the next word or sentence based on a user text. In this task, we use Penn Treebank (PTB) to evaluate the performance. We report test perplexity about each method. For this task, we use a word-based LSTM model which was used in (Kim et al., 2016). We select a medium-size model with 650-dimensional hidden states to encode each word and apply dropout (Srivastava et al., 2014) to the top of LSTMs.

Similar to the previous task, the performance of the methods is lower than the original embeddings. We conjecture the lower performance comes from that these tasks (i.e., language modeling, natural language inference) require more generalized features than other tasks. This is why these tasks are used to pretrain neural models for various NLP tasks (Cer et al., 2018; Radford et al.). Compared to others, again, AdaComp achieves the best results in terms of both metrics.

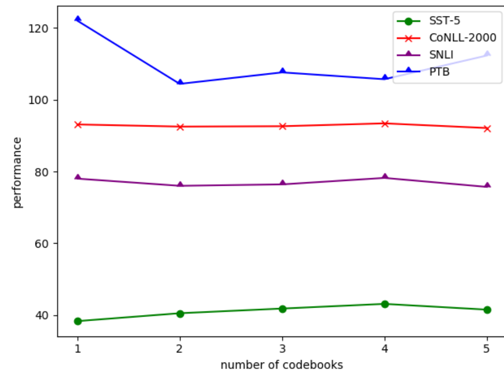
5 Analysis

5.1 Utility of pre-training and Fine-tuning AdaComp

Before AdaComp learns to compress word embeddings, we pretrain the model using the auto-encoder loss (Eq. 8). To show that pretraining step is indeed effective, we report accuracy and a ratio of code-book assignment. Here, we evaluate the performance of all tasks when we use different set of code-books (detailed in Section 5.3). Figure 3a shows the performance results. The result shows



(a)



(b)

Figure 3: Performance variation when we use different settings (the length and the number of code-books) on each task. Dashed line indicates the model which is not pretrained. Performance (y-axis) indicates evaluation metrics for each task. Note that the evaluation metric for language modeling is perplexity, thus, the performance is reversed on PTB. Best viewed in color.

that the model with pretraining performs better than the model, which is not pretrained. This is clearly evident when we use smaller code-books to represent words. We believe that the pretraining step guides our model towards basins of attraction of minima that support a better generalization. This is the similar results with (Erhan et al., 2010).

Figure 2 shows the comparison of the code-book assignment on each setting for the SNLI task. When we only pretrain the compressing model, the large portion of words, around 80%, is assigned to the largest code-book (i.e., 128). However, when we fine-tune the pre-trained models to the task, the ratio of the large one is significantly decreased. This means that fine-tuning could reduce the memory requirement by a large margin. Without the pre-training step, fine-tuning model achieves a smaller memory size than the pre-trained models. However, we have shown that pretraining leads to better

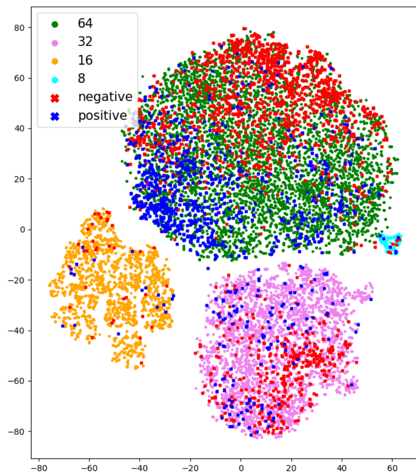


Figure 4: Visualization of the reconstructed embeddings with their code-book assignment. Best viewed in color.

performance. To achieve a reasonable memory size with reliable performance, we have applied both pretraining and finetuning to AdaComp.

5.2 Effectiveness of the length or number of code-books

We evaluate the performance of our method when we use different lengths or numbers of code-books. We first plot the results of different lengths of code-books in Figure 3a. Here, we use four code-books as default and the length of codes is divided by two along with the next smaller code-book. For example, the value *64* in the axis means (64, 32, 16, 8) and *32* means (32, 16, 8, 4).

As you can see in Figure 3a, utilizing the large size of code-books leads to improved performance than the models with smaller lengths. These results come from that larger code-books could represent more aspects of original embeddings. Figure 3b shows the performance variation of different number of code-books. Here, we use 128 code vectors and divide these vectors into several code-books. The x-axis means the number of code-books that correspond to (128), (64, 64), (64, 32, 32), (64, 32, 16, 16), (64, 32, 16, 8, 8). We observe that the performance does not depend on different code-books very much compared to lengths of code-books. To get better performance with high compression ratio, we have used four code-books in the experiments.

5.3 Code-book distribution for a task

Unlike other methods, AdaComp learns to compress embeddings with learning a downstream task. To confirm how the model assigns each word to different code-books, we visualize the code-book assignment. To this end, we project the reconstructed embeddings into 2-dimensional space using t-SNE (Maaten and Hinton, 2008), and we use the embeddings when we perform the sentiment classification task using AdaComp (64). To show important words (i.e., sentiment words) to the task, we take the sentiment words (positive and negative) from (Hu and Liu, 2004) and denote these words if they existed in the embeddings of AdaComp.

Figure 4 shows the 2-dimensional projection of the reconstructed embeddings with their assigned code-books. We observe that important sentiment words are assigned to the longest code-book, and the ratio of sentiment words are significantly decreased along with the smaller code-books. This result shows that AdaComp uses longer codes to represent task-sensitive words and shorter codes to represent less significant words to the task.

6 Conclusion

In this paper, we have described *AdaComp* that adaptively compresses word embeddings by using different lengths of code-books. To this end, we have used the Gumbel-softmax tricks and the binary-constraint networks to learn the code-book selection and its discrete codes in an end-to-end manner. To showcase the general applicability of AdaComp, we conduct four different NLP tasks, which are sentence classification, chunking, natural language inference, and language modeling. Evaluation results have clearly shown that AdaComp could obtain better results than other methods in terms of both accuracy and memory requirement. We also found that AdaComp assigns each word to different code-books by considering the significance of tasks. Although we have focused on compressing the embeddings by learning task-specific features, AdaComp could be used at NLP tasks without fine-tuning. We believe that our method can benefit simultaneously from other compression techniques, such as pruning (Han et al., 2016) and low-precision representation (Ling et al., 2016). We leave this as an avenue for future work.

Acknowledgement

This work was supported by Institute of Information communications Technology Planning Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2019-0-00079 , Artificial Intelligence

Graduate School Program(Korea University)), and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2018R1A2A1A05078380).

References

Guillaume Alain and Yoshua Bengio. 2014. What regularized auto-encoders learn from the data-generating distribution. *The Journal of Machine Learning Research*.

Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. 2015. Fixed point optimization of deep convolutional neural networks for object recognition. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP)*.

Samuel R Bowman, Gabor Angeli, Christopher Potts, and Christopher D Manning. 2015. A large annotated corpus for learning natural language inference. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Samuel R Bowman, Jon Gauthier, Abhinav Rastogi, Raghav Gupta, Christopher D Manning, and Christopher Potts. 2016. A fast unified model for parsing and sentence understanding. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, et al. 2018. Universal sentence encoder. *arXiv preprint arXiv:1803.11175*.

Yunchuan Chen, Lili Mou, Yan Xu, Ge Li, and Zhi Jin. 2016. Compressing neural language models by sparse word representations. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS)*.

Sergey Edunov, Myle Ott, Michael Auli, and David Grangier. 2018. Understanding back-translation at scale. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. 2010. Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*.

Kristina Gulordava, Laura Aina, and Gemma Boleda. 2018. How to represent a word and predict it, too: Improving tied architectures for language modelling. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Song Han, Huizi Mao, and William J Dally. 2016. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. In *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS)*.

Geoffrey E. Hinton. 2013. Neural networks for machine learning. coursera, video lectures.

Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. 2015. Distilling the knowledge in a neural network. *CoRR*, abs/1503.02531.

Minqing Hu and Bing Liu. 2004. Mining and summarizing customer reviews. In *Proceedings of the International conference on Knowledge Discovery and Data mining (KDD)*.

Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Zhiheng Huang, Wei Xu, and Kai Yu. 2015. Bidirectional LSTM-CRF models for sequence tagging. *CoRR*, abs/1508.01991.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks. In *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS)*.

Eric Jang, Shixiang Gu, and Ben Poole. 2016. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*.

Yoon Kim, Yacine Jernite, David Sontag, and Alexander M Rush. 2016. Character-aware neural language models. In *Proceedings of the Association for the Advancement of Artificial Intelligence (AAAI)*.

Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.

Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander Rush. 2017. OpenNMT: Open-source toolkit for neural machine translation. In *Proceedings of ACL 2017, System Demonstrations*, pages 67–72, Vancouver, Canada. Association for Computational Linguistics.

Namhoon Lee, Thalaiyasingam Ajanthan, and Philip HS Torr. 2019. Snip: Single-shot network pruning based on connection sensitivity. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2017. Pruning filters for efficient convnets. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

Shaoshi Ling, Yangqiu Song, and Dan Roth. 2016. Word embeddings with limited memory. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

Mason Liu and Menglong Zhu. 2018. Mobile video object detection with temporally-aware feature maps. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Yang Liu and Mirella Lapata. 2018. Learning structured text representations. *Transactions of the Association of Computational Linguistics*.

Christos Louizos, Matthias Reisser, Tijmen Blankevoort, Efstratios Gavves, and Max Welling. 2019. Relaxed quantization for discretized neural networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-sne. *Journal of machine learning research*.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS)*.

Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training.

Kaiyu Shi and Kai Yu. 2018. Structured word embedding for low memory neural network language model. In *Proceedings of the Interspeech*.

Raphael Shu and Hideki Nakayama. 2018. Compressing word embeddings via deep compositional code learning. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*.

Julien Tissier, Christophe Gravier, and Amaury Habrard. 2019. Near-lossless binarization of word embeddings. In *Proceedings of the Association for the Advancement of Artificial Intelligence (AAAI)*, volume 33.

Erik F Tjong Kim Sang and Sabine Buchholz. 2000. Introduction to the conll-2000 shared task: chunking. In *Proceedings of the International Conference on Computational Natural Language Learning (CoNLL)*.

Xiang Zhang, Junbo Zhao, and Yann LeCun. 2015. Character-level convolutional networks for text classification. In *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS)*.