# LOSS-FREE COMPRESSION OF WORD EMBEDDINGS VIA DEEP COMPOSITIONAL CODE LEARNING

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Natural language processing (NLP) models suffer from a large storage or memory footprint due to the massive number of parameters used in word embeddings. Deploying neural NLP models to mobile devices requires compressing the word embeddings without sacrificing the performance too much. For this purpose, we propose a novel hashing-based approach that constructs the embedding vectors using few basis vectors. The composition of basis vectors is controlled by the hash code assigned to each word. To maximize the compression rate, we adopt the compositional coding approach instead of binary codes. Each code is composed by multiple discrete numbers, such as $(3, 2, 1, 8)$, where the value of each component is limited in a fixed range. We propose to directly learn the discrete codes in an end-to-end neural network by reconstructing the pre-trained embeddings using the Gumbel-softmax trick. Our experiments show the loss-free compression rate achieves $98\%$ in sentiment analysis task and $94\% \sim 99\%$ in machine translation tasks. In both tasks, the proposed method improves the model performance even with a high compression rate. Comparing to other approaches such as character-level segmentation, the proposed method is language-independent and does not require modifications to the network architecture.

## 1 INTRODUCTION

In neural-based Natural Language Processing models, word embeddings play an important role. Neural word embeddings encapsulate the necessary linguistic information in continuous vectors for all words in vocabulary. However, as each word is assigned an independent embedding vector, the number of parameters in the embedding matrix can be tremendously large. For example, the network has to hold 100M embedding parameters to represent 200K words, when each embedding has 500 dimensions.

As only a small portion of the word embeddings is selected in the forward pass, the giant embedding matrix usually does not cause a speed issue. However, having a massive number of parameters in the neural network results in a large storage or memory footprint. When other components of the neural network are also large, the model may fail to fit into GPU memory during training. Moreover, as the demand for low-latency neural computation rises for mobile platforms, some neural-based models are expected to run on mobile devices. It becomes more important to compress the size of NLP models for deploying them to devices with limited memory or storage capacity.

In this work, we attempt to reduce the number of parameters used in word embeddings without hurting the performance of the model. Neural networks are known for the significant redundancy in the connections (Denil et al., 2013). In this work, we further hypothesize that the inter-similarity among words makes the word embeddings having more redundant parameters, as each word is assigned an independent vector. Some words are very similar regarding the semantics. For example, "dog" and "dogs" have almost the same meaning except one is plural. To efficiently represent these two words, it is desirable to share information between the two embeddings. However, a small portion in both vectors still have to be trained independently to capture the syntactic difference.

Follow the intuition of creating partially shared embeddings, instead of assigning each word a unique id, we represent a word $w$ with a code $C_w$ composed of $M$ components. Each component $C_w^i$ is an integer number in $[1, K]$. Ideally, similar words shall have similar codes. For example, we may want $C_{\text{dog}} = (3, 2, 4, 1)$ and $C_{\text{dogs}} = (3, 2, 4, 2)$. Once we obtained such compact codes for

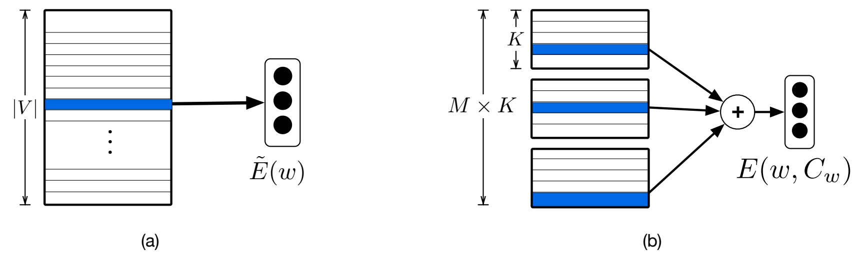


Figure 1: A comparison of embedding computation between the conventional approach (a) and compositional coding approach (b) for constructing embedding vectors

all words in the vocabulary, we assign embedding vectors to the components in the codes rather than unique words. More specifically, we create $M$ codebooks, and each codebook contains $K$ codeword vectors. The embedding of a word $E(w)$ is computed by summing up the codewords for all components in the code as

$$E(w, C_w) = \sum_{i=1}^{M} E_i(C_w^i), \tag{1}$$

where $E_i(\cdot)$ returns a codeword vector for $i$-th component. In this way, the embedding matrix will have $M \times K$ vectors, which is usually much smaller than the vocabulary size. Fig. 1 gives a intuitive comparison between our coding approach and the conventional approach (assigning unique ids).

Although the number of embedding vectors can be greatly reduced by using such coding scheme, we want to avoid the performance loss comparing to using normal embeddings. In other words, given a set of normal word embeddings $\tilde{E}(w)$, we want to find a set of codes $\hat{C}_w$ that can achieve the same effectiveness of $\tilde{E}(w)$ with $E(w, \hat{C}_w)$ for all words. A safe and straight-forward way is to minimize the squared distance between the baseline embeddings and the composed embeddings as

$$\hat{C}_w = \underset{C_w}{\operatorname{argmin}} \frac{1}{|V|} \sum_{w \in V} ||E(w, C_w) - \tilde{E}(w)||^2 \tag{2}$$

$$= \underset{C_w}{\operatorname{argmin}} \frac{1}{|V|} \sum_{w \in V} ||\sum_{i=1}^{M} E_i(C_w^i) - \tilde{E}(w)||^2 , \tag{3}$$

where $|V|$ is the vocabulary size. The baseline embeddings can be a set of pre-trained vectors such as word2vec (Mikolov et al., 2013) or Glove (Pennington et al., 2014) embeddings.

In Eq. 2, the baseline embedding matrix $\tilde{\mathbf{E}}$ is approximated by $M$ codewords selected from $M$ codebooks. The selection of codewords is controlled by the code $C_w$. Such problem of learning compositional compact codes with multiple codebooks is formalized and discussed in the research field of compression-based source coding as *multi-codebook quantization* (Jégou et al., 2011; Babenko & Lempitsky, 2014; Du & Wang, 2014; Martinez et al., 2016). Previous works learn compact codes to perform efficient similarity search of vectors. In this work, we utilize the compact codes for a different purpose, that is, constructing word embeddings with drastically fewer parameters.

Due to the discreteness in the hash codes, it is usually difficult to directly optimize the objective in Eq. 2 for code learning. In this paper, we propose a simple and straight-forward method to directly learn the codes in an end-to-end neural network. We utilize the Gumbel-Softmax trick (Maddison et al., 2016; Jang et al., 2016) to find the best discrete codes that minimize the reconstruction loss.

The contribution of this paper can be summarized as follows:

- We propose to utilize the compositional coding approach to construct the word embeddings with significantly fewer parameters. In the experiments, we show that over 98% of the embedding parameters can be eliminated in sentiment analysis task without hurting performance. In machine translation tasks, the compression rate reaches $94\% \sim 99\%$.

- We propose a way to directly learn the codes in an end-to-end neural network, with a Gumbel-softmax layer to encourage the discreteness.
- The source code for code learning will be packaged into a tool. With the learned codes and basis vectors, the computation graph for composing embeddings is fairly easy to implement, which does not require modifications to other parts in the neural network.

## 2 RELATED WORK

Existing works for compressing neural networks include low-precision computation (Vanhoucke et al., 2011; Hwang & Sung, 2014; Courbariaux et al., 2014; Anwar et al., 2015), quantization (Chen et al., 2015; Han et al., 2015a; Zhou et al., 2017), network pruning (LeCun et al., 1989; Hassibi & Stork, 1992; Han et al., 2015b; Wen et al., 2016) and knowledge distillation (Hinton et al., 2015). Network quantization is a technique to reduce the number of real weights. HashedNet (Chen et al., 2015) forces each weight matrix to have few real weights. For each position in the weight matrix, the assignment of real weights is determined by a hash function. To solve the problem that HashedNet does not capture the non-uniform nature of the networks, DeepCompression (Han et al., 2015a) groups weight values into clusters based on a pre-trained weight matrix. The weights in the same cluster share a same value. The weight assignments for each position in the weight matrix are stored in the form of Huffman codes. However, as the embedding matrix is tremendously big, the number of hash codes a model need to maintain is still very large even with Huffman coding.

Network pruning works in different way that makes a network sparse. Iterative pruning (Han et al., 2015b) prunes a weight value if its absolute value is smaller than a threshold. The remaining network weights are retrained after pruning. Some recent works (See et al., 2016; Zhang et al., 2017) also apply iterative pruning to prune 80% of the connections for neural machine translation models. In this paper, we compare the proposed method with iterative pruning. Please note that as our method produces a dense matrix composed of $M \times K$ basis vectors, there is room to further apply quantization or pruning techniques to achieve a even higher compression rate.

The problem of learning compact codes described in this paper is closely related to learning to hash (Weiss et al., 2008; Kulis & Darrell, 2009; Liu et al., 2012), which aims to learn the hash codes for vectors to facilitate the approximate nearest neighbor search. The fundamental difference between learning to hash and our approach is the objective. Learning to hash methods mainly aim to maintain the distance of input vectors in the Hamming space. Therefore, a majority of the methods in learning to hash use pairwise similarity as the objective function. In contrast, we use reconstruction loss as the objective function.

The codes in learning to hash are usually presented in the binary form. Initiated by product quantization (Jégou et al., 2011), few succeeding works such as additive quantization (Babenko & Lempitsky, 2014) and compositional coding (Du & Wang, 2014) explore the use of multiple codebooks for compression-based source coding. We also adopt the compositional coding scheme for the storage efficiency. Previous works employ alternating optimization as the learning algorithm, which alternatively optimizes the codebooks and the discrete codes. In this work, we propose a method to learn the codes directly in an end-to-end neural network.

Some recent works (Xia et al., 2014; Liu et al., 2016; Yang et al., 2017) in learning to hash also utilize neural networks as hash functions to produce the binary codes. These methods impose binary constraints in the code layer (e.g., sigmoid function) to encourage the discreteness, which cannot be applied to compositional coding approach. In our work, we encourage the discreteness with Gumbel-Softmax reparameterization trick.

As an alternative to our approach, one can also reduce the number of unique word types by forcing a character-level segmentation. Kim et al. (2016) proposed a character-based neural language model, which applies a convolutional layer after the character embeddings. Botha et al. (2017) propose to use char-gram as input features, which are further hashed to save space. Generally, using character-level inputs requires modifications to the model architecture. Moreover, some Asian languages such as Japanese and Chinese have a large vocabulary even in character level, which makes the character-based approach difficult to be applied. In contrast, our approach does not suffer from these limitations.

# 3 COMPOSITIONAL COMPACT CODES

In this section, we give a formal description of the compositional coding approach and analyze its merits. We represent a word $w$ with a compact code $C_w$ that is composed of $M$ components s.t. $C_w \in Z_+^M$. Each component $C_w^i$ is constrained to have a value in $[1, K]$, which also indicates that it takes $M \log_2 K$ bits to store each code. For convenience, $K$ is selected to be a number of a multiple of 2, so that the codes can be efficiently stored.

If we restrict each component $C_w^i$ to take 0 or 1, the code for each word $C_w$ will be a binary code. In this case, the code learning problem is equivalent to a matrix factorization problem with binary components. Forcing the compact codes to be binary numbers can be beneficial as the learning problem is usually easier to solve in the binary case, and some existing optimization algorithms in learning to hash can be reused. However, the compositional coding approach produces shorter codes and is thus more storage efficient.

As the number of basis vectors is $M \times K$ regardless of the vocabulary size, the only uncertain factor contributing to the model size is the size of the hash codes, which is proportional to the number of in-vocabulary words. Therefore, maintaining short codes is important in our work. Suppose we wish the model to have a set of $N$ basis vector, then in the binary case, each code will have $N/2$ bits. For the compositional coding approach, if we can find a $M \times K$ decomposition such that $M \times K = N$, then each code will have $M \log_2 K$ bits. For example, a binary code will have a length of 256 bits to support 512 basis vectors. In contrast, a $32 \times 16$ compositional coding scheme will produce codes of only 128 bits.

| | **#basis** | **computation** | **code length (bits)** |
|---|---|---|---|
| conventional | $\lvert V \rvert$ | 1 | - |
| binary | $N$ | $N/2$ | $N/2$ |
| compositional | $MK$ | $M$ | $M \log_2 K$ |

Table 1: A comparison of different coding approaches in terms of the number of basis vectors, code length and the number of vectors need to be indexed when constructing a word embedding

A comparison of different coding approaches is summarized in Table 1. We also report the number of basis vectors required to compose an embedding as a measure of computational cost. The conventional approach assigns a unique id to each word. Therefore, the number of basis embedding vectors is identical to the vocabulary size and the computation is basically a single indexing operation. In the case of binary codes, the computation for constructing an embedding is a summation over $N/2$ basis vectors. For the compositional approach, the number of vectors required to construct an embedding vector is $M$. Both the binary and compositional approach has significantly fewer basis vectors. The compositional coding approach provides a better balance with shorter codes and lower computational cost.

# 4 CODE LEARNING WITH GUMBEL-SOFTMAX

Let $\tilde{\mathbf{E}} \in \mathbb{R}^{|V| \times H}$ be the original embedding matrix, where each embedding vector has $H$ hidden units. By using the reconstruction loss as the objective in Eq. 2, we are actually finding an approximate matrix factorization $\tilde{\mathbf{E}} \approx \sum_{i=0}^{M} \boldsymbol{D^i A_i}$, where $\boldsymbol{A_i} \in \mathbb{R}^{K \times H}$ is a basis matrix for the $i$-th component. $\boldsymbol{D^i}$ is a $|V| \times K$ code matrix, where each row is an $K$-dimensional one-hot vector. Let $\boldsymbol{d_w^i}$ be the one-hot vector corresponding to the code component $C_w^i$, the computation of the word embeddings can be reformulated as

$$E(w, C_w) = \sum_{i=0}^{M} \boldsymbol{A_i}^\top \boldsymbol{d_w^i}. \tag{4}$$

Therefore, the learning problem of discrete codes $C_w$ can be converted to a problem of finding a set of optimal one-hot vectors $\boldsymbol{d_w^1}, ..., \boldsymbol{d_w^M}$ and source dictionaries $\boldsymbol{A_i}$ that minimize the reconstruction loss. The Gumbel-softmax reparameterization trick (Maddison et al., 2016; Jang et al., 2016) is an

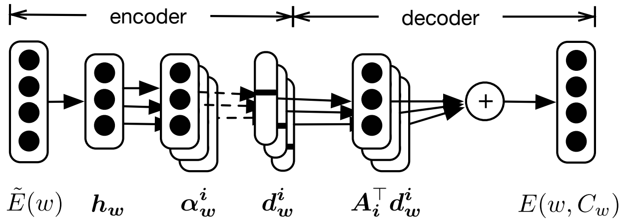


Figure 2: The network architecture for learning compositional compact codes. The Gumbel-softmax computation is marked with dashed lines.

useful tool to parameterize a discrete distribution such as the $K$-dimensional one-hot vectors $\boldsymbol{d_w^i}$ in Eq. 4. By applying the Gumbel-softmax trick, the $k$-th elemement in $\boldsymbol{d_w^i}$ is computed by

$$\left(\boldsymbol{d_w^i}\right)_k = \mathrm{softmax}_\tau(\log \boldsymbol{\alpha_w^i} + G)_k \tag{5}$$

$$= \frac{\exp((\log\ (\boldsymbol{\alpha_w^i})_k + G_k)/\tau)}{\sum_{k'=1}^{K} \exp((\log\ (\boldsymbol{\alpha_w^i})_{k'} + G_{k'})/\tau)}\ , \tag{6}$$

where $G_k$ is a noise term that is sampled from a Gumbel distribution $-\log(-\log(\mathrm{Uniform}[0,1]))$, whereas $\tau$ is the temperature of the softmax. In our model, the vector $\boldsymbol{\alpha_w^i}$ is computed by a simple neural network with a single hidden layer as

$$\boldsymbol{\alpha_w^i} = \mathrm{softplus}(\boldsymbol{\theta_i'}^\top \boldsymbol{h_w} + \boldsymbol{b_i'}), \tag{7}$$

$$\boldsymbol{h_w} = \tanh(\boldsymbol{\theta}^\top \tilde{E}(w) + \boldsymbol{b})\ . \tag{8}$$

In our experiments, the hidden layer $\boldsymbol{h_w}$ always has a size of $MK/2$. We find that a fixed temperature of $\tau = 1$ just works well. As described in Eq. 6, the Gumbel-softmax trick is applied to $\boldsymbol{\alpha_w^i}$ to obtain $\boldsymbol{d_w^i}$. Finally, the model reconstructs the embedding $E(w, C_w)$ with Eq. 4 and compute the reconstruction loss in Eq. 2. The model architecture of the end-to-end neural network is illustrated in Fig. 2, which is effectively an auto-encoder with a Gumbel-softmax middle layer. The whole neural network for coding learning has five parameters ($\boldsymbol{\theta}, \boldsymbol{b}, \boldsymbol{\theta'}, \boldsymbol{b'}, \boldsymbol{A}$).

Once the coding learning model is trained, the code $C_w$ for a word can be easily obtained by applying $\mathrm{argmax}$ to the one-hot vectors $\boldsymbol{d_w^i}$. The basis vectors (codewords) for composing the embeddings can be found as the row vectors in the weight matrix $\boldsymbol{A}$.

For general NLP tasks, one can learn the compositional codes from public available word vectors such as Glove vectors. However, for some tasks such as machine translation, the word embeddings are usually jointly learned with other parts of the neural network. For such tasks, one has to train a normal model first to obtain the embeddings for all in-vocabulary words. Then based on the trained embedding matrix, one can learn a set of task-specific codes. As the reconstructed embeddings are not identical to the original embeddings, the model parameters other than the embedding matrix have to be retrained again.

## 5 EXPERIMENTS

In our experiments, we focus on evaluating the maximum loss-free compression rate of word embeddings on two typical NLP tasks: sentiment analysis and machine translation. We compare the model performance and the size of embedding layer with the uncompressed baseline mode and the iterative pruning method (Han et al., 2015b). Please note that the sizes of other parts in the neural networks are not included in our results. For dense matrices, we report the size of dumped numpy arrays. For the sparse matrices, we report the size of dumped *compressed sparse column matrices* (`csc_matrix`) in scipy. All float numbers take 32 bits storage. We enables the "compressed" option when dumping the matrices, without this option, the file size is about 1.1 times bigger.

### 5.1 Code Learning

To learn efficient compact codes for each word, our proposed method requires a set of baseline embedding vectors. For the sentiment analysis task, we learn the codes based on the publicly available Glove vectors. For the machine translation task, we first train a normal neural machine translation (NMT) model to obtain task-specific word embeddings. Then we learn the codes using the pre-trained embeddings.

We train the end-to-end network described in Section 4 to learn the codes automatically. In each iteration, a small batch of the embeddings is sampled uniformly from the baseline embedding matrix. The network parameters are optimized to minimize the reconstruction loss of the sampled embeddings. In our experiments, the batch size is set to 128. We use Adam optimizer (Kingma & Ba, 2014) with a fixed learning rate of 0.0001. The training is run for 200K iterations. Every 1,000 iterations, we examine the loss on a fixed validation set and save the parameters if the loss decreases. We evenly distribute the model training to 4 GPUs using the *nccl* package, so that one round of code learning takes around 15 minutes to complete.

### 5.2 Sentiment Analysis

**Dataset:** For sentiment analysis, we use a standard separation of IMDB movie review dataset (Maas et al., 2011), which contains 25k reviews for training and 25K reviews for testing purpose. We lowercase and tokenize all texts with the *nltk* package. We choose the 300-dimensional uncased Glove word vectors (trained on 42B tokens of Common Crawl data) as our baseline embeddings. The vocabulary for the model training contains all words appears both in the IMDB dataset and the Glove vocabulary, which results in around 75K words. We truncate the texts of reviews to assure they are not longer than 400 words.

**Model architecture:** Both the baseline model and the compressed small model has the same computational graph except the embedding layer. The model is composed of a single LSTM layer with 150 hidden units and a softmax layer for predicting the binary label. For the baseline model, the embedding layer contains a large $75K \times 300$ embedding matrix initialized by Glove embeddings. For the compressed models based on the compositional coding approach, the embedding layer maintains the matrix of basis vectors. Suppose we use a $32 \times 16$ coding scheme, the basis matrix will then have a shape of $512 \times 300$, which is initialized by the concatenated weight matrices $[\boldsymbol{A_1}; \boldsymbol{A_2}; ...; \boldsymbol{A_M}]$ in the code learning model. The embedding parameters for both models remain fixed during the training. For the models with network pruning, the sparse embedding matrix is finetuned is finetuned during training.

**Training details:** The models are trained with Adam optimizer for 15 epochs with a fixed learning rate of $0.0001$. At the end of each epoch, we evaluate the loss on a small validation set. The parameters with lowest validation loss are saved.

**Results:** For different settings of the number of components $M$ and the number of codeword $K$, we train the code learning network. The average reconstruct loss (least-squares error) on a fixed validation set is summarized in the left of Table 2. For reference, we also report the total size (MB) of the embedding layer in the right table, which includes the sizes of the basis matrix and the hash table. We can see that increasing either $M$ or $K$ can effectively decrease the reconstruction loss. However, setting $M$ to a large number will result in longer hash codes, thus significantly increase the size of the embedding layer. Hence, it is important to choose correct numbers for $M$ and $K$ to balance the performance and model size.

To see how the reconstructed loss translates to the classification accuracy, we train the sentiment analysis model for different settings of code schemes and report the results in Table 3. The baseline model using 75k Glove embeddings achieves an accuracy of 87.18 with an embedding matrix using 78 MB of storage. In this task, forcing a high compression rate with iterative pruning degrades the classification accuracy.

For the models using compositional codes, the maximum loss-free compression rate is achieved by a $16 \times 32$ coding scheme. In this case, the total size of the embedding layer is 1.23 MB, which is equivalent to a compression rate of 98.4%. We also found the classification accuracy can be substantially improved with a slightly lower compression rate. The improved model performance may be a byproduct of the strong regularization.

| loss | M=8 | M=16 | M=32 | M=64 |
|---|---|---|---|---|
| K=8 | 29.1 | 25.8 | 21.9 | 15.5 |
| K=16 | 27.0 | 22.8 | 19.1 | 11.5 |
| K=32 | 24.4 | 20.4 | 14.3 | 9.3 |
| K=64 | 21.9 | 16.9 | 12.1 | 7.6 |

| size (MB) | M=8 | M=16 | M=32 | M=64 |
|---|---|---|---|---|
| K=8 | 0.28 | 0.56 | 1.12 | 2.24 |
| K=16 | 0.41 | 0.83 | 1.67 | 3.34 |
| K=32 | 0.62 | 1.24 | 2.48 | 4.96 |
| K=64 | 0.95 | 1.91 | 3.82 | 7.64 |

Table 2: Reconstruction loss and the size of embedding layer (MB) of difference settings

| | **#vectors** | **vector size** | **code len** | **code size** | **total size** | **accuracy** |
|---|---|---|---|---|---|---|
| Glove baseline | 75102 | 78 MB | - | - | 78 MB | 87.18 |
| prune 80% | 75102 | 21 MB | - | - | 21 MB | 86.25 |
| prune 90% | 75102 | 11 MB | - | - | 11 MB | 84.96 |
| prune 95% | 75102 | 5.31 MB | - | - | 5.31 MB | 83.88 |
| $8 \times 8$ coding | 64 | 0.06 MB | 24 bits | 0.21 MB | 0.27 MB | 82.84 |
| $8 \times 16$ coding | 128 | 0.13 MB | 32 bits | 0.28 MB | 0.41 MB | 83.77 |
| $16 \times 8$ coding | 128 | 0.13 MB | 48 bits | 0.42 MB | 0.55 MB | 85.21 |
| $8 \times 64$ coding | 512 | 0.52 MB | 48 bits | 0.42 MB | 0.94 MB | 86.66 |
| $16 \times 32$ coding | 512 | 0.52 MB | 80 bits | 0.71 MB | **1.23 MB** | 87.37 |
| $32 \times 16$ coding | 512 | 0.52 MB | 128 bits | 1.14 MB | 1.66 MB | 87.80 |
| $64 \times 8$ coding | 512 | 0.52 MB | 192 bits | 1.71 MB | 2.23 MB | **88.15** |

Table 3: Trade-off between the model performance and the size of embedding layer in IMDB sentiment analysis task

### 5.3 Machine Translation

**Dataset:** For machine translation tasks, we experiment on IWSLT 2014 German-to-English translation task (Cettolo et al., 2014) and ASPEC English-to-Japanese translation task (Nakazawa et al., 2016). The IWSLT14 training data contains 178K sentence pairs, which is a small dataset for machine translation. We utilize moses toolkit (Koehn et al., 2007) to tokenize and lowercase both sides of the texts. Then we concatenate all five TED/TEDx development and test corpus to form a test set containing 6750 sentence pairs. We apply byte-pair encoding (Sennrich et al., 2016) to transform the texts to subword level so that the vocabulary has a size of 20K for each language. For evaluation, we report *tokenized BLEU* using "multi-bleu.perl".

The ASPEC dataset contains 300M bilingual pairs in the training data with the automatic estimated quality scores provided for each pair. We only use the first 150M pairs for training the models. The English texts are tokenized by moses toolkit whereas the Japanese texts are tokenized by kytea (Neubig et al., 2011). The vocabulary size for each language is reduced to 40K using byte-pair encoding. The evaluation is performed using a standard kytea-based post-processing script for this dataset.

**Model architecture:** In our preliminary experiments, we found a $32 \times 16$ coding works well for a vanilla NMT model. As it is more meaningful to test on a high-performance model, we applied several techniques to improve the performance. The model has a standard bi-directional encoder composed of two LSTM layers similar to Bahdanau et al. (2014). The decoder contains two LSTM layers. Residual connection (He et al., 2016) with a scaling factor of $\sqrt{1/2}$ is applied to the two decoder states to compute the outputs. All LSTMs and embeddings have 256 hidden units in IWSLT14 task and 1000 hidden units in ASPEC task. The decoder states are firstly linearly transformed to 600-dimensional vectors before computing the final softmax. Dropout with a rate of 0.2 is applied everywhere except the recurrent computation. We apply Key-Value Attention (Miller et al., 2016) to the first decoder, where the query is the sum of the feedback embedding and the previous decoder state and the keys are computed by linear transformation of encoder states.

**Training details:** All models are trained by Nesterov's accelerated gradient (Nesterov, 1983) with an initial learning rate of 0.25. We evaluate the smoothed BLEU (Lin & Och, 2004) on a validation set composed of 50 batches every 7,000 iterations. The learning rate is reduced by a factor of 10 if no improvement is observed in 3 validation runs. The training ends after the learning rate is reduced three times. Similar to the code learning, the training is distributed to 4 GPUs, each GPU computes a mini-batch of 16 samples.

We firstly train a baseline NMT model to obtain the task-specific embeddings for all in-vocabulary words in both languages. Then based on these baseline embeddings, we obtain the hash codes and basis vectors by training the code learning model. Finally, the NMT models using compositional coding are retrained by plugging in the reconstructed embeddings. Note that the embedding layer is fixed in this phase, other parameters are retrained from random initial values.

**Results:** The experimental results are summarized in Table 4. All translations are decoded by the beam search with a beam size of 5. The performance of iterative pruning varies between tasks. The loss-free compression rate reaches 92% on ASPEC dataset by pruning 90% of the connections. However, with the same pruning ratio, a modest performance loss is observed in IWSLT14 dataset.

For the models using compositional coding, the loss-free compression rate is 94% for the IWSLT14 dataset and 99% for the ASPEC dataset. Similar to the sentiment analysis task, a significant performance improvement can be observed by slightly lowering the compression rate. Note that the sizes of NMT models are still quite large due to the big softmax layer and the recurrent layers, which are not reported in the table. Please refer to existing works such as Zhang et al. (2017) for the techniques of compressing layers other than word embeddings.

| | **coding** | **#vectors** | **vector size** | **code len** | **code size** | **total size** | **BLEU(%)** |
|---|---|---|---|---|---|---|---|
| De $\rightarrow$ En | baseline | 40000 | 35 MB | - | - | 35 MB | 29.45 |
| | prune 90% | 40000 | 5.21 MB | - | - | 5.21 MB | 29.34 |
| | prune 95% | 40000 | 2.63 MB | - | - | 2.63 MB | 28.84 |
| | $32 \times 16$ | 512 | 0.44 MB | 128 bits | 0.61 MB | 1.05 MB | 29.04 |
| | $32 \times 32$ | 1024 | 0.89 MB | 160 bits | 0.76 MB | 1.65 MB | 29.17 |
| | $64 \times 16$ | 1024 | 0.89 MB | 256 bits | 1.22 MB | **2.11 MB** | **29.56** |
| En $\rightarrow$ Ja | baseline | 80000 | 274 MB | - | - | 274 MB | 37.93 |
| | prune 90% | 80000 | 41 MB | - | - | 41 MB | 38.56 |
| | prune 95% | 80000 | 21 MB | - | - | 21 MB | ? |
| | $32 \times 16$ | 512 | 1.75 MB | 128 bits | 1.22 MB | **2.97 MB** | 38.10 |
| | $64 \times 16$ | 1024 | 3.50 MB | 256 bits | 2.44 MB | 5.94 MB | **38.89** |

Table 4: Trade-off between the model performance and the size of embedding layer in machine translation tasks

# 6 Qualitative Analysis

## 6.1 Examples of Learned Codes

In Table 5, we show some examples of learned codes based on the 300-dimensional uncased Glove embeddings used in the sentiment analysis task. We can see that the model learned to assign similar codes to the words with similar meanings. Such a code-sharing mechanism can largely reduce the redundancy of the word embeddings, thus help to achieve a high compression rate.

| **category** | **word** | $8 \times 8$ **code** | $16 \times 16$ **code** |
|---|---|---|---|
| animal | dog | 0 7 0 1 7 3 7 0 | 7 7 0 8 3 5 8 5 B 2 E E 0 B 0 A |
| | cat | 7 7 0 1 7 3 7 0 | 7 7 2 8 B 5 8 C B 2 E E 4 B 0 A |
| | penguin | 0 7 0 1 7 3 6 0 | 7 7 E 8 7 6 4 C F D E 3 D 8 0 A |
| verb | go | 7 7 0 6 4 3 3 0 | 2 C C 8 2 C 1 1 B D 0 E 0 B 5 8 |
| | went | 4 0 7 6 4 3 2 0 | B C C 6 B C 7 5 B 8 6 E 0 D 0 4 |
| | gone | 7 7 0 6 4 3 3 0 | 2 C C 8 0 B 1 5 B D 6 E 0 2 5 A |

Table 5: Examples of learned compositional codes based on Glove embedding vectors

### 6.2 Code Balance

Besides the performance of learned codes, we also care about the storage efficiency of the codes. If the model produces unbalance codes, some codewords will be overused, while some codewords will be used only by few words. In our experiments, when using a $8 \times 8$ coding scheme, we found 31% of the words have a subcode of "0" for the first component, while the subcode "1" is only used by 5% of the words.

Figure 3 gives a visualization of the code balance for different coding schemes. Each cell shows the percentage of a specific subcode, thus the values in each column sum to one. We found that the learned codes for a tight coding scheme are unbalanced for some components. However, when a $32 \times 32$ coding scheme is used, the problem is largely mitigated.

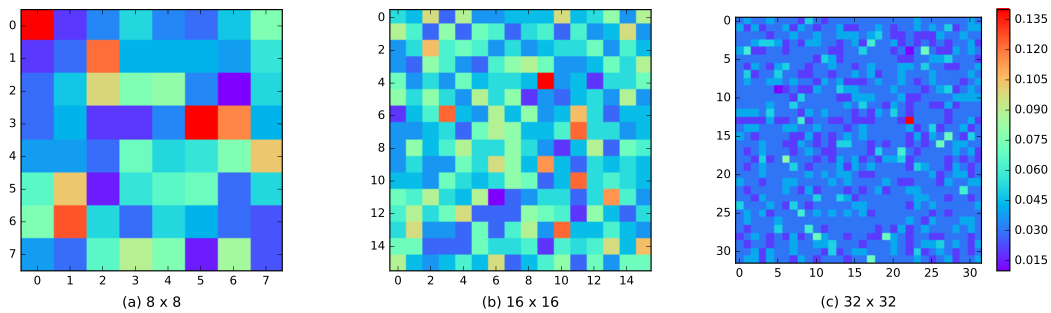


Figure 3: Visualization of code balance for different settings. Each cell in the heat map shows the percentage of a specific subcode. The results indicates the codes are evenly assigned to the words in the vocabulary when a large coding scheme (e.g., $32 \times 32$) is used.

## 7 Conclusion

In this work, we propose a novel way to reduce the number of parameters in word embeddings. Instead of assigning each unique word an embedding vector, we compose the embedding vectors using a small set of basis vectors. The selection of basis vectors is governed by the hash code of each word. We the compositional coding approach to maximize the storage efficiency. The proposed method works by eliminating the redundancy caused by representing similar words with independent embeddings. In our work, we also propose a simple way to directly learn the discrete codes in a neural network with Gumbel-softmax trick. The results show that the size of embedding layer can be reduced by 98% in IMDB sentiment analysis task and $94\% \sim 99\%$ in machine translation task without hurting performance.

Our approach achieves a high loss-free compression rate by considering the semantic inter-similarity among different words. In qualitative analysis, we found the learned codes of similar words are very close in Hamming space. As our approach maintains a dense basis matrix, it has the potential to be further compressed by applying quantization or pruning techniques to the dense matrix. The advantage of compositional coding approach will be more significant if the size of embedding layer is dominated by the hash codes.

## References

Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Fixed point optimization of deep convolutional neural networks for object recognition. *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1131–1135, 2015.

Artem Babenko and Victor S. Lempitsky. Additive quantization for extreme vector compression. *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 931–938, 2014.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Jan A. Botha, Emily Pitler, Ji Ma, Anton Bakalov, Alex Salcianu, David Weiss, Ryan T. McDonald, and Slav Petrov. Natural language processing with small feed-forward networks. In *EMNLP*, 2017.

Mauro Cettolo, Jan Niehues, Sebastian Stüker, Luisa Bentivogli, and Marcello Federico. Report on the 11th iwslt evaluation campaign, iwslt 2014. In *Proceedings of the International Workshop on Spoken Language Translation, Hanoi, Vietnam*, 2014.

Wenlin Chen, James T. Wilson, Stephen Tyree, Kilian Q. Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *ICML*, 2015.

Matthieu Courbariaux, Yoshua Bengio, and J David. Low precision arithmetic for deep learning. *CoRR, abs/1412.7024*, 4, 2014.

Misha Denil, Babak Shakibi, Laurent Dinh, Marc'Aurelio Ranzato, and Nando de Freitas. Predicting parameters in deep learning. In *NIPS*, 2013.

Chao Du and Jingdong Wang. Inner product similarity search using compositional codes. *CoRR*, abs/1406.4966, 2014.

Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015a.

Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. *CoRR*, abs/1506.02626, 2015b.

Babak Hassibi and David G. Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *NIPS*, 1992.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

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

Kyuyeon Hwang and Wonyong Sung. Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1. In *Signal Processing Systems (SiPS), 2014 IEEE Workshop on*, pp. 1–6. IEEE, 2014.

Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *CoRR*, abs/1611.01144, 2016.

Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33:117–128, 2011.

Yoon Kim, Yacine Jernite, David Sontag, and Alexander M. Rush. Character-aware neural language models. In *AAAI*, 2016.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondrej Bojar, Alexandra Constantin, and Evan Herbst. Moses: Open source toolkit for statistical machine translation. In *ACL*, 2007.

Brian Kulis and Trevor Darrell. Learning to hash with binary reconstructive embeddings. In *NIPS*, 2009.

Yann LeCun, John S. Denker, and Sara A. Solla. Optimal brain damage. In *NIPS*, 1989.

Chin-Yew Lin and Franz Josef Och. Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics. In *ACL*, 2004. doi: 10.3115/1218955.1219032.

Haomiao Liu, Ruiping Wang, Shiguang Shan, and Xilin Chen. Deep supervised hashing for fast image retrieval. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2064–2072, 2016.

Wei Liu, Jun Wang, Rongrong Ji, Yu-Gang Jiang, and Shih-Fu Chang. Supervised hashing with kernels. *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2074–2081, 2012.

Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *ACL*, 2011.

Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. *CoRR*, abs/1611.00712, 2016.

Julieta Martinez, Joris Clement, Holger H. Hoos, and James J. Little. Revisiting additive quantization. In *ECCV*, 2016.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, 2013.

Alexander H. Miller, Adam Fisch, Jesse Dodge, Amir-Hossein Karimi, Antoine Bordes, and Jason Weston. Key-value memory networks for directly reading documents. In *EMNLP*, 2016.

Toshiaki Nakazawa, Manabu Yaguchi, Kiyotaka Uchimoto, Masao Utiyama, Eiichiro Sumita, Sadao Kurohashi, and Hitoshi Isahara. Aspec: Asian scientific paper excerpt corpus. In *LREC*, 2016.

Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence o (1/k2). In *Doklady an SSSR*, volume 269, pp. 543–547, 1983.

Graham Neubig, Yosuke Nakata, and Shinsuke Mori. Pointwise prediction for robust, adaptable japanese morphological analysis. In *ACL*, pp. 529–533, 2011.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1532–1543, 2014. URL `http://www.aclweb.org/anthology/D14-1162`.

Abigail See, Minh-Thang Luong, and Christopher D. Manning. Compression of neural machine translation models via pruning. In *CoNLL*, 2016.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *ACL*, 2016.

Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on cpus. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, volume 1, pp. 4, 2011.

Yair Weiss, Antonio Torralba, and Rob Fergus. Spectral hashing. In *NIPS*, 2008.

Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *NIPS*, 2016.

Rongkai Xia, Yan Pan, Hanjiang Lai, Cong Liu, and Shuicheng Yan. Supervised hashing for image retrieval via image representation learning. In *AAAI*, 2014.

Huei-Fang Yang, Kevin Lin, and Chu-Song Chen. Supervised learning of semantics-preserving hash via deep convolutional neural networks. *IEEE transactions on pattern analysis and machine intelligence*, 2017.

Xiaowei Zhang, Wei Chen, Feng Wang, Shuang Xu, and Bo Xu. Towards compact and fast neural machine translation using a combined method. In *EMNLP*, 2017.

Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. *CoRR*, abs/1702.03044, 2017.

# A Appendix: Shared Codes

In both tasks, when we use a small code decomposition, we found some hash codes will be assigned to multiple words. Table 6 shows some samples of shared codes with their corresponding words in the sentiment analysis task. This phenomenon does not cause a problem in both tasks, as the words only have shared codes when they have almost same sentiments or target translations.

However, the shared codes cause problems when we attempted to compress the weight matrix of the softmax layer in NMT models, which is another source of large model size. When the weights are same for two different words, the model can no longer discriminate between the words, which we believe is partially the reason why the proposed method cannot be successfully applied to the softmax layer.

| **shared code** | **words** |
|---|---|
| 4 7 7 0 4 7 1 1 | homes cruises motel hotel resorts mall vacations hotels |
| 6 6 7 1 4 0 2 0 | basketball softball nfl nascar baseball defensive ncaa tackle nba |
| 3 7 3 2 4 3 3 0 | unfortunately hardly obviously enough supposed seem totally ... |
| 4 6 7 0 4 7 5 0 | toronto oakland phoenix miami sacramento denver minneapolis ... |
| 7 7 6 6 7 3 0 0 | yo ya dig lol dat lil bye |

Table 6: Examples of words sharing same codes when using a $8 \times 8$ code decomposition