LOSS-FREE REDUCTION OF WORD EMBEDDINGS VIA DEEP COMPOSITIONAL CODE LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Natural language processing (NLP) models suffers from a large storage or memory footprint due to the massive number of parameters of word embeddings. Applying neural NLP models to mobile devices requires reducing the parameter size of word embeddings without sacrificing the performance too much. For this purpose, we propose a novel hashing-based method to construct the embedding vectors for all words in the vocabulary using a small list of basis vectors. The selection of basis vectors is controlled by the hash code for each word. To maximize the storage efficiency, we use compositional codes 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 compositional codes in an end-to-end neural network minimizing the reconstruction loss based on a set of pre-trained embeddings such as Glove vectors. Our experiments on a sentiment analysis task and machine translation tasks show that the proposed method can reduce the size of embedding layer over 98% without performance loss. In sentiment analysis task, the reconstructed embeddings even improve the classification accuracy by around 1%. Comparing to other methods such as using character-level segmentation, the proposed method is language-independent and does not require modification of the model architecture.

1 INTRODUCTION

In neural-based Natural Language Processing models, word embeddings play an important role. Neural word embeddings automatically learns the representation of each word 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 a size of 500.

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 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 word embedding layer may fail to fit into GPU memory during training. In this case, one has to place the word embeddings in CPU memory, which results in the data transfer between CPU and GPU in every forward passes.

Moreover, as the demand for low-latency neural computation rises for mobile platforms, more neural-based models are expected run on mobile devices. It becomes more important to compress the size NLP models for deploying them to devices with limited memory capacity.

In this work, we attempt to reduce the number of parameters of word embeddings without hurting the performance of the model. We hypothesize that learning independent embeddings for all words causes the redundancy in embedding vectors, as the inter-similarity among words is ignored. Some words are very similar in terms of morphology or semantics. For example, "dog" and "dogs" have almost the same meaning except one is plural. To efficiently represent these two words, it desirable to share most parts of the two embeddings to capture the main meaning, except a small portion in both vectors to be trained independently.

Follow the intuition of creating partially shared embeddings, instead 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

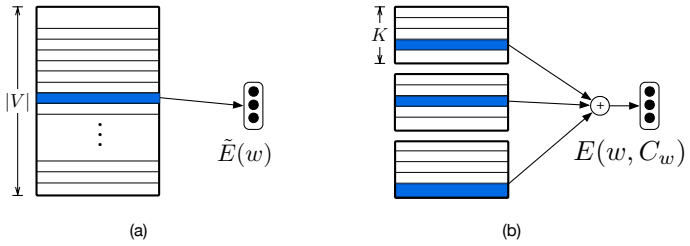

Figure 1: A comparison of embedding computation between the conventional coding approach (a) and compositional coding approach (b)

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 all words in the vocabulary, we use embedding vectors to represent the codes rather than unique words. In this case, we create M codebooks, each codebook contains K codeword vectors. The embedding of a word E(w) is computed by the sum of 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 kind of code decomposition, we want to avoid the performance loss compared to normally trained embeddings. In other words, given a set of conventional 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 baseline embeddings and the compositional embeddings as

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

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

where $|V|$ is the vocabulary size. The baseline embeddings can be a set of pre-trained vectors such as Glove (cite) or word2vec (cite) 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 codes 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 learns compact codes in order 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 less parameters.

Due to the discreteness in $\boldsymbol{C_i}$, it is usually difficult to optimize the objective. 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 decomposition that forces the model to assign similar codes to the words with similar embeddings in a trained model.

The contribution of this paper can be summarized as three folds:

- We propose to utilize a compositional compact coding approach to construct the word embeddings with drastically less parameters. In the experiments, we show that over 98% of the embedding parameters can be eliminated without hurting performance.

- We propose a way 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, the computation graph for constructing the compositional embeddings is fairly easy to implement, which does not require modification to other parts in the neural network.

2 Related Work

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 hash codes for input vectors to facilitate approximate nearest neighbor search. By mapping input vectors (e.g. image features) to the binary (Hamming) space, indexing and searching can be efficiently performed. The fundamental difference between learning to hash and our approach is in the objective. The hashing 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, as our goal is to keep the effectiveness of reconstructed embeddings, we use reconstruction loss as the objective function.

Although the codes in learning to hash methods are usually presented in binary form. Initiated by product quantization (Jégou et al., 2011), few succeeding works such as additive quatization (Babenko & Lempitsky, 2014) and compositional coding (Du & Wang, 2014) explore the use of multiple codebooks for compression-based source coding. These methods decompose vectors into multiple components in different subspaces, which are similar to our approach in terms of the code scheme. Both of them employ alternating optimization as the learning algorithm, which alternatively optimizes the codebooks and discrete codes. Our approach differs by learning the codes directly in an end-to-end neural network. As the codewords are not reused for constructing the word embeddings, we are more focused on learning high-quality compositional codes.

Besides the difference of objective function, some recent methods (Xia et al., 2014; Liu et al., 2016; Yang et al., 2017) in learning to hash also utilize neural networks as the hash function to produce binary hash codes. These methods impose binary constraints in the code layer (e.g. sigmoid function) to encourage the discreteness. However, their methods can not be applied to learn the compositional compact codes described in this paper. In our work, we encourage the discreteness by using a Gumbel-Softmax layer.

As an alternative to our approach, one can also reduce the number of unique word types in the vocabulary by forcing a character-level segmentation. (Kim et al., 2016) proposed a character-based neural language model, which requires a convolutional layer applied after character embeddings. One drawback of such approach is that the convolutional operation slows down the computation, as shown in their experiments. Moreover, some asian languages such as Japanese and Chinese has a large vocabulary even in character level, which makes their approach difficult to be applied to such languages. In contrast, our approach does not suffer from such limitation.

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 take $\log_2 K$ bits of storage 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 case of binary code, and some existing optimization algorithms in learning to hash can be reused. However, using compositional codes is more storage efficient compared to binary codes by producing shorter codes.

Besides the embedding parameters, the model also has to store the hash codes for all words in the vocabulary. The footprint of the hash codes depends on the vocabulary size and the number of bits for each single code. Suppose we wish the model to have a set of $M \times K$ basis vector, then the code length with be $\frac{1}{2}MK$ bits compared to $M \log_2 K$ bits when using a $M \times K$ code decomposition. For example, a binary code will have a length of 256 bits to support 512 basis vectors. In contrast, a 32×16 code decomposition will produce codes of only 128 bits.

	#basis	**computation**	**code length (bits)**
conventional	$\|V\|$	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. The conventional coding approach assigns an 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 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 less basis vectors. The compositional coding approach provides the better balance between the number of basis vector and the code length.

4 Code Learning with Gumbel-Softmax

By using the reconstruction loss as the objective in Eq. 2, we are actually finding an approximate matrix factorization $\tilde{\mathbf{E}} \approx \sum_{i=0}^{M} \boldsymbol{D^i} \boldsymbol{A_i}$, where $\tilde{\mathbf{E}} \in \mathbb{R}^{|V| \times N}$ is the original embedding matrix, and $\boldsymbol{A_i} \in \mathbb{R}^{K \times N}$ is a basis matrix for i-th position. $\boldsymbol{D^i}$ is a $|V| \times K$ matrix, where each row is an K-dimensional one-hot vector. Let $\boldsymbol{d_w^i}$ be the one-hot vector corresponding to the 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 is 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 in Eq. 2. The Gumbel-softmax reparameterization trick (Maddison et al., 2016; Jang et al., 2016) is an 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

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

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

where G_k is a noise term that is sampled from a Gumbel distribution $-\log(-\log(\mathrm{Uniform}[0,1]))$, whereas τ 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

$$\boldsymbol{h_w} = \tanh(\boldsymbol{\theta}^\top \tilde{E}(w) + \boldsymbol{b}) \tag{7}$$

$$\boldsymbol{\alpha_w^i} = \mathrm{softplus}(\boldsymbol{\theta_i'}^\top \boldsymbol{h_w} + \boldsymbol{b_i'})\,. \tag{8}$$

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

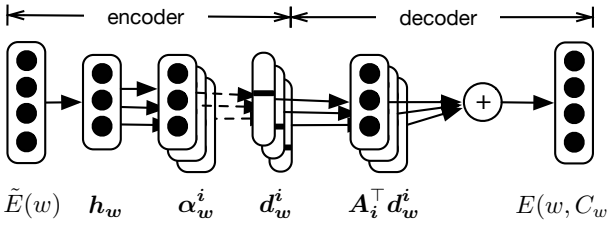

Figure 2: The network architecture for learning compositional compact codes. The Gumbel-softmax computation is marked with dashed lines.

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 argmax to the one-hot vectors $\boldsymbol{d_w^i}$. The basis vectors (codewords) for constructing the embeddings are exactly the row vectors in the matrix $\boldsymbol{A}$, which is a decoder parameter in the code learning model.

For general NLP tasks, one can learn the compositional codes from public available word vectors such as word2vec (cite) and GloVe (cite). 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.

5 Experiments

5.1 Experiment Settings

In our experiments, we focus on evaluating the maximum loss-free reduction rate of embedding numbers on two typical NLP tasks: sentiment analysis and machine translation.

5.2 Code Learning

In order 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 neural machine translation (NMT) model with a normal embedding layer 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 are sampled uniformly from the baseline embedding matrix. The network parameters are optimized to minimized the reconstruction loss of the sampled embeddings. In our experiments, the batch size is set to 128. We use Adam optimizer (cite) 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 12 minutes to complete.

5.3 Sentiment Analysis

Dataset: For sentiment analysis, we use a standard separation of IMDB movie review dataset (cite), 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 vocabulary of

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	0.57	1.15	2.30
K=16	0.43	0.86	1.73	3.46
K=32	0.65	1.30	2.60	5.20
K=64	1.01	2.03	4.06	8.12

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

Glove vectors, which results in around 75K words. We truncate the texts of reviews to assure they are not longer than 400 words. In this task, we attempt to learn an efficient compositional codes based on the Glove vectors to achieve comparable performance, but with significantly less embedding parameters.

Model architecture: Both the baseline model using Glove vectors and the small model have the same computational graph except the embedding layer. The model is composed by 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 small model, suppose we use a 32×16 code decomposition. The embedding matrix will then have shape 512×300, which is initialized by the concatenated basis matrix $[\boldsymbol{A_1}; \boldsymbol{A_2}; ...; \boldsymbol{A_M}]$. The embedding parameters for both models remain fixed during the training.

Training details: The models are trained with Adam optimizer (cite) for 15 epochs with a fixed learning rate of 0.0001. In the end of each epoch, we evaluate the model performance on a small validation set. The parameters with lowest validation loss is saved.

Results: For different settings of the number of components M and the number of codeword candidates 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 final size (MB) of embedding layer in the right table, which is the total size of embedding vectors and compact codes (Eq. (ref)). We can see that increasing either M or K can effectively improve the reconstruction loss. However, setting M to a large number will result in longer codes for representing each word, thus significantly increase the size of embedding layer. Hence, it is important to choose a correct setting of 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 few settings of code learning and report the results in Table 3. The baseline model using 75k Glove embeddings achieves an accuracy of 87.18. The uncompressed size of the embedding matrix is 85.94MB when each float number takes 32 bits of storage. For the models using code decomposition, we can observe a clear trade-off between the accuracy and the size of embedding layer.

	#vectors	**code len**	**embed size**	**accuracy**
Glove baseline	75102	-	85.94MB	87.18
8×8 code decomp.	64	24 bits	0.28 MB	82.84
8×16 code decomp.	128	32 bits	0.43MB	83.77
16×8 code decomp.	128	48 bits	0.57 MB	85.21
8×64 code decomp.	512	48 bits	1.01 MB	86.66
16×32 code decomp.	512	80 bits	**1.30 MB**	**87.37**
32×16 code decomp.	512	128 bits	**1.73 MB**	**87.80**
64×8 code decomp.	512	192 bits	**2.30 MB**	**88.15**

Table 3: A comparison of performance and size of embedding layer for different settings of code learning

Surprisingly, the model using reconstructed embeddings starts to outperform the baseline using original Glove-based embeddings with a 16×32 code decomposition. In this case, the size of 512 vectors is 0.58 MB, whereas the codes of 75k words takes 0.71 MB of storage. The uncompressed size of embedding layer is 1.30 MB, achieving a reduction rate of 98.4%. Increasing the code length

is shown to improve the classification performance, which also results in larger model sizes as a byproduct.

5.4 MACHINE TRANSLATION

Dataset: For machine translation tasks, we conduct the experiment on two language pairs: IWSLT14 German-to-English task and ASPEC English-to-Japanese task,

6 QUALITATIVE ANALYSIS

6.1 EXAMPLES OF LEARNED CODES

In Table 4, we show some examples of learned codes based the 300-dimensional uncased Glove embeddings used in the sentiment analysis task. We can see that similar words have similar codes for both categories of words. Such a code sharing mechanism can largely reduce the redundancy of the word embeddings, thus help to achieve a high reduction rate.

category	**word**	8×8 **code**	16×16 **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 4: Examples of learned compositional codes based on Glove embedding vectors

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 5: Examples of words sharing same codes when using a 8×8 code decomposition

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, then some codewords will be overused, while some codewords will be used only by few words. In our experiments, we found 31% of the words have a subcode of "0" for the first component when using a 8×8 code decomposition, whereas the subcode "1" is only used by 5% of the words.

Figure 3 gives a visualization of code balance for different settings. Each cell shows the percentage of a specific subcode, thus the values in each column sums to one. We found that the learned codes for a tight code decomposition is unbalanced for some components. However, when a large code decomposition such a 32×32 is used, the problem is largely mitigated.

7 CONCLUSION

REFERENCES

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.

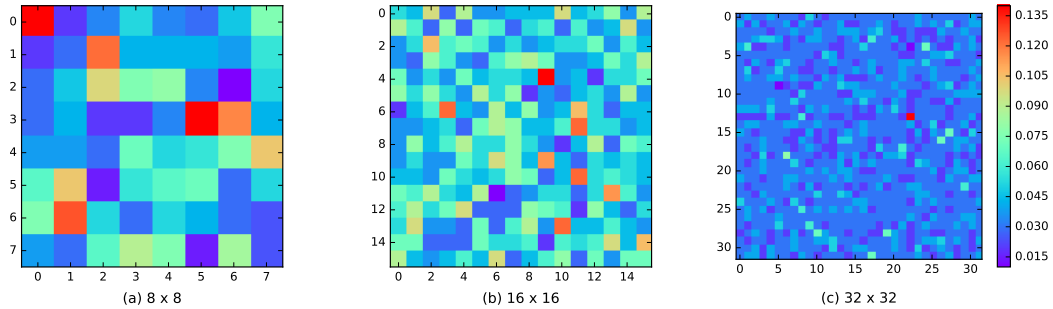

Figure 3: Visualization of code balance for different settings. Each cell in the heat map shows the percentage of a specific subcode. The results indicates the codes are evenly assigned to the words in the vocabulary when a large code decomposition (e.g., 32×32) is used.

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

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

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.

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.

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

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.