Learning K-way D-dimensional Discrete Codes for Compact Embedding Representations

Ting Chen [1] **Martin Renqiang Min** [2] **Yizhou Sun** [1]

Abstract

Conventional embedding methods directly associate each symbol with a continuous embedding vector, which is equivalent to applying a linear transformation based on a "one-hot" encoding of the discrete symbols. Despite its simplicity, such approach yields the number of parameters that grows linearly with the vocabulary size and can lead to overfitting. In this work, we propose a much more compact K-way D-dimensional discrete encoding scheme to replace the "one-hot" encoding. In the proposed "KD encoding", each symbol is represented by a D-dimensional code with a cardinality of K, and the final symbol embedding vector is generated by composing the code embedding vectors. To end-to-end learn semantically meaningful codes, we derive a relaxed discrete optimization approach based on stochastic gradient descent, which can be generally applied to any differentiable computational graph with an embedding layer. In our experiments with various applications from natural language processing to graph convolutional networks, the total size of the embedding layer can be reduced up to 98% while achieving similar or better performance.

1. Introduction

Embedding methods, such as word embedding (Mikolov et al., 2013; Pennington et al., 2014), have become pillars in many applications when learning from discrete structures. The examples include language modeling (Kim et al., 2016), machine translation (Sennrich et al., 2015), text classification (Zhang et al., 2015b), knowledge graph and social network modeling (Bordes et al., 2013; Chen &

Sun, 2017), and many others (Kipf & Welling, 2016; Chen et al., 2016). The objective of the embedding module in neural networks is to represent a discrete symbol, such as a word or an entity, with some continuous embedding vector $v \in R^d$. This seems to be a trivial problem, at the first glance, in which we can directly associate each symbol with a learnable embedding vector, as is done in existing work. To retrieve the embedding vector of a specific symbol, an embedding table lookup operation can be performed. This is equivalent to the following: first we encode each symbol with an "one-hot" encoding vector $b \in [0,1]^N$ where $\sum_j b_j = 1$ (N is the total number of symbols), and then generate the embedding vector v by simply multiplying the "one-hot" vector b with the embedding matrix $W \in R^{N \times d}$, i.e. $v = W^T b$.

Despite the simplicity of this "one-hot" encoding based embedding approach, it has several issues. The major issue is that the number of parameters grows linearly with the number of symbols. This becomes very challenging when we have millions or billions of entities in the database, or when there are lots of symbols with only a few observations (e.g. Zipf's law). There also exists redundancy in the $O(N)$ parameterization, considering that many symbols are actually similar to each other. This over-parameterization can further lead to overfitting; and it also requires a lot of memory, which prevents the model from being deployed to mobile devices. Another issue is purely from the code space utilization perspective, where we find "one-hot" encoding is extremely inefficient. Its code space utilization rate is almost zero as $N/2^N \to 0$ when $N \to \infty$, while N dimensional discrete coding system can effectively represent 2^N symbols.

To address these issues, we propose a novel and much more compact coding scheme that replaces the "one-hot" encoding. In the proposed approach, we use a K-way D-dimensional code to represent each symbol, where each code has D dimensions, and each dimension has a cardinality of K. For example, a concept of cat may be encoded as (5-1-3-7), and a concept of dog may be encoded as (5-1-3-9). The code allocation for each symbol is based on data and specific tasks such that the codes can capture semantics of symbols, and *similar codes* should reflect *similar mean-*

---

[1]Department of Computer Science, University of California, Los Angeles [2]NEC Laboratories America. Correspondence to: Ting Chen <tingchen@cs.ucla.edu>.

Proceedings of the 35th International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018. Copyright 2018 by the author(s).

ings. While we mainly focus on the encoding of symbols in this work, the learned discrete codes can have larger applications, such as information retrieval. We dub the proposed encoding scheme as "*KD encoding*".

The KD code system is much more compact than its "one-hot" counterpart. To represent a set of symbols of size N, the "KD encoding" only requires $K^D \geq N$. Increasing K or D by a small amount, we can easily achieve $K^D \gg N$, in which case it will still be much more compact and keep $D = O(\frac{\log N}{\log K})$. Consider $K = 2$, the utilization rate of "KD encoding" is $N/2^D$, which is 2^{N-D} times more compact than its "one-hot" counterpart[1].

The compactness of the code can be translated into compactness of the parametrization. Dropping the giant embedding matrix $W \in R^{N \times d}$ that stores symbol embeddings and leveraging semantic similarities between symbols, the symbol embedding vector is generated by composing much fewer code embedding vectors. This can be achieved as follows: first we embed each "KD code" into a sequence of code embedding vectors in $R^{D \times d'}$, and then apply embedding transformation function $\boldsymbol{f}(\cdot)$ to generate the final symbol embedding. By adopting the new approach, we can reduce the number of embedding parameters from $O(Nd)$ to $O(\frac{K}{\log K} d' \log N + C)$, where d' is the code embedding size, and C is the number of neural network parameters.

Due to the the discreteness of the code allocation problem, it is very challenging to learn the meaningful discrete codes that can exploit the similarities among symbols according to a target task in an end-to-end fashion. A compromise is to learn the code given a trained embedding matrix, and then fix the code during the stage of task-specific training. While this has been shown working relatively well in previous work (Chen et al., 2017; Shu & Nakayama, 2017), it produces a sub-optimal solution, and requires a multi-stage procedure that is hard to tune. In this work, we derive a relaxed discrete optimization approach based on stochastic gradient descent (SGD), and propose two guided methods to assist the end-to-end code learning. To validate our idea, we conduct experiments on three different tasks from natural language processing to graph convolutional networks for semi-supervised node classification. We achieve 95% of embedding model size reduction in the language modeling task and 98% in text classification with similar or better performance.

2. The K-way D-dimensional Discrete Encoding Framework

In this section, we introduce the "KD encoding" framework in details.

2.1. Problem Formulation

Symbols are represented with a vocabulary $V = \{s_1, s_2, \cdots, s_N\}$ where s_i corresponds to the i-th symbol. Here we aim to learn a transformation function that maps a symbol s_i to a continuous embedding vector $\boldsymbol{v}_i$, i.e. $\mathcal{T} : V \to \mathbb{R}^d$. In the case of conventional embedding method, $\mathcal{T}$ is a linear transformation of "one-hot" code of a symbol.

To measure the fitness of $\mathcal{T}$, we consider a differentiable computational graph $\mathcal{G}$ that takes discrete symbols as input $\boldsymbol{x}$ and outputs the predictions $\boldsymbol{y}$, such as text classification model based on word embeddings. We also assume a task-specific loss function $\mathcal{L}(\boldsymbol{x}, \boldsymbol{y})$ is given. The task-oriented learning of $\mathcal{T}$ is to learn $\mathcal{T}$ such that $\mathcal{L}(\boldsymbol{x}, \boldsymbol{y})$ is minimized, i.e. $\mathcal{T} = \arg\min_{\mathcal{T}} \mathcal{L}(\boldsymbol{x}, \boldsymbol{y} | \mathcal{T}, \Theta)$ where Θ are task-specific parameters.

2.2. The "KD Encoding" Framework

In the proposed framework, each symbol is associated with a K-way D-dimensional discrete code. We denote the discrete code for the i-th symbol as $\boldsymbol{c}_i = (c_i^1, c_i^2, \cdots, c_i^D) \in \mathcal{B}^D$, where $\mathcal{B}$ is the set of code bits with cardinality K. To connect symbols with discrete codes, a code allocation function $\boldsymbol{\phi}(\cdot) : V \to \mathcal{B}^D$ is used. The learning of this mapping function will be introduced later, and once fixed it can be stored as a hash table for fast lookup. Since a discrete code $\boldsymbol{c}_i$ has D dimensions, we do not directly use embedding lookup to find the symbol embedding as used in "one-hot" encoding. Instead, we want to learn an adaptive code composition function that takes a discrete code and generates a continuous embedding vector, i.e. $\boldsymbol{f} : \mathcal{B}^D \to R^d$. The details of $\boldsymbol{f}$ will be introduced in the next subsection. In sum, the "KD encoding" framework we have $\mathcal{T} = \boldsymbol{f} \circ \boldsymbol{\phi}$ with a "KD code" allocation function $\boldsymbol{\phi}$ and a composition function $\boldsymbol{f}$ as illustrated in Figure 1(a) and 1(b).

In order to uniquely identify every symbol, we only need to set $K^D = N$, as we can assign a unique code to each symbol in this case. When this holds, the code space is fully utilized, and none of the symbol can change its code without affecting other symbols. We call this type of code system *compact code*. The optimization problem for compact code can be very difficult, and usually requires approximated combinatorial algorithms such as graph matching (Li et al., 2016). Realizing the difficulties in optimization, we propose to adopt the *redundant code* system, where $K^D \gg N$, namely, there are a lot of "empty" codes with no symbol associated. Changing the code of one symbol may not affect other symbols under this scheme, since the random collision probability can be very small [2], which

---

[1] Assuming we have vocabulary size $N = 10,000$ and the dimensionality $D = 100$, it is 2^{9900} times more efficient.

[2] For example, we can set $K = 100, D = 10$ for a billion symbols, in a random code assignment, the probability of the NO
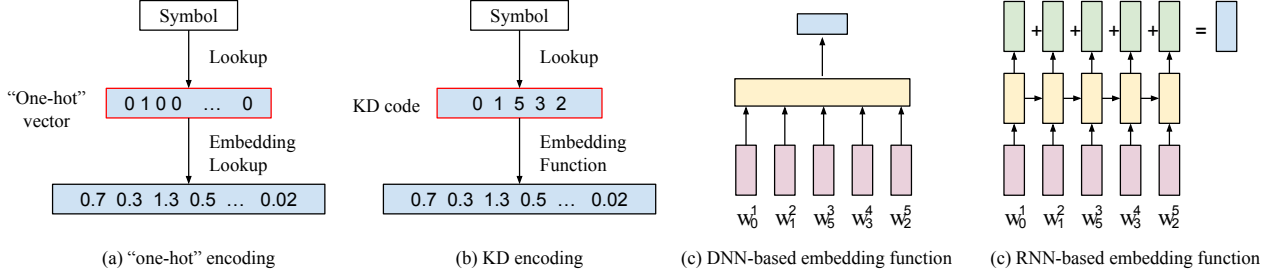
Figure 1. (a) The conventional symbol embedding based on "one-hot" encoding. (b) The proposed KD encoding scheme. (c) and (d) are examples of embedding transformation functions by DNN and RNN used in the "KD encoding" when generating the symbol embedding from its code.

makes it easier to optimize. The redundant code can be achieved by slightly increasing the size of K or D thanks to the exponential nature of their relations to N. Therefore, in both compact code or redundant code, it only requires $D = O(\frac{\log N}{\log K})$.

2.3. Discrete Code Embedding

As mentioned above, given learned $\phi(\cdot)$ and the i-th symbol s_i, we can retrieve its code via a code lookup, i.e. $c_i = \phi(s_i)$. In order to generate the composite embedding vector v_i, we adopt an adaptive code composition function $v_i = f(c_i)$. To do so, we first embed the code c_i to a sequence of code embedding vectors $(\mathcal{W}^1_{c^1_i}, \mathcal{W}^2_{c^2_i}, \cdots, \mathcal{W}^D_{c^D_i})$, and then apply another transformation $v = f_e(\mathcal{W}^1_{c^1_i}, \mathcal{W}^2_{c^2_i}, \cdots, \mathcal{W}^D_{c^D_i}; \theta_e)$ to generate v. Here $\mathcal{W}^j \in R^{K \times d'}$ is the code embedding matrix for the j-th code dimension, and f_e is the *embedding transformation function* that maps the code embedding vectors to the symbol embedding vector. The choice of f_e is very flexible and varies from task to task. In this work, we consider two types of embedding transformation functions.

The first one is based on a linear transformation:

$$v_i = H \left(\sum_j \mathcal{W}^j_{c^j_i} \right)^T,$$

where $H \in R^{d \times d'}$ is a transformation matrix for matching the dimensions. While this is simple and efficient, due to its linear nature, the capacity of the generated symbol embedding may be limited when the size of K, D or the code embedding dimension d' is small.

Another type of embedding transformation functions are nonlinear, and here we introduce one that is based on a recurrent neural network, LSTM (Hochreiter & Schmidhuber, 1997), in particular. That is, we have $(h_1, \cdots, h_j) = $ LSTM$(\mathcal{W}^1_{c^1}, \cdots, \mathcal{W}^j_{c^j})$ (see supplementary for details).

---

collision at all is 99.5%.

The final symbol embedding can be computed by summing over LSTM outputs at all code dimensions (and using a linear layer to match dimension if $d \neq d'$), i.e. $v = H(\sum_j h_j)^T$. Figure 1(c) and 1(d) illustrate the above two embedding transformation functions.

2.4. Analysis of the Proposed "KD Encoding"

To measure the parameter and model size reduction, we first introduce two definitions as follows.

Definition 1. *(Embedding parameters) The embedding parameters are the parameters θ that are used in code composition function f. Specifically, it includes code embedding matrices $\{\mathcal{W}\}$, as well as other parameters θ_e used in the embedding transformation function f_e.*

It is worth noting that we do not explicitly include the code as embedding parameters. This is due to the fact that we do not count "one-hot" codes as parameters. Also in some cases the codes are not adaptively learned, such as hashed from symbols (Svenstrup et al., 2017). However, when we export the model to embedded devices, the storage of discrete codes does occupy space. Hence, we introduce another concept below to take it into consideration as well.

Definition 2. *(Embedding layer's size) The embedding layer's size is the number of bits used to store both embedding parameters as well as the discrete codes.*

Lemma 1. *The number of embedding parameters used in KD encoding is $O(\frac{K}{\log K} d' \log N + C)$, where C is the number of parameters of neural nets.*

The proof is given in the supplementary material.

For the analysis of the embedding layer's size under "KD encoding", we assume that 32-bits floating point number is used. The total bits used by the "KD encoding" is $ND \log_2 K + 32(KDd' + C)$ consisting both code size as well as the size of embedding parameters. Comparing to the total model size by conventional full embedding, which is $32N(1 + d)$, it can still be a huge saving of model space,

especially when N, d are large.

Here we provide a theoretical connection between the proposed "KD encoding" and the SVD or low-rank factorization of the embedding matrix. We consider the scenario where the composition function $\boldsymbol{f}$ is a linear function with no hidden layer, that is $\boldsymbol{v}_i = (\sum_j \mathcal{W}^j_{\boldsymbol{c}^j_i})^T$.

Proposition 1. *A linear composition function $\boldsymbol{f}$ with no hidden layer is equivalent to a sparse binary low-rank factorization of the embedding matrix.*

The proof is also provided in the supplementary material. But the overall idea is that the "KD code" mimics an 1-out-of-K selection within each of the D groups.

The computation overhead brought by linear composition is very small compared to the downstream neural network computation (without hidden layer in linear composition function, we only need to sum up D vectors). However, the expressiveness of the linear factorization is limited by the number of bases or rank of the factorization, which is determined by K and D. And the use of non-linear composition function can largely increase the expressiveness of the composite embedding matrix and may be an appealing alternative, this is shown by the proposition 2 in supplementary.

3. End-to-End Learning of the Discrete Code

In this section, we propose methods for learning task-specific "KD codes".

3.1. Continuous Relaxation for Discrete Code Learning

As mentioned before, we want to learn the symbol-to-embedding-vector mapping function, $\mathcal{T}$, to minimize the target task loss, i.e. $\mathcal{T} = \arg\min_{\mathcal{T}} \mathcal{L}(\boldsymbol{x}, \boldsymbol{y}|\mathcal{T}, \Theta)$. This includes optimizing both code allocation function $\boldsymbol{\phi}(\cdot)$ and code composition function $\boldsymbol{f}(\cdot)$. While $\boldsymbol{f}(\cdot)$ is differentiable w.r.t. its parameters θ, $\boldsymbol{\phi}(\cdot)$ is very challenging to learn due to the discreteness and non-differentiability of the codes.

Specifically, we are interested in solving the following optimization problem,

$$\min_{\{\boldsymbol{c}\}, \theta, \Theta} \sum_i \mathcal{L}\left(\boldsymbol{x}_i, \boldsymbol{y}_i | \boldsymbol{f}_e\left(\mathcal{W}^1_{\boldsymbol{c}^1_i}, \mathcal{W}^2_{\boldsymbol{c}^2_i}, \cdots, \mathcal{W}^D_{\boldsymbol{c}^D_i}\right), \Theta\right) \quad (1)$$

where $\boldsymbol{f}_e$ is the embedding transformation function mapping code embedding to the symbol embedding, $\theta = \{\mathcal{W}, \theta_e\}$ contains code embeddings and the composition parameters, and Θ denotes other task-specific parameters.

We assume the above loss function is differentiable w.r.t. to the continuous parameters including embedding parameters θ and other task-specific parameters Θ, so they can
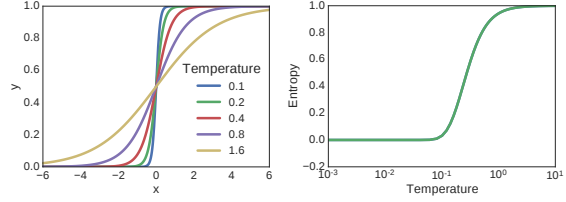
(a) The output probabilities.　　(b) The entropy.

Figure 2. The effects of temperature τ on output probability of Softmax and its entropy (when $K = 2$). As τ decreases, the probabilistic output approximates step function when $K = 2$, and generally "one-hot" vector when $K > 2$.

be optimized by following standard stochastic gradient descent and its variants (Kingma & Ba, 2014). However, each $\boldsymbol{c}_i$ is a discrete code, it cannot be directly optimized via SGD as other parameters. In order to adopt gradient based approach to simplify the learning of discrete codes in an end-to-end fashion, we derive a continuous relaxation of the discrete code to approximate the gradient effectively.

We start by making the observation that each code $\boldsymbol{c}_i$ can be seen as a concatenation of D "one-hot" vectors, i.e. $\boldsymbol{c}_i = (\boldsymbol{o}^1_i, \boldsymbol{o}^2_i, \cdots, \boldsymbol{o}^D_i)$, where $\forall j, \boldsymbol{o}^j_i \in [0, 1]^K$ and $\sum_k \boldsymbol{o}^{jk}_i = 1$, where $\boldsymbol{o}^{jk}_i$ is the k-th component of $\boldsymbol{o}^j_i$. To make it differentiable, we relax the $\boldsymbol{o}_i$ from a "one-hot" vector to a continuous relaxed vector $\hat{\boldsymbol{o}}_i$ by applying *tempering Softmax*:

$$\boldsymbol{o}^{jk}_i \approx \hat{\boldsymbol{o}}^{jk}_i = \frac{\exp(\boldsymbol{\pi}^{jk}_i/\tau)}{\sum_{k'} \exp(\boldsymbol{\pi}^{jk'}_i/\tau)}$$

Where τ is a temperature term, as $\tau \to 0$, this approximation becomes exact (except for the case of ties). We show this approximation effects for $K = 2$ with $y = 1/(1 + \exp(-x/\tau))$ in Figure 2a. Similar techniques have been introduced in Gumbel-Softmax trick (Jang et al., 2016; Maddison et al., 2016).

Since $\hat{\boldsymbol{o}}_i$ is continuous (given τ is not approaching 0), instead of learning the discrete code assignment directly, we learn $\hat{\boldsymbol{o}}_i$ as an approximation to $\boldsymbol{o}_i$. To do so, we can adjust the code logits $\boldsymbol{\pi}_i$ using SGD and gradually decrease the temperature τ during the training. Since the indexing operator for retrieval of code embedding vectors, i.e. $\mathcal{W}^j_{\boldsymbol{c}^j_i}$, is non-differentiable, to generate the embedding vector for j-th code dimension, we instead use an affine transformation operator, i.e. $(\mathcal{W}^j)^T \hat{\boldsymbol{o}}^j_i$, which enables the gradient to flow backwards normally.

It is easy to see that control of temperature τ can be important. When τ is too large, the output $\hat{\boldsymbol{o}}_i$ is close to uniform, which is too far away from the desired "one-hot" vector $\boldsymbol{o}_i$. When τ is too small, the slight differences between different logits $\boldsymbol{\pi}^j_i$ and $\boldsymbol{\pi}^{j'}_i$ will be largely magnified. Also,

the gradient vanishes when the Softmax output approaches "one-hot" vector, i.e. when it is too confident. A "right" schedule of temperature can thus be crucial. While we can handcraft a good schedule of temperature, we also observe that the temperature τ is closely related to the entropy of the output probabilistic vector, as shown in Figure 2b, where a same set of random logits can produce probabilities of different entropies when τ varies. This motivates us to implicitly control the temperature via regularizing the entropy of the model. To do so, we add the following entropy regularization term: $\mathbb{H} = -\sum_{i,j,k} \hat{o}_i^{jk} \log \hat{o}_i^{jk}$. A large penalty for this regularization term encourages a small entropy for the relaxed codes, i.e. a more spiky distribution.

Up to this point, we still use the continuous relaxation $\hat{\boldsymbol{o}}_i$ to approximate $\boldsymbol{o}_i$ during the training. In inference, we will only use discrete codes. The discrepancy of the continuous and discrete codes used in training and inference is undesirable. To close the gap, we take inspiration from Straight-Through Estimator (Bengio et al., 2013). In the forward pass, instead of using the relaxed tempering Softmax output $\hat{\boldsymbol{o}}_i$, which is likely a smooth continuous vector, we take its $\arg\max$ and turn it into a "one-hot" vector as follows, which recovers a discrete code.

$$\boldsymbol{o}_i^j = \text{one\_hot}\left(\arg\max_k \hat{o}_i^{jk}\right) \approx \text{Softmax}\left(\frac{\boldsymbol{\pi}_i^j}{\tau}\right), \quad \tau \to 0$$

We interpret the use of straight-through estimator as using different temperatures during the forward and backward pass. In forward pass, $\tau \to 0$ is used, for which we simply apply the $\arg\max$ operator. In the backward pass (to compute the gradient), it pretends that a larger τ was used. Compared to using the same temperature in both passes, this always outputs "one-hot" discrete code $\boldsymbol{o}_i^j$, which closes the previous gap between training and inference.

The training procedure is summarized in Algorithm 1, in which the `stop_gradient` operator will prevent the gradient from back-propagating through it.

3.2. Code Learning with Guidances

It is not surprising the optimization problem is more challenging for learning discrete codes than learning conventional continuous embedding vectors, due to the discreteness of the problem (which can be NP-hard). This could lead to a suboptimal solution where discrete codes are not as competitive. Therefore, we propose to use guidances from the continuous embedding vectors to mitigate the problem. The basic idea is that instead of adjusting codes according to noisy gradients from the end task as shown above, we also require the composite embedding vectors from codes to mimic continuous embedding vectors, which can be either jointly trained (online distillation guidance),

---

Algorithm 1 An epoch of code learning via Straight-through Estimator with Tempering Softmax.

Parameters: code logits $\{\boldsymbol{\pi}_i\}$, code embedding matrices $\{\mathcal{W}^j\}$, transformation parameters θ_e, and other task specific parameters Θ.

for $i \leftarrow 1$ **to** N **do**
  for $j \leftarrow 1$ **to** D **do**
    $\hat{\boldsymbol{o}}_i^j = \text{Softmax}(\boldsymbol{\pi}_i^j / \tau)$
    $\boldsymbol{o}_i^j = \text{one\_hot}(\arg\max_k \hat{o}_i^{jk})$
    $\boldsymbol{o}_i^j = \texttt{stop\_gradient}(\boldsymbol{o}_i^j - \hat{\boldsymbol{o}}_i^j) + \hat{\boldsymbol{o}}_i^j$
  end for
  A step of SGD on $\boldsymbol{\pi}_i, \{\mathcal{W}^j\}, \theta_e, \Theta$ to reduce
  $\mathcal{L}\left(\boldsymbol{x}_i, \boldsymbol{y}_i, \boldsymbol{f}_e\left((\boldsymbol{o}_i^1)^T \mathcal{W}^1, \cdots, (\boldsymbol{o}_i^D)^T \mathcal{W}^D; \theta_e\right), \Theta\right)$
end for

---

or pre-trained (pre-train distillation guidance). The continuous embedding can provide better signals for both code learning as well as the rest parts of the neural network, improve the training subsequently.

Online Distillation Guidance (ODG). A good learning progress in code allocation function $\phi(\cdot)$ can be important for the rest of the neural network to learn. For example, it is hard to imagine we can train a good model based on "KD codes" if we have $\phi(\text{"table"}) = \phi(\text{"cat"})$. However, the learning of the $\phi(\cdot)$ also depends on the rest of network to provide good signals.

Based on the observation, we propose to associate a regular continuous embedding vector $\boldsymbol{u}_i$ with each symbol during the training, and we want the "KD encoding" function $\mathcal{T}(\cdot)$ to mimic the continuous embedding vectors, while both of them are simultaneously optimized for the end task. More specifically, during the training, instead of using the embedding vector generated from the code, i.e. $\boldsymbol{f}(\boldsymbol{c}_i)$, we use a dropout average of them, i.e.

$$\boldsymbol{v}_i = m \odot \boldsymbol{u}_i + (1 - m) \odot \boldsymbol{f}(\boldsymbol{c}_i).$$

Here m is a Bernoulli random variable for selecting between the regular embedding vector or the KD embedding vector. When m is turned on with a relatively high probability (e.g. 0.7), even if $\boldsymbol{f}(\boldsymbol{c}_i)$ is difficult to learn, $\boldsymbol{u}_i$ can still be learned to assist the improvement of the task-specific parameters Θ, which in turn helps code learning. During the inference, we only use $\boldsymbol{f}(\boldsymbol{c}_i)$ as output embedding. This choice can lead to a gap between training and generalization errors. Hence, we add a regularization loss $\lambda \|\boldsymbol{u}_i - \boldsymbol{f}(\boldsymbol{c}_i)\|^2$ during the training that encourages the match between $\boldsymbol{u}_i$ and $\boldsymbol{f}(\boldsymbol{c}_i)$[3].

---

[3]Here we use stop_gradient($\boldsymbol{u}_i$) to prevent embedding vectors $\boldsymbol{u}$ being dragged to $\boldsymbol{f}(\boldsymbol{c}_i)$ as it has too much freedom.
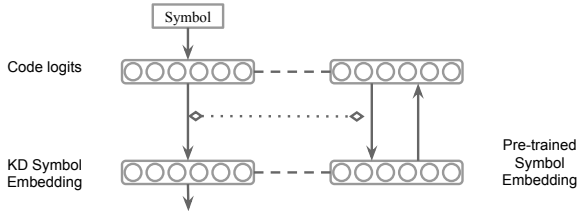
Figure 3. Online Distillation Guidance. Dashed lines denotes regularization, doted line in the middle denotes sharing of transformation function.

Pre-trained Distillation Guidance (PDG). It is important to close the gap between training and inference in the online distillation guidance process, unfortunately this can still be difficult. Alternatively, we can also adopt pre-trained continuous embedding vectors as guidance. Instead of training the continuous embedding vectors alongside the discrete codes, we utilize a pre-trained continuous embedding matrix $\mathcal{U}$ produced from the same model with conventional continuous embedding vectors. During the end-to-end training of the codes (as well as other parameters), we ask the composite embedding vector $\boldsymbol{v}_i$ generated by "KD encoding" to mimic the the given embedding vector $\boldsymbol{u}_i$ by minimizing the l_2 distance.

Furthermore, we can build an auto-encoder of the pre-trained continuous embedding vectors, and use both continuous embedding vectors as well as the code logits as guidances. In the encoding pass, a transformation function $\boldsymbol{g}(\cdot)$ is used to map $\boldsymbol{u}_i$ to the code logits $\boldsymbol{\pi}_i$. In its decoding pass, it utilizes the same transformation function $\boldsymbol{f}(\cdot)$ that is used in "KD encoding" to reconstruct $\boldsymbol{u}_i$. The loss function for the auto-encoders is

$$\mathcal{L}_{auto-encoder} = \sum_i \|\boldsymbol{f}(\boldsymbol{g}(\boldsymbol{u}_i); \tau) - \boldsymbol{u}_i\|^2$$

To follow the guidance of the pre-trained embedding matrix $\mathcal{U}$, we ask the code logits $\boldsymbol{\pi}_i$ and composite symbol embedding $\boldsymbol{v}_i = \boldsymbol{f}(\boldsymbol{\pi}_i; \tau)$ [4] to mimic the ones in the auto-encoder as follows

$$\mathcal{L}_{distillation} = \sum_i \alpha\|\boldsymbol{f}(\boldsymbol{\pi}_i; \tau) - \boldsymbol{u}_i\|^2 + \beta\|\boldsymbol{\pi}_i - \boldsymbol{g}(\boldsymbol{u}_i)\|^2$$

During the training, both $\mathcal{L}_{auto-encoder}$ and $\mathcal{L}_{distillation}$ will be added to the task-specific loss function to train jointly. The method is illustrated in the Figure 3.

Here we also make a distinction between pre-trained distillation guidance (PDG) and pre-training of codes. Firstly, PDG can learn codes end-to-end to optimize the task's loss, while the pre-trained codes will be fixed during the task

[4]Here we overload the function $\boldsymbol{f}(c_i)$ by considering that code c_i can be turned into "one-hot" $\boldsymbol{o}_i$, and $\boldsymbol{o}_i \approx Softmax(\boldsymbol{\pi}_i/\tau)$.

learning. Secondly, the PDG training procedure is much easier, especially for the tuning of discrete code learning, while pre-training of codes requires three stages and is unfriendly for parameter tuning.

4. Experiments

In this section, we conduct experiments to validate the proposed approach. Since the proposed "KD Encoding" can be applied to various tasks and applications with embedding layers involved. We choose three important tasks for evaluation, they are (1) language modeling, (2) text classification, and (3) graph convolutional networks for semi-supervised node classification. For the detailed descriptions of these tasks and other applications of our method, we refer readers to the supplementary material.

For the language modeling task, we test on the widely used English Penn Treebank (Marcus et al., 1993) dataset, which contains 1M words with vocabulary size of 10K. The training/validation/test split is provided by convention according to (Mikolov et al., 2010). Since we only focus on the embedding layer, we simply adopt a previous state-of-the-art model (Zaremba et al., 2014), in which they provide three different variants of LSTMs (Hochreiter & Schmidhuber, 1997) of different sizes: The larger model has word embedding size and LSTM hidden size of 1500, while the number is 650 and 200 for the medium and small models. By default, we use $K = 32, D = 32$ and pre-trained distillation guidance for the proposed method, and linear embedding transformation function with 1 hidden layer of 300 hidden units.

For the text classification task, we utilize five different datasets from (Zhang et al., 2015b), namely Yahoo! news, AG's news, DBpedia, Yelp review polarity ratings as well Yelp review full-scale ratings [5]. We adopt network architecture used in FastText (Joulin et al., 2016b;a), where a SoftMax is stacked on top of the averaged word embedding vectors of the text. For simplicity, we only use unigram word information but not sub-words or bi-grams, as used in their work. The word embedding dimension is chosen to be 300 as it yields a good balance between size and performance. By default, we use $K = 32, D = 32$ for the proposed method, and linear transformation with no hidden layer. That is to add code embedding vectors together to generate symbol embedding vector, and the dimension of code embedding is the same as word embedding.

For the application with graph convolutional networks, we follow the same setting and hyper-parameters as in (Kipf & Welling, 2016). Three datasets are used for comparison,

[5]YahooAnswers has 477K unique words and 131M tokens, and Yelp has 268K unique words and 94M tokens. More details available in (Zhang et al., 2015b).

Table 1. Language modeling (PTB). Compared with Conventional full embedding, and low-rank (denoted with Lr) with different compression rates.

	Model	Full	Lr(5X)	Lr(10X)	Ours
Perplexity	Small	114.53	134.01	134.89	**107.77**
	Medi.	83.38	84.84	85.53	**83.11**
	Large	78.71	81.23	81.85	**77.72**
# of emb. params. (M)	Small	2.00	0.40	**0.19**	0.37
	Medi.	6.50	1.30	0.65	**0.50**
	Large	15.00	2.99	1.50	**0.76**
# of bits (M)	Small	64.00	12.73	**6.20**	13.39
	Medi.	208.00	41.58	20.79	**17.75**
	Large	480.00	95.68	47.84	**26.00**

Table 2. Text classification. Lr denotes low-rank.

	Model	Full	Lr(10X)	Lr(20X)	Ours
Accuracy	Yahoo!	**0.698**	0.695	0.691	0.695
	AG N.	0.914	0.914	0.915	**0.916**
	Yelp P.	**0.932**	0.924	0.923	0.931
	Yelp F.	**0.592**	0.578	0.573	0.590
	DBpedia	0.977	0.977	0.979	**0.980**
# of emb. params. (M)	Yahoo!	143.26	13.857	6.690	**0.308**
	AG N.	20.797	2.019	0.975	**0.308**
	Yelp P.	74.022	7.164	3.459	**0.308**
	Yelp F.	80.524	7.793	3.762	**0.308**
	DBpedia	183.76	17.772	8.580	**0.308**
# of bits (G)	Yahoo!	4.584	0.443	0.214	**0.086**
	AG N.	0.665	0.065	0.031	**0.021**
	Yelp P.	2.369	0.229	0.111	**0.049**
	Yelp F.	2.577	0.249	0.120	**0.053**
	DBpedia	5.880	0.569	0.275	**0.108**

Table 3. Graph Convolutional Networks. Lr denotes low-rank.

	Dataset	Full	Lr(2X)	Lr(4X)	Ours
Accuracy	Cora	0.814	0.789	0.767	**0.823**
	Citese.	0.721	0.710	0.685	**0.723**
	Pubm.	0.795	0.773	0.780	**0.797**
# of emb. params. (K)	Cora	22.93	10.14	**5.8**	8.22
	Citese.	59.25	26.03	14.88	**8.22**
	Pubm.	8.00	3.61	2.06	2.69
# of bits (M)	Cora	0.73	0.32	**0.19**	0.33
	Citese.	1.90	0.83	0.48	**0.44**
	Pubm.	0.26	0.12	**0.07**	0.10

Table 4. Comparisons with more baselines in Language Modeling (Medium sized model).

Methods	PPL	Bits saved
Char-as-codes	108.14	96%
Scalar quantization (8 bits)	84.06	75%
Scalar quantization (6 bits)	87.73	81%
Scalar quantization (4 bits)	92.86	88%
Product quantization(64x325)	84.03	88%
Product quantization(128x325)	83.71	85%
Product quantization(256x325)	83.66	81%
Ours	83.11	92%

ding parameter and model size, 98% in the text classification case. In order to achieve similar level of compression, we note that low-rank factorization baseline will reduce the performance significantly.

We further compare with broader baselines on language modeling tasks (with medium sized language model for convenience): (1) directly using first 10 chars of a word as its code (padding when necessary), (2) training aware quantization (Jacob et al., 2017), and (3) product quantization (Jegou et al., 2011; Joulin et al., 2016a). The results are shown in Table 4. We can see that our methods significantly outperform these baselines, in terms of both PPL as well as model size (bits) reduction.

In the following, we scrutinize different components of the proposed model based on PTB language modeling. To start with, we test various code learning methods, and demonstrate the impact of training with guidance. The results are shown in Table 5. First, we note that both random codes as well as pre-trained codes are suboptimal, which is understandable as they are not (fully) adaptive to the target tasks. Then, we see that end-to-end training without guidance suffers serious performance loss, especially when the task specific networks increase its complexity (with larger hidden size and use of dropout). Finally, by adopting the proposed continuous guidances (especially distillation guidance), the performance loss can be overcame. We further vary the size of K or D and see how they affect the performance. As shown in Figure 4a and 4b, small K or D may harm the performance (even though that $K^D \gg N$ is satisfied),

namely Cora, Citeseer, Pubmed. Since both the number of symbols (1433, 3703, and 500 respectively) as well as its embedding dimension (16) are small, the compressible space is actually quite small. Nevertheless, we perform the proposed method with $K = 64, D = 8$ for Cora and Citeseer, and $K = 32, D = 4$ for Pubmed. Again, a linear embedding transformation function is used with one hidden layer of size 16. We do not use guidances for text classification and graph node classification tasks since the direct optimization is already satisfying enough.

We mainly compare the proposed "KD encoding" approach with the conventional continuous (full) embedding counterpart, and also compare with low-rank factorization (Sainath et al., 2013) with different compression ratios. The results for three tasks are shown in Table 1, 2, 3, respectively. In these tables, three types of metrics are shown: (1) the performance metric, perplexity for language modeling and accuracy for the others, (2) the number of embedding parameters θ used in f, and (3) the total embedding layer's size includes θ as well as the codes. From these tables, we observe that the proposed "KD encoding" with end-to-end code learning perform similarly, or even better in many cases, while consistently saving more than 90% of embe-

Table 5. Comparisons of different code learning methods.

	Small	Medium	Large
Full embedding	114.53	83.38	78.71
Random code	115.79	104.12	98.38
Pre-trained code	107.95	84.92	80.69
Ours (no guidance)	108.50	89.03	86.41
Ours (ODG)	108.19	85.50	83.00
Ours (PDG)	**107.77**	**83.11**	**77.72**

(a) Linear instantiation. (b) RNN instantiation.
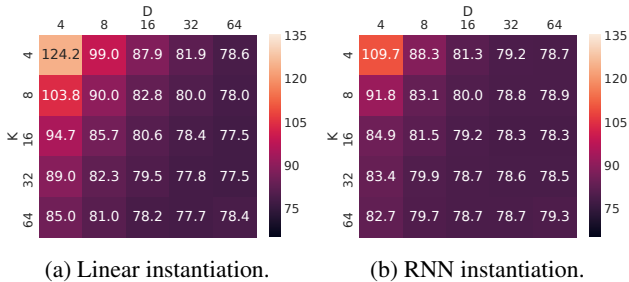
Figure 4. The effects of various K and D under different instantiation of embedding transformation function $f(\cdot)$.

Table 6. Learned codes for 10K Glove embeddings (K=6, D=4).

Code	Words
3-1-0-3	up when over into time back off set left open half behind quickly starts
3-1-0-4	week tuesday wednesday monday thursday friday sunday saturday
3-1-0-5	by were after before while past ago close soon recently continued meanwhile
3-1-1-1	year month months record fall annual target cuts

which suggests that the redundant code can be easier to learn. The size of D seems to have higher impact on the performance compared to K. Also, when D is small, non-linear encoder such as RNN performs much better than the linear counterpart, which verifies our Proposition 2. To examine the learned codes, we apply our method on the pre-trained embedding vectors from Glove (Pennington et al., 2014), which has better coverage and quality. We force the model to assign multiple words to the same code by setting $K = 6, D = 4$ (code space is 1296) for vocabulary size of 10K. Table 6 show a snippet of the learned codes, which shows that semantically similar words are assigned to the same or close-by discrete codes.

5. Related Work

The idea of using more efficient coding system traces to information theory, such as error correction code (Hamming, 1950), and Hoffman code (Huffman, 1952). However, in most embedding techniques such as word embedding (Mikolov et al., 2013; Pennington et al., 2014), entity embedding (Chen et al., 2016; Chen & Sun, 2017),

"one-hot" encoding is used along with a usually large embedding matrix. Recent work (Kim et al., 2016; Sennrich et al., 2015; Zhang et al., 2015b) explores character or sub-word based embedding model instead of the word embedding model and show some promising results. (Svenstrup et al., 2017) proposes using hash functions to automatically map texts to pre-defined bases with a smaller vocabulary size, according to which vectors are composed. However, in their cases, the chars, sub-words and hash functions are fixed and given a priori dependent on language, thus may have few semantic meanings attached and may not be available for other type of data. In contrast, we learn the code assignment function from data and tasks, and our method is language independent.

The compression of neural networks (Han et al., 2015a;b; Chen et al., 2015) has become more and more important in order to deploy large networks to small mobile devices. Our work can be seen as a way to compress the embedding layer in neural networks. Most existing network compression techniques focus on dense/convolutional layers that are shared/amortized by all data instances, while one data instance only utilizes a fraction of embedding layer weights associated with the given symbols. To compress these types of weights, some efforts have been made, such as product quantization (Jegou et al., 2011; Joulin et al., 2016a; Zhang; Zhang et al., 2015a; Babenko & Lempitsky, 2014). Compared to their methods, our framework is more general. Many of these methods can be seen as a special case of "KD encoding" using a linear embedding transformation function without hidden layer. Also, under our framework, both the codes and the transformation functions can be learned jointly by minimizing task-specific losses.

Our work is also related to LightRNN (Li et al., 2016), which can be seen as a special case of our proposed KD code with $K = \sqrt{N}$ and $D = 2$. Due to the use of a more compact code, its code learning is harder and more expensive. This work is an extension of our previous workshop paper (Chen et al., 2017) with guided end-to-end code learning. In parallel to (Chen et al., 2017), (Shu & Nakayama, 2017) explores similar ideas with linear composition functions and pre-trained codes.

6. Conclusions

In this paper, we propose a novel K-way D-dimensional discrete encoding scheme to replace the "one-hot" encoding, which significantly improves the efficiency of the parameterization of models with embedding layers. To learn semantically meaningful codes, we derive a relaxed discrete optimization technique based on SGD enabling end-to-end code learning. We demonstrate the effectiveness of our work with applications in language modeling, text classification and graph convolutional networks.

Acknowledgements

References

Babenko, A. and Lempitsky, V. Additive quantization for extreme vector compression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 931–938, 2014.

Bengio, Y., Léonard, N., and Courville, A. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

Bordes, A., Usunier, N., Garcia-Duran, A., Weston, J., and Yakhnenko, O. Translating embeddings for modeling multi-relational data. In *Advances in neural information processing systems*, pp. 2787–2795, 2013.

Chen, T. and Sun, Y. Task-guided and path-augmented heterogeneous network embedding for author identification. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pp. 295–304. ACM, 2017.

Chen, T., Tang, L.-A., Sun, Y., Chen, Z., and Zhang, K. Entity embedding-based anomaly detection for heterogeneous categorical events. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, pp. 1396–1403. AAAI Press, 2016.

Chen, T., Min, M. R., and Sun, Y. Learning k-way d-dimensional discrete code for compact embedding representations. *CoRR*, abs/1711.03067, 2017. URL http://arxiv.org/abs/1711.03067.

Chen, W., Wilson, J., Tyree, S., Weinberger, K., and Chen, Y. Compressing neural networks with the hashing trick. In *International Conference on Machine Learning*, pp. 2285–2294, 2015.

Hamming, R. W. Error detecting and error correcting codes. *Bell Labs Technical Journal*, 29(2):147–160, 1950.

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

Han, S., Pool, J., Tran, J., and Dally, W. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, pp. 1135–1143, 2015b.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Huffman, D. A. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., and Kalenichenko, D. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *arXiv preprint arXiv:1712.05877*, 2017.

Jang, E., Gu, S., and Poole, B. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

Jegou, H., Douze, M., and Schmid, C. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2011.

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

Joulin, A., Grave, E., Bojanowski, P., and Mikolov, T. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016b.

Kim, Y., Jernite, Y., Sontag, D., and Rush, A. M. Character-aware neural language models. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pp. 2741–2749. AAAI Press, 2016.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

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

Maddison, C. J., Mnih, A., and Teh, Y. W. The concrete distribution: A continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712*, 2016.

Marcus, M. P., Marcinkiewicz, M. A., and Santorini, B. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330, 1993.

Mikolov, T., Karafiát, M., Burget, L., Černockỳ, J., and Khudanpur, S. Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pp. 3111–3119, 2013.

Pennington, J., Socher, R., and Manning, C. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing*, pp. 1532–1543, 2014.

Sainath, T. N., Kingsbury, B., Sindhwani, V., Arisoy, E., and Ramabhadran, B. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 6655–6659. IEEE, 2013.

Sennrich, R., Haddow, B., and Birch, A. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

Shu, R. and Nakayama, H. Compressing word embeddings via deep compositional code learning. *https://arxiv.org/abs/1711.01068*, 2017.

Svenstrup, D. T., Hansen, J., and Winther, O. Hash embeddings for efficient word representations. In *Advances in Neural Information Processing Systems*, pp. 4935–4943, 2017.

Zaremba, W., Sutskever, I., and Vinyals, O. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

Zhang, T. Composite quantization for approximate nearest neighbor search.

Zhang, T., Qi, G.-J., Tang, J., and Wang, J. Sparse composite quantization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4548–4556, 2015a.

Zhang, X., Zhao, J., and LeCun, Y. Character-level convolutional networks for text classification. In *Advances in neural information processing systems*, pp. 649–657, 2015b.