Differentiable Product Quantization for End-to-End Embedding Compression

Ting Chen [1] **Lala Li** [1] **Yizhou Sun** [2]

Abstract

Embedding layers are commonly used to map discrete symbols into continuous embedding vectors that reflect their semantic meanings. Despite their effectiveness, the number of parameters in an embedding layer increases linearly with the number of symbols and poses a critical challenge on memory and storage constraints. In this work, we propose a generic and end-to-end learnable compression framework termed differentiable product quantization (DPQ). We present two instantiations of DPQ that leverage different approximation techniques to enable differentiability in end-to-end learning. Our method can readily serve as a drop-in alternative for any existing embedding layer. Empirically, DPQ offers significant compression ratios (14-238×) at negligible or no performance cost on 10 datasets across three different language tasks. [1]

1. Introduction

The embedding layer is a basic neural network module which maps a discrete symbol/word into a continuous hidden vector. It is widely used in NLP related applications, including language modeling, machine translation and text classification. With large vocabulary sizes, embedding layers consume large amounts of storage and memory. For example, in the medium-sized LSTM-based model on the PTB dataset (Zaremba et al., 2014), the embedding table accounts for more than 95% of the total number of parameters. Even with sub-words encoding (Sennrich et al., 2015; Kudo & Richardson, 2018), the size of the embedding layer is still very significant. In addition to words/sub-words models in the text domain (Mikolov et al., 2013; Devlin et al., 2018), embedding layers are also used in a wide range of applications such as knowledge graphs (Bordes et al., 2013; Socher et al., 2013) and recommender systems (Koren et al., 2009), where the vocabulary sizes are even larger.

Recent efforts to reduce the size of embedding layers have been made (Shu & Nakayama, 2017; Chen et al., 2018b), where the authors propose to first learn to encode symbols/words with K-way D-dimensional discrete codes (KD codes, such as 5-1-2-4 for "cat" and 5-1-2-3 for "dog"), and then compose the codes to form the output symbol embedding using an embedding composition function. However, in (Shu & Nakayama, 2017), the discrete codes are fixed before training and are therefore non-adaptive for downstream tasks. (Chen et al., 2018b) proposes to learn discrete codes in an end-to-end fashion which leads to better task performance, but their method has two major drawbacks. Firstly, they utilize complicated embedding composition functions (e.g. recurrent networks, MLPs) to convert discrete codes into embedding vectors, which are both computationally expensive and hard to learn. Secondly, as a result, an additional distillation procedure is required in order to avoid performance drop.

In this work, we propose a simple differentiable product quantization (DPQ) framework, which is an efficient module to insert discrete codes into a neural network while maintaining differentiability. The proposal is based on the observation that the discrete codes are naturally derived through the process of quantization (product quantization by Jegou et al. (2010) in particular), and by making the quantization process differentiable, we are able to learn the discrete codes in an end-to-end fashion. Under our framework, we present two concrete approximation techniques that allow differentiable learning. Compared to the existing methods (Shu & Nakayama, 2017; Chen et al., 2018b), our framework 1) establishes a simple and general framework to generate discrete codes in neural nets in a differentiable manner, 2) enables more efficient computation and better approximations, 3) achieves better task performance *and* better compression ratios, and 4) avoids the two-stage training (pre-training and distillation) as needed in (Chen et al., 2018b).

We conduct experiments on ten different datasets across three language tasks, with additional experiments on BERT (Devlin et al., 2018) pre-training, by simply replacing the original embedding layer with DPQ. The results

[1] Google Research [2] University of California, Los Angeles. Correspondence to: Ting Chen <iamtingchen@google.com>.

Proceedings of the 37^{th} International Conference on Machine Learning, Vienna, Austria, PMLR 119, 2020. Copyright 2020 by the author(s).

[1] Code at: github.com/chentingpc/dpq_embedding_compression.

show that DPQ can learn compact discrete embeddings with higher compression ratios than existing methods, at the same time achieving the same performance as the original full embeddings. Furthermore, our results are obtained from single-stage end-to-end training.

2. Method

Problem setup. An embedding function can be defined as $\mathcal{F}_{\mathcal{W}} : \mathcal{V} \rightarrow \mathbb{R}^d$, where $\mathcal{V}$ denotes the vocabulary of discrete symbols, and $\mathcal{W} \in \mathbb{R}^{n \times d}$ is the embedding table with $n = |\mathcal{V}|$. In standard end-to-end training, the embedding function is jointly trained with other neural net parameters to optimize a given objective. The goal of this work is to learn a compact embedding function $\mathcal{F}_{\mathcal{W}'}$ in the same end-to-end fashion, but the number of bits used for the new parameterization $\mathcal{W}'$ is substantially smaller than the original full embedding table $\mathcal{W}$.

Motivation. One important discovery for constructing a compact embedding layer is to represent each symbol using a learned discrete code (Shu & Nakayama, 2017; Chen et al., 2018b), and then compose the code embedding vectors to form the final symbol embedding vector via an embedding composition function. However, it is not clear how the discrete codes are generated, and what the embedding composition function should be. One could directly optimize codes as free parameters, but it is both ad-hoc and restrictive.

The key insight in this work is that discrete codes can be naturally generated from the process of quantization (product quantization (Jegou et al., 2010) in particular) of a continuous space, and the embedding composition function to get the continuous symbol embedding from the discrete codes is simply the reverse of the quantization process. The quantization process can be specified in flexible ways, and by making this quantization process differentiable we enable single-stage end-to-end learning of discrete codes via optimizing task-specific objectives.

2.1. Differentiable Production Quantization Framework

The proposed differentiable production quantization (DPQ) function is a mapping between continuous spaces, i.e. $\mathcal{T} : \mathbb{R}^d \rightarrow \mathbb{R}^d$. In between the two continuous spaces, there is a discrete space $\{1, \cdots, K\}^D$ which can be seen as a discrete bottleneck. To transform from continuous space to discrete space and back, two important functions are used: 1) a *discretization function* $\boldsymbol{\phi}(\cdot) : \mathbb{R}^d \rightarrow \{1, \cdots, K\}^D$ that maps a continuous vector into a K-way D-dimensional discrete code, and 2) a *reverse-discretization function* $\boldsymbol{\rho}(\cdot) : \{1, \cdots, K\}^D \rightarrow \mathbb{R}^d$ that maps the discrete code into a continuous embedding vector. In other words, the general DPQ mapping is $\mathcal{T}(\cdot) = \boldsymbol{\rho} \circ \boldsymbol{\phi}(\cdot)$.

Compact embedding layer via DPQ. In order to obtain a compact embedding layer, we first take a raw embedding and put it through DPQ function. More specifically, the raw embedding matrix can be presented as a query matrix $\mathbf{Q} \in \mathbb{R}^{n \times d}$ where the number of rows equals to the vocabulary size. The discretization function of DPQ computes discrete codes $\mathbf{C} = \boldsymbol{\phi}(\mathbf{Q})$ where $\mathbf{C} \in \{1, \cdots, K\}^{n \times D}$ is the discrete *codebook*. To construct the final embedding table for all symbols, the reverse-discretization function of DPQ is applied, i.e. $\mathbf{H} = \boldsymbol{\rho}(\mathbf{C})$ where $\mathbf{H} \in \mathbb{R}^{n \times d}$ is the final symbol embedding matrix. In order to make it compact for the inference, we will discard the raw embedding matrix $\mathbf{Q}$ and only store the codebook $\mathbf{C}$ and small parameters needed in the reverse-discretization function. They are sufficient to (re)construct partial or whole embedding table. In below, we specify the discretization function $\boldsymbol{\phi}(\cdot)$ and reverse-discretization function $\boldsymbol{\rho}(\cdot)$ via product keys and values. Figure 1 illustrates the proposed framework. The proposed method can also be seen as a learned hash function of finite input into a set of discrete codes, and use lookup during the inference instead of re-compute the codes.

Product keys for discretization function $\boldsymbol{\phi}(\cdot)$. Given the query matrix $\mathbf{Q}$, the discretization function computes the codebook $\mathbf{C}$. While it is possible to use a complicated transformation, in order to make it efficient, we simply leverage a key matrix $\mathbf{K} \in \mathbb{R}^{K \times d}$ with K rows where K is the number of choices for each code bit. In the spirit of *product keys* in product quantization, we further split columns of $\mathbf{K}$ and $\mathbf{Q}$ into D groups/subspace, such that $\mathbf{K}^{(j)} \in \mathbb{R}^{K \times d/D}$ and $\mathbf{Q}^{(j)} \in \mathbb{R}^{n \times d/D}$.

We can compute each of D dimensional KD codes separately. The j-th dimension of a KD code $\mathbf{C}_i$ for the i-th symbol is computed as follows.

$$\mathbf{C}_i^{(j)} = \arg\min_k \operatorname{dist}\left(\mathbf{Q}_i^{(j)}, \mathbf{K}_k^{(j)}\right) \tag{1}$$

The $\operatorname{dist}(\cdot, \cdot)$ computes distance measure between two vectors, and use it to decide which discrete code to take.

Product values for reverse-discretization function $\boldsymbol{\rho}(\cdot)$. Given the codebook $\mathbf{C}$, the reverse-discretization function computes the final continuous embedding vectors. While this can be another sophisticated transformation, we again opt for the most efficient design and employee a single value matrix $\mathbf{V} \in \mathbb{R}^{K \times d}$ as the parameter. Similarly, we leverage product keys, and split the columns of $\mathbf{V}$ into D groups/subspaces the same way as $\mathbf{K}$ and $\mathbf{Q}$, i.e. $\mathbf{V}^{(j)} \in \mathbb{R}^{K \times d/D}$. We use the code in each of D dimension to index the subspace in $\mathbf{V}$, and concatenate the results to form the final embedding vector as follows.

$$\mathbf{H}_i = [\mathbf{V}^{(1)}_{\boldsymbol{c}_i^{(1)}}, \cdots, \mathbf{V}^{(j)}_{\boldsymbol{c}_i^{(j)}}, \cdots, \mathbf{V}^{(D)}_{\boldsymbol{c}_i^{(D)}}] \tag{2}$$

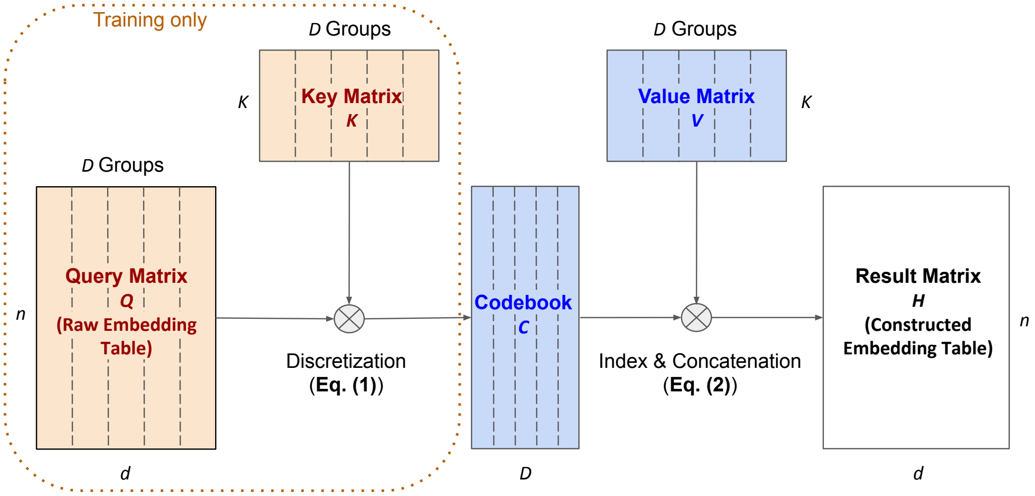


Figure 1. The DPQ embedding framework. During training, differentiable product quantization is used to approximate the raw embedding table (i.e. the query Matrix). At inference, only the codebook $\mathbf{C} \in \{1, ..., K\}^{n \times D}$ and the value matrix $\mathbf{V} \in \mathbb{R}^{K \times d}$ are needed to construct the embedding table.

Algorithm 1 Inference of embedding for i-th token.

```
Require: $\mathbf{V} \in \mathbb{R}^{K \times D \times (d/D)}$, $\mathbf{C} \in \{1, ..., K\}^{n \times D}$
  for $j \in \{1, ..., D\}$ do
    $\boldsymbol{h}_i^{(j)} = \mathbf{V}^{(j)}_{\mathbf{C}_i^{(j)}}$
  end for
  return concatenate($\boldsymbol{h}_i^{(1)}, \boldsymbol{h}_i^{(2)}, ..., \boldsymbol{h}_i^{(D)}$)
```

The inference algorithm for the i-th token is given in Algorithm 1.

Inference complexity. Since only indexing and concatenation (Eq. 2) are used during inference, both the extra computation complexity and memory footprint are usually negligible compared to the regular full embedding (which directly indexes an embedding table).

Storage complexity. Assuming the default 32-bit floating point is used, the original full embedding table requires $32nd$ bits. As for DPQ embedding, we only need to store the codebook and the value matrix: 1) codebook $\mathbf{C}$ requires $nD \log_2 K$ bits, which is the only thing that depends on vocabulary size n, and 2) value matrix $\mathbf{V}$ requires $32Kd$ bits[2], which does not explicitly depend on n and is negligible when n is large. Since typically $nD \log_2 K \ll 32nd$, the DPQ embedding is more compact than the full embedding.

Expressiveness. DPQ achieves compactness by introducing sparsity into the embedding matrix in two axis: (1) the product keys/values, and (2) top-1 selection in each group/subspace. DPQ is able to achieve compactness without decreasing the rank of the resulting embedding matrix.

Proposition 1. *The DPQ embedding matrix* $\mathbf{H}$ *is full rank given the following constraints are satisfied.*

1) *One-hot encoded* $\mathbf{C} \in \{1, ..., K\}^{n \times D}$*, denoted as* $\mathbf{B} \in \{0, 1\}^{n \times KD}$*, is full-rank.*
2) *Sub-matrices of splitted* $\mathbf{V}$*, i.e.* $\mathbf{V}^{(j)} \in \mathbb{R}^{K \times d/D}, \forall j$*, are all full-rank.*
3) $KD \geq d$*.*

The proof is given in the appendix **??**. Note that the above conditions are easy to satisfy while being compact than full embedding, since it is easy to satisfy $nD \log_2 K \ll 32nd$ with $KD \geq d$ in practice.

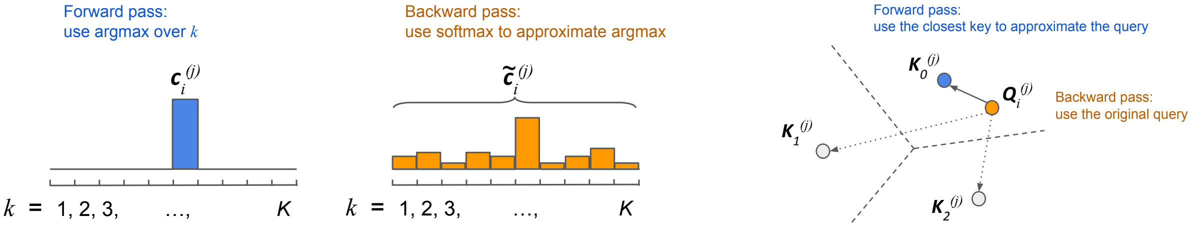
So far we have not specified designs of discretization functions such as the distance function in Eq 1. More importantly, how can we compute gradients through the $\arg\min$ function in Eq. 1? While there could be many instantiations with different design choices, below we introduce two instantiations that use two different approximation schemes for DPQ.

2.2. Softmax-based Approximation

The first instantiation of DPQ (named DPQ-SX) approximates the non-differentiable $\arg\max$ operation with a differentiable softmax function. To do so, we first specify the distance function in Eq. 1 with a softmax function as follows.

$$\mathbf{C}_i^{(j)} = \arg\max_k \frac{\exp(\langle \mathbf{Q}_i^{(j)}, \mathbf{K}_k^{(j)} \rangle)}{\sum_{k'} \exp(\langle \mathbf{Q}_i^{(j)}, \mathbf{K}_{k'}^{(j)} \rangle)} \tag{3}$$

[2] $32Kd/D$ bits if we share the weights among D groups/subspaces.



(a) Softmax-based (DPQ-SX)

(b) Centroid-based (DPQ-VQ)

Figure 2. Illustration of two types of approximation to enable differentiability in DPQ.

Method	**Dist. Metric**	**Key/Value matrices**	**Train**	**Inference**
DPQ-SX	Dot product and more	Not tied, allows different sizes	Efficient	Efficient
DPQ-VQ	Euclidean only	Tied	More efficient	Efficient

Table 1. Summary of differences between VQ and SX. DPQ-SX allows more flexibility in distance metrics and whether to tie the key and value metrices. DPQ-VQ is more efficient during training and therefore is more scalable to larger K, D values.

where $\langle \cdot, \cdot \rangle$ denotes dot product of two vectors (alternatively, other metrics such as Euclidean distance, cosine distance can also be used). To approximate the $\arg\max$, similar to (Jang et al., 2016; Chen et al., 2018b), we relax the softmax function with temperature τ:

$$\tilde{\mathbf{C}}_i^{(j)} = \exp(\langle \mathbf{Q}_i^{(j)}, \mathbf{K}_k^{(j)} \rangle / \tau) / Z \qquad (4)$$

where $Z = \sum_{k'} \exp(\langle \mathbf{Q}_i^{(j)}, \mathbf{K}_{k'}^{(j)} \rangle / \tau)$. Note that now $\tilde{\mathbf{C}}_i^{(j)} \in \Delta^K$ is a probabilistic vector (i.e. soft one-hot vector) instead of an integer $\mathbf{C}_i^{(j)}$. And one_hot$(\mathbf{C}_i^{(j)}) \approx \tilde{\mathbf{C}}_i^{(j)}$, or $\mathbf{C}_i^{(j)} = \arg\max \tilde{\mathbf{C}}_i^{(j)}$. With a one-hot code relaxed into soft one-hot vector, we can replace index operation $\mathbf{V}^{(j)}_{\tilde{\mathbf{C}}_i^{(j)}}$ with dot product to compute the output embedding vector, i.e. $\mathbf{H}_i^{(j)} = \tilde{\mathbf{C}}_i^{(j)} \mathbf{V}^{(j)}$.

The softmax approximated computation defined above is fully differentiable when $\tau \neq 0$. However, to compute discrete codes during the forward pass, we have to set $\tau \to 0$, which turns the softmax function into a spike concentrated on the $\mathbf{C}_i^{(j)}$-th dimension. This is equivalent to the $\arg\max$ operation which does not have gradient.

To enable a pseudo gradient while still be able to output discrete codes, we use a different temperatures during forward and backward pass, i.e. set $\tau = 0$ in forward pass, and $\tau = 1$ in the backward pass. So the final DPQ function can be expressed as follows.

$$\mathbf{H}_i = \mathcal{T}(\mathbf{Q}_i|\tau = 1) - \text{sg}\Big(\mathcal{T}(\mathbf{Q}_i|\tau = 1) - \mathcal{T}(\mathbf{Q}_i|\tau = 0)\Big) \qquad (5)$$

Where sg is the *stop gradient* operator, which is identity function in forward pass, but drops gradient for variables inside it during the backward pass. And $\mathcal{T}(\cdot) = \boldsymbol{\rho} \circ \boldsymbol{\phi}(\cdot)$ is the DPQ mapping defined by Eq. 1, 2 with distance function specified by Eq 4.

2.3. Centroid-based Approximation

The second instantiation of DPQ (named DPQ-VQ) uses a centroid-based approximation, which directly pass the gradient straight-through (Bengio et al., 2013) a small set of centroids. In order to do so, we need to put $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ into the same space.

First, we treat rows in key matrix $\mathbf{K}$ as centroids, and use them to approximate query matrix $\mathbf{Q}$. The approximation is based on the Euclidean distance as follows.

$$\mathbf{C}_i^{(j)} = \arg\min_k \|\mathbf{Q}_i^{(j)} - \mathbf{K}_k^{(j)}\|^2 \qquad (6)$$

Secondly, we tie the key and value matrices, i.e. $\mathbf{V} = \mathbf{K}$, so that we can pass the gradient through.

We still have the non-differentiable $\arg\min$ operation, and the input query $\mathbf{Q}_i^{(j)}$ are different from selected output centroid $\mathbf{V}^{(j)}_{\mathbf{C}_i^{(j)}}$. However, since they are in the same space, it allows us to directly pass the gradient straight-through as follows.

$$\mathbf{H}_i = \mathbf{Q}_i - \text{sg}(\mathbf{Q}_i - \mathcal{T}(\mathbf{Q}_i)) \qquad (7)$$

Where sg is again the *stop gradient* operation. During the forward pass, the selected centroid is emitted, but during the backward pass, the gradient is pass to the query directly. This provides a way to compute discrete codes in the forward pass (which are the indexes of the centroids), and update the query matrix during the backward pass.

However, it is worth noting that the Eq. 7 only approx-

Task	Dataset	Vocab Size	Tokenization	Base Model
LM	PTB Wikitext-2	10,000 33,278	Words	LSTM-based models from (Zaremba et al., 2014), three model sizes
NMT	IWSLT15 (En-Vi) IWSLT15 (Vi-En)	17,191 7,709	Words	Seq2seq-based model from (Luong et al., 2017)
	WMT19 (En-De)	32,000	Sub-words	Transformer Base in (Vaswani et al., 2017)
TextC	AG News Yahoo! Ans. DBpedia Yelp P Yelp F	69,322 477,522 612,530 246,739 268,414	Words	One hidden layer after mean pooling of word vectors, similar to fastText from (Joulin et al., 2017)

Table 2. Datasets and models used in our experiments. More details in Appendix **??**.

imates gradient for query matrix, but does not updates the centroids, i.e. the tied key/value matrix. Similar to (van den Oord et al., 2017), we add a regularization term: $\mathcal{L}_{reg} = \sum_i \|\mathcal{T}(\mathbf{Q}_i) - \text{sg}(\mathbf{Q}_i)\|^2$, which makes entries of the key/value matrix arithmetic mean of their members. Alternatively, one can also use Exponential Moving Average (Kaiser et al., 2018) to update the centroids.

A comparison between DPQ-SX and DPQ-VQ. DPQ-VQ and DPQ-SX only differ during training. They are different in how they approximate the gradient for the non-differentiable $\arg\min$ function: DPQ-SX approximates the one-hot vector with softmax, while DPQ-VQ approximates the continuous vector using a set of centroids. Figure 2 illustrates this difference. This suggests that when there is a large gap between one-hot and probabilistic vectors (large K), DPQ-SX approximation could be poor; and when there is a large gap between the continuous vector and the selected centroid (large subspace dimension, i.e. small D), DPQ-VQ could have a big approximation error.

Table 1 summarizes the comparisons between DPQ-SX and DPQ-VQ. DPQ-SX is more flexible as it does not constrain the distance metric, nor does it tie the key/value matrices as in DPQ-VQ. Thus one could use different sizes of key and value matrices. Regarding to the computational cost during training, DPQ-SX back-propagates through the whole distribution of K choices, while DPQ-VQ only back-propagates through the nearest centroid, making it more scalable (to large K, D, and batch sizes).

2.4. Implementation Details

Distance normalization. Training with straight-through estimator can be unstable as the gradient is approximated. To mitigate this problem, we apply batch normalization (Ioffe & Szegedy, 2015) for the distance measure in DPQ along the K-dimension, i.e. each centroid will have a normalized distance distribution over batch samples.

Subspace-sharing. To further reduce parameters, one can share parameters among the D groups in the Key/Value Matrices, i.e. constraining $\mathbf{K}^{(j)} = \mathbf{K}^{(j')}$ and $\mathbf{V}^{(j)} = \mathbf{V}^{(j')}, \forall j, j'$. For simplicity we refer to this as "subspace-sharing". We search over both options and utilize subspace-sharing if there is no performance drop.

3. Experiments

We conduct experiments on ten datasets across three tasks: language modeling (LM), neural machine translation (NMT) and text classification (TextC) (Zhang et al., 2015). We adopt existing architectures for these tasks as base models and only replace the input embedding layer with DPQ embeddings. The details of datasets and base models are summarized in Table 2.

We evaluate the models using two metrics: task performance and compression ratio. Task performance metrics are perplexity scores for LM tasks, BLEU scores for NMT tasks, and accuracy in TextC tasks. Compression ratios for the embedding layer is computed as follows:

$$\text{CR} = \frac{\text{\# of bits used in the full embedding table}}{\text{\# of bits used in compressed model for inference}}$$

For DPQ in particular, this can be computed as $\text{CR} = \frac{32nd}{nD\log_2 K + 32Kd}$. Further compression can be achieved with 'subspace-sharing', with which we have $\text{CR} = \frac{32nd}{nD\log_2 K + 32Kd/D}$. In this work, we focus on the embedding table in the encoder side, so we keep the decoder embedding layer (i.e. output softmax layer) as is.

3.1. Compression Ratios and Task Performance Against Baselines

Comparison with full embedding baseline on ten datasets. Table 3 summarizes the task performance and compression ratios of DPQ-SX and DPQ-VQ against baseline models that use the regular full embeddings[3]. In each

[3] For LM, results are from the medium-sized LSTM model.

Task	Metric	Dataset	Baseline	DPQ-SX	(Compr. Ratio↑)	DPQ-VQ	(Compr. Ratio↑)
LM	PPL↓	PTB	83.4	**83.2**	**(163.2)**	83.3	(58.7)
		Wikitext-2	95.6	**95.0**	**(59.3)**	95.9	(95.3)
NMT	BLEU↑	IWSLT15 (En-Vi)	**25.4**	25.3	(86.2)	25.3	(16.1)
		IWSLT15 (Vi-En)	23.0	**23.1**	**(72.0)**	22.5	(14.1)
		WMT19 (En-De)	**38.8**	**38.8**	**(18.0)**	38.7	(18.2)
TextC	Acc(%)↑	AG News	**92.6**	92.5	(19.3)	92.6	(24.0)
		Yahoo! Ans.	69.4	**69.6**	**(48.2)**	69.2	(19.2)
		DBpedia	98.1	98.1	(24.1)	**98.1**	**(38.5)**
		Yelp P	93.9	**94.2**	**(38.5)**	93.9	(24.0)
		Yelp F	**60.3**	60.1	(48.2)	60.2	(24.1)

Table 3. Comparisons of DPQ variants vs. the full embedding baseline on ten datasets across three tasks. We use ↓ to denote the lower the better, in contrast, ↑ means the higher the better.

	Small		Medium		Large	
Method	**PPL↓**	**CR↑**	**PPL↓**	**CR↑**	**PPL↓**	**CR↑**
Full	114.5	1	83.4	1	78.7	1
Shu′17	108.0	4.8	84.9	12.5	80.7	18.5
Chen′18	108.5	4.8	89.0	12.5	86.4	18.5
Chen′18+	107.8	4.8	83.1	12.5	**77.7**	18.5
DPQ-SX	**105.8**	**85.5**	**82.0**	**82.9**	78.5	**238.3**
DPQ-VQ	106.5	51.1	83.3	58.7	79.5	**238.3**

Table 4. Comparison of DPQ against recently proposed embedding compression techniques on the PTB LM task (LSTMs with three model sizes are studied). Metrics are perplexity (PPL) and compression ratio (CR).

Method	PPL↓	Compr. Ratio↑
Full	83.4	1.0
Scalar quantization (8 bits)	84.1	4.0
Scalar quantization (6 bits)	87.7	5.3
Scalar quantization (4 bits)	92.9	8.3
Product quantization(64x325)	84.0	8.3
Product quantization(128x325)	83.7	6.7
Product quantization(256x325)	83.7	5.3
Low-rank (5X)	84.8	5.0
Low-rank (10X)	85.5	10.2
Ours (DPQ-VQ)	83.3	58.7
Ours (DPQ-SX)	**82.0**	**82.9**

Table 5. Comparison of DPQ against traditional embedding compression techniques on the PTB LM task (medium-sized LSTM).

task/dataset, we report results from a configuration that gives as good task performance as the baseline (or as good as possible, if it does not match the baseline) while providing the largest compression ratio. In all tasks, both DPQ-SX and DPQ-VQ can achieve comparable or better task performance while providing a compression ratio from 14× to 163×. In 6 out of 10 datasets, DPQ-SX performs strictly better than DPQ-VQ in both metrics. *Remarkably, DPQ is able to further compress the already-compact sub-word representations used in WMT19 (En-De). This shows great potential of DPQ to learn very compact embedding layers.*

Comparison with more baselines on LM. Here we compare DPQ against the recently proposed embedding compression methods. Shu′17 from (Shu & Nakayama, 2017): a three-step procedure where one firstly trains a full model, secondly learns discrete codes to reconstruct the pre-trained embedding layer, and finally fixes the discrete codes and trains the model again; Chen′18 from (Chen et al., 2018b): end-to-end training without distillation guidance from a pre-trained embedding table; Chen′18+ from (Chen et al., 2018b): an end-to-end training with an additional distillation procedure that uses a pre-trained embedding as guidance during training. Table 4 shows the comparison between DPQ and the above methods on the PTB language modeling task using LSTMs with three different model sizes. We find that 1) other than DPQ, only Chen′18+ which requires an extra distillation procedure is able to achieve similar perplexity scores as the full embedding baseline; 2) DPQ variants (particularly DPQ-SX) are able to obtain extremely competitive perplexity scores in all cases, while offering compression ratios that are an order of magnitude larger than all the other alternatives (and is trained end-to-end in a single stage).

Table 5 shows comparisons on the PTB language modeling task (using medium-sized LSTM) with broader set of baselines (including methods that are not based on discrete codes). We find that 1) traditional compression techniques, such as scalar and product quantization as well as low-rank factorization, typically degenerates the performance significantly in order to achieve good compression ratios compared to discrete code learning-based methods (Shu & Nakayama, 2017; Chen et al., 2018b); 2) DPQ can largely improve the compression ratio while achieving similar or better task performance (perplexity in this case).

Compared with more baselines on NMT. We compare DPQ with product quantization, where we first train the full

Dataset	AG News	Yahoo!	DBPedia	Yelp P	Yelp F
Full	92.6 (1.0)	69.4 (1.0)	98.1 (1.0)	93.9 (1.0)	60.3 (1.0)
Low-rank(10×)	91.4 (10.4)	69.5 (10.2)	97.7 (10.3)	92.4 (10.4)	57.8 (10.3)
Low-rank(20×)	91.5 (21.4)	69.1 (21.5)	97.9 (21.3)	92.4 (21.5)	57.3 (21.4)
(Chen et al., 2018b)	91.6 (53.3)	69.5 (31.7)	98.0 (48.4)	93.1 (48.6)	59.0 (54.4)
DPQ-VQ	**92.6 (24.0)**	69.2 (19.2)	**98.1 (38.5)**	93.9 (24.0)	60.2 (24.1)
DPQ-SX	92.5 (19.3)	**69.6 (48.2)**	98.1 (24.1)	**94.2 (38.5)**	**60.1 (48.2)**

Table 6. Performance comparison on text classification task. The accuracy and compression ratios (in parenthesis) are shown below. The proposed method (DPQ) usually achieves better accuracies than baselines, at the same time providing better compression ratios.

Embeddings	CR	Squad 1.1	Squad 2.0	CoLA	MNLI	MRPC	XNLI
Full	1.0	90.1± 0.1 /83.1± 0.3	78.8±0.6/75.5±0.6	80.6±0.7	84.3±0.1	85.9±0.5	53.5±0.4
DPQ-SX	37.0	90.0±0.1 /83.0±0.2	78.7±0.5/75.4±0.5	80.2±0.6	83.7±0.2	85.1±0.6	53.4±0.1

Table 7. Effect of using DPQ on BERT. DPQ gives a compression ratio of 37× on the embedding table while the model's performance on downstream tasks remains competitive.

Method	BLEU↑	Compr. Ratio↑
Full	38.8	1
PQ (K=128, D=64)	28.9	31.9
PQ (K=32, D=128)	35.4	25.0
PQ (K=128, D=128)	35.7	17.0
PQ (K=32, D=256)	36.9	12.6
PQ (K=128, D=256)	37.8	8.8
DPQ-VQ (K=32, D=128)	**38.7**	**17.0**
DPQ-SX (K=32, D=128)	**38.8**	**17.0**

Table 8. Comparison of end-to-end DPQ against using PQ to reconstruct the embedding table after the model is trained, for the NMT task on WMT19 (En-DE).

embedding model, and then learn discrete codes to reconstruct the learned full embedding table. Finally, the reconstructed embedding table replaces the original embedding table for inference. In our experiment, we use auto-encoder and DPQ (with different K and D) to learn to reconstruct the full embedding table.

Table 8 shows comparisons between DPQ and PQ baselines on WMT19 (En-De) with Transformer (Vaswani et al., 2017). We can see that PQ degenerates the performance significantly. This is expected as small approximation errors in the embedding layer accumulate and can be amplified as the errors propagate through the deep neural nets, finally leading to large errors in the output space. End-to-end training does not have this problem as the whole system is jointly learned, so the networks can account for small approximation errors in the early layers.

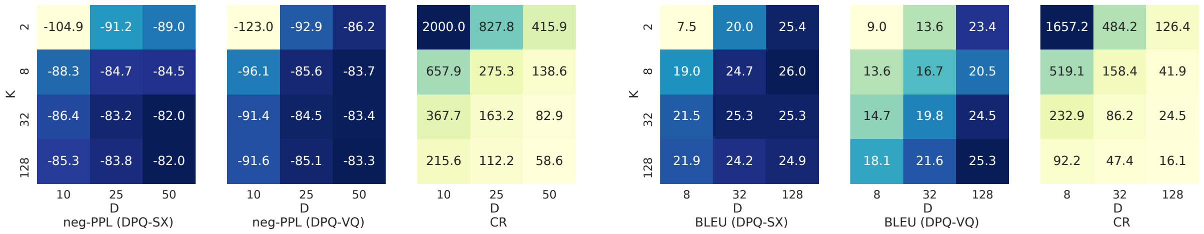
Comparison with more baselines on TextC. Table 6 provides performance comparisons on text classification tasks. We found that the proposed method (DPQ) usually achieves better accuracies than baselines, at the same time providing better compression ratios.

3.2. Applying DPQ to BERT

To further test DPQ, we replace the embedding layer in BERT with our DPQ for both pre-training and fine-tuning. We do not perform hyper-parameter search for DPQ, but simply use the best configuration from our experiments on WMT19 EnDe using Transformer, i.e. we use DPQ-SX with no subspace-sharing with $K = 32, D = 128$. For both pre-training and fine-tuning, we use the same exact configurations and hyper-parameters as in original BERT-base in (Devlin et al., 2018). Table 7 shows that DPQ performs on par with full embedding in most of the downstream tasks, while giving a compression ratio of 37× on the embedding table.

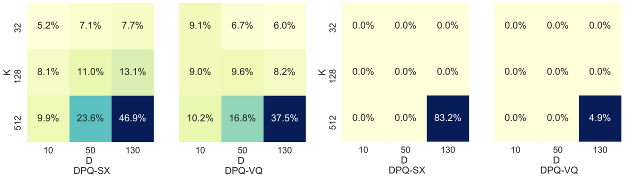
3.3. Effects of K and D

Among key hyper-parameters of DPQ are the code size: K the number of centroids per dimension and D the code length. Figure 3 shows the task performance and compression ratios for different K and D values on PTB and IWSLT15 (En-Vi). Firstly, we observe that the combination of a small K and a large D is a better configuration than the other way round. For example, in IWSLT15 (En-Vi), $(K = 2, D = 128)$ is better than $(K = 128, D = 8)$ in both BLEU and CR, with both DPQ-SX and DPQ-VQ. Secondly, increasing K or D would typically improve the task performance at the expense of lower CRs, which means one can adjust K and D to achieve the best task performance and compression ratio trade-off. Thirdly, we note that decreasing D has a much more traumatic effect on DPQ-VQ than on DPQ-SX in terms of task performance. This is because as the dimension of each sub-space (d/D) increases, the nearest neighbour approximation (that DPQ-VQ relies on)



(a) PTB (b) IWSLT15 (En-Vi)

Figure 3. Heat-maps of task performance and compression ratio for various K and D values. Darker is better. Key observations are: 1) increasing K or D typically improves the task performance at the expense of lower CRs; 2) the combination of a small K and a large D is better than the other way round.



(a) Extra training time used. (b) Extra training memory used.

Figure 4. Extra training cost incurred by DPQ, measured on a medium sized LSTM for LM trained on Tesla-V100 GPUs. For most K and D values, the extra training time is within 10%, and the extra memory usage is zero.

becomes less exact.

3.4. Computational Cost

DPQ incurs a slightly higher computational cost during training and no extra cost at inference. Figure 4 shows the training speed as well as the (GPU) memory required when using DPQ on the medium LSTM model, trained on Tesla-V100 GPUs. For most K and D values, the extra training time is within 10%, and the extra training memory is zero. For very large K and D values, DPQ-VQ has better computational efficiency than DPQ-SX (as expected). At inference, we do not observe any impact on speed or memory from DPQ.

4. Related Work

Modern neural networks have many parameters and redundancies. The compression of such models has attracted many research efforts (Han et al., 2015; Howard et al., 2017; Chen et al., 2018a). Most of these compression techniques focus on the weights that are shared among many examples, such as convolutional and dense layers (Howard et al., 2017; Chen et al., 2018a). The embedding layers are different in the sense that they are tabular and very sparsely accessed, i.e. the pruning cannot remove rows/symbols in the embedding table, and only a few symbols are accessed in each data sample. This makes the compression challenges different for the embedding layers. Other approaches such as scalar and product quantization (Jegou et al., 2010; Joulin et al., 2017), as well as compression selection criterion (May et al., 2019) are also explored.

Existing work on compressing embedding layers includes (Shu & Nakayama, 2017; Chen et al., 2018b), which also leverage discrete codes. As mentioned in Section 1, our proposed framework is more general and flexible, allowing for two approximation techniques to be used in a single-stage training process. The product keys and values in our model make it more efficient in both training and inference. Empirically, DPQ achieves significantly better compression ratios and at the same time does not need an extra distillation process.

Our work differs from traditional quantization techniques (Jegou et al., 2010) in that they can be trained in an end-to-end fashion. The idea of utilizing multiple orthogonal subspaces/groups for quantization is used in product quantization (Jegou et al., 2010; Norouzi & Fleet, 2013) and multi-head attention (Vaswani et al., 2017).

The two approximation techniques presented for DPQ in this work also share similarities with Gumbel-softmax (Jang et al., 2016) and VQ-VAE (van den Oord et al., 2017). However, we do not find using stochastic noises (as in Gumbel-softmax) useful since we aim to get deterministic codes. It is also worth pointing out that these techniques (Jang et al., 2016; van den Oord et al., 2017) by themselves cannot be directly applied to compression.

5. Conclusion

In this work, we propose a simple and general differentiable product quantization framework for learning compact embedding layers. We provide two instantiations of our framework, which can readily serve as a drop-in replacement for existing embedding layers. We empirically demonstrate the effectiveness of the proposed method in a wide variety of language tasks. Beyond embedding compression, the proposed DPQ layer can also be used for end-to-end learning of general discrete codes in neural networks.

Acknowledgements

We would like to thank Koyoshi Shindo, Mohammad Norouzi, Wang-cheng Kang, Derek Cheng, Dong Lin, Xinyang Yi, Alex Beutel for helpful discussions.

References

Anil, R., Gupta, V., Koren, T., and Singer, Y. Memory-Efficient Adaptive Optimization for Large-Scale Learning. In *arXiv*, 2019.

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., Lin, J., Lin, T., Han, S., Wang, C., and Zhou, D. Adaptive mixture of low-rank factorizations for compact neural modeling. *Neural Information Processing Systems (CDNNRIA workshop)*, 2018a.

Chen, T., Min, M. R., and Sun, Y. Learning k-way d-dimensional discrete codes for compact embedding representations. In *International Conference on Machine Learning*, 2018b.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

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*, 2015.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

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, 2010.

Joulin, A., Grave, E., Bojanowski, P., and Mikolov, T. Bag of tricks for efficient text classification. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics*, 2017.

Kaiser, Ł., Roy, A., Vaswani, A., Parmar, N., Bengio, S., Uszkoreit, J., and Shazeer, N. Fast decoding in sequence models using discrete latent variables. *arXiv preprint arXiv:1803.03382*, 2018.

Koren, Y., Bell, R., and Volinsky, C. Matrix factorization techniques for recommender systems. *Computer*, pp. 30–37, 2009.

Kudo, T. and Richardson, J. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 66–71, 2018.

Luong, M., Brevdo, E., and Zhao, R. Neural machine translation (seq2seq) tutorial. *https://github.com/tensorflow/nmt*, 2017.

May, A., Zhang, J., Dao, T., and Ré, C. On the downstream performance of compressed word embeddings. In *Advances in neural information processing systems*, pp. 11782–11793, 2019.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

Norouzi, M. and Fleet, D. J. Cartesian k-means. In *Proceedings of the IEEE Conference on computer Vision and Pattern Recognition*, pp. 3017–3024, 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. *arXiv preprint arXiv:1711.01068*, 2017.

Socher, R., Chen, D., Manning, C. D., and Ng, A. Reasoning with neural tensor networks for knowledge base completion. In *Advances in neural information processing systems*, pp. 926–934, 2013.

van den Oord, A., Vinyals, O., et al. Neural discrete representation learning. In *Advances in Neural Information Processing Systems*, pp. 6306–6315, 2017.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

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

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