Accelerating Neural Machine Translation with Partial Word Embedding Compression

Fan Zhang[1,2], **Mei Tu**[2], **Jinyao Yan**[1*]
[1] State Key Laboratory of Media Convergence and Communication, Communication University of China
[2] Samsung Research China – Beijing (SRC-B)
zhang.fan@samsung.com, mei.tu@samsung.com, jyan@cuc.edu.cn

Abstract

Large model size and high computational complexity prevent the neural machine translation (NMT) models from being deployed to low resource devices (e.g. mobile phones). Due to the large vocabulary, a large storage memory is required for the word embedding matrix in NMT models, in the meantime, high latency is introduced when constructing the word probability distribution. Based on reusing the word embedding matrix in the softmax layer, it is possible to handle the two problems brought by large vocabulary at the same time. In this paper, we propose Partial Vector Quantization (P-VQ) for NMT models, which can both compress the word embedding matrix and accelerate word probability prediction in the softmax layer. With P-VQ, the word embedding matrix is split into two low dimensional matrices, namely the shared part and the exclusive part. We compress the shared part by vector quantization and leave the exclusive part unchanged to maintain the uniqueness of each word. For acceleration, in the softmax layer, we replace most of the multiplication operations with the efficient looking-up operations based on our compression to reduce the computational complexity. Furthermore, we adopt curriculum learning and compact the word embedding matrix gradually to improve the compression quality. Experimental results on the Chinese-to-English translation task show that our method can reduce 74.35% of parameters of the word embedding and 74.42% of the FLOPs of the softmax layer. Meanwhile, the average BLEU score on the WMT test sets only drops 0.04.

Introduction

Based on the encoder-decoder framework, the neural machine translation (NMT) (Bahdanau, Cho, and Bengio 2014) has developed rapidly in recent years. However, large model size and high computational complexity prevent the NMT models from being deployed to low resource devices (e.g., mobile phones). In NMT models, to represent the discrete words, the word embedding has become a basic component which automatically learns the word representations during training. Due to the large vocabulary, the word embedding matrix requires large storage memory, which dominates the NMT model size. Take the standard Transformer model as an example. When the total vocabulary size reaches 50,000, the word embedding matrix size accounted for 36.6% of the whole model. Moreover, the impact of the large vocabulary size is more than that. In the softmax layer, when projecting the output of the top hidden layer (which is also called the context vector) onto a probability distribution over all the words in the vocabulary, an extremely high computational complexity is required. Meanwhile, an output embedding matrix with the same size of the target word embedding matrix is also required. Fortunately, based on the same shape of the two matrices, early studies (Inan, Khosravi, and Socher 2016; Press and Wolf 2016) reuse the target word embedding to project the context vector, which is proved to perform well in NMT models. But the high computational complexity to compute word probabilities still remains to be solved. In auto-regressive sequence generation configure, this operation of probability calculation even runs a couple of times to generate the whole sequence, which causes a much higher computational complexity. For instance, to generate a 10-word sequence with vocabulary size 30,000, the total FLOPs (Hunger 2005) of the softmax layer reaches 307.2 million. When deploying this model to a low resource environment, the translation speed will be quite slow.

Based on vector quantization (VQ) (Burton, Shore, and Buck 1983; Gersho and Gray 2012), many variant works greatly reduce the word embedding size by sharing parameters. This gives us the inspiration that we only need to project the context vector by the shared parameters when adopting these works in the softmax layer. This operation requires less computational complexity. However, previous VQ works (Shu and Nakayama 2017; Shi and Yu 2018) only focus on the compression of word embedding rather than the output embedding in the softmax layer. Parameter sharing in the softmax layer leads to two potential problems - parameter oscillation problem and ambiguity problem. Another big challenge in VQ works is how to maintain the quality of the model after compression. When the embedding matrix is directly compressed into a much smaller one, a big information gap occurs and thus may harm the model quality.

To address the above issues, we propose a simple but very effective method, the Partial Vector Quantization (P-VQ), to reduce the computational complexity in the softmax layer with partial word embedding compression. We tie the target word embedding matrix and the output embedding matrix following Inan, Khosravi, and Socher (2016). The tied

[*] Corresponding author: jyan@cuc.edu.cn

matrix is called the embedding matrix in the rest. Then the embedding matrix is split into the shared part and the exclusive part. The shared part is compressed by VQ, in the meantime the exclusive part is unchanged to avoid the potential problems. Besides, in the softmax layer, we replace most of the multiplication operations with the more efficient looking-up operations to reduce computational complexity. To avoid the big information gap and improve the compression quality, following the idea of curriculum learning (Bengio et al. 2009) that encourages students to learn from easy to hard, we increase the compression rate by degrees during the compression phase. In this way, we make the information gap much smaller in each compression step and eliminate it with continuous training. We evaluate our method on the Chinese-to-English translation task. The empirical results show that, P-VQ can reduce both parameter number of the embedding matrix by 74.35% and the required FLOPs in the softmax layer by 74.42%, while keeping a high translation quality on WMT test sets. Overall, the main contributions in our work are as follows,

- The proposed P-VQ takes advantage of partial embedding parameter sharing to compress the embedding matrix. Different from other VQ variant works, our method preserves the exclusive part and thus avoid the two potential problems to have a better performance.
- Based on partial compression, our method reduces the computational complexity of the softmax layer at the same time by replacing multiplication operations with the efficient looking-up operations.
- We adopt curriculum learning and compress the embedding matrix gradually, which significantly improves the compression quality.

Related Works

In the literature of neural language processing, there have been several works that try to resolve the problem caused by the large vocabulary.

Some works focus on reducing the computational complexity in the softmax layer. Hierarchical softmax methods (Goodman 2001; Morin and Bengio 2005; Mikolov et al. 2011) cluster the words in the vocabulary V into k groups first and then predict the next word by two steps: predict the group, then limit the word prediction to this group. In this way, the computational complexity $O(|V|)$ is roughly decomposed into $O(k + |V|/k)$. Unfortunately, hierarchical softmax methods even increase the size of the model because an extra weight matrix is needed for the group prediction. Binary code prediction (Oda et al. 2017) represents the words by binary codes that can speed up both training and predicting process, but the translation quality drops relatively high when the training set is large.

Some other works focus on reducing the embedding size. Liu et al. (2019) use shared-private word embedding for bilingual NMT models to reduce the embedding dimensions, it has little impact on computational complexity. Differentiated softmax (Chen, Grangier, and Auli 2015) uses a structure sparse matrix to reduce both embedding matrix size and computational complexity at the same time. But it only slightly compresses the embedding matrix size. Factorized embedding parameterization (Lan et al. 2019) build the word embedding by a low rank matrix, which can be decomposed into two lower dimensional matrices, this property can also be used to reduce computational complexity. Structure pruning (Anwar, Hwang, and Sung 2017) is mainly used in convolutional neural networks to drop weak channels, which can also cut off dimensions with smaller values in the embedding matrix.

Vector quantization (VQ) (Burton, Shore, and Buck 1983; Gersho and Gray 2012) is widely used in many tasks (Soong 1985; Faundez-Zanuy 2007; Jegou, Douze, and Schmid 2010). Many VQ variations are employed to reduce the word embedding size by using multiple codes. These variations can be approximately divided into two categories. One is additive quantization (Babenko and Lempitsky 2014). For instance, LightRNN (Li et al. 2016) uses the 2-component shared embedding for word representation, each word is assigned two codes; Deep composition (Shu and Nakayama 2017) uses a number of codes for word representation. Another is product quantization (Jegou, Douze, and Schmid 2010). The structured word embedding (Shi and Yu 2018) and the KD encoding (Chen, Min, and Sun 2018) both belong to this category. Similar with Shu and Nakayama (2017), they represent each word by a D-dimensional code with a cardinality of K, but with different code assignment strategies. The above VQ variations all have the ability to reduce the computational complexity, but they have to face the two potential problems at the same time.

Our Method

In this section, we introduce our proposed Partial Vector Quantization algorithm in details.

Problem Formulation

In neural machine translation (NMT) models, words are represented by vectors, which form the word embedding matrix $W \in R^{|V| \times d}$. Here V is the vocabulary of all the words, while d is the length of the vectors which is also called the hidden dimension. The parameter number $Param.$ of W is:

$$Param. = |V| \times d \tag{1}$$

When focusing on the decoder, by reusing $W \in R^{|V| \times d}$ in the softmax layer, the probability P_t^i of the i-th word in the vocabulary at position t can be constructed as:

$$P_t^i = \frac{exp(Y_t^i)}{\sum_{j \in V} exp(Y_t^j)} \tag{2}$$

where $Y_t \in R^{|V|}$ is the logit values of all the words:

$$Y_t = W h_{t-1} + b \tag{3}$$

In the above function, $b \in R^{|V|}$ is the bias vector. The vector $h_t \in R^d$ is the output context at position t between decoder and softmax layer (e.g., output of the Transformer decoder), which is also called the context vector.

We use the floating point operations (FLOPs) to represent the computational complexity. Following the description in Molchanov et al. (2016), for fully connected layers

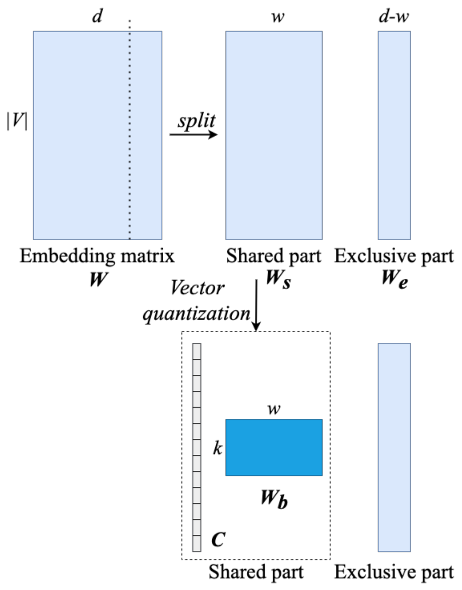

Figure 1: Partial Vector Quantization.

with bias, FLOPs can be calculated by $2I \times O$, where I and O are the input and output dimensionality respectively. Thus, to compute Y_t, the number of required FLOPs is:

$$F = 2d \times |V| \tag{4}$$

This work is devoted to reducing $Param.$ and F.

Partial Vector Quantization

Previous VQ variant works show excellent performance in the word embedding compression. However, they have two potential problems when reusing the word embedding matrix in the softmax layer: (i) parameter oscillation problem and (ii) ambiguity problem. To construct word probability distribution, the word logits Y_t is computed as Eq. (3). But in these works, parameters of W are highly shared, which results in a fact that word logits of each word is made up of the word logits of the other words. In another word, each contribution of the output probability of one word is always affected by others. These words will try to adjust the same embedding parameters during training, which may make the model hard to converge ((i) parameter oscillation problem). Due to the compression strategy, some of these works even have a potential risk that different words may have the same code representation (Chen, Min, and Sun 2018). That means these words will always have the same probability during predicting, which makes the model confusing in determining the final output word ((ii) ambiguity problem).

We note that the reason for the above problems is that the words have no unique representations, the embedding parameters of each word always share with the others. To avoid these potential problems, we propose partial compression to keep the uniqueness of each word, which is also an important technical innovation in our work.

In P-VQ, words in the vocabulary are allocated into k groups, where k is much smaller than the vocabulary size $|V|$. With a given window size $w \in [1, \ldots, d-1]$, a group embedding matrix $W_b \in R^{k \times w}$, which is also named *codebook*, is built to represent each group. Each word is assigned a code $c \in [1, ..., k]$ to find its own group. Meanwhile, an exclusive embedding matrix $W_e \in R^{|V| \times (d-w)}$ is also built for each word to retain its exclusive embedding vector v_e. In another word, the full embedding vector v^n of the n-th word in the vocabulary is concatenated by a shared group vector and an exclusive vector:

$$v^n = f^{concat}(f^{lookup}(W_b, c^n), v_e^n) \tag{5}$$

Here $f^{lookup}(a, b)$ means to find elements in a by the index b, while $f^{concat}(a, b)$ means to concatenate matrix a and matrix b by the last dimension.

In practice, as shown in Figure 1, we first split the word embedding matrix $W \in R^{|V| \times d}$ into the shared part $W_s \in R^{|V| \times w}$ and the exclusive part $W_e \in R^{|V| \times (d-w)}$ by a fixed window size $w \in [1, \ldots, d-1]$, which means:

$$W = f^{concat}(W_s, W_e) \tag{6}$$

Then W_s is compacted into the codebook $W_b \in R^{k \times w}$ and the code assignment matrix $C = \{c^n\}_{n=1}^{V}$ by VQ, where $c^n \in [1, ..., k]$ is the code of the n-th word in the vocabulary.

By sharing the embedding vectors of the shared part among words in the same group, the embedding matrix now have $k \times w + |V| \times (d - w)$ floating point parameters ($W_b \in R^{k \times w}$ and $W_e \in R^{|V| \times (d-w)}$) and $|V|$ integer parameters ($C \in R^{|V|}$). Compared with Eq. (1), the reduced floating point parameter number $\Delta Param.$ is:

$$\Delta Param. = (|V| - k) \times w \tag{7}$$

The above function shows, when the vocabulary size $|V|$ is fixed, the bigger the w is or the smaller the k is, the higher the compression rate is.

Efficient Computing in Softmax

After compact the word embedding matrix by P-VQ, we now are able to construct the word probability distribution efficiently by reducing the computational complexity of the logit value calculation in the softmax layer. According to Eq. (5), the full word embedding matrix $W \in R^{|V| \times d}$ can be restored by the codebook matrix $W_b \in R^{k \times w}$, the code assignment matrix $C \in R^{|V|}$ and the exclusive embedding matrix $W_e \in R^{|V| \times (d-w)}$:

$$W = f^{concat}(f^{lookup}(W_b, C), W_e) \tag{8}$$

For the sake of more concise expression, we hide the subscript of the position t. Combined with Eq. (3), the context vector $h \in R^d$ can be projected onto the logit values $Y \in R^{|V|}$ by Eq. (8) separately.

We first reuse the window size w to split h into $h_b \in R^w$ and $h_e \in R^{d-w}$, which means:

$$h = f^{concat}(h_b, h_e) \tag{9}$$

Then the logit values $Y \in R^{|V|}$ can be computed as:

$$Y = Y_{vq} + Y_p + b \tag{10}$$

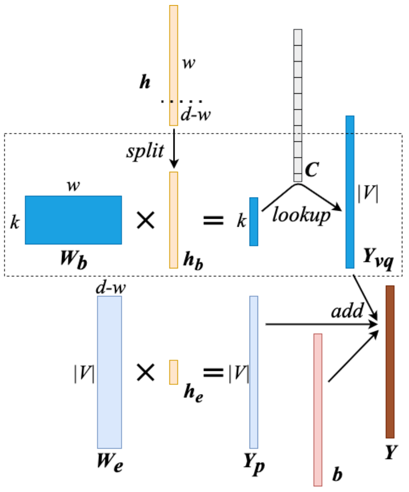

Figure 2: The efficient way to compute the word logits Y. The operations in the dashed box are the core operations to reduce the computational complexity. Here k, w, d, $|V|$ are all matrix shapes.

where

$$Y_{vq} = f^{lookup}(W_b, C)h_b \tag{11}$$

and

$$Y_p = W_e h_e \tag{12}$$

To compute Y_{vq}, considering the efficiency difference between different operations, we change the order of operations in Eq. (11) into a more efficient way:

$$Y_{vq} = f^{lookup}(W_b h_b, C) \tag{13}$$

which means instead of first reconstructing the embedding matrix of the shared part W_s and then computing the logit values Y_{vq} for all the words respectively, we first compute the logit values of all groups by the group embedding matrix W_b and then restore Y_{vq} by looking-up the group logit values with the code assignment matrix C. In another word, we use more efficient looking-up operations instead of multiplication operations with highly computational complexity. This is the key reason why our method can reduce the computational complexity.

Figure 2 shows how we compute the logit values over all the words in the vocabulary efficiently with P-VQ in the softmax layer at each position.

Curriculum Learning for Compression

When applying P-VQ algorithm, a strategy is necessary to compress the embedding matrix. In this subsection, we will discuss on this issue.

A good strategy for compression is quite important for vector quantization (VQ) methods, which significantly influences the compression quality and thus the performance of the whole model. For one compression step in previous VQ variations, Shi and Yu (2018) use k-means to learn the codebook and code assignment in each slice, while Shu and Nakayama (2017) and Chen, Min, and Sun (2018) use a straight-forward method based on the Gumble-Softmax reparameterization trick (Maddison, Mnih, and Teh 2016; Jang, Gu, and Poole 2016). They have achieved good effects in their respective tasks. However, they compress the word embedding matrix directly with a very large compression rate. This results in a big information gap between word embedding matrix and other parameters in the model, or worse yet, harms the model quality.

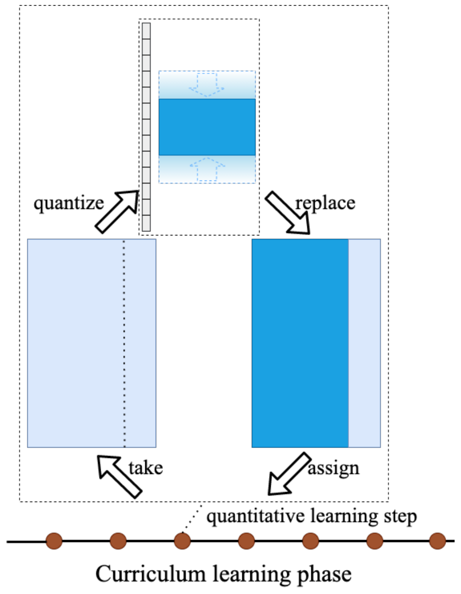

Figure 3: Curriculum learning phase. In each quantitative learning step, we first use P-VQ to compress the shared part and then replace the shared part by the quantization result.

Inspired by curriculum learning which is widely used in machine learning area (Han, Mao, and Dally 2015; Pentina, Sharmanska, and Lampert 2015; Matiisen et al. 2019) especially in machine translation (Platanios et al. 2019; Kocmi and Bojar 2017), we set the compression rate from small to large during the curriculum learning phase (i.e. compression phase) and compress the embedding matrix gradually to help training process.

To divide words into groups by the word embeddings, a pre-trained embedding matrix is required, which means we first need to pre-train a base model. Then we compress the pre-trained embedding matrix by curriculum learning.

In curriculum learning phase, to make the information gap small, we start with a small compression rate and set a big cluster number k_{begin} as current cluster number k_{curr}. Then we decrease k_{curr} by $step_k$ gradually until k_{curr} reaches the expected target number k_{end}. We implement quantitative learning every $step_c$ step. In each quantitative learning step, we first take the word embedding matrix from the training model. Second, we quantize the shared part by clustering words into k_{curr} groups to learn the codebook and code assignment. Third, we replace values of the shared part by looking-up the codebook with the code assignment. Finally, we assign the new embedding values to the word embedding

Algorithm 1: Curriculum Learning for Compression

```
params: Embedding matrix $W = f^{concat}(W_s, W_e)$;
        Other parameters $P_o$;
        Codebook $W_b$;
        Code assignment $I$;
        Total learning step $step_{max}$;
        Cluster step $step_c$;
        Begin cluster number $k_{begin}$;
        End cluster number $k_{end}$;
        Decay step of cluster number $step_k$.
while pre-training do
    $train(W_s, W_e, P_o)$;
end
$k_{curr} = k_{begin}$;
$step_{curr} = 0$;
while curriculum learning do
    if $step_{curr} \% step_c == 0$ then
        $W_b, I = f^{cluster}(W_s, k_{curr})$;   $//quantize$
        $W_s = f^{lookup}(W_b, I)$;              $//replace$
        if $k_{curr} > k_{end}$ then
            $k_{curr} = f^{max}(k_{curr} - step_k, k_{end})$;
        end
    end
    $train(W_s, W_e, P_o)$;
    $step_{curr} + +$;
    if $step_{curr} \geq step_{max}$ then
        break;
    end
end
while fine-tuning do
    $train(W_b, W_e, P_o)$;
end
```

matrix in the training model. The other training steps between two quantitative learning steps are just the same with pre-training. The word embedding matrix stays the uncompressed form (Eq. (6)) during curriculum learning phase. After curriculum learning, we first change the word embedding matrix into the compact form (Eq. (8)). Then we fix the code assignment and fine-tune the codebook to better fit the other parameters.

Figure 3 shows the operations in quantitative learning step during curriculum learning phase while Algorithm 1 shows the details through the pseudo code. In Algorithm 1, $f^{cluster}(a, b)$ means to cluster elements into b groups by the element weight matrix a, and return back the centroids and the group ids of all the elements, while $f^{max}(a, b)$ means to return back the bigger value between a and b. The operation $train(**args)$ means to update parameters in the parameter set $args$ by common step loss.

Experiments

To test our method, we conducted a set of experiments on the Chinese-to-English translation task and the English-to-French translation task. We report the experimental results of Chinese-to-English translation task in the main part while another in appendix. In the main part, we mainly compare the embedding matrix compression rate, the computational complexity reduction rate of the softmax layer, and the translation quality among different methods, measured by the number of parameters, the FLOPs, and the BLEU-4 (Papineni et al. 2002) score respectively. Please note that we only focus on the target embedding matrix and the softmax layer when measuring the number of parameters and the FLOPs.

Corpus

We use the Chinese-to-English WMT corpora as our training corpus. In data preparation phase, we clean the corpus by erasing some noisy characters and deleting sentences with length rules. After data cleaning, the corpus contains about 30 million Chinese-to-English parallel sentences. In vocabulary preparation phase, we first use the Moses tokenizer (Koehn et al. 2007) for subword tokenization of the English sentences while using the in-house tool for the Chinese sentence segmentation. Then we use BPE (Sennrich, Haddow, and Birch 2015) for the vocabulary reduction to the best of our ability. We set the BPE code size to $15,000$ for both source and target language in the BPE code learning step. After applying BPE to the corpus, we build the vocabulary with the size of $25,000$ for the source sentences and $20,000$ for the target sentences. In the evaluation step, we use WMT17, WMT18, and WMT19 test sets to evaluate the translation quality for each method.

Implementation

We use the standard Transformer (Vaswani et al. 2017) as implemented in OpenNMT-tf (Klein et al. 2017; Abadi et al. 2016) with 6 layers, 512 embedding dimensions and 8 attention heads as our baseline model. Due to the low overlap between the vocabularies of the source and target side, we require two word embedding matrices for the two sides respectively, and reuse the target word embedding matrix in the softmax layer.

We first train a baseline model as the pre-trained model. Then we only implement the compression methods to the target word embedding matrix, to test the effects of these methods in both compression and computational complexity reduction. For those methods requiring a pre-trained embedding matrix, we implement them on the pre-trained model separately and then fine-tune these models. Implementation details are as follows:

- Factorized Embedding Parameterization (FEP) (Lan et al. 2019): the lower rank is set to 128. We first train the model with FEP from scratch. To improve its translation quality, we further use the baseline model as the teacher and fine-tune it by knowledge distillation (Kim and Rush 2016).
- Structure Pruning (SP) (Anwar, Hwang, and Sung 2017): the final pruning rate of the pre-trained embedding matrix is set to 75%. During pruning phase, we adopt curriculum learning and set the learning step number T to 10000. We prune the hidden dimensions at every 10 steps. At each pruning step, the pruning rate is updated by $75\% \times t/T$, where t is the current step number.

	Param.		**FLOPs**		**Speed** (tok/s)		**BLEU** $(\%)$			
	$N(\times 10^6)$	$r(\%)$	$N(\times 10^6)$	$r(\%)$	GPU	CPU	WMT17	WMT18	WMT19	$\bar{\Delta}$
Baseline	10.24	-	20.48	-	52.98	8.41	24.23	23.50	25.68	-
FEP	2.63	74.35	5.38	73.72	0.95×	1.67×	22.68	20.88	23.12	-2.24
+kd	-	-	-	-	-	-	23.67	22.02	24.52	-1.06
SP	2.56	75	5.25	74.36	0.97×	1.65×	23.89	23.01	24.76	-0.58
DC	3.38	67.03	5.49	73.19	0.52×	1.10×	23.64	22.40	24.28	-1.03
LR	0.11	**98.89**	0.32	**98.44**	0.99×	**1.89**×	19.53	18.15	19.68	-5.35
SE	2.72	73.42	5.26	74.32	0.83×	1.41×	23.65	22.99	24.83	-0.64
P-VQ (ours)	2.63	**74.35**	5.24	**74.42**	**1.02**×	**1.76**×	**24.26**	**23.54**	**25.48**	**-0.04**

Table 1: Measurements over different methods. $N(\times 10^6)$ means the number of the parameters or FLOPs while $r(\%)$ means the reduction rate. Speed is the real translation speed of the whole model, while tok/s means how many output tokens per second when translating. $\bar{\Delta}$ means the average reduction of the BLEU score. DC, LR, SE and P-VQ (ours) are variant methods of vector quantization.

	code length	**weights** $\times 10^6$	**indices** $\times 10^6$	**Size** MB
DC	64	2.10	1.28	12.88
LR	2	0.07	0.04	**0.43**
SE	8	2.56	0.16	10.38
P-VQ (ours)	1	2.61	**0.02**	**10.02**

Table 2: Real compressed embedding sizes on the disk.

- Deep Composition (DC) (Shu and Nakayama 2017): the code length and the code depth are both set to 64. We follow the original work to learn the code assignment from the pre-trained embedding matrix.
- LightRNN (LR) (Li et al. 2016): the word allocation table size is set to 150×150. We follow the bootstrap procedure given in Li et al. (2016) to build the table. Different from the original work, due to the large difference between the model structures, we use the pre-trained embedding matrix to guide the table learning and only learn the word allocation table once. We use the minimum square error as the table learning loss.
- Structured Embedding (SE) (Shi and Yu 2018): the code length is set to 8 while 5000 for the code depth to ensure a comparable parameter number, and k-means++ (Arthur and Vassilvitskii 2006) as implemented in SciPy (Virtanen et al. 2020) is used to learn the code assignment by the pre-trained embedding matrix. When clustering, we run the cluster algorithm 10 times and use the best result, the max iteration in each run is 100.
- Partial Vector Quantization (P-VQ, our method): we set the window size $w = 384$ and the end cluster number $k_{end} = 128$. To get a balanced cluster result, we use the balanced k-means (Malinen and Fränti 2014) as our cluster algorithm, while the running settings are the same with in SE. For curriculum learning, we set the learning step $step_{max} = 10000$, the cluster step $step_c = 1000$, the begin cluster number $k_{begin} = 1024$, and the cluster number decay step $step_k = 128$.

In any training phase, we use the lazy Adam optimizer (Kingma and Ba 2014) with $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 10^{-9}$. We use the Noam decay as the learning rate scheduler with 4000 warmup steps and a factor of learning rate of 2.0. With batch size of 128 in the sentence level on 4 P40 GPUs, the training step for baseline and FEP is at least $300k$, the fine-tuning step is at least $200k$. We stop each model when the model loss is stable within $10k$ steps. In predicting phase, we set beam size to 4, and select the best BLEU score of the models in the last $10k$ steps to represent the model quality. The batch size is set to 1 when testing the real translation speed. The speed test environment configuration of GPU is Tesla P40 with CUDA 10.2 while that of CPU is GNU/Linux x86_64.

Results

Considering that FLOPs is not the only factor that affects the real inference time, we report the real translation speed on both GPU and CPU environments. Since GPUs have the capability of parallel computing which abates the impact of the matrix size on the efficiency in matrix multiplication, the actual acceleration effect in GPU environment is not obvious, or even worse because some extra operations have low efficiency in GPUs. For more concise expression, we use the abbreviations defined in subsection to represent each method. Table 1 shows the measurements over different methods.

According to the overview of Table 1, with approximately the same number of parameters and FLOPs, P-VQ (our method) has the fastest translation speed and the best translation quality among these comparable works. Although LR shows the best compression and acceleration rate, it sacrifices extremely high translation quality.

Comparison results over VQ variations show that the real translation speed in disproportion to the number of FLOPs. This problem is more serious in DC and SE. They both use a number of codes to represent words (64 in DC while 8 in SE), which means they both require the looking-up operation for multiple times. The complexity of the looking-up operation is $O(1)$ theoretically, but it seems that this operation cannot go down well with our test environments. Although it is possible to eliminate this effect by optimizing the bottom algorithm, the incompatibility is a potential trou-

	BLEU			
	WMT17	WMT18	WMT19	$\bar{\Delta}$
P-VQ	**24.26**	**23.54**	**25.48**	**-0.04**
w/o CL	23.82	22.93	24.90	-0.58

Table 3: BLEU score comparison.

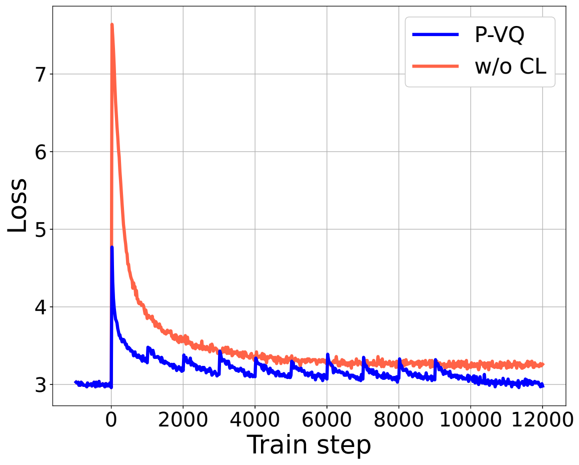

Figure 4: Training loss comparison.

ble when deploying these translation models to applications.

We also report the true storage size of the embedding matrices compressed by the VQ variations in Table 2. The floating points and integers are stored with the format of $float32$ and $int32$ respectively. Results show that, in VQ variations, when the code length increases, the code assignment matrix also occupies a relatively large memory.

In Table 1, we note that with curriculum learning, both SP and P-VQ perform better than other methods on each measurement. To show the influence of curriculum learning, we conduct another experiment about P-VQ by discarding the curriculum learning. In this experiment, we run only one quantitative learning step by the target cluster number $k_{end} = 128$ to compress the embedding matrix, and then fine-tune the compressed matrix with the other parameters. Detailed comparisons of the BLEU score and the training loss are reported in Table 3 and Figure 4 respectively.

Table 3 shows curriculum learning significantly improves the translation quality, which can be explained in Figure 4. In Figure 4, steps before 0 are the pre-training steps. As shown in Figure 4, without curriculum learning, because of the large compression rate, a huge information gap occurs in the quantitative learning step (step 0). Even after a long fine-tuning, the embedding parameters still cannot well fit the model. With curriculum learning, when we run the quantitative learning step at every 1000 steps, an information gap also occurs. But these gaps are smaller and are soon eliminated with training.

We also compare the BLEU scores of P-VQ without curriculum learning (P-VQ w/o CL in Table 3) and VQ variations (DC, SE in Table 1). Result shows that, in terms of approximately the same number of parameters, with the exclusive part, even the simplest VQ achieves a higher model quality. However, previous works already show that the performance of these VQ variations is much better than that of simple VQ. This confirms the importance of the word uniqueness, and also reflects the parameter oscillation problem we were worried about when using these methods in the softmax layer.

w	Param. $\times 10^6$	FLOPs $\times 10^6$	BLEU WMT17
128	7.73	15.45	24.36
256	5.21	10.39	24.30
384	**2.68**	**5.34**	**24.20**
448	1.41	2.81	23.47

Table 4: Impact of window size w on model performance.

k	Param. $\times 10^6$	FLOPs $\times 10^6$	BLEU WMT17
512	2.78	5.53	24.32
256	2.68	5.34	24.20
128	**2.63**	**5.24**	**24.26**
64	2.60	5.19	24.04
32	2.59	5.16	23.28

Table 5: Impact of group number k on model performance.

Furthermore, we analyze the impact of the two hyper-parameters in P-VQ, namely, the group number k and the window size w. We fix k in Table 4 to 256 and w in Table 5 to 384. As revealed by Table 4, the BLEU score remains stable during w is set from 128 to 384 and has a sharp drop when w raise to 448, which means a suitable w can well balance the model performance. Table 5 further shows that k has a smaller effect than w on the parameter number and FLOPs, which is reasonable according to Eq. (7).

Conclusion

In this work, we propose a simple but very effective method, the Partial Vector Quantization, to address the problem brought by large vocabulary in neural machine translation models. Based on partial compression, our method can reduce the computational complexity of the softmax layer by replacing multiplication operations with the efficient looking-up operations. To improve the compression quality, we follow the idea of curriculum learning and compress the embedding matrix gradually. Experimental results show that our method can significantly reduce the number of both the embedding parameters and the FLOPs of the softmax layer while keeping a high translation quality.

Acknowledgements

We would like to thank anonymous reviewers for their constructive comments. This work is partially supported by NSFC grant No.61971382.

Appendix

Different from the Chinese-to-English translation task, some translation tasks share a joint vocabulary between source and target languages (i.e., English and French) because of the high overlap between their vocabularies after tokenization. To verify the effectiveness of our method, we further evaluate our method on one of these tasks - the English-to-French translation task.

Corpus

We use English-to-French WMT corpora as our training corpus. In data preparation phase, we first clean the corpus by deleting sentences with length rules. After data cleaning, the corpus contains about 34 million English-to-French parallel sentences. We use Moses tokenizer for subword tokenization of the parallel sentences. To build the joint vocabulary, we set the BPE code size to 30,000 in the BPE code learning step and learn the BPE codes by all the parallel sentences. After applying BPE to the corpus, we get the joint vocabulary from the processed corpus with the size of 35,106. In the evaluation step, we use WMT14 test set to evaluate the translation quality.

Implementation

Compared with the Chinese-to-English translation task, the differences in implementation are as follows: (i) the English-to-French translation task requires only one word embedding matrix for the joint vocabulary and also reuse it in the softmax layer (which is also called the three-way weight tying method); (ii) for curriculum learning, we set the learning step $step_{max} = 30000$, the cluster step $step_c = 2000$. The other settings are the same as before ($w = 384$, $k_{end} = 128$, $k_{begin} = 1024$, $step_k = 128$).

Results

Experimental results of the English-to-French translation task are shown in Table 6. As the embedding size is larger than the Chinese-to-English translation task, our method in this task shows larger improvement in both compression and acceleration.

Due to the three-way weight tying method, the quality of the embedding compression has great impact on the model performance. As shown in Table 6, the translation quality only drops slightly in the English-to-French translation task after compression, which also verifies the effectiveness of our method.

		Baseline	**P-VQ (ours)**
Param.	$N(\times 10^6)$	17.97	4.57
	$r(\%)$	-	74.54
FLOPs	$(\times 10^6)$	35.95	9.12
	$r(\%)$	-	74.63
Speed	GPU	51.39	1.06×
(tok/s)	CPU	6.45	1.96×
BLEU	WMT14	39.87	39.58
	$\bar{\Delta}$	-	-0.29

Table 6: Evaluation of P-VQ on English-to-French translation task.

References

Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 265–283.

Anwar, S.; Hwang, K.; and Sung, W. 2017. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 13(3): 1–18.

Arthur, D.; and Vassilvitskii, S. 2006. k-means++: The advantages of careful seeding. Technical report, Stanford.

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

Bahdanau, D.; Cho, K.; and Bengio, Y. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* .

Bengio, Y.; Louradour, J.; Collobert, R.; and Weston, J. 2009. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, 41–48.

Burton, D. K.; Shore, J. E.; and Buck, J. T. 1983. A generalization of isolated word recognition using vector quantization. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP '83.*

Chen, T.; Min, M. R.; and Sun, Y. 2018. Learning k-way d-dimensional discrete codes for compact embedding representations. *arXiv preprint arXiv:1806.09464* .

Chen, W.; Grangier, D.; and Auli, M. 2015. Strategies for training large vocabulary neural language models. *arXiv preprint arXiv:1512.04906* .

Faundez-Zanuy, M. 2007. On-line signature recognition based on VQ-DTW. *Pattern Recognition* 40(3): 981–992.

Gersho, A.; and Gray, R. M. 2012. *Vector quantization and signal compression*, volume 159. Springer Science & Business Media.

Goodman, J. 2001. Classes for fast maximum entropy training. In *2001 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No. 01CH37221)*, volume 1, 561–564. IEEE.

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

Hunger, R. 2005. *Floating point operations in matrix-vector calculus.* Munich University of Technology, Inst. for Circuit Theory and Signal . . . .

Inan, H.; Khosravi, K.; and Socher, R. 2016. Tying word vectors and word classifiers: A loss framework for language modeling. *arXiv preprint arXiv:1611.01462* .

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

Jegou, H.; Douze, M.; and Schmid, C. 2010. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33(1): 117–128.

Kim, Y.; and Rush, A. M. 2016. Sequence-level knowledge distillation. *arXiv preprint arXiv:1606.07947* .

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

Klein, G.; Kim, Y.; Deng, Y.; Senellart, J.; and Rush, A. M. 2017. Opennmt: Open-source toolkit for neural machine translation. *arXiv preprint arXiv:1701.02810* .

Kocmi, T.; and Bojar, O. 2017. Curriculum learning and minibatch bucketing in neural machine translation. *arXiv preprint arXiv:1707.09533* .

Koehn, P.; Hoang, H.; Birch, A.; Callison-Burch, C.; Federico, M.; Bertoldi, N.; Cowan, B.; Shen, W.; Moran, C.; Zens, R.; et al. 2007. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th annual meeting of the ACL on interactive poster and demonstration sessions*, 177–180. Association for Computational Linguistics.

Lan, Z.; Chen, M.; Goodman, S.; Gimpel, K.; Sharma, P.; and Soricut, R. 2019. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942* .

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

Liu, X.; Wong, D. F.; Liu, Y.; Chao, L. S.; Xiao, T.; and Zhu, J. 2019. Shared-private bilingual word embeddings for neural machine translation. *arXiv preprint arXiv:1906.03100* .

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

Malinen, M. I.; and Fränti, P. 2014. Balanced k-means for clustering. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*, 32–41. Springer.

Matiisen, T.; Oliver, A.; Cohen, T.; and Schulman, J. 2019. Teacher-student curriculum learning. *IEEE transactions on neural networks and learning systems* .

Mikolov, T.; Kombrink, S.; Burget, L.; Černockỳ, J.; and Khudanpur, S. 2011. Extensions of recurrent neural network language model. In *2011 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, 5528–5531. IEEE.

Molchanov, P.; Tyree, S.; Karras, T.; Aila, T.; and Kautz, J. 2016. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440* .

Morin, F.; and Bengio, Y. 2005. Hierarchical probabilistic neural network language model. In *Aistats*, volume 5, 246–252. Citeseer.

Oda, Y.; Arthur, P.; Neubig, G.; Yoshino, K.; and Nakamura, S. 2017. Neural machine translation via binary code prediction. *arXiv preprint arXiv:1704.06918* .

Papineni, K.; Roukos, S.; Ward, T.; and Zhu, W.-J. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, 311–318. Association for Computational Linguistics.

Pentina, A.; Sharmanska, V.; and Lampert, C. H. 2015. Curriculum learning of multiple tasks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 5492–5500.

Platanios, E. A.; Stretcu, O.; Neubig, G.; Poczos, B.; and Mitchell, T. M. 2019. Competence-based curriculum learning for neural machine translation. *arXiv preprint arXiv:1903.09848* .

Press, O.; and Wolf, L. 2016. Using the output embedding to improve language models. *arXiv preprint arXiv:1608.05859* .

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

Shi, K.; and Yu, K. 2018. Structured Word Embedding for Low Memory Neural Network Language Model. In *Interspeech*, 1254–1258.

Shu, R.; and Nakayama, H. 2017. Compressing word embeddings via deep compositional code learning. *arXiv preprint arXiv:1711.01068* .

Soong, F. K. 1985. A Vector Quantization Approach to Speaker Recognition. *Proc Icassp* 11(2): 387–390.

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

Virtanen, P.; Gommers, R.; Oliphant, T. E.; Haberland, M.; Reddy, T.; Cournapeau, D.; Burovski, E.; Peterson, P.; Weckesser, W.; Bright, J.; et al. 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods* 17(3): 261–272.