From Fully Trained to Fully Random Embeddings: Improving Neural Machine Translation with Compact Word Embedding Tables

Krtin Kumar[1*], **Peyman Passban**[2*], **Mehdi Rezagholizadeh**[3], **Yiu Sing Lau**[4*], **Qun Liu**[3]

[1] Thomson Reuters,
[2] Amazon,
[3] Huawei Noah's Ark Lab,
[4] McGill University,
{krtinkumar, passban.peyman, mehdi.rezagholizadeh, yiusinglau17}@gmail.com, qun.liu@huawei.com

Abstract

Embedding matrices are key components in neural natural language processing (NLP) models that are responsible to provide numerical representations of input tokens (i.e. words or subwords). In this paper, we analyze the impact and utility of such matrices in the context of neural machine translation (NMT). We show that detracting syntactic and semantic information from word embeddings and running NMT systems with random embeddings is not as damaging as it initially sounds. We also show how incorporating only a limited amount of task-specific knowledge from fully-trained embeddings can boost the performance NMT systems. Our findings demonstrate that in exchange for negligible deterioration in performance, any NMT model can be run with partially random embeddings. Working with such structures means a minimal memory requirement as there is no longer need to store large embedding tables, which is a significant gain in industrial and on-device settings. We evaluated our embeddings in translating English into German and French and achieved a 5.3x compression rate. Despite having a considerably smaller architecture, our models in some cases are even able to outperform state-of-the-art baselines.

Introduction

One of the main challenges in NLP is to properly encode discrete tokens into continuous vector representations. The most common practice in this regard is to use an embedding matrix which provides a one-to-one mapping from tokens to n-dimensional, real-valued vectors (Mikolov et al. 2013; Li and Yang 2018). Typically, values of these vectors are optimized via back-propagation with respect to a particular objective function. Learning embedding matrices with robust performance across different domains and data distributions is a complex task, and can directly impact quality (Tian et al. 2014; Shi et al. 2015; Sun et al. 2016).

Embedding matrices are a key component in a seq2seq NMT model, for instance, in a transformer-base NMT model (Vaswani et al. 2017) with a vocabulary size of $50k$, 36% of the total model parameters are utilized by embedding matrices. Thus, a significant amount of parameters are utilized only for representing tokens in a model. Existing work on embedding compression for NMT systems (Khrulkov et al. 2019; Chen et al. 2018; Shu and Nakayama 2017), have shown that these matrices can by compressed significantly with minor drop in performance. Our focus in this work is to study the importance of embedding matrices in the context of NMT. We experiment with Transformers (Vaswani et al. 2017) and LSTM based seq2seq architectures, which are the two most popular seq2seq models in NMT. In Section , we compare the performance of a fully-trained embedding matrix, with a completely random word embedding (RWE), and find that using a random embedding matrix leads to drop in about 1 to 4 BLEU (Papineni et al. 2002) points on different NMT benchmark datasets. Neural networks have shown impressive performance with random weights for image classification tasks (Ramanujan et al. 2020), our experiments show similar results for embedding matrices of NMT models.

RWE uses no trainable parameters, thus it might be possible to recover the drop in BLEU score by increasing the number of parameters using additional layers. We increase the number of layers in RWE model such that the number of parameters used by the entire model is comparable to fully-trained embedding model. We find that even in comparable parameter setting RWE model performance was inferior to fully-trained model by 1 BLEU point for high and medium resource datasets. Our results suggest that even though the embedding parameters can be compressed to a large extent with only a minor drop in accuracy, embedding matrices are essential components for token representation, and cannot be replaced by deeper layers of transformer based models.

RWE assumed a fully random embedding sampled from a single Gaussian distribution; however, to control the amount of random information, and to better understand the importance of embedding matrices, we introduce *Gaussian product quantization* (GPQ). GPQ assumes that k Gaussian distributions are required to approximate the embedding matrix for NMT models. The means and variances of the k distributions are learned from a fully-trained embedding matrix trained on the same dataset as GPQ model. k is a hyper-parameter which controls the amount of information distilled from a pre-trained embedding to partially random embedding in GPQ model. GPQ has the ability to move from a fully random embedding to a fully-trained embedding by

[*]Work done while Krtin Kumar, Peyman Passban and Yiu Sing Liu were at Huawei.

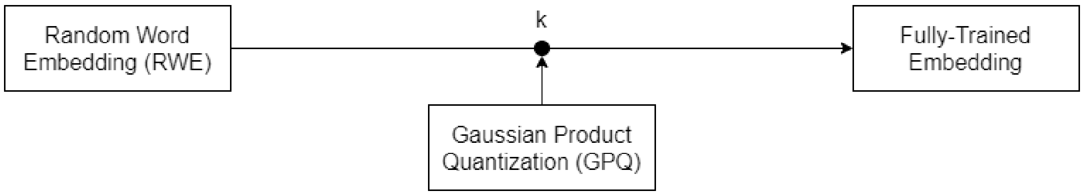
Random Word Embedding (RWE)
k
Fully-Trained Embedding
Gaussian Product Quantization (GPQ)

Figure 1: k is a hyper-parameter that controls the number of Gaussian distributions required to approximate the embedding matrix.

increasing the number of distributions k, as shown in Figure 1. Our results show that only 50 Gaussian distributions are sufficient to approximate the embedding matrix, without drop in performance. GPQ compresses the embedding matrix 5 times without any significant drop in performance, and uses only 100 floating point values for storing the means and variances. GPQ demonstrates effective regularization of the embedding matrix, by out-performing the transformer baseline model with fully-trained embedding by 1.2 BLEU points for $En \rightarrow Fr$ dataset and 0.5 BLEU for $Pt \rightarrow En$ dataset. A similar increase in performance was also observed for LSTM based models, thus further showing the effectiveness of GPQ for embedding regularization.

Product Quantization (PQ) (Jegou, Douze, and Schmid 2010) was proposed for fast search by approximating nearest neighbour search. PQ has been adapted for compressing embedding matrices for different NLP problems (Shu and Nakayama 2017; Kim, Kim, and Lee 2020; Tissier, Gravier, and Habrard 2019; Li et al. 2018). Our method is an extension of PQ, first, we incorporate variance information in GPQ, then we define *unified partitioning* to learn a more robust shared space for approximating the embedding matrix. Our extensions consistently outperform the original PQ-based models with better compression rates and higher BLEU scores. These improvement are observed for Transformer-based models as well as conventional, recurrent neural networks.

Product Quantization (PQ)

PQ is the core of our techniques, so we briefly discuss its details in this section. For more details see Jegou, Douze, and Schmid (2010). As previously mentioned, NLP models encode tokens from a discrete to a continuous domain via an embedding matrix E, as simply shown in Equation 1:

$$E \in \mathbb{R}^{|V| \times n} \tag{1}$$

where V is a vocabulary set of unique tokens and n is the dimension of each embedding. We use the notation $E_w \in \mathbb{R}^n$ to refer to the embedding of the w-th word in the vocabulary set. In PQ, first E is partitioned into g groups across columns with $G_i \in \mathbb{R}^{|V| \times \frac{n}{g}}$ representing the i-th group. Then each group G_i is clustered using K-means into c clusters. Cluster indices and cluster centroids are stored in the vector $Q_i \in \mathbb{N}^{|V|}$ and matrix $C_i \in \mathbb{R}^{c \times \frac{n}{g}}$, respectively. Figure 2 illustrates the PQ-based decomposition process.

We can obtain the quantization matrix Q (also known as the *index matrix* in the literature) and *codebook* C by

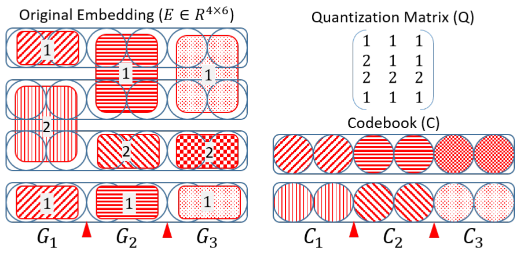

Figure 2: The matrix on the left hand side shows the original embedding matrix E, after dividing columns into 3 groups ($G_i; i \in \{1, 2, 3\}$) and applying the k-means algorithm with the cluster number 2 ($c = 2$) to each group. The digit inside each block is the cluster number. The figure on the right hand side shows the quantization matrix Q, and codebook C.

concatenating Q_i and C_i for all g groups along the columns, as shown in Equation 2.

$$\begin{aligned} Q &= \text{Concat}_{\text{column}}(Q_1, Q_2, ..., Q_g) \\ C &= \text{Concat}_{\text{column}}(C_1, C_2, ..., C_g) \end{aligned} \tag{2}$$

The decomposition process applied by PQ is reversable, namely any embedding table decomposed by PQ can be reconstructed using the matrices Q_i and C_i. The size of an embedding table compressed via PQ can be calculated as shown in Equation 3:

$$\text{Size}_{\text{PQ}} = \log_2(c) \times |V| \times g + cnf_p \text{ (in bits)} \tag{3}$$

where f_p is the number of bits defined to store parameters with floating points. In our experiments f_p is 32 for all settings. We use Equation 3 to measure the compression rate of different models.

Methodology

In this section we first explain RWE which stands for Random Word Embeddings. In RWE, all embeddings are initialized with completely random values where there is no syntactic and semantic information is available.

Next, we move from completely random embeddings to weakly supervised embeddings by incorporating some knowledge from a pre-trained embedding matrix. We propose our Gaussian Porduct Quantization (GPQ) technique in this regard, which applies PQ to a pre-trained embedding matrix and models each cluster with a Gaussian distribution. Our GPQ empowers the PQ with taking intra-cluster variance information into account. This approach is particularly useful when PQ clusters have a high variance, which might be the case for large embedding matrices.

Random Word Embeddings (RWE)

A simple approach to form random embeddings is to sample their values from a standard Gaussian distribution, as shown in Equation 4:

$$S = \mathcal{N}(E_{i,j}|\mu, \sigma^2)\ \forall(i,j)$$
$$E_i^{'} = \frac{S_i}{||S_i||}\ \forall i, \quad (4)$$

where $\mathcal{N}$ is a normal distribution with $\mu = 0$ and $\sigma = 1$, S is sampled matrix of the same dimensions as E, E' is the reconstructed matrix, E_i and S_i represent i^{th} row vector of matrices E' and S respectively. We normalize each word embedding vector to unify the magnitude of all word embeddings to ensure that we do not add any bias to the embedding matrix.

Since this approach relies on completely random values, to increase the expressiveness of embeddings and handle any potential dimension mismatch between the embedding table and the first layer of the neural model we place an *optional* linear transformation matrix W (where $W \in \mathbb{R}^{n\times m}$). The weight matrix W uses $n \times m$ trainable parameters, where n is the size of embedding vector and m is the dimension in which the model accepts its inputs. The increase in the number of parameters is negligible as it only relies on the embedding and input dimensions (n and m), which are both considerably smaller than the number of words in a vocabulary ($n, m \ll |V|$).

Gaussian Product Quantization (GPQ)

RWE considers completely random embeddings which can be a very strict condition for an NLP model. Therefore, we propose our Gaussian Product Quantization (GPQ) to boost random embeddings with prior information from a pre-trained embedding matrix. We assume that there is an existing embedding matrix E that is obtained from a model with the same architecture as ours, trained for the same task using the same dataset.

The pipeline defined for GPQ is as follows: First, we apply PQ to the pre-trained embedding matrix E to derive the quantization matrix $Q_{|V|\times g}$ and the codebook matrix $C_{c\times n}$. The codebook matrix stores centers of c clusters for all G_i groups and the quantization matrix stores the mapping index of each sub-vector in the embedding matrix to the corresponding center of the cluster in the codebook. The two matrices Q and C can be used to reconstruct the original matrix E from the PQ process.

The PQ technique only stores the center of clusters in the codebook and does not take variance of them into account. However, our GPQ technique models each cluster with a single Gaussian distribution by setting its mean and variance parameters to the cluster center and intra-cluster variance. We define $\mathcal{C}_i^j, (1 \le j \le c)$ to be the cluster corresponding to the $j-$ th row of C_i in the codebook (that is C_i^j) with the mean and variance of $\mu_i^j \in \mathbb{R}^{\frac{c}{n}}$ and $(\sigma_i^j)^2$. Then, we define entries of the codebook of our GPQ approach as following:

$$\hat{C}_i^j \sim \mathcal{N}_i^j(\mu_i^j, (\sigma_i^j)^2). \quad (5)$$

Consequently, we model each cluster of PQ with a single Gaussian distribution with two parameters. Then the embedding matrix $\hat{E}$ can be reconstructed using a lookup function that maps values Q_i from $\hat{C}_i$. We illustrate our GPQ method and compare it with PQ in Figure 3. In GPQ, we only need to store the index matrix, codebook and their corresponding variances in order to be able to reconstruct the embedding matrix.

The PQ/GPQ method relies on partitioning the embedding matrix into same size G_i groups across the columns and then clustering them. We refer to the partitioning function used in PQ as *Structured Partitioning*, and propose a new partitioning scheme, which we refer to as *Unified Partitioning*. The details of each scheme is described in the following.

Structured Partitioning The original PQ method is based on structured partitioning to partition the input matrix $E_{|V|\times n}$ into g groups of G_i of size $|V| \times \frac{n}{g}$ along the columns uniformly such that $E = \text{Concat}_{\text{column}}(G_1, G_2, ..., G_g)$. Each G_i group is clustered into c clusters along the rows using any clustering algorithm. If we choose K-means clustering algorithm for this purpose, then we can obtain the center of clusters $C_i \subset C$ (where $C_i \in \mathbb{R}^{c\times \frac{n}{g}}$), their variances $\sigma_i^2 \in \mathbb{R}^c$, and the quantization vector $Q_i \in \mathbb{N}^{|\mathbb{V}|}$ corresponding to G_i as following:

$$[C_i, \sigma_i^2, Q_i] = \text{K-means}(G_i, c). \quad (6)$$

The total number of clusters in this case is $k = c * g$. In our technique, we store the variance of clusters in addition to their mean which utilizes more number of floating point parameters compared to Equation 3, as shown in Equation 7.

$$\text{Size}_{\text{GPQ}}^{\text{Structured}} = \log_2(c) \times |V| \times g + 2cnf_p \text{ bits} \quad (7)$$

Unified Partitioning *Structured* partitioning has limited ability to exploit redundancies across groups. Motivated by this shortcoming we propose our *unified* partitioning approach in which we concatenate the g groups along rows to form a matrix $\mathcal{G} \in \mathbb{R}^{g|V|\times \frac{n}{g}}$ and then apply K-Means to the matrix $\mathcal{G}$ (instead of applying K-means to each group G_i separately).

$$\mathcal{G} = \text{Concat}_{row}(G_1, G_2, ..., G_g) \in \mathbb{R}^{(g|V|)\times(n/g)} \quad (8)$$
$$[C, \sigma^2, Q] = \text{K-means}(\mathcal{G}, c). \quad (9)$$

For better understanding we illustrate our method in Figure 4. Equation 10 defines the size of parameters for *unified* partitioning function.

$$\text{Size}_{\text{GPQ}}^{\text{Unified}} = \log_2(c) \times |V| \times g + 2c\frac{n}{g}f_p \text{ bits} \quad (10)$$

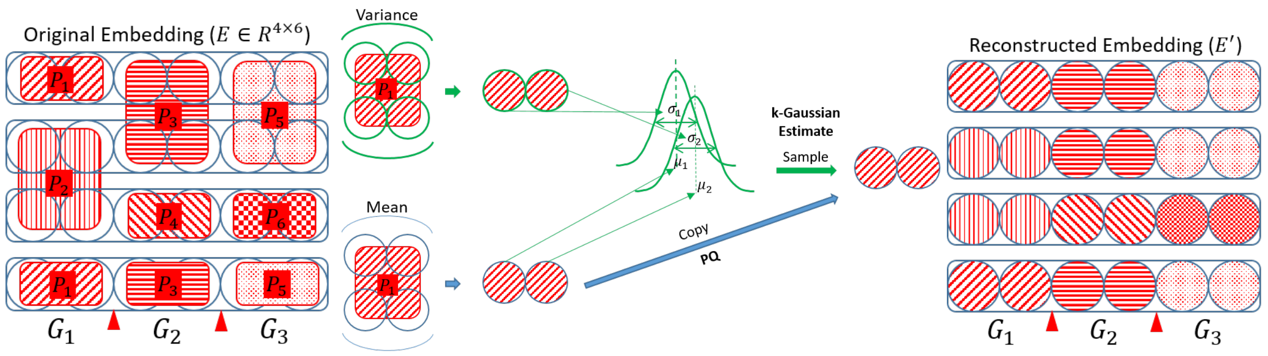

Figure 3a Figure 3b Figure 3c

Figure 3: The figure on the left hand side shows the original embedding matrix E after applying the k-means algorithm, with 2 clusters ($c = 2$) and 3 groups (G_i) ($g = 3$), which results in 6 partitions (P_i). The figure in the middle shows the difference between PQ and GPQ. The figure on the right hand side is the reconstructed matrix $E^{'}$.

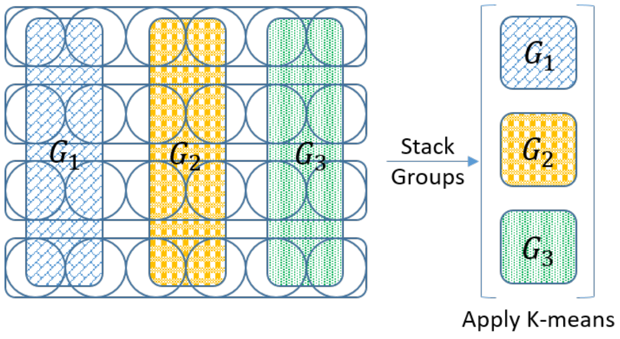

Figure 4: *Unified* partitioning function with 3 Groups.

Model	Float	Integer
PQ	25.6k	16.3M
PQ (Unified)	50	16.3M
GPQ (Unified)	100	16.3M

Table 1: Comparison of floating point vs integer parameters for vocabulary size of 32k, embedding size 512, 512 groups, and 50 clusters

Experiments

We evaluate our model for machine translation task on WMT-2014 English to French ($En \to Fr$), WMT-2014 English to German ($En \to De$), and IWSLT Portuguese to English ($Pt \to En$) datasets. We chose these pairs as they are good representatives of high, medium, and low resource language pairs.

For $En \to Fr$ experiments we used Sentence-Piece (Kudo and Richardson 2018) to extract a shared vocabulary of 32k sub-word units. We chose *newstest2013* as our validation dataset and used *newstest2014* as our test dataset. The dataset contains about 36M sentence-pairs.

For $En \to De$ experiments we use the same setup as Vaswani et al. (2017). The dataset contains about 4.5M sentence-pairs. We use a shared vocabulary of 37k sub-word units extracted using Sentence-Piece.

For $Pt \to En$ experiments, we replicate dataset configuration of Tan et al. (2019) for individual models. Specifically, the dataset contains 167k training pairs. We used a shared vocabulary of 32k subwords extracted with Sentence-Piece.

Evaluation For all language pairs, we report case-sensitive BLEU score (Papineni et al. 2002) using SacreBLEU[1] (Post 2018). We train for 100K steps and save a checkpoint every 5000 steps for $En \to Fr$ and $En \to De$ language pairs. For $Pt \to En$ translation we train for 50K steps and save checkpoint every 2000 steps. We select the best checkpoints based upon validation loss, and average best 5 checkpoints for $En \to Fr$ and $En \to De$ language pairs, and average best 3 checkpoints for $Pt \to En$. We do not use checkpoint averaging for our LSTM experiments. We use beam search with a beam width of 4 for all language pairs.

Model and Training Details

We use the Transformer model (Vaswani et al. 2017) as our baseline for all our experiments. We also report results on LSTM-based models to study our hypothesis for RNN based architectures. Our LSTM-based model uses multiple layers of bidirectional-LSTM, followed by a linear layer on the last layer to combine the internal states of LSTM. The decoder uses multiple layers of uni-directional LSTM, the final output of decoder uses attention on encoder output for generating words. All our models use weight sharing between the embedding matrix and the output projection layer (Press and Wolf 2016).

For the *transformer-base* configuration, the model hidden size h is set to 512, the feed-forward hidden size d_{ff} is set to 2048. We use different number of layers for our experiments, instead of the default 6 layers. For *transformer-small* configuration the model hidden-size n is set to 256, the feed-forward hidden size d_{ff} is set to 1024. We use different number of layers for our experiments. For *transformer-small*, the dropout

[1]https://github.com/mjpost/sacreBLEU

Model	WMT En-Fr			WMT En-De			IWSLT Pt-En		
	Layers	Model Params	BLEU	Layers	Model Params	BLEU	Layers	Model Params	BLEU
Transformer	6	60.7M	38.41	6	63M	27.03	3	11.9M	39.58
RWE+linear	6	44.4M	35.76	6	44.2M	23.04	3	3.7M	38.14
RWE+linear	8	59M	37.11	8	58.9M	25.83	6	11.1M	**40.69**

Table 2: NMT results for Random Word Embeddings (RWE), for different hyper-parameters. **M** represents Millions, Transformer is the baseline model (Vaswani et al. 2017).

Model	WMT En-Fr		WMT En-De		IWSLT Pt-En	
	Emb. Size	BLEU	Emb. Size	BLEU	Emb. Size	BLEU
Transformer Baseline	62.5 MB	39.29	72.3 MB	27.38	31.3 MB	39.88
PQ (Structured)	11.82 MB	39.95	13.65 MB	27.04	5.9 MB	40.33
GPQ (Structured)	11.92 MB	40.04	13.75 MB	27.11	5.95 MB	40.16
PQ (Unified)	11.7 MB	40.02	13.54 MB	26.87	5.86 MB	39.34
GPQ (Unified)	11.7 MB	**40.51**	13.55 MB	**27.35**	5.86 MB	**40.36**

Table 3: NMT results on Transformer baseline (Vaswani et al. 2017), with 8 layers, groups (g) equal to the embedding size n, and 50 clusters (c).

configuration was set the same as Transformer Base. For LSTM experiments, we use 2 layers and the hidden size n is set to 256 for *LSTM-small*, and 7 layers and the hidden size n is set to 512 for *LSTM-large*. We use *transformer-base* for WMT datasets, and *transformer-small* for IWSLT dataset. We use *LSTM-small* for IWSLT and *LSTM-large* for WMT datasets. For all our experiments we set *Numpy* (Van Der Walt, Colbert, and Varoquaux 2011) seed to 0 and *PyTorch* (Paszke et al. 2017) seed to 3435.

All models are optimized using Adam (Kingma and Ba 2014) and the same learning rate schedule as proposed by (Vaswani et al. 2017). We use label smoothing with 0.1 weight for the uniform prior distribution over the vocabulary (Szegedy et al. 2015; Pereyra et al. 2017).

We train all our models on 8 NVIDIA V100 GPUs. Each training batch contains a set of sentence pairs containing approximately 6144 source tokens and 6144 target tokens for each GPU worker. For implementing our method we use the OpenNMT library (Klein et al. 2017), implemented in PyTorch.

Results

Random Word Embeddings

We replace the embedding matrix of the transformer model using RWE method, and summarize our results in Table 2. We evaluate our model in two cases.

In the first case, we use exactly the same hyper-parameters as our baseline model, and only replace the embedding matrix. In this case our model has only 2 embedding parameters to store the mean and variance of the Gaussian distribution, and an additional linear layer for re-scaling the embedding vectors. Our results show that without any embedding parameters, and a linear transformation, transformer model scores only 1.44 BLEU points below the baseline for $Pt \to En$. Deterioration in the case of $En \to Fr$ was approximately 1 BLEU point higher than $Pt \to En$, and for $En \to De$ the drop is 4 BLEU points.

In the previous case, our model used at least 27% fewer parameters compared to the baseline model. In order to compare RWE in a fair parameter setting, we increase the number of layers in Transformer to approximately match the total parameters with the baseline model. In this setting our model performed within 1.3 BLEU points for $En \to De$ and $En \to Fr$. On $Pt \to En$ experiments, our model scored 1.0 BLEU point higher than the baseline with slightly fewer parameters.

Overall, our experiments show that transformer based neural networks have the capacity to efficiently learn semantic and syntactic information in deeper layers, though utility of using embedding tables for token representation is beneficial and not redundant.

Gaussian Product Quantization

We apply the GPQ method on the embedding matrix of Transformer and LSTM-based models. We design a set of experiments to investigate, 1) if we can approximate the embedding matrix in an almost discrete space using integer values, 2) to study the compression ability of our method compared to PQ.

Experiments for Discrete Space Approximation We evaluate our method on Transformer (Table 3) and LSTM-based models (Table 4). For all experiments we chose the number of groups g equal to the size of embedding vector n. This allowed us to maximize the number of groups and in turn maximize the amount of discrete information (integer values) and minimize the amount of floating-point parameters required to reconstruct the embedding matrix. We experiment

Model	WMT En-Fr		WMT En-De		IWSLT Pt-En	
	Emb. Size	BLEU	Emb. Size	BLEU	Emb. Size	BLEU
LSTM Baseline	62.5 MB	32.61	72.3	22.17	31.3	35.12
PQ (Unified)	13.67 MB	32.05	15.8 MB	22.13	6.8 MB	35.35
GPQ (Unified)	13.67 MB	**33.82**	15.8 MB	**22.14**	6.8 MB	**35.58**

Table 4: NMT results on LSTM based model, we use 128 clusters (c) and groups (g) equal to embedding size n.

Model	WMT En-Fr			WMT En-De			IWSLT Pt-En		
	Emb. Size	BLEU	c	Emb. Size	BLEU	c	Emb. Size	BLEU	c
Transformer	62.5 MB	39.29	-	72.3 MB	27.38	-	31.3 MB	39.88	-
PQ (Struct.)	1.25 MB	39.13	140	1.44 MB	25.35	160	1.22 MB	40.6	256
GPQ (Struct.)	1.25 MB	39.18	102	1.44 MB	**26.08**	116	1.1 MB	41.1	128
PQ (Unified)	1.25 MB	39.29	1024	1.44 MB	25.83	1024	1.24 MB	40.62	1024
GPQ (Unified)	1.28 MB	**39.77**	1024	1.47 MB	25.96	1024	1.25 MB	**41.32**	1024

Table 5: NMT results on Transformer baseline (Vaswani et al. 2017), with 32 groups (g) and 8 layers. *c* represents the number of clusters. *Struct.* and *unified* represents structured and unified partitioning functions respectively.

using 50 clusters for transformer based model and 128 clusters for the LSTM model. Our results for the GPQ method with unified partitioning function show that with only 50 clusters for transformer model and 128 clusters for LSTM model, we are able to perform on par or better than baseline models. This implies that in all experiments we were able to approximate the embedding matrix with approximately $\approx 100\%$ integer values. Specifically, if $g = n$, our *unified* partitioning function used only $2c$ floating points (for storing the mean and variance clusters), while $structured$ partitioning function uses $2cn$ floating points. In Table 3, GPQ with *unified* partitioning function outperforms other models consistently across all datasets. For $En \rightarrow De$, our approach is the only method that performs on par with baseline model with 5x fewer embedding parameters.

In Table 4, we report LSTM experiments for only the *unified* partitioning function as it was the best performing method. For LSTM, GPQ (*Unified*) performed better than PQ (*Unified*) for $En \rightarrow Fr$ with about 1.77 BLEU, for other languages the difference was insignificant.

Experiments for Compression Analysis Our objective is to compare the capacity of GPQ method for compressing the embedding matrix, compared to PQ. We set the number of clusters to 1024 and groups to 32, as reducing the number of groups has greater affect on compression rate. We report the results in Table 5. We find that with unified partition function GPQ method performs significantly better than the Transformer baseline for $En \rightarrow Fr$ (+0.48) and $Pt \rightarrow En$ (+0.7) datasets. For $En \rightarrow DE$ GPQ (unified) performs only 0.03 BLEU points lower than the baseline. For all language pairs GPQ (unified) is the best performing model compared to PQ and other variants.

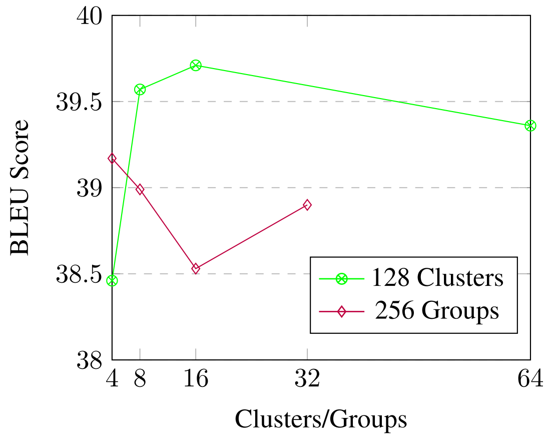

Figure 5: NMT Results on IWSLT Pt-En for GPQ method with unified partitioning function

Discussion

Cluster and Group Size Analysis

We study the effect of different cluster and group sizes on the GPQ method with *unified* partitioning function, on the $Pt \rightarrow En$ dataset and plot results in Figure 5. We experiment with fixing the groups to $g = 256$ and varying the cluster size. We find out that with only 4 clusters and only 8 floating point parameters our model performed the best. We also experiment with fixing the clusters and varying the group size. We find that group size of 4 was the worst performing setting as a result of excessive regularization. All group sizes larger than 4 had minor influence on performance. We find that we need only 8 floating point parameters, to approximate the embedding matrix without any performance drop. Thus most important information lies in a latent space which is discrete, consisting of integer values.

Model	WMT En-Fr		WMT En-De	
	Emb. Size	BLEU	Emb. Size	BLEU
Transformer Baseline	1.0x	38.41	1.0x	27.03
*SVD with rank 64	7.87x	37.44	7.89x	26.32
GroupReduce (Chen et al. 2018)	7.79x	37.63	7.88x	26.75
*Tensor Train (Khrulkov et al. 2019)	7.72x	37.27	7.75x	26.19
GPQ (Unified)	7.9x	**39.65**	7.9x	**26.84**

Table 6: NMT results on transformer based models, with 256 clusters (c), 256 groups (g) and 6 layers. Results in rows marked with * are taken from Lioutas et al. (2019)

Compression Analysis

We compare the performance of our model in a compression setting, with different compression baselines in Table 6. In particular, we compare results with two state-of-the-art methods (Khrulkov et al. 2019; Chen et al. 2018). Additionally, we compare two standard baselines reported in Lioutas et al. (2019). For all models we compare our results directly from Lioutas et al. (2019), and choose a similar compression ratio. For both $En \rightarrow Fr$ and $En \rightarrow De$ language pairs, our GPQ (unified) model performs the best compared to all other models. Additionally, our GPQ (unified) model is the only model that performs better than the baseline model for $En \rightarrow Fr$ language pair.

Importance of Variance

We extended PQ by introducing variance information in our GPQ method. Results in Table 4 highlight the importance of this information for LSTM based models. Table 6 shows that incorporating variance information can be beneficial for transformer based models in a high compression setup. Lastly, Table 3 shows that GPQ model is particularly beneficial when using $unified$ partitioning function. $Unified$ partitioning function uses much smaller number of total clusters, thus each cluster has a higher variance from the cluster centroid, compared to the case of $structured$ partitioning function. Thus, incorporating variance information using GPQ is beneficial, when cluster values have high variance.

Related Work

There are different embedding compression techniques in the literature, these methods are based on either singular value decomposition (SVD) or product quantization (PQ). In general, SVD-based methods (Chen et al. 2018; Khrulkov et al. 2019) approximate embedding matrices using fewer parameters by projecting high-dimensional matrices to lower dimensions. Therefore, the learning process is modified to work with new and smaller matrices where a multiplication of these matrices reconstruct original values. The model proposed by Chen et al. (2018) is a well-known example of SVD-based model in this field. In this technique, words are first divided into groups based upon frequency, then weighted SVD is applied on each group, which allows a compressed representation of the embedding matrix. Khrulkov et al. (2019) used a multi-linear variation instead of regular SVD. The embedding dimensions are first factorized to obtain indices of smaller tensor embedding. The tensor embedding has fewer parameters and can be used to reconstruct the original embedding matrix.

Shu and Nakayama (2017) proposed to compress the embedding matrix by learning an integer vector for all words in vocabulary (code matrix), using the Gumbel softmax technique. The code matrix is equivalent to the quantization matrix in PQ. An embedding vector is reconstructed by looking up all the values in the code matrix from the corresponding set of vector dictionaries. The vectors are added (in PQ the vectors are concatenated) to obtain the final embedding vector.

Svenstrup, Hansen, and Winther (2017) proposed a hashing based technique with a common memory for storing vectors. The method is similar to Shu and Nakayama (2017), with two key differences. First, the code matrix is not learned but assigned uniformly using a hashing function. Second, instead of a simple summation of component vectors, weighted summation is applied to obtain the final embedding vector.

(Kim, Kim, and Lee 2020) learn the code matrix by utilizing binarized code learning introduced in (Tissier, Gravier, and Habrard 2019). The key novelty introduced by Kim, Kim, and Lee (2020), is to learn different length of code vector for each word.

Conclusion

Our work points towards the need of rethinking the process of encoding tokens to real valued vectors for machine translation. We show that Transformer model is capable of recovering useful syntactic and semantic information from a random assignment of embedding vectors. Our variant of product quantization was able to approximate the embedding matrix in an almost 100% discrete space, with better performance than the baseline model. A discrete space is easier to interpret, compared to a continuous space, and can motivate future research to handle unknown tokens through optimum cluster selection.

References

Chen, P.; Si, S.; Li, Y.; Chelba, C.; and Hsieh, C.-J. 2018. Groupreduce: Block-wise low-rank approximation for neural language model shrinking. In *Advances in Neural Information Processing Systems*, 10988–10998.

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.

Khrulkov, V.; Hrinchuk, O.; Mirvakhabova, L.; and Oseledets, I. 2019. Tensorized embedding layers for efficient model compression. *arXiv preprint arXiv:1901.10787*.

Kim, Y.; Kim, K.-M.; and Lee, S. 2020. Adaptive Compression of Word Embeddings. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 3950–3959.

Kingma, D. P.; and Ba, J. 2014. Adam: A Method for Stochastic Optimization. arXiv:1412.6980.

Klein, G.; Kim, Y.; Deng, Y.; Senellart, J.; and Rush, A. M. 2017. OpenNMT: Open-Source Toolkit for Neural Machine Translation. In *Proc. ACL*.

Kudo, T.; and Richardson, J. 2018. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*.

Li, Y.; and Yang, T. 2018. Word embedding for understanding natural language: a survey. In *Guide to Big Data Applications*, 83–104. Springer.

Li, Z.; Kulhanek, R.; Wang, S.; Zhao, Y.; and Wu, S. 2018. Slim embedding layers for recurrent neural language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32.

Lioutas, V.; Rashid, A.; Kumar, K.; Haidar, M. A.; and Rezagholizadeh, M. 2019. Distilled embedding: non-linear embedding factorization using knowledge distillation. arXiv:1910.06720.

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

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.

Paszke, A.; Gross, S.; Chintala, S.; Chanan, G.; Yang, E.; DeVito, Z.; Lin, Z.; Desmaison, A.; Antiga, L.; and Lerer, A. 2017. Automatic differentiation in PyTorch. In *NIPS-W*.

Pereyra, G.; Tucker, G.; Chorowski, J.; Kaiser, L.; and Hinton, G. E. 2017. Regularizing Neural Networks by Penalizing Confident Output Distributions. *CoRR*, abs/1701.06548.

Post, M. 2018. A Call for Clarity in Reporting BLEU Scores. *Proceedings of the Third Conference on Machine Translation: Research Papers*.

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

Ramanujan, V.; Wortsman, M.; Kembhavi, A.; Farhadi, A.; and Rastegari, M. 2020. What's Hidden in a Randomly Weighted Neural Network? In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 11893–11902.

Shi, T.; Liu, Z.; Liu, Y.; and Sun, M. 2015. Learning cross-lingual word embeddings via matrix co-factorization. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, 567–572.

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

Sun, F.; Guo, J.; Lan, Y.; Xu, J.; and Cheng, X. 2016. Sparse word embeddings using l1 regularized online learning. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, 2915–2921. AAAI Press.

Svenstrup, D. T.; Hansen, J.; and Winther, O. 2017. Hash embeddings for efficient word representations. In *Advances in Neural Information Processing Systems*, 4928–4936.

Szegedy, C.; Vanhoucke, V.; Ioffe, S.; Shlens, J.; and Wojna, Z. 2015. Rethinking the Inception Architecture for Computer Vision. *CoRR*, abs/1512.00567.

Tan, X.; Ren, Y.; He, D.; Qin, T.; Zhao, Z.; and Liu, T. 2019. Multilingual Neural Machine Translation with Knowledge Distillation. *CoRR*, abs/1902.10461.

Tian, F.; Dai, H.; Bian, J.; Gao, B.; Zhang, R.; Chen, E.; and Liu, T.-Y. 2014. A probabilistic model for learning multi-prototype word embeddings. In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, 151–160.

Tissier, J.; Gravier, C.; and Habrard, A. 2019. Near-lossless binarization of word embeddings. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 7104–7111.

Van Der Walt, S.; Colbert, S. C.; and Varoquaux, G. 2011. The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2): 22.

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.