Jointly Optimizing Query Encoder and Product Quantization to Improve Retrieval Performance

Jingtao Zhan\textsuperscript{1}, Jiaxin Mao\textsuperscript{2}, Yiqun Liu\textsuperscript{1}\textsuperscript{*}, Jiafeng Guo\textsuperscript{3}, Min Zhang\textsuperscript{1}, Shaoping Ma\textsuperscript{1}

\textsuperscript{1} Department of Computer Science and Technology, Institute for Artificial Intelligence, Beijing National Research Center for Information Science and Technology, Tsinghua University, Beijing 100084, China
\textsuperscript{2} Beijing Key Laboratory of Big Data Management and Analysis Methods, Gaoling School of Artificial Intelligence, Renmin University of China, Beijing 100872, China
\textsuperscript{3} CAS Key Lab of Network Data Science and Technology, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China

jingtaozhan@gmail.com, maojiaxin@gmail.com, yiqunliu@tsinghua.edu.cn, guojiafeng@ict.ac.cn

ABSTRACT

Recently, Information Retrieval community has witnessed fast-paced advances in Dense Retrieval (DR), which performs first-stage retrieval with embedding-based search. Despite the impressive ranking performance, previous studies usually adopt brute-force search to acquire candidates, which is prohibitive in practical Web search scenarios due to its tremendous memory usage and time cost. To overcome these problems, vector compression methods have been adopted in many practical embedding-based retrieval applications. One of the most popular methods is Product Quantization (PQ). However, although existing vector compression methods including PQ can help improve the efficiency of DR, they incur severely decayed retrieval performance due to the separation between encoding and compression. To tackle this problem, we present JPQ, which stands for Joint optimization of query encoding and Product Quantization. It trains the query encoder and PQ index jointly in an end-to-end manner based on three optimization strategies, namely ranking-oriented loss, PQ centroid optimization, and end-to-end negative sampling. We evaluate JPQ on two publicly available retrieval benchmarks. Experimental results show that JPQ significantly outperforms popular vector compression methods. Compared with previous DR models that use brute-force search, JPQ almost matches the best retrieval performance with 30x compression on index size. The compressed index further brings 10x speedup on CPU and 2x speedup on GPU in query latency.

CCS CONCEPTS

- Information systems → Retrieval models and ranking
- Search
- Index compression

KEYWORDS

dense retrieval; index compression; neural ranking

1 INTRODUCTION

Traditional web search relies on Bag-of-Words (BoW) retrieval models like BM25 [40] for efficient first-stage retrieval. Recently, with the rapid progress in representation learning [5] and deep pre-trained language models [15, 31, 41], Dense Retrieval (DR) [9] has become a popular paradigm to improve retrieval performance. In this paradigm, dual-encoders are employed to embed user queries and documents in a latent vector space. To generate result candidates based on embedded queries and documents in the space, most existing DR models [16, 22, 43, 48] rely on brute-force search. They significantly outperform BoW models in terms of effectiveness and benefit downstream tasks like OpenQA [21, 27]. However, brute-force search is not suitable for practical Web search scenarios due to its high computational cost. For online services, most solutions rely on Approximate Nearest Neighbor Search (ANNS) to perform efficient vector search [10, 49].

As an important implementation of ANNS, vector compression methods [18, 25] have been adopted in many practical embedding-based retrieval applications for the following two reasons. Firstly, the size of the uncompressed embedding index is very large. For example, it is typically an order of magnitude larger than the traditional BoW index in existing DR studies [43, 47, 48]. Secondly, the embedding index is usually required to be loaded into system memory or even GPU memory [26], whose size is highly limited. Therefore, compressing the embedding index is necessary when DR is applied in practical Web search scenario. Popular compression methods include Product Quantization (PQ) [18, 25] and Locality Sensitive Hashing (LSH) [24].

Although existing vector compression methods can help improve efficiency for DR models, they suffer from a few drawbacks in practical scenarios. Many popular compression methods [18, 20, 25]
cannot benefit from supervised information because they use the task-independent reconstruction error as the loss function in training. Besides, the encoders and the compressed index are separately trained and thus may not be optimally compatible. Recently, some studies [10, 11, 45, 49] jointly train the encoders and the compressed index. However, these studies are not designed for Web search scenarios and do not take into account the characteristics of DR processes, such as the importance of negative sampling [43, 47]. Moreover, they still attempt to minimize the reconstruction error rather than to optimize retrieval performance directly [10, 11, 49]. In our experiments described in Section 5.1, we find the existing methods still lead to a significant compromise in ranking performance.

In Figure 1, we show the trade-offs between index size and ranking performance for a variety of existing first-stage retrieval models including Brute-force DR Models (before compression), Compressed ANNS DR Models (after compression), BoW models, and Late Interaction models. We report retrieval effectiveness (MRR@10) as well as the index size (log-scale) on MS MARCO Passage Ranking [4], a widely-adopted benchmark dataset. As the figure shows, although dual-encoders improve the search performance compared with BoW models, they lead to a significant increase in the index size. However, when the index is compressed by popular methods like LSH [24] and OPQ [18], the retrieval effectiveness is hurt by a large margin.

To maintain retrieval effectiveness while reducing index sizes using compression, we propose JPQ, which stands for Joint optimization of query encoding and Product Quantization. It aims to optimize the ranking performance in an end-to-end manner instead of following the existing encoding-compression two-step procedure. To achieve this goal, jointly optimizes the query encoder and PQ index with three strategies: 1) We use ranking-oriented loss for JPQ, abandoning the partial ranking loss output by dual-encoders alone [43, 47, 48] and the task-independent reconstruction loss widely-used for training PQ [10, 11, 49]. It is computed in an end-to-end manner, i.e., the actual loss produced by dual-encoders with PQ index, and thus can evaluate the ranking performance accurately. 2) Training PQ index with ranking-oriented loss is non-trivial due to problems like differentiability and overfitting. To tackle these problems, JPQ proposes to use PQ centroid optimization, which only trains a small but crucial number of PQ parameters, i.e., PQ Centroid Embeddings. Other PQ parameters are well initialized and fixed during training. 3) Besides the ranking-oriented loss, JPQ uses end-to-end negative sampling to further improve end-to-end ranking performance. Given several training queries, it performs end-to-end retrieval using current encoder and PQ parameters in training. The top-ranked irrelevant documents are utilized as negatives. Through penalizing the scores of these documents, JPQ learns to improve end-to-end ranking performance.

To verify the effectiveness and efficiency of JPQ, we conduct extensive experiments on two publicly available benchmarks [4, 12] and compare JPQ against a wide range of existing ANNS methods and first-stage retrieval models. Experimental results show that: 1) JPQ substantially reduces the index size and improves the retrieval efficiency without significantly hurting the retrieval effectiveness. Therefore, compared with existing index compression methods that trade effectiveness for efficiency, JPQ outperforms them in terms of ranking effectiveness by a large margin; compared with non-exhaustive ANNS methods that do not compress the index, JPQ achieves a better ranking performance with a 30x smaller index file. 2) JPQ outperforms existing first-stage retrieval approaches in terms of effectiveness and efficiency. JPQ is much more effective than existing DR models that use brute-force search. Specifically, it gains similar ranking performance with state-of-the-art DR models while providing 30x index compression ratio, 10x CPU speedup, and 2x GPU speedup. Compared with late-interaction models, JPQ gains similar recall with a several orders of magnitude smaller index and 5x GPU speedup. We also conduct an ablation study for JPQ on the passage ranking dataset [4]. According to the experimental results, all three strategies, namely ranking-oriented loss, PQ centroid optimization, and end-to-end negative sampling, contribute to its effectiveness.

2 BACKGROUND AND RELATED WORKS

This section introduces the background and related works. Several commonly used notations are: \( C \) denotes the set of all documents; \( N \) is the number of all documents; \( n \) is the number of documents returned by the retrieval algorithms; and \( D \) is the embedding dimension.

2.1 Dual-Encoders

Dual-encoders, i.e., the query encoder and the document encoder, represent queries and documents with embeddings. Let \( f \) be the dual-encoders, which takes the input of a query \( q \) or a document \( d \) and outputs an embedding, \( \vec{q} \) or \( \vec{d} \):

\[
\vec{q} = f(q) \in \mathbb{R}^D \quad \vec{d} = f(d) \in \mathbb{R}^D
\]

(1)

Let \( \langle \cdot, \cdot \rangle \) be the embedding similarity function. The predicted relevance score \( s(q, d) \) is:

\[
s(q, d) = \langle \vec{q}, \vec{d} \rangle
\]

(2)

Dual-encoders are mainly trained by negative sampling methods [16, 43, 48]. Some studies use all irrelevant documents [16, 23] as negatives. Some studies use hard negatives retrieved by BM25 [17, 48] or a warm-up DR model [43]. Recently, Zhan et al. [47] propose
dynamic hard negative sampling and show that hard negatives help improve top ranking performance.

2.2 Brute-force Search

Brute-force search is utilized by many previous DR studies [22, 43, 47] to retrieve candidates. We briefly formulate the search process and analyze its efficiency.

2.2.1 Search Procedure. The search procedure contains two stages, score computation and sorting. Given a query embedding \( \tilde{q} \), it computes the relevance score \( s(q, d) \) using \( \langle \tilde{q}, \tilde{d} \rangle \) for each document \( d \in C \). Then, it sorts all documents based on \( s(q, d) \) and returns the top-\( n \) documents. We formally express the procedure as follows.

\[
\text{results} = \text{sort}(d \in C \text{ based on } s(q, d)): n
\]  

(3)

2.2.2 Time Complexity. For score computation, the complexity is \( O(ND) \). For sorting, the complexity is \( O(N \log n) \). Therefore, the overall time complexity is \( O(ND + N \log n) \).

2.2.3 Index Size. The index is actually all document embeddings. Since each float consumes 4 bytes, the index size is \( 4ND \) bytes.

2.3 ANNS

ANNS achieves highly efficient vector search by allowing a small number of errors. Generally, there are two kinds of ANNS algorithms. One is non-exhaustive ANNS methods, and the other is vector compression methods. Both are important and usually combined in practice.

Non-exhaustive ANNS methods do not compress the index. They reduce the number of candidates for each query to speed up the retrieval process. Formally, let \( \Omega \) be one method, and \( \tilde{\Omega}(q, C) \) be the returned candidates for query \( q \). The search process is:

\[
\text{results} = \text{sort}(d \in \tilde{\Omega}(q, C) \text{ based on } s(q, d)): n
\]  

(4)

Popular algorithms include tree-based methods [6, 37], inverted file methods [3, 25], and graph-based methods [34].

Vector compression methods aim to compress the index but are also able to accelerate retrieval. They learn a new score function \( s^\dagger \), which is fast to compute and requires a small document index. We formulate the search process as follows:

\[
\text{results} = \text{sort}(d \in C \text{ based on } s^\dagger(q, d)): n
\]  

(5)

Popular algorithms include hashing [24] and quantization [18, 25].

2.4 Product Quantization

Product Quantization (PQ) [25] is a popular ANNS algorithm and belongs to the second kind of ANNS introduced in the previous section.

2.4.1 Approximate Score Function.

PQ replaces \( s \) with a new approximate score function \( s^\dagger \). It quantizes the document embedding \( \tilde{d} \in \mathbb{R}^D \) to \( \tilde{d}^\dagger \in \mathbb{R}^D \). Then it uses \( \tilde{d}^\dagger \) to compute similarity:

\[
s^\dagger(q, d) = \langle \tilde{q}, \tilde{d}^\dagger \rangle
\]  

(6)

Next, we introduce how it quantizes document embeddings.

2.4.2 Quantizing Document Embeddings.

PQ defines \( M \) sets of embeddings, each of which includes \( K \) embeddings of dimension \( D/M \). We call them PQ Centroid Embeddings. To quantize a document embedding, PQ picks one from each of these \( M \) sets and then concatenates the \( M \) picked embeddings as the quantized document embeddings.

Formally, let \( \tilde{c}_{i,j} \) be the \( j_{th} \) centroid embedding from the \( i_{th} \) set:

\[
\tilde{c}_{i,j} \in \mathbb{R}^D \quad (1 \leq i \leq M, 1 \leq j \leq K)
\]  

(7)

Given a document embedding \( \tilde{d} \), PQ picks the \( \varphi_i(d)_{th} \) centroid embedding from the \( i_{th} \) centroid set (\( 1 \leq i \leq M \)). Then it concatenates the \( M \) picked embeddings as \( \tilde{d}^\dagger \):

\[
\tilde{d} \rightarrow \tilde{d}^\dagger = \tilde{c}_{1,\varphi_1(d)}, \tilde{c}_{2,\varphi_2(d)}, \ldots, \tilde{c}_{M,\varphi_M(d)} \in \mathbb{R}^D
\]  

(8)

where comma denotes the concatenation operation. We call \( \varphi_i(d) \) Index Assignments.

2.4.3 Optimization Objective.

PQ trains \( (\tilde{c}_{i,j}) \) and \( \varphi \) to minimize the MSE loss between the original document embedding \( \tilde{d} \) and the quantized one \( \tilde{d}^\dagger \).

\[
\{\tilde{c}_{i,j}\}, \varphi = \arg\min ||\tilde{d} - \tilde{d}^\dagger||^2
\]  

(9)

The loss is also called reconstruction error. In this way, \( s^\dagger(q, d) \) approximates the true \( s(q, d) \).

2.4.4 Search Procedure.

PQ uses an equivalent but very efficient way to compute Eq. (6). It firstly splits the query embedding equally to \( M \) sub-vectors:

\[
\tilde{q} = \tilde{q}_1, \tilde{q}_2, \ldots, \tilde{q}_M
\]  

(10)

where \( \tilde{q}_i \) denotes the \( i_{th} \) sub-vector and is of dimension \( D/M \). Then it computes the similarity score between the query sub-vectors and PQ Centroid Embeddings:

\[
\tau_{i,j} = \langle \tilde{q}_i, \tilde{c}_{i,j} \rangle \quad (1 \leq i \leq M, 1 \leq j \leq K)
\]  

(11)

After constructing a lookup table with \( \tau_{i,j} \), PQ efficiently computes \( s^\dagger(q, d) \). For example, if \( \langle \cdot \rangle \) is inner product, PQ only needs to sum corresponding \( \tau_{i,j} \):

\[
s^\dagger(q, d) = \sum_{i=1}^{M} \tau_{i,\varphi_i(d)}
\]  

(12)

2.4.5 Time Complexity.

The major time cost is computing Eq. (12) for all documents and sorting. The overall time complexity is \( O(NM + N \log n) \). Compared with brute-force search, the speedup ratio is \( (D + \log n)/(M + \log n) \).

2.4.6 Index Size.

PQ does not explicitly store \( \tilde{d}^\dagger \). Instead, it only stores the PQ Centroid Embeddings \( (\tilde{c}_{i,j}) \) and Index Assignments \( \{\varphi_i(d)\} \). \( K \) usually equals to or is less than 256 so that \( \varphi_i(d) \) can be stored with one byte. The overall index size is \( 4KD + NM = NM \) bytes. Compared with brute-force search, the compression ratio is \( 4D/M \).
We propose JPQ, Joint optimization of query encoding and Product Quantization. In the following, we will outline the overall architecture, describe the three strategies, summarize the optimization objective, and analyze its efficiency.

3.1 Overall Architecture

Figure 2 contrasts JPQ with existing training methods. Solid arrows indicate the gradients are backpropagated, whereas the dotted arrows indicate otherwise.

On the top, Figure 2a illustrates previous methods [18, 20, 25], which follow an encoding-compression two-step procedure. They firstly train the dual-encoders with ranking loss, which does not consider PQ compression. Then they use the trained document encoder to encode all documents. Given all document embeddings, they finally train the PQ index to minimize the MSE loss (aka reconstruction error), which is task-independent and cannot benefit PQ index with supervised information.

On the bottom, Figure 2b visualizes the training process of JPQ, which follows an end-to-end paradigm with three strategies:

- JPQ proposes to use a new ranking-oriented loss, which is the actual loss produced by dual-encoders and PQ index. To compute it, JPQ firstly reconstructs the quantized document embeddings, then computes the actual relevance scores used by PQ for ranking, and finally passes the scores to a pair-wise loss function.
- Training PQ index with ranking-oriented loss is non-trivial due to problems like differentiability and overfitting. Hence, JPQ proposes PQ centroid optimization to address these problems. It initializes Index Assignments and only updates PQ Centroid Embeddings using gradient descent.
- JPQ utilizes end-to-end negative sampling to further improve ranking performance. The training query embeddings are fed into the ‘Search Negatives’ module, which uses the current PQ parameters to retrieve top-ranked irrelevant documents as negatives.

Next, we will describe the three strategies in detail.

3.2 Ranking-oriented Loss

JPQ uses ranking-oriented loss to accurately evaluate the end-to-end ranking performance. It is computed based on the actual scores utilized for ranking. We elaborate on it in the following.

Previous DR studies [17, 32, 48] usually adopt the following pair-wise ranking loss:

$$
\ell(s(q, d^+), s(q, d^-))
$$

where $s(q, d)$ is the original relevance score output by the dual-encoders, and $\ell : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ is a pair-wise loss function. Nevertheless, it is not the actual scores utilized by many ANNS algorithms during ranking. As for PQ, it quantizes documents and uses a new score $s^\dagger(q, d)$ for ranking. Since ranking with $s(q, d)$ or $s^\dagger(q, d)$ is likely to yield different ranking lists, training with the above loss cannot effectively improve the actual PQ ranking performance.

To solve this problem, we use $s^\dagger(q, d)$ to compute the ranking-oriented loss. Since $s^\dagger(q, d)$ equals to the similarity between query embedding and quantized document embeddings, JPQ firstly reconstructs the quantized document embeddings $\tilde{d}^\dagger$ from the PQ index:

$$
\tilde{d}^\dagger = \tilde{c}_1, \psi_1(d), \tilde{c}_2, \psi_2(d), \ldots, \tilde{c}_M, \psi_M(d)
$$

Secondly, JPQ computes $s^\dagger(q, d)$ using $\tilde{q}$ and $\tilde{d}^\dagger$:

$$
s^\dagger(q, d) = \langle \tilde{q}, \tilde{d}^\dagger \rangle
$$

Finally, JPQ passes $s^\dagger(q, d)$ to the pair-wise loss function:

$$
\ell(s^\dagger(q, d^+), s^\dagger(q, d^-))
$$
Because Eq. (16) utilizes the actual ranking scores, it evaluates the ranking performance accurately and helps improve it using gradient descent.

3.3 PQ Centroid Optimization

Training PQ with ranking-oriented loss is non-trivial due to the following two problems related to Index Assignments. Firstly, Index Assignments are not differentiable with respect to the ranking-oriented loss. As described in Section 2.4.4, the Index Assignments are utilized to select corresponding PQ Centroid Embeddings, which is hard to optimize using gradient decent. Secondly, even if some methods other than gradient decent can be used to update Index Assignments, e.g., exhaustive enumeration, it may cause overfitting.

Training PQ with ranking-oriented loss is non-trivial due to the following two problems related to Index Assignments. Firstly, Index Assignments are not differentiable with respect to the ranking-oriented loss. As described in Section 2.4.4, the Index Assignments are utilized to select corresponding PQ Centroid Embeddings, which is hard to optimize using gradient decent. Secondly, even if some methods other than gradient decent can be used to update Index Assignments, e.g., exhaustive enumeration, it may cause overfitting problems because the number of Index Assignments is huge, i.e., proportional to the size of the corpus.

To resolve these problems, JPQ proposes PQ centroid optimization, which initializes Index Assignments using Figure 2a and only trains a very small but crucial number of PQ parameters, PQ Centroid Embeddings. These centroid embeddings are differentiable, and it is unlikely to overfit the training data due to its small number.

Now we illustrate how PQ Centroid Embeddings are updated using gradient decent. We compute its gradients with respect to the ranking-oriented loss in three steps. Firstly, for many commonly used pair-wise loss, such as hinge loss, RankNet [7], and LambdaRank [8], we can define \( \alpha \geq 0 \) as follows:

\[
\alpha = - \frac{\partial \ell(s^*(q, d^*), s^\dagger(q, d^-))}{\partial s^\dagger(q, d^-)} = \frac{\partial \ell(s^*(q, d^*), s^\dagger(q, d^-))}{\partial s^\dagger(q, d^-)} \geq 0 \quad (17)
\]

Secondly, the gradient of \( \tilde{c}_{i,j} \) with respect to the score \( s^\dagger(q, d) \) is:

\[
\frac{\partial s^\dagger(q, d)}{\partial \tilde{c}_{i,j}} = \begin{cases} 
\tilde{q}_i, & \text{if } j = \phi_1(d). \\
0, & \text{otherwise}.
\end{cases} \quad (18)
\]

where \( \tilde{q}_i \) is the \( i \)th query embedding sub-vector described in Eq. (10). Finally, with the above two equations, we can use chain rule to derive the gradients of PQ Centroid Embeddings:

\[
\frac{\partial \ell(s^\dagger(q, d^*), s^\ddagger(q, d^-))}{\partial \tilde{c}_{i,j}} = \begin{cases} 
-a\tilde{q}_i, & \text{if } j = \phi_1(d^*), j \neq \phi_1(d^-). \\
a\tilde{q}_i, & \text{if } j \neq \phi_1(d^*), j = \phi_1(d^-). \\
0, & \text{if } j = \phi_1(d^*), j = \phi_1(d^-). \\
0, & \text{if } j \neq \phi_1(d^*), j \neq \phi_1(d^-).
\end{cases} \quad (19)
\]

According to the above gradients, PQ centroid optimization has two advantages, benefiting PQ index with supervised signals and helping it evolve with the query encoder. Firstly, the PQ index benefits from supervised signals through relevance labels and \( \alpha \). Relevance labels control the updating conditions. \( \alpha \) decides the updating weight, which is small if the ranking is correct and large otherwise. Secondly, PQ parameters directly evolve with the query encoder through \( \tilde{q}_i \), which is part of the query encoder’s output.

3.4 End-to-End Negative Sampling

Besides ranking-oriented loss, JPQ proposes end-to-end negative sampling to further improve end-to-end ranking performance. Note that negative sampling has been shown to be important for training dual-encoders [43, 47].

Our intuition is to penalize those top-ranked irrelevant documents and disregard others. The top-ranked negatives greatly affect the ranking performance, whereas low-ranked documents are mostly ignored by the truncated evaluation metric. By penalizing the top-ranked negatives, we can improve the top-ranking performance, which is the target of many popular IR systems, such as Web search engines.

The top-ranked negatives are acquired by real-time end-to-end retrieval at each training step. As shown in Figure 2b, the training query embeddings are passed to the ‘Search Negatives’ module, which uses the current PQ parameters to retrieve top-\( n \) irrelevant documents as negatives. Let \( \mathcal{D}_q^\dagger \) be the retrieved negatives and \( \mathcal{D}_q^- \) be the labeled relevant documents. \( \mathcal{D}_q^\dagger \) is formulated as:

\[
\mathcal{D}_q^\dagger = \text{sort}(d \in \mathcal{C} \setminus \mathcal{D}_q^- \text{ based on } s^\dagger(q, d)) : \hat{n}
\]

Using \( \mathcal{D}_q^\dagger \) as negatives can also be regarded as minimizing the top-\( n \) pairwise errors, which is in line with the truncated evaluation metric.

Although using top-ranked irrelevant documents as negatives has recently been explored by Zhan et al. [47], they utilize brute-force search to retrieve negatives. We use ‘BF negs’ and ‘JPQ negs’ to denote their selected negatives and ours, respectively. Figure 3 illustrates an example from TREC DL 2019 dataset [12] using t-SNE [33]. According to our intuition, the nearest neighbors to the query should be utilized as negatives. As the figure shows, our method selects the actual nearest neighbors owing to the end-to-end retrieval, whereas ‘BF negs’ do not due to the difference between PQ ranking and brute-force search.

3.5 Optimization Objective

Now we summarize the optimization objective of JPQ. JPQ uses end-to-end negative sampling and computes the ranking-oriented loss to jointly optimize the query encoder and PQ Centroid Embeddings.

Therefore, the optimization objective is formulated as follows:

\[
f^* = \arg \min_{f, \{\tilde{c}_{i,j}\}} \sum_{q} \sum_{d^* \in \mathcal{D}_q^\dagger} \sum_{d^- \in \mathcal{D}_q^-} \ell(s^\dagger(q, d^*), s^\dagger(q, d^-))
\]

3.6 Efficiency

The search time complexity and index size are the same as PQ. As we introduce PQ in Section 2.4, the search time complexity is \( O(NM + N \log N) \), and the index size is \( 4KD + NM \approx NM \) bytes. Compared with brute-force search, the speedup ratio is \( (D + \log n)/(M + \log n) \), and the compression ratio is \( 4D/M \).
Table 1: Comparison with different ANNS methods on TREC 2019 Deep Learning Track. */** denotes that JPQ performs significantly better than baselines at $p < 0.05/0.01$ level using the two-tailed pairwise t-test.

| Model                  | Index | MARCO Passage | DL Passage | MARCO Doc | DL Doc |
|------------------------|-------|---------------|------------|-----------|--------|
|                        | GB    | MRR@10 | R@100 | NDCG@10 | R@100 | GB    | MRR@100 | R@100 | NDCG@10 |
| Non-exhaust. ANNS      |       |         |       |          |       |       |         |       |          |
| Annoy [6]              | 30.96 | 0.150** | 0.274** | 0.431** | 0.157** | 11.36 | 0.304** | 0.638** | 0.553** |
| FALCONN [1]            | >25.34 | 0.297** | 0.825** | 0.567** | 0.411** | >9.20 | 0.339** | 0.869** | 0.541** |
| FLANN [36]             | 32.85 | 0.319** | 0.839** | 0.607** | 0.430** | 10.60 | 0.366** | 0.895** | 0.593   |
| IMI [3]                | 25.39 | 0.331** | 0.828** | 0.610*  | 0.431*  | 9.24  | 0.376** | 0.869** | 0.590   |
| HNSW [34]              | 25.76 | 0.334*  | 0.848** | 0.624   | 0.454   | 9.35  | 0.378** | 0.879** | 0.589   |
| Compressed ANNS        |       |         |       |          |       |       |         |       |          |
| PQ [25]                | 0.79  | 0.123** | 0.527** | 0.323** | 0.213** | 0.29  | 0.152** | 0.544** | 0.273** |
| LSH [26]               | 0.79  | 0.302** | 0.824** | 0.578** | 0.417** | 0.29  | 0.351** | 0.882** | 0.576** |
| ITQ+LSH [19]           | 0.80  | 0.296** | 0.825** | 0.582** | 0.415** | 0.29  | 0.347** | 0.874** | 0.570** |
| ScANN [20]             | 0.79  | 0.288** | 0.818** | 0.555** | 0.386** | 0.29  | 0.335** | 0.866** | 0.534** |
| HNSW+OPQ [26]          | 0.95  | 0.309** | 0.818** | 0.596** | 0.424** | 0.35  | 0.357** | 0.876** | 0.543** |
| OPQ [18]               | 0.83  | 0.305** | 0.839** | 0.594** | 0.435** | 0.30  | 0.361** | 0.892** | 0.588** |
| OPQ+ScANN              | 0.83  | 0.313** | 0.843** | 0.614** | 0.442*  | 0.30  | 0.361** | 0.897** | 0.583** |
| DPQ [11, 49]           | 0.83  | 0.311** | 0.848** | 0.601** | 0.453   | 0.30  | 0.367** | 0.899** | 0.585** |
| Ours                   | JPQ   | 0.83   | 0.341 | 0.868 | 0.677 | 0.466 | 0.30 | 0.401 | 0.914 | 0.623 |

4 EXPERIMENTAL SETUP

Here we present our experimental settings, including datasets, baselines, and implementation details.

4.1 Datasets and Metrics

We conduct experiments with two large-scale ad-hoc retrieval benchmarks from the TREC 2019 Deep Learning Track [4, 12]. Passage Retrieval has a corpus of 8.8M passages, 0.5M training queries, 7k development queries (henceforth, MARCO Passage), and 43 test queries (DL Passage). Document Retrieval has a corpus of 3.2M documents, 0.4M training queries, 5k development queries (MARCO Doc), and 43 test queries (DL Doc). For both tasks, we report the official metrics and R@100 based on the full-corpus retrieval results.

4.2 Baselines

We exploit six types of models as baselines, including different retrieval models and ANNS methods.

4.2.1 Traditional BoW Models.

BM25 [40] is a popular probabilistic BoW retrieval model that ranks documents based on the query terms appearing in each document. We use Anserini implementation [44].

4.2.2 Augmented BoW Models.

Several methods use deep language models to improve BoW models. We use them as our baselines, including doc2query [39], docT5query [38], DeepCT [13], and HDCT [14].

4.2.3 Late-Interaction Models.

ColBERT [28] stores the contextualized token embeddings and retrieves candidates with a late-interaction operation. The index size is very large because it stores token-level representations.

4.2.4 Brute-force DR Models.

Previous DR studies retrieve candidates for queries with brute-force search. We call them brute-force DR models. They share similar architectures [15, 31] but differ in training process. Baselines trained by negative sampling method include Rand Neg [23], BM25 Neg [48], ANCE [43], STAR [47], and ADORE+STAR [47]. The baseline trained by knowledge distillation is TCT-ColBERT [30].

4.2.5 Non-exhaustive ANNS DR Models.

Non-exhaustive ANNS methods accelerate search but do not compress the index. We use the following methods as baselines. Annoy [6] is based on random projection tree forest and it is hard to infer the exact index size at run time.

4.2.6 Compressed ANNS DR Models.

Compressed ANNS DR Models compress document embeddings. Unsupervised compression baselines include LSH [24], ITQ+LSH [19], PQ [25], OPQ [18], ScANN [20], OPQ+ScANN, and HNSW+OPQ [26]. Note, OPQ+ScANN uses OPQ to learn the transformation and ScANN to compress the transformed embeddings, and HNSW+OPQ [26] uses HNSW to construct proximity graphs and OPQ to compress embeddings. The supervised compression baseline is DPQ [11, 49]. It is originally designed for word embedding compression [11] and recommendation systems [49]. We implement it for document ranking.
4.3 Implementation Details

We build our models based on huggingface transformers [42] and Faiss ANNS library [26]. All dual-encoders use the 'bert-base' [15, 31] architecture. The embedding dimension $D$ is 768, and similarity function $\langle \cdot , \cdot \rangle$ is inner-product. All ANNS baselines use STAR [47] as dual-encoders. Here is how we implement JPQ. $K$ is set to 256, and $M$ is set to 16, 24, 32, 48, 64, and 96 for experiments in different parameter settings. We use OPQ [18] to learn a linear transformation of embeddings and then use PQ [25] for compression. As for JPQ’s training settings, we use the same parameters for both retrieval tasks. We use AdamW optimizer, batch size of 32, and LambdaRank [8] as pair-wise loss function. Top-200 irrelevant documents are used as hard negatives. For the query encoder, learning rate is set to $5 \times 10^{-6}$. For PQ parameters, learning rate equals to $5 \times 10^{-6}$ for $M = 16/24$, $2 \times 10^{-5}$ for $M = 32$, and $1 \times 10^{-4}$ for $M = 48/64/96$.

5 EXPERIMENTS

Now we empirically evaluate JPQ to address the following three research questions:

- **RQ1**: Can JPQ substantially compress the index without significantly hurting the ranking performance?
- **RQ2**: How does JPQ perform compared with other retrieval models?
- **RQ3**: How do different strategies contribute to the effectiveness of JPQ?

5.1 Comparison with ANNS Methods

This section compares JPQ with ANNS methods to answer RQ1. We utilize two types of ANNS baselines, i.e., vector compression methods and non-exhaustive ANNS methods. Results at a fixed trade-off setting are presented in Table 1. Results at different trade-off settings are presented in Figures 4 and 5, which show the effectiveness-memory and effectiveness-speed curves, respectively. Next, we proceed to study these results in detail.

5.1.1 Comparison with vector compression methods.

Compressed ANNS baselines as well as JPQ use roughly the same compression ratio (30x) in Table 1. According to the results, we see that JPQ leads the ranking performance by a large margin on all metrics. Among these baselines, DPQ [11, 49] is also a joint learning method. Although it benefits from supervised signals, it only marginally outperforms other unsupervised methods. The problem is that it still uses reconstruction error as loss and does not design an end-to-end negative sampling method. Conversely, JPQ adopts three strategies to improve the end-to-end ranking performance specifically, enabling it to substantially outperform both unsupervised and supervised baselines.

Now we investigate ranking performance at different compression and speedup ratios. According to Figures 4 and 5, JPQ strongly outperforms vector compression baselines regardless of different trade-off settings. Moreover, results also show that JPQ is effective even with a very high compression or speedup ratio. For example, when the compression ratio is about 140x, DPQ, the most effective baseline, incurs 28% performance loss on the document retrieval task, whereas JPQ only incurs 15% performance loss.

5.1.2 Comparison with non-exhaustive ANNS methods.

Non-exhaustive ANNS baselines are tuned to be as fast as JPQ in Table 1, specifically, 10x faster than brute-force search on CPU. According to the results, we see that JPQ is significantly more effective in both retrieval tasks, which is a bit surprising since those baselines...
Table 3: Comparison with BoW models on TREC 2019 Deep Learning Track. */** denotes that JPQ performs significantly better than baselines at $p < 0.05/0.01$ level using the two-tailed pairwise t-test.

| Model | Index | MARCO Passage | DL Passage |
|-------|-------|---------------|------------|
|       | GB MRR@10 R@100 NDCG@10 | GB MRR@100 R@100 NDCG@10 |
| Traditional BoW |       |               |            |
| BM25 [44] | 0.59 | 0.187** | 0.670** | 0.497** | 0.460 | 2.17 | 0.278** | 0.807** | 0.523** |
| Augmented BoW |       |               |            |
| doc2query [39] | 0.65 | 0.215** | 0.713** | 0.533** | 0.471 | n.a. | n.a. | n.a. | n.a. |
| DeepCT [13] | 0.48 | 0.242** | 0.754** | 0.569** | 0.455 | n.a. | n.a. | n.a. | n.a. |
| HDCT [14] | n.a. | n.a. | n.a. | n.a. | 1.71 | 0.319** | 0.843** | n.a. |
| docT5query [38] | 0.96 | 0.272** | 0.819** | 0.642 | 0.514 | 2.31 | 0.327** | 0.861** | 0.597 |
| Ours |       |               |            |
| JPQ | 0.83 | 0.341 | 0.868 | 0.677 | 0.466 | 0.30 | 0.401 | 0.914 | 0.623 |

Figure 6: Effectiveness (MRR@100) versus Index Size (log-scale) for different retrieval methods on MS MARCO Document Ranking [4]. The index size of JPQ is only 1/5833 of the size of ColBERT.

Table 4: Comparison between late-interaction models and JPQ+BERT two-stage methods on MARCO Passage dataset.

| Model | Latency (ms) | GPUs | MRR@10 |
|-------|--------------|------|--------|
| Late-Interaction | CoLBERT [28] | 458 | 1 | 0.360 |
| JPQ+BERT$_{base}$ | Re-rank Top 10 | 116 | 1 | 0.367 |
| | Re-rank Top 30 | 187 | 1 | 0.378 |
| JPQ+BERT$_{large}$ | Re-rank Top 10 | 184 | 1 | 0.372 |
| | Re-rank Top 30 | 406 | 1 | 0.387 |

5.2 Comparison with Retrieval Models

This section uses existing retrieval models as baselines to answer RQ2. We firstly compare JPQ with baselines in general, and then separately compare JPQ with different types of baselines in detail.

5.2.1 Overall Comparison.

We summarize the ranking performance, index size, and latency of representative retrieval models in Figure 1, Figure 6, and Table 2. According to the figures, although existing neural retrieval models, i.e., brute-force DR models and late-interaction models, are more effective than BoW models, they significantly increase the index size by several orders of magnitude. When the indexes of brute-force DR models are compressed by LSH [24] or OPQ [18], the retrieval effectiveness is severely hurt. Therefore, the results seem to imply that large index sizes are necessary for high-quality ranking.

In contrast with trading index size for ranking performance, JPQ achieves high ranking effectiveness with a tiny index. According to Figures 1 and 6, it outperforms BoW model by a large margin with similar or even much smaller index sizes. It gains similar ranking performance with state-of-the-art brute-force DR models while substantially compressing the index. As for the retrieval latency, Table 2 shows that JPQ is as fast as BoW models with GPU acceleration. Compared with brute-force DR models, JPQ provides 10x speedup on CPU and 2x speedup on GPU. Compared with CoLBERT, JPQ provides 5x speedup. These results highlight the effectiveness of JPQ.

5.2.2 Separate Comparison.

In the following, we use more comprehensive ranking results to separately compare JPQ with different types of retrieval models. Table 3 compares JPQ with traditional BoW models as well as the augmented variants. Results show that JPQ substantially outperforms them in terms of both accuracy (MRR) and recall. For example, its index is only one-eighth the size of docT5query [38] on the document retrieval task, and it still improves MRR@100 and R@100 by 23% and 6%, respectively.

Table 5 shows ColBERT’s retrieval performance relative to JPQ. Besides, due to CoLBERT’s high latency, we add a reranking process to JPQ and show the results in Table 4. According to Table 5, JPQ gains similar recall even if its index size is 186x smaller on passage.
We conduct experiments at different compression settings and re-visit the contributions of 'Train PQ' to more difference between $s(q, d)$ and $s(q, d)$, and therefore computing loss with $s^T(q, d)$ is more important. When more bytes are used to encode one document, the contribution of 'Train PQ' is more prominent, especially in terms of MRR@10. We think using more bytes to encode one document helps diversify the quantized document embeddings, and thus training centroid embeddings can better fit the dataset.

6 CONCLUSIONS

This paper presents JPQ, a novel framework for dense retrieval that aims to achieve high-quality ranking performance with a compact index rather than following the trend of trading index size for ranking performance. It jointly optimizes encoding and compression processes in an end-to-end manner with three carefully designed strategies, i.e., ranking-oriented loss, PQ centroid optimization, and end-to-end negative sampling. We conduct experiments on two publicly available benchmarks, where JPQ achieves impressive performance. For example, even if JPQ compresses the index by 30x and accelerates retrieval by 10x on CPU and 2x on GPU, it outperforms some competitive brute-force DR models and gains similar ranking performance with the state-of-the-art one. The results clearly demonstrate the effectiveness of JPQ and highlight that a small embedding index can still be very effective in the first-stage retrieval.

ACKNOWLEDGMENTS

This work is supported by the National Key Research and Development Program of China (2018YFC0831700), Natural Science Foundation of China (Grant No. 61732008, 61532011, 61902209, U2001212), Beijing Academy of Artificial Intelligence (BAAI), Tsinghua University Guoqiang Research Institute, Beijing Outstanding Young Scientist Program (NO. BJJWZJH012019100020098) and Intelligent Social Governance Platform, Major Innovation & Planning Interdisciplinary Platform for the “Double-First Class” Initiative, Renmin University of China.
