- *In-batch sampling*: during training, the queries to compute the loss can be randomly aggregated into batches of size b, for faster training. For each query in a given batch, the positive passages for the other $b - 1$ queries are considered as negative passages for the query [Gillick et al. 2019]. This sampling approach suffers from the same near zero loss problem as random sampling [Xiong et al. 2021], but the sampling prodedure is faster.

- *Hard negative sampling*: negative documents can be generated exploiting a classical or trained retrieval system. Each query is given as input to the retrieval system, the top documents retrieved, and the documents not corresponding to the positive ones are treated as negatives. Note that in this case we are assuming a conditional noise distribution $p(d|q, d_1)$, since we assume to know the relevant document d_1 for the query. In doing so, high-ranking documents are prioritised w.r.t. low-ranking documents, that do not impact on the user experience and do not contribute to the loss. The retrieval system used to mine the negative documents can exploit BM25 relevance model, as in DPR [Karpukhin et al. 2020], the currently neural model under training, as in ANCE [Xiong et al. 2021], or another fine-tuned neural model, as in STAR [Zhan et al. 2021b].

4 Retrieval Architectures and Vector Search

This section illustrates how the neural IR systems discussed so far are actually deployed in end-to-end systems. Section 4.1 discusses how cross-encoders and bi-encoders are deployed in ranking architectures. Since dense retrieval systems pre-computed the document embeddings, many actual systems focus on storing and searching through document embeddings. Section 4.2 formally defines the search problems in vector spaces and the embedding index, while Sections 4.3, 4.4, and 4.5 illustrate different solutions for efficiently storing and searching vectors. Section 4.6 discusses some optimisations for embedding indexes specifically tailored for dense retrieval systems.

4.1 Retrieval architectures

Pre-trained language models successfully improve the effectiveness of IR systems in the ad-hoc ranking task, but they are computationally very expensive. Due to these computational costs, the interaction-focused systems are not applied directly on the document collection, i.e., to rank all documents matching a query. They are deployed in a pipelined architecture (Figure 7) by conducting first a preliminary ranking stage to retrieve a limited number of candidates, typically 1000 documents, before re-ranking

them with a more expensive neural re-ranking system, such as cross-encoders described in Section 2.

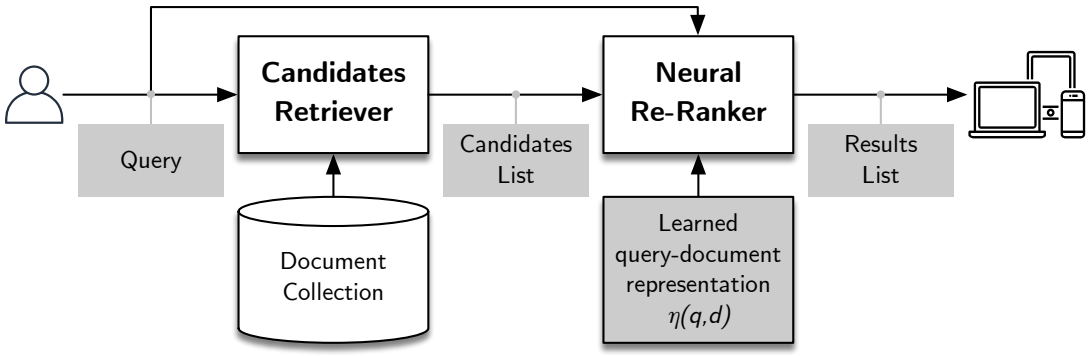

Figure 7: Re-ranking pipeline architecture for interaction-focused neural IR systems.

The most important benefit of bi-encoders discussed in Section 3 is the possibility to pre-compute and cache the representations of a large corpus of documents with the learned document representation encoder $\psi(d)$. At query processing time, the learned query representation encoder must compute only the query representation $\phi(q)$, then the documents are ranked according to the inner product of their representation with the query embedding, and the top k documents whose embeddings have the largest inner product w.r.t. the query embedding are returned to the user (Figure 8).

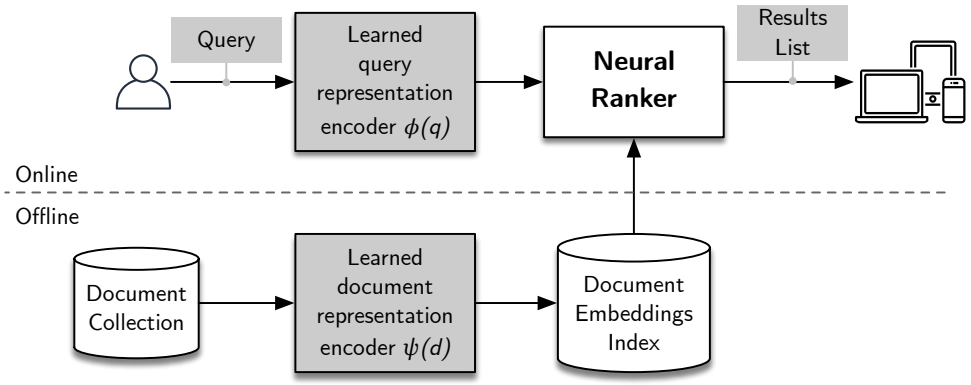

Figure 8: Dense retrieval architecture for representation-focused neural IR systems.

4.2 MIP and NN Search Problems

The pre-computed document embeddings are stored in a special data structure called *index*. In its simplest form, this index must store the document embeddings and provide a search algorithm that, given a query embedding, efficiently finds the document embedding with the largest dot product, or, more in general, with the maximum inner product.

Formally, let $\phi \in \mathbb{R}^\ell$ denote a query embedding, and let $\Psi = \{\psi_1, \ldots, \psi_n\}$ denote a set of n document embeddings, with $\psi_i \in \mathbb{R}^\ell$ for $i = 1, \ldots, n$. The goal of the *maximum inner product (MIP) search* is to find the document embedding $\psi^* \in X$ such that

$$\psi^* = \arg\max_{\psi \in \Psi} \langle \phi, \psi \rangle \tag{18}$$

A data structure designed to store Ψ is called *embedding index*. The naïve embedding index is the *flat index*, which stores the document embeddings in Ψ explicitly and performs an exhaustive search to identify ψ^*. Its complexity is $O(n\ell)$ both in space and time, so it is particularly inefficient for large n or ℓ values.
A common approach to improve the space and time efficiency of the flat index is to convert the maximum inner product search into a *nearest neighbour (NN) search*, whose goal is to find the document embedding $\psi^\dagger$ such that

$$\psi^\dagger = \arg\min_{\psi \in \Psi} \|\phi - \psi\| \tag{19}$$

Many efficient index data structures exist for NN search. To leverage them with embedding indexes, MIP search between embeddings must be adapted to use the Euclidean distance and NN search. This is possible by applying the following transformation from $\mathbb{R}^\ell$ to $\mathbb{R}^{\ell+1}$ [Bachrach et al. 2014, Neyshabur and Srebro 2015]:

$$\hat{\phi} = \begin{bmatrix} \phi / \|\phi\| \\ 0 \end{bmatrix}, \quad \hat{\psi} = \begin{bmatrix} \psi / M \\ \sqrt{1 - \|\psi\|^2 / M^2} \end{bmatrix}, \tag{20}$$

where $M = \max_{\psi \in \Psi} \|\psi\|$. By using this transformation, the MIP search solution ψ^* coincides with the NN search solution $\hat{\psi}^\dagger$. In fact, we have:

$$\min \|\hat{\phi} - \hat{\psi}\|^2 = \min \left(\|\hat{\phi}\|^2 + \|\hat{\psi}\|^2 - 2\langle \hat{\phi}, \hat{\psi} \rangle \right) = \min \left(2 - 2\langle \phi, \psi / M \rangle \right) = \max \langle \phi, \psi \rangle.$$

Hence, hereinafter we consider the MIP search for ranking with a dense retriever as a NN search based on the Euclidean distance among the transformed embeddings $\hat{\phi}$ and $\hat{\psi}$ in $\mathbb{R}^{\ell+1}$. To simplify the notation, from now on we drop the hat symbol from the embeddings, i.e., $\hat{\phi} \to \phi$ and $\hat{\psi} \to \psi$, and we consider $\ell + 1$ as the new dimension ℓ, i.e., $\ell + 1 \to \ell$.
The index data structures for exact NN search in low dimensional spaces have been very successful, but they are not efficient with high dimensional data, as in our case, due to the curse of dimensionality. It is natural to make a compromise between search accuracy and search speed, and the most recent search methods have shifted to *approx-*

imate nearest neighbor (ANN) search.
The ANN search approaches commonly used in dense retrieval can be categorised into three families: locality sensitive hashing approaches, quantisation approaches, and graph approaches.

4.3 Locality sensitive hashing approaches

Locality sensitive hashing (LSH) [Indyk and Motwani 1998] is based on the simple idea that, if two embeddings are close together, then after a "projection", using an hash function, these two embeddings will remain close together. This requires that:

- for any two embeddings ψ_1 and ψ_2 that are close to each other, there is a high probability p_1 that they fall into the same hash bucket;
- for any two embeddings ψ_1 and ψ_2 that are far apart, there is a low probability $p_2 < p_1$ that they fall into the same hash bucket.

The actual problem to solve is to design a family of LSH functions fulfilling these requirements. LSH functions have been designed for many distance metrics. For the euclidean distance, a popular LSH function $h(\psi)$ is the *random projection* [Datar et al. 2004].A set of random projections defines a family of hash functions $\mathcal{H}$ that can be used to build a data structure for ANN search. First, we sample m hash functions $h_1(\psi), \ldots, h_m(\psi)$ independently and uniformly at random from $\mathcal{H}$, and we define the function family $\mathcal{G} = \{g : \mathbb{R}^\ell \rightarrow \mathbb{Z}^m\}$, where $g(\psi) = \big(h_1(\psi), \ldots, h_m(\psi)\big)$, i.e., g is the concatenation of m hash functions from $\mathcal{H}$. Then, we sample r functions $g_1(\psi), \ldots, g_r(\psi)$ independently and uniformly at random from $\mathcal{G}$, and each function g_i is used to build a hash table H_i.
Given the set of document embeddings Ψ and selected the values of the parameters r and m, an *LSH index* is composed of r hash tables, each containing m concatenated random projections. For each $\psi \in \Psi$, ψ is inserted in the $g_i(\psi)$ bucket for each hash table H_i, for $i = 1, \ldots, r$. At query processing time, given a query embedding, we first generate a candidate set of document embeddings by taking the union of the contents of all r buckets in the r hash tables the query is hashed to. The final NN document embedding is computed performing an exhaustive exact search within the candidate set.
The main drawback of the LSH index is that it may require a large number of hash tables to cover most nearest neighbors, and it requires to store the original embeddings to perform the exhaustive exact search. Although some optimisations have been proposed [Lv et al. 2007], the space consumption may be prohibitive with very large data sets.

4.4 Vector quantisation approaches

Instead of random partitioning the input space Ψ as in LSH, the input space can be partitioned according to the data distribution. By using the k-means clustering algorithm on Ψ we can compute k *centroids* $\mu_1, \ldots, \mu_k$, with $\mu_i \in \mathbb{R}^\ell$ for $i = 1, \ldots, k$ that can be used to partition the input space Ψ. The set $M = \{\mu_1, \ldots, \mu_k\}$ is called a *codebook*. Given a codebook M, a *vector quantiser* $q : \mathbb{R}^\ell \to \mathbb{R}^\ell$ maps a vector ψ to its closest centroid:

$$q(\psi) = \arg\min_{\mu \in M} \|\psi - \mu\| \tag{21}$$

Given a codebook M, an *IVF* (Inverted File) *index* built over M and Ψ stores the set of document embeddings Ψ in k *partitions* or *inverted lists* $L_1, \ldots, L_k$, where $L_i = \{\psi \in \Psi : q(\psi) = \mu_i\}$. At query processing time, we specify to search for the NN document embeddings in $p > 0$ partitions. If $p = k$, the search is exhaustive, but if $p < k$, the search is carried out in the partitions whose centroid is closer to the query embedding. In doing so the search is not guaranteed to be exact, but the search time can be sensibly reduced. In fact, an IVF index does not improve the space consumption, since it still needs to store all document embeddings, but it can reduce the search time depending on the number of partitions processed for each query.

A major limitation of IVF indexes is that they can require a large number of centroids [Gersho and Gray 1992]. To address this limitation, *product quantisation* [Jégou et al. 2011] divides each vector $\psi \in \Psi$ into m *sub-vectors* $\psi = [\psi_1|\psi_2|\cdots|\psi_m]$. Each sub-vector $\psi_j \in \mathbb{R}^{\ell/m}$ with $j = 1, \ldots, m$ is quantised independently using its own *sub-vector quantiser* q_j. Each vector quantiser q_j has its own codebook $M_j = \{\mu_{j,1}, \ldots, \mu_{j,k}\}$. Given the codebooks $M_1, \ldots, M_m$, a *product quantiser* $pq : \mathbb{R}^\ell \to \mathbb{R}^\ell$ maps a vector ψ into the concatenation of the centroids of its sub-vector quantisers:

$$pq(\psi) = [q_1(\psi_1)|q_2(\psi_2)|\cdots|q_m(\psi_m)] = [\mu_{1,i_1}|\mu_{2,i_2}|\ldots|\mu_{m,i_m}] \tag{22}$$

Note that a product quantiser can output any of the k^m centroid combinations in $M_1 \times \ldots \times M_m$.

A *PQ* (Product Quantization) *index* stores, for each embedding $\psi \in \Psi$, its encoding $i_1, \ldots, i_m$, that requires $m \log k$ bits of storage. At query processing time, the document embeddings are processed exhaustively. However the distance computation between a query embedding ϕ and a document embedding ψ is carried out using the product quantisation of the document embedding $pq(x)$:

$$\|\psi - \phi\|^2 \approx \|pq(\psi) - \phi\|^2 = \sum_{j=1}^{m} \|q_j(\psi_j) - \phi_j\|^2 \tag{23}$$

To implement this computation, m lookup tables are computed, one per sub-vector quantiser: the j-th table is composed of the squared distances between the j-th sub-vector of ϕ, and the centroids of M_j. These tables can be used to quickly compute the sums in Eq. (23) for each document embedding.
ANN search on a PQ index is fast, requiring only m additions, and memory efficient, but it is still exhaustive. To avoid it, an *IVFPQ index* exploits inverted files and product quantisation jointly. Firstly, a *coarse quantiser* partitions the input dataset into inverted lists, for a rapid access to small portions of the input data. In a given inverted list, the difference between each input data and the list centroid, i.e., the input *residual*, is encoded with a product quantiser. In doing so, the exhaustive ANN search can be carried out only in a limited number of the partitions computed by the coarse quantiser.

4.5 Graph approaches

The distances between vectors in a dataset can be efficiently stored in a graph-based data structure called *kNN graph*. In a kNN graph $G = (V, E)$, each input data $\psi \in \Psi$ is represented as a node $v \in V$, and, for its k nearest neighbours, a corresponding edge is added in E. The computation in an exact kNN graph requires $O(n^2)$ similarity computation, but many approximate variants are available [Dong et al. 2011]. To search for an approximate nearest neighbour to an element ϕ using a kNN graph, a *greedy heuristic search* is used. Starting from a predefined entry node, the graph is visited one node at a time, keeping on finding the closest node to ϕ among the unvisited neighbour nodes. The search terminates when there is no improvement in the current NN candidate. In practice, several entry nodes are used together with a search budget to avoid local optima. For a large number of nodes, the greedy heuristic search on the kNN graph becomes inefficient, due to the long paths potentially required to connect two nodes. Instead of storing only short-range edges, i.e., edges connecting two close nodes, the kNN graph can be enriched with randomly generated long-range edges, i.e., edges connecting two randomly-selected nodes. This kind of kNN graph is a *navigable small world (NSW) graph* [Malkov et al. 2013], for which the greedy search heuristic is theoretically and empirically efficient [Kleinberg 2000].
A *hierarchical NSW (HNSW) index* stores the input data into multiple NSW graphs. The bottom layer graph contains a node for each input element, while the number of nodes in the other graphs decreases exponentially at each layer. The search procedure for approximate NN vectors starts with the top layer graph. At each layer, the greedy heuristic searches for the closest node, then the next layer is searched, starting from the node corresponding to the closest node identified in the preceding graph. At the bot-

tom layer, the greedy heuristic searches for the k closest nodes to be returned [Malkov and Yashunin 2020].

4.6 Optimisations

Implementations of the embedding indexes presented in the previou sections are available in many open-source production-ready search engines such as Lucene[3] and Vespa[4]. In the IR research community, FAISS is the most widely adopted framework for embedding indexes [Johnson et al. 2021]. Among others, FAISS includes implementations of flat, LSH, IVF, PQ, IVFPQ and HNSW indexes.

Single representation systems such as DPR [Karpukhin et al. 2020], ANCE [Xiong et al. 2021], and STAR [Zhan et al. 2021b] use flat indexes. In these cases, it is unfeasible to adopt product quantisation indexes due to their negative impact on IR-specific metrics, mainly caused by the separation between the document encoding and embedding compression phases. To overcome this limitation, several recent techniques such as Poemm [Zhang et al. 2021], JPQ [Zhan et al. 2021a] and RepCONC [Zhan et al. 2022] aim to train at the same time both phases. In doing so, during training, these techniques learn together the query and document encoders together while performing product quantisation.

Multiple representation systems such as ColBERT [Khattab and Zaharia 2020] are characterised by a very large number of document embeddings. They do not use flat indexes, due to unacceptable efficiency degradation of brute-force search, and exploit IVFPQ indexes and ANN search. With these indexes, the document embeddings are stored in a quantised form, suitable for fast searching. However, the approximate similarity scores between these compressed embeddings are inaccurate, and hence are not used for computing the final top documents. Indeed, in a first stage, ANN search computes, for each query embedding, the set of the k' most similar document embeddings; the retrieved document embeddings for each query embedding are mapped back to their documents. These documents are exploited to compute the final list of top k documents in a second stage. To this end, the set of documents computed in the first stage is re-ranked using the query embeddings and the documents' multiple embeddings to produce exact scores that determine the final ranking, according to the relevance function in Eq. (12) (see Figure 9.

Further optimisations can reduce the number of query embeddings to be processed in the first stage [Tonellotto and Macdonald 2021], or the number of documents to be processed in the second stage [Macdonald and Tonellotto 2021].

[3] https://lucene.apache.org

[4] https://vespa.ai