# Recurrent Binary Embedding for GPU-Enabled Exhaustive Retrieval from Billion-Scale Semantic Vectors

Ying Shan
Bing Ads of AI & Research Group
Microsoft Corporation
Redmond, WA

Jian Jiao
Bing Ads of AI & Research Group
Microsoft Corporation
Redmond, WA

Jie Zhu
Bing Ads of AI & Research Group
Microsoft Corporation
Redmond, WA

JC Mao
Bing Ads of AI & Research Group
Microsoft Corporation
Redmond, WA

## ABSTRACT

Rapid advances in GPU hardware and multiple areas of Deep Learning open up a new opportunity for billion-scale information retrieval with exhaustive search. Building on top of the powerful concept of semantic learning, this paper proposes a Recurrent Binary Embedding (RBE) model that learns compact representations for real-time retrieval. The model has the unique ability to refine a base binary vector by progressively adding binary residual vectors to meet the desired accuracy. The refined vector enables efficient implementation of exhaustive similarity computation with bit-wise operations, followed by a near-lossless $k$-NN selection algorithm, also proposed in this paper. The proposed algorithms are integrated into an end-to-end multi-GPU system that retrieves thousands of top items from over a billion candidates in real-time.

The RBE model and the retrieval system were evaluated with data from a major paid search engine. When measured against the state-of-the-art model for binary representation and the full precision model for semantic embedding, RBE significantly outperformed the former, and filled in over 80% of the AUC gap in-between. Experiments comparing with our production retrieval system also demonstrated superior performance.

While the primary focus of this paper is to build RBE based on a particular class of semantic models, generalizing to other types is straightforward, as exemplified by two different models at the end of the paper.

## CCS CONCEPTS

**• Computing methodologies → Massively parallel algorithms**; **Machine learning algorithms**; **Supervised learning**; **Neural networks**; **Learning latent representations**; **• Information systems → Document representation**; **Query representation**; *Sponsored search advertising*;

## KEYWORDS

Deep Learning; Deep Neural Network (DNN); Semantic Embedding; Binary Network; Information Retrieval; $k$-NN; CDSSM; CNTK; GPU; Sponsored Search

**ACM Reference Format:**
Ying Shan, Jian Jiao, Jie Zhu, and JC Mao. 2018. Recurrent Binary Embedding for GPU-Enabled Exhaustive Retrieval from Billion-Scale Semantic Vectors. In *KDD '18: The 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, August 19–23, 2018, London, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3219819.3220018

## 1 INTRODUCTION

In the age of information explosion, human attention remains a single threaded process. As the key enabler finding where to focus, information retrieval (IR) becomes ubiquitous, and is at the heart of modern applications including web search, online advertising, product recommendation, digital assistant, and personalized feed.

In IR's almost 100-year history [27], a major milestone around 60s-70s [26] was to view queries and documents as high dimensional term vectors, and measure lexical similarity using the cosine coefficient. Since then, the mainstream development continuously refined the weights of the terms. Seminal works included *term frequency* (*tf*), combined *tf* and inverted document frequency (*idf*), the *binary independence model* [35], and the less probabilistic but highly effective BM25 [24].

*Latent semantic analysis* (LSA) [9] marked the beginning of matching queries and documents at the semantic level. As a result, queries and documents relevant in semantics can score high in similarity, even though they are lexically disjoint. Inspired by LSA, a number of probabilistic topic models were proposed and successfully applied to semantic matching [4, 14, 33].

Recent trends have seen the blending of the latent analysis with DNNs. Models such as *semantic hashing* [25] and *word2vec* [21] learned word and phrase embeddings through various DNNs. Due to weak semantic constraints in the training data, they are not strictly semantic. However, the connections empowered latent analysis with the latest technologies and tools developed in the DNN community.

The *deep structured semantic model* (DSSM) [15] was among the first DNNs that learned truly semantic embeddings based on search logs. Applying user clicks as labels enabled a discriminative objective function optimized for differentiating the relevant from the irrelevant. It significantly outperformed models with objectives only loosely coupled with IR tasks. DSSM was later upgraded to the *convolutional deep structured semantic model* (CDSSM) [29], by adding word sequence features through a convolution-pooling structure.

While semantic embedding is advantageous as a representation, online retrieval has to solve the high-dimensional $k$-nearest neighbor ($k$-NN) problem. The key challenge is to achieve a balanced goal of retrieval performance, speed, and memory requirement, while dealing with the *curse of dimensionality* [1, 32].

This paper proposes a novel semantic embedding model called *Recurrent Binary Embedding* (RBE), which is designed to meet the above challenge. It is built on top of CDSSM, and inherits the benefits of being discriminative and order sensitive. The representation is compact enough to fit over a billion documents into the memory of a few consumer grade GPUs. The similarity computation of RBE vectors can fully utilize the SIMT parallelism, to the extent that a $k$-NN selection algorithm based on exhaustive search[1] is feasible in the range of real-time retrieval. As a result, the curse of dimensionality that has been haunting the decades-long research of *approximate nearest neighbor* (ANN) [7, 11] has little effect[2].

To our best knowledge, this is the first time a brute-force $k$-NN is applied to a billion-scale application, sponsored search in this case, for real-time retrieval. A salient property of RBE is that the retrieval accuracy can be optimized based on hardware capacity. Looking ahead, we expect the baseline established in this paper will be continuously refreshed by more powerful and cheaper hardware, in addition to algorithmic advances.

The paper is arranged in the following structure. Sec. 2 sets up the context of the application domain where RBE is developed. Sec. 3 states the problem formally and provides a motivational example. Sec. 4 introduces the RBE model and the architecture. Sec. 5 gives an overview of the end-to-end retrieval system based on RBE. Sec. 6 provides the details of the GPU-based $k$-NN selection algorithm. Sec. 7, 8, and 9 provide details for experiment settings, data, and results. Sec. 10 calls out the connections and differences of this paper with related work. Sec. 11 concludes the paper and points out the future directions.

## 2 SPONSORED SEARCH

RBE is discussed in the context of *sponsored search* of a major search engine. Readers can refer to [10] for an overview on this subject. In brief, sponsored search is responsible for showing *ads* (advertisement) alongside organic search results. There are three major agents in the ecosystem including the user, the advertiser, and the search platform. The goal of the platform is to display a list of ads that best match user's intent. Below is the minimum set of key concepts for the discussions that follow.

**Query:** A text string that expresses user intent. Users type queries into the search box to find relevant information

**Keyword:** A text string that expresses advertiser intent. Keywords are not visible to users, but play a pivotal role in associating advertiser intent with user intent

[1]Sometimes referred to as *brute-force* search. They will be referred to interchangeably

[2]Due to the curse of dimensionality, ANNs exploring a 5% fraction of a 128-dimensional hyper unit-cube have to search 98% of each coordinate [34]. Brute-force search that matches against the entire document set is not subject to this predicament

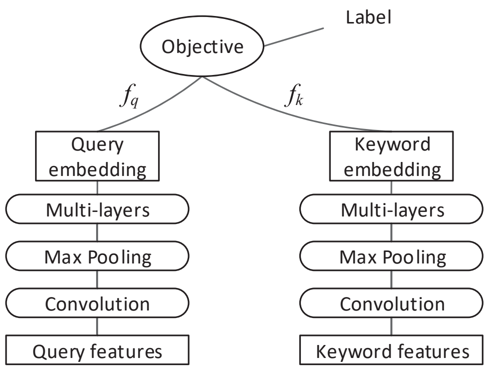

**Figure 1: The CDSSM model architecture**

**Impression:** An ad being displayed to a user, on the result page of the search engine

**Click:** An indication that an impressed ad is clicked by a user

On the backend of a paid search engine, the number of keywords are typically at the scale of billions. IR technologies are applied to reduce the amount of keywords sent to the downstream components, where more complex algorithms are used to finalize the ads to display. A click event is recorded when an impressed ad is clicked on.

To be consistent with the above context, *keyword* is used instead of *document* throughout the paper. The query and the keyword associated with a click event is referred to as a *clicked pair*, which is the source of positive samples for many paid search models, including RBE.

## 3 PROBLEM STATEMENT

Our goal is to find a vector representation that balances the retrieval performance, speed, and storage requirement as mentioned in Sec. 1. For the subsequent discussions, $q$ and $k$ will be used to denote query and keyword, respectively.

As mentioned in Sec. 1, there are many ways of representing query and keywords with vectors. This paper primarily focuses on semantic vectors produced by the CDSSM model [29].

### 3.1 A Brief Recap on CDSSM

Fig. 1 is the high-level architecture of the CDSSM model, which consists of two parallel feed-forward networks. On the left side, query features are mapped from a sparse representation called tri-letter gram[3] to a real-valued vector, through transforms including *convolution*, *max pooling*, and multiple hidden layers (*multi-layers*). The right side for keywords uses the same transformation structure, but with a different set of parameters.

For training, a positive sample is constructed from a clicked pair as mentioned in Sec. 2. Negative samples are sometimes generated from positive samples through cross sampling[4]. For each sample $s$, CDSSM produces a pair of vectors as in Fig. 1, where $\mathbf{f}_q(s)$ and $\mathbf{f}_k(s)$ are for the query, and the keyword, respectively.

[3]A form of 3-shingling at the letter level

[4]Given two pairs of positive samples $\langle q_1, k_1\rangle$ and $\langle q_2, k_2\rangle$, cross sampling produces two negative samples $\langle q_1, k_2\rangle$ and $\langle q_2, k_1\rangle$

A *sample group* $S$ includes one positive sample and a fixed number of negative samples. It is the basic unit to evaluate the objective function:

$$O(\mathcal{S};\Theta) = -\sum_{S\in\mathcal{S}} \log P(S), \tag{1}$$

where $\mathcal{S}$ is the set of all sample groups, and $\Theta$ is the set of all CDSSM parameters. In the above equation:

$$P(S) \equiv P(s^+|S) = \frac{\exp(\gamma\ \beta(\mathbf{f}_q(s^+), \mathbf{f}_k(s^+)))}{\sum_{s\in S}\exp(\gamma\ \beta(\mathbf{f}_q(s), \mathbf{f}_k(s)))}, \tag{2}$$

where $P(s^+|S)$ is the probability of the only positive query-keyword pair in the sample group, given $S$. The smoothing factor $\gamma$ is set empirically on a held-out data set. The similarity function is defined as the following:

$$\beta(\mathbf{f}_q, \mathbf{f}_k) \equiv \cos(\mathbf{f}_q, \mathbf{f}_k) = \frac{\mathbf{f}_q \cdot \mathbf{f}_k}{\|\mathbf{f}_q\|\ \|\mathbf{f}_k\|}. \tag{3}$$

## 3.2 Formulating the Goal

The goal is to design a model with specially constructed embedding vectors $\mathbf{b}_q^u$ and $\mathbf{b}_k^v$, such that objective function in (1) is minimized:

$$\arg\min_{\Theta} O(\mathcal{S};\Theta). \tag{4}$$

Unlike real-valued $\mathbf{f}_q$ and $\mathbf{f}_k$, $\mathbf{b}_q^u$ and $\mathbf{b}_k^v$ can be decomposed into a series of $u$+1 and $v$+1 binary vectors. The similarity function in (2) and (3) becomes:

$$\beta(\mathbf{b}_q^u, \mathbf{b}_k^v) \equiv \cos(\mathbf{b}_q^u, \mathbf{b}_k^v). \tag{5}$$

The full definition of $\mathbf{b}_q^u$ and $\mathbf{b}_k^v$ is deferred to Sec. 4.1. The following is an example to motivate the goal of binary decomposition. Suppose we have:

$$\begin{aligned} \mathbf{b}_q^1 &= \mathbf{b}_q^0 + \mathbf{d}_q^0 \\ \mathbf{b}_k^1 &= \mathbf{b}_k^0 + \mathbf{d}_k^0, \end{aligned} \tag{6}$$

where $\mathbf{b}_q^0$, $\mathbf{d}_q^0$, $\mathbf{b}_k^0$, $\mathbf{d}_k^0$ are binary vectors, the cosine similarity becomes:

$$\begin{aligned} \cos(\mathbf{b}_q^1, \mathbf{b}_k^1) &= \frac{\mathbf{b}_q^1 \cdot \mathbf{b}_k^1}{\|\mathbf{b}_q^1\|\ \|\mathbf{b}_k^1\|} \\ &= \frac{(\mathbf{b}_q^0\cdot\mathbf{b}_k^0 + \mathbf{b}_q^0\cdot\mathbf{d}_k^0 + \mathbf{d}_q^0\cdot\mathbf{b}_k^0 + \mathbf{d}_q^0\cdot\mathbf{d}_k^0)}{\|\mathbf{b}_q^1\|\ \|\mathbf{b}_k^1\|}. \end{aligned} \tag{7}$$

As demonstrated in (7), the binary decomposition turns similarity computation into a series of dot products of binary vectors, which can be implemented efficiently on modern hardware including GPUs. The hardware enabled computation, together with the compact representation as binary vectors, form the foundation of our work.

## 4 RECURRENT BINARY EMBEDDING

To construct the embedding vectors in (5), we propose a deep learning model called *Recurrent Binary Embedding*, or RBE. The model is learned from a training set with clicked pairs. The learned model is applied to generate query and keyword embedding vectors for retrieval.

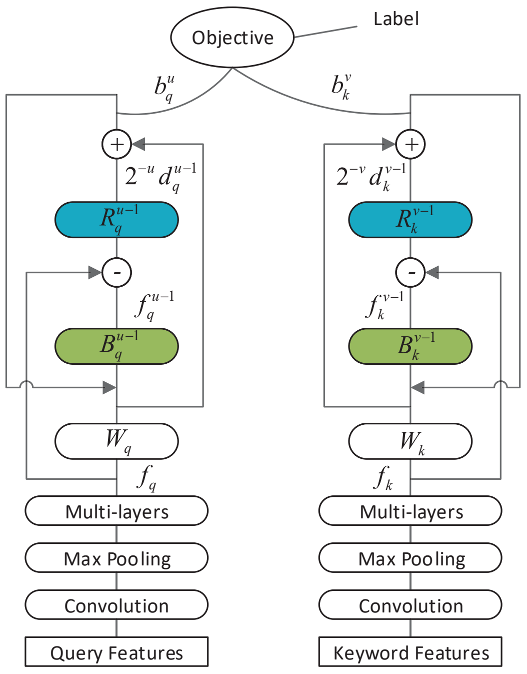

**Figure 2: The Recurrent Binary Embedding (RBE) model**

## 4.1 The Model and the Architecture

Fig. 2 is the model architecture of RBE. As compared with the CDSSM model in Fig. 1, RBE also has two separate routes for embedding, and shares the same forward processes up to the multi-layers transformations. The parts beyond the multi-layers will be referred to as *RBE layers* hereafter, and are formulated by the following equations:

$$\mathbf{b}_i^0 = \rho\ (\mathbf{W}_i \cdot \mathbf{f}_i) \tag{8}$$

$$\mathbf{f}_i^{t-1} = \tanh(\mathbf{B}_i^{t-1} \cdot \mathbf{b}_i^{t-1}) \tag{9}$$

$$\mathbf{d}_i^{t-1} = \rho\ (\mathbf{R}_i^{t-1} \cdot (\mathbf{f}_i - \mathbf{f}_i^{t-1})) \tag{10}$$

$$\mathbf{b}_i^t = \mathbf{b}_i^{t-1} + \left(\frac{1}{2}\right)^t \mathbf{d}_i^{t-1}, \tag{11}$$

where $\mathbf{f}_i \in \mathbb{R}^m$, $t \geq 1$ is the time axis discussed later in Sec. 4.5, and $\mathbf{b}_i^{t-1} \in \{-1, 1\}^n$. Bias terms are dropped for simplicity. The key idea behind the above equations is to construct the binary decomposition $\mathbf{b}_i^t$ by maximizing the information extracted from the real-valued vectors $\mathbf{f}_i$. A number of intermediate vectors are involved during the training process to achieve this objective.

**The base vector** Equation (8) is where the real-valued embedding vector $\mathbf{f}_i$ (in float) is transformed by an $n \times m$ matrix $\mathbf{W}_i$, where the index $i$ is either $q$ or $k$. It is then mapped into a binary *base vector* $\mathbf{b}_i^0$ through the binarization function $\rho$, discussed with details in Sec. 4.2

**The reconstructed vector** Equation (9) converts the $n$ dimensional binary vector $\mathbf{b}_i^{t-1}$ back to an $m$ dimensional *reconstructed vector* in float. The transformation is through an $m \times n$ matrix $\mathbf{B}_i^{t-1}$, followed by an element-wise tanh

**The residual vector** Equation (10) transforms the difference between $\mathbf{f}_i$ and $\mathbf{f}_i^{t-1}$ by an $n \times m$ matrix $\mathbf{R}_i^{t-1}$, followed by $\rho$. The transformed binary vector $\mathbf{d}_i^{t-1}$ is the *residual vector*, because it is transformed from the residual between the original embedding vector and the reconstructed vector

**The RBE embedding** Equation (11) creates a *refined vector* by recursively adding residual vectors from the previous time stamps, multiplied by a *residual weight* $2^{-t}$ detailed in Sec. 4.3. The last refined vector is the *RBE embedding*. The binary vectors adding up to form the RBE embedding are the *ingredient vectors*

At the top of Fig. 2, RBE embeddings from both sides are used to evaluate the objective function as described in (4) and (5).

## 4.2 The Binarization Function

The binarization function $\rho$ in (8) and (10) plays an important role in training the RBE model. In the forward computation, it converts float input $x$ into a binary value of either $-1$ or $+1$ as the following:

$$\rho(x) \equiv \text{sign}(x) = \begin{cases} -1, & \text{if } x \leq 0 \\ 1, & \text{otherwise.} \end{cases} \tag{12}$$

The backward computation is problematic since the gradient of the sign function vanishes almost everywhere. Different gradient estimators were proposed to address the problem.

The straight-through estimator takes the gradient of the identity function as the estimate of $\rho'(x)$ [3]. A variant of this estimator, found to have better convergence property, sets $\rho'(x) = 1$ when $|x| \leq 1$ and $\rho'(x) = 0$ otherwise [6]. An unbiased gradient estimator was proposed in [3], but did not reach the same level of accuracy in practice.

Another estimator mimics the discontinuous sign function with an annealing tanh function in backward propagation [5, 6]. The annealing tanh function approaches a step function when the *annealing slope* $\alpha$ increases:

$$\lim_{\alpha \to \infty} \tanh(\alpha x) = \text{sign}(x). \tag{13}$$

The slope is initialized with $\alpha = 1$, and is increased gently to ensure convergence. Sec. 9.2 compares the performance of the above estimators referred to later as *straight-through*, *straight-through variant*, and *annealing tanh*, respectively.

## 4.3 Residual Weights

The presence of the residual weight in (11) may seem natural and intuitive. However, a closer look reveals a profound implication to the richness of the RBE embedding.

Recall that each dimension of a binary vector in (11) takes a value of either $-1$ or 1. When two binary vectors are added without the residual weight as in (6), each dimension will end up with a value in $\{-2, 0, 2\}$. However, if the same vectors are added with the residual weight, the set of possible values becomes $\{-1.5, -0.5, 0.5, 1.5\}$. As a result, the cardinality of all RBE embeddings with two ingredient vectors increases from $3^n$ to $4^n$. In general, it can be proved[5] that for RBE embeddings with $j = t + 1$ ingredient vectors, the cardinality grows $(2^j/(j+1))^n$ times by introducing residual weights. This leads to a substantial boost in accuracy as demonstrated later in Sec. 9.3.

[5]Not elaborated here due to space limit

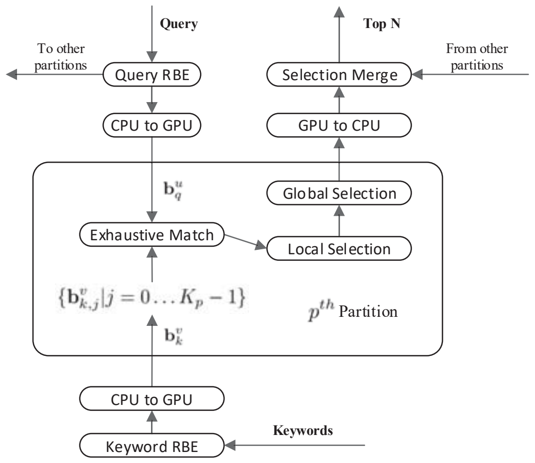

**Figure 3: The RBE GPU-enabled Information Retrieval (rbeGIR) system**

The base of the weighting schema is set to $\frac{1}{2}$ due to equal distance values for each dimension, and hardware enabled implementation with a bit-wise operation.

## 4.4 The Recurrent Pattern

RBE gets its name from the looped pattern exhibited in the equations of (9), (10), and (11). The pattern is also obvious in Fig. 2, where $\mathbf{B}_i^{t-1}$ and $\mathbf{R}_i^{t-1}$ alternate to generate the reconstructed vector and the residual vector, and to refine the base vector iteratively. While the word "recurrent" may imply connections with other network structures such as RNN and LSTM, the analogy does not go beyond the looped pattern.

The primary difference is in the purpose of the repeating structures. For RNN and LSTM-alike models, the goal is to learn a persistence memory unit. As a result, the transformations share the same set of parameters from $t-1$ to $t$. In contrast, RBE has the flexibility to decide whether the parameters of $\mathbf{B}_i^{t-1}$ and $\mathbf{R}_i^{t-1}$ should be time varying or fixed. This helps to optimize the system under various design constraints.

## 4.5 Two Time Axes

RBE has two time axes $u$ and $v$ for the query side, and the keyword side, respectively. They are the same as $t$ in the equations of (9), (10), and (11). The two time axes are independent, and can have different numbers of iterations to produce RBE embeddings. This is another flexibility RBE provides to meet various design constraints. The benefit will be made clear in Sec. 9, where RBE models with different configurations are implemented and compared.

## 5 RBE-BASED INFORMATION RETRIEVAL

A system for keyword retrieval is built based on RBE embeddings. Fig. 3 outlines the high-level architecture of the system, referred to as RBE GPU-enabled Information Retrieval, or rbeGIR.

The system uses multiple GPUs to store and process the RBE embeddings. The rounded rectangle in the middle of Fig. 3 represents the key components of the $p^{th}$ GPU, where the corresponding data partition stores RBE embeddings represented by $\{\mathbf{b}^v_{k,j} | j = 0 \ldots K_p - 1\}$. The raw keywords, $K_p$ in total for the $p^{th}$ partition, are transformed offline to vectors through the keyword side of the model in Fig. 2. They are uploaded to GPU memory from CPU memory as illustrated on the bottom of Fig. 3.

At run time, a query embedding $\mathbf{b}^u_q$ is generated on-the-fly by the query side of the RBE model. The same embedding is also sent to other GPUs as shown on the upper left side of the figure. The *exhaustive match* component inside the GPU is where the similarity function in (5) is evaluated for all pairs of $\langle \mathbf{b}^u_q, \mathbf{b}^v_{k,j} \rangle$. The similarity values are used to guide the per thread *local selection* and the per GPU *global selection* to find the best keywords from the $p^{th}$ partition. The results from all GPUs will be used to produce the top $N$ keywords, through the *selection merge* process[6].

## 5.1 Dot Product of RBE Embeddings

Section 3.2 touched upon the dot product of RBE embeddings with a specific example. More generally, from (5), (8), and (11) we have:

$$\cos(\mathbf{b}^u_q, \mathbf{b}^v_k) \propto \frac{1}{\|\mathbf{b}^v_k\|} \left( \mathbf{b}^0_q \cdot \mathbf{b}^0_k + \sum_{j=0}^{u-1} \sum_{i=0}^{v-1} \left(\frac{1}{2}\right)^{j+i+2} \mathbf{d}^j_q \cdot \mathbf{d}^i_k \right. \\ \left. + \sum_{j=0}^{u-1} \left(\frac{1}{2}\right)^{j+1} \mathbf{b}^0_q \cdot \mathbf{d}^j_k + \sum_{i=0}^{v-1} \left(\frac{1}{2}\right)^{i+1} \mathbf{b}^0_k \cdot \mathbf{d}^i_q \right), \tag{14}$$

where the magnitude of the query side embedding is dropped because it's the same for all keywords. Equation (14) decomposes the dot product of RBE embeddings into dot products of binary vectors, which can be implemented with bit-wise operations as the following:

$$\mathbf{x} \cdot \mathbf{y} = (popc(\mathbf{x} \wedge \mathbf{y}) \gg 1) + n, \tag{15}$$

where $\mathbf{x}$ and $\mathbf{y}$ are vectors in $\{-1, 1\}^n$. On the right side of (15), *popc*, "$\wedge$", and "$\gg$" are the *population count*, XOR, and logical right shift operators, respectively. Multiplying the residual weights also uses the right shift operator, which is executed at most $u + v$ times by carefully ordering the results of binary dot products[7]. Since the keyword side magnitude $\|\mathbf{b}^v_k\|$ is usually precomputed and stored with the RBE embeddings, the computation of cosine similarity boils down to a series of binary operations that can be accelerated with (15), which enables the exhaustive match component in rbeGIR.

## 5.2 Asymmetric Design

In general, increasing $u$ and $v$ in (14) improves accuracy, but comes with the cost of memory and speed. A key observation is that the memory impact of $u$ is negligible, which suggests an *asymmetric design* with $u > v$. This is feasible thanks to the independent time axes mentioned in Sec. 4.5. However, adding more ingredient vectors on the query side impacts speed. The trade-off will be studied later with experiments.

[6]The selection merge process actually relies on a GPU-based Radix sort, which is omitted in Fig. 3 for simplicity

[7]As an illustrative example, computing $\frac{1}{2}a + \frac{1}{4}b$ costs 3 shifts with $(a \gg 1) + (b \gg 2)$, but only 2 shifts with $(a + b \gg 1) \gg 1$

## 5.3 Key Advantages

With RBE embedding, storing one billion keywords needs only 14.90GB memory, instead of 238GB using float. This makes in-memory retrieval possible on a few GPUs. The similarity function is computed exhaustively for all query-keyword pairs with a $k$-NN selection algorithm discussed in Sec. 6, which reduces false negatives to almost zero. Also because of exhaustive matching, there is no need to make implicit or explicit assumptions about data distributions, resulting in consistent retrieval speed and accuracy. RBE learns application specific representations, and is more accurate than general purpose quantization algorithms. It is more accurate than other learning-based binarization algorithms due to the recurrent structure.

In addition, rbeGIR mostly uses primitive integer operations available on consumer grade GPUs. As a result, the implementation on low-end GPUs may even outperform high-end ones with higher FLOPS and price tag. It is also straightforward to port the system to different hardware platforms.

## 6 EXHAUSTIVE $k$-NN SELECTION WITH GPU

At the core of the rbeGIR system is a GPU-based brute-force $k$-NN selection algorithm designed for billion-scale retrieval. The selection algorithm starts from a local selection process that relies on a $k$-NN kernel outlined in Algorithm 1. A kernel is a function replicated and launched by parallel threads on the GPU device, each with different input data. In Algorithm 1, $I$ is the number of keywords to process per thread, $T_b$ is the number of threads per block[8], $x$ is the block id, $y$ is the thread id, and $z$ is the memory offset to the keyword vectors. The input $\mathbf{b}^v_{k,j\in\Omega_y}$ denotes the RBE embeddings of all keywords $\Omega_y$ processed by the $y^{th}$ thread. The cosine similarity function is implemented as described in Eq. 14 and Sec. 5.1.

**Algorithm 1:** The $k$-NN Kernel

```
Input:  The RBE embedding for the query $\mathbf{b}^u_q$
        The RBE embeddings for keywords $\mathbf{b}^v_{k,j\in\Omega_y}$
Output: Priority queue $p$ containing top similarity scores and
        their indices
1 $z \leftarrow x * T_b * I + y$ ;
2 $p$.clear() ;
3 for $i \leftarrow 1$ to $I$ do
4     $s \leftarrow \cos(\mathbf{b}^u_q, \mathbf{b}^v_{k,z})$;
5     $p$.insert($s$, $z$);
6     $z \leftarrow z + T_b$;
7 end
8 return $p$;
```

The returned priority lists are sent to the global selection process, and the merge selection process as mentioned in Sec. 5. Both processes leverage the Radix sort method mentioned in [19], which is one of the fastest GPU-based sorting algorithms.

[8]A GPU block is a logical unit containing a number of coordinated threads and certain amount of shared resource

## 6.1 Performance Optimization

The design of the brute-force $k$-NN takes into consideration several key aspects of the GPU hardware to save time. First, the global selection process handles only the candidates in the priority lists. This avoids extensive read and write operations in the global memory. Second, a number of sequential kernels are fused into a single thread, which takes the full advantage of thread level registers instead of much slower shared memory and global memory.

The storage of RBE embeddings is also re-arranged to utilize the warp-based memory loading pattern. Instead of organizing the embeddings by keywords, the base vectors and residual vectors are grouped and stored separately. In the case of RBE embeddings with two ingredient vectors, the base vectors are stored first in a continuous memory block, followed by the residual vectors. This makes it possible for a warp (32 consecutive threads) to load 128 bytes of memory in a single clock cycle, which improves the kernel speed significantly.

## 6.2 Negligible Miss Rate

The $k$-NN kernel computes similarity scores exhaustively for all keywords. However, the local selection process relies on a priority queue which is a lossy process. The key insight is that the miss rate of the algorithm is negligible when $N \ll C$, where $N$ is usually in the range of thousands, and the number of candidates $C$ is in the billions. As explained in Appendix B, for $N = 1000$, $C = 10^9$, and $I = 256$, the probability of missing more than two relevant keywords is less than 0.03%. In practice, setting the length of the priority queue to be 1 is sufficient.

## 7 EXPERIMENT SETTINGS

The RBE model was implemented with BrainScript in CNTK[9], and trained on a GPU cluster. The CDSSM components and the objective function were built from existing CNTK nodes. The binarization function $\rho$ was implemented as a customized node in CNTK, and exposed to BrainScript. The recurrent embedding was unfolded into a series of feed-forward layers, as detailed in Appendix A. The rbeGIR system was implemented from the ground up on a customized multi-GPU server in Fig. 4.

The convolution layer of the RBE model mapped a sliding group of three words (from either query or keyword input) to a 288 dimensional (dim) float vector. Each group of input was a sparse vector of 49292-dim tri-letter gram. The max-pooling layer produced an $m = 288$-dim vector $\mathbf{f}_i$, which was transformed to an $n = 64$-dim base vector $\mathbf{b}_i^0$ as in (8). Time varying matrices of $\mathbf{B}_i^{t-1}$ and $\mathbf{R}_i^{t-1}$ in (9) and (10) were $288 \times 64$, and $64 \times 288$, respectively. Multi-layers in Fig. 2 were not used due to limited performance gain.

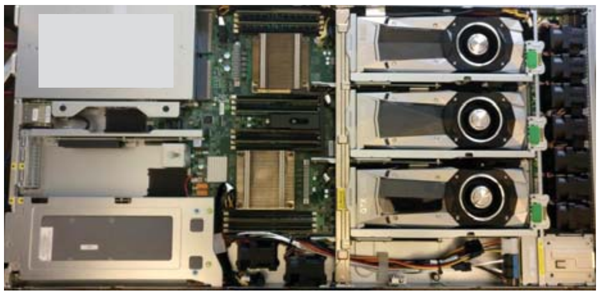
Figure 4: The rbeGIR server with four NVIDIA GeForce GTX1080 GPUs, two 8-core CPUs, and 128GB DDR memory

The rbeGIR system used 256 threads in a single GPU block, where each thread launched the $k$-NN kernel to process $I = 256$ keywords. The total number of blocks was $136 \times 136$, arranged on a two dimensional grid.

The experiments were based on the data collected from our paid search engine. The training data for the RBE model contained 175 million (M) unique clicked pairs sampled from two-year's worth of search logs. Adding 10 negative samples generated per clicked pair through cross sampling, the total number of training samples amounted to 1925M. The validation data had 1.2M pairs sampled a month after to avoid overlap. The test data consisted of 0.8M pairs labeled by human judges. Pairs labeled[10] as *bad* were considered as negative samples, and the rest were considered as positive ones.

## 8 MAIN RESULTS

The main results are reported based on a 64-dim RBE model of our choice, referred to hereafter as rbe*. The model used three ingredient vectors for queries, and two for keywords. Since ingredient vectors are stored separately, each dimension of the RBE embedding used three bits for queries, and two for keywords. Straight-through variant and residual weights were applied across all models mentioned in this section.

## 8.1 Model Accuracy

Four models in Table 1 were evaluated against rbe* with accuracy. The first model, referred to as m_1, is a CDSSM model with 64-dim embedding layers in float[11]. This sets the upper bound for binarization models including RBE. The m_2 model replaces the embedding layers in m_1 with binarization layers using the state-of-the-art straight-through variant. The m_3 model is the same as m_2 but with 128-dim embedding, which represents the performance of 2-bit binarization without changing the structure. The m_5 model has full precision embedding for queries, and RBE embedding with two ingredients for keywords. It sets the upper bound for RBE models with 2-bit binarization like rbe*.

**Table 1: Models configurations for accuracy comparison**

| Model | Dimension | $q$ (bits) | $k$ (bits) |
|---|---|---|---|
| m_1: full precision CDSSM | 64 | 32 | 32 |
| m_2: state-of-the-art | 64 | 1 | 1 |
| m_3: state-of-the-art | 128 | 1 | 1 |
| m_4: rbe* | 64 | 3 | 2 |
| m_5: hybrid rbe | 64 | 32 | 2 |

Two metrics are used for comparison in Table 2. The first metric is *log loss* defined in (1), which measures the difference between the distributions of the predicted similarity and the click labels. Log loss is applied to both the training set and the validation set. The second metric is ROC AUC measured on the test data set with

[9]https://docs.microsoft.com/en-us/cognitive-toolkit/

[10]See Sec. 8.2 for the details of labels

[11]Specifically, the CDSSM model has a 64-dim tanh layer on top of the max pooling layer as described in Sec. 7

human labels. The last column of the table is the *AUC lift* defined based on the AUC difference 0.0198 between m_3 and m_1, referred hereafter as the *reference gap*. As an example, the lift for rbe* is $0.0159/0.0198 * 100 = 80.30\%$, where the numerator is the AUC improvement from m_3 to rbe*.

**Table 2: Accuracy of embedding models**

| Model | Log Loss | ROC AUC | AUC Lift % |
|---|---|---|---|
| m_1: full precision CDSSM | 0.0293 | 0.8044 | - |
| m_2: state-of-the-art | 0.0481 | 0.7719 | -64.14 |
| m_3: state-of-the-art | 0.0425 | 0.7846 | - |
| m_4: rbe* | 0.0312 | 0.8005 | 80.30 |
| m_5: hybrid rbe | 0.0311 | 0.8011 | 83.33 |

Observing from Table 2, the 1-bit increase per dimension improves the accuracy significantly from m_2 to m_3. Without increasing bit per dimension for keywords, rbe* lifts the AUC by more than the amount from m_2 to m_3, and is only 3.03% away from the hybrid upper bound of m_5. The log loss values exhibit similar gains.

## 8.2 The Quality of Retrieval

RbeGIR was evaluated against our production retrieval system[12] with the quality of returned keywords. The production setting included one with the same amount of memory (prod_1), and another with the same amount of keywords (prod_2). Since rbeGIR does not use extra indexing structure, only the amount of memory used for keyword vectors were counted for prod_1. The rbeGIR system stored the embeddings of 1.2 billion unique keywords.

**Table 3: Top five results by quality**

| Baseline | Bad | Fair | Good | Excellent |
|---|---|---|---|---|
| prod_1 | -52.37 | -9.73 | 18.52 | 18.83 |
| prod_2 | -35.26 | 3.32 | 11.19 | 4.39 |

Table 3 reports the average quality of the top five keywords returned from each of 2000 queries. Based on a production quality guideline, query-keyword pairs were manually judged with a score of *bad*, *fair*, *good*, or *excellent*. Each column in the table is the percentage change between the counts of query-keyword pairs from rbeGIR and the baseline system by scores. As an example, rbeGIR retrieved 18.52% more good pairs than prod_1. From Table 3, rbeGIR has significantly reduced the amount of bad pairs for both prod_1 and prod_2. It found less fair pairs than prod_1, but otherwise substantially more good and excellent pairs than the production baselines. It was also observed that there were about $8-11\%$ overlap of the good or excellent pairs between rbeGIR and the baselines.

## 8.3 Recall and Latency

To evaluate the recall, 10000 queries were first matched offline with 1.2 billion keywords through exact nearest neighbor, using RBE embeddings generated by the rbe* model. The per query recall @1000 is defined by the total number of top keywords overlapping with the relevant keywords, divided by 1000. It was observed that the average recall @1000 for rbeGIR is 99.99%, which is expected per Appendix B.

[12]Unfortunately, the details of the system is not available for publishing

The latency for rbe_2 and rbe* are on average 29.92ms, and 31.17ms, respectively. Both models are in the range of real-time retrieval. Adding one more bit from rbe_2 to rbe* on the query side increased the query time by 4.18%. The latency of the full precision model was measured on a down-sampled keyword set (20M) due to the memory limit. As compared with the rbe* model using the same keyword set, the latency of the full precision model was around ten times higher.

## 9 EFFECTS OF DESIGN CHOICES

RBE and the rbeGIR system require a limited amount of tuning. Only a few design choices were experimented before finalizing the rbe* model in Sec. 8.

## 9.1 Number of Bits per Dimension

Table 4 lists models similar to rbe* but with different number of bits per dimension. These models were created by adjusting the number of iterations $u$ and $v$. The m_2 model is the same as in Table 1, with one bit for $q$ and $k$. As shown in Table 4, model accuracy improves as the number of bits per dimension increases.

**Table 4: Model accuracy with varying bits per dimension**

| Model | | | Log Loss | ROC AUC |
|---|---|---|---|---|
| m_2: | $q = 1$ | $k = 1$ | 0.0481 | 0.7719 |
| rbe_2: | $q = 2$ | $k = 2$ | 0.0333 | 0.7972 |
| rbe*: | $q = 3$ | $k = 2$ | 0.0312 | 0.8005 |
| rbe_3: | $q = 3$ | $k = 3$ | 0.0294 | 0.8034 |

The best model rbe_3 in the table is only 5.05% reference gap away from the m_1 model in Table 1. However, rbe* was chosen over rbe_3 because adding one more ingredient vector for keywords requires 50% more memory. Comparing rbe* with rbe_2 demonstrates the advantage of using asymmetric design mentioned in Sec. 5.2. Using one more bit on the query side, rbe* gains 16.67% over rbe_2 without additional memory.

## 9.2 Type of Binarization Functions

Three binarization algorithms mentioned in Sec. 4.2 were experimented with for RBE training. The original straight-through had difficulty converging when $u$ or $v$ was larger than 2, but the straight-through variant mentioned in Sec. 4.2 converged consistently. The results in Table 5 compare the AUC performance of the straight-through variant and annealing tanh.

**Table 5: Performance of different binarization methods**

| Model | | | ST Variant | Annealing Tanh |
|---|---|---|---|---|
| rbe_1: | $u = 0$ | $v = 0$ | 0.7719 | 0.7730 |
| rbe_2: | $u = 1$ | $v = 1$ | 0.7972 | 0.7950 |
| rbe*: | $u = 2$ | $v = 1$ | 0.8005 | 0.8007 |
| rbe_3: | $u = 2$ | $v = 2$ | 0.8034 | 0.8014 |

Based on Table 5, annealing tanh performs better than the straight-through variant on the rbe_1 model. However, as the number of iterations increases, the straight-through variant (referred to as "ST variant" in the table) shows better performance overall[13]. This is likely caused by the small gradient of annealing tanh, especially as the annealing slope increases over time. With more iterations, the gradient vanishes more easily due to the chain effect, making it hard to improve the binarization layers.

Based on the above experiments, rbe* adopted the straight-through variant for binarization.

### 9.3 Inclusion of Residual Weights

Table 6 reports the AUC performance for three models with and without residual weights. Using the reference gap defined in Sec. 8, the lift in accuracy ranges from 33.33% to 43.43%. Combining Table 2 and Table 6, it can be seen that almost half (39.89%) of the total gain (80.30%) from m_3 to rbe* is due to residual weights.

**Table 6: ROC AUC with and without residual weights**

| Model | | | No Weights | Weights | AUC Lift % |
|---|---|---|---|---|---|
| rbe_2: | $q = 2$ | $k = 2$ | 0.7886 | 0.7972 | 43.43 |
| rbe*: | $q = 3$ | $k = 2$ | 0.7926 | 0.8005 | 39.89 |
| rbe_3: | $q = 3$ | $k = 3$ | 0.7945 | 0.8034 | 33.33 |

## 10 RELATED WORK

RBE has binarization layers similar to binary DNNs in [2, 6, 23, 25], which focused on finding optimal ways of binarization through better gradient propagation [6], or reformulating the objective [12]. While some binary DNNS reported performance parity with the full precision counterparts, the gap was substantial in our experiments without RBE. The difference was probably in the size of the training data. With nearly two billion training samples, the side effect of binarization as a regularization process was no longer effective, and the gap had to be filled in with additional performance drivers such as RBE, which optimized the model structure.

The $k$-NN selection algorithm is related to a class of ANN algorithms such as KD-tree [11], FLANN package [22], *neighborhood graph search* [31], and *locality sensitive hashing* [8]. Unlike those algorithms, the exhaustive search is not subject to the curse of dimensionality. As compared with other brute-force $k$-NNs that have the same property, it handles billion-scale keywords, while existing methods such as [18, 30] mostly dealt with data size in the millions or smaller.

The rbeGIR system was implemented on GPUs. Based on a recent ANN called *product quantization* (PQ) [16], a billion-scale retrieval system on GPU was proposed in [34], and extended later in [17]. In order to achieve speed and memory efficiency, PQ-based approaches had to drastically sacrifice the resolution of the codebook, and rely on lossy indexing structures. In contrast, the rbeGIR system relies on a near lossless $k$-NN, and a compact representation with high and easy to control accuracy. It also does not involve extra indexing structures that may require extensive memory.

RBE is related to the efforts such as Deep Embedding Forest [36] to speed up online serving of DNNs like Deep Crossing [28]. However, the focus there was on simplifying the deep layers, rather than a compact representation of the embedding layer.

[13]The rbe* model is an exception

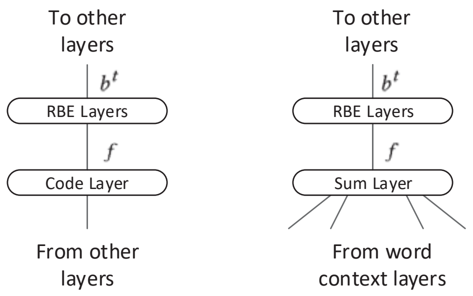

**Figure 5: The concept of RBE can be generalized to other networks such as semantic hashing (left) and word2vec (right)**

Finally, RBE is remotely related to *residual nets* [13], where the "shortcuts" in those models were constructed differently and for different purposes.

## 11 CONCLUSION AND FUTURE WORK

The RBE model proposed in this paper generates compact semantic representation that can be efficiently stored and processed on GPUs to enable billion-scale retrieval in real-time. Integrating the RBE representation with a GPU-based exhaustive $k$-NN search, the rbeGIR system is expected to set an early example for IR in the era of powerful GPUs and advanced Deep Learning algorithms.

Being able to learn the RBE representation benefits from the advance of Deep Learning, while being able to process RBE representations in real-time benefits from the advance of GPU hardware. Together, brute-force IR at billion-scale is within the reach. What is more interesting in this new era is the paradigm shift in designing IR algorithms. To tame the curse of dimensionality, the answer may lie in something more straightforward, but better utilizing the ever growing power of hardware.

To make the presentation pragmatic and intuitive, RBE is introduced in the context of CDSSM and sponsored search. We conclude this paper by claiming that RBE is not constrained by specific embedding models, and its application is broader. Part of our future work is to apply the concept of RBE on different network structures such as semantic hashing and word2vec, as illustrated in Fig. 5, where the RBE layers refer to the layers between $\mathbf{f}_i$ to $\mathbf{b}_i^t$ in Fig. 2, the *code layer* is the same as in Fig. 2 in [25], and the *sum layer* is the same as in the *CBOW* network of Fig. 1 in [20].

## 12 ACKNOWLEDGMENTS

The authors would like to thank Ray Lin and Yi Zhang from Bing Ads, and Xiaodong He from MSR for their support and discussions that benefited the development of RBE.

## APPENDIX A BUILDING AND TRAINING RBE

Figure 6 provides an implementation of the unfolded RBE model with CNTK BrainScript. Lines 01-05 define a straight-through layer using the straight-through variant as described in Sec. 4.2 . The parameters m and n are the dimensions of the output and input nodes. The *RBE* layers formulated in equations of (9), (10), and (11) are defined in the `RBEUnit` macro, from line 07 to 12. As input of `RBEUnit`, `InNode` in line 08 is the intermediate RBE embedding, and `EmbeddingNode` is the real-valued vector $\mathbf{f}_i$ as in (8). The

```
00:BrainScriptNetworkBuilder = [
01:    StraightThroughLayer(m, n) = [
02:        W = ParameterTensor{(m:n), init='glorotUniform'}
03:        b = ParameterTensor{m, initValue=0}
04:        apply(x) = StraightThrough(W*x+b)
05:    ]
06:
07:    RBEUnit(EDim, BDim, ResidualWeight, InNode, EmbeddingNode) = {
08:        ReconstructedNode = DenseLayer{EDim, activation=Tanh, bias=true}(InNode)
09:        ResidualNode = Minus(EmbeddingNode, ReconstructedNode)
10:        ResidualBinaryNode = StraightThroughLayer{BDim, EDim}(ResidualNode)
11:        RBEUnit = Plus(inBinaryNode, residualBinaryNode .* ResidualWeight)
12:    }.RBEUnit
13:
15:    numHiddenNodes = 288
14:    numBits = 64
16:    Label = Input(LabelDim)
17:    qemb = the output of the multi-layers for query as in Fig.2
18:    kemb = the output of the multi-layers for keyword as in Fig.2
19:
20:    qb0 = StraightThroughLayer{numBits, numHiddenNodes}(qemb)
21:    qb1 = RBEUnit(numHiddenNodes, numBits, CONST_HALF, qb0, qemb)
22:    qb2 = RBEUnit(numHiddenNodes, numBits, CONST_QUARTER, qb1, qemb)
23:
24:    kb0 = StraightThroughLayer{numBits, numHiddenNodes}(kemb)
25:    kb1 = RBEUnit(numHiddenNodes, numBits, CONST_HALF, kb0, kemb)
26:
27:    # Constant parameters are from CNTK CDSSM tutorial
28:    lf  = Times(CONST_PAD, Label)
29:    c   = CosDistanceWithNegativeSamples(qb2, kb1, CONST_SHIFT, CONST_NEG)
30:    s   = Slice(0, 1, c, axis=1, tag="output")
31:    ce  = CrossEntropyWithSoftmax(lf, Scale(CONST_GAMMA, c), tag="criterion")
32:]
```

**Figure 6: CNTK BrainScript for the unfolded RBE model (rbe*) with two residual vectors on the query side, and one residual vector on the keyword side**

`numHiddenNodes` (288) and `numBits` (64) are the same as in the rbe* model in Sec. 8. They are the dimensionality of the real-valued embedding and the RBE embedding, respectively. The real-valued embeddings for query and keyword are the output of the CDSSM max pooling layers, as illustrated in lines 17 and 18. Details of the CDSSM implementation can be found in the CNTK tutorial[14]. On the query side, a binary base vector is mapped from `qemb` through `StraightThroughLayer`, and two residual vectors are added to the base vector recursively through `RBEUnit`. The residual weight `CONST_HALF` at the first RBE step is set to $\frac{1}{2}$, and `CONST_QUARTER` at the second step is set to $\frac{1}{4}$. The same procedure is applied to `kemb`, but with only one `RBEUnit` transformation. The rest of the script follows the implementation of the CNTK CDSSM objective function.

## APPENDIX B PROBABILITY OF MISS

Below is a list of definitions[15] used to calculate the probability of missing relevant keywords for the $k$-NN selection algorithm introduced in Sec. 6:

- C – number of candidates
- N – number of relevant keywords, the same as in "top $N$"
- I – number of keywords per thread
- T – number of threads, and $T = C/I$
- M – number of threads with at least one relevant keyword
- L – number of missed relevant keywords

The event of missing keywords is defined under the condition of using a priority queue with length equal to 1. We start by noticing $L = N - M$, and there are $\Lambda$ combinations of distributing $N$ relevant keywords to $M$ threads[16], where $\Lambda = \binom{N-1}{M-1}$.

Suppose that $n_{i,j}$ is the number of relevant keywords in the $j^{th}$ thread of the $i^{th}$ combination. By definition, we have $n_{i,j} \geq 1$ and $\sum_{1 \leq j \leq M} n_{i,j} = N$. The number of combinations of having $n_{i,j}$ relevant keywords in the thread with a total of $I$ keywords is $\binom{I}{n_{i,j}}$[17]. Since there are $M$ independent threads, the number of combinations of all threads becomes $\prod_{j=1}^{M} \binom{I}{n_{i,j}}$. Summing up $\Lambda$

[14]https://docs.microsoft.com/en-us/cognitive-toolkit/how-do-i-train-models-in-brainscript#train-a-dssm-or-a-convolutional-dssm-model

[15]Some of the definitions are given before, but are repeated here to avoid cross referencing

[16]This is derived using the Stars and Bars method, where the stars are the relevant keywords, and the bars are the threads.

[17]The combinations of $n_{i,j} > I$ are naturally excluded because $\binom{I}{n_{i,j}} = 0$

mutually exclusive thread combinations leads to:

$$\sum_{i=0}^{\Lambda-1} \prod_{j=1}^{M} \binom{I}{n_{i,j}}.$$

Multiplying by the choices of selecting $M$ threads from the total of $T$ threads, and divided by the number of combinations of selecting $N$ relevant keywords from the entire set of $C$ candidates, the probability of having $M$ threads with at least one relevant keywords is:

$$P(M) = \frac{\binom{T}{M} \sum_{i=0}^{\Lambda-1} \prod_{j=1}^{M} \binom{I}{n_{i,j}}}{\binom{C}{N}}. \tag{16}$$

Since $L = N - M$, this is equivalent to the probability of missing $L$ relevant keywords. Table 7 summarizes the results for $N = 1000$, $C = 10^9$, and $I = 256$ based on (16).

**Table 7: Probability of missing at most $l$ keywords**

| $l$ | 0 | 1 | 2 |
|---|---|---|---|
| $P(L \leq l)$ | 88.039 | 99.256 | 99.969 |

Note that $n_{i,j}$ has to be enumerated in order to use (16). This becomes intractable when the number of missing keywords increases. However, since $P(L \leq 2)$ is already high enough, it is of no interest to go after solutions with higher $l$.

## REFERENCES

[1] Charu C Aggarwal, Alexander Hinneburg, and Daniel A Keim. 2001. On the surprising behavior of distance metrics in high dimensional spaces. In *ICDT*, Vol. 1. Springer, 420–434.

[2] Hande Alemdar, Vincent Leroy, Adrien Prost-Boucle, and Frédéric Pétrot. 2017. Ternary neural networks for resource-efficient AI applications. In *Neural Networks (IJCNN), 2017 International Joint Conference on*. IEEE, 2547–2554.

[3] Yoshua Bengio, Nicholas Léonard, and Aaron C. Courville. 2013. Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation. *CoRR* abs/1308.3432 (2013). arXiv:1308.3432 http://arxiv.org/abs/1308.3432

[4] David M Blei, Andrew Y Ng, and Michael I Jordan. 2003. Latent dirichlet allocation. *Journal of machine Learning research* 3, Jan (2003), 993–1022.

[5] Zhangjie Cao, Mingsheng Long, Jianmin Wang, and Philip S. Yu. 2017. HashNet: Deep Learning to Hash by Continuation. *CoRR* abs/1702.00758 (2017). arXiv:1702.00758 http://arxiv.org/abs/1702.00758

[6] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830* (2016).

[7] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*. ACM, 253–262.

[8] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive Hashing Scheme Based on P-stable Distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry (SCG '04)*. ACM, New York, NY, USA, 253–262. https://doi.org/10.1145/997817.997857

[9] Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. 1990. Indexing by latent semantic analysis. *Journal of the American society for information science* 41, 6 (1990), 391.

[10] Benjamin Edelman, Michael Ostrovsky, and Michael Schwarz. 2005. *Internet advertising and the generalized second price auction: Selling billions of dollars worth of keywords*. Technical Report. National Bureau of Economic Research.

[11] Jerome H Friedman, Jon Louis Bentley, and Raphael Ari Finkel. 1977. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)* 3, 3 (1977), 209–226.

[12] Abram L. Friesen and Pedro M. Domingos. 2017. Deep Learning as a Mixed Convex-Combinatorial Optimization Problem. *CoRR* abs/1710.11573 (2017). arXiv:1710.11573 http://arxiv.org/abs/1710.11573

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[14] Thomas Hofmann. 1999. Probabilistic latent semantic indexing. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 50–57.

[15] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. 2013. Learning deep structured semantic models for web search using clickthrough data. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*. ACM, 2333–2338.

[16] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2011. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33, 1 (2011), 117–128.

[17] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2017. Billion-scale similarity search with GPUs. *arXiv preprint arXiv:1702.08734* (2017).

[18] Shengren Li and Nina Amenta. 2015. Brute-force k-nearest neighbors search on the GPU. In *International Conference on Similarity Search and Applications*. Springer, 259–270.

[19] Duane G Merrill and Andrew S Grimshaw. 2010. Revisiting sorting for GPGPU stream architectures. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 545–546.

[20] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[21] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Conference on Neural Information Processing Systems (NIPS)*.

[22] Marius Muja and David G Lowe. 2014. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36, 11 (2014), 2227–2240.

[23] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*. Springer, 525–542.

[24] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.

[25] Ruslan Salakhutdinov and Geoffrey Hinton. 2007. Semantic hashing. *RBM* 500, 3 (2007), 500.

[26] Gerard Salton, Anita Wong, and Chung-Shu Yang. 1975. A vector space model for automatic indexing. *Commun. ACM* 18, 11 (1975), 613–620.

[27] Mark Sanderson and W Bruce Croft. 2012. The history of information retrieval research. *Proc. IEEE* 100, Special Centennial Issue (2012), 1444–1451.

[28] Ying Shan, T. Ryan Hoens, Jian Jiao, Haijing Wang, Dong Yu, and JC Mao. 2016. Deep Crossing: Web-Scale Modeling without Manually Crafted Combinatorial Features. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*. ACM.

[29] Yelong Shen, Xiaodong He, Jianfeng Gao, Li Deng, and Grégoire Mesnil. 2014. Learning semantic representations using convolutional neural networks for web search. In *Proceedings of the companion publication of the 23rd international conference on World wide web companion*. International World Wide Web Conferences Steering Committee, 373–374.

[30] Xiaoxin Tang, Zhiyi Huang, David Eyers, Steven Mills, and Minyi Guo. 2015. Efficient selection algorithm for fast k-nn search on gpus. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 397–406.

[31] Jingdong Wang and Shipeng Li. 2012. Query-driven Iterated Neighborhood Graph Search for Large Scale Indexing. In *Proceedings of the 20th ACM International Conference on Multimedia (MM '12)*. ACM, New York, NY, USA, 179–188. https://doi.org/10.1145/2393347.2393378

[32] Roger Weber, Hans-Jörg Schek, and Stephen Blott. 1998. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, Vol. 98. 194–205.

[33] Xing Wei and W Bruce Croft. 2006. LDA-based document models for ad-hoc retrieval. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 178–185.

[34] Patrick Wieschollek, Oliver Wang, Alexander Sorkine-Hornung, and Hendrik Lensch. 2016. Efficient large-scale approximate nearest neighbor search on the gpu. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2027–2035.

[35] Clement T Yu and Gerard Salton. 1976. Precision weightingan-effective automatic indexing method. *Journal of the ACM (JACM)* 23, 1 (1976), 76–88.

[36] Jie Zhu, Ying Shan, JC Mao, Dong Yu, Holakou Rahmanian, and Yi Zhang. 2017. Deep embedding forest: Forest-based serving with deep embedding features. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1703–1711.