Beyond Two-Tower Matching: Learning Sparse Retrievable Cross-Interactions for Recommendation

Liangcai Su*
Shenzhen International Graduate School, Tsinghua University
Shenzhen, China
sulc21@mails.tsinghua.edu.cn

Fan Yan
Huawei Noah's Ark Lab
Shenzhen, China
yanfan6@huawei.com

Jieming Zhu§
Huawei Noah's Ark Lab
Shenzhen, China
jiemingzhu@ieee.org

Xi Xiao§
Shenzhen International Graduate School, Tsinghua University
Shenzhen, China
xiaox@sz.tsinghua.edu.cn

Haoyi Duan
Zhejiang University
Hangzhou, China
howie@zju.edu.cn

Zhou Zhao
Zhejiang University
Hangzhou, China
zhaozhou@zju.edu.cn

Zhenhua Dong
Huawei Noah's Ark Lab
Shenzhen, China
dongzhenhua@huawei.com

Ruiming Tang
Huawei Noah's Ark Lab
Shenzhen, China
tangruiming@huawei.com

ABSTRACT

Two-tower models are a prevalent matching framework for recommendation, which have been widely deployed in industrial applications. The success of two-tower matching attributes to its efficiency in retrieval among a large number of items, since the item tower can be precomputed and used for fast Approximate Nearest Neighbor (ANN) search. However, it suffers two main challenges, including limited feature interaction capability and reduced accuracy in online serving. Existing approaches attempt to design novel late interactions instead of dot products, but they still fail to support complex feature interactions or lose retrieval efficiency. To address these challenges, we propose a new matching paradigm named SparCode, which supports not only sophisticated feature interactions but also efficient retrieval. Specifically, SparCode introduces an *all-to-all interaction module* to model fine-grained query-item interactions. Besides, we design a discrete code-based *sparse inverted index* jointly trained with the model to achieve effective and efficient model inference. Extensive experiments have been conducted on open benchmark datasets to demonstrate the superiority of our framework. The results show that SparCode significantly improves the accuracy of candidate item matching while retaining the same level of retrieval efficiency with two-tower models.

CCS CONCEPTS

• **Information systems** → **Recommender systems**.

* Work done during internship at Huawei Noah's Ark Lab.
§ Corresponding authors.

SIGIR '23, July 23–27, 2023, Taipei, Taiwan

ACM ISBN 978-1-4503-9408-6/23/07.
https://doi.org/10.1145/3539618.3591643

KEYWORDS

Candidate Matching, Product Quantization, Recommender System

ACM Reference Format:
Liangcai Su*, Fan Yan, Jieming Zhu§, Xi Xiao§, Haoyi Duan, Zhou Zhao, Zhenhua Dong, and Ruiming Tang. 2023. Beyond Two-Tower Matching: Learning Sparse Retrievable Cross-Interactions for Recommendation. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '23), July 23–27, 2023, Taipei, Taiwan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3539618.3591643

1 INTRODUCTION

Industrial recommender systems generally consists of candidate matching and ranking phases, where the candidate matching phase plays the role of efficiently retrieving user-preferred items out of a large pool of candidates. The matching result will directly affect the input of the ranking phase, which demonstrates significant importance to the quality of recommender systems. Due to the large size of candidate items (e.g., millions or more), matching models not only need to obtain high accuracy in recall, but also require efficient retrieval to achieve low latency.

Two-tower models [7, 22, 33, 35] are a primary paradigm for candidate matching due to their good accuracy and support for efficient top-k retrieval. As shown in Figure 1(a), a two-tower model uses dual-encoders (i.e. query and item encoders) to obtain the query[1] and item representations separately and obtains the final score by simple dot product or cosine computations. For inference, item embeddings can be pre-computed and cached, while only the user embedding is computed online. By utilizing the cached item embeddings and the support of fast Approximate Nearest Neighbor (ANN) search (e.g. Faiss [11]), two-tower matching has proven to be a good industrial practice in real-world applications [6, 33, 35]

[1]For recommender systems, user profile is taken as a query, and thus we use them exchangably.

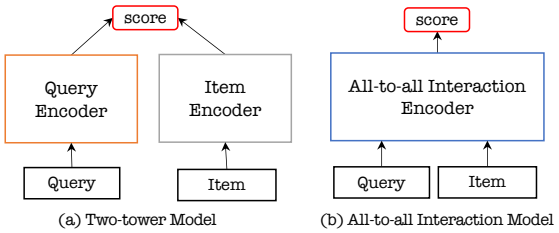


Figure 1: Two tower model and All-to-all Interaction Model.

However, the development of two tower models has the following limitations: **(a) Limitation 1: Limited feature interaction capability.** There is only one dot product interaction between the two towers, resulting in a limited ability to model fine-grained feature interactions between queries and items. Although dot product is a nice feature interaction method [27], it is not always optimal, especially in the case when rich content features are available. **(b) Limitation 2: Reduced accuracy in online serving.** In many industrial scenarios, the size of candidate item pool is so large that exhaustively comparing against all the candidate items becomes impractical due to inference time constraints. ANN search methods (e.g., HNSW [21]) have been used to build indexes and speed up retrieval, but unfortunately this may reduce recall rate since the model and the index are usually not trained end-to-end.

To address the above limitations, we propose a new matching paradigm, i.e., SparCode, for improving recommendation accuracy and retrieval efficiency. SparCode consists of two key designs: *all-to-all interaction* and *sparse inverted index*. To enhance feature interaction capability, *the all-to-all interaction module* utilizes a single expressive encoder to capture fine-grained interactions between all query features and all item features, as shown in Figure 1(b). It allows different forms of encoder structures (e.g. Self-Attention [30], MLPs, CrossNet [32]), and can learn more fine-grained and predictive feature interactions between the query and the item (i.e., early fusion) than the simple dot product in two-tower models (i.e., late fusion). Fine-grained feature interactions improve model capacity and thus result in improved accuracy, as reported in previous research [20, 25]. However, compared with the two-tower models, the nature of query-item encoding makes it hard to use the existing ANN approaches for efficient retrieval, making it impractical. To speed up model inference, a straightforward solution is to pre-compute the scores of user-item pairs and cache them. Thus, SparCode introduces the *sparse inverted index* that is widely used in sparse retrieval. However, there are two problems of building index for users: (1) The number of users is too large to be exhaustive, resulting in an unacceptably expensive precomputation. (2) Each user (index) corresponds to a large number of items, which brings great storage pressure. In respond to these problems, SparCode leverages vector quantization(VQ) as a bridge between all-to-all interaction and sparse inverted indexing, i.e. query is quantized into a series of discrete codes and corresponding code representations, where code replaces query as index and code representation is used for all-to-all interaction. Since the number of codes is controllable and much smaller than the number of queries, the problem of large number of indexes is alleviated. Further, SparCode designs a controlled sparse score function that allows each index to save only the most relevant candidates, greatly relieving storage pressure.

In summary, we present the advantages of SparCode from both model structure and inference perspectives as follows. For model structure, SparCode supports sophisticated forms of all-to-all interactions and efficient top-k retrieval of large-scale candidates by designing the linkage of VQ and sparse inverted index (for Limitation 1). SparCode introduces all-to-all interaction directly into the model structure, significantly enhancing the ability of feature interaction, rather than using all-to-all interaction models indirectly (e.g. Knowledge Distillation (KD) [3]) or using parameter-less interaction methods(e.g. ColBERT [13]). In addition, sparse inverted indexes are also supported by well-established search tools, such as ElasticSearch, ensuring usability in industrial applications.

For model inference, SparCode reduces the gap between model training and inference, since the model and index structure are trained end-to-end (for Limitation 2). In the index structure of SparCode, the index is code, which corresponds to the score of code and candidate items. The mapping between the query and code, as well as the scores, are learned during model training instead of post-training independently in ANN libraries such as Faiss. Thus, we reduce the performance drop in this stage, which is also empirically verified in our experiments (Sec. 4.4).

All in all, SparCode as an all-to-all interaction-based retrieval paradigm, achieves the goal of improving accuracy with the help of enhanced feature interactions while keeping inference efficient. Although we mainly focus on recommender system in this work, SparCode has the potential to be extended to other tasks, such as cross-modal retrieval. We conduct extensive experiments on two public datasets and show that SparCode is significantly more accurate than the two-tower model (Sec. 4.2) as well as comparably efficient to the ANN-based two-tower model (Sec. 4.4). Our contributions are summarized as follows.

- To the best of our knowledge, SparCode is the first sparse retrieval framework that supports sophisticated forms of all-to-all interactions and controllable sparse inverted indexing for recommender systems.
- SparCode converts queries to discrete codes, and thus makes pre-computed scores possible. Besides, SparCode enables efficient retrieval with a sparse inverted index structure, which has mature indexing tool support for industrial deployment.
- Our experiments on public datasets show that SparCode brings a significant improvement in accuracy, while achieving comparable efficiency to the two-tower matching.

2 RELATED WORK

2.1 Two-tower Models and Variants

Two tower matching, the combination of the two-tower model and ANN framework, is the dominant paradigm in dense retrieval, owing to achieving high accuracy and efficient top-k retrieval. Two tower matching are widely deployed in various applications [2, 6, 7, 34]. For these aforementioned partial two-tower models, BARS[39] provides experimental results and leaderboards on multiple datasets for further understanding. The feature interaction capability between query and item towers is limited by the dot product especially for scenarios with rich features. This has been verified in cross-modal retrieval tasks. For example, in text-image retrieval model VILT [14], the all-to-all interaction models performs significantly

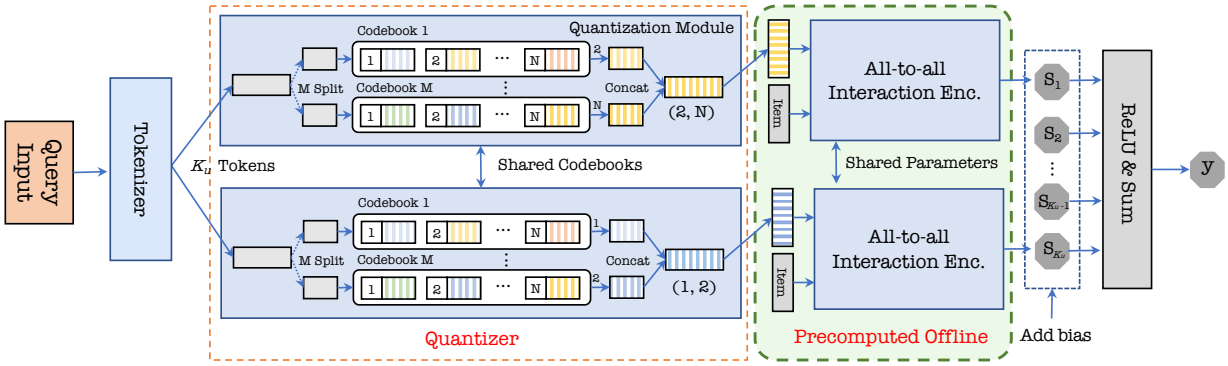


Figure 2: Overview of SparCode.

better than the standard two tower models. To sum up, enhancing feature interaction between two towers is an important direction for performance improvement.

There are two variants to overcome the above limitations: (a) using more advanced shallow interactions such as MaxSim [13], attention mechanism [9]; (b) knowledge distillation, i.e., transferring knowledge from the better interaction-based model to the two-tower model [3]. However, these variants still do not support *real* all-to-all feature interaction. In addition, some work [10, 37] attempts to mitigate the performance loss during inference, using joint training models and indexing. Unlike these methods, SparCode supports sophisticated forms of all-to-all interaction models with superior performance, and uses sparse inverted index instead of the ANN framwork.

2.2 Vector Quantization

Vector quantization refers to the discretization of vectors into codes, while product quantization (PQ) is a variant of the sharp increase in the number of codes using VQ multiple times. Traditional Product Quantization employs post-processing to convert vectors into multiple discrete codes for speeding up nearest neighbor search. In recent years, deep quantization has been applied to generating tasks successfully [23, 24, 29] with amazing results. In retrieval task, VQ is also used for multimodal unified modeling [19] or representation reconstruction [36]. In general, the code representation obtained by deep quantization is used as an alternative to the query representation to express similar information.

Different from these works, we use VQ to generate codes for the inverted indexes. The corresponding code representations of the query are used for the all-to-all interaction with the item representations.

3 OUR METHODOLOGY

3.1 Overview

In this section, we present our matching framework SparCode. An overview of SparCode is shown in Figure 2. To summarize, Tokenizer and Quantizer are designed to obtain multiple discrete representations of a query, supporting sparse inverted indexes for maintaining efficiency. All-to-all interaction-based scorer enhances the interaction between query and item for improving accuracy.

Briefly, we compare the differences between the two-tower model, the all-to-all interaction model, and our SparCode as follows.

$$\text{Two-tower model:} \quad score = E_1(q) \circ E_2(c), \tag{1}$$

$$\text{All-to-all interaction model:} \quad score = E(q, c), \tag{2}$$

$$\text{SparCode:} \quad score = \sum_k E(\mathcal{T}_k(q), c), \tag{3}$$

where q and c represent a query and item respectively; $E_1(\cdot)$ and $E_2(\cdot)$ are query and item encoder respectively, and $E(\cdot)$ refers to the all-to-all interaction encoder and obtains the score, and $\mathcal{T}(\cdot)$ refers to the Tokenizer and Quantizer, which converts q into a token embedding and looks for codes and alternative code representations.

With these equations, we highlight the design motivation of SparCode. The two tower models (i.e. Eq. 1) adopting dot product scoring is to support efficient embedding-based retrieval. The all-to-all interaction models (i.e. Eq. 2) supports arbitrarily advanced encoder for fine-grained feature interaction. Usually Eq. 2 is used for ranking in a few candidates due to its inefficient inference. SparCode (i.e. Eq. 3), based on Eq. 2, defines the overall score function as the sum of the scores between each code and candidate. That means that arbitrary queries can be replaced by codes, sharing a code's vocabulary of manageable size. Considering codes as words, the inverted indexes widely used in text retrieval [28] can be easily transferred to the recommendation task. With this definition, SparCode not only introduces a powerful all-to-all interaction but also achieves effecient retrieval similar to sparse retrieval.

3.2 Matching Framework: SparCode

3.2.1 Tokenizer. In order to be able to support sparse inverted index, inspired by sparse retrieval, we first encode the query q into multiple token embeddings. The role of the Tokenizer is to represent the query as K_u tokens. Specifically, a query in recommendation (i.e., a user) is encoded into K_u tokens, representing the multiple interests of the user.

A query may consist of images, text, sequantial information or category features. Uniformly, we represent a query $q \in \mathcal{Q}$ and a candidate item $c \in \mathcal{I}$ as a series of embeddings, i.e.,

$$H^u = [h_1^u, h_2^u, ..., h_L^u], H^u \in \mathbb{R}^{L \times D}, \tag{4}$$

$$H^v = [h_1^v, h_2^v, ..., h_P^v], H^v \in \mathbb{R}^{P \times D}, \tag{5}$$

where H^u and H^v are the L and P embeddings of a given q and c, respectively; D is the dim of each embedding in H^u and H^v.

Then, we formalize the i-th token representation as follows:

$$T_i^u = Tokenizer(H^u) \in \mathbb{R}^{D^T}, \tag{6}$$

where D^T is the embedding dim of a token depending on the tokenizer settings, usually equals to D. The specific form of the Tokenizer depends on the given task and features, e.g., if the query is a sequence of historical user clicks, the Tokenizer can be chosen from GRU [4], Self-Attention [12, 17], Capsule Network [17], etc. Similarly, we denote an item as K_c tokens $T^c \in \mathbb{R}^{K_c \times D^T}$.

3.2.2 Quantizer. For text matching tasks, even though the number of queries is very large, they share the same token table (i.e. vocabulary), which limits the number of indexes in the inverted index to the size of the token table. However, query token embeddings are dense and not shared between queries, making it impossible to build a reasonable number of indexes. Thus, in this section, Quantizer transforms query token embeddings into shared codes and their representations by discretization.

In vector quantization [29], a codebook refers to a series of numbered vectors whose numbers are called codes. The quantification is to input query and return code and the corresponding vector by looking up the codebook, resulting in an arbitrary query sharing the codebook. By utilizing VQ, we quantize token embeddings into discrete codes. We construct M codebooks $C \in \mathbb{R}^{N \times \frac{D^T}{M}}$, each of size N, i.e., holding N ordered embeddings. For T_i^u, we split T_i^u into M sub-embeddings $T_i^{u,(m)}$ and update it by:

$$\begin{aligned} \widetilde{T}_i^{u,(m)} &= Quantizer(T_i^{u,(m)}, C^{(m)}) \\ &= C_k^{(m)}, \text{where } k = \arg\min_j \|C_j^{(m)} - T_i^{u,(m)}\|_2, \end{aligned} \tag{7}$$

where $C^{(m)}$ is the m-th codebook and C_k^m is its k-th embedding; $Quantizer$ stands for looking up the most similar sub-token embedding from the given codebook, and the definition of similarity depends on the Euclidean distance between the two sub-embeddings. Thus, we get the complete updated token embedding:

$$\widetilde{T}_i^u = \text{Concat}(\widetilde{T}_i^{u,(1)}, \widetilde{T}_i^{u,(2)}, \cdots, \widetilde{T}_i^{u,(M)}). \tag{8}$$

We use the indexes of M replaced sub-embeddings to combine a corresponding discrete code. For example, in Figure 2, suppose that $M = 2$ and that the 2-nd and N-th sub-embeddings are taken from $C^{(1)}$ and $C^{(2)}$ respectively, then the discrete code is $(2, N)$.

There are two considerations for the design of codebook. Firstly, the size N of each codebook should not be set too large since an excessively large codebook affects the speed of our "$Quantizer$". Secondly, N should not be too small since we need enough model capacity to represent different queries. For the above considerations, we discuss the choice of M and N. The number of discrete codes is at most the number $N \times N \times \cdots \times N = N^M$. If M takes 1, no matter how many queries there are, there will ultimately be only N different query embeddings. In order to adequately represent the different queries, N tends to be large, but this increases the number of parameters rapidly and may reduce the speed of quantization. If M is greater than or equal to 2, it is possible to achieve a sufficient number of queries with a small number of parameters and a fast quantization speed.

Furthermore, as "arg min" is a non-differential operation, there is a non-negligible optimisation problem here. Specifically, the original token embedding T_i^u is unable to obtain the gradient from the updated token embedding $\widetilde{T}_i^u$, which ultimately results in the previous parameters (e.g., tokenizer) not being updated. We will describe the corresponding model training solution in Section 3.2.4.

3.2.3 All-to-all Interaction-based Scorer. As a parameter-free operation, the dot product of the two-tower models accomplishes both feature interaction and scoring between queries and items, which potentially limits the expressiveness of the model.

Different from the previous variants of the two-tower model, we design a parameterised, learnable scorer that supports complex interactions between queries and items, named All-to-all Interaction-based Scorer.

The proposed scorer does not restrict the specific form of feature interaction, either explicitly (inner product, FM [26], CrossNet [32], Attention [18, 30]) or implicitly (e.g. DNN). As an example, We combine inner product and MLPs to give a hybrid scoring function:

$$S_i = MLPs([sg[\widetilde{T}_i^u] \odot T_1^i; \cdots; sg[\widetilde{T}_i^u] \odot T_{K_c}^i]), i \in \{1, 2, \cdots, K\} \tag{9}$$

where $S_i \in \mathcal{R}^1$ is the matching score between a code and a candidate item c; $sg[\cdot]$ represents the stop gradient operation. Since codebooks and the rest of the model are optimized separetely, $sg[\cdot]$ is introduced to avoid affecting the parameters of codebooks. Besides, the query is represented by K_u token embeddings, the above scoring function will get K_u scores separately.

Considering a query or token is only related to a part of candidate items, we define **Sparse Score** and **Final Score** as follow:

$$\hat{y}_i = ReLU(S_i + \mathbf{b}), \tag{10}$$

$$f(q, c) = \hat{y} = \sum_{i=1}^{K} \hat{y}_i, \tag{11}$$

where $\mathbf{b}$ is a learnable bias for training.

The scoring function, especially the part corresponding to Eq. 10, is simple but critical for sparse indexing. The bias term $\mathbf{b}$ of ReLU in Eq.10 is the threshold to determine whether to set relevant score $\hat{y}_i$ as 0. If $\hat{y}_i$ equals 0, there is no need to cache $\hat{y}_i$ in advance for online serving (See Section 3.3 for details).

3.2.4 Optimization. To make the quantization process training more stable and faster, we use the exponential moving average (EMA) to update codebooks and back-propagation to update the rest of the model like [24]. In each mini-batch, the parameters in the codebooks and the remainder of the model are updated by the corresponding methods.

Model Training. The model here does not include codebooks. The matching task is the most important objective and SparCode adopts sampled softmax loss for training as follow:

$$\mathcal{L}_{Match}(q, \mathcal{I}) = \sum_{c \in \mathcal{I}_{pos}} log \frac{exp(\hat{y}_c)}{exp(\hat{y}_c) + \sum_{\hat{c} \in \mathcal{I}_{neg}} exp(\hat{y}_{\hat{c}})}, \tag{12}$$

where $\mathcal{I}_{pos}$ and $\mathcal{I}_{neg}$ represent positive samples and negative samples sampled for q respectively.

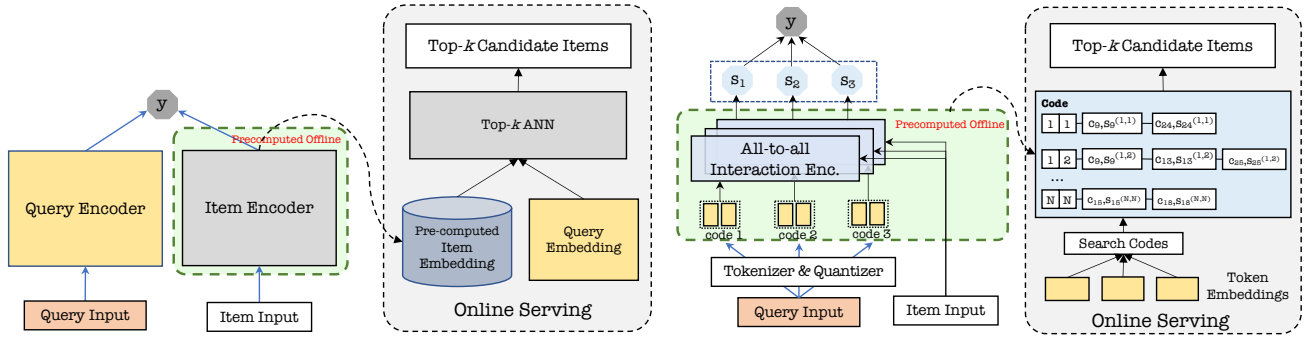


Figure 3: Comparsion of SparCode and Two-tower Matching.

As mentioned earlier, arg min is a non-differentiable operation which blocked gradient propagation, causing some parameters such as the tokenizer (or embedding table) cannot get updated. To update these parameters and make the training more stable, we introduce the commitment loss as follows:

$$\mathcal{L}_{Commit}(q) = \sum_{i=1}^{K}\sum_{m=1}^{M} \|T_i^{u,(m)} - sg[\widetilde{T}_i^{u,(m)}]\|_2^2. \tag{13}$$

Thus, the final loss is:

$$\mathcal{L}(q, \mathcal{I}) = \mathcal{L}_{Match}(q, \mathcal{I}) + \lambda \mathcal{L}_{Commit}(q), \tag{14}$$

where λ is a hyperparameter, usually set to 1 or 0.25.

Codebook Update. Following [24],we update codebooks by EMA, where the embedding in the codebook is is iteratively updated by a combination of itself and the token embeddings mapped to it. Suppose there is a series of $n_k^{(s)}$ sub-embeddings whose nearest sub-embedding is $C_k^{(m)}$ in codebook $C^{(m)}$ of the s-th mini-batch, we can update $C_k^{(m)}$ as:

$$\mathcal{N}_k^{(s)} := \mathcal{N}_k^{(s-1)} * \gamma + n_k^{(s)} * (1-\gamma),$$

$$\mathcal{V}_k^{(s)} := \mathcal{V}_k^{(s-1)} * \gamma + \sum_{j=1}^{n_k^{(s)}} T^{u,(m)}(j) * (1-\gamma),$$

$$C_k^{(m),(s)} := \frac{\mathcal{V}_k^{(s)}}{\mathcal{N}_k^{(s)}},$$

where γ is a hyperparameter that adjusts the update rate of the codebook.

3.3 Indexing and Retrieval

A query is transformed into K_u codes by Tokenizer and Quantizer, and each code has a score s_c^{code} with each item c by all-to-all interaction-based scorer. Further, the storage cost may be unacceptable if the scores of all $(code, item)$ pairs are cached. In most cases, a query is only highly relevant to part of items, which makes sparse inverted index possible since only the score of several top relevant items should be kept for each code.

Inspired by sparse retrieval and attracted by the efficiency of inverted indexes, we designed a sparse inverted index mechanism for SparCode. We use 0 as a threshold to decide whether to store the score or not for inference. We rewrite Eq. 10 to use a controllable bias instead of the bias term at training time, named "**Sparsity Control**" as follows:

$$\hat{y}_i = ReLU(S_i + \widetilde{\mathbf{b}}). \tag{15}$$

Depending on the latency requirements and memory constraints, we adjust $\widetilde{\mathbf{b}}$ in Eq. 15 to determine the sparsity of indexes.

Figure 3(b) illustrates SparCode's online service process. The right and blue part shows how the scores are cached, i.e. each code is sparsely cached with the scores of candidate items.It is important to note that these scores are pre-computed.

When serving online, the token embeddings are first obtained by tokenizer, and the code is looked up by codebooks. Then, the corresponding item score and filtered candidate itemset are loaded from the cache. For example, in Figure 3(a), if the codes are (1,1) and (1,2), we read the score set $\{s_9^{(1,1)}, s_{24}^{(1,1)}, s_9^{(1,2)}, s_{13}^{(1,2)}, s_{25}^{(1,2)}\}$ and get the merged itemset $\{c_9, c_{24}, c_{13}, c_{25}\}$. Finally, Eq. 11 is used to obtain the final scores and the items with the highest scores are taken as the recommended results.

3.4 Comparision with Two-tower Models

Figure 5 shows the comparison between SparCode and Two-tower Matching. We discuss these differences below from two perspectives: modeling stage and inference stage.

Modeling Stage. SparCode supports advanced all-to-all interaction encoders, providing better feature interaction capabilities than two-tower matching, which only supports parameter-free interactions like dot product. Additionally, SparCode's quantizer allows for shared sub-embeddings in different query representations, while each query's representation is independent in the two-tower model.

Inference Stage. As shown in the right part of Figure 3.4(a) and (b), we summarize the differences: (1) the cached content. Instead of caching the item embedding, SparCode selectively caches the pre-computed (code, item) scores. (2) the cache structure. SparCode cache is sparse hashing tables for a sparse code-based inverted index, while the two-tower matching caches matrix of item embeddings or other index structures depending on the ANN settings.

Table 1: Performance comparisons on Deezer. The second-best results are in bold and the best results are underlined.

Methods	Deezer (TT-SVD)			Deezer (UT-ALS)		
	Precision@50	Recall@50	NDCG@50	Precision@50	Recall@50	NDCG@50
Popularity	8.92%	3.01%	9.72%	8.92%	3.01%	9.72%
DropoutNet	10.04%	3.75%	10.46%	16.30%	5.77%	17.62%
MeLU	15.00%	5.12%	16.79%	13.92%	4.71%	15.49%
DeezerNet	9.58%	3.53%	9.77%	15.80%	6.63%	20.22%
TTM w/ DNN (DSSM)	10.40%	3.79%	10.54%	17.42%	6.09 %	18.60%
SparCode w/ DNN(ours)	17.80%	6.53%	19.37%	23.45%	8.41%	25.20%
TTM w/ SA	11.22%	4.11%	11.33%	19.39%	6.93%	21.15%
SparCode w/ SA(ours)	**19.08%**	**6.85%**	**20.72%**	**24.13%**	**8.69%**	**25.99%**
All-to-All SA	19.17%	6.96%	20.75%	26.35%	9.65%	28.23%

Table 2: Performance comparisons on Movielens-10M. The second-best results are in bold and the best results are underlined.

Model Type	Model	Precision@50	NDCG@50	Recall@50
TTM	DSSM	8.124%	17.690%	26.534%
	GRU4Rec	8.356%	18.473%	28.261%
	SASRec	8.754%	19.202%	29.828%
SparCode	DNN	8.393%	18.295%	27.184%
	GRU4Rec	8.888%	20.050%	29.492%
	SASRec	**9.020%**	**20.480%**	**30.142%**
All-to-All	SASRec	9.378%	21.950%	31.946%

4 EXPERIMENT

4.1 Experimental Setup

4.1.1 Datasets. We conduct experiments on two datasets Deezer[2] and Movielens-10M [3] for candidate item matching.

Deezer dataset contains 100,000 fully anonymous users and 50,000 music tracks. We follow [1], using 70,000 active warm users for training and the remaining 20,000 and 10,000 cold users for validation and testing. The dataset provides two pretrained embedding types named "TT-SVD" and "UT-ALS" to represent users, user features and songs.

Movielens-10M dataset is a classic recommendation dataset, consisting of 71,567 users and 65,133 items and over 10,000,000 user interactions. It contains abundant sequential user interactions and is widely evaluated for sequential item retrieval. We split all users into training, validation and test sets by the ratio of 8:1:1.

4.1.2 Competitors. To evaluate the results, our proposed method is compared with several powerful baselines in recent literature. For simplicity, we abbreviate the "Two-Tower Model" as "TTM" for all the tables and figures.

For Deezer, we adopt a popularity-based method called "Popularity" and three models specialized in cold-start recommendations include DropoutNet [31], MeLU [16], and DeezerNet [1]. In addition, we compare SparCode with two two-tower models: DSSM [7] and SA [12], in which DSSM is a famous two-tower model based-on DNN while SA utilizes Self-Attention module as encoder for feature encoding. We denote DSSM as "TTM w/ DNN" and SA as "TTM w/ SA". For SparCode, we choose DNN and Self-Attention modules as its tokenizer, denoted as "SparCode w/ DNN" and "SparCode w/ SA" respectively.

For Movielens-10M, we choose three most commonly used baselines including DSSM [7], GRU4Rec [4] and SASRec [12]. The sequential modeling module of these methods are treated as the user tower. Respectively, we choose DNN, GRU, and Self-Attention modules as tokenizer for SparCode to evaluate on this dataset. We set the length of the behaviour sequence to 20 for all methods.

For both recommendation datasets, we choose vanilla All-to-All Self-Attention model as a strong baseline, denoted as "All-to-All SA". Without consideration of inference speed, the user features/histories and item features are fed into a mulit-layer SA-based model simultaneously for better feature interaction. The performance gap between SparCode and All-to-All SA reveals how much effectiveness sacrificed by SparCode for better efficiency.

4.1.3 Implementation Details. We choose Adam [15] as optimizer with the learning rate of 0.001. We set the L_2 regularization factor for the embedding table as $1e^{-6}$ and the dropout rate as 0.1. Uniformly, the hidden units of DNNs is [256,256,256], and Self-Attention layers is 3. For SparCode, we search for the best query token number K_u from $\{1, 2, 4, 6, 8, 10\}$. The different K_u query token embeddings are obtained from SENet [5] or linear layers based on the output f the Tokenizer. The different K_c item token embeddings are encoded from different linear layers. The hyperparameters of the codebooks are of vital importance, including codebook number M, codebook capacity N, and the size of each embedding. We do grid search of M from $\{1, 2\}$ and N from $\{64, 128, 256, 512, 1024\}$. The item embedding size is set as 128 and 256 for "TT-SVD" and "UT-ALS" respectively on Deezer, and 64 on Movielens-10M. λ in Eq. 14 is choosen from $\{0.25, 1\}$. To ensure the accuracy of the results, we repeated the experiment five times for each experiment with different random seeds. Our partial implementation refers to FuxiCTR [40] for guidance. The source code will be released in this repository.[4].

4.2 Performance Analysis

A comparison of the performance of SparCode and baselines is shown in the Table 1 and 2. As shown in Table 1 and 2, SparCode performs well on both datasets, far better than two-tower models and even close to the "oracle", the All-to-All SA model. Specifically,

[2]https://zenodo.org/record/5121674#.YwpGvC-KFAa
[3]https://grouplens.org/datasets/movielens/
[4]https://gitee.com/mindspore/models/tree/master/research/recommend

Table 3: Effect of Interaction Type on Deezer with "TT-SVD" Embeddings. R: Recall, N: NDCG.

Model	Interaction Type	R@20	R@50	N@20	N@50
TTM w/ DNN	Dot Product	1.58%	3.79%	10.92%	10.54%
SparCode w/ DNN	Dot Product(K_c=1)	1.40%	3.39%	9.61%	9.39%
	Dot Product(K_c=4)	1.45%	3.51%	9.93%	9.69%
	MaxSim	2.46%	5.44%	17.22%	15.68%
	CrossNet	2.42%	5.30%	17.33%	15.60%
	DNN	3.12%	6.55%	21.45%	18.27%
	InnerPDNN	3.18%	6.53%	23.37%	19.96%
TTM w/ SA	Dot Product	1.70%	4.11%	11.63%	11.33%
SparCode w/ SA	Dot Product(K_c=1)	1.55%	3.72%	10.71%	10.34%
	Dot Product(K_c=4)	1.57%	3.75%	10.77%	10.43%
	MaxSim	2.39%	5.42%	16.67%	15.47%
	CrossNet	2.49%	5.55%	17.55%	16.11%
	DNN	3.18%	6.63%	23.20%	20.09%
	InnerPDNN	**3.32%**	**6.85%**	**24.11%**	**20.72%**

SparCode w/ DNN achieves 71.15% relative improvements over DSSM for Precision@50 on Deezer. Compared with other state-of-the-art retrieval models such as MeLU, DropoutNet, and DeezerNet, SparCode also yields significant effectiveness gain. For Movielens-10M, SparCode outperforms all tow-tower-models with the same sequential encoder on various metrics. For SparCode, we found that self-attention module is more suitable to be used as the tokenizer to encode interacted feature fields or sequential user behaivors.

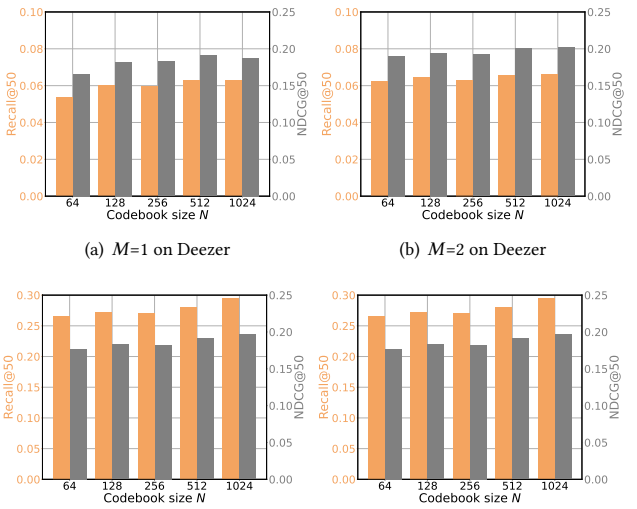
Since SA-based module performs best on both datasets, the All-to-All SA model acts like an "Oracle" which shows the performance upper bound of SparCode. Specifically, we remove the PQ and sparse indexing modules in SparCode with SA for both datasets. The results are shown as "All-to-All SA" for Deezer and "All-to-All SASRec" for Movielens-10M. For TT-SVD Embedding based Deezer, the performance gap between SparCode with SA and All-to-All SA is only 0.4% for precision@50. For UT-ALS Embedding based Deezer and Movielens-10M, the performance gap is slightly larger but is also acceptable. For instance, the relative gap on Recall@50 between SparCode w/ SA and All-to-All SA is 11%. This indicates that the use of PQ and sparse index will not significantly degrade performance. SparCode achieves a nice trade-off between effectiveness and efficiency.

4.3 Abalation Study

4.3.1 Interaction Type. We explore the impact of different types of feature interactions on SparCode. Considering that we model query and item as multiple token representations, we give a generic form for different feature interactions by rewriting Eq. 9 as follows.

- **Dot Product**: $S_i = \sum_j^p \langle sg[\widetilde{T}_i^u], T_{K_c}^i \rangle$.
- **MaxSim**(ColBERT [13]): $S_i = Max_{i \in 1,2,\odot,p}(\langle sg[\widetilde{T}_i^u], T_{K_c}^i \rangle)$.
- **CrossNet** [32]: $S_i = CrossNet([sg[\widetilde{T}_i^u]; T_1^i; \cdots; T_{K_c}^i])$.
- **DNN**: $S_i = MLPs([sg[\widetilde{T}_i^u]; T_1^i; T_2^i; \cdots; T_{K_c}^i])$.
- **InnerPDNN**: $S_i = MLPs([sg[\widetilde{T}_i^u] \odot T_1^i; \cdots; sg[\widetilde{T}_i^u] \odot T_{K_c}^i])$ (i.e. Eq. 9).

Among these interaction ways, Dot Product and MaxSim (from ColBERT [13]) are parameterless methods while the other three have optimizable parameters. The number of CrossNet layers is set as 3. InnerPDNN is the interaction way shown in Eq. 9, which is the combination of Inner Product and DNN. Table 3 shows the experimental results of various interaction types on Deezer with "TT-SVD" Embeddings. "TTM w/ DNN" is the same as DSSM.



(a) M=1 on Deezer (b) M=2 on Deezer

(c) M=1 on Movielens-10M (d) M=2 on Movielens-10M

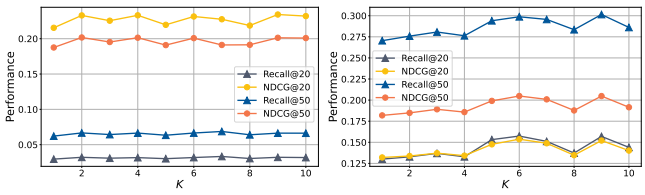
Figure 4: The number and capacity of codebook(s).

According to Table 3, complex interaction approaches (e.g., DNN, MaxSim) achieve significant performance gains compared with the simple dot production. Specifically, for DNN-based models, SparCode with MaxSim improves from 3.39% to 5.44% for Recall@50 and from 9.39% to 15.68% for NDCG@50 compared with SparCode with Dot Product. Similar phenomenon can be observed from the SA-based models, with a 45.69% and 49.61% relative improvement in Recall@50 and NDCG@50 respectively. The above results indicate that the dot product may not be sufficient to model the interaction between query and item. The DNN interaction performs significantly better than MaxSim and CrossNet. For SparCode with SA, the DNN interaction improves the Recall@50 metric from MaxSim's 5.42% and CrossNet's 5.55% to 6.63%, which is a remarkable relative improvement. Since Deezer dataset is rich in content semantics but not restricted to sparse ID features, the cross feature network may not as good as DNN to model such implict query-item interaction.

Another interesting observation is the performance of SparCode with dot product interaction is always slightly lower than that of two-tower models, suggesting that the replaced vectors from codebooks compromise some performance for efficiency. We leave how to reduce the performance gap as a future research topic.

4.3.2 Codebook Structure. Tuning codebooks is a key step in PQ. We explored the impact of hyperparameters related to the number and capacity of codebooks. We chose SparCode w/ DNN on Deezer and SparCode w/ SA on Movielens-10M for experiments and select Recall@50 and NDCG@50 as evaluation metrics. According to the results in Figure 4, we have the following observations.

Firstly, increasing the capacity N of a single codebook is helpful for better performance. On Deezer, as the codebook size increases from 64 to 1024, both Recall and NDCG are on the rise. When there is



(a) Deezer (b) Movielens-10M

Figure 5: Effect of K_u.

only one codebook(M=1), Recall@50 and NDCG@50 get improved by 16.92% and 13.29% when N increases from 64 to 1024. When M is 2, the improvement is 6.53% and 5.84%. This is because the larger M brings more code words combinations, which enhanced the representation of PQ thus the relative performance boost is not so significant. To conclude, it is recommended to increase the capacity of the codebook size for better performance, especially when there is only one codebook.

Secondly, the number of codebooks M has tremendous effects of final performance. As shown in Figure 4(a)(b), comparing M=1, N=1024 with M=2, N=64, the latter one is already more effective than the former. We believe the shared information between codebooks contributes to the increase. When M>1, different queries may share the same codewords in a specific codebook. Thus the learned information are shared in this codeword. We believe such representation sharing potentially prevents overfitting. However, it is worth noting that increasing M brings cost on computing. Choosing a proper M is also a trade-off between efficiency and effectiveness.

4.3.3 Effect of K_u. The tokenizer of SparCode encode query and item into K token vectors for later interaction. We utilize learnable linear layers or SENet [8] to transform various inputs into fixed tokens. We investigates the effects of different hyperparameter K_u. The results is shown in Fig 5. For sequential recommendation dataset like MovieLens-10M, the increase of K brings performance on various metrics, which indicates various encoded tokens represented multiple user interests, and is beneficial for better interests matching. For non-sequential recommendation dataset such as Deezer, such improvements is not very clear. Thus we can set a small K_u (e.g. $K_u = 2$) for better efficiency.

4.4 Sparsity, Performance and Speed

Sparse Indexing Mechanism play a key role in SparCode deployment. In this section, we explore the impact of cache sparsity on model performance and efficiency. To be more intuitive, we provide two evaluation metrics for sparsity, "*Sparsity*" and "*Average Items*". Specifically, "*Sparsity*" represents the ratio of **not**-cached code-item scores to the total scores. "*Average items*" represents the average number of cache scores per code. These two metrics are formulated as follows:

$$Sparsity = 1 - \frac{\#(s_c^{(m,n)} > 0)}{M^N \times |\mathcal{I}|},$$

$$Average\ Items = \frac{\#(s_c^{(m,n)} > 0)}{M^N},$$

where $m \in \{1, 2, \cdots, M\}, n \in \{1, 2, \cdots, N\}$ and "#" means "the number of".

Table 4: Performance comparison of SparCode and TDM

Model	Interaction Type	R@5	R@10	R@20	R@50
TDM-DNN	Dot Product	0.40%	0.79%	1.53%	3.71%
TDM-DNN	MLPs	0.64%	1.21%	2.31%	5.09%
SparCode-DNN	MLPs	**0.94%**	**1.75%**	**3.12%**	**6.55%**

Effect of Sparsity on Performance. We control sparsity by adjusting the bias $\widetilde{\mathbf{b}}$ in Eq. 15. The experimental results of the association of sparsity and performance are shown in Figure 6(a) and Figure 6(b), which is surprising. For Deezer, given M=2 and N=256, the setting with 99% sparsity achieves comparable performance with 0% sparsity on Recall and NDCG. This result also validates our observation that each code is highly correlated with only a small number of candidates. For memory-limited scenarios, the compressed model with a 99.947% sparsity can still exceed the performance of two-tower models (refer to Table 1). For Movielens-10M, such conclusion still holds, with only 1% scores cached to support a significant accuracy improvement. The above experimental results indicate SparCode is quite capable of online deployment even under extreme conditions.

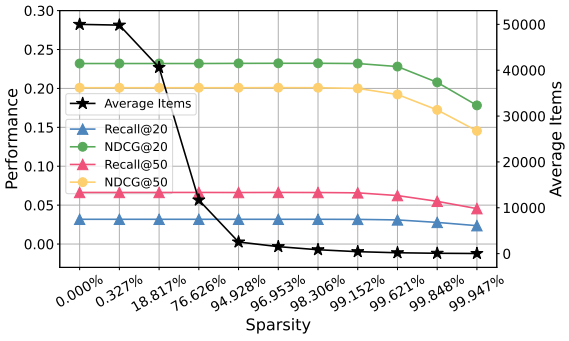
Effect of Sparsity on Inference Speed. With performance guaranteed, another concern is the inference speed. Here, we are mainly considering the time taken to process each retrieval request, excluding pre-calculation or pre-loading time. We simulate processing a real request in real scenarios. Given the resource constraints, query tower inference runs on the GPU, while the process of retrieving a top-k ANN runs on the CPU. In order to respond requests quickly, only one or a small number of requests are processed per inference. We set the batch size as 1 for comparison.

Figure 6(c) compares the inference speed between SparCode and TTMs. As shown in Figure 6(c), SparCode has similar query time with TTMs with FlatIP index, but yields significantly better performance. For two tower models, we utilize PQ methods such as IVFPQ index to get lower query time, with slight performance loss for comparison. It is worth noting that SparCode is implemented entirely in python under our experiment settings. While there is a bunch of open-source search engines which supports inverted index searching with much faster speed, SparCode can utilize these engines to achieve better query time, similar to PQ-enhanced TTM, with little performance loss.

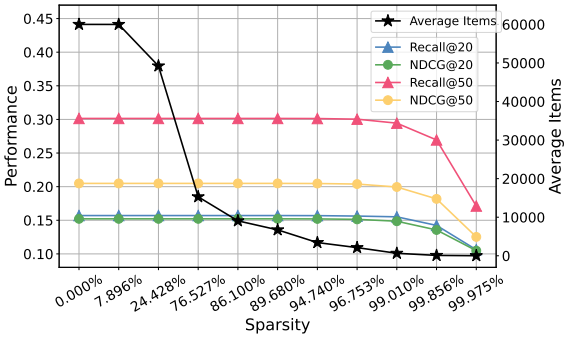
Figure 6(d) shows the effects of different sparsity on query time. With the sparsity increases, the query time decreases respectively. The evaluation performance also decreases as we have discussed.

4.5 Comparsion with TDM.

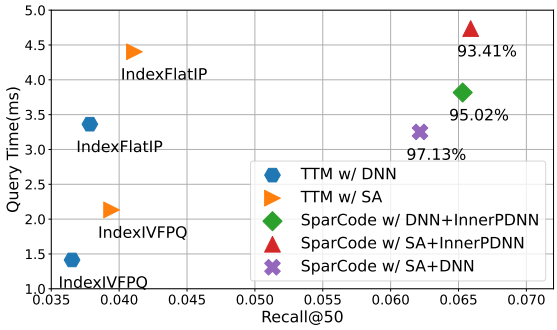
TDM [38] is a representative tree-based matching model which, like SparCode, supports advanced interaction between user and item. In this section, we compare SparCode and TDM in terms of model design and performance. We trained two YoutubeDNN models with interactions of dot product and MLPs on Deezer (TT-SVD), respectively, and trained and updated the index of the tree structure based on the trained YoutubeDNN models following the guidance of TDM. The results of SparCode and TDM experiments are shown in the Table 4, where SparCode-DNN denotes SparCode with YoutubeDNN with the same parameters as TDM.



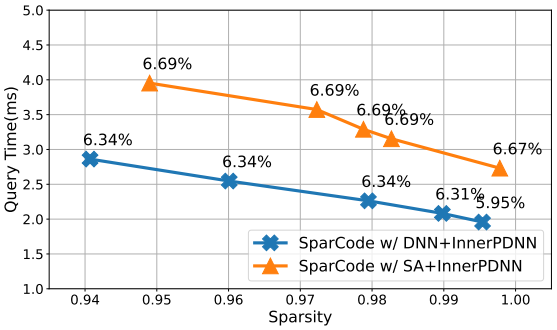
(a) Effects of sparsity on performance for Deezer



(b) Effects of sparsity on performance for Movielens-10M



(c) Comparison of query time of SparCode and TTMs. The statics annotation denotes sparsity.



(d) Comparison of query time under different Sparsity. The statics above the points denotes Recall@50.

Figure 6: Effects of sparsity on performance and query time.

According to Table 4, we have the following observations: (a) The TDM-DNN with MLPs is better than that with Dot Product, where R@20 and R@50 improved from 1.53% to 2.31% and from 3.71% to 5.09%, respectively. This improvement mainly comes from the interaction method, which again validates the limited feature interaction capability of dot product. (b) Compared to TDM-DNN with MLPs, SparCode-DNN has an average relative improvement of 35.07% in Recall metric, which reflects SparCode's accuracy advantage in model architecture.

Despite their ability to support advanced interaction methods, SparCode and TDM have many differences. First, SparCode employs a sparse inverted index with $O(1)$ inference complexity, while TDM uses a tree structure index with $O(Log(N))$ inference complexity. That is, SparCode only needs to perform a top-k retrieval once, while TDM needs to perform $O(Log(N))$ times. Secondly, SparCode is trained end-to-end, while TDM performs both model and index training. In addition, SparCode is easily controllable for the size of inverted index structures, whereas TDM usually needs to keep full binary tree indexes.

5 CONCLUSION

In this paper, we summarize two limitations of the two-tower model: limited feature interaction capability and reduced accuracy in online serving. To address the two limitations, we proposed a new matching paradigm SparCode for improving both recommendation accuracy and inference efficiency. By linking vector quantization and sparse inverted indexing, SparCode introduces an all-to-all interaction module to achieve fine-grained interaction between user and item features and is able to maintain efficient retrieval with $O(1)$ complexity. In addition, we further design the sparse fraction function to control the size of the index structure, so as to reduce the storage pressure. Extensive experimental results on two public datasets show that SparCode has far superior performance and comparable efficiency to the two-tower matching. In the future, we will further explore the application of SparCode to other tasks.

ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science Foundation of China (61972219,62276154), the Research and Development Program of Shenzhen (JCYJ20190813174403598, JCYJ20210324120012033), the Overseas Research Cooperation Fund of Tsinghua Shenzhen International Graduate School (HW2021013), Aminer ShenZhen Scientific Super Brain. We gratefully acknowledge the support of Mindspore[5], which is a new deep learning computing framework.

[5]https://www.mindspore.cn

REFERENCES

[1] Léa Briand, Guillaume Salha-Galvan, Walid Bendada, Mathieu Morlon, and Viet-Anh Tran. 2021. A Semi-Personalized System for User Cold Start Recommendation on Music Streaming Apps. *CoRR* abs/2106.03819 (2021).

[2] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. In *RecSys*. ACM, 191–198.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[4] Balázs Hidasi, Alexandros Karatzoglou, Linas Baltrunas, and Domonkos Tikk. 2016. Session-based Recommendations with Recurrent Neural Networks. In *ICLR (Poster)*.

[5] Jie Hu, Li Shen, and Gang Sun. 2018. Squeeze-and-Excitation Networks. In *CVPR*. Computer Vision Foundation / IEEE Computer Society, 7132–7141.

[6] Jui-Ting Huang, Ashish Sharma, Shuying Sun, Li Xia, David Zhang, Philip Pronin, Janani Padmanabhan, Giuseppe Ottaviano, and Linjun Yang. 2020. Embedding-based Retrieval in Facebook Search. In *KDD*. ACM, 2553–2561.

[7] 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 *Conference on Information and Knowledge Management*.

[8] Tongwen Huang, Zhiqi Zhang, and Junlin Zhang. 2019. FiBiNET: combining feature importance and bilinear feature interaction for click-through rate prediction. In *Proceedings of the 13th ACM Conference on Recommender Systems, RecSys 2019, Copenhagen, Denmark, September 16-20, 2019*, Toine Bogers, Alan Said, Peter Brusilovsky, and Domonkos Tikk (Eds.). ACM, 169–177. https://doi.org/10.1145/3298689.3347043

[9] Samuel Humeau, Kurt Shuster, Marie-Anne Lachaux, and Jason Weston. 2020. Poly-encoders: Architectures and Pre-training Strategies for Fast and Accurate Multi-sentence Scoring. In *ICLR*. OpenReview.net.

[10] Yunjiang Jiang, Han Zhang, Yiming Qiu, Yun Xiao, Bo Long, and Wen-Yun Yang. 2022. Givens Coordinate Descent Methods for Rotation Matrix Learning in Trainable Embedding Indexes. *CoRR* abs/2203.05082 (2022).

[11] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Trans. Big Data* 7, 3 (2021), 535–547.

[12] Wang-Cheng Kang and Julian J. McAuley. 2018. Self-Attentive Sequential Recommendation. In *ICDM*. IEEE Computer Society, 197–206.

[13] Omar Khattab and Matei Zaharia. 2020. ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT. In *SIGIR*. ACM, 39–48.

[14] Wonjae Kim, Bokyung Son, and Ildoo Kim. 2021. Vilt: Vision-and-language transformer without convolution or region supervision. In *International Conference on Machine Learning*. PMLR, 5583–5594.

[15] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1412.6980

[16] Hoyeop Lee, Jinbae Im, Seongwon Jang, Hyunsouk Cho, and Sehee Chung. 2019. MeLU: Meta-Learned User Preference Estimator for Cold-Start Recommendation. In *KDD*. ACM, 1073–1082.

[17] Chao Li, Zhiyuan Liu, Mengmeng Wu, Yuchi Xu, Huan Zhao, Pipei Huang, Guoliang Kang, Qiwei Chen, Wei Li, and Dik Lun Lee. 2019. Multi-Interest Network with Dynamic Routing for Recommendation at Tmall. In *CIKM*. ACM, 2615–2623.

[18] Yi Li, Jieming Zhu, Weiwen Liu, Liangcai Su, Guohao Cai, Qi Zhang, Ruiming Tang, Xi Xiao, and Xiuqiang He. 2022. PEAR: Personalized Re-ranking with Contextualized Transformer for Recommendation. In *Companion Proceedings of the Web Conference 2022*. 62–66.

[19] Alexander H. Liu, SouYoung Jin, Cheng-I Lai, Andrew Rouditchenko, Aude Oliva, and James R. Glass. 2022. Cross-Modal Discrete Representation Learning. In *ACL (1)*. Association for Computational Linguistics, 3013–3035.

[20] Fangyu Liu, Yunlong Jiao, Jordan Massiah, Emine Yilmaz, and Serhii Havrylov. 2022. Trans-Encoder: Unsupervised sentence-pair modelling through self- and mutual-distillations. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

[21] Yury A. Malkov and Dmitry A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (2020), 824–836.

[22] Kelong Mao, Jieming Zhu, Jinpeng Wang, Quanyu Dai, Zhenhua Dong, Xi Xiao, and Xiuqiang He. 2021. SimpleX: A Simple and Strong Baseline for Collaborative Filtering. In *Proceedings of the 30th ACM International Conference on Information Knowledge Management* (Virtual Event, Queensland, Australia) *(CIKM '21)*. Association for Computing Machinery, New York, NY, USA, 1243–1252.

[23] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. 2021. Zero-shot text-to-image generation. In *International Conference on Machine Learning*. PMLR, 8821–8831.

[24] Ali Razavi, Aäron van den Oord, and Oriol Vinyals. 2019. Generating Diverse High-Fidelity Images with VQ-VAE-2. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 14837–14847.

[25] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, 3980–3990. https://doi.org/10.18653/v1/D19-1410

[26] Steffen Rendle. 2010. Factorization Machines. In *ICDM 2010, The 10th IEEE International Conference on Data Mining, Sydney, Australia, 14-17 December 2010*, Geoffrey I. Webb, Bing Liu, Chengqi Zhang, Dimitrios Gunopulos, and Xindong Wu (Eds.). IEEE Computer Society, 995–1000. https://doi.org/10.1109/ICDM.2010.127

[27] Steffen Rendle, Walid Krichene, Li Zhang, and John R. Anderson. 2020. Neural Collaborative Filtering vs. Matrix Factorization Revisited. In *RecSys 2020: Fourteenth ACM Conference on Recommender Systems, Virtual Event, Brazil, September 22-26, 2020*. ACM, 240–248.

[28] Stephen E. Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Found. Trends Inf. Retr.* 3, 4 (2009), 333–389.

[29] Aäron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. 2017. Neural Discrete Representation Learning. In *NIPS*. 6306–6315.

[30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc.

[31] Maksims Volkovs, Guang Wei Yu, and Tomi Poutanen. 2017. DropoutNet: Addressing Cold Start in Recommender Systems. In *NIPS*. 4957–4966.

[32] Ruoxi Wang, Rakesh Shivanna, Derek Zhiyuan Cheng, Sagar Jain, Dong Lin, Lichan Hong, and Ed H. Chi. 2021. DCN V2: Improved Deep & Cross Network and Practical Lessons for Web-scale Learning to Rank Systems. In *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*, Jure Leskovec, Marko Grobelnik, Marc Najork, Jie Tang, and Leila Zia (Eds.). ACM / IW3C2, 1785–1797. https://doi.org/10.1145/3442381.3450078

[33] Ji Yang, Xinyang Yi, Derek Zhiyuan Cheng, Lichan Hong, Yang Li, Simon Xiaoming Wang, Taibai Xu, and Ed H. Chi. 2020. Mixed Negative Sampling for Learning Two-tower Neural Networks in Recommendations. In *Companion of The 2020 Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*. 441–447.

[34] Xinyang Yi, Ji Yang, Lichan Hong, Derek Zhiyuan Cheng, Lukasz Heldt, Aditee Kumthekar, Zhe Zhao, Li Wei, and Ed H. Chi. 2019. Sampling-bias-corrected neural modeling for large corpus item recommendations. In *Proceedings of the 13th ACM Conference on Recommender Systems, RecSys 2019, Copenhagen, Denmark, September 16-20, 2019*, Toine Bogers, Alan Said, Peter Brusilovsky, and Domonkos Tikk (Eds.). ACM, 269–277. https://doi.org/10.1145/3298689.3346996

[35] Yantao Yu, Weipeng Wang, Zhoutian Feng, and Daiyue Xue. 2021. A dual augmented two-tower model for online large-scale recommendation. *DLP-KDD* (2021).

[36] Ziyun Zeng, Jinpeng Wang, Bin Chen, Tao Dai, and Shu-Tao Xia. 2021. Pyramid Hybrid Pooling Quantization for Efficient Fine-Grained Image Retrieval. *CoRR* abs/2109.05206 (2021).

[37] Han Zhang, Hongwei Shen, Yiming Qiu, Yunjiang Jiang, Songlin Wang, Sulong Xu, Yun Xiao, Bo Long, and Wen-Yun Yang. 2021. Joint Learning of Deep Retrieval Model and Product Quantization based Embedding Index. In *SIGIR*. ACM, 1718–1722.

[38] Han Zhu, Xiang Li, Pengye Zhang, Guozheng Li, Jie He, Han Li, and Kun Gai. 2018. Learning Tree-based Deep Model for Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, Yike Guo and Faisal Farooq (Eds.). ACM, 1079–1088. https://doi.org/10.1145/3219819.3219826

[39] Jieming Zhu, Quanyu Dai, Liangcai Su, Rong Ma, Jinyang Liu, Guohao Cai, Xi Xiao, and Rui Zhang. 2022. BARS: Towards Open Benchmarking for Recommender Systems. In *SIGIR '22: Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*.

[40] Jieming Zhu, Jinyang Liu, Shuai Yang, Qi Zhang, and Xiuqiang He. 2021. Open Benchmarking for Click-Through Rate Prediction. In *Proceedings of the 30th ACM International Conference on Information Knowledge Management* (Virtual Event, Queensland, Australia) *(CIKM '21)*. Association for Computing Machinery, New York, NY, USA, 2759–2769.