
# Experimental Analysis of Large-scale Learnable Vector Storage Compression

Hailin Zhang*
Peking University
z.hl@pku.edu.cn

Penghao Zhao*
Peking University
penghao.zhao@stu.pku.edu.cn

Xupeng Miao
Carnegie Mellon University
xupeng@cmu.edu

Yingxia Shao
Beijing University of Posts and Telecommunications
shaoyx@bupt.edu.cn

Zirui Liu*
Peking University
zirui.liu@pku.edu.cn

Tong Yang*
Peking University
yangtongemail@gmail.com

Bin Cui*†
Peking University
bin.cui@pku.edu.cn

## ABSTRACT

Learnable embedding vector is one of the most important applications in machine learning, and is widely used in various database-related domains. However, the high dimensionality of sparse data in recommendation tasks and the huge volume of corpus in retrieval-related tasks lead to a large memory consumption of the embedding table, which poses a great challenge to the training and deployment of models. Recent research has proposed various methods to compress the embeddings at the cost of a slight decrease in model quality or the introduction of other overheads. Nevertheless, the relative performance of these methods remains unclear. Existing experimental comparisons only cover a subset of these methods and focus on limited metrics. In this paper, we perform a comprehensive comparative analysis and experimental evaluation of embedding compression. We introduce a new taxonomy that categorizes these techniques based on their characteristics and methodologies, and further develop a modular benchmarking framework that integrates 14 representative methods. Under a uniform test environment, our benchmark fairly evaluates each approach, presents their strengths and weaknesses under different memory budgets, and recommends the best method based on the use case. In addition to providing useful guidelines, our study also uncovers the limitations of current methods and suggests potential directions for future research.

**PVLDB Reference Format:**
Hailin Zhang, Penghao Zhao, Xupeng Miao, Yingxia Shao, Zirui Liu, Tong Yang, and Bin Cui. Experimental Analysis of Large-scale Learnable Vector Storage Compression. PVLDB, 17(4): 808 - 822, 2023.
doi:10.14778/3636218.3636234

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/HugoZHL/Hetu/tree/embedmem/tools/EmbeddingMemoryCompression.

*School of Computer Science & Key Lab of High Confidence Software Technologies, Peking University
†Institute of Computational Social Science, Peking University (Qingdao)

Proceedings of the VLDB Endowment, Vol. 17, No. 4 ISSN 2150-8097.
doi:10.14778/3636218.3636234

## 1 INTRODUCTION

In recent years, embedding techniques have been widely used in database-related research areas, such as cardinality estimation [48, 61], query optimization [8, 118], language understanding [43], entity resolution [21], document retrieval [34], graph learning [45, 105] and advertising recommendation [68]. These applications, especially recommendation [30, 32, 82, 108] and retrieval [24, 42, 77, 97], often rely on large amount of embedding vectors to learn semantic representations and extract meaningful patterns and similarities. However, the sheer volume of learnable vectors poses considerable storage challenges in practical deployment scenarios. For example, Meta [71] proposed a deep learning recommendation model (DLRM) equipped with billions of embedding vectors that can take 96 terabytes memory to serve.

The management of these large amount of learnable vectors has become a critical concern for database communities (e.g., cloud-native vector database [27]). One way to address the issue is to involve multiple distributed instances, which may also bring significant communication overheads [69, 93]. Another way is to compress the embedding vectors without compromising the accuracy or the utility of models. During the past few years, various compression methods have been proposed, including hashing, quantization, and so on. However, the performance and the effectiveness of these techniques remain largely unexplored. It is still an open question for data scientists to select from existing compression techniques when the storage of embeddings becomes unbearable.

| | Categorical Features | | | Numerical Features | | Target Label |
|---|---|---|---|---|---|---|
| | User ID | Gender | Ads ID | Dwell Time | Price | Click |
| | 1 | Male | 4 | 1.2 | 99.00 | 1 |
| | 2 | Female | 2 | 3.3 | 34.40 | 1 |
| | 3 | Male | 3 | 4.0 | 12.99 | 0 |
| | 4 | Female | 1 | 2.7 | 67.80 | 0 |

**Figure 1: An example of input data for DLRMs.**

In this paper, we study the above problem by revisiting the embedding compression methods under recommendation and retrieval scenarios since they have the most severe learnable vector storage pressure due to the high-dimensional sparse data [58] and the huge volume of corpus. Figure 1 illustrates an example of input data for DLRMs, which consists of multiple columns of categorical and numerical features, along with a column of target labels. A typical

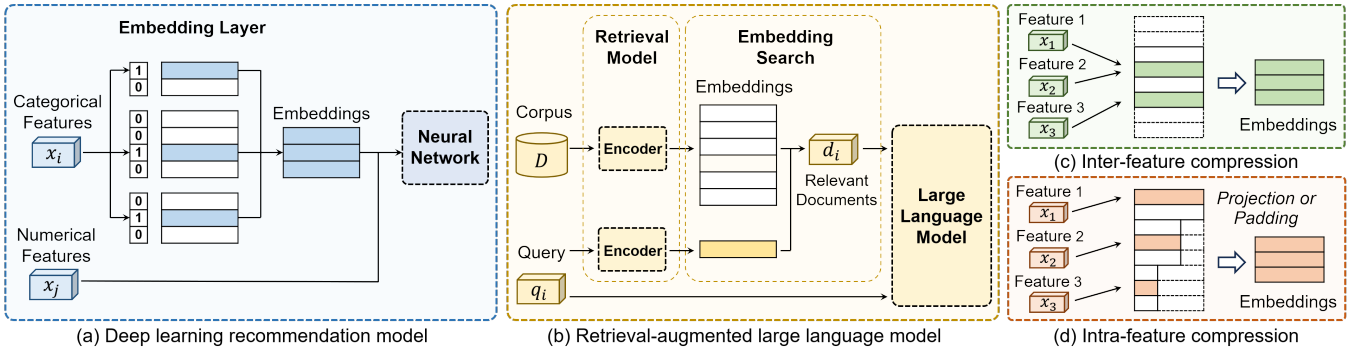


**Figure 2: (a) A typical DLRM. (b) A typical retrieval-augmented LLM. (c) An example of inter-feature compression, the original 8 features now share 4 embeddings. (d) An example of intra-feature compression, each embedding is compressed individually.**

DLRM vectorizes the categorical features into dense embeddings and feeds them along with numerical features into a downstream neural network to make predictions, as shown in Figure 2(a). The embedding layer maintains trainable embedding vectors for all categorical features. It also provides embedding read and write primitives which are similar to the key-value storage [99, 114, 115]. Unfortunately, most existing key-value storage compression techniques [13, 80] are not suitable for DLRMs because of several special characteristics of embeddings, such as skew distribution of embedding popularity [68, 69, 81, 96, 111, 113], frequently accessing and updating of multiple embeddings especially during training. For example, if trained on the Criteo dataset, embeddings are accessed and updated more than 30 times per epoch, with the most popular embeddings being updated almost every iteration. To address the memory issue, many embedding compression methods have been proposed and can be classified into two categories: inter-feature compression and intra-feature compression.

Considering that the storage bottleneck is mainly caused by the increasing number of unique features, **inter-feature compression forces features to share embeddings within a limited memory space**, as shown in Figure 2(c). Inter-feature compression is commonly used in industrial applications [114], and it requires an encoding function to maintain the mapping from features to embeddings. According to whether the encoding function is predetermined or updated during training, we further divide these methods into static encoding [74, 84, 109] and dynamic encoding [14, 17, 40].

Inspired by features' different importance, **intra-feature compression assigns each feature an individually compressed embedding**. Figure 2(d) shows an example, where each feature has its own embedding of distinct dimensions, and the final embeddings are obtained by projection or padding. According to the compression paradigm, intra-feature compression can be further divided into quantization, dimension reduction, and pruning. Quantization is a common compression method in deep learning models that uses data types with fewer bits [28, 53]. Dimension reduction [25, 64, 117] and pruning [46, 64] provide features with embeddings of different dimensions and sparsities, respectively.

Despite the existence of numerous proposed embedding compression methods, a thorough evaluation and analysis remains lacking. To the best of our knowledge, no previous work provides a comprehensive overview of this field. The experiments of existing approaches are often limited to specific cases with restricted metrics and settings. Consequently, the advantages, disadvantages, and applicability of these compression methods have yet to be explored. While benchmarks for DLRMs have been established [75, 121], they primarily focus on model design and do not consider embedding compression. The absence of a comprehensive evaluation framework for various compression methods makes it difficult to reproduce and compare existing techniques, which significantly undermines the practical value of research in this domain.

In addition to DLRMs, retrieval models also have large embedding tables for similarity-based embedding search. Although existing work focuses more on designing efficient embedding search algorithms [7, 79, 86], the emergence of retrieval-augmented large language models (LLMs) [4, 29, 52] brings new challenges to embedding vector storage. A typical retrieval-augmented LLM is shown in Figure 2(b). Since LLMs already consume a lot of memory [5, 78], embedding tables cannot be stored in GPUs or other accelerators, resulting in high search latency. It is currently unclear whether existing learnable vector compression methods are suitable for embeddings generated from retrieval models.

Motivated by the aforementioned issues in this research field, we aim to provide an in-depth analysis and a comprehensive experimental evaluation of embedding compression methods. In this paper, we carry out experiments using a unified evaluation framework to uncover the strengths and weaknesses of each method in various scenarios. We summarize our contributions as follows:

- We propose a new taxonomy of embedding compression methods according to their unique properties. On this basis, we provide a new perspective to understand and analyze their characteristics.
- We construct a unified modular evaluation framework for experiments. We build a general pipeline that can implement a wide variety of compression methods without much effort.
- We comprehensively evaluate representative embedding compression methods using rich metrics in DLRM scenarios. We further discuss the strengths and weaknesses of these methods.
- We apply the embedding compression methods to a retrieval-augmented LLM and analyze their performance.
- We discuss the guidelines, challenges, and promising research directions of embedding compression methods.

## 2 PRELIMINARIES

### 2.1 DLRM

The general architecture of DLRM is depicted in Figure 2(a). A DLRM consists of two parts: an embedding layer mapping each categorical feature into a dense embedding vector, and a neural network containing interaction layers and fully connected layers. Numerical features are fed along with embeddings into the neural network. Many works have been done to improve the performance of the neural network part, such as WDL [15], DCN [90], and DIN [119].

In DLRMs, the categorical feature $x$ can be interpreted as a one-hot vector by encoding function $\mathcal{I}(x)$ to obtain the corresponding row vector $e$ from the embedding table $E \in \mathbb{R}^{n \times d}$ by $e = \mathcal{I}(x)^T E$; or $e = \mathcal{E}(\mathcal{I}(x))$ where $\mathcal{E}$ denotes the embedding layer function. Using $k$ to denote the number of categorical feature fields, and $x_{num}$ to denote numerical features, the downstream neural network is a function $f$ with parameters $\theta$ that inputs embeddings and outputs predictions $\hat{y} = f(e_{i_1}, e_{i_2}, ..., e_{i_k}, x_{num}; \theta)$ for the loss function $\mathcal{L}$. After the forward pass, optimizer such as Adam [44] is applied to update the embeddings and other model parameters. In summary, the optimization of DLRM can be formalized as:

$$\min_{E,\theta} \mathbb{E}_{(X,y)\sim\mathcal{D}} \mathcal{L}(y, f(\mathcal{E}(\mathcal{I}(x)), x_{num}; \theta)). \tag{1}$$

The notations are detailed in Table 1.

### 2.2 Retrieval-augmented LLM

The general structure of retrieval-augmented LLM is depicted in Figure 2(b). A typical model [4, 29, 52] consists of three parts: a retrieval model [42, 77, 100], an embedding search algorithm [7, 79, 86], and an LLM [5, 51, 78]. The retrieval model has two encoders $f_q, f_d$ that encode queries $q$ and all documents $D$ into embeddings separately. The embedding search algorithm $S$ takes query embedding $f_q(q)$ as input, and search similar documents within the embedding table $\mathcal{E} = f_d(D)$. After obtaining relevant documents $S(f_q(q_i), \mathcal{E})$, both the query and the documents serve as input to the LLM $f_{llm}$. The size of the corpus used in industrial applications is at least one million level [73, 110], resulting in a large amount of memory required for embedding table storage. For simplicity, we currently focus on the inference performance of retrieval-augmented LLMs.

### 2.3 Problem Definition

In this section, we discuss the problem of embedding compression in detail. In DLRMs, we use $\mathcal{E}^*$ to denote the compressed embedding layer with trainable parameters $E^*$. The encoding function $\mathcal{I}^*$ for the compressed embedding layer can be one-hot or multi-hot, depending on actual needs. The problem of learning the parameters of a DLRM with a compressed embedding layer can be modified to:

$$\begin{aligned} \min_{\mathcal{E}^*,\mathcal{I}^*,E^*,\theta} & \mathbb{E}_{(X,y)\sim\mathcal{D}} \mathcal{L}(y, f(\mathcal{E}^*(\mathcal{I}^*(x)), x_{num}; \theta)), \\ \text{s.t.} & \mathcal{M}(\mathcal{E}^*, f) \le M_{budget}. \end{aligned} \tag{2}$$

Besides the model parameters $E^*$ and $\theta$, the embedding layer $\mathcal{E}^*$ and the encoding function $\mathcal{I}^*$ are also variables that determine the loss. The optimization process can be decomposed into two parts: the first part determines $\mathcal{E}^*$ and $\mathcal{I}^*$ through the compression method, and the second part trains the model parameters $E^*$ and $\theta$. In this paper, our goal is to provide advice on choosing a proper compression method in the first part.

**Table 1: Commonly used notations.**

| Notation | Explanation |
|---|---|
| $\mathcal{D}$ | Set of data samples |
| $E^{(*)}$ | Parameters of (compressed) embedding layer |
| $\mathcal{E}^{(*)}$ | (Compressed) embedding layer |
| $k$ | Number of categorical fields |
| $X$ | Features of a sample |
| $x, x_{num}$ | Categorical feature, numerical feature |
| $q$ | Input query for retrieval-augmented LLM |
| $\mathcal{I}^{(*)}$ | (Compressed) encoding function |
| $V$ | Set of features |
| $n$ | Number of features |
| $m$ | Number of rows in inter-feature compression |
| $d(')$ | (Reduced) embedding dimension |
| $e$ | Embedding vector |
| $r$ | Density in pruning methods |
| $f, f_q, f_{llm}$ | Neural network of DLRM, query encoder, LLM |
| $S$ | Embedding search function |
| $Dec$ | Embedding decompress function |
| $\theta$ | Model parameters except embeddings |
| $y, \hat{y}$ | Ground truth label, prediction |
| $\mathcal{L}, \mathcal{L}_{llm}$ | Loss functions for DLRM and LLM |
| $\mathcal{M}, M_{budget}$ | Memory usage function, memory budget |
| $CR$ | Compression ratio |

In retrieval-augmented LLMs, we use $Dec$ to denote the decompress function, and abuse some common notations such as $\mathcal{E}^*$ for compressed embeddings and $y$ for labels. Assuming our target is to minimize the objective function $\mathcal{L}_{llm}$, then the problem of choosing the most proper compression method can be formed as:

$$\begin{aligned} \min_{\mathcal{E}^*} & \mathbb{E}_{q\sim\mathcal{D}} \mathcal{L}_{llm}(y, f_{llm}(q, S(f_q(q), Dec(\mathcal{E}^*)))), \\ \text{s.t.} & \mathcal{M}(\mathcal{E}^*, f_q, f_{llm}, S) \le M_{budget}. \end{aligned} \tag{3}$$

Since we are targeting the inference stage, the only variable is the compression algorithm. In practical applications, search algorithms are usually performed in batches, so the decompression can also be a batch operation to avoid storing the complete embedding table.

The compressed embeddings $\mathcal{E}^*$ should meet the memory constraint. The memory function $\mathcal{M}$ outputs the memory usage of the whole model during inference. In real scenarios, especially on-device situations, the memory budget is often smaller than the memory usage of models with the full embedding table. Since the memory usage of other parts is fixed, the memory constraint can be simplified as $\mathcal{M}(\mathcal{E}^*) \le M_{budget}$. In addition to inference memory constraints, more metric constraints can be applied, such as low latency requirements in online service scenarios, training time or training memory constraints in time- or memory-limited scenarios.

Some methods cannot compress embedding layers within a given memory budget; they can only compress to a specific target memory. For instance, quantization methods directly adopt INT8 or INT16 to replace the original FLOAT32 data type, reducing the memory usage to 25% or 50%. To measure the compression ability, we define $CR$ (compression ratio) as the ratio of the original memory to the compressed memory as $CR = \frac{\mathcal{M}(\mathcal{E})}{\mathcal{M}(\mathcal{E}^*)}$.

## 2.4 Scope

In this section, we discuss the scope of this paper. We focus on embedding compression for DLRMs and retrieval-augmented LLMs, with practical applications of at least millions of embeddings. We do not focus on the embeddings in NLP models. Although works have been done to compress the memory usage of embedding tables in NLP [6, 9, 10, 85, 87], the number of unique vocabularies in mainstream LLMs such as Bert [20] and GPT-3 [5] is no more than 0.1 million, which is not as memory-intensive as DLRMs.

There are several research directions that can be easily confused with our study: feature selection, embedding search. Although feature selection [59, 63, 92] does reduce memory usage by directly pruning useless features, it can be seen as the upstream process of embedding compression. Embedding search [39, 65, 66] is a subsequent stage of embedding compression in retrieval tasks, and its related research is orthogonal to memory compression.

# 3 OVERVIEW OF EMBEDDING COMPRESSION

In this section, we present an overview and a new taxonomy of embedding compression methods. We first divide all methods into inter-feature and intra-feature compression based on whether the features share parameters or have individually compressed embeddings. We further divide the methods according to their properties and techniques. The detailed information of inter-feature and intra-feature compression methods is listed in Table 2 and 3, respectively.

## 3.1 Inter-feature Compression

To address the memory bottleneck caused by the explosive growth of features, a direct approach is to keep only a small number of embeddings for features to share, as shown in Figure 2(c). Compression is generally performed within fields to ensure that features which share embeddings have similar semantics. Inter-feature compression needs to maintain a new mapping from features to embeddings, instead of the original one-hot encoding. Early methods utilize hash functions [87, 95] to map features into multi-hot vectors, then lookup from hash embedding tables for sub-embeddings to construct final embeddings. Following this idea, the problem can be simplified as finding an encoding function $\mathcal{I}^*$, and a corresponding row-compressed embedding layer $\mathcal{E}^*$. Based on whether the encoding function is fixed during training, the methods can be further divided into static encoding and dynamic encoding.

*3.1.1 Static Encoding.* Static encoding uses fixed encoding functions during training. Mapping features into a smaller number of embeddings is essentially a hashing process. Thus, many hash functions have served as encoding functions in industry [114]. While early works explored the form of encoding functions, recent works further explored the form of embedding layers. We use $m$ to denote the number of embeddings after compression.

**DoubleHash** [109] uses two hash functions, and sums the two sub-embeddings together. More hash functions lead to less collision rate since the bucket size is enlarged from $m$ to $m^2$.

**CompoEmb** [84] recursively divides the original feature index by the row sizes of hash embedding tables and gets the remainders as the new indices. As long as the product of row sizes is greater than the number of features, no features will share the exact same embedding. The sub-embeddings are aggregated by multiplication. BinaryCode [102] follows the idea, splitting the binary respresentation of the original index in succession style or skip style, where the former is essentially the CompoEmb. Some following work [56] also adopts CompoEmb to implement lightweight embedding layers.

**MEmCom** [74] stores scale and bias weights for each feature. Given a feature as input, a embedding is indexed by a hash function, then multiplied and added with scale and bias to get the final embedding.

Methods above share some sub-embeddings among features, resulting in degraded model quality. They only form the encoding function, enabling simple and flexible memory compression. In contrast, the following methods design new embedding layers.

**DHE** (Deep Hash Embedding [41]) radically replaces embedding tables with multi-layer perceptrons (MLPs). It maps features to integers using many hash functions and then applies transformations to approximate a uniform or Gaussian distribution as input to MLPs. Equipped with complex MLPs, DHE achieves good model quality, but requires much more time to train and infer. In Table 2, the symbol $d_i$ means the number of hidden units in each MLP layer.

**TT-Rec** (Tensor-Train Recommendation [33, 107]) borrows the idea of tensor-train decomposition (abbreviated as TT). In TT, a tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \ldots I_t}$ can be decomposed $\mathcal{A} \approx \mathcal{G}_1 \mathcal{G}_2 \ldots \mathcal{G}_t$, with each TT-core $\mathcal{G}_i \in \mathbb{R}^{R_{i-1} \times I_i \times R_i}$, $R_0 = R_t = 1$. TT-Rec factorizes the row size $|V| \leq \Pi_{i=1}^{t} m_i$ and the column size $d \leq \Pi_{i=1}^{t} d_i$ into intergers, and decomposes the embedding table by $E \approx \mathcal{G}_1 \mathcal{G}_2 \ldots \mathcal{G}_t$, where $\mathcal{G}_i \in \mathbb{R}^{R_{i-1} \times m_i \times d_i \times R_i}$. To obtain the embedding, TT-Rec looks up the tensors and conducts matrix multiplication according to the decomposition. Its row decomposition is the same with CompoEmb, but the matrix multiplication requires much more time than simple aggregation. TT-Rec is also adopted by following work [89].

**ROBE** (Random Offset Block Embedding [19]) stores an 1-D array instead of 2-D matrix for embedding layer. It uses hash functions to generate indices, then concatenates the sub-embeddings retrieved at the indices. ROBE can reduce running time with simple design, but requires more epochs to converge due to the randomness.

The exploration of embedding layer design is both creative and effective. They either improve the model quality at the cost of more computation, or simplify the embedding structure.

**Dedup** [120] conducts similarity-based deduplication [50, 88] on embedding models. It adopts L2LSH [36], a local-sensitive hashing algorithm on Euclidean (L2) distance, to efficiently deduplicate similar parameter blocks. Dedup can only be applied when the parameters are fixed, so uncompressed embeddings still need to be trained. We classify Dedup as static encoding, because the encoding function is determined by an one-pass LSH process. Since it deduplicates embeddings by value, we directly deduplicate the entire embedding table to speed up compression and serving, regardless of feature fields. Dedup hashes the embedding content while other hashing-based methods hash the input indices, so we distinguish them explicitly in our experimental analysis.

In summary, static encoding methods are simple, effective, and capable of compression at any memory budget. Their encoding functions, which remain constant during training, are often simple hash functions that take up no storage space. The focus of research work gradually shifts from hash functions to embedding layers. The former guarantees memory constraints while the latter further

**Table 2: Summary of inter-feature compression. Space Complexity reflects the memory of encoding function and embedding layer; Time Complexity reflects the time of embedding lookup process; Freq / Impo-aware indicates whether the method is frequency- or importance-aware; Com Cap shows the compression capability within a certain range of memory budgets.**

| Subcategory | Method | Techniques | Space Complexity | Time Complexity | Freq / Impo-aware | Com Cap |
|---|---|---|---|---|---|---|
| Static Encoding | CompoEmb [84] | Hash | $O(md + nd/m)$ | $O(d)$ | / | Yes |
| | DoubleHash [109] | Hash | $O(md)$ | $O(d)$ | Frequency-aware | Yes |
| | BinaryCode [102] | Hash | $O(\sqrt{n} \cdot d)$ | $O(d)$ | / | No |
| | MemCom [74] | Hash | $O(md + n)$ | $O(d)$ | / | Yes |
| | DHE [41] | Hash, MLP | $O(d \cdot d_i + d_i^2)$ | $O(d \cdot d_i + d_i^2)$ | / | Yes |
| | TT-Rec [107] | TensorTrain (Hash) | $O(R^2 \sum(m_i + d_i))$ | $O(R^2 d)$ | Frequency-aware | Yes |
| | ROBE [19] | Hash, 1-D Array | $O(md)$ | $O(d)$ | / | Yes |
| | Dedup [120] | LSH | $O(n + md)$ | $O(d)$ | / | Yes |
| Dynamic Encoding | MGQE [40] | VQ (PQ) | $O(nk' + md)$ | $O(k' + d)$ | Frequency-aware | Yes |
| | LightRec [57] | VQ (AQ) | $O(nk' + md)$ | $O(k'd)$ | / | Yes |
| | AdaptEmb [17] | Hash, Frequency | $O(m + md)$ | $O(d)$ | Frequency-aware | Yes |
| | CEL [14] | Clustering | $O(n + 2B/b \cdot d)$ | $O(d)$ | Frequency-aware | Yes |

guarantees model quality. Another line of research is similarity-based deduplication, which performs post-training compression.

*3.1.2 Dynamic Encoding.* Dynamic encoding allows encoding functions to be updated during training, which is naturally suitable for online learning. They adopt trainable indices or build data structures to store and adjust the mapping. They tend to incorporate more information but only achieve mediocre compression ratios.

**MGQE** (Multi-Granularity Quantized Embedding [40]) extends DPQ (Differentiable Product Quantization) [9] to fit recommendation data. DPQ is based on PQ (Product Quantization) [37], a VQ (Vector Quantization) technique in embedding search. PQ splits embeddings into several parts, clusters the partial embeddings respectively, then reconstructs embeddings with the nearest centroids. DPQ introduces supervised learning to train the centroids, minimizing the distances between the original and the reconstructed embeddings. The uncompressed embeddings are kept during training to determine and update the nearest sub-embeddings. After training, the uncompressed embeddings are dropped, and the nearest sub-embeddings are adopted to reconstruct the final embeddings. DPQ and other similar works [10, 85] focus on NLP word embeddings, and MGQE extends DPQ for highly-skewed recommendation data, providing more centroid embeddings for features with higher frequency. The memory usage can only be reduced during inference, and the compression ratio is relatively low due to the storage of centroids indices. Besides PQ, other VQ techniques such as AQ (Additive Quantization) are also adopted for embedding compression [57]. The centroids in AQ are summed to reconstruct embeddings. In Table 2, $k'$ is the number of parts in VQ.

**AdaptEmb** (DeepRec Adaptive Embedding [17]) allocates unique embeddings for high-frequency features and shared embeddings for others. It dynamically converts the feature's embedding from shared to exclusive if the frequency becomes high enough. It incurs extra memory to store high frequency features during inference.

**CEL** (Clustered Embedding Learning [14]) compresses the embeddings of two special fields (users and items) that are clustered with only one embedding per cluster. During training, items are dynamically reassigned to the more proper clusters based on their history interactions, and clusters are split if associated with too many interactions. CEL has limited compression ratios with the storage of the cluster structure, and takes more training time due to cluster adjustment. In Table 2, the total number of interactions is $B$ and the cluster will split iff it has more than $2b$ associated interactions.

Dynamic encoding requires extra data structures to store dynamic codes, so sometimes only supports limited compression ratios. They incorporate frequency information, but the dynamic encoding function brings some overheads during training.

## 3.2 Intra-feature Compression

Instead of sharing embeddings and modifying the encoding function, intra-feature compression compresses embeddings individually to form a new embedding layer $\mathcal{E}^*$. These methods can be further divided into quantization, dimension reduction, pruning.

*3.2.1 Quantization.* Quantization is a common compression technique in deep learning training [35, 70] and inference [2]. It is stable and simple to use, since it does not affect the original training paradigm; however, low-precision data types will lead to a slight loss of model quality and limited compression ratios.

**FP16** [112] is only used for storage, while during training the retrieved embeddings are converted to FP32. When rounding updated parameters back into FP16, there are two choices: nearest rounding and stochastic rounding. The former selects the nearest value in FP16, appearing to have a systematic bias in model update accumulation since a relatively small update will never take effect if always discarded. The latter first computes the rounded-up value and the rounded-down value, then draws a random number from a Bernoulli distribution with the distances to these values; this approach is not biased yet brings higher variance in optimizer. In practice, stochastic rounding is chosen for better model quality.

**INT8/16** [101, 104] are integer data types of low-precision, treated as bins of values, where two manually designed parameters scale and bias are further required to restore the original FP32 value. FP data types have unequal intervals between values, while INT data types have equal intervals. INT data types also adopt stochastic rounding for better model quality.

Directly using FP16 or INT8/16 is simple and has almost no overhead. In order to obtain better model quality, the following methods try to search or learn the proper scale for INT-type compression.

**Table 3: Summary of inter-feature compression. Space Complexity reflects the memory of encoding function and embedding layer; Time Complexity reflects the time of embedding lookup process; Freq / Impo-aware indicates whether the method is frequency- or importance-aware; Com Cap shows the compression capability within a certain range of memory budgets.**

| Subcategory | Method | Techniques | Space Complexity | Time Complexity | Freq / Impo-aware | Com Cap |
|---|---|---|---|---|---|---|
| Quantization | FP16 [112] | FP16 | $O(nd/2)$ | $O(d)$ | / | No |
| | Post4Bits [26] | Greedy Search | $O(nd/8)$ | $O(d)$ | / | No |
| | MixedPrec [104] | FP16, INT8 | $O(nd/4)$ | $O(d)$ | / | No |
| | Int8/16 [101] | INT8, INT16 | $O(nd/4)$ | $O(d)$ | / | No |
| | ALPT [55] | Learnable Scale | $O(nd/4+n)$ | $O(d)$ | Importance-aware | No |
| Dimension Reduction | NIS [38] | Policy Gradient | $O(nd')$ | $O(d'd)$ | Importance-aware | Yes |
| | ESAPN [60] | Policy Gradient | $O(nd')$ | $O(d'd)$ | Both | No |
| | MDE [25] | Heuristic | $O(nd')$ | $O(d'd)$ | Frequency-aware | Yes |
| | AMTL [103] | MLP | $O(nd')$ | $O(d)$ | Both | No |
| | AutoEmb [116] | DARTS | $O(nd')$ | $O(d'd)$ | Both | No |
| | AutoDim [117] | DARTS | $O(nd')$ | $O(d'd)$ | Importance-aware | No |
| | SSEDS [76] | One-shot NAS | $O(nd')$ | $O(d'd)$ | Importance-aware | Yes |
| | OptEmbed [64] | One-shot NAS | $O(nd')$ | $O(d)$ | Importance-aware | No |
| Pruning | DeepLight [18] | Structural Prune | $O(rnd)$ | $O(rd)$ | Importance-aware | Yes |
| | PEP [62] | Mask, Threshold | $O(rnd)$ | $O(rd)$ | Importance-aware | No |
| | HAM [98] | STE, Hard Mask | $O(rnd)$ | $O(rd)$ | Importance-aware | Yes |
| | AutoSrh [46] | Mask, DARTS | $O(rnd)$ | $O(rd)$ | Both | Yes |

**Post4Bits** [26] performs a post-training greedy search on scale and bias of INT4 data type. The minimum and the maximum values are searched step by step to minimize the model loss.
**ALPT** [55] makes the scale alternatively trained with the model parameters to improve the model quality. The idea of learnable scale comes from LSQ [22].

In summary, quantization involves little overhead and achieves certain compression ratios.

*3.2.2 Dimension Reduction.* Generally, the larger the dimension, the more information the embedding can represent. As the recommendation data is highly skewed [111], a natural idea is to assign different dimensions for features with different frequency or importance. To align the dimension of embeddings for subsequent neural networks, there are two ways: zero-padding and projection. The second scheme is inspired by SVD [6] and is adopted by most methods, since the learnable projection matrices can represent all the linear transformations including zero-padding. The symbol $d'$ in Table 3 means the reduced dimension.
**MDE** (Mixed Dimension Embedding [25]) represents feature frequency with the inverse of feature cardinality within field. The dimensions are proportional to $p^\alpha$, where $p$ is the frequency and $\alpha$ is a hyper-parameter.

MDE is the only dimension reduction method that compresses memory during training, with no learnable structures involved. All of the following methods adopt learnable structures to determine dimensions, incurring much more training overhead.
**NIS** (Neural Input Search [38]) uses a policy network to determine the dimensions. It splits the embeddings into chunks. For each column chunk, it builds projection matrices and uses a controller to sample row chunks. In the reward $R = R_Q - \lambda \cdot C_M$, $R_Q$ is the model quality and $C_M$ is the memory cost at inference. The chunk-based search is also used in [12].
**ESAPN** (Embedding Size Adjustment Policy Network [60]) uses a series of projection matrices to convert dimensions larger and larger until the final dimension. Each feature field is assigned a policy network, which inputs the feature frequency and the current dimension and outputs whether enlarge the dimension. If the dimension is enlarged, the transformed vector is used as initialization. It takes the improvement of the current state as reward.

The above two methods adopt policy network to learn dimensions, incurring much training overhead for the trials of different settings. The memory budget can be considered in reward function just as NIS to enforce memory constraint at inference.
**AutoEmb** [116] forms MLP-based controllers, which take frequency and other contextual information as input and output probabilities of dimensions. Controllers and other model parameters are trained alternatively using DARTS [106] solution for bi-level optimization.
**AutoDim** [117] defines field-wise architectural weights to compute probabilities of dimensions via gumbel-softmaxing. It also alternatively trains the architectural weights and the other model parameters using DARTS. After training, the dimension with the highest probability is selected for further re-training.

The above two methods both utilize DARTS, incurring lower training overhead than policy network. However, they cannot search within a given memory budget.
**AMTL** (Adaptively-Masked Twins-based Layer [103]) introduces two MLPs for features with high- and low-frequency respectively, to output scores for positions where embeddings should be truncated.
**SSEDS** (Single-Shot Embedding Dimension Search [76]) multiplies the pre-trained uncompressed embeddings with field-dimension-wise masks to conduct single-shot NAS. It uses the masks' gradients to represent value importance, which is further used to truncate the embeddings according to memory budget.
**OptEmbed** [64] jointly learns masks for both row and column. Row masks that threshold the embeddings' $L_1$ norms are multiplied onto the original embeddings for supernet training. After determining the row masks, OptEmbed conducts an evolutionary search to determine the column masks for embeddings truncation. Then the compressed embeddings are retrained to fit the masked parameters.

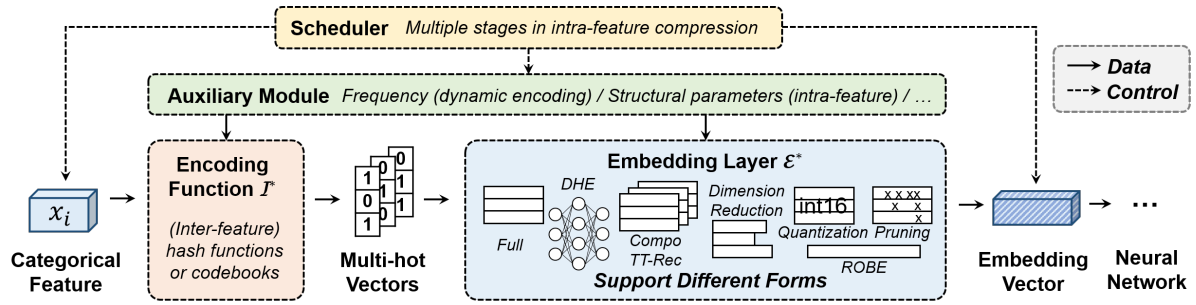


**Figure 3: Overview of the evaluation framework.**

SSEDS and OptEmbed utilize one-shot NAS to make the training process faster. While SSEDS takes the memory budget into consideration, OptEmbed does not support flexible memory budget.

In summary, dimension reduction methods aim to assign a suitable dimension for each feature. Except for MDE and AMTL, all methods require significant time for complex training or retraining. Except for AMTL and OptEmbed, other methods use projection matrices, which result in increased inference latency. Only MDE, NIS and SSEDS can compress to a given memory budget. There are also other methods that jointly optimize the embedding dimension and model components using rule [83], DARTS [106] or one-shot NAS [94], which do not meet our plug-and-play requirement.

*3.2.3 Pruning.* Pruning is a common technique in the compression of deep learning models [23, 31, 54]. According to the lottery ticket hypothesis [23], a dense neural network contains a subnetwork that can match the test accuracy of the dense network. Similar to dimension reduction that assigns different dimensions, pruning assigns different sparsity for different features. The pruned sparse embeddings are stored in sparse tensor format in practice. The symbol $r$ in Table 3 means the density of embeddings.

**DeepLight** [18] uses structural pruning, a common pruning method in DL models [1]. It progressively thins out embeddings by filtering small-magnitude values, until reaching the memory budget.

Except for DeepLight, all the others adopt learning methods, involving more training overhead for better model quality.

**PEP** (Plug-in Embedding Pruning [62]) defines a learnable threshold for pruning. After joint training the threshold and the other parameters, the model is retrained to fit the pruned embeddings.

**HAM** (Hard Auxiliary Mask [98]) first pre-trains the uncompressed embeddings with Soft Orthogonal [3] regularizations, then alternatively trains learnable masks and other parameters, and finally re-trains the pruned embeddings.

**AutoSrh** [16, 46] sorts features by frequency and partitions them into blocks, with each block assigned with learnable masks for pruning. Masks and other model parameters are alternatively trained using DARTS. After training, parameters are filtered according to memory budget, then re-trained to fit the sparse embeddings.

The above methods learn masks or thresholds for pruning. HAM and AutoSrh update learnable masks alternatively with parameters, which is similar to SSEDS and OptEmbed in dimension reduction. Like dimension reduction, pruning attempts to allocate more memory to more important features. Pruning methods achieve good model quality with significant training overhead. They are flexibly adapted to a given memory budget, but require system support for sparse tensor storage and computation.

## 4 EVALUATION FRAMEWORK

We design and implement a unified modular evaluation framework for embedding compression, as shown in Figure 3. Generally, all existing embedding compression methods can be implemented with these 4 modules: encoding function, embedding layer, scheduler, and auxiliary module. The encoding function inputs features and outputs one-hot or multi-hot vectors. The embedding layer stores embedding-related parameters, such as one or several embedding tables, MLPs, 1-D arrays, and sparse matrices, etc. For sparse matrices, we implement CSR and COO formats, and the framework adaptively chooses the format under a given memory budget. The embedding layer outputs the corresponding embeddings based on the encoded vectors, then the embeddings are fed into neural networks along with numerical features for predictions. We omit the neural network part from the figure because it is not our focus. The optional auxiliary module contains data structures that assist model training, such as the frequency information in dynamic encoding, the learnable masks in pruning, the architecture weights in dimension reduction. The scheduler manages the entire training process, switches training stages, and schedules proper data to train certain parts of the model. For example, DARTS-based methods use training data and evaluation data to update model parameters and architecture weights respectively, while NAS-based methods usually require a re-training stage for further improvement.

The framework integrates 14 representative methods for experimental comparison, which are listed in Section 5.1.2. Besides existing methods, our framework supports any new method that applies this compression pipeline. We expect more compression methods to be proposed based on our framework.

The framework is implemented on Hetu [67], an efficient deep learning system. Our framework consists of 10 thousand lines of code in Python for the modules. We also implement some necessary C++/CUDA computing kernels. The framework does not explicitly consider distributed scenarios: data parallelism can be simply applied, while model parallelism that partitions embedding layers are not necessary because embedding layers have already been compressed. Some other orthogonal system optimizations such as data prefetch [68], or DL compilation [11] are not applied, as they do not affect our analysis.

## 5 EXPERIMENTS AND ANALYSIS

In this section, we experimentally evaluate the embedding compression methods on DLRM (Section 5.2). We design experiments to reveal the influence of neural network models (Section 5.3) and embedding dimensions (Section 5.4). We also apply the embedding compression methods to retrieval-augmented LLM (Section 5.5). We later discuss challenges and future directions (Section 5.6).

### 5.1 Experiment Settings of DLRM

*5.1.1 Models and Datasets.* We experiment on three popular models: DLRM[1] [72], WDL [15], and DCN [90]. We evaluate three click-through rate (CTR) datasets: Avazu [91], Criteo [49], and Company, where the former two are widely used in academia and have been employed in recommendation benchmarks [75, 121], and the latter is collected from a recommendation scenario in Tencent containing ad features. The statistics of the datasets are listed in Table 4.

**Table 4: Overview of the datasets.**

| Datasets | # Fields | # Features | # Samples |
|---|---|---|---|
| Avazu | 22 | 9,449,445 | 40,428,967 |
| Criteo | 26 | 33,762,577 | 45,840,617 |
| Company | 43 | 66,102,027 | 35,682,429 |

Feature frequency follows a power law [68, 69, 81, 96, 111, 113]. For example, in Avazu and Criteo, the top 10% features with the highest frequency account for more than 95% of the occurrences in samples, while for the long-tail part, more than 80% of the features have less than 5 occurrences. Compression methods are inspired to allocate different amount of memory to features.

*5.1.2 Compared Methods.* We choose 14 representative methods for comparison. For static encoding, **CompoEmb** uses multiple hash functions, and DoubleHash, MEmCom can be regarded as its variants; **TT-Rec** is a special variant of CompoEmb that borrows the idea of tensor-train decomposition; **DHE** and **ROBE** explore different forms of embeddings; **Dedup** is the state-of-the-art similarity-based deduplication method. For dynamic encoding, **MGQE** learns the codes of sub-embeddings and has a simpler PQ structure than LightRec; **AdaptEmb** employs feature frequency and is more general than another frequency-based method CEL. For quantization, we choose **INT8/16** for fixed quantization with uniform value distribution; **ALPT** is the state-of-the-art method that learns the quantization scale. For dimension reduction, **MDE** is the only heuristic-based method; **AutoDim** and **OptEmbed** are the state-of-the-art methods using trainable structural parameters and one-shot NAS respectively; we do not evaluate policy-gradient-based methods because they are time consuming and perform poorly. For pruning, **DeepLight** is the only structural pruning method; **AutoSrh** is the state-of-the-art method that learns the pruning structure.

*5.1.3 Environment and Hyperparameters.* We use the Adam optimizer [44] with the learning rate grid-searched from [0.001, 0.01, 0.1], and the batch size is 64 (for Company) or 128 (for Avazu and Criteo). We conduct every single experiment on an Nvidia RTX TITAN 24 GB GPU card. We tested different dimensions on uncompressed embeddings, and selected 16 as the embedding dimension.

For simplicity, we implement all methods on GPU. The location of the embeddings does not affect model accuracy or memory usage. If the embedding layer is on CPU, embeddings need to be transferred to GPU with additional communications, and compute-intensive methods like TT-Rec and DHE which already have the highest latency will be slower. These two factors only affect the absolute value of processing time, not the relative ranking of each method.

*5.1.4 Metrics.* We employ AUC (area under the ROC curve) to measure model quality. In recommendation systems, an improvement of 0.001 in AUC is considerable. We measure memory usage by the actual memory consumption of the embedding layer at inference. This is more effective than using the number of parameters or sparsity rate, which do not account for the compression effect of quantization methods or the additional memory cost of sparse formats. For training memory, we include the memory of the auxiliary modules. For training time, we measure the total time of training to convergence, including all stages. Inference latency is the forward pass time of a batch using well-trained model checkpoints.

### 5.2 Performance on DLRM

Table 5 shows the results on DLRM under different inference memory budgets. We use the uncompressed embedding table as the baseline method, and its memory usage as the baseline memory usage. By default, the inference memory budgets are 50%, 10%, 1%, 0.1% of the baseline memory. Methods that cannot achieve these compression ratios are compressed as much as possible, with their actual memory usage listed in parentheses; these results are not explicitly compared with those normal ones.

*5.2.1 Ability of Compression.* **Hash-based methods (including LSH) and pruning methods are the most capable compression methods, achieving all compression ratios.** Static encoding methods and AdaptEmb can simply adjust the number of rows, while pruning methods can flexibly change the sparsity. The storage of auxiliary mapping in LSH-based Dedup can be reduced by using large-sized tensor blocks. All the other methods have certain limitations on their compression capabilities. Other dynamic encoding methods need to store feature-to-embedding mappings with memory proportional to the number of features, leading to an upper bound of 16× compression ratio. Quantization methods are limited to several specific compression ratios: 2× and 4× for simple INT8/16; 1.8× and 3.2× for ALPT which requires more memory for feature-wise step sizes. Dimension reduction methods learn optimal dimensions based on model quality rather than memory budgets: MDE and AutoDim assign features with at least one dimension, leading to an upper bound of 16× compression ratio; AutoDim and OptEmbed achieve specific compression ratios within $2 - 2.7\times$.

*5.2.2 Model AUC.* **In general, static encoding, quantization, and pruning methods achieve the best model AUC.** For each memory budget, the methods with top-3 AUC are highlighted in bold, coupled with an underlined ranking. There is no single method that performs best in all situations. Specifically, quantization methods perform well around 25% or 50% of the baseline memory, since

[1] In Section 5 we use the term DLRM to refer to this particular model, rather than the general deep learning recommendation models in the previous sections.

**Table 5: Overall performance on DLRM.**

| Methods | | Metrics | Avazu | | | | Criteo | | | | Company | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 50% | 10% | 1% | 0.1% | 50% | 10% | 1% | 0.1% | 50% | 1% |
| Baseline | Full | AUC | 0.7543 (100.0%) | | | | 0.8061 (100.0%) | | | | 0.7583 (100.0%) | |
| | | TrainMem | 300.0% | | | | 300.0% | | | | 300.0% | |
| | | TrainTime | 1m36s | | | | 8m39s | | | | 9m45s | |
| | | Latency | 0.56ms | | | | 0.88ms | | | | 0.61ms | |
| Static Encoding | CompoEmb | AUC | 0.7491 | 0.7480 | 0.7472 | / | 0.8060 | **0.8053**$_{3}$ | 0.8016 | / | 0.7503 | 0.7228 |
| | | TrainMem | **150.0%** | **30.0%** | **3.0%** | / | **150.0%** | **30.0%** | **3.0%** | / | **150.0%** | **3.0%** |
| | | TrainTime | **6m42s**$_{3}$ | **8m51s**$_{2}$ | **5m23s**$_{1}$ | / | 54m20s | **51m41s**$_{3}$ | **50m27s**$_{3}$ | / | 2h20m | **1h58m**$_{3}$ |
| | | Latency | 3.27ms | 3.34ms | 2.83ms | / | 4.47ms | 3.55ms | 3.92ms | / | 6.67ms | 6.57ms |
| | TT-Rec | AUC | 0.7497 | 0.7537 | 0.7542 | 0.7517 | 0.7876 | 0.7993 | 0.8025 | 0.7997 | 0.7255 | 0.7329 |
| | | TrainMem | **149.3%** | **29.8%** | **2.9%** | **0.3%** | **149.4%** | **30.0%** | **3.0%** | **0.3%** | **149.9%** | **2.9%** |
| | | TrainTime | 7h29m | 1h37m | 36m32s | **40m48s**$_{3}$ | 12h8m | 12h4m | 3h29m | 4h30m | 127h27m | 5h53m |
| | | Latency | 84.41ms | 21.28ms | 5.00ms | 4.28ms | 246.39ms | 82.72ms | 11.99ms | 6.10ms | 505.89ms | 24.05ms |
| | DHE | AUC | **0.7583**$_{1}$ | **0.7581**$_{1}$ | **0.7586**$_{1}$ | **0.7563**$_{1}$ | 0.8056 | **0.8053**$_{3}$ | 0.8027 | 0.8004 | 0.7532 | **0.7442**$_{2}$ |
| | | TrainMem | **149.9%** | **30.0%** | **3.0%** | **0.3%** | **150.0%** | **30.0%** | **3.0%** | **0.3%** | **150.0%** | **3.0%** |
| | | TrainTime | 3h11m | 45m15s | 1h12m | 12h53m | 5h44m | 2h10m | 4h35m | 12h14m | 7h45m | 34h29m |
| | | Latency | 5.81ms | 3.37ms | 3.02ms | 5.91ms | 12.86ms | 6.52ms | 6.26ms | 8.58ms | 17.05ms | 20.35ms |
| | ROBE | AUC | 0.6612 | 0.6522 | 0.6477 | 0.6407 | 0.7510 | 0.7479 | 0.7514 | 0.7490 | 0.5945 | 0.5828 |
| | | TrainMem | **150.0%** | **30.0%** | **3.0%** | **0.3%** | **150.0%** | **30.0%** | **3.0%** | **0.3%** | **150.0%** | **3.0%** |
| | | TrainTime | 11m11s | 11m51s | 1h6m | 47m18s | 5h13m | 5h20m | 5h13m | 5h21m | **8m6s**$_{1}$ | **8m4s**$_{1}$ |
| | | Latency | 0.66ms | **0.66ms**$_{3}$ | **0.67ms**$_{3}$ | 0.65ms | **0.92ms**$_{1}$ | **0.87ms**$_{1}$ | **0.95ms**$_{2}$ | **0.91ms**$_{2}$ | 0.67ms | 0.67ms |
| | Dedup | AUC | **0.7547**$_{2}$ | **0.7555**$_{2}$ | **0.7573**$_{3}$ | **0.7560**$_{2}$ | **0.8076**$_{1}$ | **0.8071**$_{1}$ | **0.8057**$_{3}$ | **0.8035**$_{1}$ | **0.7553**$_{3}$ | **0.7420**$_{3}$ |
| | | TrainMem | 300% | 300% | 300% | 300% | 300% | 300% | 300% | 300% | 300% | 300% |
| | | TrainTime | **2m44s**$_{2}$ | **5m58s**$_{1}$ | **9m54s**$_{2}$ | **11m12s**$_{1}$ | **10m10s**$_{2}$ | **10m7s**$_{1}$ | **18m48s**$_{1}$ | **29m9s**$_{1}$ | **10m1s**$_{2}$ | **10m24s**$_{2}$ |
| | | Latency | 0.71ms | 0.82ms | 0.68ms | **0.64ms**$_{2}$ | 1.29ms | **1.04ms**$_{3}$ | **0.97ms**$_{3}$ | **0.86ms**$_{1}$ | 0.69ms | **0.65ms**$_{3}$ |
| Dynamic Encoding | MGQE | AUC | 0.7286 (16.0%) | | 0.7332 (11.1%) | | 0.7908 (16.1%) | | 0.7891 (11.2%) | | 0.6881 (16.0%) | 0.6477 (11.1%) |
| | | TrainMem | 322.3% | | 317.4% | | 322.7% | | 317.8% | | 322.2% | 317.4% |
| | | TrainTime | 20m0s | | 34m33s | | 56m37s | | 37m45s | | 3h36m | 1h13m |
| | | Latency | 3.40ms | | 3.24ms | | 4.33ms | | 4.37ms | | 6.72ms | 6.74ms |
| | AdaptEmb | AUC | 0.7513 | 0.7484 | 0.7407 | 0.7302 | 0.8051 | 0.8026 | 0.7990 | 0.7921 | **0.7566**$_{2}$ | 0.7136 |
| | | TrainMem | 154.4% | 35.3% | 9.2% | 6.5% | 154.4% | 35.3% | 9.2% | 6.5% | 154.4% | 9.2% |
| | | TrainTime | 13m58s | **10m26s**$_{3}$ | **13m13s**$_{3}$ | 45m14s | 52m3s | **49m43s**$_{2}$ | **48m36s**$_{2}$ | **1h46m**$_{2}$ | 2h7m | 2h11m |
| | | Latency | 4.63ms | 3.59ms | 3.40ms | 3.13ms | 5.79ms | 4.40ms | 5.01ms | 4.55ms | 7.32ms | 7.30ms |
| Quantization | INT8/16 | AUC | **0.7539**$_{3}$ | 0.7524 (25.0%) | | | **0.8071**$_{2}$ | 0.8045 (25.0%) | | | 0.7493 | 0.7536 (25.0%) |
| | | TrainMem | 250.0% | 225.0% | | | 250.0% | 225.0% | | | 250.0% | 225.0% |
| | | TrainTime | **2m24s**$_{1}$ | 4m31s | | | **9m44s**$_{1}$ | 8m8s | | | **10m16s**$_{3}$ | 9m16s |
| | | Latency | **0.65ms**$_{3}$ | 0.65ms | | | **1.01ms**$_{3}$ | 0.88ms | | | **0.63ms**$_{3}$ | 0.58ms |
| | ALPT | AUC | 0.7545 (56.3%) | | 0.7511 (31.3%) | | 0.8062 (56.3%) | | 0.8057 (31.3%) | | 0.7562 (56.3%) | 0.7532 (31.3%) |
| | | TrainMem | 268.8% | | 243.8% | | 268.8% | | 243.8% | | 268.8% | 243.8% |
| | | TrainTime | 3m5s | | 2m51s | | 15m46s | | 12m45s | | 16m28s | 17m22s |
| | | Latency | 0.63ms | | 0.64ms | | 0.92ms | | 1.01ms | | 0.59ms | 0.61ms |
| Dimension Reduction | MDE | AUC | 0.7502 | 0.7504 (8.1%) | | | 0.8044 | 0.8033 (9.3%) | | | **0.7632**$_{1}$ | / |
| | | TrainMem | **150.0%** | 24.3% | | | **150.0%** | 27.9% | | | **150.0%** | / |
| | | TrainTime | 7m28s | 4m54s | | | **33m48s**$_{3}$ | 34m58s | | | 1h35m | / |
| | | Latency | 2.66ms | 2.82ms | | | 3.86ms | 3.02ms | | | 5.18ms | / |
| | AutoDim | AUC | 0.7504 (46.4%) | | | | 0.8036 (46.0%) | | | | 0.7595 (37.3%) | |
| | | TrainMem | 562.5% | | | | 562.5% | | | | 562.5% | |
| | | TrainTime | 1h6m | | | | 5h13m | | | | 2h27m | |
| | | Latency | 1.11ms | | | | 2.68ms | | | | 5.27ms | |
| | OptEmbed | AUC | 0.7484 (50.0%) | | | | 0.8019 (46.3%) | | | | 0.7610 (47.0%) | |
| | | TrainMem | 300.0% | | | | 300.0% | | | | 300.0% | |
| | | TrainTime | 16m17s | | | | 30m34s | | | | 1h57m | |
| | | Latency | 0.57ms | | | | 0.79ms | | | | 0.65ms | |
| Pruning | DeepLight | AUC | 0.7520 | **0.7554**$_{3}$ | **0.7579**$_{2}$ | **0.7526**$_{3}$ | 0.8030 | 0.8050 | **0.8067**$_{2}$ | **0.8019**$_{3}$ | 0.7536 | **0.7504**$_{1}$ |
| | | TrainMem | 306.3% | 306.3% | 306.3% | 306.3% | 306.3% | 306.3% | 306.3% | 306.3% | 306.3% | 306.3% |
| | | TrainTime | 3h32m | 3h45m | 5h25m | 8h10m | 3h11m | 8h41m | 15h38m | 15h41m | 31h25m | 27h50m |
| | | Latency | **0.59ms**$_{2}$ | **0.63ms**$_{2}$ | **0.62ms**$_{2}$ | **0.64ms**$_{2}$ | **0.93ms**$_{2}$ | **0.91ms**$_{2}$ | **0.91ms**$_{1}$ | **0.93ms**$_{3}$ | **0.56ms**$_{1}$ | **0.58ms**$_{1}$ |
| | AutoSrh | AUC | 0.7518 | 0.7520 | 0.7526 | 0.7507 | **0.8066**$_{3}$ | **0.8071**$_{1}$ | **0.8068**$_{1}$ | **0.8033**$_{2}$ | 0.7529 | 0.7215 |
| | | TrainMem | 306.3% | 306.3% | 306.3% | 306.3% | 306.3% | 306.3% | 306.3% | 306.3% | 306.3% | 306.3% |
| | | TrainTime | 22m36s | 28m43s | 29m55s | **32m39s**$_{2}$ | 2h9m | 2h2m | 2h6m | **2h9m**$_{3}$ | 4h21m | 4h43m |
| | | Latency | **0.56ms**$_{1}$ | **0.57ms**$_{1}$ | **0.57ms**$_{1}$ | **0.62ms**$_{1}$ | 1.43ms | 1.36ms | 1.49ms | 1.44ms | **0.57ms**$_{2}$ | **0.59ms**$_{2}$ |

they do not change the original training paradigm. DHE adopts novel MLP structures and performs well on Avazu dataset. Pruning methods use more memory to emphasize important features, regardless of memory budgets, and they perform well on Criteo dataset, especially when the memory budget is small. Dedup achieves near-optimal performance on all datasets, demonstrating the strength of similarity-based deduplication. Dimension reduction methods can also achieve good model AUC by capturing feature importance, but the results are mostly not comparable due to different compression ratios. Which methods are suitable for different datasets remains an open question, which we leave as future work.

Not all methods achieve better AUC with larger memory budgets. For TT-Rec and DHE, the dimension of matrix multiplication increases as the memory increases, making the optimization more difficult. For Dedup, within small memory, pooly-trained embeddings may be replaced by well-trained ones, thus improving model quality. For MGQE, larger memory only means the embeddings are split into more parts, with the centroids memory unchanged. For pruning, noisy redundant parameters may be removed as the memory decreases, leading to an increase in AUC.

*5.2.3 Training Memory.* **Static encoding methods (except for Dedup) and MDE use the least memory during training, only three times the inference memory budget considering the optimizer states.** Memory consumption during training is different from inference. Many methods require training uncompressed embeddings or other memory-intensive auxiliary modules. For the Adam optimizer, we also need to store the first- and the second-order momentum, making the memory usage at least three times that of the inference process. Training memory is described as a ratio of the baseline memory, independent of dataset size.

Static encoding methods (except for Dedup) and MDE have no extra structures, and the trained parameters are directly used for inference. They have a linear relationship between training memory and inference memory, where the exact multiple depends on the optimizer. AdaptEmb records feature frequency during training. Quantization only quantizes the embeddings, not the optimizer states, so its training memory is large. MGQE, Dedup, dimension reduction methods (except MDE) and pruning methods, require full-embedding training which is at least three times the baseline memory. Among them, AutoDim requires the most memory because it simultaneously trains all candidate dimensions.

*5.2.4 Training Time.* **Simple hash-based methods (including Dedup) and INT8/16 are fast to converge.** We employ the early stopping strategy and record the time for each method to converge, including all stages. Methods with top-3 least training time are highlighted in bold with an underlined ranking. Generally speaking, the larger the dataset, the longer it takes for DLRM to converge.

Dedup and INT8/16 are the fastest to converge. They do not change the training paradigm and involve negligible deduplication and (de)quantization overhead. CompoEmb and AdaptEmb are also fast to converge, requiring minor modifications to the training process. Other inter-feature compression methods either involve complex computations, or require more epochs to converge due to relaxed abstraction of embedding tables. These also result in large variance in their training times. Dimension reduction and pruning methods have longer training times due to the introduction of warm-up, search, retraining stages, and the alternative learning of model parameters and structural parameters.

There is not a clear relationship between the training time and the memory budget. On the one hand, more memory may lead to greater training complexity; on the other hand, less memory may make it harder to achieve convergence.

*5.2.5 Inference Latency.* **Dedup, ROBE, OptEmbed, quantization methods, and pruning methods have the lowest inference latency.** After training, the model checkpoints are saved for inference. The methods with top-3 least inference latency are highlighted in bold with an underlined ranking. We use the same batch size in training and inference. Criteo has greater latency than Avazu with more embeddings to compute. The batch size of Company is smaller than other datasets, so the latency is not comparable.

Dedup, ROBE, OptEmbed, quantization, and pruning all lookup the embeddings from only one table (or array), resulting in low inference latency. Dedup conducts similarity-based deduplication, with no need to consider field information; ROBE designs an array to share all embeddings. Except for Dedup and ROBE, inter-feature compression methods have to perform compression within fields, considering that features of the same field have similar semantics. TT-Rec and DHE have the largest inference latency due to time-consuming matrix multiplications. Quantization only incurs negligible dequantization process during inference. Sparse tensors in pruning may have fewer memory accesses with no additional overheads. MDE and AutoDim introduce additional matrix multiplications to align dimensions, thereby increasing inference latency.

If the time complexity is constant, the inference latency hardly changes with the memory budget. In contrast, TT-Rec and DHE perform more complex computations with larger memory, resulting in greater latency. However, when the memory is too small, more fields participate in compression, also leading to greater latency.

*5.2.6 Commercial Dataset.* **After analyzing the results on the commercial dataset Company, we find that the conclusions are consistent with those of the public datasets, despite some minor differences.** The training time variance is larger on Company, mainly because the larger Company dataset is more difficult for compression methods to train. Compression methods perform similarly on the three datasets, because 1) the public datasets are also collected from real recommendation scenarios; 2) our conclusions are robust enough to be generalized to larger datasets. Since we use compression ratios to study the performance, the absolute size of the embedding table has little impact on the conclusions.

*5.2.7 Discussion on Taxonomy.* **The current taxonomy is based on the compression paradigm, which determines the implementation.** For example, dynamic encoding records dynamic mappings, quantization adopts low-precision data types, and pruning stores embeddings in sparse formats. In experiments, methods of the same category have a certain degree of similarity, but there may be differences in some metrics due to different techniques used, such as simple hashing, complex computation, similarity-based deduplication, and VQ techniques in inter-feature compression, heuristics, policy gradient, DARTS, and one-shot-NAS techniques in intra-feature compression. Our analysis considers both paradigms and techniques, making the conclusions more comprehensive.

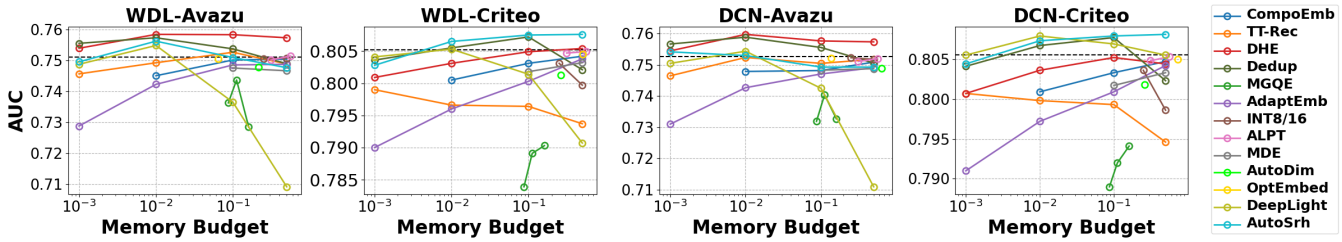


**Figure 4: AUC of WDL and DCN.**

## 5.3 Impact of Neural Network Model

For another two recommendation models WDL [15] and DCN [90], we plot the AUC of each compression method at each inference memory budget in Figure 4. Despite minor differences compared to DLRM, the ranking of the methods remains almost the same. The neural networks' memory consumption and processing time have minor differences and do not impact the conclusions.

From the experimental results, we can see that the optimization of the model and the selection of the compression method are orthogonal, as the compression methods are decoupled from the downstream neural network. Therefore, the conclusions we draw on DLRM can be applied to other models as well.

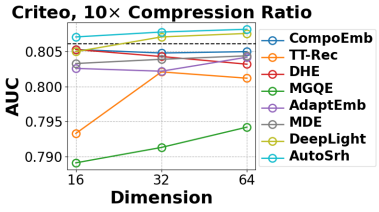


**Figure 5: AUC vs dimension.**

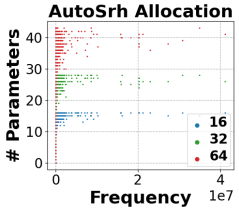


**Figure 6: Allocation.**

## 5.4 Impact of Dimension

In Section 5.2, we align the embedding dimension to the baseline. However, methods that capture feature frequency or importance generally prefer larger dimensions to allocate more memory for more important features [12, 46, 55, 62, 64, 76]. In this section, we enlarge the dimension to explore the potential of these methods.

Figure 5 shows the AUC for each compression method with dimension 16, 32, and 64. The inference memory budget is fixed at 10% of the baseline memory with dimension 16. In general, methods that adopt feature importance, including dynamic encoding, dimension reduction, and pruning, have a certain increase in AUC as the dimension increases. In contrast, static encoding methods mostly do not benefit from larger dimensions.

In Figure 6, we visualize the actual memory allocated for each feature in AutoSrh. Each point represents a feature: the x-axis is its frequency, and the y-axis is the number of assigned parameters. The allocated memory does not necessarily depend on frequency, as frequency is only one factor of feature importance. As the dimension becomes larger, AutoSrh allocates more memory to important features, explaining the effect of dimension increase.

When the dimension is enlarged, MGQE and pruning methods increase the training memory linearly despite better AUC. Therefore, choosing the right dimensions requires a careful trade-off between model quality and training overhead.

## 5.5 Performance on Retrieval-augmented LLM

In this section, we apply compression methods to generated embeddings in a retrieval-augmented LLM. The entire generated embeddings are produced by neural networks and present only at inference, different from parametric embeddings in DLRM that are trainable parameters and present throughout training. Therefore, compression methods that involve the training process, such as AutoML-based methods, are not suitable for generated embeddings.

We select applicable compression methods or their variants for evaluation, including **TT** (tensor-train decomposition), **Dedup**, **PQ**, **MagPQ** (PQ within embedding groups that are split by magnitude), **INT8/16**, **SVD** (dimension reduction), **MagSVD** (SVD within embedding groups that are split by magnitude), **Pruning** (pruning values of low magnitude). MagPQ and MagSVD are variants of MGQE and MDE respectively, replacing missing frequency information with embeddings' L2-norms.

We experiment with RAG [52] which uses DPR [42] for retrieval and BART [51] for generation. We experiment on the open-domain QA dataset Natural Questions (NQ) [47], with cleaned Wikipedia articles (21 million) as the search corpus, following previous research [24, 42, 52, 77, 110]. We retrieve top 10 documents for each query. The embedding dimension is 768, which is much larger than DLRM. Since the entire embeddings are generated after training, we apply compression methods at the inference stage. Each experiment is conducted on an Nvidia A100 40GB GPU card. Table 6 presents three metrics: Exact Match (EM) score, compression time, and batched-decompression latency with a batch size of 1024.

*5.5.1 Ability of Compression.* **TT, Dedup, and Pruning can reach all compression ratios.** Similar to the DLRM experiment, we compress under four memory budgets. TT essentially performs two SVDs with moderate dimensions to enable a wide range of compression ratios. In contrast, (Mag)SVD cannot support small memory budgets, because their memory scales linearly with the corpus cardinality. Pure SVD cannot support large memory budgets, because it is difficult to decompose with large intermediate dimensions. Dedup and Pruning have adjustable thresholds, making them applicable for almost any memory budget. (Mag)PQ cannot support large compression ratios due to complex computations in clustering. INT8/16 only support several fixed compression ratios.

*5.5.2 Embedding Quality.* **INT8/16, (Mag)SVD, Pruning achieve best EM scores under large memory budgets, while (Mag)PQ achieve best EM scores under small memory budgets.** INT8/16 and PQ have been implemented in the well-known embedding search library Faiss [39] due to their effectiveness. MagPQ and MagSVD show comparable performance to PQ and SVD with less memory, thanks to magnitude-aware compression. TT uses two SVDs, which greatly degrades performance. Dedup's block-wise deduplication may not perform well on this retrieval-related task.

*5.5.3 Compression Time.* **INT8/16 has the smallest compression time, followed by Pruning.** INT8/16 requires almost no computation. Pruning uses an efficient binary search algorithm to determine the threshold. Dedup uses L2LSH for deduplication which is only efficient under large memory budgets when the block size is large. (Mag)SVD is only efficient under small memory budgets when the intermediate dimensions are small. TT and (Mag)PQ are computationally expensive, resulting in long compression times.

*5.5.4 Batched-decompression Latency.* **INT8/16 has the smallest latency, followed by Pruning, Dedup, and (Mag)PQ.** The decompression of INT8/16 and CSR-format Pruning is fast with little overhead. When the memory budget is small, Pruning uses COO format, which is very slow for high-dimensional embeddings. Dedup and (Mag)PQ look up embeddings from tensor blocks or centroids, with no computation overhead. TT and (Mag)SVD adopt matrix multiplication, leading to large decompression latency.

## 5.6 Further Discussions and Future Directions

*5.6.1 Challenges.* Currently, all compression methods have certain drawbacks, requiring users to carefully trade-off based on practical needs. For DLRM, there is no single method that performs well in all metrics. For retrieval-augmented LLM, research on embedding compression is still in its early stages, with only a few specialized methods available. Therefore, more comprehensive and advanced methods are expected in both fields.

On the other hand, the relationship between datasets and compression methods has not been studied. Our experiments show that different methods perform better on different datasets in DLRM, but it is unclear why. It is currently difficult to determine a proper method for a given dataset without actual experiments.

*5.6.2 Future Directions.* A straightforward idea is to combine the advantages of different compression methods in DLRM. For dynamic encoding, state-of-the-art static encoding and pruning methods can be integrated to achieve better model quality in online scenarios. Quantization can be used as a plug-in module, contributing a fixed compression ratio with very low cost; another possible improvement is to assign data types with different bits to different features, borrowing ideas of capturing feature importance. Dimension reduction and pruning require pre-training, where static encoding can be applied to avoid large training memory.

Currently, embedding compression for retrieval tasks mainly uses quantization or PQ [39]. To the best of our knowledge, we are the first to study other embedding compression methods for retrieval. We anticipate that compression methods specifically designed for retrieval will emerge in the future and can be combined with embedding search to further improve performance. Inspired by data skewness in DLRM, we are also curious whether retrieval datasets also have such properties, which we leave as future work.

Moreover, studying the impact of recommendation data distribution on compression methods is also a promising direction. At present, for a given dataset we can only determine compression methods experimentally. A deeper understanding of data will not only help in the selection of compression methods, but also inspire the development of more advanced methods.

**Table 6: Overall performance on RAG.**

| Methods | Metrics | 50% | 10% | 1% | 0.1% |
|---|---|---|---|---|---|
| Full | EM | 41.14 (100.00%) | | | |
| TT | EM | 22.02 | 11.47 | 6.54 | 4.38 |
| | Time | 1h35m | 42m46s | 18m11s | 13m22s |
| | Latency | 39.54s | 9.14s | 2.21s | 482.77ms |
| Dedup | EM | 25.93 | 15.57 | 5.51 | 4.68 |
| | Time | 11m3s | 11m29s | 33m36s | 2h34m |
| | Latency | 4.14ms | 3.55ms | 3.00ms | 2.63ms |
| PQ | EM | 35.32 (3.14%) | | 24.57 (1.58%) | |
| | Time | 43m21s | | 40m33s | |
| | Latency | 8.29ms | | 9.09ms | |
| MagPQ | EM | 35.29 (2.99%) | | 33.99 (1.57%) | |
| | Time | 1h9m | | 40m30s | |
| | Latency | 11.52ms | | 10.71ms | |
| INT8/16 | EM | 41.02 | 38.06 (25.00%) | | |
| | Time | 4m60s | 5m3s | | |
| | Latency | 0.0513ms | 0.0472ms | | |
| SVD | EM | / | 31.02 | 3.88 | / |
| | Time | / | 25m48s | 10m4s | / |
| | Latency | / | 16.86ms | 11.58ms | / |
| MagSVD | EM | 41.11 | 30.89 | 4.04 | / |
| | Time | 50m6s | 17m16s | 13m0s | / |
| | Latency | 65.83ms | 35.56ms | 25.57ms | / |
| Pruning | EM | 37.29 | 17.15 | 4.24 | 4.04 |
| | Time | 14m37s | 11m2s | 12m13s | 13m25s |
| | Latency | 3.13ms | 3.42ms | 3.23ms | 1.33s |

## 6 CONCLUSION

In this paper, we surveyed existing embedding compression methods and proposed a new taxonomy. We modularized the compression pipeline and implemented a unified evaluation framework. We conducted a comprehensive experimental evaluation to analyze the performance of each method under different memory budgets. The experimental results reveal the pros and cons of each method, provide suggestions for method selection in different situations, and shed light on promising research directions.

## ACKNOWLEDGMENTS

This work is supported by National Key R&D Program of China (2022ZD0116315), National Natural Science Foundation of China (U22B2037 and U23B2048), and PKU-Tencent joint research Lab. Yingxia Shao's work is supported by the National Natural Science Foundation of China (Nos. 62272054, 62192784), Beijing Nova Program (No. 20230484319), the Fundamental Research Funds for the Central Universities (No. 2023PY11) and Xiaomi Young Talents Program. Bin Cui and Xupeng Miao are the co-corresponding authors.

# REFERENCES


[1] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. 2017. Structured Pruning of Deep Convolutional Neural Networks. *ACM Journal on Emerging Technologies in Computing Systems* 13, 3 (2017), 32:1–32:18.

[2] Ron Banner, Yury Nahshan, and Daniel Soudry. 2019. Post training 4-bit quantization of convolutional networks for rapid-deployment. In *Advances in Neural Information Processing Systems 32 (NeurIPS)*.

[3] Nitin Bansal, Xiaohan Chen, and Zhangyang Wang. 2018. Can We Gain More from Orthogonality Regularizations in Training Deep Networks?. In *Advances in Neural Information Processing Systems 31 (NeurIPS)*.

[4] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and Laurent Sifre. 2022. Improving Language Models by Retrieving from Trillions of Tokens. In *Proceedings of the 39th International Conference on Machine Learning (ICML)*.

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems 33 (NeurIPS)*.

[6] Patrick H. Chen, Si Si, Yang Li, Ciprian Chelba, and Cho-Jui Hsieh. 2018. GroupReduce: Block-Wise Low-Rank Approximation for Neural Language Model Shrinking. In *Advances in Neural Information Processing Systems 31 (NeurIPS)*.

[7] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighborhood Search. In *Advances in Neural Information Processing Systems 34 (NeurIPS)*.

[8] Tianyi Chen, Jun Gao, Hedui Chen, and Yaofeng Tu. 2023. LOGER: A Learned Optimizer towards Generating Efficient and Robust Query Execution Plans. *Proceedings of the VLDB Endowment* 16, 7 (2023), 1777–1789.

[9] Ting Chen, Lala Li, and Yizhou Sun. 2020. Differentiable Product Quantization for End-to-End Embedding Compression. In *Proceedings of the 37th International Conference on Machine Learning (ICML)*.

[10] Ting Chen, Martin Renqiang Min, and Yizhou Sun. 2018. Learning K-way D-dimensional Discrete Codes for Compact Embedding Representations. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*.

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[12] Tong Chen, Hongzhi Yin, Yujia Zheng, Zi Huang, Yang Wang, and Meng Wang. 2021. Learning Elastic Embeddings for Customizing On-Device Recommenders. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining (KDD)*.

[13] Xubin Chen, Ning Zheng, Shukun Xu, Yifan Qiao, Yang Liu, Jiangpeng Li, and Tong Zhang. 2021. KallaxDB: A Table-less Hash-based Key-Value Store on Storage Hardware with Built-in Transparent Compression. In *Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN)*.

[14] Yizhou Chen, Guangda Huzhang, Anxiang Zeng, Qingtao Yu, Hui Sun, Heng-Yi Li, Jingyi Li, Yabo Ni, Han Yu, and Zhiming Zhou. 2023. Clustered Embedding Learning for Recommender Systems. In *Proceedings of the Web Conference (WWW)*.

[15] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. 2016. Wide & Deep Learning for Recommender Systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems (DLRS@RecSys)*.

[16] Weiyu Cheng, Yanyan Shen, and Linpeng Huang. 2020. Differentiable Neural Input Search for Recommender Systems. *CoRR* abs/2006.04466 (2020).

[17] DeepRec. 2021. Adaptive Embedding. https://github.com/alibaba/DeepRec/blob/main/docs/docs_en/Adaptive-Embedding.md.

[18] Wei Deng, Junwei Pan, Tian Zhou, Deguang Kong, Aaron Flores, and Guang Lin. 2021. DeepLight: Deep Lightweight Feature Interactions for Accelerating CTR Predictions in Ad Serving. In *Proceedings of the 14th ACM International Conference on Web Search and Data Mining (WSDM)*.

[19] Aditya Desai, Li Chou, and Anshumali Shrivastava. 2022. Random Offset Block Embedding (ROBE) for compressed embedding tables in deep learning recommendation systems. In *Proceedings of Machine Learning and Systems (MLSys)*.

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*.

[21] Muhammad Ebraheem, Saravanan Thirumuruganathan, Shafiq R. Joty, Mourad Ouzzani, and Nan Tang. 2018. Distributed Representations of Tuples for Entity Resolution. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1454–1467.

[22] Steven K. Esser, Jeffrey L. McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S. Modha. 2020. Learned Step Size quantization. In *8th International Conference on Learning Representations (ICLR)*.

[23] Jonathan Frankle and Michael Carbin. 2019. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. In *7th International Conference on Learning Representations (ICLR)*.

[24] Luyu Gao and Jamie Callan. 2021. Condenser: a Pre-training Architecture for Dense Retrieval. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[25] Antonio A. Ginart, Maxim Naumov, Dheevatsa Mudigere, Jiyan Yang, and James Zou. 2021. Mixed Dimension Embeddings with Application to Memory-Efficient Recommendation Systems. In *IEEE International Symposium on Information Theory (ISIT)*.

[26] Hui Guan, Andrey Malevich, Jiyan Yang, Jongsoo Park, and Hector Yuen. 2019. Post-Training 4-bit Quantization on Embedding Tables. In *Workshop on Systems for ML at NeurIPS*.

[27] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, Zhenshan Cao, Yanliang Qiao, Ting Wang, Bo Tang, and Charles Xie. 2022. Manu: A Cloud Native Vector Database Management System. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3548–3561.

[28] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*.

[29] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. 2020. Retrieval Augmented Language Model Pre-Training. In *Proceedings of the 37th International Conference on Machine Learning (ICML)*.

[30] Peng Han, Silin Zhou, Jie Yu, Zichen Xu, Lisi Chen, and Shuo Shang. 2023. Personalized Re-ranking for Recommendation with Mask Pretraining. *Data Science and Engineering* 8, 4 (2023), 357–367.

[31] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning both Weights and Connections for Efficient Neural Network. In *Advances in Neural Information Processing Systems 28 (NeurIPS)*.

[32] Teng-Yue Han, Pengfei Wang, and Shaozhang Niu. 2023. Multimodal Interactive Network for Sequential Recommendation. *Journal of Computer Science and Technology* 38, 4 (2023), 911–926.

[33] Oleksii Hrinchuk, Valentin Khrulkov, Leyla Mirvakhabova, Elena D. Orlova, and Ivan V. Oseledets. 2020. Tensorized Embedding Layers. In *Findings of the Association for Computational Linguistics (EMNLP)*.

[34] Ruihong Huang, Shaoxu Song, Yunsu Lee, Jungho Park, Soo-Hyung Kim, and Sungmin Yi. 2020. Effective and Efficient Retrieval of Structured Entities. *Proceedings of the VLDB Endowment* 13, 6 (2020), 826–839.

[35] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2017. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. *The Journal of Machine Learning Research* 18 (2017), 187:1–187:30.

[36] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the 30th Annual ACM Symposium on the Theory of Computing (STOC)*.

[37] Hervé Jégou, 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.

[38] Manas R. Joglekar, Cong Li, Mei Chen, Taibai Xu, Xiaoming Wang, Jay K. Adams, Pranav Khaitan, Jiahui Liu, and Quoc V. Le. 2020. Neural Input Search for Large Scale Recommendation Models. In *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery & Data Mining (KDD)*.

[39] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[40] Wang-Cheng Kang, Derek Zhiyuan Cheng, Ting Chen, Xinyang Yi, Dong Lin, Lichan Hong, and Ed H. Chi. 2020. Learning Multi-granular Quantized Embeddings for Large-Vocab Categorical Features in Recommender Systems. In *Companion Proceedings of The Web Conference*.

[41] Wang-Cheng Kang, Derek Zhiyuan Cheng, Tiansheng Yao, Xinyang Yi, Ting Chen, Lichan Hong, and Ed H. Chi. 2021. Learning to Embed Categorical Features without Embedding Tables for Recommendation. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining (KDD)*.

[42] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick S. H. Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[43] Hyeonji Kim, Byeong-Hoon So, Wook-Shin Han, and Hongrae Lee. 2020. Natural language to SQL: Where are we today? *Proceedings of the VLDB Endowment* 13, 10 (2020), 1737–1750.

[44] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations (ICLR)*.

[45] Adrian Kochsiek and Rainer Gemulla. 2021. Parallel Training of Knowledge Graph Embedding Models: A Comparison of Techniques. *Proceedings of the VLDB Endowment* 15, 3 (2021), 633–645.

[46] Shuming Kong, Weiyu Cheng, Yanyan Shen, and Linpeng Huang. 2023. AutoSrh: An Embedding Dimensionality Search Framework for Tabular Data Prediction. *IEEE Transactions on Knowledge and Data Engineering* 35, 7 (2023), 6673–6686.

[47] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur P. Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. 2019. Natural Questions: a Benchmark for Question Answering Research. *Transactions of the Association for Computational Linguistics* 7 (2019), 452–466.

[48] Suyong Kwon, Woohwan Jung, and Kyuseok Shim. 2022. Cardinality Estimation of Approximate Substring Queries using Deep Learning. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3145–3157.

[49] Criteo Labs. 2014. Kaggle display advertising challenge dataset. https://labs.criteo.com/2014/02/kaggle-display-advertising-challenge-dataset.

[50] Seulki Lee and Shahriar Nirjon. 2020. Fast and scalable in-memory deep multi-task learning via neural weight virtualization. In *Proceedings of the 18th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*.

[51] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*.

[52] Patrick S. H. Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems 33 (NeurIPS)*.

[53] Hao Li, Soham De, Zheng Xu, Christoph Studer, Hanan Samet, and Tom Goldstein. 2017. Training Quantized Nets: A Deeper Understanding. In *Advances in Neural Information Processing Systems 30 (NeurIPS)*.

[54] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2017. Pruning Filters for Efficient ConvNets. In *5th International Conference on Learning Representations (ICLR)*.

[55] Shiwei Li, Huifeng Guo, Lu Hou, Wei Zhang, Xing Tang, Ruiming Tang, Rui Zhang, and Ruixuan Li. 2023. Adaptive Low-Precision Training for Embeddings in Click-Through Rate Prediction. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*.

[56] Yang Li, Tong Chen, Peng-Fei Zhang, and Hongzhi Yin. 2021. Lightweight Self-Attentive Sequential Recommendation. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management (CIKM)*.

[57] Defu Lian, Haoyu Wang, Zheng Liu, Jianxun Lian, Enhong Chen, and Xing Xie. 2020. LightRec: A Memory and Search-Efficient Recommender System. In *Proceedings of the Web Conference (WWW)*.

[58] Xiangru Lian, Binhang Yuan, Xuefeng Zhu, Yulong Wang, Yongjun He, Honghuan Wu, Lei Sun, Haodong Lyu, Chengjun Liu, Xing Dong, Yiqiao Liao, Mingnan Luo, Congfei Zhang, Jingru Xie, Haonan Li, Lei Chen, Renjie Huang, Jianying Lin, Chengchun Shu, Xuezhong Qiu, Zhishan Liu, Dongying Kong, Lei Yuan, Hai Yu, Sen Yang, Ce Zhang, and Ji Liu. 2022. Persia: An Open, Hybrid System Scaling Deep Learning-based Recommenders up to 100 Trillion Parameters. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery & Data Mining (KDD)*.

[59] Weilin Lin, Xiangyu Zhao, Yejing Wang, Tong Xu, and Xian Wu. 2022. AdaFS: Adaptive Feature Selection in Deep Recommender System. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery & Data Mining (KDD)*.

[60] Haochen Liu, Xiangyu Zhao, Chong Wang, Xiaobing Liu, and Jiliang Tang. 2020. Automated Embedding Size Search in Deep Recommender Systems. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*.

[61] Jie Liu, Wenqian Dong, Dong Li, and Qingqing Zhou. 2021. Fauce: Fast and Accurate Deep Ensembles with Uncertainty for Cardinality Estimation. *Proceedings of the VLDB Endowment* 14, 11 (2021), 1950–1963.

[62] Siyi Liu, Chen Gao, Yihong Chen, Depeng Jin, and Yong Li. 2021. Learnable Embedding sizes for Recommender Systems. In *9th International Conference on Learning Representations (ICLR)*.

[63] Fuyuan Lyu, Xing Tang, Dugang Liu, Liang Chen, Xiuqiang He, and Xue Liu. 2023. Optimizing Feature Set for Click-Through Rate Prediction. In *Proceedings of the Web Conference (WWW)*.

[64] Fuyuan Lyu, Xing Tang, Hong Zhu, Huifeng Guo, Yingxue Zhang, Ruiming Tang, and Xue Liu. 2022. OptEmbed: Learning Optimal Embedding Table for Click-through Rate Prediction. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management (CIKM)*.

[65] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems* 45 (2014), 61–68.

[66] Yury A. Malkov and Dmitry A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 4 (2020), 824–836.

[67] Xupeng Miao, Xiaonan Nie, Hailin Zhang, Tong Zhao, and Bin Cui. 2023. Hetu: a highly efficient automatic parallel distributed deep learning system. *Science China Information Sciences* 66, 1 (2023).

[68] Xupeng Miao, Yining Shi, Hailin Zhang, Xin Zhang, Xiaonan Nie, Zhi Yang, and Bin Cui. 2022. HET-GMP: A Graph-based System Approach to Scaling Large Embedding Model Training. In *Proceedings of the International Conference on Management of Data (SIGMOD)*.

[69] Xupeng Miao, Hailin Zhang, Yining Shi, Xiaonan Nie, Zhi Yang, Yangyu Tao, and Bin Cui. 2022. HET: Scaling out Huge Embedding Model Training via Cache-enabled Distributed Framework. *Proceedings of the VLDB Endowment* 15, 2 (2022), 312–320.

[70] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed Precision Training. In *6th International Conference on Learning Representations (ICLR)*.

[71] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie Amy Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, K. R. Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. 2022. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*.

[72] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* abs/1906.00091 (2019).

[73] Tri Nguyen, Mir Rosenberg, Xia Song, Jianfeng Gao, Saurabh Tiwary, Rangan Majumder, and Li Deng. 2016. MS MARCO: A Human Generated MAchine Reading COmprehension Dataset. In *Proceedings of the Workshop on Cognitive Computation at NeurIPS (CoCo@NeurIPS)*.

[74] Niketan Pansare, Jay Katukuri, Aditya Arora, Frank Cipollone, Riyaaz Shaik, Noyan Tokgozoglu, and Chandru Venkataraman. 2022. Learning Compressed Embeddings for On-Device Inference. In *Proceedings of Machine Learning and Systems (MLSys)*.

[75] NVIDIA AI platform. 2020. MLPerf Benchmark. https://mlperf.org.

[76] Liang Qu, Yonghong Ye, Ningzhi Tang, Lixin Zhang, Yuhui Shi, and Hongzhi Yin. 2022. Single-shot Embedding Dimension Search in Recommender System. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*.

[77] Yingqi Qu, Yuchen Ding, Jing Liu, Kai Liu, Ruiyang Ren, Wayne Xin Zhao, Daxiang Dong, Hua Wu, and Haifeng Wang. 2021. RocketQA: An Optimized Training Approach to Dense Passage Retrieval for Open-Domain Question Answering. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*.

[78] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *The Journal of Machine Learning Research* 21 (2020), 140:1–140:67.

[79] Jie Ren, Minjia Zhang, and Dong Li. 2020. HM-ANN: Efficient Billion-Point Nearest Neighbor Search on Heterogeneous Memory. In *Advances in Neural Information Processing Systems 33 (NeurIPS)*.

[80] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A Space-Efficient Key-Value Storage Engine For Semi-Sorted Data. *Proceedings of the VLDB Endowment* 10, 13 (2017), 2037–2048.

[81] Geet Sethi, Bilge Acun, Niket Agarwal, Christos Kozyrakis, Caroline Trippel, and Carole-Jean Wu. 2022. RecShard: statistical feature-based memory optimization for industry-scale neural recommendation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[82] Pengyang Shao, Le Wu, Lei Chen, Kun Zhang, and Meng Wang. 2022. FairCF: fairness-aware collaborative filtering. *Science China Information Sciences* 65, 12 (2022).

[83] Jiayi Shen, Haotao Wang, Shupeng Gui, Jianchao Tan, Zhangyang Wang, and Ji Liu. 2021. UMEC: Unified model and embedding compression for efficient recommendation systems. In *9th International Conference on Learning Representations (ICLR)*.

[84] Hao-Jun Michael Shi, Dheevatsa Mudigere, Maxim Naumov, and Jiyan Yang. 2020. Compositional Embeddings Using Complementary Partitions for Memory-Efficient Recommendation Systems. In *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery & Data Mining (KDD)*.

[85] Raphael Shu and Hideki Nakayama. 2018. Compressing Word Embeddings via Deep Compositional Code Learning. In *6th International Conference on Learning Representations (ICLR)*.

[86] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnaswamy, and Rohan Kadekodi. 2019. Rand-NSG: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *Advances in Neural Information Processing Systems 32 (NeurIPS)*.

[87] Dan Svenstrup, Jonas Meinertz Hansen, and Ole Winther. 2017. Hash Embeddings for Efficient Word Representations. In *Advances in Neural Information Processing Systems 30 (NeurIPS)*.

[88] Manasi Vartak, Joana M. F. da Trindade, Samuel Madden, and Matei Zaharia. 2018. MISTIQUE: A System to Store and Query Model Intermediates for Model Diagnosis. In *Proceedings of the International Conference on Management of Data (SIGMOD)*.

[89] Qinyong Wang, Hongzhi Yin, Tong Chen, Zi Huang, Hao Wang, Yanchang Zhao, and Nguyen Quoc Viet Hung. 2020. Next Point-of-Interest Recommendation on Resource-Constrained Mobile Devices. In *Proceedings of the Web Conference (WWW)*.

[90] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. 2017. Deep & Cross Network for Ad Click Predictions. In *Proceedings of the ADKDD'17*.

[91] Steve Wang and Will Cukierski. 2014. Avazu Click-Through Rate Prediction. https://kaggle.com/competitions/avazu-ctr-prediction.

[92] Yejing Wang, Xiangyu Zhao, Tong Xu, and Xian Wu. 2022. AutoField: Automating Feature Selection in Deep Recommender Systems. In *Proceedings of the Web Conference (WWW)*.

[93] Zehuan Wang, Yingcan Wei, Minseok Lee, Matthias Langer, Fan Yu, Jie Liu, Shijie Liu, Daniel G. Abel, Xu Guo, Jianbing Dong, Ji Shi, and Kunlun Li. 2022. Merlin HugeCTR: GPU-accelerated Recommender System Training and Inference. In *Proceedings of the 16th ACM Conference on Recommender Systems (RecSys)*.

[94] Zhikun Wei, Xin Wang, and Wenwu Zhu. 2021. AutoIAS: Automatic Integrated Architecture Searcher for Click-Trough Rate Prediction. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management (CIKM)*.

[95] Kilian Q. Weinberger, Anirban Dasgupta, John Langford, Alexander J. Smola, and Josh Attenberg. 2009. Feature hashing for large scale multitask learning. In *Proceedings of the 26th International Conference on Machine Learning (ICML)*.

[96] Carole-Jean Wu, Robin Burke, Ed H. Chi, Joseph A. Konstan, Julian J. McAuley, Yves Raimond, and Hao Zhang. 2020. Developing a Recommendation Benchmark for MLPerf Training and Inference. *CoRR* abs/2003.07336 (2020).

[97] Zhijing Wu, Yiqun Liu, Jiaxin Mao, Min Zhang, and Shaoping Ma. 2022. Leveraging Document-Level and Query-Level Passage Cumulative Gain for Document Ranking. *Journal of Computer Science and Technology* 37, 4 (2022), 814–838.

[98] Tesi Xiao, Xia Xiao, Ming Chen, and Youlong Chen. 2022. Field-wise Embedding Size Search via Structural Hard Auxiliary Mask Pruning for Click-Through Rate Prediction. In *Proceedings of the Workshop on Deep Learning for Search and Recommendation (DL4SR) at CIKM*.

[99] Minhui Xie, Kai Ren, Youyou Lu, Guangxu Yang, Qingxing Xu, Bihai Wu, Jiazhen Lin, Hongbo Ao, Wanhong Xu, and Jiwu Shu. 2020. Kraken: memory-efficient continual learning for large-scale real-time recommendations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[100] Lee Xiong, Chenyan Xiong, Ye Li, Kwok-Fung Tang, Jialin Liu, Paul N. Bennett, Junaid Ahmed, and Arnold Overwijk. 2021. Approximate Nearest Neighbor Negative Contrastive Learning for Dense Text Retrieval. In *9th International Conference on Learning Representations (ICLR)*.

[101] Zhiqiang Xu, Dong Li, Weijie Zhao, Xing Shen, Tianbo Huang, Xiaoyun Li, and Ping Li. 2021. Agile and Accurate CTR Prediction Model Training for Massive-Scale Online Advertising Systems. In *Proceedings of the International Conference on Management of Data (SIGMOD)*.

[102] Bencheng Yan, Pengjie Wang, Jinquan Liu, Wei Lin, Kuang-Chih Lee, Jian Xu, and Bo Zheng. 2021. Binary Code based Hash Embedding for Web-scale Applications. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management (CIKM)*.

[103] Bencheng Yan, Pengjie Wang, Kai Zhang, Wei Lin, Kuang-Chih Lee, Jian Xu, and Bo Zheng. 2021. Learning Effective and Efficient Embedding via an Adaptively-Masked Twins-based Layer. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management (CIKM)*.

[104] Jie Amy Yang, Jianyu Huang, Jongsoo Park, Ping Tak Peter Tang, and Andrew Tulloch. 2020. Mixed-Precision Embedding Using a Cache. *CoRR* abs/2010.11305 (2020).

[105] Renchi Yang, Jieming Shi, Xiaokui Xiao, Yin Yang, Juncheng Liu, and Sourav S. Bhowmick. 2020. Scaling Attributed Network Embedding to Massive Graphs. *Proceedings of the VLDB Endowment* 14, 1 (2020), 37–49.

[106] Yao Yao, Bin Liu, Haoxun He, Dakui Sheng, Ke Wang, Li Xiao, and Huanhuan Cao. 2023. i-Razor: A Differentiable Neural Input Razor for Feature Selection and Dimension Search in DNN-Based Recommender Systems. *IEEE Transactions on Knowledge & Data Engineering* 01 (2023), 1–14.

[107] Chunxing Yin, Bilge Acun, Carole-Jean Wu, and Xing Liu. 2021. TT-Rec: Tensor Train Compression for Deep Learning Recommendation Models. In *Proceedings of Machine Learning and Systems (MLSys)*.

[108] Zhiyang Yuan, Wenguang Zheng, Peilin Yang, Qingbo Hao, and Yingyuan Xiao. 2023. Evolving Interest with Feature Co-action Network for CTR Prediction. *Data Science and Engineering* 8, 4 (2023), 344–356.

[109] Caojin Zhang, Yicun Liu, Yuanpu Xie, Sofia Ira Ktena, Alykhan Tejani, Akshay Gupta, Pranay Kumar Myana, Deepak Dilipkumar, Suvadip Paul, Ikuhiro Ihara, Prasang Upadhyaya, Ferenc Huszar, and Wenzhe Shi. 2020. Model Size Reduction Using Frequency Based Double Hashing for Recommender Systems. In *Proceedings of the 14th ACM Conference on Recommender Systems (RecSys)*.

[110] Hailin Zhang, Yujing Wang, Qi Chen, Ruiheng Chang, Ting Zhang, Ziming Miao, Yingyan Hou, Yang Ding, Xupeng Miao, Haonan Wang, Bochen Pang, Yuefeng Zhan, Hao Sun, Weiwei Deng, Qi Zhang, Fan Yang, Xing Xie, Mao Yang, and Bin Cui. 2023. Model-enhanced Vector Index. *CoRR* abs/2309.13335 (2023).

[111] Jia-Dong Zhang and Chi-Yin Chow. 2015. GeoSoCa: Exploiting Geographical, Social and Categorical Correlations for Point-of-Interest Recommendations. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*.

[112] Jian Zhang, Jiyan Yang, and Hector Yuen. 2018. Training with low-precision embedding tables. In *Workshop on Systems for ML at NeurIPS*.

[113] Yuanxing Zhang, Langshi Chen, Siran Yang, Man Yuan, Huimin Yi, Jie Zhang, Jiamang Wang, Jianbo Dong, Yunlong Xu, Yue Song, Yong Li, Di Zhang, Wei Lin, Lin Qu, and Bo Zheng. 2022. PICASSO: Unleashing the Potential of GPU-centric Training for Wide-and-deep Recommender Systems. In *38th IEEE International Conference on Data Engineering (ICDE)*.

[114] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. 2020. Distributed Hierarchical GPU Parameter Server for Massive Scale Deep Learning Ads Systems. In *Proceedings of Machine Learning and Systems (MLSys)*.

[115] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. 2019. AIBox: CTR Prediction Model Training on a Single Node. In *Proceedings of the 28th ACM International Conference on Information & Knowledge Management (CIKM)*.

[116] Xiangyu Zhao, Haochen Liu, Wenqi Fan, Hui Liu, Jiliang Tang, Chong Wang, Ming Chen, Xudong Zheng, Xiaobing Liu, and Xiwang Yang. 2021. AutoEmb: Automated Embedding Dimensionality Search in Streaming Recommendations. In *IEEE International Conference on Data Mining (ICDM)*.

[117] Xiangyu Zhao, Haochen Liu, Hui Liu, Jiliang Tang, Weiwei Guo, Jun Shi, Sida Wang, Huiji Gao, and Bo Long. 2021. AutoDim: Field-aware Embedding Dimension Searchin Recommender Systems. In *Proceedings of the Web Conference (WWW)*.

[118] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. 2022. QueryFormer: A Tree Transformer Model for Query Plan Representation. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1658–1670.

[119] Guorui Zhou, Xiaoqiang Zhu, Chengru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. 2018. Deep Interest Network for Click-Through Rate Prediction. In *Proceedings of the 24th ACM SIGKDD Conference on Knowledge Discovery & Data Mining (KDD)*.

[120] Lixi Zhou, Jiaqing Chen, Amitabh Das, Hong Min, Lei Yu, Ming Zhao, and Jia Zou. 2022. Serving Deep Learning Models with Deduplication from Relational Databases. *Proceedings of the VLDB Endowment* 15, 10 (2022), 2230–2243.

[121] 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 (CIKM)*.