TT-REC: TENSOR TRAIN COMPRESSION FOR DEEP LEARNING RECOMMENDATION MODEL EMBEDDINGS

Chunxing Yin [1 2] **Bilge Acun** [1] **Xing Liu** [1] **Carole-Jean Wu** [1]

ABSTRACT

The memory capacity of embedding tables in deep learning recommendation models (DLRMs) is increasing dramatically from tens of GBs to TBs across the industry. Given the fast growth in DLRMs, novel solutions are urgently needed in order to enable DLRM innovations. At the same time, this must be done in a fast and efficient way without having to exponentially increase infrastructure capacity demands. In this paper, we demonstrate the promising potential of Tensor Train decomposition for DLRMs (TT-Rec), an important yet under-investigated context. We design and implement optimized kernels (TT-EmbeddingBag) to evaluate the proposed TT-Rec design. TT-EmbeddingBag is 3× faster than the SOTA TT implementation. The performance of TT-Rec is further optimized with the batched matrix multiplication and caching strategies for embedding vector lookup operations. In addition, we present mathematically and empirically the effect of weight initialization distribution on DLRM accuracy and propose to initialize the tensor cores of TT-Rec following the sampled Gaussian distribution. We evaluate TT-Rec across three important design space dimensions—memory capacity, accuracy, and timing performance—by training MLPerf-DLRM with Criteo's Kaggle and Terabyte data sets. TT-Rec compresses the model size by 4× to 221× for Kaggle, with 0.03% to 0.3% loss of accuracy correspondingly. For Terabyte, our approach achieves 112× model size reduction which comes with no accuracy loss nor training time overhead as compared to the uncompressed baseline.

Our code is available on Github at facebookresearch/FBTT-Embedding.

1 INTRODUCTION

Deep neural networks (DNNs) are witnessing an unprecedented growth in all dimensions: data, model complexity, and the cost of infrastructure required for model training and deployment. For instance, at Facebook, the amount of data used in machine learning (ML) tripled in one year (2019–20), which led to an eight-fold increase in the amount of computation required for training (Hazelwood, 2020). Similarly, the number of parameters in state-of-the-art language models have increased exponentially, currently at over 175 billion parameters in OpenAI's GPT-3 (Brown et al., 2020). In response, there is considerable interest to design domain-specific accelerators and at-scale infrastructures (Acun et al., 2021; Alibaba, 2019; Amazon, 2019; Chung et al., 2018; Fowers et al., 2018; Hazelwood et al., 2018; Jouppi et al., 2017; Lee & Rao, 2019; Mattson et al., 2020a; NVIDIA, 2020a; Ovtcharov et al., 2015; Reddi et al., 2020). To sustain the fast cadence of ML innovations, we must achieve orders-of-magnitude reductions in the infrastructure demand while maintaining or even improving model accuracy.

In this work, we consider a new algorithmic approach to cope with the large memory requirement of DNNs, focusing on the critical use-case of embedding tables in deep learning-based recommendation models (DLRMs). These models represent one of the most resource-demanding deep learning workloads, consuming more than 50% of training and 80% of the total AI inference cycles at Facebook's data centers (Gupta et al., 2020; Naumov et al., 2020). Furthermore, the memory capacity requirement for state-of-the-art recommendation models has grown into the scale of terabytes (Lui et al., 2021; Yi et al., 2018; Zhao et al., 2019; 2020). From systems perspective, the large embedding tables that contribute to more than 99% of the total recommendation model capacity are important targets for efficiency optimization. Our approach uses *tensorization* to address the large memory capacity demand of embedding tables in a DLRM.

At a high level, tensorization replaces layers of a neural network with an approximate and structured low-rank form (Novikov et al., 2015). The tensorization form is parametric – its "shape" determines the design trade-off between storage capacity, execution time, and model accuracy. Furthermore, tensorized representation can be fine-tuned with respect to the architecture of a given hardware plat-

[1]Facebook AI, USA [2]Georgia Institute of Technology, USA. Correspondence to: Bilge Acun <acun@fb.com>.

Proceedings of the 4^{th} MLSys Conference, San Jose, CA, USA, 2021.

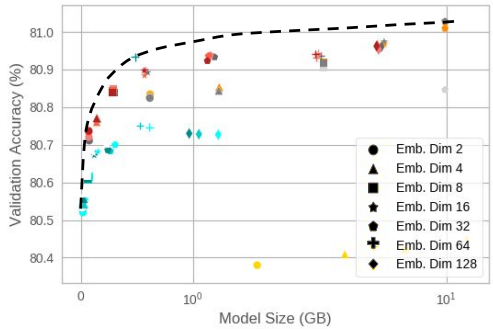

Figure 1. The design space demonstrates the potential of DLRM accuracy and model size tradeoff with respect to the tunable parameters, e.g., ranks of the tensorization method (colors), dimensions of embedding (shapes), and number of compressed embedding tables (brightness). The data points that fall onto the Pareto Frontier (the black curve) represent optimal settings that maximize DLRM accuracy (the y-axis) given a memory size (the x-axis).

form. Figure 1 illustrates the design space with respective to the tunable parameters, such as ranks of the tensorization method, dimensions of the embedding and number of tables to compress. Data points on the Pareto frontier (black curve) represent the optimal settings that maximize the recommendation model accuracy (y-axis) given a corresponding memory size (x-axis). The parameters of these optimal data points vary, depending on the model's characteristics (embedding dimensions), the tensorization setting (i.e., tensor ranks), and the memory capacity of the underlying training system. Given the large configuration space, parameters need to be carefully studied in order to achieve highest possible model quality given a target model size.

We design a **T**ensor-**T**rain compression technique for deep learning **Rec**ommendation models, called *TT-Rec*. The core idea is to replace large embedding tables in a DLRM with a sequence of matrix products. This method is analogous to techniques that use lookup tables to trade-off memory capacity and bandwidth with computation. TT-Rec suits well for accelerators like GPUs, which have a relatively higher compute-to-memory (FLOPs-per-Byte) ratio and limited memory capacity. Since tensor products skew the initialization distribution, we propose a new way to initialize the element distribution of the tensor form. Furthermore, to mitigate increases in training time when the tensor form must be decompressed, we introduce a cache structure that exploits the unique sparse feature distribution in DLRMs, in which we store the most accessed embedding vectors in the uncompressed format. Since these cached embedding vectors are learned without compression, using this cache design can also help recovering model accuracy. Thus, TT-Rec uses a hybrid approach to learn features and deliver on-par model accuracy while requiring orders-of-magnitude less memory capacity.

We show significant compression ratios and improved training time performance at the same time with a judicious design and parameterization of the tensor-train compression technique. The compute-to-memory ratio of the underlying hardware can also be taken into account when parameterizing the proposed technique. While prior works have demonstrated tensor-train compression techniques for embedding layers (Hrinchuk et al., 2020), this paper is the first to explore and customize tensor-train compression techniques for DLRMs, with a particular focus on minimizing the significant memory capacity requirement of the embedding layers (over tens to hundreds of GBs, or over 99% of the total model size).

The main contributions of this paper are as follows:

- This work applies tensor-train compression for a new application context—the embedding layers of DLRMs.
- TT-Rec offers a flexible design space between memory capacity, training time and model accuracy. It is an effective approach especially for online recommendation training. The orders-of-magnitude lower memory requirement with TT-Rec also unlocks a range of modern AI training accelerators for DLRM training.
- Our in-depth design space characterization shows the importance of choosing the right number of embedding tables to compress and the dimension of the compressed tensors. In particular, we quantify the potential trade-off between memory requirements and accuracy.
- To recover accuracy loss, we propose to use a sampled Gaussian distribution for the weight initialization of the tensor cores. Furthermore, to accelerate the training performance of TT-Rec, we introduce a separate cache structure to store frequently-accessed embedding vectors in the uncompressed format, which we show empirically also improves model accuracy.
- We demonstrate the promise of TT-Rec: on Terabyte, TT-Rec achieves higher model accuracy (0.19% to 0.42% over the baseline) while reducing the total memory requirement of the embedding tables by 22× to 112× with a small amount of 10% training time increase on average.

2 BACKGROUND

Deep learning recommendation models. Figure 2 depicts the generalized model architecture for DLRMs. The left box represents the uncompressed baseline, while the right box depicts the proposed TT-Rec design. There are two primary components: the Multi Layer Perceptron (MLP) layer modules and the Embedding Tables (EMBs). The MLP layers are used to process continuous features, such as user age, while the EMBs are used to process categorical features by encoding sparse, high-dimensional inputs into dense, vector representation. The encoded vectors are processed by an interaction operation followed by a top MLP

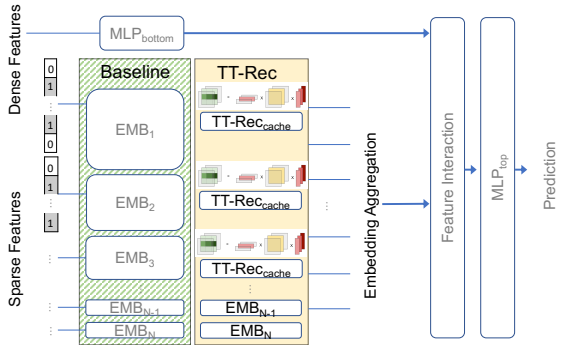

Figure 2. Generalized model architecture for DLRMs. The shaded structure (with orthogonal green stripes) is the baseline DLRM and
TT-Rec design that would replace the baseline in the stack.

layer.

The sparse embedding tables pose infrastructure challenges from both the perspectives of storage and bandwidth requirement. There are tens of millions of rows in an embedding table, where the number of rows are growing exponentially for future recommendation models, resulting in memory requirement into the TB scale (Zhao et al., 2020). Furthermore, the EMB lookup operation gathers multiple embedding vectors simultaneously across the tables, making the execution memory bandwidth bound. To address the ever-increasing memory capacity and bandwidth challenges, this work examines a fundamentally different approach—instead of gathering dense embedding vectors from EMBs that store information in the latent space representation, we seek methods to replace large EMBs with a sequence of small matrix products using tensor train decomposition.

Tensor-train decompositions. Similar to matrix decomposition, such as Singular Value Decomposition (SVD) (Xue et al., 2013) and Principal Component Analysis (PCA) (Wold et al., 1987), Tensor-Train (TT) is an approach to tensor decomposition by decomposing multidimensional data into product of smaller tensors. TT is a simple and robust method for model compression (Oseledets, 2011) and has been studied extensively for deep learning application domains, such as computer vision (Yang et al., 2017) and natural language understanding (Hrinchuk et al., 2020). However, such method has not been investigated for the deep learning recommendation space; thus, its potential remains unknown. Here, we describe the fundamental principle of TT-decomposition and its application to recommendation embedding.

Assume $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \ldots I_d}$ is a $d-$dimensional tensor, where I_k is the size of dimension k. $\mathcal{A}$ can be decomposed as

$$\mathcal{A}(i_1, i_2, \ldots, i_d) = \mathcal{G}_1(:, i_1, :)\mathcal{G}_2(:, i_2, :) \ldots \mathcal{G}_d(:, i_d, :),$$

where $\mathcal{G}_k \in \mathbb{R}^{R_{k-1} \times I_k \times R_k}$, and $R_0 = R_k = 1$ to keep product of the sequence of matrices a scalar. The sequence $\{R_k\}_{k=0}^d$ is referred as to **TT-ranks**, and each 3-dimensional tensor $\mathcal{G}_k$ is called a **TT-core**.

The TT decomposition can also be generalized to compress a matrix $W \in \mathbb{R}^{M \times N}$. We assume that M and N can be factorized into sequences of integers, i.e., $M = \prod_{i=1}^d m_k$, and $N = \prod_{i=1}^d n_k$. Correspondingly, we reshape the matrix W as a $2d$-dimensional tensor $\mathcal{W} \in \mathbb{R}^{(m_1 \times n_1) \times (m_2 \times n_2) \times \cdots \times (m_d \times n_d)}$, where

$$\begin{aligned} &\mathcal{W}((i_1, j_1), (i_2, j_2), \ldots, (i_d, j_d)) \\ &\quad = \mathcal{G}_1(:, i_1, j_1, :)\mathcal{G}_2(:, i_2, j_2, :) \ldots \mathcal{G}_d(:, i_d, j_d, :) \end{aligned} \tag{1}$$

and each 4-d tensor $\mathcal{G}_k \in \mathbb{R}^{R_{k-1} \times m_k \times n_k \times R_k}$, $R_0 = R_d = 1$. Let R, m, and n be the maximal r_k, m_k and n_k respectively for $k = 1, \ldots, d$. TT format reduces the space for storing the matrix from $O(MN)$ to $O(dR^2 \max(m, n)^2)$. Table 2 in Section 6.6 illustrates the detailed embedding table sizes and their corresponding compressed dimensions.

3 Proposed Design of TT-Rec

In this section, we present the proposed design, called *TT-Rec*. TT-Rec customizes the TT-decomposition method to compress embedding tables in deep learning recommendation models (§ 3.1). An overview of our design, where the large embedding tables are replaced with TT-Rec, is shown in Figure 2. In order to compensate the accuracy loss from replacing embedding vectors with vector-matrix multiplications, we introduce a new way to initialize the element distribution for the TT-cores (§ 3.2). This is an important step, since the distribution of the weights resulting from multiplication of the TT-cores are skewed from TT-cores' initial distribution due to the product operation.

3.1 Customizing TT-decomposition for Embedding Table Compression

Each embedding lookup can be interpreted as a one-hot vector matrix multiplication $w_i^T = e_i^T W$, where e_i is a vector with i-th position to be 1, and 0 anywhere else. In more complex scenarios, an embedding lookup represents a weighted combination of multiple items $w = \sum_k i_k^T W$. We compress the embedding table W as in Equation (1), and hence embedding lookup operation of row $i = \sum_{i=1}^d i_k \prod_{j=i+1}^d I_j$ is be represented the following:

$$w_i = \mathcal{G}_1(:, i_1, :, :)\mathcal{G}_2(:, i_2, :, :) \ldots \mathcal{G}_d(:, i_d, :, :) \tag{2}$$

Let $w_i^{(k)} \in \mathbb{R}^{\prod_{i=1}^{k-1} n_i \times n_k R_k}$ be the partial product of the first k TT-cores in Equation 2. The tensor multiplication of $w_i^{(k)} \mathcal{G}_{k+1}(:, i_k, :, :)$ can be unfolded and formulated as a matrix-matrix multiplication where $w_i^{(k)} \in \mathbb{R}^{\prod_{i=1}^{k} n_i \times R_k}$ and $\mathcal{G}_{k+1}(:, i_k, :, :) \in \mathbb{R}^{R_k \times n_{k+1} R_{k+1}}$.

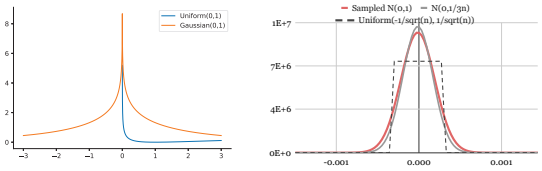

Figure 3. The probability density function (PDF) of product of three independent and identically distributed (i.i.d.) random variables (left) from Uniform(0,1) and from $\mathcal{N}(0,1)$; and from (right) sampled Gaussian distribution (red), comparing to $\mathcal{N}(0,1)$ (grey) and uniform distribution.

Distribution	KL-divergence	Accuracy
uniform($(\frac{-1}{\sqrt{n}}, \frac{1}{\sqrt{n}})$)	0	79.263 %
$\mathcal{N}(0,1)$	$c - \frac{1}{6n}$	78.123 %
$\mathcal{N}(0,1/2)$	$c - \frac{1}{3n} + 0.34$	78.371 %
$\mathcal{N}(0,1/8)$	$c - \frac{4}{3n} + 1.4$	78.823 %
$\mathcal{N}(0,1/3n)$	-0.17	79.256 %
$\mathcal{N}(0,1/9n^2)$	$\frac{1}{2}\ln\frac{18}{\pi n} - 1.5n$	79.220 %

Table 1. The accuracy of uncompressed DLRM with embedding tables initialized from various Gaussian distributions, $c = 0.5\ln\frac{2}{\pi n} < -7$, compared with uniform distribution.

In uncompressed models, storing and updating an embedding table of size $M \times N$ requires for $O(MN)$ space, while in TT-Rec, we propose to only learn the gradient of the loss function L with respect to the TT-cores through backward propagation (Equation 3). Let $\frac{\partial L}{\partial y}$ be the gradient with respect to the output y, and $\frac{\partial L}{\partial w_i^{(k+1)}} = \frac{\partial L}{\partial y}$ for $k = d$,

$$\begin{aligned} \frac{\partial L}{\partial \mathcal{G}_k(:, i_k, :, :)} &= (w_i^{(k)})^T \frac{\partial L}{\partial w_i^{(k+1)}} \\ \frac{\partial L}{\partial w_i^{(k)}} &= \frac{\partial L}{\partial w_i^{(k+1)}} \mathcal{G}_k^T(:, i_k, :, :) \end{aligned} \tag{3}$$

Not all sparse features are equally important is a key observation used in the design of TT-Rec. Data samples in industry-scale recommendation use cases often follow a Power or Zipfian distribution (Wu et al., 2020). A small subset of sparse features capture a significant portion of training samples that index to the set of the features in the embedding tables. This observation is particularly important to reduce the data movement overhead when GPUs are employed as AI training accelerators.

One of the performance optimization potential unlocked by TT-Rec is that the collection of DLRMs that require memory capacities larger than that of training accelerators can now easily run on accelerators. In addition to optimizing TT-Rec's performance using GPUs, we introduce a caching scheme to retain the frequently-accessed embedding vectors/rows in the EMBs. The cache enables TT-Rec to exploit the aforementioned temporal locality by storing most frequently-accessed embedding rows in the uncompressed format. By doing so, TT-Rec minimizes the need of computation. The detail of the performance optimization implementation are described in detail later in Section 4.2.

3.2 Weight Initialization

Replacing an embedding vector lookup with a sequence of tensor computation to approximate the original vector may introduce an accuracy loss in TT-Rec. To compensate the accuracy loss, we introduce a new way to initialize the weights of the TT-cores. Typically, larger TT-ranks provide lower compression ratios while achieving model accuracy closer to that of the baseline. However, when increasing the TT-rank value in TT-Rec, we find that the corresponding accuracy improvements saturate quickly despite the decreasing compression ratios, as we show later in detail in Section 6.6. This led us to investigate the initialization behavior of the uncompressed baseline and TT-Rec—the initial distribution of the TT-cores can significantly influence the model quality.

Since the TT decomposition approximates the full tensor, we hope to find the best configuration of the uncompressed model and have TT-Rec approximate the same configuration. For the uncompressed DLRMs, uniform random distribution usually outperforms normal distribution. For validation, we initialize DLRMs with different forms of Gaussian distribution. Then, we determine the correlation between the model accuracy and the distance between the Gaussian and uniform distribution.

To approximate a uniform distribution on $[a, b]$ by a Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$, we want to minimize the KL-divergence

$$\mathcal{D}_{KL}(P|Q) = -\int_{-\infty}^{\infty} P(x) \ln \frac{P(x)}{Q(x)} dx$$

where $P(x) = \frac{1}{b-a}$ if $x \in [a, b]$, otherwise 0; and $Q(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{(x-\mu)^2}{2\sigma^2}}$. Minimizing the KL-divergence for a given uniform distribution using the first-order approach results in

$$\mu = \frac{a+b}{2}, \quad \sigma^2 = \frac{(b-a)^2}{12}.$$

Therefore, to best approximate the uniform distribution used in the DLRM (Uniform($\frac{-1}{\sqrt{n}}, \frac{1}{\sqrt{n}}$)), we should adopt $\mathcal{N}(0, \frac{1}{3n})$ for initialization. Table 1 shows the accuracy of the DLRMs that are initialized with various Gaussian forms, where accuracy gap is proportional to the KL-divergence between the Gaussian and uniform distribution.

For TT-Rec, we want the product of TT cores to either approximate Uniform($\frac{-1}{\sqrt{n}}, \frac{1}{\sqrt{n}}$) or $\mathcal{N}(0, \frac{1}{3n})$ during initialization. Initializing the TT-cores is challenging since the distribution of product of random variables is non-trivial. In practice, both uniform and normal distributions can be used

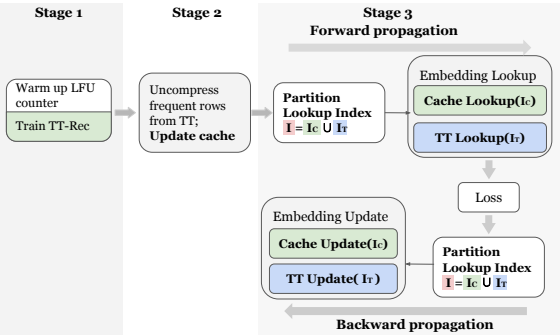

Figure 4. Multi-stage training process with caching.

to reasonably initialize TT-cores.However, the TT product of these distributions do not serve as an appropriate approximation for uniform distribution as Figure 3 (left) shows.

To better approximate the best Gaussian distribution shown in Table 1, we reduce the amount of values close to zero in each core by sampling the Gaussian distribution. Our sampling method is shown in Algorithm 3 in Appendix A. We validate the product of the TT-cores with Algorithm 3 in Appendix A. In Figure 3-(right), we compare the product with $\mathcal{N}(0, 1/3n)$, which serves as the best approximation to Uniform$(-1/\sqrt{n}, 1/\sqrt{n})$. We show later in Section 6.2 that our algorithm achieves the highest accuracy in training.

4 Performance Optimizations

Replacing embedding vectors with TT-core multiplications trades-off memory with computation. Depending on the embedding vector lookup patterns and the underlying system architectures, training time overhead from the computations can vary. In order to mitigate the associated performance overhead, we introduce a cache structure as part of TT-Rec to leverage the observation – a small subset of features (rows in embedding tables) constitute most of the embedding table accesses. Thus, here, we describe the execution flow for deep learning recommendation model training: forward propagation and backward propagation implementations in Section 4.1. Then, we introduce TT-Rec's cache design to improve the training time performance in Section 4.2.

4.1 TT-Rec Implementation

The embedding layer takes row indices as input in Compressed Sparse Row (CSR) format. The input is translated into multiple embedding bags, in which the embedding vectors belonging to an embedding bag are pooled together by summation or average. Let $indices[]$ be an integer array of length n, and $offsets[]$ be an integer array of length $m \leq n$ to specify the starting index in $indices[]$ of each embedding bag. Specifically, to compute for jth embedding bag with embedding matrix W, the algorithm will summarize or average the rows $\{W(indices[k],:)|offsets[j] \leq k < offsets[j+1]\}$. In TT-Rec, this is computed as a sequence of small matrix-matrix products.

$$\begin{aligned} output_i &= \sum_{k=offset[i]}^{offset[i+1]} \alpha_{indices[k]} w_{indices[k]} \\ &= \sum_{k=offset[i]}^{offset[i+1]} \alpha_{indices[k]} \mathcal{G}_1(i_1^k,:,:)\ldots\mathcal{G}_d(:,i_d^k,:), \end{aligned} \tag{4}$$

where i_j^k indexes the slice in $\mathcal{G}_j$ that is used for computing vector $W(indices[k],:)$, and α_i is the per sample weight associated with each embedding vector.

To obtain a high-performance implementation of TT-EmbeddingBag, we use a batched GEMM implementation from cuBLAS (NVIDIA, 2020). The algorithm computes and stores the intermediate GEMM results $tr_i = tr_{i-1}\mathcal{G}_i$, where $tr_1 = \mathcal{G}_1$, for all B vectors by a single batched GEMM kernel. Given a batch of queried embedding vectors, the algorithm reduces the batch to embedding bags in parallel. Algorithm 1 of Appendix A shows the pseudocode of the batched embedding kernel for the 3-dimensional TT decomposition. This algorithm can be generalized to any arbitrary TT dimension by extending lines 5-10 to set up the pointers to intermediate GEMM results.

The backward propagation algorithm for the 3-dimensional TT compression is illustrated by Algorithm 2 of Appendix A. Based on the chain rule as described in Equation 3, we compute the gradient w.r.t. the embedding bags, and accumulate the gradients into each TT-core.

4.2 A Least-Frequently-Used Cache for TT-Rec

While TT-Rec reduces the memory requirement of embedding tables, this method introduces latency through additional computations from (1) reconstructing values of the queried embedding rows from the TT format in the forward propagation, (2) determining the gradient for each TT-core in the backward propagation, and (3) recomputing the intermediate GEMM results for the gradient computation.

Recomputation of the intermediate results, in Algorithm 2 (line 3) of Appendix A, can be eliminated by storing tensors from the forward pass. This reduces the latency but comes with slightly increased memory footprint and memory allocation time. The dominating source of performance optimization potential, however, comes from the aforementioned distribution of sparse features in the training samples. The row access frequency follows the Power Law distribution in the largest embedding tables, i.e. a few embedding vectors are recurrently accessed throughout training.

To consider the characteristics of recommendation data sam-

Emb. Table Dimensions		TT-Core Shapes			# of TT Parameters			Memory Reduction		
# Rows	Emb. Dim.	$\mathcal{G}_1$	$\mathcal{G}_2$	$\mathcal{G}_3$	$R = 16$	$R = 32$	$R = 64$	$R = 16$	$R = 32$	$R = 64$
10131227	16	(1, 200, 2, R)	(R, 220, 2, R)	(R, 250, 4, 1)	135040	495360	1891840	1200	327	86
8351593	16	(1, 200, 2, R)	(R, 200, 2, R)	(R, 209, 4, 1)	122176	449152	1717504	1094	297	78
7046547	16	(1, 200, 2, R)	(R, 200, 2, R)	(R, 200, 4, 1)	121600	448000	1715200	927	252	66
5461306	16	(1, 166, 2, R)	(R, 175, 2, R)	(R, 188, 4, 1)	106944	393088	1502976	817	222	58
2202608	16	(1, 125, 2, R)	(R, 130, 2, R)	(R, 136, 4, 1)	79264	291648	1115776	445	121	32
286181	16	(1, 53, 2, R)	(R, 72, 2, R)	(R, 75, 4, 1)	43360	160448	615808	106	28	7
142572	16	(1, 50, 2, R)	(R, 52, 2, R)	(R, 55, 4, 1)	31744	116736	446464	72	19	5

Table 2. The original dimensions of Kaggle's 7 largest embedding tables in DLRM and their respective TT decomposition parameters.

ples, we incorporate a software cache structure to store the frequently-used embedding vectors on the GPU. The cache stores an uncompressed copy of the frequently accessed embedding vectors. As Section 6.5 later shows, we perform performance sensitive analysis over a wide range of cache sizes and empirically determine that devoting 0.01% of the respective embedding tables as TT-Rec caches is sufficient from both model accuracy's and training time's perspectives.

Given the sequence of embedding bags, the indices are first partitioned into two different groups: $cached\_indices$ and $tt\_indices$. The cached embedding rows are directly fetched from the cache whereas the embedding vectors from the $tt\_indices$ group are computed, following the forward propagation operations described in Algorithm 1. Then, during the backward propagation, the cached, uncompressed vectors can be simply updated with $W' = W + \eta \frac{\partial L}{\partial W}$, while the non-cached vectors are updated to TT-cores as described in Algorithm 2. In this way, the weights of the two index sets are learned separately by the cache and the TT cores.

In order to offset the cache population overheads, TT-Rec adopts a semi-dynamic cache, where the frequently-accessed embedding vectors are loaded into cache only every 100s to 1000s of iterations, initialized from TT cores. In order to track the frequencies of the all the existing indices, an open addressing hash table is used. On the other hand, the learned weights in the cache would be discarded when an eviction happens. In practice, this strategy does not affect training accuracy as the evicted cache lines are not accessed frequently and therefore contribute less to the overall model. We chose this strategy as decomposing the evicted vectors and updating the decomposed parameters with the existing TT cores are equivalent to dynamically tracking TT decomposition for a streaming matrix, which is a challenging algebraic problem itself.

Figure 4 summarizes the multi-stage training process. The model training starts with the TT embedding tables only. The first few iterations (e.g. 10% training samples) are used to warm up the cache state. The most frequently accessed embedding vectors will be stored in the cache as uncompressed. Depending on the phase behavior, one might consider updating the cache and repeat the warm up process periodically. Based on our empirical observation, the set of the most-frequently-accessed vectors are stable over window, indicating little periodic warm-up need.

5 Experimental Setup

Deep Learning Recommendation Model Parameters and Data Sets: We implement the proposed TT-Rec design over the open-source MLPerf reference implementation of the DLRM recommendation model architecture (MLPerf-DLRM) (Mattson et al., 2020a;b; MLPerf, 2020; Naumov et al., 2019). We train TT-Rec with the Criteo Kaggle Display Advertising Challenge Dataset (Kaggle) (Criteo Labs, 2014) and Criteo Terabyte Click Logs (Criteo Labs, 2013). In both the datasets, each data sample consists of 13 numerical features and 26 categorical features, and a binary label. Kaggle contains 7 days of ads click data, whereas Terabyte contains 24 days of click data (4.3 billion records). The 26 categorical features are interpreted into 26 embedding tables in TT-Rec, where each row in the embedding table corresponds to an element in that category. Figure 6-Baseline bars show the size and composition of the embedding tables in the two datasets. Table 2 summarizes the dimensions of the 7 largest embedding tables when training MLPerf-DLRM with Kaggle and the respective TT decomposition parameters.

To study the effectiveness of TT-Rec, we adopt the same hyperparamters as specified in the MLPerf-DLRM reference implementation, including the embedding dimensions, MLP dimensions, learning rate, and batch size. Both datasets are trained with the SGD optimizer. Note, as the dimension of the embedding increases from 64 to 512, the total memory requirement is over 96 GB, exceeding the latest GPU memory capacity. This is when TT-Rec shines. The uncompressed baseline has to run on CPUs or multiple GPUs via model parallelism (which requires extra all-to-all communication overheads) while TT-Rec enables recommendation training on GPUs with data parallelism.

For accuracy evaluation, we report the test accuracy (%) as well as the BCE loss. We train TT-Rec for a single epoch using all the data samples in Kaggle. For Terabyte,

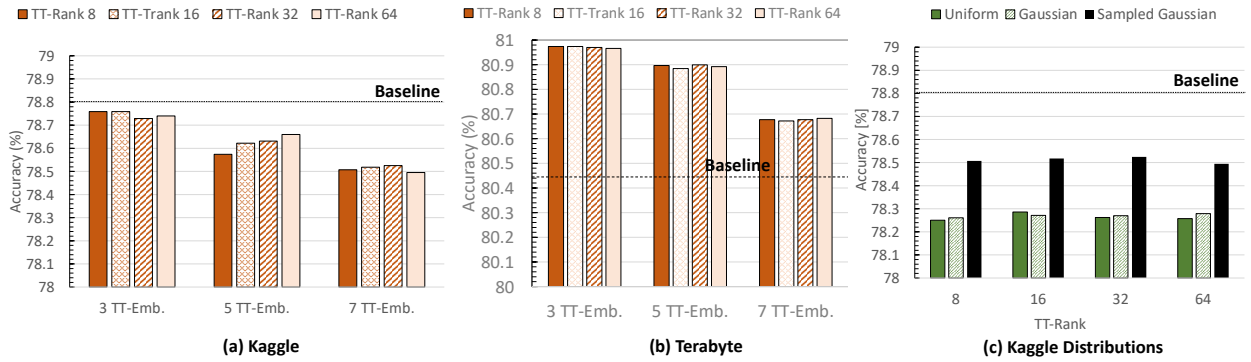

Figure 5. (a) and (b) Validation accuracy of TT-Rec with various tables compressed and TT ranks training with Kaggle and Terabyte respectively. (c) Validation accuracy of TT-Rec using different initialization techniques on Kaggle with 7 TT-Embedding tables.

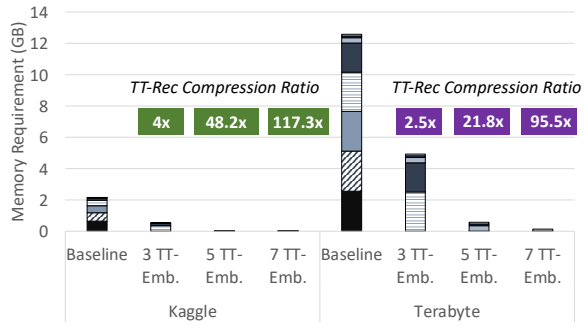

Figure 6. TT-Rec shows significant model size reduction when compressing different number of embeddings for Kaggle in green, and Terabyte in purple labels (TT-rank of 32).

we downsize the negative training samples by 0.875, as specified by the MLPerf-DLRM benchmark.

All the evaluation results are obtained by training Kaggle dataset on an NVIDIA Tesla V100-SXM2 GPU with an Intel Xeon E5-2698 CPU. Terabyte dataset results are obtained on the same system but using eight CPUs with a single GPU due to its larger system memory requirement.

6 Experimental Results

TT-Rec: Overall, TT-Rec demonstrates to be an effective technique. Here, we first present the TT-Rec results without using our custom designed cache. For Kaggle, TT-Rec reduces the overall model size requirement by 4× to 221×, from 2.16 GB to less than 0.5 GB. The different capacity reduction comes with validation accuracy loss ranging from 0.03% to 0.3%, and a small amount of training time increase by 7.4% on average. Similarly, for Terabyte, the overall model size requirement is reduced by 112×, from 12.57 to 0.11 GB. The model quality experiences no degradation, TT-Rec outperforms the uncompressed baseline by 0.2% to 0.5% on validation accuracy. Finally, this comes with a training time increase by 13.9% on average.

6.1 Memory Capacity Requirement with TT-Rec

TT-Rec achieves significant compression ratios for the embedding tables of Terabyte and Kaggle, by as much as 327× and by an average of 181× (with TT-rank of 32). In the uncompressed baseline, the 7 largest tables constitutes 99% of the model. For Kaggle, with TT-Rec, the memory requirement of the 7 embedding tables is reduced from 2.16 GB to only 18 MB, leading to 112× model size reduction.

Figure 6 compares the memory capacity requirement (y-axis) between the baseline and TT-Rec (x-axis) across the 3, 5, and 7 largest embedding tables. As illustrated in Figure 6, the model size requirement also becomes significantly lower when TT-Rec trains the less number of the large embedding tables in the TT-Emb. format – for TT-Emb. of 5 and 3, the overall model size is reduced by 48× and 4×, respectively. For Terabyte, TT-Rec achieves 2.6, 21.8, and 95.5× model size reduction for TT-Emb. of 3, 5 and 7, respectively. This impressive memory capacity reduction unlocks industry-scale multi-GB/TB DLRMs that cannot be previously trained using commodity training accelerators, such as GPUs (40 GB of HBM2 in the latest NVIDIA A100 GPUs (NVIDIA, 2020b) or TPUs 16-32 GB of HBM (Chao & Saeta, 2019)), to enjoy significantly higher throughput performance in state-of-the-art training accelerators.

6.2 Model Accuracy with TT-Rec

To achieve accuracy-neutral while still enjoying TT-Rec's memory reduction benefit, Figure 5(a) and (b) compare the validation accuracy of the uncompressed baseline with that of TT-Rec for Kaggle and Terabyte, respectively. Figure 5(a) shows that, when TT-Rec trains the largest 3, 5, and 7 embedding tables in the TT-Emb. format (x-axis), the optimal TT-rank to achieve a nearly accuracy-neural result varies, with the optimal rank of 8, 32, and 64, respectively.

Interestingly, for Terabyte, Figure 5(b) shows that TT-Rec

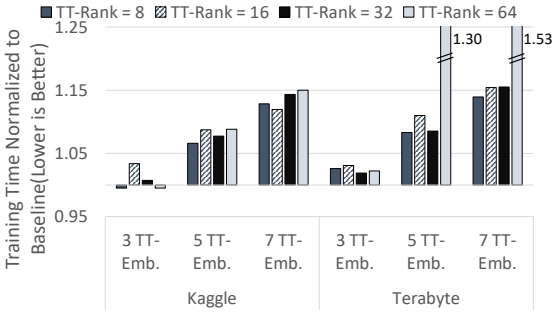

Figure 7. Training time comparison across TT-Rec settings. Baseline takes 12.14ms/iter on Kaggle, and 12.64ms/iter on Terabyte.

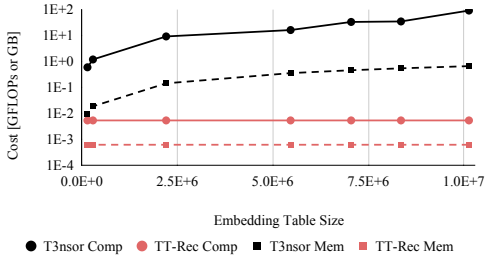

Figure 8. System resource requirement of T3nsor and TT-Rec.

achieves higher validation accuracies (y-axis) across the board. As expected, with more embedding tables trained in the TT-Emb. format, TT-Rec brings significantly higher model size reduction at the expense of model accuracy degradation. Increasing the number of the large embedding tables trained in the TT-Emb. format from 3 to 7 improves the model size reduction from 2.6 to 95.5× while the validation accuracy degrades from 80.975% to 80.682%. Note, even though the validation accuracy is lowered, the model accuracy (TT-Emb. of 7) still outperforms the uncompressed baseline of 80.45%.

Using larger TT-ranks produces more accurate models at the expense of lower compression ratios. We notice that, although mathematically larger TT-ranks should produce more accurate approximations to the full tensor, increasing the rank does not always compensate the loss of accuracy. We believe that such accuracy loss is caused by the weight initialization distribution, as we describe next.

Figure 5(c) presents the TT-Rec accuracy results using the different weight initialization strategies, described in §3.2. Recall, the model accuracy difference between TT-Rec and the baseline strongly correlates with the distance between the distribution of the full matrix generated by the set of the tensor cores and the uniform distribution. Thus, the sampled Gaussian distribution is expected to outperform the other distributions as the full matrix generated by TT-Rec approximate $\mathcal{N}(0, \frac{1}{3n})$ the best. The accuracy results in Figure 5(c) verify this expectation empirically.

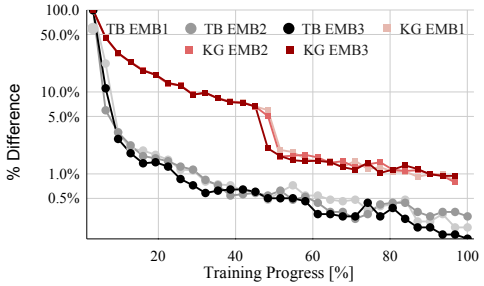

Figure 9. The set of frequently-accessed embedding rows over time stabilize at around 5% and 50% of the training run for Terabyte and Kaggle, respectively.

6.3 Training Time Performance of TT-Rec

To have a full picture of TT-Rec's potential, we present the training time performance of TT-Rec next. Figure 7 depicts the normalized training time of TT-Rec (y-axis), using TT-ranks of [8, 16, 32, 64] across the TT-Emb. settings of [3, 5, 7] (x-axis). Higher model size reduction ratios come with higher training time overheads. Increasing the number of the large embedding tables trained in the TT-Emb. format from 3 to 7 reduces the model sizes by 46.5 and 37.4× for Kaggle and Terabyte, respectively, while the training time using the optimal TT-rank increases by 12.5% and 11.8%, respectively. Depending on the importance of the three axes—*memory capacity requirement*, *model quality* and *training time performance* for DLRM training—TT-Rec offers a flexible design space that can be navigated according to the desired optimization goal.

6.4 TT-Embedding Kernel Implementation Efficiency

To quantify the efficiency of our TT-Rec implementation, we compare the performance of the TT-Embedding kernel with the baseline PyTorch EmbeddingBag operator (Paszke et al., 2019) and the state-of-the-art TT embedding library, called T3nsor (Hrinchuk et al., 2020), for word embedding of NLP use cases. Figure 8 shows that our TT-Embedding implementation is order-of-magnitude more efficient than that of T3nsor, from the perspectives of compute (represented by circle) and memory (represented by square) requirement, over the number of rows in embedding table (x-axis). T3nsor decompresses embedding tables on the fly; thus, it requires the same amount of memory footprint during training as that of the PyTorch Embedding Bag operator. Our implementation achieves a memory footprint reduction of $\frac{\#Emb\ Rows}{Batch\ Size}$ yielding roughly 10,000× lower memory footprint requirement as compared to T3nsor and the PyTorch implementation. The overall training time of TT-Rec is on par with that of the baseline using the PyTorch Embedding Bag operator (Figure 7; TT-Emb. of 3; Kaggle), and is 2.4× faster than T3nsor on average.

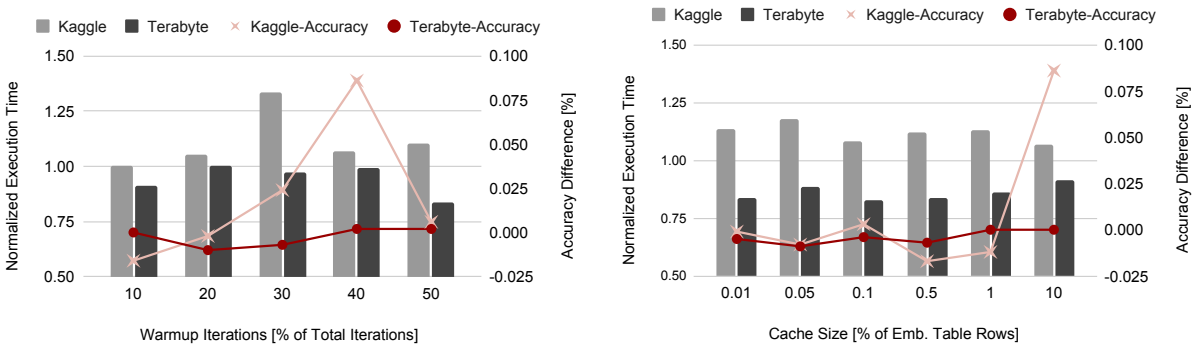

Figure 10. The impact of warm-up iterations on the TT-Rec training time and model accuracy (left). The model training time and accuracy over the cache sizes with respect to the embedding table (right).

6.5 Analysis for TT-Rec Cache Performance

As described in §4.2, we introduce a cache structure in TT-Rec to capture the small subset of embedding rows with data locality. Based on the available memory capacity on the training system, TT-Rec caches an uncompressed version of frequently-accessed rows to reduce the computation need.

To illustrate the feature reuse potential in the context of TT-Rec, Figure 9 depicts the percentage of changes in the set of the most-frequently-accessed 10k embedding rows over the training run, for the three largest embedding tables (EMB1, EMB2, and EMB3). We count the cumulative row access frequencies every 3% of training progress and measure the difference between each consecutive points (y-axis of Figure 9 in the log-scale). This difference is indicative of training phase stability: the lower the difference is, the more stable the set of frequently-accessed rows is.

Figure 10(a) shows the impact of warm-up iterations on the TT-Rec training time and model accuracy. During the warm-up period, the cache structure is being filled up with most-frequently-accessed embedding vectors, using the aforementioned LFU replacement. Thus, the longer the warm-up period is, the better the TT-Rec cache captures the most-frequently-accessed embedding vectors and the higher the cache hit rate is for the remaining training iterations.

For Kaggle, as the warm-up iterations increase from 10% to 30% of the total training iterations, the total training time increases by 33%. This is because the training time speedup from the slightly higher cache hit rate (from the warm-up period) is not sufficient to compensate the warm-up time overhead. As warm-up continues to increase beyond 30% of the training, we start seeing the training time overhead to decrease. This happens as the hit rate improves with more warm-up iterations. In contrast, with the larger Terabyte dataset, TT-Rec cache can effectively and consistently reduce the training time across the different warm-up iterations. It improves the end-to-end training time by up-to 19% with negligible accuracy impact. Overall, with the cache

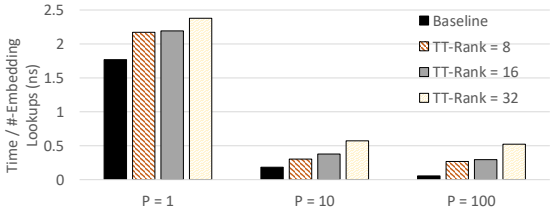

Figure 11. Performance of TT-Rec for MLPerf-DLRM (P of 1) and embedding-dominated DLRMs (P of 10 and 100).

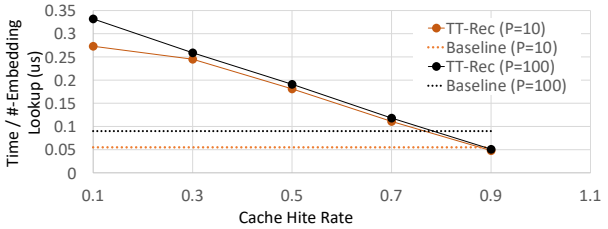

Figure 12. Performance comparison for TT-Rec's EmbeddingBag kernel and the baseline.

support, TT-Rec receives additional accuracy improvement of 0.09% on Kaggle and 0.02% on Terabyte.

Another important design parameter for TT-Rec is the size of the LFU cache. Figure 10(b) shows the corresponding model training time and accuracy over the cache sizes ranging from 0.01% to 10% of the respective embedding tables. For Kaggle and Terabyte, devoting 0.01% worth of the embedding table memory requirement is sufficient.

6.6 TT-Rec Performance with Large Pooling Factors

To understand Rec-TT training performance for other categories of DLRMs that are embedding-dominated (Gupta et al., 2020; Gupta et al., 2020; Ke et al., 2020), we develop a suite of microbenchmarks to synthetically generate embedding vectors with a polling factor (P) of 10 and 100. The pooling factor is defined as the average number of embedding lookups required per training sample. The larger P, the more embedding vectors are looked up and gathered, and

the embedding operations are more expensive. Both Kaggle and Terabyte datasets correspond to P = 1.

Figure 11 compares the performance of the non-cached TT-Rec kernel with PyTorch EmbeddingBag for P =[1, 10, 100] across various TT-ranks. As seen in Figure 11, the performance per training sample is better as P increases. This is because as the number of embedding lookups increase, the overhead of EmbeddingBag is amortized, in both the baseline and TT-Rec cases. Furthermore, the performance gap between the non-cached TT-Rec kernel and EmbeddingBag increases as P increases. This is because there exists higher reuses of embedding vectors when P is larger and EmbeddingBag benefits from such reuse.

To reduce the execution time of TT-Rec as P increases, Figure 12 compares the performance of TT-Rec with caching enabled with EmbeddingBag. To quantify the timing performance per training sample, we synthetically generate data samples to control the cache hit rate. As expected, as the cache hit rate increases (the x-axis), the performance of TT-Rec improves and eventually outperforms the baseline EmbeddingBag when the cache hit rate reaches 90%.

7 Related Work

General Model Compression Techniques: Among the many compression techniques for training, the commonly-used ones include magnitude pruning (Zhu & Gupta, 2017), variational dropout (Kingma & Welling, 2013), and l_0 regularization (Louizos et al., 2017). Other efforts propose to impose structured sparsity in model weights upfront (Child et al., 2019; Gray et al., 2017). Such approaches can significantly reduce both training and inference cost, but have not been proven as effective solutions for deep learning recommendation models. Furthermore, while these methods are generally applicable, the prior works are fundamentally different from our proposed TT-decomposition approach.

Embedding Table Compression Techniques: The seminal work by Weinberger et al. examined feature hashing, allowing multiple elements to map to the same embedding vector; thus, it reduces the embedding space (Weinberger et al., 2009). However, hash collisions can yield significant accuracy losses. For example, Zhao et al. observed an intolerable degree of accuracy loss if hashing were applied to terabyte-scale recommendation models (Zhao et al., 2020). Guan et al. proposed a 4-bit quantization scheme to compress pre-trained models for inference (Guan et al., 2019). This design is feasible for recommendation inference although quantization for training is challenging and often comes with accuracy trade-off. A recent technique proposed is to generate embedding vectors on the fly using dense neural networks (Kang et al., 2020). This work achieves a comparable accuracy against full embedding with 1/4th model size. However, it comes with $9x$ runtime overhead. Other works also explored the potential of low-rank approximation on the embedding tables (Ghaemmaghami et al., 2020; Hrinchuk et al., 2020) but experienced critical accuracy degradation. None provides a computationally-efficient interface for industry-scale DLRMs.

Tensorization: Tensor methods have been extensively studied for compressing DNNs. One of the most common method is the Tucker factorization (Cohen et al., 2016), which can generate high-quality DNNs when compressing fully-connected layers. Tensor Train (TT) and Tensor Ring (TR) decomposition techniques have been recently studied in the context of DNNs (Hawkins & Zhang, 2019; Wang et al., 2018). But previous work has explored the accuracy trade-off for fully-connected and convolution layers only. In particular, TT decomposition offers a structural way to compress DNNs and thus is capable of preserving the DNN weights. TR can preserve the weights with moderately lower compression ratios than that of TT (Wang et al., 2018). Despite interest in TT-based methods, to the best of our knowledge, ours is the first to consider them in the context of DLRMs. In our analysis, we comprehensively study the design space of how memory size reduction, model quality, and training time overheads trade-off. Finally, we also present an efficient implementation for TT-Rec, which we will release as open source upon the acceptance of this work.

8 Conclusion

The spirit of TT-Rec is to use principled, parameterized algorithmic methods to help control the explosive demands on computational infrastructure. This strategy complements innovations in the infrastructure itself. TT-Rec specifically attacks the considerable memory requirements of embedding layers of modern recommendation models, whose memory requirements in industrial applications at scale can require hundreds of GBs to TBs of memory. TT-Rec replaces otherwise large embedding tables with a sequence of matrix products, reducing the total model memory capacity requirement by 112 times with only a small amount of 13.9% training time overhead while maintaining the same model accuracy as the baseline. Such significant memory capacity reductions can be achieved with a relatively small increase in training time through clever caching strategies, making online recommendation model training more practical.

Acknowledgements

We would like to thank Richard Vuduc and Mark Tygert for the rigorous discussions and thoughtful feedback. We also thank Hsien-Hsin Lee and Kim Hazelwood for supporting this work.

REFERENCES

Acun, B., Murphy, M., Wang, X., Nie, J., Wu, C.-J., and Hazelwood, K. Understanding training efficiency of deep learning recommendation models at scale. In Proceedings of the IEEE International Symposium on High Performance Computer Architecture, 2021.

Alibaba. Alibaba unveils ai chip to enhance cloud computing power. https://www.alibabagroup.com/en/news/article?news=p190925, 2019.

Amazon. AWS Inferentia: High performance machine learning inference chip, custom designed by AWS. https://aws.amazon.com/machine-learning/inferentia/, 2019.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. arXiv:2005.14165, 2020.

Chao, C. and Saeta, B. Cloud TPU: Codesigning architecture and infrastructure, 2019. URL https://www.hotchips.org/hc31/HC31_T3_Cloud_TPU_Codesign.pdf.

Child, R., Gray, S., Radford, A., and Sutskever, I. Generating long sequences with sparse transformers. arXiv:1904.10509, 2019.

Chung, E., Fowers, J., Ovtcharov, K., Papamichael, M., Caulfield, A., Massengill, T., Liu, M., Ghandi, M., Lo, D., Reinhardt, S., Alkalay, S., Angepat, H., Chiou, D., Forin, A., Burger, D., Woods, L., Weisz, G., Haselman, M., and Zhang, D. Serving DNNs in real time at datacenter scale with project brainwave. IEEE Micro, 38:8–20, March 2018.

Cohen, N., Sharir, O., and Shashua, A. On the expressive power of deep learning: A tensor analysis. In Proceedings of the Conference on learning theory, 2016.

Criteo Labs. Download terabyte click logs. https://labs.criteo.com/2013/12/download-terabyte-click-logs/, 2013.

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

Fowers, J., Ovtcharov, K., Papamichael, M., Massengill, T., Liu, M., Lo, D., Alkalay, S., Haselman, M., Adams, L., Ghandi, M., Heil, S., Patel, P., Sapek, A., Weisz, G., Woods, L., Lanka, S., Reinhardt, S., Caulfield, A., Chung, E., and Burger, D. A configurable cloud-scale dnn processor for real-time ai. In Proceedings of the International Symposium on Computer Architecture, June 2018.

Ghaemmaghami, B., Deng, Z., Cho, B., Orshansky, L., Singh, A. K., Erez, M., and Orshansky, M. Training with multi-layer embeddings for model reduction. arXiv:2006.05623, 2020.

Gray, S., Radford, A., and Kingma, D. P. Gpu kernels for block-sparse weights. arXiv:1711.09224, 3, 2017.

Guan, H., Malevich, A., Yang, J., Park, J., and Yuen, H. Post-training 4-bit quantization on embedding tables, 2019.

Gupta, U., Hsia, S., Saraph, V., Wang, X., Reagen, B., Wei, G., Lee, H. S., Brooks, D., and Wu, C. DeepRecSys: A system for optimizing end-to-end at-scale neural recommendation inference. In Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture, pp. 982–995, 2020.

Gupta, U., Wu, C.-J., Wang, X., Naumov, M., Reagen, B., Brooks, D., Cottel, B., Hazelwood, K., Jia, B., Lee, H.-H. S., Malevich, A., Mudigere, D., Smelyanskiy, M., Xiong, L., and Zhang, X. The Architectural Implications of Facebook's DNN-based Personalized Recommendation. In Proceedings of the IEEE International Symposium on High Performance Computer Architecture, 2020.

Hawkins, C. and Zhang, Z. Bayesian Tensorized Neural Networks with Automatic Rank Selection. arXiv:1905.10478, 2019.

Hazelwood, K. Deep learning: It's not all about recognizing cats and dogs, 2020. URL https://databricks.com/speaker/kim-hazelwood.

Hazelwood, K., Bird, S., Brooks, D., Chintala, S., Diril, U., Dzhulgakov, D., Fawzy, M., Jia, B., Jia, Y., Kalro, A., Law, J., Lee, K., Lu, J., Noordhuis, P., Smelyanskiy, M., Xiong, L., and Wang, X. Applied machine learning at Facebook: A datacenter infrastructure perspective. In Proceedings of the International Symposium on High Performance Computer Architecture, 2018.

Hrinchuk, O., Khrulkov, V., Mirvakhabova, L., Orlova, E., and Oseledets, I. Tensorized embedding layers for efficient model compression, 2020.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R.,

Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-datacenter performance analysis of a tensor processing unit. In Proceedings of the ACM/IEEE International Symposium on Computer Architecture, 2017.

Kang, W.-C., Cheng, D. Z., Yao, T., Yi, X., Chen, T., Hong, L., and Chi, E. H. Deep hash embedding for large-vocab categorical feature representations. arXiv:2010.10784, 2020.

Ke, L., Gupta, U., Cho, B. Y., Brooks, D., Chandra, V., Diril, U., Firoozshahian, A., Hazelwood, K., Jia, B., Lee, H. S., Li, M., Maher, B., Mudigere, D., Naumov, M., Schatz, M., Smelyanskiy, M., Wang, X., Reagen, B., Wu, C., Hempstead, M., and Zhang, X. RecNMP: Accelerating personalized recommendation with near-memory processing. In Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture, 2020.

Kingma, D. P. and Welling, M. Auto-encoding variational bayes. arXiv:1312.6114, 2013.

Lee, K. and Rao, V. Accelerating facebook's infrastructure with application-specific hardware. https://engineering.fb.com/data-center-engineering/accelerating-infrastructure/, March 2019.

Louizos, C., Welling, M., and Kingma, D. P. Learning sparse neural networks through l_0 regularization. arXiv:1712.01312, 2017.

Lui, M., Yetim, Y., Özgür Özkan, Zhao, Z., Tsai, S.-Y., Wu, C.-J., and Hempstead, M. Understanding capacity-driven scale-out neural recommendation inference. Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2021.

Mattson, P., Cheng, C., Coleman, C., Diamos, G., Micikevicius, P., Patterson, D., Tang, H., Wei, G.-Y., Bailis, P., Bittorf, V., Brooks, D., Chen, D., Dutta, D., Gupta, U., Hazelwood, K., Hock, A., Huang, X., Jia, B., Kang, D., Kanter, D., Kumar, N., Liao, J., Narayanan, D., Oguntebi, T., Pekhimenko, G., Pentecost, L., Reddi, V. J., Robie, T., John, T. S., Wu, C.-J., Xu, L., Young, C., and Zaharia, M. MLPerf training benchmark. Proceedings of the Conference on Systems and Machine Learning, 2020a.

Mattson, P., Reddi, V. J., Cheng, C., Coleman, C., Diamos, G., Kanter, D., Micikevicius, P., Patterson, D., Schmuelling, G., Tang, H., Wei, G.-w., and Wu, C.-J. Mlperf: An industry standard benchmark suite for machine learning performance. IEEE Micro, 40(2):8–16, 2020b.

MLPerf. MLPerf Training. https://mlperf.org/training-overview/#overview, 2020.

Naumov, M., Mudigere, D., Shi, H.-J. M., Huang, J., Sundaraman, N., Park, J., Wang, X., Gupta, U., Wu, C.-J., Azzolini, A. G., Dzhulgakov, D., Mallevich, A., Cherniavskii, I., Lu, Y., Krishnamoorthi, R., Yu, A., Kondratenko, V., Pereira, S., Chen, X., Chen, W., Rao, V., Jia, B., Xiong, L., and Smelyanskiy, M. Deep Learning Recommendation Model for Personalization and Recommendation Systems. arXiv:1906.00091, 2019.

Naumov, M., Kim, J., Mudigere, D., Sridharan, S., Wang, X., Zhao, W., Yilmaz, S., Kim, C., Yuen, H., Ozdal, M., Nair, K., Gao, I., Su, B.-Y., Yang, J., and Smelyanskiy, M. Deep learning training in Facebook data centers: Design of scale-up and scale-out systems. arXiv:2003.09518, 2020.

Novikov, A., Podoprikhin, D., Osokin, A., and Vetrov, D. Tensorizing neural networks. Advances in Neural Information Processing Systems, pp. 442–450, 2015.

NVIDIA. 2.8.13. cublasgemmbatchedex, cublas :: Cuda toolkit documentation, 2020. URL https://docs.nvidia.com/cuda/cublas/index.html#cublas-GemmBatchedEx.

NVIDIA. NVIDIA DGX A100: the universal system for AI infrastructure, 2020a. URL https://www.nvidia.com/en-us/data-center/dgx-a100/.

NVIDIA. NVIDIA A100 Tensor Core GPU, 2020b. URL https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet.pdf.

Oseledets, I. V. Tensor-train decomposition. Society for Industrial and Applied Mathematics, 45(4):1600–1621, 2011.

Ovtcharov, K., Ruwase, O., Kim, J.-Y., Fowers, J., Strauss, K., and Chung, E. Toward accelerating deep learning at scale using specialized hardware in the datacenter. In Proceedings of the IEEE Symposium on High-Performance Chips, August 2015.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. PyTorch: An imperative style, high-performance deep learning library. arXiv:1912.01703, 2019.

Reddi, V. J., Cheng, C., Kanter, D., Mattson, P., Schmuelling, G., Wu, C.-J., Anderson, B., Breughe, M., Charlebois, M., Chou, W., Chukka, R., Coleman, C., Davis, S., Deng, P., Diamos, G., Duke, J., Fick, D., Gardner, J. S., Hubara, I., Idgunji, S., Jablin, T. B., Jiao, J., John, T. S., Kanwar, P., Lee, D., Liao, J., Lokhmotov, A., Massa, F., Meng, P., Micikevicius, P., Osborne, C., Pekhimenko, G., Rajan, A. T. R., Sequeira, D., Sirasao, A., Sun, F., Tang, H., Thomson, M., Wei, F., Wu, E., Xu, L., Yamada, K., Yu, B., Yuan, G., Zhong, A., Zhang, P., and Zhou, Y. MLPerf inference benchmark. Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture, 2020.

Wang, W., Sun, Y., Eriksson, B., Wang, W., and Aggarwal, V. Wide Compression: Tensor Ring Nets. Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pp. 9329–9338, 2018.

Weinberger, K., Dasgupta, A., Langford, J., Smola, A., and Attenberg, J. Feature hashing for large scale multitask learning. In Proceedings of the 26th annual international conference on machine learning, pp. 1113–1120, 2009.

Wold, S., Esbensen, K., and Geladi, P. Principal component analysis. Chemometrics and intelligent laboratory systems, 2(1-3):37–52, 1987.

Wu, C.-J., Burke, R., Chi, E. H., Konstan, J., McAuley, J., Raimond, Y., and Zhang, H. Developing a recommendation benchmark for mlperf training and inference. arXiv:2003.07336, 2020.

Xue, J., Li, J., and Gong, Y. Restructuring of deep neural network acoustic models with singular value decomposition. Proceedings of Interspeech, 2013.

Yang, Y., Krompass, D., and Tresp, V. Tensor-train recurrent neural networks for video classification. Proceedings of the International Conference on Machine Learning, 2017.

Yi, X., Chen, Y.-F., Ramesh, S., Rajashekhar, V., Hong, L., Fiedel, N., Seshadri, N., Heldt, L., Wu, X., and Chi, E. H. Factorized deep retrieval and distributed tensorflow serving. Proceedings of the Conference on Systems and Machine Learning, 2018.

Zhao, W., Zhang, J., Xie, D., Qian, Y., Jia, R., and Li, P. AIBox: CTR prediction model training on a single node. In Proceedings of the ACM International Conference on Information and Knowledge Management, 2019.

Zhao, W., Xie, D., Jia, R., Qian, Y., Ding, R., Sun, M., and Li, P. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems. arXiv:2003.05622, 2020.

Zhu, M. and Gupta, S. To prune, or not to prune: exploring the efficacy of pruning for model compression. arXiv:1710.01878, 2017.

APPENDIX

A ALGORITHM DETAILS

Algorithm 1 and Algorithm 2 below shows the details of the forward and back-propagation algorithm for TT-Rec embedding tables. Algorithm 3 shows how the TT-cores are initialized using our proposed Sampled Gaussian method.

Algorithm 1: Forward prop. of TT-Embedding

while $S\_idx < offsets[m]$ **do**
  $E\_idx = min(offsets[S\_idx + B], offsets[m])$
  for $k = S\_idx$ to E_idx **do**
    $idx[j][k] = i_j^k$ in Eqn(4)
    $a[k] = \&\mathcal{G}_1[idx[1][k]][0]$
    $b[k] = \&\mathcal{G}_0[idx[0][k]][0]$
    $c[k] = \&tr_0[k][0]$
    $a[k + B] = \&\mathcal{G}_2[idx[2][k]][0]$
    $b[k + B] = \&tr_0[k][0]$
    $c[k + B] = \&tr_1[k][0]$
  end
  for j = 0 to d-2 **do**
    ▷ Batched GEMM kernel calls
    $c[jB : (j + 1)B] = a[jB : (j + 1)B] * b[jB : (j + 1)B]$
  end
  ▷ Reduce embedding rows to output
  $output[S\_idx : E\_idx] = \sum_{j=offsets[i]}^{offsets[i+1]} c[j]$
  $S\_idx = E\_idx$
end

Algorithm 2: Backward prop. of TT-Rec Embeddings

while $S\_idx < offsets[m]$ **do**
  $E\_idx = min(offsets[S\_idx + B], offsets[m])$
  Recompute for tr_i's as in Algorithm 1
  for $k = S\_idx$ to E_idx **do**
    $idx[j][k] = i_j^k$ in Eqn(4)
    $a0[k] = \&\mathcal{G}_0[idx[0][k]][0]$
    $b0[k] = \&tr_0[k][0]$
    $c0[k] = \&tr\_\mathcal{G}_1[k][0]$
    $a1[k] = \&tr_0[k][0]$
    $b1[k] = \&\mathcal{G}_1[idx[1][k]][0]$
    $c1[k] = \&tr\_\mathcal{G}_0[k][0]$
    $a0[k + B] = \&tr_0[k][0]$
    $b0[k + B] = \&dx$
    $c0[k + B] = \&tr\_\mathcal{G}_2[k][0]$
    $a1[k + B] = \&dx$
    $b1[k + B] = \&\mathcal{G}_2[idx[2][k]][0]$
    $c1[k + B] = \&tr_0[k][0]$
  end
  for j = d-2 to 0 **do**
    ▷ Batched GEMM calls to compute $\partial\mathcal{G}_j$
    $c0[jB : (j + 1)B] = a0[jB : (j + 1)B] * b0[jB : (j + 1)B]$
    ▷ Batched GEMM calls to compute ∂x
    $c1[jB : (j + 1)B] = a1[jB : (j + 1)B] * b1[jB : (j + 1)B]$
    $\partial\mathcal{G}_j[idx[k]] += tr\_\mathcal{G}_j[k]$
  end
  $start\_idx = end\_idx$
end

Algorithm 3: Sampled Gaussian initialization.

for d = 0 to tt-dim **do**
  $\mathcal{G}_d$ = random.normal(0,1)
  for each entry $\mathcal{G}_d(i, j, k, l)$ in $\mathcal{G}_d$ **do**
    while $\mathcal{G}_d(i, j, k, l) \leq 2$ **do**
      $\mathcal{G}_d(i, j, k, l)$ = random.normal(0,1)
  $\mathcal{G}_d / = (\sqrt{1/3n})^{1/d}$