The trade-offs of model size in large recommendation models : A 10000 × compressed criteo-tb DLRM model (100 GB parameters to mere 10MB)

Anonymous Author(s)
Affiliation
Address
email

Abstract

Embedding tables dominate industrial-scale recommendation model sizes, using up to terabytes of memory. A popular and the largest publicly available machine learning MLPerf benchmark on recommendation data is a Deep Learning Recommendation Model (DLRM) trained on a terabyte of click-through data. It contains 100GB of embedding memory (25+Billion parameters). DLRMs, due to their sheer size and the associated volume of data, face difficulty in training, deploying for inference, and memory bottlenecks due to large embedding tables. This paper analyzes and extensively evaluates a generic parameter sharing setup (PSS) for compressing DLRM models. We show theoretical upper bounds on the learnable memory requirements for achieving $(1 \pm \epsilon)$ approximations to the embedding table. Our bounds indicate exponentially fewer parameters suffice for good accuracy. To this end, we demonstrate a PSS DLRM reaching 10000× compression on criteo-tb without losing quality. Such a compression, however, comes with a caveat. It requires 4.5 × more iterations to reach the same saturation quality. The paper argues that this tradeoff needs more investigations as it might be significantly favorable. Leveraging the small size of the compressed model, we show a 4.3× improvement in training latency leading to similar overall training times. Thus, in the tradeoff between system advantage of a small DLRM model vs. slower convergence, we show that scales are tipped towards having a smaller DLRM model, leading to faster inference, easier deployment, and similar training times.

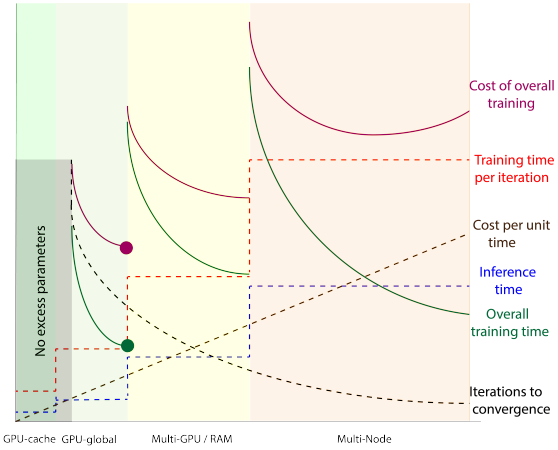
1 Introduction

Recently, recommendation systems have emerged as one of the largest workloads in machine learning [1]. Recommendation systems form the backbone of a good user experience on online platforms such as e-commerce and web search, where there is a flood of information. Thus, considerable effort goes into building recommendation systems. Deep learning recommendation models give a state-of-the-art performance. However, recommendation models suffer from a critical challenge - sparse features with millions of categorical values[2, 3] These state-of-the-art [2, 4, 5, 6, 7, 8] methods learn a dense representation of the categorical values in a parameter structure called *embedding table*.

Most parameters in recommendation models come from embedding tables. For example, in the popular criteo-tb MLPerf benchmark model, the embedding tables are around 100GB, whereas other parameters only amount to 10MB. Industrial-scale recommendation models are one of the largest models built. The size of the embedding table can go as large as hundreds of terabytes. For example, a research article from Facebook discusses the training of a model of size 50TB over 128 GPUs [3]. The scale of these models leads to some unfavorable effects - slower inference time, slower training time per iteration, and significant engineering challenges in training/deployment. In light of these issues, many works have investigated learning of compressed representation of embedding tables

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

Figure 1: This not-to-scale illustration shows tradeoffs of having larger parameters (and hence the memory footprint of the model). Each line has its own scale. As we move towards the right, memory increases, and we need to have memory farther away from computational hardware. This increases the latency and thus inference time and training time per iteration. The energy cost also increases due to larger active hardware. We observe that the convergence becomes faster as the number of parameters increases. The solid lines show the overall training time and costs, showing that more parameters are not always better.



using various principles : (1) compositional embeddings [9] (2) exploiting power-law in the observed frequencies of tokens [10, 11, 12, 13, 14, 15] (3) low-rank decomposition [16], and weight sharing methods [17, 18]. Parameter-sharing methods will be the focus of our paper.

The general idea in machine learning has primarily shifted to larger models and more data. Larger models lead to better capacity, faster convergence, and better generalization. However, the community is realizing that the current route to DL success is unsustainable[19]. It stands to reason that we must proceed cautiously on this path of building larger models. The large-scale nature of DLRM comes from blown-up embedding parameters - a seemingly inadvertent effect of naive usage of embedding tables. This paper evaluates the tradeoffs between smaller models achieved by parameter sharing methods and large models using naive embedding tables. We find that compressed DLRM models have seemingly no downside in training efficiency and quality.

This paper provides a theoretical answer to the question, "How much can embedding tables be compressed without significantly losing quality." We present a general parameter sharing setup (PSS) for embedding tables and prove that we only require memory logarithmic $(\mathcal{O}(1/\epsilon^2 \max(d^2, d.\log(n))))$ in n to approximate an embedding table $E \in R^{n \times d}$ up to a factor of ϵ for subsequent computations. This result motivates us to evaluate the extents of compression that can be achieved for embedding tables. We find that we can obtain 10000× compression (10MB embedding tables) on the criteo-tb dataset to achieve the same target quality - an order of magnitude higher compression than previously reported. These small models improve inference times and training time per iteration, lower training costs, eliminate engineering challenges and make the models easy to deploy on low-resource devices. While large compression is possible in theory and as we will show in our results, one important aspect of training compressed models needs to be solved - slow convergence.

A general observation in machine learning is that the number of iterations required to converge to a target quality decreases with the increase in parameters. We also observe this effect in the training of parameter-shared recommendation models. The higher the compression, the slower the convergence: a concern since training times are one of the significant aspects of the model. However, a highly compressed model completely changes the storage of model parameters (e.g., multiple nodes to a single node, RAM to GPU, etc.). Hence, we observe a steep decrease in the latency of embedding lookup/gradient update operations. An illustration of this tradeoff is presented in figure 1. The question of how this improvement in latency compensates for increased training iterations is worth investigating. Unfortunately, existing parameter-sharing methods do not demonstrate much training time per iteration improvement due to various reasons such as high algorithmic complexity, poor cache efficiency, and sub-optimal implementation strategies. This paper talks about various aspects of implementation. With our PSS, we demonstrate that, indeed, the advantage of faster training time

per iteration almost compensates for the slower convergence in compressed models. For example a 1,000× compressed model trains in $1.1\times$ time whereas a 10,000× compressed model trains at $1.04\times$ time as compared to original model.

This paper provides a theoretical peek into the compression of large recommendation models. Also, it resolves one of the major roadblocks in training compressed recommendation models by leveraging the system advantage of compressed models. We show 10000× compression (10× higher than best known) on DLRM on criteo-tb, making the total model size 20MB. Such a small DLRM can now be easily deployed on client devices that are resource constrained.

2 Theory of parameter-shared embedding tables

2.1 General parameter sharing embedding table setup

Definition 1 (**Parameter shared setup**). *Under parameter shared setup (PSS) for embedding table of size $n \times d$, we have a set of weights M and recovery function $\mathcal{M} : R^{|M|} \times \{0, .., n-1\} \to R^d$. The embedding of a token $i \in \{0, .., n-1\}$ is recovered as $\mathcal{M}(M, i)$*

M can be represented with any memory layout(eg. 2D/1D array). We denote the size of M as $|M|$ and memory to store $\mathcal{M}$ as $|\mathcal{M}|$. Note the inherent trade-off between $|M|$ and $|\mathcal{M}|$. We can make M large in order to get a very simple $\mathcal{M}$ (for example, M=E and $\mathcal{M}(M, i)$=$M[i]$) or we can keep M very small at the cost of complicated $\mathcal{M}$ (for example M = [0,1] and $\mathcal{M}$ combines these bits to get bit representation of E). An effective PSS for embedding table $E \in R^{n\times d}$ would have $(|M| + |\mathcal{M}|) \ll nd$. In context of training a PSS in an end-to-end manner, we additionally want (1) $\mathcal{M}$ is differentiable (2) $\mathcal{M}(M, i)$ computation is cheap.

One of the key questions we want to answer is how small a PSS can we create for a ground truth embedding table E? i.e. how small can $(|M| + |\mathcal{M}|)$ be. For an arbitrary E, it is impossible to encode E in sublinear memory accurately. So, in the next section, we define a $(1 \pm \epsilon)$ PSS

2.2 $(1 \pm \epsilon)$ parameter sharing setup for table E

While creating an alternative representation of embedding table E, let us see what we would like to achieve. First, we would like to preserve all the pairwise inner products between two embeddings of tokens from the embedding table. Additionally, we also want to minimally affect the subsequent computations performed on the embeddings retrieved from the embedding table. In recommendation models, the embeddings retrieved are passed through neural network layers, which will perform inner products on embeddings. With this in mind, we propose the following definition of approximate PSS for embedding the table.

Definition 2 (**$(1 \pm \epsilon)$ parameter sharing setup**). *A PSS $(M, \mathcal{M})$ is an $(1 \pm \epsilon)$ PSS for an embedding table E if the following holds,*

$$\begin{aligned} &\forall x \in R^d \ \forall i \in \{0, .., n-1\} \\ &\quad |\langle \mathcal{M}(M, i), x\rangle - \langle E[i], x\rangle| \le \epsilon \|x\|_2 \end{aligned}$$

We note that the $(1 \pm \epsilon)$ PSS for embedding table E also preserves the pairwise distances between the embeddings of different tokens, as is mentioned in the following result.

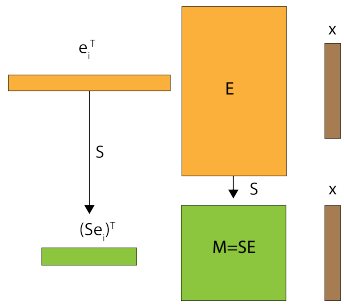
Theorem 3. *Let $(M, \mathcal{M})$ is a $(1 \pm \epsilon)$ PSS for embedding table E, we have ,*

$$\begin{aligned} &\forall i, j \in \{0, ..., n-1\}, \\ &\mid \ \|\mathcal{M}(M, i) - \mathcal{M}(M, j)\|_2 - \|(E[i] - E[j])\|_2 \ \mid \le 2\epsilon \end{aligned}$$

Proof. Presented in the appendix. We first analyze the norm of recovered embedding from PSS and then extend the result to pairwise distances. □

Naive application of JL-Lemma will give us an existence proof of memory M of size $n \log(n)$. In the next section, we show the existence of M with size $dlog(n)$, which is much smaller. Also, note that the condition for $(1 \pm \epsilon)$ PSS is quite different from that of l2-subspace embedding [20]. As we shall see in the proof, this leads to a logarithmic dependence on n (whereas l2-subspace embeddings for $A \in R^{n\times d}$ has no dependence on n) Also, as we shall see, the considerations in $(1 \pm \epsilon)$ PSS are different to those previously discussed in the literature for linear algebra problems.

Figure 2: An i illustration of JLT based setup of PSS, the computation involving E is performed in the sketched-space using much lesser memory of learnable parameters M. The sketching is achieved using sketching matrix S



2.3 $(1 \pm \epsilon)$ PSS via Johnson–Lindenstrauss Transforms

In this section, we answer the question of how small the $|M|$ and $|\mathcal{M}|$ can be. In order to do this, we give a construction of a $(1 \pm \epsilon)$ PSS using the Johnson–Lindenstrauss Transforms(JLT) matrices. We will use the following definition of JLT matrices from [20]

Definition 4. *A randomly generated matrix $S \in R^{k \times n}$ is a $JLT(\epsilon, \delta, f)$ if with probability at least $(1 - \delta)$, for any f-element subset $V \subset R^n$ the following holds,*

$$\forall v_1, v_2 \in V \ |\langle Sv_1, Sv_2 \rangle| \leq \epsilon \|v_1\|_2 \|v_2\|_2$$

The construction of $(1 \pm \epsilon)$ PSS using JLT is presented in the following theorem,

Theorem 5. *Let the embedding table be $E \in R^{n \times d}$. Consider a matrix S which is $JLT(\epsilon/\sigma(E), \delta, 9^d + n))$ where $\sigma(E)$ denotes the maximum singular value of E. Then M and $\mathcal{M}$ defined by*

$$M{=}SE \quad \mathcal{M}(M, i){=}(Se_i)^\top(M)$$

is a $(1 \pm \epsilon) PSS$ with probability $(1 - \delta)$ where $e_i \in R^n$ is a one-hot encoding of integer i (i.e $e_i[i] = 1$ and rest all elements of e_i are 0)

Proof. The complete proof is presented in the appendix. The intuition is that we want to maintain the computations of the form $e_i^\top Ex$, which can be seen as an inner product between $e_i \in R^n$ and $Ex \in R^n$. So the inner products between vectors of sets $A = \{e_0, ..., e_{n-1}\}$ and column space of E need to be preserved. Once we observe this, we apply JLT to the set of n discrete points and the $1/2 - net$ of column space of E which contains less than 9^d points, and get our result. □

Note that when using a JLT matrix S as described in theorem 5, the storage cost of memory M is $|M| = |SE| = kd$. For the mapping $\mathcal{M}$, we only need to store the matrix S. Thus $|\mathcal{M}| = |S|$. The cost of storing S will depend on the exact choice of JLT. Since (Se_i) is a $\mathcal{O}(1)$ operation as e_i is a one-hot encoded vector, the cost of $\mathcal{M}(M, i)$ is just that of vector-matrix multiplication between (Se_i) and M. Again, the exact cost depends on the choice of S. Using the random matrix formed with i.i.d normal variables, the most basic form of JLT, we can give the following bound on the memory of learnable parameters M

Corollary 6. *Let E be the embedding table under consideration. Also, let $S \in R^{k \times n}$ matrix where each element of S is i.i.d $\mathcal{N}(0, \frac{1}{k})$ with $k = \Omega(\sigma^2(E)/\epsilon^2(log(1/\delta) + \max(d, log(n)))$, then with a probability $(1 - \delta)$, $(M = SE, \mathcal{M}(M, i) = (Se_i)^\top(M))$ is a $(1 \pm \epsilon)$ PSS for E.*

Thus we obtain an upper bound on M. $|M| = \mathcal{O}(1/\epsilon^2 \max(d^2, d.log(n)))$ when using the standard normal matrix as JLT for embedding tables with bounded singular values. In this case, the mapping storage cost is $|\mathcal{M}| = kn$ as the matrix S is dense. The cost of applying $\mathcal{M}$ will be kd and hence, $\mathcal{O}(1/\epsilon^2 \max(d^2, d.log(n)))$. In the next section, we will see how to reduce the cost of storing S in practice. This logarithmic dependence on n in the case of embedding tables with very large n explains why we can achieve a very high compression of 10000× for the criteo-tb dataset.

2.4 $(1 \pm \epsilon)$ PSS in practice

$(1 \pm \epsilon)$ PSS from a learned embedding table E The above discussion directly gives us an algorithm to compute a $(1 \pm \epsilon)$ of a trained embedding table E. We just need to compute $M = SE$ and while retrieving an embedding compute $\mathcal{M}(M, i) = (Se_i)^\top M$.

Training end-to-end $(1 \pm \epsilon)$ PSS for embedding table E We can also directly train an $(1 \pm \epsilon)$ PSS in an end-to-end manner. The idea is to directly train the compressed M. Thus, we have a matrix of learnable weights $M \in R^{k \times d}$. Let us now look at how the forward and backward pass of embedding retrieval mapping $\mathcal{M}$ looks like

$$\text{forward(i)} = (Se_i)^\top M$$

The forward function takes an integer i and returns a $R^{1 \times d}$ array which is embedding of i.

$$\text{backward(i}, \Delta) = (Se_i)\Delta$$

The backward pass takes in all the arguments of the forward pass along with the $\Delta \in R^{1 \times d}$ which is gradients of loss with respect to the output of the forward pass. We can back-propagate further to W using the above formulation. The result of the backward pass is a $k \times d$ matrix.

If we use a sparse S such as with sparse JLT, we can achieve sparse gradients of W. That is, only a few entries of the gradient for W are non-zero. As we will see in section 3, whether to propagate sparse or dense gradients is a vital implementation choice. We were able to achieve better training per iteration by exploiting this choice.

Cost considerations for PSS using JLT. In using PSS in end-to-end training, we would be learning the memory M while keeping the mapping $\mathcal{M}$ constant. Thus, unlike in sketching for linear algebra [20], we care about the costs of (1) storage of learnable parameters $|M|$, (2) Cost of storing mapping $|\mathcal{M}|$, and (3) cost of computing $\mathcal{M}(M, i)$. It is clear that using standard normal JLT is not feasible due to storage of matrix S will be more expensive than storing E itself! There are a lot of sparse JLT(ϵ, δ, f) matrices S which will reduce the storage cost of S [21, 22, 23]. [23] showed that the matrix needs to have a minimum column sparsity of $\Omega(\epsilon^{-1} log(f/\delta) log(1/\epsilon))$ and hence the cost of storing S while using JLT matrices is lower bounded by $\Omega(n\epsilon^{-1} log((n + 9^d)/\delta) log(1/\epsilon))$ which can still be considerably high. In the next section, we see relaxations of conditions on S for practical purposes.

2.5 Practical PSS and existing SOTA methods

This section will briefly review practical relaxations of PSS defined above.

Sparse JL with independence relaxation: The issue with storage costs of S can be alleviated in practice by relaxing the complete independence condition on entries of S. For example, the JLT matrix (which requires the same bound on k as given in theorem 5) suggested by [21], which selects each entry to be 0 with probability 2/3, ±1 with probability 1/6 independently can be generated on-the-fly using universal hashing. [24] analyze why simpler hash functions can work well with data having enough entropy. The benefit of using universal hashing is that mapping $\mathcal{M}$ can now be stored in $O(1)$ memory. Thus using the [21] sparse matrix, we can obtain $(1 \pm \epsilon)$ PSS using total setup memory of $|M| + |\mathcal{M}| = \mathcal{O}(1/\epsilon^2 \max(d^2, d.log(n)))$ and cost of applying mapping $\mathcal{M}$ to be $\mathcal{O}(d)$ (hash computation and lookups) under valid independence relaxation.

Existing parameter-sharing based embedding compression methods can also be seen as PSS with varied distributions over S. We state a few of them below.

- **Hashing Trick** In this method, entire embeddings for a token i is drawn from a randomly hashed location. In terms of PSS, we can define the mapping function $\mathcal{M}_h$ over a 2D matrix $\mathcal{M}_h \in R^{k \times d}$ where $k < n$.

  $$\mathcal{M}_h(M_h, i)[:] = M_h[h(i), :]$$

  where h is a hash function.
- **QR decomposition**[9]. In this method, embedding for a token i is drawn in chunks from separate memory vectors. In terms of PSS, we can define the mapping function $\mathcal{M}_q$ over, say l, pieces of 2D memory $M_1, M_2, ..M_l \in R^{k \times d/l}$ as follows

  $$\mathcal{M}_q(\{M_1, M_2, ..M_k\}, i)[j * (d/l) : (j + 1) * (d/l)] = M_j[h_j(i), :] \tag{1}$$

  In words, the jth chunk of ith embedding is recovered using the chunk at the location $h_j(i)$ in memory M_r

Table 1: State of the art in embedding compression on criteo datasets. (the ones in grey are not PSS)

Method	Dataset	Compression	Quality
HashingTrick	Crieto Kaggle	4×	worse
QR Trick	Criteo Kaggle	4×	similar/slightly worse
MD Embedding	Criteo Kaggle	16×	better/similar
TT-Rec	Criteo kaggle/Criteo TB	112× / 117 ×	better/simiar
ROBE	Criteo Kaggle/Criteo TB	1000×	better/similar

- **HashedNet [17] and ROBE-Z [18]** : In HashedNets, authors proposed mapping model weights randomly into a parameter array. ROBE-Z extended this idea by hashing chunks of embedding vector instead of individual elements. In terms of PSS, we can define the mapping function $\mathcal{M}_r$ over 1D memory M_r as

$$\mathcal{M}_r(M_r, i)[j * Z : (j+1) * Z] = M_r[h(i,j) : h(i,j) + Z]$$

  In words, the jth chunk of ith embedding is recovered using the chunk at the location $h(i,j)$ from M_r. Here, $h : \mathbf{N}^2 \rightarrow \{0, .., |M_r|\}$ is a hash function. If we set $Z = 1$, then we get the mapping function for HashedNet.

We can summarize the state-of-the-art embedding compression in the table 1. Our PSS theory suggests that we should be able approximate the embedding table in memory of the order of $\mathcal{O}$(log(n). Thus, it is important to question if $1000\times$ compression is the best compression we can achieve on a dataset like CriteoTB where we have large (~400GB sized) embedding tables. As we shall see, one of the significant roadblocks in aiming to achieve higher compression is training time. The higher the compression, the more iterations are needed to train the model to a certain quality. In the next section, we discuss how to overcome this bottle-neck preventing training of highly compressed PSS.

3 Road to 10000× compression with PSS

Table 4 shows the quality of embeddings for the criteo-kaggle dataset over varying compression rates across different popular recommendation models. We can see that the state-of-the-art compression of the criteo-kaggle dataset is $1000\times$. It is a consequence of PSS theory that larger embedding tables (prevalent in industrial scale recommendation models) such as those for criteo-tb should obtain more compression. Our goal is to study higher compression for large embedding tables and improve over existing SOTA (1000x).

A significant roadblock in building highly compressed embedding tables is the slow convergence. Table 4 shows that across different deep learning recommendation models, the number of iterations required to converge increases with higher compression. Specifically, for $1000\times$ compression, it takes up to $4\times$ iterations, and for models with $10000\times$ compression, most models do not converge in 15 epochs. If a model does not converge, it is hard to judge if the convergence is too slow or its capacity is consumed. In order to be able to train for larger iterations, we should be able to train the highly compressed models faster. Indeed, we expect some system benefits with highly compressed models, as shown in figure 1. We find that existing methods cannot train highly compressed embedding tables because of the implementation choices. We propose the following implementation choices.

Hashing chunks is better than hashing elements: As suggested in [18], we find that hashing chunks instead of individual elements is indeed helpful in reducing the latency of PSS systems. As shown in table 3(d,e), both forward and backward passes are affected a lot by chunk sizes. It is also interesting to note that the effect of chunk sizes diminishes after size 16. Hence, we can choose a chunk size of 32 for hashing.

Dense gradients are suitable for highly compressed PSS Generally, in embedding tables, implementations use sparse gradients. Dense gradients are particularly wasteful in embedding tables, where in each iteration, only a few weights are involved in the computation. However, the same does not apply to highly compressed PSS. In the case of PSS, the algorithm for sparse gradient computation is more involved than that for dense gradients. The algorithms for sparse and dense gradient computation are specified in algorithm 1. Also, the scatter operation is much faster when the memory size of embeddings is smaller. Hence, dense gradients work well with higher compressions. From table 3(b,c), we can see that at higher compression rates, the backward pass is up to $8\times$ faster with dense gradients as compared to sparse gradients.

Figure 3: These experiments are run on a single GPU for a simple embedding lookup and loss is taken to be sum of all retrieved elements with a batch size of 10240 and embedding dimension=128. ■ higher latency (worse) ■ lower latency (better)

(a) Fixed chunk size (16)

Forward pass (ms) [sparse=false]				
	compression			
n	10x	100x	1000x	10000x
4M	0.40	0.31	0.27	0.27
8M	0.36	0.32	0.29	0.27
16M	0.37	0.31	0.30	0.27
32M	0.37	0.25	0.31	0.26
64M	0.37	0.37	0.30	0.28
128M	0.37	0.37	0.32	0.29
256M		0.36	0.31	0.30

(b) Fixed chunk size (16)

Backward pass (ms) [sparse=false]				
	compression			
n	10x	100x	1000x	10000x
4M	2.55	0.59	0.35	0.35
8M	4.81	0.83	0.37	0.35
16M	9.15	1.27	0.41	0.34
32M	17.81	2.13	0.54	0.35
64M	35.10	3.85	0.73	0.36
128M	69.73	7.42	1.08	0.40
256M		14.34	1.78	0.47

(c) Fixed chunk size (16)

Backward pass (ms) [sparse=true]				
	compression			
n	10x	100x	1000x	10000x
4M	2.67	2.48	2.56	2.52
8M	2.53	2.58	2.48	2.53
16M	2.52	2.68	2.58	2.45
32M	2.51	2.64	2.56	2.55
64M	2.51	2.66	2.63	2.60
128M	2.52	2.65	2.65	2.54
256M	2.54	2.52	2.53	2.62

(d) Fixed compression 1000×

Forward pass (ms) [sparse=false]						
	chunk size					
n	1	2	4	8	16	32
4M	0.68	0.58	0.55	0.32	0.31	0.31
8M	0.75	0.62	0.54	0.33	0.31	0.31
16M	0.77	0.65	0.54	0.54	0.31	0.31
32M	0.80	0.67	0.54	0.26	0.24	0.23
64M	0.92	0.79	0.65	0.39	0.37	0.37
128M	0.88	0.75	0.65	0.39	0.37	0.37
256M	0.88	0.75	0.65	0.40	0.36	0.36

(e) Fixed compression 1000×

Backward pass (ms) [sparse=false]						
	chunk size					
n	1	2	4	8	16	32
4M	1.29	1.10	0.80	0.64	0.58	0.57
8M	1.56	1.38	1.07	0.88	0.82	0.80
16M	2.02	1.87	1.54	1.33	1.27	1.27
32M	2.96	2.79	2.40	2.20	2.12	2.12
64M	4.69	4.54	4.13	3.98	3.85	3.85
128M	8.15	8.00	7.70	7.48	7.43	7.42
256M	15.17	15.01	14.61	14.40	14.34	14.34

(f) Fixed compression 1000×

Backward pass (ms) [sparse=true]						
	chunk size					
n	1	2	4	8	16	32
4M	3.17	3.04	2.61	2.54	2.50	2.48
8M	3.41	2.97	2.64	2.64	2.66	2.61
16M	3.56	3.07	2.74	2.58	2.54	2.62
32M	3.68	3.13	2.73	2.63	2.53	2.50
64M	3.94	3.36	2.91	2.64	2.52	2.64
128M	3.98	3.22	2.77	2.60	2.51	2.60
256M	4.00	3.27	2.74	2.61	2.53	2.50

Algorithm 1 Backward Pass for PSS

Require: $\mathcal{G}_O \in R^{B\times d}, \mathcal{I} \in R^{B\times d}$, sparse $\in \{\text{True}, \text{False}\}$, M: compressed memory.

$\triangleright$ $\mathcal{G}_O$ is the gradient w.r.t output, and $\mathcal{I}$ is the mapping matrix showing the locations from which the output was accessed

Ensure: $\mathcal{G}_i$ is gradient w.r.t input.

if sparse **then**

$\text{unique}, \text{invIdx} \leftarrow \text{findUnique}(\mathcal{I}.\text{flatten}(), \text{returnInverse} = \text{True})$

$g \leftarrow \text{Tensor}((\text{unique.shape},))$

$g.\text{scatter_add}(\text{invIdx}, \mathcal{G}_o.\text{flatten}()$

$\mathcal{G}_i = \text{sparseTensor}(\text{location} = \text{unique}, \text{values} = \text{g}, \text{size} = \text{M.size})$

else if dense **then**

$\mathcal{G}_i \leftarrow \text{Tensor}(\text{M.size})$

$\mathcal{G}_i.\text{scatter_add}(0, \mathcal{I}.\text{flatten}(), \mathcal{G}_o.\text{flatten}())$

end if

Forward/Backward kernel optimizations Latency for PSS, and CUDA kernels in general, is very sensitive to the usage of shared memory, occupancy, and communication between CPU and GPUs. We also optimize our PSS code to minimize the data movement costs, implement custom kernels to fuse operations to improve shared memory usage and optimize CUDA grid sizes to obtain the best performance.

With the implementation choices mentioned above, we are ready to train a $10000\times$ compressed PSS for criteo-tb. We implement our PSS using ROBE-style hashing.

4 Experimental results on Criteo datasets

We perform experiments on criteo-kaggle and criteo-tb datasets in order to confirm the following hypothesis,

1. Theory in section 2 dictates that embedding table ($E \in R^{n\times d}$) can be represented in memory logarithmic in n. High compression should be possible in these datasets. Specifically, higher compression should be possible in larger embedding tables.
2. The system advantage of smaller PSS should compensate for the convergence advantage of the original model.

Datasets: Criteo-kaggle and criteo-tb datasets have 13 integer and 26 categorical features. criteo-kaggle data was collected over seven days, whereas the criteo-tb dataset was collected over 23 days. criteo-tb is one of the largest recommendation datasets in the public domains with around 800 million token values making the embedding tables of size around 400GB. (with d = 128). criteo-kaggle is smaller and has a 2GB sized embedding table. Note that industrial-scale models are much larger than

Figure 4: Quality and convergence on criteo-kaggle of 5 popular models vs. compression. The standard deviation of all AUC results is within 0.0009.

(a) Epochs needed to converge to target AUC set to minimum of value achieved by original model. (max 15 epochs)

Training epochs for reaching desired quality of model (criteo-kaggle)						
	TargetAUC	**ORIG**	**10x**	**100x**	**1000x**	**10000x**
DLRM	0.8029	1.00	1.52	1.78	2.78	-
DCN	0.7973	1.00	1.00	1.87	1.93	-
AUTOINT	0.7968	1.00	1.00	2.00	2.50	14.93
DEEPFM	0.7951	1.00	1.00	1.00	3.93	12.68
XDEEPFM	0.7989	1.62	1.50	1.74	3.93	-
FIBINET	0.8011	1.00	0.93	1.00	2.99	-

(b) Quality of models for criteo-kaggle dataset is largely maintained till 1000x compression after which either the convergence is too slow / capacity of model drops (max 15 epochs)

Criteo-kaggle quality vs excess parameters						
	DLRM	**DCN**	**AUTOINT**	**DEEPFM**	**XDEEP**	**FIBINET**
ORIG	0.8031	0.7973	0.7968	0.7957	0.8007	0.8016
10x	0.8032	0.7982	0.7972	0.7961	0.8016	0.8021
100x	0.8029	0.7978	0.7968	0.7953	0.7998	0.8011
1000x	0.8048	0.7991	0.7987	0.7951	0.7989	0.8011
10000x	0.8001	0.7967	0.7957	0.7943	0.795	0.7963

Figure 5: Quality and convergence on criteo-tb with DLRM vs. compression. The standard deviation of AUC results is within 0.0009.

(a) Epochs required to reach 0.8025 AUC for DLRM model on criteo-tb dataset. Relative time computes the ratio of A*B to original model

		orig	$10^2\times$	$10^3\times$	$10^4\times$
A	Epochs	1.00	1.94	1.9	4.35
B	Time /1000 itr	50.6	31.04	29.6	11.6
A*B	Relative Time	1	1.19	1.11	1.04

(b) $10^4\times$ compression also reaches the target AUC. For $10^5\times$ convergence is too slow/mode capacity is reached. (max 15 epochs)

0.8025 AUC	orig	$10^2\times$	$10^3\times$	$10^4\times$	$10^5\times$
DLRM	Yes	Yes	Yes	Yes	No

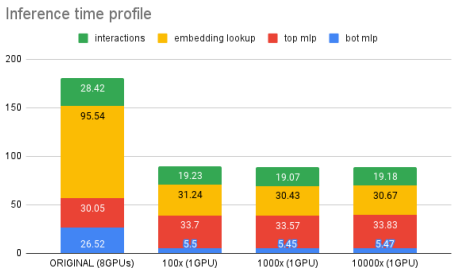
the DLRM model we talk about here. For example, Facebook recently published a model sized 50TB [3]. One can extrapolate the benefits of PSS to industrial-scale models based on this case study.

Models: Facebook MLPerf DLRM[2] model, available under Apache-2.0 license, for the criteo-tb dataset achieves the target AUC (0.8025) with the embedding memory of around 100GB. This model uses a maximum cap of 40M indices per embedding table, leading to a total of 204M embeddings. This model cannot be trained on a single GPU (like V100) and is trained using a multiple GPUs (4 or 8). For the criteo-tb dataset, we use the DLRM MLPerf model. For criteo-kaggle dataset, we use an array of state-of-the-art models DLRM[2], DCN[4], AUTOINT[5], DEEPFM [6], XDEEPFM [8] and FIBINET. We use our PSS implementation described in 3 as a compressed model.

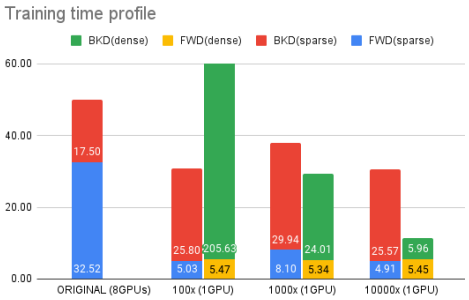
Quality of model vs. Excess parameters: Tables 4(b) and 5(b) show the results for the two datasets across different values of compression. In table 4(b), we can see that across different models, the quality of the model is maintained until 1000× compression. As criteo-tb embedding tables are much larger than the criteo-kaggle dataset, according to the section 2, we should see larger values of compression. Indeed, we obtain a 10000× compression without loss of quality of the model for criteo-tb. This level of compression is unprecedented in embedding compression literature. These experiments validate our first hypothesis and provide a new state-of-the-art embedding compression. In the rest of the section, we evaluate how the system advantage of PSS compares against the original embeddings in various aspects.

Inference time In figure 6a, we compare the inference time of test data (89M samples) in the criteo-tb dataset with a batch size of 16384 with PSS on a single Quadro RTX-8000 GPU and the original model on 8 Quadro RTX-8000 GPUs. The total time for inference for the original model is around 203 seconds, whereas PSS is around 97 seconds. In the original model trained on 8GPUs, embedding lookup is model-parallel, whereas the rest of the computation (bot-mlp, top-mlp, and interactions) is data-parallel. There is a steep improvement of over 3× in embedding time lookup as we move from a distributed GPU setup for embedding tables to a PSS on a single GPU. The extra computation time in PSS is much smaller than all-to-all communication costs in the original embedding table lookup. We include the time required for data distribution (initial) in bot-mlp and data-gather (final) in top-mlp timings. This communication cost is high, and we can see that we are better off performing the entire computation on a single GPU.

Model training time per iteration vs excess parameters Figure 6b records the time-taken for 1000 iterations of training. While using embedding tables, we can choose to back-propagate sparse or dense embedding gradients. The general idea is to use sparse gradients when very few gradients are non-zeros. In the original model, we can only use sparse gradients as using dense gradients is prohibitive w.r.t to computation and communication. In PSS, however, we compare both modes for gradient back-propagation. We see that dense gradients perform exceptionally well at higher rates of compression. The performance of sparse gradients is constant across different compression rates as the workload is similar. Generally, it seems good to use dense gradients for PSS when the effective



(a) [criteo-tb] DLRM inference time taken for over 89M samples with a batch size of 16384. Communication causes embedding lookup for original model much slower than that for PSS models when entire model is located on a single GPU



(b) DLRM training time for 1000 iterations of training with a mini-batch size of 2048 and SGD optimizer

memory size (i.e., the final memory of compressed embedding tables) is small. It is noteworthy that the training time per iteration reduces significantly with higher compression. For example, with 1000x compression, the training time is 1.7× lesser than the time taken by the original model, whereas with 10000x compression, the time per iteration is 4.37 × lesser.

Model convergence and overall time vs. excess parameters We observe a consistent trend in convergence: as the number of excess parameters in the models reduces, the convergence becomes slower. This can be seen across models and datasets as shown in tables 4(a) and 5(a). As an example, table 5 shows that with 10000× compression, we require 4.55 epochs as compared to the original model's one epoch. This might seem unfavorable at first. However, as seen in the previous section, there is a significant gain in training time per iteration. As seen in table 5, this gain largely compensates for the disadvantage in terms of convergence, making the overall training times similar. For example, while we need 4.55 epochs with 10000× compressed PSS, the training time per iteration goes down by a factor of 4.37. Hence, the overall training time is only 1.04× original time.

Engineering challenges and costs vs. excess parameters: The excess parameters in a recommendation model primarily appear due to the construction of embedding tables. The industrial-scale embedding tables can go as large as hundreds of terabytes. With the increase in the model's size, the model needs to be distributed across different nodes and GPUs. The complexities of efficiently running a distributed model training include many considerations such as non-uniform memory allocation and communication costs. The increasing literature detailing the engineering solutions to training such models is evidence of the fact that training such models require engineering ingenuity to solve the challenges involved [3, 25, 26, 27, 28, 29]. For example, in [3], the authors detail their solution to train a model of size 50TB on a new distributed system. Similarly, in [25], authors talk about their optimizations of recommendation models on CPU clusters. In this paper, we argue that the large-scale nature of embedding-based recommendation models appears due to embedding tables where a complete $n \times d$ table with blown up n does not add value to model capacity while adding significant engineering challenges. Distributed systems also imply significant energy costs. PSS can avoid these downsides by fitting entire learnable parameters on a single GPU/node.

5 Limitations and future work

In this paper, we show that we can compress the embedding tables for CTR prediction models as much as 10000×. The theory presented in the paper shows promise of generalizing these results to other models such as retrieval or ranking problems. However, more experiments are needed to evaluate PSS approaches in these problems. We evaluate CTR models in this paper and extensions of this work to other models is left for future work.

6 Conclusion

Today's prevalent idea in deep learning(DL) is overparameterized models with stochastic iterative learning. While this paradigm has led to success in DL, the sustainability of this route to success is a grim question [19]. We evaluate the value of embedding-based excess parameters in DLRM models. We conclude that compressed DLRM models are better for fast inference and are easy to train and deploy as expected. More importantly, they can also be trained in the same overall time despite having a slower convergence rate. The critical observation is that compressed models enjoy a

system advantage they can exploit, which reduces time per iteration. The paper also highlights that the correct way to compare the quality of models of highly different sizes is to run the models for equal time rather than equal iterations.

References

[1] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. The architectural implications of facebook's dnn-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–501. IEEE, 2020.

[2] 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. Deep learning recommendation model for personalization and recommendation systems. arXiv:1906.00091, 2019.

[3] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, et al. High-performance, distributed training of large-scale deep learning recommendation models. *arXiv preprint arXiv:2104.05158*, 2021.

[4] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. Deep & cross network for ad click predictions. In *Proceedings of the ADKDD*, 2017.

[5] Weiping Song, Chence Shi, Zhiping Xiao, Zhijian Duan, Yewen Xu, Ming Zhang, and Jian Tang. Autoint: Automatic feature interaction learning via self-attentive neural networks. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 1161–1170, 2019.

[6] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. Deepfm: a factorization-machine based neural network for ctr prediction. *arXiv preprint arXiv:1703.04247*, 2017.

[7] Tongwen Huang, Zhiqi Zhang, and Junlin Zhang. Fibinet: combining feature importance and bilinear feature interaction for click-through rate prediction. In *Proceedings of the 13th ACM Conference on Recommender Systems*, pages 169–177, 2019.

[8] Jianxun Lian, Xiaohuan Zhou, Fuzheng Zhang, Zhongxia Chen, Xing Xie, and Guangzhong Sun. xdeepfm: Combining explicit and implicit feature interactions for recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1754–1763, 2018.

[9] Hao-Jun Michael Shi, Dheevatsa Mudigere, Maxim Naumov, and Jiyan Yang. Compositional embeddings using complementary partitions for memory-efficient recommendation systems. arXiv:1909.02107, 2019.

[10] Antonio Ginart, Maxim Naumov, Dheevatsa Mudigere, Jiyan Yang, and James Zou. Mixed dimension embeddings with application to memory-efficient recommendation systems. arXiv:1909.11810, 2019.

[11] Manas R Joglekar, Cong Li, Mei Chen, Taibai Xu, Xiaoming Wang, Jay K Adams, Pranav Khaitan, Jiahui Liu, and Quoc V Le. Neural input search for large scale recommendation models. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2387–2397, 2020.

[12] Haochen Liu, Xiangyu Zhao, Chong Wang, Xiaobing Liu, and Jiliang Tang. Automated embedding size search in deep recommender systems. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2307–2316, 2020.

[13] Siyi Liu, Chen Gao, Yihong Chen, Depeng Jin, and Yong Li. Learnable embedding sizes for recommender systems. *arXiv preprint arXiv:2101.07577*, 2021.

[14] Weiyu Cheng, Yanyan Shen, and Linpeng Huang. Differentiable neural input search for recommender systems. *arXiv preprint arXiv:2006.04466*, 2020.

[15] Xiangyu Zhao, Chong Wang, Ming Chen, Xudong Zheng, Xiaobing Liu, and Jiliang Tang. Autoemb: Automated embedding dimensionality search in streaming recommendations. *arXiv preprint arXiv:2002.11252*, 2020.

[16] Chunxing Yin, Bilge Acun, Carole-Jean Wu, and Xing Liu. Tt-rec: Tensor train compression for deep learning recommendation models. *Proceedings of Machine Learning and Systems*, 3, 2021.

[17] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *International conference on machine learning*, pages 2285–2294. PMLR, 2015.

[18] Aditya Desai, Li Chou, and Anshumali Shrivastava. Random offset block embedding array (robe) for criteotb benchmark mlperf dlrm model: 1000× compression and 2.7× faster inference. *arXiv preprint arXiv:2108.02191*, 2021.

[19] Neil C Thompson, Kristjan Greenewald, Keeheon Lee, and Gabriel F Manso. Deep learning's diminishing returns: The cost of improvement is becoming unsustainable. *IEEE Spectrum*, 58(10):50–55, 2021.

[20] David P Woodruff. Sketching as a tool for numerical linear algebra. *arXiv preprint arXiv:1411.4357*, 2014.

[21] Dimitris Achlioptas. Database-friendly random projections: Johnson-lindenstrauss with binary coins. *Journal of computer and System Sciences*, 66(4):671–687, 2003.

[22] Anirban Dasgupta, Ravi Kumar, and Tamás Sarlós. A sparse johnson: Lindenstrauss transform. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 341–350, 2010.

[23] Daniel M Kane and Jelani Nelson. Sparser johnson-lindenstrauss transforms. *Journal of the ACM (JACM)*, 61(1):1–23, 2014.

[24] Michael Mitzenmacher and Salil P Vadhan. Why simple hash functions work: exploiting the entropy in a data stream. In *SODA*, volume 8, pages 746–755. Citeseer, 2008.

[25] Dhiraj Kalamkar, Evangelos Georganas, Sudarshan Srinivasan, Jianping Chen, Mikhail Shiryaev, and Alexander Heinecke. Optimizing deep learning recommender systems training on cpu cluster architectures. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[26] Biye Jiang, Chao Deng, Huimin Yi, Zelin Hu, Guorui Zhou, Yang Zheng, Sui Huang, Xinyang Guo, Dongyue Wang, Yue Song, et al. Xdl: an industrial deep learning framework for high-dimensional sparse data. In *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data*, pages 1–9, 2019.

[27] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, et al. Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications. *arXiv preprint arXiv:1811.09886*, 2018.

[28] Maxim Naumov, John Kim, Dheevatsa Mudigere, Srinivas Sridharan, Xiaodong Wang, Whitney Zhao, Serhat Yilmaz, Changkyu Kim, Hector Yuen, Mustafa Ozdal, et al. Deep learning training in facebook data centers: Design of scale-up and scale-out systems. *arXiv preprint arXiv:2003.09518*, 2020.

[29] Jie Amy Yang, Jongsoo Park, Srinivas Sridharan, and Ping Tak Peter Tang. Training deep learning recommendation model with quantized collective communications. 2020.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] It is clear in theory
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] see section 6
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes]
   (b) Did you include complete proofs of all theoretical results? [Yes] see supplementary
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [No] Our code is a set of patches on exisitng codes and hence is difficult to package. we will publish links when accepted
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] yes in supplementary
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] See table captions
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] In experimental section
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [Yes] In models section in experiments
   (c) Did you include any new assets either in the supplemental material or as a URL? [No]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Appendix

B Proof of Theorem 5

Let the embedding table be $E \in R^{n \times d}$. Consider a matrix S which is $JLT(\epsilon/\sigma(E), \delta, 9^d + n))$ where $\sigma(E)$ denotes the maximum singular value of E. Then M and $\mathcal{M}$ defined by

$$M{=}SE \quad \mathcal{M}(M,i){=}(Se_i)^{\top}(M)$$

is a $(1 \pm \epsilon)PSS$ where $e_i \in R^n$ is a one-hot encoding of integer i (i.e $e_i[i] = 1$ and rest all elements of e_i are 0)

Proof. Johnson Lindenstrauss transforms are a class of transformations which preserve pair wise inner products (equivalently norms) of a set of vectors in a space Definition of JLT transform from [20] is A random matrix $S \in R^{k \times n}$ forms a $JLT(\epsilon, \delta, f)$ if with probability $(1 - \delta)$ for any f-element subset $V \subset R^n$, for all v_1, v_2 it holds that

$$|\langle Sv_1, Sv_2 \rangle - \langle v_1, v_2 \rangle| \leq \epsilon \|v_1\|_2 \|v_2\|_2 \tag{2}$$

Choice of S matrix and inner product preservation Consider the following sets of points.

1. $\{e_i\}_{i=0}^{n-1}$
2. $1/2 - net$ over the set $\{u | u \in \text{Column-space}(E)\ \|u\|\|_2 = 1\}$

We use the lemma 5 from [20] to have the number of points in set 2 bounded by 9^d. Let S be a matrix that is JLT($\delta, \epsilon, (n + 9^d)$). For these $(n + 9^d)$ points with probability $(1 - \delta)$, the matrix S preserves inner products as per definition of JLT given above. Let us consider 4 cases w.r.t to the following condition

$$|\langle Sv_1, Sv_2 \rangle - \langle v_1, v_2 \rangle| \leq \epsilon \|v_1\|_2 \|v_2\|_2 \tag{3}$$

1. $v_1, v_2 \in$ Column-space(E). We use the same argument as given on page 12 [20] and conclude that condition holds for these v_1, v_2
2. $v_1, v_2 \in \{e_i\}_{i=0}^{n-1} \cup 1/2 - net$. condition holds due to JLT
3. $v_1 \in \{e_i\}_{i=0}^{n-1}, v_2 \in$ column-space(E). Let $\|v_2\| = 1$. We will prove the condition for this case and for all other non-unit-norm cases will happen due to scaling. Using argumnet from page 12 [20], we can represent v_2 as a sum of vectors in $1/2 - net$.

$$v_2 = v^0 + v^1 + v^2 + \ldots. \text{ where } v^i \in \text{1/2-net} \tag{4}$$

such that $\|v^i\| \leq \frac{1}{2^i}$

$$\begin{aligned}
&|\langle Sv_1, Sv_2 \rangle - \langle v_1, v_2 \rangle| \\
&= |\sum_i (\langle Sv_1, Sv^i \rangle) - \sum_i (\langle v_1, v^i \rangle)| \\
&= |\sum_i (\langle Sv_1, Sv^i \rangle) - (\langle v_1, v^i \rangle)| \\
&\leq \sum_i (\langle Sv_1, Sv^i \rangle) - (\langle v_1, v^i \rangle)| \\
&\leq (\epsilon) \sum_i \|v_1\|_2 \|v^i\|_2 \\
&= \epsilon \sum_i \|v^i\|_2 \\
&\leq \epsilon \sum_i \frac{1}{2^i} \\
&= \epsilon
\end{aligned}$$

Thus, using the matrix S, for all $v1, v2 \in \{e_i\} \cup$ column-space(E) it holds that ,

$$|\langle Sv_1, Sv_2 \rangle - \langle v_1, v_2 \rangle| \leq \epsilon \|v_1\|_2 \|v_2\|_2 \tag{5}$$

proving the statement for $(1 \pm \epsilon)$ PSS Let e_is be one hot encoded vectors such that we have that

$$E[i] = e_i^\top E \tag{6}$$

Let $M = SE$ and $\mathcal{M}(M, i) = (SE)^\top Se_i$

$$\langle \mathcal{M}(M,i), x\rangle - \langle E[i]^\top, x\rangle \tag{7}$$
$$= \langle (SE)^\top Se_i, x\rangle - \langle E[i]^\top, x\rangle \tag{8}$$
$$= \langle Se_i, SEx\rangle - \langle (e_i^\top E)^\top, x\rangle \qquad = \langle Se_i, SEx\rangle - \langle e_i, Ex\rangle \tag{9}$$

Note that both e_i and Ex are vectors that belong to our f point set V. Thus

$$\langle \mathcal{M}(M,i), x\rangle - \langle E[i]^\top, x\rangle \tag{10}$$
$$= \langle Se_i, SEx\rangle - \langle e_i, Ex\rangle \tag{11}$$
$$\leq \epsilon \|e_i\|_2 \|Ex\|_2 \tag{12}$$
$$\leq \epsilon \|Ex\|_2 \tag{13}$$
$$\leq \epsilon \sigma(E) \|x\|_2 \tag{14}$$

Note that $\|e_i\| = 1$ Let σ be the max singular value of E. We can push $\sigma(E)$ into ϵ and select a transform $JLT(\epsilon/\sigma(E), \delta, (n + 9^d))$ to obtain a $\epsilon - l_2$ parameter shared setup.

□

C Proof of theorem 3

Let $(M, \mathcal{M})$ is a $(1 \pm \epsilon)$ PSS for embedding table E, we have ,

$$\forall i, j \in \{0, ..., n-1\} \quad | \ \|\mathcal{M}(M,i) - \mathcal{M}(M,j)\|_2 - \|(E[i] - E[j])\|_2 \ | \leq 2\epsilon$$

Proof. Using definition of PSS

$$|\langle \mathcal{M}(M,i), \mathcal{M}(M,i)\rangle - \langle \mathcal{M}(M,i), E[i]\rangle| \leq \epsilon \|\mathcal{M}(M,i)\|_2$$

$$\langle \mathcal{M}(M,i), \mathcal{M}(M,i)\rangle \leq \langle \mathcal{M}(M,i), E[i]\rangle + \epsilon \|\mathcal{M}(M,i)\|_2$$
$$\text{and } \langle \mathcal{M}(M,i), \mathcal{M}(M,i)\rangle \geq \langle \mathcal{M}(M,i), E[i]\rangle - \epsilon \|\mathcal{M}(M,i)\|_2$$

$$\langle \mathcal{M}(M,i), \mathcal{M}(M,i)\rangle \leq \langle E[i], E[i]\rangle + \epsilon(\|E[i]\|_2 + \|\mathcal{M}(M,i)\|_2)$$
$$\text{and } \langle \mathcal{M}(M,i), \mathcal{M}(M,i)\rangle \geq \langle E[i], E[i]\rangle - \epsilon(\|E[i]\|_2 + \|\mathcal{M}(M,i)\|_2)$$

$$\langle \mathcal{M}(M,i), \mathcal{M}(M,i)\rangle \leq \langle E[i], E[i]\rangle + \epsilon(\|E[i]\|_2 + \|\mathcal{M}(M,i)\|_2)$$
$$\text{and } \langle \mathcal{M}(M,i), \mathcal{M}(M,i)\rangle \geq \langle E[i], E[i]\rangle - \epsilon(\|E[i]\|_2 + \|\mathcal{M}(M,i)\|_2)$$

Let $\|\mathcal{M}(M,i)\| = m$ and $\|E[i]\| = e$

$$m^2 - \epsilon m \leq e^2 + \epsilon e \text{ and } m^2 + \epsilon m \geq e^2 - \epsilon e$$

Adding $(1/4)\epsilon^2$ on both sides

$$m^2 - \epsilon m + 1/4\epsilon^2 \leq e^2 + \epsilon e + \epsilon^2 \text{ and } m^2 + \epsilon m + 1/4\epsilon^2 \geq e^2 - \epsilon e + 1/4\epsilon^2$$

$$(m - 1/2\epsilon)^2 \leq (e + 1/2\epsilon)^2 \text{ and } (m + 1/2\epsilon)^2 \geq (e + 1/2\epsilon)^2$$

$$m \leq (e + \epsilon) \text{ and } m \geq e - \epsilon$$

Thus,

$$|m - e| \leq \epsilon$$

Thus,

$$\|\mathcal{M}(M, i)\|_2 = \|E[i]\|_2 \pm \epsilon \tag{15}$$

Now we can look at the pairwise distances

$$\begin{aligned}
&\|\mathcal{M}(M, i) - \mathcal{M}(M, j)\|_2 \\
&= \|(\mathcal{M}(M, i) - E[i]) - (\mathcal{M}(M, j) - E[j]) + (E[i] - E[j])\|_2 \\
&\leq \|(\mathcal{M}(M, i) - E[i])\|_2 + \|(\mathcal{M}(M, j) - E[j])\|_2 + \|(E[i] - E[j])\|_2 \\
&\leq 2\epsilon + \|(E[i] - E[j])\|_2
\end{aligned}$$

$$\begin{aligned}
&\|\mathcal{M}(M, i) - \mathcal{M}(M, j)\|_2 \\
&= \|(\mathcal{M}(M, i) - E[i]) - (\mathcal{M}(M, j) - E[j]) + (E[i] - E[j])\|_2 \\
&\geq -\|(\mathcal{M}(M, i) - E[i])\|_2 - \|(\mathcal{M}(M, j) - E[j])\|_2 + \|(E[i] - E[j])\|_2 \\
&\geq -2\epsilon + \|(E[i] - E[j])\|_2
\end{aligned}$$

Thus,

$$\left| \ \|\mathcal{M}(M, i) - \mathcal{M}(M, j)\|_2 - \|(E[i] - E[j])\|_2 \ \right| \leq 2\epsilon$$

□

D Data

D.1 PSS rigorous evaluation

- Page 1 : memory vs compression.
- page 2: memory vs chunk size
- page 3: compression vs chunk size

Some obervations

- (sparse/dense) original embeddings should be run with sparse gradients. The dense embeddings are too time consuming as they require updating large amounts of memory (vacuously) in each iteration. Note that a lot of optimizers in deep learning libraries like pytorch do not yet have support for sparse gradient updates.
- When the entire embedding table can fit on the GPU, the forward and backward times do not change much.
- While we can go upto 64M tokens (for m = 128) on single gpu using original embeddings and sparse gradient propogation, the simulated embedding tables can be much larger with PSS and compression.
- Best forward times we can achieve with PSS (around 0.27ms/iteration) is almost 2× the forward times for original embedding lookup. (0.12 ms / iteration). This is expected since, we have to perform additional hash computations in PSS. Hence, if both original embeddings and compressed embeddings are at the same distance from the computational resource, then PSS will have a disadvantage.
- (compression, PSS) Higher the compression, better are timings for both backward pass (using dense propagation) and forward pass.
- A nice trade-off can be seen between using sparse or dense gradients with PSS. If we use dense gradients, the time in backward propogation is affected by the overall size of embedding parameters. Thus, for higher compression and smaller n, we have smaller times where as if n becomes larger and compression is smaller, the time taken increases. The cost of dense gradient propogation becomes quite high at sufficiently large n and sufficiently small compression. On the other hand, the cost of sparse gradient propogation is uniform across different values of n and compression. This is because the algorithmic complexity and memory accessed is similar. This gives us a guideline as to when to use sparse / dense gradients with PSS. Note that in our final results, we see a good overall improvement by leveraging the dense gradient propogation at high compression.
- Higher the chunk size better is the time in forward pass. backward pass is largely unaffected by chunk size largely due to the way it is implemented.

Original time(ms)							
		n					
call	**sparse**	4M	8M	16M	32M	64M	128M
backward	FALSE	25.68	50.72	100.72	200.67		
	TRUE	0.35	0.35	0.38	0.38	0.35	
forward	FALSE	0.21	0.22	0.22	0.22		
	TRUE	0.12	0.12	0.12	0.12	0.12	

Table 2: Time taken for original embeddingn table

Forward pass (ms) [sparse=false]				
	compression			
n	10x	100x	1000x	10000x
4M	0.40	0.31	0.27	0.27
8M	0.36	0.32	0.29	0.27
16M	0.37	0.31	0.30	0.27
32M	0.37	0.25	0.31	0.26
64M	0.37	0.37	0.30	0.28
128M	0.37	0.37	0.32	0.29
256M		0.36	0.31	0.30

Table 3: batch size = 10240 and dimension = 128

Forward pass (ms) [sparse=true]				
	compression			
n	10x	100x	1000x	10000x
4M	0.57	0.54	0.34	0.33
8M	0.51	0.56	0.34	0.32
16M	0.56	0.56	0.36	0.34
32M	0.55	0.57	0.36	0.33
64M	0.56	0.57	0.52	0.35
128M	0.52	0.57	0.57	0.35
256M	0.55	0.53	0.51	0.36

Table 4: batch size = 10240 and dimension = 128

Backward pass (ms) [sparse=false]				
	compression			
n	10x	100x	1000x	10000x
4M	2.55	0.59	0.35	0.35
8M	4.81	0.83	0.37	0.35
16M	9.15	1.27	0.41	0.34
32M	17.81	2.13	0.54	0.35
64M	35.10	3.85	0.73	0.36
128M	69.73	7.42	1.08	0.40
256M		14.34	1.78	0.47

Table 5: batch size = 10240 and dimension = 128

Backward pass (ms) [sparse=true]				
	compression			
n	10x	100x	1000x	10000x
4M	2.67	2.48	2.56	2.52
8M	2.53	2.58	2.48	2.53
16M	2.52	2.68	2.58	2.45
32M	2.51	2.64	2.56	2.55
64M	2.51	2.66	2.63	2.60
128M	2.52	2.65	2.65	2.54
256M	2.54	2.52	2.53	2.62

Table 6: batch size = 10240 and dimension = 128

Forward pass (ms) [sparse=false]						
	PSS chunk size					
n	1	2	4	8	16	32
4M	0.68	0.58	0.55	0.32	0.31	0.31
8M	0.75	0.62	0.54	0.33	0.31	0.31
16M	0.77	0.65	0.54	0.54	0.31	0.31
32M	0.80	0.67	0.54	0.26	0.24	0.23
64M	0.92	0.79	0.65	0.39	0.37	0.37
128M	0.88	0.75	0.65	0.39	0.37	0.37
256M	0.88	0.75	0.65	0.40	0.36	0.36

Table 7: batch size = 10240 and dimension = 128

Forward pass (ms) [sparse=true]						
	PSS chunk size					
n	1	2	4	8	16	32
4M	0.87	0.62	0.57	0.57	0.52	0.37
8M	0.91	0.76	0.70	0.71	0.57	0.57
16M	0.94	0.80	0.68	0.70	0.56	0.56
32M	0.92	0.82	0.68	0.69	0.57	0.54
64M	0.93	0.80	0.68	0.67	0.56	0.57
128M	0.93	0.80	0.68	0.67	0.56	0.56
256M	0.93	0.80	0.68	0.67	0.55	0.37

Table 8: batch size = 10240 and dimension = 128, fixed compression = 100x

Backward pass (ms) [sparse=false]						
	PSS chunk size					
n	1	2	4	8	16	32
4M	1.29	1.10	0.80	0.64	0.58	0.57
8M	1.56	1.38	1.07	0.88	0.82	0.80
16M	2.02	1.87	1.54	1.33	1.27	1.27
32M	2.96	2.79	2.40	2.20	2.12	2.12
64M	4.69	4.54	4.13	3.98	3.85	3.85
128M	8.15	8.00	7.70	7.48	7.43	7.42
256M	15.17	15.01	14.61	14.40	14.34	14.34

Table 9: batch size = 10240 and dimension = 128, fixed compression = 100x

Backward pass (ms) [sparse=true]						
	PSS chunk size					
n	1	2	4	8	16	32
4M	3.17	3.04	2.61	2.54	2.50	2.48
8M	3.41	2.97	2.64	2.64	2.66	2.61
16M	3.56	3.07	2.74	2.58	2.54	2.62
32M	3.68	3.13	2.73	2.63	2.53	2.50
64M	3.94	3.36	2.91	2.64	2.52	2.64
128M	3.98	3.22	2.77	2.60	2.51	2.60
256M	4.00	3.27	2.74	2.61	2.53	2.50

Table 10: batch size = 10240 and dimension = 128, fixed compression = 100x

Forward pass (ms) [sparse=false]				
	compression			
chunk	10x	100x	1000x	10000x
1	0.89	0.90	0.73	0.28
2	0.77	0.80	0.61	0.27
4	0.65	0.65	0.54	0.27
8	0.40	0.39	0.32	0.28
16	0.37	0.37	0.30	0.29
32	0.37	0.37	0.30	0.30

Table 11: batch size = 10240 and dimension = 128

Forward pass (ms) [sparse=true]				
	compression			
chunk	10x	100x	1000x	10000x
1	0.93	0.93	0.87	0.34
2	0.80	0.80	0.77	0.32
4	0.69	0.68	0.71	0.33
8	0.69	0.67	0.72	0.34
16	0.56	0.57	0.49	0.34
32	0.56	0.55	0.54	0.36

Table 12: batch size = 10240 and dimension = 128

Backward pass (ms) [sparse=false]				
	compression			
chunk	10x	100x	1000x	10000x
1	35.94	4.69	1.44	0.37
2	35.78	4.53	1.26	0.37
4	35.38	4.13	0.96	0.36
8	35.15	3.91	0.83	0.36
16	35.10	3.85	0.72	0.36
32	35.11	3.86	0.70	0.37

Table 13: batch size = 10240 and dimension = 128

Backward pass (ms) [sparse=true]				
	compression			
chunk	10x	100x	1000x	10000x
1	4.08	3.89	2.77	2.87
2	3.32	3.20	2.76	2.88
4	2.78	2.78	2.61	2.63
8	2.62	2.60	2.58	2.58
16	2.51	2.68	2.63	2.55
32	2.49	2.50	2.59	2.57

Table 14: batch size = 10240 and dimension = 128