



PRE-TRAIN AND SEARCH: EFFICIENT EMBEDDING TABLE SHARDING WITH PRE-TRAINED NEURAL COST MODELS

Daochen Zha¹ Louis Feng² Liang Luo² Bhargav Bhushanam² Zirui Liu¹ Yusuo Hu² Jade Nie²
Yuzhen Huang² Yuandong Tian² Arun Kejariwal² Xia Hu¹

ABSTRACT

Sharding a large machine learning model across multiple devices to balance the costs is important in distributed training. This is challenging because partitioning is NP-hard, and estimating the costs accurately and efficiently is difficult. In this work, we explore a “*pre-train, and search*” paradigm for efficient sharding. The idea is to pre-train a universal and once-for-all neural network to predict the costs of all the possible shards, which serves as an efficient sharding simulator. Built upon this pre-trained cost model, we then perform an online search to identify the best sharding plans given any specific sharding task. We instantiate this idea in deep learning recommendation models (DLRMs) and propose NeuroShard for embedding table sharding. NeuroShard pre-trains neural cost models on augmented tables to cover various sharding scenarios. Then it identifies the best column-wise and table-wise sharding plans with beam search and greedy grid search, respectively. Experiments show that NeuroShard significantly and consistently outperforms the state-of-the-art on the benchmark sharding dataset, achieving up to 23.8% improvement. When deployed in an ultra-large production DLRM with multi-terabyte embedding tables, NeuroShard achieves 11.6% improvement in embedding costs over the state-of-the-art, which translates to 6.6% end-to-end training throughput improvement. To facilitate future research of the “*pre-train, and search*” paradigm in ML for Systems, we open-source our code at <https://github.com/daochenzha/neuroshard>

1 INTRODUCTION

Deep learning recommendation models (DLRMs) are one of the most important machine learning applications (Zhang et al., 2019; Cheng et al., 2016; Naumov et al., 2019; Tan et al., 2023b; Zhou et al., 2023). For example, DLRMs account for more than 50% of training and 80% inference demands in Meta (Naumov et al., 2020; Gupta et al., 2020). A challenge in DLRMs is how to deal with sparse categorical features. For instance, a single categorical feature in the YouTube recommendation model contains tens of millions of video IDs (Covington et al., 2016). To handle the categorical features, modern DLRMs use embedding tables, which are hash tables that map a categorical index to a vector.

Unfortunately, embedding tables are often the storage and efficiency bottlenecks in production-scale DLRMs. On the one hand, the embedding tables can be extremely large. For example, the embedding tables in the Meta DLRMs demand multi-terabyte memory (Acun et al., 2021; Mudigere et al., 2022). Thus, modern distributed training systems for DL-

RLMs often have to adopt model parallelism to partition the tables and place them on different devices, such as GPUs and CPUs (Acun et al., 2021; Covington et al., 2016; Liu et al., 2017; Gomez-Uribe & Hunt, 2015; Lian et al., 2022). On the other hand, embedding tables often incur significant computation and communication costs. For instance, it is reported that embedding tables account for 48% of the total computation and 65% of the total communication costs in one of the Meta DLRMs (Zha et al., 2022b).

The left side of Figure 1 shows the computation and communication costs for embedding tables in a typical distributed training workflow of DLRMs¹ (Naumov et al., 2019). It exploits a combination of model parallelism, i.e., partitioning embedding tables and placing them to multiple GPUs, and data parallelism, i.e., duplicating the fully connected layers and partitioning the training data. In the forward pass, each GPU queries the other GPUs with its sparse features to look up the embeddings from their tables (forward computation) and obtain the embeddings through an all-to-all communication (forward communication). In the backward pass, the gradients are sent back to the GPUs with another all-to-all communication (backward communication) and applied to the embeddings (backward computation).

¹Department of Computer Science, Rice University, USA
²Meta Platforms, Inc., USA. Correspondence to: Daochen Zha <daochen.zha@rice.edu>.

¹This work focuses on sharding among GPU devices. We will study CPU or mixed CPU-GPU sharding scenarios in the future.

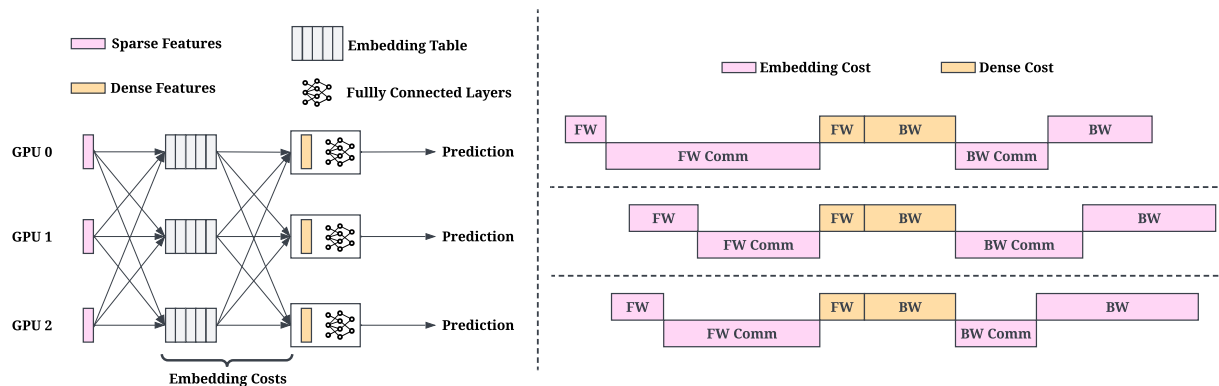


Figure 1. Left: an illustrative distributed training workflow of DLRMs (Naumov et al., 2019) on three GPUs. Right: typical GPU traces in the fully synchronous mode (Mudigere et al., 2022). For each GPU, the top and bottom threads are for computation and communication, respectively, and they can overlap. FW and BW stand for forward and backward computations. FW Comm and BW Comm mean forward and backward communications. We only visualize the major costs; the other costs are often neglectable or hidden due to overlapping. The embedding forward operations (pink FW) often do not start at the same time across GPUs because of the different ending times of the embedding backward operations (pink BW) in the previous training iteration.

An important design factor that can significantly impact the embedding costs is embedding table sharding, i.e., the strategy of partitioning and placing embedding tables. If not carefully partitioned, the embedding tables can easily lead to imbalances, where some devices have significantly more computation and communication costs than others, leading to a straggler effect in the synchronous training setting (Mudigere et al., 2022). While some heuristic-based sharding strategies have been proposed (Acun et al., 2021; Lui et al., 2021), they rely on oversimplified cost functions so they often have unsatisfactory sharding performance.

Recently, reinforcement learning (RL) has shown promise in embedding table sharding (Zha et al., 2022a;b). The idea is to make sharding an optimization problem, which aims to identify a sharding plan that can minimize the overall cost. These methods formulate sharding as a Markov decision process (MDP), whose states and rewards are computation and communication costs estimated by neural networks. Then they train another policy network to solve the MDP to minimize the overall embedding costs. These learning-based methods have achieved a significant improvement over the heuristic sharding strategies (Zha et al., 2022a;b).

Despite the successes of RL-based approaches, it is difficult to deploy them. **1)** They only consider table-wise sharding, i.e., they treat tables as the smallest units in sharding and focus on how to assign each table to a device. However, it is very likely that one table is extremely large or costly, which makes it a memory and computation bottleneck. Adopting these approaches may lead to an out-of-memory error or an undesirable balance. **2)** The policy network is often trained on very few sharding tasks, so frequent re-training will be needed to handle unseen tasks. **3)** The policy in RL is notoriously unstable with a high variance (Henderson

et al., 2018; Zha et al., 2019; 2021d; Lai et al., 2020a); that is, even if we train the same policy on the same MDP with multiple independent runs, some of the runs may work well but the others could fail. However, we often demand a stable sharding solution in production.

Motivated by the recent successes of “pre-train, prompt, and predict” in large language models (Liu et al., 2023; Brown et al., 2020; Touvron et al., 2023; Chuang et al., 2023; Tang et al., 2023), we explore a “pre-train, and search” paradigm for efficient sharding and present NeuroShard, illustrated in Figure 2. To handle the extremely large or costly tables, we incorporate *column-wise* sharding into the optimization process, where a table can be partitioned into two smaller tables, each with half the columns of the original one. Unlike RL-based methods that stochastically train a policy network, we pre-train general neural cost models on augmented data to cover comprehensive sharding scenarios. Once trained, the cost models serve as a simulator to efficiently estimate the embedding costs for any sharding plans. Built upon the pre-trained cost models, NeuroShard identifies the best column-wise and table-wise sharding plans with beam search and greedy grid search, respectively. NeuroShard not only outperforms the state-of-the-art but also can be easily deployed in production. In summary, we make the following contributions.

- We provide a comprehensive analysis of the computation and communication costs of embedding tables. We observe: **1)** partitioning a table column-wisely will increase the overall cost so that column-sharding has a tradeoff between overall cost and balance, **2)** multi-table computation cost has a non-linear correlation with the sum of single-table costs, and **3)** the communication cost is mainly determined by table dimensions.

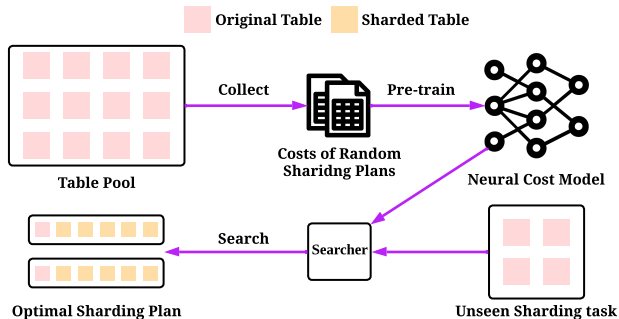


Figure 2. NeuroShard identifies the optimal sharding plan with “pre-train, and search”.

- Motivated by the observations, we devise NeuroShard, a “pre-train, and search” framework which searches for column-wise and table-wise sharding plans on pre-trained cost models. Unlike the RL-based methods whose cost models only have limited coverage, our cost models are trained for a once-for-all purpose, i.e., training one universal cost model for all the sharding tasks. Moreover, our cost models can readily support column-wise sharding since it is trained on an augmented table pool with various table dimensions.
- NeuroShard achieves up to 23.8% improvement over the state-of-the-art without out-of-memory error on the benchmark sharding dataset (Naumov et al., 2019)².
- NeuroShard has been deployed to an ultra-large production DLRM to shard multi-terabyte embedding tables to hundreds of GPUs. NeuroShard achieves an 11.6% improvement in embedding costs, which translates to 6.6% end-to-end training throughput improvement, which is a significant speedup since the production DLRM has been heavily optimized.

2 UNDERSTANDING EMBEDDING COST

This section analyzes the embedding costs to motivate the sharding algorithm design. The right-hand side of Figure 1 shows a typical trace in one training iteration, which mainly consists of the computation/communication of the embeddings and the computation of the fully connected layers. Note that there are some other costs that are not visualized in the figure, such as the communication of the sparse features before embedding lookup, and the weight synchronization of the fully connected layers, etc. These costs are often neglectable or hidden because of the overlapping of computation and communication.

²https://github.com/facebookresearch/dlrm_datasets

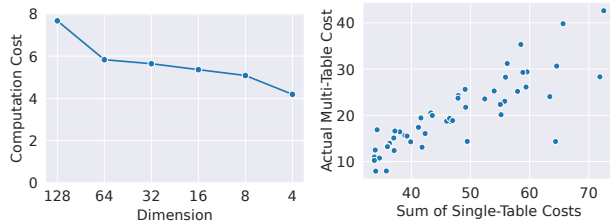


Figure 3. Left: computation costs w.r.t. dimensions. We observe similar patterns for other tables (see Appendix A.1). Right: the sum of single-table costs versus the actual multi-table cost.

We first explain why imbalances can cause more embedding costs using the example trace. In the trace of GPU 1, the embedding backward communication and computation take more time than the other GPUs, which makes its embedding forward computation in the next iteration start later than its peer GPUs. The embedding forward computation of GPU 1 again takes a longer time than those of the other GPUs. As such, the above delays are finally accumulated, forcing GPU 1 to start embedding forward communication significantly later than other GPUs. This imbalance issue results in significant idle times for GPU 0 and GPU 2.

To reduce the accumulated delay, we need to balance the computation and communication costs associated with embeddings across all GPUs. In the following, we analyze the costs separately. All the results are collected on 2080Ti GPUs with a modern embedding implementation from FBGEMM (Khudia et al., 2021), which fuses multiple table lookups as a single operation. Appendix A provides more details on all the analytical experiments.

2.1 Computation Cost Analysis

Computation costs are mainly decided by the table configurations and the lookup indices. Some important factors were used to quantify the computation costs (Zha et al., 2022a). **1) Dimension:** the number of columns of the table. A larger dimension often means a higher computation cost since it technically leads to more memory bandwidth use. **2) Hash size:** the number of rows of the table. It indirectly impacts the computation costs by affecting the caching/prefetching behaviors. **3) Pooling factor:** the number of embedding indices in a lookup. We often calculate the mean pooling factor of a batch of indices. A higher mean pooling factor leads to a higher computation cost since it determines the workloads of the lookup. **4) Indices distribution:** the access pattern and distribution of the lookup, i.e., some indices can be accessed more frequently than others. It indirectly impacts the costs by affecting cache effectiveness. Also, the number of unique embeddings accessed in a batch can also influence the caching. Fewer indices being accessed will often lead to smaller costs.

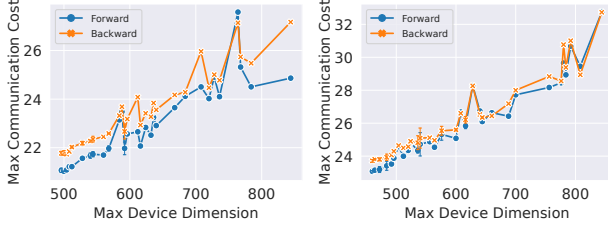


Figure 4. Max forward/backward communication cost versus max device dimensions among 4 GPUs (left) and 8 GPUs (right).

The previous analysis suggests that the actual computation costs have complex and non-linear correlations with the above factors (Zha et al., 2022a), which makes cost estimation hard. Here, we perform two analytical experiments to provide a deeper understanding and motivate the algorithm designs of column-wise sharding and table-wise sharding.

First, we study the impact of dimension. We randomly choose a table from the benchmark sharding dataset named DLRM (Naumov et al., 2019)². The left-hand side of Figure 3 visualizes its computation costs (forward + backward) with varying dimensions of $\{128, 64, 32, 16, 8, 4\}$. As expected, a larger dimension corresponds to a higher computation cost. However, we also have the following observation.

Observation 1 *When partitioning a table into two halves column-wisely, the computation cost of each shard is larger than half the cost of the original table.*

For example, the cost of dimension 64 is much larger than the half of cost of dimension 128. This could be explained by parallelism and operation fusion; the fused embedding table operation can achieve better optimization in the CUDA kernel than in each operation alone. This also indicates a trade-off in column-wise sharding: while partitioning tables into smaller tables could improve load balance, it may increase the overall computation cost. Thus, the column-wise sharding algorithm needs to strike a balance between the load balance and the overall computation cost.

Then, we investigate whether we can estimate the computation cost of a multi-table operation with the sum of the single-table costs. This is important because if this assumption holds, we could well balance the computation costs with tools such as mixed integer linear program (Sethi et al., 2022). We randomly sample 50 subsets of tables from the DLRM dataset, where each subset contains 10 tables. Then we plot their relationships on the right-hand side of Figure 3.

Observation 2 *Multi-table computation cost has a non-linear relationship with the sum of single-table costs.*

Evidently, table-wise sharding algorithms must consider the nonlinearity of the multi-table costs.

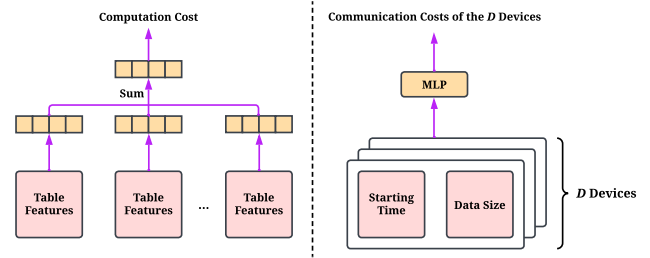


Figure 5. The neural architectures of the computation cost model (left) and the communication cost model (right).

2.2 Communication Cost Analysis

In distributed training, every pair of GPUs need to communicate in both the forward and backward passes. Thus, each GPU could have a different communication latency. Our goal is to minimize the max communication cost among all the GPUs since the slowest one will become the bottleneck. Intuitively, the communication costs depend on the sizes of the data to be sent in the GPUs, where the data size of a GPU can be estimated by the product of the batch size and the device dimension, which is defined as the sum of the dimensions of the tables in the device. Since all the GPUs have the same batch size, the device dimension becomes the determining factor of communication balance.

We conduct an experiment to understand the relationship between the max device dimension and the max forward/backward communication cost. We randomly sample a subset of tables (40 tables for 4 GPUs, and 80 tables for 8 GPUs) from the DLRM dataset and select a random dimension for each table from $\{128, 64, 32, 16, 8, 4\}$. Then we shard these tables to 4 or 8 GPUs with varying max device dimensions. We benchmark 50 assignments in Figure 4.

Observation 3 *The max forward/backward communication cost among all the GPUs positively correlates with the max device dimension among all the GPUs.*

The above observation motivates us to adopt an alternative way to balance communication costs, i.e., minimizing the max device dimension among all the GPUs. Balancing the device dimensions is much easier since the device dimension is simply the sum of the dimensions of the tables.

3 NEUROSHARD FRAMEWORK

Motivated by the three observations above, we propose NeuroShard, an embedding table sharding framework based on pre-trained neural cost models and online search. Figure 6 shows the workflow. The main idea is to pre-train neural networks to predict the computation and communication costs, which can serve as a sharding simulator to quickly estimate the embedding costs for any sharding tasks and

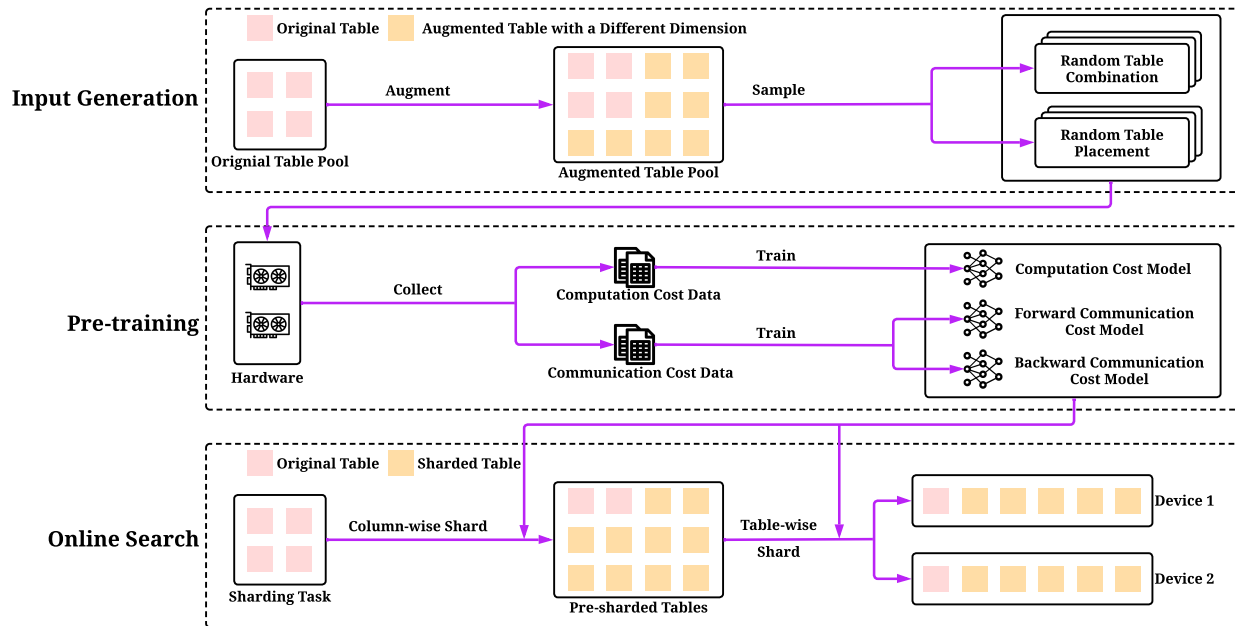


Figure 6. Overall workflow of NeuroShard. We first generate random inputs based on the augmented tables (top row). Then we run a micro-benchmark to collect the costs and pre-train three neural networks to predict the costs (middle row). Finally, we perform an online search based on the pre-trained cost models for embedding table sharding (bottom row).

any sharding plans. Then we perform an online search on the pre-trained cost models to identify the best sharding plan without real GPU execution. In what follows, we describe how to generate synthetic inputs for training data collection (Section 3.1) and how to train neural networks for cost prediction (Section 3.2). Once the cost models are pre-trained, we then present how to perform an online search to optimize column-wise sharding and table-wise sharding (Section 3.3).

3.1 Generating Synthetic Inputs

Pre-training is mainly a data-centric procedure (Zha et al., 2023b;a), where high-quality cost data plays an essential role. This subsection describes how to generate table inputs that can cover different table combinations and placements for benchmarking the computation and communication costs, which includes table augmentation, random table combination generation, and random table placement generation. Note that to achieve the best coverage, the generation strategy should consider the infrastructure for model training and the embedding tables in the model. For example, if a model has a lot of embedding tables but only a very limited number of GPUs is used, the generated inputs should cover the table combinations that have lots of tables. In the following, we mainly discuss the high-level strategy on how to achieve good coverage. We will introduce the instantiation of the strategy on the DLRM dataset in Section 4.

Table augmentation. In DLRMs, each embedding table corresponds to a sparse feature, which is often collected

from users or items. In real-world applications, we often have a pool of embedding tables, where the machine learning engineers will perform feature selection, i.e., choosing a subset of embedding tables from the pool for model training. In this procedure, the dimensions of the tables could be adjusted. Further, column-wise sharding will also generate new tables with different dimensions. Thus, the generated table inputs should be able to cover tables with different dimensions. To achieve this, we perform table augmentation. Specifically, for each table, we generate augmented tables with different dimensions. For example, suppose the original table has a dimension of 64, we could generate 5 augmented tables with dimensions of 128, 32, 16, 8, and 4 to accommodate the potential dimension adjustment in the model design and column-wise sharding. The table augmentation results in an augmented table pool, which will be used for data generation. Appendix B.1 summarizes the detailed augmentation process.

Random table combination generation. The table combinations will be used to benchmark computation costs. The generated combinations should cover different numbers of tables on a GPU. To achieve this, we first uniformly sample the number of tables T in a certain range and then randomly select a subset of T tables from the augmented table pool. Appendix B.2 summarizes the detailed generation process.

Random table placement generation. The table placements will be used to benchmark communication costs. The generated placements should cover different degrees of balance for the device dimensions and different communication

starting timestamps. **1) To simulate different degrees of balance, we adopt a greedy strategy equipped with randomness.** Specifically, we first randomly sample a subset of tables from the table pool and sort the tables in descending order based on the dimension. Starting from the table with the largest dimension, with a probability of p , where p is uniformly sampled in $[0, 1]$ for each table placement, we assign the current table to the GPU with the lowest sum of the table dimensions so far, and with a probability of $(1-p)$, we randomly assign the current table. Here, p can indirectly control the degree of balance. When $p = 1$, this strategy will greedily balance dimensions in each step so the sharding plan can well balance the dimensions. When $p = 0$, the sharding plan will be random so that the dimensions will be very likely to be imbalanced. Since p is randomly selected, we can cover sharding plans with different degrees of balance. **2) We randomly generate the communication starting timestamp of each GPU.** Recall that in the trace analysis (the right-hand side of Figure 1), the delays caused by previous operations can be accumulated to make the forward communication start significantly later than the other GPUs. Naturally, we also need to take the delays into account when benchmarking the communication costs since the communication could have different behaviors when they do not start simultaneously across the GPUs. To simulate the delays, we randomly select a starting timestamp in a certain range for each GPU. Appendix B.3 summarizes the detailed generation procedure.

3.2 Pre-training Neural Cost Models

Given the generated inputs, we run micro-benchmark, e.g., PARAM Benchmarks³, to collect the actual computation and communication costs for cost model training. Appendix C provides more details of cost model training.

Figure 5 illustrates the neural architectures of the computation and communication cost models. **1) Computation cost model:** the architecture follows (Zha et al., 2022a). Specifically, each table is represented with some features, including dimension, hash size, pooling factor, and indices distribution. Given a table combination, we use a shared MLP to process all the table features to obtain table representations. We obtain a fixed-dimension representation of a table combination by performing an element-wise sum of all the table representations. Finally, we use another MLP to produce the computation costs (forward + backward). **2) Communication cost model:** we use an MLP to predict the communication costs of all the GPUs based on the starting timestamps and the transferred data sizes. We train two separate models for forward and backward communications.

Deployment of the neural cost models. In real-world DL-

RMs, the indices distributions could shift over time. Thus, we may need to re-train/fine-tune and redeploy the cost models to tackle the potential shifts. To enable a smooth re-deployment, we often need to have strict version control to ensure that one training job is always associated with the same version of cost models. This is particularly important for checkpointing since we need a consistent sharding plan when resuming the training. We find a re-training interval of three months is sufficient in our production environment. One could also periodically calculate the prediction errors of the cost model by sampling a batch of table indices and trigger re-training or fine-tuning when the error exceeds a certain threshold. Note that we often only need to re-train when the indices distributions shift. Re-training is not needed when table dimension changes, as the table augmentation has already encompassed various table dimensions.

3.3 Online Search

The pre-trained cost models serve as a universal simulator for embedding table sharding. They can estimate the embedding cost of any sharding plan for any sharding task efficiently by summing up the predicted computation, forward communication, and backward communication costs. In this subsection, we introduce how to leverage this simulator to minimize the embedding cost with search. Figure 7 shows an overview. In the outer loop, we search the column-wise sharding plan with beam search. In the inner loop, we find the best max dimension constraint with a greedy grid search for table-wise sharding. The presented search algorithm is mainly motivated by the observations in Section 2.

Optimization problem formulation. We denote the column-wise sharding plan as $\mathbf{c} = [c_1, c_2, \dots, c_m]$, where c_i means, in step i , we shard the table of index c_i into two halves column-wisely and append the resultant new table to the end of the table list. Let $\mathbf{t} = [t_1, t_2, \dots, t_T]$ denote the table-wise sharding plan that assigns T tables to D GPU devices, where $t_i \in \{1, 2, \dots, D\}$. \mathbf{t} depends on \mathbf{c} since \mathbf{t} operates on the column-wise sharded tables. Both \mathbf{c} and \mathbf{t} must satisfy some constraints. For example, the embedding operations in FBGEMM (Khudia et al., 2021) require that the dimension must be dividable by 4. For \mathbf{c} , the sharding plan has to satisfy the GPU memory constraints. We denote their legal plan spaces as \mathcal{C} and \mathcal{T} , respectively, where \mathcal{T} depends on the selected \mathbf{c} . The objective is

$$\arg \min_{\mathbf{c} \in \mathcal{C}, \mathbf{t} \in \mathcal{T}} f(\mathbf{c}, \mathbf{t}). \quad (1)$$

$f(\mathbf{c}, \mathbf{t})$ is the simulated embedding cost. The dependency of \mathbf{t} on \mathbf{c} naturally makes the search of \mathbf{c} as the outer loop and the search of \mathbf{t} as the inner loop.

Column-wise sharding with beam search. Column-wise sharding can remove oversized tables and costly tables to enable a better balance. However, from Observation 1, column-

³<https://github.com/facebookresearch/param>

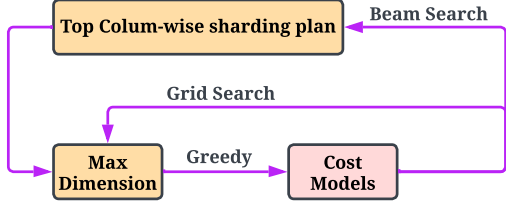


Figure 7. The search process. The outer loop finds the best column-wise sharding plan with beam search. The inner loop focuses on the max dimension constraint of greedy allocation with grid search.

wise sharding increases the overall computation cost. Thus, the desirable sharding plan should enable a balance with minimum steps. We propose a beam search strategy to reduce the search space. The main idea is that we often only need to column-wisely shard tables with large sizes and high computation costs. Specifically, in each iteration i , we identify the top N costly tables and the top N tables with the largest sizes as the candidates (with duplicates removed). Similarly, we only consider the top K best column-wise sharding plans found in the previous step. For each of the top K plans, we add each of the candidate tables to it to obtain a new sharding plan and run the inner loop to get the cost. We again identify the top K best new sharding plans for the next sharding step. We perform L sharding steps and output the sharding plan with the minimum cost. L , K , and N are hyperparameters to balance optimality and efficiency.

Table-wise sharding with greedy grid search. Given a list of column-wisely sharded tables, we describe how to perform table-wise sharding in the inner loop. We propose a grid search strategy to find the best balance of computation and communication costs based on two ideas. **1)** Motivated by Observation 2, we propose a greedy algorithm to balance the multi-table computation costs. **2)** Inspired by Observation 3, we make the max device dimension a constraint for the greedy algorithm to achieve the communication balance, where the best max device dimension is identified with grid search. Specifically, in each step we execute the following: **1)** Choose a max device dimension max_dim . **2)** Sort the tables in descending order based on the computation cost predicted by the cost model. **3)** Starting from the table with the highest cost, we assign tables one by one to the device with the lowest device cost so far subject to the memory and max_dim constraints, where the device cost is predicted by the cost model. **4)** Evaluate the embedding cost with the cost models. We grid search max_dim as follows. Given a starting value M_s , an ending value M_e , and the total number steps M , we try all the values in $[M_s, M_e]$ with a step size of $(M_e - M_s)/(M - 1)$. We empirically fix M_s to be the average dimension across device and M_e to be $1.5 * M_s$. M is a hyperparameter to control the granularity of the search.

Implementation with caching. The most expensive part in the search is predicting the computation cost. It needs to be

Algorithm 1 BeamSearch

```

1: Input:  $T$  embedding tables, beam search hyperparameters ( $N$ ,  $K$ , and  $L$ )
2: Best global column-wise sharding plan  $\mathbf{c}^* \leftarrow []$ 
3: Best global table-wise sharding plan  $\mathbf{t}^* \leftarrow NULL$ 
4: Best column-wise sharding plans  $C_p \leftarrow \{[]\}$ 
5: Initialize a global cache  $global\_cache$ 
6: for  $outer\_loop = 1, 2, \dots, L$  do
7:   Column-wise sharding plans in the next step  $C'_p \leftarrow []$ 
8:   for each plan  $\mathbf{c}_p$  in  $C_p$  do
9:     Obtain the candidate tables based on  $\mathbf{c}_p$  by merging the top  $N$  costly tables and the top  $N$  tables with the largest sizes with duplicates removed
10:    for each candidate table  $t$  do
11:       $col\_plan \leftarrow \mathbf{c}_p$  with  $t$  appended in the end
12:       $cost, \mathbf{t} \leftarrow GreedyGridSearch(global\_cache, col\_plan)$ 
13:      Append  $(col\_plan, cost)$  to  $C'_p$ 
14:      if a lower  $cost$  is observed then
15:         $\mathbf{c}^* \leftarrow col\_plan$ 
16:         $\mathbf{t}^* \leftarrow \mathbf{t}$ 
17:      end if
18:    end for
19:   end for
20:    $C_p \leftarrow$  plans with the top  $K$  lowest cost in  $C'_p$ 
21: end for
22: Return  $\mathbf{c}^*, \mathbf{t}^*$ 
    
```

called for $O(LKNMTD)$ times (T is the number of tables, and D is the number of GPUs), where each call requires a forward pass of the cost model. Fortunately, we find there are lots of duplicated calls in the search. This is because if we only make small changes to the column-wise sharding plan or max_dim , the cost model will be very likely to be asked to predict the cost for the same set of tables in most of the steps. Thus, we can naturally use a life-long hash map as a cache for acceleration. In practice, the cache hit rate can reach 95% (see Table 3). We summarize the beam search in Algorithm 1 and greedy grid search in Algorithm 2.

4 EXPERIMENTS

The experiments aim to answer the following research questions. **RQ1:** How does NeuroShard compare with the state-of-the-art sharding algorithms (Section 4.1)? **RQ2:** How accurate are the neural cost models (Section 4.2)? **RQ3:** How does each search design contribute to the performance (Section 4.3)? **RQ4:** How do the hyperparameters impact the performance (Section 4.4)? **RQ5:** Can NeuroShard boost end-to-end training throughput (Section 4.5)?

Datasets. The public large datasets (e.g., Criteo, Avazu, and KDD) often do not match the industrial-scale data and are

Table 1. Embedding table cost in milliseconds (averaged over 100 randomly constructed sharding tasks) of NeuroShard against baselines with 4 or 8 GPUs and maximum table dimensions from 4 to 128. The top-1 and top-2 results are highlighted in boldface and underlined, respectively. “-” indicates that the method cannot scale, i.e., at least one of the 100 tasks suffers from memory explosion. The bottom row summarizes the improvement of NeuroShard over the strongest baseline.

Category	Method	4 GPUs						8 GPUs					
		4	8	16	32	64	128	4	8	16	32	64	128
Random	-	25.47	30.10	-	-	-	-	33.40	36.43	-	-	-	-
Greedy	Size-based	24.45	29.60	30.86	37.80	41.59	-	31.75	34.30	37.70	46.07	54.57	-
	Dim-based	23.51	28.46	29.76	35.98	38.71	-	27.54	32.20	34.78	42.35	47.54	-
	Lookup-based	18.69	24.34	26.83	34.62	38.69	-	21.83	26.66	30.59	<u>39.07</u>	47.47	-
	Size-lookup-based	18.38	24.18	26.81	33.94	-	-	21.27	26.55	30.34	39.28	48.17	-
Reinforcement Learning	AutoShard	<u>17.99</u>	<u>22.08</u>	-	-	-	-	<u>20.79</u>	-	-	-	-	-
	DreamShard	18.78	<u>22.59</u>	<u>25.08</u>	<u>30.74</u>	-	-	21.40	<u>25.30</u>	<u>26.90</u>	-	-	-
Planning	TorchRec	19.20	26.00	28.24	34.88	<u>38.13</u>	<u>47.22</u>	22.34	28.99	32.71	39.90	<u>47.43</u>	60.58
Cost Modeling	NeuroShard	<u>17.74</u>	<u>21.75</u>	<u>23.11</u>	<u>28.86</u>	<u>31.55</u>	<u>39.99</u>	<u>20.68</u>	<u>23.23</u>	<u>25.64</u>	<u>32.30</u>	<u>38.30</u>	<u>49.10</u>
Improvement of NeuroShard		+1.4%	+1.5%	+8.5%	+6.5%	+20.9%	+18.1%	+0.5%	+8.9%	+4.9%	+21.0%	+23.8%	+23.4%

unsuitable for evaluating sharding algorithms. Please find more discussions in Appendix D. Following the previous work (Zha et al., 2022a;b), we use the DLRM dataset (Nau-mov et al., 2019)², which contains 856 synthetic tables whose indices distributions are similar to the production workloads in Meta. These 856 tables serve as the table pool. **We construct synthetic sharding tasks as diversely as possible to test sharding algorithms in different scenarios.** We consider two sets of sharding tasks, which aim to shard tables to 4 and 8 GPUs, and each GPU has a memory constraint of 4 GBs for embedding tables. We randomly sample a dimension for each table in $\{4, 8, \dots, 2^j\}$, where $2 \leq j \leq 7$, and 2^j specifies the maximum possible dimension for a table. A larger 2^j makes the sharding task more challenging since the tables will have more diverse dimensions and larger sizes. Given the number of GPUs d and max dimension 2^j , we sample a sharding task by randomly choosing T tables from the table pool, where $10 \leq T \leq 60$ for 4 GPUs and $20 \leq T \leq 120$ for 8 GPUs. For each pair of d and 2^j , we randomly construct 100 sharding tasks. We provide a detailed description of the sharding tasks generation in Appendix D. In Section 4.1, we consider all the above d - 2^j pairs. For the other experiments, we mainly focus on a maximum dimension of 128 and 4 GPUs. Also, we consider a real-world sharding task in a production model, which aims to shard hundreds of tables to 128 GPUs (Section 4.5).

Baselines. We consider baselines in several categories (detailed in Appendix E): 1) **Random** sharding, 2) **Greedy** algorithms that balance various heuristic costs (Acun et al., 2021; Lui et al., 2021), 3) **Reinforcement Learning** algorithms proposed in (Zha et al., 2022a;b), and 4) **Planning** algorithm provided in TorchRec⁴.

⁴<https://github.com/pytorch/torchrec>

Evaluation protocol. For each pair of d and 2^j , we apply each sharding algorithm to generate sharding plans for the 100 sharding tasks and collect real embedding costs from GPUs. To collect the costs, we run the embedding operations on GPUs to simulate computation and communication and use a timer to measure the time spent on each device. We report the maximum cost across devices for each sharding task since the maximum embedding cost will become the bottleneck. If any of the sharding plans generated by an algorithm causes memory error, it means the algorithm cannot scale to the setting defined by d and 2^j , so we denote the performance as “-”. If all the sharding plans are valid, we report the mean embedding cost across the 100 tasks. For the real-world production sharding task, we report both embedding costs and end-to-end training throughput improvements. The embedding costs are directly obtained from the traces collected during model training.

Implementation details. For the generation of the synthetic input, we augment the table pool with dimensions $\{4, 8, 16, 32, 64, 128\}$. We randomly select 1 to 15 tables for the table combination generation and N_{place} tables for the table placement generation, where $10 \leq N_{\text{place}} \leq 60$ for 4 GPUs and $20 \leq N_{\text{place}} \leq 120$ for 8 GPUs. We generate 100K samples for each cost model. We randomly select a starting timestamp from 0 to 20 milliseconds. For the online search, we set $N = 10$, $K = 3$, $L = 10$, and $M = 11$. All the experiments are conducted on a server with eight 2080Ti GPUs. We provide more details in Appendix E.

4.1 Comparison with the State-of-the-art Methods

To answer **RQ1**, we compare NeuroShard with the baselines on different numbers of GPUs and max table dimensions in Table 1. We make the following observations.

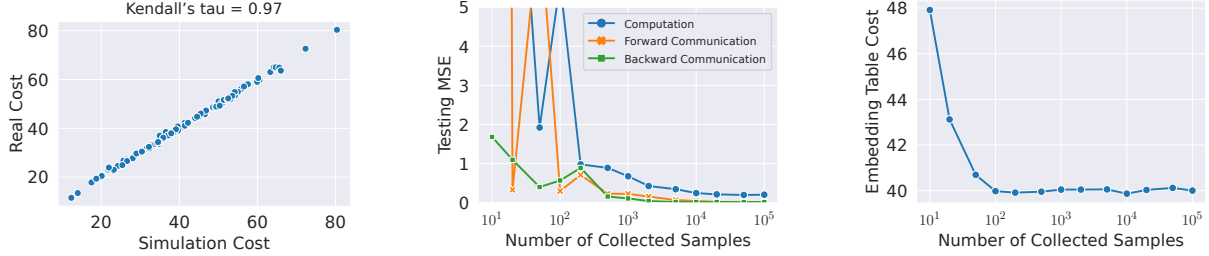


Figure 8. Left: the scatter plot of the simulation costs estimated by the neural cost models and the real costs measured on GPUs. Middle: the performances of the neural cost models w.r.t. the number of samples used in training. Right: the embedding table cost for the sharding tasks with a maximum dimension of 128 and 4 GPUs using the neural cost models that are trained with different numbers of samples.

1) NeuroShard significantly outperforms the baselines in all the sharding tasks with an improvement ranging from +0.5% to +23.8%, which demonstrates the superiority of NeuroShard. An advantage of NeuroShard over AutoShard and DreamShard stems from its column-wise sharding. The large or costly tables will be partitioned to avoid memory explosion or become the bottleneck in sharding. This can not be achieved by DreamShard and AutoShard. 2) NeuroShard successfully scales to sharding tasks with high table dimensions. When the table dimension goes large, all the baselines except TorchRec tend to fail. This is because it becomes harder to find a sharding plan that can satisfy the memory constraint with larger tables. Unlike the greedy and RL-based methods, NeuroShard searches for the best column-wise sharding plan so that it can partition the large tables. Surprisingly, the RL-based methods fail even when the dimension is small. A possible reason is that the stochastic policies in RL are very hard to train with high variance. 3) Learning-based methods tend to perform better than heuristic costs. This is because the neural cost models are trained in a data-driven manner so they can provide a better cost estimation to boost the sharding performance. 4) While TorchRec also scales well, NeuroShard achieves much better performances. This is because TorchRec still relies on a heuristic cost function, which is inaccurate. Whereas the cost models in NeuroShard are pre-trained in a data-driven manner so that they can estimate table costs more accurately.

4.2 Analysis of Neural Cost Models

To study **RQ2**, we design experiments to understand how accurate the cost models are and how accurate they need to be to enable good sharding plans. First, we report the testing mean-squared-error (MSE) losses of all the pre-trained neural costs models in Table 2. We observe that the largest MSE is 0.26, which suggests that the prediction error is within 0.6 milliseconds ($0.6 \times 0.6 = 0.36 > 0.26$). Note that the real costs may have some variance when collecting them. Thus, an error of 0.6 milliseconds is highly accurate. The left-hand side of Figure 8 plots the real costs and simulation costs for 100 random sharding plans. The results

again verify the high accuracy of the cost models.

Then, we visualize how many samples are needed to train the cost models in the middle and right-hand side of Figure 8. As expected, all the cost models become more accurate when we have more samples. Interestingly, even with only 10² samples, NeuroShard can achieve very strong performance. This suggests that we only need sufficiently but not perfectly accurate cost models. This is desirable in practical use since it means we do not need many samples. Note that the result does not imply that we can use a simpler model. The current neural architecture of NeuroShard is already very shallow. An even simpler network (i.e., a linear one) may not work due to the non-linearity of the costs.

4.3 Ablation Study

To investigate **RQ3**, we report the results with one of the proposed beam search, greedy grid search, and caching removed in Table 3. We make two observations. 1) The performance drops significantly when removing beam search or greedy grid search, which demonstrates the necessity of performing a joint search with both of them. 2) The sharding takes significantly more time when removing the caching mechanism. This is because the cache has a more than 95% hit rate, which can significantly accelerate the sharding speed.

4.4 Hyperparameter Analysis

To understand **RQ4**, we analyze the impact of the hyperparameters in NeuroShard. Recall that we have 4 hyperparameters in the online search to balance between optimality and efficiency, i.e., N , K , L , and M . We visualize their impacts in Figure 9. We make two observations. 1) A larger value for all the above four hyperparameters leads to better performance. This is because a larger value will result in more search iterations. 2) Larger values also lead to more sharding time. Thus, we should specify an appropriate value (not too large nor too small) for each of the hyperparameters to strike a balance between optimality and efficiency.

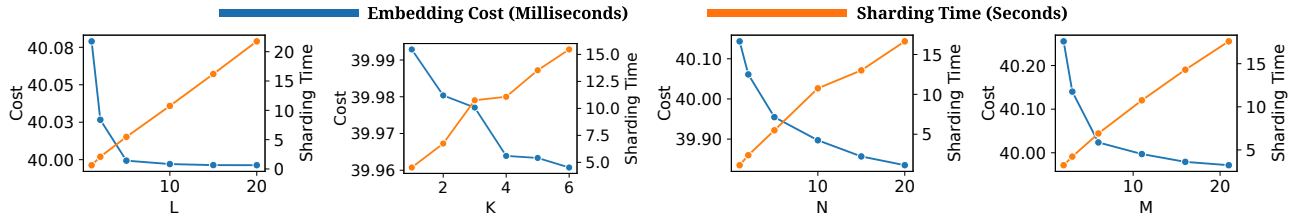


Figure 9. Impact of NeuroShard’s hyperparameters.

Table 2. Testing MSE of the neural cost models.

	DLRM (4 GPUs)	DLRM (8 GPUs)	Production (128 GPUs)
Computation	0.21	0.21	0.26
Forward Communication	0.02	0.05	0.05
Backward Communication	0.02	0.04	0.15

Table 3. Ablation study with a maximum dimension of 128 and 4 GPUs. w/o beam search means removing the beam search in col-wise sharding. w/o greedy grid search suggests not grid-searching the table dimension threshold. w/o caching disables the caching mechanism of computation costs. Results on 8 GPUs are in Appendix G.

	Cost (Milliseconds)	Success Rate	Sharding Time (Seconds)	Cache Hit Rate
w/o beam search	-	87.0%	0.05	79.1%
w/o greedy grid search	42.90	100.0%	2.12	82.2%
w/o caching	39.99	100.0%	95.87	0.0%
Full NeuroShard	39.99	100.0%	10.75	95.4%

4.5 Application of NeuroShard to Production Models

To answer **RQ5**, we deploy NeuroShard to an ultra-large production DLRM. We used a state-of-the-art hardware platform with RDMA network fabrics, which is detailed in (Mudigere et al., 2022). The model has nearly a thousand embedding tables that demand multi-terabyte memory. The task is to shard these tables to 128 GPUs. We compare NeuroShard with the baselines on embedding cost and training throughput in Table 4. Because we observed out-of-memory errors without a column-wise sharding, for the baselines except for TorchRec, we first apply the column-wise sharding plan proposed by NeuroShard and then run the baselines. Compared with the state-of-the-art (DreamShard), NeuroShard achieves 11.6% improvement in embedding costs, which translates to 6.6% end-to-end training throughput improvement. Note that >5% is considered very significant in our production model since it has been heavily optimized. Also, NeuroShard can shard tables significantly faster in deployment; DreamShard requires RL training when applied to this model, while NeuroShard does not need further training since it directly leverages pre-trained cost models. Note that NeuroShard is trained using the data collected one month before the deployment, so there could

Table 4. Embedding cost and overall training throughput improvement of NeuroShard and baselines on a production model. The results are collected from a training cluster with 128 GPUs.

Sharding Algorithm	Embedding Cost (Milliseconds)	Training Throughput Improvement
Random	118.3	-
Size-based	107.6	+4.0%
Dim-based	90.8	+13.9%
Lookup-based	102.4	+11.9%
Size-lookup-based	109.2	+12.8%
AutoShard	86.6	+32.4%
DreamShard	61.6	+45.3%
TorchRec	86.4	+34.6%
NeuroShard	55.2	+54.9%

be a distribution shift in table indices. The results suggest that we do not need to re-train NeuroShard for at least 1 month in production use.

In this experiment, the whole process (collecting data + training NeuroShard) takes roughly one hour with 128 GPUs, which is minor compared to the 6.6% throughput improvement because 1) training a recommendation model can take up to a week so that NeuroShard can save several hours for each run, and 2) we do not need to frequently re-train NeuroShard, as discussed above.

5 RELATED WORK

DLRMs. DLRMs have been widely adopted in many recommendation scenarios (Zhang et al., 2019; Cheng et al., 2016; Naumov et al., 2019; He et al., 2017; Wang et al., 2020; Lin et al., 2019; Chuang et al., 2020; Chang et al., 2020; Zhou et al., 2021; 2022; Tan et al., 2021a;b; 2020; Liu et al., 2019; Tan et al., 2019; 2023a). To train DLRMs on ultra-large data and model sizes, distributed training solutions have been developed (Acun et al., 2021; Covington et al., 2016; Zhou et al., 2019; Liu et al., 2017; Gomez-Uribe & Hunt, 2015). Embedding table sharding is an important design factor in the distributed training of DLRMs, which has been rarely studied in the literature. NeuroShard provides a deployable embedding table sharding solution to boost the distributed training efficiency of DLRMs.

Embedding Table Sharding. Several recent papers have

Algorithm 2 GreedyGridSearch

```

1: Input:  $T$  embedding tables,  $D$  GPU devices, pre-
   trained neural cost models, grid search hyperparameter
    $M$ ,  $global\_cache$ ,  $col\_plan$ 
2: Generate  $T'$  column-wise sharded tables using  $col\_plan$ 
   where  $T' = T + |col\_plan|$ 
3: Sort the  $T'$  tables in descending order based on the costs
   predicted by the computation cost model
4: Best table-wise sharding plan  $t^* \leftarrow NULL$ 
5: Best computation cost  $cost^* \leftarrow Inf$ 
6: for  $inner\_loop = 1, 2, \dots, M$  do
7:   Get  $max\_dim$  based on  $M$ 
8:   Initialize table-wise sharding plan  $t \leftarrow []$ 
9:   for each of the  $T'$  tables do
10:    Get candidate GPUs that will not cause memory error
    with device dimension smaller than  $max\_dim$ 
11:    for each of the candidate GPUs do
12:      if the tables in GPU are in  $global\_cache$  then
13:        Get  $cost$  from  $global\_cache$ 
14:      else
15:        Get  $cost$  with the computation cost model
        and store the cost into  $global\_cache$ 
16:      end if
17:      Append the GPU with the lowest cost to  $t$ 
18:      if a lower  $t$  is observed then
19:         $t^* \leftarrow t, cost^* \leftarrow cost$ 
20:      end if
21:    end for
22:  end for
23: end for
24: Return  $cost^*, t^*$ 

```

studied embedding table sharding. The pioneering work relies on heuristic cost functions and greedy strategies for sharding (Acun et al., 2021; Lui et al., 2021). RecShard formulates sharding as an optimization problem with mixed integer linear program (Sethi et al., 2022). However, it does not consider the non-linearity of the table costs. FlexShard (Sethi et al., 2023) presents a tailored sharding strategy for sequential DLRM. SurCo (Ferber et al., 2022) solves embedding table sharding by learning linear Surrogate costs. Our previous work uses reinforcement learning (RL) to optimize sharding with learned cost models (Zha et al., 2022a;b). While RL-based methods have achieved significant improvement, they cannot handle very large or costly tables, are expensive to train, and are unstable with high variance due to the stochastic policies in RL. In contrast, NeuroShard pre-trains cost models for a once-for-all purpose and jointly searches for column-wise and table-wise sharding plans. NeuroShard outperforms the RL-based methods in the benchmark sharding dataset and boosts the end-to-end training throughput of a production-scale model.

Embedding Table Compression. In parallel, researchers have studied how to compress embedding tables (Zhang et al., 2020; Shi et al., 2020; Zhao et al., 2020; Joglekar et al., 2020; Liu et al., 2021; Kang et al., 2020; 2021; Pansare et al., 2022; Desai et al., 2022; Lan et al., 2019; Chen et al., 2015). Embedding table sharding is an orthogonal direction to these methods. This is because we often still need to perform table sharding after compression since the compressed tables can be still too large to fit on a single GPU’s memory. Thus, sharding and compression can complement each other with their efficiency improvements. Moreover, embedding compression may lead to an accuracy drop since the embeddings may lose information. Whereas sharding is a lossless optimization on how to partition and place tables so it can improve efficiency without any accuracy loss.

Device placement. Another task that is related to embedding table sharding is the device placement of operations in the neural network. The existing work in this research line either uses reinforcement learning (Mirhoseini et al., 2017; 2018; Gao et al., 2018b; Addanki et al., 2019; Paliwal et al., 2019; Gao et al., 2018a; Goldie & Mirhoseini, 2020) or cost modeling (Lawler et al., 1993; Jia et al., 2019; 2018; Narayanan et al., 2019; Tarnawski et al., 2020). Our work is also based on cost modeling. Unlike regular operations, it is hard to estimate the cost of an embedding operation since the cost depends not only on the operation but also on the indices distributions. Our work presents a learning-based solution for embedding table sharding with pre-trained cost models and online search.

6 CONCLUSION AND FUTURE WORK

In this work, we present NeuroShard, an embedding table sharding framework based on pre-trained neural cost models and online search. We have developed various strategies to train universal and accurate cost models for estimating embedding costs. With the pre-trained cost models, the online search requires minimum computational resources without real GPU execution. We show that NeuroShard not only outperforms the existing sharding algorithms on the benchmark dataset but also significantly boosts the training throughput of an ultra-large production DLRM. In the future, we will extend NeuroShard to row-wise sharding for partitioning large tables. Also, we plan to investigate CPU sharding or mixed CPU-GPU sharding scenarios. Last, we will explore “pre-train, and search” for other system problems.

ACKNOWLEDGEMENTS

Rice University authors are, in part, supported by NSF (#IIS-2224843). The views are those of the authors and should not be interpreted as representing any funding agencies. We’d like to thank the helpful feedback from the anonymous reviewers and Ankur Mallick during the shepherding process.

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 *HPCA*, 2021.
- Addanki, R., Venkatakrishnan, S. B., Gupta, S., Mao, H., and Alizadeh, M. Placeto: learning generalizable device placement algorithms for distributed machine learning. In *NeurIPS*, 2019.
- Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. In *NeurIPS*, 2020.
- Chang, C.-Y., Chen, N., Chiang, W.-T., Lee, C.-H., Tseng, Y.-H., Wang, C.-J., Chen, H.-H., and Tsai, M.-F. Query expansion with semantic-based ellipsis reduction for conversational ir. In *TREC*, 2020.
- Chen, W., Wilson, J., Tyree, S., Weinberger, K., and Chen, Y. Compressing neural networks with the hashing trick. In *ICML*, 2015.
- Cheng, H.-T., Koc, L., Harmsen, J., Shaked, T., Chandra, T., Aradhye, H., Anderson, G., Corrado, G., Chai, W., Ispir, M., et al. Wide & deep learning for recommender systems. In *DLRS Workshop*, 2016.
- Chuang, Y.-N., Chen, C.-M., Wang, C.-J., Tsai, M.-F., Fang, Y., and Lim, E.-P. Tpr: Text-aware preference ranking for recommender systems. In *CIKM*, 2020.
- Chuang, Y.-N., Tang, R., Jiang, X., and Hu, X. Spec: A soft prompt-based calibration on mitigating performance variability in clinical notes summarization. *arXiv preprint arXiv:2303.13035*, 2023.
- Covington, P., Adams, J., and Sargin, E. Deep neural networks for youtube recommendations. In *RecSys*, 2016.
- Desai, A., Chou, L., and Shrivastava, A. Random offset block embedding (robe) for compressed embedding tables in deep learning recommendation systems. *MLSys*, 2022.
- Ferber, A., Huang, T., Zha, D., Schubert, M., Steiner, B., Dilkina, B., and Tian, Y. Surco: Learning linear surrogates for combinatorial nonlinear optimization problems. *arXiv preprint arXiv:2210.12547*, 2022.
- Gao, Y., Chen, L., and Li, B. Post: Device placement with cross-entropy minimization and proximal policy optimization. In *NeurIPS*, 2018a.
- Gao, Y., Chen, L., and Li, B. Spotlight: Optimizing device placement for training deep neural networks. In *ICML*, 2018b.
- Goldie, A. and Mirhoseini, A. Placement optimization with deep reinforcement learning. In *ISPD*, 2020.
- Gomez-Uribe, C. A. and Hunt, N. The netflix recommender system: Algorithms, business value, and innovation. *ACM Transactions on Management Information Systems (TMIS)*, 6(4):1–19, 2015.
- Gupta, U., Wu, C.-J., Wang, X., Naumov, M., Reagen, B., Brooks, D., Cotel, B., Hazelwood, K., Hempstead, M., Jia, B., et al. The architectural implications of facebook’s dnn-based personalized recommendation. In *HPCA*, 2020.
- He, X., Liao, L., Zhang, H., Nie, L., Hu, X., and Chua, T.-S. Neural collaborative filtering. In *WWW*, 2017.
- Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. Deep reinforcement learning that matters. In *AAAI*, 2018.
- Jia, Z., Lin, S., Qi, C. R., and Aiken, A. Exploring hidden dimensions in parallelizing convolutional neural networks. In *ICML*, 2018.
- Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks. In *MLSys*, 2019.
- Joglekar, M. R., Li, C., Chen, M., Xu, T., Wang, X., Adams, J. K., Khaitan, P., Liu, J., and Le, Q. V. Neural input search for large scale recommendation models. In *KDD*, 2020.
- Kang, W.-C., Cheng, D. Z., Chen, T., Yi, X., Lin, D., Hong, L., and Chi, E. H. Learning multi-granular quantized embeddings for large-vocab categorical features in recommender systems. In *WWW*, 2020.
- Kang, W.-C., Cheng, D. Z., Yao, T., Yi, X., Chen, T., Hong, L., and Chi, E. H. Learning to embed categorical features without embedding tables for recommendation. In *KDD*, 2021.
- Khudia, D., Huang, J., Basu, P., Deng, S., Liu, H., Park, J., and Smelyanskiy, M. Fbgemm: Enabling high-performance low-precision deep learning inference. *arXiv preprint arXiv:2101.05615*, 2021.
- Kulkarni, T. D., Narasimhan, K., Saeedi, A., and Tenenbaum, J. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *NeurIPS*, 2016.
- Kumar, A., Zhou, A., Tucker, G., and Levine, S. Conservative q-learning for offline reinforcement learning. *NeurIPS*, 2020.
- Lai, K.-H., Zha, D., Li, Y., and Hu, X. Dual policy distillation. In *IJCAI*, 2020a.

- Lai, K.-H., Zha, D., Zhou, K., and Hu, X. Policy-gnn: Aggregation optimization for graph neural networks. In *KDD*, 2020b.
- Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., and Soricut, R. Albert: A lite bert for self-supervised learning of language representations. In *ICLR*, 2019.
- Lawler, E. L., Lenstra, J. K., Kan, A. H. R., and Shmoys, D. B. Sequencing and scheduling: Algorithms and complexity. *Handbooks in operations research and management science*, 4:445–522, 1993.
- Li, Y., Chen, Z., Zha, D., Zhou, K., Jin, H., Chen, H., and Hu, X. Automated anomaly detection via curiosity-guided search and self-imitation learning. *IEEE Transactions on Neural Networks and Learning Systems*, 33(6): 2365–2377, 2021.
- Li, Y., Zha, D., Zhang, T., Yan, F. Y., Suh, G. E., and Delimitrou, C. Towards handling metastable failures in distributed systems with offline reinforcement learning. In *ICLR*, 2023.
- Lian, X., Yuan, B., Zhu, X., Wang, Y., He, Y., Wu, H., Sun, L., Lyu, H., Liu, C., Dong, X., et al. Persia: An open, hybrid system scaling deep learning-based recommenders up to 100 trillion parameters. In *KDD*, 2022.
- Lin, S.-C., Chuang, Y.-N., Yang, S.-F., Tsai, M.-F., and Wang, C.-J. Negative-aware collaborative filtering. In *RecSys*, 2019.
- Liu, D. C., Rogers, S., Shiao, R., Kislyuk, D., Ma, K. C., Zhong, Z., Liu, J., and Jing, Y. Related pins at pinterest: The evolution of a real-world recommender system. In *WWW*, 2017.
- Liu, N., Tan, Q., Li, Y., Yang, H., Zhou, J., and Hu, X. Is a single vector enough? exploring node polysemy for network embedding. In *KDD*, 2019.
- Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., and Neubig, G. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35, 2023.
- Liu, S., Gao, C., Chen, Y., Jin, D., and Li, Y. Learnable embedding sizes for recommender systems. In *ICLR*, 2021.
- Lui, M., Yetim, Y., Özkan, Ö., Zhao, Z., Tsai, S.-Y., Wu, C.-J., and Hempstead, M. Understanding capacity-driven scale-out neural recommendation inference. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 162–171. IEEE, 2021.
- Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., Kumar, N., Norouzi, M., Bengio, S., and Dean, J. Device placement optimization with reinforcement learning. In *ICML*, 2017.
- Mirhoseini, A., Goldie, A., Pham, H., Steiner, B., Le, Q. V., and Dean, J. A hierarchical model for device placement. In *ICLR*, 2018.
- Mudigere, D., Hao, Y., Huang, J., Jia, Z., Tulloch, A., Sridharan, S., Liu, X., Ozdal, M., Nie, J., Park, J., et al. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *ISCA*, 2022.
- Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. Pipedream: generalized pipeline parallelism for dnn training. In *SOSP*, 2019.
- Naumov, M., Mudigere, D., Shi, H.-J. M., Huang, J., Sundaraman, N., Park, J., Wang, X., Gupta, U., Wu, C.-J., Azzolini, A. G., et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.
- Naumov, M., Kim, J., Mudigere, D., Sridharan, S., Wang, X., Zhao, W., Yilmaz, S., Kim, C., Yuen, H., Ozdal, M., et al. Deep learning training in facebook data centers: Design of scale-up and scale-out systems. *arXiv preprint arXiv:2003.09518*, 2020.
- Oh, J., Guo, Y., Singh, S., and Lee, H. Self-imitation learning. In *ICML*, 2018.
- Paliwal, A., Gimeno, F., Nair, V., Li, Y., Lubin, M., Kohli, P., and Vinyals, O. Reinforced genetic algorithm learning for optimizing computation graphs. In *ICLR*, 2019.
- Pansare, N., Katukuri, J., Arora, A., Cipollone, F., Shaik, R., Tokgozoglou, N., and Venkataraman, C. Learning compressed embeddings for on-device inference. *MLSys*, 2022.
- Sethi, G., Acun, B., Agarwal, N., Kozyrakis, C., Trippel, C., and Wu, C.-J. Recshard: Statistical feature-based memory optimization for industry-scale neural recommendation. In *ASPLOS*, 2022.
- Sethi, G., Bhattacharya, P., Choudhary, D., Wu, C.-J., and Kozyrakis, C. Flexshard: Flexible sharding for industry-scale sequence recommendation models. *arXiv preprint arXiv:2301.02959*, 2023.
- Shi, H.-J. M., Mudigere, D., Naumov, M., and Yang, J. Compositional embeddings using complementary partitions for memory-efficient recommendation systems. In *KDD*, 2020.

- Tan, Q., Liu, N., and Hu, X. Deep representation learning for social network analysis. *Frontiers in big Data*, 2:2, 2019.
- Tan, Q., Liu, N., Zhao, X., Yang, H., Zhou, J., and Hu, X. Learning to hash with graph neural networks for recommender systems. In *WWW*, 2020.
- Tan, Q., Zhang, J., Liu, N., Huang, X., Yang, H., Zhou, J., and Hu, X. Dynamic memory based attention network for sequential recommendation. In *AAAI*, 2021a.
- Tan, Q., Zhang, J., Yao, J., Liu, N., Zhou, J., Yang, H., and Hu, X. Sparse-interest network for sequential recommendation. In *WSDM*, 2021b.
- Tan, Q., Liu, N., Huang, X., Choi, S.-H., Li, L., Chen, R., and Hu, X. S2gae: Self-supervised graph autoencoders are generalizable learners with graph masking. In *WSDM*, 2023a.
- Tan, Q., Zhang, X., Liu, N., Zha, D., Li, L., Chen, R., Choi, S.-H., and Hu, X. Bring your own view: Graph neural networks for link prediction with personalized subgraph selection. In *WSDM*, 2023b.
- Tang, R., Chuang, Y.-N., and Hu, X. The science of detecting llm-generated texts. *arXiv preprint arXiv:2303.07205*, 2023.
- Tarnawski, J. M., Phanishayee, A., Devanur, N., Mahajan, D., and Nina Paravecino, F. Efficient algorithms for device placement of dnn graph operators. In *NeurIPS*, 2020.
- Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.
- Wang, C.-J., Chuang, Y.-N., Chen, C.-M., and Tsai, M.-F. Skewness ranking optimization for personalized recommendation. In *UAI*, 2020.
- Zha, D., Lai, K.-H., Zhou, K., and Hu, X. Experience replay optimization. In *IJCAI*, 2019.
- Zha, D., Lai, K.-H., Wan, M., and Hu, X. Meta-aad: Active anomaly detection with deep reinforcement learning. In *ICDM*, 2020.
- Zha, D., Lai, K.-H., Huang, S., Cao, Y., Reddy, K., Vargas, J., Nguyen, A., Wei, R., Guo, J., and Hu, X. Rlcard: a platform for reinforcement learning in card games. In *IJCAI*, 2021a.
- Zha, D., Lai, K.-H., Zhou, K., and Hu, X. Simplifying deep reinforcement learning via self-supervision. *arXiv preprint arXiv:2106.05526*, 2021b.
- Zha, D., Ma, W., Yuan, L., Hu, X., and Liu, J. Rank the episodes: A simple approach for exploration in procedurally-generated environments. In *ICLR*, 2021c.
- Zha, D., Xie, J., Ma, W., Zhang, S., Lian, X., Hu, X., and Liu, J. Douzero: Mastering douzhu with self-play deep reinforcement learning. In *ICML*, 2021d.
- Zha, D., Feng, L., Bhushanam, B., Choudhary, D., Nie, J., Tian, Y., Chae, J., Ma, Y., Kejariwal, A., and Hu, X. Autoshard: Automated embedding table sharding for recommender systems. In *KDD*, 2022a.
- Zha, D., Feng, L., Tan, Q., Liu, Z., Lai, K.-H., Bhargav, B., Tian, Y., Kejariwal, A., and Hu, X. Dreamshard: Generalizable embedding table placement for recommender systems. In *NeurIPS*, 2022b.
- Zha, D., Lai, K.-H., Tan, Q., Ding, S., Zou, N., and Hu, X. B. Towards automated imbalanced learning with deep hierarchical reinforcement learning. In *CIKM*, 2022c.
- Zha, D., Bhat, Z. P., Lai, K.-H., Yang, F., and Hu, X. Data-centric ai: Perspectives and challenges. In *SDM*, 2023a.
- Zha, D., Bhat, Z. P., Lai, K.-H., Yang, F., Jiang, Z., Zhong, S., and Hu, X. Data-centric artificial intelligence: A survey. *arXiv preprint arXiv:2303.10158*, 2023b.
- Zhang, C., Liu, Y., Xie, Y., Ktena, S. I., Tejani, A., Gupta, A., Myana, P. K., Dilipkumar, D., Paul, S., Ihara, I., et al. Model size reduction using frequency based double hashing for recommender systems. In *RecSys*, 2020.
- Zhang, S., Yao, L., Sun, A., and Tay, Y. Deep learning based recommender system: A survey and new perspectives. *ACM Computing Surveys (CSUR)*, 52(1):1–38, 2019.
- Zhao, X., Wang, C., Chen, M., Zheng, X., Liu, X., and Tang, J. Autoemb: Automated embedding dimensionality search in streaming recommendations. In *SIGIR*, 2020.
- Zhou, G., Mou, N., Fan, Y., Pi, Q., Bian, W., Zhou, C., Zhu, X., and Gai, K. Deep interest evolution network for click-through rate prediction. In *AAAI*, 2019.
- Zhou, H., Tan, Q., Huang, X., Zhou, K., and Wang, X. Temporal augmented graph neural networks for session-based recommendations. In *SIGIR*, 2021.
- Zhou, H., Fan, J., Huang, X., Li, K. H., Tang, Z., and Yu, D. Multi-interest refinement by collaborative attributes modeling for click-through rate prediction. In *CIKM*, 2022.
- Zhou, H., Zhou, S., Duan, K., Huang, X., Tan, Q., and Yu, Z. Interest driven graph structure learning for session-based recommendation. In *PAKDD*, 2023.

A DETAILS OF ANALYTICAL EXPERIMENTS

In this section, we provide more details on the three analytical experiments performed in Section 2.

A.1 Impact of Dimension

In this experiment, we aim to study how the dimension impacts the computation costs of the tables. In the DLRM dataset (Naumov et al., 2019), we have the table indices and hash sizes of the tables. The dimensions are not specified. Thus, we vary the dimension from the set of $\{128, 64, 32, 8, 4\}$ to study the influence of dimension. Note that for all the above five cases, we use the same hash size and the same table indices, and the only difference is the dimension. We have tried multiple tables and observed similar patterns. The results for some other randomly selected tables are visualized in Figure 10.

A.2 Multi-table Costs vs the Sum of Single-table Costs

This analytical experiment aims to reproduce the results in (Zha et al., 2022a) to understand the relationship between the multi-table costs and the sums of the single-table costs. We randomly sample 50 data points to plot the scatters, where each data point consists of 10 tables from the 856 tables in the DLRM dataset (Naumov et al., 2019). For each of the data points, we measure two costs on a single GPU:

- **Multi-table cost:** we directly run the fused operation on GPU (forward and backward passes). We first run the operation 10 times to warm up the hardware. Then we run another 100 times and use the median cost as the multi-table cost.
- **Sum of single-table costs:** we collect the cost for every single table following the same process. Then we sum the single-table costs.

We observe that the sum of single-table costs is often larger than the multi-table costs because the fused operation is faster.

A.3 Communication Cost Analysis

This analytical experiment aims to provide an understanding of the relationship between the max device dimension and the max forward/backward communication cost. To simulate the scenarios with different max device dimensions, we use the same random table placement generation strategy described in Section 3.1, with a greedy strategy to balance the dimensions equipped with randomness to cover different scenarios. A complete generation process is summarized in Algorithm 5. For each data point (i.e., a placement), we run the forward all-to-all communication and the backward

communication to collect the costs from the GPUs in a single server. Similar to computation costs, we first run the communication 10 times to warm up the hardware. Then we run the communication another 100 times and use the median costs as the communication costs. Note that each GPU will have a different locally measured cost. We only focus on the max cost since it is the bottleneck.

B DETAILS OF SYNTHETIC INPUTS GENERATION

In this section, we give more details about how we generate synthetic data to have good coverage for benchmarking.

B.1 Table Augmentation

The key idea is to generate various dimensions for a single table. Given a list of dimensions, for each table in the pool, we associate the table with each of the dimensions as an augmented table. Algorithm 3 summarizes the augmentation procedure.

Algorithm 3 Table Augmentation

```

1: Input:  $T$  embedding tables, a set of dimensions  $\mathcal{D}$ 
2: Augmented table pool  $\mathcal{P} \leftarrow \{\}$ 
3: for each of the  $T$  tables do
4:   for each of the dimensions in  $\mathcal{D}$  do
5:     The augmented table  $aug\_table \leftarrow$  the selected
     table with the selected dimension
6:     Add  $aug\_table$  to  $\mathcal{P}$ 
7:   end for
8: end for
9: Return  $\mathcal{P}$ 

```

B.2 Random Table Combination Generation

We summarize the process of generating random data table combinations in Algorithm 4.

Algorithm 4 Random Table Combination Generation

```

1: Input: Augmented table pool  $\mathcal{P}$ , min number of table  $T_{\min}$ , max number of table  $T_{\max}$ , number of table combinations we aim to generate  $N_{\text{com}}$ 
2: Table combinations  $\mathcal{T}_{\text{com}} \leftarrow \{\}$ 
3: for  $i = 0, 1, \dots, N_{\text{com}}$  do
4:   Uniformly sample the number of tables  $T$  in  $[T_{\min}, T_{\max}]$ 
5:   Randomly sample  $T$  tables from  $\mathcal{P}$  and append this combination to  $\mathcal{T}_{\text{com}}$ 
6: end for
7: Return  $\mathcal{T}_{\text{com}}$ 

```

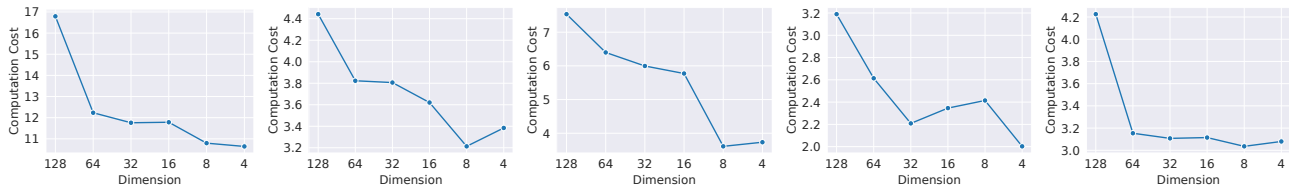


Figure 10. Computation costs w.r.t. dimensions for some other randomly selected tables.

B.3 Random Table Placement Generation

We summarize the table placement generation in Algorithm 5.

Algorithm 5 Random Table Placement Generation

- 1: **Input:** Augmented table pool \mathcal{P} , min number of table T_{\min} , max number of table T_{\max} , number of table placements we aim to generate N_{place} , number of GPU devices D .
 - 2: Table placements $\mathcal{T}_{\text{place}} \leftarrow \{\}$
 - 3: **for** $i = 0, 1, \dots, N_{\text{place}}$ **do**
 - 4: Uniformly sample the number of tables T in $[T_{\min}, T_{\max}]$
 - 5: Randomly sample T tables from \mathcal{P}
 - 6: Sort the T tables in descending order based on the table dimension
 - 7: Uniformly sample a probability of applying greedy strategy $p \in [0, 1]$
 - 8: **for** each of the T tables **do**
 - 9: Randomly sample $p' \in [0, 1]$
 - 10: Obtain the candidate GPU devices that will not cause a memory error.
 - 11: **if** $p' \leq p$ **then**
 - 12: Assign the current table to the candidate GPU with the lowest device dimension
 - 13: **else**
 - 14: Randomly assign the current table to one of the candidate GPUs
 - 15: **end if**
 - 16: **end for**
 - 17: Append the placement to $\mathcal{T}_{\text{place}}$
 - 18: **end for**
 - 19: Return $\mathcal{T}_{\text{place}}$
-

C DETAILS OF COST MODEL TRAINING

In this section, we provide more details of the neural architecture and loss functions for training the neural cost models.

Neural architectures. For the computation cost model, we use an MLP with a size of 128-32 to process the table features and another MLP with a size of 32-64. For the communication cost model, we use an MLP with a size of

128-64-32-16.

Loss functions. We use mean squared error (MSE) to update both the computation cost model and communication cost model. Specifically, let \mathbf{x} be the features (can be either table features or communication features), y be the ground truth, and g be the cost model (can be either computation cost model or communication model). Let N_{sample} be the number of samples. Then the loss is

$$\mathcal{L} = \sum_{i=1}^{N_{\text{sample}}} \text{MSE}(\mathbf{x}_i, y_i), \quad (2)$$

where $\text{MSE}(\cdot, \cdot)$ represents the MSE loss. In practice, we can use mini-batch training.

D DATASETS

We use the benchmark dataset for evaluating embedding table sharding (Naumov et al., 2019). It is publicly available as https://github.com/facebookresearch/dlrm_datasets. It contains 856 synthetic tables whose indices distributions are similar to the production workloads in Meta. The statistics of the datasets are well summarized in previous work. Please see the appendices in (Zha et al., 2022a;b) for details.

We summarize the 12 sharding tasks we used in our experiments in Table 5. All the sharding tasks have a constraint of 4 GB memory in each GPU.

Discussion of public datasets. Table 6 compares the scale of DLRM datasets with several large-scale public datasets. DLRM has significantly more tables, a larger average hash size, and a larger average pooling factor than these public datasets. The embedding costs on these datasets will always be very small, no matter how we do sharding. For example, DLRM has at least 30x more tables than Criteo, at least 200x larger average hash size, and a 15x larger average pooling factor. So, as the embedding cost of the DLRM dataset is from the range of 17 ms to 40 ms (Table 1 in our paper), the embedding cost of Criteo could be only around 1 ms or even smaller. Thus, there is no need to do sharding on the Criteo dataset. So these datasets are not suitable for evaluating embedding table sharding algorithms.

Table 5. Sharding tasks generated in the experiments.

Number of GPUs	Range of the Number of Tables	Range of Table Dimensions
4	10-60	4
4	10-60	4, 8
4	10-60	4, 8, 16
4	10-60	4, 8, 16, 32
4	10-60	4, 8, 16, 64
4	10-60	4, 8, 16, 64, 128
8	20-120	4
8	20-120	4, 8
8	20-120	4, 8, 16
8	20-120	4, 8, 16, 32
8	20-120	4, 8, 16, 64
8	20-120	4, 8, 16, 64, 128

Table 6. Comparison of embedding table feature statistics between some popular public recommendation datasets and the industrial-scale DLRM dataset.

Dataset	# of Tables	Avg. hash size	Avg. pooling factor	Link	
Public	Criteo	26	17,839	1	https://www.kaggle.com/c/criteo-display-ad-challenge
	Avazu	23	67,152	1	https://www.kaggle.com/c/avazu-ctr-prediction/data
	KDD	10	601,908	1	https://www.kaggle.com/c/kddcup2012-track2/data
Industrial-Scale	DLRM	856	4,107,458	15	https://github.com/facebookresearch/dlrm_datasets

E BASELINES

In this section, we introduce the details of all the baselines used in our experiments.

E.1 Greedy Algorithms

The greedy sharding algorithms have been used in previous papers of distributed recommender systems (Acun et al., 2021; Lui et al., 2021). The main idea is to use a greedy algorithm to balance the costs by assigning the table to the device with the lowest cost so far in each step, where the costs are estimated in different ways. Specifically, greedy algorithms consist of two steps as follows.

- **Designing a cost function:** we give each table a cost to quantify the expected running time on the device. The cost is the objective that we want to balance.
- **Greedy allocation:** the objective is to balance the sum of the costs in each device. To achieve this, we first sort the embedding tables in descending order based on the costs defined by the cost function. The sorting can make it more easily to achieve a balance if we allocate the tables greedily. Then, we assign tables starting from the table with the highest cost. In each step, we make a greedy decision by assigning the current table to the device that has the lowest sum of the cost so far. This sorting-enhanced greedy strategy can enable each device to have roughly a similar sum of the costs.

The four baselines differ in how the cost function is designed. This can significantly impact the balance since the cost function determines our optimization objective. We summarize the used cost functions as follows:

- **Size-based:** we use the table size as the cost function. The idea is that balancing table size can reduce the risk of getting out-of-memory errors. Also, table size is also positively correlated with dimension so it can also reflect the workloads.
- **Dim-based:** we use the table dimension as the cost function. Table dimension is an important feature to represent the cost since it can decide both computation and communication workloads. Thus, it is natural to balance the sums of dimensions (i.e., the device dimension).
- **Lookup-based:** we use the product of the table dimension and the mean pooling factor as the cost function. The intuition is that the table dimension and the pooling factor can determine the computation workload in embedding lookup.
- **Size-lookup-based:** we use the product of the table dimension, the mean pooling factor, and the table size as the cost function. This is a more comprehensive cost function that considers both lookup cost and table sizes.

E.2 Reinforcement Learning Algorithms

We have included two state-of-the-art reinforcement learning algorithms for embedding table sharding (Zha et al., 2022a;b). They share similar ideas with differences in optimization objectives and training methods. We summarize them as follows.

- **AutoShard** (Zha et al., 2022a): it trains an LSTM controller to perform sharding to balance computation cost. The objective is the degree of balance, which is defined as the min cost divided by the max cost. The code is available as <https://github.com/daochenzha/autoshard>
- **DreamShard** (Zha et al., 2022b): it extends AutoShard by also balancing communication. It also extends the cost model to communication. It additionally introduces an estimated MDP to make training and inference much faster. The code is available at <https://github.com/daochenzha/dreamshard>

E.3 Planing Algorithms

In parallel to reinforcement learning, planning algorithms identify the sharding plan with search. TorchRec provides a planning-based sharding strategy. For a fair comparison, we allow TorchRec to search for both column-wise and table-wise sharding plans. However, TorchRec still relies on heuristic costs so we do not see a clear improvement of TorchRec over the greedy algorithms. The code is publicly available at <https://github.com/pytorch/torchrec>.

F HYPERPARAMETERS AND CONFIGURATIONS

In this section, we list all the hyperparameters of NeuroShard. We also list the hardware/software configurations

- **Generating synthetic inputs:** we augment the table pool with dimensions $\{4, 8, 16, 32, 64, 128\}$. We randomly select 1 to 15 tables for the table combination generation and N_{place} tables for the table placement generation, where $10 \leq N_{\text{place}} \leq 60$ for 4 GPUs and $20 \leq N_{\text{place}} \leq 120$ for 8 GPUs. The starting timestamp for the communication data is sampled from 0 to 20 milliseconds. We generate 100K samples for each of the cost models.
- **Training neural cost models:** We use 80% of the data for training, 10% of the data for validation, and 10% of the data for testing. We set the batch size to 512. We use Adam optimizer with a learning rate of 0.001 with the other configurations as the default. We train

1000 epochs and save the model that can deliver the best results on the validation data.

- **Online search:** we set $N = 10$, $K = 3$, $L = 10$, and $M = 11$.
- **Software:** we use PyTorch 1.9.1, and FBGEMM 0.0.1
- **Hardware:** we conduct the experiments on a server with 48 Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz processors, 188 GB memory, and eight NVIDIA GeForce RTX 2080 Ti GPUs.

G ADDITIONAL ABLATION RESULTS

The results are shown in Table 7.

H ADDITIONAL DISCUSSION OF SEARCH WITH REINFORCEMENT LEARNING

In this work, we used beam search and greedy grid search to identify in search. However, this search process could be further accelerated by training a meta-policy (Zha et al., 2020; Lai et al., 2020b) with reinforcement learning (Zha et al., 2021a) and transferring it across tasks. Here, we highlight several potential strategies that we are trying. **1) Hierarchical reinforcement learning:** The idea is to decompose the sharding task into several sub-tasks (Kulkarni et al., 2016; Zha et al., 2022c). This can be naturally applied to the embedding table sharding problem, as there are two hierarchies of search: column-wise sharding and table-wise sharding. **2) Self-imitation learning:** The idea is to select the highly-rewarded samples and use supervise losses to encourage the policy to reproduce the good behaviors (Oh et al., 2018; Zha et al., 2021c;b; Li et al., 2021). This could be helpful since we often have lots of system logs of sharding. The idea is to select good sharding plans from the system log and use supervised losses to train a policy. **2) Offline reinforcement learning:** The idea is to learn the optimal strategy based on offline data (Kumar et al., 2020; Li et al., 2023). This can also be applied to the offline sharding log.

Note that the reinforcement learning meta-policy could also be combined with search to guide the search process.

I ARTIFACT

This section is for readers who are interested in reproducing our results. In what follows, we will describe how to download the dataset, how to install NeuroShard and its dependencies, how to run NeuroShard, and how to collect latencies from the hardware.

I.1 Artifact check-list (meta-information)

- **Algorithm:** Pre-trained models, beam search, grid search.

Table 7. Ablation study with a maximum dimension of 128 and 8 GPUs. w/o beam search means removing the beam search in col-wise sharding. w/o greedy grid search suggests not grid-searching the table dimension threshold. w/o caching disables the caching mechanism of computation costs.

	Cost (Milliseconds)	Success Rate	Sharding Time (Seconds)	Cache Hit Rate
w/o beam search	-	63.0%	0.09	75.4%
w/o greedy grid search	55.97	100.0%	4.09	79.8%
w/o caching	49.10	100.0%	164.21	0.0%
Full NeuroShard	49.10	100.0%	20.79	93.0%

- **Program:** Implemented using PyTorch with Python.
- **Compilation:** PyTorch 1.8.0 and Python 3.8.0.
- **Data set:** Synthetic embedding data open-sourced by Meta.
- **Run-time environment:** Ubuntu 18.04.6 LTS.
- **Hardware:** Eight NVIDIA GeForce RTX 2080 Ti GPUs.
- **Metrics:** Latency in milliseconds.
- **Output:** Command line outputs.
- **Experiments:** Generate sharding plans and collect latencies.
- **How much disk space required (approximately)?:** 20 GB.
- **How much time is needed to prepare workflow (approximately)?:** 2 hours.
- **How much time is needed to complete experiments (approximately)?:** 20+ hours.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT.
- **Archived (provide DOI)?:** <https://zenodo.org/badge/latestdoi/556106683>

I.2 Description

I.2.1 How delivered

The artifact is zipped and available at <https://zenodo.org/badge/latestdoi/556106683>. The source code is also publicly released on GitHub at <https://github.com/daochenzha/neuroshard>, with a README file that provides step-to-step instructions to run the code.

I.2.2 Hardware dependencies

Our results are collected from a server with eight NVIDIA GeForce RTX 2080 Ti GPUs. Other types of GPUs are also acceptable but the results may vary since the collected latencies are GPU-dependent.

I.2.3 Software dependencies

To run the code, Python 3.8.0 or higher (we used 3.8.0) is required. Additionally, an appropriate version of CUDA must be installed to utilize GPUs. Lastly, FBGEMM, an open-sourced embedding table operation, must be installed.

I.2.4 Data sets

The synthetic dataset is publicly available at https://github.com/facebookresearch/dlrm_datasets.git. The dataset can be downloaded with Git LFS by running the following commands:

```
git lfs install --skip-smudge
git clone \
  https://github.com/facebookresearch/dlrm_datasets.git
cd dlrm_datasets
git lfs pull \
  --include=embedding_bag/2021/fbgemm_t856_bs65536.pt.gz
gzip -d embedding_bag/2021/fbgemm_t856_bs65536.pt.gz
```

A file named `fbgemm_t856_bs65536.pt` will be obtained in `embedding_bag/2021/` after running the above commands, and its size is 4.0 GB. This file contains synthetic indices for embedding lookups, which share similar indices distributions as the Meta production environment.

I.3 Installation

Firstly, we need to install Python 3.8.0+ and CUDA. Secondly, we install PyTorch 1.8.0 with the following:

```
pip3 install torch==1.8.0
```

Thirdly, we need to install the open-sourced FBGEMM, which is available at https://github.com/pytorch/FBGEMM/tree/main/fbgemm_gpu.

Note that, except for A100 or V100 GPUs, FBGEMM needs to be built manually following the instructions, which is expected to take 0.5 to 1 hour. Finally, clone the NeuroShard code and install it:

```
git clone \
  https://github.com/anonymoussubmission/neuroshard.git
cd neuroshard
pip3 install -e .
```

I.4 Experiment workflow

The experiment consists of four main steps:

1. We process the raw synthetic data with a script and randomly construct some sharding tasks for evaluation.
2. We do micro-benchmarking on hardware to collect computation and communication costs. These cost data will be saved in a file.
3. Based on the collected cost data, we pre-train cost models. Specifically, we train three models, including a computation cost model, a forward communication cost model, and a backward communication cost model.
4. Using the pre-trained cost models, we can run the on-line search in NeuroShard for embedding table sharding. We can also run the baseline heuristic sharding algorithms. The performance can be evaluated by using the cost models (simulation) or collecting costs from the hardware.

Now we walk through the above steps one by one. To begin with, we need to ensure that the current directory is in `neuroshard/` and put the `fbgemm_t856_bs65536.pt` file in the current directory.

In Step 1, we run the following:

```
python3 tools/gen_dlr_data.py \
  --data fbgemm_t856_bs65536.pt
```

Here, `--data` specifies the path of the raw data. The expected output is:

```
Processing DLRM data...
Generating table configs...
```

The processed data will be saved in `data/dlr_data` by default. After this, we generate sharding tasks with

```
python3 tools/gen_tasks.py --max_dim 128
```

Here, `--max_dim` specifies the maximum table dimension (the second row in Table 1). We used 128 as an example to show how to run the code. The expected output is:

```
100 sharding tasks generated!
```

The sharding tasks will be saved in `data/tasks/4_gpus` by default. **In Step 2**, we collect cost data with micro-benchmarking. We first collect computation cost data:

```
python3 collect_compute_cost_data.py --data_size 10
```

`--data_size` specifies how many data samples will be generated. We use a small number here so that the program runs faster. To reproduce the results, `--data_size` should be set as 100K. The expected final two lines should be similar to the following (the exact values in the first line may differ):

```
2997,436,2332,3728,535 3.4415176584173777
Device 0 finished!
```

Similarly, we can collect communication cost data by running the following script:

```
python3 collect_comm_cost_data.py --data_size 10
```

We also set `--data_size` to be small here to make it run faster. 100K should be set instead to reproduce the results. The last line of the expected output should be:

```
Evaluator sub-process terminated!
```

After running the above two scripts, the cost data is expected to be stored in `data/cost_data/`. **In Step 3**, based on the collected cost data, we pre-train neural cost models. We train the computation cost model by running the following command:

```
python train_compute_cost_model.py --epochs 3
```

`--epochs` means the number of epochs of training. We set it to be 3 here to make the training faster. To reproduce the result, `--epochs` needs to be set to 1000. The last line of the expected output is (the exact values may differ):

```
Final result, train MSE: 8.861547689180117,
valid MSE 8.398209571838379, test MSE: 8.432512283325195
```

Then we do similar things again for training communication cost models. We can run the following command:

```
python3 train_comm_cost_model.py --epochs 3
```

Again, we set `--epochs` to be 3 to make the training faster, and 1000 is needed to reproduce the results. The last line of the expected output is (the exact values may differ):

```
Final result, train MSE: 31.95864486694336,
valid MSE 42.04985046386719, test MSE: 23.29010772705078
```

After training, there should be three models stored in `models/`. **In Step 4**, we evaluate different sharding algorithms with simulation or with real hardware. To get simulation results, we can use the following commands to run NeuroShard and the heuristic baselines:

```
python3 eval_simulator.py --alg neuroshard
python3 eval_simulator.py --alg random
python3 eval_simulator.py --alg dim_greedy
python3 eval_simulator.py --alg lookup_greedy
python3 eval_simulator.py --alg size_greedy
python3 eval_simulator.py --alg size_lookup_greedy
```


Similarly, we evaluate the cost on real hardware with

```
python3 eval.py --alg neuroshard
python3 eval.py --alg random
python3 eval.py --alg dim_greedy
python3 eval.py --alg lookup_greedy
python3 eval.py --alg size_greedy
python3 eval.py --alg size_lookup_greedy
```

I.5 Evaluation and expected result

For each of the scripts in Step 4, the final result will be printed out in the terminal. The cost models need to be trained with `--data_size` of 100K and `--epochs` of 3 in order to get similar results shown below. For the simulation result of NeuroShard, the expected output is:

```
Average: 39.0441405081749
Valid 100 / 100
```

The first line is the average latency. The second line means NeuroShard succeeds on all the tasks, i.e., no memory error. If running `size_lookup_greedy`, the result will be:

```
Average: 47.95177095494372
Valid 94 / 100
```

It only succeeds in 94 out of 100 tasks. For the results on real hardware, the expected output of NeuroShard is:

```
Average: 39.98647058823529
Valid 100 / 100
```

The expected result of `size_lookup_greedy` is:

```
Average: 48.010588235294115
Valid 94 / 100
```

The simulation and real costs are consistent. The above procedure only provides one column of the results in Table 1. To get full results, we need to customize the `--max_dim`, `--T_range`, and `--max_mem` in `gen_tasks.py`, and also the corresponding arguments in `collect_comm_cost_data.py`, `eval.py`, and `eval_simulator.py`.

I.6 Notes

The cost is highly dependent on the GPUs used, and different PyTorch versions may produce varying results. Additionally, a result itself may have a variance, so the results obtained on another machine may differ from those shown above.