AutoShard: Automated Embedding Table Sharding for Recommender Systems

Daochen Zha[1], Louis Feng[2], Bhargav Bhushanam[2], Dhruv Choudhary[2], Jade Nie[2], Yuandong Tian[2], Jay Chae[2], Yinbin Ma[2], Arun Kejariwal[2], Xia Hu[1]
{daochen.zha,xia.hu}@rice.edu,{lofe,bbhushanam,choudharydhruv,qnie,yuandong,jchae,yinbin,akejariwal}@fb.com
[1]Department of Computer Science, Rice University, Houston, TX, USA
[2]Meta Platforms, Inc., Menlo Park, CA, USA

ABSTRACT

Embedding learning is an important technique in deep recommendation models to map categorical features to dense vectors. However, the embedding tables often demand an extremely large number of parameters, which become the storage and efficiency bottlenecks. Distributed training solutions have been adopted to partition the embedding tables into multiple devices. However, the embedding tables can easily lead to imbalances if not carefully partitioned. This is a significant design challenge of distributed systems named embedding table sharding, i.e., how we should partition the embedding tables to balance the costs across devices, which is a non-trivial task because 1) it is hard to efficiently and precisely measure the cost, and 2) the partition problem is known to be NP-hard. In this work, we introduce our novel practice in Meta, namely AutoShard, which uses a neural cost model to directly predict the multi-table costs and leverages *deep reinforcement learning* to solve the partition problem. Experimental results on an open-sourced large-scale synthetic dataset and Meta's production dataset demonstrate the superiority of AutoShard over the heuristics. Moreover, the learned policy of AutoShard can transfer to sharding tasks with various numbers of tables and different ratios of the unseen tables without any fine-tuning. Furthermore, AutoShard can efficiently shard hundreds of tables in seconds. The effectiveness, transferability, and efficiency of AutoShard make it desirable for production use. Our algorithms have been deployed in Meta production environment. A prototype is available at https://github.com/daochenzha/autoshard

KEYWORDS

Recommender System; Reinforcement Learning; Cost Modeling

ACM Reference Format:
Daochen Zha[1], Louis Feng[2], Bhargav Bhushanam[2], Dhruv Choudhary[2], Jade Nie[2], Yuandong Tian[2], Jay Chae[2], Yinbin Ma[2], Arun Kejariwal[2], Xia Hu[1]. 2022. AutoShard: Automated Embedding Table Sharding for Recommender Systems. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '22), August 14–18, 2022, Washington, DC, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3534678.3539034

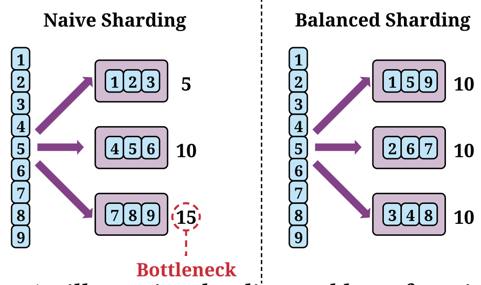


Figure 1: An illustrative sharding problem of partitioning 9 embedding tables across 3 devices with naive sharding and balanced sharding. Blue blocks are embedding tables, whose numbers indicate the costs measured by the operation execution time. Purple blocks are shards, whose costs are usually smaller than the sum of the table costs within the shard due to parallelism (e.g., 5 < 1+2+3 and 10 < 4+5+6). The slowest shard will become the bottleneck since the other shards have to wait until it finishes. Optimizing the naive sharding in this task can achieve 1.5X speedup (i.e., 15/10 = 1.5).

1 INTRODUCTION

Embedding learning has become an important technique for modeling categorical features in deep recommendation models [37]. It maps sparse categorical features into dense vectors by performing embedding lookup in embedding tables. The learned vectors are then used for complex feature interactions and can greatly help us improve the prediction results (e.g., DeepFM [4], AutoInt [29], and deep learning recommendation model (DLRM) [25]).

However, industrial recommendation models often demand an extremely large number of parameters for embedding tables, which become the storage and efficiency bottlenecks [38]. A typical example is YouTube Recommendation Systems [5], where a single categorical feature contains tens of millions of video IDs, which leads to gigantic embedding tables. The ultra-large embedding tables also result in training efficiency problems. For instance, more than 48% of the operation kernel time is spent on embedding tables in a Meta production model (see Figure 2 for the breakdown). Similar observations are also reported in [2], showing that embedding table sizes have a significant impact on the training throughput. The memory and efficiency requirements motivate the distributed training solutions, where model-parallelism is exploited to partition and feed the embedding tables into multiple devices [1, 2, 9, 24, 25, 38]. The embedding lookup for a certain index will then be performed by querying the device that actually holds the corresponding table.

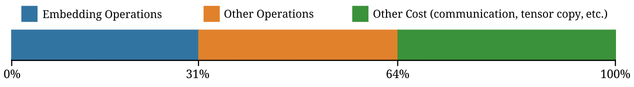


Figure 2: A breakdown of the computation time of one iteration of a Meta production model. Embedding operations account for 31% of the iteration time and 48% of the total operation execution time in GPU (i.e., 0.31 / 0.64 ≈ 48%).

While the model-parallelism enables training models with very large embedding sizes, it poses a significant design challenge named *embedding table sharding*, i.e., how we should partition the embedding tables across devices[1]. Figure 1 presents an illustrative example of why optimizing the sharding can significantly accelerate the training. If not carefully partitioned (left-hand side of Figure 1), the tables could lead to imbalances among GPUs, where all the GPUs are forced to wait for the slowest GPU. In contrast, a balanced sharding (right-hand side of Figure 1) can significantly accelerate the embedding operation by reducing the waiting time. Motivated by this, we investigate the following research question: *given an arbitrary set of embedding tables, how can we shard the embedding tables to balance the costs across devices?*

It is non-trivial to achieve this goal because of two major challenges. **First**, we need to efficiently estimate the cost (i.e., the kernel time of the embedding operators), which serves as the optimization objective. Unfortunately, the cost is hard to estimate. Unlike many other partition problem, the total cost of multiple tables in a shard is not the sum of the single table costs within the shard due to parallelism and operator fusion. As a result, it is hard to estimate the cost without actually running the operators; however, running the operators is computationally expensive. **Second**, we need an efficient algorithm to solve the partition problem, which is known to be NP-hard[2]. The ever-increasing number of embedding tables makes it infeasible to adopt a brute-force approach, i.e., iterating through all the possible sharding plans and outputting the best one. For practical use, the sharding algorithm is expected to propose an effective sharding plan for hundreds of tables in a reasonable time.

To address the above challenges, we present our novel practice in Meta, namely AutoShard, based on cost modeling and deep reinforcement learning (RL). To efficiently estimate the cost, we develop a neural cost model, which is trained with regression to the multi-table cost data collected from running micro-benchmarks on GPUs. To optimize the partitioning, we formulate the table sharding as a Markov decision process (MDP), where in each step, we allocate one table to a shard. The process ends when all the tables are allocated and we obtain a reward indicating the sharding quality at the final step. Then we leverage deep RL to learn an LSTM policy to optimize the sharding strategy. The cost model and the sharding policy are jointly trained towards convergence. In summary, we make the following contributions:

- Provide an in-depth analysis of the main influential factors of the cost of embedding operators via a case study on an modern embedding bag implementation from FBGEMM [16].

[1]In this work, we focus on embedding table sharding among GPU devices when the tables fit on the GPU memory. We note that system memory [12], non-volatile memory [11], and SSD [38] can also be used to store tables at the cost of decreased throughput [2]. We defer hybrid sharding strategies for these scenarios to future work.
[2]https://en.wikipedia.org/wiki/Partition_problem

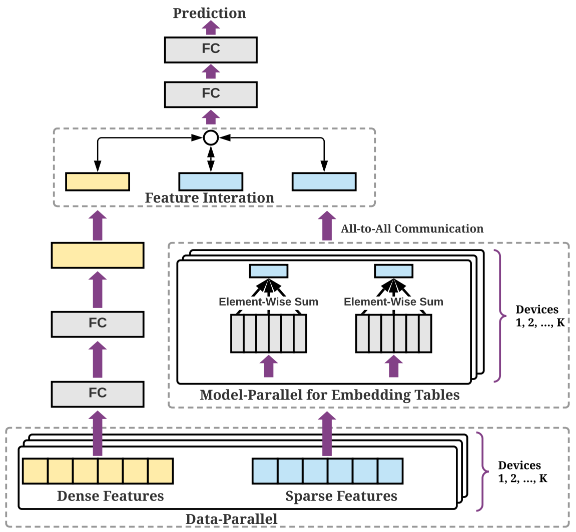


Figure 3: A typical recommendation model with dense and sparse features [25]. The system exploits a combination of model parallelism (i.e., the embedding tables are partitioned into different devices) and data parallelism (i.e., replicating MLPs on each device and partitioning training data into different devices). The embedding vectors obtained from embedding lookup are appropriately sliced and transferred to the target devices through an all-to-all communication.

- Propose AutoShard for embedding table sharding. It uses neural cost model to predict the kernel time of the operator and leverages deep RL to solve the partition problem. It can propose an effective sharding plan for hundreds of tables in seconds with a single CPU core.
- Conduct extensive experiments on an open-sourced large-scale synthetic dataset (for reproducibility) and Meta's production dataset. AutoShard significantly outperforms the heuristic sharding strategies. In particular, AutoShard can well transfer to various scenarios. The trained policy of AutoShard can be directly applied to solve a wide range of sharding tasks with various numbers of tables and different ratios of the unseen tables without any fine-tuning, achieving the same level of balance and speedup.

2 PRELIMINARIES

We start with a background of distributed recommender systems, and then formulate the embedding table sharding problem.

2.1 Training Deep Learning Recommendation Models at Scale

Industrial recommendation models usually require massive memory and high training throughput. Therefore, distributed training is employed in large-scale industrial systems [1, 2, 5, 8, 21, 40]. We take DLRM[3] [24] as an example to introduce the system design.

Figure 3 depicts an overview of DLRM. The system exhibits a combination of data parallelism and model parallelism. To exploit data parallelism, each trainer will hold a copy of MLP layers,

[3]https://github.com/facebookresearch/dlrm

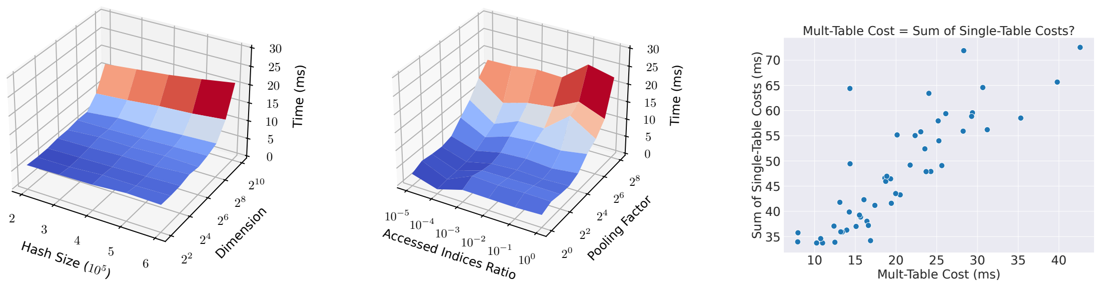


Figure 4: Impact of hash size and dimension on the single table cost (left); impact of indices distribution and pooling factor on the single table cost (middle); multi-table cost versus the sum of single table costs (right).

which will be trained on its own mini-batch of data. The model parameters are updated in the fully synchronous mode. For embedding tables, model parallelism is adopted to shard the tables into multiple devices. In the forward pass of embedding lookup, each device will perform a lookup for all the indices concerning the tables in the device (including the indices from the other devices' mini-batches). The obtained vectors will be transferred to the corresponding devices via an all-to-all communication. Then each device will receive all the embedding vectors for its own mini-batch, which will be interacted with the dense features, followed by an MLP layer for predictions. In the backward pass, the gradients will be similarly transferred with another all-to-all communication so that each device will receive all the gradients for the tables within the device. The embedding lookup and the dense computations are often executed in parallel. Therefore, the bottleneck depends on the model design, i.e., whether the dense cost or the embedding cost is more significant. In our production models, we observe that the embedding lookup latency often dominates the dense costs.

2.2 Problem Formulation

In the production environment, the possible embedding tables will often stay unchanged for a period of time since the raw features for a recommendation task are relatively steady. Nevertheless, different subsets of the tables could be used to build recommendation models. For example, a machine learning engineer may conduct feature selection by experimenting with different table combinations.

Following the above intuition, we formalize the embedding table sharding problem. Let $\mathcal{T}_p = \{T_1, T_2, ..., T_N\}$ be a pool of embedding tables, where N is the total number of tables in the pool. A sharding task S can be represented as a triple $S = (\mathcal{T}, \mathcal{D}, \mathcal{M})$, where $\mathcal{T} \subseteq \mathcal{T}_p$ is subset of the tables, $\mathcal{D} = \{1, 2, ..., K\}$ is a set of shard IDs with K shards in total, and $\mathcal{M} = \{M_1, M_2, ..., M_K\}$ is the memory constraints for all the shards. A sharding plan π can be represented as a mapping from each table to a shard. Then each device will get its own shard to process, which leads to a set of actual memory usages $\hat{\mathcal{M}} = \{\hat{M}_1, \hat{M}_2, ..., \hat{M}_K\}$ (which is obtained by summing the tables sizes in each shard) and a set of costs $C = \{C_1, C_2, ...C_K\}$ in terms of operation execution time. Embedding table sharding aims to optimize the sharding plan π such that the maximum cost across shards is minimized subject to the memory constraints:

$$\min_{\pi} \quad \max(C) := \max_{k} C_k \quad \text{s.t.} \quad \hat{M}_k \leq M_k, \forall k \in \mathcal{D}. \tag{1}$$

3 ANALYSIS OF EMBEDDING TABLE COST

This section analyzes the table costs on a modern embedding bag implementation from FBGEMM[4] [16] with a 2080Ti GPU.

3.1 Analysis of Single-Table Cost

The cost of a table is mainly determined by the characteristics of the table itself and indices lookup. Table characteristics include *hash size*, which is defined as the number of entries of the table, and *dimension*, which means the dimension of the embedding vectors. Characteristics of indices lookup include *pooling factor*, which is the average number of lookup indices per query, and *indices distribution*, which determines the indices accessing frequencies. We study the table characteristics and indices lookup on synthetic tables.

We first visualize the impact of table characteristics. We fix the pooling factor to be 32 and the indices to be uniformly distributed. Then we vary the hash size and table dimension to plot the kernel time in the left-hand side of Figure 4. As expected, we can see that a higher dimension will significantly increase the kernel time. This is because the dimension is positively correlated to the amount of data to be fetched. An interesting observation is that the hash size only has a moderate impact on the cost, which could be partly explained by the $O(1)$ time complexity for hash table lookup.

Similarly, we study the impact of indices lookup by fixing hash size to be 10^6 and dimension to be 32. For the indices distribution, some indices could be accessed far more frequently than others [2]. We simulate this behavior by restricting the indices access to a subset of the indices with a pre-defined accessed indices ratio. For example, a ratio of 1.0 is equivalent to the uniform distribution, while a ratio of 10^{-2} suggests only one percent of the indices will be accessed, which means the indices are sparsely distributed and those one percent of the indices are warm indices. The middle of Figure 4 shows the impact of the pooling factor and accessed indices ratio. A larger pooling factor will significantly increase the time, which is expected since it indicates more indices per lookup. In contrast, a sparse indices distribution tends to decrease the time. We speculate that this is caused by the caching mechanism.

The above analysis motivates a series of greedy sharding algorithms to balance different combinations of the above four factors, which will be elaborated in Section 5.1. However, the designed heuristics are sub-optimal since the actual running time has non-linear and complex relationships with these four factors.

[4]https://github.com/pytorch/FBGEMM/

3.2 Analysis of Multi-Table Cost

Since we usually have multiple tables in a shard, we naturally need to estimate the multi-table cost. A naive way to achieve this is to sum the costs of the single tables within the shard (single-sum for short). However, this is often inaccurate due to the parallelism of GPU. The right-hand side of Figure 4 plots the multi-table cost versus the single-sum of 50 randomly sampled table combinations from MetaSyn (see Section 5.1 for dataset details), where each sample contains 10 tables. **First**, we observe that the multi-table cost is significantly smaller than the single-sum. This is because the tables can be batched and accelerated with parallelism. **Second**, while the single-sum is positively correlated with the multi-table cost in general, it is still a poor estimator in many cases. The discrepancy between the multi-table cost and single-sum will amplify the difficulty of cost estimation since we need to consider all the possible table combinations, which grow exponentially with more tables.

4 METHODOLOGY

An overview of AutoShard is illustrated in Figure 5. Our framework on the highest level consists of four modules: *1) a micro-benchmark* that measures actual costs of embedding operators (Section 4.1), *2) a cost model* which approximates multi-table costs based on the data collected from micro-benchmark (Section 4.2), *3) an environment* that formulates the sharding process as a Markov Decision Process (MDP) by allocating one table in each step (Section 4.3), and *4) an RL policy* that optimizes the sharding strategies in a trial-and-error fashion (Section 4.4). Finally, we summarize the overall training procedure in Section 4.5.

4.1 Micro-Benchmarking Embedding Operators

This subsection introduces how we efficiently and precisely measure the latency of embedding tables. A naive way to achieve this is to train the recommendation models and profile the training with tools such as Pytorch Profiler[5]. Then we can parse the generated trace files to obtain the kernel time of embedding operators. However, this strategy is inefficient because a lot unnecessary time will be spent on data processing, and running irrelevant operators.

Since we only need the latency of embedding operators, we design and implement a micro-benchmark, which only benchmarks embedding operators alone. Due to space limitation, we introduce the main steps of micro-benchmarking here and provide more details in Appendix B.1: *1) initialization:* we initialize the operators with the specified arguments of embedding tables and load the indices data. *2) warmup*: we run the embedding operator several times to warm up the device. This step is important for enabling an accurate measurement because it allows the CUDA to complete the necessary preparation for the operator. *3) benchmarking*: run embedding operator several times and return the mean latency[6].

4.2 Modeling Multi-Table Costs

While micro-benchmark can accurately and efficiently measure the latency, it still needs to run the operators, which remains computationally expensive for production use. To further accelerate the cost

[5]https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html
[6]The micro-benchmark is currently maintained as a separate open-source effort for all the PyTorch operators at https://github.com/facebookresearch/param

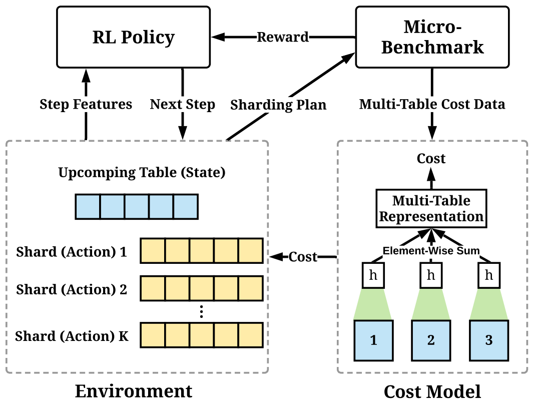


Figure 5: A high-level overview of AutoShard. The RL policy interacts with the environment and allocates one embedding table in each step. Based on the shards generated from the environment, the micro-benchmark measures the actual latencies of embedding table operators and produces a reward signal to update the RL policy. The cost model approximates multi-table costs based on the data collected from the micro-benchmark. The estimated costs will be used to enhance the action representations in RL training.

estimation, we develop a neural cost model to predict the operator cost based on the data collected from the micro-benchmark. The neural cost model can be easily deployed since it only requires a forward pass of a shallow neural network to obtain the cost.

Cost estimation can be formulated as a regression task, which takes as input the features of multiple tables and outputs the latency. The lower right corner of Figure 5 illustrates the neural architecture of the cost model. For each table, we use a shared MLP (green) to generate the single table representations based on the table features. Then we sum up the single table representations to obtain a multi-table representation (we have tried other reductions, such as max and mean, and found that sum works better), followed by another MLP to predict the cost. This design can flexibly accommodate different numbers of tables and obtain a final representation with a fixed dimension. We empirically use the following features: table dimension, hash size, pooling factor, table size, and indices distributions (17 features). We provide more details of these features in Appendix B.2. Formally, let $(\mathbf{X}, \mathbf{y})$ be the collected data. $\mathbf{X}$ is the table features where each row represents the features of multiple tables and has variable lengths; $\mathbf{y}$ denotes a vector of ground-truth costs obtained by running micro-benchmark on GPUs. We train the cost model f with mean squared error (MSE) loss $L_{\text{cost}} = (\mathbf{y} - f(\mathbf{X}))^2$. The performance of the cost model is reported in Table 3 of Appendix.

4.3 Formulating Sharding as MDP

This subsection describes why and how we formulate the sharding procedure as a sequential decision process. A naive strategy to solve the sharding problem is to treat it as a black-box optimization problem, where we sample and evaluate a sharding plan in each iteration. However, this will lead to an extremely large search space since each table can be possibly assigned to any shard.

To tackle this problem, we decompose the generation of a sharding plan into multiple steps, where we only assign one table to

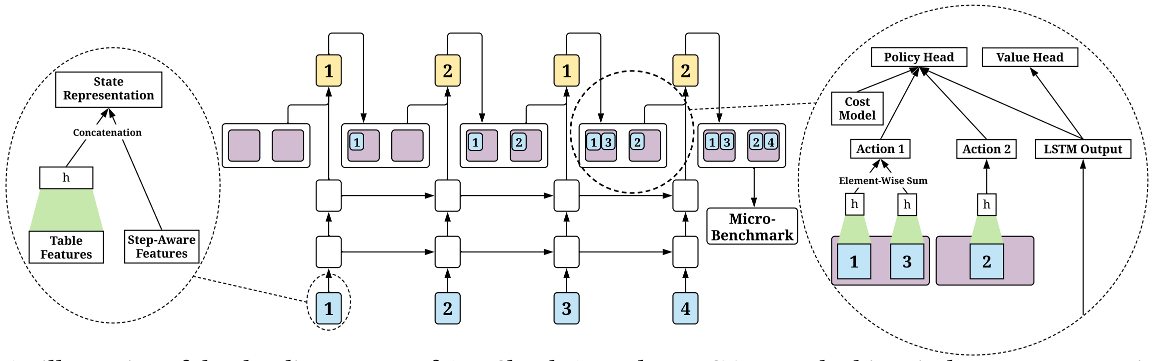


Figure 6: An illustration of the sharding process of AutoShard. A two-layer LSTM encodes historical state representations and in each step outputs a shard ID (yellow) that allocates the table at hand (blue) to a shard (purple). State representation is obtained by concatenating the table representation and step-aware features (dotted circle in the left). Action representation is the sum of the single tables' representations (dotted circle in the right). A policy head produces a shard ID based on the state/action representations and the cost model. A value head will approximate state values to reduce the variance of RL training.

a shard in each step. Then after scanning all the tables, we can eventually obtain a sharding plan. This formulation has two desirable properties. **First,** it can significantly decrease the decision space. Specifically, the decision space of the sharding policy is only the number of shards K in each step. Although the policy needs to perform more steps, the decisions made across steps are very similar (i.e., they all aim to assign a table to achieve load balance) so that the policy in one step may learn to reuse the knowledge learned from other steps and improve learning efficiency. **Second,** it can implicitly encourage transferable strategies. By associating one table with one step, a model trained on very few tables can easily transfer to more tables by simply adding more steps without re-training. This cannot be achieved by black-box optimization.

We formulate the above process as an MDP with the state, action, and reward defined as follows. **State:** The features of the upcoming table and a step-aware feature which is the ratio of the remaining tables to be assigned. **Action:** The shard IDs with K actions in total. The multi-table cost features and the predicted costs from the cost model can serve as the action features. **Reward**: The agent will receive zero rewards for all the intermediate steps and a final reward indicating the quality of the sharding plan. Specifically, if the sharding plan meets the memory constraint, we run the micro-benchmark to obtain the shard latencies of all the shards. The reward is calculated by the ratio between the maximum latency and minimum latency, i.e., $min(C)/max(C)$, to encourage the agent to balance the costs across shards. The reward is in the range of $[0, 1]$, where a higher reward suggests a better balance. Alternatively, if the sharding plan cannot meet the memory constraints, we penalize this behavior with a negative reward, which is determined by the shard that is the most seriously affected by the memory explosion, i.e., $\max_k((\hat{M}_k - M_k)/M_k)$. A higher reward indicates that the sharding plan is closer to meeting the memory constraints.

4.4 Optimizing Sharding Strategy with RL

This subsection presents how we solve the above MDP with RL. We will first elaborate on the neural architecture design and then describe how we train the model weights.

Figure 6 illustrates the neural architecture and how it makes predictions to shard tables. The model takes as input the state features and actions features, and outputs an action probability vector where

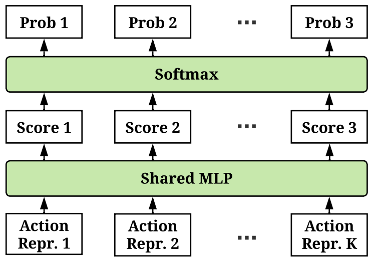


Figure 7: An illustration of the policy head.

each action corresponds to a shard ID (policy head) and a scalar value indicating the value of the state (value head). The model is instantiated with a two-layer LSTM to process the state and actions with the following procedure. **1)** The state representation is obtained by combining the table features and step-aware features (dotted circle in the left). **2)** The state representations will be fed into the LSTM sequentially so that historical state information will be encoded into LSTM output as well. **3)** The multi-table representations obtained in the cost model will be concatenated with the predicted costs from the cost model to construct the action representations (dotted circle in the right). **4)** Each action representation will be concatenated with the state representation, followed by an MLP (which is shared for all actions) to produce a confidence score for the action, shown in Figure 7. **5)** The scores for all the actions will be processed by a Softmax layer to obtain the probability vector, where the probabilities sum to one. Similarly, the LSTM output will be fed into another MLP to generate a state value in the value head. For an upcoming table, the model will sample a shard ID (action) based on the probabilities from the policy head. The tables will be allocated to the shards one by one following this procedure.

We train the model with IMPALA [7], a distributed actor-critic algorithm enhanced by V-trace targets to achieve off-policy learning. The V-trace correction can tackle delayed model weights update, which is helpful in our problem in that evaluating a sharding plan is slow and may result in substantial delays. Here, we only briefly introduce IMPALA since RL itself is not our focus; one can adopt other RL algorithms as well under our framework. We first introduce the V-trace targets and then describe the loss for updating the policy and value heads. Let s_t, a_t, and r_t be the state, action, and reward at step t, respectively. We consider an n-step trajectory $(s_t, a_t, r_t)_{t=t'}^{t=t'+n}$. The V-trace target for $s_{t'}$ is defined as

$$V_{\text{target}}(s_{t'}) = V(s_{t'}) + \sum_{t=t'}^{t'+n-1} \gamma^{t-t'} (\pi_{i=t'}^{t-1} c_i) \delta_t V, \tag{2}$$

where $V(s_{t'})$ is the output of the value head for $s_{t'}$, $\delta_t V = \rho_t (r_t + \gamma V(s_{t+1}) - V(x_t))$ is the temporal difference, and c_i and ρ_t are truncated importance sampling weights that tackle the delayed update of the model. Then the loss at step t is defined as

$$L_t = \rho_t \log \pi(a_t|s_t)(r_t + \gamma V_{\text{target}}(s_{t+1} - V(s_t)) + \frac{1}{2}(v_t - V(s_t))^2, \tag{3}$$

where $\pi(a_t|s_t)$ and $V(s_t)$ correspond to policy and value heads, respectively. The training can be batched to update the losses for multiple steps at a time in each iteration.

4.5 Training of AutoShard

This subsection summarizes the overall training procedure. To improve the sample efficiency, the cost model and the LSTM policy-value network are jointly trained with shared table representations and data. Specifically, the MLP for processing the tables features (i.e., the green parts in Figure 5 and Figure 6) is shared. Similarly, the data collected by the RL agent will be reused to generate cost data to train the cost model. The whole training process is summarized in Algorithm 1. In each training iteration, we collect a batch of trajectories by interacting with the micro-benchmark to update the policy-value network (line 4). The collected data will be stored in a buffer and reused to train the cost model (line 8). Since the main bottleneck is data collection, we parallelize line 4 with multiple processes operating on different GPUs. Once the cost model and the policy-value network are trained, they can be directly applied to any new sharding tasks by sequentially predicting the shard IDs.

Algorithm 1 Training of AutoShard

```
1:  Input: Training tasks $\mathcal{S}_{\text{train}} = \{S_i\}_{i=1}^n$, batch size $B_1$ for policy-value network and $B_2$ for cost model, number of update iterations $I$ for cost model, number of data collection steps $T$
2:  Initialize the cost model and policy-value network
3:  for iteration = 1, 2, ... until convergence do
4:      Collect a set of trajectories with $T$ steps $\{s_t, a_t, r_t\}_{t=1}^T$ from a randomly sampled task from $\mathcal{S}_{\text{train}}$ and store the generated cost data into a buffer
5:      if more than $B_1$ sets of new trajectories are collected then
6:          Update policy-value network with Eq. 3
7:          for iteration = 1, 2, ..., $I$ do
8:              Sample a batch of cost data with size $B_2$ from the buffer and update the cost model with MSE loss
9:          end for
10:     end if
11: end for
```

5 EXPERIMENTS

The experiments are conducted on both synthetic datasets and production datasets at Meta. We aim to answer the following questions: **Q1:** How does AutoShard compare with the heuristic sharding strategies (Section 5.2)? **Q2:** How large is the search space of AutoShard and can simple random search be competitive with it (Section 5.3)? **Q3:** How does each component of AutoShard contribute to the performance (Section 5.4)? **Q4:** Can AutoShard transfer to unseen tables, and sharding tasks with more tables (Section 5.5)? **Q5:** How efficient is the training/inference of AutoShard (Section 5.6)?

5.1 Experimental Settings

Datasets. The public recommendation datasets often cannot match the scale of the industrial models to enable a valid evaluation of embedding table sharding. For example, Criteo[7], one of the most popular datasets, only has 26 sparse features with a cardinality of at most one million. Thus, our experiments are mainly conducted on an open-sourced large-scale synthetic dataset[8] (**MetaSyn**), which shares similar indices accessing patterns to Meta production embedding tables. As tabulated in Table 1, MetaSyn consists of hundreds of embedding tables with very large and diverse hash sizes and pooling factors. Since MetaSyn does not specify table dimensions, we randomly select a dimension for each table from $\{16, 32\}$ (we purposely make the dimensions small so that our results can be reproduced on GPUs with 10 GB memory). For verification, we also conduct experiments on the Meta production embedding tables (**MetaProd**), which have a similar scale as MetaSyn except for larger table dimensions. We keep the details of MetaProd confidential. MetaSyn and MetaProd will serve as the table pools, where each sharding task is constructed by a randomly sampled subset of the tables. Intuitively, with more tables, we also need more devices so that the tables can fit on the GPU memory. We empirically set the number of devices to be 1/10 of the total number of tables for all the tasks since this setting can ensure sufficient memory for all the sharding algorithms on both MetaSyn and MetaProd; that is, we use 8 devices for 80 tables, and 16 devices for 160 tables, etc. We provide more details in Appendix A.

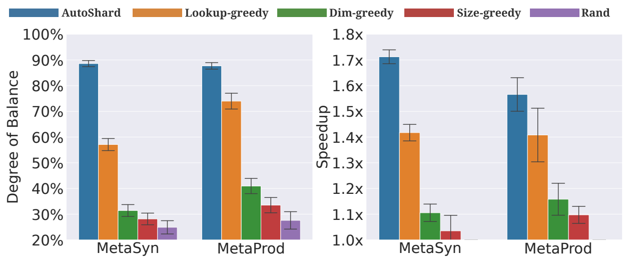
Heuristic Algorithms. We compare AutoShard against several deployed heuristic sharding algorithms. They mainly consist of two steps: *(1) cost function:* each table will be assigned an estimated cost, and *(2) greedy algorithm:* the tables are first sorted in descending order based on the costs. Starting from the table with the highest cost, greedy algorithm will assign tables one-by-one to the device with the lowest sum of the costs so far, so that each device will have roughly an equal sum of the costs in the end. We consider the heuristics with the following cost functions, which have been proven to show strong performance in prior work [26]: the size (the product of dimension and hash size) of the table (**size-greedy**), the dimension of the table (**dim-greedy**), the product of the dimension and mean pooling factor of the table (**lookup-greedy**). We further include a random sharding baseline (**rand**).

Metrics. We evaluate the performance with the following metrics. **Degree of balance:** the ratio between the minimum latency and the maximum latency across shards. 100% suggests perfect balancing where each shard has equal latency, and 0% indicates the worst-case of load balance. **Speedup:** the speedup over random sharding which is the most naive strategy. Specifically, the speedup is calculated by $max(C^{random})/max(C)$, where C^{random} and C

[7]https://www.kaggle.com/c/avazu-ctr-prediction/data
[8]https://github.com/facebookresearch/dlrm_datasets

Table 1: Statistics of Meta synthetic embedding tables.

Attribute	Value
Number of Tables	856
Batch Size	65,536
Max/Mean/Min Hash Sizes	12,543,670 / 4,107,458 / 1
Max/Mean/Min Pooling Factors	193 / 15 / 0



(a) Degree of balance (b) Speedup

Figure 8: Performance of AutoShard against baselines. We report the mean and standard deviation across five runs.

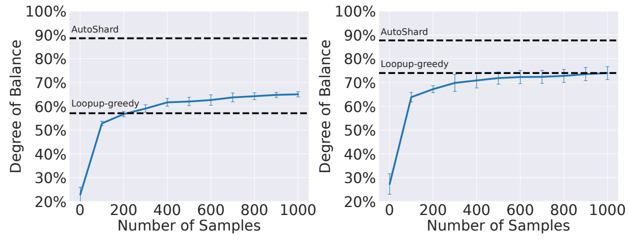
are the sets of actual embedding table costs of random sharding strategy and the sharding algorithm at hand, respectively.

Implementation Details. All the hyperparameters are tuned based on MetaSyn and the same set of hyperparameters are used on MetaProd. Specifically, we set $B_1 = 8$, $B_2 = 512$, $I = 20$, and $T = 100$. We use the IMPALA implementation in [17] with the default hyperparameters for RL training. We use 2080 Ti and V100 GPUs for MetaSyn and MetaProd, respectively. We run all the experiments five times with random seeds 0, 1, 2, 3, and 4 and report the means and standard deviations. We provide more details in Appendix C.

5.2 Comparison with the Heuristics

To study **Q1**, we conduct experiments on the tasks of sharding 80 tables to 8 devices. Specifically, we randomly sample 90 training tasks, where each task consists of 80 randomly sampled tables from the pool. Then we sample another 10 different tasks with the same procedure for the testing purpose. AutoShard is trained on the 90 training tasks. We collect the mean result on the same 10 testing tasks for all the algorithms. Note that we have purposely separated the training and testing tasks to test whether AutoShard can generalize to different table combinations from the pool.

We summarize the results in Figure 8. We make the following observations. **First**, all the sharding algorithms outperform the random sharding, which is expected since random sharding may easily result in imbalances. **Second**, AutoShard performs significantly and consistently better than the heuristics on both synthetic and production data for both metrics, which demonstrates the effectiveness of AutoShard. **Third**, look-greedy appears to be the strongest heuristic algorithm. This is expected because it considers both table dimensions and pooling factors, which can essentially quantify the workload for the indices lookup. Nevertheless, there is still a clear gap between lookup-greedy and AutoShard. This is because AutoShard can achieve a more accurate estimation of the cost by considering indices distributions and multi-table costs, and leveraging RL to optimize the sharding process.



(a) MetaSyn (b) MetaProd

Figure 9: Performance of random search with five runs. Note that searching 1000 samples is extremely time-consuming and are often impractical for production use (it takes 9,783 seconds for MetaSyn and 14,599 seconds for MetaProd).

Table 2: Ablation study of AutoShard on MetaSyn.

	Degree of Balance	Speedup
w/o cost modeling	61.3%±12.9%	1.420±0.203
w/o dimension feature	87.6%±1.5%	1.706±0.056
w/o hash size feature	87.8%±0.9%	1.702±0.047
w/o pooling factor feature	45.9%±1.9%	1.271±0.035
w/o size feature	87.0%±1.1%	1.683±0.049
w/o distribution features	84.3%±1.3%	1.688±0.035
Full version of AutoShard	**88.6%±1.2%**	**1.712±0.027**

5.3 Comparison with Random Search

To answer **Q2**, we implement a random search algorithm to understand the difficulty of identifying a strong sharding plan in the search space. We choose random search because it is shown to be a strong baseline in neural architecture search when the search space is restricted [19]. Specifically, we treat the tables as decisions, whose possible choices are the device IDs, and use random search to optimize the degree of balance. We follow the setting in Section 5.2, which results in an extremely large search space[9] of $8^{80} \approx 1.76 \times 10^{72}$. Note that search is infeasible in production because it requires lots of GPU resources. This experiment is designed solely for understanding of the massive search space of AutoShard.

Figure 9 plots the performance of random search w.r.t. the number of samples. Although random search can achieve competitive performance with lookup-greedy after hours of searching, it is far behind the AutoShard, which verifies the difficulty of embedding table sharding. In contrast, AutoShard shows clear advantages in terms of both effectiveness and efficiency. It can achieve strong performance with only a forward pass without the search.

5.4 Ablation Studies

For **Q3**, we consider several ablations: 1) we remove the cost model and only use raw features to train RL, 2) instead of sharding with RL, we greedily assign tables like the heuristics with the only difference that we use the cost model to estimate the cost, and 3) we remove each of the table features to study the feature importance.

Table 2 summarizes the results. **First**, we observe a significant performance drop when removing the cost model, which verifies the necessity of cost modeling. **Second**, the pooling factor feature is a very important feature, which is expected since pooling factor can indicate the number of lookup indices. **Finally**, removing either

[9]The search space grows exponentially with more tables. For the most difficult problem in Section 5.5, i.e., 800 tables and 80 devices, the search space is $80^{800} \approx 2.96 \times 10^{1523}$.

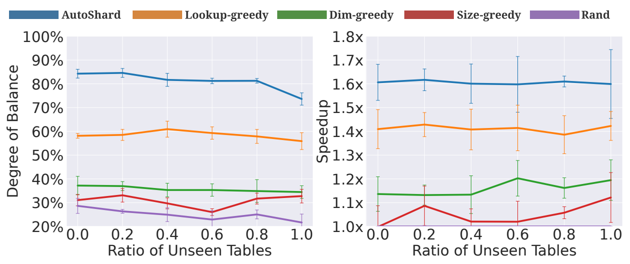


Figure 10: Degree of balance (left) and speedup (right) of AutoShard on MetaSyn with different ratios of unseen tables. Note that when the ratio is 0.0, the results are worse than those shown in Figure 8. This is because we only use half of the tables in this experiment. A possible reason is that more tables can improve the generalization ability of AutoShard, which leads to better results on the testing tasks.

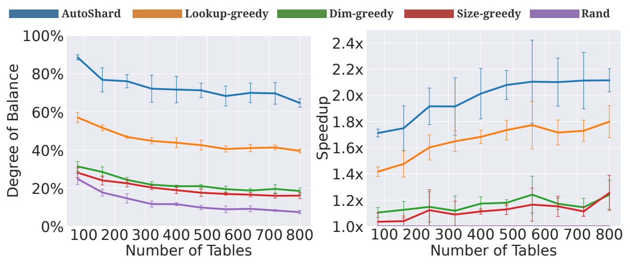


Figure 11: Degree of balance (left) and speedup (right) of AutoShard on MetaSyn with up to 800 tables. We directly transfer the model trained on 80 tables without fine-tuning.

of the features will degrade the performance, which suggests the designed features are complimentary.

5.5 Analysis of Transferability

To investigate **Q4**, we evaluate the transferability of AutoShard on unseen tables and sharding tasks with more tables.

To test AutoShard on unseen tables, we split the original table pools in half, where the first sub-pool is used to train AutoShard, and the tables in the second sub-pool are unseen in training. Then we randomly mix the tables from the two sub-pools to construct the sharding tasks based on a specified ratio of unseen tables. A ratio of 0.0 suggests that all the tables are from the first sub-pool (i.e., all the tables in the sharding tasks are seen in training), while a ratio of 1.0 means all the tables are from the second sub-pool (i.e., all the tables are unseen). A high ratio will make the transferability task more challenging. We report the performance w.r.t. different ratios of unseen tables in Figure 10. **First**, AutoShard only shows a moderate performance decrease when a part of the tables are unseen. Specifically, when the ratio is between 0.0 to 0.8, AutoShard can achieve at least 80% degree of balance and 1.6X speedup. **Second**, when all the tables are unseen (i.e., the ratio is 1.0), AutoShard can still achieve more than 70% degree of balance and around 1.6X speedup, which significantly outperform the baselines.

To test whether AutoShard can scale to hundreds of tables, we compare AutoShard with baselines on tasks that shard up to 800 tables to 80 GPU devices. Note that in this experiment we have not trained any new models but instead directly apply the model

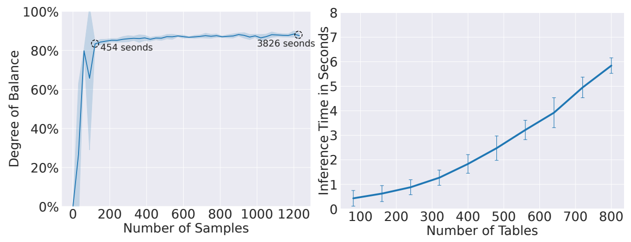


Figure 12: Training curve on four 2080 Ti GPUs (left) and inference time with a single CPU core (right).

trained on 80 tables. We plot the results in Figure 11 and make the following observations. **First**, the degree of balance decreases for all the sharding algorithms. This is expected because the task becomes more challenging with more tables. **Second**, AutoShard can significantly outperform the baselines in all the settings. In particular, we surprisingly observe an increase in speedup with more tables. This is because random sharding will perform poorly with more tables. This again demonstrates the superiority of AutoShard.

Overall, we conclude that AutoShard can well transfer to unseen tables and hundreds of tables, making it a desirable choice in handling complex training tasks in the production environment.

5.6 Analysis of Training/Inference Efficiency

We analyze the training and inference time to answer **Q5**. The left-hand side of Figure 12 plots the training curve of AutoShard on four GPUs. We observe that AutoShard can achieve more than 80% degree of balance within around 150 samples or 454 seconds. This is highly efficient for production use since we only need to train AutoShard offline periodically (e.g. we can run a daily or weekly training job). The right-hand side of Figure 12 shows the inference time with a single CPU core. AutoShard can shard hundreds of tables in seconds. This cost is neglectable in production use.

6 RELATED WORK

Deep recommendation models. Deep-learning-based recommendation models have shown superior performance in many recommendation scenarios [4, 10, 25, 37]. Due to the ultra-large-scale of the data and the features in industrial applications, distributed training solutions have been developed to improve the training efficiency [2, 5, 8, 21, 40]. Despite these efforts, embedding table sharding remains to be a critical challenge for distributed training. AutoShard is the first learning-based sharding algorithm that can optimize the sharding strategy in an end-to-end fashion.

Tackling large embedding tables. How to deal with the ultra-large embedding tables of recommendation models has been a long-standing challenge. One line of work aims to reduce the embedding table size, such as sharing the embeddings across related features [27, 36], searching the vocabulary sizes or the table dimensions [13, 39], pruning [22], quantilization [14], and hashing [15]. Our work is orthogonal to these methods since AutoShard can be applied to compressed tables as well. A related work explored using the tiered memory hierarchy to store the embedding tables [26]. They exploited the unequal access patterns of embedding tables to improve the efficiency by placing hot rows in the GPU memory. Our efforts complement [26] by providing an end-to-end learning-based framework for cost approximation and partitioning optimization.

Deep RL. Deep RL has shown promise in accomplishing goal-oriented tasks [23, 28, 30, 33, 35]. Recently, deep RL has been applied to various machine learning model design tasks, such as neural architecture search [41], pipeline search [18, 20, 34], data augmentation/sampling [6, 31, 32]. Our work also falls into this line of studies but we instead focus on optimizing the model efficiency. Our work is also related to applying RL for classical combinational optimization [3]. Unlike [3], we tackle a real-world combinational optimization challenge in industrial recommender systems.

7 CONCLUSIONS AND FUTURE WORK

This work presents a novel solution for embedding table sharding practiced at Meta. The proposed algorithm, namely AutoShard, uses a cost model to efficiently estimate the table cost and leverages deep RL to solve the partition problem. The empirical results suggest that AutoShard is effective, transferable, and efficient. Through developing AutoShard, we show the promise in applying RL to optimize the industrial-level system designs. We have open-sourced a prototype of AutoShard to motivate and facilitate future exploration in this direction. In the future, we will extend AutoShard to tackle more complex sharding tasks by modeling the cost of the communication and the tiered memory hierarchy.

REFERENCES

[1] [n.d.]. Amazon DSSTNE: Deep Scalable Sparse Tensor Network Engine. https://github.com/amazon-archives/amazon-dsstne.

[2] Bilge Acun, Matthew Murphy, Xiaodong Wang, Jade Nie, Carole-Jean Wu, and Kim Hazelwood. 2021. Understanding training efficiency of deep learning recommendation models at scale. In *HPCA*.

[3] Thomas Barrett, William Clements, Jakob Foerster, and Alex Lvovsky. 2020. Exploratory combinatorial optimization with reinforcement learning. In *AAAI*.

[4] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. 2016. Wide & deep learning for recommender systems. In *DLRS Workshop*.

[5] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *RecSys*.

[6] Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. 2018. Autoaugment: Learning augmentation policies from data. *arXiv preprint arXiv:1805.09501* (2018).

[7] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. 2018. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *ICML*.

[8] Carlos A Gomez-Uribe and Neil Hunt. 2015. The netflix recommender system: Algorithms, business value, and innovation. *ACM Transactions on Management Information Systems (TMIS)* 6, 4 (2015), 1–19.

[9] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. 2020. The architectural implications of facebook's dnn-based personalized recommendation. In *HPCA*.

[10] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural collaborative filtering. In *WWW*.

[11] Doo Seok Jeong and Cheol Seong Hwang. 2018. Nonvolatile memory materials for neuromorphic intelligent machines. *Advanced Materials* 30, 42 (2018), 1704729.

[12] Biye Jiang, Chao Deng, Huimin Yi, Zelin Hu, Guorui Zhou, Yang Zheng, Sui Huang, Xinyang Guo, Dongyue Wang, Yue Song, et al. 2019. XDL: an industrial deep learning framework for high-dimensional sparse data. In *KDD Workshop*.

[13] Manas R Joglekar, Cong Li, Mei Chen, Taibai Xu, Xiaoming Wang, Jay K Adams, Pranav Khaitan, Jiahui Liu, and Quoc V Le. 2020. Neural input search for large scale recommendation models. In *KDD*.

[14] Wang-Cheng Kang, Derek Zhiyuan Cheng, Ting Chen, Xinyang Yi, Dong Lin, Lichan Hong, and Ed H Chi. 2020. Learning multi-granular quantized embeddings for large-vocab categorical features in recommender systems. In *WWW*.

[15] Wang-Cheng Kang, Derek Zhiyuan Cheng, Tiansheng Yao, Xinyang Yi, Ting Chen, Lichan Hong, and Ed H Chi. 2021. Learning to Embed Categorical Features without Embedding Tables for Recommendation. In *KDD*.

[16] Daya Khudia, Jianyu Huang, Protonu Basu, Summer Deng, Haixin Liu, Jongsoo Park, and Mikhail Smelyanskiy. 2021. FBGEMM: Enabling High-Performance Low-Precision Deep Learning Inference. *arXiv preprint arXiv:2101.05615* (2021).

[17] Heinrich Küttler, Nantas Nardelli, Thibaut Lavril, Marco Selvatici, Viswanath Sivakumar, Tim Rocktäschel, and Edward Grefenstette. 2019. Torchbeast: A pytorch platform for distributed rl. *arXiv preprint arXiv:1910.03552* (2019).

[18] Kwei-Herng Lai, Daochen Zha, Guanchu Wang, Junjie Xu, Yue Zhao, Devesh Kumar, Yile Chen, Purav Zumkhawaka, Minyang Wan, Diego Martinez, et al. 2021. TODS: An Automated Time Series Outlier Detection System. In *AAAI*.

[19] Liam Li and Ameet Talwalkar. 2020. Random search and reproducibility for neural architecture search. In *UAI*.

[20] Yuening Li, Zhengzhang Chen, Daochen Zha, Kaixiong Zhou, Haifeng Jin, Haifeng Chen, and Xia Hu. 2021. Automated Anomaly Detection via Curiosity-Guided Search and Self-Imitation Learning. *IEEE Transactions on Neural Networks and Learning Systems* (2021).

[21] David C Liu, Stephanie Rogers, Raymond Shiau, Dmitry Kislyuk, Kevin C Ma, Zhigang Zhong, Jenny Liu, and Yushi Jing. 2017. Related pins at pinterest: The evolution of a real-world recommender system. In *WWW*.

[22] Siyi Liu, Chen Gao, Yihong Chen, Depeng Jin, and Yong Li. 2021. Learnable Embedding sizes for Recommender Systems. In *ICLR*.

[23] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

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

[25] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. 2019. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091* (2019).

[26] Geet Sethi, Bilge Acun, Niket Agarwal, Christos Kozyrakis, Caroline Trippel, and Carole-Jean Wu. 2022. RecShard: Statistical Feature-Based Memory Optimization for Industry-Scale Neural Recommendation. *arXiv preprint arXiv:2201.10095* (2022).

[27] Hao-Jun Michael Shi, Dheevatsa Mudigere, Maxim Naumov, and Jiyan Yang. 2020. Compositional embeddings using complementary partitions for memory-efficient recommendation systems. In *KDD*.

[28] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of go without human knowledge. *nature* (2017).

[29] Weiping Song, Chence Shi, Zhiping Xiao, Zhijian Duan, Yewen Xu, Ming Zhang, and Jian Tang. 2019. Autoint: Automatic feature interaction learning via self-attentive neural networks. In *CIKM*.

[30] Daochen Zha, Kwei-Herng Lai, Songyi Huang, Yuanpu Cao, Keerthana Reddy, Juan Vargas, Alex Nguyen, Ruzhe Wei, Junyu Guo, and Xia Hu. 2021. RLCard: a platform for reinforcement learning in card games. In *IJCAI*.

[31] Daochen Zha, Kwei-Herng Lai, Mingyang Wan, and Xia Hu. 2020. Meta-AAD: Active anomaly detection with deep reinforcement learning. In *ICDM*.

[32] Daochen Zha, Kwei-Herng Lai, Kaixiong Zhou, and Xia Hu. 2019. Experience Replay Optimization. In *IJCAI*.

[33] Daochen Zha, Wenye Ma, Lei Yuan, Xia Hu, and Ji Liu. 2021. Rank the Episodes: A Simple Approach for Exploration in Procedurally-Generated Environments. In *ICLR*.

[34] Daochen Zha, Zaid Pervaiz Bhat, Yi-Wei Chen, Yicheng Wang, Sirui Ding, Anmoll Kumar Jain, Mohammad Qazim Bhat, Kwei-Herng Lai, Jiaben Chen, Na Zou, et al. 2022. AutoVideo: An Automated Video Action Recognition System. In *IJCAI*.

[35] Daochen Zha, Jingru Xie, Wenye Ma, Sheng Zhang, Xiangru Lian, Xia Hu, and Ji Liu. 2021. DouZero: Mastering DouDizhu with Self-Play Deep Reinforcement Learning. In *ICML*.

[36] Caojin Zhang, Yicun Liu, Yuanpu Xie, Sofia Ira Ktena, Alykhan Tejani, Akshay Gupta, Pranay Kumar Myana, Deepak Dilipkumar, Suvadip Paul, Ikuhiro Ihara, et al. 2020. Model size reduction using frequency based double hashing for recommender systems. In *RecSys*.

[37] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. 2019. Deep learning based recommender system: A survey and new perspectives. *ACM Computing Surveys (CSUR)* 52, 1 (2019), 1–38.

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

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

[40] Guorui Zhou, Na Mou, Ying Fan, Qi Pi, Weijie Bian, Chang Zhou, Xiaoqiang Zhu, and Kun Gai. 2019. Deep interest evolution network for click-through rate prediction. In *AAAI*.

[41] Barret Zoph and Quoc V Le. 2017. Neural architecture search with reinforcement learning. In *ICLR*.

A DATASET DETAILS

We will not provide the details of MetaProd for data privacy and only discuss MetaSyn, which is open-sourced[10] and shares similar patterns to Meta production recommendation workloads.

A.1 Data Format

MetaSyn consists of three PyTorch tensors saved in a single file, including an indices tensor, an offsets tensor, and a length tensor. We denote them as `indices`, `offsets`, and `lengths`, respectively. `indices` is 1-dimensional, where each element is an integer index. The indices are ordered by `(table_id, batch_offset)`; that is, if we scan the tensor from the left to the right, we will first get a batch of indices for the first table, and then obtain a batch of indices for the second table, etc. `offsets` is also 1-dimensional and specifies the starting position and the ending position for one lookup. It is also ordered by `(table_id, batch_offset)`. For example, suppose the batch size is 65,536. Then `start = offsets[65536]` and `end = offsets[65537]` will specify the starting and ending positions of the first indices lookup in the second table. The indices tensor between the starting and ending positions, i.e., `indices[start:end]` correspond to the first instance in the batch and the second table. `lengths` is 2-dimensional and is of the shape `[num_tables, batch_size]`, where each element is the pooling factor of the corresponding indices lookup. `lengths` is provided for correctness validation since it can be inferred from the other two tensors. `indices` and `offsets` share the same data format with the batched embedding bag operator from FBGEMM and can be directly fed into it.

A.2 Distributions

We visualize the distribution of hash sizes, the distribution of mean pooling factors, the relation between the hash sizes and pooling factors, and the distribution of indices accessing frequency of MetaSyn in Figure 13. We make the following observations: 1) the hash sizes for most tables are on the scale of millions, while some can reach tens of millions. 2) the pooling factor generally follows a power-law distribution, where the majority is less than 50, while some can be as large as 200. 3) there is no clear relationship between the hash size and pooling factor; 4) most of the indices are accessed less than ten times, while some of them can reach 10^5. The highly diverse table characteristics and indices patterns will easily result in imbalances if not carefully partitioning the tables. Specifically, the costs of the tables will also be diverse, i.e., some tables will have extremely high costs while some others could have very low costs. As a result, if we do not shard the tables carefully, some tables with high costs can be easily put into the same shard, resulting in a very high cost for the shard.

A.3 Data Processing

Recall that the 856 tables in MetaSyn will serve as the table pool in our experiments. For the ease of use, we separate `indices` and `offsets` into a list of `indices` and a list of `offsets`, respectively, where each element corresponds to one table. The offsets for each table will be re-indexed starting from 0. In training, we merge indices and the offsets of the selected tables to construct the input of the operator. For the embedding tables, we randomly choose the table dimension from $\{16, 32\}$ since the MetaSyn does not provide this information. We purposely make the dimensions small to facilitate reproducibility on GPUs with small memory. Note that, for MetaProd, the table dimension is specified by the production model and is larger than MetaSyn. For the parameters of embedding tables, we randomly initialize them with fp16 precision.

[10]https://github.com/facebookresearch/dlrm_datasets

Algorithm 2 `benchmark_op`

```
1: Input: A PyTorch operator, the arguments of the operator
2: Feed random tensors to GPU to clear the cache
3: Initialize the start and end CUDA events for time measuring
4: Run the forward and backward passes of the operator
5: Collect latency from the CUDA events
6: Return the latency
```

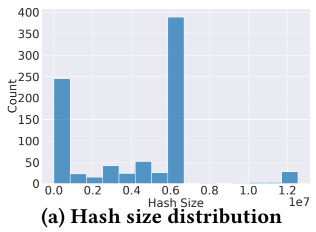
Algorithm 3 Micro-Benchmark

```
1: Input: A PyTorch operator, the arguments of the operator, the
   number of warmup iterations $W$, the number of measuring
   iterations $B$, the number of removed highest/lowest costs $R$
2: for iteration = 1, 2, ... W do
3:     Call benchmark_op with the operator and the arguments
4: end for
5: Initialize a Python list costs to store the costs
6: for iteration = 1, 2, ... B do
7:     Call benchmark_op and append the result to costs
8: end for
9: Sort the costs and remove the $R$ highest/lowest costs.
10: Return the mean of the costs
```

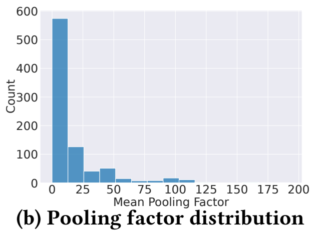
B MODEL DETAILS

B.1 Details of Micro-Benchmark

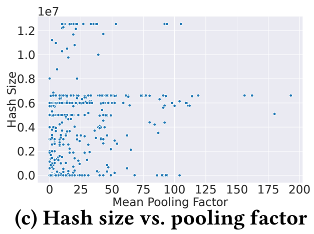
Precisely measuring the kernel time of the embedding operator requires non-trivial engineering efforts because of the warmup overhead of the GPU, caching mechanism, and variance. Specifically, naively running an operator multiple times and using the mean latency as the cost will not result in an accurate estimation. First, the warmup stage of the CUDA devices will cause significant overhead, leading to a higher estimation of the cost. Second, the L1 and L2 caches will cache the tensors in the previous iterations and result in a lower estimation of the cost. Third, some certain anomalous runs could have very high or low latency due to variance. However, the mean latency will be sensitive to anomalies. Fortunately, we found that some engineering tricks can well tackle the above problems to enable a precise measurement of the latency. The overall procedure is summarized in Algorithm 3 with an inner function in Algorithm 2. Before actually measuring the time, we first run several warmup iterations to warm up the GPU (line 2-3 of Algorithm 3). For each run, we first clear the cache to remove the impact of caching (line 2 of Algorithm 2). To tackle the anomalies, we remove the highest and the lowest costs and return the mean value of the remaining costs. We empirically set $W = 5$, $B = 10$, and $R = 2$. We have performed sanity checks and concluded that the time collected from



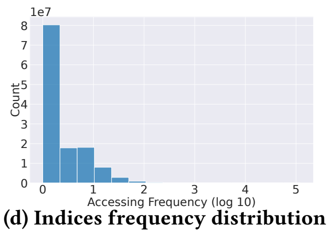
(a) Hash size distribution



(b) Pooling factor distribution



(c) Hash size vs. pooling factor



(d) Indices frequency distribution

Figure 13: Data distributions of MetaSyn.

Table 3: Mean absolute error (MAE) and mean squared error (MSE) of the cost model on 100 randomly sampled 10 tables following the unseen tables setting in Section 5.5, where the training tables are the first half of the pool, and the testing tables are the second half. The prediction error of the multi-table cost is around 1.3 milliseconds on the training tables and 3 milliseconds on the unseen tables.

MAE (training)	MSE (training)	MAE (testing)	MSE (testing)
1.321	3.408	3.061	10.835

the micro-benchmark is consistent with the kernel time obtained by profiling. In essence, the micro-benchmark is general and can be applied to other operators as well. Thus, we have deployed it in the production environment to support general micro-benchmarking. It is also open-sourced under PARAM Benchmarks[11].

B.2 Details of Features

The features are important for learning. Recall that we have used the following features in the cost model and the RL policy: table dimension, hash size, pooling factor, table size, indices distributions, and step-aware feature. We elaborate on these features below.

- **Table dimension:** it is the dimension of each embedding vector in the table. This feature is normalized to have a mean of 0 and a standard deviation of 1.
- **Hash size:** it is the number of rows in a table, which is determined by the feature cardinality. This feature is also normalized.
- **Pooling Factor:** it is obtained by dividing the total number indices with the batch size. This feature is also normalized.
- **Table size:** it is obtained by calculating the parameter size in GBs. We do not normalize this feature.
- **Indices distributions:** Since the majority of the indices are visited very few times, we divide frequencies into several bins, where the size of each bin grows exponentially. Specifically, we use the following 17 bins: $(0, 1]$, $(1, 2]$, $(2, 4]$, $(4, 8]$, $(8, 16]$, $(16, 32]$, $(32, 64]$, $(64, 128]$, $(128, 256]$, $(256, 512]$, $(512, 1024]$, $(1024, 2048]$, $(2048, 4096]$, $(4096, 8192]$, $(8192, 16384]$, $(16384, 32768]$, and $(32768, \infty)$. We count the number of times each index appears in a batch of indices and put the count into the corresponding bin. Then we calculate the ratio for each bin, which results in 17 features. For example, the first feature is the ratio of the indices for the bin $(0, 1]$. The fourth feature is the ratio of the indices for the bin $(4, 8]$.
- **Step-aware feature:** it is defined as the the ratio of the tables that have already been assigned.

[11]https://github.com/facebookresearch/param

C IMPLANTATION DETAILS

C.1 Neural Architecture

Architecture of the cost model. We use a two-layer MLP to obtain the representation for every single table. The input dimension is 21, which is the number of table features. The hidden dimension is 128. The output size is 32; that is, the size of a single table representation is 32. For multi-table cases, we sum the table representations to obtain multi-table representation, whose size is also 32. Finally, we use another two-layer MLP to produce the multi-table cost, where the hidden dimension is 64, and the output size is 1.

Architecture of the policy-value network. We first use one-layer MLP to map the state-aware feature to a 32-dimensional representation. Then we concatenate the single table 32-dimensional representation (shared with the cost model) of the upcoming table to obtain a 64-dimensional state representation. The state representation is processed by a two-layer LSTM with a hidden size 64. For actions, we similar obtain a multi-table representation for each shard. We use a one-layer MLP to map the shard cost to a 32-dimensional cost representation. The action representation is obtained by concatenating the multi-table representation and the cost representation, and is 64-dimensional. For the policy head, we concatenate the state representation, the action representation, and their dot-product and use a four-layer MLP with size 128-128-64–1 to generate the action confidence. Then SoftMax is applied to obtain the probabilities. For the value head, we use a two-layer MLP with a size 64-1 to produce the state value.

C.2 Hyperparameters Configuration

We summarize all the hyperparameters of AutoShard below.

- **Cost model training**: batch size $B_2 = 512$, the number of update iteration $I = 20$, the buffer size is 5000.
- **RL training**: batch size $B_1 = 8$, number of data collection steps $T = 100$, number of learning threads is 1, entropy weight is 0.001, baseline weight is 0.5, discounting factor is 1.0, gradient is clipped with threshold 40, and the total number of training steps is 100,000.
- **Optimizer**: We use Adam optimizer with a learning rate of 0.001. The other hyperparameters are kept as default.

C.3 Hardware

For MetaSyn, we conduct all the experiments on a server with 48 Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz processors, 188 GB memory, and four NVIDIA GeForce RTX 2080 Ti GPUs. For MetaProd, the server has a similar hardware environment but with NVIDIA V100 GPUs.