AdaEmbed: Adaptive Embedding for Large-Scale Recommendation Models

Fan Lai[2]*, Wei Zhang[1], Rui Liu[1], William Tsai[1], Xiaohan Wei[1], Yuxi Hu[1], Sabin Devkota[1], Jianyu Huang[1], Jongsoo Park[1], Xing Liu[1], Zeliang Chen[1], Ellie Wen[1], Paul Rivera[1], Jie You[1], Chun-cheng Chen[1], Mosharaf Chowdhury[2]
[1]*Meta* [2]*University of Michigan*

Abstract

Deep learning recommendation models (DLRMs) are using increasingly larger embedding tables to represent categorical sparse features such as video genres. Each embedding row of the table represents the trainable weight vector for a specific instance of that feature. While increasing the number of embedding rows typically improves model accuracy by considering more feature instances, it can lead to larger deployment costs and slower model execution.

Unlike existing efforts that primarily focus on optimizing DLRMs for the given embedding, we present a complementary system, AdaEmbed, to reduce the size of embeddings needed for the same DLRM accuracy via in-training embedding pruning. Our key insight is that the access patterns and weights of different embeddings are heterogeneous across embedding rows, and dynamically change over the training process, implying varying embedding importance with respect to model accuracy. However, identifying important embeddings and then enforcing pruning for modern DLRMs with up to billions of embeddings (terabytes) is challenging. Given the total embedding size, AdaEmbed considers embeddings with higher runtime access frequencies and larger training gradients to be more important, and it dynamically prunes less important embeddings at scale to automatically determine per-feature embeddings. Our evaluations in industrial settings show that AdaEmbed saves 35-60% embedding size needed in deployment and improves model execution speed by 11-34%, while achieving noticeable accuracy gains.

1 Introduction

Deep learning recommendation models (DLRMs) are important to many online services, including Google advertisement display [9, 10], Netflix movie recommendations [15, 27], and Amazon e-commerce [40], and comprise up to 65% of AI cycles in Meta's datacenters [13, 18]. Unlike conventional machine learning (ML) counterparts that train models on continuous input features (e.g., color values of images), DLRM inputs consist of continuous dense features (e.g., timestamp) and categorical sparse features (e.g., video genres). Each sparse feature is often associated with an embedding table, where each instance of that feature is represented by a trainable embedding row (weight vector). In the forward and backward passes of model execution, the model reads and updates the embedding weights of accessed rows.

Because the accuracy of a DLRM typically increases with larger embeddings (e.g., by considering more feature instances), modern DLRM embedding size is ever growing (up to terabytes and billions of embeddings [13, 50]). This introduces multiple challenges. First, DLRMs often have stringent throughput and latency requirements for (online) training and inference [26, 45], but gigantic embeddings make computation [34], communication [4, 39] and memory optimizations [13, 52] challenging. To achieve desired model throughput, practical deployments often have to use hundreds of GPUs to hold embeddings [35]. Meanwhile, designing better embeddings (e.g., number of per-feature embedding rows and which embedding weights to retain) remains challenging because the exploration space increases with larger embeddings and requires intensive manual efforts [32, 49].

Unlike existing DLRM efforts that have primarily focused on optimizing the model's execution speed for the given embeddings – e.g., by balancing embedding sharding [35, 52], accelerating embedding retrieval [39, 44], compressing embeddings [19, 48], or elastic resource scaling [45, 51] – we explore a complementary opportunity: *Can we fundamentally reduce the size of embedding needed for the same accuracy, by dynamically optimizing the per-feature embedding during model training?* Or, equivalently, *can we improve model accuracy for the given embedding size?* This is because unlike classic ML models, the DLRM model output (accuracy) is determined by the input data (e.g., accessed instances) and their embedding weights, and the input data is typically organized chronologically during training to account for the diverse and non-stationary user preferences [53]. Therefore, the access patterns and the weights of embeddings vary across embeddings rows and the training process (§2.2). This implies an opportunity to admit and prune embedding rows based on their heterogeneous importance to improve model accuracy.

In this paper, we introduce an *automated in-training pruning system* to Adaptively optimize per-feature Embeddings (AdaEmbed) for better model accuracy. For the given embedding size, AdaEmbed scalably identifies and retains embeddings that have larger importance to model accuracy at particular times during training. As a result, not only does it reduce human effort in embedding design, but it also cuts down the embedding size, thus the computational, network,

*Work done while the author was working at Meta.

and memory resources, needed to achieve the same accuracy. AdaEmbed is complementary to and supports existing DLRM efforts with a few lines of code changes (§3).

Unfortunately, identifying important embeddings out of billions is non-trivial. To maximize the overall model accuracy, we should retain the embedding rows that affect model inputs more often (e.g., are frequently accessed) and that affect model outputs the most (e.g., have larger weights) (§4.1). However, the non-stationary data distribution during training leads to the spatiotemporal variation in the access frequency of different embeddings. e.g., new videos are posted and become popular, while some old ones lose popularity. Moreover, embedding weights change over training iterations and so does their impact. Once we prune an embedding's weights from the GPU memory, we cannot accurately capture their importance to model accuracy as training moves on. Based on our analytical insights, embeddings with larger runtime gradients and higher access frequencies tend to accumulate larger embedding weights, and AdaEmbed prioritizes them when deciding which ones to retain. Moreover, we group features with similar feature-level characteristics (e.g., vector dimensions), and then identify important embeddings across feature groups to optimize the per-feature embedding size and which embedding to retain (§4.1).

Enforcing in-training pruning after identifying important embeddings is not straightforward either. Pruning for practical DLRMs can require reallocating millions of embedding rows and tens of gigabytes of embedding weights per training iteration, whereas each iteration takes only a few hundred milliseconds [4, 35]. While frequent pruning allows admitting important embeddings in a timely manner, thereby improving model accuracy, it can slow down model training by many hundred times (§4.2). To achieve a sweet spot between timely pruning and low overhead, AdaEmbed initiates pruning selectively when perceiving big changes in the importance distribution of all embeddings, thus reducing the number of pruning rounds needed while ensuring high accuracy. However, existing DLRM systems face difficulty in dynamically admitting and pruning embeddings at scale because they often rely on static and/or fixed-size embedding storage [1–3, 44]. AdaEmbed introduces a shim layer, Virtually Hashed Physically Indexed (VHPI) embedding, to support various embedding designs. VHPI decouples the management of embeddings from their physical weights, whereby it recycles the weight vector of embeddings to avoid intense memory allocation (§4.3).

We have implemented a system prototype of AdaEmbed (§5) and evaluated it using five industry models and months of data across hundreds of GPUs (§6). Our evaluations show that AdaEmbed can reduce 35-60% embedding size, implying comparable resource savings, and improve model execution speed by 11-34% without compromising model accuracy. Meanwhile, it achieves noticeable accuracy gains under the same embedding size, thus being able to reducing manual efforts by automatically finding better per-feature embeddings.

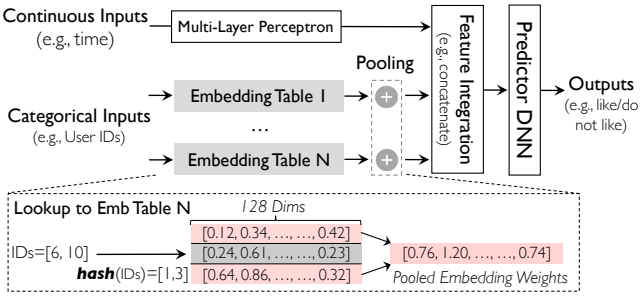


Figure 1: ***DLRM models consist of large embedding tables.***

Overall, we make the following contributions in this paper:

1. We propose an in-training pruning system, AdaEmbed, to automatically optimize DLRM embeddings.
2. We introduce embedding importance to capture important embeddings and employ VHPI embedding to enforce scalable pruning, with few changes to existing designs.
3. We evaluate AdaEmbed in various real-world settings to show its resource savings and accuracy gains.

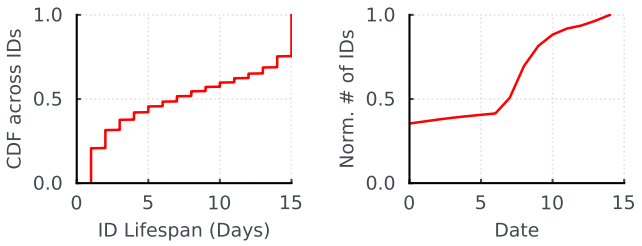
2 Background and Motivation

We start with a quick primer on DLRMs (§2.1), followed by the challenges it faces and inefficiencies of the state-of-the-art based on our analysis of real-world experiments (§2.2). Next, we highlight the opportunities that motivate our work (§2.3).

2.1 Deep Learning Recommendation Models

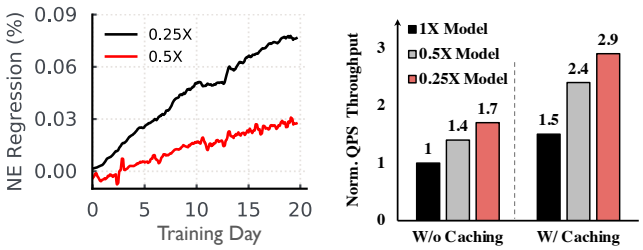
As shown in Figure 1, a DLRM consists of a combination of fully connected multiple-layer perceptrons (MLPs) to capture continuous dense features (e.g., timestamp), and a set of embedding tables to map various categorical sparse features (e.g., user and video IDs) to a dense representation. DLRMs can contain up to thousands of sparse features: each feature is typically associated with an embedding table, and each table can have millions of rows [15, 35, 52]. Each embedding row is a multi-dimensional weight vector (e.g., 128 floats) corresponding to a specific feature instance (e.g., a specific user ID of feature "User IDs").

DLRMs differ from traditional computer vision (CV) and natural language processing (NLP) models in that they require training on large volumes of data organized chronologically, to keep up with the latest recommendation trends. Hence, the distribution of training data changes over the training process. In the forward pass of model computation, each input sample includes a set of embedding IDs for each table to extract the corresponding embedding weights (vectors). To reduce the computation complexity, embedding weights of a sample will be pooled per table using the element-wise *pooling* operator, which typically takes the sum or maximum along each vector dimension (Figure 1). The pooled embedding weights of mini-batch samples are packed together with their intermediate outputs of dense features, forming a batch input to deeper layers. In the backward pass, the weights of the accessed embeddings are updated using the gradient.



(a) Lifespan of instances. (b) Number of instances.

Figure 2: ***The number of sparse feature instances (IDs) increases rapidly over time, while the lifespan of instances is heterogeneous.***



(a) Large embeddings improve NE. (b) Large embeddings hurt QPS.

Figure 3: ***Compared to the full (1×) model, smaller embedding sizes hurt model NE (i.e., larger NE regression), but improve QPS. 0.25× and 0.5× denote using 25% and 50% of the full model size.***

Due to the enormous number of sparse feature instances, their embedding weights can occupy more than 99% size of a commonly used model (up to several terabytes) [21]; so DLRMs exhibit much larger memory intensity than conventional ML models (e.g., ResNet). As such, practical DLRM deployments use a combination of model parallelism for sparse feature layers and data parallelism for MLPs. The former allocates different embedding partitions across workers to avoid replicating them, and the latter enables concurrent processing of dense feature inputs [13]. Even so, model deployments often require hundreds of GPUs to achieve the desired model throughput (a few hundred milliseconds per iteration) [4, 35].

2.2 Challenges in DLRM Deployment

Due to its significant impact on revenue and numerous iterations needed to train a DLRM model, DLRM deployments follow the "*achieve better accuracy and run as fast as possible*" paradigm [35, 45, 52]. The execution speed and accuracy of a DLRM model are respectively measured by Query-Per-Second (QPS) throughput and Normalized Entropy (NE) loss [22]. Larger QPS and smaller NE indicate better performance, and any relative > 0.02% NE gain is considered to be significant [13, 46]. However, optimizing both aspects leads to novel tussles and challenges in real-world deployments.

Larger embedding sizes improve NE Embedding size of modern DLRMs is ever-growing to accommodate more embedding rows for sparse features and their instances [35, 44]. Figure 2 reports the size of the instance set over 15 days' data in a real-world DLRM system. We observe that even though a small portion of the trained instances will seldom be accessed again in later days (Figure 2(a)), the total number of unique instances increases by 1.5× every week (Figure 2(b)). As DLRMs are often trained on months of data and retrained over time, the size of the instance set will eventually far exceed the embedding size. To cap the embedding size, existing designs often perform hashing on the raw instance IDs, and then use the hashed IDs to access their embedding rows [3].

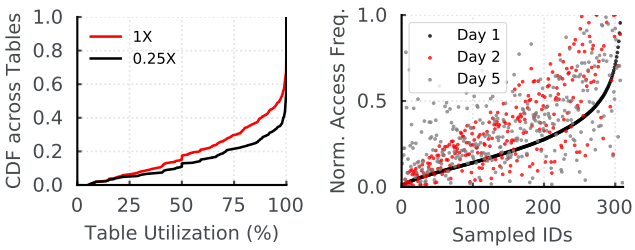
Intuitively, using more embedding rows implies more instances are considered, thus enabling better data coverage for better NE. Figure 3(a) reports the impact of the embedding size on the NE regression at different times of training. NE regression denotes the accuracy degradation of using a smaller embedding size w.r.t. the full-size model.We notice that (i) using a smaller embedding size can greatly hurt NE. For example, reducing the number of embedding rows by 75% (i.e., 0.25× model) results in ∼0.02% NE regression on Day 2; Worse, (ii) this NE regression inflates as the training evolves over time as more instances are spawned.

Large embedding sizes hurt QPS However, using more embeddings can slow down model execution and consume more machine resources in multiple execution phrases: (i) slower embedding access if we can not retain all embeddings in high-bandwidth GPUs; (ii) longer communication as we may need to transfer more embeddings over the network [4, 50]; and (iii) longer computation as more embeddings need to be computed on. Figure 3(b) shows, compared to the full model, 0.5× model achieves 1.4× QPS speedup in the same resource setting. Here, we note that state-of-the-art DLRM optimizations [35, 44], which cache and prefetch the embeddings to be accessed in future batches, cannot eliminate the QPS drop (Figure 3(b)). More importantly, they can be insufficient for online training and model serving as we may not know the input data in advance.

2.3 Opportunities for In-Training Pruning

For a given DLRM, recent advances have made considerable progress for efficient communication [4, 19, 39] and/or computation [13, 26, 35]. Instead, we focus on a complementary opportunity that *reduces the embedding size needed without NE regression, by adaptively pruning embeddings during model training*. Our approach is based on the following observations.

Handcrafted embeddings are suboptimal Designing optimal embeddings (e.g., deciding the number of per-feature embedding rows and which embedding weights to retain) is as yet an open problem in the ML community [14]. Hence, DLRM systems often decide the embedding size using human-defined rules, e.g., by estimating the feature popularity [14] and/or hyper-parameter tuning by model experts before training takes place [52]. Not only does this require great human effort and resources to explore, but it can also be suboptimal due to limited adaptivity at runtime (e.g., deciding which instance's embedding to retain if many instances are generated).



(a) Utilization of embedding tables. (b) Heterogeneous ID access.

Figure 4: ***Embedding access varies across IDs and over time, leading to distinct table utilization in existing embedding designs.***

Worse, existing DLRM systems often treat per-feature embedding tables individually for ease of management. This can underutilize or overload individual tables as data distribution changes over time. Indeed, when we analyze the table utilization in a one-day training window (i.e., number of accessed embeddings over the total number of embeddings on that day), we notice large heterogeneity (Figure 4(a)). Intuitively, tables that are fully utilized can degrade NE because many instances are hashed to the same embedding row, leading to hash collisions. However, underutilized tables cannot trade in their space during training because of the suboptimal predetermined embedding size and inelastic embedding designs.

Embeddings have heterogeneous characteristics Figure 4(b) zooms into individual embedding rows, where the sampled IDs (i.e., x-axis) are ordered based on their access frequency on Day 1. We notice that the access frequency of embeddings varies across embedding IDs and over time, since user preferences change over time. We have similar observations on embedding weights too (§4.1). Since the model output (accuracy) is determined by the input instance (e.g., which embedding is accessed) and embedding weights, this implies a potential to identify and retain more important embeddings during training to maximize final model accuracy.

3 AdaEmbed Overview

In this paper, we introduce an *automated in-training pruning system*, AdaEmbed, to adaptively optimize per-feature embeddings at scale for better model accuracy. Unlike existing efforts for model pruning, which focus on conventional models [8,11,20] and/or prune model size when training completes [32], AdaEmbed automatically identifies and retains important embeddings for the given embedding size to improve performance while training is ongoing. Our evaluations in industrial settings show that in addition to saving resource throughout training, AdaEmbed provides superior model accuracy to its post-training pruning counterparts (§6.4).

AdaEmbed is a complementary system that acts as a shim layer atop today's embedding designs (Figure 5). It has a central coordinator and a set of distributed on-worker agents:

- *AdaEmbed Coordinator*: It gathers the embedding information from agents, determines the global pruning decision, and orchestrates the agent to enforce the pruning.

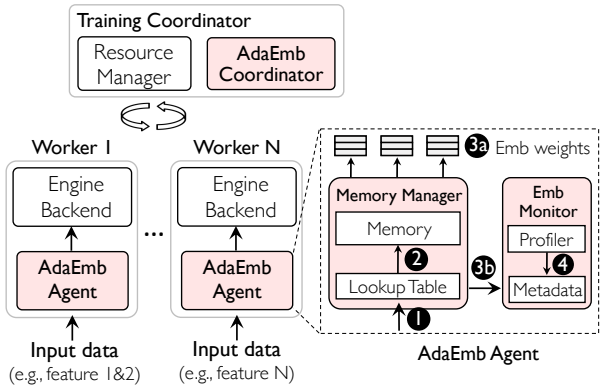


Figure 5: ***AdaEmbed overview and its in-training execution flow. AdaEmbed components are in red.***

- *Memory Manager*: It is located inside each AdaEmbed agent and manages the physical memory for today's embedding designs. At runtime, it receives the pruning decision from the coordinator and executes pruning on local embedding weights.
- *Embedding Monitor*: It resides along with the memory manager to track embedding importance and reports the profiling results of the importance to the coordinator.

Figure 6 illustrates the interface of AdaEmbed, which supports existing DLRM systems in a few lines of code.

```
1  import AdaEmbed
2
3  def dlrm_model_training():
4    # Wrap existing embedding modules
5    emb_agent = AdaEmbed.create_agent(
6      emb_tables=model.embs, pruning_config=config)
7
8    for _ in range(num_iterations):
9      input_ids = get_next_data_batch()
10
11     # Look up physical embedding address
12     emb_physical_ids = emb_agent.look_up(input_ids)
13     feedback = model.train_step(emb_physical_ids)
14
15     # Update embedding importance with feedback
16     emb_agent.update_importance(input_ids, feedback)
```

Figure 6: ***AdaEmb supports existing DLRMs with minor changes.***

Training Lifecycle Similar to current DLRM deployments, ❶ each worker is in charge of a subset of sparse features, which is determined by the embedding partition of model parallelism. The worker processes the input data (i.e., a list of embedding IDs) of those features. ❷ However, the inputs are first forwarded to AdaEmbed agent to look up the physical address of each embedding's weights (Line 12). ❸ The physical address is then used to fetch the embedding weights for read and write operations. The rest of model training adheres to existing designs. ❹ After each training iteration, the embedding monitor updates the embedding importance with the training feedback (Line 16). Periodically, it samples the importance of different embedding rows and notifies the coordinator of the profiling results. The coordinator determines how to prune embeddings subject to the total embedding size and guides the

memory manager to admit and prune embeddings at scale.

4 AdaEmbed Design

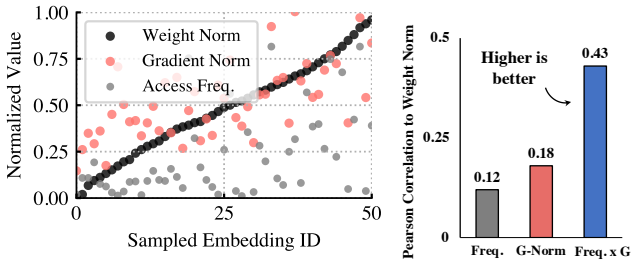
Practical DLRMs often contain hundreds of sparse features and up to billions of embedding rows [13, 50]. They run across hundreds of GPUs on non-stationary model inputs to get the desired model execution speed [4, 35]. These lead to the following challenges toward practical in-training pruning of embedding rows:

- *Heterogeneity*: The characteristics of embeddings (e.g., data distribution and embedding weights) vary across instances of the same feature. This, as well as the physical size of the embedding row, differs across features too. How to measure which embeddings are important to retain for better model accuracy (§4.1)?
- *Dynamics and Scalability*: The importance of individual embeddings varies over iterations at a sub-second speed. As such, improving model accuracy requires pruning in a timely manner to maximize the number of important embeddings. However, identifying important embeddings out of billions distributed across hundreds of workers, and then pruning on terabytes of embedding weights can lead to large overhead. How to orchestrate pruning under training dynamics (§4.2)? Additionally, how to efficiently enforce pruning on each worker's memory to avoid throughout degradation (§4.3)?
- *Extensibility*: Existing systems are built atop a variety of embedding designs, such as key-value storage [44, 52] or highly optimized but fixed-size tensors [2, 3]. How to provide generic systems support to minimize modifications to existing DLRM systems (§4.3)?

4.1 Embedding Monitor: Identify Important Embeddings

Given the embedding size, we aim to trade the less important embedding rows for the more important ones. This requires us to consider the importance of each embedding row in terms of the contribution of its embedding weights to model accuracy, as well as its physical size. However, determining the optimal pruning strategy during training is challenging. First, the model output (accuracy) is affected by the complex interplay between input feature instances (e.g., which item IDs appear) and their embedding weights. Even with full model information after training completes, pruning is still a fundamental open problem in the ML literature [11, 32]. Second, during model training, this interplay becomes more intractable because of the large spatiotemporal variations in the distribution of model inputs and embedding weights (Figure 7(a)). Worse, once we prune an embedding's weight vector, it is difficult to assess its impact on model accuracy as training moves on. These challenges are amplified by the need to account for feature-level heterogeneity too (e.g., different weight vector sizes across features).

AdaEmbed employs the embedding monitor to capture the embedding importance of individual rows within the feature, and then extends it to identify important rows across features.



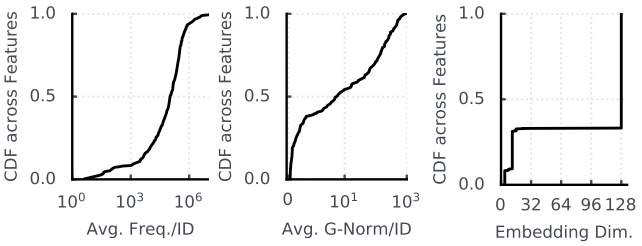
(a) Heterogeneous emb. characteristics. (b) Pearson analysis.

Figure 7: ***(a) Embedding gradient and access frequency are heterogeneous, (b) while their combination reports a larger correlation to the embedding weights. A correlation value > 0.4 indicates a positive and medium to strong association.***

Intra-Feature Embedding Importance For embeddings of the same feature, we introduce a data- and model-aware importance metric $EI(i)$ to capture the importance of each row i to model accuracy. Instead of relying on the embedding weights that become stale after being pruned, $EI(i)$ is the runtime combination of access frequency and gradient, i.e., $EI(i) = freq_t(i) \times \|\nabla g_t(i)\|$. $\|\nabla g_t(i)\|$ is the L2-norm of i's gradient in iteration t, and $freq_t(i)$ is the access frequency. So the embedding with a higher access frequency and a larger gradient norm is deemed more important. Here, collecting $EI(i)$ introduces negligible overhead, because the embedding gradient is already generated during back-propagation of training regardless of AdaEmbed. Since the gradient is generated and shared by mini-batch samples [35], the importance of pruned-but-accessed embeddings will continue to be updated.

Our importance design is motivated by multiple factors:

- Intuitively, the output of sparse feature layers (i.e., pooled embedding weights) is often derived by taking the sum or maximum of input embedding weights (§2.1); so we should retain the embeddings that affect many model inputs (i.e., frequently accessed) and that affect model outputs more (i.e., larger weights). While we do not have information about future weights after pruning an embedding, we observe a strong correlation between our frequency-gradient combined metric and the final embedding weights when training converges (Figure 7). This is because frequent weight updates with large gradients typically result in larger weights.
- Theoretically, embedding rows are designed for training different bins of data instances: each bin holds only one type of category instance (i.e., a specific ID), and bins can have different data volumes (i.e., different access frequencies of IDs). Now, we want to select and retain certain bins (embeddings). This, in concept, is similar to the importance sampling problem in the ML literature [17, 25]: To improve model convergence by selecting the right bins to train the model, the optimal solution is to select bin i with a probability proportional to the aggregate gradients



(a) Heter. frequency. (b) Heter. grad. norms. (c) Heter. dimensions.

Figure 8: ***Magnitudes of embedding access frequencies and gradients vary across features, making it hard to compare*** $EI(i)$.

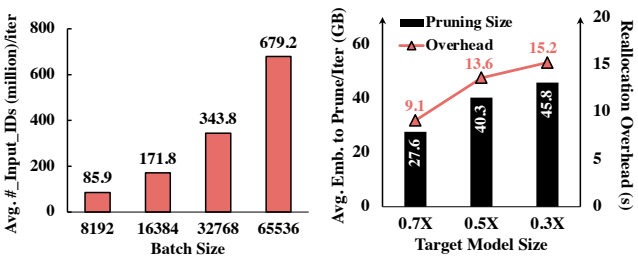
of training all that bin's data. In our formulation, the training samples within the same bin are identical, because they correspond to the same specific ID. Therefore, the aggregate gradients herein is equivalent to the product of the number of training samples and the gradient of the individual sample (i.e., $EI(i) = freq_t(i) \times \|\nabla g_t(i)\|$).

Empirically, our fleet-wide evaluations show that our importance design outperforms its alternatives (§6.4).

Since the gradient and access frequency can fluctuate during training (e.g., due to the randomness in sampling mini-batches), we need to account for these uncertainties in $EI(i)$. Here, the embedding monitor considers $EI(i)_t = freq_t(i) \times \|\nabla g_t(i)\| + EI(i)_{t-1}$, whereby we reduce uncertainties in individual iterations and only need to update the importance of accessed embeddings. This is because the importance of not accessed embeddings remains unchanged as $freq_t(i)$ = 0. In reality, only a subset of embeddings are accessed, so we can reduce the overhead significantly (§4.2). Moreover, to account for the temporal variation, we use a moving average that decays $EI(i)$ by a factor of 0.8 every T iterations.

Inter-Feature Group Pruning Retaining important embeddings subject to the total size naturally leads to a global pruning design, in which we hope to allocate different embedding sizes to individual features. However, the values of embedding importance can vary across features by orders of magnitude. This can be due to features with fewer instances often having larger average access frequencies per embedding, and/or different initialization mechanisms of the embedding weights leading to gradients of different magnitudes (Figure 8). As such, directly using the intra-feature importance for comparison across features can result in a large bias, as embeddings with greater importance values are not necessarily more important than those of other features. Moreover, as the dimension of embedding vectors of different features can vary (Figure 8(c)), deciding which embeddings are more valuable to retain becomes intricate when large embedding importance and vector size are in conflict.

Because we rely on the relative ranking of importance to determine pruning (e.g., prune the tail 40% less important embeddings), we can tackle the comparison bias across features using the popular normalization philosophy [16]; i.e., by normalizing each embedding's importance by that



(a) Number of accessed IDs. (b) Size of weights to prune.

Figure 9: ***(a) Each iteration accesses millions of embeddings. (b) Pruning needs to reallocate a large amount of embedding weights.***

feature's distribution of all embeddings' importance. This way, the embedding importance of different features takes on similar ranges of values, and the more important embeddings of each feature are still prioritized because of having larger relative importance values after normalization. The embedding monitor normalizes the embedding importance of each feature by the 95th percentile of its distribution (i.e., $EI(i)/EI_{95th}(feature(i))$) to avoid outliers.

Next, to account for different weight vector sizes across features, AdaEmbed groups features with the same embedding dimension and then performs global pruning within the feature group. In reality, DLRMs are configured with only a handful of distinct embedding dimensions (Figure 8(c)) to reduce hyper-parameter tuning and/or to achieve better parallelism (e.g., balancing embedding sharding [35, 52]). This implies a big opportunity to group many features, which already forms a large shared embedding size for inter-feature group pruning. By default, AdaEmbed initializes the per-group embedding size based on the number of in-group features and the total embedding size (i.e., $\frac{num\_group\_features \times group\_feature\_dim}{num\_features \times avg\_feature\_dim} \times total\_size$) to uniformly allocate the space to each dimension. Note that unused embedding storage will be picked up by other groups (§4.3). When developers have more advanced information about features (e.g., feature importance), AdaEmbed provides APIs for customizing feature groups and sizes (§5).

Our evaluations show that with importance normalization and group pruning, AdaEmbed achieves better resource savings and model accuracy (§6.3).

4.2 AdaEmbed Coordinator: Prune at Right Time

In real-world DLRM systems, each training iteration involves updating the importance of millions of embedding rows in terabyte-sized models (Figure 9(a)). At that scale, orchestrating hundreds of workers to prune leads to a trade-off between the pruning overhead and quality. Frequent pruning allows for better decision quality, i.e., maximizing the number of important embeddings all the time for potentially better model accuracy. Yet, pruning can require cleaning up and creating tens of gigabytes of embedding weights, which can take many seconds and significantly slow down the sub-second training iterations (Figure 9(b)). This trade-off becomes more

Algorithm 1: Pseudo-code of AdaEmbed runtime

```
1:  weight_table ← EmbWeights()          ▷ Physical weight tables
2:  emb_meta ← Init(weight_table)        ▷ VHPI metadata
3:  pruning_start ← false                ▷ Enforce pruning or not
4:  Function UpdateEmbs (input_ids, feedback):
        /* Monitor: Update embedding importance
           asynchronously to model training.        */
5:      UpdateImport(input_ids, feedback)
6:      if pruning_start == true then
7:          EnforcePruning()             ▷ Stall training
8:          pruning_start ← false
9:  Function MonitorImportance (ProfilingInterval Δ):
        /* Coordinator: asynchronously inspect big changes on
           the importance distribution via profiling across
           workers.                                 */
10:     last_dist ← null
11:     while training == true do
12:         if mod(current_time, Δ) == 0 then
13:             cur_dist← ProfileImportance()
14:             pruning_start←Diff(last_dist, cur_dist) > p
15:             last_dist ← cur_dist
16: Function EnforcePruning ():
        /* Memory manager: Identify embedding rows to admit
           and prune subject to the given embedding size.   */
17:     admit_emb, evict_emb ← IdentifyRecycleEmbs(
18:                 emb_meta, weight_table.size)
        /* Redistribute the lookup mapping from the embedding
           ID to the weight vector, whereby admitted embedding
           rows can recycle the weight vector of pruned ones. */
19:     RedistLookup(emb_meta, admit_emb, evict_emb)
        /* Reset embedding weights for admitted embeddings. */
20:     weight_table.ResetEmbs(admit_emb)
```

intractable as a result of training dynamics; e.g., stochastic gradient descent can introduce large noise to embedding gradients, thus the embedding importance. As such, pruning too frequently can also be suboptimal (§6.4).

To find the sweet spot between pruning overhead and quality, AdaEmbed Coordinator decides the right time to prune to reduce the number of pruning rounds needed, and instructs the memory manager to minimize the overhead in each pruning round when pruning embedding weights (§4.3). Algorithm 1 outlines how AdaEmbed Coordinator orchestrates efficient embedding pruning. The embedding monitor updates the importance of accessed embeddings after each training iteration (Line 4), and periodically profiles embedding importance (Line 9). The results of the profiling will be sent to the coordinator. In the event of big changes in the importance distribution, the coordinator initiates a new pruning round and notifies the memory manager of the pruning decision (Line 9). The memory manager on each worker then executes pruning and admits new embedding weights at scale (Line 16).

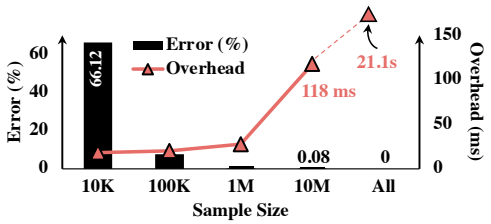


Figure 10: ***Profiling can get accurate results with little overhead.***

Intuitively, pruning cares about the importance ranking of individual embeddings instead of their dynamic importance. Therefore, AdaEmbed coordinator relies on the importance distribution of all embeddings again, and initiates pruning if the importance distribution has changed greatly since the last pruning round. To effectively gather the importance distribution across hundreds of machines, each local agent samples a small portion, P, of embedding importance values on that agent. The coordinator then can estimate how many embeddings have crossed the pruning boundary, i.e., the number of embedding rows whose importance ranking has fallen below or risen above the X_{th} percentile of the distribution since the last pruning round. X_{th} is the cut-off importance boundary determined by the size limit (i.e., $\sum_{emb \in top\_Xth} size(emb) < total\_size$), and the agent will prune the weight vector of the embeddings whose importance value is smaller than the cut-off importance.

As shown in Figure 10, while more samples, P, allow for a more precise estimate of X_{th} importance, this will also increase the coordination overhead, such as in collecting importance distributed across hundreds of machines and then computing distribution changes. In fact, we can use the concentration theorem in the probability sampling [47] to decide the right number of samples.[1] This gives us $\sim$ 5M embedding rows out of billions to sample on each machine, in order to ensure a deviation from the global ground truth of less than 1%. In addition to having a smaller computation overhead, this results in negligible network traffic, $5M \times 4bytes \sim 20$ megabytes as $EI(i)$ is a 4-byte float, over tens of Gbps network to the coordinator. As suggested by today's data validation systems [7, 33], we consider a big change to have occurred and initiate pruning when more than c =5% of the total embeddings cross the boundary (i.e., we need to prune and admit more than c% embeddings), and issue this lightweight profiling per minute. This avoids the large overhead caused by pruning in each training iteration, while ensuring that the current embedding allocation is at most c% worse than what we can achieve through pruning in each iteration. We show that profiling achieves a small deviation and little overhead (i.e., the 5M sample size in Figure 10).

Convergence Analysis As described in §4.1, our design of embedding importance draws inspiration from importance

[1] The minimum number of samples P needed to ensure $Pr[|\bar{X} - E[\bar{X}]| < \varepsilon] > \delta$ is $P = \frac{(X_{max} - X_{min})^2 \ln(2/\delta)}{2\varepsilon^2}$ for the distribution of variable X. $E[\bar{X}]$, X_{max} and X_{min} are the expectation, maximum and minimum of X, respectively.

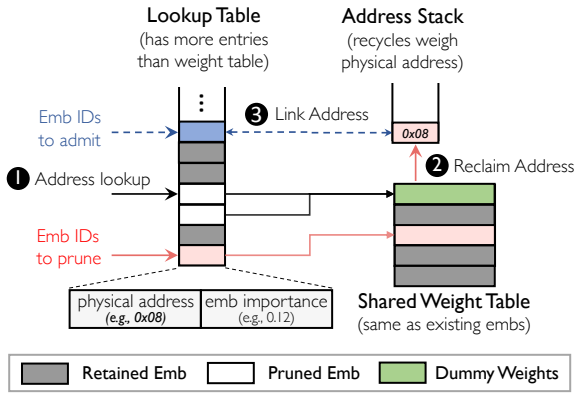


Figure 11: ***VHPI employs lookup table to link each embedding to the weight vector, and recycles the vector of pruned embeddings without intense memory allocation.***

sampling, which has been shown in ML theory [17, 25, 31] for its ability to reduce gradient variance and accelerate training convergence. Empirically, our extensive evaluations using months of real-world data and models demonstrate that AdaEmbed consistently improves model accuracy by pruning at the right time, as opposed to pruning too frequently or infrequently (§6.4).

4.3 Memory Manager: Prune Weights at Scale

As the reallocation of embedding weights is hundreds of times slower than each training iteration (Figure 9(b)), reducing the number of pruning rounds needed is still far from achieving negligible overhead in practice (§6.2). To avoid intense memory reallocation, the memory manager of AdaEmbed employs a Virtually Hashed Physically Indexed (VHPI) design to decouple the management of embeddings from their physical weight vectors, whereby AdaEmbed can recycle the weight vectors of different embeddings to enable efficient pruning for a variety of existing embedding designs.

VHPI primarily consists of two parts (Figure 11):

- *Lookup table*: It stores the metadata information of each embedding instance, including the embedding importance (a `float32`), and the physical address (a `int64`) to that embedding's weight vector. Compared to the weight vector, often a vector of 128 `float`, this payload information introduces a negligible memory footprint ($\frac{3}{128} \sim 2\%$).
- *Weight table*: It is a monolithic physical table for embedding weight vectors. It remains the same as the embedding table of today's DLRM systems, but it is shared across features under the orchestration of the memory manager.

Weights vectors of the pruned embeddings are not retained, while the metadata of all embeddings is always maintained in the lookup table. So the lookup table can include more entries (i.e., embedding IDs) than the weight table. This allows us to adaptively determine the link between embeddings and weight vectors to recycle weight vectors. Moreover, this can improve model accuracy by reducing hash collision (§6.4), as we can make the lookup table very large to accommodate

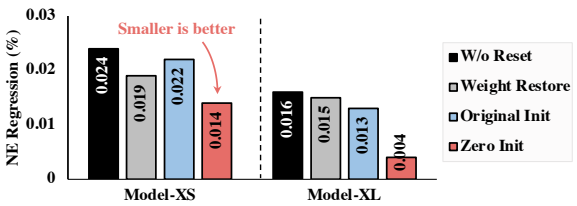


Figure 12: ***Zero initialization performs better (0.5× model).***

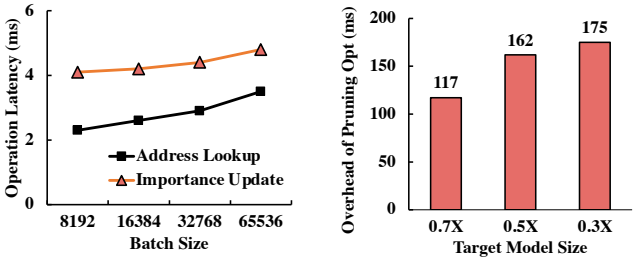
many embedding entries without expanding the weight table.

The memory manager performs two primitive operations for weight pruning at runtime (Figure 11):

- *Address lookup*: It looks up the physical weight address for each embedding ID to access its embedding weights. ❶ If that embedding row is pruned, to avoid breaking existing designs (e.g., missing weights due to pruning), lookup returns a shared physical address that points to a weights vector containing constant zeros. Access to this dummy vector will be folded on the execution backend due to the same entry, reducing redundant execution.
- *Weight allocation*: It executes the pruning decision to prune and admit embeddings. ❷ To prune an embedding row, VHPI first de-links and reclaims the current physical address of that embedding's weight. It then sets the address of the pruned embedding's lookup entry to the address of the shared dummy vector, redirecting the future access. ❸ To admit an embedding, VHPI pops an available physical address and links this address with the lookup entry, thereby recycling the physical memory. Meanwhile, the memory manager resets the weight vector values to clean up the previously pruned weight state.

However, it is not straightforward to reset (i.e., reinitialize) the weight values for admitted embeddings, because the model herein is partially trained and the values of embedding weights already differ by orders of magnitude (Figure 7(a)). Improper initialization (e.g., random initialization) can introduce a large amount of noise to the retained embeddings. Eventually, this will hurt model accuracy, especially considering the noise from millions of admitted embeddings in each pruning round.

Here, we investigated four popular strategies to reset weight vectors (Figure 12): (1) *w/o reset*: inherit the weights of pruned embeddings without resetting them; (2) *weight restore*: evict previously pruned weights to extra storage (e.g., disk) and reinstate the weights when that embedding is reclaimed; (3) *original initialization*: randomly initialize embedding weights as at the start of training; and (4) *zero initialization*: reset embedding weights to zeros. Intuitively, the restored weights will become too stale since they were pruned (often thousands of iterations ago). Original initialization and w/o reset can introduce large noise, as the weights have already been of differing magnitudes. Here, we advocate resetting the weight vector values to zeros, as this can avoid large noise while allowing the admitted embedding to learn from scratch. Indeed, our real-world evaluations report that zero initialization outperforms its alternatives (Figure 12).



(a) Control-plane overhead. (b) Data-plane overhead.

Figure 13: ***VHPI operations introduce little overhead.***

Overhead Analysis Among all the operations involved in VHPI, address lookup and importance update to the lookup table take place every iteration and consume a few milliseconds (Figure 13(a)). Weight pruning to the weight table consumes a few hundred milliseconds (Figure 13(b)), but it occurs every hundreds of iterations. Overall, these operations lead to little end-to-end overhead in large-scale deployments (§6.2).

5 Implementation

We implemented a system prototype of AdaEmbed to support distributed DLRM deployment across GPUs. Our implementation requires minor changes to existing DLRM systems.

AdaEmbed Backend AdaEmbed backend is implemented as GPU operators for fast execution. The VHPI metadata (e.g., embedding importance and weight address) are hosted on GPUs to process embeddings in parallel. The address lookup and importance update operations require no change to existing DLRM systems. As we need to reset the weight vector, the weight allocation operation requires existing frameworks to expose an API to access their weight table, but this requires a few lines of code change. The local agent interacts with the coordinator via TCP connections.

Fault Tolerance As a shim layer, AdaEmbed can be integrated into existing DLRM checkpoints by adding its state information to the model state. This not only minimizes the modification to existing designs, but also ensures that the saved AdaEmbed state conforms to the embedding weights at that time. When training is resumed, the model reloads the checkpoint, which restores the AdaEmbed state too. At runtime, AdaEmbed runs a lightweight daemon to back up VHPI metadata after each pruning round, and to resume its components if the current instances crash.

Interfaces AdaEmbed exposes Python APIs as the frontend (Figure 6), and it can also take *json* as input (Figure 14).

6 Evaluation

We evaluate AdaEmbed in real-world DLRM systems across hundreds of GPUs. Our evaluation results on different industrial models and months of data are summarized as follows:

- AdaEmbed can reduce 35-60% embedding size and improve model execution speed by 11-34% without compromising model accuracy (§6.2).

```
"adaembed_configs": {
  "total_emb_size": "1 TB",// Total embedding size
  "feature_configs": {
    "default_group": {...},
    "group_1": {  // Features to use group pruning
      "features": ["feature1", ...],
      "total_emb_size": "200 GB",
    } ...         // Other feature groups
  }
}
```

Figure 14: ***Example embedding configuration in AdaEmbed.***

- AdaEmbed can reduce manual efforts by automatically finding better per-feature embeddings, achieving noticeable accuracy improvements (§6.2-§6.3);
- AdaEmbed improves performance over a wide range of settings and outperforms its design counterparts (§6.4);

6.1 Methodology

Experimental setup We use models and data from industry DLRM systems in the evaluation. Table 1 depicts high-level statistics of the model. They span different scales and recommendation tasks, including click-through rate prediction and ranking. We train each model on 14 days' data to obtain the model *lifetime NE*, which indicates the cumulative model accuracy throughout training, and then test the model on the 15th day's data to get the *evaluation NE*. Each day has many terabytes of data input.

The training batch size of each model is 65536, requiring tens of GPU nodes for the desired QPS. Each GPU node has 8 A100 GPUs with 40 GB of GPU memory. The GPUs are interconnected using 200 Gbps RoCE NICs.

Baselines To the best of our knowledge, AdaEmbed is the first system to support in-training embedding pruning, and is complementary to existing DLRM efforts. Our evaluations cover two primary baselines: (i) *w/o AdaEmbed*: an industry DLRM system without AdaEmbed support. Based on the access frequency of embedding rows in previous days, rows that are less frequently accessed are removed before training starts. This generates a pruned model derived from the full model; and (ii) different variants of AdaEmbed with changes in the pruning algorithm (§6.4). Here, we focus on the performance improvement of the *w/ AdaEmbed* setup, i.e., the setup using AdaEmbed.

Metrics We care about the (i) *memory saving* to achieve the same model accuracy as with the full model (i.e., without NE regression)[2], because we want to minimize the embedding size for better model throughput and resource savings in deployment; (ii) *NE gain* that we can achieve using the same embedding size, since it not only minimizes manual efforts in configuring DLRM embeddings, but also implies higher revenues; and (iii) *overhead* that AdaEmbed introduces in model execution speed (i.e., QPS).

[2]A smaller Normalized Entropy (NE) loss indicates better model accuracy.

Model	# of Sparse Features (Approximate Value)	Raw Emb Size (Approximate Value)	# of GPUs	w/ Same Model NE		w/ Same Emb Size	
				Memory Saving	QPS Speedup	Avg. NE Gain (%)	QPS Overhead
Model-XS	1000	200 GB	32	≈ 35%	1.1×	0.015	0.4%
Model-S	600	350 GB	32	≈ 45%	1.2×	0.018	0.2%
Model-M	1000	1 TB	64	≈ 40%	1.2×	0.028	1.6%
Model-L	1000	1.1 TB	64	≈ 55%	1.3×	0.021	1.3%
Model-XL	800	1.5 TB	128	≈ 60%	1.3×	0.026	1.1%

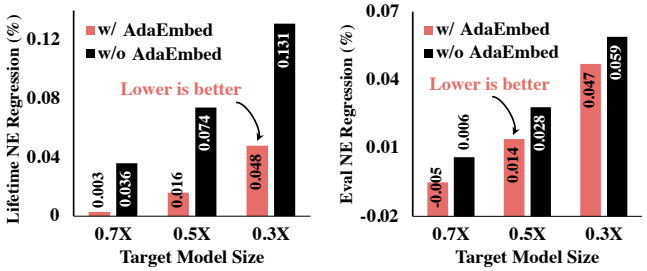
Table 1: ***Summary of improvements. AdaEmbed reduces the embedding size needed for the same model accuracy (NE), while improving NE using the same embedding size. We report the approximate memory saving, since evaluating all memory settings is unaffordable.***

6.2 End-to-End Performance

Table 1 summarizes the key memory saving, NE gain, and overhead of five models at different scales. Meanwhile, Figure 16 zooms into three representative models and reports their performance under different target embedding sizes. In our evaluations, NE regression measures the accuracy loss w.r.t. the full model (i.e., 1× model), and any > 0.02% NE gap is considered to be significant [13, 30, 52].

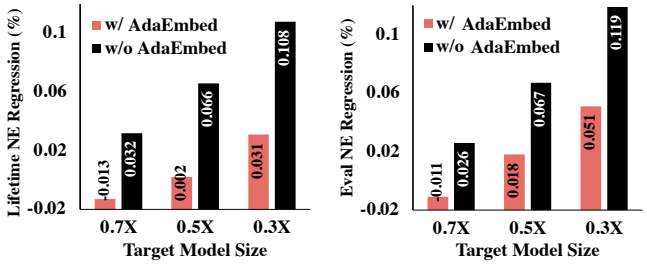
AdaEmbed cuts resource needs and improves QPS We first evaluate how many embedding sizes we can reduce without sacrificing model NE. Yet, evaluating all embedding sizes to get accurate memory saving is unaffordable because training with each setup takes thousands of GPU hours. So, we enumerate 0.7× (i.e., cut the embedding size by 30%), 0.6×, 0.5×, 0.4×, and 0.3× of the full model size to approximate this embedding saving with no accuracy drop. Table 1 reports that (i) AdaEmbed reduces the model embedding size by 35-60% with no reduction in model accuracy. This implies that we can reduce the machine usage by nearly the same amount (e.g., using 50% fewer GPUs); (ii) the resource savings are more encouraging for large models (e.g., Model-XL vs. Model-XS). One reason behind this is that large models provide gigantic GPU memory for AdaEmbed to reallocate embeddings via inter-feature group pruning (§6.3); and (iii) alternatively, reducing the fundamental embedding size provides 1.1-1.3× faster model execution speed (i.e., QPS) when running the model on the same machines.

AdaEmbed achieves better NE under the same size Figure 16 illustrates that with AdaEmbed, models can achieve 0.011-0.077% better NE using the same embedding size. We notice that (i) AdaEmbed achieves consistently better NE across models and under different target embedding sizes than the baseline; (ii) we can achieve NE gains with smaller embedding sizes (e.g., 0.7× models) even when compared to the full model. This is because AdaEmbed can automatically learn better per-feature embeddings, like the size and which embeddings to retain. Meanwhile, pruning less important embeddings can reduce model overfitting, thereby improving model generalization (accuracy) [6]; and (iii) the lifetime NE



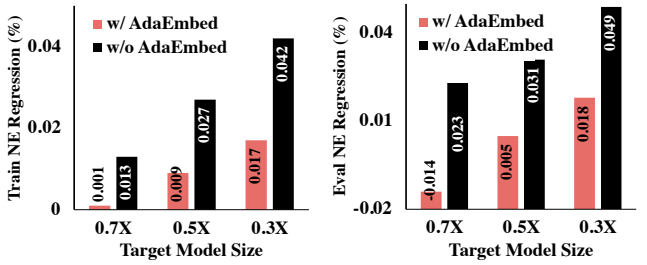
(a) Model-XS (Lifetime NE).

(b) Model-XS (Eval NE).



(c) Model-M (Lifetime NE).

(d) Model-M (Eval NE).



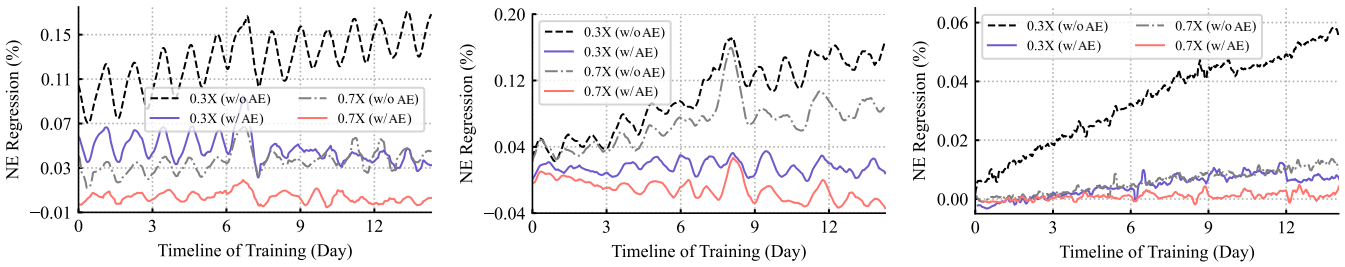
(e) Model-XL (Lifetime NE).

(f) Model-XL (Eval NE).

Figure 16: ***AdaEmbed achieves better lifetime NE and evaluation NE. Better lifetime NE implies potentially better model accuracy for online learning deployment, while better evaluation NE indicates better accuracy after offline training (i.e., prior to launching online training). Both NEs are important metrics.***

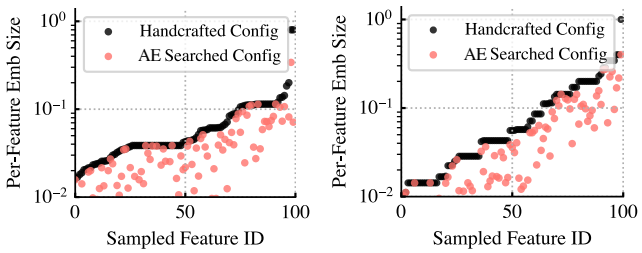
gain is more prominent than that of the evaluation NE, because the former is closer to the online deployment (i.e., retraining on real-time data), where AdaEmbed is able to adapt to the latest data distribution.

AdaEmbed introduces negligible overhead As shown in Table 1, compared to the same-size model in the baseline,



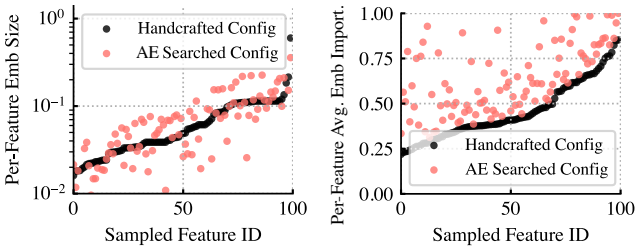
(a) Model-XS (b) Model-M (c) Model-XL

Figure 15: ***Models with AdaEmbed achieve consistently better NE over time. Troughs are due to data distribution shifting over days.***



(a) Model-XS (norm. value) (b) Model-XL (norm. value)

Figure 17: ***For the same NE w.r.t. 1× model, AdaEmbed learns better per-feature embedding configuration using smaller size.***



(a) AE learns per-feature config. (b) AE gathers larger importance.

Figure 18: ***For the same size (0.5× model), AdaEmbed retains more important embeddings to achieve better NE (Model-XS).***

AdaEmbed introduces negligible (< 2%) QPS overhead across scales of the deployment (e.g., from 32 to 128 GPUs and 200 GB to 1.5 TB models), because (i) AdaEmbed largely parallelizes operations (e.g., asynchronous importance update and multi-threading); (ii) coordinator selectively initiates pruning rounds; and (iii) the memory manager introduces VHPI to avoid intense reallocation of the physical weight. Note that the memory overhead is ∼2% as AdaEmbed introduces only two small buffers (i.e., the lookup address and embedding importance) in VHPI lookup table (§4.3).

6.3 Performance Breakdown

We next break down AdaEmbed performance by time, the characteristics of sparse features, and design components.

Breakdown by Time Figure 15 breaks down model NE by time, with each data point on the line representing the moving average of the NE over hourly data (i.e., window NE regression). The training encompasses 14 days of data. We observe that with AdaEmbed, we can achieve consistently small NE regression than the baseline over time.

Moreover, we notice that this NE regression exhibits diurnal variation (e.g., in Model-XS and Model-M). This is because the data distribution (e.g., user preference) of recommendation tasks can change drastically over days. As such, at the beginning of training on a new day's data, the smaller model (e.g., 0.3× model) will experience a larger NE regression as it has less space to accommodate new embedding IDs. However, as the model gradually adapts to the new distribution, this regression tones down. We note that AdaEmbed experiences less NE fluctuation due to its ability to identify

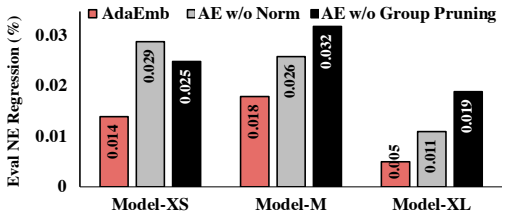
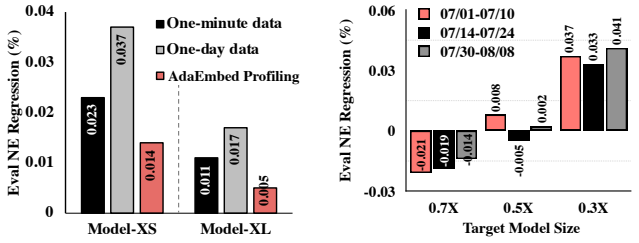


Figure 19: ***Performance breakdown of AdaEmbed (AE) design.***

and retain important embeddings on the fly.

Breakdown by Embedding Features We next investigate whether AdaEmbed can reduce manual efforts by learning to use better embedding configurations. First, in achieving the same NE as the 1× model, AdaEmbed learns to use smaller embeddings for many features (Figure 17). Moreover, using the same embedding size w.r.t. the 0.5× model, AdaEmbed gathers larger average embedding importance on each feature than the handcrafted setup (Figure 18), implying that more important embeddings are retained under the same total size. More importantly, we notice that (i) our group pruning shares similar preferences to the handcrafted configuration. Specifically, AdaEmbed tends to allocate more embeddings to those features that the model expert also values highly. However, (ii) some features are allocated fewer embeddings but AdaEmbed eventually achieves better NE, indicating that AdaEmbed can automatically find better embedding configurations.

Breakdown by Components We break down our design into two variants (i) (AdaEmbed w/o Norm): disable importance normalization in group pruning; and (ii) (AdaEmbed



(a) Impact of pruning interval. (b) Impact of dataset.

Figure 20: ***AdaEmbed achieves improvement across settings.***

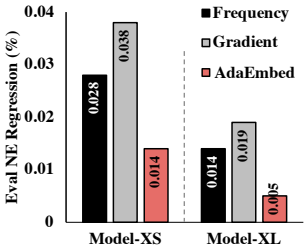


Figure 21: ***AdaEmbed outperforms importance alternatives.***

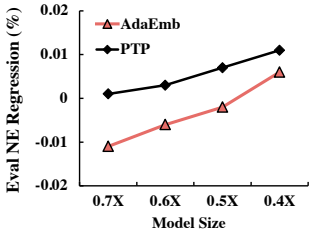


Figure 22: ***AE outperforms post-training pruning (PTP).***

w/o Group): completely disable group pruning, so the per-feature embedding size is resized to X% of the full model. We notice that both normalization and group pruning contribute to better NE (Figure 19). This is because (i) group pruning allows greater flexibility to resize the per-feature embedding using the shared gigantic weight table; and (ii) importance normalization helps to reduce the inter-feature heterogeneity by prioritizing important embeddings of each feature when comparing embedding importance globally.

6.4 Sensitivity and Ablation Studies

Impact of pruning frequency AdaEmbed Coordinator initiates a pruning round when the importance distribution radically changes. Next, we evaluate the impact of pruning frequency by deterministically enforcing pruning after training every-minute ($\sim$ 50 training iterations) and every-day data ($\sim$ 70K training iterations). Figure 20(a) reports that pruning too frequently and infrequently (i.e., pruning every one-minute and one-day data) both lead to suboptimal NE. The former is due to large training noise affecting instantaneous embedding importance, while the latter is due to AdaEmbed missing to admit important embeddings in a timely manner. Instead, the selective pruning of AdaEmbed achieves better performance by relying on the overall importance distribution at runtime.

Impact of different data Figure 20(b) reports the NE performance of model-S on three distinct datasets. Each training spans 10 days' training data, and we report the evaluation NE on the data of day 11. While the NE gain varies slightly as the data distribution varies across dates, AdaEmbed consistently achieves 50% memory savings with no NE regression.

Alternatives of embedding importance We next experiment with different embedding importance designs in training 10 days' data. Here, we consider using the frequency, gradient, and their combination (i.e., AdaEmbed design) as the embedding importance. We notice our frequency-gradient combination outperforms the alternatives. We note that this is consistent with the results of our Pearson analysis too, i.e., their combination has a stronger correlation to final embedding weights (Figure 8(b)). Instead, the access frequency and gradient only consider the data distribution and model characteristics, respectively, while DLRM accuracy depends on both aspects.

In-training vs. post-training pruning We compare AdaEmbed to its post-training pruning (PTP) counterpart like [20]. After model training is complete, PTP reduces the embedding size by pruning less important embeddings, as measured by our importance design. In fact, deploying PTP in real is often impractical (e.g., due to the need for online learning), and cannot achieve memory savings and/or QPS improvement during model training. Moreover, Figure 22 reports that AdaEmbed (i.e., in-training embedding pruning) can achieve better NE than PTP under the same embedding size, as the in-training design can adapt to the model performance at runtime and continuously optimize embeddings.

7 Related Work

Deep Learning Recommendation Systems Existing systems primarily focus on accelerating DLRM execution. NEO [18] co-optimizes embedding sharding and data parallelism. AIBox [52] and HierPS [51] overlap training execution on CPUs (using solid-state drives) and GPUs. Ekko [39] accelerates DLRM training over wide-area networks. TT-Rec [48] replaces embedding tables with matrix products to reduce memory footprints. Check-N-Run [13] reduces the bandwidth consumption for model checkpoints. Fleche [44] and Kraken [45] share the idea of sharing the weight table across features, but they focus on caching frequently accessed embeddings. AdaEmbed goes one step further by identifying the heterogeneous embedding importance to improve model accuracy during model training.

Optimizations for Deep Learning Recent ML advances have proposed various innovations for deep learning. TASO [23] and PET [41] perform tensor optimizations to improve model computation. Superneurons [42] and PipeSwitch [5] optimize instantaneous GPU memory by prefetching model layers based on their computation order. Similarly, ByteScheduler [37] and BytePS [24] accelerate the communication of distributed DNN training. ModelKeeper [28] warms up model training to reduce the amount of training execution needed. Egeria [43] adaptively freezes the training of model layers and bypasses their computation. These existing works focus primarily on conventional models, whereas DLRM models are often bottlenecked by memory-

intensive embeddings. Moreover, AdaEmbed is complementary to these efforts as AdaEmbed can further improve their optimized DLRM models.

Model Pruning Model pruning has been extensively studied to reduce model computation during training [11, 32], or to generate smaller models after training completes [8, 38]. Importance sampling [17, 29] performs weighted sampling on training data to achieve faster training convergence. Existing pruning systems and theories primarily focus on conventional CV and/or NLP counterparts by pruning only the dense layers [12, 20, 36]. However, in DLRMs, the gigantic embedding tables have become the bottleneck. This difference introduces novel challenges since the dense layers and embedding tables are distinct components with unique characteristics. For instance, dense layers are shared and accessed by all input samples, whereas each embedding row corresponds to a specific feature instance and is only accessed by it, leading to the heterogeneous importance of embeddings. Therefore, existing solutions are ill-suited for DLRMs.

8 Conclusion

This paper introduces AdaEmbed, an in-training embedding pruning system for better DLRM accuracy. AdaEmbed identifies embedding rows with larger importance to model accuracy, and then adaptively prunes less important embeddings to cap the total embedding size at scale. Our evaluations demonstrate that AdaEmbed can reduce manual efforts by automatically learning to use better per-feature embeddings, whereby it saves 35-60% embedding size needed in deployment, and achieves noticeable improvements on model accuracy and model execution speed.

Acknowledgments

We thank our shepherd, Deepak Narayanan, and the anonymous reviewers for their insightful feedback that significantly improved the final paper. This work was supported in part by NSF grants CNS-1909067, CNS-1900665, and CNS-2106184.

References

[1] HugeCTR: a high efficiency GPU framework designed for Click-Through-Rate (CTR) estimating training. https://developer.nvidia.com/nvidia-merlin/hugectr.

[2] TensorFlow. https://www.tensorflow.org/.

[3] TorchRec. https://github.com/pytorch/torchrec.

[4] Saurabh Agarwal, Ziyi Zhang, and Shivaram Venkataraman. Bagpipe: Accelerating deep recommendation model training. *arXiv preprint arXiv:2202.12429*, 2022.

[5] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. Pipeswitch: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 499–514. USENIX Association, November 2020.

[6] Brian R. Bartoldson, Ari S. Morcos, Adrian Barbu, and Gordon Erlebacher. The generalization-stability tradeoff in neural network pruning. In *NeurIPS*, 2020.

[7] Eric Breck, Marty Zinkevich, Neoklis Polyzotis, Steven Whang, and Sudip Roy. Data validation for machine learning. In *SysML*, 2019.

[8] Shih–Kang Chao, Zhanyu Wang, Yue Xing, and Guang Cheng. Directional pruning of deep neural networks. In *NeurIPS*, 2020.

[9] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. Wide & deep learning for recommender systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, DLRS 2016, page 7–10, New York, NY, USA, 2016. Association for Computing Machinery.

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

[11] Xiaocong Du, Bhargav Bhushanam, Jiecao Yu, Dhruv Choudhary, Tianxiang Gao, Sherman Wong, Louis Feng, Jongsoo Park, Yu Cao, and Arun Kejariwal. Alternate model growth and pruning for efficient training of recommendation systems. In *arxiv.org/abs/2105.01064*, 2021.

[12] Xiaocong Du, Bhargav Bhushanam, Jiecao Yu, Dhruv Choudhary, Tianxiang Gao, Sherman Wong, Louis Feng, Jongsoo Park, Yu Cao, and Arun Kejariwal. Alternate model growth and pruning for efficient training of recommendation systems. In *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2021.

[13] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. Check-n-run: a checkpointing system for training deep learning recommendation models. In *NSDI*, 2022.

[14] A.A. Ginart, Maxim Naumov, Dheevatsa Mudigere, Jiyan Yang, and James Zou. Mixed dimension embeddings with application to memory-efficient recommendation systems. In *ISIT*, 2021.

[15] Carlos A. Gomez-Uribe and Neil Hunt. The netflix recommender system: Algorithms, business value, and innovation. *ACM Trans. Manage. Inf. Syst.*, 6(4), December 2016.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. 2016.

[17] Siddharth Gopal. Adaptive sampling for sgd by exploiting side information. In *ICML*, 2016.

[18] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, and et al. The architectural implications of facebook's dnn-based personalized recommendation. In *HPCA*, 2020.

[19] Vipul Gupta, Dhruv Choudhary, Peter Tang, Xiaohan Wei, Xing Wang, Yuzhen Huang, Arun Kejariwal, Kannan Ramchandran, and Michael W. Mahoney. Training recommender systems at scale: Communication-efficient model and data parallelism. KDD, 2021.

[20] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. In *NIPS*, 2015.

[21] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, and et al. Applied machine learning at facebook: A datacenter infrastructure perspective. In *HPCA*, 2018.

[22] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, and Joaquin Quiñonero Candela. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, ADKDD'14. Association for Computing Machinery, 2014.

[23] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: Optimizing deep learning computation with automatic generation of graph substitutions. In *SOSP*, 2019.

[24] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous gpu/cpu clusters. In *OSDI*, 2020.

[25] Angelos Katharopoulos and François Fleuret. Not all samples are created equal: Deep learning with importance sampling. In *ICML*, 2018.

[26] Liu Ke, Udit Gupta, Carole-Jean Wu, Benjamin Youngjae Cho, Mark Hempstead, Brandon Reagen, Xuan Zhang, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, and Xiaodong Wang. Recnmp: Accelerating personalized recommendation with near-memory processing. In *ISCA*, 2020.

[27] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[28] Fan Lai, Yinwei Dai, Harsha V. Madhyastha, and Mosharaf Chowdhury. ModelKeeper: Accelerating dnn training via automated training warmup. In *NSDI*, 2023.

[29] Fan Lai, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. Oort: Efficient federated learning via guided participant selection. In *OSDI*, 2021.

[30] Xiangru Lian, Binhang Yuan, Xuefeng Zhu, Yulong Wang, Yongjun He, Honghuan Wu, Lei Sun, Haodong Lyu, Chengjun Liu, Xing Dong, et al. Persia: a hybrid system scaling deep learning based recommenders up to 100 trillion parameters. *arXiv preprint arXiv:2111.05897*, 2021.

[31] Rui Liu, Tianyi Wu, and Barzan Mozafari. Adam with bandit sampling for deep learning. In *NeurIPS*, 2020.

[32] Siyi Liu, Chen Gao, Yihong Chen, Depeng Jin, and Yong Li. Learnable embedding sizes for recommender systems. In *ICLR*, 2021.

[33] Ankur Mallick, Kevin Hsieh, Behnaz Arzani, and Gauri Joshi. Matchmaker: Data drift mitigation in machine learning for large-scale systems. *MLSys*, 2022.

[34] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, and et al. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *KDD*, 2021.

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

[36] Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning. In *ASPLOS*, 2020.

[37] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *SOSP*, 2019.

[38] Victor Sanh, Thomas Wolf, and Alexander M. Rush. Movement pruning: Adaptive sparsity by fine-tuning. In *NeurIPS*, 2020.

[39] Chijun Sima, Yao Fu, Man-Kit Sit, Liyi Guo, Xuri Gong, Feng Lin, Junyu Wu, Yongsheng Li, Haidong Rong, Pierre-Louis Aublin, and Luo Mai. Ekko: A Large-Scale deep learning recommender system with Low-Latency model update. In *OSDI*, 2022.

[40] Brent Smith and Greg Linden. Two decades of recommender systems at amazon.com. In *IEEE Internet Computing*, 2017.

[41] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *OSDI*, 2021.

[42] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic gpu memory management for training deep neural networks. In *PPoPP*, 2018.

[43] Yiding Wang, Decang Sun, Kai Chen, Fan Lai, and Mosharaf Chowdhury. Egeria: Efficient dnn training with knowledge-guided layer freezing. In *EuroSys*, 2023.

[44] Minhui Xie, Youyou Lu, Jiazhen Lin, Qing Wang, Jian Gao, Kai Ren, and Jiwu Shu. Fleche: An efficient gpu embedding cache for personalized recommendations. EuroSys, 2022.

[45] Minhui Xie, Kai Ren, Youyou Lu, Guangxu Yang, Qingxing Xu, Bihai Wu, Jiazhen Lin, Hongbo Ao, Wanhong Xu, and Jiwu Shu. Kraken: Memory-efficient continual learning for large-scale real-time recommendations. In *SC*, 2020.

[46] Li Yan, Choudhary Dhruv, Wei Xiaohan, Yuan Baichuan, Bhushanam Bhargav, Zhao Tuo, and Lan Guanghui. Frequency-aware sgd for efficient embedding learning with provable benefits. ICLR, 2022.

[47] Ying Yan, Liang Jeff Chen, and Zheng Zhang. Error-bounded sampling for analytics on big sparse data. 2014.

[48] Chunxing Yin, Bilge Acun, Carole-Jean Wu, and Xing Liu. TT-Rec: Tensor train compression for deep learning recommendation models. In *MLSys*, 2021.

[49] Zi Yin and Yuanyuan Shen. On the dimensionality of word embedding. NIPS'18, 2018.

[50] Chaoliang Zeng, Layong Luo, Qingsong Ning, Yaodong Han, Yuhang Jiang, Ding Tang, Zilong Wang, Kai Chen, and Chuanxiong Guo. FAERY: An FPGA-accelerated embedding-based retrieval system. In *OSDI*.

[51] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems. *MLSys*, 2020.

[52] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. AIBox: CTR prediction model training on a single node. CIKM, 2019.

[53] Guorui Zhou, Chengru Song, Xiaoqiang Zhu, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *KDD*, 2018.