PERSIA: An Open, Hybrid System Scaling Deep Learning-based Recommenders up to 100 Trillion Parameters

Xiangru Lian\textsuperscript{1}, Binhang Yuan\textsuperscript{3}, Xuefeng Zhu\textsuperscript{2}, Yulong Wang\textsuperscript{2}, Yongjun He\textsuperscript{3}, Honghuan Wu\textsuperscript{2}, Lei Sun\textsuperscript{2}, Haodong Lyu\textsuperscript{2}, Chengjun Liu\textsuperscript{2}, Xing Dong\textsuperscript{2}, Yiqiao Liao\textsuperscript{2}, Mingnan Luo\textsuperscript{2}, Congfei Zhang\textsuperscript{2}, Jingru Xie\textsuperscript{2}, Haonan Li\textsuperscript{2}, Lei Chen\textsuperscript{2}, Renjie Huang\textsuperscript{2}, Jianying Lin\textsuperscript{2}, Chengchun Shu\textsuperscript{2}, Xuezhong Qiu\textsuperscript{2}, Zhishan Liu\textsuperscript{2}, Dongying Kong\textsuperscript{2}, Lei Yuan\textsuperscript{2}, Hai Yu\textsuperscript{3}, Sen Yang\textsuperscript{2}, Ce Zhang\textsuperscript{3}, Ji Liu\textsuperscript{1}

\textsuperscript{1}Kwai Inc., USA; \textsuperscript{2}Kuaishou Technology, China; \textsuperscript{3}ETH Zürich, Switzerland;
\{firstname.lastname}@\{1.kwai.com;2.kuaishou.com;3.inf.ethz.ch\}

ABSTRACT

Deep learning based models have dominated the current landscape of production recommender systems. Furthermore, recent years have witnessed an exponential growth of the model scale—from Google’s 2016 model with 1 billion parameters to the latest Facebook’s model with 12 trillion parameters. Significant quality boost has come with each jump of the model capacity, which makes us believe the era of 100 trillion parameters is around the corner. However, the training of such models is challenging even within industrial scale data centers. This difficulty is inherited from the staggering heterogeneity of the training computation—the model’s embedding layer could include more than 99.90% of the total model size, which is extremely memory-intensive; while the rest neural network is increasingly computation-intensive. To support the training of such huge models, an efficient distributed training system is in urgent need. In this paper, we resolve this challenge by careful co-design of both the optimization algorithm and the distributed system architecture. Specifically, in order to ensure both the training efficiency and the training accuracy, we design a novel hybrid training algorithm, where the embedding layer and the dense neural network are handled by different synchronization mechanisms; then we build a system called PERSIA (short for parallel recommendation training system with hybrid acceleration) to support this hybrid training algorithm. Both theoretical demonstrations and empirical studies up to 100 trillion parameters have been conducted to justified the system design and implementation of PERSIA. We make PERSIA publicly available (at https://github.com/PersiaML/Persia) so that anyone would be able to easily train a recommender model at the scale of 100 trillion parameters.

1 INTRODUCTION

A recommender system is an important component of Internet services today. Tasks such as click-through rate (CTR) and buy-through rate (BTR) predictions are widely adopted in industrial applications, influencing the ad revenues at billions of dollar level for search engines such as Google, Bing and Baidu [78]. Moreover, 80% of movies watched on Netflix [30] and 60% of videos clicked on YouTube [25] are driven by automatic recommendations; over 40% of user engagement on Pinterest are powered by its Related Pins recommendation module [58]; over half of the Instagram community has visited recommendation based Instagram Explore to discover new content relevant to their interests [12]; up to 35% of Amazon’s revenue is driven by recommender systems [18, 104]. At

Figure 1: Model sizes of different recommender systems, among which only XDL and AIBox (via PaddlePaddle) are open-source. PERSIA is an open-source training system for deep learning-based recommender systems, which scales up models to the scale of 100 trillion parameters.

Kwai, we also observe that recommendation plays an important role for video sharing—more than 300 million of daily active users explore videos selected by recommender systems from billions of candidates.

Racing towards 100 trillion parameters. The continuing advancement of modern recommender models is often driven by the ever increasing model sizes—from Google’s 2016 model with 1 billion parameters [24] to Facebook’s latest model (2022) with 12 trillion parameters [62] (See Figure 1). Every jump in the model capacity has been bringing in significantly improvement on quality, and the era of 100 trillion parameters is just around the corner.

Interestingly, the increasing parameter comes mostly from the embedding layer which maps each entrance of an ID type feature (such as an user ID [50, 83] and a session ID [79, 85, 86]) into a fixed length low-dimensional embedding vector. Consider the billion-scale of entrances for the ID type features in a production recommender system (e.g., [28, 89]) and the wide utilization of feature crosses [23], the embedding layer usually dominates the parameter space, which makes this component extremely memory-intensive. On the other hand, these low-dimensional embedding vectors are concatenated with diversified Non-ID type features (e.g., image [95, 98], audio [87, 96], video [20, 46], social network [27, 33],...
etc.) to feed a group of increasingly sophisticated neural networks (e.g., convolution, LSTM, multi-head attention) for prediction(s) [24, 63, 107, 116, 121, 122]. Furthermore, in practice, multiple objectives can also be combined and optimized simultaneously for multiple tasks [48, 60, 93, 94, 118]. These mechanisms make the rest neural network increasingly computation-intensive.

**Challenges.** This paper is motivated by the challenges that we were facing when applying existing systems, e.g., XDL from Alibaba [38] and PaddlePaddle from Baidu [9], to the training of models at the 100 trillion parameter scale. This is a challenging task—as illustrated in Figure 1, most open source systems were only designed for a scale that is at least one order of magnitude smaller; even the largest proprietary systems such as the one reported by Facebook [62] just two month ago (September 2021) are still 8.3× smaller. The goal of this paper is to enable, to our best knowledge, the first open source system that is able to scale in this regime.

As 100 trillion parameters require at least 200TB to simply store the model (even in fp16), a distributed training system at this scale often consists of hundreds of machines. As so, the communication among workers is often a system bottleneck. While state-of-the-art systems [38, 45, 63, 66, 70, 100, 114, 115] often employ a carefully designed heterogeneous architecture (e.g., CPU/GPU, DRAM/SSD) to accommodate the heterogeneity in a recommender model (as illustrated in Figure 2), all of them are using a homogeneous training algorithm (either synchronous or asynchronous stochastic gradient based algorithms), for the model as a whole. At the scale of 100 trillion parameters and hundreds of workers, this homogeneity in the training algorithm starts to become a significant problem:

(1) Synchronous algorithms always use the up-to-date gradient to update the model to ensure the model accuracy. However, the overhead of communications for synchronous algorithms starts to become too expensive to scale out the training procedure, causing inefficiency in running time.

(2) While asynchronous algorithms have better hardware efficiency, it often leads to a “significant” loss in model accuracy at this scale—for production recommender systems (e.g., Baidu’s search engine [114]), Recall that even 0.1% drop of accuracy would lead to a noticeable loss in revenue—this is also consistent with our observation at Kwai.

Motivated by these challenges, we ask:

**Q1. Can we design an algorithm that can take benefits from both synchronous and asynchronous updates avoiding their disadvantages, to further scale up a recommender system with 100 trillion parameters?**

**Q2. How can we design, optimize, and implement a system to efficiently support such an algorithm?**

**Persia.** In this paper, we describe Persia, an open source distributed training system developed at Kwai to support models at the scale of 100 trillion parameters. As we will show in Section 6, this is only possible given its significant speed-up over state-of-the-art systems including both XDL and PaddlePaddle, achieved by the careful co-design of both the training algorithm and the training system. Open Source and Reproducibility: We also believe that the ability to train such a large model should be made easily and widely available to everyone instead of just being locked in the hands of a handful of largest companies. We make Persia open source so that everyone who has access to cloud computing service, e.g., Google cloud platform (where we perform Persia’s capacity test) can easily setup a distributed training system to reproduce our 100-trillion-parameter model test and to train their own models at this scale.

**Contributions.** Persia is enabled by a set of technical contributions. The core technical hypothesis of Persia is that, by using a hybrid and heterogeneous training algorithm, together with a heterogeneous system architecture design, we can further improve the performance of training recommender systems over state-of-the-arts.

Our first contribution is a natural, but novel hybrid training algorithm to tackle the embedding layer and dense neural network modules differently—the embedding layer is trained in an asynchronous fashion to improve the throughput of training samples, while the rest neural network is trained in a synchronous fashion to preserve the statistical efficiency. We also provide a rigorous theoretical analysis on its convergence behavior and further connect the characteristics of a recommender model to its convergence to justify its effectiveness.

This hybrid algorithm requires us to revisit some decisions with respect to the system architecture and optimizations, to unleash its full potential. Our second contribution is on the system side. We design a distributed system to manage the hybrid computation resources (CPUs and GPUs) to optimize the co-existence of asynchronicity and synchronicity in the training algorithm.

We further implement a wide range of system optimizations for memory management and communication optimization—such system optimizations are the key to fully unleash the potential of our hybrid training algorithm. We also develop a fault tolerance strategy to handle various potential failures during the training procedure.

**Last but not least,** we evaluate Persia using both publicly available benchmark tasks and real-world tasks at Kwai. We show that Persia scales out effectively and leads to up to 7.12× speedup.

---

1One can follow our detailed configuration of Persia over Google cloud platform at this link: https://github.com/PersiaML/tutorials/blob/main/src/kubernetes-integration/index.md.
compared to alternative state-of-the-art approaches [9, 38]. We further conduct larger-scale scalability experiments up to 100 trillion parameters on public clouds to ensure public reproducibility.

**Overview** The rest of the paper is organized as follows. We first provide some preliminaries for deep learning recommender systems in Section 2. We introduce our hybrid training algorithm in Section 3 and discuss **Persia** system design and implementation in Section 4. We show the theoretical analysis of the hybrid algorithm in Section 5, present the experimental results in Section 6, summarize related work in Section 7, and conclude in Section 8.

## 2 Preliminaries

This section introduces the basic principle of a typical deep learning based recommendation system (e.g., DLRM [62]) illustrated in Figure 2. We first formalize the distributed recommender training problem; and then give an anatomy of the existing architectures.

### 2.1 Problem Formalization

A typical recommender system takes the training sample in the form:

\[ x_\xi \in \mathbb{R}^{N_{ID}} \times \mathbb{R}^{N_{NID}} \times \mathbb{R}^{N_{Y}} \]

where \( \xi \) denotes the index of the sample in the whole dataset. \( x_{\xi}^{ID} := \{x_1, x_2, \ldots \} \) is the collection of ID type features in the sample, \( x_{\xi}^{NID} \) denotes the number of IDs, \( x_{\xi}^{Y} \) denotes the label. The ID type feature is the sparse encoding of large-scale categorical information. For example, one may use a group of unique integers to record the microvideos (e.g., noted as \( \langle \text{VideoIDs} \rangle \)) that have been viewed by a user; similar ID type features may include location (\( \langle \text{LocIDs} \rangle \)), relevant topics (\( \langle \text{TopicIDs} \rangle \)), followed video bloggers (\( \langle \text{BloggerIDs} \rangle \)), etc. In our formalization, \( x^{ID} \) is the collection of all ID type features—for the above example, it can be considered as:

\[ x^{ID} := \{\langle \text{VideoIDs} \rangle, \langle \text{LocIDs} \rangle, \langle \text{TopicIDs} \rangle, \langle \text{BloggerIDs} \rangle, \ldots \} \]

The Non-ID type feature \( x^{NID} \) can include various visual or audio features. And the label \( y \) may include one or multiple value(s) corresponding to one or multiple recommendation tasks(s).

The parameter \( w \) of the recommender system usually has two components:

\[ w := [w^{emb}, w^{nn}] \in \mathbb{R}^{N_{emb} + N_{nn}} \]

where \( w^{emb} \in \mathbb{R}^{N_{emb}} \) is the parameter of the embedding layer and \( w^{nn} \in \mathbb{R}^{N_{nn}} \) is the parameter of the rest dense neural network. We use: \( \text{lookup}_{w^{emb}}(x^{ID}) \) to denote the concatenation of all embedding vectors that has correspondence in \( x^{ID} \); \( NN_{w^{nn}}(\cdot) \) to denote a function parameterized by \( w^{nn} \) implemented by a deep neural network that takes the looked up embeddings and Non-ID features as input and generates the prediction. In the remaining part of this paper, “NN” is short for “neural network”. The recommender system predicts one or multiple values \( \hat{y} \) by:

\[ \hat{y}_\xi = NN_{w^{nn}}(\text{lookup}_{w^{emb}}(x_{\xi}^{ID}), x_{\xi}^{NID}) \]

It is worth mentioning that while the \( w^{nn} \) involved computation can be \( 10^7 \times \) more than the \( w^{emb} \) involved computation, the size of \( w^{emb} \) can be \( 10^7 \times \) larger than that of \( w^{nn} \), especially when \( w^{emb} \) contains many cross features. The imbalance between model scale and model computation intensity is one of the fundamental reasons why the recommender system needs an elaborate training system other than a general purpose deep learning system. Formally, the training system essentially solves the following optimization

\[
\min_w f(w) := \mathbb{E}_\xi [F(w; \xi)].
\]

if we use \( L \) to denote some loss function over the prediction and the true label(s) \( y \), \( F \) can be materialized as:

\[
F(w; \xi) := L(NN_{w^{nn}}(\text{lookup}_{w^{emb}}(x_{\xi}^{ID}), x_{\xi}^{NID}), y_\xi).
\]

Lastly, we use:

\[
\nabla_{w^{emb}} F := \nabla_{w^{emb}} F(w^{emb}, w^{nn}; \xi)
\]

\[
\nabla_{w^{nn}} := \nabla_{w^{nn}} F(w^{emb}, w^{nn}; \xi)
\]

to denote the gradients of \( w^{emb} \) and \( w^{nn} \) respectively.

### 2.2 Anatomy of Existing Architectures

Existing distributed systems for deep learning based recommender models are usually built on top of the parameter server (PS) framework [47], where one can add elastic distributed storage to hold the increasingly large amount of parameters of the embedding layer. On the other hand, the computation workload does not scale linearly with the increasing parameter scale of the embedding layer—in fact, with an efficient implementation, a lookup operation over a larger embedding table would introduce almost no additional computations. Thus, a fundamental goal of these systems is to resolve the inconsistency between the computation-intensive neural network and the memory-intensive embedding layer.

One natural solution is to deploy the training task over a hybrid infrastructure as an extension of the original PS framework: the large number of embedding parameters can be shard across multiple CPU PS nodes for storage and update, whilst the intensive FLOP computation can be assigned to GPU nodes. Thus, the optimization in these systems focuses on avoiding frequent communications of the memory-intensive embedding layer. Towards this end, different designs have been proposed. For example, Alibaba proposes XDL [38] that moves the computation of the embedding layer from the GPU worker to the CPU PS node so that only the embedding outputs (instead of all parameters) are transformed to the GPU workers along with the neural network parameters; Baidu designs a hierarchical PS framework [59] using a hierarchical storage to cache the frequently used parameter close to the GPU workers—only the infrequently used parameters would trigger the slow communication from SSD to GPU.

**Our Motivation.** Existing systems such as XDL and PaddlePaddle update the model in either the pure synchronous fashion or the pure asynchronous fashion. Both generally perform well on training small models. However, when training a large-scale model, the synchronous updating suffers from low hardware efficiency while the asynchronous updating usually incurs low accuracy (recall that a tiny loss in accuracy often means a huge loss in revenue
in the recommender system). Therefore, it motivates us to design a (sync-async) hybrid algorithm and an efficient system that is able to take benefits from both synchronous and asynchronous updates but avoid their disadvantages.

3 HYBRID TRAINING ALGORITHM

This section starts with explaining the intuition of the proposed hybrid algorithm, followed by the detailed algorithm description.

3.1 Why Hybrid Algorithm?

Training a recommendation mode requires the following essential steps in each iteration as shown in Figure 3-left:

- Preparation of embedding for the training sample(s);
- Forward propagation of the neural network;
- Backward propagation of the neural network;
- Synchronization of the parameters of the neural network;
- Update of embedding based on the corresponding gradients.

One illustration of the training workflow is shown in Figure 3-left. Note that a homogeneous view of the embedding layer and NN would set obstacles for either hardware or statistical efficiency. For example, consider the fully synchronous mode that we illustrate in Figure 3-right, the above five steps have to happen sequentially—it would be hard to speed up the training procedure based on this mechanism with low hardware efficiency.

Asynchronous distributed training. It is important to consider appropriate system relaxations for stochastic gradient-based optimizations deployed in a hybrid distributed runtime. There are two key observations from the ML community about asynchronous distributed training of neural networks:

- Asynchronous updating is efficient with sparse access. When individual updating (e.g., SGD iteration) only modifies a small portion of the model’s parameters, overwrites are rare and introduce barely no bias into the computation when they do occur [55, 65].
- Staleness limits the scalability and convergence of asynchronous SGD. On the other hand, when the updates are heavily overlapped (e.g., dense neural networks), the bias introduced by the discrepancy would limit SGD’s scalability and convergence [19, 52].

Without considering the statistical efficiency, the fully asynchronous mode (second row in Figure 3-right) would provide the optimal hardware efficiency for distributed recommender model training, where the time of preparing embeddings, synchronizing dense parameters, and updating embedding parameters can be hidden within the computation time for forward and backward propagations of the dense neural network. Unfortunately, as we will illustrate in Section 6.2, the asynchronicity would hurt the statistical efficiency and diminish the generalization performance, which is not acceptable for production recommender models [114].

Hybrid training algorithm. Based on these observations, one may consider to design different mechanisms for the embedding and dense neural network separately to optimize the training efficiency. For this purpose, we introduce a natural, but novel sync-async hybrid algorithm, where the embedding module trains in an asynchronous fashion while the dense neural network is updated synchronously. Considering the inherited heterogeneity of deep learning based recommender systems, this design would exploit the strength whilst avoid the weaknesses of asynchronous SGD from the algorithmic perspective. Briefly, as we illustrate in the third and forth rows in Figure 3-right, a naive hybrid mode is able to hide the steps of preparing embeddings, and updating embedding parameters within the synchronous training of the dense neural network; further, with some advanced system optimizations (e.g., overlapping computation and communication listed in Section 4.2.3), the synchronization of the dense parameters would also be able to mostly hidden within the backward propagation of the dense neural network. Thus, the hybrid algorithm would achieve almost similar hardware efficiency as the fully asynchronous mode without sacrificing the statistical efficiency.

3.2 Asynchronous updating for \( w_{\text{emb}} \)

To discuss the asynchronous algorithm to update embedding parameter \( w_{\text{emb}} \), given training sample indexed by \( \xi \), we introduce the following operations:

- \( \xi = x^{ID}_\xi \): fetch the subset of parameters to generate the embedding of the ID type feature \( x^{ID}_\xi \)—we use the notation \( w_{\text{emb}}^{\xi} \) to represent this subset of parameters activated by the sample.
- \( \xi = x^{ID}_\xi, f^{\text{emb}}_{\xi} \): communicate the gradient \( f^{\text{emb}}_{\xi} \) w.r.t \( x^{ID}_\xi \) to the storage of parameter \( w_{\text{emb}} \).

Algorithm 1 shows the asynchronous updating algorithm for the embedding layer \( w_{\text{emb}} \). Both forward computation task and backward computation task will be executed without any lock for synchronization: the forward computation task takes training samples, retrieves the embedding vectors, and sends the corresponding embedding vectors to the proceeding compute units (which handle the NN SGD computation); the backward computation task will receive the gradients of the embedding vectors, and sends it to the parameter storage for parameter updates.

---

\( \text{Algorithm 1 Asynchronous updating algorithm for } w_{\text{emb}}. \)

**Context:** Forward task for embedding layer.

**Input:** Embedding layer lookup (-), training data \( D \).

**while** Not converge do

/* Without any lock: */

- Select a sample from the training set: \( x^{ID}_\xi \sim D \).
- Get embedding vector(s) w.r.t the sample: \( w^{\xi}_{\text{emb}} \leftarrow \text{get}(x^{ID}_\xi) \).
- Send the embedding vector(s) to proceeding units: \( w^{\xi}_{\text{emb}} \downarrow \).

**end while**

**Context:** Backward task for embedding layer.

**Input:** Embedding layer lookup (-), optimizer \( t^{\text{emb}} \).

**while** Activation’s gradients keeps arriving do

/* Without any lock: */

- Receive gradient of embedding: \( F^{\xi}_{\text{emb}} \).
- Send gradients to parameter storage: \( \text{put}(x^{ID}_\xi, F^{\xi}_{\text{emb}}) \).
- Update embedding parameter: \( w_{t+1}^{\text{emb}} \leftarrow w_t^{\text{emb}} - \Omega^{\text{emb}} \left( F^{\xi}_{\text{emb}} \{ \xi \} \right) \).

**end while**

---

This category of optimization includes SGD, Adam[43], etc. We slightly abuse the term “SGD” to generally refer such optimizations for the rest of the paper.
Algorithm 2: Synchronous updating algorithm for $w_{\text{nn}}$.

**Context:** A task for dense module (indexed by $k$, $k = 1, 2, ..., K$).

**Input:** Neural network $NN_{w_{\text{nn}}}$ (.), embeddings $w_{\text{emb}}^\text{b,}(.)$, optimizer $\Omega_{\text{nn}}$.

**while** Not converge **do**

Randomly select $b$ buffered embeddings: $w_{\text{emb}}^\text{b,1, ... , b, l}^\text{b,}$.

**Forward of batch $B$:**

$F_B = \sum_{i=1}^{b} \left( NN_{w_{\text{nn}}} \left( w_{\text{emb}}^\text{b,} \sum_{\text{i}} \Omega_{\text{nn}}^\text{b,} \right), y_i \right)$;

/* With locks as synchronization barrier: */

**Backward of batch $B$: compute gradient $F_{B}^{\text{nnr}}, F_{B}^{\text{emb'}}$;**

Sync gradients with optimization: $F_{B}^{\text{nnr}} = 1/k \sum_{k=1}^{K} F_{B}^{\text{nnr}}$

**Update dense parameter:** $w_{\text{nn}}^{t+1} \leftarrow w_{\text{nn}}^{t} - \Omega_{\text{nn}} \left( F_{B}^{\text{nnr}} \right)$.

Send back activation’s gradients: $[F_{B}^{\text{emb'}}_1, F_{B}^{\text{emb'}}_2, ..., F_{B}^{\text{emb'}}_b] \uparrow$

**end while**

### 3.3 Synchronous updating for $w_{\text{nn}}$

Our training algorithm for the rest neural network looks similar to the standard distributed training of deep neural networks—the key difference is that our algorithm concatenates the embedding activations as part of input. The algorithm is illustrated in Algorithm 2. Notice that we adopt a mini-batch based SGD algorithm in contrast to the sample-based SGD in the asynchronous counterpart.

Since the hybrid algorithm is quite different from what has been assumed by existing systems [9, 38, 47, 59, 73]. To fully unleash its potential, we have to carefully design the system and optimize its performance—we introduce the system design and implementation of Persia in Section 4. On the other hand, it is also important to understand the statistical efficiency of the proposed hybrid training algorithm, thus we provide a theoretical analysis about the convergence guarantee of the hybrid algorithm in Section 5.

### 4 SYSTEM DESIGN AND IMPLEMENTATION

In this section, we first introduce the design of Persia to support the hybrid algorithm; then we discuss a wide range of implementations to optimize the computation and communication utilization.

#### 4.1 System Design

The system design includes two main fundamental aspects: i) the placement of the training workflow over a heterogeneous cluster, and ii) the corresponding training procedure over the hybrid infrastructure. To enlighten the system implementation, we also list the implementation goals of the Persia system here.

**Workflow placement over heterogeneous cluster.** To support the distributed training of deep learning based recommender model, a straightforward utilization of the PS paradigm (provided by general purpose deep learning frameworks, e.g., TensorFlow [11]) would place the storage and update of both embedding and NN parameters in a group of PS nodes (i.e., CPU machines) and the computation of forward and backward propagations in a group of worker nodes (i.e., GPU machines). However, this would be far from efficient and even impossible for deployment. For example, such recommender model would easily exceed the GPU RAM; and the uniform view of embedding and NN modules would introduce a large amount of unnecessary network traffic.

Some optimized PS architectures are proposed to optimize the training of deep learning recommender models by rearrange the functionalities in the PS paradigm. For example, XDL [38] designs the advanced model server to extend the original functionality of PS node to manage the learning (forward and backward propagations) of the embedding module. The hierarchical PS architecture proposed by Baidu [114] adopts a colocated PS framework with a sophisticated caching schema to reduce the communication overhead.

New challenges for the layout of 100-trillion-parameter models. In order to support the recommender models with one or two magnitude larger models, Persia should provide efficient autoscaling. Thus, we introduce the following modules, where each module
can be dynamically scaled for different model scales and desired training throughput:

- A data loader that fetches training data from distributed storages such as Hadoop, Kafka, etc;
- An embedding parameter server (embedding PS for short) that manages the storage and update of the parameters in the embedding layer \( w_{\text{emb}} \);
- A group of embedding workers that runs Algorithm 1 for getting the embedding parameters from the embedding PS; aggregating embedding vectors (potentially) and putting embedding gradients back to embedding PS;
- A group of NN workers that runs the forward-/backward-propagation of the neural network \( \text{NN}_{\text{nn}}(\cdot) \).

Considering the heterogeneity between the embedding and NN modules, Persia adopts different communication paradigms for training process: i) the PS paradigm between embedding PS and embedding workers (running on CPU nodes) to manage the training of the embedding layer, while ii) the AllReduce paradigm among NN workers (running on GPU nodes) for the NN.

**Distributed training procedure.** Logically, the training procedure is conducted by Persia in a data dispatching based paradigm as below (see Figure 4):

1. The data loader will dispatch the ID type feature \( x_{\text{ID}}^{\text{ID}} \) to an embedding worker—the embedding worker will generate a unique sample ID \( \xi \) for this sample, buffer this sample ID with the ID type feature \( x_{\text{ID}}^{\text{ID}} \) locally, and return this ID \( \xi \) back to the data loader; the data loader will associate this sample’s Non-ID type features and labels with this unique ID.
2. Next, the data loader will dispatch the Non-ID type feature and label \((x_{\text{NID}}^{\text{NID}}, y_{\xi})\) to a NN worker.

3. Once a NN worker receives this incomplete training sample, it will issue a request to pull the ID type features’ \( x_{\text{ID}}^{\text{ID}}(\xi) \) embedding \( w_{\text{emb}}^{\text{emb}}(\xi) \) from some embedding worker according to the sample ID \( \xi \)—this would trigger the forward propagation in Algorithm 1, where the embedding worker will use the buffered ID type feature \( x_{\text{ID}}^{\text{ID}}(\xi) \) to get the corresponding \( w_{\text{emb}}^{\text{emb}}(\xi) \) from the embedding PS.
4. Then the embedding worker performs some potential aggregation of original embedding vectors. When this computation finishes, the aggregated embedding vector \( w_{\text{emb}}^{\text{emb}}(\xi) \) will be transmitted to the NN worker that issues the pull request.
5. Once the NN worker gets a group of complete inputs for the dense module, it will create a mini-batch and conduct the training computation of the NN according to Algorithm 2. Note that the parameter of the NN always locates in the device RAM of the NN worker, where the NN workers synchronize the gradients by the AllReduce Paradigm.
6. When the iteration of Algorithm 2 is finished, the NN worker will send the gradients of the embedding \( (F_{\text{emb}}')(\xi) \) back to the embedding worker (also along with the sample ID \( \xi \)).
7. The embedding worker will query the buffered ID type feature \( x_{\text{ID}}^{\text{ID}}(\xi) \) according to the sample ID \( \xi \); compute gradients \( F_{\text{emb}}' \) of the embedding parameters and send the gradients to the embedding PS, so that the embedding PS can finally compute the updates according the embedding parameter’s gradients by its SGD optimizer and update the embedding parameters.

**System implementation goals.** To efficiently support the hybrid training algorithm, we listed the following system design goals for Persia:

**Fill the asynchronicity and synchronicity gap.** One central functionality of Persia system is to handle the heterogeneity inherited from the hybrid training algorithm. This would request Persia to seamlessly connect the forward- and backward-propagation during the training phase—the coordination of embeddings and gradients transmitted in a large-scale cluster is a unique challenge for Persia.

**Utilize the heterogeneous clusters efficiently.** To achieve the actual performance gain, Persia needs to include different mechanisms to fully utilize the computation resources (e.g., CPUs, GPUs) given diversified link connections. This demands an efficient memory management and a group of optimized communication mechanisms.

**Provide effective fault tolerance.** With the large number of machines that training requires, effective fault tolerance is necessary. Furthermore, the hybrid algorithm would request more complex mechanisms to manage the heterogeneous computation and communication, which poses additional challenges for fault tolerance.

4.2 System Optimizations and Design Decisions

We enumerate the implementation details to fulfill the design goals.

4.2.1 Fill the Async/Sync Gap. To fill the gap between synchronous and asynchronous updates, both embedding and NN workers implement some buffering mechanisms.

**NN worker buffer mechanism.** Since we adopt a GPU-pull based schema for the hybrid training procedure between the NN

---

**Figure 4: The architecture of Persia.** Persia includes a data loader module, an embedding PS module, a group of embedding workers over CPU nodes, and a group of NN workers over GPU instances.
worker and the embedding worker, each NN worker will locally maintain an input sample hash-map keyed on the sample ID $\xi$ and valued on tuples of Non-ID type feature $x_{\xi}^{\text{NID}}$ and label $y_{\xi}$. In the forward propagation, once a NN worker receives the Non-ID type feature and label, it will first insert the key-value pair $(\text{key} : \xi, \text{value} : (x_{\xi}^{\text{NID}}, y_{\xi}))$ to the input sample hash-map; and then send the request of the embedding vector to the embedding worker. Later, when the embedding vector $w_{\xi}^{\text{emb}}$ arrives in the NN worker from an embedding worker, the NN worker will pop the key-value pair $(\text{key} : \xi, \text{value} : (x_{\xi}^{\text{NID}}, y_{\xi}))$ from the input sample hash-map, consume $w_{\xi}^{\text{emb}}, x_{\xi}^{\text{NID}}, y_{\xi}$ in the mini-batch for the SGD computation within the GPU. In the backward propagation, once the computation is done, the NN worker will use the sample ID $\xi$ to locate the embedding worker, and then send the gradient of the embedding $F_{\xi}$ to this embedding worker.

Embedding worker buffering mechanism. To support the pull request from NN worker, an embedding worker also needs to locally maintain a ID type feature hash-map that is keyed on sample ID $\xi$ and valued on ID type feature $x_{\xi}^{D}$. In the forward propagation, when the embedding worker receives a new ID type feature, it will generate a unique ID for this sample, store the key-value pair $(\text{key} : \xi, \text{value} : x_{\xi}^{D})$ in the ID type feature hash-map. Then the embedding worker will get the corresponding embedding parameter $w_{\xi}^{\text{emb}}$ from embedding PS and return it to the NN worker. In the backward propagation, once the embedding worker receives the gradient of the embedding vector $F_{\xi}$ from a NN worker, the embedding worker will find the corresponding ID type feature $x_{\xi}^{D}$. Note that this is implemented as a local search in the ID type feature hash-map—the sample ID $\xi$ serves as the key to retrieve $x_{\xi}^{D}$. Later, the embedding worker can put the gradient of the embedding parameters back to the embedding PS.

4.2.2 Persia Memory Management. Memory management is an important component for Persia to efficiently utilize the hybrid infrastructure, especially for the embedding PS which is responsible for maintaining trillions of embedding parameters. Comprehensive, the embedding PS works like a standard PS, which is not significantly different from a distributed key-value store. As we illustrated in Figure 5, when retrieving the embedding parameters, an embedding worker first runs an identical global hashing function to locate the embedding PS node that stores the parameters; once the request arrives in the PS node, the parameter can be acquired in an LRU cache as we explain below.

LRU cache implementation. Persia leverages LRU cache\footnote{Standard LRU implementation is based on hash-map and doubly linked list, e.g., [5].} to maintain the embedding parameters in RAM. We use array-list and hash-map to implement the LRU. Instead of a doubly linked list where the pointer stores a memory address, we adopt an array-list design where the pointer stores the index of the pre- or post-entrace in the array; similarly, the hash-map’s value also stores the corresponding embedding parameter’s index in the array instead of the memory address. Besides the pre- and post- indices, each item in the array also includes two fields: the embedding vector and the optimizer state.

Figure 5: Memory management of Persia’s embedding PS node. The design is based on a LRU cache implemented by hash-map and array-list.

and the optimizer states corresponding to this embedding vector. There are two advantages by utilizing an array-based linked list: i) this mechanism avoids frequent allocation and deallocation of fragmented memory blocks—this cost is not negligible when each linked list may contain billions of entries; ii) since pointer (that stores the memory address) does not exist in the data structure, serialization and deserialization become straightforward memory copy, which is far more convenient and efficient—this is helpful for periodic saving and loading checkpoints for fault tolerance and model sharing. Note that to fully utilize the CPU cores and RAM in the embedding PS node, we utilize multiple threads in the LRU implementation. Each thread manages a subset of the local hash-map and the corresponding array-list; when there is a request of get or put, the corresponding thread will lock its hash-map and array-list until the execution is completed.

4.2.3 Communication Optimization. To fully utilize the computation power in the heterogeneous cluster, Persia implements a range of system optimizations to decrease the communication overhead.

Optimized communication among NN workers. The optimization of AllReduce communication paradigm among NN workers in Persia is the key for hiding communication overhead within the backward computation of the neural network. This functionality is implemented based on Bagua [29], an open-source general-purpose distributed learning system optimized for data parallelism, also released by Kwai. Currently, Persia utilizes Bagua’s centralized synchronous full-precision communication primitive (equivalent to AllReduce) by default, in an attempt to preserve the accuracy. Persia leverages Bagua for the synchronization among NN workers because additional system optimization enabled by Bagua can be directly adopted, including tensor bucketing, memory flattening, hierarchical communications, etc. Notice that there are other communication primitives provided by Bagua which could potentially further improve the training throughput; however, it is unclear if these communication primitives would hurt the statistical efficiency, we leave this exploration as an interesting future work.

Optimized remote procedure call. The point-to-point communication: i) between NN workers and embedding workers, and ii) between embedding workers and embedding PS is implemented by remote procedure call (RPC). Unlike the traditional usage of RPC, where the communication is mainly responsible for transmitting small objects with complex serialization and deserialization mechanism, Persia demands a RPC implementation that is efficient for communicating tensors stored in a large continuous memory space.
As so, Persia abandons the protocol buffer based implementation (adopted by gRPC [4] and bRPC [2] for other learning systems), which introduces significant overhead for communicating tensors; instead, Persia adopts a simple but efficient zero-copy serialization and deserialization mechanism targeting for tensors which directly uses memory layout to serialize and deserialize tensors. Further, as we mentioned in Section 4.2.2, since tensors on host RAM are allocated in large pages, the TLB lookup time is also reduced significantly, which accelerates the copying procedure from host RAM to the RAM of the network adapter.

**Workload balance of embedding PS.** The embedding parameter storage is implemented as a sharded PS to support the query and update of the parameter in the embedding layer. The key challenge in embedding PS implementation is about the workload balance of query and update about the embedding parameter. Initially, we adopt a straightforward design by distributing the embedding parameter according to the feature groups. A sub-group of CPU instances are allocated to manage a partition of semantic independent embbeddings. We find that in practice, this would easily lead to congestion in the access of some feature groups during training—the access of training data can irregularly lean towards a particular embedding during group during the online learning procedure in industrial-scale applications. We solve this issue by adopting an alternative partition of the embedding parameter: the embeddings inside a feature group are first uniformly shuffled and then evenly distributed across embedding PS nodes. We observe that this design effectively diminishes the congestion of the embedding parameter access and keeps a balanced workload for the embedding PS.

**Communication compression.** To reduce the network traffic, we adopt both lossless and lossy compression mechanisms for the communication request between embedding and NN workers. The network bandwidth connecting GPU instances is limited—besides the AllReduce operation that leverages this bandwidth, another noticeable utility is the communication of embedding vectors in the forward propagation (4) in Figure 4) and its gradient in the backward propagation (6) in Figure 4). The communication between GPUs enabled by advanced connections like GPUDirect RDMA is 10x faster than that between GPU and CPU nodes through PCI-e [7], which could leave the communication of embedding activation and its gradients a bottleneck. Note that although plenty of lossy compression schema has been proposed for distributed learning, one should be cautious of applying them in distributed recommender model training—for a commercial recommender system, as mentioned before, even a drop of 0.1% accuracy is not affordable [114]. To reduce such network traffic, we apply a lossless compression mechanism for the index component and a discreet lossy compression mechanism for the value component.

**Lossless compression.** For the index component of the embedding, instead of representing a batch of samples as a list of vectors, where each vector containing all IDs (represented by int64) of a sample, we represent a batch as a hash-map, where the key is unique IDs in the whole batch, and the value corresponding to each unique ID is the indices of the samples in the batch containing this ID. Since the batch size is relatively small (\( \leq 65535 \)), the indices can be represented using uint16 instead of int64 without losing any information.

**Lossy compression.** For the value component, We adopt a discreet fp32 to fp16 compression. Notice that a uniform mapping from fp32 to fp16 would harm the statistic efficiency significantly, so we defines an nonuniform mapping method: suppose \( \parallel \cdot \parallel_\infty \) represents the \( \infty \) norm of a vector, \( \kappa \) represents a relatively large constant scalar. In the compression side, each fp32 vector block \( v \) is first scaled by \( \frac{\kappa}{\|v\|_\infty} \) and then converted to a fp16 block vector. In the decompression side, the compressed block vector \( \hat{v} \) communicated as fp16 is first converted back to a fp32 vector and then divided by \( \frac{\kappa}{\|v\|_\infty} \).

4.2.4 Fault Tolerance. Failure can happen frequently considering the large number of nodes that participate in the hybrid training. Towards this end, Persia implements a group of fault tolerance mechanisms to handle different failures in the cluster. Two observations are interesting:

- The infrequent loss of parameter update of the embedding layer is usually negligible for convergence, while the responsive time of the embedding parameter query would be important for the end-to-end training time;
- Any drop of the model synchronization by the NN worker that runs the dense synchronous training algorithm is vital for convergence. Based on these observations, Persia implements the following mechanisms for different components to handle different system failures.

Since the data loader should be able to run on top of any other popular distributed storage systems (e.g., HDFS), Persia relies on their own recovery schema once an instance failure happens. Notice that Persia mainly considers the online training setting, where no shuffling schema is required by Persia’s data loader.

The embedding PS should be responsive during the hybrid training execution. For this purpose, the embedding PS node will put in-memory LRU cache (introduced in Section 4.2.2) in a shared memory space—by this fashion, once a process-level failure happens, the process can automatically restart and attach to the consistent shared-memory space without influencing any other instances of the embedding PS. Additionally, embedding PS nodes will periodically save the in-memory copy of the embedding parameter shard, with the advance of our LRU implementation, check-pointing is very efficient (also see Section 4.2.2).

The embedding worker has no fault recovery schema—once a failure happens, the local buffer of the ID type feature hash-map will be simply abandoned without any recovery attempts.

By contrast, the dense module cannot afford any drop of model synchronization. As so, the NN worker would also periodically save the synchronized model as the checkpoint with the same frequency as the embedding PS. Once a failure of GPU instances happens, all the GPU instances will abandon their local copy of the model, load the latest checkpoint, and continue the execution of the dense synchronous training algorithm.
5 THEORETICAL ANALYSIS

In this section, we show the convergence rate of the proposed hybrid algorithm adopted by Persia. In short, the proposed hybrid algorithm converges and admits a similar convergence rate (or total complexity equivalently) to the standard synchronous algorithm. That is to say that the hybrid algorithm takes a significant benefit from asynchronicity in the system efficiency but sacrifices very little for the convergence rate.

To show the detailed convergence rate, we first provide the essential updating rule of the hybrid algorithm in Persia:

\[
\begin{align*}
    w_{t+1}^{\text{emb}} &= w_{t}^{\text{emb}} - T \bar{F}_t^{\text{emb}}, \\
    w_{t+1}^{\text{nn}} &= w_{t}^{\text{nn}} - T \bar{F}_t^{\text{nn}},
\end{align*}
\]

where \( F_t^{\text{emb}} \) and \( F_t^{\text{nn}} \) are notations for gradients of \( w_t^{\text{emb}} \) and \( w_t^{\text{nn}} \), short for respectively:

\[
F_t^{\text{emb}} := \nabla \bar{w}_t^{\text{emb}} F_t(D(t), w_t^{\text{nn}}, \bar{w}_t),
\]

\[
F_t^{\text{nn}} := \nabla \bar{w}_t^{\text{nn}} F_t(D(t), w_t^{\text{nn}}, \bar{w}_t),
\]

where \( D(t) \leq t \) denotes some iterate earlier than the current iterate \( t \). This is because of the asynchronous update for the embedding layer. Since the dense neural network adopts the synchronous update, it always uses the up to date value of \( w_t^{\text{nn}} \), while the used value of the embedding layer may be from some early iterate. This is the cost of using asynchronous updating, but we will see that the cost is very minor comparing to our gain in system efficiency.

To ensure the convergence rate, we still need to make some commonly used assumptions as follows:

**Assumption 1.** We make the following commonly used assumptions for analyzing the stochastic algorithm:

- **Existence of global minimum.** Assume \( f^* := \min_w f(w) \) exists.

- **Lipschitzian gradient.** The stochastic gradient function \( F_t(\cdot; \bar{w}_t) \) is differentiable, and \( L \)-Lipschitzian for all \( \bar{w}_t \):

\[
\| F_t'(w; \bar{w}_t) - F_t'(\bar{w}; \bar{w}_t) \| \leq L \| w - \bar{w} \|, \\
\forall \bar{w}_t \in \mathbb{R}^n, \forall w \in \mathbb{R}^{N_{\text{emb}} + N_{\text{nn}}}. (3)
\]

- **Bounded variance.** The variance of the stochastic gradient is bounded: there exists a constant \( \sigma > 0 \) such that

\[
\mathbb{E} \| F'(w; \bar{w}_t) - F'(\bar{w}; \bar{w}_t) \|^2 \leq \sigma^2 \quad \forall w. (4)
\]

- **Bounded staleness.** All sparse model update delays are bounded: there exists a constant \( \tau \geq 0 \) such that

\[
t - D(t) \leq \tau, \quad \forall t. (5)
\]

In practice \( \tau \) is the number of samples whose embeddings are retrieved but their gradients are not yet updated into the model parameters. In Persia this value is less than 5 for most cases.

**Remark 1.** The assumptions of existence, bounded variance, and bounded staleness are commonly used ones. The bounded staleness assumption is due to the asynchronous update for \( w^{\text{emb}} \). In practice, \( t - D(t) \leq \tau \) means that any element in \( w^{\text{emb}} \) is updated up to \( \tau \) times between reading it and writing it within training the same sample. In Persia this value of \( \tau \) is less than 5 for most scenarios. The frequency assumption reflects the intrinsic property of the recommendation system. For each sample \( x, \bar{x} \) if every ID type feature \( x \) has the same probability of being in \( \bar{x}_t^D \), then \( \alpha = \frac{1}{N_{\text{emb}}} \). On the other hand, if an ID type feature \( x \) is contained by every sample \( \bar{x}_t^\prime \)’s ID type features \( x_t^D \), then \( \alpha = 1 \).

Then we obtain the following convergence result:

**Theorem 1.** Denote \( \alpha \) to be the constant such that for each ID \( \bar{x} \) the probability of a sample containing it is smaller than \( \alpha \). Under Assumption 1, with learning rate \( T \) in (2) as

\[
Y = \frac{1}{L + \sqrt{T \alpha} + 4 \alpha L \alpha}
\]

Persia admits the following convergence rate:

\[
\frac{\sum_{t=0}^{T-1} \mathbb{E} \| f'(w_t) \|^2}{T} \leq \sigma^2 \frac{1}{\sqrt{T} + \frac{1}{T}} \frac{\alpha \sigma}{T}
\]

where \( \leq \) means “smaller than or equal to up to a constant factor”, and \( L, G \) are all treated as constants for simplicity. For example \( a_t \leq b_t \) means that there exists a constant \( C > 0 \) such that \( a_t \leq C b_t \) for all \( t \).

The complete proof is provided in the preprint version of this paper [53]. Note that the first two terms in (6) are exactly the convergence rate of vanilla SGD [59]. The third (additional) term is caused by the staleness of the asynchronous update in the embedding layer of the model. The staleness upper bound \( \tau \) is typically proportional to the number of workers. The ID frequency upper bound \( \alpha \) is a value smaller than 1. Notice that \( \alpha = 1 \) leads a convergence rate exactly matching the asynchronous SGD [59]. It indicates that Persia (the hybrid algorithm) guarantees to be no worse than the asynchronous algorithm in convergence rate. When \( \alpha \ll 1 \), which is mostly always true in real recommendation systems, the hybrid algorithm—Persia—admits a very similar convergence rate to the synchronous algorithm, since the third term is dominated by the second term. Therefore, this theorem suggests that the hybrid algorithm follows the same convergence efficiency in terms of iterations.

6 EXPERIMENTS

In this section, we introduce experiments to evaluate the design and implementation of Persia, focusing on the following questions:

- Can Persia provide significant boost of end-to-end training time comparing with the state-of-the-art systems?
- Is the hybrid training algorithm statistically efficient in terms of convergence w.r.t training iterations?
- Can Persia provide high training throughput and scale out well especially for trillion-parameter scale models?

**Benchmark.** We evaluate Persia over three open-source benchmarks and one real-world production microvideo recommendation workflow at Kwai:

- Taobao-Ad (open source [10]): predict the advertisement CTR from Taobao’s website for 8 days with 26 million records.
We appreciate Google’s great support for coordinating the computation resources for this experiment.

6.1 End-to-end Evaluation

We first compare the end-to-end training time that each system needs to achieve a given AUC, using the three open source benchmarks over a heterogeneous cluster with 8 GPU workers. We report the results of both synchronous and asynchronous modes of XDL, and conduct a careful trial of different modes in PaddlePaddle and report the result under the optimal setting. Figure 6 illustrates significant performance improvements from Persia (e.g., for the Taobao-Ad benchmark, Persia is 17.1× faster than that of the synchronous and asynchronous modes of XDL, and 1.71× faster than PaddlePaddle)—same level of speedup also appears in the Avazu-Ad and Criteo-Ad benchmarks.

On our production Kwai-Video dataset, XDL fails to run the task to convergence in 100 hours—we compare the training sample throughput and find that Persia’s throughput is 19.31× of XDL. For PaddlePaddle, we fail to run the training task since PaddlePaddle does not support some deep learning operators (e.g., batch normalization) required in our model.

6.2 Evaluation of Hybrid Algorithm

The hybrid algorithm is one of the main reasons behind Persia’s scalability. Here, we evaluate its convergence behavior and leave it scalability and performance to the next section. Figure 7 illustrates the convergence behaviors of different systems. We see that the hybrid algorithm shows almost identical convergence when comparing with the fully synchronous mode, converging to comparable AUC as illustrated in Table 2 (but as we will see in the next section, the hybrid algorithm is much faster compared with the fully synchronous algorithm). We see that test AUC gap between the hybrid mode and synchronous mode is always less than 0.1% in the three open-source benchmarks, and less than 0.001% in the

Baseline systems. We consider two state-of-the-art baselines: XDL [38] and PaddlePaddle [9]—XDL is a specialized distributed recommendation framework developed by Alibaba; PaddlePaddle is a general purpose deep learning framework from Baidu with a special Heter mode that implements the design of ABox [115]. Baidu’s recommendation training system, according to private communications we had with members of the PaddlePaddle development community.

Cluster setup. We conduct experiments on two clusters. Most of the training is conducted over heterogeneous clusters inside Kwai’s production data center—for Persia, we include up to 64 Nvidia V100 GPUs, and 100 CPU instances (each with 52 cores and 480GB RAM). The instances are connected by a network with the bandwidth of 100 Gbps. The baseline systems (XDL and PaddlePaddle) are equipped with the same amount of computation resources for each individual setting.

We further conduct larger-scale scalability experiments over Google cloud platform, for both capacity and public reproducibility reasons with a heterogeneous cluster including:

- 8 a2-highgpu-8g instances (each with 8 Nvidia A100 GPUs) as NN workers;
- 100 c2-standard-30 instances (each with 30vCPUs, 120GB RAM) as embedding workers;
- 30 m2-ultramem-416 instances (each with 416vCPUs, 12TB RAM) as embedding PS.

Table 1: Model scales for the benchmarks.

| Benchmark      | Sparse # parameter | Dense # parameter |
|----------------|--------------------|-------------------|
| Taobao-Ad      | 29 Million         | 12 Million        |
| Avazu-Ad       | 134 Million        | 12 Million        |
| Criteo-Ad      | 540 Million        | 12 Million        |
| Kwai-Video     | 2 Trillion         | 34 Million        |
| Criteo-Syn\(_1\) | 6.25 Trillion      | 12 Million        |
| Criteo-Syn\(_2\) | 12.5 Trillion      | 12 Million        |
| Criteo-Syn\(_3\) | 25 Trillion        | 12 Million        |
| Criteo-Syn\(_4\) | 50 Trillion        | 12 Million        |
| Criteo-Syn\(_5\) | 100 Trillion       | 12 Million        |

Figure 6: End-to-end training performance of four benchmarks: Taobao-Ad, Avazu-Ad, Criteo-Ad, Kwai-Video (from left to right).
production Kwai-video benchmark; by contrast, the gap between the asynchronous mode and the synchronous mode is much higher (from 0.5% to 1.0%); further, as we allow more aggressive asynchronicity in PaddlePaddle, the gap is more significant. We want to emphasize that even 0.1% decrease of accuracy for the production recommender model might lead to a loss of revenue at the scale of a million dollar—this is NOT acceptable.

### 6.3 Scalability and Capacity

We first evaluate the scalability of Persia over clusters with different scales of GPUs in terms of training sample throughput, as illustrated in Figure 8. We see that Persia, with a hybrid algorithm, achieves much higher throughput compared to all other systems. On three open source benchmarks, Persia reaches nearly linear speedup with significantly higher throughput comparing with XDL and PaddlePaddle. For the Kwai-video benchmark, Persia achieves 3.8× higher throughput compared with the fully synchronous algorithm. Note that the fully asynchronous algorithm can achieve faster throughput than the hybrid algorithm; however, as illustrated in the previous section, it incurs a lower AUC compared with the hybrid algorithm.

We further investigate the capacity of Persia for training different scales of trillion-parameter models in Google cloud platform. Figure 9-left illustrates that the throughput of Persia when varying the number of entrances in the embedding table and fixing the embedding output dimension to 128—this is corresponding to the model scale of Criteo-Syn in Table 1. We see that Persia shows stable training throughput when increasing the model size even up to 100 trillion parameters. This can be viewed as concrete evidences that Persia has the capacity to scale out to the largest recommender model that is never reported before. Figure 9-right shows that for the 100 trillion-parameter model, Persia also achieves 2.6× higher throughput than the fully synchronous mode; on the other hand, asynchronous mode introduces further speedup (1.2× faster than that of the hybrid algorithm)—this generally would introduce statistical inefficiency for convergence, which is not revealed by the capacity evaluation.

| Benchmark   | Persia-Hybrid | Sync | Async | PaddlePaddle |
|-------------|---------------|------|-------|--------------|
| Taobao-Ad   | 65.86%        | 65.93% (XDL) | 65.66% (XDL) | 59.14%       |
| Avazu-Ad    | 77.93%        | 78.02% (XDL) | 77.10% (XDL) | 73.19%       |
| Criteo-Ad   | 80.47%        | 80.51% (XDL) | 80.01% (XDL) | 77.08%       |
| Kwai-Video  | 72.63%        | 72.63%       | 72.16%       | -            |

Table 2: Final test AUC of four benchmarks.

### 7 RELATED WORK

We briefly discuss the relevant recommender systems and system relaxations for distributed large scale learning. Detailed discussion can be found in more comprehensive surveys (e.g., [110] for recommender system, and [59] for distributed deep learning).

#### 7.1 Recommender Systems

Recommender systems are critical tools to enhance user experience and promote sales and services [110], where deep learning approaches have shown great advance recently and been deployed ubiquitously by tech companies such as Alibaba [121], Amazon [70], Baidu [114], Facebook [63], Google [24, 118], Kuaishou [103], Pinterest [58], Netflix [30], etc. Generally, deep learning based approaches take the sparse ID type features to generate dense vectors through an embedding layer, and then feed the embedding activations into the proceeding neural components to expose the correlation and interaction. Diversified features are included to power recommender systems [111], such as sequential information (e.g., session based user id [79, 85, 86]), text [109, 117, 125], image [95, 98], audio [87, 96], video [20, 46], social network [27, 33], etc. Additionally, various neural components have been explored for recommender systems, including multi-layer perceptron (MLP) [105, 106], convolutional neural network (CNN) [72, 85], recurrent neural network (RNN) [101, 102], Autoencoder [71, 124], and deep reinforcement learning [117, 119]. Furthermore, multiple relevant goals for recommendation can be learned simultaneously by multi-task learning [24, 63, 107, 116, 121, 122].

Due to the massive scale of both training data and recommender models, distributed learning techniques are applied to train industrial-scale recommender systems [63, 70, 114, 118, 121]. The heterogeneity in computation naturally leads to the idea of utilizing heterogeneous distributed computation resources to handle such training procedure [38, 45, 61, 63, 66, 70, 100, 114, 115]. For example, XDL [38] adopts the advanced model server to replace classic parameter server to reduce the traffic between CPU nodes and GPU nodes; Baidu proposes a hierarchical PS architecture [114] to implement a sophisticated caching schema to reduce the communication overhead; HET [61] adopts an advanced cache mechanism at the GPU worker side to leverage the skewness of embeddings by caching.
frequently updated entrances in the worker’s limited local memory. However, as far as we know, there is a lack of a carefully designed distributed training solution to support the heterogeneity inherited from the training data (sparse vs. dense) and gradient decent based optimization (memory bounded vs. computation bounded) over mixed hardwares and infrastructures (CPU instances vs. GPU instances).

### 7.2 Distributed Deep Learning

Distributed deep learning can be categorized as data parallelism and model parallelism. In **data parallel** distributed deep learning, two main categories of systems are designed—parameter server [26, 35, 40, 47, 112] and AllReduce [21, 39, 73, 113]. In a PS architecture, models are stored in a single node or distributively in multiple nodes; during the training phase, workers periodically fetch the model from PS, conduct forward/backward propagation, and push the gradients to the PS, while the PS aggregates the gradients and updates the parameters. With an AllReduce paradigm, all workers collaborate with their neighbors for model/gradient exchanges. Different strategies are proposed to speed up the expensive parameter/gradient exchange phase. To reduce communication volumes, lossy communication compression methods are introduced, such as quantization [13, 16, 99, 108], sparsification [14, 88, 91, 97], sketching [36], and error compensation [82]. In an attempt to get rid of the latency bottleneck, decentralized communication approaches are proposed [44, 49, 54, 56, 81]. Additionally, local SGD is discussed to optimize for the number of communication rounds during training [31, 57, 77, 90]. To remove the synchronization barrier, asynchronous update methods are proposed [52, 64, 68, 76, 120, 123]. There are also approaches that combines multiple strategies listed above [15, 17, 44, 56, 80]. On the other hand, researches about **model parallelism** attempt to study how to allocate model parameters and training computation across compute units in a cluster to maximize training throughput and minimize communication overheads. Optimizations are proposed for both operation partitioning approach [37, 74, 75, 92] and pipeline parallel approach [32, 34, 84]. Recently, there are also approaches that combine both data and model parallelism [51, 67, 70].

General purpose deep learning systems have been one of the main driving forces behind the rapid advancement of machine learning techniques with the increasing scalability and performance for distributed deep learning. Popular options include TensorFlow [11], PyTorch [8], MXNet [22], PaddlePaddle [9], MindSpore [6], etc. Extensions and modifications have been made based on these general purpose learning systems for efficient distributed learning (e.g., Horovod [73], BytePS [41], Bagua [29], Megatron [75], ZeRO [69], SageMaker [42], etc.). However, even including these extensions, the current general purpose deep learning systems do not consider the challenges about handling the heterogeneity over a hybrid infrastructure. Thus, it is difficult to directly use these general purpose learning systems for the training tasks of industrial-scale recommender systems over a heterogeneous cluster.

### 8 CONCLUSION

We introduce **Persia**, a distributed system to support efficient and scalable training of recommender models at the scale of 100 trillion parameters. We Archive the statistical and hardware efficiency by a careful co-design of both the distributed training algorithm and the distributed training system. Our proposed hybrid distributed training algorithm introduces elaborate system relaxations for efficient utilization of heterogeneous clusters, while converging as fast as vanilla SGD. We implement a wide range of system optimizations to overcome the unique challenges raised by the hybrid algorithmic design. We evaluate **Persia** using both publicly available benchmark tasks and production tasks at Kwai. We show that **Persia** leads to up to \(7.12 \times\) speedup compared to alternative state-of-the-art approaches; additionally, **Persia** can also scale out to 100-trillion-parameter model on Google cloud platform.

**Persia** has been released as an open source project on github with concrete instructions about setup over Google cloud platform—we hope everyone from both academia and industry would find it easy to train 100-trillion-parameter scale deep learning recommender models.

### REFERENCES

[1] [n.d.]. Avazu Advertisement CTR. https://www.kaggle.com/c/avazu-ctr-prediction.
[2] [n.d.] bRPC Industrial-grade RPC framework used throughout Baidu. http://incubator.apache.org/clutch/brpc.html.
[3] [n.d.] Criteo-Advertisement CTR. https://labal.criteo.com/download-criteo-hb-click-log-data/.
[4] [n.d.] gRPC A high performance, open source universal RPC framework. https://grpc.io/.
[5] [n.d.] How to implement LRU cache using HashMap and Doubly Linked List. https://zhuanlan.zhihu.com/p/39233727.
[6] [n.d.] MindSpore AI computing Framework bridge the gap between AI research and application. https://www.paddlepaddle.org.cn./.
[7] [n.d.] NVIDIA GPU Direct. https://developer.nvidia.com/gpudirect.
[8] [n.d.] Open source machine learning framework that accelerates the path from research prototyping to production deployment. https://pytorch.org/.
[9] [n.d.] Parallel Distributed Deep Learning: Machine Learning Framework from Industrial Practice. https://www.paddlepaddle.org.cn/.
[10] [n.d.] Taobao Advertisement CTR. https://www.kaggle.com/pavansanagapati/ad-display-click-data-on-taobao.com.
[11] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A system for large-scale machine learning. In 12th [USENIX] symposium on operating systems design and implementation ([OSDI] 16). 265–283.
[12] Bilge Acun, Matthew Murphy, Xiaodong Wang, Jane Nie, Carole-Jean Wu, and Kim Hazelwood. 2020. Understanding Training Efficiency of Deep Learning Recommendation Models at Scale. arXiv preprint arXiv:2013.05497 (2020).
[13] Dan Altschul, Torsten Hoefler, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2016. QSGD: Communication-efficient SGD via gradient quantization and encoding. arXiv preprint arXiv:1610.02132 (2016).
[14] Dan Altschul, Torsten Hoefler, Mikael Johansson, Sarit Krahiart, Nikola Konstantinov, and Cedric Bengli. 2018. The convergence of sparsified gradient methods. In Proceedings of the 32nd International Conference on Neural Information Processing Systems. 5977–5987.
[15] Debraj Basu, Deepak Narayanan, Amar Phansriayee, Vivek Seshadri, Nikhil Devanur, Greg Gangler, and Phil Gibbons. 2018. Pipedream: Fast and efficient pipeline parallel dnn training. arXiv preprint arXiv:1806.03377 (2018).
[16] Jingyuan Chen, Hanwang Zhang, Xiangnan He, Liqiang Nie, Wei Liu, and Tat-Seng Chua. 2015. Item2Vec: Deep neural networks for recommender systems with item-and component-level attention. In Proceedings of the 10th ACM conference on Conference on Neural Information Processing Systems-Volume 1 (2015). 191–198.
[17] Aleksandr Beznosikov, Samuel Horváth, Peter Richtárik, and Mher Safaryan. 2019. Qsparse-local-SGD: Distributed SGD with quantization, sparsification, and local computations. arXiv preprint arXiv:1906.02367 (2019).
[18] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Debraj Basu. 2019. Qsparse-Local-SGD: Distributed SGD with quantization, sparsification, and local computations. arXiv preprint arXiv:1906.02367 (2019).
[19] Debraj Basu, Samuel Horváth, Peter Richtárik, and Mher Safaryan. 2019. Qsparse-Local-SGD: Distributed SGD with quantization, sparsification, and local computations. arXiv preprint arXiv:1906.02367 (2019).
[20] Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and scalable coordination library for distributed parameter servers. In Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2643–2653.
[21] Jianmin Chen, Xingbao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. 2012. Very deep convolutional networks for large-scale image recognition. In Proceedings of the 29th ACM International Conference on Information & Management Sciences, 785–794.
[22] Suresh Krishna and Ravi Krishna. 2020. Accelerating Recommender Systems via Hardware” scale-in”. arXiv preprint arXiv:2009.05230 (2020).
[23] Yiming Jiang, Bin Cui, Ce Zhang, and Lele Yu. 2017. Heterogeneity-aware distributed parameter servers. In Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data. 463–478.
[24] Shuiguang Deng, Longtao Huang, Guandong Xu, Xindong Wu, and Zhaohui Wu. 2016. Deep learning for high-dimensional sparse data. In Proceedings of the 20th ACM SIGKDD Conference on Knowledge Discovery & Data Mining. 191–198.
[25] Charles Sugnet, Mark Ulrich, and Jure Leskovec. 2018. Pixie: A system for processing of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining. 2643–2653.
[26] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Njeim, Quoc V Le, Yonghui Wu, et al. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In NeurIPS.
[27] Po-Jen Hsieh, Chia-Ling Chiang, and Hsing-Fang Hung. 2015. Immersive recommendation: News and event recommendations using personal digital traces. In Proceedings of the 25th International Conference on World Wide Web 51–62.
[28] Shuangshuang Deng, Longtao Huang, Guandong Xu, Xindong Wu, and Zhaohui Wu. 2016. Deep learning for trust-aware recommendations in social networks. IEEE transactions on neural networks and learning systems 28, 5 (2016), 1364–1377.
[29] Shuangshuang Deng, Longtao Huang, Guandong Xu, Xindong Wu, and Zhaohui Wu. 2016. Deep learning for trust-aware recommendations in social networks. IEEE transactions on neural networks and learning systems 28, 5 (2016), 1364–1377.
[30] Joonseok Lee, Sami Abu-El-Haija, Balakrishnan Varadarajan, and Apostol Ntaflos, Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. 2012. Very deep convolutional networks for large-scale image recognition. In Proceedings of the 29th ACM International Conference on Information & Management Sciences, 785–794.
[31] Shuangshuang Deng, Longtao Huang, Guandong Xu, Xindong Wu, and Zhaohui Wu. 2016. Deep learning for trust-aware recommendations in social networks. IEEE transactions on neural networks and learning systems 28, 5 (2016), 1364–1377.
[32] Changat Eksombatchai, Pranav Jindal, Jerry Zitou Liu, Yuchen Liu, Rahul Sharma, Charles Sugnet, Mark Ulrich, and Jure Leskovec. 2018. Pix: A system for recommending 0.5 billion items to 200+ million users in real-time. In Proceedings of the 2018 world wide web conference. 1775–1784.
[33] Shouduo Gan, Xiangru Lian, Rui Wang, Jianbin Chang, Chengjun Li, Tengxu Sun, Jiawei Jiang, et al. 2021. BAGUA: Scaling up Distributed Learning with System Relaxations. arXiv preprint arXiv:2107.01499 (2021).
[34] Roberto Amatriain, Kang Lu, Mike Webber, and Andrew Charles. 2019. Recommendations. Business, value, and innovation. ACM Transactions on Management Information Systems (TAMIS) 6, 4 (2015), 1–15.
[35] Farnaz Haddadpour, Mohammad Mahdi Kamani, Mehrdad Mahdavi, and Vivek R Cadambe. 2019. Local SGD with periodic averaging: Tighter analysis and adaptive synchronization. arXiv preprint arXiv:1910.13589 (2019).
[36] Shuangshuang Deng, Longtao Huang, Guandong Xu, Xindong Wu, and Zhaohui Wu. 2016. Deep learning for trust-aware recommendations in social networks. IEEE transactions on neural networks and learning systems 28, 5 (2016), 1364–1377.
[37] Franci Nollet, Mohammad Mahdi Kamani, Mehrdad Mahdavi, and Vivek R Cadambe. 2019. Local SGD with periodic averaging: Tighter analysis and adaptive synchronization. arXiv preprint arXiv:1910.13589 (2019).
[38] Farnaz Haddadpour, Mohammad Mahdi Kamani, Mehrdad Mahdavi, and Vivek R Cadambe. 2019. Local SGD with periodic averaging: Tighter analysis and adaptive synchronization. arXiv preprint arXiv:1910.13589 (2019).
[39] Farnaz Haddadpour, Mohammad Mahdi Kamani, Mehrdad Mahdavi, and Vivek R Cadambe. 2019. Local SGD with periodic averaging: Tighter analysis and adaptive synchronization. arXiv preprint arXiv:1910.13589 (2019).
[40] Farnaz Haddadpour, Mohammad Mahdi Kamani, Mehrdad Mahdavi, and Vivek R Cadambe. 2019. Local SGD with periodic averaging: Tighter analysis and adaptive synchronization. arXiv preprint arXiv:1910.13589 (2019).
[41] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014).
[42] Anastasia Koloskova, Sebastian Stich, and Martin Jaggi. 2019. Decentralized stochastic optimization and gossip algorithms with compressed communication. In International Conference on Machine Learning. PMLR, 3478–3487.
[43] Sungwook Han, Sangtae Park, and Jure Leskovec. 2015. DeepWalk: online learning of social representations. In Proceedings of the 21st ACM SIGKDD Conference on Knowledge Discovery & Data Mining. 1017–1027.
[44] Shuangshuang Deng, Longtao Huang, Guandong Xu, Xindong Wu, and Zhaohui Wu. 2016. Deep learning for trust-aware recommendations in social networks. IEEE transactions on neural networks and learning systems 28, 5 (2016), 1364–1377.
[94] Youjie Li, Mingchao Yu, Songze Li, Salmen Avestimehr, Nan Sun, Kim, and Alexander Schwing. 2018. Pipe-SGD: a decentralized pipelined SGD framework for distributed deep net training. In Proceedings of the 32nd International Conference on Neural Information Processing Systems. 8056–8067.

[50] Zhi Li, Hongke Zhao, Qi Liu, Zhenyu Huang, Tao Mei, and Enhong Chen. 2018. Learning from history and present: Next-item recommendation via discriminatively exploiting user behaviors. In Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 1734–1743.

[51] Zhuhuan Li, Siyuan Zhaung, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. 2021. TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models. arXiv preprint arXiv:2102.07888 (2021).

[52] Xiangru Lian, Yujin Huang, Yuncheng Li, and J. Liu. 2015. Asynchronous parallel stochastic gradient for nonconvex optimization. Advances in Neural Information Processing Systems 28 (2015), 2777–2785.

[53] Xiangru Lian, Binhang Yuan, Xufeng Zhu, Yulong Wang, Yongjun He, Hongjuan Wu, Wei Sun, Haodong Lyu, Chengjun Liu, Xing Dong, et al. 2021. Parsia: A Hybrid System Scaling Deep Learning Based Recommenders up to 100 Trillion Parameters. arXiv preprint arXiv:2111.05897 (2021).

[54] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and J. Liu. 2017. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. In Proceedings of the 31st International Conference on Neural Information Processing Systems. 5356–5364.

[55] Xiangru Lian, Huan Zhang, Cho-Jui Hsieh, Yijun Huang, and J. Liu. 2016. A Comprehensive Linear Speedup Analysis for Asynchronous Stochastic Parallel Optimization from Zeroth-Order to First-Order. NIPS (2016).

[56] Xiangru Lian, Weihang Zhang, Ce Zhang, and Ji Liu. 2019. Asynchronous decentralized parallel stochastic gradient descent. In International Conference on Machine Learning, PMLR, 3043–3052.

[57] Tao Lin, Sebastian U Stich, Kumar Kshitij Patel, and Martin Jaggi. 2019. Don’t Use Large Mini-batches, Use Local SGD. In International Conference on Learning Representations.

[58] David C Liu, Stephanie Rogers, Raymond Shiao, Dmitry Kislyuk, Kevin C Ma, David C Liu, Stephanie Rogers, Raymond Shiao, Dmitry Kislyuk, Kevin C Ma, et al. 2020. Horovod: fast and easy distributed training on (Whimpy) Heterogeneous Clusters through Inte-
APPENDIX

Lemma 1. Under Assumption 1, for any \( \delta \in \mathbb{R}^{N_{\text{emb}}} \) the following inequality holds:

\[
\left\| f' \left( \mathbf{w}_{t+1}^{\text{emb}} + \delta, \mathbf{w}_{t+1}^{\text{nn}} \right) - f' \left( \mathbf{w}_{t}^{\text{emb}}, \mathbf{w}_{t}^{\text{nn}} \right) \right\| \leq \alpha L \| \delta \| , \quad \forall \mathbf{w}_{t}^{\text{emb}} \in \mathbb{R}^{N_{\text{emb}}}, \mathbf{w}_{t}^{\text{nn}} \in \mathbb{R}^{N_{\text{nn}}}.
\]

Proof. Define Hessian matrix w.r.t. the sparse weights \( \mathbf{w}_{t}^{\text{emb}} \):

\[
\mathbf{H}_{f}^{\text{emb}} := \begin{pmatrix}
\frac{\partial^2 f (\mathbf{w}_{t}^{\text{emb}}, \mathbf{w}_{t}^{\text{nn}})}{\partial \mathbf{w}_{t}^{\text{emb}} \partial \mathbf{w}_{t}^{\text{emb}}} & \cdots & \frac{\partial^2 f (\mathbf{w}_{t}^{\text{emb}}, \mathbf{w}_{t}^{\text{nn}})}{\partial \mathbf{w}_{t}^{\text{emb}} \partial \mathbf{w}_{t}^{\text{nn}}} \\
\frac{\partial^2 f (\mathbf{w}_{t}^{\text{emb}}, \mathbf{w}_{t}^{\text{nn}})}{\partial \mathbf{w}_{t}^{\text{nn}} \partial \mathbf{w}_{t}^{\text{emb}}} & \cdots & \frac{\partial^2 f (\mathbf{w}_{t}^{\text{emb}}, \mathbf{w}_{t}^{\text{nn}})}{\partial \mathbf{w}_{t}^{\text{nn}} \partial \mathbf{w}_{t}^{\text{nn}}}
\end{pmatrix}
\]

From the Lipschitz assumption (3) and the stochastic gradient function \( F'(\cdot ; \xi) \) is differentiable, we obtain the Hessian matrix of \( F(\cdot ; \xi) \) is \( \leq LI \), where \( I \) is the identity matrix. Noting that since for each sample, the \( \mathbf{w}_{t}^{\text{emb}} \) is used by a lookup operation (??) where only a subset of the weights get used. For any element \( \mathbf{w}_{t}^{\text{emb}} \), if \( i \notin \Omega_{\xi} \), \( \mathbf{w}_{t}^{\text{emb}} \) will have no effect on the value of \( F'(\cdot ; \xi) \). This implies the corresponding elements are zero in \( F(\cdot ; \xi) \)’s Hessian matrix, and the following inequality holds:

\[
\mathbf{H}_{f}^{\text{emb}} \leq LI_{\xi},
\]

where

\[
(I_{\xi})_{i,j} = \begin{cases} 1, & \text{if } i = j \text{ and } i \in \Omega_{\xi}, \\ 0, & \text{otherwise}. \end{cases}
\]

Note that from the definition of \( f \) in (1), we obtain the Hessian of \( f \) is the expectation of the Hessian of \( F \):

\[
\mathbf{H}_{f}^{\text{emb}} = \mathbb{E}_{\xi} \mathbf{H}_{f}^{\text{emb}} \leq \mathbb{E}_{\xi} I_{\xi} \leq LI_{\xi}.
\]

Finally following the definition of \( \alpha \), we obtain \( \mathbf{H}_{f}^{\text{emb}} \leq LI_{\xi} \leq \alpha L \), which leads to the \( \alpha L \) Lipschitzian continuity in the sparse weights (7).

Proof to Theorem 1

Proof. Since \( f \) is defined as the average of \( F \), from Lipschitzian gradient assumption (3), it can be derived that \( f \) is also Lipschitzian:

\[
\left\| f' \left( \mathbf{w} + \delta \right) - f' \left( \mathbf{w} \right) \right\| \leq L \| \delta \| , \quad \forall \mathbf{w}, \delta \in \mathbb{R}^{N_{\text{emb}}+N_{\text{nn}}}.
\]

From the algorithm update rule (2) and the Lipschitz property (8), we obtain

\[
f \left( \mathbf{w}_{t+1}^{\text{emb}}, \mathbf{w}_{t+1}^{\text{nn}} \right) = f \left( \mathbf{w}_{t}^{\text{emb}}, \mathbf{w}_{t}^{\text{nn}} \right) \leq f \left( \mathbf{w}_{t}^{\text{emb}}, \mathbf{w}_{t}^{\text{nn}} \right) + \frac{L}{2} \left\| \mathbf{w}_{t+1}^{\text{emb}} + \mathbf{w}_{t+1}^{\text{nn}} - \mathbf{w}_{t}^{\text{emb}} - \mathbf{w}_{t}^{\text{nn}} \right\|^2 \]

\[
= -\gamma \left( f' \left( \mathbf{w}_{t}^{\text{emb}}, \mathbf{w}_{t}^{\text{nn}} \right), F_{t}^{\text{emb'}} + F_{t}^{\text{nn'}} \right) + \frac{L \gamma^2}{2} \left\| F_{t}^{\text{emb'}} + F_{t}^{\text{nn'}} \right\|^2
\]

Taking the expectation on both sides, we obtain

\[
\mathbb{E} \left[ f \left( \mathbf{w}_{t+1}^{\text{emb}}, \mathbf{w}_{t+1}^{\text{nn}} \right) - f \left( \mathbf{w}_{t}^{\text{emb}}, \mathbf{w}_{t}^{\text{nn}} \right) \right] \leq -\gamma \mathbb{E} \left[ \left( f' \left( \mathbf{w}_{t}^{\text{emb}}, \mathbf{w}_{t}^{\text{nn}} \right), F_{t}^{'} \right) \right] + \frac{L \gamma^2}{2} \mathbb{E} \left\| F_{t}^{'} \right\|^2.
\]
We show the bound for the terms $T_1$ and $T_2$ respectively, in the following paragraphs. For $T_2$, from the unbiased property of stochastic gradients (1) and the bounded variance assumption (4) we obtain

$$T_2 = \mathbb{E} \left[ \left| f_t' \right|^2 \right]$$

$$= \mathbb{E} \left[ \left| f_t' - f' \left( w_{D(t)}^{\text{emb}}, w_t^{\text{nn}} \right) + f' \left( w_{D(t)}^{\text{emb}}, w_t^{\text{nn}} \right) \right|^2 \right]$$

$$= \mathbb{E} \left[ \left| f_t' - f' \left( w_{D(t)}^{\text{emb}}, w_t^{\text{nn}} \right) \right|^2 + \left| f' \left( w_{D(t)}^{\text{emb}}, w_t^{\text{nn}} \right) \right|^2 \right] + 2 \mathbb{E} \left[ \left( f_t' - f' \left( w_{D(t)}^{\text{emb}}, w_t^{\text{nn}} \right) \right) \cdot f' \left( w_{D(t)}^{\text{emb}}, w_t^{\text{nn}} \right) \right] = 0, \text{ from (1)}$$

$$\leq \sigma^2 + \mathbb{E} \left[ \left| f' \left( w_{D(t)}^{\text{emb}}, w_t^{\text{nn}} \right) \right|^2 \right]. \quad (10)$$

For $T_1$, we have

$$T_1 = \mathbb{E} \left[ \left( f' \left( w_t^{\text{emb}}, w_t^{\text{nn}} \right), f_t' \right) \right]$$

$$= \mathbb{E} \left[ \left( f' \left( w_t^{\text{emb}}, w_t^{\text{nn}} \right), f' \left( w_{D(t)}^{\text{emb}}, w_t^{\text{nn}} \right) \right) \right]$$

$$= \frac{1}{2} \mathbb{E} \left[ \left| f' \left( w_t^{\text{emb}}, w_t^{\text{nn}} \right) \right|^2 + \left| f' \left( w_{D(t)}^{\text{emb}}, w_t^{\text{nn}} \right) \right|^2 - \left| f' \left( w_t^{\text{emb}}, w_t^{\text{nn}} \right) - f' \left( w_{D(t)}^{\text{emb}}, w_t^{\text{nn}} \right) \right|^2 \right]. \quad (11)$$

where the last step comes from the fact that for any $a, b$ the following equality holds: $\langle a, b \rangle = \frac{1}{2} \| a - b \|^2 - \frac{1}{2} \| a \|^2 - \frac{1}{2} \| b \|^2$.

Using the bounds for $T_1$ (11) and $T_2$ (10) in (9), we obtain

$$\mathbb{E} \left[ f \left( w_{t+1}^{\text{emb}}, w_{t+1}^{\text{nn}} \right) - f \left( w_t^{\text{emb}}, w_t^{\text{nn}} \right) \right]$$

$$\leq -\frac{\gamma L}{2} \mathbb{E} \left[ \left| f' \left( w_t^{\text{emb}}, w_t^{\text{nn}} \right) \right|^2 + \left| f' \left( w_{D(t)}^{\text{emb}}, w_t^{\text{nn}} \right) \right|^2 - \left| f' \left( w_t^{\text{emb}}, w_t^{\text{nn}} \right) - f' \left( w_{D(t)}^{\text{emb}}, w_t^{\text{nn}} \right) \right|^2 \right]$$

$$+ \frac{L \sigma^2}{2} \mathbb{E} \left[ \left| f' \left( w_{D(t)}^{\text{emb}}, w_t^{\text{nn}} \right) \right|^2 \right] = -\frac{\gamma L}{2} \mathbb{E} \left[ \left| f' \left( w_t^{\text{emb}}, w_t^{\text{nn}} \right) \right|^2 \right] - \frac{\gamma}{2} (1 - \gamma L) \mathbb{E} \left[ \left| f' \left( w_{D(t)}^{\text{emb}}, w_t^{\text{nn}} \right) \right|^2 \right]$$

$$+ \frac{L \sigma^2}{2} \mathbb{E} \left[ \left| f' \left( w_{D(t)}^{\text{emb}}, w_t^{\text{nn}} \right) \right|^2 \right] \leq -\frac{\gamma L}{2} \mathbb{E} \left[ \left| f' \left( w_t^{\text{emb}}, w_t^{\text{nn}} \right) \right|^2 \right] + \frac{L \sigma^2}{2} \mathbb{E} \left[ \left| f' \left( w_t^{\text{emb}}, w_t^{\text{nn}} \right) - f' \left( w_{D(t)}^{\text{emb}}, w_t^{\text{nn}} \right) \right|^2 \right]$$

$$\leq -\frac{\gamma L}{2} \mathbb{E} \left[ \left| f' \left( w_t^{\text{emb}}, w_t^{\text{nn}} \right) \right|^2 \right] + \frac{L \sigma^2}{2} \mathbb{E} \left[ \left| f' \left( w_t^{\text{emb}}, w_t^{\text{nn}} \right) - f' \left( w_{D(t)}^{\text{emb}}, w_t^{\text{nn}} \right) \right|^2 \right]. \quad (12)$$

where the last inequality is obtained by choosing a sufficiently small learning rate $\gamma$ satisfying $1 \geq \gamma L$.

We then bound $\omega_t$ in order to get the final convergence rate result. Denoting

$$L_{\text{emb}} := \min \{ \ell, \alpha \} L,$$

$$F_t^{\text{emb}}(t) := \nabla_{w} f \left( w_{D(t)}^{\text{emb}}, w_t^{\text{emb}} \right),$$

$$F_t^{\text{nn}}(t) := \nabla_{w} f \left( w_{D(t)}^{\text{emb}}, w_t^{\text{emb}} \right).$$
it follows from Lemma 1 that

\[ \omega_t = \mathbb{E} \left[ \left\| f' \left( w_t^{emb}, w_t^{nn} \right) - f' \left( w_{D(t)}^{emb}, w_{D(t)}^{nn} \right) \right\|^2 \right] \]

\[ \leq L_{emb}^2 \mathbb{E} \left[ \left\| w_t^{emb} - w_{D(t)}^{emb} \right\|^2 \right] \]

\[ \leq \gamma^2 L_{emb}^2 \mathbb{E} \left[ \left\| \sum_{k=D(t)}^{t-1} F_k^{emb} \right\|^2 \right] \]

\[ \leq \gamma^2 L_{emb}^2 \mathbb{E} \left[ \left\| \sum_{k=1}^{D(t)} F_k^{emb} - \sum_{k=1}^{D(t)} F_k^{emb} + \sum_{k=1}^{D(t)} F_k^{emb} \right\|^2 \right] \]

\[ \leq \gamma^2 L_{emb}^2 \mathbb{E} \left[ \sum_{k=1}^{D(t)} \left\| F_k^{emb} \right\|^2 \right] \]

\[ \leq \gamma^2 L_{emb}^2 \mathbb{E} \left[ \sum_{k=1}^{D(t)} \left\| F_k^{emb} \right\|^2 \right] \]

\[ \leq \gamma^2 \tau L_{emb}^2 \sigma^2 + \gamma^2 L_{emb}^2 \mathbb{E} \left[ \sum_{k=1}^{D(t)} \left\| F_k^{emb} \right\|^2 \right] \]

\[ \leq \gamma^2 \tau L_{emb}^2 \sigma^2 + \gamma^2 L_{emb}^2 \mathbb{E} \left[ \sum_{k=1}^{D(t)} \left\| F_k^{emb} \right\|^2 \right] \]

\[ \leq \gamma^2 \tau L_{emb}^2 \sigma^2 + \gamma^2 L_{emb}^2 \mathbb{E} \left[ \sum_{k=1}^{D(t)} \left\| F_k^{emb} \right\|^2 \right] \]

(13)

where the third last step and the last step come from the fact that for any \(a_1, a_2, \ldots, a_n\), we have the following inequality: \(\|a_1 + a_2 + \cdots + a_n\|^2 \leq n \sum_{i=1}^n \|a_i\|^2\). We then bound the right side of (13) with \(\omega_t\) itself:

\[ \omega_t \leq \gamma^2 L_{emb}^2 \tau \sigma^2 + \gamma^2 L_{emb}^2 \mathbb{E} \left[ \sum_{k=1}^{D(t)} \left\| F_k^{emb} \right\|^2 \right] \]

\[ = \gamma^2 L_{emb}^2 \tau \sigma^2 + \gamma^2 L_{emb}^2 \mathbb{E} \left[ \sum_{k=1}^{D(t)} \left\| F_k^{emb} \right\|^2 \right] \]

\[ \leq \gamma^2 L_{emb}^2 \tau \sigma^2 + \gamma^2 L_{emb}^2 \mathbb{E} \left[ \sum_{k=1}^{D(t)} \left\| F_k^{emb} \right\|^2 \right] \]

\[ \leq \gamma^2 L_{emb}^2 \tau \sigma^2 + \gamma^2 L_{emb}^2 \mathbb{E} \left[ \sum_{k=1}^{D(t)} \left\| F_k^{emb} \right\|^2 \right] \]

(14)

Summing (14) from \(t = 0\) to \(t = T - 1\) we obtain

\[ \sum_{t=0}^{T-1} \omega_t \leq \gamma^2 L_{emb}^2 \tau \sigma^2 T + 2 \gamma^2 L_{emb}^2 \mathbb{E} \left[ \sum_{t=0}^{T-1} \omega_t + \sum_{t=0}^{T-1} \left\| f' \left( w_t \right) \right\|^2 \right] \]

Rearranging the terms, we obtain

\[ \left( 1 - 2 \gamma^2 L_{emb}^2 \right) \sum_{t=0}^{T-1} \omega_t \leq \gamma^2 L_{emb}^2 \tau \sigma^2 T + 2 \gamma^2 L_{emb}^2 \mathbb{E} \left[ \sum_{t=0}^{T-1} \left\| f' \left( w_t \right) \right\|^2 \right] \]

By choosing a sufficiently small learning rate \(\gamma\) satisfying \(1/2 \geq \gamma \tau L_{emb}\), it follows that

\[ \sum_{t=0}^{T-1} \omega_t \leq 2 \gamma^2 L_{emb}^2 \tau \sigma^2 T + 4 \gamma^2 L_{emb}^2 \mathbb{E} \left[ \sum_{t=0}^{T-1} \left\| f' \left( w_t \right) \right\|^2 \right] \]

(15)
Now we get back to (12). Summing (12) from $t = 0$ to $t = T - 1$ we obtain

$$
\mathbb{E} \left[ f \left( w_t^{\text{emb}}, w_t^{\text{nn}} \right) - f \left( w_0^{\text{emb}}, w_0^{\text{nn}} \right) \right] \\
\leq - \frac{1}{2} \sum_{t=0}^{T-1} \mathbb{E} \left[ \left\| f' \left( w_t^{\text{emb}}, w_t^{\text{nn}} \right) \right\|^2 \right] + \frac{L^2}{2} \sigma^2 T + \frac{y}{2} \sum_{t=0}^{T-1} \omega_t
$$

where the last step comes from choosing a sufficiently small learning rate $\gamma$ satisfying $1/4 \geq \gamma T L$. Rearranging the terms we obtain:

$$
\begin{align*}
&\leq \frac{4}{T} \mathbb{E} \left[ f \left( w_0^{\text{emb}}, w_0^{\text{nn}} \right) - f \left( w_t^{\text{emb}}, w_t^{\text{nn}} \right) \right] + \frac{L^2}{2} \sigma^2 T + 4 \gamma^2 \eta T L \sigma T \\
&\leq \frac{4}{T} \mathbb{E} \left[ f \left( w_0^{\text{emb}}, w_0^{\text{nn}} \right) - f \left( w_t^{\text{emb}}, w_t^{\text{nn}} \right) \right] + 2 L \sigma^2 + y \eta T L \sigma^2 T \\
&\leq \frac{4}{T} \mathbb{E} \left[ f \left( w_0^{\text{emb}}, w_0^{\text{nn}} \right) - f' \right] + 3 L \sigma^2,
\end{align*}
$$

where the last step comes from the definition of $\eta$ and the second last step still comes from the our choosing a sufficiently small learning rate $\gamma$ satisfying $1/4 \geq \gamma T L$.

By choosing the learning rate $\gamma$ to be the same as the optimal learning rate for SGD $\frac{1}{L + \sqrt{L} \sigma + 4 \tau L}$, we obtain

$$
\sum_{t=0}^{T-1} \mathbb{E} \left[ \left\| f' \left( w_t^{\text{emb}}, w_t^{\text{nn}} \right) \right\|^2 \right]
\leq \frac{\sqrt{L} \sigma}{\sqrt{T}} + \frac{L}{T} + \frac{\tau L \text{emb}}{T}
\leq \frac{\sqrt{L} \sigma}{\sqrt{T}} + \frac{L}{T} + \frac{\tau \min \{1, \alpha\}}{T}
$$

completing the proof. □