Hierarchical Training: Scaling Deep Recommendation Models on Large CPU Clusters

Yuzhen Huang, Xiaohan Wei, Xing Wang, Jiyan Yang, Bor-Yiing Su,
Shivam Bharuka, Dhruv Choudhary, Zewei Jiang, Hai Zheng, Jack Langman
Facebook, 1 Hacker Way, Menlo Park, CA 94065
{yuzhenhuang,ubimeteor,xingwang,chocjy,boryiingsu,shivamb,choudharydhruv,zeweijiang,haizheng,youknowjack}@fb.com

ABSTRACT

Neural network based recommendation models are widely used to power many internet-scale applications including product recommendation and feed ranking. As the models become more complex and more training data is required during training, improving the training scalability of these recommendation models becomes an urgent need. However, improving the scalability without sacrificing the model quality is challenging. In this paper, we conduct an in-depth analysis of the scalability bottleneck in existing training architecture on large scale CPU clusters. Based on these observations, we propose a new training architecture called **Hierarchical Training**, which exploits both data parallelism and model parallelism for the neural network part of the model within a group. We implement hierarchical training with a two-layer design: a tagging system that decides the operator placement and a net transformation system that materializes the training plans, and integrate hierarchical training into existing training stack. We propose several optimizations to improve the scalability of hierarchical training including model architecture optimization, communication compression, and various system-level improvements. Extensive experiments at massive scale demonstrate that hierarchical training can speed up distributed recommendation model training by 1.9x without model quality drop.

CCS CONCEPTS

• **Computing methodologies** → **Distributed algorithms**; • **Computer systems organization** → *Distributed architectures.*

KEYWORDS

distributed training, optimization, system for machine learning

ACM Reference Format:
Yuzhen Huang, Xiaohan Wei, Xing Wang, Jiyan Yang, Bor-Yiing Su,, Shivam Bharuka, Dhruv Choudhary, Zewei Jiang, Hai Zheng, Jack Langman. 2021. Hierarchical Training: Scaling Deep Recommendation Models on Large CPU Clusters. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '21), August 14–18, 2021, Virtual Event, Singapore.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3447548.3467084

KDD '21, August 14–18, 2021, Virtual Event, Singapore

ACM ISBN 978-1-4503-8332-5/21/08…$15.00
https://doi.org/10.1145/3447548.3467084

1 INTRODUCTION

The wide adoption of the recommendation and personalization applications including apps/news/friends/videos recommendation, groups suggestions, feed ranking, etc. have greatly improved the Internet and Apps experience [7, 9, 15, 49]. With the recent advent of deep learning, most of these personalization workloads are powered by neural network based machine learning models, e.g., DLRM [31] and Wide & Deep [6], for better recommendation accuracy. At Facebook, the training of these recommendation models takes up more than 50% of the total AI training cycles [2].

In these recommendation models [6, 13, 28, 31, 50], embeddings and higher-order interactions are used to learn from the sparse categorical features, and then deep neural network is applied to improve the generalization of the models. The embedding tables learned for the large number of the sparse categorical features are in general memory-intensive and can consume up to tens of terabytes of the memory [15, 36, 47]. This makes the recommendation models different from the content understanding models used in computer vision, speech recognition, etc., which are mainly compute-intensive and can be better accelerated by GPUs. These recommendation models can be trained efficiently using large scale CPU clusters where each training job contains tens to hundreds of CPU nodes at Facebook [15, 17]. A hybrid of model parallelism and data parallelism scheme is used to train these recommendation models for the sparse part and the dense part [48]. More specifically, the embedding tables of the sparse categorical features are partitioned onto a dedicated set of parameter servers which are accessed in a model parallel manner, while the deep neural network part (i.e., the dense part) of the model is replicated across all trainers and is trained using data parallelism.

To further improve the user experience and provide more accurate and satisfactory recommendation, the deep learning models are becoming larger and more complex. For example, more complicated layers, e.g., deep attention, skip layers, etc., and wider and heavier layers are added to the models to improve model quality [38, 41]. At the same time, more training data are used to train these complex recommendation models. The growing model size and data size call for the need of improving the training scalability of the large scale distributed training system.

The existing distributed training architecture presented above has been demonstrated to have good scalability in terms of the training throughput [48]. To increase the number of sparse categorical features, one can add more sparse parameter servers to the system to hold the additional embedding tables. To boost the training throughput, more trainers can be added to improve the processing power. However, increasing the training throughput

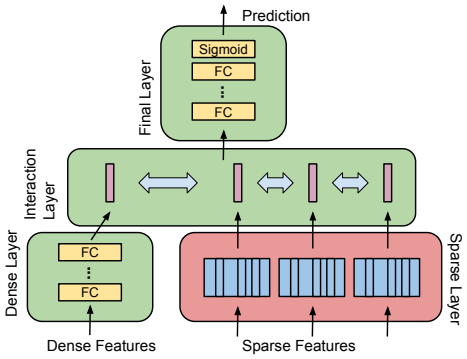


Figure 1: The DLRM-like Model

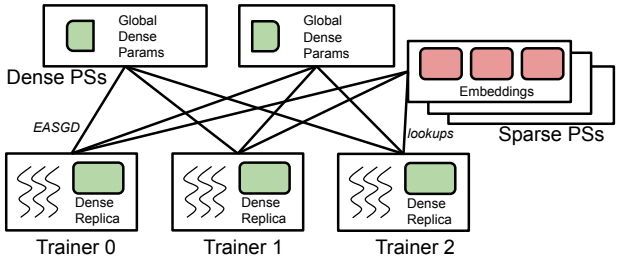


Figure 2: Existing EASGD-based Architecture

does not come for free in the existing architecture. When increasing the trainers in the system, we observe a non-negligible model quality drop which blocks us from scaling out.

In this paper, we identify through experiments that in existing system, increasing the number of dense replica will hurt the model quality severely, but increasing the parallelism and staleness within one dense replica will not degrade the model quality a lot. Based on these observations, we propose a new distributed training architecture called **Hierarchical Training** to greatly improve the scalability. In hierarchical training, we replace a trainer with a group of training nodes, which we call a *group*. This setup greatly improves the processing power within a group while at the same time keeps the number of dense replicas small. To realize hierarchical training in our existing training stack: we design a two-layer system: (1) *The Tagging System* decides the operators placement strategy in a group using a rule-based tagging process and an iterative greedy refinement process; (2) *The Net Transformation System* is the back-end to handle the net partitioning and replication, and finally materialize the training plans for all nodes.

Our main contributions can be summarized as follows:

- We conduct an in-depth analysis of the scalability limitation of the current distributed training architecture on recommendation models and identify potential opportunity to increase the training throughput while at the same time preserve the model quality. (Section 2)
- We design a novel architecture called Hierarchical Training (HT) based on the observations and implement it with a two-layer system. (Sections 3)
- We propose a set of optimizations for HT including model architecture optimization and communication compression. We highlight challenges we face when shipping complex models with HT. (Section 4)
- We conduct a comprehensive evaluation for HT on massive scale recommendation workloads. (Section 5)

2 BACKGROUND AND MOTIVATION

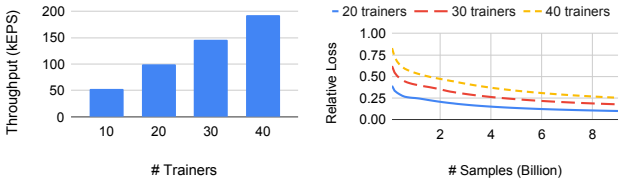
2.1 DLRM

The models powering most of the deep learning recommendation workfloads at Facebook are in the form of DLRM [31]. As shown in Figure 1, a DLRM-like model is composed of four major layers. The sparse layer contains the embedding tables which are used to process the sparse categorical features. One can scale the models by adding more sparse features. The dense layer is several MLPs (multilayer perceptron) processing the continuous features (i.e., the dense features). The interaction layer takes the output of the dense layer and the sparse layer, computes the second-order interaction among them by taking the dot product between all the combination among the embedding vectors and the processed dense features. And then the final layer of MLPs are applied on the interaction and a sigmoid function will be applied to obtain the final prediction. Note that this is a high level description of the DLRM-like models, and each layer can be more complicated in reality.

The model can be expressed as a directed graph in the context of Caffe2 where each node represents an operator and each directed edge represents data dependency between two operators [3]. For example, a FC operator is followed by a Relu operator. The model graph is also known as the training net. It contains the forward and backward computation operators for one batch of the training data. The training process is to iteratively executing the training net until the model converges or a certain number of samples are processed. A typical model graph for DLRM-like models have hundreds to thousands of operators.

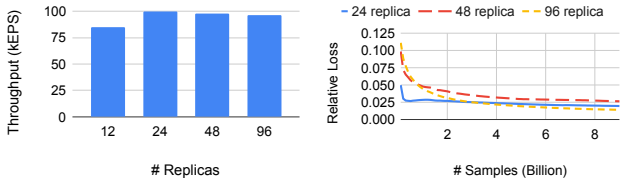
2.2 The Existing Training Architecture

Here we describe how we map the DLRM-like models to the existing training system at Facebook. In the existing training architecture, there are mainly three types of nodes: trainers, sparse parameter servers, dense parameter servers, as shown in Figure 2. From the system perspective, there are two parts in a DLRM-like model: the sparse part which is the sparse layer holding the embedding tables as mentioned in Section 2.1, and the dense part which contains everything else of the model, including the dense layer, the interaction layer and the final layer. This is because the system scales the sparse part and the dense part individually using different mechanisms. The sparse part (the sparse embedding tables) are partitioned onto a set of dedicated sparse parameter servers (sparse PSs). These PSs are shared by all trainers and can be easily scaled by adding more machines when more embedding tables are added. The dense part, on the other hand, are replicated on each trainer. In other words, each trainer holds one copy of the dense parameters from the dense part of the model. It accesses and updates the local copy of the dense parameters through local training. The synchronization of the dense parameters across all trainers is executed in



(a) Training Throughput **(b) Model Quality**

Figure 3: Effect of Inter-trainer Asynchronicity



(a) Training Throughput **(b) Model Quality**

Figure 4: Effect of Intra-trainer Asynchronicity

background through the dense parameter servers (dense PSs) using the ShadowSync framework [48] with the EASGD algorithm [46]. This allows the system to scale linearly by adding more trainers as the synchronization happens in the background and never blocks the iterative training process.

To fully utilize multiple CPU cores of each machine, trainers are multi-threaded and each trainer processes a number of copies of the dense part of the training net iteratively. Though the dense part of the training net has multiple copies/replicas on one trainer, there is only one dense parameter copy on each trainer shared by all training net replicas on the same trainer. The replicas of the training net are executed by a thread pool concurrently and they access and update the shared local dense parameters in a lock-free manner (similar to Hogwild! style update [33]). This design introduces asynchronicity and staleness in each trainer, because when a thread/net updates the parameter after reading it, other threads/nets may have already updated them. We found that such kind of asynchronicity at a certain range is acceptable.

During the training execution, each net in each trainer executes the following logic iteratively: reads a batch of data, sends requests to sparse PSs to perform embedding lookup operations, runs forward/backward computation on the local copy of the dense parameters, and finally updates the local dense parameters as well as the embedding tables on sparse PSs. In the background, each trainer has a dedicated net talking to the global dense parameter servers to synchronize the copy of the dense parameters using the EASGD algorithm [46] asynchronously. For simplicity, we call the existing training architecture the EASGD-based training system.

The system is expressing both the model parallelism and data parallelism: model parallelism is used to partition and place the embedding tables on sparse PSs; data parallelism is used within a trainer and across the trainers as multiple net replicas within each trainer are processing different batches of the data concurrently.

2.3 Asynchronicity Analysis in Existing System

To improve the model quality, more complex models and more training data are being explored. This calls for the needs of increasing the training throughput of the system, i.e., improving the training scalability, while at the same time preserving the model quality. Following the definition from ShadowSync [48], we use *Examples Per Second (EPS)*, the number of examples processed by the system per second, to evaluate the throughput of a training system.

Here we analyze the asynchronicity in the EASGD-based system. There are two types of asynchronicity for the dense part: *inter-trainer* and *intra-trainer*. Inter-trainer asynchronicity is introduced by the background EASGD synchronization algorithm. This is because the EASGD is running in the background, and dense parameters communication is not blocking the forward/backward training. Intra-trainer asynchronicity is incurred by the asynchronous update within one trainer. In each trainer, there are multiple replicated nets executing at the same time and they will access and update the same local copy of the dense parameters asynchronously. We define the staleness of the dense parameters introduced here as the number of writes conducted by other net replicas between a net read (during the forward pass) and write (during the backward pass) the same parameter. The maximum dense staleness for a trainer is bounded by the number of concurrent net replicas.

Intuitively both inter-trainer asynchronicity and intra-trainer asynchronicity may deteriorate the model quality to different extents and we conduct experiments to better understand their effects. We first analyze the effect of inter-trainer asynchronicity by changing the number of trainers for EASGD-based system. As shown in Figure 3a, the training throughput grows linearly with the number of trainers. However, we observed non-trivial model quality degradation when scaling out as shown in Figure 3b (the lower the better). Specifically, with 40 trainers, we observe 0.25% loss increase (relative loss compared with the 10-trainer case) which is significant. We believe it is because of the nature of the background model averaging algorithm. That is, the model quality drops a lot when there are many dense replicas in the system. Besides, adding dense replica will make each trainer see less training data given the total number of data fed into the system is unchanged.

Observation 2.1. *The EASGD-based system incurs non-neglectable model quality drop when increasing the number of trainers.*

We then conduct experiments to test the effect of intra-trainer asynchronicity as shown in Figure 4. Though the training throughput is not increasing because each trainer only has 18 physical CPU cores, we found that increasing the number of concurrent nets within a trainer will not severely hurt the model quality. The loss value is increased by around 0.02% for 96 nets compared with 12 nets when training with 9 billions of examples, which is much smaller compared with the degradation brought by inter-trainer asynchronicity.

Observation 2.2. *The EASGD-based system is not sensitive to intra-trainer asynchronicity, i.e., increasing the number of net replicas within a trainer will not severely hurt the model quality.*

Based on these observations, we have concluded that model quality is more sensitive to inter-trainer asynchronicity compared

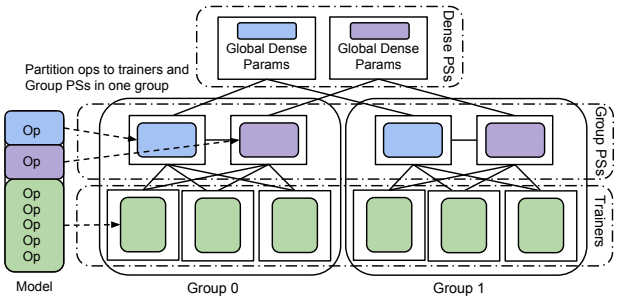


Figure 5: Hierarchical Training Architecture

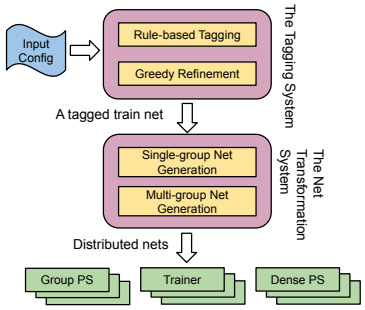


Figure 6: Hierarchical Training System Components

to the intra-trainer asynchronicity. This motivates us to redesign the system and introduce hierarchical training.

3 DESIGN

3.1 Hierarchical Training

We design **Hierarchical Training** (HT). We increase the processing power for a "trainer" while keeping the total number of "trainers", i.e., total number of dense replicas, small. We replace a trainer in the EASGD-based system with a group of nodes to increase the processing power. We call it a ***group***. A group consists of two types of nodes: a set of trainers and a set of group parameter servers (group PSs). The model graph (i.e., the training net) is partitioned to the group PSs and trainers within a group. An overall architecture is shown in Figure 5 where we have 2 groups and each group has 2 group PSs and 3 trainers. The trainers read data batches, conduct embedding lookups with the sparse PSs and initiate the iterative training loop as in the EASGD-based system (the sparse PSs are omitted in the figure for simplicity). The group PSs within a group participate in the computation and they jointly store one copy of the dense parameters of the model. The dense copy in each group are partitioned among group PSs. In this architecture, the total amount of data parallelism is $num\_trainers \times num\_concurrent\_nets\_per\_trainer$ and is preserved as the EASGD-based system. However, we only create num_groups dense replicas instead of $num\_trainers$ dense replicas compared to the EASGD-based system.

3.1.1 Computation and Communication Pattern. We introduce the computation and communication pattern within a group for hierarchical training. We first consider the partitioning of one net replica. In each group, the training net replica is partitioned into $k+1$ parts if there are k group PSs per group. Each group PS gets one partition of the net and all the trainers within a group gets a copy of the last partition. In other words, operators on trainers are replicated while operators on group PSs are partitioned; see Figure 5 for an example of $k = 2$: in each group, each group PS holds one operator, and each trainer holds the remaining 5 operators. Trainers control the training loop. When one or more operators are placed on a group PS, a trainer will send the input tensor to the group PS for execution. The group PS can then finish all the local execution, and then the output will be returned to the trainer who initiates the computation. As all the concurrent net replicas for all trainers within a group access and read the same copy of the dense parameters stored on the group PSs, if there are t trainers in a group, the maximum staleness on dense parameters is $t \times r$, where r is the number of concurrent nets per trainer. This effectively means we increase the number of threads in Hogwild!-style update within a group in hierarchical training.

Following the EASGD-based training system, hierarchical training uses background averaging algorithms, e.g., EASGD, to synchronize the local dense parameter copy among different groups. The communication is between group PSs and the global dense PSs as all the dense parameters are stored on group PSs. Note that other background synchronization algorithms introduced in ShadowSync [48] can also be used here.

3.1.2 Parallelism Analysis. Hierarchical training introduces model parallelism within a group as it distributes the operators in a net to different group PSs. At the same time, it also increases the data parallelism degree for a dense replica in a group as HT replaces a trainer with a group of nodes which results in more parallelism. HT also implies pipeline parallelism as group PSs in a group can be viewed as pipeline workers each responsible for the computation of a certain part of the model. They also work on different batches of the data concurrently due to the data parallel execution.

3.2 System Implementation

Now we address the following questions for hierarchical training: (1). How do we realize hierarchical training into the existing system? (2). How do we decide the net/graph partitioning for nodes in a group, i.e., which operators should be on trainers and which operators should be on group PSs? (3). As hierarchical training introduces extra communication overhead within a group, how do we minimize the communication overhead?

We answer these questions through a two-layer system design for hierarchical training as shown in Figure 6. The first layer is the tagging system for operator placement on group PSs and trainers. It takes in a user config and generates a single tagged train net. Each operator in the train net is tagged by a *node_name* field, indicating which device this operator will be run on. The second layer is the net transformation system which takes the tagged train net from the tagging system as the input, applies net transformation on the net and finally generates the distributed nets for each device, e.g., trainers, group PSs, dense PSs, etc., in the system. We describe the

hierarchical training implementation in the Caffe2 stack [3] while the design is general and can be applied to other commonly used deep learning frameworks, e.g., PyTorch [32], Tensorflow [1].

3.2.1 The Tagging System. The tagging system decides the operator placement for operators in the train net. Specifically, we need to decide (1). whether an operator should be placed on trainers or group PSs; and (2). if an operator is going to be placed on group PS, which group PS it should be placed on. We consider the placement in one group as the placements for all groups are identical. The goal of the tagging system is to generate an operator placement strategy such that: (1). the communication between trainers and group PSs is minimized; (2). the computation on trainers and group PSs are balanced; (3). trainers and group PSs will not run out of memory.

The problem can be modeled as a mixed integer linear programming problem and can be solved by a solver like Xpress [43]. However, the train net can have up to thousands of operators and solving it using the solver is infeasible. To give a relatively reasonable solution, we develop a rule-based tagging process based on prior knowledge of the models and a greedy refinement process to refine the operator placement from the rule-based process.

The rule-based tagging process works as the followings. We consider the operator placement problem as a bi-partition problem to put operators on trainers or group PSs. First of all, we tag all operators related to the initialization of the parameter weights, e.g., FC weights and bias, to make the initialization run on the group PSs. This step ensures that there is only one copy of the parameters in each group. We also tag all the related operators in the same modules, e.g., the corresponding FC operators and the FCGradient operators. Second, based on prior knowledge of the deep learning recommendation models, we develop several rules to tag some operators on the trainers. Third, we run a connected component detection algorithm on the train net to identify several connected components, each corresponds to a set of operators running on one group PS. Finally, we allocate those components on different group PSs in a way that balancing the size of the dense weights on each group PSs. The rule-based tagging process is not optimal but gives us a reasonably good initial operator placement. We can improve the placement by plugging in more advanced rules based on the knowledge of the models.

Then we apply a greedy refinement process to further tune the operator placement plan generated by the above rule-based process. The refinement runs as the followings: Given the current placement, it identifies a device which is the bottleneck of the system. A bottleneck device is a device with the highest resource utilization. We consider the following three dimensions for utilization: CPU, network and memory. If one device is the bottleneck, it tries to move one of the operator from the bottleneck device to another device if the move can alleviate the bottleneck of the system. The algorithm runs the above process iteratively until the placement is balanced or a pre-defined search time limit is reached.

The tagging process is a combination of a knowledge-based process and a greedy process but in practice it is sufficient to yield satisfactory operator placement plan. We will show in Section 5.3 about the effectiveness of the tagging process.

3.2.2 The Net Transformation System. We develop a net transformation system to transform a tagged training net into distributed training nets running on different hosts. The primary goal here is to partition some parts of the nets to express model parallelism, and replicate some other parts to express data parallelism. The tagging system adds annotations/tags to the train net. With the annotations, the net transformation framework can infer the model parallelism and data parallelism schemes used for different parts of the net. For computations that are related to embedding lookups, the corresponding operators will be partitioned and placed to the available sparse PSs. Hierarchical training architecture expresses model parallelism within a group, and data parallelism among the groups. Therefore, we need to partition the train nets and place the operators to the group PSs based on the tagged train net. Then for the remaining operators that happen on the trainer, we replicate them and place them on all trainers. For the ShadowSync EASGD synchronization, we create one net per group PS that iterates over all the parameters owned by the group PSs and syncs them with the dense PSs. Once the distributed nets for one group are created, we replicate the distributed nets to each group with small manipulations to place the nets to the correct hosts.

The net transformation system is a generic design which allows us to define different combinations of model and data parallelism, and execute partitioned or replicated nets on different hosts.

4 SYSTEM OPTIMIZATIONS

In this section, we introduce several optimizations specific to HT.

4.1 Model Architecture Optimization

We first identify potential improvement by modifying the models. The fully-connected layer (FC) is one of the most common layers in the dense part of the DLRM-like recommendation models. It is represented by the FC Op and FCGradient Op in the forward and backward computation respectively which are similar to matrix multiplication [3], i.e., $\mathbf{AB} = \mathbf{C}$ where $\mathbf{A}$ is the input matrix with size $input\_dim \times hidden\_dim$, $\mathbf{B}$ is the weight matrix with size $hidden\_dim \times output\_dim$ and $\mathbf{C}$ is the output. To better parallelize the FC, we introduce two schemes for FC splits: vertical split and horizontal split. In vertical split, we split the weight matrix vertically, and the final result of the FC is the concatenation of the sub matrix multiplication. In horizontal split, we split both the input matrix and the weight matrix along the $hidden\_dim$. The result is the sum of the sub matrix multiplication. Both schemes of FC split help with paralleling the computation on group PSs and each sub matrix operation can be placed on one group PS.

We also apply co-partitioning for two consecutive FCs to avoid unnecessary traffic. If the $output\_dim$ is larger than $hidden\_dim$ for the first FC weight and the $hidden\_dim$ is larger than the $output\_dim$ of the second FC weight, we apply co-partitioning to the two consecutive FCs. The first FC is better to use vertical split while the second one is better to use horizontal split. Before any optimization, the first FC will be distributed to different group PSs and the results will be sent back to trainers for aggregation (e.g., concat), and then for the second FC, we redistribute the input (i.e., the output of the previous FC) to different group PSs. To avoid the unnecessary traffic, we enable co-partitioning of the two FCs so that the intermediate results between the two FCs do not need to be sent back to trainers, and thus saves precious network bandwidth.

4.2 Communication Compression

We employ two techniques to reduce the network traffic introduced by the operator partitioning in Hierarchical Training [16] - ***Quantization*** can typically reduce the traffic by 2x-4x depending on the bit precision. We found that using specialized formats like bfloat16 can compress communication with almost no impact to model accuracy in lieu of increased latency of quantization and dequantization operations. Lower bit-precision like INT8 and INT4 can typically provide more compression but also comes at the cost of model accuracy. We observed that gradient information sent between sub-networks is extremely sensitive to lower bit precision, and hence INT8 is only applied to layer activations in the forward pass, while gradients are quantized to bfloat16. ***Sparsification*** is another technique we employ that further boosts the EPS by zeroing out low magnitude values in gradients. This introduces bias in the system which can be compensated by employing error correction techniques [24].

Another important lever that helps improve the efficiency of hierarchical training is the batch size for computation. Larger batches lead to more computation during each batch of training, thereby boosting EPS and CPU utilization. But this also increases the network communication which scales linearly with the number of layer activations. Hence communication compression and large batch can organically complement each other in boosting EPS.

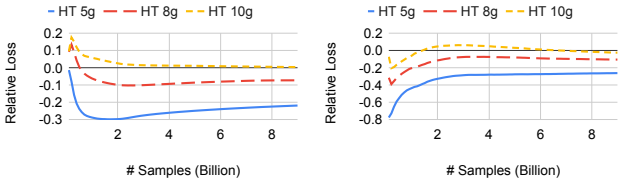
4.3 System Optimizations for Scaling

Hierarchical training increases the number of distributed nodes from dozens to more than 200. This adds pressure on the underlying infrastructure to meet end-to-end training latency guarantees while maintaining high reliability. We applied several optimizations to improve the model publishing performance, memory management scheme, and the overall fault-tolerance strategy.

During online training, models incrementally train on fresh data and publish new inference files at an interval of two hours. The models stop training to save inference files to disk with low latency and resumes training quickly to avoid loss of training data. A background thread uploads the files from disk to remote storage to serve predictor traffic. Hierarchical training allows training of larger model which increases the write latency of inference files to local disk and can lead to training data loss. Moreover, a lower end-to-end publishing latency between two consecutive snapshots is required to avoid model accuracy degradation from over-fitting. We optimized our model publishing logic to directly write to remote storage and avoid the slow temporary write to local disk.

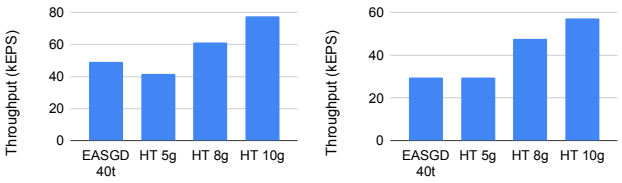
Hierarchical training executes complex computations inside group PS and trainers. Dynamic memory of these hosts are utilized while training large models which places heavy computational components on a single node. Moreover, sparse PS host large embedding tables, making these hosts susceptible to run out of memory. We utilize a detection framework to identify fused components which cannot be sharded across trainer and group PS and fail early. We also deploy an auto-tuning framework with dynamic memory accounting to estimate the memory requirements of sparse PS and shard large embedding tables across multiple nodes.

Training at scale with large number of nodes increases the surface area for failures due to communication challenges between nodes and routine maintenance of hosts. We use checkpoint-based failure recovery strategy for fault tolerance and increase the number of retries. We also implement a controller to defer the host maintenance events to much later so that we can group multiple events and perform maintenance on the hosts altogether.



(a) Model-A (b) Model-B

Figure 7: Relative loss compared with 40 trainers EASGD



(a) Model-A (b) Model-B

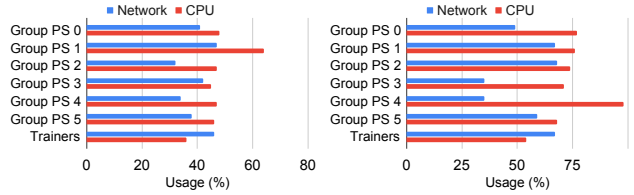
Figure 8: Training throughput (EPS)

5 EXPERIMENTS

In this section, we conduct experiments to evaluate the effectiveness of hierarchical training on DLRM-like models for click-through-rate prediction tasks. We use internal models and dataset for the experiments. The models consist hundreds of sparse categorical features and the dense part is of several hundreds megabytes. We will omit the detailed descriptions for the models and data. Hyper-parameters, e.g., the learning rate, batch size, etc., are the same for all the experiments. For all the comparisons except those in Section 5.3, we employed all system optimization techniques introduced in Section 4. We use enough sparse PSs and readers for all jobs to ensure they are not the system bottleneck. The total number of nodes (gang size) we used ranges from 100 to 200.

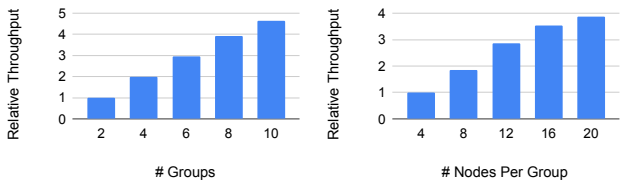
5.1 HT vs. EASGD

First of all, we demonstrate the model quality and training throughput improvement for hierarchical training (HT) compared to the EASGD-based training system. We take two models: **Model-A** and **Model-B**, where Model-A uses full precision computations on fully-connected layers, and Model-B uses 8-bit low precision computations for FC layers on the forward pass. Both scenarios are commonly used in real world recommendation model training. Both models are trained using 9 billion samples. The loss we use is the normalized entropy (NE) loss between click-through-rate prediction probability and the true labels. The detailed description of the loss can be found in [19]. We use averaged EPS (examples



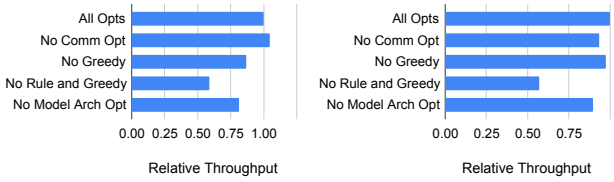
(a) Model-A **(b) Model-B**

Figure 9: CPU and Network Utilization



(a) Inter-group Scalability **(b) Intra-group Scalability**

Figure 10: Scalability



(a) Model-A **(b) Model-B**

Figure 11: Effect of Optimizations

per second) throughout the training to measure the throughput of the system. Note that the EPS is stable during training.

In each of the two models, we use EASGD with 40 trainers (*EASGD 40t*) as the baseline and compare it with hierarchical training (HT) with 5, 8, 10 groups (*HT 5g*, *HT 8g*, *HT 10g*), where each group contains 6 trainers and 6 group PSs. We do not consider using more nodes in HT as an efficiency loss because sparse PSs and readers already take up more than half of the total nodes.

We first evaluate the *relative loss* (the lower the better) for HT runs compared with 40 trainers EASGD baseline over the training process in Figure 7. HT versions achieved neutral to better loss compared with the baseline. The gain is larger when fewer groups, e.g., 5 groups, are used in HT. This is because fewer dense replicas can yield to better model quality as also observed in Section 2. Specifically, with 5 groups, HT achieves more than 0.2% model quality gain compared with the baseline which is significant.

Figure 8 shows the corresponding training throughput (EPS) for HT and EASGD baseline. HT with 10 groups is 1.6x and 1.9x of the EPS compared with the EASGD baseline for Model-A and Model-B respectively. Also note that with 10 groups, HT achieves neutral model quality for both models as shown in Figure 7. This result demonstrates that HT can speedup the training by up to 1.9x while at the same time preserving the model quality. From another perspective, it also shows that with similar training throughput, HT can achieve up to 0.2% loss improvement compared with the EASGD-based system.

We also report the CPU and network utilization for group PSs and trainers for the HT 10 groups case on both models in Figure 9. The utilization of group PSs from different groups are similar so we only plot the utilization for the first group. Different trainers also have similar utilization because they are identical, and we only plot the averaged usage for trainers. Utilization for HT 5 groups and 8 groups are similar with HT 10 groups because changing the number of groups does not change the utilization of the group PSs and trainers within a group. The result suggests that for Model-A, the CPU and network utilization for the training nodes (group PSs and trainers) are relatively balanced, and there is no obvious bottleneck. This demonstrates the effectiveness of our operator placement strategy, including the rule-based tagging and greedy refinement process. For Model-B, the CPU and network are more utilized, and more specifically, the CPU on group PS 4 becomes a bottleneck. Further improving the efficiency of hierarchical training will be left as future work.

5.2 Scalability

In this section, we measure the scalability of hierarchical training on Model-A. We first evaluate the *inter-group* scalability by increasing the number of groups in HT as shown in Figure 10a. Each group has 6 trainers and 6 group PSs. We report the relative speedup compared with the 2 groups setting. The result shows that HT can scale linearly to 10 groups as long as other system components do not become the bottleneck. This is expected as similar to the EASGD-based system, the model averaging for the dense part is running in the background and allows HT to increase the training throughput almost linearly with more groups.

We then evaluate the effect of changing the (# trainers, # group PSs) pair in each group from $(2, 2)$ to $(10, 10)$ in an 8-group HT run on Model-A and measure the averaged EPS. Figure 10b shows the relative speedup compared with the (2 trainers, 2 group PSs) setting. We found that when we have more nodes per group, the model split becomes worse and the communication traffic among trainers and group PSs increased. Thus, the EPS grows sublinearly when increasing the nodes per group. A reasonable number of nodes per group could be around 12 (6 trainers and 6 group PSs).

5.3 The Effectiveness of System Optimizations

Finally, we evaluate the effectiveness of various optimization techniques for hierarchical training mentioned in Section 3.2 and Section 4. In particular, we look at the following techniques in both model-A and model-B: (1) greedy refinement tagging process; (2) rule-based tagging + greedy refinement; (3) model architecture optimization, and (4) communication compression. We remove each of them independently from the HT baseline and measure the EPS. In both model-A and model-B experiments, we use 5 groups with 6 trainers and 6 group PSs in each group.

Figure 11a and Figure 11b show that the tagging strategy is important to the efficiency of HT. Disabling both the rule-based tagging and greedy refinement can reduce the EPS by more than

40%. The greedy refinement can improve the efficiency considerably if the rule based tagging strategy is not good enough, e.g., for Model-A, but gives little improvement if the rule-based process is already good, e.g., for Model-B. Model architecture optimizations can also boost the EPS as they help model parallelism to achieve a more balanced computation/communication among training nodes in a group. Finally, communication compression does not always help. The reason is that the compression/decompression operators also take some CPU cycles and it might outweighs the benefit of less communication sizes, e.g., for Model-A. This indicates that taking the CPU cycles for compression/decompression into considerations in the tagging process can potentially further improve the training efficiency. We leave this as future work.

6 RELATED WORK

Models for Recommendation Systems. The recommendation models has evolved from large scale logistic regression [18] and factorized machine [13, 28, 34] to more advanced neural network such as DLRM [31] form Facebook and Deep & Wide [6] from Google. Baidu and Alibaba use a similar architecture for ads recommendation [47] and product recommendation [49] respectively. Youtube, Netflix, Microsoft, etc, also report similar architecture for various personalization workloads [7, 9, 10]. At Facebook, DLRM-like models are powering the majority of the recommendation products including instagram story ranking, news feed ranking, and group recommendation, etc [2, 14, 15, 17]. Our previous work provides an in-depth analysis about model architecture, hardware and system configurations [2]. DLRM-like models are also widely used in recommendation models research [12, 26, 37].

Parallelism Schemes. *Data Parallelism* is a commonly used scheme for distributed training which partitions the input dataset among multiple devices. Each device holds a replica of the model and the model are synchronized using parameter server [1, 8, 21, 22, 27, 42] or collective communication primitive [11, 40]. *Model Parallelism* splits the model into different devices to achieve parallelism, e.g., STRADS [25], NOMAD [44, 45] and Mesh-Tensorflow [35]. PipeDream [30], GPipe [20] adopt *pipeline parallelism* and partition the model into multiple devices and splits a batch of data into mini batches to fully utilize all the devices in the pipeline. FlexFlow [23], Tofu [39] and TensorOpt [4] employ different strategies to discover the parallelism schemes for different operators automatically. Hierarchical training combines data parallelism and model parallelism in the execution of each group, and utilizes a two-phase tagging strategy for operator placement.

Parameter Sychronization. Parameter server architecture stores the model on distributed parameter servers and allows multiple workers to update the parameters in RPC-like style, e.g, Parameter Server [27], Petuum [42], DistBelief [8], etc. At Facebook, we also use a dedicated set of sparse parameter servers to host the embeddings for recommendation models [2]. Hogwild! [33] exploits the sparsity of data and lets multiple threads update the shared model in one machine in a lock-free manner. Another line of work focuses on model averaging algorithms [51], e.g., EASGD [46], BMUF [5]. Decentralized parallel SGD [29] relies on peer-to-peer communication to get rid of the centralized parameter servers. ShadowSync [48] makes the model averaging happen in the background and avoid the synchronization blocking the training process. Hierarchical training extends ShadowSync across groups and uses Hogwild!-style update within each group.

7 CONCLUSION

Scaling deep learning based recommendation models training without sacrificing the model quality is challenging. In this paper, we provide an in-depth analysis of the challenges and identify that the model quality is more sensitive to inter-trainer asynchronicity than intra-trainer asynchronicity. Based on these observations, we propose **Hierarchical Training** to scale the real-world recommendation models. We address the operator placement problem, develop optimization strategies for communication reduction, and integrate hierarchical training into our existing training stack. Our experiments verify the effectiveness of hierarchical training on real-world models. Our work demonstrates that a thorough understanding of the system asynchronicity and a design that leverages the characteristic of the models could significantly boost training scalability on large CPU clusters.

8 ACKNOWLEDGEMENT

We would like to thank Lin Yang, Chunzhi Yang, Xi Tao, Bangsheng Tang, Yunchen Pu, Ximing Chen, Huayu Li, Chonglin Sun, Ashwin Bahulkar, Shuai Yang, Xi Liu, Sherman Wong, Tristan Rice, Wenqi Cao, Hassan Eslami, James March, Jeff Kerzner, Dianshi Li, Isabel Gao, Ashot Melik-Martirosian, Joyce Xu, Carole-Jean Wu, Max Leung, Amit Nagpal, Bingyue Peng, John Bocharov, Rocky Liu, Wenlin Chen, Yantao Yao, Shuo Chang, Jason Chen, Liang Xiong, Hagay Lupesko, Stanley Wang, Shri Karandikar, Mohamed Fawzy for the helpful discussions and consistent support.

REFERENCES

[1] Martín 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.

[2] Bilge Acun, Matthew Murphy, Xiaodong Wang, Jade Nie, Carole-Jean Wu, and Kim Hazelwood. 2020. Understanding Training Efficiency of Deep Learning Recommendation Models at Scale. *arXiv preprint arXiv:2011.05497* (2020).

[3] Caffe2 Operators Catalog. 2021. https://caffe2.ai/docs/operators-catalogue.html. (2021).

[4] Zhenkun Cai, Kaihao Ma, Xiao Yan, Yidi Wu, Yuzhen Huang, James Cheng, Teng Su, and Fan Yu. 2020. TensorOpt: Exploring the Tradeoffs in Distributed DNN Training with Auto-Parallelism. *CoRR* abs/2004.10856 (2020). arXiv:2004.10856 https://arxiv.org/abs/2004.10856

[5] Kai Chen and Qiang Huo. 2016. Scalable training of deep learning machines by incremental block training with intra-block parallel optimization and blockwise model-update filtering. In *2016 ieee international conference on acoustics, speech and signal processing (icassp)*. IEEE, 5880–5884.

[6] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. 2016. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*. 7–10.

[7] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*. 191–198.

[8] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. 2012. Large scale distributed deep networks. *Advances in neural information processing systems* 25 (2012), 1223–1231.

[9] Ali Mamdouh Elkahky, Yang Song, and Xiaodong He. 2015. A multi-view deep learning approach for cross domain user modeling in recommendation systems. In *Proceedings of the 24th International Conference on World Wide Web*. 278–288.

[10] Carlos A Gomez-Uribe and Neil Hunt. 2015. The netflix recommender system: Algorithms, business value, and innovation. *ACM Transactions on Management*

Information Systems (TMIS) 6, 4 (2015), 1–19.

[11] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv:1706.02677* (2017).

[12] Hui Guan, Andrey Malevich, Jiyan Yang, Jongsoo Park, and Hector Yuen. 2019. Post-training 4-bit quantization on embedding tables. *arXiv preprint arXiv:1911.02079* (2019).

[13] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. 2017. DeepFM: a factorization-machine based neural network for CTR prediction. *arXiv preprint arXiv:1703.04247* (2017).

[14] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S Lee, David Brooks, and Carole-Jean Wu. 2020. Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 982–995.

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

[16] Vipul Gupta, Dhruv Choudhary, Ping Tak Peter Tang, Xiaohan Wei, Xing Wang, Yuzhen Huang, Arun Kejariwal, Kannan Ramchandran, and Michael W. Mahoney. 2020. Fast Distributed Training of Deep Neural Networks: Dynamic Communication Thresholding for Model and Data Parallelism. *CoRR* abs/2010.08899 (2020). arXiv:2010.08899 https://arxiv.org/abs/2010.08899

[17] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. 2018. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 620–629.

[18] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, et al. 2014. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*. 1–9.

[19] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, and et al. 2014. 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, New York, NY, USA. https://doi.org/10.1145/2648584.2648589

[20] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2018. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *arXiv preprint arXiv:1811.06965* (2018).

[21] Yuzhen Huang, Tatiana Jin, Yidi Wu, Zhenkun Cai, Xiao Yan, Fan Yang, Jinfeng Li, Yuying Guo, and James Cheng. 2018. FlexPS: Flexible Parallelism Control in Parameter Server Architecture. *Proc. VLDB Endow.* 11, 5 (2018), 566–579. https://doi.org/10.1145/3187009.3177734

[22] Yuzhen Huang, Xiao Yan, Guanxian Jiang, Tatiana Jin, James Cheng, An Xu, Zhanhao Liu, and Shuo Tu. 2019. Tangram: bridging immutable and mutable abstractions for distributed data analytics. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 191–206.

[23] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2018. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358* (2018).

[24] Sai Praneeth Karimireddy, Quentin Rebjock, Sebastian Stich, and Martin Jaggi. 2019. Error Feedback Fixes SignSGD and other Gradient Compression Schemes. In *International Conference on Machine Learning*. 3252–3261.

[25] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A Gibson, and Eric P Xing. 2016. Strads: A distributed framework for scheduled model parallel machine learning. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–16.

[26] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. 2019. Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 740–753.

[27] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 583–598.

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

[29] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. 2018. Asynchronous decentralized parallel stochastic gradient descent. In *International Conference on Machine Learning*. PMLR, 3043–3052.

[30] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 1–15.

[31] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *arXiv preprint arXiv:1906.00091* (2019).

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*. 8026–8037.

[33] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *Advances in neural information processing systems* 24 (2011), 693–701.

[34] S. Rendle. 2010. Factorization Machines. In *2010 IEEE International Conference on Data Mining*. 995–1000. https://doi.org/10.1109/ICDM.2010.127

[35] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. 2018. Mesh-tensorflow: Deep learning for supercomputers. *arXiv preprint arXiv:1811.02084* (2018).

[36] Hao-Jun Michael Shi, Dheevatsa Mudigere, Maxim Naumov, and Jiyan Yang. 2020. Compositional embeddings using complementary partitions for memory-efficient recommendation systems. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 165–175.

[37] Qingquan Song, Dehua Cheng, Hanning Zhou, Jiyan Yang, Yuandong Tian, and Xia Hu. 2020. Towards automated neural interaction discovery for click-through rate prediction. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 945–955.

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *arXiv preprint arXiv:1706.03762* (2017).

[39] Minjie Wang, Chien-chin Huang, and Jinyang Li. 2019. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–17.

[40] Yidi Wu, Kaihao Ma, Xiao Yan, Zhi Liu, Zhenkun Cai, Yuzhen Huang, James Cheng, Han Yuan, and Fan Yu. 2021. Elastic Deep Learning in Multi-Tenant GPU Clusters. *IEEE Transactions on Parallel and Distributed Systems* (2021).

[41] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2017. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1492–1500.

[42] Eric P Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. 2015. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data* 1, 2 (2015), 49–67.

[43] Xpress Optimization. 2021. https://www.fico.com/en/products/fico-xpress-optimization. (2021).

[44] Fan Yang, Fanhua Shang, Yuzhen Huang, James Cheng, Jinfeng Li, Yunjian Zhao, and Ruihao Zhao. 2017. Lftf: A framework for efficient tensor analytics at scale. *Proceedings of the VLDB Endowment* 10, 7 (2017), 745–756.

[45] Hyokun Yun, Hsiang-Fu Yu, Cho-Jui Hsieh, SVN Vishwanathan, and Inderjit Dhillon. 2013. Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. *arXiv preprint arXiv:1312.0193* (2013).

[46] Sixin Zhang, Anna E Choromanska, and Yann LeCun. 2015. Deep learning with elastic averaging SGD. *Advances in neural information processing systems* 28 (2015), 685–693.

[47] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. 2020. Distributed Hierarchical GPU Parameter Server for Massive Scale Deep Learning Ads Systems. *arXiv preprint arXiv:2003.05622* (2020).

[48] Qinqing Zheng, Bor-Yiing Su, Jiyan Yang, Alisson Azzolini, Qiang Wu, Ou Jin, Shri Karandikar, Hagay Lupesko, Liang Xiong, and Eric Zhou. 2020. ShadowSync: Performing Synchronization in the Background for Highly Scalable Distributed Training. *arXiv preprint arXiv:2003.03477* (2020).

[49] Guorui Zhou, Na Mou, Ying Fan, Qi Pi, Weijie Bian, Chang Zhou, Xiaoqiang Zhu, and Kun Gai. 2019. Deep interest evolution network for click-through rate prediction. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 5941–5948.

[50] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. 2018. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1059–1068.

[51] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. 2010. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*. 2595–2603.