ShadowSync: Performing Synchronization in the Background for Highly Scalable Distributed Training

Qinqing Zheng*[1] Bor-Yiing Su[2] Jiyan Yang[2] Alisson Azzolini[2] Qiang Wu*[3] Ou Jin*[4]
Shri Karandikar[2] Hagay Lupesko[2] Liang Xiong[2] Eric Zhou[2]

[1]*University of Pennsylvania*
[2]*Facebook*
[3]*Horizon Robotics*
[4]*Cruise*

June 30, 2020

Abstract

Ads recommendation systems are often trained with a tremendous amount of data, and distributed training is the workhorse to shorten the training time. Meanwhile, a commonly used technique to prevent overfitting in Ads recommendation is **one pass training**. In this scenario, the total amount of data is fixed. When we express data parallelism on n workers, each worker only processes $1/n$ data. The larger the number of workers, the less data each worker observes. While the training throughput can be increased by simply adding more workers, it is also increasingly challenging to preserve the model quality.

To address this problem, we propose the **ShadowSync** framework, in which the model parameters are synchronized across workers, yet we isolate synchronization from training and run it in the background. In contrast to common strategies including synchronous SGD, asynchronous SGD, and model averaging on independently trained sub-models, where synchronization happens in the foreground, ShadowSync synchronization is neither part of the backward pass nor happens every k iterations.

ShadowSync is simple but effective. Our framework is generic to host various types of synchronization algorithms, and we propose 3 approaches under this theme. The superiority of ShadowSync is confirmed by experiments on training deep neural networks for click-through-rate prediction. Our methods all succeed in making the training throughput linearly scale with the number of trainers. Comparing to their foreground counterparts, our methods exhibit neutral to better model quality and better scalability when we keep the number of parameter servers the same. In our training system which expresses both replication and Hogwild parallelism, ShadowSync also accomplishes the highest example level parallelism number comparing to the prior arts.

1 Introduction

Improving model quality is a race that never ends. In order to accomplish the goal, machine learning practitioners often train with more and more data, use more and more features, or innovate on the model

*These authors contributed to this work while they were working at Facebook.

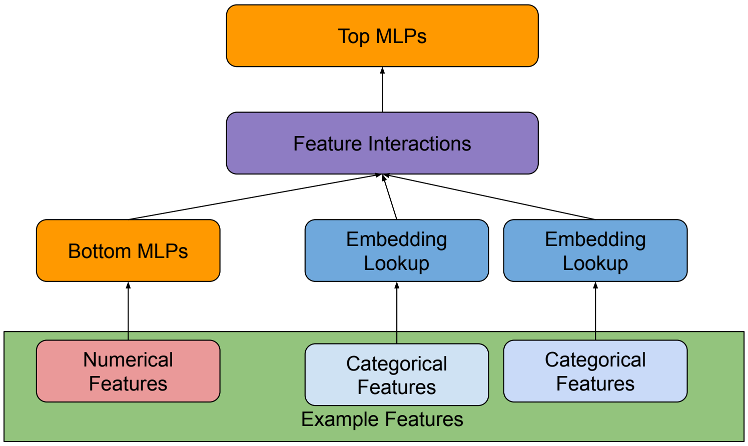


Figure 1: Overview of the DLRM [18] model architecture.

architecture to capture more meaningful interactions among the features. However, both increasing data and increasing model complexity have direct impact on the training speed. As a result, to finish the training job in a reasonable amount of time, distributed training becomes inevitable for training complicated models on large dataset.

Unfortunately, distributed training is extremely challenging. In practice, most of the training algorithms are based on stochastic gradient descent (SGD) [3]. However, SGD is a sequential algorithm, in which we need to process each batch sequentially. When we express parallelism at the batch level and coordinate multiple workers to train one batch, the sequential update assumption is satisfied. This is the common synchronous training algorithm. But the scalability of the algorithm is bounded by the largest batch size we can use without sacrificing the model quality. When we express parallelism on the updates and allow different workers to update the parameters concurrently, we are breaking the sequential update assumption and maintaining good model convergence is difficult. With that, there are many works that propose various ideas to improve training speed while preserving model convergence quality. These ideas are based on different synchronization strategies, to define how different workers synchronize with each other and update the parameters.

Distributed training for a *Deep Learning Recommendation Model* (DLRM) [18] is even more challenging; see Figure 1 as an illustration of the DLRM model architecture. This is because in general one-pass training is used for the DLRM model as it is easy to see overfitting when training with multiple passes [27]. With that, the amount of training data is fixed, and we cannot iterate the training data again to further optimize the model. With this very strict setting, when we express data parallelism on n workers, each worker only processes $1/n$ data. The larger the number of workers, the less data each worker observes and the harder synchronization becomes. Therefore, increasing the scalability while preserving the model quality becomes extremely hard.

To the best of our knowledge, all the existing synchronization algorithms have incorporated synchronization in the training process. Section 2 summarizes the state-of-the-art synchronization algorithms in more details. However, having synchronization as part of the training process tends to make it an overhead in training. Usually to ensure good training convergence, we need to synchronize often. The more often we synchronize, the more time we spend on synchronization and hence it increases the end-to-end training time. This can be manifested by the fact that there are much active research attention on quantizations and gradient sparsification in order to reduce the synchronization overhead. For example, the ternary gradient [22], deep compression [11] and one-bit quantization [20] works claim 3x, 4x and 10x training speedup by reducing the communication cost.

In this work, we propose the **ShadowSync** framework to perform synchronization in the background, so that the synchronization does not interfere the foreground training process. With that, we are able to achieve linear scalability in terms of `EPS` (Definition 1). We have also empirically validated that the model quality is on par with or better than the case when we sync in the foreground.

Definition 1 (Examples Per Second)**.** *We define Examples Per Second (*`EPS`*) as the number of examples per second processed by the distributed training system. This is the primary metric we will use to measure the training performance.*

Even though the **ShadowSync** idea is widely applicable to any model architectures, our research focuses on click-through-rate prediction[17] models that are similar to the DLRM [18] architecture. So in this paper we will focus on illustrating how to integrate the ShadowSync framework to the distributed training system that can train DLRM-like models. In our distributed training system, we express both model parallelism and data parallelism for training, and hogwild parallelism [19] and replication parallelism for optimization. With that, we are able to get the highest `ELP` (Definition 2) number among all the prior arts to the best of our knowledge.

Definition 2 (Example Level Parallelism)**.** *We define Example Level Parallelism (*`ELP`*) as the maximum number of examples processed by the distributed training system concurrently at any point of training time.*

The main contributions of this paper are in the following:

1. We introduce a new framework called ShadowSync to synchronize parameters in the background. The framework is generic to host various synchronization algorithms. In essence, it splits the duty of synchronization and training and thus is flexible to accommodate new algorithms in the future.

2. We propose the ShadowSync EASGD, ShadowSync BMUF, and ShadowSync MA algorithms which all sync parameters in the background. This shows how simple it is to develop new synchronization algorithms in the ShadowSync framework.

3. We empirically demonstrate that the ShadowSync idea enables us to scale training `EPS` linearly because training is not interrupted by syncing. When we increase the scale of training, we see the ShadowSync algorithms outperform the foreground variants in both the relative error changes and the absolute error metrics.

4. We compare the ShadowSync algorithms, and conclude that all of them have the same training `EPS` , while ShadowSync EASGD has slightly better quality, and ShadowSync BMUF/MA consume fewer compute resources because they do not need the extra sync parameter servers.

5. We compare the `ELP` for our distributed training system with other state-of-the-art works, and claim that we can accomplish the highest `ELP` among all the distributed training works to the best of our knowledge.

2 Related Work

Synchronization strategy is one of the most important factors affecting model convergence in distributed training. Therefore, this is an area that the researchers are actively innovating on. We can classify the existing synchronization algorithms into two primary variants – synchronous SGD algorithms and asynchronous SGD algorithms.

Synchronous SGD algorithms [2, 9, 23] have become one of the most popular optimizer for training neural network models in a distributed training system. One big batch of examples are partitioned into many mini-batches, with one worker works on one or more mini-batches. Gradients computed using the local mini-batch of data on each worker are aggregated using all-reduce collectives and then applied to the parameters in each worker. This procedure is equivalent to performing a single-machine SGD update on the parameter based on the original big batch. The parallelism is expressed at batch level, in which we allow multiple workers to train on one big batch cooperatively. There are no parallel updates to the parameters. This strategy honors the sequential update assumption in the SGD algorithm and in general has better convergence guarantee. Various methods have been proposed to further optimize the collectives or the scheduling of the mini-batches. However, there are still fundamental limitations of the synchronous SGD algorithms: stragglers in the workers will slow down the synchronization, the barriers forced at the synchronization stage introduce extra overhead, and any failures in any workers will affect the other normal workers.

On the other hand, asynchronous SGD algorithms increase the parallelism by removing the dependency among the nodes. It allows training the model with different data shards and updating the parameters in parallel. In a standard asynchronous SGD algorithm, the parameter servers maintain the global parameters. Each worker sends requests to the parameter servers to pull the global parameters, processes a batch of examples to calculate gradients, and sends the gradients back to the parameter servers which update the global parameters accordingly. Famous works in this class include Hogwild! [19] which is a lock-free implementation of asynchronous SGD that can achieve a nearly optimal rate of convergence for certain problems, and the DistBelief [7] framework that utilizes the parameter server and worker architecture. The study of convergence behavior of asynchronous SGD can be found in [4, 25, 26, 6, 10].

As the parameter servers in the asynchronous SGD algorithms can easily become a bottleneck, researchers have proposed new algorithms that replicate the model parameters in each worker and allow the workers to train on their local parameter replicas independently, and insert a synchronization layer to perform parameter averaging to sync the parameter replicas in the workers. The EASGD algorithm [24] uses parameter servers to host the central parameters. Each worker will sync their local parameter replica with the central parameters every k iterations.

Instead of using parameter servers, the model averaging algorithm (MA) [28] utilizes the AllReduce primitive to aggregate the sub-models every k iterations. Contrasted with the EASGD algorithm, the network topology of MA is decentralized. Another example in this category is the blockwise model-update filtering (BMUF) algorithm [5] which first calculates the averaged model parameters, computes the difference between the averaged parameters and the local parameters, and then uses the difference as a gradient to update the local parameters. The Slow Momentum algorithm [21] extends the idea of BMUF and synchronize the workers periodically with the momentum optimizer. The AllReduce primitive introduces huge synchronization overhead though. The asynchronous decentralized parallel SGD [16] and the stochastic gradient push [1] algorithm rely on peer-to-peer communications among the workers, and will perform the gossiping-style synchronization. These methods require less communication as nodes are designed to not rely on the same parameter and instead periodically pull towards each other.

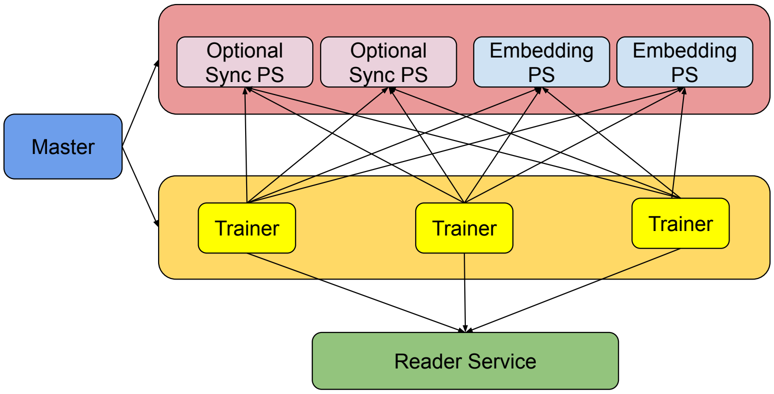


Figure 2: Overview of our distributed training system. For architectures similar to DLRM, the training of embedding layer happens in the embedding parameter servers and model parallelism is performed. The training of interaction & MLP layers happen in the trainers and data parallelism is expressed.

Among all these different algorithms, the synchronization always happens in the foreground. It either is incorporated in the backward pass, or is added every k training iterations.

3 ShadowSync

3.1 System Overview

Our system is designed to support models in a similar architecture as the DLRM [18]. It is capable of expressing model parallelism and/or data parallelism simultaneously, depending on the specific design of the particular model.

The DLRM is composed of three layers of architectures. See Figure 1 as an illustration. The bottom layer contains embedding look-up table, in which the categorical features are transformed into embeddings, and MLP that transfers numerical features to embeddings for the next layer. The middle layer is feature interactions, in which the interactions among the embeddings are defined. The top layer is primarily the multi-layer perceptrons (MLPs). We refer readers to Naumov et al. [18] for the details.

The architecture of the distributed training system is summarized in Figure 2. To map DLRM to the system, we express model parallelism on the embedding look-up layers, and data parallelism on the bottom MLP, feature interaction, and top MLP layers. To express model parallelism on the embeddings, we partition the embeddings into smaller shards to fit the shards into the memory of embedding parameter servers (embedding PS). The embeddings in the embedding parameter servers are shared among all the trainers. To express data parallelism on the rest of the model layers, we replicate the parameters of these layers in each trainer. With that, each trainer will process different batch of examples in parallel, and update its own local copy of the parameters of the bottom MLP, feature interaction, and top MLP layers. We design such architecture because the embedding tables are often gigantic, and cannot fit in the memory of one machine. So we have to partition and serve them in embedding servers. On the other hand, the MLP and interaction layer parameters are often small and can fit in the memory of one machine, so we can replicate them in each trainers and scale `EPS` linearly.

In the system, a master coordinates the overall training process. It assigns different roles (trainers/Embedding PSs/Sync PSs) to the workers, and send training plans to them for execution. The trainers are the workers who control the training loop. Each trainer connects to a shared reader service. It has a local queue that fetches new batch of examples from the reader service. The reader service is a distributed system which consumes the raw data in the distributed storage, and then convert the raw data to feature tensors after the feature engineering step so that the trainers can focus on training without being bottlenecked on the data reading.

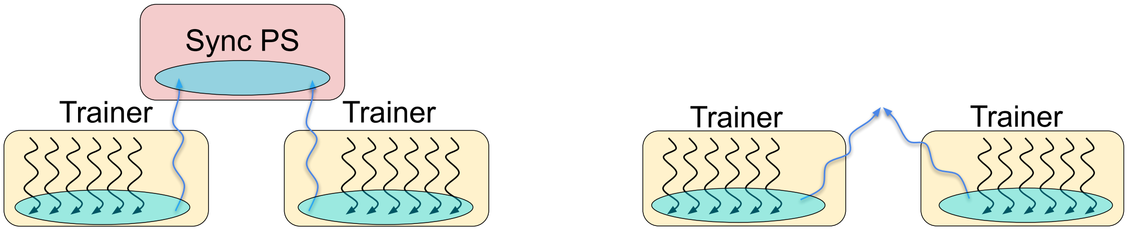
The trainers are multi-threaded and are training multiple batches of examples concurrently. The connection between the trainers and the parameter servers (PS) forms a bipartite graph. Each trainer can connect to each parameter server. Given a batch of examples, one trainer thread sends the embedding lookup requests to the corresponding embedding PSs who host the embedding tables. If one embedding is partitioned into multiple shards and placed on multiple embedding PSs, we will perform local embedding pooling on each PS for the embedding shard, then the partial pooling from the shards will be returned to the trainer to perform the overall embedding pooling to get the final embedding lookup results. Another optimization we have performed on the embedding PSs is to ensure that the workload are distributed evenly among the embedding PSs. We accomplish this goal by profiling the cost of embedding lookup in advance, and then solve a bin packing problem to distribute the workload (the embedding lookup cost) among the embedding PSs (the available bins) evenly. With this optimization, we are able to ensure that the trainers are not bottlenecked by the shared embedding PSs. The sync parameter servers (sync PS) are optional and only exist if we use centralized algorithm, e.g., EASGD, as the synchronization algorithm. To balance the load for the sync PSs, we applied the similar optimizations to profile the costs and then solve the bin packing problem to shard and distribute the parameters to the available sync PSs. Details are explained in Section 3.2.

In brief, the embedding lookup is expressed as model parallelism and is executed in the embedding PSs. After all the embedding lookups are returned, the trainer has all the embeddings and the numerical features needed for current mini-batch. The interaction layers and the MLP layers are thus executed in the trainers. This part is expressed as data parallelism in which each trainer thread is processing its own batch of examples in parallel. Similarly, for the backward pass, the gradient calculation and parameter updates for the MLP and the interaction layers are performed in the trainers. Then the trainers send the gradients to the embedding PSs to update the embeddings.

Even though the system is designed with the DLRM [18] in mind, it is a generic architecture. Basically, we express data parallelism in the trainers, and model parallelism in the PSs. More than that, there can be additional PSs for special purposes (like the sync PSs). This architecture is very similar to the DistBelief architecture [7] and the parameter server architecture [15]. The parameter servers are capable of executing arbitrary operations that are defined by the master. The most common use case is to perform partial embedding pooling. But if more complicated model is expressed, such as adding attention layer to the embedding lookup, we can also perform the required MLP layers on the embedding PSs as well.

3.2 Optimization Strategies

Within a trainer, we have created multiple worker threads to process the example batches in parallel. The simplest idea is to let one thread process one batch of examples. A more complicated parallelism is to explore intra-op parallelism, so that we can use multiple threads for executing the layers for one example batch. But this is beyond the scope of this paper. In this work we will assume that we use one thread to process one example batch. That is, if we have 24 threads, we will process 24 example batches concurrently. Given that we have performed different parallelization strategies for different parts of the model, the optimization strategies for different parts are also different.



(a) ShadowSync for centralized algorithms. The shadow threads will talk to Sync PSs. There is no interaction among the trainers.

(b) ShadowSync for decentralized algorithms. Sync PSs are absent and the shadow threads will communicate with each other.

Figure 4: ShadowSync for data parallelism optimization. The black arrows represent worker threads. They update local replica of parameters in the Hogwild manner. The blue arrows represent shadow threads whose job is synchronization.

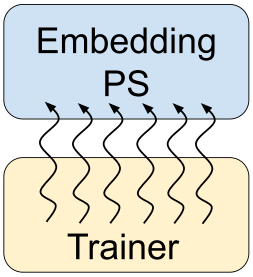


Figure 3: For model parallelism, worker threads optimize the embedding tables using Hogwild.

The embedding tables are big and thus partitioned into many shards and hosted in different embedding PSs. Therefore, there is only one copy of the embedding tables in the whole system. With that, we are using the Hogwild [19] algorithm to optimize the embedding tables. See Figure 3. There is no lock involved in the accesses of the tables. When an embedding PS receives one request from a trainer, it is either doing the embedding lookup or the embedding update in a lock-free way. Every embedding PS has multiple threads so that it can handle many requests in parallel. Different optimization techniques can be used when updating the embeddings, such as Adagrad [8], Adam [14], Rmsprop [13] or other algorithms. All the auxiliary parameters for the optimizers (for example, the accumulation of the squared gradient for the Adagrad optimizer) collocate with the actual embeddings on the embedding PSs.

For the interaction and MLP layers, we express data parallelism on them. The parameters for these layers are replicated across all the trainers. Locally, all the worker threads within one trainer access the intra-trainer shared parameter memory space, and also the shared auxiliary parameters for the optimizer of choice. These accesses are performed in a Hogwild [19] manner as well. So the reads and the updates to the local parameters are lock-free. This strategy has broken the Hogwild assumption that the parameter accesses are sparse. In our case, all the threads are accessing the same parameters in parallel. In practice, this strategy works pretty well and still provides very good model convergence. See Section 4 for the experimental results.

In the model parallelism regime, the worker threads access the shared parameters in the embedding PSs. In contrast, for the interaction and MLP layers, i.e., data parallelism regime, the scope of worker threads is restricted to the local parameter space on individual trainer. In order to synchronize among the trainers, we create one **shadow thread**, who is independent to the worker threads, to carry out the synchronization without interrupting the foreground training. We call this background synchronization framework **ShadowSync**. See Figure 4 for an illustration. Depending on the specific sync algorithm of choice, the shadow threads could either communicate with each other or just with the sync PSs.

This framework has a number of appealing properties. First, as we have separated the duty of training and synchronization into different threads, training is never stalled by the synchronization need. For the computational time and network communication cost, the huge overhead of syncing the parameters is removed from the training loop. As illustrated in experiments in Section 4, when using two sync PSs, syncing

Algorithm	Batch Size	# Hog.	# Rep.	ELP
ShadowSync	200	24	20	**96000**
EASGD [24]	128	1	16	2048
DC-ASGD [26]	128	16	1	2048
BMUF [5]	N.A.	1	64	64 × B
DownpourSGD [7]	N.A.	1	200	200 × B
ADPSGD [16]	128	1	128	16384
LARS [23]	32000	1	1	32000
SGP [1]	256	1	256	65536

Table 1: ELP comparison between ShadowSync and the other optimization algorithms. #Hog. refers to the number of threads who access the shared parameters in a Hogwild fashion. #Rep. refers to the number of replicated parameters in the system.

in the foreground becomes a bottleneck and the training speed plateaued with more than 14 trainers. On the other hand, we are able to scale linearly with the ShadowSync framework on the same setting.

Second, our system is capable of expressing different sync algorithms. For the centralized algorithms like EASGD [24], we need a place to host the central parameters. In our architecture, we have chosen to allocate dedicated sync PSs for this purpose. The central parameters are hosted on the sync PSs, and then the trainers will sync their local replication of the parameters to the sync PSs. The synchronization is a network heavy operation, so we allow partitioning the parameters into shards, and use multiple sync PSs to sync the parameters. We can also support decentralized algorithms like model averaging [28] or BMUF [5], for which we do not need central sync PSs. So we apply the all-reduce collectives to sync among the trainers directly.

Last but important, in the practical realization of our system, the development of sync algorithms can be completely separated from training code. This makes the system easy to modify and experiment, without much engineering effort.

The two-level data parallelism (Hogwild within a trainer and replication across trainers) we have expressed in training helps us to accomplish very high ELP numbers. In Table 1, we have compared the ELP we have accomplished with other state-of-the-art optimization algorithms. In the experimental Section 4, the maximum number of trainers we have used is 20, which seems to be a moderate number. But when we include batch size and the concurrent Hogwild updates, the ELP we are able to accomplish is very high. We have accomplished 96000 ELP with 20 trainers. The batch sizes of the BMUF [5] and the DownpourSGD [7] work are not disclosed in the papers, so their ELP should be the amount of data parallelism they have expressed times B, which is the batch size of training. To the best of our knowledge, the Stochastic Gradient Push [1] work is the best reported distributed training so far. It can scale to 256 GPUs, with each GPU training on a batch of 256 examples. But even with that, the ELP that is accomplished in the paper is 65536. We understand that the model and the datasets are different in all these works, so that the ELP comparisons might not be very representative. We plan to evaluate all these different algorithms on the same model and the same datasets as future work.

3.3 ShadowSync Algorithms

In this section we present the formal algorithmic description of the ShadowSync concept. Three representative algorithms under this framework are introduced, which incorporate the synchronization strategy of EASGD [24], model averaging [28], and BMUF [5] into the execution plan of shadow threads respectively. We call these algorithms *Shadow EASGD*, *Shadow MA* and *Shadow BMUF*.

Let h denote the embedding of categorical features and w denote the weights on MLP and interaction layers. The goal is to minimize the objective function $f_D(w, h)$ defined by the model architecture and training data D. In our framework, assume there are n trainers. Recall there is only one copy of h on the embedding PSs and n replications of w on trainers. Let $w^{(i)}$ denote the replica on trainer i, and $D^{(i)}$ denote the dataset consumed by trainer i. Our system is solving the following optimization problem

$$\min_{w^{(1)},\ldots,w^{(n)},\, h} \sum_{i=1}^{n} f_{D^{(i)}}(w^{(i)}, h), \;\; \text{subject to } w^{(1)}, \cdots, w^{(n)} \text{ in sync.} \tag{1}$$

The constraint in Equation 1 is used to promote the consistency across the weight replicas, and different algorithms use different strategies to derive the sync updates. For example, depending on the topology of the chosen algorithm, the shadow thread on trainer i will sync with replicas on other trainers directly, or indirectly, through a hub copy on the sync PS. When the training ends, one can either output the average of $w^{(i)}$s, select the best replica on a validation dataset, or even simply pick an arbitrary replica.

Algorithm 1: ShadowSync Framework

```
Input: $w_0$, $h_0$
Init embedding tables on embedding PSs: $h \leftarrow h_0$
(Optional) Init MLP & interaction params on sync PSs: $w^{\text{PS}} \leftarrow w_0$
trainer $i$ do in parallel with others
    Init local MLP and interaction param $w^{(i)} \leftarrow w_0$
    worker threads do in parallel
        while data is not all consumed do
            Update $h$ on embedding PSs
            Update local param $w^{(i)}$
    shadow thread do
        while data is not all consumed do
            Sync local param $w^{(i)}$ with Sync PS or other trainers
```

Algorithm 2: Shadow EASGD on Trainer i

```
Input: elastic param $\alpha$
shadow thread do
    while data is not all consumed do
        $w^{\text{PS}} \leftarrow (1 - \alpha) w^{\text{PS}} + \alpha w^{(i)}$
        $w^{(i)} \leftarrow (1 - \alpha) w^{(i)} + \alpha w^{\text{PS}}$
```

Algorithm 1 summarizes the ShadowSync idea. We first initialize the embedding tables by h_0. The initialization of MLP and interaction layers w_0 are fed to all the trainers. If we use centralized algorithms,

Algorithm 3: Shadow MA on Trainer i

```
Input: elastic param $\alpha$, total number of trainers $n$
Init MA global param $w^{\text{global}} \leftarrow w_0$
shadow thread do
    while data is not all consumed do
        $w^{\text{global}} \leftarrow w^{(i)}$                              // make a copy of local param
        $w^{\text{global}} \leftarrow \texttt{AllReduce}(w^{\text{global}})/n$
        $w^{(i)} \leftarrow (1-\alpha)w^{(i)} + \alpha w^{\text{global}}$
```

Algorithm 4: Shadow BMUF on Trainer i

```
Input: step size $\eta$, elastic param $\alpha$, total number of trainers $n$
Init BMUF global param $w^{\text{global}}, w^{\text{copy}} \leftarrow w_0$
shadow thread do
    while data is not all consumed do
        $w^{\text{copy}} \leftarrow w^{(i)}$                                // make a copy of local param
        $w^{\text{copy}} \leftarrow \texttt{AllReduce}(w^{\text{copy}})/n$
        $w^{\text{desc}} \leftarrow w^{\text{copy}} - w^{\text{global}}$                 // compute descent direction
        /* can do momentum update, Nesterov acceleration etc.                  */
        $w^{\text{global}} \leftarrow w^{\text{global}} + \eta w^{\text{desc}}$
        $w^{(i)} \leftarrow (1-\alpha)w^{(i)} + \alpha w^{\text{global}}$
```

the Sync PSs need to be present and be initialized by w_0 too. The worker threads on each trainer will optimize their own local weight and the embedding table in the lock-free manner. In other words, if there are m worker threads spawned per trainer, the embedding h will be updated using nm Hogwild threads across the trainers, and the local copy $w^{(i)}$ will be updated by m Hogwild threads within trainer i. For decentralized algorithms, the update of $w^{(i)}$ will involve copies on other trainers, whereas for centralized algorithms, $w^{(i)}$ will just sync with w^{PS}.

Algorithm 2, 3, 4 describe the synchronization updates of Shadow EASGD, Shadow MA and Shadow BMUF. Contents of worker threads and initialization that are repeating Algorithm 1 are omitted. For MA, each trainer will host an extra copy of weights w^{global}, which is used to aggregate the training results via `AllReduce`. Similarly we have w^{copy} and w^{global} for BMUF, where w^{global} hosts the global model in sync and w^{copy} is used for `AllReduce`. To sync, BMUF defines the difference between the latest averaged model and current w^{global} as the descent direction, then make a step along it. Considering the descent direction as a surrogate gradient, one can incorporate techniques like momentum update and Nesterov acceleration into the updates.

The sync update of Shadow EASGD is essentially the same as original EASGD. Given elastic parameter α, it will do convex interpolation between w^{PS} and $w^{(i)}$. Note that the interpolation is asymmetric: $w^{(i)}$ and w^{PS} are not equal after this update. Intuitively, the PS is in sync with other trainers, and the worker threads didn't stop training, so that both of them would like to trust their copy of weights. Similar interpolation is happening for both Shadow MA and Shadow BMUF. This is a major modification from the original methods. Our experiments have verified it is essential to improve the model quality in the ShadowSync setting. Take MA for example, the `AllReduce` primitive is time-consuming and the worker threads would have consumed a fair amount of data in the `AllReduce` period. If we directly copy the averaged weight

$w^{\mathtt{global}}$ back, we will lose the updates to the local parameter replicas when the background synchronization is happening in parallel.

4 Experiments

We conducted numerical experiments on training a variety of machine learning models for click-through-rate prediction tasks. All the algorithms were applied to training production models using real data. Due to privacy issues, the detailed description of specific datasets, tasks and model architectures will be omitted in this paper, yet we will report the sizes of datasets when presenting the experiments. For readers interested in recommendation models and systems, we refer them to [12, 18] for details. In the sequel, we name the internal models and datasets Model-A to Model-C, and Dataset-1 to Dataset-3, respectively. For simplicity, we refer to the ShadowSync algorithms (see Section 3.3) as S-EASGD, S-BMUF and, S-MA, and refer to the original fixed rate algorithms as FR-EASGD, FR-BMUF, and FR-MA.

As explained in section 1, to prevent overfitting, we use one-pass training for all the experiments. After the training ends, the embedding h and the weights replica $w^{(1)}$ on the first trainer are returned as the output model (this is for simplicity, an alternative is to return the average of all the weight replicas). The hardware configurations are identical and consistent across all the experiments. All the trainers and PSs use Intel 20-core 2GHz processor, with hyperthreading enabled (40 hyperthreads in total). For network, we use 25Gbit Ethernet. We set 24 worker threads per trainer.

Section 4.1 compares ShadowSync scheme to fixed rate scheme in the aspects of the model quality and scalability. As the typical pair of competitors, S-EASGD and FR-EASGD are first picked for this set of experiments. We are interested in answering the following questions:(1) What is the best sync rate of FR-EASGD? What is the average sync rate of S-EASGD, and how does the quality of the model obtained by S-EASGD compare to FR-EASGD? (2) What is the scaling behavior of S-EASGD and FR-EASGD? Could they achieve linear EPS scaling while maintaining model quality? Similar comparison for BMUF and MA algorithms are presented in Section 4.2. Section 4.3 focuses on the comparison of S-EASGD, S-BMUF and S-MA within the ShadowSync framework. S-BMUF and S-MA are typical *de-centralized* algorithms – the usage of sync PSs is eliminated. Those lightweight optimizers are suitable for scenarios where the computation resource is on a tight budget. We are thus curious about whether the performance of S-BMUF and S-MA are on par with S-EASGD. Finally, in Section 4.4 we provide a justification for the choice of 24 Hogwild worker threads in the setup.

4.1 Shadow EASGD vs Fixed Rate EASGD

4.1.1 Model Quality

The very first thing we are interested is to compare the qualities of models returned by S-EASGD and FR-EASGD. We studied their performance on training Model-A on Dataset-1. This dataset comprises $48,727,971,625$ training examples and

$1,001,887,500$ testing examples. The performance of FR-EASGD might be sensitive to the hyper-parameter **sync gap**, which is the number of iterations between two synchronizations. We tested 4 values for it: 5, 10, 30, and 100. We shall use FR-EASGD-5 to denote FR-EASGD with sync gap 5, and similarly for other numbers. To be fair in comparison, all the other hyper-parameters such as elastic parameter, learning rate were set the same as in the production setting, for both S-EASGD and FR-EASGD.

The experiment was first carried out in 11 trainers, 12 embedding PSs and 1 sync PS. Table 2(a) reports the training and evaluation losses obtained. The reported loss is an internal metric used as the objective

	Sync Gap	Train Loss	Eval loss
S-EASGD	5.21	**0.78926**	**0.78451**
FR-EASGD	5	0.78942	0.78483
	10	0.78937	0.78508
	30	0.78942	0.78523
	100	0.78969	0.78531

(a) 11 trainers

	Sync Gap	Train Loss	Eval loss
S-EASGD	1.008	**0.78958**	0.78565
FR-EASGD	5	0.78971	0.78565
	10	0.78977	0.78589
	30	0.7899	**0.78491**
	100	0.79008	0.78557

(b) 20 trainers

Table 2: Model quality of training Model-A on Dataset-1.

value in recommendation models. It is similar to the *normalized entropy* introduced in [12]. We also report the *average sync gap* for S-EASGD, calculated using metrics measured during training:

$$\begin{aligned} \text{avg sync gap} &= \frac{\text{num of iterations trained per sec}}{\text{num of EASGD syncs per sec}} \\ &= \frac{\texttt{EPS}/\text{batch size}}{\text{sync PSs network usage per sec}/\text{size of weight params } w}. \end{aligned} \tag{2}$$

Table 2(a) shows that the evaluation loss of FR-EASGD kept increasing as the sync gap goes up, the smallest gap 5 achieves the lowest evaluation error. The training loss of FR-EASGD does not show any pattern correlated with sync gap. The average sync gap of S-EASGD is 5.21, very close to 5. For both training and evaluation loss, S-EASGD outperforms FR-EASGD over all tested sync gaps.

In practice, a common pain point for distributed optimization is that training at scale could degrade the model convergence and thus hurt the model quality. While 11 trainers is at moderate scale, we compare the performance of S-EASGD to FR-EASGD for the same task using 20 trainers, 29 embedding PSs and 6 sync PSs. The results are reported in Table 2(b). The best sync gap for FR-EASGD was 30. This suggested that the optimal sync rate would vary over different system configurations; one would need to carefully tune this hyper-parameter for FR-EASGD. The average sync gap for S-EASGD is 1.008. The sync gap was extremely low due to we underspecified the compute resources of the reader service. The data reading becomes the bottleneck and the training slows done. The evaluation performance of S-EASGD is slightly worse than the best FR-EASGD, but comparable to FR-EASGD-5. Our interpretation is that, the hyper parameters we use in this experiment favors the case when sync gap is about 30. When we reduce the sync gap, the workers are tightly synced, and the amount of exploration is reduced. Thus the final evaluation results are slightly worse for both the S-EASGD and the FR-EASGD-5 cases. One interesting phenomenon is that FR-EASGD-5 and FR-EASGD-100 has comparable evaluation performance. Our experiments suggested that small sync gap could slow down the convergence in the early stage of training, yet it is beneficial when the training moves towards the end. We conjecture that a time-varying sync gap would be favorable for FR-EASGD under our setting.

4.1.2 Scalability

Another important property of distributed optimization algorithms is the **scalability**. Ideally, as we increase the training scale, we would like to see the EPS to grow linearly as the number of trainers, while the model quality drop is small and tolerable. To explore the scaling behavior, we apply S-EASGD and FR-EASGD to train Model-B on Dataset-2. Dataset-2 is a smaller dataset that contains 3,762,344,639 training

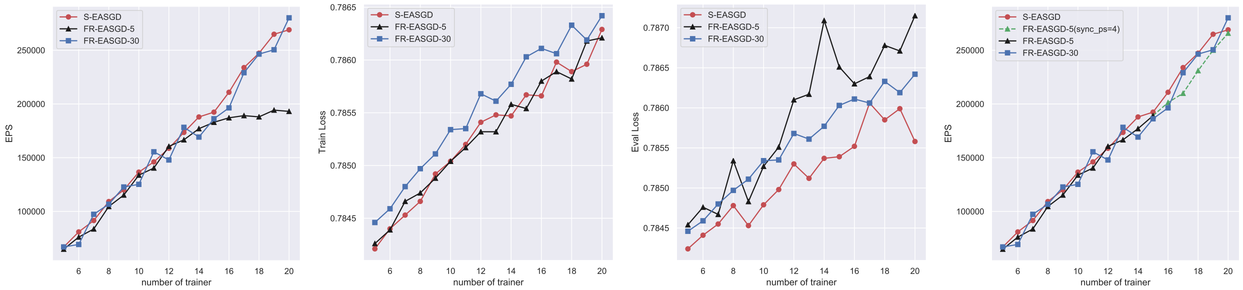


Figure 5: The scaling behavior of S−EASGD and FR−EASGD for training Model−B on Dataset−2. The 1st panel shows the EPS stagnation of FR−EASGD-5. The root cause is that the sync PSs are saturated for high frequency synchronization.

examples and 2,369,568,296 testing examples. We vary the number of trainers from 5 to 20. To ensure enough computing resource, we overspecify the number of embedding PSs to be the same as trainers. The number of sync PSs is fixed to be 2. We tested both FR−EASGD-5 and FR−EASGD-30, since sync gap 5 and 30 are the best results obtained in the previous section. As before, we use the same hyper-parameters for both FR−EASGD and S−EASGD for the sake of fairness.

Figure 5 illustrated how S−EASGD and FR−EASGD trade model quality for data processing speed. Starting from the left, the 1st panel plots EPS as a function of number of trainers. Both S−EASGD and FR−EASGD-30 achieve linear EPS growth. Yet for FR−EASGD-5, its EPS barely increases after the number of trainers goes up to 14. To explain the reason why FR−EASGD-5 reached this plateau, we investigated the hardware utilization of all the machines and identified the sync PSs as the bottleneck. When more trainers are added into training, the network bandwidths of the sync PSs will be saturated at certain point. For FR−EASGD, the synchronization is foreground and integrated into the training loop, hence the network bandwidth needs grow as 24x (the number of worker threads) compared to S−EASGD. When the sync gap is small, the sync PSs can easily get saturated. Increasing the number of sync PSs to 4 solves the problem. See the last panel of Figure 5. We also calculated the average sync gap of S−EASGD as before. For runs with 15 - 20 trainers, the gaps are 8.60, 8.76, 10.43, 10.93, 11.95, and 12.48. This also suggests another strength of S−EASGD comparing to FR−EASGD: it is less demanding for computing resource even for high frequency synchronization.

The 2nd and 3rd panels plot the training and evaluation loss for all the methods. For S−EASGD and FR−EASGD-30, both training and evaluation loss gently increases in comparable speed, with small fluctuations. FR−EASGD-5 is not stable in terms of evaluation loss, and has some spikes in the curve. In addition, S−EASGD demonstrated the best generalization property. Its evaluation losses are the lowest everywhere.

For each method, we also calculate the relative increase [1] of losses when the number of trainers is 10 and 20, comparing with the 5-trainer case. The results are summarized in Table 3. Clearly, S−EASGD enjoys the mildest loss increase, especially for evaluation.

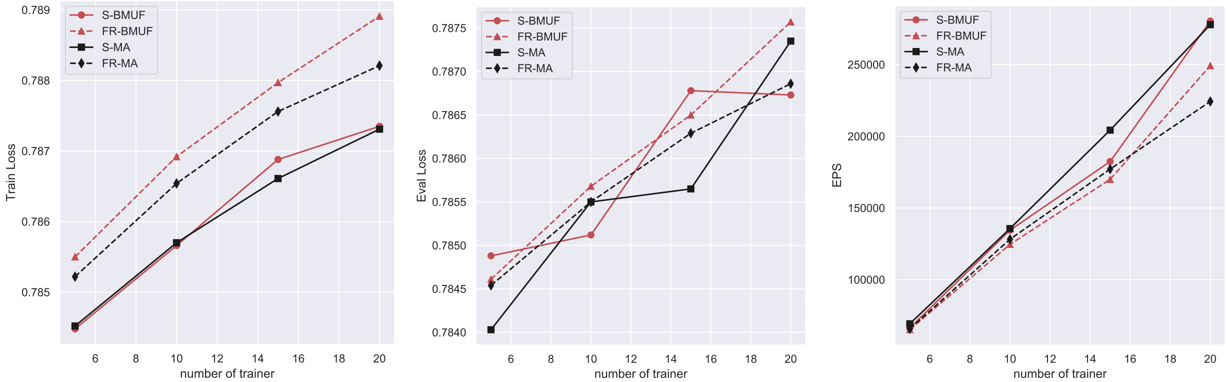
4.2 ShadowSync vs Fix Rate: BMUF & MA

We present a similar but simplified experiment for BMUF and MA type of algorithms. We apply the fixed rate and ShadowSync versions of those algorithm to training Model−B on Dataset−2, where the number of trainers are 5, 10, 15, and 20 respectively. We inspected the average sync rate of S−BMUF, which was 2

[1] The relative increase of loss is defined as $(\text{loss}_{\text{new}} - \text{loss}_{\text{old}})/\text{loss}_{\text{old}}$.

		S-EASGD	FR-EASGD-5	FR-EASGD-30
10 Trainer	Train	**0.084%**	0.099%	0.096%
	Eval	**0.062%**	0.093%	0.112%
20 Trainer	Train	0.230%	0.249%	**0.210%**
	Eval	**0.177%**	0.333%	0.250%

Table 3: Relative loss increase comparing to the 5-trainer result.



(a)

(b)

Figure 6: (a): Model quality of BMUF and MA under ShadowSync and fixed rate frameworks for training `Model-B` on `Dataset-2`. (b): The `EPS` scaling of BMUF and MA algorithms.

syncs per minute for 5 trainers and 0.8 for 20 trainers. For `S-MA`, the numbers were 2.9 and 1.0. We then set the sync rate of `FR-BMUF` and `FR-MA` to be 1 per minute. The losses are reported in Figure 6a. The performance of ShadowSync algorithms are comparable and even superior to the fixed rate versions. The `EPS` plot is deferred to Figure 6b. Here the synchronization is not a bottleneck for all the experiments, and all the algorithms can scale linearly.

4.3 ShadowSync Algorithms

`S-EASGD` is a representative *centralized* algorithm, where the parameter exchange happens in a single location. One shortcoming of `S-EASGD` is it requires extra machines for synchronization purpose only, and the number of sync PSs need to increase if we want to further reduce the sync gap. In contrast, for *decentralized* algorithms the synchronization happens across trainers directly. `S-BMUF` and `S-MA` are two instances under ShadowSync framework. We are thus interested in comparing `S-BMUF` and `S-MA` to `S-EASGD`.

We applied those methods to training `Model-B` on `Dataset-2`, using 5, 10, 15 and 20 trainers. The setup is the same as in Section 4.1. The number of embedding PSs are the same as the number of trainers, and we use 2 sync PSs for `S-EASGD`. The same hyper-parameters are deployed to all 3 methods. One exception we made is the elastic parameter α for `S-BMUF`. `S-BMUF` tends to update the model more conservatively than `S-MA`: it would make a step towards to average model rather than taking it directly. In light of this, we hypothesized `S-BMUF` will converge slower than `S-MA`. Hence, in addition to the standard α used before, we tested a larger value for it to make more aggressive parameter sharing. See Figure 7 for the results. Increasing α does improve the performance of `S-BMUF`. `S-EASGD` has best training performance, followed

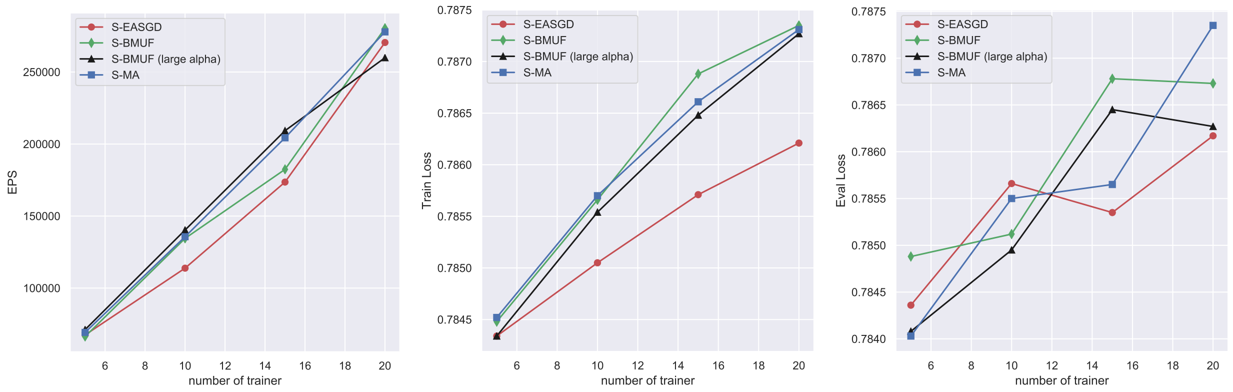


Figure 7: Performance of ShadowSync algorithms `S-EASGD`, `S-BMUF` and `S-MA` for training `Model-B` on `Dataset-2`.

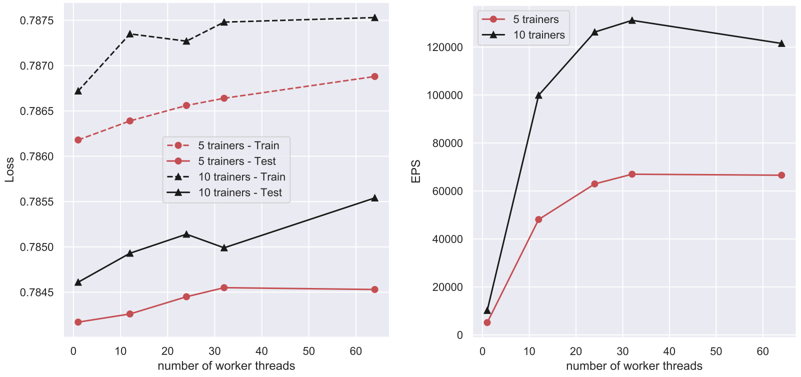


Figure 8: Performance of `S-EASGD` with varying number of worker threads for training `Model-C` on `Dataset-3`.

by `S-BMUF` with larger elastic parameter. However, the evaluation performance is mixed. None of those algorithm stands at the leading place. To summarize, our experiments suggest that `S-BMUF` and `S-MA` are capable to perform comparably good as `S-EASGD`. This is heart-stirring message for users with limited computation budget.

4.4 Hogwild vs Single Thread

Finally, we justify the usage of 24 worker threads with Hogwild update throughout our experiments. We shall train `Model-C` on `Dataset-3` using `S-EASGD`. This dataset contains 1,967,190,757 training samples and 4,709,234,620 evaluation samples. The baseline of our experiment is `S-EASGD` using single-thread training. For Hogwild, we tried 12, 24, 32 and 64 worker threads. All hyper-parameters are set to be the same. We run this experiment under 5 trainers and 10 trainers setup, respectively. For 5-trainer training, we use 1 sync PS, and 4 embedding PSs. For 10-trainer training, we use 1 sync PS and 6 embedding PSs. Results are shown in Figure 8.

The left panel plots training and evaluation losses versus the number of worker threads. We do observe an increasing pattern, however the quality drop is mild compared to the `EPS` gain, plotted in the right panel. The right panel also shows the `EPS` almost stops growing when 24 or more threads are used, for both 5-

trainer and 10-trainer cases. We find that the trainers became the bottleneck in those cases, as the memory bandwidth is saturated (the interaction layers are memory bandwidth demanding). With 12 worker threads, the memory bandwidth utilization is around 50%. After we double the number of worker threads to be 24, we already saturate the memory bandwidth: the average utilization is around 70%, while some hot trainers have 89% utilization.

5 Conclusion

In this paper, we described a new **ShadowSync** framework that synchronizes parameters in the background. This framework isolates training from synchronization. We described the **ShadowSync EASGD**, **ShadowSync BMUF**, and **ShadowSync MA** algorithms under this framework, and have shown that these algorithms can scale linearly with similar or better model quality compared to their foreground variants. We also described how we integrate the ShadowSync framework into our distributed training system, which expresses both model parallelism and data parallelism (with both Hogwild parallelism and replication parallelism) to accomplish the extremely high `ELP` numbers.

Acknowledgement

We would like to thank Mohamed Fawzy, Ozgur Ozkan, Mike Rabbat, Chonglin Sun, Chenguang Xi and Tommy Yu for helpful discussions and consistent support.

References

[1] M. Assran, N. Loizou, N. Ballas, and M. Rabbat. Stochastic gradient push for distributed deep learning. *arXiv preprint arXiv:1811.10792*, 2018.

[2] T. Ben-Nun and T. Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Comput. Surv.*, 52(4):65:1–65:43, 2019.

[3] O. Bousquet and U. von Luxburg. Stochastic learning. *Advanced Lectures on Machine Learning*, pages 146–168.

[4] S. Chaturapruek, J. C. Duchi, and C. Ré. Asynchronous stochastic convex optimization: the noise is in the noise and sgd don't care. In *Advances in Neural Information Processing Systems*, pages 1531–1539, 2015.

[5] K. Chen and Q. Huo. 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)*, pages 5880–5884. IEEE, 2016.

[6] C. De Sa, M. Feldman, C. Ré, and K. Olukotun. Understanding and optimizing asynchronous low-precision stochastic gradient descent. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 561–574, 2017.

[7] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. aurelio Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems 25*, pages 1223–1231. Curran Associates, Inc., 2012.

[8] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[9] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv:1706.02677*, 2017.

[10] I. Hakimi, S. Barkai, M. Gabel, and A. Schuster. Taming momentum in a distributed asynchronous environment. *arXiv preprint arXiv:1907.11612*, 2019.

[11] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[12] X. He, J. Pan, O. Jin, T. Xu, B. Liu, T. Xu, Y. Shi, A. Atallah, R. Herbrich, S. Bowers, and et al. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, ADKDD'14, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450329996. URL https://doi.org/10.1145/2648584.2648589.

[13] G. Hinton, N. Srivastava, and K. Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on*, 14:8, 2012.

[14] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[15] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 583–598, 2014.

[16] X. Lian, W. Zhang, C. Zhang, and J. Liu. Asynchronous decentralized parallel stochastic gradient descent. *arXiv preprint arXiv:1710.06952*, 2017.

[17] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, D. Golovin, et al. Ad click prediction: a view from the trenches. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230, 2013.

[18] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[19] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 693–701. Curran Associates, Inc., 2011.

[20] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[21] J. Wang, V. Tantia, N. Ballas, and M. Rabbat. Slowmo: Improving communication-efficient distributed sgd with slow momentum. *arXiv preprint arXiv:1910.00643*, 2019.

[22] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in Neural Information Processing Systems 30*, pages 1509–1519. Curran Associates, Inc., 2017.

[23] Y. You, I. Gitman, and B. Ginsburg. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*, 2017.

[24] S. Zhang, A. E. Choromanska, and Y. LeCun. Deep learning with elastic averaging sgd. In *Advances in Neural Information Processing Systems*, pages 685–693, 2015.

[25] S.-Y. Zhao and W.-J. Li. Fast asynchronous parallel stochastic gradient descent: A lock-free approach with convergence guarantee. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[26] S. Zheng, Q. Meng, T. Wang, W. Chen, N. Yu, Z.-M. Ma, and T.-Y. Liu. Asynchronous stochastic gradient descent with delay compensation. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 4120–4129. JMLR. org, 2017.

[27] G. Zhou, X. Zhu, C. Song, Y. Fan, H. Zhu, X. Ma, Y. Yan, J. Jin, H. Li, and K. Gai. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1059–1068, 2018.

[28] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*, pages 2595–2603, 2010.