Asynchronous Stochastic Gradient Descent for Extreme-Scale Recommender Systems

Lewis Liu,[1,*] **Kun Zhao**[2,*]
[1] University of Montreal, Quebec
[2] Alibaba Group
allis.algo2@gmail.com, kun.zhao@alibaba-inc.com

Abstract

Recommender systems are influential for many internet applications. As the size of the dataset provided for a recommendation model grows rapidly, how to utilize such amount of data effectively matters a lot. For a typical Click-Through-Rate(CTR) prediction model, the amount of daily samples can probably be up to hundreds of terabytes, which reaches dozens of petabytes at an extreme-scale when we take several days into consideration. Such data makes it essential to train the model parallelly and continuously. Traditional asynchronous stochastic gradient descent (ASGD) and its variants are proved efficient but often suffer from stale gradients. Hence, the model convergence tends to be worse as more workers are used. Moreover, the existing adaptive optimizers, which are friendly to sparse data, stagger in long-term training due to the significant imbalance between new and accumulated gradients.

To address the challenges posed by extreme-scale data, we propose: 1) Staleness normalization and data normalization to eliminate the turbulence of stale gradients when training asynchronously in hundreds and thousands of workers; 2) *SWAP*, a novel framework for adaptive optimizers to balance the new and historical gradients by taking sampling period into consideration. We implement these approaches in TensorFlow and apply them to CTR tasks in real-world e-commerce scenarios. Experiments show that the number of workers in asynchronous training can be extended to 3000 with guaranteed convergence, and the final AUC is improved by more than 5 percentage.

Introduction

Deep Learning (DL), which involves a class of neural networks with deep architectures, is now a workhorse powering state-of-the-art results on a wide spectrum of data-driven tasks. Since the model Wide & Deep (Cheng et al. 2016) was proposed, it has been widely used and developed in many recommender systems. For a typical industrial scenario, where the daily data can be up to hundreds of terabytes or even more, great amount of computation power has to be leveraged. In view of the training efficiency, distributed DL, where a model is trained by hundreds and thousands of workers concurrently, becomes necessary. Besides, many experiences show that the model will perform better as the training data accumulated. Because of this, industry models are often trained using data of several days, weeks or even months, which makes the total data size aggravated.

To learn models from such huge data, effective training methods must be developed. Currently, the main challenges are:

- Data parallel training with hundreds and thousands of workers, which compute gradients locally and update the global parameters asynchronously, is necessary to consume so much data. The asynchrony causes stale gradients, which may lead to model divergence. The problem will get worse as workers increased. How to guarantee the model convergence as well as training speed?
- Adaptive optimizers are proved stable in many cases, especially for sparse data. But they will come to a bottleneck with continuous gradients accumulation (Zeiler 2012). Some variants with historical gradients decay at each training step perform worse convergence. How to balance the value between the new and the historical information in long-term training? And how to balance the role between dense and sparse features?

To fill the mentioned above, we analyze the behavior of gradient staleness in asynchronous distributed training, and then clarify the key factors that impact model convergence in extremely large-scale clusters. Based on our analysis, a staleness normalization framework is proposed to improve the convergence in asynchronous distributed training. Besides, we try to take time as a factor of training samples to address the impact of infinite accumulated data, based on which a method to balance the new and the historical data is proposed. It is suitable to any adaptive optimizers and performs excellent. In general, our main contributions are:

- We analyze the behavior of gradient staleness as the number of distributed workers changes and also validate our ideas by experiments. This will serve as a theoretical basis for many other works. Based on this, we can quantify the optimization methods.
- To tackle the problems caused by huge data, we propose a staleness normalization framework for large-scale asynchronous distributed training, and a periodic recession based optimization method for infinite accumulated training data. Both of them can be integrated into any SGD

[*]Equal contributions.

based optimizers. These methods are valuable and feasible for industrial scenarios.

Related Work

In recent years, distributed DL attracts great attentions. One of the implementation developed and analyzed is the decentralized parallel stochastic gradient descent (D-PSGD). (Lian et al. 2017a) gives an theoretical breakthrough on non-convex optimization for D-PSGD, which first justifies the potential advantage of decentralized algorithms over the centralized ones and identifies regimes in which decentralized algorithms can be faster with convergence guarantees. More detailed experiments are further conducted in (Tang et al. 2018; McMahan et al. 2016). A more recent work (Lian et al. 2017b) demonstrates that an asynchronous decentralized stochastic gradient decent algorithm (AD-PSGD) is proven to be robust in a heterogeneous environment. In addition, a co-designed decentralized ML system and training algorithm is proposed in (Xie et al. 2018). However, the challenges are imposed under decentralized settings when the data size is promoted to industrial level. Some researches extend decentralized SGD with a pipe (Harlap et al. 2018; Li et al. 2018) and perform well in ONE server equipped with several GPU cards. But they are not verified when server count reaches hundreds or thousands. On the other hand, the centralized architecture with parameter server (PS) (Ho et al. 2013; Li et al. 2014) indicates an alternative and more controllable regime in distributed DL literature. In this scenario, stochastic gradient descent (SGD) using mini-batch lies in the core of data-parallel distributed training algorithms. Unfortunately, synchronous SGD(SSGD) is often slow or even infeasible due to the need of waiting for the slowest node in every iteration. Industry practitioners resorted to asynchronous SGD(ASGD), which removes the synchronization requirements, and greatly speeds up the training procedure. However, the poor convergence results limit the scalability of ASGD.

When applying ASGD to large-scale distributed training with thousands of workers, we meet the problems due to the nature of asynchronous regime. An observation in (Huo and Huang 2016) reveals that the variance of gradient estimators in distributed stochastic updates deteriorate the convergence rate. The most well-developed shortage of ASGD, named as gradient staleness, is illustrated through a series of works (Lian et al. 2015; Zhang et al. 2016; Zheng et al. 2017). Such staleness is accumulated through the training procedure especially in massively distributed systems. A few methods are proposed in prior works to improve ASGD convergence against the gradient staleness. Backup gradients (Chen et al. 2016), gradient compensation (Zheng et al. 2017) and learning rate decay (Zhang et al. 2016) have chronologically been proposed to alleviate the impact of stale workers. (Lian et al. 2015) proved that the model trained with ASGD has a bounded convergence rate. Its follower (Zhang et al. 2016) took gradient staleness as a decay factor of learning rate, which makes a good result when tens of workers training concurrently. (Zheng et al. 2017) proposed a method to compensate the gap between the current model parameters and stale gradients and got good results on small staleness cases. However, few former works point out the factors of ASGD convergence when scaling to thousands of workers and the corresponding effective solutions.

In the other hands, many experiences show that the model tends to be better as the training data accumulates. To promote these needs, a lot of adaptive optimization methods (Sutskever et al. 2013; Kingma and Ba 2014; Zeiler 2012; Duchi, Hazan, and Singer 2011) are proposed. The main point of these methods is that not only the current batch of data but also the historical information will guide the direction of model convergence. In many cases, they are proved more effective and stable than the raw SGD. However, when the training data comes to petabytes, it harms the continuous training procedure due to the infinite momentums.

Methods for Asynchronous Training

In this section, we propose two strategies for performance improvement: staleness-normalization and data-normalization for the prospect of reducing the turbulence of weight update and smoothing the loss function.

Background of Distributed SGD

Parameter Server (PS) terminology is used in which model parameters are stored on PS and gradients are calculated on workers. Let w represents the model parameters, B represents the labeled training set in a iteration and n represents the size of B, η represents the learning rate. The serialized training procedure of iteration t can be denoted as follows,

$$w_{t+1} = w_t - \eta \frac{1}{n} \sum_{x \in B} \nabla f(x, w_t). \tag{1}$$

In distributed Asynchronous SGD(ASGD), all workers pull and push w in a lock-free way, without waiting for other workers. w on parameter server is updated as follows,

$$w_{t+1} = w_t - \eta \frac{1}{n} \sum_{x \in B_i} \nabla f(x, w_{t-\tau_{i,t}}), \tag{2}$$

where $\tau_{i,t}$ represents the staleness of the *i-th* worker at timestamp t.

Staleness is defined as the version difference between the pulled w and pushed w. That is, the version of newly-updated variable w_{t+1} minus that of w in $\nabla f(x, w_t)$ term. For example, staleness is always **1** in Synchronous SGD. Figure 1 intuitively illustrate parameter updating procedure in Equ. 2. ASGD does not enforce a barrier among workers after each iteration and therefore allows updates coming from an old version of parameters. Instead, Synchronous SGD accumulates the updates from all workers before the beginning of next iteration. Staleness ($\tau_{i,t}$) here can be denoted as the number of pushes raised by all workers during the period between timestamp t (current push of worker i) and timestamp $t - \tau_{i,t}$ (previous pull of worker i).

Trend of Staleness as Workers Scales

Here we describe how staleness changes when ASGD scales to thousands of workers. First of all, we pose some mild assumptions on the updating behaviors of both the PS side and

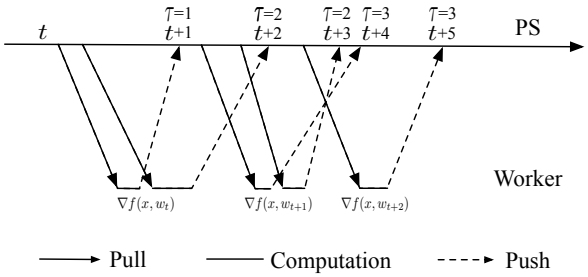

Figure 1: Illustration of asynchronous updates in Parameter Server (PS) architecture. Each worker pulls a replica of parameter w from PS, computes the gradients (∇f) with the pulled parameter and then push them back to PS. From the perspective of PS, the timestamp (t) of parameter is added by 1 after receiving each gradient. The staleness (τ) of each update is defined as the timestamp difference between the push and the pull.

clients by casting the arrivals of push/pull requests as a *Poisson process*. We first assume that due to different manufacturing quality of workers, the pushing/pulling rate of each worker is subjected to a normal-like distribution with non-negative values as follows: The pushing rate (at which a worker submits its own gradients to PS) $\gamma_i > 0$ of each worker i is sampling from a normal-like distribution with finite mean and variance. Let $N^i(t)$ denotes the number of pushes during period $(0, t]$ for worker i, then N^i is a Poisson process with arrival rate γ_i.

Next we will show that the compound behavior of multiple workers is eligible to be treated as a whole Poisson arrival via properties from Poisson process, which helps analyze the correlation between the system scale and gradient delay from a macro view.

Superposition of Poisson Process

Suppose we have two independent Poisson process with arrival rate γ_i and γ_j respectively, then the combined process of arrivals from both processes is a Poisson process with arrival rate $\gamma_i + \gamma_j$ (Ross et al. 1996).

We denote the average pushing rate of each worker by $\overline{\gamma}$ and the variance of γ_i by δ_γ^2. Following the superposition property of Poisson process, the total arrival of pushes seen from PS side is a compound Poisson process with the compound arrival rate $\gamma = \sum_{i=1}^{n} \gamma_i$. Furthermore, the mean of this compound rate can be expressed as $E\gamma = \sum_i^n E(\gamma_i) = n\overline{\gamma}$.

Note that the distribution of γ_i only works for the selecting machine process. Once the cluster of workers for training is decided, the concrete pushing rate of each worker will be determined. Hence in the modeled Poisson process for a worker, γ_i keeps a constant. However, the distribution of the staleness determined by computational performance of one worker as well as others depends on the prior distribution of pushing rates. With the compound result, the staleness over the network will scale as the number of workers scales.

Linearly Scaling of Averaged Staleness

For the *i-th* worker, its staleness during a pull-push period Δt_i is the count of pushes raised by all the workers to PS. It is concluded that for any cluster of workers subjected to Poisson process models with pushing rate γ, the mean of staleness increases linearly with the number of workers. The result is computed through conditional expectations as follows.

$$\begin{aligned} E(\tau_i) = E(N(\Delta t_i)) &= E(E(N(\Delta T_i))|\Delta t_i = \Delta T_i) \\ &= E_{\Delta t_i}(((n-1)\overline{\gamma} + \frac{1}{\Delta t_i}) \cdot \Delta t_i) \\ &= (n-1)\overline{\gamma}E(\Delta t_i) + 1 = n. \end{aligned}$$

Staleness Normalization

Inspired by the results above, we proceed to operate the learning rate with the scaling property of staleness to stabilize the gradient update and achieve a convergence speed up. (Zhang et al. 2016) divides the fine tuned learning rate η by τ directly and achieves a better performance in a few workers. However, when applied in large-scale cases, the staleness tends to hundreds or thousands, where the new learning rate will slow the training procedure terrifically. Our main idea is to scale down the behavior of current cluster down to a smaller one via dividing the learning rate by a appropriate staleness factor, which is exactly the *normalized staleness* obtained by a linear transformation between random variables. In what follows, we denote by τ the staleness of the current cluster, and by τ' the target staleness (corresponding to a smaller cluster) which we expect to transform into. Considering that the larger cluster possess a staleness with greater a mean and variance leading to the worse performance, the transforming coefficients are composed of means and variances at the two scales following to the following proposition. Let $f(\cdot)$ be any probability density function (pdf). Suppose that μ is a real number and σ stands for a positive real number. Then X is a random variable with pdf $(1/\sigma)f((x-\mu)/\sigma)$ if and only if there exists a random variable Y with pdf $f(y)$ and $X = \sigma Y + \mu$. The detailed proof of the property can be found in Chapter 3 of (Casella and Berger 2002). Another insight of our idea gives that the staleness also varies under a normal-like distribution affected by the prior distribution of inherent pushing rates. Consequently, by combining the means (μ and μ') and variances (δ and δ') of the current and smaller cluster respectively with Proposition , we derive the τ-normalization formula according to distribution transformation between normal distributions (Durrett 2010) as $\tau' = \delta'(\tau - \mu)/\delta + \mu'$. In view of Proposition , we can further write $\tau' = \delta'\tau/\delta$. Applying this normalized τ to the current cluster, we obtain the actual learning rate as $\eta' = \eta/\tau'$.

The derivation above implies that different models have different tolerance for stale gradients, and a specific strategy may not be appropriate to other models. For generality, we promote the method in (Zhang et al. 2016) by considering a function of staleness τ as below,

$$\eta' = \rho(\tau) \cdot \eta. \tag{3}$$

Here η' is the adjusted learning rate, and $\rho(\tau) \in (0, 1]$ is the discounted factor as a function in τ that admits various options. We choose

$$\rho(\tau) = \tau^{-k}, k \in N. \tag{4}$$

By setting $k = 2$, we may get a more reasonable factor of learning rate in large-scale cases.

Furthermore, we consider different stages in a training procedure. Given a loss function f with parameters w, the training procedure is to fit for the training samples by updating w iteratively. For a continuous differentiable convex function, Δw is gradual and not saltatoric. In the begining of the training procedure, w changes fast and it is sensitive to gradient updates. When the model is stable, the stale gradients may not harm so much. Here we take the last Δw into consideration to make different decisions in different training stages, even for the same τ. We write

$$\eta'_t = \rho(\tau \cdot \text{trans}(\Delta w_{t-1})) \cdot \eta_t, \tag{5}$$

where the last Δw is used to measure how we care about the staleness. trans(Δw) also admits a number of options, such as below, where λ is a weight.

$$\text{trans}(\Delta w) = \text{sigmoid}(\lambda \cdot |\Delta w|), \tag{6}$$

We remark that our τ-normalization protocol is a superset of Staleness-aware ASGD (Lian et al. 2015), in that τ always normalizes to SSGD scenario in their work. Our staleness normalization protocol is derived from the fact that the staleness scales linearly with workers. It avoids directly divide the staleness, which can be tens of thousand when using thousands of workers. Instead, it uses a specific number of workers as the baseline, which is set to 1K workers throughout our evaluation, and then fine-tune the learning rate. To formally state our decay strategy over different workers, our scheme can be formulated as

$$\rho = \begin{cases} 1 & \tau \leq \mu' \\ \frac{1}{(\tau)^2} & \tau > \mu' \end{cases}, \tag{7}$$

where the part of identity mapping helps keep a meaningful value of learning rate from being decayed to a considerably small level which hinder the convergence performance instead. Figure 2 illustrates three different settings of the factor ρ, where Linear stands for the strategy to directly divide the learning rate of the current cluster by its own staleness, and Linear1K denotes the case where the staleness is approximately normalized to a cluster scale of 1K, Linear500 shows a normalization to the scale of 500 similarly. From Figure 2 we can observe that, the decay factor will decrease slower as the staleness increases. For example, when two staleness value τ_0 and $2\tau_0$ are compared, the impact of $2\tau_0$ is $2\tau_0/\tau_0 = 2$ times less than that of τ_0, which result in a less negative impact of the large staleness values compared to the vallina update strategy.

Incorporate Staleness into Optimizers

The learning rate will affect not only the current gradients but also the momentums in many optimizers. It is not meaningful to punish the accumulated gradients just because the

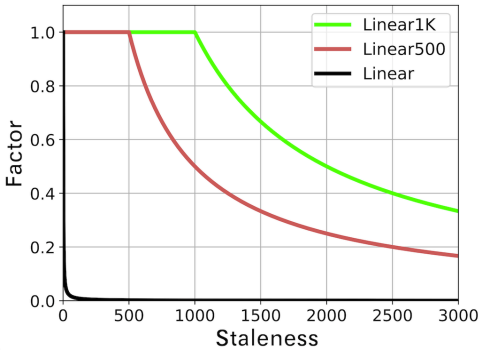

Figure 2: Normalized decay factor.

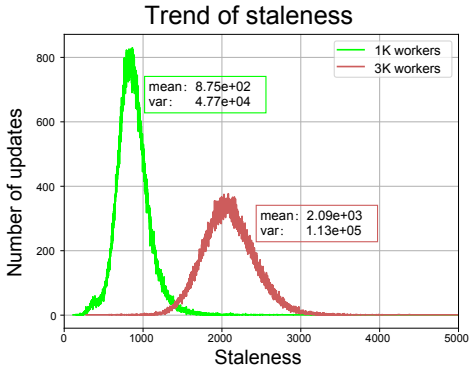

Figure 3: Trend of staleness when training on 1K and 3K workers. X-axis represents the measured staleness and Y-axis represents the corresponding updates number during a given period, in which about 260K updates are recorded in parameter server.

current gradient is stale. We apply ρ to the gradient g_t at each iteration instead of to the learning rate as follows,

$$\hat{g}_t = \rho(\tau \cdot trans(\Delta w_{t-1})) \cdot g_t. \tag{8}$$

We can easily find that the decayed gradients will lead to a decayed learning rate in some way, which aims a discounted update. Such as, in Adagrad, $\Delta w = \frac{\eta}{\sqrt{v_t + \epsilon}} g_t$, where $v_t = v_{t-1} + g_t^2$. And when $\hat{g}_t$ is used, $\hat{v}_t = v_{t-1} + (\rho(\tau) g_t)^2$, we can get $\Delta \hat{w} < \Delta w$.

Data Normalization

Aside from the turbulence brought by staleness updates, in most cases, the sampled data at different workers also show a distribution deviation, which may lead to a considerable variance of updates when the number of workers becomes large. What is worse, the so-called internal covariate shift (ICS) (Ioffe and Szegedy 2015) of a network layer dates from the parameter updates of the preceding layers. Such variation consequences become much more significant in distributed asynchronous updates.

BatchNorm (Ioffe and Szegedy 2015) was proposed as a major advanced technique through the evolution of DL

Algorithm 1: Local Batch Normalization

$output_0$ can be input or other layer's output
$w^{(i)}$ is the parameters of the $layer_i$
L is the number of dense layers
for i in range(1, L) **do**
  $output_i = \text{Dense}(w^{(i)}, output_{i-1})$
  $output_i = \text{Normalize}(output_i)$
  $output_i = \text{Active}(output_i)$
end for

methods from algorithmic aspect, which was widely used to speed up the learning process and reduce the cost of parameter tuning. Empirically, at most time, BatchNorm behaves to benefit controlling the change of the layers' input distributions during training. Furthermore, it is recently discovered that BatchNorm impacts network training in a fundamental way. That is, it makes the landscape of the underlying optimization problem significantly more smooth, which leads to the fact that the derived gradients become more predictive and a wider range of learning rates are allowed. However, up to now, no research takes BatchNorm as a solution for large-scale asynchronous DL training. Here we present data normalization to intentionally solve the aforementioned problems in such training environments. Our key idea is that applying BatchNorm to each worker asynchronously.

In a search or recommendation system in practice, Wide&Deep (W&D) model (Cheng et al. 2016) was widely used. Our model extended W&D, in which the number of sparse feature parameters reaches 30 billions. The sparse feature parameters are not updated frequently as usual, and so that stale gradients problem will not happen to sparse features. We just operated normalization on the dense layers. Our algorithm combined with this W&D model is illustrated in Algorithm 1.

In particular, we use a simple normalization instead of the standard Gaussian as follows.

$$\text{Normalize}(x) = \frac{x - E(x)}{E(x)} = \frac{x}{E(x)} - 1. \tag{9}$$

By introducing two coefficients α and β, we have

$$\text{Normalize}(x) = \alpha\frac{x}{E(x)} - \beta, \tag{10}$$

where α and β are trainable parameters. In this normalization scheme, only the expectation of the input needs to be computed. Some computation cost can be saved compared to the standard Gaussian, such as the variance. More importantly, it avoids a poor model accuracy.

Samples Weighted According to Period

To use the second-order information of gradients while keeping computation and storage feasible, adaptive optimizers such as Adagrad (Duchi, Hazan, and Singer 2011), AdaDelta (Zeiler 2012), Adam (Kingma and Ba 2014), etc were proposed by considering only the diagonal of the inverse Hessian matrix. Adaptive optimizers have been widely adopted for their stability. However, *vanishing update* problem in these adaptive optimizers comes with big data. Take Adagrad as an example. After training with sufficient large iterations, the accumulated gradients could possibly be very large and the Δw gradually converges to zero, which means the model will not be improved as more data comes in. Prior researches (Zeiler 2012) introduce a factor λ to decay v_t in each iteration in the following way.

$$v_t = \lambda v_{t-1} + (1 - \lambda)g_t^2.$$

When applying this on extreme-scale data with hundreds of millions of iteration, the value of data that is trained earlier vanishes. It is not always reasonable for a industrial recommender system. Usually, the industrial model is trained in a incremental way. That is, the new model parameters will be initialized based on the ones of the model yesterday, and then be fine tuned using the new training data. In such cases, the samples generated in the same day should be treated equally, and the order of them does not mean a *weight*. Our main idea is, in the same period, how much information of the samples will be preserved should not be determined by the appearance order but the training algorithm.

Here we introduce a trade-off framework called *SWAP, Samples Weighted According to Period*, to balance the new coming data and the historical ones. It performs excellent in long-term incremental training which may last for days or weeks. Let T play for the *Period*. In the same *Period*, the samples are treated equally. We incorporate *SWAP* with Adagrad and the improved algorithm 2 is as below.

In the same period, both dense and sparse parameters are treated in the same way. That is, by setting t with global step instead of the update times, the learning rate for sparse parameters will be larger than for the dense parameters in some way. Usually, sparse features used for updating sparse parameters are less than the dense ones, and the whole training procedure may benefit from niched learning rate. *SWAP* can also be applied to other optimizers. It is a kind of trade-off between large accumulated data and preserving information based on data aging. Usually, λ_0 is chosen according to the amount of training data, the period T, and prior knowledge of your task. It indicates that how much to discount every T steps. For example, if we divide one day into 5 periods and set λ_0 to 0.9, the samples appeared in the first period will be weighted by 0.9^5 when training finished.

Algorithm 2: Adagrad-SWAP

Initialize w_0, v_0, λ_0, T
Define t as the global training step.
if t mod T = 0 **then**
  $\lambda_t = \lambda_0$
else
  $\lambda_t = 1.0$
end if
$v_t \leftarrow \lambda_t \cdot v_{t-1} + g_t^2$
$w_t \leftarrow w_{t-1} - \frac{\eta}{\sqrt{v_t + \epsilon}} g_t$
$t \leftarrow t + 1$

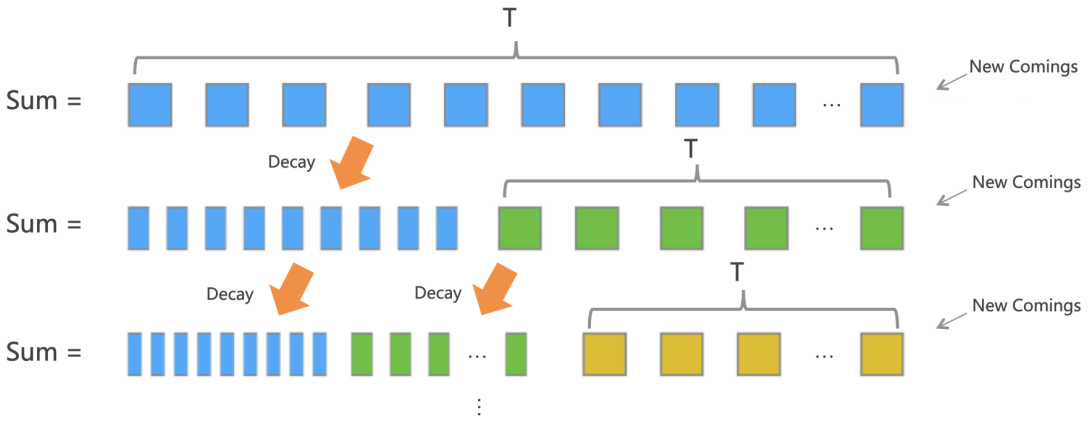

Figure 4: How *SWAP* works in the training procedure.

Experiments

Aforementioned algorithms are helpful in training with thousands of workers on huge data. In this section, we conduct experiments on the CTR model for a e-commerce search engine. The experiments include:

- How staleness changes as training cluster scales.
- Effects of the proposed staleness normalization and data normalization mechanism for convergence.
- The effects of improved optimizers with *SWAP* for long-term training.

E-Commercial Model and Dataset

The E-Commercial model is a DNN model composed of a stack of fully-connected layers and an embedding layer, in which attention mechanism is applied. It is a variant of Wide and Deep Model (Cheng et al. 2016) (Ni et al. 2018). The training data consists of records of browsing and purchases, queries and product information on a e-commerce web site. The model predicts whether a product will be clicked (0 or 1) or not to deliver a better online exhibition. Area under ROC curve (AUC) is used to measure the accuracy of such a binary classification model. The size of a model checkpoint is about 250 GB. The amount of daily training samples is about 7 billions and the size is more than 100TB. The model is trained in a incremental way. i.e, it uses samples of day-1 as training set and day-2 as test set, and refines the model day by day.

Computation Environment

The model was trained in a CPU commodity cluster with tens of thousands of nodes. Moreover, many types of tasks including online serving tasks, data cleaning tasks and offline training tasks concurrently share the underlying hardware. Each node in the cluster is equipped with a 64-core CPU and 512GB memory. Node are connected with Ethernet of which the bandwidth is around 2.5GB/s. During training, the resource of each worker (and parameter server) is limited to 10 physical cores and 20 GB memory.

Software Implementation

The staleness is calculated for each trainable parameter w of the model. To measure the staleness τ, we attach a version tag V to each w. Workers get V during pulling w. For the dense features, V is a scalar value. For the sparse features, such as id embeddings, V is a vector with each element representing the version tag of one id. As a result, the shape of V equals the first dimension of w. After local computation, the workers push the gradients as well as V to servers, and difference between the current V and pushed V is the staleness τ. V will be increased by one once the corresponding w is updated. We extend TensorFlow V1.4 as the training framework with the proposed operators and provide several new interfaces including *VersionedVariable* and *AdagradSWAPOptimizer*. Later the implementation will be integrated into higher TensorFlow releases.

The Trend of Staleness as Scaling

To measure the distribution of staleness, we record the staleness for about 260K updates on parameter server and the results are shown in Fig. 3. The mean of the staleness distribution on 1K worker and 3K worker is about 875 and 2090, while the variance is 4770 and 11300 respectively, which is roughly consistent with our theoretical analysis.

Moreover, an intuitive finding from Fig. 3 is that larger amount of workers bring more turbulent updates. At the same time, more turbulent updates bring unstable convergence. To our best knowledge, this is the first theoretical and quantitative analysis of staleness for large-scale ASGD training. It not only reveals the reason why ASGD suffers from poor convergence that were also reported and analyzed by prior researches, but also enables a proper normalization of gradient, in which the parameters will be adjusted according to the distribution.

Effects of Staleness Normalization

To show how much the normalized gradients benefit, we carry out experiments with 3K workers. Our method will be compared with two baselines: no normalization and linearly

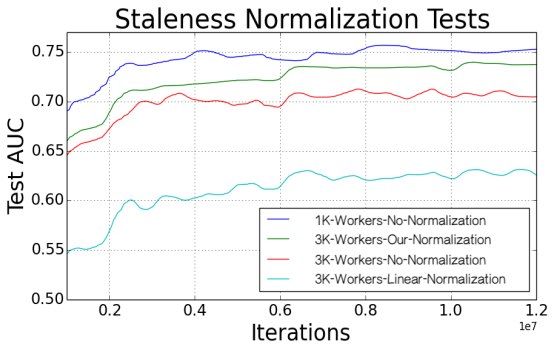

Figure 5: Gradient decay with normalized staleness for training on 3K workers. The model is trained with 12 million iteration (about 7 billion samples). We compare the convergence performance in terms of prediction AUC on three settings, vanilla ASGD, prior linearly normalization (Zhang et al. 2016), and our method.

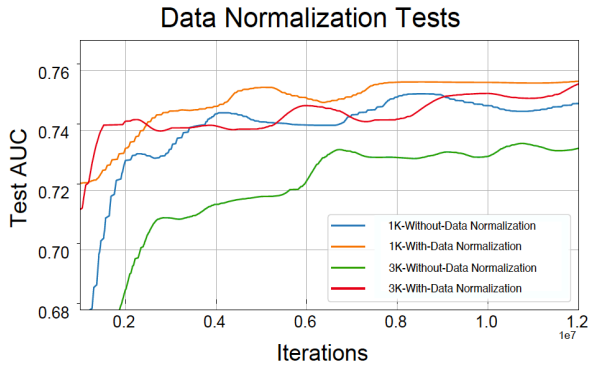

Figure 6: Data normalization on 1K and 3K workers. We employ data normalization over staleness normalization, and compare the obtained convergence performance in terms of prediction AUC.

normalization in (Zhang et al. 2016). The AUC trained in 1K workers is what we expected.

By setting $k = 2$ in Equ. 4, we can see from Fig. 5, our staleness normalization method accelerates the convergence and also performs better in the final result. Specially, linearly normalization (Zhang et al. 2016) shows the worst convergence, which mainly suffers from the large staleness and is infeasible for large-scale scenarios. An interesting findings is that no-normalization method performs not too bad. We believe it is because the extensive sparse embeddings in our DNN architecture and they can accommodate more turbulent sparse embedding gradient compared with dense CNN model in (Zhang et al. 2016). When the number of workers scales from 1K to 3K, we find that the two baselines perform worse but our method almost achieves the same AUC as in 1K workers.

Effects of Data Normalization

We apply data normalization based on the experiments of staleness normalization and measure how much it benefits the convergence performance. Here we treat data normalization as a compensation of staleness normalization and never use it solely. From Fig. 6, we can see the data normalization benefits not only in accelerating the convergence, but most importantly, increasing the overall AUC metrics. For example, training on 1K and 3K workers can achieve the AUC around 0.75 and 0.74 while data normalization improves the result by 1% and 2% respectively.

Effects of *SWAP*

Here we evaluate the benefits of *SWAP* in long-term training with huge accumulated data. The optimizer used in experiment is *Adagrad*, which performs stable and effective for sparse features. The model is trained in a incremental way. That is, the model checkpoint trained yesterday will be restored as the initialization of the model to be trained today. Our experiments will cover 10 days' data. *SWAP* will be

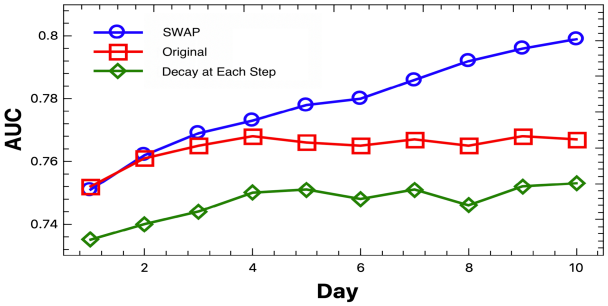

Figure 7: Train the CTR task with *SWAP* or not. Illustrate the trend of AUC as data accumulated.

compared with two baselines: the original *Adagrad* and the one proposed in research (Zeiler 2012) with $\lambda = 0.999999$. We record 10 final AUCs for each optimizer as date goes on and illustrate them in Figure 7. We can see that the original *Adagrad* will meet a bottle-neck when 3 or 4 days' data accumulated, and the reason we have explained before. The optimizer that decays accumulators at each step is not significant in such cases. *Adagrad-SWAP*, in which we set decay rate with 0.8 and $T = 1.4 * 10^7$ shows good result as training data accumulation.

Conclusion

In this paper, we propose some solutions for training a large-scale DNN model with petabytes dataset for a recommender system, including staleness normalization and data normalization for improving convergence when training asynchronously with thousands of workers, and *SWAP*, an effective framework as a compensation to the existing optimizers to adapt for huge dataset. The proposed approaches facilitate DNN training on industry-level data and industry-level model. Experiments on practical products show significant training acceleration and improved performance.

References

Casella, G.; and Berger, R. L. 2002. *Statistical inference*, volume 2. Duxbury Pacific Grove, CA.

Chen, J.; Pan, X.; Monga, R.; Bengio, S.; and Jozefowicz, R. 2016. Revisiting distributed synchronous SGD. *arXiv preprint arXiv:1604.00981* .

Cheng, H.-T.; Koc, L.; Harmsen, J.; Shaked, T.; Chandra, T.; Aradhye, H.; Anderson, G.; Corrado, G.; Chai, W.; Ispir, M.; et al. 2016. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*, 7–10. ACM.

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

Durrett, R. 2010. *Probability: theory and examples*. Cambridge university press.

Harlap, A.; Narayanan, D.; Phanishayee, A.; Seshadri, V.; Devanur, N.; Ganger, G.; and Gibbons, P. 2018. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377* .

Ho, Q.; Cipar, J.; Cui, H.; Lee, S.; Kim, J. K.; Gibbons, P. B.; Gibson, G. A.; Ganger, G.; and Xing, E. P. 2013. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*, 1223–1231.

Huo, Z.; and Huang, H. 2016. Asynchronous stochastic gradient descent with variance reduction for non-convex optimization. *arXiv preprint arXiv:1604.03584* .

Ioffe, S.; and Szegedy, C. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* .

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

Li, M.; Andersen, D. G.; Park, J. W.; Smola, A. J.; Ahmed, A.; Josifovski, V.; Long, J.; Shekita, E. J.; and Su, B.-Y. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *OSDI*, volume 14, 583–598.

Li, Y.; Yu, M.; Li, S.; Avestimehr, S.; Kim, N. S.; and Schwing, A. 2018. Pipe-SGD: A Decentralized Pipelined SGD Framework for Distributed Deep Net Training. In *Advances in Neural Information Processing Systems*, 8045–8056.

Lian, X.; Huang, Y.; Li, Y.; and Liu, J. 2015. Asynchronous parallel stochastic gradient for nonconvex optimization. In *Advances in Neural Information Processing Systems*, 2737–2745.

Lian, X.; Zhang, C.; Zhang, H.; Hsieh, C.-J.; Zhang, W.; and Liu, J. 2017a. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. In *Advances in Neural Information Processing Systems*, 5330–5340.

Lian, X.; Zhang, W.; Zhang, C.; and Liu, J. 2017b. Asynchronous Decentralized Parallel Stochastic Gradient Descent. *arXiv preprint arXiv:1710.06952* .

McMahan, H. B.; Moore, E.; Ramage, D.; Hampson, S.; et al. 2016. Communication-efficient learning of deep networks from decentralized data. *arXiv preprint arXiv:1602.05629* .

Ni, Y.; Ou, D.; Liu, S.; Li, X.; Ou, W.; Zeng, A.; and Si, L. 2018. Perceive Your Users in Depth: Learning Universal User Representations from Multiple E-commerce Tasks. In *SIGKDD*, 596–605. ACM.

Ross, S. M.; Kelly, J. J.; Sullivan, R. J.; Perry, W. J.; Mercer, D.; Davis, R. M.; Washburn, T. D.; Sager, E. V.; Boyce, J. B.; and Bristow, V. L. 1996. *Stochastic processes*, volume 2. Wiley New York.

Sutskever, I.; Martens, J.; Dahl, G.; and Hinton, G. 2013. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, 1139–1147.

Tang, H.; Lian, X.; Yan, M.; Zhang, C.; and Liu, J. 2018. D^2: Decentralized Training over Decentralized Data. *arXiv preprint arXiv:1803.07068* .

Xie, P.; Kim, J. K.; Ho, Q.; Yu, Y.; and Xing, E. 2018. Orpheus: Efficient Distributed Machine Learning via System and Algorithm Co-design. In *Proceedings of the ACM Symposium on Cloud Computing*, 1–13. ACM.

Zeiler, M. D. 2012. ADADELTA: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701* .

Zhang, W.; Gupta, S.; Lian, X.; and Liu, J. 2016. Staleness-aware async-SGD for distributed deep learning. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, 2350–2356. AAAI Press.

Zheng, S.; Meng, Q.; Wang, T.; Chen, W.; Yu, N.; Ma, Z.-M.; and Liu, T.-Y. 2017. Asynchronous Stochastic Gradient Descent with Delay Compensation. In *International Conference on Machine Learning*, 4120–4129.