Fast Distributed Deep Learning via Worker-adaptive Batch Sizing

Chen Chen\textsuperscript{1}, Qizhen Weng\textsuperscript{1}, Wei Wang\textsuperscript{1}, Baochun Li\textsuperscript{2}, Bo Li\textsuperscript{1}
\textsuperscript{1}Hong Kong University of Science and Technology \textsuperscript{2}University of Toronto

ABSTRACT
Deep neural network models are usually trained in cluster environments, where the model parameters are iteratively refined by multiple worker machines in parallel. One key challenge in this regard is the presence of stragglers, which significantly degrades the learning performance. In this paper, we propose to eliminate stragglers by adapting each worker’s training load to its processing capability; that is, slower workers receive a smaller batch of data to process.

Following this idea, we develop a new synchronization scheme called LB-BSP (Load-balanced BSP). It works by coordinate setting the batch size of each worker so that they can finish batch processing at around the same time. A prerequisite for deciding the workers’ batch sizes is to know their processing speeds before each iteration starts. For the best prediction accuracy, we adopt NARX, an extended recurrent neural network that accounts for both the historical speeds and the driving factors such as CPU and memory in prediction. We have implemented LB-BSP for both TensorFlow and MXNet. EC2 experiments against popular benchmarks show that LB-BSP can effectively accelerate the training of deep models, with up to 2× speedup.

KEYWORDS
Distributed Deep Learning, Batch Size, Load Balancing

1 INTRODUCTION
Due to its computation-intensive nature, deep learning models are typically trained on large clusters in a distributed manner \cite{13, 22, 29, 50}: many worker machines iteratively refine the model parameters using a subset of training data, i.e., a sample batch, and communicate with the parameter servers with respect to updates to the model parameters.

While existing distributed training practices \cite{22, 29, 37, 67} are mainly conducted on dedicated clusters with homogeneous workers, we believe that it would become increasingly more common to train new deep learning models using shared clusters. Due to their shared nature, these clusters are heterogeneous in their hardware configurations and time varying in their resource availability. Naturally, workers running on less capable computing hardware would make slower progress, and have the potential to become stragglers.

Stragglers may negatively affect the performance of distributed deep learning training in two possible ways. Each iteration in the convergence process may need to take a longer period of time; or alternatively depending on the mechanism used for worker coordination, more iterations may be required for convergence. A typical mechanism for worker coordination, for example, is called Bulk Synchronous Parallel (BSP) model. It makes sure that workers are synchronized at the end of each iteration, and the amount of time it takes in each iteration may then be longer as fast workers have to wait for stragglers.

In this paper, we seek to improve the overall performance of distributed training of deep models by reducing the amount of time taken in each iteration, and minimizing the number of iterations at the same time. At a high level, in a shared cluster with a heterogeneous environment, stragglers are mainly caused by the mismatch between the workload and processing capability at each worker, and the intuitive solution is to apportion the workload to these workers according to their processing capabilities.

That said, existing load balancing approaches \cite{14, 17, 33, 38} are not suitable for deep learning training. They are designed for cases where each iteration is sufficiently long so that stragglers can be detected and tackled, but iterations in deep learning training are quite short in general, lasting for a few seconds or even less than a second. In fact, it’s unnecessary and even infeasible to address stragglers within iterations. For deep learning workloads, we find it’s more appropriate to detect stragglers at the iteration boundaries, and tackle them by adjusting the size of their sample batches.

In this work, we propose Load-Balanced Bulk Synchronous Parallel (LB-BSP), which adaptively apportions the size of each worker’s sample batch with the objective of equalizing their processing times. This mechanism is referred to as worker-adaptive batch sizing. LB-BSP works effectively without introducing the extra overhead of fine-grained progress monitoring or load transfers, and can be easily implemented in modern deep learning frameworks.

While each worker’s processing time increases monotonically as its workload (represented by its batch size) increases, it remains a challenge how each worker’s workload can be accurately apportioned so that the processing times of all the workers can be equalized. We first formulate our objective as a general optimization problem, and then customize the problem further with the specific characteristics of deep
learning training. The result of such customization is a linear optimization problem that can be directly solved.

One prerequisite, however, for solving our optimization problem is the knowledge of each worker’s processing speed of each sample before each iteration runs. Such processing speed may be affected by the available resources, such as CPU and memory. We use the Nonlinear AutoRegressive eXogenous model (NARX) [32], an extended recurrent neural network that makes speed predictions with those driving resources accounted.

We have implemented LB-BSP as a Python library that can be integrated in both TensorFlow and MXNet (§4). Our experimental evaluation using benchmark deep learning workloads shows that LB-BSP outperforms the existing mechanisms, especially when resource availability is highly heterogeneous across the workers. In an emulated production cluster built with Google Traces [57], the convergence speed under LB-BSP can be \(2\times\) that of BSP. In another GPU-cluster, the performance improvement of LB-BSP over BSP is 41%. With respect to predicting the processing speed, we find NARX surpasses the second best approach by 15%.

2 BACKGROUND AND MOTIVATION

2.1 Distributed Deep Learning

Training process of deep learning models. Deep learning (DL) [48] has dramatically improved the state-of-the-art in many domains such as computer vision [35, 46] and natural language processing [23, 59]. Given a neural network model, the goal of DL training is to find the model parameters \(\omega^*\) that minimizes the loss function \(L(\omega)\) over the entire training dataset, i.e.,

\[
\omega^* = \arg\min_{\omega} L(\omega) = \arg\min_{\omega} \frac{1}{|S|} \sum_{s \in S} l(s, \omega). \tag{1}
\]

Here \(S\) is the labeled training set, and \(l(s, \omega)\) is the loss value for a sample \(s \in S\) with the current parameters \(\omega\).

Mini-batch Stochastic Gradient Descent (SGD) [29, 51] is a popular algorithm to train the model parameters \(\omega^*\). Its basic idea is to iteratively update \(\omega\) with the gradients calculated from sample (mini-)batches:

\[
\omega^{k+1} = \omega^k - \eta g^k, \text{ where } g^k = \frac{1}{|B^k|} \sum_{s \in B^k} \nabla l(s, \omega^k). \tag{2}
\]

Here \(B^k\) is a batch sampled from \(S\) for iteration \(k\), and \(\eta\) is the learning rate. Such an iteration repeats until the model parameters \(\omega\) finally converge.

Distributed deep learning. Due to the ever increasing model complexity and data volume, it has become commonplace to train DL models in a distributed manner, with a dedicated parameter server (PS) and multiple workers. As shown in Fig. 1, in each iteration, each worker \(i\) generates a sample batch \(B_i\) from its input stream, based on which it then calculates the gradient \(g_i\) with the latest global model parameters \(\omega\).

Distributed DL training has been conducted in various cluster environments. Here we broadly classify those clusters into two types: dedicated clusters and non-dedicated clusters.

1) Dedicated clusters usually consist of homogeneous workers and are monopolized by only one training job. Dedicated clusters are typically used by production service providers or model developers, to whom fast model convergence is of the highest priority.

2) Non-dedicated clusters are usually built from heterogeneous machines in which DL workloads coexist with other analytic workloads. Non-dedicated clusters are typically adopted by cost-sensitive organizational entities or budget-limited individuals. For example, large-scale shared clusters [25, 57] often span multiple machine generations or specific configurations, with dynamic, low resource utilization in general; to reduce resource wastage due to over-subscription, the cluster operator can run long-lasting, delay-tolerant DL workloads in the background, using the leftover resources. Besides, in a DL training work, a user might want to combine the use of multiple GPUs with different computing capabilities—mainly because they are the only accessible ones or among the cheapest in EC2 spot market [2]. We expect that, with the increasing popularity of DL, more training work would be conducted in non-dedicated clusters.

The two types of clusters differ significantly with regard to the straggling factors. In dedicated clusters, all workers have a similar computation capability, and the stragglers are usually transient. However, in non-dedicated clusters with heterogeneous hardwares or dynamic resources, workers’ computational capabilities can be highly different. The resulting straggling factors are usually non-transient and salient, in face of which, as we show next, the distributed DL process would suffer remarkable performance degradation.
2.2 Existing Coordinating Schemes

Given the prevalence of stragglers, how to coordinate workers critically affects the learning performance. Several Coordinating schemes have been developed in this regard: Bulk Synchronous Parallel (BSP), Asynchronous Parallel (ASP), Stale Synchronous Parallel (SSP) and its extension DynSSP. However, they all have limitations in the presence of stragglers in non-dedicated clusters.

**Bulk Synchronous Parallel.** In BSP, workers synchronize at the end-of-iteration barrier and cannot proceed until the model parameters have been fully updated by all workers. This ensures the computation to use up-to-date parameters, which in turn yields high-quality model refinement. However, the price paid is low hardware efficiency: fast workers have to wait for stragglers to complete in each iteration, wasting computing cycles and delaying the training progress.

**Asynchronous Parallel.** To avoid resource wastage, many learning problems are solved in the ASP model, where workers asynchronously proceed to the next iteration before the parameters have been fully updated. The ASP model wastes no computing cycle and attains highest hardware efficiency. However, this requires synchronization, the computation often uses stale parameters, which may require much more iterations to converge and yield low-quality updates that poison the learning performance.

**Stale Synchronous Parallel.** The SSP model [26, 41] comes as a middle ground between BSP and ASP. In SSP, fast workers synchronize with stragglers only when the staleness of parameters (the number of missing updates) is above a bounded amount. However, when extending SSP, it also behaves poorly facing non-transient stragglers.

**DynSSP.** The recently proposed DynSSP [44] is built upon SSP, and it incorporates heterogeneity-aware dynamic learning rate to achieve improved statistical efficiency. However, because it has same hardware efficiency as SSP, it also behaves poorly facing non-transient stragglers.

2.3 Load-balancing DL workloads

In typical load balancing approaches, there are two basic steps: first to detect stragglers, and second to tackle stragglers by load adjustment. For example, FlexRR [38], a recently proposed load-balancing approach for Machine Learning workloads, detects stragglers by measuring the task progress and computes the iteration boundaries. Here we justify that with the following properties of DL workloads:

- **[P1] Short iterations.** DL iterations are usually very short [13, 21, 37], because the sample batches shall be small enough to avoid sacrificing the learning efficiency [18, 51]. This rules out the intra-iteration straggler detecting and load transferring steps adopted in FlexRR, which are too time-consuming and meanwhile bring non-negligible computation and communication overheads.

- **[P2] Strong iterative-ness.** DL training process is composed of thousands of iterations sharing the identical training operations. When analyzing a worker’s progressing state, historical information in the past iterations plays a valuable reference.

- **[P3] Load inner-homogeneity.** For DL workloads, each training sample consumes a fixed number of CPU cycles, which is totally determined by the pre-defined DL model. Therefore, the batch size accurately controls the iteration load. This is different from traditional workloads, like SQL queries running under MapReduce [30] or Spark [64], where the additional load brought by one data item is not constant but dependent to the item value (e.g., whether the key is selected for further processing).

- **[P4] Load indivisibility.** From an engineering perspective, adjusting DL loads at runtime is actually infeasible. In state-of-the-art DL frameworks like TensorFlow [13], MXNet [21] and Caffe [43], all samples in the input batch would be processed not one by one but—for fast processing speed—as an indivisible matrix, e.g., a tensor in TensorFlow or a NDArray in MXNet. The size of such matrix, i.e., the worker’s load, is fixed during the iterations.
The above DL properties indicate that stragglers shall and could be detected at the iteration boundaries, and meanwhile batch size is an ideal tool to adjust the loads of workers. This motivates our load-balancing solution, worker-adaptive batch sizing, which will be elaborated in following sections.

### 3 LOAD-BALANCED BSP

In this section, we introduce Load-balanced Bulk Synchronous Parallel (LB-BSP), a BSP-based worker-coordinating scheme for distributed DL. The basic intuition of LB-BSP is to adaptively adjust the batch size of each worker so as to equalize their batch processing times. We first formulate the problem, and then respectively elaborate our solutions for CPU-clusters and GPU-clusters.

#### 3.1 Problem Formulation

By batch processing time, we refer to the time spent on the whole procedure that a worker pulls model parameters from the PS, calculates the gradient from its sample batch, and finally pushes it to the PS. Here we divide the batch processing time \( t \) into the following two parts:

- **computation time** \( t^p \)—the time taken to compute gradient from the sample batch. Under a given environment, it monotonously increases with the worker batch size.
- **communication time** \( t^m \)—the time taken to transmit parameters/gradient from/to the PS. The transmission amount is decided by the DL model, irrespective of the batch size.

With LB-BSP, our objective is to equalize the batch processing time by right-sizing the batches for each worker. This objective can be described as an optimization problem. That is, given \( n \) workers with initial batch size \( x \), we want to find the worker batch sizes \( x = (x_1, x_2, ..., x_n) \) that can minimize the maximal batch processing time among all workers:

\[
\begin{align*}
& \text{Minimize} \quad \max_{i \in \{1,2,...,n\}} t_i, \\
& \text{subject to} \quad t_i = t^p_i + t^m_i, \quad i = 1, \ldots, n; \\
& \quad t^p_i = \Gamma_i(x_i), \quad i = 1, \ldots, n; \\
& \quad \sum_{i=1}^n x_i = X.
\end{align*}
\]

Here \( \Gamma_i(\cdot) \) is the function between worker-\( i \)'s computation time \( t^p_i \) and its batch size \( x_i \); by the last constraint where

\( X = nx \), we want to ensure that the total number of samples processed in each iteration is the same as in vanilla BSP. Table 1 summaries the important notations.

Next, we will customize and solve this optimization problem respectively for CPU-clusters and GPU-clusters, based on their particular execution characteristics.

#### 3.2 LB-BSP in CPU-clusters

CPU-clusters refer to those clusters without any computing accelerators like GPUs. Although training with CPU is slow, it still shares the DL market due to the prevalence of idled CPU resources and the scarcity and expensiveness of GPUs [22, 29]. For CPU-clusters, the initial optimization problem can be simplified from the following aspects:

1. **Computation Time**

   - On CPU-workers, the forward/backward propagation for gradient calculation is usually computationally intensive [40]. In our EC2 measurements, when training Inception-V3 model on ImageNet dataset with 32 c5.2xlarge instances, the computation time \( t^p \) takes up more than 99% of the batch processing time \( t \). Thus, as a reasonable approximation, we can consider \( t^p \) as \( t \).

2. **Communication Time**

   - In a DL iteration, all the samples share the identical training operations and consume a fixed amount of CPU cycles. Therefore, the computation time \( t^p \) is strictly proportional to the batch size. To further confirm that, we respectively train the ResNet-32 and Inception-V3 model (introduced in §5.1) with different types of EC2 instances.

   - We vary the batch size \( x \) and record the corresponded computation time \( t^p \) in Fig. 3. The measurement results suggest strong linearity between \( x \) and \( t^p \) in each case. Let \( \nu \) be the ratio of \( x \) to \( t^p \), i.e., the sample processing speed, then we have \( t^p = \Gamma(x) = x/\nu \).

   - Given the above simplifications, we can easily get the solution to the optimization problem described in Eq. 3: \( x_i = \frac{c_i}{\nu} \times X \). Hence, to get the targeted batch size on each worker, we only need to know their sample processing speeds.

   - However, in typically non-dedicated environments—like training with the leftover resources of shared clusters, the sample processing speed is not a constant but varies with

---

**Table 1: Summary of important notations.**

| Symbol | Description |
|-------|-------------|
| \( t \) | batch processing time |
| \( t^p \) | computation time |
| \( t^m \) | communication time |
| \( \Gamma_i(\cdot) \) | function of \( x \) and \( t_i \) |
| \( x_i \) | sample processing speed |
| \( X \) | total number of samples |
| \( n \) | total number of workers |

---

\(^1\) Communication and computation might be partially overlapped under TensorFlow and MXNet [67]; for clarity, the communication time in our definition excludes the overlapped periods.
the available resources. Because batch size must be specified prior to each iteration (§2.3), we need to predict each worker’s sample processing speed before the iteration starts.

3.2.1 Predicting Sample Processing Speed.

Accurate speed prediction is important but difficult. When training DL models at a large scale under LB-BSP, the prediction error for merely one worker could delay all other workers. Speed prediction is also challenging, because non-dedicated CPU-clusters usually suffer both transient and non-transient stragglers: ideally, a good prediction approach shall be robust to random perturbations, and can meanwhile agilely react to the resource variations. We have surveyed a series of prediction technologies, and finally find NARX [32, 36] is the most appropriate one.

Potential methods. An intuitive prediction approach is to simply use the speed of the last iteration as the predicted speed for the iteration to start. However, this approach is not robust to temporary speed perturbations. Another intuitive approach is to use the Exponential Moving Average (EMA) as the predicted speed. This approach, by considering the older speeds with an exponentially decreasing weights, is robust to transient fluctuations. However, EMA fails to react timely to sharp speed changes caused by sudden resource variations.

The speed prediction problem facing us is basically a classical research problem—time series prediction, for which many statistical or learning based techniques have been proposed. The typical statistical approach is Autoregressive Integrated Moving Average (ARIMA) [34], which makes predictions based on statistics like moving average and the variation of deviations. Later on, models based on Recurrent Neural Network (RNN) [24] (like plain RNN and LSTM [42]), due to their ability to maintain inner memory, have been applied in forecasting real-world time series like stock price [56] or transport flow [55].

However, for our problem, the prediction performance of all the above approaches is limited by their blindness to underlying resources. Actually, the sample processing speed varies with the CPU/memory resources. This is supported by Fig. 4, in which the DL processes are slowed down after we restrict their resource usage. Taking those driving resources into consideration can help to distinguish the non-transient stragglers factors from the random perturbations. We find that the Nonlinear AutoRegressive eXogenous (NARX) model [32, 36] is a good fit for that purpose.

NARX. The NARX model we use is basically an extended recurrent neural network that takes three series as inputs: past values of sample processing speeds (s), current and past values of the two driving resources: CPU/memory usage (c/m). Essentially, NARX aims to learn a nonlinear function F(·) between predicted speed and a limited view (specified with a look-back window) of the input series:

$$v^k = F(v^{k-1}, \ldots, v^{k-d_v}, c^k, \ldots, c^{k-d_c}, m^k, \ldots, m^{k-d_m}).$$  \hspace{1cm} (4)

Here $v^k$, $c^k$ and $m^k$ respectively represents the value of speed, CPU and memory usage in iteration $k$; $d_v$, $d_c$ and $d_m$ represent the corresponded look-back window size of each series. Fig. 5 shows the unfolded architecture of NARX, in which the values of each series within the look-back window are fed into a feedforward neural network.

In practice, to ensure high prediction accuracy, we maintain a NARX model for each worker, which is trained online with the incrementally collected execution data (i.e., v, c and m). To avoid high model complexity, as in other prediction works [16, 19], the look-back window sizes for all the three input series are set to 2, and we include only one hidden layer in the feedforward network. Such a simple NARX model, with less than 20 parameters, can avoid over-fitting and achieve fast convergence. In practice (§5.4), we find it can meanwhile make predictions accurate enough, beating all those potential approaches we surveyed.

3.3 LB-BSP in GPU-clusters

Training DL models in GPU-clusters, i.e., with GPU-equipped workers, is prevalent due to their strong parallel processing
capabilities [37, 67]. Compared with CPUs, GPUs have much more kernels for concurrent data processing. Thus, the relationship between a GPU-worker’s computation time \( t^p \) and its batch size \( x \) is more complicated.

To demystify the relationship \( \Gamma(\cdot) \) between \( t^p \) and \( x \) for GPU-workers, we have conducted a series of measurements in EC2 GPU instances, as shown in Fig. 6. The models we run are a 7-layer Convolutional Neural Network and Inception-V3; the revealed relationships are all non-linear. In particular, we have following observations:

1) **Minimum Saturation Point** \((x^s)\)—the threshold of batch size \( x \) below which the computation time \( t^p \) would be almost constant. That point \( x^s \) exists because, when \( x \) is too small that only a portion of the parallel computing kernels are utilized, GPU would be unsaturated and those busy kernels then dominate \( t^p \) [49, 52]. Generally, \( x^s \) would be smaller for datasets with larger samples or more complicated DL models. Setting \( x \) smaller than \( x^s \) unnecessarily wastes resources and meanwhile does not help reduce \( t^p \), so the optimal solution of \( x \) would always be no less than \( x^s \).

2) **Out-Of-Memory Point** \((x^o)\)—the threshold of \( x \) above which the GPU would encounter Out-Of-Memory error. During GPU computing, the data must reside on the GPU DRAM, which is fast but limited [53, 62]. To avoid the Out-Of-Memory error, \( x \) must be set no larger than \( x^o \).

3) **Linearity between \( x^s \) and \( x^o \)**. Within the interval \([x^s, x^o]\), the computation time \( t^p \) in both figures increases linearly with \( x \). Let \( m \) be the line slope and \( b \) be the intercept, then we have \( t^p = \Gamma(x) = m \times x + b \).

Those observations are consistent with our later measurements for ResNet-32 (Fig. 12 in §5.5). Therefore, for GPU-clusters, the non-linear constraint \( t^p = \Gamma_i(x_i) \) in Eq. 3 can be equivalently translated into the following linear constraints:

\[
t^p_i = m_i \times x_i + b_i, \quad i = 1, \ldots, n; \\
x_i^s \leq x_i \leq x_i^o, \quad i = 1, \ldots, n; \\
\tag{5}
\]

Then, how to obtain the coefficients in the updated constraints—\( m_i, b_i, x_i^s, x_i^o \)—to solve the optimization problem? After surveying the current use cases of GPU, we find that fine-grained GPU sharing is very rare, due to the technical difficulties and large overheads [4]. For TensorFlow, shared use of GPU is not currently or even planned to be supported [3]. Therefore, we can limit our focus to cases where one GPU is dedicated to one DL work.

Performance of a dedicated GPU is stable, as indicated by Fig. 6, where the measurement in each case spans for 5 hours. Therefore, in GPU-clusters we don’t need to make online predictions for the coefficients \( m_i, b_i, x_i^s, x_i^o \); they can be obtained by offline profiling or with a fast profiling phase at the beginning of the training process.

Additionally, because a GPU-worker is typically tens of times faster than a CPU-worker, the communication time \( t^m \) in GPU-clusters is no longer negligible. Thus it shall be predicted for each iteration. Different from the speed prediction problem in §3.2.1, for dedicated GPU-workers there are no evident driving factors for \( t^m \), and meanwhile the external network state is relatively stable during the much shorter GPU iterations, so for simplicity we can use EMA for \( t^m \) estimation.

After we have obtained all those coefficients and the predicted communication time \( t^m \), the resultant linear optimization problem can be easily solved with existing mathematical toolboxes.

### 3.4 Weighted Gradient Aggregation

Recall that our objective under LB-BSP is to simultaneously achieve best hardware efficiency and statistical efficiency. By solving the optimization problem in Eq. 3, we eliminate resource wastage and realize the first part—best hardware efficiency as ASP. Next, we show that with weighted gradient aggregation, LB-BSP can also make the second part—best statistical efficiency as BSP.

One obstacle facing LB-BSP is that workers have different batch sizes under it. Due to such heterogeneous batch sizes, the naively aggregated gradient would be biased, impairing the statistical efficiency. We will first elaborate that problem and then propose weighted gradient aggregation to solve it.

**Naive aggregation causes biased gradient under LB-BSP.**

Under BSP, the aggregated gradient \( g \) for parameter updating is the naive average of the gradients from workers. Suppose there are \( n \) workers and \( g_i \) is the gradient calculated on worker-\( i \) (\( i = 1, 2, \ldots, n \)), then

\[
g = \frac{1}{n} \sum_{i=1}^{n} g_i, \quad \text{where} \quad g_i = \frac{1}{|B_i|} \sum_{s \in B_i} \nabla l(s, \omega). \tag{6}
\]

Here \( B_i \) is the batch on worker-\( i \). Getting rid of \( g_i \), we have

\[
g = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{|B_i|} \sum_{s \in B_i} \nabla l(s, \omega) = \sum_{i=1}^{n} \sum_{s \in B_i} \frac{1}{|B_i|} \cdot \nabla l(s, \omega). \tag{7}
\]
This implies that, when workers have different batch sizes \(\{|B_j|\}\) under LB-BSP, the ponderance of different samples in parameter updating, i.e., \(\frac{1}{n||B_j||}\), is also different. This violates the DL principle that each sample shall be treated equally in the training process, and the resultant gradient \(g\) is biased to samples in small batches.

**Weighted Gradient Aggregation.** To avoid biased gradient, we propose weighted gradient aggregation—using a worker’s batch size as the weight when aggregating its gradient. Then, suppose the total batch size \(\sum_{j=1}^{n} |B_j| = X\), we have

\[
g = \frac{1}{\sum_{j=1}^{n} |B_j|} \sum_{j=1}^{n} |B_j| \cdot g_j = \sum_{j=1}^{n} \frac{1}{X} \cdot \nabla l(s, \omega). \quad (8)
\]

Obviously, each sample always plays an equal role in parameters updating, regardless of the batch size heterogeneity. After the fix, LB-BSP has following properties:

1) **No parameter staleness.** With barriers, LB-BSP ensures that all workers share the latest model parameters.

2) **Identical per-iteration progress as BSP.** As a constraint in Eq. 3, the total number of samples processed in each iteration under LB-BSP is the same with BSP.

3) **Unbiased gradient.** With weighted gradient aggregation, the model is updated with the unbiased average of gradients from all the samples processed in the iteration.

Given the above properties, for training dataset satisfying the i.i.d. requirement, i.e., samples being independent and identically distributed, LB-BSP can always achieve the identical statistical efficiency with BSP, which is known to be optimal. This will be confirmed in our later evaluations.

### 3.5 Discussions

**Worker-adaptive batch sizing in ASP.** Although LB-BSP is built upon BSP, its basic philosophy of worker-adaptive batch sizing can also be integrated into ASP to avoid stale parameters. Nonetheless, under ASP every worker proceeds independently without barriers, so it’s hard to simultaneously and coordinately adjust worker batch size, leading to a performance worse than LB-BSP.

**Uneven sample access frequency.** In distributed DL, a worker may have access to the whole or only a shard of the training dataset. In the latter case, under LB-BSP, when faster workers keep training with larger batches, samples in their data shards would be accessed more frequently, leading to biased training results if the initial dataset is not well shuffled. Actually this problem widely exists under each worker-coordinating scheme. For example, under BSP, when the training datasets are not equally partitioned among workers, samples in small shards would be accessed more frequently; under ASP, similar with LB-BSP, samples possessed by faster workers would be trained with more frequently. We will address this problem in our subsequent work.

**End-of-iteration network burst.** One well-known problem of BSP is the end-of-iteration network burst: in GPU clusters, when many workers communicate with the PS simultaneously at the barrier, the network would be a bottleneck. This problem is orthogonal to our work, and the methods proposed by other works like Poseidon [67] can be adopted to mitigate that bottleneck.

### 4 IMPLEMENTATION

Our LB-BSP implementation pursues following objectives:

- **Generally applicable.** We want LB-BSP to be pluggable for multiple DL frameworks. Thus, we implement its core algorithm in a separated Python Library, BatchSizeManager, which is applicable for both TensorFlow and MXNet.

- **Light-weight.** We want to minimize the overheads of LB-BSP on normal DL process. So we adopt a series of techniques like non-blocking batch-size updating, Thrift RPC, NARX model reuse, etc.

Fig. 7 shows the LB-BSP architecture for CPU-clusters and Alg. 1 describes the detailed workflow. At the start of iteration \(k\), each worker pushes its execution states (last sample processing speed \(v^{k-1}\), current CPU/RAM usage \(c^k/m^k\)) to the BatchSizeManager and pulls the new batch size \(|B^k|\).

The LB-BSP implementation for GPU-clusters shares a similar architecture, but the reported worker state is the communication time and the batch sizes are calculated without NARX models. Moreover, while the batch size updating


```python
new_batch_size = rpcClient.update_batch_size(taskid, last_speed, cpu_usage, memory_usage)
```

After receiving the RPC calls, BatchSizeManager would use the NARX models to make speed predictions and decide each worker’s batch size for balanced loads. Yet, one challenge is how to efficiently train the NARX models online, with the incrementally collected execution information, which will be addressed next.

### 4.2 Online NARX Training

In our implementation, the NARX models are written in Keras [5], a high-level neural network API. Because accurate NARX training requires enough input samples (one sample corresponds to one DL iteration), we use the NARX models for prediction only after the first 500 iterations. In practice we find that 500 samples are enough for accurately training our NARX models, which are quite simple (§3.2.1). Within the first 500 iterations, we can use EMA or, if that DL job is recurring, the past NARX models trained in former runs.

Moreover, for fast convergence, parameters of the NARX models are initialized by reusing available pre-trained models, even though those models are trained for other workers or DL applications. Such a model-reuse approach has been shown [63] to largely speed up the model convergence. Besides, we enable early stopping in the training process. That training process would stop if the loss value reduces less than 0.0001 for 4 consecutive steps. Furthermore, to pursue low overhead, we train the NARX models in separated threads with low priority (the priority is explicitly specified through Python `threading2` library [7]), and models of different workers are trained in a round-robin manner—at any time the number of models that can be trained concurrently is limited to half of the total number of CPU cores.

### 5 EVALUATION

Our evaluation seeks to answer the following questions:

- **How does LB-BSP perform on DL convergence?** In face of non-transient stragglers (§5.2), LB-BSP outperforms all existing schemes by being optimal in both hardware efficiency and statistical efficiency. In an emulated production cluster (§5.3), it can speed up the DL convergence by more than 2x. In another GPU-cluster (§5.5), it can well adapt to the network variation and improve the hardware efficiency by around 41%, compared with BSP.

- **How effective is the NARX prediction approach?** By diving deep into the speed prediction results (§5.4), we confirm that NARX is robust to random variations and can meanwhile react promptly to non-transient stragglers; as a result, it brings a convergence speed 15% better than that with the second best approach.

- **How about the extra overhead of LB-BSP?** We evaluate how the batch adjusting process would prolong the iteration time in clusters of different scales (§5.6). That gap is less than 1.1% even in a large cluster with 96 workers.

#### 5.1 Experimental Setup

**Experimental Platforms.** We conduct our experiments on three different Amazon EC2 clusters. **Cluster-A**, for experiments with finely-controlled stragglers, is composed of 1 PS server (c5.4xlarge) and 32 workers (c5.2xlarge), each
worker having 8 cores and 16GB memory. We include 32 workers in the cluster because it’s a reported scale at which distributed DL applications empirically benefit most [67]. Cluster-B contains 32 worker nodes of different types to emulate the production environment. Cluster-C is a heterogeneous cluster with 8 GPU-instances, used for evaluating LB-BSP in GPU-clusters. Operating system of all the instances is 64-bit Ubuntu Server 16.04 LTS (HVM). For generality, in Cluster-A and Cluster-B we run TensorFlow, while in Cluster-C we run MXNet.

Datasets and Models. We use two typical image classification datasets. (1) CIFAR-10 [45], a dataset containing 50K 32 × 32 colored images of 10 classes; (2) ILSVRC12 [58], a subset of ImageNet22K that has 1.28 million of training images in 1000 categories. On CIFAR-10 dataset, we train the ResNet-32 model, a 32-layer deep residual neural network [39]; on ImageNet dataset, we train the Inception-V3 model [9], an improved version of GoogleLeNet [60]. The initial batch size of each worker is set to 128 for ResNet-32, and 32 for Inception-V3. Meanwhile, the initial learning rate is set to 0.1 for ResNet-32, and 0.045 for Inception-V3. All those hyper-parameters are the default settings in the TensorFlow official github repository [10].

Metrics. The model training efficiency is quantified as the runtime required to converge to a given tolerance. It’s further decoupled into two aspects: statistical efficiency and hardware efficiency. Statistical efficiency is measured as the number of updates (gradients) the PS receives until model convergence, and hardware efficiency is measured as the per-update time—the average time each update takes.

Schemes Compared. We compare LB-BSP directly with BSP, ASP, SSP, and indirectly with DynSSP. TensorFlow currently only support ASP and BSP, so we implemented SSP with a plugged worker-coordinator, which will dictate the fastest workers to wait as long as the staleness threshold is reached. For DynSSP, we didn’t implement it because its underlying data structure is not open-sourced for TensorFlow or MXNet, but we can indirectly justify the performance relationship between DynSSP and LB-BSP, bridged by BSP and SSP. Note that FlexRR is not compared because it assumes divisible loads and is simply incompatible with TensorFlow or MXNet (§2.3).

5.2 Results with Fine-tuned Stragglers

Straggler Injection. We use Cluster-A to evaluate the performance of LB-BSP in different heterogeneity levels. For that purpose, we inject artificial stragglers by running on each worker a competing process; that process consumes designated cpu cycles and memory space. In particular, to realize dynamicity, we make that competing process periodically run or sleep with certain probability; to realize heterogeneity, in different workers that probability and the resource consumption of the competing process are different. By tuning those configurations, we create three heterogeneity levels: Homo—no stragglers at all; Hetero-L2 (L3)—sample process periodically run or sleep with certain probability; to realize heterogeneity, in different workers that probability and the resource consumption of the competing process are different. By running those configurations, we create three heterogeneity levels: Homo—no stragglers at all; Hetero-L2 (L3)—sample process periodically run or sleep with certain probability; to realize heterogeneity, in different workers that probability and the resource consumption of the competing process are different.

CIFAR-10 Results. Fig. 8 shows the performance under different synchronization schemes when training ResNet-32 model on CIFAR-10 dataset. Here the training process converges, in our definition, when the loss function is below the targeted value (0.6) for 10 consecutive iterations. Besides, in SSP the staleness threshold is set to 10 iterations (clocks). Moreover, we repeat each training process for 3 times, and report min/max value of each metric via error bars.

As revealed by Fig. 8, LB-BSP is the best one among all the schemes, its superiority increasing with the level of heterogeneity. More specifically, in each case, LB-BSP requires the same number of updates for convergence as BSP, and
### Table 2: Cluster-B composition.

| Instance Type | CPU, Mem (core, GiB) | Num |
|---------------|----------------------|-----|
| m4.2xlarge    | (8, 32)              | 17  |
| c3.2xlarge    | (8, 16)              | 10  |
| r4.2xlarge    | (8, 64)              | 2   |
| m4.4xlarge    | (16, 64)             | 2   |
| m4.xlarge     | (4, 16)              | 1   |

### Figure 10: For both models in Cluster-B, convergence speed under LB-BSP is > 2× of that under BSP.

### 5.3 Exp. in Emulated Production Cluster

#### Cluster Setup
To further verify the effectiveness of LB-BSP in production environments, we resort to Cluster-B—an emulated production cluster that is carefully designed based on the released Google traces [57]. Those traces disclosed the machine configurations (CPU/Memory) of a reported Google cluster (in normalized form), together with the activities of all the involved jobs/tasks during a selected month—including their resource consumptions and start/end times. When creating Cluster-B, we scale down the totally 12,583 machines of that Google cluster to 32 Amazon EC2 instances, with the former’s hardware heterogeneity proportionally preserved—by accordingly selecting the EC2 instance types and the instance quantity of each type. Table 2 shows the instance composition of Cluster-B, which actually serves as the most general case where each worker machine for the training process is randomly selected from that reported Google cluster.

Meanwhile, we have also emulated the resource dynamicty of that Google cluster. For each instance in Cluster-B, we randomly map it to one machine in the Google cluster; during the training process, in that instance we launch a set of faked tasks that share exactly the same start/end times and resource (CPU/memory) consumptions with those submitted to the mapped Google machine.

#### General performance of LB-BSP
By building up Cluster-B, we are actually emulating the scenarios where a shared-cluster operator utilizes the left-over resources for training DL models in the background. In Cluster-B we respectively train the ResNet-32 and Inception-V3 model for a fixed amount of time. Fig. 10 shows their convergence curves, which are smoothed with a window size of 30 iterations. We find that, given the particular convergence criterion on loss value (0.6 for ResNet-32 and 11.0 for Inception-V3), the convergence speed under LB-BSP is more than 2× of that under BSP for both models. Such a huge improvement suggests it be highly beneficial to load-balance DL workloads with LB-BSP in shared production clusters.
Table 3: RMSE and normalized per-update time when applying different prediction approaches in LB-BSP.

| Method   | Configuration | RMSE | Normalized per-update Time |
|----------|---------------|------|----------------------------|
| Memoryless | -             | 11.85| 0.584                      |
| EMA      | $\alpha=0.2$ | 7.35 | 0.469                      |
| ARIMA    | $(p,d,q)=(2,2,1)$ | 9.57 | 0.523                      |
| SimpleRNN | look-back=2   | 8.34 | 0.491                      |
| LSTM     | look-back=2   | 9.17 | 0.515                      |
| NARX     | look-back=2   | 4.78 | 0.423                      |

5.4 Deep Dive into NARX Performance

Accurate sample processing speed prediction is crucial for the performance of LB-BSP. As elaborated in Sec. 3.2.1, we have resorted to the NARX model for speed prediction. Then, how on earth does the NARX approach perform? We answer that by diving deep into the speed prediction results in the above experiments conducted in Cluster-B.

**Visual analysis.** To get a visual understanding of NARX prediction performance, we randomly select a period (iteration 1200~1300) from the ResNet-32 training process on one m4.2xlarge instance of Cluster-B; the actual and predicted sample processing speeds are presented in Fig. 11. From it we observe that the benefit of NARX is twofold. On the one hand, NARX is robust to transient perturbations: when there are “spikes” (like the sharp wave around iteration 1206) in the actual speed curve, the predicted curve fluctuates much less. This is because NARX predicts also with the worker’s available memory and CPU amounts, which are relatively stable during those “spikes”. On the other hand, when the actual speed increases not for randomness but for non-transient deterministic factors like increased CPU/memory resources (e.g., around iteration 1270), the predicted speed can promptly catch up with the actual speed.

**Comparison with other approaches.** We further compare NARX with other approaches we surveyed in §3.2.1, as listed in Table. 3. Here the memoryless method means to simply take last iteration’s sample processing speed as the predicted one. Regarding the EMA approach, the smoothing factor $\alpha$ (weight of the latest observation) is set to be 0.2. As for the statistical prediction approach—ARIMA, its order of the autoregressive model ($p$), degree of differencing ($d$), and order of the moving average ($q$) are respectively set to 2, 2 and 1, based on its model selection techniques [54]. Finally, for SimpleRNN (plain RNN) and LSTM, their look-back window size is set to 2, the same as in NARX.

Then, we replace the NARX approach respectively with each of those prediction approaches when training ResNet-32 model under LB-BSP in Cluster-B. For each approach, we record the average root-mean-square error (RMSE) of the prediction results, as well as the corresponded per-update time (normalized by the per-update time under BSP). From the results in Table. 3, the NARX approach, due to its ability to perceive CPU/memory resource variations, can do the best among all those candidates—it surpasses the second best by around 40% in RMSE and 15% in per-update time.

5.5 Micro-benchmarks in GPU Cluster

**Cluster Setup.** We resort to Cluster-C to verify the effectiveness of LB-BSP in GPU-clusters. Cluster-C contains 8 single-GPU workers: four g2.2xlarge instances (with a NVIDIA GRID K520 GPU), two p2.xlarge instances (with a NVIDIA Tesla K-80 GPU), and two g3.4xlarge instances (with a NVIDIA Tesla M60 GPU). Each worker is installed with CUDA-8.0 and cuDNN-5. Meanwhile, the DL model trained is ResNet-32 (on CIFAR-10 dataset), and the initial batch size is 380. For generality, in Cluster-C the deployed DL framework is switched to MXNet.

Fig. 12 shows the profiled relationships between computation time and batch size, i.e., $\Gamma(\cdot)$ (§3.1) for GPU-instances in Cluster-C. In particular, the $[x^o, x^o]$ pairs of g2.2x, p2.x and g3.4x instances are [58, 384], [92, 1184] and [103, 788], respectively.

Recall that LB-BSP in GPU-clusters can react to hardware heterogeneity and network variations. To verify that, we select one g2.2x worker as the testee, and periodically rotate its link bandwidth (with Wonder Shaper [12]) between an abundant state (480Mbps) and a deficient state (160Mbps). For performance comparison between LB-BSP and BSP, we record the testee’s temporal batch size and the corresponded iteration time, as shown in Fig. 13.

LB-BSP outperforms BSP in two aspects. First, in face of the hardware (GPU) heterogeneity, LB-BSP sets each worker’s batch size based on their computing capabilities: the batch size of the g2.2xlarge instances is adjusted from 380 to 235
to compensate for their relatively low speeds. This brings a performance improvement of around 27%. Second, when the testee’s bandwidth drops, its prolonged communication time would be perceived under LB-BSP (with EMA), which further reduces its batch size as a remediation. This limits the negative influence of bandwidth drop to less than 3%. As a result, compared with BSP, LB-BSP improves the hardware efficiency by totally more than 41%.

5.6 System Overhead and Scalability
In TensorFlow, the worker overheads brought by LB-BSP come from two perspectives. First, to get the sample processing speed in an iteration, we need to process the worker execution logs (Time1ine object). Besides, each worker shall communicate with the BatchSizeManager to get the updated batch size. For CPU-clusters, both processes are blocking (§4) and would take extra time in each iteration. Next, we measure those overheads at different cluster scales.

We respectively train ResNet-32 and Inception-V3 model for 1000 iterations in three clusters—first Cluster-A, then its enlarged versions with doubled/tripled size. Fig. 14 shows the average time respectively spent on log processing and batch size updating, normalized by the iteration time. Here the error bars show the \(5^{th}\)/\(95^{th}\) percentile. Even in the largest cluster with 96 workers, the total overheads are less than 1.1% of the iteration time for both models—such a cost (slowdown) is much less compared with the benefit (speedup) from LB-BSP.

Besides, we also confirm that performance of the PS is almost not affected by the co-located BatchSizeManager: we migrate the BatchSizeManager to a dedicated machine, and again train ResNet-32 in the 96-worker cluster. Our measurements show that, without BatchSizeManager, the push/pull interactions between the PS and workers are sped up merely by 1.3% on average. So there is no need to provision extra machine for BatchSizeManager.

6 RELATED WORK
Straggler Treatment. Stragglers have long been an annoying problem for parallel computing; basically the cures are of two kinds: load balancing and redundant execution.

Load balancing is the most fundamental methodology to solve the straggler problem. In traditional multi-core systems [14, 17, 33], work stealing has been exploited to balance the loads among different computing cores: once a core becomes idle, it will steal work from other busy ones. Recently, FlexRR [38] was proposed for load balancing in data-parallel iterative Machine Learning frameworks; it combines SSP with an intra-iteration work reassignment mechanism. As justified in §2.3, such fine-grained techniques do not fit DL workloads.

Redundant execution [15, 66] is typically adopted to mitigate stragglers in data analytics frameworks like MapReduce and Spark, and it works by speculatively launching multiple copies of the straggling tasks and picking up only the one finishing earliest. Recently, it has been also applied in distributed learning systems: Chen et al. [20] have proposed to train DL models with backup workers and use gradients from those finishing earlier. However, all those approaches are suboptimal: they merely mitigate the worst stragglers instead of fundamentally eliminating all the progress inconsistency; worse, they waste extra computing resources.

Batching Manipulation. Batching is necessary when the input is a long-lasting stream or too large to be processed all at once. In real-time streaming systems [61, 65], improper batching interval (size or time) would cause either low throughput or long end-to-end latency. Thus, some works [27, 68] have explored how to adaptively adjust the batching interval when faced with dynamic data rates or operating conditions. Yet, such adjustments focus only at the front-end batching interface, without involving the back-end parallel workers. Meanwhile, from an algorithmic perspective of DL, some [28, 31] have proposed to adaptively increase the batch size during the training process to yield faster convergence; those works are orthogonal to ours. To the best of our knowledge, we are the first that has identified the superiority of and then leveraged coordinated batch size tuning to load-balance different workers in the distributed DL training process.

7 CONCLUSION
In this work, we presented LB-BSP, an adaptive mechanism that improves the performance of distributed deep learning training by apportioning the workload to each worker according to its processing capability. Called worker-adaptive batch sizing, we mathematically formulate the problem as an optimization problem, and solve it for CPU-clusters and GPU-clusters, respectively. In particular, we employ a NARX model to predict the processing speed of workers in CPU-clusters. We have implemented LB-BSP as a Python library and integrated it into both TensorFlow and MXNet, two of the modern and most popular deep learning frameworks. Our extensive experiments have shown that it is able to outperform BSP by 2x.
REFERENCES

[1] Apache Thrift. https://thrift.apache.org/.
[2] EC2 Spot Instances. https://aws.amazon.com/ec2/spot/.
[3] GPU sharing is not supported in TensorFlow. https://github.com/tensorflow/tensorflow/issues/2210.
[4] GPU sharing is rare in practice. https://www.quora.com/Can-I-run-multiple-deep-learning-models-on-the-same-GPU/.
[5] Keras. https://keras.io/.
[6] Python Psutil. https://psutil.readthedocs.io/en/latest/.
[7] Python Threading2. https://pypi.python.org/pypi/threading2.
[8] Stress-ng: a tool to load and stress a computer system. http://manpages.ubuntu.com/manpages/artful/man1/stress-ng.1.html.
[9] TensorFlow Inception-V3. https://github.com/tensorflow/models/tree/master/inception.
[10] TensorFlow Models in Official Github Repository. https://github.com/tensorflow/models.
[11] TensorFlow Timeline. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/client/timeline.py.
[12] Wonder Shaper. https://github.com/chenc10/wondershaper.
[13] Robert D Blumofe and Charles E Leiserson. 1999. Scheduling multi-threaded computations by work stealing. JACM 46, 5 (1999), 720–748.
[14] Erasmo Cadenas, Wilfrido Rivera, Rafael Campos-Amezcua, and Roberto Cadenas. 2016. Wind speed forecasting using the NARX model, case: La Mata, Oaxaca, Mexico. Neural Computing and Applications 27, 8 (2016), 2417–2428.
[15] Jianmin Chen, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, and Andrew Ng. 2012. Large scale distributed deep networks. In NIPS.
[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In IEEE CVPR.
[17] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. 2017. Heterogeneity-aware distributed parameter servers. In ACM SIGMOD.
[18] Alex Krizhevsky and Geoffrey Hinton. 2009. Learning multiple layers of features from tiny images. (2009).
[19] Alex Krizhevsky and Geoffrey Hinton. 2009. Learning multiple layers of features from tiny images. (2009).
