# DLion: Decentralized Distributed Deep Learning in Micro-Clouds

Rankyung Hong
University of Minnesota
Minneapolis, MN, USA
hongx293@umn.edu

Abhishek Chandra
University of Minnesota
Minneapolis, MN, USA
chandra@umn.edu

## ABSTRACT

Deep learning (DL) is a popular technique for building models from large quantities of data such as pictures, videos, messages generated from edges devices at rapid pace all over the world. It is often infeasible to migrate large quantities of data from the edges to centralized data center(s) over WANs for training due to privacy, cost, and performance reasons. At the same time, training large DL models on edge devices is infeasible due to their limited resources. An attractive alternative for DL training distributed data is to use *micro-clouds*—small-scale clouds deployed near edge devices in multiple locations. However, micro-clouds present the challenges of both computation and network resource heterogeneity as well as dynamism. In this paper, we introduce *DLion*, a new and generic decentralized distributed DL system designed to address the key challenges in micro-cloud environments, in order to reduce overall training time and improve model accuracy. We present three key techniques in *DLion*: (1) *Weighted dynamic batching* to maximize data parallelism for dealing with heterogeneous and dynamic compute capacity, (2) *Per-link prioritized gradient exchange* to reduce communication overhead for model updates based on available network capacity, and (3) *Direct knowledge transfer* to improve model accuracy by merging the best performing model parameters. We build a prototype of *DLion* on top of TensorFlow and show that *DLion* achieves up to 4.2× speedup in an Amazon GPU cluster, and up to 2× speed up and 26% higher model accuracy in a CPU cluster over four state-of-the-art distributed DL systems.

## CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Computing methodologies** → **Machine learning**.

## KEYWORDS

Edge computing, Deep learning, Micro-clouds, Resource allocation

**ACM Reference Format:**
Rankyung Hong and Abhishek Chandra. 2021. DLion: Decentralized Distributed Deep Learning in Micro-Clouds. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '21), June 21–25, 2021, Virtual Event, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3431379.3460643

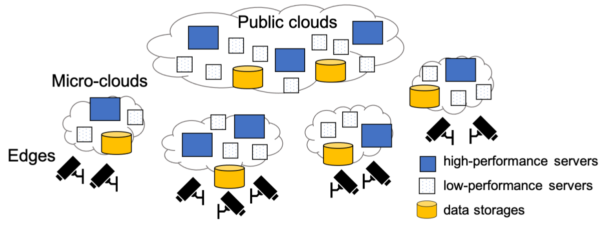


**Figure 1: Distributed deep learning (DL) in micro-clouds.**

## 1 INTRODUCTION

Deep learning (DL) is a popular technique to build models from large volumes of input data for applications in many domains [10, 15, 24]. Traditionally, DL models are trained in a cluster or data center environment as a one-time solution for a fixed set of training data. Recently, the advent of sensors, mobile, and IoT devices has led to increasingly large volumes of continuously generated data from the edge [33, 47, 49]. Such data has created the need for DL models to keep evolving using data continuously generated from edge devices across the globe, and could be used for online-learning or incremental-learning [6, 35, 37].

However, migrating such large amounts of data into centralized cloud(s) over WANs for training is likely to be prohibitive due to cost, performance, or privacy reasons. For instance, such data is hard to move because of WAN bandwidth constraints, or because it could contain a lot of personal information such as pictures or videos generated by user devices or recorded using surveillance cameras. The need for geo-distributed data analysis has also been shown for many other analytics tasks [17, 19, 23, 27].

Federated learning [5, 7, 28] has been proposed to train models at the edge without data movement. However, federated learning can only train much smaller-scale models like traditional machine learning algorithms due to limited resources of the edges, such as computation, storage, or energy, which are significantly constrained for training large deep learning models.

An attractive alternative is to carry out distributed deep learning across *micro-clouds* [4, 13, 14, 38]: an emerging type of infrastructure to support the exponentially growing large amounts of data generated by edge devices [20, 44, 50] such as surveillance cameras, mobile phones, or various sensors (Figure 1). Micro-clouds often provide better computation, storage, and energy capabilities than edges, and better data locality than public clouds. While there has been growing interest in using the edge for DL *inference* [2, 21, 45], where models trained in the cloud are deployed at the edge for faster inference; in this paper, we argue for the use of micro-cloud environments for DL *training* to efficiently build DL models in-situ and to support such learning.

There are two major system challenges in enabling distributed deep learning in micro-clouds.

**1. Compute resource heterogeneity and dynamism.** Different micro-clouds can have different number of servers equipped with different performance hardware. Servers in the micro-clouds can also be shared by other applications, so the available compute capacity may dynamically change.

**2. Network resource heterogeneity and dynamism.** Servers in a micro-cloud communicate with each other over LAN, whereas servers in different micro-clouds are connected via WAN. Network capacities in LANs may vary due to network resource contention with other applications, while bandwidths in WANs are much more scare and fluctuating than in LANs.

Existing distributed deep learning systems do not fully address these challenges. General purpose distributed DL systems [1, 8] do not consider system heterogeneity, resulting in much longer training times in the presence of compute heterogeneity or network bottlenecks. Recent research has addressed the network bottleneck issue by reducing the amounts of data transmitted over the network [18, 46], and system heterogeneity by skipping updates from stragglers [31]. However, these approaches typically trade off training time and model accuracy, and do not comprehensively consider all challenges of DL learning in micro-cloud environments. Federated learning [5, 7, 28] handles system heterogeneity, but DL training is not feasible due to extremely limited resources at edges.

In this paper, we present *DLion*, a new and generic distributed deep learning system designed for deep learning in heterogeneous environments such as micro-clouds. It builds on a decentralized system architecture because it naturally fits in such heterogeneous environments. The goal of *DLion* is to reduce training time while achieving higher model accuracy for distributed DL in micro-clouds. We assume that training data continuously generated from edges can be collected to nearby micro-clouds, and DL models then periodically start or resume training process with the collected data on *DLion* system. Input data collection/movement is an interesting research problem by itself, and we consider it to be beyond the scope of this paper.

*DLion* employs three key techniques: (1) *Weighted dynamic batching* to handle compute heterogeneity, (2) *Per-link prioritized gradient exchange* to handle network heterogeneity, and (3) *Direct knowledge transfer* to improve model accuracy. We implement a prototype of *DLion* on top of TensorFlow and deploy *DLion* on an Amazon GPU cluster and a local CPU cluster emulating micro-clouds. To evaluate the effectiveness of *DLion* on different types of applications, we train Cipher CNN model over CIFAR10 on CPUs and MobileNet over ImageNet on GPUs. Our experiments show the efficacy of *DLion* towards achieving accurate and efficient distributed DL in micro-clouds: *DLion* provides up to 4.2× speedup over four state-of-the-art distributed DL systems in the GPU cluster, and up to 2× speed up and 26% higher model accuracy in CPU cluster.

We make the following major contributions:

- We propose a new distributed deep learning system for DL training in heterogeneous environments such as micro-clouds. The system achieves shorter training time and higher model accuracy by handling system heterogeneity having dynamically changing computation capacity and network bandwidth.
- We design *DLion* as a generic and flexible system. The system is well-modularized, so it is very easy to adopt different distributed DL systems and algorithms in *DLion*. For example, we have implemented three state-of-the-art existing distributed DL systems (Ako [46], Gaia [18], and Hop [31]), with a maximum additional 23 lines of code per system.
- We build a *DLion* prototype on top of TensorFlow and demonstrate its effectiveness on both CPU-based and GPU-based clusters with Cipher and MobileNet models training. We show that *DLion* outperforms four state-of-the-art distributed DL systems in terms of training time and accuracy in micro-clouds environments.

## 2 BACKGROUND AND MOTIVATION

We first describe the general terms used in distributed deep learning and centralized and decentralized system architectures that state-of-the-art distributed DL systems utilize. We then introduce DL learning in micro-clouds and discuss challenges and motivation in designing a distributed DL system in such environments.

### 2.1 Distributed Deep Learning

**Deep Learning.** We consider supervised learning using minibatch stochastic gradient descent (SGD) [36] to minimize the loss value of the function $f$ over the training dataset $x$ (Eq. 1).

$$\text{Learning:} \quad \min_{x \in R^n} f(x; w) = \frac{1}{m} \sum_{i=1}^{m} f_i(x; w_t) \tag{1}$$

A DL model consists of a set of parameters called *weights*, and operators. The meaning of training a DL model is to find the best values for the weights, which lead to the smallest loss value.

$$\text{Gradient Calculation:} \quad g_t = \frac{1}{m} \sum_{i=1}^{m} \nabla_w f_i(x; w_t) \tag{2}$$

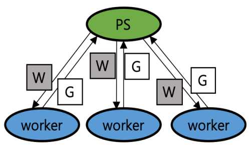
$$\text{Model (Weight) Update:} \quad w_{t+1} = w_t - \eta g_t \tag{3}$$

The weights are tuned by iterations of *gradient* $g_t$ calculation (Eq. 2) and model (*weight* $w_t$) update (Eq. 3) over minibatches. A *minibatch* is composed of $m$ training data samples from the training data $x$ and *batch size* indicates the size of a minibatch. An *iteration* indicates a cycle of gradient calculation and model update over a minibatch. An *epoch* indicates a set of iterations trained over one pass of the whole training data. Batch size and learning rate $\eta$ are tunable model parameters.

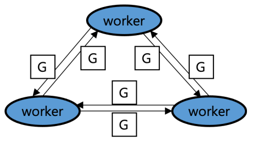
**Distributed Deep Learning.** Multiple workers in a cluster collaborate to train a model over partitioned training data. $n$ workers calculate their own gradients locally based on a minibatch size of $m$ in parallel. *Local batch size* indicates the minibatch size processed in a worker, which is $m$, and *global batch size* indicates the total batch size across all workers at an iteration, which is $n \times m$.

$$w_{t+1} = w_t - \eta \frac{1}{n} \sum_{j=1}^{n} \frac{1}{m} \sum_{i=1}^{m} \nabla_w f_i(x; w_t) \tag{4}$$

The model update in distributed DL systems follows Eq. 4. Weights are updated based on the average of the $n$ gradients.



(a) Centralized DL Systems



(b) Decentralized DL Systems

**Figure 2: Distributed DL system architectures**

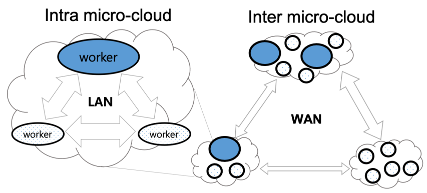


**Figure 3: Deep learning training in micro-clouds; workers in a micro-cloud communicate over LANs, whereas workers in different micro-clouds are connected over WANs. Large solid ellipses are workers with more computation capacity than small dotted ellipses.**

## 2.2 Distributed Deep Learning Systems

Distributed deep learning systems allow users to train their DL models using a cluster of multiple workers where training data are distributed. General purpose DL systems like TensorFlow [1] or MXNet [8] utilize central components called *parameter servers* (PS) for model updates in a *centralized* manner as shown in Figure 2a. All workers pull the synchronized weights from PSs to calculate gradients for the next iteration. However, in such a centralized architecture, PSs can be a communication bottleneck and the performance depends on their optimal deployment. On the other hand, *decentralized* distributed DL systems such as Ako [46], Hop [31] and Prague [30] synchronize models without PSs as shown in Figure 2b. Workers exchange local gradients with each other, and update their local model based on collected gradients. The workload imposed on PSs can be offloaded to all the workers and there is no PS placement problem in this architecture. *Hybrid* distributed DL systems such as Gaia [18] employ the decentralized architecture to exchange gradients between PSs over WANs while learning in a centralized manner in LANs.

## 2.3 DL Learning in Micro-Clouds

The system model that we target is to train deep learning models in micro-clouds as shown in Figure 3. Workers in a micro-cloud are connected over LANs, whereas workers in different micro-clouds communicate over WANs. Available compute capacities of individual workers may vary: some of them utilize only CPUs while others use GPUs, and the number of processing units may vary across workers. In addition, available compute and network capacities can fluctuate over time due to resource sharing with other applications.

## 2.4 Challenges and Motivation

There are two major challenges factored in designing a distributed DL system running on micro-clouds.

**Compute resource heterogeneity and dynamism.** *How to effectively handle different computation capacities of workers in micro-clouds to shorten training time while retaining model accuracy?* Unlike public clouds, micro-clouds may have various types of CPUs and/or GPUs and the number of units may vary per worker because different providers may set up their own micro-clouds for various application-specific purposes. Besides, the cluster can be shared by multiple applications, so the available computation resources can vary over time. The state-of-the-art distributed DL systems hold an implicit assumption that the computation power of workers are identical and steady, so the overall performance can be bounded by the slowest worker and system resource cannot be fully utilized, especially if they employ a synchronous training strategy. Thus, we study techniques to effectively work on such heterogeneous computation resource environments for faster training time with minimum impact on accuracy.

**Network resource heterogeneity and dynamism.** *How to effectively communicate with workers over various types of network environments ranging from LANs to WANs, from homogeneous to heterogeneous, and from steady to dynamic network bandwidths to reduce training time while improving model accuracy?* Most state-of-the-art distributed DL systems address the network bottleneck issue caused by scarce network capacity in distributed DL training. However, they target a certain type of network environment such as a homogeneous LAN or a heterogeneous WAN. Besides, these techniques reduce the running time, but typically at the cost of accuracy. We propose a general technique to work well in all types of network environments while achieving *both* faster training time and higher model accuracy.

## 3 OUR APPROACH: *DLION*

We propose *DLion*, a new decentralized distributed DL system for DL training in micro-clouds to address the challenges discussed above. In this section, we introduce design goals of *DLion* and describe key techniques and relevant exploratory studies to handle compute and network resource heterogeneity and dynamism for reducing training time and improving accuracy.

## 3.1 Design Goals and Overview

*DLion* is designed using a decentralized training architecture. The philosophy of a decentralized architecture fits well in distributed micro-cloud environments, which are inherently geo-distributed and loosely coupled. It also obviates the need for centralized control to consider where and how many parameter servers to deploy in the system. *DLion* has the following design goals to meet the challenges outlined above.

**Maximize data parallelism** to reduce training time with a minimal cost on accuracy by handling different computation capacities of workers.

**Reduce communication cost** among workers and guarantee model convergence by handling available network bandwidth for faster training while retaining accuracy.

**Improve model accuracy** by directly sharing knowledge among workers to compensate for any adverse impact on accuracy.

*DLion* employs three key techniques to accomplish the aforementioned goals: (1) **Weighted dynamic batching** (§ 3.2) to maximize data parallelism, (2) **Per-link prioritized gradient exchange** (§ 3.3) to reduce communication cost among workers and guarantee model convergence, and (3) **Direct knowledge transfer** (§ 3.4) to improve model accuracy. Figure 4 shows how these techniques fit into the workflow of *DLion* workers at each training iteration. We next describe these techniques in detail. The results in this section

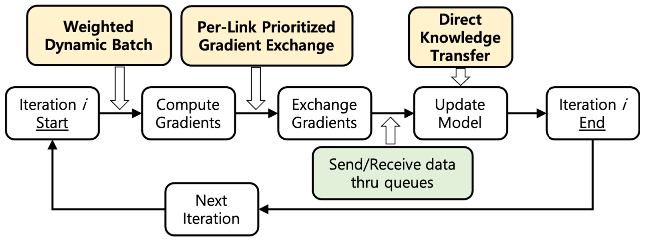


**Figure 4: Three key techniques of *DLion* and the workflow for a worker at each training iteration**

are based on experiments with Cipher model over CIFAR10 [26] in an emulated 6-worker cluster: more details of the experimental setup are provided in Section 5.1.

## 3.2 Weighted Dynamic Batching

Traditional DL training is performed in a single machine with a fixed batch size. If the training is performed on multiple workers in a cluster, the total size of batches processed by all workers for an iteration is called a *global batch size (GBS)*. The batch size processed at a machine for an iteration is called a *local batch size (LBS)*. GBS can increase by having either larger LBS or larger cluster size. In this paper, we do not focus on elastic cluster, and assume $n$ workers in the system.

There are advantages and disadvantages of training DL models with large global batch size [25, 40]. The major gain is to expedite training by processing an epoch within much shorter time. On the other hand, the drawback is that very large GBS deteriorates final model accuracy. Therefore, it is critical to find an appropriate GBS which results in shorter training time without significant accuracy drop. *DLion* presents a GBS controller that automatically adjusts GBS. It does not change the learning rate, as prior work[40] has shown that the same training performance can be achieved by varying GBS without decaying the learning rate.

Workers in micro-clouds may be heterogeneous, having different computational capacities. If workers have homogeneous computation capacity, it makes sense to have an even LBS share ($LBS = \frac{GBS}{n}$). However, it is inefficient in heterogeneous environments because more powerful workers would have to wait for less powerful workers to complete gradient computation. In addition, the computation power of individual workers can also be fluctuating over time. So *DLion* uses an LBS controller to automatically and dynamically assign a desired LBS per worker by considering available computation capacity at that time.

The high-level idea of the **weighted dynamic batching** technique is to dynamically determine the GBS and assign an appropriate LBS to workers based on their available computation capacities for better data parallelism. The technique is composed of three modules; (1) global batch size (GBS) controller, (2) local batch size (LBS) controller, and (3) weighted model update module.

**Global batch size (GBS) controller.** The GBS controller is designed to systematically increase GBS in an automatic manner, unlike traditional schedule-based approaches like [40] that require definitive user input such as a fixed total training epoch, and a good knowledge or intuition about how much and when to increase GBS prior to the training.

The design of the GBS controller is informed by two findings from our empirical results shown in Figure 5. This figure shows how

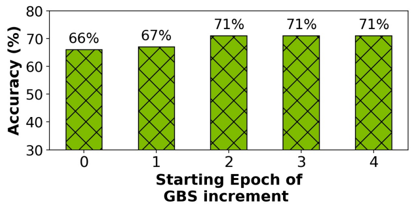


**Figure 5: Model accuracy for Cipher model trained for 30 epochs on 6 workers with initial LBS=32, as GBS is doubled beginning at different starting epochs. Accuracy does not change for the later epochs.**

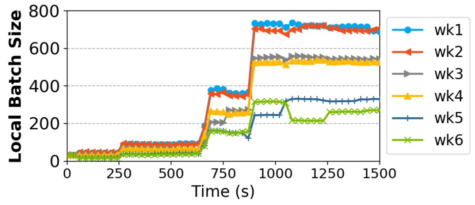


**Figure 6: An example of how local batch size is adjusted for workers with different compute capacities in a heterogeneous computation environment. The GBS increases around time=250s, 600s, and 800s, resulting in corresponding changes in LBS.**

the model accuracy varies as GBS is increased (doubled) beginning at different epochs during the training phase. The first finding is that the accuracy is lower if GBS rapidly increases at an early phase of the training (epoch = 0 or 1). The second finding is that the impact on the accuracy by GBS increment is relatively stable after the early phase of training (epoch=2 onwards). These findings agree with previous research [12, 40].

Based on these findings, the GBS controller adjusts GBS in two phases: warm-up and speed-up. In the warm-up phase, GBS increases in arithmetic progression ($GBS_{t+1} = GBS_t + C_{warmup}$). GBS increment stops if GBS is greater than 1% of the total training data size in order to avoid a drop in accuracy based on the first finding. In the speed-up phase, GBS increases in geometric progression ($GBS_{t+1} = GBS_t \times C_{speedup}$). GBS increment stops if GBS is greater than 10% of the total training data size according to the existing study [40]. System parameters for the GBS controller such as $C_{warmup}$, $C_{speedup}$, and duration of the two phases are configurable.

**Local batch size (LBS) controller.** The LBS controller automatically and dynamically determines LBS for each worker based on their available computation capacity. More powerful workers have larger LBS and less powerful workers have smaller LBS, and $GBS = \sum_{i=1}^{n} LBS_i$, where $LBS_i$ is LBS of worker $i$.

The LBS controller uses a simple and intuitive way to measure the available computation capacity of each worker. It is to find a relationship between local batch sizes and elapsed times to execute an iteration through a linear regression algorithm instead of collecting hardware specs of each worker. LBS controller calculates relative computation power ($RCP_i$) for each worker, a maximum local batch size that worker $i$ can process during a given unit time. After sharing $RCP_i$ with other workers, the LBS controller can determine a final LBS for each worker based on Eq. 5.

$$LBS_i = GBS \frac{RCP_i}{\sum_{j=1}^{n} RCP_j} \quad (5)$$

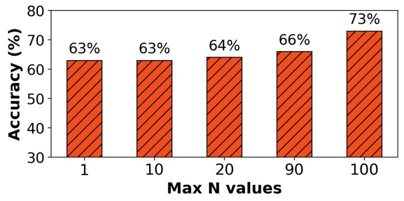


**Figure 7: Accuracy of Max N integrated with *DLion* with different N values; model accuracy trained until being fully converged in CPU cluster on homogeneous system environment**

Figure 6 shows the local batch size changes for 6 workers, as computed by GBS and LBS controllers in a heterogeneous computation environment where the 6 workers have heterogeneous CPU cores (24/24/12/12/4/4). As GBS is incremented by the GBS controller, the LBS for each worker is automatically adjusted by the LBS controller based on its available computation power.

**Weighted model update module.** Each worker $j$ computes its gradients over its LBS at each iteration $t$ as follows:

$$g_t^j = \frac{1}{LBS_j} \sum_{i=1}^{LBS_j} \nabla_w f_i(x; w_t) \tag{6}$$

As Eq. 6 shows, the gradient calculation of individual workers is based on different sample sizes ($LBS_j$). The sample size can have an impact on the final weight computation, since larger sample sizes typically provide more statistically robust mean values and a smaller margin of error, while smaller sample sizes could skew the mean values towards outliers. To account for the different sample sizes, we introduce a new confidence coefficient $db_j^k$ called dynamic batching weight and a new weighted model update equation:

$$w_{t+1}^k = w_t^k - \eta \frac{1}{n} \sum_{j=1}^{n} db_j^k g_t^j \tag{7}$$

Each worker $k$ uses Eq. 7 to update its weights based on gradients received from other workers. $db_j^k = LBS_j/LBS_k$, a ratio of LBS of workers $j$ and $k$, and compensates for the relative LBS of each worker. For example, when worker $k$ receives gradients from worker $j$ where $LBS_j > LBS_k$, it applies a dynamic batching weight greater than 1 ($db_j^k > 1$) to worker $j$'s gradients ($g_t^j$). If $LBS_j < LBS_k$, then the worker $k$ applies a dynamic batching weight less than 1 ($db_j^k < 1$) to worker $j$'s gradients ($g_t^j$). If all workers have a fixed LBS like a traditional DL learning, then the dynamic batching weight is equal to 1 ($db_j^k = 1$), which makes the new weighted model update equation (Eq. 7) equivalent to the original distributed DL model update equation (Eq. 4).

## 3.3 Per-Link Prioritized Gradient Exchange

The goal of network capacity-aware techniques is to speed up training and retain model accuracy by reducing communication overhead and guarantee model convergence based on available network bandwidth. Network bandwidth is an expensive resource in DL training. When DL models are huge or computation units are very powerful like GPUs, the network resource becomes more expensive since the size of data and data generation speed increase.

We propose a **per-link prioritized gradient exchange** technique to exchange gradients between workers in the presence of network constraints. There is a tradeoff between reducing network transmission cost and maintaining model accuracy. If a system considers only network bandwidth constraints, it could shorten training time by reducing the amount of data exchange, but could have a high impact on model accuracy. On the other hand, if a system takes into account only the data quality when exchanging gradients, it may have poor performance due to network congestion in environments with scarce network capacity. Therefore, *DLion* uses two complementary modules to balance out these two factors in exchanging gradients: a **data quality assurance module** to select a subset of gradients based on the importance of gradient values, and a **transmission speed assurance module** to dynamically determine the size of the partial gradients based on the available network bandwidth at that moment.

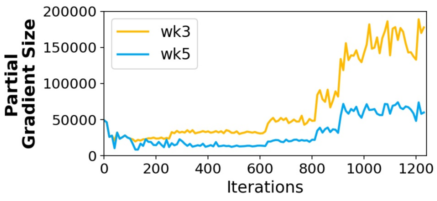
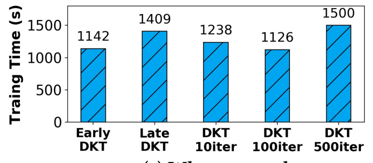


**Figure 8: Different gradient size with different network bandwidth per communication link (worker1 → worker3 and worker1 → worker5). There is no dynamism on network bandwidth per link for this experiment**

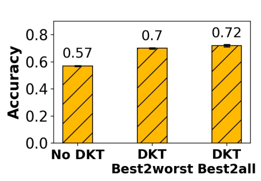
**Data quality assurance module.** This module selects a subset of important gradients to exchange with other workers, to minimize the impact on model accuracy. It uses a simple but powerful algorithm called **Max N** to identify the most statistically significant gradient values. The Max N algorithm selects those partial gradients whose absolute values are greater than or equal to N% of the maximum absolute value. Each weight variable has their own value distribution and convergence speed, so Max N is applied per weight variable. The mechanism and purpose of Max N is similar to the significance threshold S used in `Gaia` [18], except that Max N only considers gradient values to determine their significance. As N increases, the size of partial gradients increases. If N is 1, gradient values within 1% of max are exchanged with workers, while if N is 100, it is equivalent to exchanging whole gradients with workers. We next discuss how the value of N is determined automatically.

**Transmission speed assurance module.** Network heterogeneity and dynamism motivates the transmission speed assurance module because different links may have different bandwidth and the bandwidths are dynamically fluctuating over time. As the network capacity for a worker can change over time, the parameter N ($0 < N \leq 100$) of Max N is dynamically determined by the transmission speed assurance module based on the currently available network bandwidth.

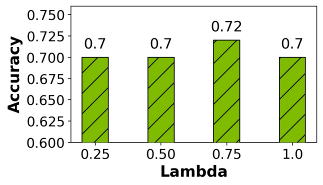
Figure 7 shows the model accuracy with different N values of Max N integrated with *DLion*. As seen from the figure, larger N values lead to higher accuracy. With this finding, the key role of the transmission speed assurance module is to automatically find the largest N value for each communication link based on per-link network capacity at each iteration. The maximum size of partial gradients that worker $i$ sends to worker $j$ is computed as $BW\_net_j/Iter\_com_i$,



(a) When-to-send



(b) Whom-to-send



(c) How-to-merge

**Figure 9: Emperical study on the effectiveness of direct knowledge transfer; Cipher models trained with CPUs until 70% accuracy for (a), for 1500 seconds for (b) and (c)**

where $BW\_net_j$ is the available network bandwidth of the communication link to worker $j$ and $Iter\_com_i$ is the number of iterations worker $i$ can proceed during a unit time. Figure 8 shows the per-link prioritized gradient exchange technique sends different size of partial gradients to two communication links having different network bandwidths.

## 3.4 Direct Knowledge Transfer

The techniques discussed above mainly focus on the reduction of training time with minimal impact on accuracy. To further compensate for potential accuracy loss due to these techniques, *DLion* employs a technique of **direct knowledge transfer** through periodic weight exchange [41, 42] between workers. Unlike gradient exchange, weight exchange shares model weights of the best worker having the smallest loss value at that moment, to directly transfer the knowledge accumulated on local model to other workers. All workers periodically share an average of last $l$ losses with each other, and send a request to the best worker to pull the best weights. However, the effectiveness of direct knowledge transfer (DKT) depends on several decisions: **when-to-send**: when to exchange the best weights, **whom-to-send**: whether to exchange the best weights with all workers or a subset of workers, and **how-to-merge**: how to merge new weights to the local model. We explore them empirically.

Figure 9 shows exploratory results regarding the factors of direct knowledge transfer technique. Figure 9a shows that periodic weight exchange with a moderate period (every 100 iterations) has the shortest training time. If the frequency of weight exchange is too short, it consumes a large amount of network resource, so it has rather longer training time. If the frequency is too long, it takes longer training time since it does not take advantage of the use of DKT. Interestingly, frequent weight exchange at early learning phase has a comparable performance with the best one from which we can infer it is important to share knowledge earlier rather than late in learning.

Figure 9b shows three different variants of whom-to-send direct knowledge transfer. We compare a model not using DKT (`No_DKT`) with two other variants (`DKT_Best2worst` and `DKT_Best2all`). The figure shows that transferring the best knowledge to all workers leads to the best accuracy. The benefit of model synchronization across all workers compensates the cost of network resources used by sending model weights to all.

We also explore how to effectively merge the best knowledge to local model. A recent work [41] has introduced a parameter $\lambda$ indicating the ratio of the best knowledge to local knowledge $w_{local} = w_{local} - \lambda(w_{local} - w_{best})$. If $\lambda = 0$, workers are not using DKT, which is the equivalent to `No_DKT` having the lowest accuracy as shown in Figure 9b. If $\lambda = 1$, the best weights are replaced with local weights, which leads to the fastest training progress at the early training phase, but does not have the best result at the end. The best option for the direct knowledge transfer technique may vary depending on individual application.

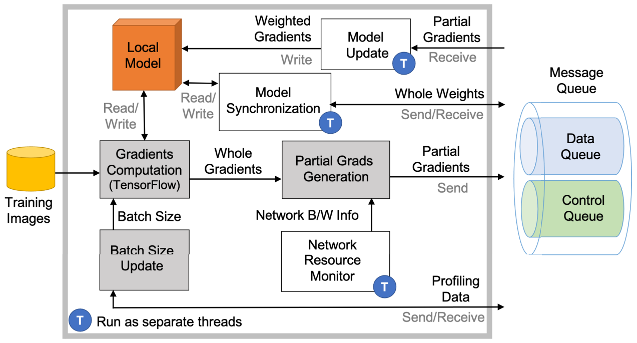


**Figure 10: Architecture of a *DLion* worker**

## 4 IMPLEMENTATION

*DLion* is built on top of TensorFlow and uses Redis for data and control messages queues. We describe key components and operations of a *DLion* worker in Section 4.1 and implementation details in Section 4.2.

## 4.1 Key Components and Operations

Key components of a *DLion* worker are presented in Figure 10. The main training workflow (colored in grey) is to compute gradients, generate/send partial gradients, and periodically update batch size. Three other independent modules run in parallel in separate threads. Model update module applies gradients of other workers to the local model whenever it receives them. Model synchronization module periodically gets the best model weights and merge them to the local model. Network resource monitor returns available network bandwidths of individual connections to neighbor workers upon the request by the partial gradient generation module. Next we present the operational details and workflow of these modules.

**Batch size update module.** Before starting model training, local batch size (LBS) controller (§ 3.2) is invoked to profile the compute capacity of workers at that moment. As a result of profiling, this module determines LBS of a worker used for the next update. The controller is invoked periodically and whenever global batch size (GBS) is changed by GBS controller (§ 3.2).

**Gradients computation module.** Given the LBS, this module calculates gradients, passes the gradients to the partial gradients generation module, and updates local model every iteration. If it is configured with synchronous strategy, it pauses proceeding next iteration until getting a go-signal through control queue.

**Table 1: Lines of code changes to emulate systems in *DLion***

| APIs | Baseline | Hop | Gaia | Ako |
|---|---|---|---|---|
| **generate_partial_gradients** | 1 | 1 | 1 | 23 |
| **synch_training** | 0 | 20 | 0 | 0 |

**Partial gradients generation module.** Upon a call to this module with newly computed local gradients, it requests the current available network bandwidth from the network resource monitor and generates partial gradients for individual neighbor workers based on per-link prioritized gradient exchange technique (§ 3.3). The generated partial gradients are sent to each worker.
**Model update module.** Upon the arrival of partial gradients of a neighbor worker, it executes weighted model update (§ 3.2) by applying a dynamic batching weight to the partial gradients and aggregates them to the local model.
**Model synchronization module.** Each worker periodically shares its average of the last $l$ loss values with workers, sends a direct knowledge transfer (DKT) request to the best worker having the smallest loss (§ 3.4). Upon the receipt of DKT request, the best worker sends its model weights to all workers sent the DKT request. Once this module receives the best weights, it merges them to the local model. The frequency and $\lambda$ of DKT are configurable.

## 4.2 Generic and Flexible *DLion*

We implement *DLion* on top of TensorFlow. Gradient calculation and model update are performed by TensorFlow core, and other modules are written in Python using TensorFlow APIs. Redis PUB/SUB and Redis Lists, an in-memory data store, are used for data exchange in *DLion*. There are two different message queues, control and data queues. Control queue is used for signaling among workers for training synchronization as well as signaling among threads in a worker, and data queue is used to exchange gradients and weights among workers. Since *DLion* modules and communication queues are implemented separately from DL core, *DLion* can be easily integrated with other cores like MXNet.

We have implemented *DLion* as a generic and flexible framework. Modules are configurable and easy to plugin different algorithms. With an API `build_model`, various DNN models can be defined and trained in *DLion*. We use two different models, Cipher and MobileNet, by simply calling the API with different model name for our evaluation. With an API `enqueue`, DL core can send local gradients to other workers. Internally, the `enqueue` calls `generate_partial_gradients` and `send_data` APIs. Users can easily implement their own algorithms to generate partial gradients in the API `generate_partial_gradients`. In the API `send_data`, the generated partial gradients are divided into indices and data and separately sent to workers with unique keys through data queue. The granularity of data transmission is not the whole weight variables, but individual weight variables. Lastly, *DLion* has an API `synch_training` where various configurable synchronization mechanisms are implemented including synchronous, asynchronous, and bounded synchronous training strategies. It internally maintains each worker's current iteration and received weight variable ids. Based on the information, it can skip or proceed to the next training iteration as well as identify straggler workers.

We have implemented four state-of-the-art distributed DL systems in the *DLion* framework for comparison: `Baseline` (send all gradients to all workers), `Hop` [31], `Gaia` [18] and `Ako` [46].

**Table 2: Actual network bandwidth between 6 Amazon regions**

| (Mbps) | V | O | I | M | S1 | S2 |
|---|---|---|---|---|---|---|
| **Virginia (V)** | - | 190 | 181 | 53 | 58 | 56 |
| **Oregon (O)** | 187 | - | 91 | 41 | 93 | 84 |
| **Ireland (I)** | 171 | 92 | - | 73 | 30 | 41 |
| **Mumbai (M)** | 53 | 41 | 73 | - | 85 | 79 |
| **Seoul (S1)** | 58 | 88 | 40 | 85 | - | 79 |
| **Sydney (S2)** | 56 | 84 | 36 | 79 | 72 | - |

Table 1 shows the lines of code changes for their partial gradient selection algorithms and synchronization mechanisms using the APIs. The small code changes show how easily we have implemented these other systems in our framework, illustrating its generality.

## 5 EVALUATION

We evaluate *DLion* by comparing it with four state-of-the-art decentralized training systems on various heterogeneous environments.

## 5.1 Methodology

*5.1.1* ***Applications and Datasets.*** We evaluate *DLion* on two deep learning tasks for image classification; Cipher model over CIFAR10 [26] and MobileNet, a well-known DNN model over ImageNet [11]. CIFAR10 is 28x28 gray-scale handwriting digits, which contains 60K training images and 10K testing images, and each image is labeled as one of 10 classes. ImageNet consists of 1.2M training images and 50K testing images, and each image is labeled as one of 1000 classes. We pre-processed it to 256x256 RGB images and randomly selected 100 classes to obtain faster convergence time for experiments due to monetary cost incurred by using Amazon GPU instances. Cipher model consists of 3 convolutional and 2 fully-connected layers with ReLU and Maxpooling applied. We use 10, 20, 100 kernels and 200 neurons like `Ako` [46]. The size of Cipher model is 5MB. MobileNet consists of 28 layers and its model size is 17MB.

*5.1.2* ***Experimental Platforms.*** We use two different platforms to evaluate *DLion*.
**CPU-based emulated micro-clouds.** The CPU cluster is composed of 6 machines in our local cluster. Each machine is equipped with 24 CPU cores, 60GB memory, and 1Gbps network bandwidth and runs on 64-bit Ubuntu 16.04 with TensorFlow 1.14, and Redis-server 5.0.10. Linux command `stress` and `tc` are used to emulate heterogeneous compute and network capacity environments, respectively. We emulate network bandwidth for micro-clouds by using actual measurement in 6 different Amazon regions shown in Table 2. Cipher model is trained with CIFAR10 on CPU cluster.
**GPU-based emulated micro-clouds.** The GPU cluster is composed of 6 GPU instances in Amazon. We use two different types of GPU instances, p2.xlarge and p2.8xlarge. Instance p2.xlarge is equipped with 1 GPU, 4 vCPUs, 61GB RAM and 1Gbps network bandwidth, and instance p2.8xlarge is equipped with 8 GPUs, 32 vCPUs, 488GB RAM, and 1Gbps network bandwidth. We emulate network capacity using command `tc` due to the high monetary cost of training model in multiple regions. We train MobileNet with ImageNet on the GPU cluster.

*5.1.3* ***Performance Metrics.*** We use three performance metrics to evaluate the effectiveness of distributed DL training systems.
• Model accuracy for a given training time in order to show how fast systems train models for a given time.

**Table 3: Emulation details for micro-cloud environments (* stands for environments used in AWS GPU cluster)**

| Environments | Computation (No.CPU cores or AWS instance types*) | Network (Mbps) |
|---|---|---|
| Homo A | No emulation | LAN |
| Homo B | No emulation | 50/50/50/50/50/50 |
| Homo C* | 6×p2.xlarge | LAN |
| Hetero CPU A | 24/24/12/12/6/6 | LAN |
| Hetero CPU B | 24/24/24/24/24/4 | LAN |
| Hetero NET A | No emulation | 50/50/35/35/20/20 |
| Hetero SYS A | 24/24/12/12/6/6 | 50/50/35/35/20/20 |
| Hetero SYS B | 24/24/12/12/6/6 | 20/20/35/35/50/50 |
| Hetero SYS C* | 2×p2.8xlarge + 4×p2.xlarge | 190/190/140/140/100/100 |
| Dynamic SYS A | Homo B→Hetero SYS A→Hetero SYS B | |
| Dynamic SYS B | Hetero SYS B→Hetero SYS A→Homo B | |

• Training (execution) time until a target model accuracy is reached, with model accuracy being measured every 20 iterations during training.
• Model accuracy trained until model is fully converged in order to show the highest accuracy a model can obtain given infinite training time.

The first two metrics measure the training speed, while the last metric measures the best model accuracy achievable.

#### 5.1.4 *Comparison Systems.*
We evaluate the effectiveness of *DLion* by comparing it with four state-of-the-art decentralized training systems implemented in our *DLion* framework (see Section 4): (1) `Baseline`, exchanging whole gradients with all workers every iteration, (2) `Ako` [46], partitioning gradients based on available network capacity and computation power and sending a block of the partitioned gradients in turn, (3) `Gaia` [18], exchanging only a subset of gradients causing more than S% change on model weights, and (4) `Hop` [31], exchanging whole gradients but advancing iterations by not receiving gradients of stragglers called backup workers. We set the threshold S to 1% for `Gaia`, and the number of backup worker to 1 and a staleness bound to 5 for `Hop` like their evaluation settings. We set the minimum N for max N algorithm to 0.85, the period of direct knowledge transfer to 100 iterations and $\lambda = 0.75$ for *DLion*. Most numbers shown in figures are the average of three runs and error bars mark 95% confidence interval.

#### 5.1.5 *Experimental Setup.*
The evaluation is performed in different environments with various combinations of homogeneous/heterogeneous and computation/network capacities, as well as dynamically changing resources over time. Table 3 shows the details of all the emulated micro-cloud environments. The unit for computation and network is the number of CPU cores and Mbps, respectively. Homo C and Hetero SYS C are used for experiments in the Amazon GPU cluster and the others are used in the CPU cluster. Homo A and Homo C are homogeneous, best-case environments with no emulation and LAN, and are used as baseline for comparison with other environments. For Dynamic SYS A and B, each environment is applied for 500 seconds in the given order.

## 5.2 Evaluation Results

We now present the results of our evaluation for different environments. We begin by evaluating *DLion* in heterogeneous system (both CPU/GPU and network) environments (Sections 5.2.1 and 5.2.2). We then evaluate *DLion* in the presence of CPU heterogeneity, network heterogeneity, and dynamism, in order to gain a better understanding of the benefit of the various *DLion* techniques.

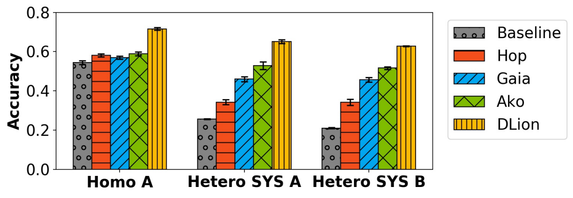


**Figure 11: Handling homogeneous and heterogeneous system (compute + network) environments in CPU cluster**

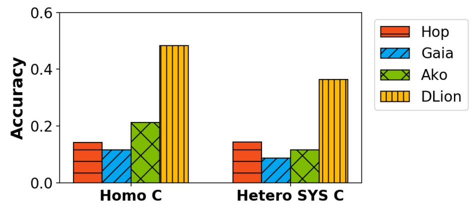


**Figure 12: Handling homogeneous and heterogeneous system (compute + network) environments in GPU cluster; model accuracy of MobileNet trained for 2 hours**

#### 5.2.1 *System Heterogeneity.*
We present the performance of *DLion* in heterogeneous systems where both computation and network capacities are heterogeneous. In Hetero SYS A, workers with more computation power have more available network bandwidth, whereas in Hetero SYS B, workers with more computation power have less network bandwidth. We train Cipher model for 1500 seconds on CPU cluster those three environments.

Figure 11 shows the model accuracy achieved by each model for the given training time (higher is better). *DLion* outperforms the state-of-the-art decentralized deep learning systems both in homogeneous and heterogeneous system environments. Specifically, accuracy improvement of *DLion* in Hetero SYS A and Hetero SYS B respectively is 155% and 199% over `Baseline`, 90% and 84% over `Hop`, 42% and 38% over `Gaia`, and 23% and 22% over `Ako`. *DLion* performs much better in heterogeneous systems because it takes into account not only the available computation power for better data parallelism, but also available network bandwidth to reduce communication cost while retaining high accuracy. Interestingly, we see that *DLion* outperforms all other systems even for the homogeneous full CPU/network capacity environment Homo A by 32% over `Baseline`, 23% over `Hop`, and 26% over `Gaia`, and 22% over `Ako`. This is because *DLion* utilizes the direct knowledge transfer, a supplementary technique to increase the maximum accuracy.

#### 5.2.2 *System Robustness in Heterogeneous GPU cluster.*
To evaluate the robustness of *DLion*, we train a much bigger MobileNet model over ImageNet, a much bigger dataset, on the Amazon GPU cluster for 2 hours in homogeneous (Homo C) and heterogeneous (Hetero SYS C) environments. A special characteristic of the environments is that the gap between required network capacity and available computation power is huge. In other words, GPU-based powerful computation capacity and larger model size of MobileNet generates a huge amount of data to exchange between workers,

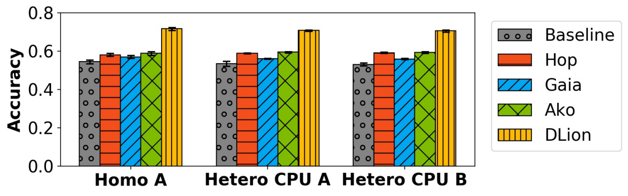


**Figure 13: Handling homogeneous and heterogeneous compute resource environments**

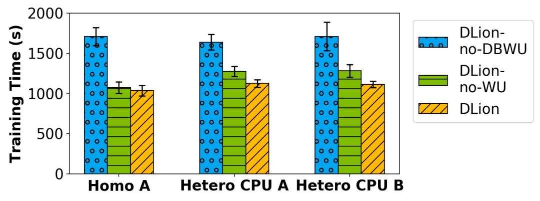


**Figure 14: Effect of dynamic batching (DB) based on GBS & LBS controllers and weighted model updates (WU) in heterogeneous compute capacity environments**

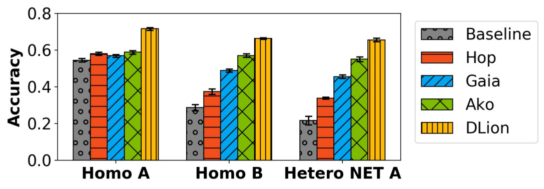


**Figure 15: Handling homogeneous and heterogeneous network resource environments**

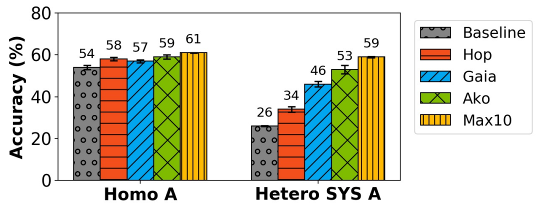


**Figure 16: Max10 algorithm comparison with existing systems on both homogeneous and heterogeneous system environments**

which leads to a severe network bottleneck issue. Although Homo C uses LANs, it suffers from the same network bottleneck issue. In Hetero SYS C using WANs, the network bottleneck issue becomes more serious.

Figure 12 shows that our design is robust for both GPU-based and CPU-based DL systems. *DLion* achieves much higher model accuracy both in homogeneous and heterogeneous system environments. Its accuracy improvement in Homo C is 3.4× over `Hop`, 4.2× over `Gaia`, and 2.3× over `Ako`, and the improvement in Hetero SYS is 2.5× over `Hop`, 4.2× over `Gaia`, and 3.1× over `Ako`. Direct knowledge transfer technique in *DLion* plays a key role for faster model convergence in such environments since it is a more direct way to share learned knowledge among workers.

### 5.2.3 *Heterogeneous Compute Resources.*

To better understand the benefits of *DLion*'s CPU-aware techniques, we now evaluate the performance of *DLion* on heterogeneous compute capacity environments while keeping the network capacity of workers homogeneous. Figure 13 shows accuracy of Cipher model trained for 1500 seconds on the CPU cluster for three different compute resource environments: Homo A, Hetero CPU A, and Hetero CPU B. As mentioned before, Homo A is the best-case homogeneous environment. In Hetero CPU A, different computation capacities are evenly distributed across workers, whereas Hetero CPU B has a distinct straggler in a cluster. The average of accuracy improvement of *DLion* is 32% over `Baseline`, 21% over `Hop`, 26% over `Gaia`, and 20% over `Ako`.

The first finding from the figure is that *DLion* outperforms other systems both in homogeneous and heterogeneous computation environments. This shows the benefit of the weighted dynamic batching technique which results in better load balancing and parallelism across workers. Second, the difference in the amount of available computation resource—the total number of CPU cores used for Home A, Hetero CPU A, and Hetero CPU B are 144, 88, and 114, respectively—is not reflected on the accuracy, with each system having almost the same accuracy in all three environments. This indicates that system performance is bounded more by network capacity than compute capacity.

To further understand the performance gain of the weighted dynamic batching technique of *DLion*, we measure training time until Cipher model reaches 70% accuracy with three variants of *DLion*: (i) `DLion-no-DBWU`, which does not have dynamic batching based on GBS and LBS controllers nor weighted model update, (ii) `DLion-no-WU`, which has dynamic batching but does not apply weighted model update, and (iii) *DLion* with both dynamic batching and weighted model update enabled. All other features and techniques are the same across the three.

Figure 14 (lower is better) shows a noticeable performance gain by dynamic batching technique: 37%, 22%, and 25% training speed-up in Homo A, Hetero CPU A, and Hetero CPU B environments, respectively. The effect of weighted model update is clearly visible in heterogeneous compute environments: a further 12% and 13% training speed-up in Hetero CPU A and Hetero CPU B. There is no statistically significant improvement due to weighted model update in Homo A, since the weighted model update equation (Eq. 7) reduces to the general distributed DL model update equation (Eq. 4) in homogeneous compute environments. Therefore, we conclude that the combination of dynamic batching with weighted model update achieves the best results both in homogeneous and heterogeneous compute resource environments.

### 5.2.4 *Heterogeneous Network Resources.*

We next evaluate the performance of *DLion* on heterogeneous and constrained network capacity environments while compute capacity of workers remains homogeneous and identical. We train Cipher model for 1500 seconds on the CPU cluster with three different homogeneous and heterogeneous network environments. Homo A uses LANs while both Homo B and Hetero NET A are WAN environments. Homo B has homogeneous network capacity across workers though it is constrained, while the workers in Hetero NET A have different network bandwidths.

First, Figure 15 shows that *DLion* outperforms all other systems in all three cases, and especially performs much better in heterogeneous network environment. Accuracy improvement of *DLion* in Homo B and Hetero NET A respectively is 132% and 202% over `Baseline`, 78% and 94% for `Hop`, 36% and 44% over `Gaia`, and

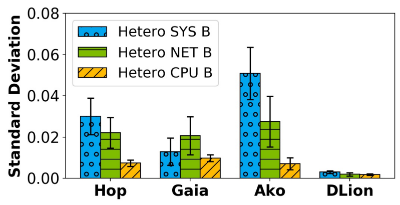


**Figure 17: The deviation of model accuracy among workers in various heterogeneous environments**

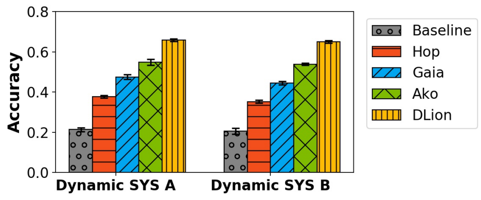


**Figure 18: The highest accuracy of cipher model trained with different systems in dynamic heterogeneous environments where resources are dynamically changing over time**

16% and 19% over `Ako`. This shows the benefit of the network-aware gradient and weight exchange techniques in *DLion*.

Second, we also see that system performance depends more on the available network bandwidth, with the achieved accuracy in LANs being much higher than the ones in WANs. This is because distributed DL training in general spends more time for communication than computation. This is why systems sharing whole gradients like `Hop` and `Baseline` have greater accuracy difference between LANs and WANs. Instead, the approach of exchanging small gradients like `Ako`, `Gaia`, and *DLion* is more effective than skipping slower workers like `Hop` in the environments of Homo B and Hetero NET A like WANs.

In addition, to understand the sole benefit of max N algorithm, we train Cipher model in the CPU cluster for 1500s without any support from the other *DLion* techniques such as weighted dynamic batching, per-link prioritized gradient exchange, and direct knowledge transfer. Figure 16 shows max N (N=10) outperforms state-of-the-art distributed DL systems in both homogeneous and heterogeneous system environments.

#### 5.2.5 *Deviation of Model Accuracy.*

One of the strengths of *DLion* is its significantly small deviation of accuracy across workers. Figure 17 shows the standard deviation of accuracy among workers based on CPU-based experiments in three different heterogeneous environments; Hetero SYS B, Hetero NET B, and Hetero CPU B. *DLion* has much smaller accuracy deviation than other systems because direct knowledge transfer technique periodically synchronizes model weights across workers. `Ako` has the biggest accuracy deviation among workers due to its asynchronous training strategy where some workers advance training iterations without consideration of slower workers' progress. Similarly, `Hop` has second highest accuracy deviation because it uses bounded synchronous training strategy and backup worker technique ignoring updates from stragglers. Lastly, `Gaia` is less sensitive than `Ako` and `Hop` because it uses a kind of bounded synchronous training strategy by blocking progress to the next iteration until important gradients are delivered to all workers.

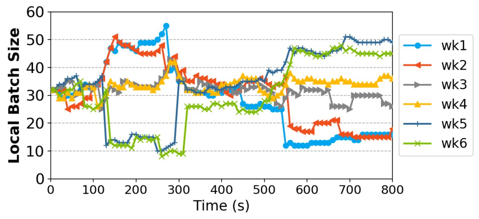


**Figure 19: Dynamically adjusting local batch sizes of 6 workers when available computation resources are changing**

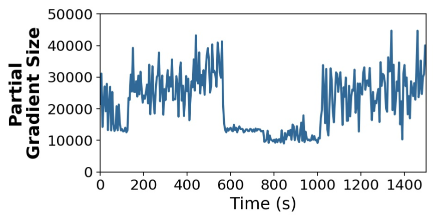


**Figure 20: Change of partial gradient size (the number of gradients) adjusted by Per-link prioritized gradient exchange technique in consideration of dynamically changing network bandwidth**

#### 5.2.6 *Dynamic Resource Changes.*

We evaluate how effectively *DLion* handles dynamically changing resources compared to existing systems. As shown in Table 3, the combination of sub-environments are the same between Dynamic SYS A and Dynamic SYS B. However, Dynamic SYS A has more resource at early training phase, whereas Dynamic SYS B has more capacity at later training phase. Figure 18 shows *DLion* outperforms other systems in both environments. Accuracy improvement of *DLion* in Dynamic SYS A and Dynamic SYS B respectively is 209% and 216% over `Baseline`, 75% and 85% over `Hop`, 38% and 46% over `Gaia`, and 20% and 21% over `Ako`. This shows that *DLion* is able to handle dynamic resource changes more effectively than the other systems.

Figure 19 shows how the LBS controller dynamically changes LBS of 6 workers in a dynamic compute resource environment. Here LBS is initialized to 32 and GBS is fixed to 192 ($32 \times 6$). The available number of CPU cores of workers changes: it is homogeneous (24/24/24/24/24/24 for 0-100s and 12/12/12/12/12/12 for 300-500s periods) and heterogeneous (24/24/12/12/4/4 for 100-300s and 4/4/12/12/24/24 for 500-800s periods). The figure shows a different LBS is assigned to each worker based on their available computation power at that moment.

Figure 20 shows the per-link prioritized gradient exchange technique handles dynamism of network bandwidth by changing partial gradient size based on changing available network bandwidth where available network bandwidths are set to 30 Mbps for 0-100s and 600-1000s periods and to 100Mbps for the remaining periods.

#### 5.2.7 *Effect on Improving Model Accuracy.*

To understand how well *DLion* accomplishes its accuracy improvement goal, we train the Cipher model in Homo A until it is fully converged. Figure 21 shows the final model accuracy and required training time to reach the accuracy of the systems. *DLion* reaches the highest model accuracy among other systems because the direct knowledge transfer technique directly propagates the best training knowledge to all workers. *DLion* obtains 26% and 24% higher accuracy with 59% and 36% faster training time over `Baseline` and `Hop`, respectively,

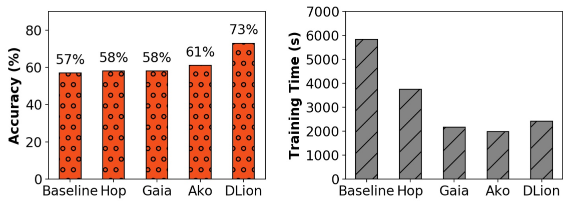


**Figure 21: The highest model accuracy and elapsed training time until Cipher model is fully converged**

and 25% and 18% higher accuracy with 11% and 21% slower training time over `Gaia` and `Ako`, respectively. Even though `Ako`, `Gaia` and `Hop` exchange partial gradients or skip gradients from stragglers, they have higher accuracy than `Baseline` sharing whole gradients. We analyze the result that more errors are accumulated on local model by sharing larger gradients since all those systems do not synchronize model across workers unlike *DLion* using direct knowledge transfer.

## 6 RELATED WORK

**Distributed Deep Learning Systems.** General purpose distributed DL systems [1, 8] use parameter servers (PS) to provide scalability. These systems have implicit assumption that machines are identical and connected with high-speed network bandwidth. In addition, recent research [16, 39, 51] has done great deal of work on performance improvement on multi-GPU environments. However, they also target homogeneous environments. Besides, the performance can significantly vary depending on cluster configuration related to PSs. In contrast, our work employs decentralized design and focuses on handling heterogeneous resources.

**Resource-aware Distributed DL Systems.** There have been many distributed DL systems considering heterogeneous resources. AD-PSGD [29] and HetPipe [34] have addressed heterogeneous computation environments by using a decentralized architecture based on asynchronous parallel SGD. However, they mainly focus on GPU clusters. Our work covers both CPU-based and GPU-based environments because micro-clouds are composed of commodity machines with CPUs/GPUs, connected over WANs/LANs.

Ako [46], Gaia [18], Hop [31], and Prague [30] consider distributed DL issues in heterogeneous system environments on CPU and/or GPU cluster. While Gaia and Ako address the network bottleneck issue by exchanging partial gradients, Hop and Prague reduce the number of workers sending/receiving whole gradients by excluding slow workers from training. Some research [3, 43] specifically focuses on gradient compression problem, which is complementary to our work, and their compression algorithms can be placed in the data quality assurance module in *DLion*. None of these approaches comprehensively consider all the challenges in micro-cloud environments, as done by our work.

**Federated Learning.** Recently, federated learning [5, 7, 28] has emerged as a new paradigm for distributed training on edges over locally generated data driven by privacy concerns. While it focuses on system and data heterogeneity to effectively select subset of edge devices for training purposes, federated learning can only train much smaller-scale models like traditional machine learning algorithms due to limited resources of the edges. Our work, on the other hand, is targeting DL training in micro-clouds.

**Deep Learning Inference at Edges.** One of the main streams in DL research is fast prediction during inference. Recent research [2, 21, 45] has proposed algorithms to compact pre-trained models by reducing precision or pruning model weights to speed up inference in edge or micro-cloud environments. Our work, on the other hand, is focused on the orthogonal stream of DL *training*.

**General Parallel Computing.** Many studies [9, 32, 48] take into account resources of machines in cluster for better scheduling to improve performance and resource efficiency. Our work also considers computation resource to enable more powerful machines to process more data for better performance, but is specific to DL, which is a different application than considered in much of the prior work.

**Geo-Distributed Data Analytics.** Data analytics with geographically distributed data has gained much attention in recent years. Since a large amount of data are shuffled and processed over networks, previous research [17, 19, 22, 23, 27] has proposed techniques such as minimizing the amount of intermediate data, merging multiple queries, or placing queries based on the available network bandwidth. Similarly, a key factor affecting system performance in our work is network bandwidth because DL training causes a huge data transmission among machines over the network, but reducing the impact on overall model accuracy is a key goal of our work.

## 7 CONCLUSION

In this paper, we proposed *DLion*, a new and generic decentralized distributed deep learning system for fast learning and high accuracy in micro-clouds. *DLion* is designed to handle heterogeneous and dynamically changing compute and network resources in such environments, to reduce training time and improve model accuracy. We presented three key techniques in *DLion*: (1) *Weighted dynamic batching* to maximize data parallelism, (2) *Per-link prioritized gradient exchange* to reduce communication overhead, and (3) *Direct knowledge transfer* to improve model accuracy. We built a prototype of *DLion* on top of TensorFlow and showed that *DLion* achieves up to 4.2× speedup in an Amazon GPU cluster, and up to 2× speed up and 26% higher model accuracy in a CPU cluster over four state-of-the-art distributed DL systems.

## ACKNOWLEDGMENTS

This work is supported in part by NSF grant CNS-1717834.

## REFERENCES

[1] M. Abadi et al. 2016. Tensorflow: a system for large-scale machine learning.. In *OSDI*, Vol. 16. 265–283.
[2] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 972–986.
[3] Dan Alistarh, Torsten Hoefler, Mikael Johansson, Nikola Konstantinov, Sarit Khirirat, and Cédric Renggli. 2018. The convergence of sparsified gradient methods. In *Advances in Neural Information Processing Systems*. 5973–5983.
[4] Kashif Bilal, Osman Khalid, Aiman Erbad, and Samee U Khan. 2018. Potentials, trends, and prospects in edge technologies: Fog, cloudlet, mobile edge, and micro data centers. *Computer Networks* 130 (2018), 94–120.
[5] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečnỳ, Stefano Mazzocchi,

H Brendan McMahan, et al. 2019. Towards federated learning at scale: System design. *arXiv preprint arXiv:1902.01046* (2019).

[6] Francisco M Castro, Manuel J Marín-Jiménez, Nicolás Guil, Cordelia Schmid, and Karteek Alahari. 2018. End-to-end incremental learning. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 233–248.

[7] Zheng Chai, Ahsan Ali, Syed Zawad, Stacey Truex, Ali Anwar, Nathalie Baracaldo, Yi Zhou, Heiko Ludwig, Feng Yan, and Yue Cheng. 2020. TiFL: A Tier-based Federated Learning System. *arXiv preprint arXiv:2001.09249* (2020).

[8] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[9] Dazhao Cheng, Yuan Chen, Xiaobo Zhou, Daniel Gmach, and Dejan Milojicic. 2017. Adaptive scheduling of parallel jobs in spark streaming. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 1–9.

[10] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555* (2014).

[11] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. (2009).

[12] Aditya Devarakonda, Maxim Naumov, and Michael Garland. 2017. Adabatch: Adaptive batch sizes for training deep neural networks. *arXiv preprint arXiv:1712.02029* (2017).

[13] Yehia Elkhatib, Barry Porter, Heverson B Ribeiro, Mohamed Faten Zhani, Junaid Qadir, and Etienne Rivière. 2017. On using micro-clouds to deliver the fog. *IEEE Internet Computing* 21, 2 (2017), 8–15.

[14] Nelson Mimura Gonzalez, Walter Akio Goya, Rosangela de Fatima Pereira, Karen Langona, Erico Augusto Silva, Tereza Cristina Melo de Brito Carvalho, Charles Christian Miers, Jan-Erik Mångs, and Azimeh Sefidcon. 2016. Fog computing: Data analytics and cloud distributed processing on the network edges. In *2016 35th International Conference of the Chilean Computer Science Society (SCCC)*. IEEE, 1–9.

[15] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Advances in neural information processing systems*. 2672–2680.

[16] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. 2018. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377* (2018).

[17] Benjamin Heintz, Abhishek Chandra, and Ramesh K Sitaraman. 2015. Optimizing grouped aggregation in geo-distributed streaming analytics. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 133–144.

[18] K. Hsieh et al. 2017. Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds.. In *NSDI*. 629–647.

[19] Anand Padmanabha Iyer, Aurojit Panda, Mosharaf Chowdhury, Aditya Akella, Scott Shenker, and Ion Stoica. 2018. Monarch: gaining command on geo-distributed graph analytics. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*.

[20] Samvit Jain, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, and Joseph Gonzalez. 2019. Scaling Video Analytics Systems to Large Camera Deployments. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*. ACM, 9–14.

[21] Hyuk-Jin Jeong, Hyeon-Jae Lee, Chang Hyun Shin, and Soo-Mook Moon. 2018. IONN: Incremental offloading of neural network computations from mobile devices to edge servers. In *Proceedings of the ACM Symposium on Cloud Computing*. 401–411.

[22] Rathinaraja Jeyaraj, VS Ananthanarayana, and Anand Paul. 2019. MapReduce scheduler to minimize the size of intermediate data in shuffle phase. In *2019 IEEE/ACIS 18th International Conference on Computer and Information Science (ICIS)*. IEEE, 30–34.

[23] Albert Jonathan, Abhishek Chandra, and Jon Weissman. 2018. Multi-Query Optimization in Wide-Area Streaming Analytics. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 412–425.

[24] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. 2014. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 1725–1732.

[25] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2016. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836* (2016).

[26] A. Krizhevsky and G. Hinton. 2009. *Learning multiple layers of features from tiny images.* Technical Report. Citeseer.

[27] Dhruv Kumar, Jian Li, Abhishek Chandra, and Ramesh Sitaraman. 2019. A ttl-based approach for data aggregation in geo-distributed streaming analytics. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 3, 2 (2019), 1–27.

[28] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. 2018. Federated optimization in heterogeneous networks. *arXiv preprint arXiv:1812.06127* (2018).

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

[30] Qinyi Luo, Jiaao He, Youwei Zhuo, and Xuehai Qian. 2020. Prague: High-Performance Heterogeneity-Aware Asynchronous Decentralized Training. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 401–416.

[31] Qinyi Luo, Jinkun Lin, Youwei Zhuo, and Xuehai Qian. 2019. Hop: Heterogeneity-aware decentralized training. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 893–907.

[32] Vicent Sanz Marco, Ben Taylor, Barry Porter, and Zheng Wang. 2017. Improving spark application throughput via memory aware task co-location: A mixture of experts approach. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. 95–108.

[33] Carlos Navarro-Racines, Jaime Tarapues, Philip Thornton, Andy Jarvis, and Julian Ramirez-Villegas. 2020. High-resolution and bias-corrected CMIP5 projections for climate change impact assessments. *Scientific Data* 7, 1 (2020), 1–14.

[34] Jay H Park, Gyeongchan Yun, Chang M Yi, Nguyen T Nguyen, Seungmin Lee, Jaesik Choi, Sam H Noh, and Young-ri Choi. 2020. HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism. *arXiv preprint arXiv:2005.14038* (2020).

[35] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H Lampert. 2017. icarl: Incremental classifier and representation learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2001–2010.

[36] Herbert Robbins and Sutton Monro. 1951. A stochastic approximation method. *The annals of mathematical statistics* (1951), 400–407.

[37] Doyen Sahoo, Quang Pham, Jing Lu, and Steven CH Hoi. 2017. Online deep learning: Learning deep neural networks on the fly. *arXiv preprint arXiv:1711.03705* (2017).

[38] Mahadev Satyanarayanan, Zhuo Chen, Kiryong Ha, Wenlu Hu, Wolfgang Richter, and Padmanabhan Pillai. 2014. Cloudlets: at the leading edge of mobile-cloud convergence. In *6th International Conference on Mobile Computing, Applications and Services*. IEEE, 1–9.

[39] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[40] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. 2017. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489* (2017).

[41] Yunfei Teng, Wenbo Gao, Francois Chalus, Anna E Choromanska, Donald Goldfarb, and Adrian Weller. 2019. Leader Stochastic Gradient Descent for Distributed Training of Deep Learning Models. In *Advances in Neural Information Processing Systems*. 9824–9834.

[42] Jianyu Wang and Gauri Joshi. 2018. Adaptive communication strategies to achieve the best error-runtime trade-off in local-update SGD. *arXiv preprint arXiv:1810.08313* (2018).

[43] Linnan Wang, Wei Wu, Junyu Zhang, Hang Liu, George Bosilca, Maurice Herlihy, and Rodrigo Fonseca. 2020. FFT-based Gradient Sparsification for the Distributed Training of Deep Neural Networks. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*. 113–124.

[44] Shiqiang Wang, Tiffany Tuor, Theodoros Salonidis, Kin K Leung, Christian Makaya, Ting He, and Kevin Chan. 2018. When edge meets learning: Adaptive control for resource-constrained distributed machine learning. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 63–71.

[45] Xiaofei Wang, Yiwen Han, Victor CM Leung, Dusit Niyato, Xueqiang Yan, and Xu Chen. 2020. Convergence of edge computing and deep learning: A comprehensive survey. *IEEE Communications Surveys & Tutorials* 22, 2 (2020), 869–904.

[46] P. Watcharapichat et al. 2016. Ako: Decentralised deep learning with partial gradient exchange. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, 84–97.

[47] Christopher R Wren, Yuri A Ivanov, Darren Leigh, and Jonathan Westhues. 2007. The MERL motion detector dataset. In *Proceedings of the 2007 workshop on Massive datasets*. 10–14.

[48] Luna Xu, Ali R Butt, Seung-Hwan Lim, and Ramakrishnan Kannan. 2018. A heterogeneity-aware task scheduler for spark. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 245–256.

[49] Jing Yuan, Yu Zheng, Xing Xie, and Guangzhong Sun. 2011. Driving with knowledge from the physical world. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. 316–324.

[50] Chaoyun Zhang, Paul Patras, and Hamed Haddadi. 2018. Deep learning in mobile and wireless networking: A survey. *arXiv preprint arXiv:1803.04311* (2018).

[51] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. 2017. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 USENIX Annual Technical Conference (ATC 17)*. 181–193.