DC2: Delay-aware Compression Control for Distributed Machine Learning

Ahmed M. Abdelmoniem, Marco Canini
KAUST

***Abstract*—Distributed training performs data-parallel training of DNN models which is a necessity for increasingly complex models and large datasets. Recent works are identifying major communication bottlenecks in distributed training. These works seek possible opportunities to speed-up the training in systems supporting distributed ML workloads. As communication reduction, compression techniques are proposed to speed up this communication phase. However, compression comes at the cost of reduced model accuracy, especially when compression is applied arbitrarily. Instead, we advocate a more controlled use of compression and propose DC2, a delay-aware compression control mechanism. DC2 couples compression control and network delays in applying compression adaptively. DC2 not only compensates for network variations but can also strike a better trade-off between training speed and accuracy. DC2 is implemented as a drop-in module to the communication library used by the ML toolkit and can operate in a variety of network settings. We empirically evaluate DC2 in network environments exhibiting low and high delay variations. Our evaluation of different popular CNN models and datasets shows that DC2 improves training speed-ups of up to 41× and 5.3× over baselines with no-compression and uniform compression, respectively.**

***Index Terms*—Machine Learning, Distributed Training, Delay-aware Control, Adaptive Gradient Compression**

I. Introduction

Machine learning (ML) is used as an integral part of many applications to create predictive models from large-scale data and aid in decision making. ML has shown unprecedented performance in applications such as topic modeling (e.g., [5]), speech recognition (e.g., [14]), and image or video classification (e.g., [20], [41]). In these applications, ML systems train increasingly larger models on massive datasets collected from user activities, photos, videos, text, etc. [8], [38], [39].

Many of the large data-mining organizations nowadays dedicate a large portion of their (geo)-distributed computing resources to deploy advanced ML systems. These systems store, process, analyze, and exploit the massive wealth of data collected on a global scale. However, scaling ML training jobs over distributed resources faces a fundamental challenge: easily, communication between distributed worker machines becomes a bottleneck that severely degrades the scalability and performance of distributed training [17], [34], [43]. This problem is exacerbated when communicating in shared environments (e.g., public cloud [26], [42]) or over the wide-area network (e.g., geo-distributed training [16] or federated learning [23]).

Popular ML toolkits like TensorFlow or PyTorch readily support distributed training wherein learning occurs over an array of worker nodes. In the data-parallel setting, the data are stored on many nodes (or partitioned among them) and the workers run in parallel, each one using its own replica of the model. However, this method requires that workers periodically communicate model updates among each other. In particular, under the synchronous replication mode (which is typically used in practice), after each iteration of the training algorithm, the workers need to synchronize the models by communicating and aggregating the gradient (or parameter updates) before proceeding with the next training iteration. The time taken to transfer gradients, especially for large models, introduces a delay that challenges the scalability of distributed training.

Gradient compression is commonly proposed to reduce this transfer time; however, the work in this area (see [46] for a recent survey) typically ignores several practical considerations concerning production jobs. In production settings, training jobs typically are associated with service-level objectives and/or practical resource constraints that impose a training time deadline; thus, the timely completion of these jobs is of paramount importance. When network delays are predictable, the job completion time could be estimated and feasible deadlines (or time limits) can be set. However, in environments where network delays are highly variable, the time spent in the communication phase during the training becomes unpredictable [25], [26], [42], [43]. The variability in communication time makes the planning of the training jobs within the imposed deadlines a hard problem.

To this end, in this paper, we seek to make distributed training systems more robust and resilient to variations in network conditions so they can finish their jobs in a predictable time [27] and consequently meet their time (or monetary) budgets. We propose DC2, a delay-aware compression control mechanism that: 1) mitigates the variance of communication times, so that dynamic network conditions do not over-extend the duration of training jobs; 2) is generally applicable to different types of communication network settings.

In designing DC2, we address the following challenges: C1) *How to adapt the communication to time-varying network conditions?* We strike a balance between mitigating network variations and maintaining the convergence proprieties of distributed training; C2) *How to handle the dynamics of different networks?* We adapt to available network bandwidth by relying on explicit or implicit signals from various metrics.

We make the following contributions:

1) We propose coupling of the network dynamics with some form of communication volume control to achieve better training speed-ups with respect to both time and accuracy.

2) We propose,*DC2* , a simple control system that adapts the communicated volume to the network delay variations.
3) We employ *DC2* in distributed training and evaluate it in both static and dynamic network conditions. The results show *DC2* can improve the training speed-up in highly variable network conditions by up to $\approx 41\times$ and $\approx 5.3\times$ over baselines with no-compression baseline and uniform compression, respectively.

II. Background

Our focus is on distributed training in the data-parallel mode via synchronous stochastic optimization algorithms such as stochastic gradient descent (SGD) or its variants (e.g., ADAM [22] or ADAGrad [9]).[1] We offer a brief description of this process and discuss its communication aspects.

Distributed training in a data-parallel mode: Consider a setup with n workers. Each worker holds a copy of the model parameters $p \in \mathbb{R}^d$ (i.e., the weights and biases) and has access to a disjoint partition D_i of the training dataset D. The training process can be seen as solving the following optimization:

$$\min_{p\in\mathbb{R}^d} \frac{1}{n}\sum_{i=1}^{n} f_i(p), \tag{1}$$

where $f_i(p)$ is the loss function for worker i.

An optimizer (e.g, SGD) proceeds iteratively to update the model parameters. At each iteration, every worker selects a so-called mini-batch of b training samples and performs the feed-forward and back-propagation passes of the DNN model. The latter pass calculates the stochastic gradient g_i of the loss function with respect to p. Then the gradients g_i are communicated across all workers and aggregated into the stochastic gradient g. This communication phase is typically supported via either a parameter server architecture or peer-to-peer collective operations (e.g., all-reduce or all-gather). Finally, the resulted gradient g_j at iteration j is used to update the model parameters. For example, at iteration $j+1$, SGD applies the following update rule: $p_{j+1} = p_j - \eta g_j$ (where η is the learning rate).

Influence of communication on training time: Gradient aggregation is potentially a performance-crucial step to total training time. Since workers wait for gradient aggregation to complete before proceeding to the next training iteration, when a large volume of data (10s-1000s of MBs) needs to be communicated over the network interconnecting the workers, the time taken for gradient aggregation slows down the training process. Popular distributed ML toolkits like TensorFlow and PyTorch in part mitigate this issue by partially overlapping computation with communication. This is enabled by the backward-propagation phase that incrementally computes, in a layer by layer fashion, the partial gradient [10]. Thus, communication can start as soon as the earliest gradient vector is computed (i.e., that of the last DNN layer, following the back-propagation order). However, as reported in many studies, due to the fast and ever increasing processing speeds of modern DNN accelerators (e.g., GPUs or TPUs), this communication phase remains predominantly the performance bottleneck, especially at large scale and for large DNN models [11], [18], [37], [42], [43].

Gradient compression: To accelerate training, several works have proposed various compression techniques that aim to reduce communication time by decreasing the transmitted data volume. Two broad classes of compressors exist. Quantization methods (e.g., [2], [3], [35], [44]) reduce the bit-width precision of gradient elements (e.g., use 8 bits instead of 32). Sparsification methods (e.g., [1], [25])) transmit only a subset of gradient elements, say $k\%$ out of d elements; k is known as the compression ratio.

Gradient compression is lossy operation and can adversely affect model accuracy. Although DNN training is a stochastic process, compression introduces an error in the aggregated gradient g. As the level of compression increases (e.g., choosing smaller k), so does the resulting error in the compressed gradient. While the literature on compression methods has established sound convergence results, a larger error implies that: (i) in theory, a larger number of iterations might be necessary to converge (i.e., finding p that minimizes the loss function); and (ii) in practice, the model accuracy might decrease. However, the relationship between the compression level and update error (and ultimately, model accuracy) is not straightforward since it depends on the actual data. Thus, when gradient compression is used in practice, the compression level remains an additional hyper-parameter to be carefully tuned with each training job.

III. The Case for Adaptive Compression

Training time is subject to communication delay and, as a result, is directly influenced by the performance variability of the underlying network. This means that network congestion events can significantly slowdown training. To make matters worse, because gradient aggregation acts as a synchronization barrier among workers, when even a single worker experiences congestion, gradient aggregation completes only once this network-impaired worker finishes. The adverse effects of network performance variations can be severe when workers span a shared network (as it is common in the public Cloud), where performance variability is the norm rather than the exception [26]. This scenario, instead of a dedicated network fabric, is the main focus of this paper.

A slowdown in training performance is undesirable: it increases monetary costs and can result in violations of service-level objectives (SLOs), which are costly for production DNN-training workloads. To minimize training costs, a ML job needs to maintain a high utilization of the DNN accelerators (e.g., GPUs, TPUs or FPGAs). However, network congestion events induce periods of time where DNN accelerators are underutilized. Since Cloud tenants are charged based on the duration of VM allocations rather than the actual usage of compute resources, longer training times entail higher costs, proportionately to the slowdown. Moreover, production DNN-training workloads typically are associated with SLOs that

[1] In this paper we do not consider asynchronous data-parallel methods because they are not commonly used in practice due to their weaker convergence guarantees and slow rates of convergence [7].

specify an expected time budget for training a model. SLO violations must be minimized to avoid financial losses or other adverse consequences.

A possible approach to avoid undesired slowdown is to control the communication phase of distributed training, aiming to ensure that communication time is roughly constant and stays predictable. Although gradient compression is agnostic to network conditions, we observe that its compression level can be adapted in response to network dynamics. Our main insight is that compression does not need to be applied uniformly across the entire training process. Instead, the level of compression can vary from one iteration to the next. Importantly, prior works [3], [10] show that this variation does not invalidate the convergence guarantees of the chosen compressor (granted their assumptions hold for the chosen compressor, which is expected in practice). Besides theoretical convergence bounds, our empirical results (§V) demonstrate that adaptive compression does not negatively affect convergence. The theoretical analysis of convergence guarantees under adaptive compression is left for future work.

At this point, the reader may wonder: if compression accelerates training, why not just use it consistently throughout training? Uniformly applying compression for the entire training process has a major drawback: it forces the decision of a single compression level to be taken *in advance*, thus without the ability to choose the compression level in an informed fashion, the trade-off between training accuracy and time is ignored. This is because the training time depends on both data reduction (which compression achieves) and network conditions (that may vary in the future). If the predetermined level is low, accuracy may be mostly unaffected, but data reduction might not be sufficient to achieve a training time within a budget. Whereas if the predetermined level is high, accuracy is sacrificed regardless of whether the network conditions could have afforded a lower compression level and better accuracy.

Since it is not possible to know future network dynamics in advance, we argue that compression should be exposed as a controllable knob to be employed as a mean to counter network congestion when the need arises. Fig. 1 graphically illustrates the role of adaptive compression for a fictitious example purporting the trade-off between accuracy and time. Effectively this approach retains the ability to avoid affecting model accuracy when network conditions are stable and not congested while employing compression proportionately to maximize the likelihood that training finishes within the time budget even if it comes at a small expense of minor accuracy reduction. Next, we describe our proposal towards this objective.

IV. Delay-aware Compression Control

DC2's goal is to keep network delays minimal by controlling the volume of data being exchanged during the gradient aggregation phase in response to the variable network dynamics.

To achieve this, we borrow from the principles and practices of network congestion control, that is, the idea of injecting data into the network in proportion to available bandwidth and responding to congestion signals. Secondly, we adapt the compression level knob to modulate the volume of data transmitted.

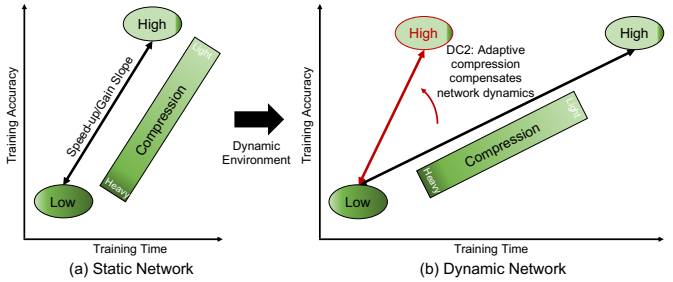


Fig. 1: With static network conditions (a), uniformly-applied gradient compression navigates the trade-off between training accuracy and time. With dynamic network conditions (b), adaptive compression is necessary to apply compression in proportion to network congestion so as to retain both high accuracy and short training time.

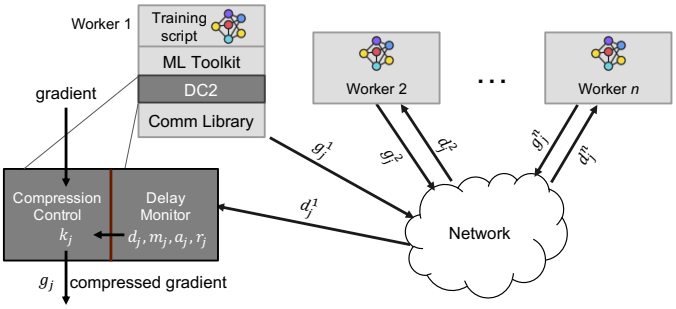


Fig. 2: High-level design of *DC2* .

Thus, *DC2* is a form of application-level congestion control similar in spirit to adaptive bit-rate algorithms commonly used for video streaming applications.

At a high-level, the design of *DC2* is admittedly straightforward — a main benefit of which is a simple and practical solution to adaptive compression control. *DC2* consists of two components (c.f. Fig. 2 for a graphical illustration):

Delay Monitor: this component maintains certain measurements and statistics of the network-level communication properties. The monitoring data are the following time series over iteration j of training: d_j, the communication phase delay; m_j, the minimum communication delay as witnessed up until iteration j; a_j, the average delay over the previous w samples $(d_j, \cdots, d_{(j-w)})$; r_j, the average delay differences over the previous w samples $(d_j - d_{(j-1)}, \cdots, d_{(j-w)} - d_{(j-w-1)})$.

Compression Control: this component embodies control logic that uses the monitoring data to adjust the gradient compression level. *DC2* is versatile and supports a wide range of flavors of adaptive compression logic. More details below.

DC2 is intended to be realized as a lightweight shim-layer that interposes between the ML toolkit and the communication library at each worker end point. That is, adopting *DC2* does not require changes to the training script nor to the ML toolkits. Rather, *DC2* integrates in the software stack as a proxy communication library that delegates the actual communication of the compressed gradient to existing communication libraries (e.g., OpenMPI or NCCL for collective operations or gRPC for parameter-server based communication).

Algorithm 1: Adaptive Compression Algorithms

Input: k_j: current ratio at iteration j
Output: k_{j+1}: the new ratio for iteration $j+1$

Function *DA1*(k_j)
  return $k_j - \frac{d_j - a_j}{d_j - m_j}$;

Function *DA2*(k_j)
  if $\frac{d_j - a_j}{d_j - m_j} > d_{\text{var}}$ **then return** $k_j - \frac{k_j}{2}$;
  if $\frac{a_j - d_j}{d_j - m_j} < d_{\text{var}}$ **then return** $k_j + k_{inc}$;
  return k_j

Function *DA3*(k_j)
  if $\frac{d_j - a_j}{d_j - m_j} > d_{\text{var}}$ **then return** $k_j - k_{dec}$;
  if $\frac{a_j - d_j}{d_j - m_j} < d_{\text{var}}$ **then return** $k_j + k_{inc}$;
  return k_j

Function *DA4*(k_j)
  if $\frac{r_j}{m_j} < 0$ **then**
    return $k_j + k_{inc}$
  else
    return $k_j \times \left(1 - \beta \times \frac{r_j}{m_j}\right)$
  end

Function *DA5*(k_j)
  if $d_j < \alpha \times m_j$ **then**
    return $k_j + k_{inc}$
  else
    if $d_j > \alpha \times a_j$ **then**
      return $k_j \times \left(1 - \beta \times \frac{d_j - \alpha \times a_j}{d_j - \alpha \times m_j}\right)$
    else
      if $\frac{r_j}{m_j} < 0$ **then**
        return $k_j + k_{inc}$
      else
        return $k_j \times \left(1 - \beta \times \frac{r_j}{m_j}\right)$
      end
    end
  end

A. Algorithms for Adaptive Compression

DC2 design is generic and allows for different control logic to be implemented. At each iteration, *DC2* invokes a user-specified adaptive compression algorithm to obtain the ratio k_{j+1} based on the current ratio k_j and the data tracked by the delay monitor.[2] In addition, to avoid over- or under-compression, *DC2* enforces a bound on the ratio to stay in $[k_{min}, k_{max}]$. These parameters are chosen by the user based on their needs.

Given this flexibility, we now propose 5 algorithms inspired by traditional as well as recent network congestion control schemes. The algorithms are shown in Algorithm 1. To keep their pseudo-code succinct, the state of the delay monitor is accessed by directly referencing the monitored variables (d_j, m_j, a_j, r_j). The common objective of these control algorithms is to minimize and keep the average delay a_j close to the minimum delay m_j. A brief description of these algorithms follows. Table I summarizes the algorithms' parameters and their default values. An illustration of the adaptive behavior of these algorithms in diverse scenarios will be given in Section V-E.

[2]For simplicity, we use the sparsification ratio of k in the discussion. The case of quantization bit-width as compression level is analogous but more involved since this level is discrete. Note that, for sparsification, due to the differences in the current ratio of k_j among the workers, they communicate different data volumes and indexes. Therefore, the values and indexes are collected via all-gather collective operation then the aggregation is performed [30], [31].

TABLE I: Parameters used in Algorithm 1.

Symbol	Meaning	Default
d_{var}	Allowed delay variation from target	0.05
k_{inc}	Additive increase step size	0.005
k_{dec}	Additive decrease step size	0.005
k_{max}	Maximum value for k	0.3
k_{min}	Minimum value for k	0.005
α	Allowed slack from the low and high	1.25
β	Multiplicative decrease factor	0.8

Proportional to normalized delay variation (DA1): this controller reacts in proportion to the delay deviation from the normalized target average a_j. k_j is updated proportionally to the delay difference from average divided by the difference from the minimum. This algorithm is simple but not recommended due to its oscillatory behavior as a result of its proportional reaction to delay dynamics. Therefore, we omit its results.

Additive-Increase Multiplicative-Decrease (DA2): this controller aims to maintain the delay around the target average d_j and uses the well-known AIMD rule [6] to adjust the knob k_j when the delay is outside the range $[a_j - d_{var}, a_j + d_{var}]$;

Additive-Increase Additive-Decrease (DA3): similar to DA2, this controller aims to maintain the delay around the target average d_j but applies additive decrease (AD) instead of multiplicative decrease (MD). AIAD (and similarly MIMD) control logic is known not to reach stability for congestion control problems [6];

Timely-Normalized Delay Gradient (DA4): This controller is inspired by Mittal et al. [29]. It reacts to the normalized gradient of changes in the delay ($\frac{r_j}{m_j}$) and adjusts the knob to stay at the target point. It applies Additive Increase (AI) by $\frac{k_j}{2}$ if the gradient is negative or else applies Multiplicative Decrease (MD) by $1 - \beta \times \frac{r_j}{m_j}$;

Timely-Thresholds Delay Gradient (DA5): This controller is similar to DA4 but introduces low (αm_i) and high (αa_i) thresholds and applies AI and MD if d_j are below or above them, respectively. Otherwise, DA4's logic is applied.

Parameters sensitivity: The aforementioned algorithms can be sensitive to the choice of the parameters in Table I. We choose these values empirically; k_{min} and k_{max} are chosen based on reasonable extremes for the compression, i.e., 0.5% and 30%, respectively. We did not conduct an exhaustive sensitivity analysis of all parameters, which we leave for future work. However, in our experience, the algorithms are not very sensitive to these parameters as long as they are reasonably set. This is because *DC2* controls the application data volume at each worker rather than controlling the sending rate, which is accomplished by the congestion control mechanism employed by the transport layer. The latter (i.e., sending rate) is more sensitive to parameter settings because it regulates access to the shared network resources among competing entities.

V. Experimental Evaluation

This evaluation answers the following questions:

- What benefits, in terms of training speed-up and model quality, does *DC2* provide when performance variability affects

TABLE II: Summary of the benchmarks used in this work.

Task	Neural Network	Model	Dataset	Training Parameters	Quality metric	Baseline quality	Training iterations
Image Classification	CNN	ResNet-20 [13]	CIFAR-10 [24]	269,467	Top-1 Accuracy	93.75%	2700
		VGG16 [39]	CIFAR-10 [24]	14,982,987		96.8%	2700
		ResNet-50 [13]	ImageNet [8]	25,559,081		73.75%	4950
Language Modeling	RNN	LSTM-2Layers [15] 1500 hidden units	PTB [28]	66,034,000	Test Perplexity	103.4	1800

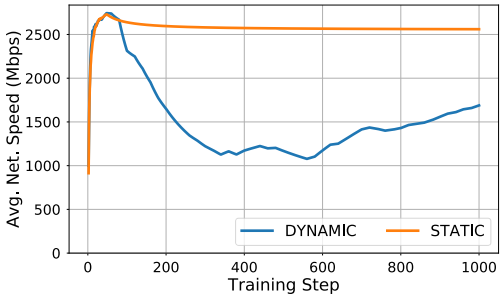


Fig. 3: Running average of network transfer speed measured by the application during the first epoch of training ImageNet benchmark in static and dynamic network settings.

network performance (*dynamic network scenario*)? (§V-B)
• Are there drawbacks to employ *DC2* in a network with little to no performance variability (*static network scenario*)? (§V-C)
• Does adaptive compression have an impact on model convergence? (§V-D)

Finally, we demonstrate (§V-E) that our proposed control algorithms (D2-D5) exhibit well behaved control dynamics in both network scenarios.

A. Experimental settings

Environment: We perform our experiments on 8 server-grade machines equipped with dual 2.6 GHz 16-core Intel Xeon Silver 4112 CPU, 512GB of RAM, and 25 Gbps network interface cards. Each machine has an NVIDIA V100 GPU with 16GB of memory. The servers run Ubuntu 18.04, Linux kernel 4.15.0. We use PyTorch 1.1.0 with CUDA 10.2 as the ML toolkit. We use Horovod 0.16.4 configured with OpenMPI 4.0.0 for collective operations of the distributed training.

Scenarios: We run the experiments in two network scenarios:
• *Static network:* the network observes minor delay variations, as it occurs when distributed training jobs run in dedicated clusters or a private cloud. In this case, ML workloads run in our dedicated cluster with all the available network bandwidth.
• *Dynamic network:* the network observes significant delay variations, as it occurs in public clouds where network performance variations and unpredictability has been observed [36] and is documented for VMs training ML models [26]. As done in previous work [26], [42], we emulate a dynamic network by rate limiting workers' transmission speed with performance variations at random. We take care to apply the same network speed profile across experiments to obtain a fair comparison.

Fig. 3 shows an example contrasting the two network scenarios based on the average network goodput (speed) as measured from the application level. This example was chosen as it closely matches reported network performance of distributed training in public clouds [26], [42]. Note, in the static scenario, distributed training is not able to exploit all available bandwidth.

Benchmarks: Table II lists the 4 ML workloads we use to evaluate *DC2*. We use both Convolution Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) models for image classification and language modeling tasks, respectively. We train the models using the SGD optimizer with momentum for the number of iterations listed in the table. Using other optimizers, while possible, is out of scope.

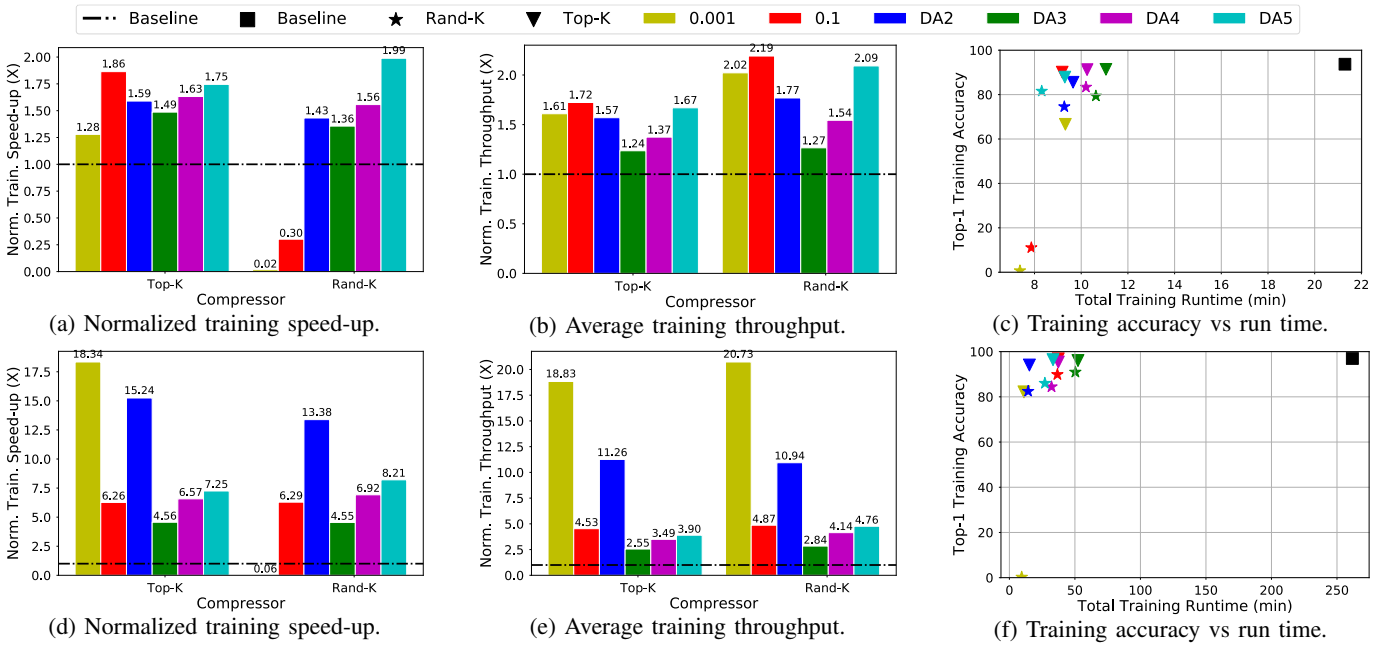
Due to the long training time that the ImageNet dataset entails, our training experiments with ImageNet last 5 epochs. We deem this sufficient to evaluate the speed up benefits of *DC2*. However, to verify that *DC2* converges for ImageNet, we also run the experiments for 90 epochs on a shared cluster where we are not allowed any control on the network.

Compressors: We experiment with two popular gradient compression methods: Random-k and a low-overhead version of Top-k (henceforth simply called Top-k).

Random-k uniformly at random selects $k\%$ of gradient elements. Inspired by the work of Lin et al. [25], the low-overhead version of Top-k randomly samples a subset of the gradient (e.g., 1%) and finds a threshold to select $\approx k\%$ of gradient elements by magnitude. Generally, Random-k has lower overheads than Top-k on a GPU. However, Top-k is known to perform better than Random-k [3]. We find that when Random-k is used, the PTB benchmark does not converge. Therefore, we do not show these results.

The gradient compression literature is vast and there are many techniques that have been proposed (for a recent survey see [46]). The choice of a particular compressor is largely orthogonal to *DC2*. In a sense, *DC2* applies to any compressor that admits a controllable compression level. We pick Random-k and Top-k because their selection criteria are at two opposite extremes of the spectrum (i.e., random vs. largest elements by magnitude), which gives us some confidence that other compressors are likely to lie somewhere in between. However, we leave it for future work to apply *DC2* to other compressors.

Error Compensation (EC) is a mechanism that improves convergence of compression methods in some cases [19], [25]. Note that EC comes at the cost of higher memory consumption of the limited resources of DNN accelerators (in our case, GPUs) and it might not always be usable, especially with a large mini-batch size. When EC cannot be used, the compression level generally has to be larger (e.g., $k \geq 0.1$) than when EC is used. We use EC throughout our experiments except for CIFAR-10 benchmarks.



(a) Normalized training speed-up. (b) Average training throughput. (c) Training accuracy vs run time.

(d) Normalized training speed-up. (e) Average training throughput. (f) Training accuracy vs run time.

Fig. 4: Training Performance of ResNet20 [(a),(b),(c)] and VGG16 [(d),(e),(f)] on CIFAR-10 in the dynamic scenario.

We use the case of no-compression as the *baseline*. We compare *DC2* with compression to both the baseline as well as uniformly applying compression throughout a training run. We use $k = 0.001$ and $k = 0.1$ as uniform compression ratios for the two compressors. These values represent extreme and moderate ratios, respectively. We set the seed of the random number generators so that the stochastic behavior of SGD and network dynamics is consistent for all schemes. We use a measurement window w of 50 iterations.

Metrics: We quantify the performance of the schemes (i.e., *DC2* methods, or uniform Random-k, Top-k) via these metrics:

• *Normalized Training Speed-up:* We evaluate the model quality at iteration T (the end of training) and divide it by the time taken to complete T iterations. We normalize this quantity by the same measurement calculated for the baseline. This is the normalized training speed-up relative to the baseline;

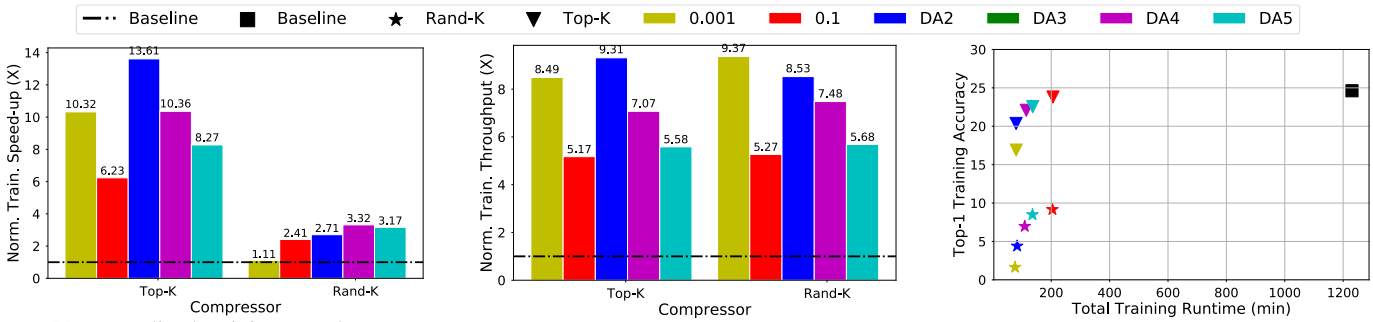
• *Normalized Average Training Throughput:* is the average throughput normalized by the baseline's throughput which shows the speed-ups from compression irrespective of its impact on model quality;

• *Quality vs Total Run Time:* The model quality (either top-1 accuracy or test perplexity) evaluated after T iterations in relation to the total run time to finish T iterations.

B. Dynamic Network Scenario

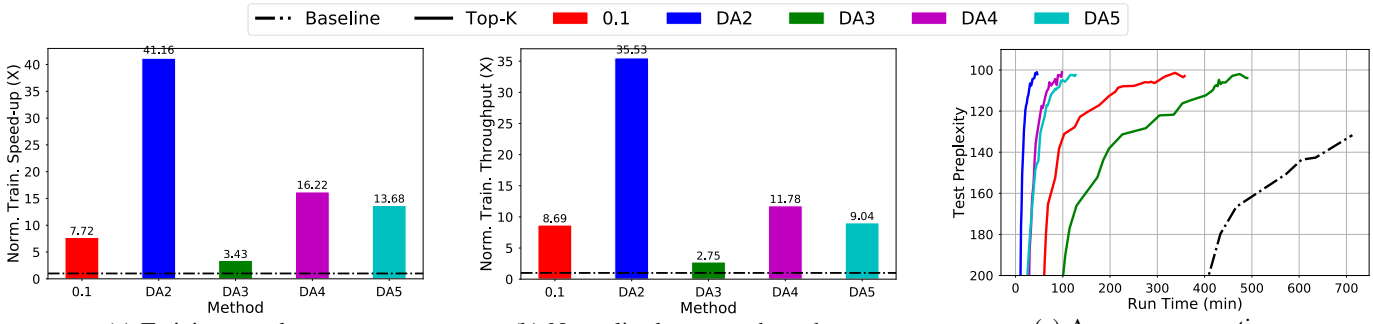
ResNet20 on CIFAR-10: Fig. 4a shows that, except for uniform Random-k compressors, both uniform Top-k and *DC2*-assisted compressors can achieve accuracy gains over no-compression baseline. Moreover, *DC2* achieves nearly same speed-ups of uniform compressors for Top-k and improves by $\approx 6.7\times$ over Random-k with ratio 0.1. We note that ratio 0.001 results in 0 speedup and divergence for Random-k and results in accuracy of 67 which is a huge reduction by 26.75% compared to baseline. Fig. 4b shows that, *DC2* are able to maintain nearly same average training throughput as the uniform compression. However, both the throughput and the speed-up results for Random-k highlights the benefits of the network-aware adaptation of *DC2* for the compressors. Specifically, without unnecessarily compromising the accuracy, *DC2* forces more aggressive compression to lower the communication time only whenever the network delays are high (Fig. 10c). Fig. 4c shows the training accuracy for both uniform compression and *DC2* reaches (or approaches) that of the baseline for Top-k (or Random-k). However, unlike *DC2* -based Random-k, Random-k even with a mild uniform ratio of 0.1 which enjoys high training throughput, can not converge. This signifies the benefits of adopting *DC2* as the regulator of compressed volume.

***VGG16 on CIFAR-10*:** Fig. 4d shows that *DC2* improves the training speed-up over no-compression baseline and uniform compression with ratio 0.1 by up to $\approx 15.2\times$ and $2.5\times$, respectively. Even though, Top-k with ratio 0.001 shows the highest speed-up, model accuracy is at 82 which is a significant reduction of 14.8% compared to baseline. Similar to ResNet20, Random-k with ratio 0.001 results in 0 speedup and model divergence. Fig. 4e shows *DC2* adaptions can compensate for the network variations and suppress the communication time (esp., DA2 which significantly improves the throughput). Even though, the throughput of ratio 0.001 is almost two times of DA2, it resulted in divergence and significant quality loss for both Random-k and Top-k, respectively. Notably, as Fig. 4f shows, VGG16 seems to be more robust and tolerant to compression compared to ResNet20. Therefore, both Top-k and Random-k compressors can converge to reasonable accuracy. This robustness allowed, less robust compression methods such as Random-k, to achieve comparable speed-ups



(a) Normalized training speed-up. (b) Average training throughput. (c) Training accuracy vs run time

Fig. 5: Performance of training ResNet50 on ImageNet in the dynamic network scenarios.



(a) Training speed-up. (b) Normalized average throughput. (c) Accuracy vs run-time.

Fig. 6: Performance of training an LSTM on PTB using Top-k compressor in dynamic network scenario.

of Top-k. The reason for this tolerance could be attributed to the parameter size and architectural differences between ResNet20 and VGG16 models.

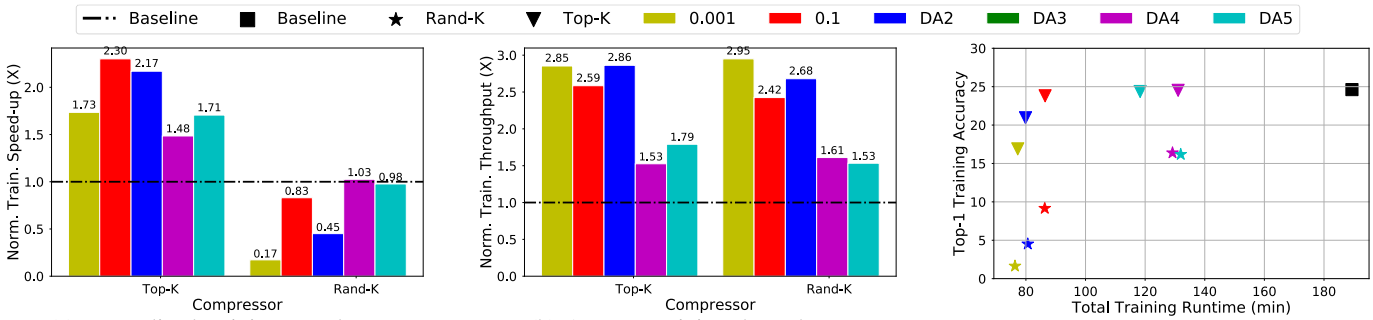
***ResNet50 on ImageNet*:** Fig. 5a shows that *DC2* methods have remarkable gains in terms of training speed-up over uniform compressors by up to ($\approx 2.2\times$ at ratio 0.1, $\approx 1.3\times$ at ratio 0.001). Fig. 5b shows that *DC2* methods (e.g., DA2) achieve on average training throughput as fast as the extreme compressor (i.e., ratio of 0.001) which shows that DC2 can adapt the compression rate to match the network condition and squeeze the communication as necessary to maintain high training throughput. Fig. 5c shows, accuracy of *DC2* methods are close to and higher than that of the moderate and extreme uniform compressor, respectively. This shows that *DC2* can maintain the training throughput with little to no impact on the accuracy.

RNN-LSTM on PTB: In this benchmark, due to the length of the no-compression baseline in the dynamic scenario, we force stop the experiment at step 1845 ($\approx$700 minutes). The results of Fig. 6a show that *DC2* methods achieve speed-ups over no-compression by up to $\approx 41\times$ and by $\approx 5.3\times$ over the uniform ratio compressor. The results of Fig. 6b show that all methods significantly improve the training throughput over no-compression. Moreover, these results show that the speed-ups compared to the training throughput are higher for *DC2* methods and lower for the uniform compressor. This suggests that the gains of *DC2* methods are not only from throughput gains but also the volume adaptions, only whenever necessary, in response to network delay variations which boosts the convergence. Fig. 6c shows the test perplexity over run-time of all the methods and show that compression (esp., DA2 scheme of *DC2*) can significantly improve the speed for reaching the target convergence values.

C. Static Network Scenario

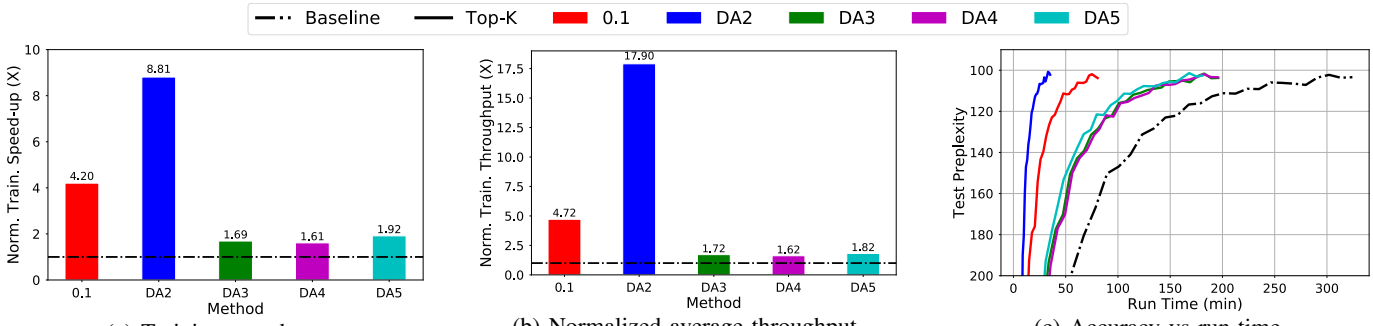
ResNet50 on ImageNet: Fig. 7a shows that *DC2* methods achieve comparable speed-ups to the moderate and extreme uniform compressors, respectively. And, Fig. 7b shows that *DC2* methods and uniform compression have comparable training throughput. Fig. 7c shows that moderate methods (e.g., 0.1, DA4 and DA5) achieve higher accuracy than aggressive ones (e.g., 0.001 and DA2). So, *DC2* achieves a middle-ground performance between mild and extreme uniform compressors in static scenario. Moreover, *DC2* provides the benefit of not requiring prior knowledge of nor performing hyper-parameter search for the best compression ratio in this scenario.

RNN-LSTM on PTB: The speedup results shown in Fig. 8a show that *DC2* methods achieve speed-ups over no-compression by up to $\approx 8.8\times$ and by $\approx 2.1\times$ over the uniform ratio compressor. The results of Fig. 8b show that all methods improve mildly the training throughput over no-compression (except for, DA2 which boosts it to $18\times$ the baseline). This translates to mild speedups over the baseline compared to the dynamic scenario presented earlier. Fig. 8c shows the test perplexity over run-time and show that all methods reach the same (or slightly better) model quality over the baseline. It also shows that compression (esp., DA2) can significantly improve the convergence speed for reaching these quality values.



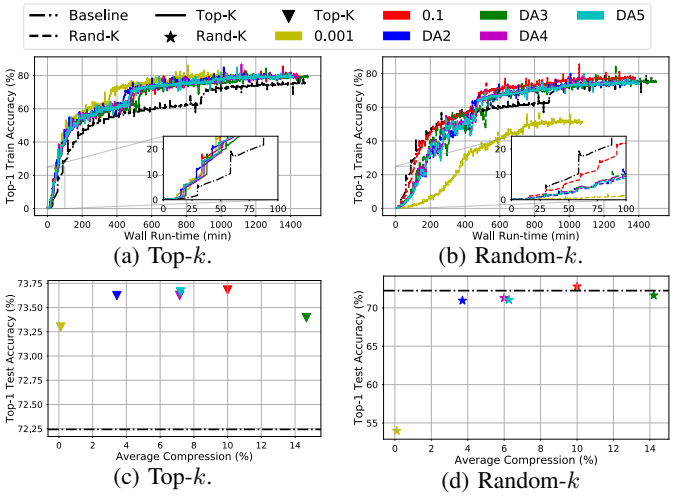
(a) Normalized training speed-up. (b) Average training throughput. (c) Training accuracy vs run time.

Fig. 7: Performance of training ResNet50 on ImageNet dataset in the static network scenarios.



(a) Training speed-up. (b) Normalized average throughput. (c) Accuracy vs run-time.

Fig. 8: Performance of training an LSTM on PTB using Top-k compressor in the static network scenario.



(a) Top-k. (b) Random-k.

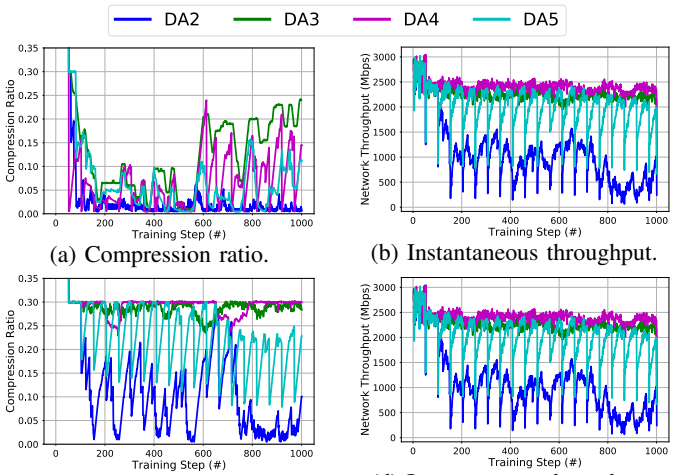
(c) Top-k. (d) Random-k

Fig. 9: Convergence Experiments: Training ResNet50 on ImageNet using Top-k [a,c] and Random-k [b,d] compressors.

D. Convergence Experiments

To evaluate the convergence of the compressors with *DC2* for large datasets such as ImageNet, we run distributed training of the ResNet50 model till convergence (i.e., 90 epochs) in an allocations-based shared cluster. In this cluster, for each experiment, a single compute-server equipped with 8 NVIDIA V100-32GB GPUs is allocated for a time limit of 24 hours. The workers communicate intra-node via PCIe interface. The monitor module is fed with delay information generated following the profile of the dynamic network shown in Fig. 3.

The results in Fig. 9 show the convergence behaviour for (a) Top-k and (b) Random-k and the average compression ratio used to achieve a final test accuracy for (c) Top-k and (d) Random-k. As the experiment runs towards completion nearing the end of training time limit, we make the following observations: i) the average compression ratio (or average communicated volume normalized by the baseline) of *DC2* methods are $\approx (0.034, 0.145, 0.071, 0.072)$ or $(3.4\%, 14.5\%, 7.1\%, 7.2\%)$ for (DA2, DA3, DA4, DA5) controllers, respectively. ii) Top-k converges to close but slightly higher accuracy over the baseline, for all methods (i.e., 0.1, 0.001 and *DC2* adaption schemes), which conforms to results in Fig. 5c which showed Top-k achieves accuracy values close to baseline. We note that uniform compressor with ratio 0.001 (or 0.1%) achieves the lowest accuracy among all methods; iii) Even though, Fig. 5c showed Random-k exhibited significantly lower quality values compared to baseline, most methods converge to the same accuracy level as of the baseline. Specifically, only *DC2* adaption schemes and uniform Random-k compressor with ratio 0.1 are able to converge to baseline accuracy. In contrast, Random-k with ratio of 0.001 achieves $\approx 53\%$ test accuracy, which is significantly lower by 20 accuracy points compared to the baseline accuracy. Moreover, for this compression ratio, running the experiment for more epochs did not help with achieving higher levels of accuracy. These results show that adaptations of compression level by *DC2* does not harm convergence rather it helps compression methods achieve baseline accuracy.



(a) Compression ratio. (b) Instantaneous throughput.

(c) Compression ratio. (d) Instantaneous throughput.

Fig. 10: *DC2* controllers' dynamics in a dynamic [(a),(b)] and static [(c),(d)] environment.

E. Dynamics of DC2 *Controllers*

Here, we study the dynamics of the various *DC2* control methods discussed in Section IV-A. To show the adaptive dynamics of different compression controllers presented in Algorithm 1, we present their compression level and throughput dynamics during the first epoch of ImageNet training on 8 nodes in the dedicated cluster.

Fig. 10 shows the dynamics of *DC2* methods and the corresponding network throughput in static and dynamic networks, respectively. The results show that most of *DC2* methods, in both static and dynamic settings, can adjust the compression ratio in response to the network condition, which reflects on the throughput of the workers. Moreover, the results suggest that DA2, DA4, and DA5 are the preferred methods to provide the best performance. DA2 tends to stay at low ratios, so it is more recommended for high-quality (or low-error) compressors (e.g., Top-k) or robust models (e.g., VGG16). DA3 method tends to over-shoot the ratio and have more oscillations than the former methods. DA1 is extremely oscillatory due to its proportional control law, and hence, for clarity, its dynamics' results were not shown. In summary, results show that *DC2* is an effective system for adapting training speed to varying network conditions via an adaptive control of the compression.

VI. Related work

Distributed Training: Recently, there have been many advances in ML toolkits that support distributed DNN training. These advances are fueled by a vat body of work over the past decade to further our understand of the learning process as an optimization problem solved in a distributed setting. Numerous proposals analyze and optimize the convergence behavior of distributed, asynchronous and/or low-precision versions of SGD. First, [48] analyzes a parallel version of SGD. Recently, more work try to speed up distributed SGD by scaling the batch size [21], [32] or adapting the learning rate [4], [12]. While these methods improve the optimizer, communication remains a practical dominant bottleneck in scaling distributed training.

Scaling Distributed ML: Many recent works tackle the challenges in scaling distributed ML jobs. As communication is predominately the bottleneck [26], [34], [42], [43], compression methods aim to reduce communication time by means of sparsification [1], [25], quantization [2] or delayed aggregation [47]. For instance, [25] proposes a low-overhead Top-k by sampling gradient elements and calculate a threshold to sparsify the Top-k elements. Others works [3], [19] study the convergence of distributed SGD and prove guarantees of various compressors. Even though these methods reduce the communication time, they do not provide a practical way of choosing/tuning their communication reduction knobs. *DC2* fills this gap by adapting these knobs via low-complexity controllers without requiring expensive any coordination among the controllers across workers. Concurrently with *DC2*, DBW [45] adapts the number of parameter servers used to perform gradient aggregation without further delay based on a threshold on the estimated gain of the loss function. However, the benefits of DBW in real benchmarks and use cases are questionable as the experiments use only simplistic settings and both computation and communication are simulated using generic distributions. Orthogonal works distribute the aggregation load among workers by forming a hierarchical aggregation tree [26], optimize the computation-communication overlap [33] or devise an in-network aggregation function to minimize communication time [34]. Largely, adopting and choosing the right setting for many DNN training optimizations to scale distributed training jobs remains an open problem. For instance, [40] proposed MLSL to scale distributed training to up to thousands of nodes across the cloud. These efforts could benefit from our adaptive schemes such as *DC2* that can automate the process of tuning knobs of communication reduction methods and dynamically adapt them to the conditions of the environment.

VII. Conclusions

We identified a practical problem for distributed ML workloads running in diverse network conditions. Most works leverage gradient compression as means of reducing the communication time without addressing the question of *when and by how much compression should be applied in various network conditions?* In this work, we tackled this question. First, we identified the importance of adapting compression to variable network dynamics. Then, similar in spirit to the idea of network congestion control, we leveraged controllable compression knobs to regulate the communicated volume to maintain communication delays under control. This effectively forms a feed-back closed-loop system between the network environment and the compression controller. We designed *DC2* that employs a network monitor and a compression controller. We implemented *DC2* as a drop-in component to an existing communication library. We evaluated *DC2* on distributed training in static and dynamic network scenarios. The results showed that *DC2* improved the training speed-up in dynamic settings by $\approx 2.7\times$, $\approx 2.2\times$ and $\approx 5.3\times$ compared to uniform compression for CIFAR-10, ImageNet, and PTB benchmarks, respectively.

REFERENCES

[1] A. F. Aji and K. Heafield. Sparse Communication for Distributed Gradient Descent. In *EMNLP-IJCNLP*, 2017.

[2] D. Alistarh, D. Grubic, J. Z. Li, R. Tomioka, and M. Vojnovic. QSGD: Communication-efficient SGD via gradient quantization and encoding. In *NeurIPS*, 2017.

[3] D. Alistarh, T. Hoefler, M. Johansson, S. Khirirat, N. Konstantinov, and C. Renggli. The convergence of sparsified gradient methods. In *NeurIPS*, 2018.

[4] L. Balles, J. Romero, and P. Hennig. Coupling adaptive batch sizes with learning rates. In *Uncertainty in Artificial Intelligence (UAI)*, 2017.

[5] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet allocation. *Journal of Machine Learning Research*, 3:993–102, 2003.

[6] D.-M. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN Systems*, 17(1):1 – 14, 1989.

[7] W. Dai, Y. Zhou, N. Dong, H. Zhang, and E. P. Xing. Toward understanding the impact of staleness in distributed machine learning. In *ICLR*, 2019.

[8] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *IEEE CVPR*, 2009.

[9] J. C. Duchi, E. Hazan, and Y. Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

[10] A. Dutta, E. H. Bergou, A. M. Abdelmoniem, C.-Y. Ho, A. N. Sahu, M. Canini, and P. Kalnis. On the Discrepancy between the Theoretical Analysis and Practical Implementations of Compressed Communication for Distributed Deep Learning. In *AAAI*, 2020.

[11] Z. Hao Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. In *USENIX ATC*, 2017.

[12] C. Hardy, E. Le Merrer, and B. Sericola. Distributed deep learning on edge-devices: Feasibility via adaptive compression. In *IEEE International Symposium on Network Computing and Applications*, 2017.

[13] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *IEEE CVPR*, pages 770–778, 2015.

[14] G. Hinton, L. Deng, D. Yu, G. Dahl, A. R. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 2012.

[15] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computing*, 9(8):1735–1780, 1997.

[16] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu. Gaia: Geo-distributed machine learning approaching LAN speeds. In *USENIX NSDI*, 2017.

[17] C. C. Hung, L. Golubchik, and M. Yu. Scheduling jobs across geo-distributed datacenters. In *ACM SoCC*, 2015.

[18] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *USENIX ATC*, 2019.

[19] S. P. Karimireddy, Q. Rebjock, S. U. Stich, and M. Jaggi. Error Feedback Fixes SignSGD and other Gradient Compression Schemes. In *ICML*, 2019.

[20] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and F. F. Li. Large-scale video classification with convolutional neural networks. In *IEEE CVPR*, 2014.

[21] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. In *ICLR*, 2017.

[22] D. P. Kingma and J. Ba. ADAM: A Method for Stochastic Optimization. In *ICLR*, 2015.

[23] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtarik, A. T. Suresh, and D. Bacon. Federated Learning: Strategies for Improving Communication Efficiency. In *NeurIPS Workshops*, 2016.

[24] A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[25] Y. Lin, S. Han, H. Mao, Y. Wang, and W. Dally. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. In *ICLR*, 2018.

[26] L. Luo, P. West, J. Nelson, A. Krishnamurthy, and L. Ceze. PLink: Discovering and Exploiting Locality for Accelerated Distributed Training on the public Cloud. In *MLSys*, 2020.

[27] L. Mai, C. Hong, and P. Costa. Optimizing Network Performance in Distributed Machine Learning. In *USENIX HotCloud*, 2015.

[28] M. P. Marcus, B. Santorini, M. A. Marcinkiewicz, and A. Taylor. Treebank-3. https://catalog.ldc.upenn.edu/LDC99T42.

[29] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. TIMELY: RTT-Based Congestion Control for the Datacenter. In *ACM SIGCOMM*, 2015.

[30] NVIDIA NCCL. https://docs.nvidia.com/deeplearning/nccl/.

[31] OpenMPI MPI_Allgather. https://www.open-mpi.org/doc/v4.0/man3/MPI_Allgather.3.php.

[32] N. Parikh. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *Foundations and Trends in Optimization*, 2014.

[33] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In *ACM SOSP*, 2019.

[34] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. K. Ports, and P. Richtárik. Scaling Distributed Machine Learning with In-Network Aggregation. In *USENIX NSDI*, 2021.

[35] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs. In *INTERSPEECH*, 2014.

[36] R. Shea, F. Wang, H. Wang, and J. Liu. A deep investigation into network performance in virtual machine based cloud environments. In *IEEE INFOCOM*, 2014.

[37] S. Shi, Q. Wang, and X. Chu. Performance modeling and evaluation of distributed deep learning frameworks on gpus. In *IEEE DataCom*, 2018.

[38] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv 1909.08053*, 2019.

[39] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

[40] S. Sridharan, K. Vaidyanathan, D. Kalamkar, D. Das, M. E. Smorkalov, M. Shiryaev, D. Mudigere, N. Mellempudi, S. Avancha, B. Kaul, and P. Dubey. On Scale-out Deep Learning Training for Cloud and HPC. In *SysML*, 2018.

[41] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *IEEE CVPR*, 2015.

[42] H. Tang, S. Gan, C. Zhang, T. Zhang, and J. Liu. Communication Compression for Decentralized Training. In *NeurIPS*, 2018.

[43] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer. A survey on distributed machine learning. *ACM Computing Surveys*, 53(2):1–33, May 2020.

[44] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li. TernGrad: Ternary gradients to reduce communication in distributed deep learning. In *NeurIPS*, 2017.

[45] C. Xu, G. Neglia, and N. Sebastianelli. Dynamic backup workers for parallel machine learning. In *IFIP Networking Conference*, 2020.

[46] H. Xu, C.-Y. Ho, A. M. Abdelmoniem, A. Dutta, E. H. Bergou, K. Karatsenidis, M. Canini, and P. Kalnis. Compressed Communication for Distributed Deep Learning: Survey and Quantitative Evaluation. Technical report, KAUST, Apr 2020. http://hdl.handle.net/10754/662495.

[47] S. Zheng, Q. Meng, T. Wang, W. Chen, N. Yu, Z.-M. Ma, and T.-Y. Liu. Asynchronous stochastic gradient descent with delay compensation. In *ICML*, 2017.

[48] M. A. Zinkevich, M. Weimer, A. Smola, and L. Li. Parallelized stochastic gradient descent. In *NeurIPS*, 2010.