ScaDLES: Scalable Deep Learning over Streaming data at the Edge

Sahil Tyagi
Department of Intelligent Systems Engineering
Luddy School of Informatics, Computing and Engineering
Indiana University Bloomington
Indiana, USA
styagi@iu.edu

Martin Swany
Department of Intelligent Systems Engineering
Luddy School of Informatics, Computing and Engineering
Indiana University Bloomington
Indiana, USA
swany@iu.edu

***Abstract*—Distributed deep learning (DDL) training systems are designed for cloud and data-center environments that assumes homogeneous compute resources, high network bandwidth, sufficient memory and storage, as well as independent and identically distributed (IID) data across all nodes. However, these assumptions don't necessarily apply on the edge, especially when training neural networks on streaming data in an online manner. Computing on the edge suffers from both systems and statistical heterogeneity. Systems heterogeneity is attributed to differences in compute resources and bandwidth specific to each device, while statistical heterogeneity comes from unbalanced and skewed data on the edge. Different streaming-rates among devices can be another source of heterogeneity when dealing with streaming data. If the streaming rate is lower than training batch-size, device needs to wait until enough samples have streamed in before performing a single iteration of stochastic gradient descent (SGD). Thus, low-volume streams act like stragglers slowing down devices with high-volume streams in synchronous training. On the other hand, data can accumulate quickly in the buffer if the streaming rate is too high and the devices can't train at line-rate. In this paper, we introduce *ScaDLES* to efficiently train on streaming data at the edge in an online fashion, while also addressing the challenges of limited bandwidth and training with non-IID data. We empirically show that *ScaDLES* converges up to $3.29\times$ faster compared to conventional distributed SGD.**

***Index Terms*—Deep learning, Distributed training, Streaming data, Federated learning, Adaptive compression**

I. Introduction

With the advent of big data and IoT, the number of smart devices has grown exponentially over the years. These devices capture data across a wide range of modalities, such as image/video in smartphones and surveillance camera feeds, audio and speech from smart speakers, text/language on phone/tablet keyboards etc. The data collected on the devices can either be moved to a centralized server in the cloud or persist locally. Local storage is practical when network bandwidth is limited and data privacy is a concern.

Locally storing data presents its own challenges due to limited capacity on edge/fog devices. The problem is exacerbated on devices with high-inflow streaming data. The data lifetime is also influenced by device streaming rates as high-volume streams may require more frequent storage purge or handling via other means. Commercial solutions offer data storage in the cloud for finite time, but this violates data privacy, incurs high communication cost of data movement and the subscription-based cost to store that data. *Distributed deep learning (DDL)* typically assumes centralized data, where each process/device samples training data in an IID fashion at every iteration. However, this is not necessarily true for streaming data which can be skewed not just in volume, but can be unbalanced and have non-IID distribution as well. Another consequence of training on devices with varying flow-rates is that high-inflow devices may have to wait on low-inflow ones until they gather enough samples corresponding to the mini-batch set for training. Thus, devices with low-volumes of streaming data can be essentially perceived as *stragglers* that slow down distributed training.

In DDL, gradients computed locally are aggregated into a global update which is propagated back to the devices before proceeding to the next iteration. The size of the gradients communicated is of the same scale as the number of trainable parameters in the network, which can span over hundreds of millions or even billions for modern language and vision models. Using single-precision (32-bit) floats to represent gradients means that hundreds of megabytes or even gigabytes of data needs to be exchanged at every iteration. Thus, heterogeneity in data inflow among devices, unbalanced-ness in device-local data, finite memory/storage and limited bandwidth violate assumptions of conventional distributed training designed for HPC and cloud.

In this paper, we build a streaming-based distributed training framework cognizant of the aforementioned issues that we call ***ScaDLES:*** **{Sca}**lable **{D}**eep **{L}**earning over **{S}**treaming data at the **{E}**dge. *ScaDLES*[1] is designed to train across devices with heterogeneous volumes of streaming data in an online manner. Instead of waiting for all workers to accumulate enough samples corresponding to the mini-batch, we choose a variable min-batch size for each device based on its streaming rate. As a result, there is no additional wait-time on account of low-volume devices. To aggregate gradients across workers, we perform *weighted aggregation* such that a device with a larger batch size is weighted more than those with smaller batches. We empirically show how this weighted

[1] Code available at https://github.com/sahiltyagi4/ScaDLES

gradient aggregation approach converges faster than typical distributed SGD.

To tackle the issues of limited memory, storage and expensive disk IO, we compare two simple data storage policies: **Stream Persistence** and **Truncation**. We simulate streams and implement these policies with Apache Kafka [1], a popular distributed stream processing platform.

Lastly, to deal with limited bandwidth and high communication cost of gradient reduction, we propose an adaptive compression technique where we scale the compression ratio based on gradient variance and adjusting to critical regions [3] in the training phase. We apply this adaptive method on Top-k gradient sparsification [4]. *ScaDLES* works in an online, black-box manner that we validate by simulating streams with different degrees of heterogeneity, both on IID and non-IID data, and compare performance with conventional distributed SGD.

II. Challenges in Streaming DL

DDL training on streaming data presents unique challenges that can severely impact training time and/or convergence quality. Using data streams simulated on Kafka and Pytorch's [5] distributed data-parallel [6] module, we observe the effects of heterogeneous streams, skewness in training data, limited memory/storage and communication cost of synchronizing model updates on the overall training time and model convergence.

A. Heterogeneity in device streaming rates

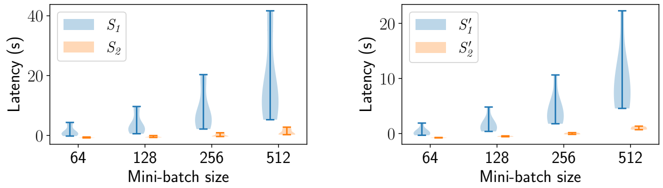
In conventional DDL, multiple devices train a local model replica on partitions sampled from the entire training dataset, and aggregate gradient updates at the end of each iteration either via parameter servers [7] or Allreduce using communication libraries like Open MPI [8] and NCCL [9]. With distributed SGD, parameter update w at iteration $(t+1)$ for N devices optimizing loss function $\mathcal{L}(\cdot)$ on a sample x_i of size b_i from distribution $\mathcal{X}_i$ and learning rate η is given by Eqn. (1).

$$w_{t+1} = w_t - \eta \frac{1}{N} \sum_{n=1}^{n=N} \frac{1}{|b_i|} \sum_{i \in b_i} \frac{\partial}{\partial w_t} \mathcal{L}(x_{(i,n)}, w_t) \qquad (1)$$

Each device trains on the same mini-batch size b, making global batch-size $N \cdot b$. However, when dealing with streaming data, devices can have different streaming rates. Devices with high-volume streams can readily collect b samples, while those with sparse inflow rates need to wait until samples equal to b are collected. With an inflow rate of p samples/sec., a device would have to wait about (b/p) seconds before proceeding to perform forward-backward pass. We consider such variances in streaming rates among devices as ***streaming heterogeneity***. Heterogeneity can be inter or intra-device as well; the streaming rate on a device itself can vary based on traffic, usage, time of day, etc. To understand how streaming heterogeneity can affect wall-clock time due to latency incurred while gathering a mini-batch, we sample streaming rates from different distributions and compute the latency incurred to collect different batch sizes.

TABLE I
Devices in a cluster sampled with varying streaming rates

Distribution	**Sample set**	**Mean**	**Std. Dev.**
Uniform	S_1	38	24
	S_2	300	112
Normal	S_1'	64	24
	S_2'	256	28



(a) Latency in S_1 and S_2

(b) Latency in S_1' and S_2'

Fig. 1. Streaming latency across batches when device stream-rates are sampled from different distributions.

In Table I, we use two sets each of uniform and normal distribution to sample streaming rates for devices. These two distributions capture inflow heterogeneity that we typically expect to see in real-world settings. Uniform distribution samples evenly across a given range, thus giving more heterogeneous streaming rates. On the other hand, rates sampled in normal distribution are centered around the mean so it resembles a more homogeneous setting w.r.t the device streaming rates. Sets S_1 and S_1' have a smaller mean as well as variance, while S_2 and S_2' represent a higher mean and larger standard deviation. $[S_2, S_2']$ denote higher streaming rates compared to $[S_1, S_1']$.

The batch size is an important *hyperparameter* in deep learning, i.e., a factor that influences convergence in neural networks. A small batch size cannot be efficiently parallelized, while a very large batch size increases generalization error [10]. For now, we don't take these considerations into account and only see the latency incurred to gather different batch sizes when we sample streaming rates from the described distributions. Fig. 1 shows the streaming latency across each set for different batches. Latency increases with larger batches as more training samples need to be collected. *Thus, the device with the lowest streaming rate (and maximum latency) effectively becomes a straggler in synchronous training as other devices wait on it to gather a mini-batch, perform computation and send its local gradients for reduction.*

B. Data skewness in deep learning

While collaboratively training models, data on a device can be skewed either in volume, properties, or both. For unbalancedness due to volume, imagine a traffic surveillance system where devices capture identically distributed data like frames of individual vehicles (car, bike, trucks, etc.), but the

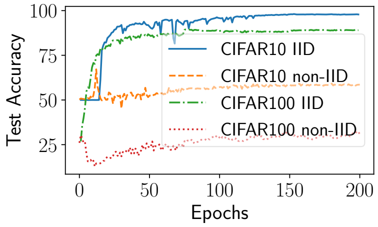
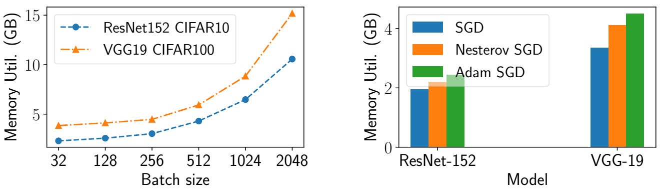


Fig. 2. Test accuracy for ResNet152 on CIFAR10 and VGG19 on CIFAR100 with IID and non-IID data.



(a) Memory util. vs. batch-size

(b) Memory varies with SGD variant

Fig. 3. GPU memory utilization in DDL.

volume of data on each device varies with the traffic density on the route where the camera is installed. Skewness due to data properties is introduced when the distribution of device-local data varies significantly from the overall data distribution. For example, a vehicle recognition model running on a video surveillance system installed in a subway captures images of trains, while devices installed on the airport cover flying vehicles only. Thus, training data has non-IID distribution as it has partial labels only (like a train or a plane). Privacy-sensitivity and large volumes of data on constrained networks make it unfeasible to move it to a centralized location like the cloud.

We train two popular image classifers: ResNet152 [11] and VGG19 [12] on skewed data to observe the impact of unbalanced and non-IID distribution on convergence. We induce non-IID distribution of CIFAR10 and CIFAR100 [13] by mapping a subset of labels to a unique device. We train on 10 devices for CIFAR10 such that a single label resides on one device, while we train CIFAR100 on 25 devices by mapping 4 labels to a single device. Using **Top-5 test accuracy** as the performance metric, Fig. 2 shows the result of training ResNet152 on CIFAR10 and VGG19 on CIFAR100. For comparison, we also show the corresponding performance of training with data partitioned in an IID manner. The model quality degrades considerably on non-IID data for both models and datasets.

C. Limited Memory and storage

With high volumes of streaming data, further processing and storage can be costly or even unfeasible due to limited physical resources. Limited memory presents challenges even in data-center settings where GPU memory is significantly lower than system memory. Training a neural network on a GPU requires storing model parameters (a.k.a weights), gradients computed in backward pass, activation maps as well as training batches. Fig. 3 shows how GPU memory utilization varies on NVIDIA V100 GPUs based on the mini-batch size and the kind of optimization used. Keeping all other hyperparameters fixed, memory usage increases in a near-exponential fashion with batch size (Fig. 3a). From Fig. 3b, memory consumption also increases as we move from mini-batch SGD to Nesterov's momentum [14], and then to Adam optimizer [15]. Nesterov's momentum needs more memory than mini-batch SGD since it keeps parameter updates from the previous timestep as well. Adam optimizer consumes even more memory since it stores both first and second order updates from previous timestep. Even though devices designed for the edge are now more capable than ever, its still more resource constrained than dedicated data-center hardware. This makes training neural networks on the edge even more challenging.

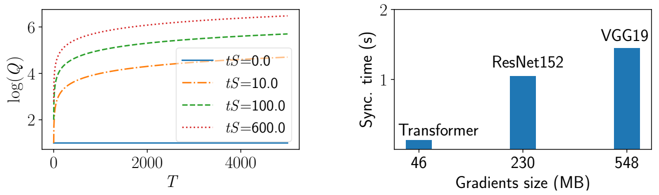
Since neural network training is compute-intensive, it is difficult to train models on streaming data at *line-rate*. High synchronization overhead to aggregate updates further inhibits linear scaling of DDL. As a result, *data quickly accumulates if the streaming rate is higher than the processing rate*.

The size of streaming queues can quickly blow up in distributed training, be it on-disk or in-memory. We formulate this as follows: suppose each device D_i among $[D_1, D_2, ...D_n]$ devices has a fixed streaming rate of $S^{(i)}$ samples/second. The devices collaboratively train a model with average batch-size b_i (i.e., $b_i = \sum_{j=1}^{j=n} b_j/n$). Consider a scenario where a devices' streaming rate is larger than the training batch-size, i.e., $S^{(i)} > b_i$. A single iteration in distributed training involves calculating loss, computing gradients, aggregate and apply updates; let's denote this time on device i as t_i. At initial timestep $ts = 0$, $S^{(i)}$ samples arrive each second at i which then processes b_i samples from it. In the time t_i that i completes one training iteration, about $(t_i \cdot S^{(i)})$ more samples arrive in addition to the residual $(S^{(i)} - b_i)$ that weren't used. Thus, there are $(S^{(i)} - b_i) + t_i \cdot S^{(i)}$ samples enqueued in the streaming buffer at timestep $ts = 1$. At timestep $ts = 2$, there are $2(t_i + 1)S^{(i)} - 2b_i$ samples in the buffer.

The queue size increases over time on account of residual samples from previous timesteps. We generalize the number of accumulated samples Q_i on device i after T timesteps in Eqn. (2) and note that Q_i scales linearly with T.

$$Q_i = (t_i \cdot S_i - b_i) \cdot T + S^{(i)} \quad \forall \quad t_i \cdot S^{(i)} \geq b_i \tag{2}$$

As a timestep corresponds to an iteration in DDL, buffer size can increase dramatically when T is large, which is typical for neural networks to run for thousands of iterations. To limit Q_i from blowing up, one could argue to set b_i to $t_i \cdot S^{(i)}$. In that case, Q_i is always equal to $S^{(i)}$ irrespective of the value of T . However, b_i is a hyperparameter that may require careful tuning. Using $t_i \cdot S^{(i)}$ as batch size can be impractical if the streaming rate is too high or too low. A small batch-size doesn't leverage parallelism while a large b_i would hurt generalization performance.



(a) Streaming queue grows over time (b) Gradient synchronization time

Fig. 4. DDL on streaming data is limited by memory/storage as well as network bandwidth.

TABLE II
Data accumulated with streaming in DDL

Model	t	S	**Data accumulated at** T **steps (GB)**		
	(s)	(img/s)	$T = 10^3$	$T = 10^4$	$T = 10^5$
ResNet152	1.2	100	0.35	3.5	34.33
		600	2.06	20.6	200.6
VGG19	1.6	100	0.47	4.69	46.8
		600	2.75	27.5	274.83

Assuming high streaming rates and considerable iteration times in DDL due to limited compute and bandwidth at the edge, Eqn. (2) reduces to

$$Q_i = (T \cdot t_i \cdot S^{(i)} + S^{(i)}) \quad \text{if} \quad (t_i \cdot S^{(i)}) \gg b_i \qquad (3)$$

We simulate how Q_i increases with T as stated in Eqn. (3). The results are illustrated for different tS values in Fig. 4a. The y-axis takes the log (with base 10) of the samples accumulated. As tS increases, so does the corresponding buffer size. We note that when $tS \approx 0$, the buffer only holds S samples at any given time. Such a setting describes a hypothetical system where the total iteration time is negligible regardless of the streaming rate.

To gauge buffer requirements with streaming data in real-world settings, we measure the space needed to store $[32 \times 32]$ colored images for training ResNet152 and VGG19. For mini-batch size 64, the models have average iteration times of 1.2 and 1.6 seconds respectively. Table II tabulates how training samples accumulate after $1K, 10K, 100K$ timesteps for these iteration times. As T increases, so does the storage requirements to hold the data. Optimized stream processing platforms like Kafka reduce memory footprint by storing messages on-disk as partitions, and then deleting the data based on some retention policy once the messages are successfully consumed. However, persisting data on-disk becomes unfeasible especially on the edge as data keeps accumulating with the iterations.

D. Synchronization overhead

Although training neural networks on GPUs can significantly reduce the computation time, DDL can still incur significant overhead due to periodic gradient synchronization. Training ResNet152 and VGG19 on 8 NVIDIA K80s takes about 80 to 90% of the total iteration time in gradient synchronization. Additionally, communication cost tends to increase with the number of devices participating in training. Increasing the network bandwidth brings down the synchronization cost to only a certain extent and saturating thereafter [2].

The 8 GPUs are connected via 5Gbps ethernet for which we plot the communication time to synchronize gradients for Transformer [16], ResNet152 and VGG19. Fig. 4b incurs higher communication time as the model size increases.

III. Background and Related Work

A. Handling unbalanced and non-IID data

Neural network training under constraints like privacy-sensitivity, skewness due to non-IID and unbalanced data has been well studied under the premise of *federated learning*. In federated training, devices train on local, skewed data while a global shared model is learned by periodically aggregating updates from other devices. For example, FedAvg [19] collects updates only from a fraction of total clients after certain local epochs to reduce frequent communication cost. FedProx [23] extends FedAvg to include partial work (to address systems heterogeneity) and adds a proximal term to the local objective function to deal with statistical heterogeneity in non-IID data. Sparse Tenary Compression (STC) [18] reduces communication by combining Terngrad [24] quantization with Top-*k* [25] sparsification, and is shown to perform better than FedAvg on both IID and non-IID data. Zhao et al. [17] account for data skewness due to weight divergence among devices' local model replica and measure it with earth mover's distance (EMD) between device-local and overall data distribution. To facilitate development of federated learning systems and algorithms, benchmarks like FedML [20] and LEAF [21] have been developed.

B. Dealing with limited memory

As neural networks have grown in size over the years, so has their resource requirements. The memory space to hold a model comprises of trainable parameters and a computation graph to store gradients and activation maps computed in forward-backward pass. Micikevicius, et al. [26] proposed automatic mixed-precision (AMP) training with half-precision (16-bit) floating points to reduce the memory footprint by half. Gradient checkpointing [27] trades memory for computation by flushing intermediate data from the computation graph to reduce memory utilization. This comes at the cost of increased computation as flushed activation maps need to be recomputed whenever needed. Memory consumption can be reduced by decreasing the training batch size as well. However, this reduces parallelizability and increases the overall training time. Deep learning frameworks like PyTorch implement `torchvision.datasets.DatasetFolder` and `ImageFolder` to avoid loading entire training data to memory by reading samples from disk one batch size at a time. But these dataloaders adhere to a rigid format for specifying labels in the underlying directory structure. Thus, training a model on a particular dataset may require considerable preprocessing and designing custom dataloaders. For massive datasets, on-disk storage can quickly blow up too as seen previously.

C. Reducing communication cost

Algorithms like FedAvg and FedProx minimize communication overhead by choosing a low-frequency, high-volume communication strategy with occasional gradient synchronization. On the other hand, gradient compression either via sparsification, quantization or low-rank approximations uses a high-frequency, low-volume approach to reduce communication cost in DDL. Sparsification techniques like Topk-*k* [25] and Deep Gradient Compression (DGC) [28] apply sparse updates by sending only a subset of the gradients and setting remaining values to 0. The bit-width of floating-point gradients is reduced with quantization methods. Automatic mixed-precision (AMP) training described earlier uses half-precision gradients to achieve 2× compression. Another quantization method called Quantized SGD (QSGD) [29] quantizes gradients while balancing the trade-off between precision and accuracy. On the other hand, Terngrad [24] limits gradients across three quantization levels [-1,0,+1]. Low-rank approximations like PowerSGD [35] minimize update cost by performing low-rank updates that effectively work as regularization. All these compression techniques use a fixed compression ratio throughout training. Using a high compression ratio incurs higher communication cost, while a small compression ratio may trim too much useful information from the gradients. Accordian [31] dynamically compresses gradients by detecting critical regions in training by tracking gradient variance. We extend this further by developing an adaptive compression strategy by comparing entropy loss between the original and compressed gradients.

IV. SCADLES

We propose *ScaDLES* to address the challenges described in section II and accelerate DDL training on heterogeneous streams in both IID and non-IID settings.

Heterogeneous streams: The approaches described w.r.t federated training in section III consider either systems heterogeneity or statistical heterogeneity due to skewed and non-identical data. Training neural networks synchronously on multiple devices with heterogeneous data streams suffers from stragglers. A device i among n devices with the lowest streaming rate $S^{(i)} \in [S^{(1)}, S^{(2)}, ...S^{(n)}]$ can become a bottleneck depending on the mini-batch size since all other devices have to wait on i to gather enough training samples b_i and proceed an iteration. Additionally, streaming rates can vary at intra-device level at the edge too, depending on factors like battery level, time of day, usage, etc. This wait-time incurred due to streaming latency can thus slow down training.

To mitigate the impact of streams with lower inflows, we propose performing variable computation where we set $b_i \, \forall \, i \propto S^{(i)}$. Thus, we minimize streaming latency by setting device batch size to its streaming rate. As a result, some devices with high volume streams train on a large batch-size while the low volume devices use a smaller batch-size, and wait-times due to streaming latency are avoided. Since the amount of work done on each device is different, we perform weighted aggregation rather than a simple average to get the shared global updates. At iteration t, device i trains with batch-size corresponding to its streaming rate $S_i^{(t)}$ and scales the computed gradients $g_i^{(t)}$ by factor $r_i^{(t)}$ and updates the parameters as:

$$r_t^{(i)} = \frac{S_t^{(i)}}{\sum_{j=1}^{n} S_t^{(j)}} \quad : \sum_{j=1}^{n} r_t^{(j)} = 1.0 \tag{4a}$$

$$\tilde{g}_t = \sum_{j=1}^{n} r_t^{(j)} \cdot g_t^{(j)} \tag{4b}$$

$$w_{t+1} = w_t - \eta_{scaled} \cdot \tilde{g}_t \tag{4c}$$

The global batch-size with weighted gradients from Eqn. 4a is $\sum_{j=1}^{n} S^{(j)}$. As streaming rates can vary both inter and intra-device, so does the global batch-size. To ensure extremely high streaming rates don't increase the global batch-size so much that it degrades generalization performance, we add a *linear scaling rule* as suggested in [32], [33]. Essentially, linear scaling adjusts the learning rate in proportion to the batch-size, i.e., learning rate is increased if the batch-size increases, and vice versa. When the batch-size is multiplied by factor *k*, multiply the base learning rate by *k* as well. If the base global batch-size is B, then we scale the base learning rate as

$$\eta_{scaled} = \gamma_{scaled} \cdot \eta \quad : \gamma_{scaled} = \frac{\sum_{j=1}^{n} S_j}{B}$$

Even with a linear-scaling rule for training with larger batches, model quality still may suffer with extremely large batches in high volume streams. Likewise, using a batch-size too small is not efficient from parallelization perspective. Thus, we set $b_i = S^{(i)}$ as long as the device batch-size is bounded in the range $b_{min} \leq b_i \leq b_{max}$, else we use the corresponding min-max for training.

Limited memory and storage: The buffer size can grow quickly due to continuous data streams and considerable iteration times at the edge. The accumulated data can either reside in memory like a buffered queue, and reside on-disk to reduce memory footprint like in Kafka. By default, we could keep all the data streaming in and store it until processed successfully. We refer to this policy as *Stream Persistence*. As seen from Table II, accumulated samples keep increasing over the iterations depending on the stream-rate. Looking at Eqn. 2, buffer size grows to $\mathcal{O}(S^{(i)}T)$ after T iterations. Stream persistence makes sense especially when devices have sufficient memory or storage to hold the data, like in data-centers, cloud or high-capacity fog devices. In *Stream Truncation*, we discard the residual samples and hold just enough data corresponding to the device's streaming rate $S^{(i)}$. As a result, storage requirements for stream truncation is $\mathcal{O}(S^{(i)})$ at any given time, which is significantly smaller than stream persistence.

Unbalanced and Non-IID data: Fig. 2 demonstrates how model quality degrades when training with non-identical data. This happens since each device contains only a subset of training labels that are not representative of the entire distribution. Thus, parameters learned by device-local model replicas are skewed, and so is the aggregated model. To deal with non-IID data, we propose randomized ***data-injection*** where a fraction of the training devices share partial training samples with other devices. Particularly, at every iteration a device randomly chooses a subset α of the total devices D to share fraction β of its streaming data $\beta S(i) \in [\alpha D]$. Together, (α, β) determine what set of devices share how much of their training samples with other devices in DDL. Data injection helps improve the overall data distribution by making the device local data more representative of the complete dataset. However, this implies a trade-off between high model quality on account of better data distribution and privacy concern arising from moving data away from the devices. Privacy violation is greatly minimized by choosing only a subset of devices randomly (from α) and broadcasting only partial data (determined by β).

High communication cost: Federated algorithms reduce communication cost either with low-frequency, high-volume or high-frequency, low-volume communication strategy. We focus our efforts on the latter by looking at various gradient compression techniques. Rather than using a static compression ratio throughout training that can be detrimental to the final model accuracy, we look into adaptive compression. Prior work keeps track of the moving average of gradient variance to detect critical regions in training [30], [31], [34]. Gradients are large initially, but get smaller as the model evolves and training continues. Thus, we can use low compression in the beginning and higher compression later. We implement an adaptive compression strategy with Top-*k* sparsification which compares entropy loss between compressed and uncompressed gradients. Gradients compressed to top *k%* are used if the variance between compressed and uncompressed tensors falls below threshold δ; original, uncompressed tensors are communicated otherwise. The intuition is that if the top *k%* gradients have most of the information as the uncompressed gradients within the margin of δ, then remaining gradients are relatively less meaningful that don't greatly contribute towards model update and can thus be ignored. We track of the variance of compressed and uncompressed gradients at every iteration by keeping exponential weighted moving average (EWMA) and implement the communication rule for adaptive compression on gradients g for a device as follows:

$$\text{send}(\text{Top}k(g)) \;\; \text{if} \;\; \frac{||g|^2 - |\text{Top}k(g)|^2|}{|g|^2} \leq \delta \;\; \text{else} \;\; \text{send}(g)$$

Compression threshold, denoted by δ determines the degree of relaxation we impose on the compressed tensors to be eligible for communication. A small δ penalizes compressed tensors more severely and performs reduction only when the compressed data captures most of the relevant gradients in the original tensor. Constraints are loose with a larger δ which allows more iterations to use compressed tensors for synchronization.

TABLE III
NEURAL NETWORKS EVALUATED

Model	Parameters	Data	Devices	Labels/device
ResNet152	$60.2M$	IID Cifar10	16	10
		nonIID Cifar10	10	1
VGG19	$143.7M$	IID Cifar100	16	100
		nonIID Cifar100	25	4

V. Evaluation

A. Cluster setup

We simulate streaming data with Kafka by sampling streaming rates from uniform and normal distributions described in Table I. The hardware used to evaluate *ScaDLES* in our experiments comprises of 4 servers each with with 48-core Intel Xeon E5-2650, 128 GB system memory and 8 NVIDIA K80 GPUs connected with 5 Gbps ethernet. We mimic CUDA-aware edge devices by spawning them as `nvidia-docker` containers on CentOS linux 7.9.2009 with docker engine 20.10.17. Each device running as a container is allocated 4vCPUs, 12 GB system memory and 1 K80 GPU running NVIDIA driver 465.19.01 on CUDA 11.3 and PyTorch 1.10.1. We create a docker swarm network on the 5 Gbps network interface to facilitate communication for gradient synchronization among containers.

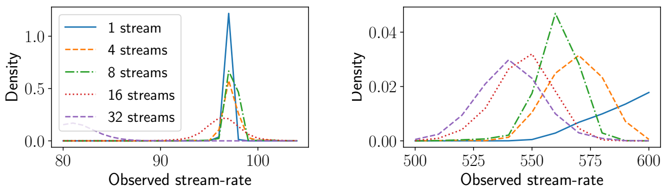
B. Data, models and hyperparameters

We evaluate two popular neural networks across different streaming distributions, training dataset and cluster configurations. ResNet152 uses SGD optimizer with momentum 0.9 and weight decay 0.0001 while adopting a learning rate schedule with initial lr 0.1 that decays by 0.2 after 75, 150 and 225 epochs. We also train VGG19 momentum SGD of 0.9 and weight decay 0.0005 with an intial lr 0.01 that decays by 0.3 after 75, 150 and 200 epochs. The model quality for both neural networks is measured by the *Top-5 test accuracy*.

The cluster setup for both IID and non-IID data is outlined in Table III. Training with IID data is performed on 16 devices where each device is equipped with a K80 GPU. We partition non-IID data by mapping a device to a unique subset of labels. Non-IID CIFAR10 is trained on 10 devices where each device contains only a single label. We train non-IID CIFAR100 on 25 devices such that each device is mapped to 4 unique labels.

C. Streaming data for DDL

We use Apache Kafka v3.1.0 to spawn a docker container that runs a broker as well as producers. The broker-producer container is allocated 16vCPUs, 32 GB system memory and *no* GPU since it doesn't participate in model training. We configure the container with 8 network threads, 4 IO-threads and 1 partition per topic. The sole purpose of this container is to host the Kafka broker and launch multiple producer processes such that each process publishes to a unique topic



(a) 100 samples/sec (b) 600 samples/sec

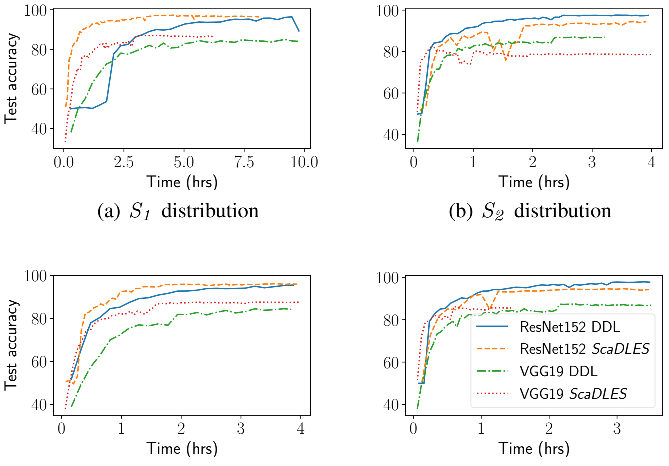
Fig. 5. Effective streaming rates achieved when scaling to multiple topics.

corresponding to a device. Thus, there are as many topics as the number of training devices. Each producer process controls the streaming rate corresponding to a device's topic. As for data consumption, the training devices have a kafka consumer running on them. The consumer implements a custom PyTorch dataloader that batches the data and integrates into a typical training loop that is common in deep learning training.

Since we run multiple producers from a single container (and not a separate container for every producer), we measure the effective streaming rate achieved with our proposed setup to ensure we meet the target stream-rates in our experiments. We scale up the number of concurrent producers (i.e., topics) and measure the observed streaming rate. For e.g., 32 streams in Fig. 5a imply 32 producers publishing to 32 topics at 100 samples/sec. Fig. 5 shows the density estimates of observed streaming rates with targets of 100 and 600 samples/sec. For each target rate, we scale up the number of concurrent producers to $1, 4, 8, 16$ and 32. We achieve nearly the same target of 100 samples/sec as shown in Fig. 5a. For the 600 samples/sec target, the effective streaming rate decreases noticeably beyond 16 concurrent streams. We could likely improve this by increasing the number of network threads and partitions per topic, but this setup sufficed for the evaluations we perform in this paper.

D. Weighted aggregation in heterogeneous streams

We use a batch-size corresponding to a devices' streaming rate in *ScaDLES* to avoid wait-times on account of possible streaming latency. To enforce bounds on the batch-size used, we set b_{min} and b_{max} to 8 and 1024 respectively, although stream-rate for any device remains within this range regardless of the streaming distribution. The waiting time can especially be long in highly heterogeneous streams when a devices' streaming rate is lower than the mini-batch size configured prior training. There is no waiting time high-volume streams with low-batch size settings. However, the buffer size can grow quickly over time in that case. We compare *ScaDLES* with conventional DDL training for batch-size 64 irrespective of the device streaming rates. We look at the convergence curves and buffer buildup over training epochs to compare the two. The streaming rates for the 16 devices were sampled from the distributions outlined in section II. Uniform distributions are more heterogeneous compared to normal distributions (2/3rd values lie within 1 standard deviation from the mean in the latter).



(a) S_1 distribution (b) S_2 distribution

(c) S_1' distribution (d) S_2' distribution

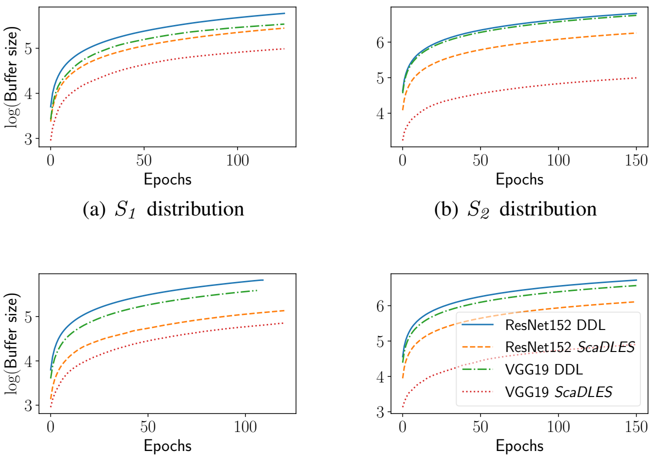
Fig. 6. Convergence in conventional DDL vs. weighted aggregation approach in *ScaDLES*.

Fig. 6a shows results for *ScaDLES* and conventional DDL using device stream-rates sampled from S_1 that converges 3.33× and 1.92× faster in *ScaDLES*. Conventional DDL converged with higher final accuracy in S_2 due to large batches used for training by *ScaDLES* with this distribution; about 4.5K in *ScaDLES* compared to only 1K in DDL. Linearly scaling the learning rate at batches this large did not significantly improve *ScaDLES*' generalization performance. Devices sampled from S_1' achieved around 3.6× and 4× speedup with *ScaDLES* while still achieving higher final accuracy. Using larger batches and linear scaling proved to be beneficial in this case. Lastly, Fig. 6d uses S_2' distribution where ResNet152 performs similarly for both *ScaDLES* and DDL, while VGG19 performs better with our approach.

E. Managing limited memory and storage

We first look at how streaming data gets accumulated in a device with the default persistence policy. For the same runs described in the previous section, we plot how samples get accumulated over the iterations for different sampling distributions in Fig. 7. We plot logarithm of the accumulated samples with base 10. The buffer size is smaller in *ScaDLES* compared to DDL training for the same persistence policy. This is because *ScaDLES* uses batch-size $S^{(i)}$ while we use a smaller batch-size 64 in conventional DDL. S_2 and S_2' have larger buffer sizes since they represent higher volume streams compared to S_1 and S_1'. DDL occupies 2× and 3.5× more space with ResNet152 and VGG19 in S_1. *ScaDLES* holds 3.6× and 641× less data than DDL for S_2. Comparing Fig. 6b and Fig. 7b, we see the lower buffer size in *ScaDLES* came at the cost of lower final accuracy due to large-batch training. *ScaDLES* has 4.7× and 5× smaller buffer in S_1', and 3.9× and 42× lesser data with S_2' distribution.

Although *ScaDLES* accumulates lesser samples than conventional DDL, we can further lower the buffer size for continuous data streams. With stream truncation, data in



(a) S_1 distribution

(b) S_2 distribution

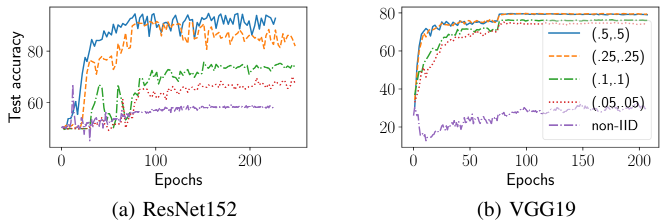
(c) S_1' distribution

(d) S_2' distribution

Fig. 7. Buffer size increases with training iterations

TABLE IV
BUFFER-SIZE REDUCTION WITH TRUNCATION POLICY

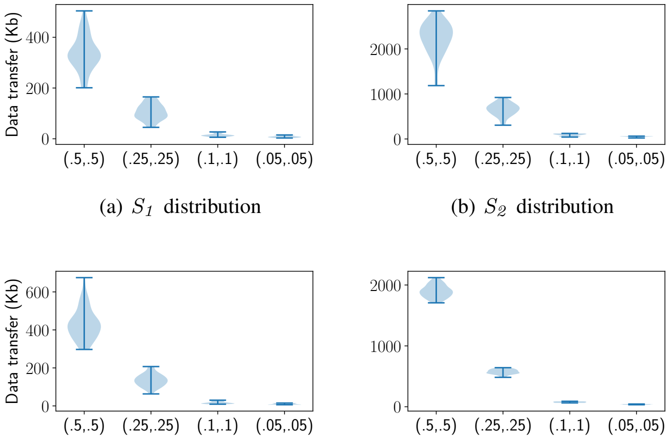
Dist.	Model	Persistence	Truncation	Reduction
S_1	ResNet152	2.9×10^5	129	2238×
	VGG19	1×10^5	118	848×
S_2	ResNet152	4.36×10^6	633	6889×
	VGG19	4×10^6	523	7830×
S_1'	ResNet152	6.2×10^5	143	4340×
	VGG19	3.7×10^5	129	2861×
S_2'	ResNet152	3.6×10^6	384	9429×
	VGG19	2.5×10^6	360	6956×



(a) ResNet152

(b) VGG19

Fig. 8. Different (α, β) configurations for data-injection in non-IID training.



(a) S_1 distribution

(b) S_2 distribution

(c) S_1' distribution

(d) S_2' distribution

Fig. 9. Data transfer overhead at each iteration to handle non-IID data with data-injection.

buffer exceeding the samples that just streamed in is simply discarded. The buffer size with truncation policy is constant as long as the streaming rate is continuous. On the other hand, persistence policy grows with each passing iteration. Table IV shows the final buffer size to reach 95% and 84% accuracy on ResNet152 and VGG19. The table also reports reduction with truncation relative to persistence policy. We observed buffer reductions from 850× to 9400× depending on the distribution.

F. Data-injection for non-IID and skewed data

Data-injection helps improve overall model quality when dealing with non-IID data. The degree of data-injection is determined by (α, β) parameters that determine the subset of devices to send partial data. We evaluate four (α, β) sets in *ScaDLES*: $(0.5, 0.5)$, $(0.25, 0.25)$, $(0.1, 0.1)$ and $(0.05, 0.05)$. A value of $(0.5, 0.5)$ means half of the devices share half of the samples in their current batch. We plot the convergence curves for different streaming distributions in Fig. 8 and note significantly better performance than training merely with non-IID data.

Some additional networking cost is associated with data-injection as a subset of devices send partial data to other devices. For CIFAR10 and CIFAR100 datasets, each sample is an image 3 Kilobytes in size. For different streaming distributions and (α, β) parameters, we look at data exchange among the devices in Fig. 9. The overhead is minimal and ranges anywhere from 150 to 2000 kilobytes on average for each training iteration.

G. Adaptive compression

We look at reduction in the overall communication volume, i.e., cumulative single-precision floats communicated achieving target accuracy to evaluate the performance of adaptive compression in *ScaDLES*. **Compression ratio (CR)** measures the degree of compression by comparing the tensor size of compressed gradients to that of the original tensors. CR of 0.1 means compressed tensors are $1/10$-th the size of uncompressed gradients. The update size of ResNet152 (230 MB) and VGG19 (548 MB) at this CR reduces to just 23 and 55 MB respectively.

Using the adaptive compression rule described in section IV, we measure the usage of compressed gradients for communication with **Compression-to-No-Compression (CNC) ratio**. CNC ratio compares iterations using compressed gradients for communication to the total iterations used throughout training. The latter includes the iterations that use compression as well iterations that use the original, uncompressed gradients for

TABLE V
COMMUNICATION REDUCTION IN ADAPTIVE COMPRESSION

Model	CR	δ	CNC ratio	Accuracy	Floats sent
ResNet152	0.1	0.1	0.29	97.55%	4.43×10^{11}
		0.2	0.99	96.81%	0.56×10^{11}
		0.3	1.0	98.41%	0.4×10^{11}
		0.4	1.0	98.57%	0.4×10^{11}
	0.01	0.1	0	97.39%	6.02×10^{11}
		0.2	0.17	97.47%	4.99×10^{11}
		0.3	0.43	96.72%	2.56×10^{11}
		0.4	0.99	94.97%	6.32×10^{8}
VGG19	0.1	0.1	0	85.45%	1.3×10^{12}
		0.2	0.08	84.74%	1.19×10^{12}
		0.3	1.0	81.91%	1.3×10^{10}
		0.04	1.0	81.78%	1.3×10^{10}
	0.01	0.1	0	84.68%	1.3×10^{12}
		0.2	0	83.98%	1.3×10^{12}
		0.3	0	83.94%	1.3×10^{12}
		0.4	0.004	84.39%	1.29×10^{12}

communication:

$$\text{CNC ratio} = \frac{T_{compressed}}{T_{compressed} + T_{uncompressed}}$$

CNC ratio of 0 means that compressed tensors *were not* used even for a single iteration, while CNC of 1.0 implies all training iterations used *only* the compressed tensors for exchange since there are no iterations that used the original gradients for communication. To measure the impact of adaptive compression on model convergence, we tabulate the CNC ratio, accuracy and overall reduction in communication volume for different (CR, δ) configurations in Table V. ResNet152 running with CR 0.1 and any δ beyond 0.2 is faster as it converges by exchanging fewer floats. A δ of 0.1 barely used any compression for the two CRs in either of the models. The most communication efficient configuration for ResNet152 used CR 0.01 and δ 0.4, although it results in slightly lower test accuracy. The pattern of low communication overhead accompanied with degradation in final accuracy was also observed in VGG19 for CR 0.1 and δ 0.4. The CNC ratio is high in both cases implying compression is enabled for most iterations and thus, accuracy drop can be attributed to model degradation commonly associated with compression. An interesting observation in VGG19 using CR 0.01 is the communication volume is same across all δ values and the total floats exchanged is the same as training without any compression. This means the adaptive strategy is not using compression in this configuration, which is further corroborated by the CNC ratio that is 0 across all δ.

H. Overall performance of ScaDLES

Last, we look at the overall performance gains in *ScaDLES* by combining weighted aggregation in heterogeneous streams, data-injection for non-IID data, buffer reduction with stream truncation and reducing communication with adaptive compression (using CR 0.1 and δ of 0.3 in our final evaluation). We compare against conventional DDL with fixed batch-size 64, persistence policy and the same training schedule as *ScaDLES* described in section V.

TABLE VI
SCADLES' PERFORMANCE GAINS OVER CONVENTIONAL DDL

Model	Dist.	Acc. drop	Buffer red. (GB)	Speedup
ResNet152	S_1	-0.06%	0.6	$\mathbf{1.89}\times$
	S_2	-0.32%	5.9	$\mathbf{1.15}\times$
	S_1'	-0.13%	0.8	$\mathbf{3.29}\times$
	S_2'	-0.21%	4.03	$\mathbf{1.42}\times$
VGG19	S_1	-1.93%	0.26	$\mathbf{1.56}\times$
	S_2	-4.18%	3.91	$\mathbf{2.83}\times$
	S_1'	-2.03%	0.35	$\mathbf{2.06}\times$
	S_2'	-1.59%	2.58	$\mathbf{2.13}\times$

For the same streaming distribution, *ScaDLES'* performance is measured relative to conventional DDL in terms of drop in test accuracy, reduction in buffer size using truncation policy (in Gigabytes) and overall training speedup to convergence w.r.t wall-clock time. A negative accuracy drop means *ScaDLES* achieved lower accuracy by that margin. Table VI shows the results for IID training. ResNet152 trains on *ScaDLES* with a maximum of drop of 0.32% in final model accuracy compared to conventional DDL with stream-rates sampled from S_2. Training with *ScaDLES* is also much faster; ranging from $1.15\times$ to $3.29\times$ faster than DDL. For high-volume streams S_2 and S_2', truncation policy saves up to 5.9 GB in the occupied buffer-size. *ScaDLES* achieved lower final accuracy in VGG19 by as much as 4.18%. We observed that VGG19 is more sensitive to the combination of *ScaDLES*' large-batch training and adaptive compression than ResNet152. However, VGG19 still reduces buffer-size by up to 3.91 GB and reduces wall-clock training time by $1.56\times$ to $2.83\times$ over conventional DDL.

As for training with non-IID data, conventional DDL was unable to reach the same convergence targets as *ScaDLES*' data-injection strategy for either of the neural networks. Conventional DDL with non-IID data saturated ResNet152 at 56% test accuracy, while VGG19 did not improve beyond 35%. For the same non-IID training, *ScaDLES* achieved at least 93.6% and 77.8% accuracy for ResNet152 and VGG19 across the four stream-rate distributions.

VI. CONCLUSION

This paper presents the notion of training neural networks efficiently over streaming data at the edge. Streaming data presents challenges affecting both parallel and statistical efficiency of distributed training. *ScaDLES* addresses the problem of heterogeneous streaming rate among devices with weighted aggregation where each device trains on the samples accumulated in the stream and avoids wait-time or large buffer accumulation. Since it is difficult to perform training at line-rate, samples streaming into the device can accumulate over time. *ScaDLES* uses a simplistic truncation policy to keep the buffer size in check. Devices on the edge commonly have non-IID and unbalanced data. Data-injection strategy improves model convergence significantly in such scenarios. Lastly, we propose an adaptive compression technique to deal with limited bandwidth on the edge and high communication cost in large deep learning models. We simulate different degrees of streaming heterogeneity by sampling from both uniform and normal distributions and evaluate popular image classifiers. Our empirical evaluation shows that *ScaDLES* can converge anywhere from $1.15\times$ to $3.29\times$ faster than DDL. At the same time, *ScaDLES* reduces the number of accumulated samples in the buffer by $848\times$ to $9429\times$.

References

[1] J. Kreps, N. Narkhede and J. Rao, Kafka: a Distributed Messaging System for Log Processing.

[2] Zhen Z., Chaokun C., Haibin L., Yida W., Raman A., and Xin J. Is Network the Bottleneck of Distributed Training? In Proceedings of the Workshop on Network Meets AI and ML (NetAI '20).

[3] Achille A., Rovere M., Soatto S. (2017), Critical Learning Periods in Deep Neural Networks. ArXiv, abs/1711.08856.

[4] Dan A., Torsten H., Mikael J., Sarit K., Nikola K., and Cédric R.. 2018. The convergence of sparsified gradient methods. In Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18).

[5] Adam P., Sam G., Francisco M., Adam L., James B., Gregory C., Trevor K., Zeming L., Natalia G., Luca A., Alban D., Andreas K., Edward Y., Zach D., Martin R., Alykhan T., Sasank C., Benoit S., Lu Fang, Junjie B., and Soumith C.. 2019. PyTorch: an imperative style, high-performance deep learning library. Proceedings of the 33rd International Conference on Neural Information Processing Systems.

[6] Shen L., Yanli Z., Rohan V., Omkar S., Pieter N., Teng L., Adam P., Jeff S., Brian V., Pritam D., and Soumith C.. 2020. PyTorch distributed: experiences on accelerating data parallel training. Proc. VLDB Endow.

[7] Li M., Andersen D.G., Park J.W., Smola A., Ahmed A., Josifovski V., Long J., Shekita E.J. and Su B. (2014). Scaling Distributed Machine Learning with the Parameter Server. BigDataScience '14.

[8] Gabriel E., Fagg G.E., Bosilca G., Angskun T., Dongarra J.J., Squyres J.M., Sahay V., Kambadur P., Barrett B.W., Lumsdaine A., Castain R.H., Daniel D.J., Graham R.L., and Woodall T.S. (2004). Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. PVM/MPI.

[9] NCCL: NVIDIA Collective Communications Library. Available here: https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/index.html.

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

[11] He K., Zhang X., Ren S. and Sun J. Deep Residual Learning for Image Recognition. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 770-778.

[12] Simonyan K. and Zisserman A. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition. CoRR, abs/1409.1556.

[13] CIFAR-10 and CIFAR-100: https://www.cs.toronto.edu/ kriz/cifar.html.

[14] Nesterov Y. (1983). A method for solving the convex programming problem with convergence rate $O(1/k^2)$. Proceedings of the USSR Academy of Sciences, 269, 543-547.

[15] Kingma D.P., and Ba J. (2015). Adam: A Method for Stochastic Optimization. CoRR, abs/1412.6980.

[16] Vaswani A., Shazeer N.M., Parmar N., Uszkoreit J., Jones L., Gomez A.N., Kaiser L. and Polosukhin I. (2017). Attention is All you Need. ArXiv, abs/1706.03762.

[17] Zhao Y., Li M., Lai L., Suda N., Civin D. and Chandra V. (2018). Federated Learning with Non-IID Data. ArXiv, abs/1806.00582.

[18] Sattler F., Wiedemann S., Müller K. and Samek W. (2020). Robust and Communication-Efficient Federated Learning From Non-i.i.d. Data. IEEE Transactions on Neural Networks and Learning Systems, 31, 3400-3413.

[19] McMahan H.B., Moore E., Ramage D., Hampson S. and Arcas B.A. (2017). Communication-Efficient Learning of Deep Networks from Decentralized Data. AISTATS.

[20] He C., Li S., So J., Zhang M., Wang H., Wang X., Vepakomma P., Singh A., Qiu H., Shen L., Zhao P., Kang Y., Liu Y., Raskar R., Yang Q., Annavaram M. and Avestimehr S. (2020). FedML: A Research Library and Benchmark for Federated Machine Learning. ArXiv, abs/2007.13518.

[21] Caldas S., Wu P., Li T., Konecný J., McMahan H.B., Smith V. and Talwalkar A.S. (2018). LEAF: A Benchmark for Federated Settings. ArXiv, abs/1812.01097.

[22] Wang H., Kaplan Z., Niu D. and Li B. (2020). Optimizing Federated Learning on Non-IID Data with Reinforcement Learning. IEEE INFOCOM 2020 - IEEE Conference on Computer Communications, 1698-1707.

[23] Sahu A., Li T., Sanjabi M., Zaheer M., Talwalkar A.S. and Smith V. (2020). Federated Optimization in Heterogeneous Networks. arXiv: Learning.

[24] Wen W., Xu C., Yan F., Wu C., Wang Y., Chen Y. and Li H.H. (2017). TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning. ArXiv, abs/1705.07878.

[25] Alistarh D., Hoefler T., Johansson M., Khirirat S., Konstantinov N. Renggli C. (2018). The Convergence of Sparsified Gradient Methods. NeurIPS.

[26] Micikevicius P., Narang S., Alben J., Diamos G.F., Elsen E., García D., Ginsburg B., Houston M., Kuchaiev O., Venkatesh G. and Wu H. (2018). Mixed Precision Training. ArXiv, abs/1710.03740.

[27] Griewank A. and Walther A. (2000). Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. ACM Trans. Math. Softw., 26, 19-45.

[28] Lin Y., Han S., Mao H., Wang Y. and Dally W.J. (2018). Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. ArXiv, abs/1712.01887.

[29] Alistarh D., Grubic D., Li J., Tomioka R. and Vojnovic M. (2016). QSGD: Communication-Optimal Stochastic Gradient Descent, with Applications to Training Neural Networks.

[30] Achille A., Rovere M. and Soatto S. (2017). Critical Learning Periods in Deep Neural Networks. ArXiv, abs/1711.08856.

[31] Agarwal S., Wang H., Lee K., Venkataraman S. and Papailiopoulos D. (2021). Accordion: Adaptive Gradient Communication via Critical Learning Regime Identification. ArXiv, abs/2010.16248.

[32] Goyal P., Dollár P., Girshick R.B., Noordhuis P., Wesolowski L., Kyrola A., Tulloch A., Jia Y. and He K. (2017). Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. ArXiv, abs/1706.02677.

[33] Smith S.L., Kindermans P. and Le Q.V. (2018). Don't Decay the Learning Rate, Increase the Batch Size. ArXiv, abs/1711.00489.

[34] Frankle J., Schwab D.J. and Morcos A.S. (2020). The Early Phase of Neural Network Training. ArXiv, abs/2002.10365.

[35] Vogels T., Karimireddy S.P. and Jaggi M. (2019). PowerSGD: Practical Low-Rank Gradient Compression for Distributed Optimization. NeurIPS.