Mercury: Efficient On-Device Distributed DNN Training via Stochastic Importance Sampling

Xiao Zeng, Ming Yan, Mi Zhang
Michigan State University

ABSTRACT

As intelligence is moving from data centers to the edges, intelligent edge devices such as smartphones, drones, robots, and smart IoT devices are equipped with the capability to altogether train a deep learning model on the devices from the data collected by themselves. Despite its considerable value, the key bottleneck of making on-device distributed training practically useful in real-world deployments is that they consume a significant amount of training time under wireless networks with constrained bandwidth. To tackle this critical bottleneck, we present Mercury, an importance sampling-based framework that enhances the training efficiency of on-device distributed training without compromising the accuracies of the trained models. The key idea behind the design of Mercury is to focus on samples that provide more important information in each training iteration. In doing this, the training efficiency of each iteration is improved. As such, the total number of iterations can be considerably reduced so as to speed up the overall training process. We implemented Mercury and deployed it on a self-developed testbed. We demonstrate its effectiveness and show that Mercury consistently outperforms two status quo frameworks on six commonly used datasets across tasks in image classification, speech recognition, and natural language processing.

CCS CONCEPTS

• **Computing methodologies → Distributed artificial intelligence**.

KEYWORDS

Machine Learning Systems, On-Device Distributed Training, Edge Computing, AIoT, Importance Sampling

ACM Reference Format:
Xiao Zeng, Ming Yan, Mi Zhang. 2021. Mercury: Efficient On-Device Distributed DNN Training via Stochastic Importance Sampling. In *The 19th ACM Conference on Embedded Networked Sensor Systems (SenSys'21), November 15–17, 2021, Coimbra, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3485730.3485930

1 INTRODUCTION

The recent past few years have witnessed the success of deep learning (DL) in a wide spectrum of areas including computer vision [6], speech recognition [15], and natural language processing [34]. Part of this success is attributed to the highly efficient distributed training frameworks [42, 44] that train deep neural networks (DNN) in parallel machines inside data centers owned by large organizations such as Facebook, Google, Amazon, and Microsoft.

As AI chipsets become pervasive, machine intelligence is moving from the data centers to the edges [45]. Edge devices such as smartphones, drones, robots, autonomous vehicles, and smart IoT devices at homes powered by DL models are able to not only perform on-device inferences for a variety of tasks [8, 9, 20, 43] but also altogether train a DL model locally on the devices from the data collected by themselves in a distributed manner.

There are a wide range of real-world applications that are built upon on-device distributed training, where it could take far too long for an individual edge device to learn by experiencing all the scenario by itself but would quickly acquire new knowledge from collectively learning from each other. For example, smart cameras deployed at different locations of the smart home can collaboratively train DL models to detect dangerous individuals or events; a swarm of drones can collaboratively learn to recognize important terrain or city landmarks for navigation; and a team of robots can collaboratively learn to assist human workers in a warehouse or factory by recognizing daily objects or voice commands.

Status Quo and their Limitations. Despite its considerable value, the key bottleneck of making on-device distributed training practically useful in real-world deployments is that they consume a significant amount of training time. The root cause of such performance bottleneck is the limited wireless network bandwidth: in data centers, communication between parallel machines is conducted via high-bandwidth network such as 50 Gbps Ethernet or 100 Gbps InfiniBand [39]. In contrast, in on-device distributed training, communication between participating edge devices is conducted via wireless network such as Wi-Fi. The bandwidth of wireless network, however, can be much more constrained than the bandwidth in data centers. Such limited bandwidth considerably slows down the communication and hence the overall training process.

The objective of this work is to tackle this critical bottleneck to *enhance the training efficiency of on-device distributed training while retaining training correctness without compromising the training quality* (i.e., accuracies of the trained models). Although a number of training acceleration methods have been proposed [2, 12, 17, 19, 27, 28], as we will discuss in §2.2, these methods either compromise the training quality to gain training efficiency or are designed for distributed training in data center setting where they achieve limited training efficiency enhancement in on-device setting given the significant gap in network bandwidth between the two settings.

Overview of the Proposed Approach. The limitations of existing methods motivate us to rethink the on-device distributed training framework design by taking a path that is different from existing ones. To this end, we present Mercury, an importance sampling-based framework that enables efficient on-device distributed training without compromising the training quality. In existing approaches, each participating device in every training iteration *randomly* samples a mini-batch from local data to compute its local gradients. This random sampling strategy, at its core, assumes that each sample in the local data is equally important to model training. However, *not all the samples contribute equally to model training*. Training on less important samples not only has limited contribution to model accuracy but also slows down the training process. Given that, instead of sampling data in a random manner, Mercury focuses on samples that provide more important information in each iteration. As such, the training efficiency of each iteration is improved, and the total number of iterations can be reduced so as to speed up the overall training process.

The design of Mercury involves three key challenges.

- **Challenge#1**: Importance sampling incurs computation cost. and without careful design, could easily overshadow the benefit it brings. Reducing the computation cost of importance sampling without affecting its effectiveness and compromising the training quality represents a significant challenge.
- **Challenge#2**: Since importance computation is performed on the local data of each device, the importance only reflects the local importance distribution within each device rather than the global importance distribution across all the distributed devices. As a result, a device may repeatedly learn globally trivial samples, which lowers the training efficiency.
- **Challenge#3**: Even though the computation cost of importance sampling can be reduced, the cost is still not zero. Moreover, compared to Ethernet or InfiniBand used in the data center setting, wireless network in the on-device setting is more susceptible to interference and thus its bandwidth could experience variations over time in real-world deployments. The design of Mercury should take bandwidth variation into account as well.

To address the first challenge, Mercury incorporates a *group-wise importance computation and sampling technique* that cuts the computation cost of importance sampling by only re-computing the importance of a subset of samples within each iteration. Moreover, by constructing the mini-batch based on the re-computed importance in a stochastic manner, the training correctness is guaranteed.

To address the second challenge, Mercury incorporates an *importance-aware data resharding technique* that efficiently reshuffles the data among devices to balance the importance distribution. It achieves the same effect with much lower overhead compared to importance-agnostic data resharding by prioritizing the transfer of more important samples.

To address the third challenge, Mercury incorporates a *bandwidth-adaptive computation-communication scheduler* which schedules the execution of importance computation and data resharding in a bandwidth-adaptive manner to further improve the training speedup by completely masking out the costs of importance sampling and data resharding.

System Implementation and Evaluation Results. We have implemented Mercury using TensorFlow and deployed it on a self-developed testbed that consists of 12 NVIDIA Jetson TX1 as edge devices. We evaluate Mercury on six commonly used datasets using a diverse set of DL models across three important AI tasks including image classification, speech recognition, and natural language processing. We compare Mercury against two status quo baselines TicTac [12] and AdaComm [36]. Our results show that Mercury consistently outperforms the baselines in training efficiency, achieving 3.74×, 3.21×, 4.08×, 2.21×, 2.74×, and 1.85× training speedup on the six datasets respectively. Moreover, Mercury is robust to wireless bandwidth variations in real-world deployments, and is able to maintain training speedup as the number of participating devices scales up. Finally, federated learning can be regarded as a constrained case of on-device distributed training where data in each edge device cannot be exchanged with each other. Our results show that Mercury is able to enhance the training efficiency of federated learning under diverse non-IID distributions.

In summary, our work makes three major contributions:

- To the best of our knowledge, Mercury represents the first on-device distributed training framework based on stochastic importance sampling which achieves high training speedup while retaining training correctness without compromising the accuracies of the trained models.
- We provide a performance model of the proposed importance sampling-based on-device distributed training framework. Guided by the performance model, we propose three novel techniques: group-wise importance computation and sampling, importance-aware data resharding, and bandwidth-adaptive computation-communication scheduling, which altogether fully exploit the benefits brought by importance sampling. We also provide a theoretical proof on the training correctness.
- We implemented Mercury and deployed it on a self-developed testbed. We demonstrate its effectiveness and show that Mercury consistently outperforms two status quo frameworks on six datasets across three important AI tasks. We believe our work represents a significant step towards making on-device distributed training practically useful.

2 BACKGROUND AND MOTIVATION

In this section, we first provide a brief background on distributed DNN training (§2.1). We then discuss the existing approaches for enhancing the training efficiency and their limitations in the context of on-device setting, which is the key motivation of this work (§2.2).

2.1 Distributed DNN Training

Distributed training uses Stochastic Gradient Descent (SGD) or its variants to train the DL model in an iterative manner. In each SGD iteration, each client randomly samples a mini-batch from its local data to compute its local gradients. These local gradients are then aggregated from the distributed clients to update the model parameters $\mathbf{w}$ as follows:

$$\mathbf{w}^{t+1} = \mathbf{w}^{t} - \eta_t \frac{1}{K} \sum_{k=1}^{K} \mathbf{g}^{k} \tag{1}$$

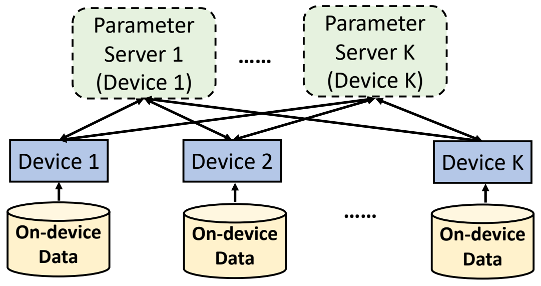


Figure 1: The multi-parameter server (multi-PS) architecture adopted by Mercury for on-device distributed training.

where $\mathbf{g}^k = \frac{1}{|\mathcal{B}_k|}\sum_{i\in\mathcal{B}_k}\nabla l(x_i, \mathbf{w}^t)$ are the gradients at client k and $\mathcal{B}_k$ is the mini-batch of client k.

Modern distributed training systems use parameter server (PS) architecture for gradient aggregation. In PS [25, 38, 44], one or more machines play the role of servers to aggregate computed gradients from worker machines, update the model, and send the updated model or aggregated gradients back to all workers.

As shown in Figure 1, Mercury adopts a multi-PS architecture and treats each edge device as a parameter server too. By doing this, the communication is not bottlenecked at a single device given the low network bandwidth in the on-device setting.

2.2 Existing Approaches and Their Limitations

The total training time of distributed training T can be generally estimated as the total number of training iterations until convergence E multiplies the sum of the time consumed by local computation T_{cp} (e.g., compute the gradients of the model parameters) and the communication time consumed by transmitting the model gradients between the parameter server and the client T_{cm} in each iteration as follows:

$$T = E \cdot (T_{cp} + T_{cm}). \tag{2}$$

To reduce the total training time T, existing approaches can be in general grouped into the following three categories.

Approach#1: Gradient Compression. To reduce T, one common approach is to reduce the communication time in each iteration T_{cm} in Eq. (2) via gradient compression. This can be achieved by quantizing gradients using smaller number of bits [2, 27, 39] or selecting important gradients to transfer via sparsification [17, 28, 37]. However, communication in on-device setting is conducted through wireless networks whose bandwidth is much more constrained than Ethernet or InfiniBand used in data centers. Given such limited bandwidth, the contribution of gradient compression to reducing T_{cm} in Eq. (2) is limited. Although many works adopt aggressive gradient compression that is able to push the limit, they gain training efficiency by compromising the training quality, which sacrifices the accuracy of the trained model.

Approach#2: Local SGD. In distributed SGD, global synchronization among clients is performed in each iteration for aggregating gradients. Such synchronization, however, is expensive in terms of communication under bandwidth-constrained networks. To reduce the communication overhead, the approach named local SGD [7, 29, 36, 46] has been proposed where the idea is to allow each client to perform multiple steps of SGD before each round of global synchronization. Similar to gradient compression, although aggressively performing local SGD enhances training efficiency by reducing the rounds of communication, it also suffers from compromising the accuracy of the trained model.

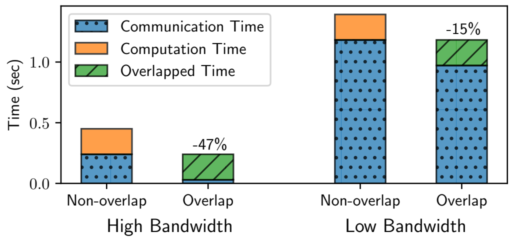


Figure 2: Communication-computation overlapping in high bandwidth and low bandwidth settings.

Approach#3: Communication-Computation Overlapping. DL models in general can be represented as directed acyclic graphs (DAGs). The structures of DAGs provide an opportunity to overlap gradient computation and gradient aggregation in a pipelined fashion to mask out the communication cost. This overlapping technique can be seen as a mechanism to reduce the sum $(T_{cp} + T_{cm})$ in Eq. (2). Such reduction can be achieved by either identifying the optimal gradient transfer order [5, 12, 19, 44] or using staled model weights [26]. However, the limited bandwidth in on-device setting makes the communication-to-computation ratio much higher than the one in data center setting. As a result, the effectiveness of such overlapping technique is significantly diminished since it is no longer able to mask out the communication cost under such high communication-to-computation ratio. To demonstrate this, Figure 2 shows the cost breakdown of communication-computation overlapping technique. As shown, although it provides significant cost reduction (47%) under high bandwidth network in the data center setting, the cost reduction is reduced to 15% in low bandwidth network in the on-device setting.

3 MERCURY OVERVIEW

The limitations of existing approaches motivate us to design Mercury by taking a different path. In this section, we first introduce the underlying principle behind the design of Mercury (§3.1). We then introduce a rudimentary framework designed upon the principle and its performance model (§3.2). Lastly, we briefly overview the techniques involved in Mercury which transform the rudimentary framework into a highly efficient one for on-device distributed training (§3.3).

3.1 Design Principle

The underlying principle behind the design of Mercury is to exploit *data efficiency* to improve the efficiency of on-device distributed training. In standard distributed training, in each SGD iteration, each device *randomly* samples a mini-batch from its local data to compute its local gradients. This random sampling strategy, at its

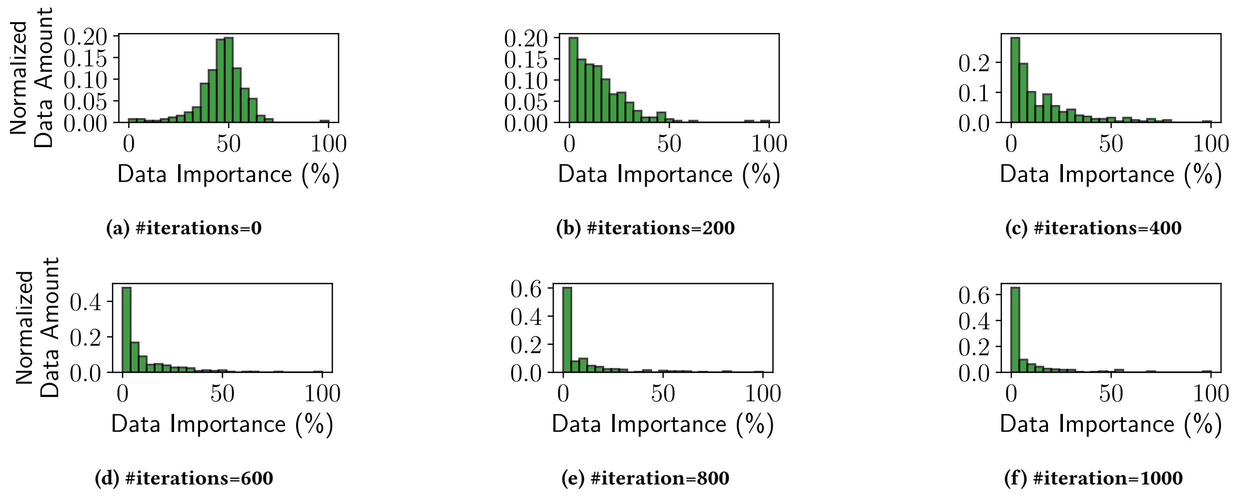


Figure 3: Distributions of data importance as the number of iterations increases during training.

core, assumes that each data sample in the local data is equally important during the overall model training process. In practice, however, redundant information across data samples exist, and *not all the data samples contribute equally to model training*: some of them provide more information to the training and therefore are more important, while the others provide less information and are thus less important. As an example, Figure 3 illustrates the importance distribution of the data in CIFAR-10 as the number of iterations increases during training. As shown, as the training process proceeds, more data samples become less important. In particular, after 200 iterations, the importance distribution quickly concentrates to a small portion of data samples and the rest have limited contribution to the model training. This observation sheds light on the low training efficiency of standard distributed training where mini-batch is generated by random sampling.

Based on this insight, we propose to incorporate *importance sampling* [48] to improve the *training efficiency of each SGD iteration.* The key idea of importance sampling is that in each training iteration, it focuses on samples that provide more important information and contribute more to the model training.

Formally, let x denote a data sample in the mini-batch, p denote the uniform distribution adopted by random sampling, and q denote a new distribution adopted by importance sampling. To ensure training correctness, the gradient $\nabla l(x)$ (here we leave model weights $\mathbf{w}^t$ out for simplicity) should be multiplied by $p(x)/q(x)$ when shifting from random sampling to importance sampling [1]:

$$\mathbb{E}_{x\sim q}\left[\frac{p(x)}{q(x)}\nabla l(x)\right] = \mathbb{E}_{x\sim p}\left[\nabla l(x)\right] \tag{3}$$

if $q(x) > 0$ whenever $p(x) > 0$.

To ensure gradient variance reduction to accelerate the training process, $q(x)$ should be proportional to the sample's gradient norm ($||\nabla l(x)||$) [1]:

$$q(x) \propto ||\nabla l(x)|| \tag{4}$$

The above derivation implies that the gradient norm of the data sample can be used as an indicator of its importance. In practice, the feed-forward loss is often used as a computation-efficient alternative to gradient norm [21]. We thus use feed-forward loss as the metric to measure the importance of each data sample.

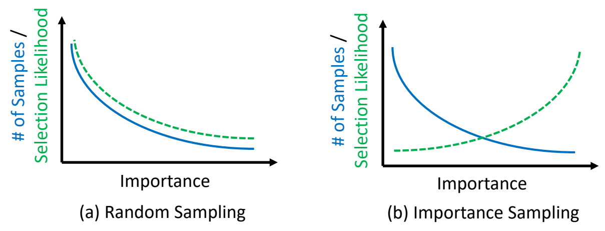


Figure 4: Comparison between (a) random sampling and (b) importance sampling.

Figure 4 provides a conceptual comparison between random sampling and importance sampling. The blue solid curve depicts the distribution of the importance values of data samples. In random sampling, each sample has the equal probability to be selected regardless of its importance. As such, if there are more samples with lower importance in a dataset, samples with lower importance have higher likelihood to be selected (green dotted curve in Figure 4a). In contrast, in importance sampling, samples with higher importance have higher likelihood to be selected (green dotted curve in Figure 4b). As such, the mini-batch generated by importance sampling carries more important information than random sampling. Such advantage enhances the training efficiency within each training iteration, which in turn reduces the total number of iterations and speeds up the overall training process.

3.2 Performance Model

Inspired by the benefits brought by importance sampling, Figure 5a illustrates a rudimentary importance sampling-based framework for on-device distributed training. As shown, the framework is built upon the standard distributed SGD. In each SGD iteration, the importance sampling-based framework replaces *random sampling* (i.e., randomly select samples to construct the mini-batch) in the standard distributed training with two new operations: 1) *importance computation* which computes the importance (i.e., gradient norm) for every sample in the local dataset, and 2) *importance sampling* which selects samples based on their computed importance to construct the mini-batch.

The *performance model* of the importance sampling-based framework can be formulated as follows: let T_{is} denote the increased local computation cost in each SGD iteration caused by importance sampling, and E_{is} denote the corresponding total number of iterations until convergence. The training speedup of importance sampling-based framework over standard random sampling-based distributed training can be derived as:

$$Speedup = \frac{E \cdot (T_{cp} + T_{cm})}{E_{is} \cdot (T_{cp} + T_{cm} + T_{is})} = \frac{1}{\frac{E_{is}}{E} \cdot (1 + \frac{T_{is}}{T_{cp}+T_{cm}})}. \quad (5)$$

As shown in Eq. (5), the rudimentary importance sampling-based framework, however, could perform worse than the standard random sampling-based distributed training (i.e., $Speedup < 1$) if the computation cost incurred by importance sampling T_{is} overshadows the benefit brought by the reduction of total training iterations (i.e., $E_{is} < E$). As such, to ensure $Speedup > 1$, a better design is needed.

3.3 Overall Design

Figure 5b illustrates the overall design of Mercury. Guided by the performance model in Eq. (5), Mercury incorporates three techniques that effectively avoid the pitfalls of the rudimentary framework. First, Mercury incorporates a *group-wise importance computation and sampling scheme* that considerably reduces the computation cost of importance sampling without compromising its effectiveness and training quality (§4.1). Second, it incorporates an *importance-aware data resharding scheme* for balancing importance distribution among devices to further accelerate the training process (§4.2). Lastly, it incorporates a *bandwidth-adaptive computation-communication scheduler* that schedules the execution of importance computation and data resharding in parallel in a bandwidth-adaptive manner such that their costs can be completely hidden behind the standard distributed SGD Mercury is built upon (§4.3).

In the following, we describe the details of these three techniques and provide a theoretical proof on Mercury's training correctness.

4 DESIGN DETAILS

4.1 Group-wise Importance Computation and Sampling

The first key technique in Mercury is *group-wise importance computation and sampling*: a technique that considerably reduces the computational cost of importance sampling without compromising its effectiveness and biasing convergence. In the rudimentary importance sampling-based framework introduced in §3.2, in each SGD iteration, the importance of *all* the local data are computed to derive their importance distribution. The all-inclusiveness of this approach naturally leads to costly computation. In contrast,

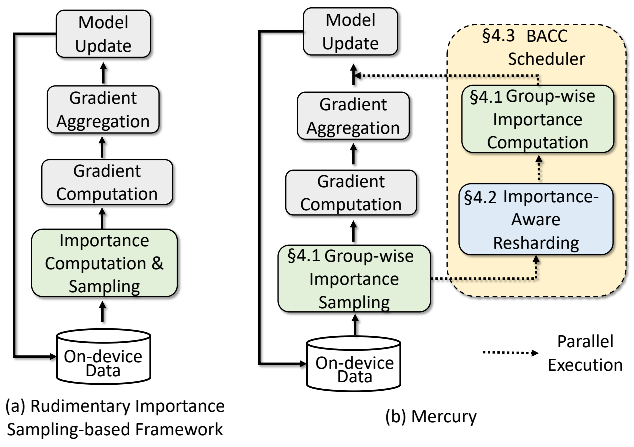
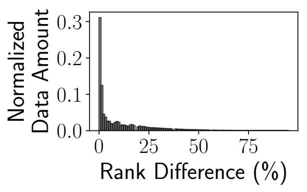

Model Update
Gradient Aggregation
Gradient Computation
Importance Computation & Sampling
On-device Data
(a) Rudimentary Importance Sampling-based Framework
Model Update
Gradient Aggregation
Gradient Computation
§4.1 Group-wise Importance Sampling
On-device Data
§4.3 BACC Scheduler
§4.1 Group-wise Importance Computation
§4.2 Importance-Aware Resharding
Parallel Execution
(b) Mercury

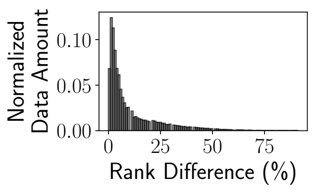
Figure 5: Comparison between (a) rudimentary importance sampling-based framework and (b) Mercury.

our group-wise technique foregoes such all-inclusiveness to cut the computational cost: during each SGD iteration, it only computes the importance of a *subset* (i.e., group) of the local data. To guarantee unbiased convergence, it adopts a *stochastic* approach to construct the mini-batch based on the computed group-wise importance.

Our technique is inspired by an observation that *the importance of each sample of the local data does not change abruptly across multiple SGD iterations.* Figure 6 depicts the difference of importance ranks between two iterations. About 50% of data samples change less than 5% in importance ranking and about 70% of data samples change less than 10% after one training iteration (Figure 6a). Even after 10 iterations, about 65% of data samples change less than 10% in importance ranking (Figure 6b). In other words, if one sample is identified as important, it would possibly stay as an important sample for the following multiple SGD iterations. Therefore, it is not necessary to re-compute the importance of each sample in the local training dataset for every SGD iteration. Instead, we can reuse the importance computation results from previous iterations to further cut the computation cost.



(a) Distribution of importance rank changes after 1 iteration.



(b) Distribution of importance rank changes after 10 iterations.

Figure 6: Distribution of importance rank changes across iterations.

Based on this observation, our group-wise importance computation and sampling technique consists of two steps as shown in Figure 7.

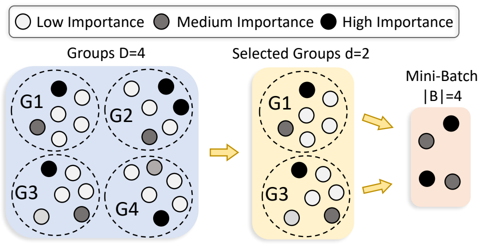


Figure 7: Illustration of group-wise importance computation and sampling. Each dot represents one sample and the darkness indicates its importance.

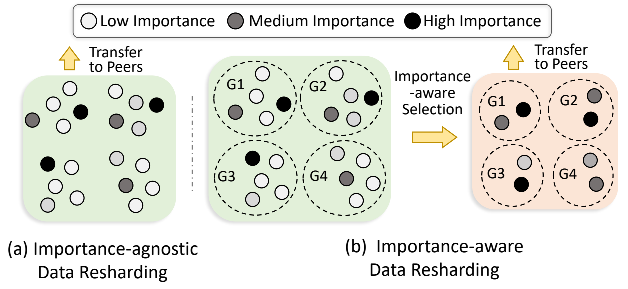

○ Low Importance ● Medium Importance ● High Importance
Transfer to Peers
Importance -aware Selection
G1
G2
G3
G4
(a) Importance-agnostic Data Resharding
(b) Importance-aware Data Resharding

Figure 8: Comparison between (a) importance-agnostic data resharding and (b) importance-aware data resharding.

Step#1: Compute Group-wise Importance. In the first step, we split the complete local dataset on each edge device into D non-overlapping groups. At each SGD iteration, instead of re-computing the importance for all the samples, only the importance of samples in one of the D groups is re-computed. In doing so, the computational cost of importance sampling becomes $1/D$ of the all-inclusive one per SGD iteration. The group is selected in a round-robin fashion such that every group is selected once every D SGD iterations.

Since we only update the sample importance of one group in each SGD iteration and reuse the previously computed importance for the remaining $D-1$ groups, there is a discrepancy between the derived importance distribution and the importance distribution derived by re-computing the importance of all the samples. This is because as model parameters used to compute the sample importance are updated due to SGD, the sample importance computed in different iterations do not belong to the same distribution.

Step#2: Construct Mini-Batch based on Group-wise Importance. To mitigate the effect caused by the discrepancy, in the second step, we need to consider two levels of importance to construct a mini-batch. The first is the *inter-group level importance*, which reflects the relative freshness of a group compared to the other groups. The second is the *intra-group level importance*, which reflects the importance of samples within the same group.

To accommodate the *inter-group level importance*, we first select d groups out of D groups, and assign the probability of selecting group i as:

$$r_i = \frac{\exp(\beta t_i)}{\sum_{n=1}^{D} \exp(\beta t_n)} \tag{6}$$

where t_i is the relative step index when the importance of group i is updated and $\beta > 0$ is the amplifying factor. A larger β encourages the selection of newer groups.

To accommodate the *intra-group level importance*, we select $|\mathcal{B}| \times \frac{M_i}{\sum_{n=1}^{d} M_n}$ from each selected group i using importance sampling where M_i is the size of group i. The probability of a sample being selected is proportional to its feed-forward loss within its group. In other words, instead of selecting samples randomly with uniform distribution $p_{i,j} = 1/M_i$, the probability of selecting sample j in group i is given by:

$$q_{i,j} = \frac{I_{i,j}}{\sum_{n=1}^{M_i} I_{i,n}} \tag{7}$$

where $I_{i,j}$ is the importance of the sample j in group i. Finally, the mini-batch is constructed by combining the selected samples from the selected groups. To guarantee unbiased convergence, we reweigh the computed gradients among samples by multiplying the gradient with $p_{i,j}/q_{i,j}$ to obtain the final unbiased gradient [22].

4.2 Importance-aware Data Resharding

The second key technique in Mercury is *importance-aware data resharding*: a technique for balancing importance distribution among edge devices to accelerate the training process. As importance update is performed on the local dataset at each device, the local importance rank within each device does not reflect the global importance rank that accumulatively considers the importance of all the samples from all the devices. As a consequence, an edge device may repeatedly learn globally trivial samples, which lowers the training efficiency. This problem is exacerbated when data distribution across edge devices is non-IID (identically and independently distributed), which is common in distributed settings.

Such problem can be resolved by data resharding, a technique that redistributes samples among workers. In data centers, data resharding is commonly applied and can be easily achieved due to the availability of high-bandwidth network. However, in on-device settings where the network bandwidth is much more constrained, shuffling a large amount of data among edge devices could significantly delay the training process.

To this end, we propose importance-aware data resharding that only redistributes important samples. Instead of blindly shuffling data among edge devices, we only select non-trivial samples to shuffle to maximize data resharding efficiency with minimum communication overhead. As shown in Figure 8, suppose after data resharding is determined, a device with N samples and D groups needs to swap $N_p < N$ samples with other devices and the budget only allows $N_o < N_p$ samples to be transferred. For importance-agnostic data resharding, it directly transfers N_p samples without taking the sample importance information into consideration. In contrast, importance-aware data resharding selects $N_o \frac{M_i}{\sum_j M_j}$ samples from each group i where the group size is M_i. The probability of a sample being selected is proportional to its importance within its belonging group as in Eq. (7). As a result, we are able to select important samples to be redistributed without change in group size or local dataset size.

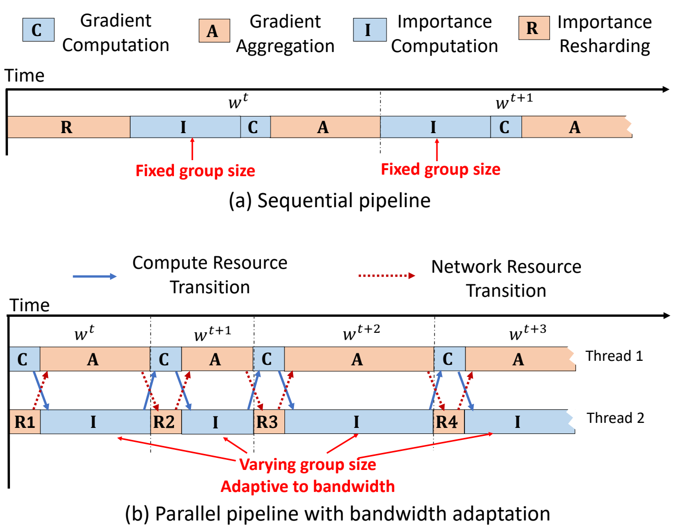


Figure 9: Comparison between (a) sequential pipeline and (b) parallel pipeline with bandwidth adaptation. w^t represents the model weight at iteration t. R = R1 + R2 + R3 + R4 in length.

4.3 BACC Scheduler

A naive way of incorporating importance sampling into the standard SGD pipeline is shown in Figure 9a. As shown, the importance computation and data resharding is *sequentially* inserted in the training process, incurring unnecessary blocking and overhead.

As shown in Figure 9b, Mercury incorporates a *bandwidth-adaptive computation-communication (BACC) scheduler* which schedules the execution of importance computation and data resharding *in parallel* in a bandwidth-adaptive manner to further enhance the training efficiency by *completely masking out the costs of importance sampling and data resharding*. Specifically, since gradient aggregation only uses network resource while gradient computation only uses compute resource, the group-wise importance computation and importance-aware data resharding can be overlapped with gradient aggregation and gradient computation respectively and executed in parallel. This can be achieved by creating two threads, one for standard distributed SGD operations and the other for group-wise importance computation and importance-aware data resharding. In doing so, the overheads incurred by these two techniques can be masked out and the training speedup can be further improved.

Given that both gradient aggregation and data resharding depend on network bandwidth which could experience variations over time in real-world deployments, achieving full overlapping that completely masks out the costs of importance computation and data resharding requires the scheduler to be *bandwidth-adaptive*. We propose two modifications to adapt to the bandwidth variation. First, to fully overlap importance computation with gradient aggregation, instead of using a fixed group size, Mercury adopts varying group sizes such that computing the importance of the samples in a group consumes the same amount of time as gradient aggregation. To compensate for the reduction of importance variety in groups of small sizes and ensure unbiased convergence, the importance of each data sample sampled from groups of different sizes will be reweighed using its group size. Second, to fully overlap data resharding with gradient computation, Mercury adopts a breakpoint-resume technique: data resharding pauses when gradient aggregation begins and resumes when it ends. By doing these, the costs of importance sampling can be completely hidden behind the standard distributed SGD in a bandwidth-adaptive manner.

Figure 9b illustrates how our proposed BACC scheduler is able to achieve full overlapping to completely mask out the costs of importance computation and data resharding. Specifically, after gradient computation, Thread 1 yields its compute resource to Thread 2 to perform importance computation. After gradient aggregation is complete, Thread 2 yields the compute resources to Thread 1 to perform gradient computation for the next iteration $t + 1$. Similarly, after gradient aggregation, Thread 1 yields network resource to Thread 2 to perform data resharding. When gradient aggregation for the next iteration begins, Thread 2 pauses data resharding and yields network resource back to Thread 1 to perform gradient aggregation for the next iteration.

4.4 Proof of Training Correctness

Lastly, we provide the theoretical result showing that our proposed importance sampling-based on-device distributed training is guaranteed to converge to the same solution as the standard distributed SGD *without bias*. It should be noted that our proof does not require the data distribution across edge devices to be IID, and hence our proof is valid for both IID and non-IID data distributions.

Sketch of Proof: we first show that the stochastic gradient information averaged at the server is unbiased. Then we show that its variance is bounded.

1. The gradient averaged across all workers is unbiased. We use $\nabla l_{k,i,j}$ to represent the gradient from the sample j in group i of device k for the simplicity of notation. Using the proposed sampling technique in §4.1, the expectation of aggregated gradient $\mathbf{g}$ is

$$\mathbb{E}(\mathbf{g}^t) = \frac{1}{\sum_{k=1}^{K} N_k} \sum_{k=1}^{K} N_k \frac{1}{N_k} \sum_{i=1}^{D_k} \sum_{j=1}^{M_{k,i}} \nabla l_{k,i,j} = \nabla l(\mathbf{w}^t).$$

where $M_{k,i}$ is the number of samples in group i at node k; N_k and D_k are the number of samples and the number of groups at node k. It shows that the averaged gradient is unbiased.

2. The gradient averaged across all the devices has bounded variance. Note that the variance of $\mathbf{g}^t$ is

$$\mathbf{Var}(\mathbf{g}^t) = \sum_{k=1}^{K} (\frac{N_k}{\sum_j N_j})^2 \mathbf{Var}(\mathbf{g}_k).$$

Since we only change the probability of choosing the samples, the variance $\mathbf{Var}(\mathbf{g}_k)$ at each node k is still bounded. Therefore, the variance of $\mathbf{g}^t$ is bounded. For simplicity, we use the same notation $\mathcal{V}$ for this bound.

2. Convergence rate. Since the loss function l has a Lipschitz continuous gradient with constant L, we have the following inequality:

$$l(\mathbf{w}^t) - l(\mathbf{w}^{t+1}) \\ \geq \langle \nabla l(\mathbf{w}^t), \mathbf{w}^t - \mathbf{w}^{t+1} \rangle - \frac{L}{2} \|\mathbf{w}^t - \mathbf{w}^{t+1}\|^2.$$

Taking the expectation over all possible $\mathbf{g}^t$, summing this inequality from $t = 0$ and taking the expectation over all possible $\{\mathbf{w}^t\}$, we

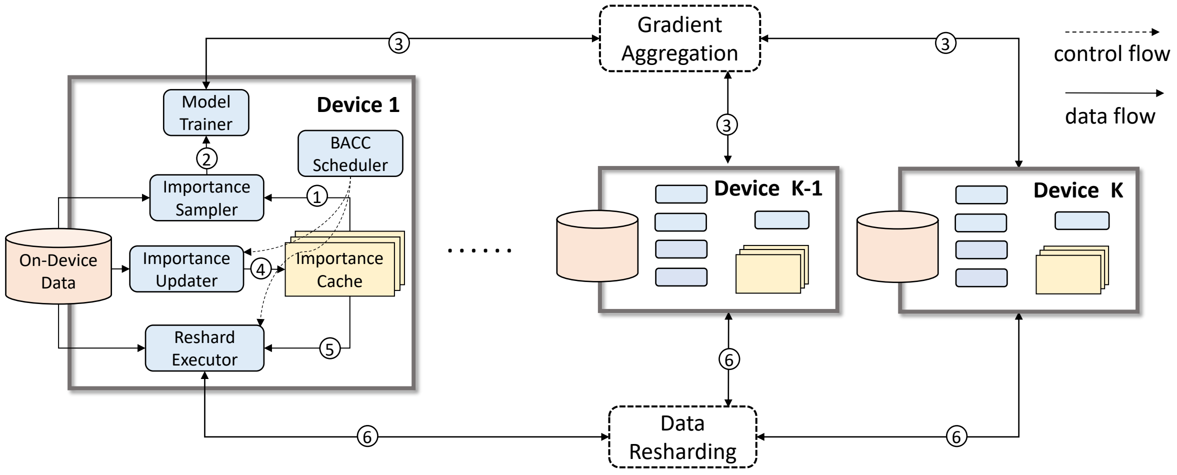


Figure 10: System architecture of Mercury.

have

$$l(\mathbf{w}^0) - \mathbb{E}l(\mathbf{w}^{t+1})$$
$$\geq \sum_{i=0}^{t}\left(\eta_i - \frac{L\eta_i^2}{2}\right) \min_{i\in\{0,1,\ldots,t\}} \mathbb{E}\|\nabla l(\mathbf{w}^i)\|^2 - \sum_{i=0}^{t}\frac{L\eta_i^2}{2}\mathcal{V}.$$

The last inequality holds because we choose $\eta_i < 2/L$. Therefore, we obtain

$$\min_{i\in\{0,1,\ldots,t\}} \mathbb{E}\|\nabla l(\mathbf{w}^i)\|^2 \leq \frac{l(x^0) - l^* + \sum_{i=0}^{t}\frac{L\eta_i^2}{2}\mathcal{V}}{\sum_{i=0}^{t}\left(\eta_i - \frac{L\eta_i^2}{2}\right)},$$

where l^* is the minimum function value for l. If the learning rate is set as $\eta_t = \eta = \frac{1}{\sqrt{T}}$, then we have

$$\min_{i\in\{0,1,\ldots,T\}} \mathbb{E}\|\nabla l(\mathbf{w}^i)\|^2 \leq \frac{l(\mathbf{w}^0) - l^* + \frac{L}{2}\mathcal{V}}{\sqrt{T} - \frac{L}{2}}.$$

Thus, we have $\min_{i\in\{0,1,\ldots,T\}} \mathbb{E}\|\nabla l(\mathbf{w}^i)\| \leq \epsilon$ after $O(1/\epsilon^4)$ iteration, and the learning rate η is in the order of ϵ^2.

5 IMPLEMENTATION

Testbed. We designed and developed our own testbed due to the lack of off-the-shelf ones. Specifically, we use 12 NVIDIA Jetson TX1 as the edge devices. Each TX1 has an integrated small form factor mobile GPU that is designed for next-generation intelligent edge devices to execute DL-powered tasks onboard. For wireless networking, we use Netgear Nighthawk X6S AC4000 Tri-band Wi-Fi routers to connect all the TX1, and use Linux *tc*, *qdisc*, and *iptables* to control the network bandwidth to conduct our experiments.

System Implementation. We have implemented Mercury using TensorFlow v1.12.0. Figure 10 shows the system architecture of Mercury. Specifically, Mercury is implemented as a distributed training framework that spans across edge devices. In each training iteration, the *Importance Sampler* first constructs a mini-batch from on-device data using group-wise importance sampling (①) based on the importance distribution of local data stored in *Importance Cache*. The mini-batch is then fed into the *Model Trainer* to compute the local gradients (②). The local gradients from all the participating edge devices are aggregated, and the aggregated gradients are sent to the *Model Trainer* to update the model (③). Meanwhile, the *Importance Updater* re-computes the data importance and updates the *Importance Cache* (④) while the edge device is performing gradient aggregation. Finally, the *Reshard Executor* identifies important data samples from *Importance Cache* (⑤) and communicates with other devices to perform importance-aware data resharding (⑥) while the device is performing gradient computation. The executions of *Reshard Executor* and *Importance Updater* are triggered and scheduled by the *BACC Scheduler* at runtime.

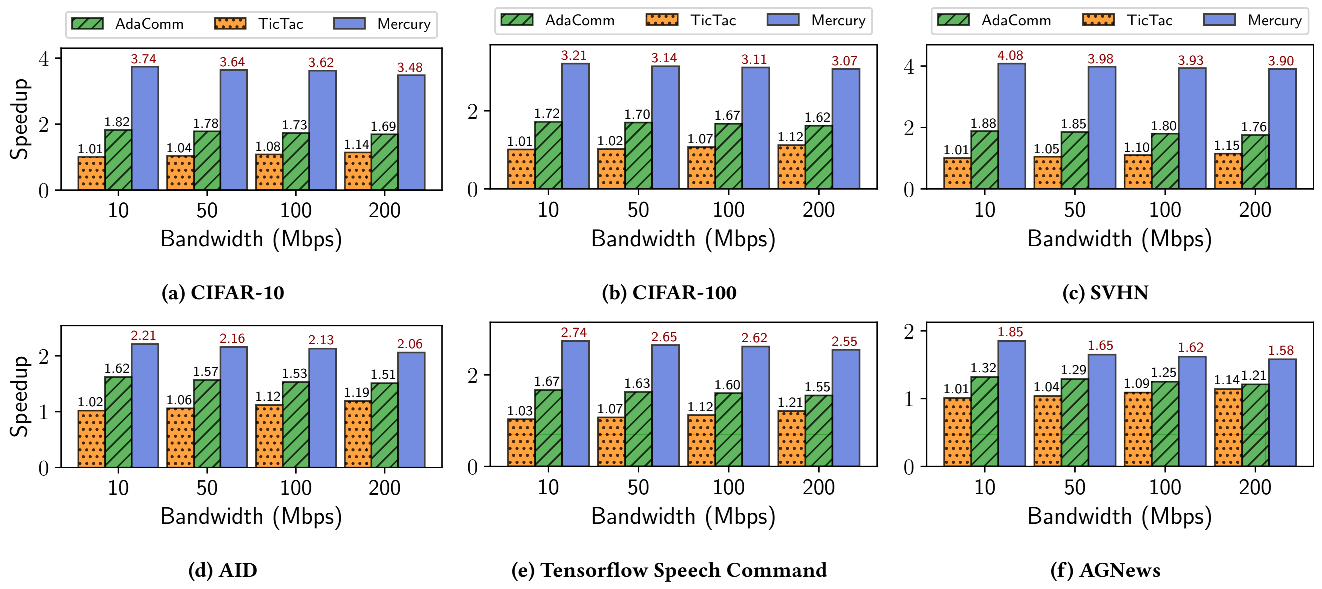
6 EVALUATION

In this section, we evaluate the performance of Mercury with the aim to answer the following questions:

- **Q1 (§6.2):** *Does Mercury outperform status quo? If so, what are the reasons?*
- **Q2 (§6.3):** *How effective is each core technique incorporated in the design of Mercury?*
- **Q3 (§6.4):** *Can Mercury adapt to wireless network bandwidth variation well?*
- **Q4 (§6.5):** *Does Mercury scale well when the number of edge devices increases?*
- **Q5 (§6.6):** *Can Mercury improve the training efficiency of federated learning (FL) where each edge device's data is stored locally and not exchanged or transferred?*

6.1 Experimental Methodology

Tasks, Datasets, and DL Models. To demonstrate the generality of Mercury across tasks, datasets, and DL models, we evaluate Mercury on six public datasets that are commonly used for benchmarking using a diverse set of DL models across three important



(a) CIFAR-10 **(b) CIFAR-100** **(c) SVHN**

(d) AID **(e) Tensorflow Speech Command** **(f) AGNews**

Figure 11: Overall performance comparison between Mercury, TicTac and AdaComm. Each bar represents the training speedup over standard distributed SGD in total training time.

tasks: image classification, speech recognition, and natural language processing. These tasks, datasets, and models are selected for the on-device training settings for edge devices.

- **Image Classification.** For the task of image classification, we use four datasets. Specifically, we select CIFAR-10 [24], CIFAR-100 [24], and SVHN [40] since they are three of the most commonly used datasets for image classification. Both CIFAR-10 and CIFAR-100 consist of 50, 000 training images and 10, 000 test image in 10 classes. SVHN consists of 73, 257 training and 26, 032 test images. We use ResNet-18 [14] to train on CIFAR-10 and SVHN, and ResNet-50 [14] to train on CIFAR-100. In addition, we select AID [41] as our fourth image classification dataset. AID is a large-scale aerial image dataset collected from Google Earth. It has 10, 000 images in 30 classes including airport, mountain, desert, forest, etc. We randomly select 80% images for training and the remaining 20% images for testing. We select this dataset to emulate the application of on-device distributed training across a swarm of drones. We use MobileNetV2 [31] to train on this dataset.
- **Speech Recognition**. For the task of speech recognition, we select Tensorflow Speech Command [33] as our dataset. This dataset consists of 105, 829 audio utterances of 35 short words, recorded by a variety of different people. The recorded audio clips are intended for simple speech instructions such as *go*, *stop*, *yes*, *no*, etc. The training set has 84, 843 samples and the test set has 11, 005 samples. We extract the 2-D spectrograms from the raw audio clips, and use VGG-13 [32] as the model.
- **Natural Language Processing**. For the task of natural language processing, we select AG News Corpus [47] as our dataset. This dataset contains news articles from the AG's corpus. It has 120, 000 training and 7, 600 testing samples. Each sample consists of a couple of sentences and is labeled as one of the four classes: *world*, *sports*, *business* and *sci/tech*. We use a two-layer LSTM with attention [3, 16] to train on this dataset.

Baselines. The goal of Mercury is to enhance the training efficiency of on-device distributed training while retaining algorithm correctness guarantees without compromising the accuracies of the trained models. For fair comparison, we compare Mercury against two status quo frameworks which share the same goal[1].

- **TicTac** [12]. TicTac is a status quo distributed training framework for training acceleration. It re-orders parameter transfer nodes in a computational graph to increase the overlapping between computation and communication, and outperforms TensorFlow by 1.19× in terms of training efficiency.
- **AdaComm** [36]. AdaComm is another status quo efficient distributed training framework. AdaComm reduces the total training time by allowing each worker to perform multiple local SGD before communication and adaptively balancing the number of local SGD iterations and communication.

Evaluation Metrics. We use two metrics to evaluate the performance of Mercury and the baselines.

- **Total Training Time**. We use total training time as our first metric. Total training time is defined as the wall-clock time from the start to the convergence of the training process. This metric jointly considers the number of iterations until convergence as well as the training time per iteration.

[1]Gaia [17] is the status quo communication-efficient distributed training framework based on gradient compression. However, Gaia obtains training efficiency by compromising the accuracies of the trained models. Thus, we did not include it as a baseline.

- **Training Quality**. The second metric is training quality. We use the Top-1 test accuracy of the trained DL model to measure the training quality.

Training Details. For all the experiments, we use Adam as the optimizer with $\beta_1 = 0.9$ and $\beta_2 = 0.999$. We use local batch size of 32 for CIFAR-10, CIFAR-100, SVHN and AID. For Tensorflow Speech Command and AG News, we use local batch size of 64. For model update, the learning rate is set to 0.004.

6.2 Overall Performance

We begin with comparing the overall performance of Mercury with TicTac and AdaComm. To do so, we run training experiments on four edge devices, and measure their total training time speedups over standard distributed SGD on the six datasets. To provide a comprehensive evaluation, we did our experiments under various network bandwidths ranging from 10 Mbps to 200 Mbps, emulating scenarios with different wireless network bandwidth availability.

Figure 11 shows the results. Overall, we observe that Mercury significantly outperforms TicTac and AdaComm on all the six datasets across all the network bandwidths. Specifically, Mercury achieves up to 3.74×, 3.21×, 4.08×, 2.21×, 2.74× and 1.85× speedups over standard distributed SGD on six datasets respectively. In contrast, TicTac achieves a training speedup from 1.01× to 1.21× while AdaComm achieves a training speedup from 1.21× to 1.88×.

We also observe that Mercury achieves higher training speedups as the network bandwidth becomes more constrained. Specifically, across all six datasets, Mercury achieves the highest performance gain under 10 Mbps, the lowest bandwidth in our experiment. This is because under more constrained bandwidth, the communication time is longer and Mercury can thus spend more time on computing the latest importance rank. This result demonstrates that the more constrained the bandwidth is, the more superiority Mercury has.

Why Mercury Outperforms TicTac and AdaComm? To understand why Mercury is able to achieve higher training speedups compared to TicTac and AdaComm, we use CIFAR-10 as an example, and show the test accuracy curves of Mercury (blue), TicTac (orange) and AdaComm (green) during the complete training process in Figure 12. We have two observations.

First, Mercury outperforms TicTac by a large margin from the beginning to the convergence. Specifically, compared to TicTac, Mercury uses 2.7× and 2.4× less communication rounds on CIFAR-10 and 2.7× less communication rounds on Tensorflow Speech Command to converge. This is because under wireless network setting where bandwidth is limited, the communication time dominates the overall training timing, making overlapping computation with communication not effective anymore (Figure 2). Therefore, TicTac provides little runtime reduction in each iteration. In contrast, Mercury aims at improving the training efficiency *per iteration* to reduce the total number of iterations (communication rounds) without additional overhead.

Second, although AdaComm converges faster than Mercury in early stage when the number of communications is smaller than 10×10^3 (Figure 12a), AdaComm begins to lose effect and converges slower than Mercury. This is because although AdaComm can also reduce communication rounds, it achieves such reduction by performing local training steps in early stage of training. To guarantee the training quality, AdaComm has to perform less local steps and the acceleration of AdaComm begins to slow down after the early stage. In addition, performing multiple local steps increase the computation time in each round. In contrast, with the training correctness guarantee, Mercury is able to increase training efficiency throughout the entire training process without additional overheads. Therefore, even though AdaComm achieves faster convergence during early stage of training, Mercury eventually outperforms AdaComm in terms of both number of communications and total training time.

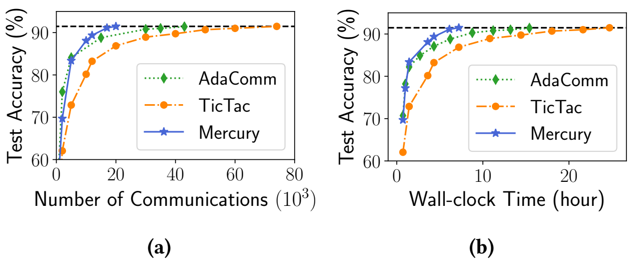


Figure 12: Test accuracy curves during the complete training process in terms of (a) the number of communication rounds and (b) wall-clock time comparison between Mercury, TicTac and AdaComm.

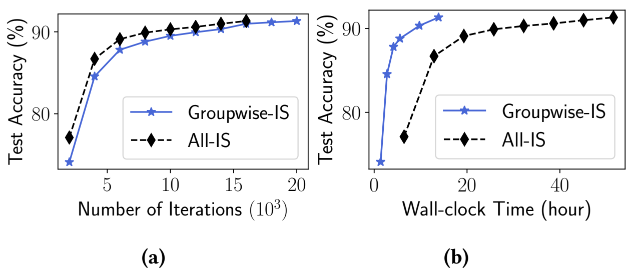


Figure 13: Test accuracy curves in terms of (a) the number of iterations and (b) wall-clock time comparison between group-wise (Groupwise-IS) and all-inclusive (All-IS) importance computation and sampling.

6.3 Component-wise Analysis

Next, we evaluate the effectiveness of each of the three key techniques incorporated in Mercury. The experimental setup is the same as in §6.2. We use CIFAR-10 to conduct our experiments. The results altogether show that with the three proposed techniques, Mercury can achieve higher performance gain compared to the rudimentary importance sampling-based framework.

Component#1 Analysis: Group-wise vs. All-inclusive Importance Computation and Sampling. We evaluate the effectiveness of the proposed group-wise importance computation and sampling technique described in §4.1. To do so, we compare it against the all-inclusive importance computation and sampling strategy, which re-computes the importance of every sample in the local dataset at each SGD iteration. The results are shown in Figure 13.

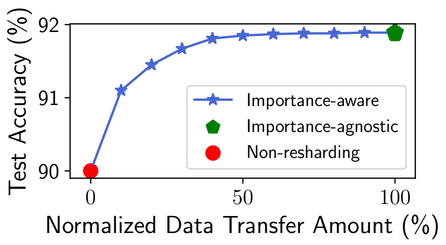
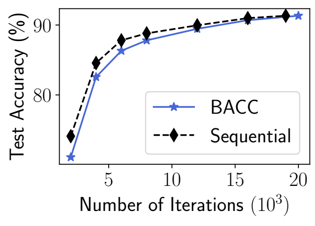
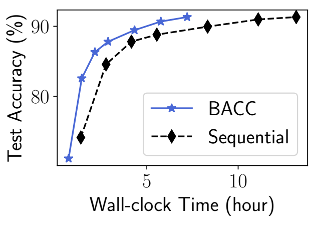


Figure 14: Performance comparison between non-resharding, importance-agnostic data resharding, and importance-aware data resharding.



(a)



(b)

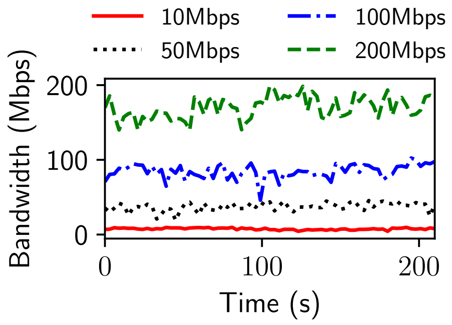
Figure 15: Test accuracy curves in terms of (a) the number of iterations and (b) wall-clock time using BACC and sequential pipeline.

As we can see, while group-wise importance computation and sampling uses 20% more iterations to converge to the same test accuracy as the all-inclusive importance computation and sampling scheme (Figure 13a), group-wise is 4.2× faster than the all-inclusive when translated into total training time (Figure 13b). This is because even though re-computing the importance of every sample indeed improves the training quality per iteration, the significant computation cost it incurs significantly prolongs the training time per iteration. In contrast, by sacrificing marginal training quality per iteration, group-wise computation and importance sampling is able to considerably cut the training time per iteration, leading to a much reduced total training time.

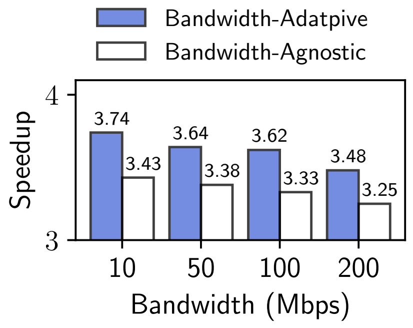
Component#2 Analysis: Importance-aware vs. Importance-agnostic Data Resharding. Next, we evaluate the effectiveness of the proposed importance-aware data resharding technique described in §4.2. To do so, we compare it against the importance-agnostic data resharding strategy. Different from importance-aware resharding which selects the important samples during resharding, importance-agnostic treats every sample equally and randomly reshuffles the samples across edge devices. Figure 14 shows the results. We have two observations.

First, similar to the findings from many other works [10, 11, 30], we observe that compared to non-resharding, data resharding helps boost the test accuracy, and in our case can increase the test accuracy up by 1.84%.

Second, compared to importance-agnostic data resharding, the proposed importance-aware strategy is able to converge to the same test accuracy, but with only 50% of its data transfer amount. This is because importance-aware resharding prioritizes shuffling more important samples, which considerably improves the resharding efficiency and reducing the network traffic for data resharding.



(a) A snapshot of bandwidth variations.



(b) Training speedup comparison between bandwidth-adaptive and bandwidth-agnostic.

Figure 16: Adaptation to bandwidth variations.

Component#3 Analysis: BACC Scheduler vs. Sequential Pipeline. To evaluate the effectiveness of the proposed BACC scheduler described in §4.3, we compare it against the sequential pipeline, which performs data resharding, group-wise importance computation and sampling, gradient computation and gradient aggregation sequentially. As shown in Figure 15, although BACC uses 5% more iterations to converge (Figure 15a), when translated into total training time, it is 2.2× faster than the sequential pipeline (Figure 15b).

6.4 Adaptation to Bandwidth Variations

We also take a closer look at the bandwidth adaptation performance of the BACC scheduler proposed in Mercury. When deploying our testbed in the real-world settings, we observed that the bandwidth variations across different bandwidths are 10% on average (Figure 16a shows a snapshot). To examine the bandwidth adaptation performance of the BACC scheduler, we compare it against a bandwidth-agnostic solution where fixed group sizes and fixed-time data resharding are adopted. Figure 16b shows the speedup comparison across four bandwidths. As shown, without bandwidth adaptation, the training speedup drops from 3.74×, 3.64×, 3.62×, 3.48× to 3.43×, 3.38×, 3.33×, 3.25×, which validates the effectiveness of Mercury in adapting to bandwidth variations.

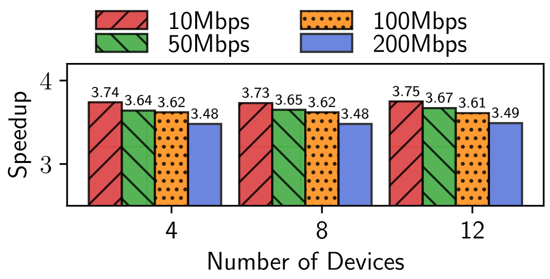


Figure 17: Scaling performance of Mercury on 4, 8, 12 edge devices under different network bandwidths.

6.5 Scaling Performance

To evaluate the scaling performance of Mercury, we use CIFAR-10 as the example and examine the training speedups of Mercury under different network bandwidths as Mercury scales up its number of edge devices from 4 to 8 and 12. Figure 17 illustrates the scaling performance of Mercury. As shown, Mercury is able to maintain its training speedup over standard distributed SGD under various network bandwidths as the number of edge devices increases.

6.6 Application to Federated Learning

Federated learning can be regarded as a constrained case of on-device distributed training where each edge device's data is stored locally and not exchanged or transferred. To examine whether Mercury can improve the training efficiency of federated learning, we retain group-wise importance sampling while disabling data resharding and BACC in Mercury, and evaluate its performance under different non-IID data distributions. Specifically, we generated three non-IID CIFAR-10 datasets using Dirichlet distribution [18] with three different concentration parameters: $\alpha = 10.0$ (low non-IID), $\alpha = 1.0$ (medium non-IID) and $\alpha = 0.5$ (high non-IID). We train ResNet-18 with group-wise importance sampling (Groupwise-IS) and without importance sampling (Non-IS) using 12 edge devices. As shown in Figure 18, our results show that Mercury is able to enhance the training efficiency of federated learning, achieving 2.75×, 2.5× and 2.41× training speedups over non importance sampling counterpart when $\alpha = 10.0$, $\alpha = 1.0$ and $\alpha = 0.5$.

7 RELATED WORK

Distributed Training in Data Centers. The design of Mercury was inspired by distributed training frameworks in data center settings. The work that is most similar to Mercury is AdaComm [36] and TicTac [12]. Specifically, AdaComm adaptively adjusts the number of local SGD iterations to reduce communication cost, and TicTac overlaps gradient computation with communication to reduce per-iteration overhead. Although techniques proposed in these work are effective in accelerating the training process, they are designed for distributed training in the data center setting. However, there is limited performance gain when directly applying these techniques to the on-device setting given the significant gap in network bandwidth between these two settings. Such limited performance gain motivates us to rethink the distributed training framework design for the on-device setting and to take a path that is different from existing ones.

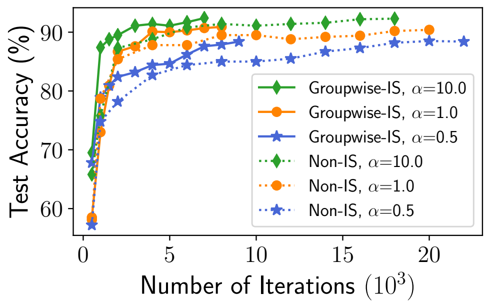


Figure 18: Performance of Mercury on federated learning under low ($\alpha = 10.0$), medium ($\alpha = 1.0$), and high ($\alpha = 0.5$) non-IID data distributions.

On-Device Distributed Training and Federated Learning. Federated learning [13, 23, 29, 35] can be regarded as a constrained case of on-device distributed training. In federated learning, data privacy is strictly enforced: during training, participating devices do not share their local data with each other. As we have demonstrated in §6.6, with the proposed group-wise importance computation and sampling technique, Mercury is able to enhance the training efficiency of federated learning under diverse non-IID distributions.

Importance Sampling and Curriculum Learning. Importance sampling shares similarities with curriculum learning [4] in terms of selecting important data to generate mini-batch for training. However, Mercury uses stochastic importance sampling which ensures that the training converges to the same solution as the standard distributed SGD without bias, while curriculum learning does not provide such guarantee.

8 CONCLUSION AND FUTURE WORK

In this paper, we present the design, implementation, and evaluation of Mercury, an importance sampling-based framework that enables efficient on-device distributed training without compromising the accuracies of the trained models. Mercury addresses the key bottleneck of on-device distributed training, and contributes novel techniques that take a different path from existing approaches. We implemented Mercury and conducted a rich set of experiments with a self-developed testbed on six commonly used datasets across tasks in image classification, speech recognition, and natural language processing. Our results show that Mercury consistently outperforms the status quo. Therefore, we believe Mercury represents a significant contribution to making on-device distributed training practically useful in real-world deployments. As our future work, we plan to extend Mercury to supporting more AI tasks such as object detection and scene understanding.

9 ACKNOWLEDGEMENT

We sincerely thank the anonymous reviewers and our shepherd for their valuable feedback. This work was partially supported by NSF Awards CNS-1617627, CNS-1814551, DMS-2012439, PFI:BIC-1632051, and Facebook Faculty Research Award on Systems for Machine Learning.

REFERENCES

[1] Guillaume Alain, Alex Lamb, Chinnadhurai Sankar, Aaron Courville, and Yoshua Bengio. 2015. Variance reduction in sgd by distributed importance sampling. *arXiv preprint arXiv:1511.06481* (2015).

[2] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Communication-efficient SGD via gradient quantization and encoding. In *Proceedings of the Advances in Neural Information Processing Systems.* 1709–1720.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[4] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning.* 41–48.

[5] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. 2016. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems.* ACM, 4.

[6] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *Proceedings of the 2009 IEEE conference on computer vision and pattern recognition.* Ieee, 248–255.

[7] Sanghamitra Dutta, Gauri Joshi, Soumyadip Ghosh, Parijat Dube, and Priya Nagpurkar. 2018. Slow and stale gradients can win the race: Error-runtime trade-offs in distributed SGD. *arXiv preprint arXiv:1803.01113* (2018).

[8] Biyi Fang, Xiao Zeng, Faen Zhang, Hui Xu, and Mi Zhang. 2020. FlexDNN: Input-Adaptive On-Device Deep Learning for Efficient Mobile Vision. In *ACM/IEEE Symposium on Edge Computing (SEC).*

[9] Biyi Fang, Xiao Zeng, and Mi Zhang. 2018. NestDNN: Resource-Aware Multi-Tenant On-Device Deep Learning for Continuous Mobile Vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking (MobiCom).* New Delhi, India, 115–127.

[10] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[11] Mert Gürbüzbalaban, Asu Ozdaglar, and Pablo Parrilo. 2015. Why random reshuffling beats stochastic gradient descent. *arXiv preprint arXiv:1510.08560* (2015).

[12] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. 2019. TicTac: Accelerating Distributed Deep Learning with Communication Scheduling. In *Proceedings of the 2nd SysML Conference.*

[13] Chaoyang He, Songze Li, Jinhyun So, Xiao Zeng, Mi Zhang, Hongyi Wang, Xiaoyang Wang, Praneeth Vepakomma, Abhishek Singh, Hang Qiu, et al. 2020. FedML: A research library and benchmark for federated machine learning. *arXiv preprint arXiv:2007.13518* (2020).

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 770–778.

[15] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Brian Kingsbury, et al. 2012. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal processing magazine* 29 (2012).

[16] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[17] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R Ganger, Phillip B Gibbons, and Onur Mutlu. 2017. Gaia: Geo-distributed machine learning approaching LAN speeds. In *NSDI.* 629–647.

[18] Tzu-Ming Harry Hsu, Hang Qi, and Matthew Brown. 2019. Measuring the effects of non-identical data distribution for federated visual classification. *arXiv preprint arXiv:1909.06335* (2019).

[19] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. 2019. Priority-based parameter propagation for distributed DNN training. *arXiv preprint arXiv:1905.03960* (2019).

[20] Shuang Jiang, Zhiyao Ma, Xiao Zeng, Chenren Xu, Mi Zhang, Chen Zhang, and Yunxin Liu. 2020. SCYLLA: QoE-aware Continuous Mobile Vision with FPGA-based Dynamic Deep Neural Network Reconfiguration. In *IEEE International Conference on Computer Communications (INFOCOM).*

[21] Angelos Katharopoulos and François Fleuret. 2017. Biased importance sampling for deep neural network training. *arXiv preprint arXiv:1706.00043* (2017).

[22] Angelos Katharopoulos and François Fleuret. 2018. Not all samples are created equal: Deep learning with importance sampling. *arXiv preprint arXiv:1803.00942* (2018).

[23] Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. 2016. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492* (2016).

[24] Alex Krizhevsky. 2009. *Learning multiple layers of features from tiny images.* Technical Report. Citeseer.

[25] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI).* 583–598.

[26] Youjie Li, Mingchao Yu, Songze Li, Salman Avestimehr, Nam Sung Kim, and Alexander Schwing. 2018. Pipe-SGD: A Decentralized Pipelined SGD Framework for Distributed Deep Net Training. In *Proceedings of the Advances in Neural Information Processing Systems.* 8045–8056.

[27] Hyeontaek Lim, David G Andersen, and Michael Kaminsky. 2018. 3LC: Lightweight and Effective Traffic Compression for Distributed Machine Learning. *arXiv preprint arXiv:1802.07389* (2018).

[28] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2017. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887* (2017).

[29] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017. Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics.* PMLR, 1273–1282.

[30] Qi Meng, Wei Chen, Yue Wang, Zhi-Ming Ma, and Tie-Yan Liu. 2017. Convergence analysis of distributed stochastic gradient descent with shuffling. *arXiv preprint arXiv:1709.10432* (2017).

[31] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 4510–4520.

[32] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[33] tensorflow. 2018. Tensorflow Speech Command Dataset. https://www.tensorflow.org/tutorials/sequences/audio_recognition

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the Advances in neural information processing systems.* 5998–6008.

[35] Jianyu Wang, Zachary Charles, Zheng Xu, Gauri Joshi, H Brendan McMahan, Maruan Al-Shedivat, Galen Andrew, Salman Avestimehr, Katharine Daly, Deepesh Data, et al. 2021. A Field Guide to Federated Optimization. *arXiv preprint arXiv:2107.06917* (2021).

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

[37] Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. 2018. Gradient sparsification for communication-efficient distributed optimization. In *Proceedings of the Advances in Neural Information Processing Systems.* 1299–1309.

[38] Pijika Watcharapichat, Victoria Lopez Morales, Raul Castro Fernandez, and Peter Pietzuch. 2016. Ako: Decentralised deep learning with partial gradient exchange. In *Proceedings of the Seventh ACM Symposium on Cloud Computing.* ACM, 84–97.

[39] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2017. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in neural information processing systems.* 1509–1519.

[40] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. 2016. *Data Mining: Practical machine learning tools and techniques.* Morgan Kaufmann.

[41] Gui-Song Xia, Jingwen Hu, Fan Hu, Baoguang Shi, Xiang Bai, Yanfei Zhong, Liangpei Zhang, and Xiaoqiang Lu. 2017. AID: A benchmark data set for performance evaluation of aerial scene classification. *IEEE Transactions on Geoscience and Remote Sensing* 55, 7 (2017), 3965–3981.

[42] Eric P Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. 2015. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data* 1, 2 (2015), 49–67.

[43] Xiao Zeng, Biyi Fang, Haichen Shen, and Mi Zhang. 2020. Distream: Scaling Live Video Analytics with Workload-Adaptive Distributed Edge Intelligence. In *ACM Conference on Embedded Networked Sensor Systems (SenSys).*

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

[45] Mi Zhang, Faen Zhang, Nicholas D Lane, Yuanchao Shu, Xiao Zeng, Biyi Fang, Shen Yan, and Hui Xu. 2020. Deep Learning in the Era of Edge Computing: Challenges and Opportunities. *Fog Computing: Theory and Practice* (2020), 67–78.

[46] Sixin Zhang, Anna E Choromanska, and Yann LeCun. 2015. Deep learning with elastic averaging SGD. *Advances in neural information processing systems* 28 (2015), 685–693.

[47] Xiang Zhang, Junbo Zhao, and Yann LeCun. 2015. Character-level convolutional networks for text classification. In *Proceedings of the Advances in neural information processing systems.* 649–657.

[48] Peilin Zhao and Tong Zhang. 2015. Stochastic optimization with importance sampling for regularized loss minimization. In *Proceedings of the International Conference on Machine Learning.* 1–9.