COLLABORATIVE DISTRIBUTED DEEP LEARNING SYSTEMS ON THE EDGES

By

Xiao Zeng

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Electrical Engineering — Doctor of Philosophy

2021

ABSTRACT

COLLABORATIVE DISTRIBUTED DEEP LEARNING SYSTEMS
ON THE EDGES

By

Xiao Zeng

Deep learning has revolutionized a wide range of fields. In spite of its success, most deep learning systems are proposed in the cloud, where data are processed in a centralized manner with abundant compute and network resources. This raises a problem when deep learning is deployed on the edge where distributed compute resources are limited. In this dissertation, we propose three distributed systems to enable collaborative deep learning on the edge. These three systems target different scenarios and tasks. The first system dubbed Distream is a distributed live video analytics system based on the smart camera-edge cluster architecture. Distream fully utilizes the compute resources at both ends to achieve optimized system performance. The second system dubbed Mercury is a system that addresses the key bottleneck of collaborative learning. Mercury enhances the training efficiency of on-device collaborative learning without compromising the accuracies of the trained models. The third system dubbed FedAce is a distributed training system that improves training efficiency under federated learning setting where private on-device data are not allowed to be shared among local devices. Within each participating client, FedAce achieves such improvement by prioritizing important data. In the server where model aggregation is performed, FedAce exploits the client importance and prioritizes important clients to reduce stragglers and reduce the total number of rounds. In addition, FedAce conducts federated model compression to reduce the per-round communication cost and obtains a compact model after training completes. Extensive experiments show that the proposed three systems are able to achieve significant improvements over status-quo systems.

ACKNOWLEDGMENTS

I would like to express the deepest appreciation to many people. First, I am honored and grateful to have Dr. Mi Zhang as my academic advisor. He has been not only an excellent supervisor but also a great friend to me since we met for the first time. I would not have made such much achievement and completed by Ph.D. degree without him. I would also like to thank Dr. Xiaobo Tan, Dr. Xiaoming Liu and Dr. Jiliang Tang who serve as my committee members and provide guidance for my academic career.

Second, I would like to thank my fellows in the lab, including Biyi Fang, Wei Du, Dong Chen, Jiajia Li, Yu Zheng, Wei Ao, Shen Yan, Xiaoyu Zhang, Haochen Sun, Kai Cao, Kaixiang Lin and Lai Wei for their encouragement and help during my life at Michigan State University. I am also grateful for all the staff and faculty in the ECE department.

Lastly, I would like to thank my family who have been providing tremendous support and encouragement. I have left them for so many years and I am truly thankful for their understanding. Especially, I would like to thank my wife, who has been staying with me since we came to the the U.S. to fulfil my Ph.D. degree. I am also blessed to have my son, Lucas, during the last year of my Ph.D. career. He is the best gift ever given to me in my Ph.D. life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Collaborative Deep Learning Inference on the Edge

1.1 Introduction

Video cameras are ubiquitous. Today, cameras have been deployed at scale at places such as traffic intersections, university campuses, and grocery stores. Driven by the recent breakthrough in deep learning (DL) [49], organizations that have deployed these cameras start to use DL-based techniques for *live video analytics* [86, 35, 40]. Analyzing live videos streamed from these distributed cameras is the backbone of a wide range of applications such as traffic control and security surveillance. As many of these applications require producing analytics results in real-time, achieving *low-latency*, *high-throughput*, and *scalable* video stream processing is crucial [77].

Live video analytics systems require high-resolution cameras to capture high-quality visual data for analytics. As camera number scales up, these always-on cameras collectively generate hundreds of gigabytes of data every single second, making it infeasible to transmit such gigantic volume of data to data centers in the cloud for real-time processing due to insufficient network bandwidth and long transmission latency between cameras and the cloud [88].

The key to overcoming this bottleneck is to *move compute resources close to where data reside*. Status quo live video analytics systems hence stream camera feeds to local edge clusters for data

	Architecture	Video Analytics Pipeline Partition	Workload Adaptive	Cross-Camera Workload Balancing
VideoStorm [86]	Centralized	N/A	No	N/A
Chameleon [40]	Centralized	N/A	No	N/A
NoScope [44]	Centralized	N/A	No	N/A
FilterForward [13]	Camera-Only	Fixed	No	No
VideoEdge [35]	Distributed	Dynamic	No	No
Distream	**Distributed**	**Dynamic**	**Yes**	**Yes**

Table 1.1: Comparison between Distream and status quo live video analytics systems.

analytics where much higher network bandwidth is provided [86, 40, 44, 65]. To move compute resources *even closer* to data sources, major video analytics providers (e.g., Avigilon, Hikvision, NVIDIA) are replacing traditional video cameras with "smart cameras". Equipped with onboard DL accelerators, these smart cameras are not only able to perform basic video processing tasks such as background subtraction and motion detection, but also capable of executing complicated compute-intensive DL-based pipelines to detect and recognize the objects and a variety of their attributes [27, 82, 41]. Since each smart camera brings extra compute resources to process video streams generated by itself, this *smart camera-edge cluster architecture* is the key to enabling live video analytics at scale [35, 13, 41].

Motivation & Limitations of Status Quo. In real-world deployments, depending on what areas the cameras are covering, the number of objects of interest (e.g., people, vehicles, bikes) captured by each camera is different and can vary significantly over time. For example, for surveillance systems deployed on university campuses, a camera that covers the building entrance captures much larger numbers of people before and after classes than any other time; a camera that points at the emergency exit where people rarely visit has no objects of interest captured most of the time. As a consequence, the workload of recognizing the captured objects and their attributes produced at each camera is different and inherently dynamic over time.

As listed in Table 1.1, in recent years, live video analytics systems such as VideoStorm [86],

Chameleon [40] and NoScope [44] have emerged. These systems enable efficient processing of a large number of camera streams, but are designed to process the streams within a centralized cluster.

With the emergence of smart cameras, recent systems start to leverage the compute resources inside smart cameras for distributed live video analytics. FilterForward [13] proposes a camera-only solution which identifies important video frames and filter out unimportant ones directly on smart cameras. VideoEdge [35], on the other hand, proposes a fully distributed framework that partitions the video analytics pipeline across cameras and cluster with the objective to optimize the resource-accuracy tradeoff. While these systems aim to optimize live video analytics from a variety of perspectives, they are *agnostic* to the workload dynamics in real-world deployments described above, making them fall short in two situations: on one hand, failing to utilize the compute resources inside idle cameras could considerably jeopardize system throughput; on the other hand, failing to alleviate the workloads from cameras that are overloaded by bursty workloads could incur significantly high latency, causing the system not able to meet the latency service level objective (SLO) imposed by the live video analytics applications.

Overview of the Proposed Approach. In this paper, we present Distream – a distributed framework based on the smart camera-edge cluster architecture – that is able to *adapt* to the workload dynamics in real-world deployments to achieve low-latency, high-throughput, and scalable DL-based live video analytics. The underpinning principle behind the design of Distream is to adaptively balance the workloads across smart cameras as well as partition the workloads between cameras and the edge cluster. In doing so, Distream is able to fully utilize the compute resources at both ends to jointly maximize the system throughput and minimize the latency *without sacrificing the video analytics accuracy*.

The design of Distream involves three key challenges.

- **Challenge#1: Cross-Camera Workload Balancing**. One key obstacle to achieving high-throughput low-latency live video analytics is caused by the imbalanced workloads across cameras. Therefore, the first challenge lies in designing a scheme that balances the workloads across cameras. However, the cross-camera workload correlation, the heterogeneous onboard compute capacities of smart cameras, and the overhead of workload balancing altogether make designing such a scheme not trivial.
- **Challenge#2: Camera-Cluster Workload Partitioning**. Another key obstacle to achieving high-throughput low-latency live video analytics is caused by the imbalanced workloads between smart cameras and the edge cluster. To balance the workloads between cameras and edge cluster, the video analytics pipeline should be partitioned based on the workload ratio of the two sides. However, the possible options to partition the video analytics pipeline are quite limited in number, making workload partitioning between camera and edge cluster by nature *coarse-grained*. As a result, the partitioned video analytics pipeline may not match the workload ratio of the two sides.
- **Challenge#3: Adaptation to Workload Dynamics**. Given the dynamics of workloads in real-world deployments, the optimal solutions for cross-camera workload balancing and camera-cluster workload partitioning *vary over time*. Being able to adapt to such workload dynamics is a must for high-performance live video analytics systems. Designing such an adaptation scheme, however, is not trivial, as the optimal pipeline partitioning solution for each camera can be *different*. More importantly, since the workloads are jointly executed between cameras and edge cluster, for the whole system to achieve the best performance, the optimal pipeline partitioning solutions for all the cameras need to be *jointly* determined. This is a much more

challenging problem compared to the single-pair workload partitioning problem tackled in the literature [16, 14].

To address the first challenge, Distream incorporates a cross-camera workload balancer that takes the cross-camera workload correlation, heterogeneous compute capacities of smart cameras, as well as the overhead of workload balancing into account, and formulates the task of cross-camera workload balancing as an optimization problem. In particular, the proposed cross-camera workload balancer incorporates a long-short term memory (LSTM)-based recurrent neural network which is able to enhance the performance of cross-camera workload balancing by predicting incoming workloads in the near future to avoid migrating workloads to cameras that are going to experience high workloads.

To address the second challenge, Distream incorporates a stochastic partitioning scheme that partitions the video analytics pipeline in a *stochastic* manner. In doing so, it provides much more partition flexibility and much finer partition granularity. As such, Distream is able to partition the pipeline to match the workload ratio of the smart camera and edge server.

To address the third challenge, Distream incorporates a workload adaption controller which triggers the cross-camera workload balancer when cross-camera workload imbalance is detected. Moreover, it formulates the task of jointly identifying the optimal pipeline partitioning solutions for all the cameras as an optimization problem with the objective to maximize the overall system throughput subject to the latency SLO imposed by the live video analytics applications.

System Implementation & Summary of Evaluation Results. We implemented Distream and deployed it on a self-developed testbed that consists of 24 smart cameras and a 4-GPU edge cluster. We evaluate Distream with 500 hours of distributed video streams from two real-world video datasets: one from six traffic cameras deployed in Jackson Hole, WY [7] for traffic monitoring

application, and the other from 24 surveillance cameras deployed on a university campus for security surveillance application. We compare Distream against three baselines: Centralized, which processes all the workloads on the edge cluster; Camera-Only, which processes all the workloads on smart cameras; and VideoEdge [35]. Our results show that Distream consistently outperforms the baselines in terms of throughput, latency, and latency SLO miss rate by a large margin due to its workload adaptation schemes. Moreover, our scaling experiments show that Distream is able to scale up system throughput nearly linearly while maintaining a low latency with negligible overheads. Finally, we show that the workload adaptation techniques proposed in Distream could benefit existing live video analytics systems and enhance their performance as well.

Summary of Contributions. To the best of our knowledge, Distream represents the first distributed framework that enables workl-oad-adaptive live video analytics under the smart camera-edge cluster architecture. It identifies a key performance bottleneck and contributes novel techniques that address the limitations of existing systems. We believe our work represents a significant step towards turning the envisioned large-scale live video analytics into reality.

1.2 Background and Motivation

1.2.1 Live Video Analytics Pipeline

Modern live video analytics pipelines typically adopt a cascaded architecture which consists of a front-end object detector followed by a back-end task-specific module to perform a variety of analytics tasks on each of the detected object of interest within a video frame.

There are two types of object detectors that have been commonly used in existing live video analytics systems [40]. The first type is the CNN-based object detector (*e.g.*, YOLO [68], SSD [57]

and Faster-RCNN [69]) which extracts and identifies all the objects of interest in a frame with one single inference. However, such an object detector has to be constantly extracting features from frames and performing inference even if there is no object of interest appearing in video streams. In many scenarios in real-world deployments, however, the objects of interest may only appear in video streams for short periods of time. In such case, a significant amount of compute resources is wasted. The second type of object detector is to first use a light-weight background subtractor [93] to extract the regions where objects of interest reside from the frame. It sends these regions to a classifier to identify the object within each region. As such, it produces objects of interest only when they appear in the frame. While Distream is a generic framework which supports both types of object detectors, in this work, we focus on the background subtraction-based detector to illustrate our ideas.

The task-specific module in general can be represented as a directed acyclic graph (DAG). In this work, we use attribute recognition as a concrete example of the task-specific module to illustrate our ideas. Specifically, each vertex in the DAG represents a DNN classifier that recognizes a particular attribute of the object; each directed edge represents a data flow from one classifier to another. For example, the task-specific DAG in Figure 1.1 consists of three branches. Based on the type of the detected object (vehicle, person, or others), the task-specific module selects one of the three branches, each of which employs a cascaded sequence of classifiers to further identify the attributes of the object[1].

Based on the pipeline illustrated in Figure 1.1, going through each classifier within the DAG is regarded as an individual *workload*[2]. Therefore, identifying an object of interest and its attributes

[1] Although we adopted the design choice of treating each task independently, Distream is flexible to support multi-task learning where a DNN inference produces multiple results. It should be noted that not all the tasks can be combined into a single multi-task DNN model. Thus our DAG formulation is general enough to support all the cases.

[2] We did not include capturing images and background extraction as workloads mainly because these steps consume much less computation compared to DNN-based inference and thus can be executed locally fast enough without offloading.

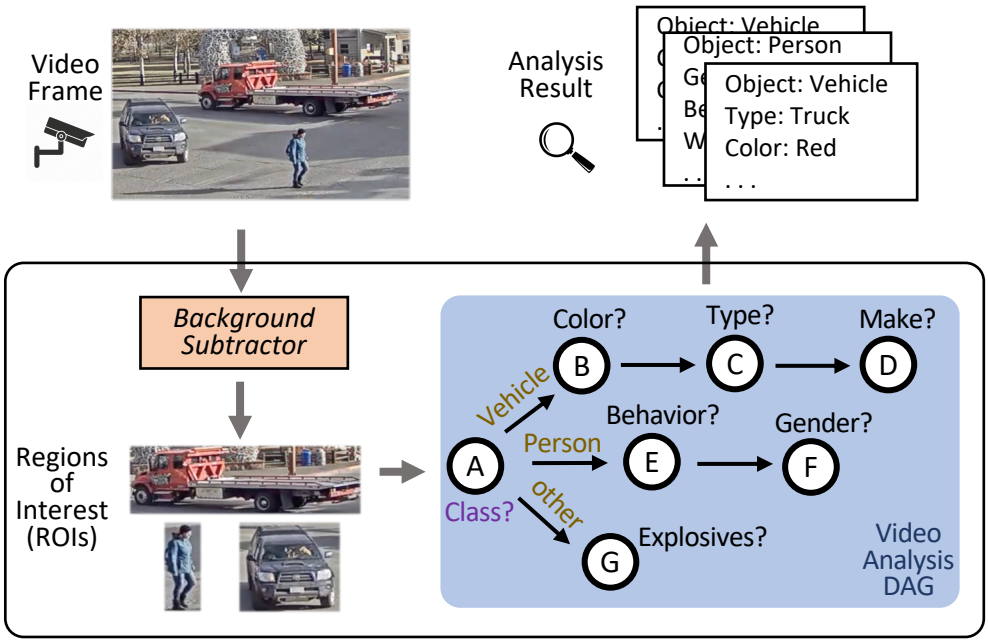


Figure 1.1: Live video analytics pipeline used in Distream.

within a video frame produces multiple workloads to be processed by the live video analytics system.

1.2.2 Workload Dynamics in Real-world Deployments

To illustrate the workload dynamics in real-world deployments, we collected a dataset from 24 surveillance cameras deployed on a university campus (§2.6.1). Given the limited space, we pick a representative video clip to make our points. Specifically, Figure 1.2 shows the workloads generated by four surveillance cameras deployed at a university building on a weekday between 12pm to 12:30pm. Among them, CAM1 monitors a square next to the building entrance; CAM2 monitors a sideway outside the building; CAM3 and CAM4 cover two different corridors inside the building.

As illustrated, depending on the views captured by the cameras, the workloads of real-world video streams have three key characteristics. (i) *The workloads are different across cameras*: for

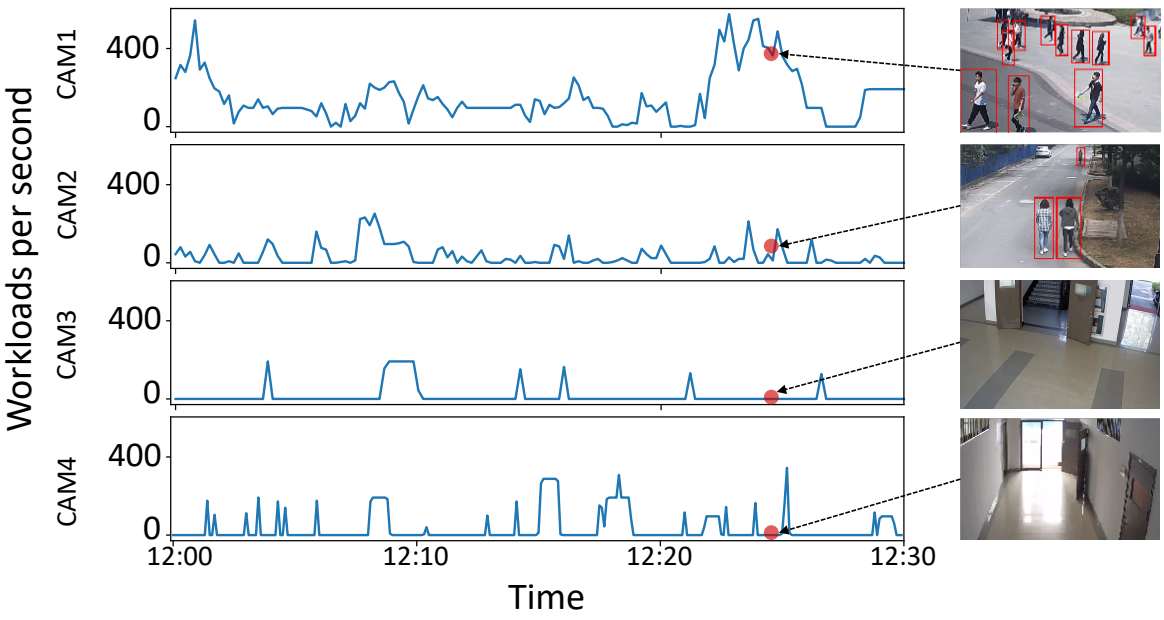


Figure 1.2: Workload dynamics in real-world deployment.

CAM1 that monitors busy areas, more workloads are generated as pedestrians pass by more often; for CAM3 that monitors a less busy indoor corridor, much less workloads are generated sporadically and it has no workloads most of the time. (ii) *The workload generated at each camera is dynamic over time*: this is obvious because the content captured by each camera is changing over time. (iii) *The amount of workloads generated at each camera can vary significantly*: the workload difference between bursty and non-bursty durations can be more than 1000×.

1.2.3 Need for Workload Adaptation

Existing live video analytics systems based on the smart camera-edge cluster architecture, however, are agnostic to the real-world workload dynamics illustrated above. To demonstrate how existing systems fall short under such workload dynamics, we compare the system performance between Distream that is workload-adaptive and a workload-agnostic baseline that allocates compute resource proportional to the compute capacity. We use the same dataset in §2.2 and a testbed that consists of four smart cameras (2 Jetson-TX1 and 2 Jetson-TX2) and 1-GPU edge cluster (§1.4) to

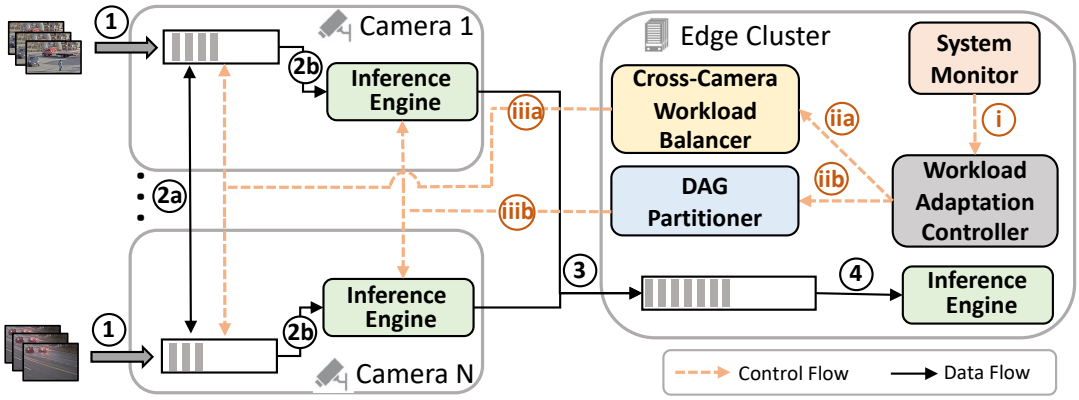


Figure 1.3: System architecture of Distream.

conduct our experiments.

Figure 1.4 illustrates our comparison results. As shown, Distream achieves significant improvement in terms of throughput and latency compared to the workload-agnostic scheme. Specifically, the median and peak throughputs of Distream are 302 IPS and 562 IPS respectively, which is 1.4× and 2.3× improvement over the workload-agnostic baseline, which only achieves 213 IPS median and 240 IPS peak throughputs. Distream also achieves 0.23s median and 3.92s maximum latency, which reduces medium and maximum latency by 57× and 40.7× compared to the workload-agnostic scheme which achieves 13.1s median latency and 159.73s maximum latency.

To understand why the workload-agnostic scheme falls short, we use one smart camera as an example and plot its workloads generated over a two mins video clip in Figure 1.5. Specifically, the black solid line depicts the generated workloads over time; the red dashed line depicts the camera's processing capability (i.e., the maximum number of workloads it can process per unit time) based on the workload-agnostic scheme. As shown, under the workload-agnostic scheme, although the workload generated at the smart camera is dynamic over time, the processing capability of the camera stays the same and does not adapt to the workload variation. Consequently, when the generated workload exceeds its processing capability, the camera experiences *under-provisioning*

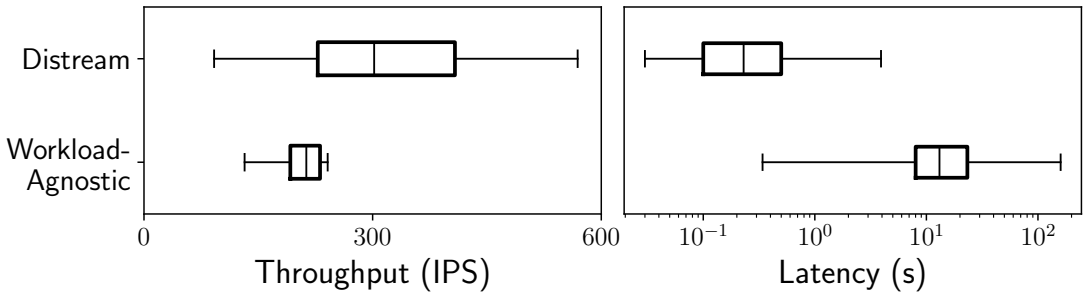


Figure 1.4: Comparison of throughput (left) and latency (right) between workload-agnostic and workload-adaptive (ours) schemes.

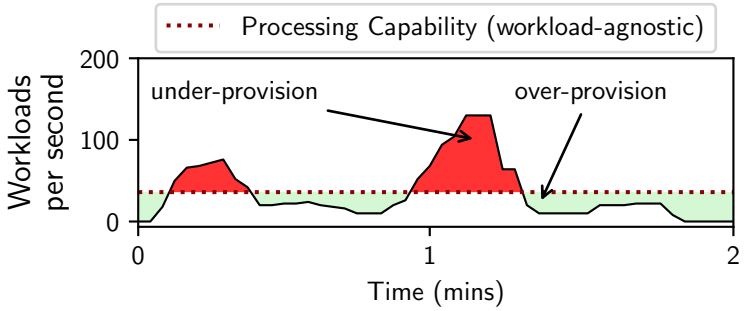


Figure 1.5: Illustration on where the workload-agnostic scheme falls short under dynamic workloads in real-world deployments.

(red region) so that it fails to process the workloads in time; when the workload is below its processing capability, the camera becomes *over-provisioning* (green region) and thus wastes its compute resources that could have been used to process workloads generated from other cameras. These observations altogether highlight the need for workload adaptation to avoid the occurrence of either under-provisioning or over-provisioning, which motivates the design of Distream.

1.3 Distream Design

1.3.1 Overall Architecture

Figure 2.5 illustrates the overall architecture of Distream. As shown, Distream is designed as a distributed framework that spans across smart cameras and the edge cluster. In the data plane, as soon as the smart camera captures a video frame, it runs the onboard background subtractor to extract regions of interest (ROIs) within the frame and appends them one by one into a local queue (①). Each ROI in the queue is either offloaded to another camera (②a) or partially processed inside the local *inference engine* according to its partitioned DAG whose partition point is determined by the *DAG partitioner* (②b). The result of the partially-processed DAG is then transferred to the edge cluster and buffered in a queue (③), which will be processed by the *inference engine* inside the edge cluster (④) to complete the remaining unprocessed part of the DAG. All the inference results from the cameras are gathered at the edge cluster. In the control plane, the *system monitor* continuously monitors the workloads at each camera and edge cluster, and sends the collected workload information to the *workload adaptation controller* (ⓘ). Once the workload imbalance across cameras is detected, the *workload adaptation controller* triggers the *cross-camera workload balancer* (ⓘⓘa) to balance the workloads across cameras (ⓘⓘⓘa). Meanwhile, based on the current workloads at each camera and edge cluster, the *workload adaptation controller* adaptively identifies the optimal DAG partition point for each camera that balances the workload between each camera and the edge cluster. Such optimal DAG partition points are sent to the *DAG partitioner* (ⓘⓘb) for DAG partitioning, and the corresponding DAG partition result is sent to the *inference engine* at each camera (ⓘⓘⓘb).

In the following, we describe the three key components of Distream: *cross-camera workload balancer* (§1.3.2), *DAG partitioner* (§1.3.3), and *workload adaptation controller* (§1.3.4) in detail.

1.3.2 Balancing the Workloads across Smart Cameras

The first key component of Distream is the *cross-camera workload balancer*, which balances the workloads at the level of the extracted regions of interest (ROIs) across cameras by migrating part of the workloads from heavily loaded cameras to idle or lightly loaded ones (Figure 1.6). To achieve this, the *cross-camera workload balancer* needs to accommodate three key considerations:

- **Consideration#1: cross-camera workload correlation**. As also being observed by many existing works [40, 41, 38], workloads on nearby cameras may exhibit strong correlations due to mobility of objects of interest: a camera may have high workload within a short time period if its nearby cameras are experiencing high workloads. The *cross-camera workload balancer* needs to take such correlations into account to avoid migrating workloads to cameras that are going to experience high workloads.
- **Consideration#2: heterogeneous compute capacities**. Smart cameras may have heterogeneous onboard compute capacities caused by different generations of compute hardware. To prevent any camera from becoming the bottleneck, the *cross-camera workload balancer* needs to balance the workloads proportional to each camera's compute capacity.
- **Consideration#3: workload balancing overheads**. In practice, migrating workloads across cameras incurs overheads. The *cross-camera workload balancer* needs to take such overheads into account such that the overheads do not overshadow the benefits brought by load balancing.

The *cross-camera workload balancer* takes these three considerations into account, and formulates the task of cross-camera workload balancing as an optimization problem.

Formally, let N denote the total number of smart cameras, w_i denote the workload of camera i before load balancing, and let $\sum_j x_{ji}$ and $\sum_j x_{ij}$ denote the workloads that are moved into and moved out of camera i respectively. Therefore, the workload of camera i after load balancing can

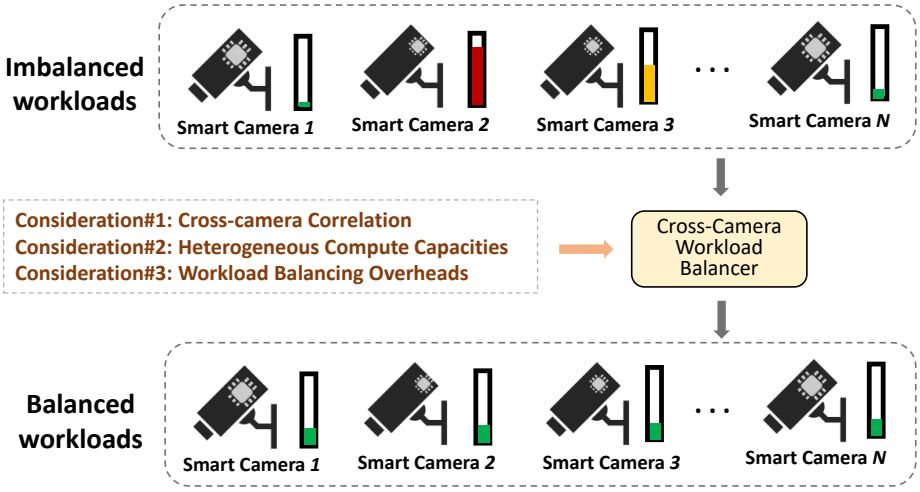


Figure 1.6: Overview of the cross-camera workload balancer.

be calculated as $u_i = (w_i + \sum_j x_{ji} - \sum_j x_{ij})$. And the goal of the *cross-camera workload balancer* is to minimize ν, the workload imbalance index across smart cameras after workload balancing, which can be calculated as

$$\nu = (\frac{u_{max}}{\bar{u}} - 1) \tag{1.1}$$

where u_{max} and $\bar{u}$ represent the maximum workload and average workload of all cameras respectively. By minimizing the workload imbalance index, the *cross-camera workload balancer* is able to achieve a good balance between workload balancing efficiency and fairness without incurring thrashing.

To accommodate **Consideration#1**, we take the cross-camera workload correlation into account by incorporating a workload predictor to predict the future workload at each camera (not from migration) based on their current workloads and the cross-camera workload correlations. Specifically, we employ long-short term memory (LSTM)-based recurrent neural network [61] as the workload predictor because it shows better long-term forecasting ability in predicting time series data compared to other methods [60]. The predicted workload $\hat{w}_i$ at camera i output by the workload predictor is added to u_i as an adjustment to obtain a more accurate future workload

estimate as

$$u_i = w_i + \sum_j x_{ji} - \sum_j x_{ij} + \hat{w}_i \tag{1.2}$$

To accommodate **Consideration#2**, we take the heterogeneous compute capacities of smart cameras into account by multiplying u_i with a normalization factor $a_i = C_i/\bar{C}$ where C_i is the compute capacity of camera i and $\bar{C}$ is the average compute capacity of all the smart cameras. As a result, the workload at each camera after load balancing gets normalized as $u'_i = u_i * a_i$ and the workload imbalance index ν gets normalized as

$$\nu' = (\frac{u'_{max}}{\bar{u'}} - 1) \tag{1.3}$$

To accommodate **Consideration#3**, a triggering threshold β is added into our optimization objective. The cross-camera workload balancing is not triggered when ν' is below β.

Putting all the pieces together, our cross-camera workload balancing scheme can be formulated as

$$\min_{x_{ij} \geq 0} \quad \max(\nu' - \beta, 0) \tag{1.4a}$$

$$s.t. \quad u_i = w_i + \sum_j x_{ji} - \sum_j x_{ij} + \hat{w}_i \tag{1.4b}$$

$$a_i = C_i/\bar{C} \tag{1.4c}$$

$$u'_i = u_i * a_i \tag{1.4d}$$

$$\nu' = (\frac{u'_{max}}{\bar{u'}} - 1) \tag{1.4e}$$

Solving the above nonlinear optimization problem is computationally hard. To enable real-time cross-camera workload balancing, we utilize a heuristic to efficiently solve the optimization

problem. Specifically, our heuristic starts with all $x_{ij} = 0$ and increases x_{ij} iteratively. In each iteration, we select cameras that have the largest and the smallest workloads to form a migration pair. Then we increase the migrated workload x_{ij} by Δx (i.e., $x_{ij} = x_{ij} + \Delta x$), where $\Delta x = 1\% * u'_i$. We repeat such iteration until v' is below a threshold or v' does not improve between two consecutive iterations.

As shown in §1.5.6, such heuristic is able to solve the optimization problem in an efficient manner and incurs negligible overheads.

1.3.3 Partitioning the Workload between Smart Cameras and Edge Cluster

The second key component of Distream is the *DAG partitioner*, which partitions the workloads between smart cameras and edge cluster at the DAG level. To achieve this, the *DAG partitioner* needs to accommodate two key considerations:

- **Consideration#1: DAG is conditionally executed**. As the contents of video streams are changing, the execution flow in DAG is changing correspondingly. Take the DAG in Figure 1.1 as an example: if a 'Vehicle' is detected, the upper path of the DAG (i.e., the 'Color', 'Type' and 'Make' classifiers) will be executed; if a 'Person' is detected, the middle path of the DAG (i.e., 'Behavior' and 'Gender' classifiers) will be executed. However, to identify the optimal DAG partition point, the *DAG partitioner* needs to consider all the possible partition points from all the possible execution paths.
- **Consideration#2: DAG Partitioning is by nature coarse-grained**. Ideally, to balance the workloads between cameras and the edge cluster, the DAG partition point should be set based on the workload ratio of the two sides. However, the possible partition points in a DAG are the vertices in the DAG, which are discrete and limited in number. This makes DAG partitioning

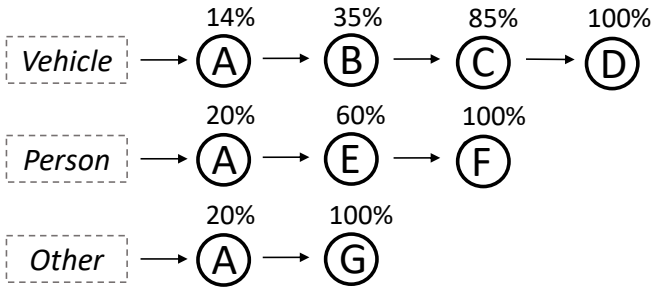


Figure 1.7: Full DAG profiling. The percentage on top of each vertex is the normalized accumulated inference cost.

by nature coarse-grained. As a result, the DAG partition point may not match the workload ratio of the two sides.

The *DAG partitioner* takes these two considerations into account, and designs a two-step scheme to partition the DAG.

Step#1: Full DAG Profiling. In our first step, we follow the topological order of the DAG to extract all the possible execution paths in the DAG. As an example, Figure 1.7 shows three execution paths extracted from the DAG in Figure 1.1. For each extracted execution path, we then profile the inference cost of each classifier, and label the normalized accumulated inference cost on top of each classifier. For example, in Figure 1.7, the normalized accumulated inference cost of executing the 'Vehicle' path up to classifier B is 35%.

Step#2: Stochastic DAG Partitioning. In our second step, we propose a stochastic DAG partitioning scheme to partition the DAG in a fine-grained manner such that the partitioned DAG could better match the workload ratio of the camera and the edge cluster. Specifically, for each execution path obtained in Step#1, the goal of our stochastic DAG partitioning scheme is to find a *stochastic path execution plan* such that the expected normalized accumulated inference cost of executing the path matches the optimal DAG partition point $par \in [0, 1]$ provided by the *workload adaptation controller* (§1.3.4).

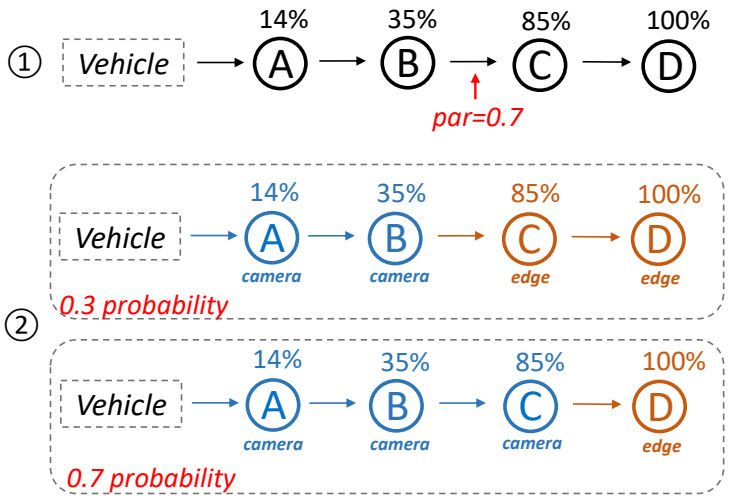


Figure 1.8: Stochastic DAG partition. Vertexes in blue color are classifiers allocated to run at the camera, and vertexes in brown color are classifiers allocated to run at the edge cluster.

Take the 'Vehicle' path in Figure 1.8 as an illustrative example. For the profiled 'Vehicle' path, assume we need to achieve $par = 0.7$ workload partitioning (i.e., 70% workload allocated to the camera and 30% workload allocated to the edge cluster) determined by the *workload adaptation controller*. Our stochastic DAG partitioning scheme first identifies two vertexes (vertex B, vertex C) along the 'Vehicle' path such that $par = 0.7$ falls between their normalized accumulated inference cost (Figure 1.8 (1)). Given the normalized accumulated inference costs of these two vertexes ($Cost_B = 0.35$ for vertex B, $Cost_C = 0.85$ for vertex C), our stochastic DAG partitioning scheme then determines the probabilities for partitioning the 'Vehicle' path at vertex B (P_B) and vertex C (P_C) respectively as:

$$\begin{aligned} P_B &= (par - Cost_C)/(Cost_B - Cost_C) \\ P_C &= 1 - P_B \end{aligned} \tag{1.5}$$

In this example, the solutions are $P_B = 0.3$ and $P_C = 0.7$. Therefore, the *stochastic path execution plan* is to partition the path at vertex B with a probability of 0.3 and to partition the

path at vertex C with a probability of 0.7 (Figure 1.8 (2)). As such, the expected normalized accumulated inference cost of executing the path matches the optimal DAG partition point *par* ($0.3 * 35\% + 0.7 * 85\% = 0.7$).

Compared to the discrete DAG partitioning approach, our stochastic DAG partitioning scheme provides much more partition flexibility and granularity. With such flexibility and granularity, Distream is able to better partition the DAG between smart cameras and edge cluster to match the workload ratio of two sides.

1.3.4 Workload Adaptation Controller

The third key component of Distream is the *workload adaptation controller*, which is the core to achieve our workload adaptation objective. The *workload adaptation controller* performs two tasks. First, when the workload imbalance index across smart cameras is greater than a threshold β, the *workload adaptation controller* triggers the *cross-camera workload balancer* to balance the workloads across the cameras as described in §1.3.2. Second, the *workload adaptation controller* follows a camera-cluster workload partition schedule to periodically (at interval γ) update the optimal DAG partition point for each camera to balance the workloads between each camera and edge cluster (Figure 1.9). Such optimal partition points are sent to the *DAG partitioner* for DAG partitioning as described in §1.3.3.

The DAG partition points serve as knobs to control the DAG execution across cameras and edge cluster, and thus control the workload split between them. To identify the optimal DAG partition points, the *workload adaptation controller* needs to accommodate two key considerations:

- **Consideration#1: the optimal DAG partition point is different for each camera**. The smart cameras have heterogeneous compute capacities and the workload at each camera can be different due to the discretization of cross-camera workload balancing. This requires the *workload*

adaptation controller to identify the optimal DAG partition point for each camera.

- **Consideration#2: the optimal DAG partition points of all cameras need to be determined jointly**. Because the workloads are partitioned between cameras and edge cluster, the system performance (throughput and latency) is *jointly* determined by all cameras and edge cluster. Therefore, the optimal DAG partition points for all cameras are linked together and need to be jointly adjusted as well.

The *workload adaptation controller* takes these two considerations into account, and formulates the task of identifying optimal DAG partition points as an optimization problem.

Specifically, our optimization problem aims to jointly identify the optimal DAG partition points for all the cameras with the objective to maximize the overall system throughput subject to the latency service level objective (SLO) imposed by the live video analytics applications. Below, we describe how we model the throughput and the latency followed by the complete optimization formulation.

Throughput. As a DAG is partitioned into two parts and executed on a camera and edge cluster respectively, the throughput of the entire live video analytics system is the sum of the throughput of cameras and the throughput of the edge cluster.

Formally, let N denote the number of cameras in the system, par_i denote the DAG partition point for camera i, and $TP^{cam_i}(par_i)$ denote the throughput of camera i given DAG partition point par_i, and TP^{edge} denote the throughput at the edge cluster which depends on the all partition points $\{par_i\}$. $TP^{cam_i}(par_i)$ and TP^{edge} can be accurately measured by monitoring the number of workloads processed per time unit at camera i and edge cluster respectively. Therefore, the throughput of N cameras TP^{cams}, the throughput of the edge cluster TP^{edge}, and the throughput of the entire system TP^{system} can be computed as:

$$TP^{cams} = \sum_{i=1}^{N} TP^{cam_i}(par_i) \tag{1.6a}$$

$$TP^{edge} = TP^{edge}(par_1, par_2, ..., par_N) \tag{1.6b}$$

$$TP^{system} = TP^{cams} + TP^{edge} \tag{1.6c}$$

Latency. The latency of a live video analytics system is defined as the time elapsed between receiving a workload and producing the inference result of the workload. Given that, the latency is the sum of two parts: 1) the workload queuing time, and 2) the workload processing time for producing the inference result. Because workloads are partitioned across camera and edge, the latency of two sides is thus computed separately. As such, the latency at camera i and at edge cluster are computed as:

$$L^{cam_i} = \frac{w_i}{TP^{cam_i}} + \frac{1}{TP^{cam_i}} \tag{1.7a}$$

$$L^{edge} = \frac{w_{edge}}{TP^{edge}} + \frac{1}{TP^{edge}} \tag{1.7b}$$

where w_i is the number of pending workloads to be processed at camera i, and w_{edge} is the number of pending workloads to be processed at the edge cluster.

Complete Optimization Formulation. Putting all the pieces together, the complete optimization problem can be formulated as[3]:

[3]Distream operates at DAG node level when partitioning workloads between cameras and edge cluster. Defining the optimization objective in terms of DAG node is able to reflect the performance gain in a more straightforward manner.

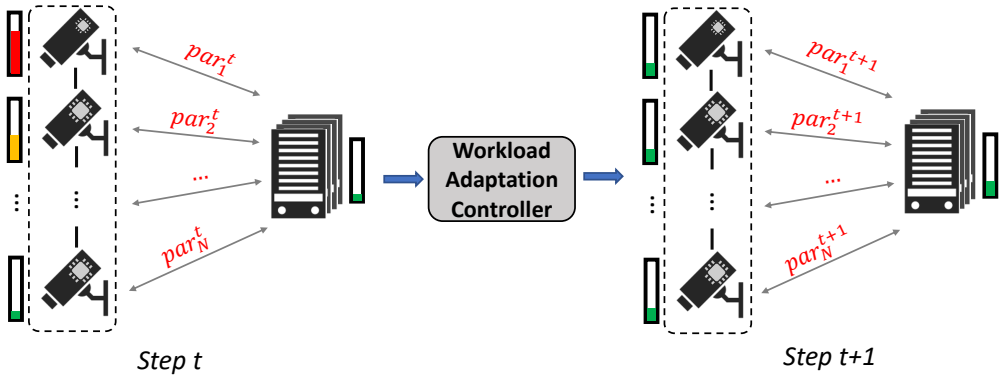


Figure 1.9: Overview of camera-cluster workload partition in workload adaptation controller. The optimal DAG partition points of all cameras are jointly determined.

$$\max_{par_i \forall i} \quad TP^{system} \tag{1.8a}$$

$$s.t. \quad L^{cam_i} \leq L_{Max}, \forall i \tag{1.8b}$$

$$L^{edge} \leq L_{Max} \tag{1.8c}$$

where L_{Max} is the latency SLO imposed by the specific application.

As shown, since L^{cam_i} and L^{edge} are determined by TP^{cam_i}, w_i, TP^{edge} and w_{edge}, which are determined by par_i, the workloads received at cameras and the edge cluster are not independent but negatively correlated with each other. This tradeoff illustrates the significance of identifying the optimal DAG partition points to balance workloads between cameras and the edge cluster to maximize the throughput under the latency SLO.

The optimization problem formulated above is non-convex. Instead of exhaustively searching over all possible partition points for all cameras, we adopt a heuristic to solve the optimization problem efficiently. Specifically, our heuristic is iterative and each iteration has two phases. In

the first phase, it starts with the objective of finding the preliminary DAG partition points that maximize the throughput without considering the latency SLO. In the second phase, we iteratively target the bottleneck camera with the largest latency and adjust its preliminary DAG partition point to meet the latency SLO. In practice, our heuristic is highly effective with negligible overheads even if the number of cameras scales up (§1.5.6).

1.4 System Implementation

We implemented Distream using about 2500 lines of Golang code and 500 lines of python code. In this section, we provide details on how the key parts of Distream were implemented.

Testbed. We followed the design choice that is widely adopted by existing live video analytics systems in real-world deployments to develop our own testbed to evaluate the performance of Distream. Specifically, our testbed consists of 24 smart cameras and a single edge cluster. Among the 24 cameras, 18 were prototyped using Nvidia Jetson TX1 [3] while the other six were prototyped using Nvidia Jetson TX2 [6]. Both Jetson TX1 and TX2 are designed for embedded systems with onboard DL workloads[4]. Jetson TX2 is an upgraded version of Jetson TX1 with larger compute capacity. For the edge cluster, we use a desktop server equipped with a cluster of 4 Nvidia Titan X GPUs [1]. We followed the standard configuration of existing IP video surveillance systems to connect and set up the network bandwidth between smart cameras and edge cluster [5]. Specifically, all the 24 smart cameras and the edge cluster are wire-powered and interconnected through a single switch (D-Link DGS-1510-28X [2]) to form a local network. Each smart camera has a Fast Ethernet link (10/100 Mbps) to the switch, and the bandwidth from the switch to the edge cluster is 10Gbps. By default, we set the camera bandwidth to 10Mbps. We also evaluate the system

[4] As AI chipsets evolve, we conjecture that AI hardware with such compute capacity will be widely deployed inside various edge devices such as smart cameras.

performance under 50Mbps.

LSTM Workload Predictor. We use a two-layer LSTM with 256 neurons in each layer as our workload predictor. Specifically, the workload predictor takes the arrived workloads of all cameras as input and predicts the upcoming workloads of each camera in the next second. We observed that the amount of workloads generated at each camera is highly relevant to the time period in a day. Thus we divide a day into six periods as 00:00-04:00, 04:00-8:00, 8:00-12:00, 12:00-16:00, 16:00-20:00 and 20:00-24:00. For each period, we train a specialized LSTM using workloads observed from that period. We find that using six specialized LSTMs reduces 10.2% mean square error compared to using a universal LSTM. Since running a specialized LSTM is fast on CPU (less than 1ms for one step), and it only makes predictions each second, our LSTM workload predictor incurs negligible overheads. Although the predictor could potentially benefit from periodic re-training given the dynamics of workloads over time, it is not the focus of this work.

Video Analytics Pipeline. We use OpenCV 3.2.0 to read video streams and implemented the Gaussian mixture model-based background subtraction method in [4] to extract regions of interest (ROIs) with length of history set to 500 and threshold set to 60. It maintains an estimate of background image and uses subtraction operation to extract foreground regions, followed by blob detection operations to extract ROIs. The number of extracted ROIs per frame varies depending on the nature of the scenes captured by the cameras, ranging from 0 to 30. For *inference engine*, we designed an efficient DNN model based on MobileNetV2 [70] for each classifier, and we are able to compress over 90% of model weights using pruning [58] and knowledge distillation [30] without accuracy loss. Given that each DNN model has only about 2MB memory footprint, the *inference engine* is able to load all the DNN models into a single GPU, avoiding model loading or switching time. Similar to [71], we adopt batched inference [15] to improve video analytics

throughput. We profiled the inference latency with batch size of 1, 8, 16, 32 and 64 respectively and set the batch size to 8 at the camera side and 32 at the edge cluster given its better performance tradeoff.

Scheduling. We implemented a monitor proxy at each camera and edge cluster to keep track of the runtime information including workload status, network bandwidth, throughput and latency. The proxy periodically reports these information to the *system monitor* in the edge cluster every 50ms. When performing the cross-camera workload balancing, the ROI is encapsulated into the migrating workload because the target camera does not have the video frame from the source camera.

Communication. Distream uses ZeroMQ [8] for high-speed low-cost inter-process communication among cameras. For remote communication between cameras and edge cluster, we implemented a low-cost remote procedure call (RPC) to transfer control data. Specifically, the cross-camera workload balancing instructions and DAG partition points are transferred to cameras immediately after they are generated by the *cross-camera workload balancer* and the *DAG partitioner* respectively. When performing cross-camera workload balancing and camera-cluster workload partitioning, we use batched RPCs for communication between two entities to amortize the communication overhead. The maximum RPC batch size is 20.

1.5 Evaluation

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

- **Q1 (§1.5.2)**: *Does Distream outperform status quo live video analytics systems? If so, what are the reasons?*
- **Q2 (§1.5.3)**: *How effective is each core technique incorporated in the design of Distream?*

- **Q3 (§1.5.4)**: *How is the performance of Distream affected by the system hyper-parameters and network bandwidth?*
- **Q4 (§1.5.5)**: *Does Distream scale well?*
- **Q5 (§1.5.6)**: *How much overheads does Distream incur?*
- **Q6 (§1.5.7)**: *Do the techniques proposed in Distream also benefit status quo live video analytics systems?*

1.5.1 Experimental Methodology

Applications. We use Traffic Monitoring and Campus Surveillance as two representative applications to evaluate Distream. For each application, we use a representative real-world multi-camera dataset to benchmark the performance.

- **Application#1: Traffic Monitoring.** For this application, we use publicly available live video streams from six traffic cameras deployed at different traffic intersections and roads in Jackson Hole, WY [7]. These six live video streams have dynamic contents, reflecting diverse traffic conditions in the city across space and time. To obtain a representative dataset, we sampled 200 5-minute video clips across 48 hours of video streams from each of the six cameras at a frame rate of 15 FPS and 1280×720P frame resolution. The sampled video clips cover diverse traffic conditions at different time of the day including rush hours, light traffic time, and night time.
- **Application#2: Campus Surveillance.** Due to lack of publicly available large-scale datasets, we self-collected a dataset from 24 surveillance cameras deployed on a university campus. These cameras cover diverse areas of the campus, including indoor areas such as dining halls, cafeterias, and hallways as well as outdoor areas such as university entrances, building en-

trances, squares, streets, and traffic intersections. For security purposes, some of the surveillance cameras are particularly covering areas where people rarely visit. Similar to the Traffic Monitoring application, we sampled 200 5-minute-long video clips across 48 hours of video streams from each of 24 cameras at a frame rate of 15 FPS and 1280×720P frame resolution to obtain a representative dataset.

Baselines. We compare Distream against three baselines that cover all three types of architectures listed in Table 1.1.

- **Centralized**. This baseline represents the centralized approach adopted by a number of existing live video analytics systems such as VideoStorm [86], Chameleon [40] and NoScope [44] where video streams are sent to edge cluster for processing.
- **Camera-Only**. This baseline represents the approach at the other end of the spectrum where video streams are only processed locally on smart cameras (e.g., FilterForward [13]).
- **VideoEdge-Lossless (VideoEdge-L)**. VideoEdge [35] is the status quo live video analytics system that utilizes compute resources across cameras and cluster to process video streams in a distributed manner. Different from Distream, VideoEdge is workload-agnostic and aims to achieve optimized resource-accuracy tradeoff, which may trade for resources with loss of accuracy. Since Distream does not sacrifice accuracy, for a fair comparison, we implemented VideoEdge with the resource-accuracy tradeoff disabled, which we refer to as VideoEdge-Lossless (VideoEdge-L) and use it as our third baseline. As VideoEdge is workload agnostic and the only prior knowledge given is the compute cost of each classifier in the DAG as well as the compute capacity ratio of cameras and cluster, we partitioned the DAG and place the partitions according to the camera-cluster compute ratio to implement VideoEdge-L.

Evaluation Metrics. We use three metrics to evaluate the performance of Distream and the base-

lines.

- **Throughput**. Live video analytics systems need to process streaming video frames in a continuous manner, and high-throughput processing is essential to keeping up with the incoming video streams. In our case, workloads are inferences involved in the DAG. Thus, we use the number of inferences processed per second (IPS) to measure the throughput.
- **Latency**. Live video analytics applications require producing analytics results within a short period of time. We thus use latency which is defined as the elapsed time from when the workload is generated to when the inference result of the workload is produced as our second metric to measure the system's responsiveness. As such, latency can be calculated as the sum of network latency, workload queuing time, and workload processing latency.
- **Latency Service Level Objective (SLO) Miss Rate**. Lastly, we measure the latency service level objective (SLO) miss rate, which quantifies the percent of the workloads that do not meet the latency requirement set by a live video analytics application. In this work, we set the latency SLO to 3 seconds, which is a reasonable requirement for live video analytics systems.

1.5.2 Overall Performance

We begin with comparing the overall performance of Distream and baselines on both applications. Since the Traffic Monitoring dataset involves six video streams, we use six smart cameras (4 TX1 and 2 TX2) with each allocated to one video stream and incorporate one GPU in the edge cluster. We will scale to 24 smart cameras and 4-GPU edge cluster when evaluating the scaling performance of Distream in §1.5.5.

Throughput and Latency. Table 1.2 and Table 1.3 list the throughput and latency of Distream and baselines on the Campus Surveillance and Traffic Monitoring application respectively. Specif-

	Throughput				Latency			
	avg	med	75th	99th	avg	med	75th	99th
Distream	**2.9**×	**3.1**×	**2.9**×	**2.5**×	**128.2**×	**189.3**×	**147.9**×	**112**×
VideoEdge-L	2.1×	2.2×	1.9×	1.5×	1.5×	2.4×	1.6×	1.4×
Centralized	1.9×	2.0×	1.7×	1.3×	1.4×	2.2×	1.5×	1.2×

Table 1.2: Performance gain over Camera-Only (Campus Surveillance).

ically, we report the average, 50th (median), 75th, and 99th percentile of throughput and latency performance gain over Camera-Only.

We have three major observations. First, Distream has achieved higher throughput than the baselines. Specifically, it achieves 2.9× and 1.6× average throughput gain on two applications, while the best baseline (VideoEdge-L) only achieves 2.1× and 1.2×. Second, Distream achieves higher peak throughput (99th) than the baselines. The higher peak throughput indicates that Distream is able to better utilize the distributed compute resources to survive through the workload bursts such as the ones illustrated in Figure 1.2. Third, Distream archives significant latency reduction compared to the baselines. In particular, Distream achieves 128.2× and 184× average latency reduction, while VideoEdge-L only achieves 1.5×.

Figure 1.10 and Figure 1.11 provides a more comprehensive view of the comparison by plotting the full distribution of throughput and latency, including the 1st, 25th, 50th (median), 75th and 99th percentiles in the box plot. In terms of throughput, Distream has a much wider throughput range, which indicates that Distream is able to handle a much wider range of amounts of workloads than the baselines. In terms of latency, the average latency of Distream for the Campus Surveillance and Traffic Monitoring application is 0.34s and 0.27s, which is much less than the baselines, which ranges from 28.54s to 43.93s for Campus Surveillance and 9.98s to 49.68s for Traffic Monitoring. This result indicates that the throughput improvement in Distream does not come at the cost of sacrificing its latency.

	Throughput				Latency			
	avg	med	75th	99th	avg	med	75th	99th
Distream	**1.6**×	**1.6**×	**1.9**×	**2.0**×	**184**×	**604.3**×	**446.8**×	**52.6**×
VideoEdge-L	1.2×	1.2×	1.2×	1.2×	1.5×	1.8×	1.5×	1.2×
Centralized	1.1×	1.1×	1.1×	1.1×	1.2×	1.4×	1.2×	1.1×

Table 1.3: Performance gain over Camera-Only (Traffic Monitoring).

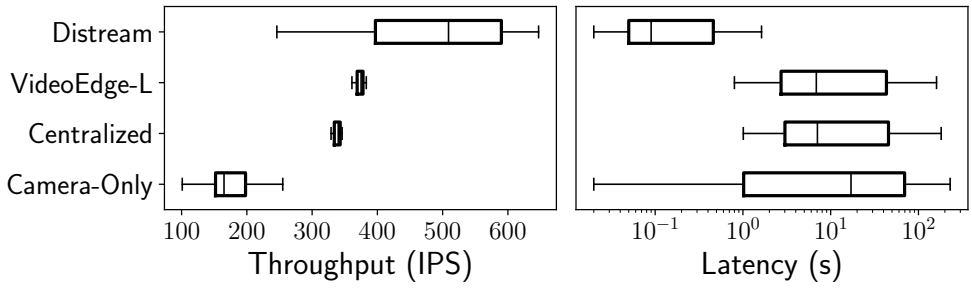


Figure 1.10: Throughput and latency distribution on Campus Surveillance.

Latency SLO Miss Rate. Figure 1.12 compares the latency SLO miss rate of Distream against baselines. As shown, Distream outperforms the baselines by a large margin. In particular, Distream is able to achieve a near-zero latency SLO miss rate (0.7% and 1.5%) on the Campus Surveillance and Traffic Monitoring application respectively, while the baselines have much higher miss rates (at least 60.7% Campus Surveillance and 93.8% for Traffic Monitoring).

Why Distream Outperforms the Baselines? Distream outperforms both Centralized and Camera-Only because Distream is able to leverage the compute resources at both cameras and edge cluster sides while Centralized and Camera-Only could not. For VideoEdge-L, Distream is able to achieve better performance for the following two reasons:

Reason#1: Higher Resource Utilization. In terms of throughput, Distream outperforms the baselines because it is able to utilize the idle compute resources at both cameras and the edge cluster to process the workloads to enhance the throughput. To see this, we continuously track the workload imbalance index across smart cameras and the edge cluster. As shown in Figure 1.13 (a), Distream is able to achieve a near-zero workload imbalance for about 60% of the time. In contrast,

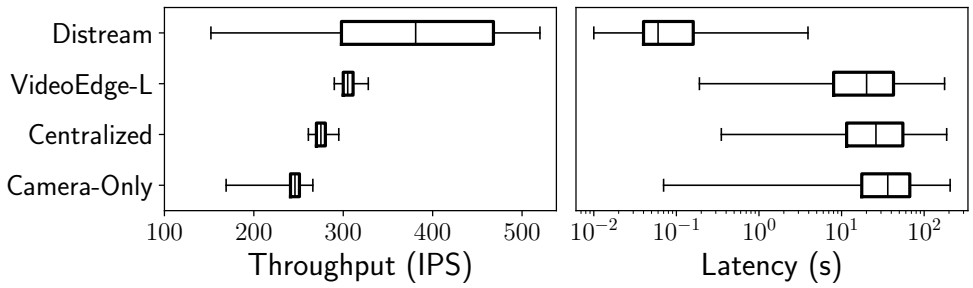


Figure 1.11: Throughput and latency distribution on Traffic Monitoring application.

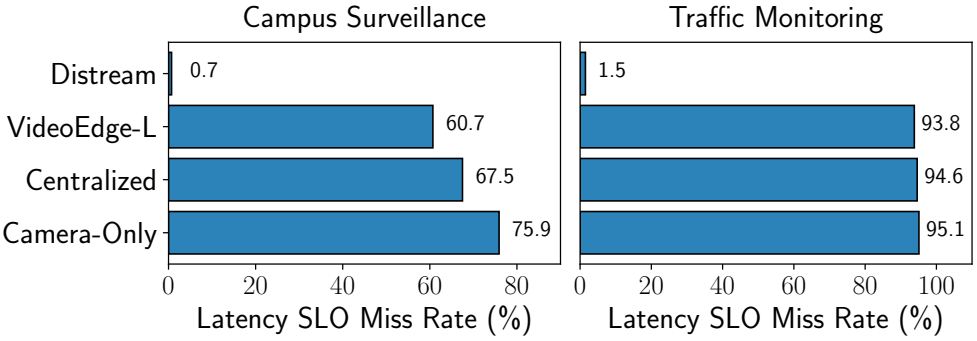


Figure 1.12: Latency SLO miss rate comparison.

VideoEdge-L is experiencing significant workload imbalance (more than 200% imbalance index) for 90% of the time for being agnostic to the dynamics of the workloads.

Reason#2: Less Accumulated Workloads. In terms of latency, Distream outperforms the baselines because with higher compute resource utilization, Distream is able to process the workloads at a much faster rate than the baselines, preventing the workloads from being accumulated at cameras or edge cluster. As such, Distream has much less workloads waiting in the queue, which significantly reduces the workload queuing time and hence the latency. To see this, we continuously track the accumulated workloads in the local queue at each camera and the edge cluster. As shown in Figure 1.13 (b), Distream is able to achieve a near-zero amount of accumulated workloads for about 80% of the time. In contrast, VideoEdge-L is experiencing significantly high amount of accumulated workloads for most of the time. As such, the workload queuing time in Distream is much lower, which is the root cause of its low latency.

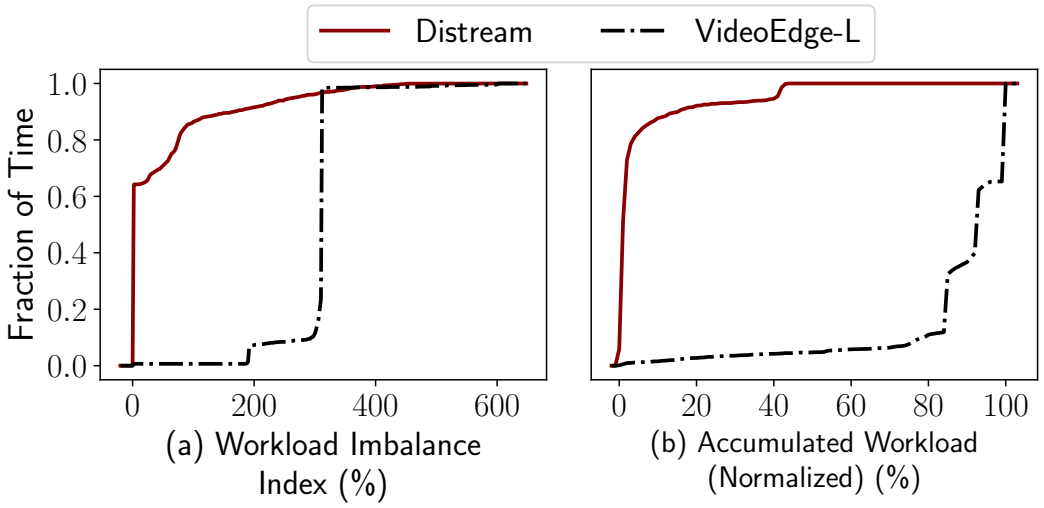


Figure 1.13: Illustration on why Distream outperforms baselines.

1.5.3 Component-wise Analysis

The design of Distream enables adaptive cross-camera workload balancing and adaptive camera-cluster workload partitioning. In this section, we implemented two breakdown versions of Distream to take a closer look at the contribution of each component.

- **Distream-L** has adaptive cross-camera workload balancing, but has static camera-cluster workload partitioning based on the compute capacity ratio of cameras and cluster.
- **Distream-P** has adaptive camera-cluster workload partitioning, but does not enable cross-camera workload balancing.

Table 1.4 shows the comparison results on the Campus Surveillance application. As shown, the performance gains brought by both Distream-L and Distream-P are similar and significant. On average, Distream-L achieves 2.4× throughput gain and 5.9× latency reduction respectively compared to Camera-Only. Distream-P achieve 2.5× throughput gain and 6.3× latency reduction respectively compared to Camera-Only. This result indicates the importance of both adaptive cross-camera workload balancing and adaptive camera-cluster workload partitioning to the whole

system. Distream is able to combine the advantages of both Distream-L and Distream-P to achieve better performance than using only one of them.

	Throughput				**Latency**			
	avg	med	75th	99th	avg	med	75th	99th
Distream-L	2.4×	2.5×	2.3×	2.2×	5.9×	8.2×	5.5×	5.2×
Distream-P	2.5×	2.5×	2.3×	2.1×	6.3×	9.2×	6.6×	5.1×

Table 1.4: Component-wise analysis of Distream. Performance gains are over Camera-Only.

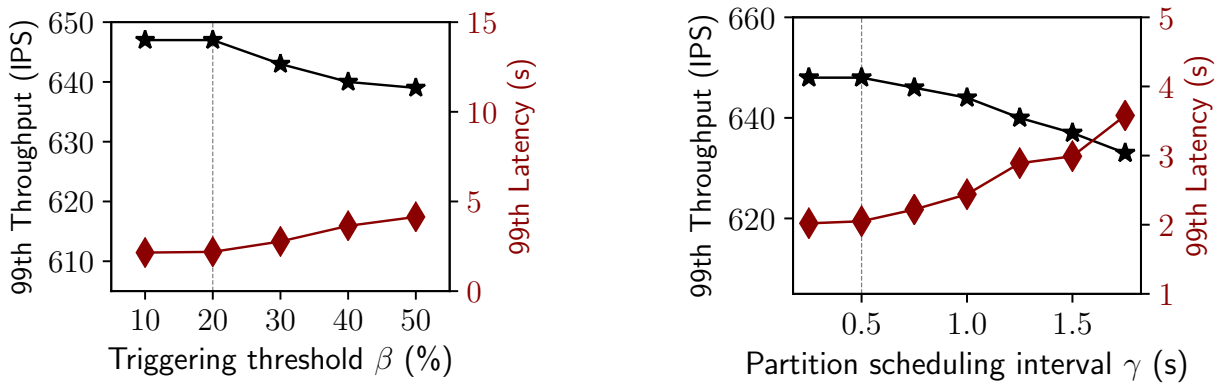
1.5.4 Sensitivity Analysis

The design of Distream involves two key system hyper-parameters: (i) the cross-camera workload balancing threshold β (§1.3.2), and (ii) the camera-cluster workload partition scheduling interval γ (§1.3.4). In this section, we evaluate the impact of these system hyper-parameters on the performance of Distream. In addition, since cross-camera workload balancing requires migrating workloads through the network, we also evaluate the impact of network bandwidth on the performance of Distream.

Impact of Cross-Camera Workload Balancing Threshold β. Figure 1.14 (a) shows Distream's sensitivity to the cross-camera workload balancing threshold β. We only show 99th percentile throughput (left y-axis, black) and 99th percentile latency (right y-axix, red) because they are sensitive to hyper-parameter changes and can better reflect the impact on the system performance. We observe that when β is below 20%, the performance of Distream does not change much. When β exceeds 20%, the 99th percentile throughput begins to drop and 99th latency increases. This is because a larger β would trigger load balancing less often. As a result, the system may suffer from performance loss due to workload imbalance. Therefore, we set β to be 20%.

Impact of Camera-Cluster Workload Partition Scheduling Interval γ. Figure 1.14 (b) shows Distream's sensitivity to the camera-cluster workload partition scheduling interval γ. We see that

the 99th percentile throughput and 99th percentile latency do not change much when the interval is less than 0.5s. When γ is between 0.5s to 1.5s, the system performance has slightly declined. When γ is greater than 1.5s, the system performance significantly declines as the interval increases. Therefore, we set γ to 0.5s.



(a) Impact of cross-camera workload balancing threshold β.

(b) Impact of camera-cluster workload partition schedule interval γ.

Figure 1.14: Impact of system hyper-parameters on the performance of Distream.

Impact of Network Bandwidth. We also examine Distream's sensitivity to network bandwidth. We evaluate Distream with network bandwidth at 10Mbps (low bandwidth) and 50Mbps (high bandwidth), which covers the bandwidth range in standard video surveillance systems on the market. Table 1.5 lists our evaluation results. As shown, both the throughput and latency are very similar between 10Mbps to 50Mbps. This result indicates that Distream is robust to network bandwidth changes and is able to achieve high performance even with a network bandwidth of 10Mbps.

	Throughput (IPS)				Latency (s)			
	avg	med	75th	99th	avg	med	75th	99th
Low Bandwidth (10Mbps)	489.1	509	590	647	0.34	0.09	0.47	2.05
High Bandwidth (50Mbps)	490.9	510	593	650	0.31	0.07	0.41	1.75

Table 1.5: Performance of Distream under low (10Mbps) and high (50Mbps) network bandwidths.

1.5.5 Scaling Performance

To evaluate the scaling performance of Distream, we use the Campus Surveillance dataset and examine the throughput and latency as Distream scales up its number of smart cameras from 6 to 12, 18, and 24. To match the compute resources with new workloads brought by the new cameras, we add one GPU to the edge cluster whenever six smart cameras are added to the system.

Figure 1.15 illustrates the scaling performance of Distream in terms of throughput (upper) and latency (lower). As shown, Distream is able to scale its throughput up nearly linearly. Specifically, Distream achieves a 3.95× average throughput from 489.1 IPS when processing videos streamed from 6 cameras to 1931.4 IPS when processing videos streamed from 24 cameras. Meanwhile, Distream is able to maintain a similar low latency when scaling up.

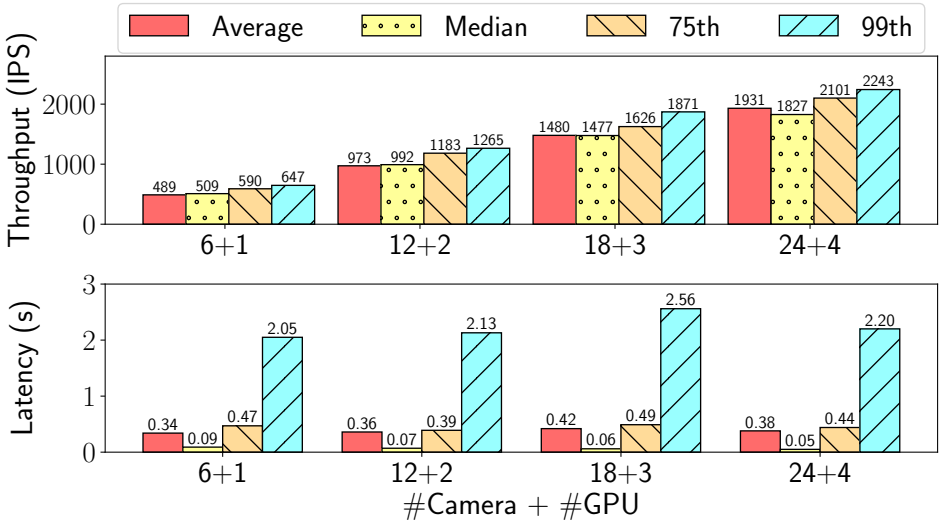


Figure 1.15: Scaling performance of Distream.

1.5.6 System Overheads

The system overheads incurred by the design of Distream come from two sources. The first source comes from solving the optimization problems for cross-camera workload balancing in §1.3.2 and identifying the optimal DAG partition points in §1.3.4. Their overhead is 16us and 4us with six cameras and one GPU, and 21us and 8us when scaled up to 24 cameras and 4 GPUs. This result indicates that our proposed heuristics are highly efficient and incur negligible overheads. The second source of overhead comes from the RPC call for migrating workloads across smart cameras (§1.3.4). As shown in Table 1.6, even if under the low bandwidth condition (i.e., 10Mbps), each RPC call only takes 2.1ms on average for workload migration. When we employ batching RPC, this overhead is amortized when the RPC batch size increases, making the overheads of cross-camera workload balancing negligible.

Batch Size	Total Migration Time	Avg RPC Overhead
1	2.1ms	2.1ms
10	7.4ms	0.74ms
20	9.2ms	0.46ms

Table 1.6: Cross-camera workload balancing overheads of Distream.

1.5.7 Benefits to Existing Systems

As summarized in Table 1.1, existing live video analytics systems are agnostic to the workload dynamics in real-world deployments. Instead, they focus on designing different techniques to optimize the resource-accuracy tradeoff of live video analytics systems. In this section, we add the workload adaptation techniques proposed in Distream onto VideoEdge [35] to examine how Distream could enhance its performance. We select VideoEdge instead of other status quo live video analytics systems listed in Table 1.1 because VideoEdge is also a fully distributed framework that

partitions the video analytics pipeline across cameras and cluster.

Figure 1.16 compares the performance between VideoEdge and VideoEdge + Distream. As illustrated, VideoEdge + Distream is able to generate a better resource-accuracy trade-off curve than VideoEdge. In particular, Distream is able to boost the accuracy by 21% when compute resource is limited, and is able to boost the accuracy by 7% when compute resource is adequate. Such improvement is made possible due to the higher resource utilization brought by the workload adaptation mechanism of Distream. With higher resource utilization, VideoEdge does not need to sacrifice as much accuracy as before to trade for resources.

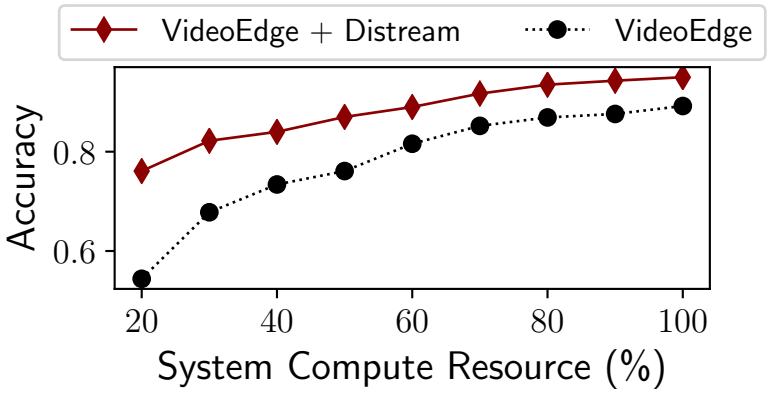


Figure 1.16: Resource-accuracy tradeoff curve comparison between VideoEdge and VideoEdge + Distream.

1.6 Related Work

Live video analytics is a killer application not only for mobile vision [36, 21, 20, 42] but also for large-scale distributed settings. Our work is closely related to DL-based live video analytics systems with distributed cameras. A majority of these systems are designed to process video streams in a centralized cluster. For example, Focus [32] customizes small DL classifiers to generate object indexes in each video frame to enable fast offline video query. NoScope [44] uses specialized DL

models to reduce inference overheads for throughput gain with small accuracy loss. Besides them, Chameleon [40], VideoStorm [86], and AWStream [85] leverage resource-accuracy tradeoffs for system optimization. Chameleon [40] reduces the overhead caused by searching for optimal tradeoff configuration. VideoStorm [86] exploits the variety of quality and lag goals in video analytics tasks and optimizes compute resources in GPU clusters. AWStream [85] is focused on adapting to network bandwidth variation to achieve low-latency high-accuracy wide-area streaming analytics. Different from these works, instead of trading accuracy for less compute resource consumption, Distream achieves high throughput and low latency by maximizing the resource utilization across cameras and edge cluster without sacrificing the accuracy.

The closest work to Distream is VideoEdge [35]. Although Distream and VideoEdge are both built upon a distributed architecture that involves smart cameras and edge cluster for live video analytics, they are different in three aspects. First, Distream focuses on enabling workload-adaptive live video analytics while VideoEdge is workload agnostic. Second, VideoEdge targets optimizing resource-accuracy tradeoff while Distream focuses on maximizing throughput and attaining latency SLO without compromising accuracy. Third, VideoEdge performs video analytics pipeline placement across cameras, edge and cloud where the dynamic network bandwidth is the main bottleneck. In contrast, Distream performs workload allocation across cameras and edge cluster based on workload dynamics to eliminate the bottleneck in compute resources. In fact, Distream is complementary to VideoEdge: as shown in §5.7, Distream is able to provide a better resource-accuracy tradeoff curve when integrated with VideoEdge.

1.7 Conclusion and Future Work

In this paper, we present the design, implementation, and evaluation of Distream, a distributed workload-adaptive live video analytics system based on the smart camera-edge cluster architecture. Distream addresses the key challenges brought by workload dynamics in real-world deployments, and contributes novel techniques that complement existing live video analytics systems. We have implemented Distream and conducted a rich set of experiments with a testbed consisting of 24 cameras and a 4-GPU edge cluster on two real-world distributed video datasets. Our experimental results show that Distream consistently outperforms the status quo in terms of throughput, latency, and latency SLO miss rate. Therefore, we believe Distream represents a significant contribution to enabling live video analytics at scale. In the current form, Distream treats cross-camera workload balancing and camera-cluster partitioning as two separate components. We plan to work on a joint solution as our future work. We will also work on designing common programming and network abstraction to support live video analytics application development based on our framework, and plan to expand our framework to the wireless settings.

Chapter 2

Collaborative Deep Learning Training on the Edge

2.1 Introduction

The recent past has witnessed the success of deep learning (DL) in a wide spectrum of areas including computer vision [18], automatic speech recognition [29], and natural language processing [76]. Part of this success is attributed to the highly efficient distributed training systems [51, 84, 87] that train DL models in parallel machines inside data centers owned by large organizations such as Google, Facebook, Amazon, and Microsoft. As intelligence is moving from data centers to devices, intelligent edge devices such as smartphones, drones, robots, and smart cameras are equipped with machine intelligence powered by deep learning to not only perform on-device inferences but also altogether train a DL model from the data collected by themselves (i.e., on-device distributed training).

Depending on the specific application settings, on-device distributed training can in general be classified into two categories: *federated learning*, and *collaborative learning*. In federated learning, data privacy is strictly enforced: during training, participating devices do not share their local data with each other. In addition, participating devices are usually located at different geographical locations and a different subset of the devices participate in each round of the training process. As

such, federated learning requires the support of a cloud server to coordinate the distributed training process and select which devices to participate in each round. In contrast, in collaborative learning, data privacy is not a concern given that the participating devices usually belong to the same individual, group, organization, or company. Moreover, participating devices are fixed throughout the training process and are located within a geographically close area. Thus they are able to communicate with each other via a local wireless network without the support of a cloud server.

In this work, we focus on collaborative learning. There are a wide range of real-world applications that are built upon collaborative learning. For example, in the U.S., each consumer on average owns more than three connected devices (e.g., smartphones, tablets, smart watches). In this setting, personal data are distributed across all those connected devices, and there is no privacy concern since all the devices are owned by the same person. With wireless network available at home or in the office, collaborative learning is the most practical solution to collaboratively learn a DL model from those distributed personal data.

Motivation & Limitations of Status Quo. Despite its considerable value, the key bottleneck of making collaborative learning systems 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 [81]. In contrast, in collaborative learning systems, communication between participating devices is conducted via wireless network such as Wi-Fi. The bandwidth of wireless network, however, is much more constrained, and can be *10,000 times less* than the bandwidth in data centers. Such limited bandwidth considerably slows down the communication and hence the overall training process.

Our *goal* in this work is to tackle this critical bottleneck to *enhance the training efficiency*

of on-device collaborative learning while retaining training correctness without compromising the training quality (i.e., accuracies of the trained models). Although a number of training acceleration approaches have been proposed [10, 55, 56, 25, 39, 33], as we will discuss in §2.2 in detail, these approaches either compromise the training quality to gain training efficiency or are designed for distributed training in data centers where they achieve limited training efficiency enhancement in on-device collaborative learning given the significant gap in network bandwidth between the data center setting and the on-device setting.

Overview of the Proposed Approach. The limitations of existing approaches motivate us to re-think the collaborative learning framework design by taking a different path from existing ones. To this end, we present Distream, an importance sampling-based framework that enables efficient on-device collaborative learning without compromising the training quality. In existing approaches, each participating device in each training iteration *randomly* samples a mini-batch from its 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 during model training. In practice, however, *not all the samples in the local data contribute equally to model training due to information overlapping among different samples*. Training on less important samples not only contributes little to model accuracy but also considerably slows down the training process. Given that, instead of random sampling, Distream focuses on samples that provide more important information in each iteration. By 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 whole training process.

The design of Distream 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 considerably 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 Distream should take bandwidth variation into account as well.

To address the first challenge, Distream incorporates a *group-wise importance computation and sampling scheme* 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, Distream incorporates an *importance-aware data resharding scheme* 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, Distream incorporates a *bandwidth-adaptive computation-communication scheduler* that schedules the execution of importance computation and data resharding in a bandwidth-adaptive manner to further improve the speedup by completely masking out the costs of importance sampling and data resharding.

Implementation & Evaluation Results. We implemented Distream using TensorFlow 1.10 and deployed it on a self-developed testbed that consists of 12 NVIDIA Jetson TX1 as edge devices. We evaluate Distream on six commonly used datasets using a diverse set of DL models across three most important task domains: computer vision, speech recognition, and natural language processing. We compare Distream against two status quo baselines TicTac [25] and AdaComm [78]. Our results show that Distream consistently outperforms the baselines in training efficiency, achieving 3.7×, 3.2×, 4.0×, 2.2×, 2.7×, 1.8× on the six datasets. Moreover, Distream is able to maintain training speedup as the number of participating devices scales up, and is robust to wireless bandwidth variations in real-world deployments.

In summary, we make three major contributions:

- To the best of our knowledge, Distream is the first efficient on-device collaborative learning framework based on importance sampling that 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 collaborative learning framework. Guided by the performance model, we propose three novel techniques, namely, 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 of Distream.
- We implemented Distream and deployed it on a self-developed testbed. We demonstrate its effectiveness and show that Distream consistently outperforms two status quo frameworks on six datasets across computer vision, speech recognition, and natural language processing.

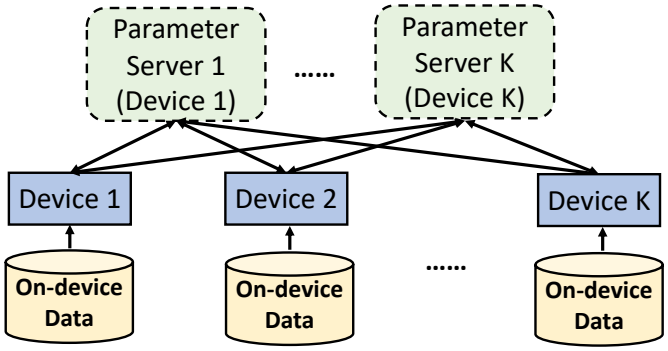


Figure 2.1: The multi-parameter server architecture of on-device collaborative learning systems.

2.2 Background and Motivation

In this section, we first introduce the architecture of on-device collaborative learning systems (§2.2.1). We then discuss the existing techniques for enhancing their training efficiency followed by explaining their limitations, which are the key motivation of this work (§2.2.2).

2.2.1 Architecture of On-Device Collaborative Learning Systems

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

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta_t \frac{1}{K} \sum_{k=1}^{K} \mathbf{g}^k \tag{2.1}$$

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

Modern distributed training systems use parameter server (PS) architecture for gradient aggre-

gation. In PS [87, 80, 52], 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 2.1, Distream adopts a multi-PS architecture and treats each end 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.

2.2.2 Existing Approaches and Their Limitations

The total training time of collaborative learning 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 edge device T_{cm} in each iteration as follows:

$$T = E \cdot (T_{cp} + T_{cm}). \tag{2.2}$$

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

Gradient Compression. To reduce T, one common approach is to reduce the communication time in each iteration T_{cm} in Eq. (2.2). This is achieved by quantizing gradients using smaller number of bits [81, 10, 55] or selecting important gradients to transfer via sparsification [79, 56, 33]. 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. (2.2) 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 and hence sacrifice the accuracy of the trained model, which contradicts the goal of this work.

Local-update SGD. In distributed SGD, global synchronization among nodes is performed in each iteration for exchanging and averaging gradient information, which is an expensive and lengthy operation in bandwidth-constrained network. A technique called local-update SGD [89, 19, 62, 78] is proposed to reduce the communication overhead. The idea is to allow each node to perform multiple local updates before global synchronization in each iteration. Similar to the gradient compression approach, although aggressively performing multiple local updates gains training efficiency by reducing communication iterations, it again compromises the accuracy of the trained model.

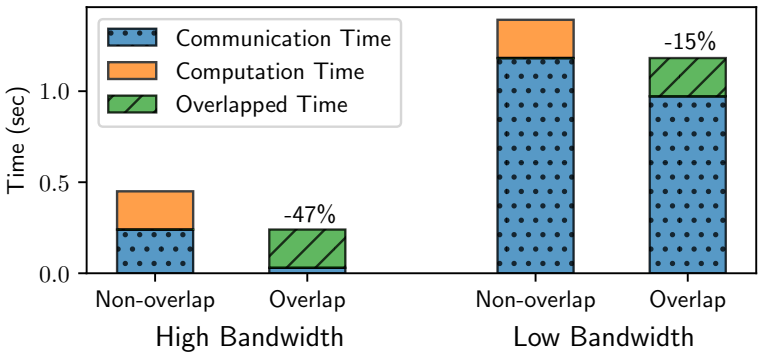
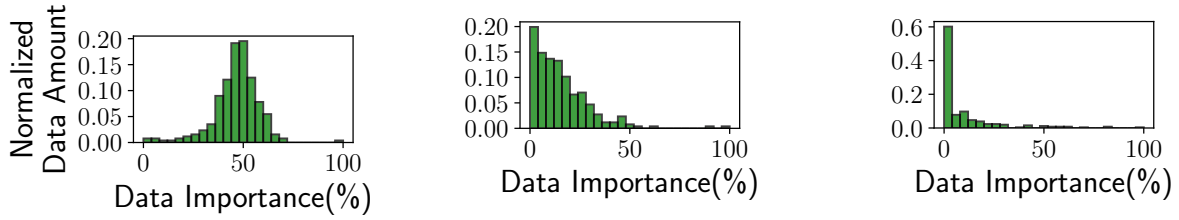


Figure 2.2: Comparison in communication-computation overlapping between high bandwidth and low bandwidth setting.

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 optimization technique can be seen as a mechanism to reduce the sum $(T_{cp} + T_{cm})$ in Eq. (2.2). Such reduction can be achieved either by identifying the optimal gradient transfer or-



(a) #iterations=0 (b) #iterations=200 (c) #iterations=600

Figure 2.3: Distributions of data importance as the number of iterations increases during training.

der [25, 39, 17, 87] or using staled model weights [54]. Although this computation-communication overlapping technique does not compromise the accuracy of the trained model, the limited bandwidth in on-device collaborative learning causes the communication-to-computation ratio to be much higher than the one in data center setting. As a result, the effectiveness of overlapping techniques significantly diminishes because it is no longer able to mask out the communication cost under such high communication-to-computation ratio. To demonstrate this, Figure 2.2 shows the cost breakdown of overlapping technique. As shown, although it provides significant cost reduction (47%) under high bandwidth network in data center setting, it only reduces 15% in low bandwidth network in on-device collaborative learning setting.

2.3 Distream Overview

The limitations of existing approaches motivate us to design Distream by taking a different path. In this section, we first introduce the key insight that Distream is designed upon (§2.3.1). We then introduce a rudimentary framework designed upon the key insight and its performance model (§2.3.2). Lastly, we briefly overview the techniques we design under the guidance of the performance model in Distream that transform the rudimentary framework into a highly efficient one for on-device collaborative learning (§2.3.3).

2.3.1 Key Insight

The key insight behind the design of Distream is to *exploit data efficiency to improve the training efficiency of on-device collaborative learning*. 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 core, assumes that each data sample in the local data is equally important during the whole model training process. In practice, however, *not all the data contribute equally to model training*. Some of them provide more information to the training and thus are more important, while the others provide less information and thus are less important. As an example, Figure 2.3 illustrates the importance distribution of the data in CIFAR-10 as the number of iterations increases during training. As shown, more and more data samples become less important as the training process proceeds. In particular, after 200 iterations, the importance of data quickly concentrates to a small portion of data samples and the rest contribute little to the model training. This observation sheds light on the low training efficiency of standard SGD where mini-batch is generated by random sampling.

Based on this insight, we propose to incorporate *importance sampling* [91] 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 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 $\mathbf{w}^t$ out for simplicity) should be multiplied

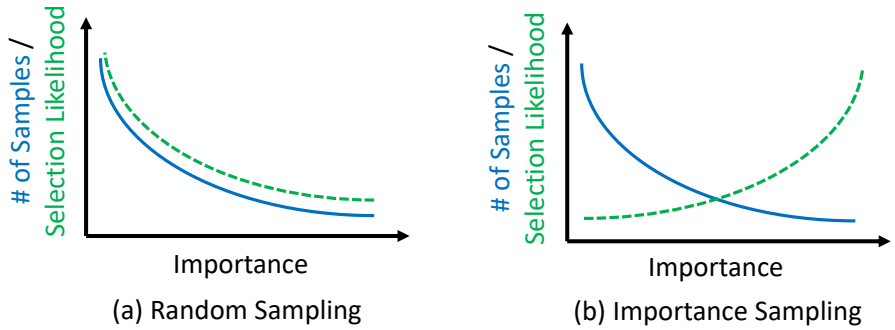


Figure 2.4: A conceptual comparison between (a) random sampling and (b) importance sampling.

by $p(x)/q(x)$ when shifting from random sampling to importance sampling [9]:

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

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 ($||\nabla l(x)||$) [9]:

$$q(x) \propto ||\nabla l(x)||.$$

The above derivation implies that *a sample's gradient norm can be used as an indicator of its importance*. The larger gradient norm a sample has, the more contribution the sample can make to the model training.

Figure 2.4 provides a conceptual comparison between random and importance sampling. The blue solid curve depicts the distribution of the importance values of 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 2.4a). In contrast, in importance sampling, samples with higher importance have higher likelihood to be selected (green dotted

curve in Figure 2.4b). 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 whole training process.

2.3.2 Rudimentary Framework and Performance Model

Inspired by the benefits brought by importance sampling, we propose an importance sampling-based framework for on-device collaborative learning. Our framework is built upon the standard distributed SGD. Figure 2.5a illustrates the operations involved in each SGD iteration. As shown, the only difference between them is that the importance sampling-based framework replaces *random sampling* (i.e., randomly select samples to construct the mini-batch) in the standard distributed SGD with two new operations: *importance computation* which computes the importance (i.e., gradient norm) for every sample in the local dataset, and *importance sampling* which selects samples based on their computed importance to construct the mini-batch.

The performance model of such 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 SGD is:

$$\begin{aligned} Speedup &= \frac{E \cdot (T_{cp} + T_{cm})}{E_{is} \cdot (T_{cp} + T_{cm} + T_{is})} \\ &= \frac{1}{\frac{E_{is}}{E} \cdot (1 + \frac{T_{is}}{T_{cp}+T_{cm}})}. \end{aligned} \tag{2.3}$$

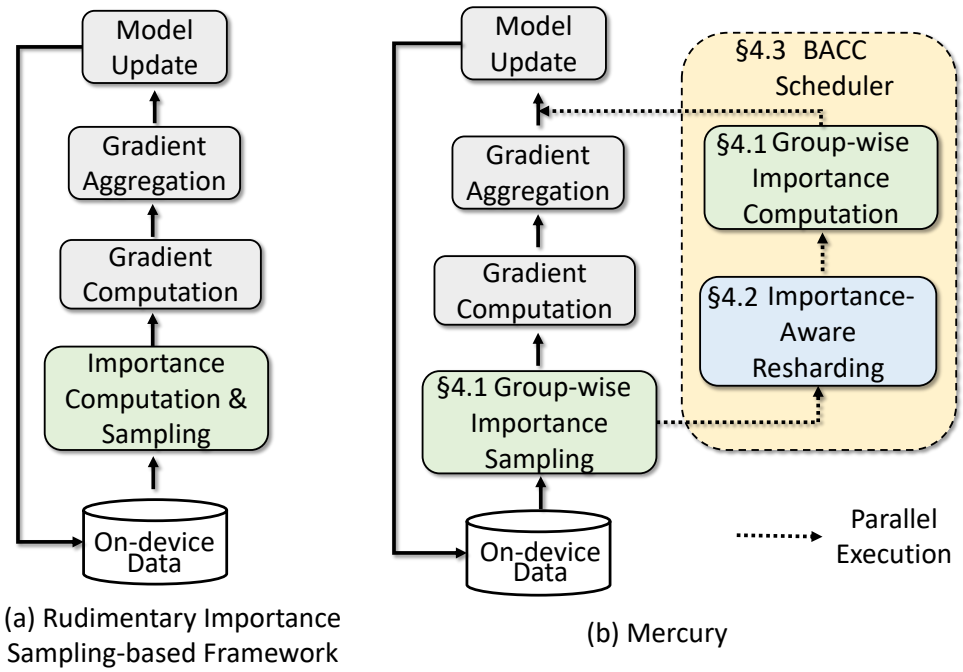


Figure 2.5: Conceptual Comparison between rudimentary importance sampling-based framework and Mercury.

As shown in Eq. (2.3), the importance sampling-based framework could perform worse than the standard random sampling-based distributed SGD (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., $E_{is} < E$).

2.3.3 Overall Design

Figure 2.5b illustrates the overall design of the proposed Distream framework. Compared to the rudimentary framework illustrated in Figure 2.5a, Distream incorporates three key innovations guided by the performance model in Eq. (2.3) which effectively avoid the pitfalls of the rudimentary framework. First, Distream incorporates a group-wise importance computation and sampling technique that considerably reduces the computation cost of importance sampling without com-

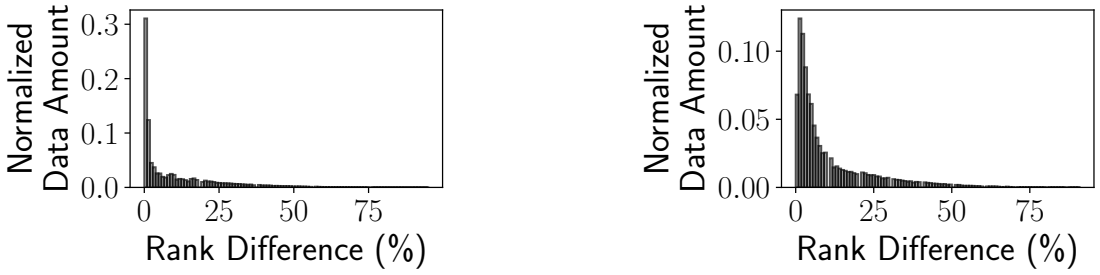
promising its effectiveness and training quality. Second, it incorporates an importance-aware data resharding technique for balancing importance distribution among edge devices to further accelerate the training process. Lastly, it incorporates a bandwidth-adaptive computation-communication scheduler 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 Distream is built upon.

In the following, we describe the details of these three techniques in §2.4.1, §2.4.2 and §2.4.3, and prove Distream's training correctness in §2.4.4.

2.4 Design Details

2.4.1 Group-wise Importance Computation and Sampling

The first key technique in Distream is *group-wise importance computation and sampling*: a technique that considerably reduces the computational cost of importance sampling without compromising its effectiveness and unbiased convergence. In the standard importance sampling-based framework introduced in §2.3.2, 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



(a) Distribution of rank difference after 1 iteration

(b) Distribution of rank difference after 10 iteration

Figure 2.6: Distributions of rank difference.

leads to costly computation. In contrast, our group-wise stochastic importance sampling 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 then 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 2.6 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% in importance ranking after one training iteration (Figure 2.6 (a)). Even after 10 iterations, about 65% of data samples change less than 10% in importance ranking (Figure 2.6 (b)). 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.

Based on this observation, our group-wise importance computation and sampling (Figure 2.7) consists of two steps:

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

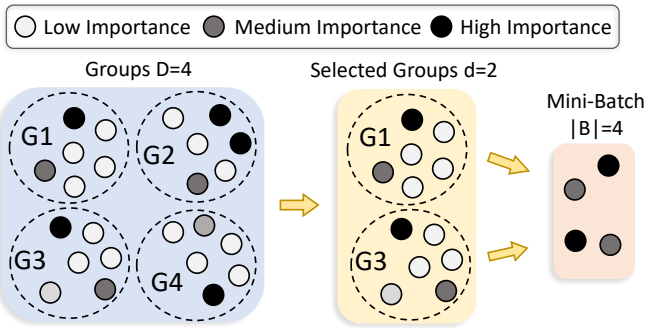


Figure 2.7: Illustration of group-wise importance computation and sampling. Each dot represents one sample and the darkness indicates its importance.

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 a 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:

$$r_i = \frac{\exp(\beta t_i)}{\sum_{n=1}^{D} \exp(\beta t_n)} \tag{2.4}$$

where t_i is the 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 $|\mathscr{B}| \times \frac{M_i}{\sum_{n=1}^{d} M_n}$ from each selected group i using importance sampling where M_i is the size of group i. To guarantee unbiased convergence, the probability of a sample being selected is proportional to its loss within its group. In other words, the probability of selecting sample j in group i is given by:

$$p_{i,j} = \frac{I_{i,j}}{\sum_{n=1}^{M_i} I_{i,n}} \tag{2.5}$$

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 selected groups. To ensure unbiasedness, we reweigh the gradients among samples when computing average gradients to obtain the final unbiased gradient as proposed in [46].

2.4.2 Importance-aware Data Resharding

The second key technique in Distream is *importance-aware data resharding*: a technique for balancing importance distribution among workers to accelerate the training process. As importance update is performed on the local dataset at each edge device, the local importance rank within each edge device does not reflect the global importance rank that accumulatively considers the importance of all the samples from all edge devices. As a consequence, an edge device may repeatedly learn globally trivial samples, which considerably lowers the training efficiency. This problem is exacerbated when data classes are not evenly distributed across edge devices, which is common in on-device settings. Such problem can be resolved by data resharding, a technique that randomly redistributes samples among workers. In data centers, data resharding 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 signif-

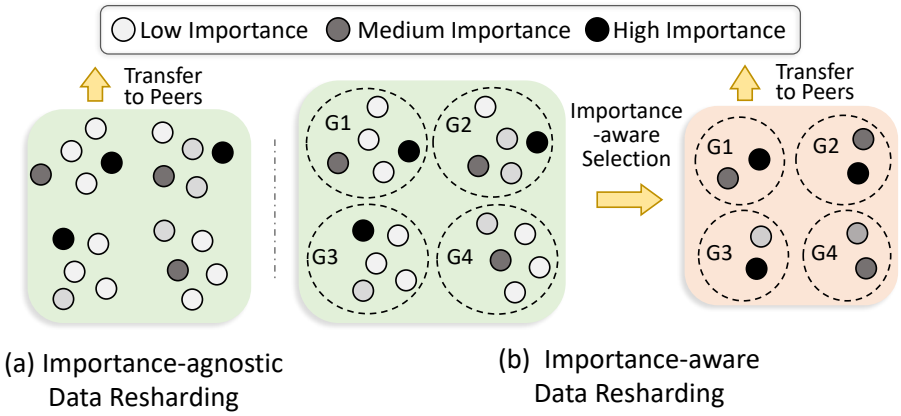


Figure 2.8: Importance-agnostic data resharding vs. importance-aware data resharding.

icantly delays the training process.

To this end, we propose importance-aware data resharding that only redistributes important samples. Instead of blindly shuffling data among workers, we only select non-trivial samples to shuffle to maximize data resharding efficiency with minimum communication overhead. As shown in Figure 2.8, suppose after data resharding is determined, a worker with N samples and D groups needs to swap $N_p < N$ samples with other workers 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. (2.5). As a result, we are able to select important samples to be redistributed without change in group size or local dataset size.

2.4.3 BACC Scheduler

A naive implementation of importance sampling for each device in collaborative learning is shown in Figure 2.9a. The importance computation and data resharding is sequentially inserted in the original training process, incurring unnecessary blocking and overhead.

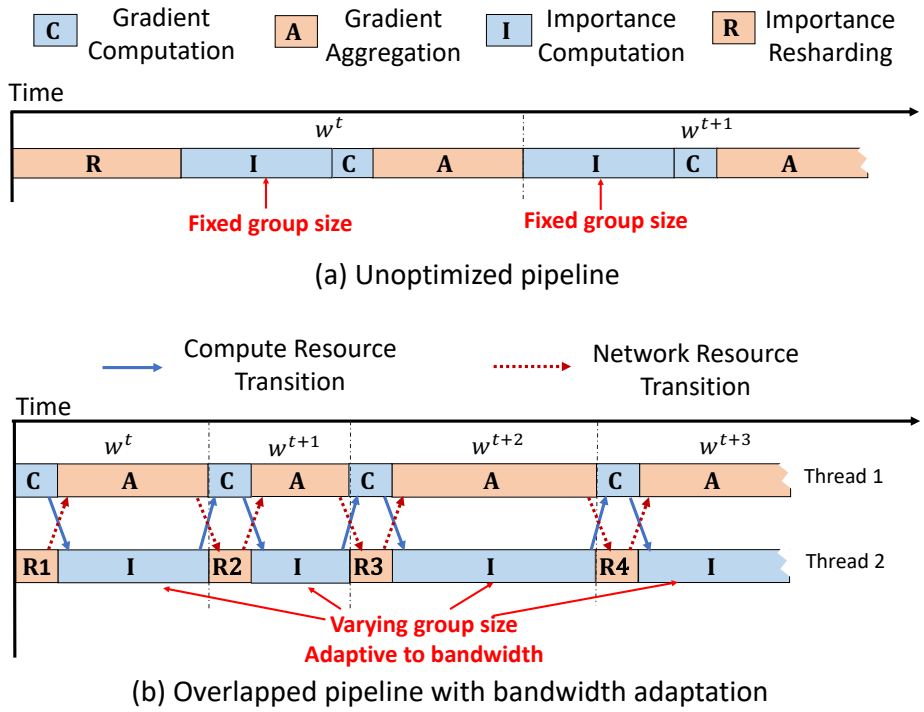


Figure 2.9: (a) Unoptimized Pipeline vs. (b) Optimized pipeline. w^t represents the model weight at iteration t. R = R1 + R2 + R3 + R4 in length.

As shown in Figure 2.9b, Distream 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 improve the speedup by completely masking out the costs of importance sampling and data resharding. Specifically, since gradient aggregation only uses network resource and 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 is 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, Distream 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 size and ensure unbiasedness, 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, Distream adopts 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 2.9b 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.

2.4.4 Proof of Training Correctness

Lastly, due to page limitation, we briefly provide the theoretical result showing that the collaborative learning under Distream is guaranteed to converge to the same solution as the standard

distributed SGD.

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 §2.4.1, 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 $\mathscr{V}$ for this bound.

2.5 Implementation

Testbed. We designed and developed our own testbed due to lack of off-the-shelf ones. Specifically, we use 12 NVIDIA Jetson TX1 as edge devices. Each TX1 has an integrated small form

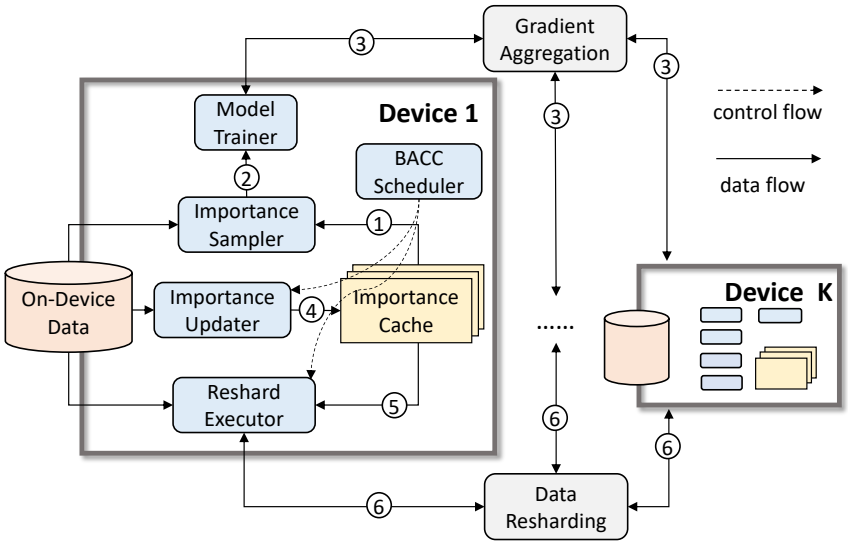


Figure 2.10: Distream implementation.

factor mobile GPU, which is designed for next-generation intelligent edge devices to execute DL workloads onboard. For wireless networking, we use Netgear Nighthawk X6S AC4000 Tri-band Wi-Fi routers to connect all the TX1, and use Linux tools *tc*, *qdisc*, and *iptables* to control the network bandwidth to conduct our experiments.

Framework Implementation. We implemented Distream using TensorFlow 1.10. Figure 2.10 shows the implementation of our Distream framework. Distream is distributed training framework that spans across multiple edge devices. In each training iteration, *Importance Sampler* 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 fed into *Model Trainer* to compute local gradient (②). The local gradients from all devices are aggregated and the aggregated gradients are sent to *Model Trainer* to update the model (③). Meanwhile, *Importance Updater* computes the data importance and updates *Importance Cache* (④) when the device is performing gradient aggregation. *Reshard Executor* identifies important data samples from *Importance Cache* (⑤) and communicates with other devices to perform importance-aware data resharding (⑥) when the device is performing gradient computation. The executions of *Re-*

shard Executor and *Importance Updater* are triggered and scheduled by the *BACC Scheduler* at runtime.

2.6 Evaluation

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

- **Q1 (§2.6.2):** *Does Distream outperform status quo? If so, what are the reasons?*
- **Q2 (§2.6.3):** *How effective is each core technique incorporated in the design of Distream?*
- **Q3 (§2.6.4):** *Can Distream adapt to wireless network bandwidth variation well?*
- **Q4 (§2.6.5):** *Does Distream scale well when the number of edge devices increases?*

2.6.1 Experimental Methodology

Task Domains, Datasets, and DL Models. To demonstrate the generality of Distream across domains, datasets, and DL models, we evaluate Distream on six commonly used datasets using a diverse set of DL models across three most important task domains: computer vision, speech recognition, and natural language processing.

- **Computer Vision.** In the domain of computer vision, we use four datasets. Specifically, we select CIFAR-10, CIFAR-100, and SVHN as they are three of the most commonly used computer vision datasets for deep learning algorithm evaluation. Both CIFAR-10 and CIFAR-100 [48] consist of $50,000$ training images and $10,000$ test image in 10 classes. SVHN [82] consists of $73,257$ training and $26,032$ test images. We use ResNet18 [28] to train on CIFAR-10 and SVHN training, and use RetNet50 [28] to train on CIFAR-100. In addition, we select

AID [83] as our fourth computer vision 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 collaborative learning across a swarm of drones. We use MobileNetV2 [70] to train on this dataset.

- **Speech Recognition**. In the domain of speech recognition, we select Tensorflow Speech Command [75] 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 raw audio clips, and use VGG-13 [73] as the model.
- **Natural Language Processing**. In the domain of natural language processing, we select AG News Corpus [90] 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 [11, 31] to train on this dataset.

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

- **TicTac** [25]. TicTac is a status quo distributed training framework for training acceleration.

[1] Gaia [33] 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. Therefore, we did not include it as a baseline.

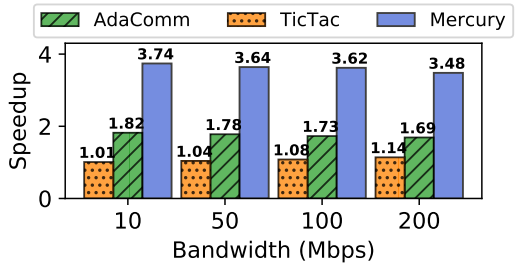
It proposes two heuristics to re-order parameter transfer nodes in a computational graph to increase the overlapping between computation and communication, outperforming TensorFlow by 1.19× shorter in total training time.

- **AdaComm** [78]. AdaComm is a status quo communicati-on-efficient distributed training framework. AdaComm reduces training time by allowing each worker to perform multiple local SGD iterations before communication and adaptively balancing between the number of local SGD iterations and communication.

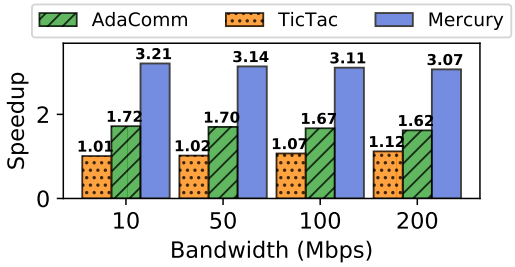
Evaluation Metrics. We use two metrics to evaluate the performance of Distream and the baselines.

- **Total Training Time**. We use total training time as our first metric. Total training time is defined as the wall-clock time from the beginning to the convergence of training process. This metric contains not only information about the number of iterations until convergence but also information about the training time per iteration.
- **Training Quality**. The second metric is training quality. We use top-1 test accuracy of the trained DL model to measure the training quality.

Training Details. During training, the batch size of each device is set to 32, and we adopt SGD optimizer and cosine-annealing learning rate decay [59].

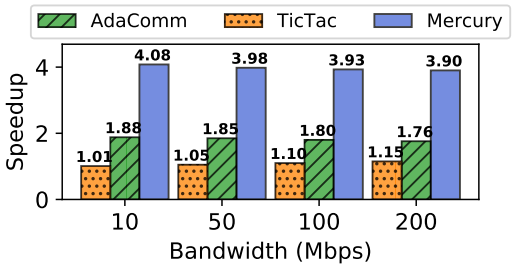


(a) CIFAR-10

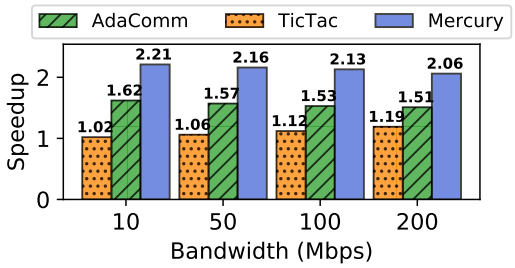


(b) CIFAR-10

Figure 2.11: Overall performance comparison on Cifar10 and Cifar100.

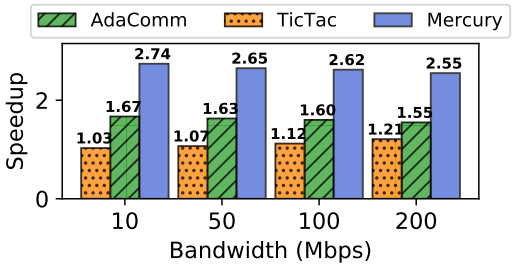


(a) SVHN

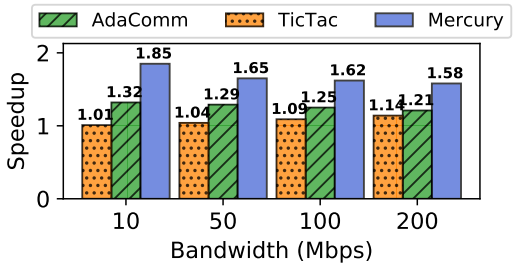


(b) AID

Figure 2.12: Overall performance comparison on SVHN and AID.



(a) Tensorflow Speech Command



(b) AGNews

Figure 2.13: Overall performance comparison on Tensorflow Speech Command and AGNews.

2.6.2 Overall Performance

We first compare the overall performance of Distream 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 2.11, Figure 2.12 and Figure 2.13 shows the results. We see that Distream significantly outperforms TicTac and AdaComm on all the six datasets across all the network bandwidths, with improvements from 1.6× to 3.9×. Specifically, Distream 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 1.01× to 1.21× speedup and AdaComm achieves 1.2× to 1.9× in

wireless settings where bandwidths are considerably constrained.

We also notice that Distream achieves higher speedups when bandwidth becomes more constrained. For example, across six datasets, Distream 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 larger and Distream can spend more time on computing latest importance rank. This result demonstrates that the more constrained the bandwidth is, the more superiority Distream has.

Why Distream Outperforms TicTac and AdaComm. To understand why Distream is able to achieve higher speedups compared to TicTac and AdaComm, we use *CIFAR-10* as an example, and depict the test accuracy of Distream (blue), TicTac (orange) and AdaComm (green) during the complete training process in Figure 2.14. We have two observations.

First, Distream outperforms TicTac by a large margin from the beginning until convergence. Specifically, Distream uses 2.7× and 2.4× fewer communication rounds on *CIFAR-10* and 2.7× on *Tensorflow Speech Command* to converge compared to TicTac. This is because under wireless network setting where bandwidth is limited, the communication time dominates training overhead overlapping computation with communication is not effective anymore (Figure 2.2). Therefore, TicTac provides little runtime reduction in each iteration. In contrast, Distream 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 Distream in early stage when the number of communications is smaller than 10×10^3 (Figure 2.14 (left)), AdaComm begins to lose effect and converges slower than Distream. 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

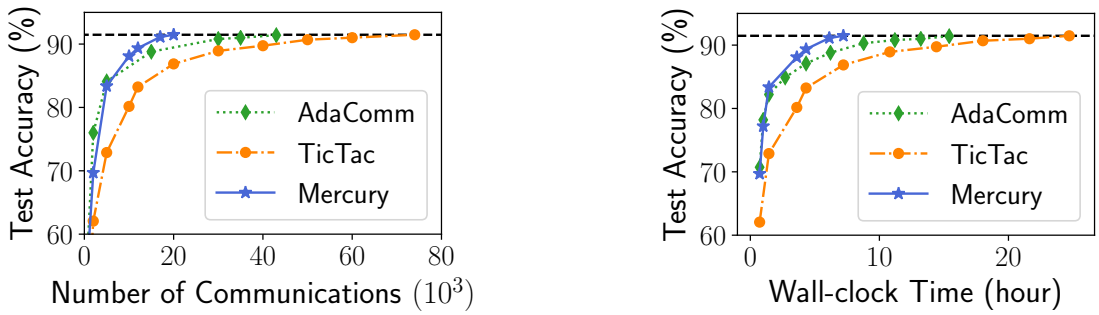


Figure 2.14: Test accuracy curves during the complete training process in terms of number of communication rounds (left) and wall-clock time (right).

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, Distream 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, Distream eventually outperforms AdaComm in terms of both number of communications and total training time.

2.6.3 Component-wise Analysis

Next, we evaluate the effectiveness of each of the three key techniques incorporated in Distream. The experimental setup is the same as in §2.6.2. We only include the results obtained in *CIFAR-10*

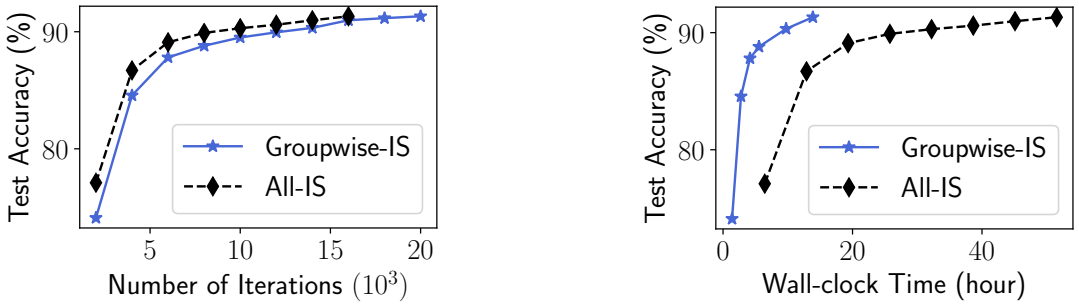


Figure 2.15: Test accuracy curves in terms of number of iterations (left) and wall-clock time (right) using group-wise (Groupwise-IS) and all-inclusive importance computation and sampling (All-IS).

here since we achieved similar results on the other five datasets. The experiment results altogether show that with the three proposed techniques, Distream 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 §2.4.1. 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 2.15.

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 2.15 (left)), group-wise is 4.2$\times$ faster than the all-inclusive when translated into total training time (Figure 2.15 (right)). 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. Impor-tance-agnostic Data Resharding. Next, we evaluate the effectiveness of the proposed importance-aware data resharding technique described in §2.4.2. 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. We also compare to one extreme case where data resharding was not performed at

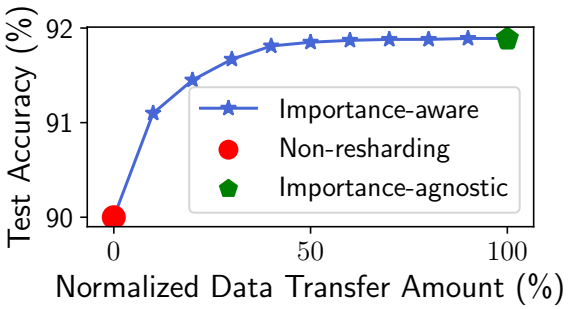


Figure 2.16: Importance-aware vs. importance-agnostic data resharding, vs. non-resharding.

all. Figure 2.16 shows the results. We have two observations.

First, similar to the findings from many other works [22, 23, 64], we observe that compared to non-resharding, data resharding is able to boost the test accuracy up by 1.84%, demonstrating the necessity of data resharding for achieving state-of-the-art test accuracy.

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.

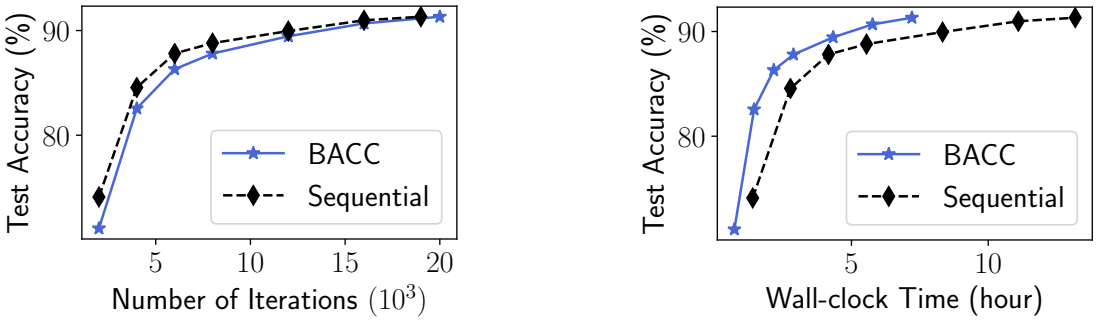


Figure 2.17: Test accuracy curves in terms of number of iterations (left) and wall-clock time (right) using BACC and sequential pipeline.

Component#3 Analysis: BACC Scheduler vs. Sequential Pipeline. To evaluate the effectiveness of the proposed BACC scheduler described in §2.4.3, we compare it against the sequential

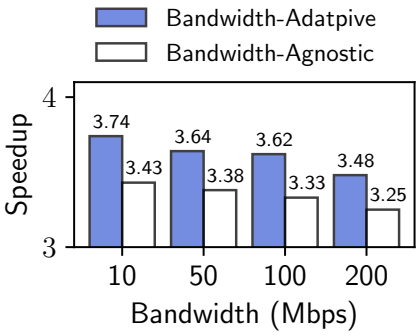


Figure 2.18: Speedup on CIFAR-10.

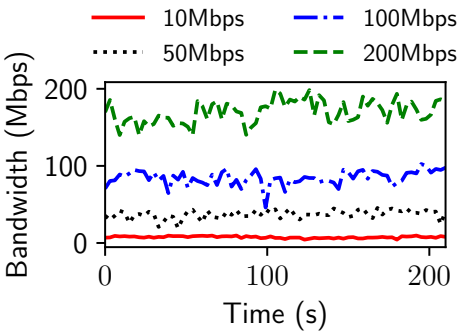


Figure 2.19: Bandwidth snapshot.

pipeline, which performs data resharding, group-wise importance computation and sampling, gradient computation and gradient aggregation sequentially. As shown in Figure 2.17, although BACC uses 5% more iterations to converge (Figure 2.17 (left)), when translated into total training time, it is 2.2× faster than the sequential pipeline (Figure 2.17 (right)).

2.6.4 Bandwidth Adaptation Performance

We take a closer look at the bandwidth adaptation performance of the BACC scheduler proposed in Distream. When deploying our testbed in the real-world settings, we observed that the bandwidth variations across different bandwidths are 10% on average (Figure 2.19 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 2.18 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 Distream in adapting to bandwidth variations.

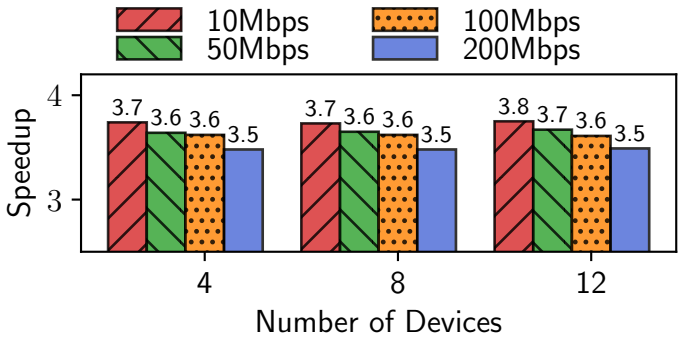


Figure 2.20: Scaling performance of Distream on 4, 8, 12 edge devices under various network bandwidths.

2.6.5 Scaling Performance

Finally, we evaluate the scalability of Distream by running it on more edge devices under different bandwidths. Again, we only show the results of CIFAR-10 here due to page limitation since we observed similar results on the other five datasets. As shown in Figure 2.20, when scaling from 4 devices to 8 and 12 devices, Distream gracefully maintains the speedup over standard distributed SGD under various network bandwidths. This result demonstrates that Distream is able to maintain its superiority over standard distributed SGD when the number of edge devices scales up.

2.7 Related Work

The design of Distream was inspired by distributed training frameworks in data center setting. The most similar works to Distream are AdaComm [78] which adaptively adjusts the number of local SGD iterations to reduce communication cost, and TicTac [25] which overlaps gradient computation with communication to reduce per-iteration overhead. Although many techniques were proposed in these works to accelerate training process, they are designed for distributed training systems in data centers. Unfortunately, there is limited performance gain when directly applying these techniques to on-device setting given the significant gap in network bandwidth

between these two settings.

The design of Distream was also inspired by existing works in federated learning [62, 47] – the other category of on-device distributed training as we discuss in introduction. However, the training efficiency optimization techniques involved in those works achieve training efficiency at the cost of compromising the accuracies of the trained models, which motivates our work for designing techniques that improve training efficiency without such compromise. In fact, our proposed group-wise importance computation and sampling technique and BACC scheduler can also be applied for enhancing the training efficiency of federated learning.

2.8 Conclusion

In this paper, we present the design, implementation, and evaluation of Distream, an importance sampling-based framework that enables efficient on-device collaborative learning without compromising the accuracies of the trained models. Distream addresses the key bottleneck of collaborative learning, and contributes novel techniques that take a different path from existing approaches. We implemented Distream and conducted a rich set of experiments with a self-developed testbed on six commonly used datasets across computer vision, speech recognition, and natural language processing. Our results show that Distream consistently outperforms the status quo. Therefore, we believe Distream represents a significant contribution to making collaborative learning systems practically useful in real-world deployments.

Chapter 3

Collaborative Learning for Efficient Federated Learning

3.1 Introduction

Recent years have witnessed the booming of edge devices. It is expected that by 2030, the number of edge devices will increase from 8.74 billion in 2020 to 25.44 billion [12]. These edge devices, such as smartphones, drones, wearables, are revolutionizing the way we live, work, and interact with the world. Equipped with a variety of sensors, these devices are collecting a massive volume of data from people as well as surrounding environments every day. These data are of great value to technology companies for improving their products and providing better services and customer experience. In order to gain real value out of these data, the status quo approach is to upload these decentralized data onto central servers in the cloud where powerful machine learning models such as deep neural networks are trained. However, these user data often contain private or privacy-sensitive information such as faces, license plates, house numbers, etc. As a result, these data are protected from leakage by law, and uploading the data to central cloud servers for further processing is not a feasible solution.

With the emergence of AI chips, the majority of the edge devices are now able to not only perform on-device model inference, but also train a deep learning model using the data collected

by themselves. There is a significant interest in leveraging the on-device computational capabilities to collaboratively train a globally shared model from the decentralized data while keeping the data locally on each device [72]. Many people have shifted from centralized server to edge setting for machine learning model development and training.

Driven by this trend, Google proposed federated learning [47] in 2016. Federated learning is an emerging machine learning paradigm that has recently attracted considerable attention due to its wide range of applications in mobile scenarios [63, 47, 74]. Different from the standard machine learning approach which requires all the training data to be centralized in a server or in a datacenter, it enables geographically distributed edge devices such as mobile phones to collaboratively learn a shared model without exposing the training data on each edge device. As such, federated learning enables distributing the knowledge across smartphones without exposing users' private data.

Federated learning uses distributed stochastic gradient descent (SGD) and requires a central to coordinate the iterative training process. Before each round begins, the server selects a subset of clients and distributes the latest global model to all the participating devices. Each device computes the gradient or update of the model parameters using its local data. The server aggregates the gradients or models update from all devices, averages them, and updates the global model. The central server sends the updated model to the clients to perform local training of next round. In such manner, each device benefits from obtaining a better model than the one trained only on the locally stored private data.

Although FL addresses the privacy issue, it still has many problems and suffers from significant performance degradation due to three problems. The first is low training efficiency. In federated learning, the training data is massively distributed across a large number of clients. Existing approaches adopt random sampling when it selects clients to participate in training before each round begins. In each round, the clients adopt random sampling strategy to sample mini-batches when

performing local training. Consequently, the training efficiency in federated learning is low and it requires more steps to converge. The second problem is limited network bandwidth. In data centers, communication between the server and working nodes is conducted via Gbps Ethernet or InfiniBand network with even higher bandwidth [81]. In contrast, communication in federated learning relies on wireless networks such as 4G and Wi-Fi. Both uplink and downlink bandwidths of those wireless networks are at Mbps scale, which is much lower than the Gbps scale in the data center setting. The limited bandwidth in federated learning illustrates the necessity of reducing the communication cost to accelerate the training process. The last problem is federated learning is low accuracy due to the adoption of small capacity model. Although existing edge devices such as smartphones are able to train neural network models, they have much less compute resources compared to centralized cloud servers. Moreover, these devices often need to concurrently run multiple models that perform different tasks in real time. It thus requires the collaboratively trained model in federated learning to be computationally efficient. Therefore, federated learning usually sacrifices accuracy and uses a small model in order to reduce the computational cost.

In this work, we propose FedAce, an efficient federated learning framework that addresses the aforementioned problems. It is featured by two key components. The first component is federated importance sampling, which not only performs important data selection within each client in local model training, but also selects important clients to participate in each round of training. By focusing on important clients and data, FedAce is able to increase training efficiency and reduce training time. The second component is federated model compression, which iteratively compresses the globally shared model throughout the whole training process. The global model is compressed using a global mask aggregated by locally computed masks, which represents the opinions on the locations of important parameters from distributed clients. After training completes, only important parameters are kept, and the final global model is computationally efficient.

As the exchanged information between clients and the central server are in the form of model, we can significantly reduce the communication cost by transferring a compressed model. As a result, the training time is significantly reduced.

We implement FedAce using FedML [26], a research library and benchmark for federated learning. We examine the performance of FedAce on four federated learning datasets across computer vision, speech recognition, and natural language processing tasks. Our experiment results show that FedAce is able to increase training efficiency and achieve significant training speedup compared to existing FL frameworks. Meanwhile, FedAce achieves significant model compression and communication reduction ratios with no or little accuracy loss.

3.2 Related Work

Federated Learning. As an emerging field, federated learning has recently attracted a lot of attention. [63] developed a federated learning approach based on iterative model averaging to tackle the statistical challenge, especially for the mobile setting. [74] proposed a distributed learning framework based on multi-task learning and demonstrated that their approach is able to address the statistical challenge and is robust to stragglers. [47] developed structured and sketched update techniques to reduce uplink communication cost in federated learning.

Distributed SGD Training. Our work is related to distributed training acceleration in federated learning. Most existing works are focused on on reducing communication time to cut the wall-clock time. This is achieved by quantizing gradients using smaller number of bits [81, 10, 55] or selecting important gradients to transfer via sparsification [79, 56, 33]. Another research field aims at increasing mini-batch size for less communication. In [37], the authors proposed to use larger mini-batch size to reduce communication iterations. Although this approach effectively reduces the

wall-clock time, it comes with the cost of lowering the accuracy of the trained model. Lastly, deep learning models in general can be represented as directed acyclic graphs (DAGs). The structures of DAGs provide an opportunity to overlap computation operations and communication operations in a pipelined fashion to mask out the communication cost. This optimization technique can be seen as a mechanism to reduce the total training time by identifying the optimal gradient transfer order [25, 39, 17, 87] or using staled model weights [54].

Model Compression. Our work is also related to model compression. Deep learning models have been increasingly computationally expensive. Model compression aims to reduce the computational intensity of deep neural networks by pruning out redundant model parameters. It has received a lot of attentions in recent years due to the imperative demand on running deep learning models on resource-constrained edge devices [24]. The most popular technique for model compression is parameters pruning, where the least important parameters are removed to reduce the model size and inference cost. Depending on the structure of pruned parameters, there are two types of pruning technique. The first is unstructured pruning, which focuses on pruning model parameters [24]. Although unstructured pruning parameters achieves higher model compression rate and effective at reducing model size, it does not necessarily reduce computational costs when it is applied to CNNs. The other is structured pruning [50], which identifies and prunes unimportant filters in CNNs. Structured pruning achieves lower compression rate, it can effectively reduce runtime cost.

Our approach also differs significantly from prior work on federated learning, where they focus on either tackling statistical and straggler challenges [63, 74] or reducing uplink communication cost alone [47]. Our approach is also different from prior work on model compression [24, 66] and communication reduction for distributed learning [10, 81, 56], where they treat them as two *separate* tasks. In other words, the novelty of our proposed approach lies at integrating model

compression and communication reduction into a single optimization framework based on training data distributed at different devices.

3.3 Background and Motivation

In this section, we first introduce the background knowledge of federated learning. Then we compare the difference between federated learning and standard distributed learning. Finally we discuss the drawbacks of existing federated learning approaches and provide improving opportunities for federated learning, which is the motivation of FedAce.

3.3.1 Architecture of Federated Learning

Federated learning is new training paradigm for training machine learning models using geographically distributed clients under the orchestration of a central server. The training process is conducted in the form of iterative rounds. In each round, the central server selects a group of clients randomly from all the available clients to participate in the training. The central server will first send to latest global model to selected clients in the current round. After receiving the latest model from the central server, each selected client computes the model update or gradient using its local data, which will be sent to and averaged in the central server. The central server will aggregate the model updates or gradients from clients and update the global model:

$$W^{t+1} = \sum_{k=1}^{K} \frac{n_k}{n} W_k^{t+1} \tag{3.1}$$

where W^{t+1} is the updated global model for round $t+1$. K is the number of selected clients that participate in the training and w_k^{t+1} is the updated model of client k. n_k is the number of data in

client k and we have $\sum_{k=1}^{K} n_k = n$. The overview of training process is shown in Figure 3.1.

3.3.2 Federated Learning Characteristics

Although both federated learning and distributed aim at collaboratively training a global model, the setting where federated learning is focused on is fundamentally different from distributed training. We present the key differences between federated learning and distributed training as follows.

Data Heterogeneity. The first difference is that there is data heterogeneity in federated learning. Specifically, the local data across clients are non-i.i.d distributed. This includes not only the variation in the number of local data, but also the feature skewness and label skewness [43]. In contrast, the data is independent and identically distributed across workers and the quantity of local data is the same. Data heterogeneity imposes an significant challenge in federated learning and it is regarded as the major cause of the model performance loss.

Client Heterogeneity. In federated learning, the clients from millions of users across world could be selected to participate in training. Thus, the hardware and compute capacities of participating clients could be different. Moreover, as the central server only selects a subset of clients to participate in joining in each round, the client heterogeneity is also changing in each round. In distributed training, each work has the same compute capability.

Network Heterogeneity. In federated learning, each client is unreliably connected to the central server and the bandwidth is small. As the clients are distributed across continents or the world, the network bandwidths between clients and the central server are heterogeneous. In distributed training, as the clients are located in local region, the bandwidths among clients are the same and each client is reliably connected to the central server.

These differences make existing optimization techniques for distributed SGD training fall short, and highlight the necessity to develop a new training framework to optimize the system perfor-

mance for federated learning. Table 3.1 summarizes the key differences of distributed training and federated learning.

	Federated Learning	**Distributed Training**
Data distribution	non-i.i.d	i.i.d
Number of local data	small, typically < 500	large, typically > 1000
Data Accessibility	invisible to other clients	visible to other clients
Number of clients	typically >=500	typically <=50
Participating clients	<10%	100%
Client location	wide area	local region
Client connectivity	unreliably connected	always connected
Network bandwidth	low	high

Table 3.1: Comparison between federated learning and distributed training

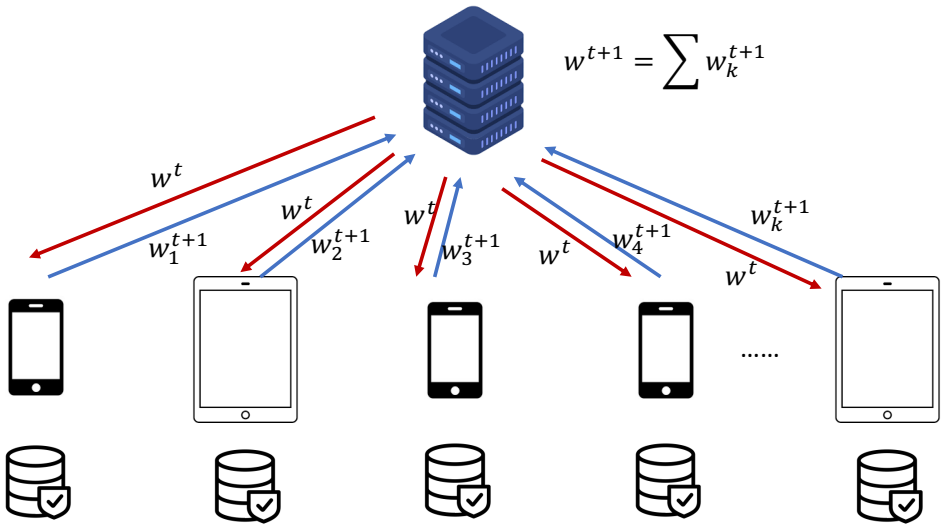


Figure 3.1: Federated learning overview

3.3.3 Scope and Improving Opportunity

Federated learning is challenging research area and many problems remain to be solved, such as data non-iidness, slow convergence and adversarial attacks. In this work, FedAce only focuses

on addressing three problems in federated learning. Next, we identify and discuss impossible improvement for each problem.

Low Training Efficiency. Current practices in federated learning adopt random sampling in both data selection and client selection. While this simple solution facilitates the implementation and deployment of federated learning applications, the training efficiency is low. This is because the local data at each client contributes differently to training. As there is data heterogeneity and the local data cannot be shared among clients, each client also contributes differently. Consequently, using random sampling strategy in data and client selection wastes much training effort on unimportant samples and clients. This motivates us to adopt *non-random sampling strategy* when we select clients for training and construct mini-batch for local train in order to increase the training efficiency.

Large Communication Cost. As discussed before, the clients are distributed across the world, and their bandwidths are limited and heterogeneous. Hence, the communication overhead in each round is large, making federated learning prohibitively impractical to train a global model within a short period time. To mitigate this problem, we should not only increase the training efficiency for reduction of training steps, but also reduce the overhead communication cost. Therefore, the second improving opportunity lies at reducing the communication cost to further reduce the total training time.

Low Model Accuracy. In federated learning, the participating clients are usually resource-constrained edge devices (e.g., smartphones), where the network resources and compute resources are limited. Training a large model not only results in slow convergence due to large per-round communication cost, but also a final model inefficient for inference. Given such resource constraints, the common practice of existing work is to train a small neural network for efficient inference after the training. However, neural networks that achieve state-of-the-art accuracy in most learning tasks (especially

difficult ones) are models with large capacities and deep architectures. Current practice of federated learning could not leverage the superior accuracy brought by large neural networks. As a sanity check, we train a large model and a small model (similar to the large model but with 10% weights) under the same federated setting. As shown in Figure 3.2, there is an accuracy gap between the two models. Therefore, it is not trivial to enhance the model performance while keeping the trained model compact and efficient.

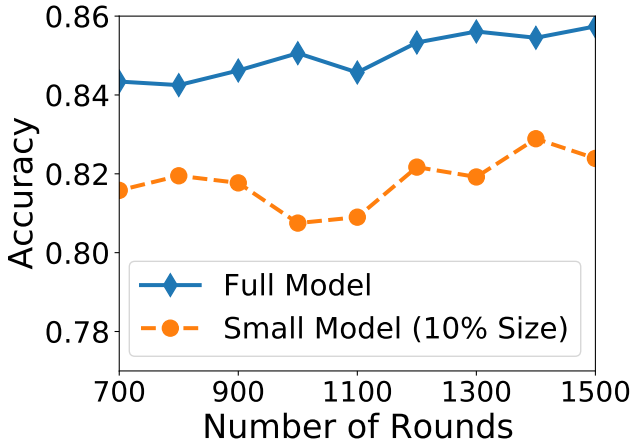


Figure 3.2: Training process federated learning

3.4 FedAce Design

In this section, we present the detailed design of FedAce. First we describe the system architecture of FedAce. Then we introduce two algorithmic components, which are the backbones of FedAce.

3.4.1 Overall Architecture

FedAce is a distributed framework that spans clients and the central server, as illustrated in Figure 3.3. At client side, the important data sampler selects important data to construct mini-batches to perform local training. After local training completes and before the client sends the locally

updated model to the central server, the local compressor computes a local mask that indicates important parameters to be preserved and unimportant parameters to be pruned. At central server side, the important client sampler selects important from available clients to participate in training for the next round. The important client sampler is also responsible for maintaining the importance estimate of available clients. The global compressor aggregates the local masks sent from clients into a global mask, and prune the global model using the global mask.

Next we will describe the detailed approaches behind these components.

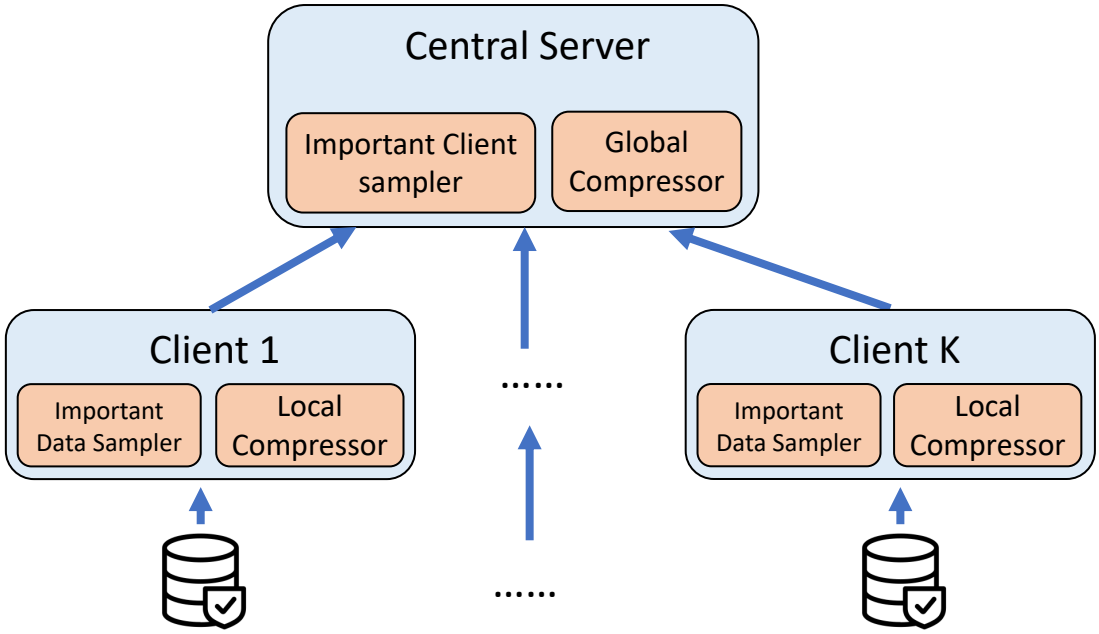


Figure 3.3: Overview of FedAce

3.4.2 Federated Importance Sampling

Federated learning employs a hierarchical selection strategy to train a global model: the central server first selects a subset of clients to participate in training; then each participating client selects important data to perform local training. Accordingly, FedAce adopts federated importance sam-

pling, which consists of two importance sampler to increase training efficiency in a hierarchical way: in the high level at central server side, it uses important client sampler to select important clients to participate in training; in the low level at client side, each client uses data importance sampling to sample important data, as illustrate in Figure 3.4.

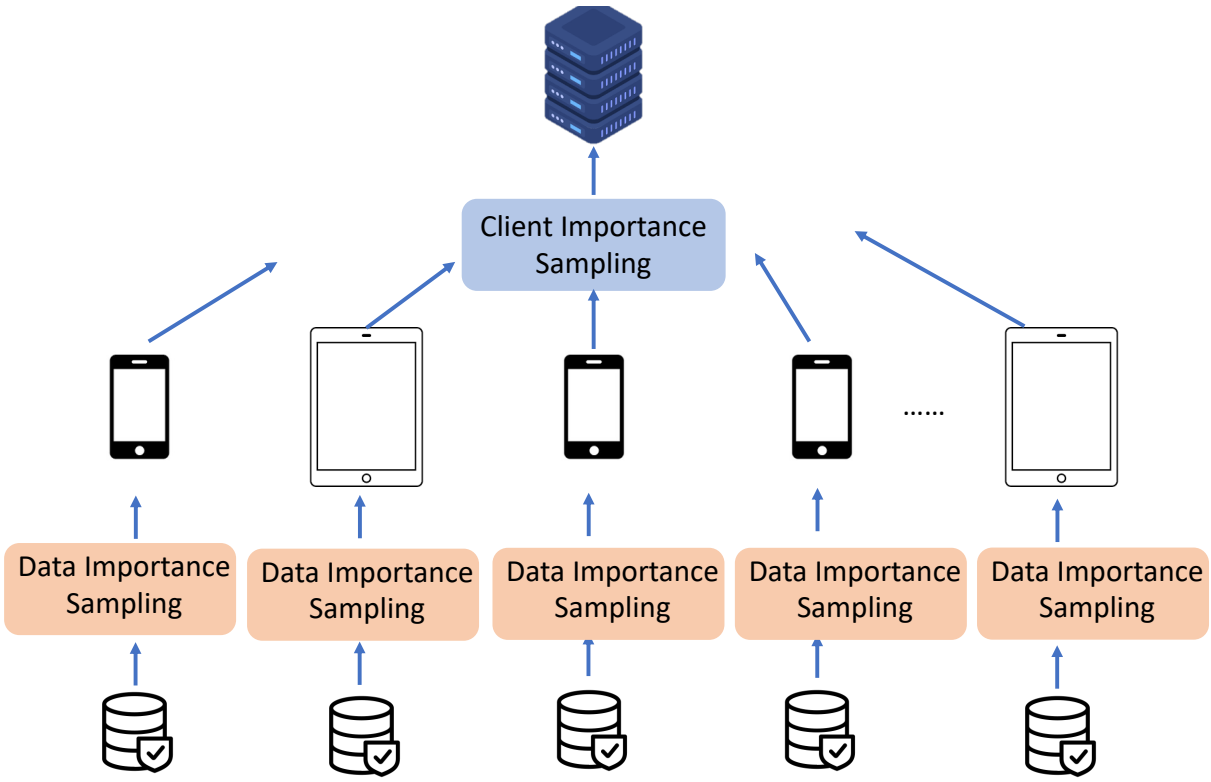


Figure 3.4: Overview of federated importance sampling.

3.4.2.1 Data Importance Sampling

The data sampler at each client maintains an estimate of importance distribution P of its local data D. Then when constructing the mini-batches, the data is sampled from the distribution P without replacement. We employ data loss as the importance indicator when computing importance distribution P as data loss is an good approximation of the data importance and computing loss is

cheap, as presented in Mercury. Therefore, the importance distribution P can be computed as

$$p_i = \frac{l_i}{\sum_{n=1}^{|D|} l_n} \tag{3.2}$$

where $|D|$ is the total number of data at local client.

Different from distributed training, the number of data in each client is small and communication process takes longer time as the clients are distributed around the world, the cost of computing the losses for all data can be entirely hidden when it is paralleled with communication process. More importantly, as federated learning adopts local SGD where the local model is updated by performing multiple steps and the communication cost is large, the cost of loss computation is comparably small even if it is carried out without parallelization.

3.4.2.2 Client Importance Sampling

Instead of selecting client randomly, the central server in FedAce prioritizes the important clients to participate in training. Before each round begins, the central server collects client importance information from connected clients and selects the important clients based on the collected importance information. To collect client importance, FedAce sends the latest model to clients. Once the clients received the model sent by the central server for importance collection, it will perform inference for all of its local data. The importance of client K is computed as:

$$I_k = \bar{L}_k \frac{n_k}{n} \tag{3.3}$$

where $\bar{L}_k$ is the average sample loss at client k, n_k is the number of local data at client k, and n is the total number of data. In other words, a client needs to have a large number of important data to

be viewed as an important client. Then the probability of selecting a client k is computed as:

$$p_k = \frac{I_k}{\sum_{k=1}^{K} I_k} \tag{3.4}$$

One limitation of this strategy is that the central server has to send the latest model to all clients so that all clients can estimate their importance and send the importance back to central server to identify important clients. This may cause large overhead because the server has to send the model to clients even though only some of clients will be selected to participate in the training. FedAce adopts two techniques to achieve a good balance between the selection of important client and the incurred cost. In the first technique, the central server does not update the importance of all clients in every round and reuses the cached client importance. To calibrate the staleness in the cached importance information, the client importance is re-weighted using a staleness term for balancing exploration and exploitation:

$$I_k = \bar{L'}_k \frac{n_k}{n} * \exp(\beta \frac{||W_k - W||}{||W||}) \tag{3.5}$$

$\bar{L'}_k$ represents the last updated loss of client k. W_k is the model weight used to compute $\bar{L'}_k$, and W is latest model weight at central server. $\exp(\beta \frac{||W_k - W||}{||W||})$ is a staleness term to calibrate the importance of a client. If this term is large, it means the difference between W_k and W is large and the model W_k used to compute $\bar{L'}_k$ is staled. In such case, The central server should prioritize this client to participate in training, as this client has a higher chance of becoming more important since last update. $\beta > 0$ is a scaling factor determined empirically. A large β emphasizes the freshness of importance estimate and encourages FedAce to focus more on exploration.

The second technique to reduce overhead is to allow a subset of clients to perform local training

while allowing another non-overlapping subset of clients to perform importance update simultaneously. Specifically, the least recently updated clients in the non-overlapping subset of clients will be selected to update their client importance. Since the importance collection process is not dependent on the local training process, it can be executed in parallel with the local training process. Because the importance collection process only requires feed-forward inference to obtain the sample losses, and only requires downloading global model without the need of uploading the locally updated model, it takes shorter time than local training process. Therefore, we can ensure that before each round begins, the central server is able to have the client importance information updated for client sampling. This selection strategy is able to select important clients while ensuring on-time client importance update with minimum overhead.

3.4.3 Federated Model Compression

Federated model compression aims to compress the model in each round in order to save per-round cost, and obtain an efficient and compact model after training completes. To compress the model, we need to identify and prune unimportant parameters. To achieve that, we compute the binary mask to indicate the unimportant parameters in the model (0 for unimportant parameters and 1 for important parameters). In each round, FedAce computes such a binary mask and apply the computed mask to the global model to obtain a pruned model. The pruned model will be used to proceed the training of next round. Next, we will describe the generation of the mask and two techniques for maintaining accuracy after compression.

Mask Generation. At a high level, the generation of mask consists of three steps, as shown in Figure 3.5. First, after receiving the latest model, each client not only trains the local model using its own data, but also computes the local mask. The local mask will be sent the central server together with the updated model when local training ends. Then, the central server aggregates the

local mask to generate the global mask, which will be applied to the aggregated global model.

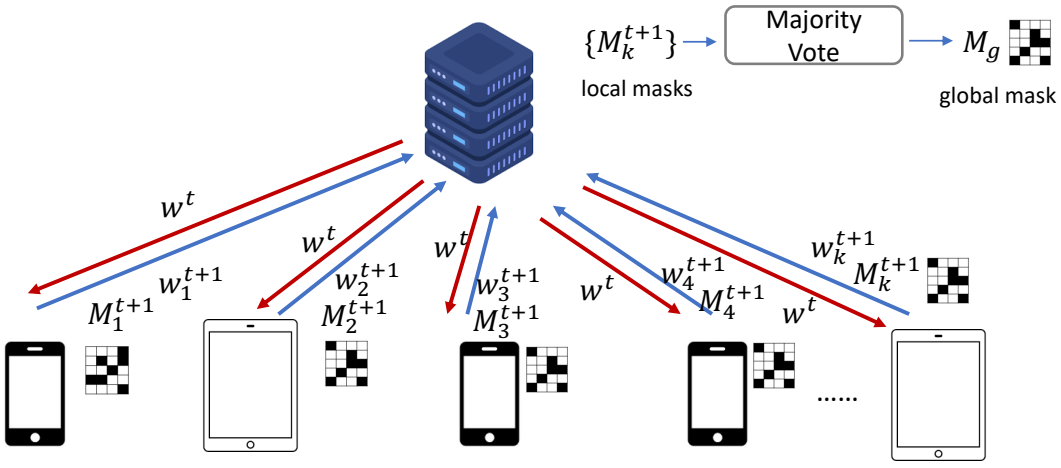


Figure 3.5: Overview of federated model compression.

FedAce adopts unstructured pruning and uses magnitude-based approach to determine the important parameters. Specifically, the local mask at client k can be computed as:

$$M_k = \sigma(W_k^{abs}, s) \tag{3.6}$$

where W_k^{abs} is obtained by taking the absolute value of the elements of the weights W_k at client k. $\sigma(W, s)$ is an activation function where the elements in W that are greater than s will be set to one and to zero if otherwise.

After the local masks are generated, they will be sent to the central server together with the model weights. Then, the central server not only aggregates the model weights as it normally does in federated learning, but also aggregates the local masks to create a global mask. We use majority vote as the aggregation method for local masks. Specifically, FedAce first sums the collected local masks together. Those summed elements smaller than a threshold s in the summed mask indicate the corresponding weights of the global model will be pruned based on the agreement from the

majority of the clients:

$$M_g = \sigma(\sum_{k=1}^{K} M_k, s), \tag{3.7}$$

In other words, those parameters that most local masks agree to preserve will be kept and others will be pruned. After the global mask M_g is computed, the pruned global model is obtained by

$$W_g = W \odot M_g \tag{3.8}$$

where $\odot$ is the element-wise multiplication. Using masks brings extra communication cost to the training, as the clients need to upload the local masks to the central server. To reduce cost, we use bit operation to encode the binary mask into last bit of the weight. Changing the last bit does not affect the model accuracy, which has been verified in many prior works on model quantization [92].

One problem of federated model compression is that the model accuracy after compression may drop significantly, especially with the existence of data non-i.i.dness, which causes the clients to disagree with each other on pattern of optimal mask. To address this issue, we proposed two techniques to maintain the accuracy. The first is compression rate schedule, the second is mask caching.

Compression Rate Schedule. One key prerequisite for accurately identifying the unimportant weights to prune is to train the model to nearly convergence when the weights become stable. However, this prerequisite does not hold in the case of federated model compression because pruning is performed throughout the training process where the model weights may not be stabilized. As a result, generating local masks from local models that are not converged may falsely remove important weights and preserve redundant weights.

To achieve the prerequisite, FedAce incorporates a warm-up stage and an adaptive model sparsity policy during the federated learning process. Specifically, FedAce does not perform model

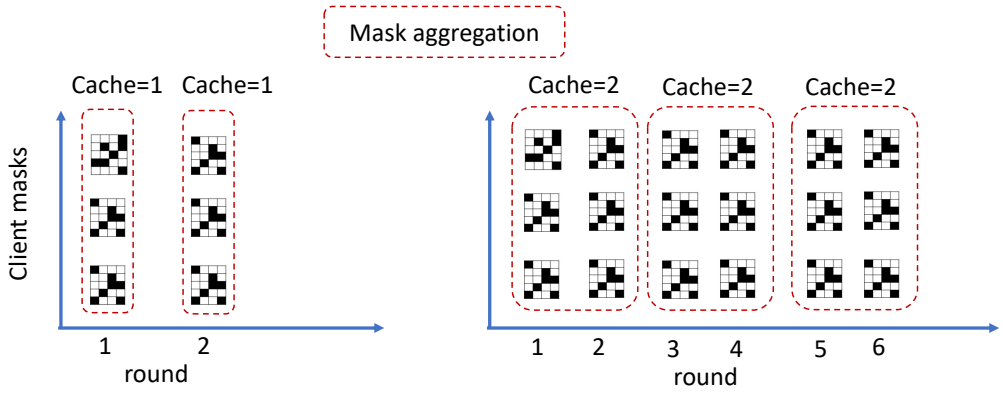


Figure 3.6: Examples of mask caching.

compression in the warm-up stage. The goal is to allow the model quickly identify important and unimportant parameters. After the warm-up stage, the federated model compression begins and the compression rate begins to increase exponentially. In doing so, we allow the compressed model to regain accuracy slowly, especially in the late stage where mistakenly pruning an important parameter leads to a significant accuracy drop.

Mask Caching. Given the non-iid characteristic of the federated paradigm, the local masks of the clients may have different views on the importance of the model weights. This issue is exacerbated by the practice that at each round of the FL, only a subset of the clients is selected to participate in the training process. If the global mask is generated at the end of every round, the global mask could be easily biased by that specific subset of the clients, and thus deviated from the optimal global mask. As a consequence, the global model pruned using the biased global mask could lead to sub-optimal performance.

To address this challenge, FedAce caches the received local masks in the central server over a number of rounds from more clients before generating the global mask. As such, the local masks collected from a larger pool of clients are able to capture a better global view of the importance of the model weights. Figure 3.6 illustrates the idea of mask caching. Specifically, there is no caching when cache equals 1 (left chart in Figure 3.6). When cache equals 2, the masks will be cached and

aggregated every two rounds (right chart in Figure 3.6).

3.5 Experiments

In this section, we evaluate the effectiveness and performance of FedAce under federated learning on four federated datasets across compute vision, speech recognition and natural language modeling tasks.

3.5.1 Setup

Implementation and hardware environment. We have implemented FedAce using FedML [26]. FedML is a research library and benchmark for federated machine learning. Users can develop variants of federated learning algorithms using this framework. We implemented FedAce using 900 lines of Python code. All the experiments are run on a desktop server with 4 RTX-8000 GPUs.

Datasets and deep learning models.

1) The first dataset is *Fed-CIFAR10*, which is constructed by partitioning *CIFAR-10* into 1000 shards to simulate 1000 clients. In order to simulate the data heterogeneity across devices, we partition the data shards using latent Dirichlet allocation (LDA) [34] and each shard has an imbalanced label distribution, also known as label skewness [43]. We use ResNet18 [28] in this dataset. Because the statistics parameters of batch normalization relied on local data, we follow [67] to replace the batch normalization layer with group normalization layer.

2) The third dataset is *FEMNIST* [67]. It has 749068 images of digits and English characters, with a total of 62 classes. The dataset is partitioned into 3400 shards with different shard sizes. For this dataset, we use a CNN model with two convolutional layers and two liner layers.

3) The second dataset is Fed-TSC, which is constructed by partitioning Tensorflow Speech Com-

mand dataset into 1000 shards. This dataset consists of 105,829 audio utterances of 35 short words, recorded by a variety of different people [75]. Similarly, we partition the data shards using LDA distribution simulate label skewness. We convert the audios into spectrum and we use VGG-13 for classification.

4) The final dataset is *StackOverFlow* [67]. StackOverflow is a language modeling dataset. It consists of questions and answers from a website called stackoverflow. The dataset contains 342,477 unique users which we use as clients. We perform next-word prediction in this task. We use a one layer LSTM with 670 hidden dimension to predict the next word in the sentence. Each word is embedded in a 96-dimensional space with a RNN. The details of the datasets and used model can be found in [67].

Baselines. While there are many works aiming at improving federated learning, most of these works are focused on addressing the data non-iidness [45, 53] or preserving the user privacy. The approaches proposed in FedAce are complementary to these works. Therefore, we adopt FedAvg as the baseline in this experiment, which is a state-of-the-art federated learning training method for federated learning where each client trains for one or more epochs and the local model updates are sent to and averaged in the remote central server. For both FedAvg and FedAce, we use the same training hyperparameters. Specifically, at client side , we adopt SGD optimizer and the learning rate is 0.01 and we use cosine-annealing learning rate decay [59]. The batch size is 20. At server side, we use adam optimzer and the learning rate is 0.001. The number of local epochs are set to 1.

3.5.2 Results

We show that FedAce achieves superior performance compared to FedAvg by highlighting the key results:

- FedAce achieves 3.24× to 6.54× speedup over FedAvg without loss of accuracy.
- FedAce reduces network traffic by 2.4× to 3.44× in each round.
- FedAce obtains a compact and efficient sparse model after training with 17.2% to 41.2% of original model size.

3.5.2.1 End-to-End Performance Comparison

We first compare the end-to-end performance of FedAce and baseline. Table 3.2 shows the speedups obtained by FedAce over FedAvg. The overall speedups are 5.28×, 4.7×, 5.28×, and 5.28× on Fed-Cifar10, Femnist,Fed-TSC and StackOverflow respectively. Specifically, the speedups are contributed by two sources. The first is the reduction in the number of rounds contributed by adopting importance-aware data selection and client selection strategy. The second source is the reduction in the communication cost contributed by compressing the model. All of the speedups obtained by FedAce without hurting accuracy. It indicates that FedAce can increase the training efficiency and reduce the training overhead.

Dataset	**# Rounds Reduction**	**Avg Comm. Cost Reduction**	**Speedup**	**Final Model Sparisty**
Fed-Cifar10	2.2×	2.4×	5.28×	69.3%
Femnist	1.7×	2.76×	4.7×	75.3%
Fed-TSC	1.9×	3.44×	6.54×	82.8%
StackOverflow	1.6×	2.03×	3.24×	58.8%

Table 3.2: End-to-end performance.The numbers indicate the speedups of FedAce over FedAvg.

3.5.2.2 Sensitivity Analysis

In this section, we perform sensitivity analysis to show that FedAce is robust to various hyperparameters.

Impact of Number of Local Epochs. We first analyze the impact of the number of local epochs. We run FedAce with different number of local epochs on Femnist in this experiment. The result is shown in Figure 3.7. We can see that when the number of local epochs increases, the speedups obtained by FedAce becomes smaller. This is because when the number of local epochs increases, the local model drifts more from the global optimal model. As a result, the estimated importance distributions of local data and clients are less accurate, limiting the improvement brought by importance sampling. Furthermore, as the number of local epochs increase, the mask of the locally trained model is encoded with local information, making it diverge from the masks generated by other clients. FedAce needs more rounds to achieve the same accuracy. Even though the speedups becomes smaller, FedAce is still able to achieve significant speedup over baseline.

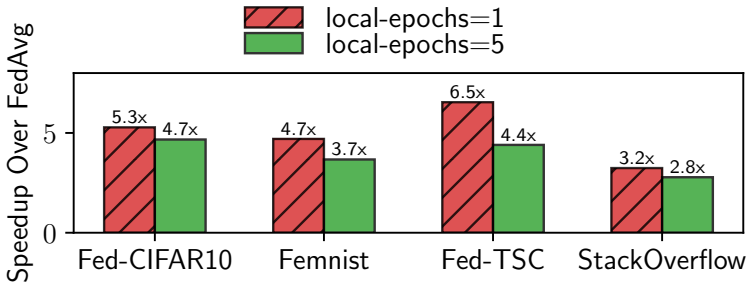


Figure 3.7: Performance under different number of epochs

Impact of Number of Clients. We then study the impact of number clients. We run FedAce with different number of clients on Femnist in this experiment. The number of clients affects the number local data in the device, thus the non-iidness of local data is also affected. We do not enable model compression in this experiment to eliminate its influence. Figure 3.8 shows the speedups brought by data sampling and client sampling under various number of clients. We can see that when the number of client increases, the speedup achieved by data importance sampling becomes smaller, while the speedup achieved by client importance sampling increases. This is because when the number of clients increases, the importance diversity in local data is small and the data sampling

does not provide much performance gain even if important data is selected. However, in this case, the speedup achieved by client becomes larger, since the importance diversity of clients becomes dominant. This demonstrates the needs to combine two sampling strategies to achieve importance sampling in federated learning.

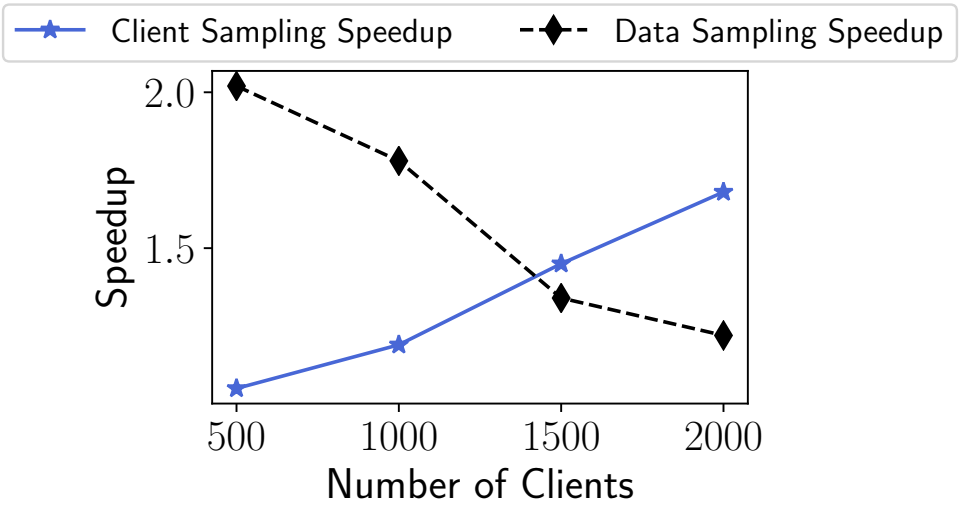


Figure 3.8: Performance gain of FedAce in three federated learning datasets

Compression Rate Schedule. In this experiment, we analyze the effect of compression rate schedule. We compare the proposed compress rate schedule and constant compress rate (set as final sparsity from the beginning till the end).

Table 3.3 and Table 3.4 shows the accuracy of different compression rate schedule strategies on Femnist and StackOverflow. We can see that with the same final sparsity, the constant schedule suffers from significant accuracy drop. FedAce with warm-up strategy only is able to mitigate the accuracy loss. FedAce with FedAce with warm-up and exponential increase schedule suffers from the least accuracy drop. This is because by gradually increasing the compression rate, FedAce can have more time to regain accuracy after pruning. This demonstrates that the compression rate schedule is effective in maintaining accuracy.

Number of Cached Masks. Finally, we study the impact of the number of cached mask in cen-

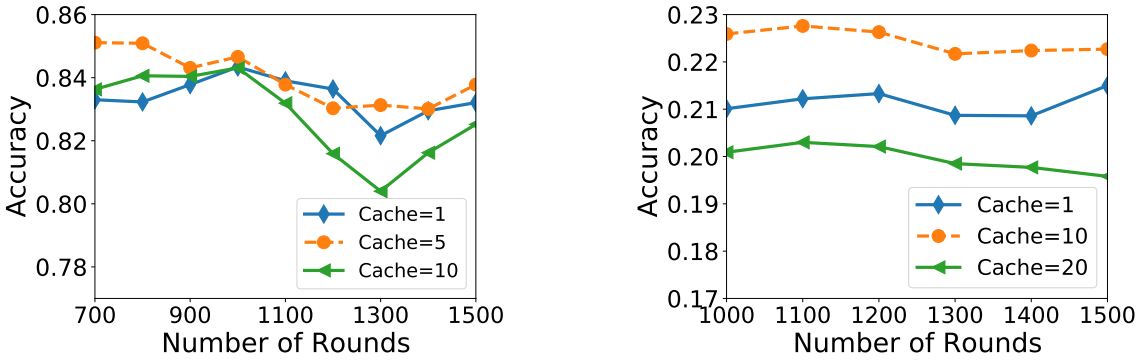
Dataset	Compression Rate Schedule	Model Sparsity	Accuracy Change(%)
Femnist	Constant	75.3%	-1.93
	warmup only		-0.64
	warm-up + exponential increase		**-0.05**
StackOverflow	Constant	58.8%	-0.92
	warmup only		-0.32
	warm-up + exponential increase		**-0.02**

Table 3.3: Performance of compression rate schedule.

Dataset	Compression Rate Schedule	Model Sparsity	Accuracy Drop
Femnist	Constant	90%	-3.32
	Schedule (warm-up only)		-2.92
	Schedule (warm-up + exponential increase)		**-2.53**
StackOverflow	Constant	90%	-2.03
	Schedule (warmup only)		-1.23
	Schedule (warm-up + exponential increase)		**-0.49**

Table 3.4: Performance of compression rate schedule under high compression rate.

tral server. Figure 3.9 plots the accuracy of FedAce using different number of cached masks on two datasets. We have two observations. On one hand, using a larger number of cached mask can achieve better accuracy than using no cached mask (cache=1). For example, using cache=5 in Feminst dataset achieves higher accuracy and using cache=10 achieves higher accuracy in StackOverflow. The reason is, by increasing the number of cached masks, FedAce is able to aggregate more local information in order to determine the global optimal mask. On the other hand, keeping increasing the number of cached masks may hurt accuracy. For example, using cache=10 in Feminst and cache=20 in StackOverflow is worse than using cache=1. This is because the mask are computed by local models, which are updated in each round. Aggregating too many local masks computed in different rounds introduces staleness in local masks aggregation.



(a) Femnist

(b) StackOverflow

Figure 3.9: Test accuracy comparison under different mask cache settings.

3.6 Conclusion

In this work, we propose FedAce, an efficient training framework for federated learning. FedAce addresses two key problems in federated learning. The first problem is federated learning requires a large number of communication rounds due low sampling efficiency in both data selection and client selection. The second problem is the model trained in federated learning has to be a small model due to inference speed requirement and constrained bandwidth, resulting in poor performance. To address the above challenges, FedAce has two components: a importance sampler that selects important data for local training and important clients in each round to increase training efficiency, and a model compressor that progressively compresses the large model. Experiment results on four federated learning datasets show that the FedAce can significantly reduce the training overhead by 3.2$\times$ to 6.5$\times$ without accuracy drop.

Chapter 4

Conclusion

In this dissertation, we propose three collaborative distributed deep learning systems for the edge: Distream, Mercury, and FedAce, to enable deep learning-based serving and training for distributed edge systems respectively. Distream addresses the key challenges brought by workload dynamics in real-world deployments, and contributes novel techniques that complement existing live video analytics systems. Mercury represents a distributed training framework for efficient distributed on-device deep learning under distributed edge setting. Mercury improves training quality and reduce convergence wall-clock time without additional overhead by adapting to the bandwidth changes using three novel techniques: group-wise importance sampling, importance-aware resharding and overlapping. FedAce is an efficient federated learning system that improves training efficiency by combining data and client importance sampling and adaptive model compressing.

Through experiments, we show that these three systems outperform state-of-the-art approaches and existing solutions can benefit from the proposed techniques for additional performance gain. Therefore, we believe these three works represent a significant contribution to enabling large-scale deep learning serving and training system on distributed edge architecture.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Nvidia titan x, 2016.

[2] 24-port gigabit stackable smart managed switch with 4 10gbe sfp+ ports, 2017.

[3] Nvidia jetson tx1, 2017.

[4] Opencv background subtraction, 2017.

[5] Networking solutions for ip surveillance, 2018.

[6] Nvidia jetson tx2, 2018.

[7] Jacksonhole, 2019.

[8] F. Akgul. *ZeroMQ*. Packt Publishing Ltd, 2013.

[9] G. Alain, A. Lamb, C. Sankar, A. Courville, and Y. Bengio. Variance reduction in sgd by distributed importance sampling. *arXiv preprint arXiv:1511.06481*, 2015.

[10] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *Proceedings of the Advances in Neural Information Processing Systems*, pages 1709–1720, 2017.

[11] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[12] D. Bastos, M. Shackleton, and F. El-Moussa. Internet of things: A survey of technologies and security risks in smart home and city environments. 2018.

[13] C. Canel, T. Kim, G. Zhou, C. Li, H. Lim, D. G. Andersen, M. Kaminsky, and S. R. Dulloor. Scaling video analytics on constrained edge nodes. *arXiv preprint arXiv:1905.13536*, 2019.

[14] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314, 2011.

[15] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 613–627, 2017.

[16] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl.

Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62, 2010.

[17] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 4. ACM, 2016.

[18] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proceedings of the 2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[19] S. Dutta, G. Joshi, S. Ghosh, P. Dube, and P. Nagpurkar. Slow and stale gradients can win the race: Error-runtime trade-offs in distributed sgd. *arXiv preprint arXiv:1803.01113*, 2018.

[20] B. Fang, X. Zeng, F. Zhang, H. Xu, and M. Zhang. Flexdnn: Input-adaptive on-device deep learning for efficient mobile vision. In *Proceedings of the 5th ACM/IEEE Symposium on Edge Computing (SEC)*, 2020.

[21] B. Fang, X. Zeng, and M. Zhang. 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)*, pages 115–127, 2018.

[22] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[23] M. Gürbüzbalaban, A. Ozdaglar, and P. Parrilo. Why random reshuffling beats stochastic gradient descent. *arXiv preprint arXiv:1510.08560*, 2015.

[24] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[25] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. In *Proceedings of the 2nd SysML Conference*, 2019.

[26] C. He, S. Li, J. So, M. Zhang, H. Wang, X. Wang, P. Vepakomma, A. Singh, H. Qiu, L. Shen, et al. Fedml: A research library and benchmark for federated machine learning. *arXiv preprint arXiv:2007.13518*, 2020.

[27] K. He, G. Gkioxari, P. Dollár, and R. Girshick. Mask r-cnn. In *Computer Vision (ICCV), 2017 IEEE International Conference on*, pages 2980–2988. IEEE, 2017.

[28] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In

Proceedings of the IEEE conference on computer vision and pattern recognition, pages 770–778, 2016.

[29] G. Hinton, L. Deng, D. Yu, G. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, B. Kingsbury, et al. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal processing magazine*, 29, 2012.

[30] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[31] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[32] K. Hsieh, G. Ananthanarayanan, P. Bodik, P. Bahl, M. Philipose, P. B. Gibbons, and O. Mutlu. Focus: Querying large video datasets with low latency and low cost. *arXiv preprint arXiv:1801.03493*, 2018.

[33] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu. Gaia: Geo-distributed machine learning approaching {LAN} speeds. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 629–647, 2017.

[34] T.-M. H. Hsu, H. Qi, and M. Brown. Measuring the effects of non-identical data distribution for federated visual classification. *arXiv preprint arXiv:1909.06335*, 2019.

[35] C.-C. Hung, G. Ananthanarayanan, P. Bodik, L. Golubchik, M. Yu, P. Bahl, and M. Philipose. Videoedge: Processing camera streams using hierarchical clusters. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 115–131. IEEE, 2018.

[36] L. N. Huynh, Y. Lee, and R. K. Balan. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 82–95, 2017.

[37] F. N. Iandola, M. W. Moskewicz, K. Ashraf, and K. Keutzer. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2592–2600, 2016.

[38] S. Jain, G. Ananthanarayanan, J. Jiang, Y. Shu, and J. Gonzalez. Scaling video analytics systems to large camera deployments. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, pages 9–14, 2019.

[39] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko. Priority-based parameter propagation for distributed dnn training. *arXiv preprint arXiv:1905.03960*, 2019.

[40] J. Jiang, G. Ananthanarayanan, P. Bodik, S. Sen, and I. Stoica. Chameleon: scalable adapta-

tion of video analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 253–266. ACM, 2018.

[41] J. Jiang, Y. Zhou, G. Ananthanarayanan, Y. Shu, and A. A. Chien. Networked cameras are the new big data clusters. In *Proceedings of the 2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges*, pages 1–7, 2019.

[42] S. Jiang, Z. Ma, X. Zeng, C. Xu, M. Zhang, C. Zhang, and Y. Liu. Scylla: Qoe-aware continuous mobile vision with fpga-based dynamic deep neural network reconfiguration. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 1369–1378. IEEE, 2020.

[43] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings, et al. Advances and open problems in federated learning. *arXiv preprint arXiv:1912.04977*, 2019.

[44] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. Noscope: optimizing neural network queries over video at scale. *Proceedings of the VLDB Endowment*, 10(11):1586–1597, 2017.

[45] S. P. Karimireddy, S. Kale, M. Mohri, S. Reddi, S. Stich, and A. T. Suresh. Scaffold: Stochastic controlled averaging for federated learning. In *International Conference on Machine Learning*, pages 5132–5143. PMLR, 2020.

[46] A. Katharopoulos and F. Fleuret. Not all samples are created equal: Deep learning with importance sampling. *arXiv preprint arXiv:1803.00942*, 2018.

[47] J. Konečnỳ, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*, 2016.

[48] A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[49] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[50] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[51] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, Broomfield, CO, Oct. 2014. USENIX Association.

[52] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J.

Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, 2014.

[53] T. Li, A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, and V. Smith. Federated optimization in heterogeneous networks. *arXiv preprint arXiv:1812.06127*, 2018.

[54] Y. Li, M. Yu, S. Li, S. Avestimehr, N. S. Kim, and A. Schwing. Pipe-SGD: A decentralized pipelined sgd framework for distributed deep net training. In *Proceedings of the Advances in Neural Information Processing Systems*, pages 8045–8056, 2018.

[55] H. Lim, D. G. Andersen, and M. Kaminsky. 3lc: Lightweight and effective traffic compression for distributed machine learning. *arXiv preprint arXiv:1802.07389*, 2018.

[56] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

[57] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[58] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell. Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270*, 2018.

[59] I. Loshchilov and F. Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

[60] L. Ma, D. Van Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 631–645. ACM, 2018.

[61] X. Ma, H. Zhong, Y. Li, J. Ma, Z. Cui, and Y. Wang. Forecasting transportation network speed using deep capsule networks with nested lstm models. *IEEE Transactions on Intelligent Transportation Systems*, 2020.

[62] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics*, pages 1273–1282. PMLR, 2017.

[63] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, et al. Communication-efficient learning of deep networks from decentralized data. In *Proceedings of the 20 th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2017.

[64] Q. Meng, W. Chen, Y. Wang, Z.-M. Ma, and T.-Y. Liu. Convergence analysis of distributed stochastic gradient descent with shuffling. *arXiv preprint arXiv:1709.10432*, 2017.

[65] S. A. Noghabi, L. Cox, S. Agarwal, and G. Ananthanarayanan. The emerging landscape of edge computing. *GetMobile: Mobile Computing and Communications*, 23(4):11–20, 2020.

[66] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[67] S. Reddi, Z. Charles, M. Zaheer, Z. Garrett, K. Rush, J. Konečnỳ, S. Kumar, and H. B. McMahan. Adaptive federated optimization. *arXiv preprint arXiv:2003.00295*, 2020.

[68] J. Redmon and A. Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[69] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[70] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

[71] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram. Nexus: a gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019.

[72] R. Shokri and V. Shmatikov. Privacy-preserving deep learning. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 1310–1321. ACM, 2015.

[73] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[74] V. Smith, C.-K. Chiang, M. Sanjabi, and A. S. Talwalkar. Federated multi-task learning. In *Advances in Neural Information Processing Systems*, pages 4427–4437, 2017.

[75] tensorflow. Tensorflow speech command dataset, 2018.

[76] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Proceedings of the Advances in neural information processing systems*, pages 5998–6008, 2017.

[77] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 374–389. ACM, 2017.

[78] J. Wang and G. Joshi. Adaptive communication strategies to achieve the best error-runtime trade-off in local-update SGD. *arXiv preprint arXiv:1810.08313*, 2018.

[79] J. Wangni, J. Wang, J. Liu, and T. Zhang. Gradient sparsification for communication-efficient distributed optimization. In *Proceedings of the Advances in Neural Information Processing Systems*, pages 1299–1309, 2018.

[80] P. Watcharapichat, V. L. Morales, R. C. Fernandez, and P. Pietzuch. Ako: Decentralised deep learning with partial gradient exchange. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 84–97. ACM, 2016.

[81] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in neural information processing systems*, pages 1509–1519, 2017.

[82] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[83] G.-S. Xia, J. Hu, F. Hu, B. Shi, X. Bai, Y. Zhong, L. Zhang, and X. Lu. Aid: A benchmark data set for performance evaluation of aerial scene classification. *IEEE Transactions on Geoscience and Remote Sensing*, 55(7):3965–3981, 2017.

[84] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data*, 1(2):49–67, 2015.

[85] B. Zhang, X. Jin, S. Ratnasamy, J. Wawrzynek, and E. A. Lee. Awstream: adaptive wide-area streaming analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 236–252. ACM, 2018.

[86] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *NSDI*, volume 9, page 1, 2017.

[87] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on {GPU} clusters. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 181–193, 2017.

[88] M. Zhang, F. Zhang, N. D. Lane, Y. Shu, X. Zeng, B. Fang, S. Yan, and H. Xu. Deep learning in the era of edge computing: Challenges and opportunities. *Fog Computing: Theory and Practice*, pages 67–78, 2020.

[89] S. Zhang, A. E. Choromanska, and Y. LeCun. Deep learning with elastic averaging sgd. *Advances in neural information processing systems*, 28:685–693, 2015.

[90] X. Zhang, J. Zhao, and Y. LeCun. Character-level convolutional networks for text classification. In *Proceedings of the Advances in neural information processing systems*, pages 649–657, 2015.

[91] P. Zhao and T. Zhang. Stochastic optimization with importance sampling for regularized loss minimization. In *Proceedings of the International Conference on Machine Learning*, pages 1–9, 2015.

[92] C. Zhu, S. Han, H. Mao, and W. J. Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.

[93] Z. Zivkovic and F. Van Der Heijden. Efficient adaptive density estimation per image pixel for the task of background subtraction. *Pattern recognition letters*, 27(7):773–780, 2006.