

MDLdroid: a ChainSGD-reduce Approach to Mobile Deep Learning for Personal Mobile Sensing

Yu Zhang , Student Member, IEEE, Tao Gu , Senior Member, IEEE, Member, ACM, and Xi Zhang 

Abstract—Personal mobile sensing is fast permeating our daily lives to enable activity monitoring, healthcare and rehabilitation. Combined with deep learning, these applications have achieved significant success in recent years. Different from conventional cloud-based paradigms, running deep learning on devices offers several advantages including data privacy preservation and low-latency response for both model inference and update. Since data collection is costly in reality, Google’s Federated Learning offers not only complete data privacy but also better model robustness based on data from multiple users. However, personal mobile sensing applications are mostly user-specific and highly affected by environment. As a result, continuous local changes may seriously affect the performance of a global model generated by Federated Learning. In addition, deploying Federated Learning on a local server, e.g., edge server, may quickly reach the bottleneck due to resource limitation. Towards pushing deep learning on devices, we present MDLdroid, a novel decentralized mobile deep learning framework to enable resource-aware on-device collaborative learning for personal mobile sensing applications. To address resource limitation, we propose a ChainSGD-reduce approach which includes a novel *chain-directed* Synchronous Stochastic Gradient Descent algorithm to effectively reduce overhead among multiple devices. We also design an agent-based *multi-goal* reinforcement learning mechanism to balance resources in a fair and efficient manner. Our evaluations show that our model training on off-the-shelf mobile devices achieves 2x to 3.5x faster than single-device training, and 1.5x faster on average than the existing master-slave approach.

Index Terms—Mobile deep learning, neural networks, distributed computing, resource allocation, reinforcement learning.

I. INTRODUCTION

WITH the rapid development of mobile and wearable devices, recent years have witnessed an explosion of mobile sensing applications. These applications gain an insight into people’s life based on rich personal sensing data, e.g., understanding biological contexts in daily living [1], recognizing activities in ambient assisted living areas [2], and monitoring personal health in a smart home or hospital [3]. Machine learning has been commonly used to process sensing data. However, traditional machine learning techniques require manual and complex feature engineering. Deep Learning (DL) has gained an increasing popularity due to its higher model accuracy. Besides, its automated feature extraction capability and the ability of scaling with data make it an ideal solution for processing multi-modal sensing data [4]. It is advocated that DL will be the next key enabler for advanced personal mobile sensing [5].

Y. Zhang and X. Zhang are with the School of Computer Science and IT, RMIT University, Melbourne, VIC 3000, Australia (E-mail: zac.lhjzyzoo@gmail.com; zaibuer@gmail.com).

T. Gu is with the Department of Computing, Macquarie University, NSW 2109, Australia (E-mail: tao.gu@mq.edu.au).

A successful DL application requires a huge amount of data to train a model, in which large computation resources will be involved. The commercial solution is to transfer data from local to cloud, *offloading* heavy training workloads to the cloud [6] or edge servers [7], and then downloading the pre-trained models [5] for on-device inference. However, personal sensing applications are intrinsically highly *privacy-sensitive* [8], *user-specific* (i.e., different preferences or health conditions) [2], *low-latency* interactive [9], and can be easily affected by *local scenario changes* (i.e., long-term behavior changes or ambient environment changes) [5], thus the cloud-based approach may suffer from severe data privacy concerns, compromised personalization, inferior model performance without *continual training*, and unacceptable network latency due to data transfer and model downloading [10]. Although transfer [11] and meta learning [12] can adapt to user-specific models, they may still suffer from strong *domain-shift* by sensing data dynamics and lack of related source datasets or models. Mobile DL can effectively preserve sensing data privacy (i.e., no data transfer), and enable quick local model inference and update response for different on-device learning purposes (i.e., training from scratch, continual training or model adaptation) [4], [10], thus presenting a promising direction for personal sensing applications.

Mobile Deep Learning Challenges Existing work [4] reveals that mobile DL poses two challenges—*resource constraint* and *insufficient sensing data* (i.e., costly data collection and labeling with various real-world conditions). Collaborative deep learning, e.g., Google’s Federated Learning (FL) [13], has been proposed to ensure model training efficiency and robustness based on *multiple* users’ data [14]. Since FL mainly relies on a central cloud or edge sever for intensive global *model aggregations* (i.e., a large number of selected users involved) during model training, several limitations arise in the context of personal sensing applications, e.g., non-independent and identically distributed (non-IID) personal data problem [15], de-personalization issue and considerable network latency [10], lower resistance to attacks and less resource efficiency based on the intrinsic master-slave network structure [16]–[18].

In contrast, applying a decentralized structure for mobile DL can theoretically offer high attack resistance and reliable fault tolerance [18]. Existing approaches propose a theoretical decentralized structure based on a *fixed directed* graph network such as Ring-Allreduce [19]. These approaches have been applied in high-performance cloud environments where resources are rich and stable. However, when applied to mobile devices where their resources are scarce and dynamic, the training process can easily suspend or crash due to the low level of resources available. Our work is motivated by

collaborative DL, but we move towards *local resource-aware decentralized collaborative* mobile DL without central server support for personal sensing applications. Our approach aims to mitigate resource overhead and reduce latency for efficient model aggregations during training.

Decentralized Mobile Scheduling Since resources on devices are constrained and heterogeneous (i.e., the *Non-stationary* problem), the design of *resource-aware* scheduling for efficient model aggregation can be extremely difficult in a decentralized DL framework [20]. In addition, a collaborative DL process typically yields large communication overhead in model aggregation [21], energy balance among multiple devices can be critical. A generic optimization algorithm [22] can be applied to perform task scheduling, but the recent *Multi-agents Reinforcement Learning* (MARL) approach [23] may work better in such a *non-stationary* and multi-device scenario. However, when applied to resource-constrained devices, MARL-based scheduling may experience severe resource conflict which will affect or interrupt training.

Our Approach Aiming to push DL to mobile devices, we present MDLDroid, a novel decentralized **Mobile Deep Learning** framework to enable resource-aware on-device collaborative learning for personal mobile sensing applications. MDLDroid targets to fully operate on multiple off-the-shelf **Android** smartphones connected in a mesh network, and achieve high training accuracy and reliable execution of the state-of-the-art DL models.

In the previous version of MDLDroid (MDLDroid-v1), we have addressed two major challenges.

- To address the challenge of *resource constraint* on device, we propose a *ChainSGD-reduce* approach which essentially uses a novel *chain-directed* Synchronous Stochastic Gradient Descent (S-SGD) algorithm to effectively reduce resource overhead among devices for training. The key idea is to decentralize the S-SGD algorithm [24] running on a single device to multiple devices with dynamic *chain-directed* model aggregation. Specifically, each device runs a descendant model for training in which model aggregation task is managed by any two devices at a time to achieve *minimal-peak* (i.e., minimum peak memory and communication) of resource overhead for each device.
- Due to the constrained and heterogeneous resource on devices, a *resource-aware* scheduling is critical for efficient model aggregation in a decentralized DL framework. To this end, we propose a single agent-based scheduler based on Reinforcement Learning (RL) in the ChainSGD-reduce approach, named *Chain-scheduler*, aiming to self organize scheduling tasks using dynamic information of on-device resource. Chain-scheduler includes three unique techniques: 1) an effective reward function design to map each perceived scheduling action to a reward, aiming to optimize the given constraints; 2) a continuous environment learning strategy to repeat learning if the scheduling environment is changed; 3) a Threshold-based Decaying Greedy-Exploration (TDGE) strategy to further accelerate the RL learning process, aiming to save on-device resources used for scheduling tasks.

In this work, we extend MDLDroid-v1 to MDLDroid-v2 by significantly improving its overall performance in terms of

training accuracy and resource efficiency when scaling up the network size.

- Training accuracy of ChainSGD may have a notable decrease when the network size increases in MDLDroid-v1, yielding inferior model performance. Although largely increasing training and model aggregation iterations may recover the accuracy to the state-of-the-arts, the resource overhead may easily arise beyond the capacity of devices. To meet both resource-constraint and network scaling requirements, we propose a novel ChainSGD gradient-scale method based on the L1 weight normalization analysis [25] to ensure training accuracy and maintain model performance.
- Learning scalability of our Chain-scheduler may be limited by network scaling in MDLDroid-v1, which may seriously slow down the entire end-to-end training process (i.e., slow RL convergence by network scaling) leading to large resource overhead on device. To address this issue, we propose a momentum-based penalty function to dynamically adapt the proposed scheduling reward threshold to make the learning of the scheduler more resource-efficient.
- We comprehensively improve the system performance and end-to-end experience from a number of key aspects, e.g., training speed-up on single device, fast peer-to-peer communication, scheduling scalability, efficient aggregation timeout mechanism, and model attack defense.

To further explore the end-to-end performance of our system in reality, we present a case study using a real-world gesture recognition application. The results show that MDLDroid achieves a reliable training accuracy with low resource overhead.

Key contributions of this paper are summarized as follows.

- To the best of our knowledge, MDLDroid presents the first decentralized mobile DL framework based on a mesh network to enable resource-aware on-device collaborative learning for personal mobile sensing applications (§III).
- We propose a novel ChainSGD-reduce approach, in particular a *chain-directed* S-SGD algorithm, to minimize the resource overhead of model aggregation tasks in a decentralized framework (§III-B).
- We design an agent-based *multi-goal* reinforcement learning mechanism, Chain-scheduler, with an *accelerated* reward function to manage and balance resources in a fair and efficient manner (§III-C).
- We evaluate MDLDroid on off-the-shelf Android smartphones with a number of state-of-the-art DL models using 6 public personal mobile sensing datasets (§V). Results indicate that MDLDroid accelerates training effectively, outperforming the state-of-the-arts.

Implication Towards a mobile DL paradigm, MDLDroid offers a resource-aware mobile collaborative DL framework for privacy-preserving personal sensing applications, exploring specific local sensing features and guaranteeing the performance of DL models. Leveraging on-device collaborative learning, people can directly build their models using a small amount of data on smartphones, and easily exchange with their friends or colleagues' models to ensure a state-of-the-art model performance without data privacy concerns. In practice, the proposed ChainSGD-reduce approach can be widely applied in

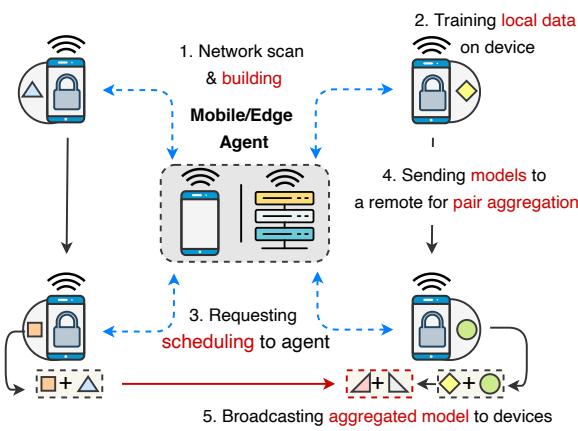


Fig. 1: MDLdroid Architecture and Workflow.

various learning scenarios (*e.g.*, collaborative and distributed learning models), especially for the resource heterogeneous and constrained DL scenarios, and scale to network size. To further extent, MDLdroid can be applied to other edge devices (*e.g.*, embedded and Internet of Things (IoT) devices) for building collaborative edge intelligence and IoT applications.

II. MOTIVATION

To investigate the limitations of deploying mobile DL on device in current solutions, we conduct four preliminary studies to discover the bottleneck of on-device collaborative learning, which motivates our proposal. The results are summarized as follows: 1) single-device training on off-the-shelf smartphone is inefficient due to resource constraints; 2) the centralized framework (*e.g.*, FL [13]) has severe memory limitations for master model aggregation on device; 3) the sequential model aggregation (*i.e.*, run-time memory usage is $O(2)$) on master device can still cause considerable extra training time cost and accuracy degradation; 4) severe resource conflict (*i.e.*, memory conflict between training and agent tasks) exists in on-device MARL. Please refer to our conference paper [26] for the details of these experiments.

III. MDLDROID FRAMEWORK

In this section, we detail the system architecture of MDLdroid, and present the proposed ChainSGD-reduce approach.

A. System Architecture

Since the framework is designed to operate full-scale DL on Android based on a mesh network, we employ Bluetooth Low Energy (BLE), Bluetooth Socket (BS) in MDLdroid-v1 and Wi-Fi Direct (WD) in MDLdroid-v2 to build the mesh network due to accessibility and low energy consumption. In principle, any on-device mesh-based protocol can be applied (§III-B). Especially, MDLdroid can also reload a pre-trained model to continually train with new local sensing data. In short, Fig. 1 presents a complete workflow of MDLdroid, where a *mobile agent* (MA) can be deployed on either a mobile device or a stationary edge server for task scheduling.

B. ChainSGD-reduce Approach

Collaborative learning relies on the distributed *stochastic gradient descent* (SGD) algorithms [13], [27]. To achieve reliable training accuracy, a central server typically gathers local gradient parameters Δw_i^k from all machines and aggregates them to be global gradient parameters $\Delta w^{(k)}(t)$ after each training iteration. The central server then updates the local parameters $w^{(k)}(t+1)$ for all machines via an *one-to-many* broadcasting:

$$\Delta w^{(k)}(t) = \frac{1}{N} \sum_{i=1}^N \Delta w_i^k \quad (1)$$

$$w^{(k)}(t+1) = w^{(k)}(t) - \eta \Delta w^{(k)}(t)$$

where $w^{(k)}(t)$ denotes the $k^{(th)}$ parameters at each training iteration t , Δw_i denotes a local gradient parameter from the i th machine, N is the number of machines and η is the learning rate.

Asynchronous SGD vs. Synchronous SGD The distributed SGD algorithms are mainly classified into two categories—Asynchronous SGD (A-SGD) and S-SGD [24]. A-SGD can be more communication efficient and runs with no strong dependency among machines, but may suffer from an uncertain training accuracy degradation issue due to its delay model updating mechanism [28]. S-SGD representing in Eq. (1), on the other hand, runs quite stable without this issue, but the overall training time depends on the slowest machine [24]. By given the *resource-constrained* training condition on mobile device, S-SGD can effectively achieve a higher accuracy and more reliable training performance than A-SGD, which motivates S-SGD in our approach.

Chain-directed Synchronous SGD In a decentralized DL framework, the relationship between training nodes and the central node is decoupled. Therefore, the centralized model aggregation can be separated into multiple descendant aggregations by structural transformation. The existing work [29] present decentralized SGD algorithms based on a *fixed directed* network. We define a decentralized topology with a *fixed directed* graph as (V, E) , where V denotes a set of devices and E represents a set of edges. When we have N training devices, $V = \{1, 2, \dots, N\}$ and $E \in \mathbb{R}^{V \times V}$. We define *directed* edges as $(i, j) \in E$, which means device i can send its gradient parameter to device j for model aggregation. The number of device j 's neighbors is m , and the number of device j is n , where $m, n \in N$. With these settings, we can transform the centralized model aggregation (CentralSGD) in Eq. (1) to the decentralized neighbor aggregation (NeighborSGD) as follows.

$$\Delta w^{(k)}(t) = \frac{1}{n} \sum_{j=1}^n \left(\frac{\sum_{i=1}^m \Delta w_i^k + m \Delta w_j^k}{2m} \right) \quad (2)$$

In NeighborSGD, each device j first receives all gradient parameters $\frac{\sum_{i=1}^m \Delta w_i^k}{m}$ from its m neighbors and aggregate with its local gradient parameter. The global gradient parameters will then be averaging of n device j s at each training iteration t . However, although the model aggregation is divided by j descendant neighbor aggregations, the single device j still requires to concurrently perform m model aggregations, which may cause significant memory overhead revealed by

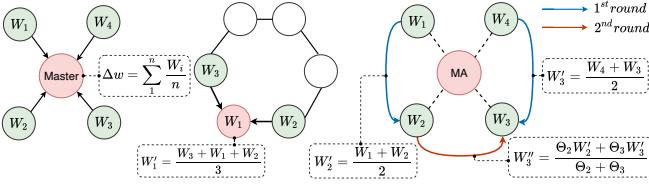


Fig. 2: Model aggregation structure comparison: CentralSGD vs. NeighborSGD vs. ChainSGD. (W_i' denotes the gradient parameters of each device and the prime symbol denotes the number of aggregation rounds.)

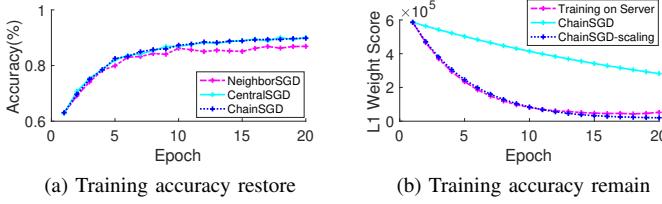


Fig. 3: Training accuracy preserving using ChainSGD.

our preliminary study. On the other hand, considering the dynamicity of resources on device is uncertain during training, the neighbor model aggregation based on the *fixed directed* decentralized topology may cause distinct latency if device j pauses the process due to low-resource condition.

To reduce memory overhead and latency, we propose a ChainSGD-reduce approach with a mesh-based decentralized topology. In this approach, m is constantly managed as one for every neighbor aggregation to achieve a *minimal-peak* (*i.e.*, minimal peak memory footprint and communication overhead for each model aggregation) resource overhead for both devices i and j . Our approach also includes an agent-based RL Chain-scheduler to schedule the neighbor aggregation task as a dynamic *chain-directed* graph in a resource-efficient way.

Compared to both CentralSGD and NeighborSGD, Fig. 2 demonstrates that the major differences of ChainSGD-reduce are **twofold**: 1) the model aggregation is managed only with *one* of neighbors at a time on each device for resource overhead reduction (*i.e.*, pair-wise model aggregation); 2) the order of the aggregation tasks at each training iteration is dynamically *scheduled* (*i.e.*, dynamic topology enabled by Chain-scheduler) depending on the real-time heterogeneous resource conditions of devices (§III-C).

When ChainSGD-reduce is applied to NeighborSGD in Eq. (2), one of our key observations is that the training accuracy notably degrades comparing to that of CentralSGD. Fig. 3a shows that the training accuracy of NeighborSGD using LeNet on HAR is lower than that of CentralSGD by 3% when the number of connected devices is 6 (*i.e.*, $N = 6$). The accuracy gap between NeighborSGD and CentralSGD becomes larger with N increases. It is essentially because that the global gradient parameters $\Delta w^{(k)}(t)$ decompose through the model aggregation process using NeighborSGD. To restore training accuracy, we thus formulate ChainSGD as the following *pair*

aggregation function $W(j, i)$.

$$W(j, i) = \begin{cases} \Delta w_i^k = \frac{\theta_i \Delta w_i^k + \theta_j \Delta w_j^k}{\theta_i + \theta_j} \\ \theta'_i = \theta_i + \theta_j \quad \theta^o = 1 \end{cases} \quad (3)$$

where θ is a *reversal* parameter, and $\theta(t)$ represents the number of devices for model aggregation at iteration t . θ^o denotes as an initial value of the reversal parameter as default 1 on each device before each model aggregation iteration. Specifically, as the reversal parameter θ on each device records how many devices are processed by pair aggregation in a iteration, the pair-aggregated gradient parameters (*e.g.*, both of the local Δw_i^k and remote Δw_j^k in Eq. (3)) can be easily reversed to the sum of the gradient parameters by multiplying θ_i and θ_j , respectively, before the current pair aggregation. In this reversal way, the aggregated gradient parameters by $W(j, i)$ can be equally restored to CentralSGD, which can be simply proved by mathematical induction. Once a *pair aggregation* is done, the local θ_i will be updated with a remote θ_j as θ'_i for the next iteration. The global gradient parameters of each iteration represented as $\Delta w^{(k)}(t)$ are processed with $N - 1$ times of pair aggregations $W(j, i)$ (*e.g.*, N denotes the number of connected devices), hence each model aggregation iteration will be stopped to finalize the global gradient parameters once the $\theta(t)$ reaches N . Besides, each pair of devices is dynamically scheduled by Chain-scheduler (§III-C) to perform the pair aggregation in each iteration. In our ChainSGD-reduce protocol, the message for model aggregation requires both the local gradient parameters Δw^k and the *reversal* parameter θ encoded in a $(\Delta w^k, \theta)$ packet.

Since ChainSGD-reduce offers a dynamic model aggregation process, multiple *pair aggregation* can be performed in *parallel* based on resource conditions of devices within each iteration to reduce latency showing in Fig. 2. In addition, Fig. 3a presents a pre-result that the training accuracy of ChainSGD is consistent with that of CentralSGD with no accuracy drop.

As ChainSGD essentially requires the local gradient parameters Δw_i^k as input in Eq. 3 to offer a higher-level SGD process for efficient model aggregations, most of gradient descent optimization algorithms (*i.e.*, classic or momentum-based SGD optimizations) are compatible to output the Δw_i^k in MDLdroid. In practice, we employ Adam optimizer [30] as default to achieve a fast and stable learning convergence performance.

ChainSGD Gradient-scale With the network size increases, though ChainSGD in Eq. (3) performs an identical training accuracy as CentralSGD, we observe that both approaches present a serious *slow* learning convergence rate (*i.e.*, training loss rate) by the given training iterations compared to training on server with a full-size dataset, resulting in a severe training accuracy drops. Although we can practically increase either the number of synchronization rounds (*i.e.*, model aggregation rounds) or training iterations to reach the same accuracy on server, it may lead to resource overhead which is beyond the capacity of a device. To meet both resource-constrained and scaling requirements, we propose a ChainSGD gradient-scale to guarantee a fast convergence rate in each model aggregation process.

To analyze the slow learning convergence issue in ChainSGD, we apply L1 weight normalization analysis [25] which is an effective way to monitor dynamic changes of model parameters during training. In particular, we reveal that the average decay rate of L1 weight score (*i.e.*, the value of model parameters by L1 normalization process) on each device is much slower than that of training on server, shown in Fig. 3b based on a 6-device connected network by training LeNet on HAR. With this observation, our key idea of gradient-scale is to *reinforce* the decay of L1 weight score by scaling down the model parameters after local gradient update in Eq. (1) at each iteration. To calculate the gradient-scale at each iteration, we first scale up $\theta(t)$ times by L1 weight score of the global gradient parameters as $\theta(t)N1(\Delta w^k(t))$. We then measure the absolute difference between the local L1 weight score as $N1(w_i^k(t))$ and the scale-up global L1 weight score, and define that the difference divided by the maximum value between local and global L1 weight scores represents the "reinforced" gradient-scale. Fig. 3b presents the pre-result that the decay rate of L1 weight score using the gradient-scale can match with or even lower than that of training on server, resulting in an identical or higher training accuracy performance within the same training iterations. In short, we formulate the ChainSGD gradient-scale as follows.

$$S(t) = \left| \frac{N1(w_i^k(t)) - \lambda \theta(t)N1(\Delta w^k(t))}{\max(N1(w_i^k(t)), \theta(t)N1(\Delta w^k(t)))} \right| \quad (4)$$

where $N1$ represents as a L1 weight normalization function, and λ denotes as a decay coefficient as default 0.95.

Fault-tolerant and Resource-aware Broadcasting MDL-droid offers a fault-tolerant strategy to ensure the stability for on-device training: 1) *File cache*: model parameters and necessary records can be saved as files backup in run-time for training recovery; 2) *Training device fault*: once a training device is lost after the re-connection attempts, *File cache* will be first performed. Then MA (*i.e.*, the mobile agent deployed on device or edge server) will skip the device for scheduling until reconnected. If re-connection is successful, MA will require the device to send over the last model parameters and iteration records. After that, MA will send back the current iteration records for synchronization, and the device will receive the global aggregated model parameters until next iteration; 3) *MA scheduling failure*: if the training devices cannot connect to MA to get scheduling paths for model aggregation (*i.e.*, MA may go down during run-time), each device will continually communicate with each remote device by the previous scheduling paths to complete model aggregation. Once MA is reconnected, scheduling service will be recovered without suspending the training process.

In ChainSGD-reduce, we employ a binomial-tree broadcast algorithm [31] with a message forwarding function to reduce the number of broadcast round from N to $\lceil \log_2 N \rceil$. Please refer to our conference paper [26] for the detailed design of the resource-aware broadcasting.

Aggregation Timeout Mechanism Since ChainSGD-reduce builds on S-SGD, the training process may experience long latency for low-resource devices, *e.g.*, the training on low-resource devices (*i.e.*, the conditions including low memory, low battery or multi-process occupancy) will be temporarily

suspended and the state of the device in our system turns to *busy* (S_{busy}) until being *free* (S_{free}) (§III-C). To avoid the latency, we design a timeout mechanism for the pair aggregation. For the *busy* devices, if the latency is larger than a given fixed time window (*e.g.*, we set 5mins as default), the Chain-scheduler on MA will schedule to skip their gradient parameters sync until next aggregation iteration. If the other *free* devices complete all pair aggregations before the time window, they will be managed as "sleep" for saving resource until reaching the time window.

Model Attack Defense Although a decentralized framework is theoretically high attack-resistant and reliable fault-tolerant [18], the security issues on both communication and model aggregation may still be concerned in reality. To improve overall security of MDLdroid, we apply MD5 to encrypt all messages by the communication component. Besides, we employ the model poisoning check [17] before each pair aggregation to prevent and defense model attacks. If a remote model file is detected as poisoned, the local device will skip the pair aggregation, and ask MA to schedule a new remote to sync.

In summary, the ChainSGD-reduce algorithm is updated in Algorithm 1 below.

ALGORITHM 1: ChainSGD-reduce Algorithm

```

1 Initialize parameters  $w_0$ , number of iteration  $t$ , number of
   connected devices  $N$ , training dataset  $D_{train}$ ;
2 for All devices  $i \in N$  do
3   for  $t$  Iterations do
4     Train model on local  $d_{train}^t \in D_{train}$ ;
5     while true do
6       if Get an incoming message  $(\Delta w_j^k, \theta_j)$  then
7         Pair aggregation with local  $(\Delta w_i^k, \theta_i)$  in
          Eq. (3);
8         if  $\theta_i$  equals  $N$  then
9           Notice MA for broadcasting and Break;
10        else if Get a remote neighbor  $j$  scheduled by
          MA using the agent-based RL model in
          Algorithm 2 then
11          Send  $(\Delta w_i^k, \theta_i)$  to neighbor  $j$  and Break;
12        end
13        Get global  $(\Delta w^k(t), \theta(t))$  via broadcasting;
14        Send  $\Delta w^k(t)$  if assigned a forwarding list by MA;
15        Calculate gradient scale  $S(t)$  in Eq. (4) ;
16        Update local
           $w_i^{(k)}(t+1) = S(t)(w_i^k(t) - \eta \Delta w^k(t));$ 
17      end
18    end

```

C. Resource-aware Chain-scheduler

Since resources (*e.g.*, memory footprint, CPU usage, and battery consumption) on device are constrained strictly and changed dynamically (*i.e.*, the *Non-stationary* problem), a device may be dynamically switched to *busy* state due to other high priority tasks, leading to notable latency as training being suspended. Also, the energy balance is critical due to the communication overhead caused by the intensive model aggregations. Applying naive scheduling approaches (*e.g.*, using generic optimization algorithm [22]) may not be able to handle such complex non-stationary and multi-device

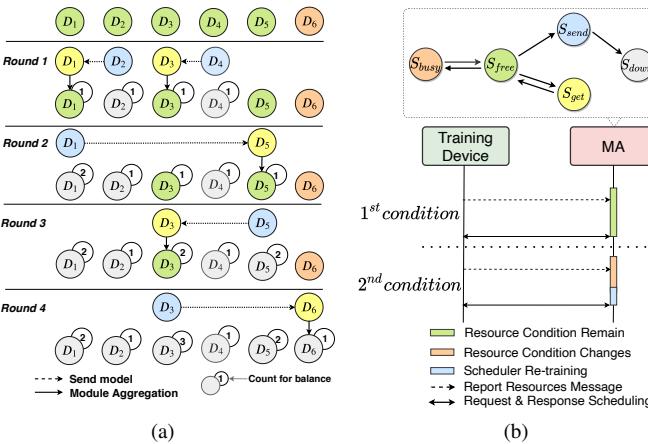


Fig. 4: (a) Process of scheduling by busy condition; (b) State transition diagram and re-training mechanism-v2.

scenario. To dynamically schedule model aggregation tasks for the heterogeneous resource-efficiency design in MDLDroid, we propose Chain-scheduler using an agent-based RL model based on dynamic on-device resource conditions to enable a *resource-aware* scheduling in our ChainSGD-reduce approach.

Due to the observation of the memory overhead using MARL analyzed in our preliminary experiment, we design a single agent-based RL mechanism in Chain-scheduler. The agent task is separated into an individual device as the MA which is responsible of resource-aware scheduling, and other devices are mainly responsible of running training tasks. With a mesh network, MA can globally monitor all training devices' resource condition in real-time with low-energy cost, and dynamically schedule model aggregation tasks for resource efficiency and energy balance. Hence, we define two optimal goals for Chain-scheduler, *e.g.*, reducing training latency caused by the *busy* devices, and balancing communication overhead and battery consumption across the network.

Mathematically, Chain-scheduler deals with the following constrained optimization function:

$$\arg \min N(T(t)) + N(E(t)) \quad (5)$$

s.t. $M_i \leq M_Max_i$, $B_i \leq B_Max_i$

where T and E denote *training time* and *energy balance* (*i.e.*, energy variance for devices) across the network, respectively. $N(x) = (x - x_{min}) / (x_{max} - x_{min})$ is a standard normalization function to transform *training time* and *energy balance* to be at the same scale. t is the time index which represents the current training iteration. We denote M_Max and B_Max as the maximum memory and maximum battery offered by each of target devices (*i.e.*, depending on different device specifications), respectively. Hence, our objective is to minimize both *training time* and *energy balance* at each iteration t in Eq. 5 by scheduling to achieve resource-efficiency. Please refer to our conference paper [26] for the detailed design of both T and E .

Multi-goal Reward Function We employ a DQN model [32] in MA to learn the scheduling environment based on the optimization function in Eq. (5). In our protocol, all train devices are required to continually report their resource condition to MA. We select five essential resource parameters

including: *free memory* (*i.e.*, remaining memory), *battery* (*i.e.*, remaining battery), *in-use* (*i.e.*, whether if under intensive use condition), *charge* (*i.e.*, whether if charging), *cpu* (*i.e.*, current CPU), to identify whether the device is in *busy* (S_{busy}) or *free* (S_{free}). Next, we present the design of the Chain-scheduler structure:

- **State:** We design five states for Chain-scheduler to make crucial scheduling decisions based on our protocol. $s = \{s_t, i = 1, 2, \dots, T\}$ represents as a set of devices' states, where t denotes the learning step. $States = (S_{free}, S_{busy}, S_{send}, S_{get}, S_{done})$, where S_{send} denotes that the device is sending model parameters, and S_{get} represents the device is getting model parameters. Specifically, the state of devices can be only defined as one of the five states at any learning step. Fig. 4b illustrates the transition relationship of these states.
- **Action:** We define $a = \{a_i, i = 1, 2, \dots, N\}$, where N denotes the number of devices. Each action a represents one of devices selected by the scheduler to "interact" (*i.e.*, sending model gradient parameters for pair aggregation) with other devices in the scheduling environment (*i.e.*, multiple devices connected in a decentralized framework), and the state of each device will transfer to the next after acting each action.

- **Reward:** $r = \{r_t, t = 1, 2, \dots, T\}$ is defined by the reward function $r(s_t, a, s_{t+1})$ in Eq. (6). Each action acted by the scheduler will achieve a reward value from the environment, and the scheduler uses the achieved reward values by different actions to optimize the "best" scheduling policy.

With these settings, we summarize the design of **reward function** based on a three-stage mechanism aiming to solve the optimization function in Eq. (5): 1) we design a decaying penalty function $\rho + tp/2(N - 1)$ aiming to set S_{busy} as the lowest priority to reduce training latency from concurrent perspective; 2) we design a penalty function $\alpha - \beta n_{agg}$ and an incentive function $\alpha + \beta n_{agg}$ for efficient energy balance, where n_{agg} records the count of $S_{free} \rightarrow S_{get}$; 3) we design a *termination* function to efficiently stop the learning if *invalid* action is selected or the learning step exceeds limits. For the detailed design of the reward model, please refer to our conference paper [26]. In summary, we present our **reward function** as follows.

$$r(t) = \begin{cases} \alpha - \beta n_{agg} & S_{free} \rightarrow S_{get} \\ \alpha + \beta n_{agg} & S_{free} \rightarrow S_{send} \\ \rho + tp/2(N - 1) & S_{busy} \\ -1 & Exceeds\ limits \\ -1 & Select\ invalid\ action \\ 1 & Completed \end{cases} \quad (6)$$

where α , β and ρ denote initial reward, initial busy penalty and battery cost per aggregation, respectively. n_{agg} denotes the times of pair aggregations. By default, we set α to -0.04 , β to 0.1 and ρ to -0.8 . Fig. 4a presents the learning process of Chain-scheduler.

Accelerated Reward Function To further accelerate the RL learning process, we propose a threshold-based decaying greedy-exploration (TDGE) strategy which extends the existing decaying greedy-exploration (DGE) strategy [33]. One key observation from our empirical study is that more exploration

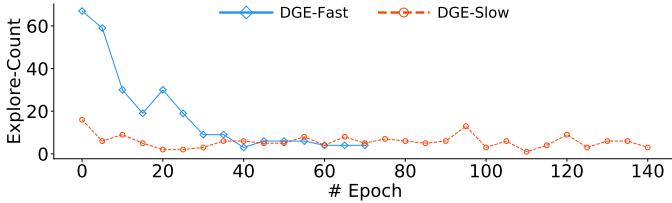


Fig. 5: RL training speed comparison by using different exploration rate.

achieves more learning time reduction. Fig. 5 shows DGE-Fast (using 2x exploration rate) completes earlier than DGE-Slow. Our intuition is to ensure the model to fully explore in the beginning until the reward reaches the given threshold, then switching the exploration to exploitation starting epsilon decaying from the predefined value, aiming to accelerate the learning process. Specifically, since the reward function is designed to optimize both latency and battery balance at the same time by Eq. (5), the threshold is approximately defined as the sum of each optimal reward value. The estimated reward value of optimal battery balance is defined as $f(N)$, where the estimated reward value of optimal latency is defined as $g(N)$, both shown in Eq. (8). Following this, we define the threshold $TR(N)$ by separating the calculation of the optimal reward into two functions in Eq. (9), which imitates an architecture in defining optimization function of Chain-scheduler.

$$f(N) = \begin{cases} \beta(N \bmod 2) + f(\lfloor N/2 \rfloor) & N > 1 \\ 0 & N = 0 \& 1 \end{cases} \quad (7)$$

$$g(N) = \beta \log_2 N \quad (8)$$

$$TR(N) = f(N) + g(N) - \Phi \quad (9)$$

where Φ denotes a fixed offset value as default 0.1.

Learning Scalability With the increase of connected devices, the proposed accelerated learning process may reach its performance limit for fast learning in practice. Through our experiments, we observe that RL has a notable slow convergence when scaling up devices. Since TDGE manages the exploration-to-exploitation control by the fixed threshold in Eq. (9), the model may fail to reach the threshold (*e.g.*, threshold is relatively large) to switch to exploitation in time by the scale-up condition, leading to slow convergence. To improve learning scalability, we propose to dynamically adapt the threshold to best-fit with the learning process. Technically, we extend the fixed offset value Φ in Eq. (9) to a *momentum-based* penalty function formulated as follows.

$$\Phi(t+1) = g\Phi(t) + (1-g)ReLU(r(t) - r(t-1)) \quad (10)$$

where g denotes as a momentum coefficient by default 0.95, and we set $\Phi(0)$ to 0 in the beginning of learning. Besides, we employ a *ReLU* function to ensure $\Phi(t)$ positive aiming to decay the threshold. In addition, the increment of $\Phi(t)$ is dynamically calculated by the growth gradient of the latest two reward values. Leveraging the penalty function, the threshold can be gradually decayed to not only effectively control the model exploration, but also efficiently enable switching to exploitation in an earlier time to achieve the fast convergence for scalability. In our evaluation, we mark this penalty-based exploration strategy as TDGE-v2.

Efficient Continuous Environment Learning As analyzed in the motivation section, the *Non-Stationary* scenario has a negative impact on the performance of the RL model. To address this, we design a repeating environment learning mechanism to mitigate the impact in MDLdroid-v1, including three *re-learning* conditions. However, the previous mechanism remains two major limitations. Firstly, since the RL learning process can be easily restarted by any resource condition changes on devices, the mechanism may be sensitive to the environment with high-frequent re-learning, resulting in considerable resource overhead. To wisely manage the re-learning, we simplify two conditions in MDLdroid-v2. Especially, in terms of the second condition shown in Fig. 4b, Chain-scheduler will only perform the re-learning *once* nearly the end of each training iteration if the environment changes, and dynamically manage to finish within the slot. Secondly, training Chain-scheduler from scratch to adapt different environments may be costly especially for initialization. To achieve resource efficiency, we simulate to train Chain-scheduler with a number of different scheduling environments in advance, and deploy a pre-trained RL model in practice to make the learning process more efficient than the previous version.

In summary, the Chain-scheduler algorithm is updated in Algorithm 2.

ALGORITHM 2: Chain-scheduler Algorithm

```

1 Initialize target network weights  $\theta$ , action-value function  $Q$ , experience memory  $D$ , state  $s$ , scheduling list  $L(s)$ , epsilon  $\epsilon = 1.0$ ,  $\epsilon_{new} = 0.3$ , decay rate  $\delta = 0.02$ , threshold  $TR$  in Eq. (9), penalty coefficient  $\Phi$  in Eq. (10);
2 while  $Epoch < maxEpoch$  do
3   for  $t \leftarrow 1$  to  $maxStep$  do
4     if  $random(0,1) < \epsilon$  then
5       Randomly explore an action from validAction();
6     else
7       Take action  $a$  with  $\epsilon$ -greedy policy based on  $\arg \max_{a_t} Q(L(s_t), a_t; \theta)$ ;
8     end
9     Get  $s_{t+1}$  and  $r_t$  by acting the action in environment using Eq. (6);
10    Set  $s_{t+1} = s_t$  and  $L_{t+1} = L(s_{t+1})$ ;
11    Store state transition  $(L_t, a_t, r_t, L_{t+1})$  in  $D$ ;
12    Randomly sample and label a batch of state transitions from  $D$ ;
13    Fit target network for scheduling policy learning;
14    if  $s_{t+1}$  is terminated then
15      if  $reward > TR$  then
16         $\epsilon \leftarrow \epsilon_{new}$  and mark the epsilon decaying as open;
17      else
18        update  $\Phi$  in Eq. (10) and  $TR$  in Eq. (9);
19      end
20      break;
21    end
22  end
23 update  $\epsilon$  by decaying  $\delta$  if open;
24 end

```

IV. SYSTEM IMPLEMENTATION

We implement MDLdroid based on an open-source DL library (*i.e.*, DL4J). In particular, we essentially modify DL4J to enable the proposed ChainSGD-reduce approach on device.

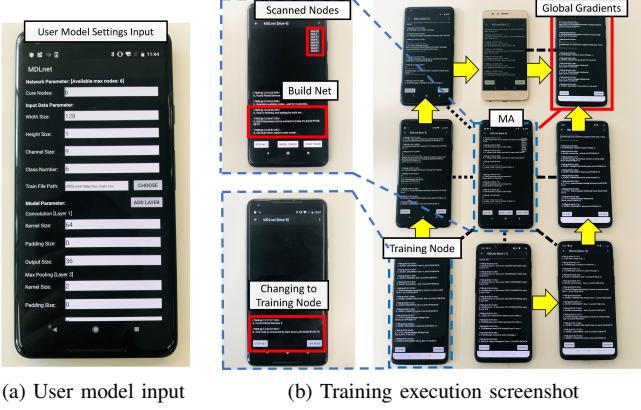


Fig. 6: Experiment screenshot.

We also tailor our implementation for execution on Android smartphone. We employ 9 off-the-shelf Android smartphones detailed in Table III and a laptop running Linux (*e.g.*, Ubuntu 19.04). Fig. 6 presents the experiment screenshots.

We further improve the system performance and end-to-end experience in MDLdroid-v2.

Training Speedup Our analysis shows that reading data from CSV files in MDLdroid-v1 seriously slows down training on single device. Since the input data of training represents as a multi-dimensional array and using CSV file can only save the data as 2D array, we develop an array shape converter to intensively transform the shape of each batch input data during loading data process, in which leads to a notable latency. To speed up training, we save all datasets to NPY files in MDLdroid-v2 since NPY file can save data in a buffered multi-dimensional array to efficiently improve the reading data and training speed. Besides, due to the nature of NPY file format, the size of all datasets has largely reduced by over 50% on average without losing accuracy, which is updated in Table I.

Fast Communication For accessibility and low-energy, in MDLdroid-v1 we use BLE and BS to build a mesh network. Due to the low-energy nature of BLE, the continual resource monitoring messages costs little, but the speed of sending model gradient parameters for pair aggregation presents a major slow-down. To provide a fast communication for model aggregation, we deploy Wi-Fi Direct (WD) on Android in MDLdroid-v2.

Scheduling Scalability In MDLdroid-v1, we select one smartphone as MA to run Chain-scheduler separately. With the network size rising up, the MA may reach the maximum resource threshold of the smartphone due to the dynamic continual environment learning. To improve scheduling scalability, in MDLdroid-v2 we deploy the MA on a Linux-based resourceful edge server. In addition, we offer two options to allow the MA either run on device or edge server depending on different usage scenarios (*e.g.*, on device for outdoor activity, and on edge server for indoor use).

Real-world Personal Sensing Application To demonstrate the end-to-end performance of MDLdroid in real-world applications, we apply MDLdroid in developing a privacy-preserving application, FinDroidHR [9], for hand gestures recognition. Fig. 7 shows the real locations of different users

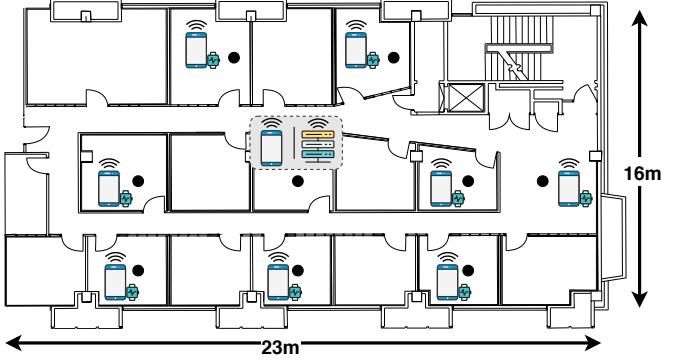


Fig. 7: Illustration of real-world locations using MDLdroid.

TABLE I: Dataset Specifications

Datasets	Type	Task	Subject	Class	Sample	Rate	#-C	#-TR	#-TE
sEMG [35]	EMG	GR	37	6	14695	50Hz	8	15	3
FinDroidHR [9]	IMU&HR	GR	8	6	2128	100Hz	7	13	3
MHEALTH [36]	IMU	HBM	10	9	3255	50Hz	23	40	18
UniMiB [37]	IMU	FDR	30	8	8430	50Hz	1	35	4
HAR [38]	IMU	ADLs	30	6	10299	50Hz	9	65	26
OPPORTUNITY [39]	IMU	ADLs	12	11	16837	50Hz	77	225	15
PAMAP2 [40]	IMU	ADLs	9	12	12397	100Hz	9	400	36

and the edge server running the MA. The result will be presented in the next section.

V. EVALUATION

In this section, we design three sets of experiments to evaluate MDLdroid extensively. We first evaluates the performance of Chain-scheduler compared with existing model aggregation scheduling approaches in distributed DL frameworks. We then compare the performance of MDLdroid with Federated Learning (FL) [13] in structure level and explore optimized resource-accuracy trade-off options (*i.e.*, model aggregation frequency and training iteration). Finally, we examine the training performance of the improved ChainSGD in MDLdroid-v2 compared with Gossip SGD (GoSGD) [34] in different data distribution scenarios, the resource efficiency of MDLdroid-v2 compared with different state-of-the-art DL models, and its end-to-end performance in reality.

A. Evaluation Set-up and Methodology

To evaluate MDLdroid, we select 6 public personal mobile sensing datasets and 1 self-collected dataset shown in Table I. These datasets are typically used for building a variety of personal mobile applications. We employ three state-of-the-art DL models including LeNet, MobileNet, and TCN with detailed model configurations shown in Table II. Practically,

TABLE II: Model configuration specifications

Models	Configuration (Type/Stride/Padding/Dilation)
LeNet [41]	Conv1/S1/P _{Same} ∈ [1, 36] → Pool/S2 → Conv2/S1/P _{Same} ∈ [1, 72] → Pool/S2 → FC ∈ [1, 300] → Output
MobileNet [42]	Conv1/S2 ∈ [1, 32] → ConvDW1/S1 ∈ [1, 32] → ConvP1/S1 ∈ [1, 64] → ConvDW2/S2 ∈ [1, 64] → ConvP2/S1 ∈ [1, 128] → ConvDW3/S1 ∈ [1, 128] → ConvP3/S1 ∈ [1, 256] → AvgPool → Output
TCN [43]	ConvC1/S1/D1 ∈ [1, 36] → ConvC2/S1/D1 ∈ [1, 36] → ConvC3/S1/D2 ∈ [1, 36] → ConvC4/S1/D2 ∈ [1, 36] → AvgPool → Output

TABLE III: Mobile device specifications

Device	ROM	RAM	CPU	Battery	OS
OnePlus 6	128GB	8GB	Snapdragon 845	3300mAh	Android 8.1.0
Pixel 2 XL	64GB	4GB	Snapdragon 835	3520mAh	Android 8.1.0
Huawei Honor 8	32GB	4GB	HiSilicon Kirin 950	3000mAh	Android 8.0.0
Samsung Gear S3	4GB	768MB	Exynos 7 Dual 7270	380mAh	Tizen 4.0.0.4

we scale down the layers of the standard MobileNet and TCN to fit sensor data.

We conduct all evaluations in an indoor environment. For performance evaluation, the participating smartphones are placed in proximity with a range from 1m to 5m for any two shown in Fig. 6b. For evaluation in a real-world application, all participating users are located in a workplace with a size of 23m x 16m shown in Fig. 7. To evaluate battery consumption, we discharge all smartphones. Before each experiment, we charge the battery of smartphones full to ensure that each experiment is in the same initial battery condition. In addition, we keep training computational and communication costs identical (*i.e.*, the same system architecture) as fair comparison for all baselines.

For evaluation, we set up both IID and non-IID data distribution scenarios [15]. In IID, we randomly distribute each dataset equally among all the devices to fairly test the resource-efficiency performance of MDLdroid. In non-IID, we first separate each dataset by subject ID, and then randomly select a number of subject's dataset to match the size of devices (*i.e.*, each device keeping each subject's data), aiming to compare the training performance of MDLdroid with the baselines. For each scenario, we pre-load a given sub-dataset to each device in advance to simplify our evaluation.

Hyper-parameters We choose Adam [30] as the default SGD optimization with a fixed learning rate of 0.001, and set the batch size to 16. For all DL models, the parameters and noise are randomly initialized by a uniform distribution in $[-1, 1]$. We report top-1 accuracy throughout the evaluation.

B. Performance of Chain-scheduler

We first compare our TDGE approach with the baselines, *i.e.*, DGE and the threshold-based greedy-exploration (TGE) (*i.e.*, the exploration only relies on the given threshold without decaying) approaches (§III-C), and evaluate the performance of the exploration strategy when training Chain-scheduler. Secondly, we select two existing resource-agnostic schedulers (*i.e.*, *Tree-scheduler* [44] and *Ring-scheduler* [19]) in distributed DL frameworks as the baselines to evaluate the performance of Chain-scheduler. Thirdly, we evaluate the improvements of TDGE-v2 and Pre-TDGE-v2 (*i.e.*, using pre-trained RL model by TDGE-v2) compared to TDGE in terms of resource-efficiency and scalability on an edge server.

For fair benchmarking, we simulate resource dynamicity in reality for each device in MDLdroid. Specifically, we randomly allocate resources for each training device while assuring a maximum of 50% of the devices being in *busy* state. To evaluate our re-learning mechanism, we randomly modify the resource state of some devices being *busy* or *free* to emulate the conditions mentioned in Fig. 4b. We run each experiment 50 times and report the performance and resource usage presented in the following sections.

1) Exploration Strategy and Scheduling Performance:

To fully examine the performance of our TDGE approach and Chain-scheduler, we conduct three experiments which the results are summarized as follows: 1) our TDGE outperforms TGE and DGE in training time, and accelerates the process

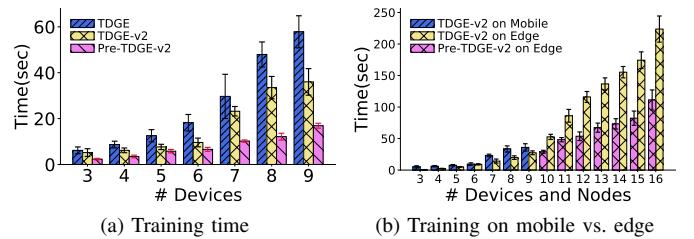


Fig. 8: Chain-scheduler training comparison using TDGE-v2.

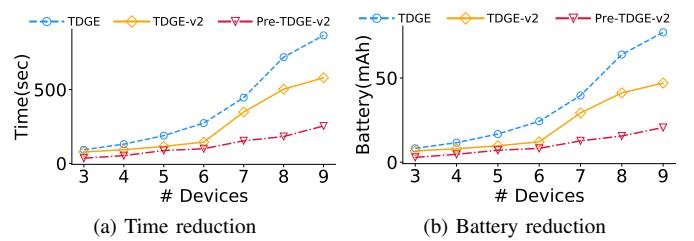


Fig. 9: Re-learning using TDGE-v2.

of Chain-scheduler training; 2) the TDGE outperforms DGE under an *intensive* re-learning scenario (*i.e.*, randomly 15 times re-learning during 20 Epoch training iterations) in resource-efficiency; 3) Chain-scheduler outperforms Tree-scheduler and Ring-scheduler, achieving the best trade-off between training time and energy variance. Please refer to our conference paper [26] for the detailed results about these experiments.

2) Efficient Exploration Strategy: In this experiment, we investigate whether applying TDGE-v2 to train Chain-scheduler performs a faster learning convergence than using TDGE when the network size increases. We also evaluate whether using Pre-TDGE-v2 to avoid learning from scratch can further reduce the learning time compared to both TDGE-v2 and TDGE. Fig. 8a clearly shows that both TDGE-v2 and Pre-TDGE-v2 outperform TDGE with less training time. With the network size increases, the training time of TDGE-v2 and Pre-TDGE-v2 is largely reduced by 1.5x and 2.9x on average over that of TDGE, respectively. Since we apply Pre-TDGE-v2 to avoid learning the environment from scratch, the training time is significantly further reduced to make Chain-scheduler much more efficient.

3) Performance in Resource-efficient Re-learning: In this experiment, we continue to evaluate the resource-efficiency (*i.e.*, training time and battery consumption) of different exploration strategies under the same *intensive* re-learning scenario. Fig. 9a indicates that both training time and battery consumption of TDGE-v2 and Pre-TDGE-v2 are notably less than that of TDGE. Besides, compared to TDGE-v2, Pre-TDGE-v2 saves both time and battery by 2x on average. The result also shows that the battery consumption of training Chain-scheduler is minor (*e.g.*, the maximum battery of TDGE with 9 devices is 77 mAh out of 3520 mAh), but the training time is long. Thus, applying Pre-TDGE-v2 to train Chain-scheduler is more practical with much less training time even under such *intensive* scenario.

4) Learning Scalability on Mobile vs. Edge: We now evaluate the learning scalability performance of Chain-scheduler running on edge server vs. device. Due to the availability of

smartphones in our lab, we report the testbed result with 9 smartphones, and the simulation result with 16 nodes. Fig. 8b demonstrates that running on the edge server achieves with less training time (*e.g.*, 2.6x less on average). When the network size increases (*e.g.*, larger than 9 nodes), the training time of TDGE-v2 on the edge shows a notably increase, and the time cost may easily beyond the limitation of running on device in practice. However, applying Pre-TDGE-v2 on the edge effectively reduces the training time by 2x on average. Thus, moving Chain-scheduler to the edge server can practically reduce the risk of the resource bottleneck on device when the network size goes large, and Pre-TDGE-v2 can essentially improve the performance of learning scalability.

C. Performance of MDLdroid

To give a comprehensive evaluation for the performance of MDLdroid, we first compare MDLdroid with the FL [13] (*i.e.*, a master-slave structure) in structure level from training accuracy and resource used perspectives. Besides, we choose a server-based approach to further ensure the training accuracy to be reliable. we next explore the optimized resource-accuracy trade-off options of MDLdroid.

1) **MDLdroid vs. FL:** We compare the performance of MDLdroid vs. FL from three perspectives, *e.g.*, peak-memory overhead (*i.e.*, initial memory footprint by loading libraries is excluded), training time, and network energy balance, and the summarized results present that MDLdroid achieves low-memory footprint, faster training, and better energy balance comparing to FL. Besides, since MDLdroid requires a training device to periodically report its resource condition via a compressed tiny BLE message (*i.e.*, the size is less than 20K), the actual battery consumption of reporting resources is much smaller than that of sending model parameters via BS or WD, *e.g.*, each device roughly drains 15 mAh out of 3600 mAh for the BLE messages by training PAMA2 with 20 Epoch in a network with 9 devices. Please refer to our conference paper [26] for the complete results of this experiment.

2) **Trade-off between Resource and Accuracy:** Both model aggregation frequency (*i.e.*, periodic aggregation [45]) and training iteration *Epoch* can significantly impact the balance between resource and accuracy. We evaluate the trade-off between resource and accuracy from three aspects, *e.g.*, training accuracy by a given threshold, battery consumption by maximum battery, and training time, using LeNet on all datasets in Table I. The summarized results show that the 1-E20 (*i.e.*, 1 model aggregation sync after each iteration—20 training iterations) as the optimized resource-accuracy option to achieve the best performance. Please refer to our conference paper [26] for detailed results.

D. Improvement in MDLdroid-v2

We design seven experiments as follows to fully evaluate the resource-efficiency improvements in MDLdroid-v2. We also compare the improved ChainSGD with GoSGD [34] (*i.e.*, a state-of-the-art decentralized A-SGD approach) in both IID and non-IID scenarios. In particular, GoSGD-B originally

uses a *Bernoulli* function to manage the model aggregation frequency, and GoSGD keeps using the same optimized frequency option (*i.e.*, 1-E20) as ChainSGD in this set of evaluation. In practice, GoSGD-B may have fewer model aggregations (*i.e.*, depending on the batch size) comparing to GoSGD by our settings. We then conduct a real-world case study to examine the end-to-end performance of MDLdroid-v2.

1) **Training Accuracy Guarantee:** As aforementioned, the training accuracy may present a notably drop when the network size increases. In this experiment, we evaluate whether ChainSGD gradient-scale can guarantee fast training convergence to keep the same accuracy level when network scales in the IID scenario. We train LeNet on all datasets in a network size up to 9 devices using 4 approaches, respectively. Since both GoSGD and GoSGD-B are A-SGD approach, the output models on different devices will be naturally different. For a fair comparison, we refer to [34] to finally process an averaged model, and evaluate on the same test dataset after each training. Fig. 10 clearly shows that the training accuracy on all datasets in MDLdroid-v2 keeps the same accuracy level without a major drop, *e.g.*, accuracy on each dataset is higher than the red dash line as the given threshold, while the other approaches have a notably accuracy drop when network scales. In particular, the accuracy on sEMG and OPPORTUNITY in a network with 9 devices dramatically increases by 41.16% and 69.54%, respectively, compared to MDLdroid-v1. The proposed ChainSGD gradient-scale approach therefore guarantees the training accuracy and improves the scalability of MDLdroid.

2) **Training in Non-IID:** In MDLdroid, multiple users' data will be more likely to be distributed as in a non-IID scenario in reality. In this experiment, we continually investigate whether MDLdroid-v2 can achieve a better training performance than the baselines in a non-IID scenario. We select three datasets with subject ID (*e.g.*, HAR, sEMG, and MHEALTH) in Table I using LeNet. We run each training 5 times in a network size ranged from 2 to 9 devices, and randomly select the number of subject's datasets in each training. We also test each device individually. Fig. 11 presents that MDLdroid-v2 achieves a higher training accuracy on average comparing to the baselines, especially sEMG. Taking a closer look at the training performance in a network with 9 devices, Fig. 12 shows that MDLdroid-v2 outperforms the baselines with a larger median of accuracy at 20-Epoch on all datasets, *e.g.*, 92.3%, 74.8%, and 87.9% on HAR, sEMG, and MHEALTH, respectively. We hence conclude that MDLdroid-v2 achieves a superior training performance on average over the baselines in non-IID.

3) **Out-of-sync Condition:** Since we design an efficient timeout mechanism (§III-B) to skip the delayed devices in network if in case, these out-of-sync devices may easily affect training accuracy due to missing the gradient parameter aggregation. In this experiment, we evaluate whether gradient-scale can keep the same training accuracy on the out-of-sync condition (*i.e.*, the worst condition which the selected devices are lost throughout the training). We train LeNet on HAR, and assume the worst case that the out-of-sync devices ranged

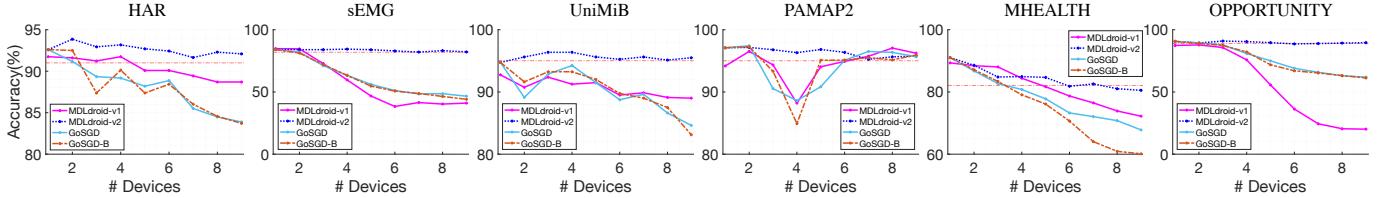


Fig. 10: Training accuracy preserving using ChainSGD-scaling by network scaling.

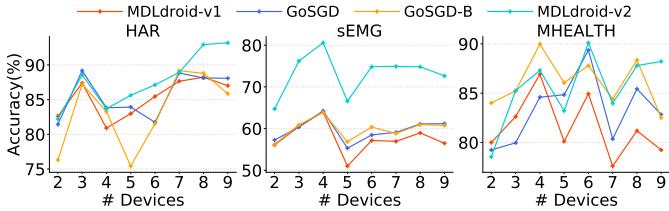


Fig. 11: Non-IID accuracy comparison by network scaling.

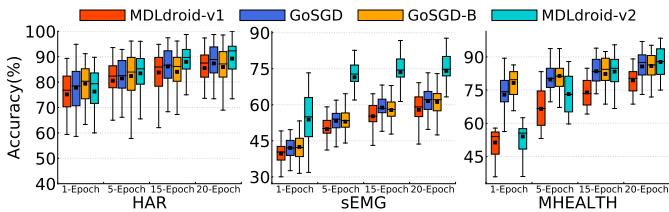


Fig. 12: Non-IID training performance comparison. (Black line in the box represents median; dot represents mean; box represents 25% and 75% percentiles.)

from 1 to 7 (*i.e.*, at least 2 devices alive) do not perform any pair aggregation throughout the entire training process. Based on this, Fig. 13a presents that the training accuracy of both v1 and v2 progressively decreases with the number of out-of-sync devices increases. However, v2 still outperforms v1 with a higher accuracy in each case. Especially, v2 remains the accuracy by 91.21% with 4 out of 7 devices missing their contributions (*i.e.*, 57% out-of-sync rate). As a result, the gradient-scale in v2 keeps the training accuracy with a minimal drop on the worse condition. In addition, since our system enables the re-connection mechanism to each device (§III-B), the impact of the out-of-sync condition can be practically mitigated if the devices are able to re-connect to continually sync with peers during the training.

4) Training Speedup: Since we improve the way of reading input data on device, we now evaluate the actual training speed improvement of MDLdroid-v2. Fig. 14a presents a

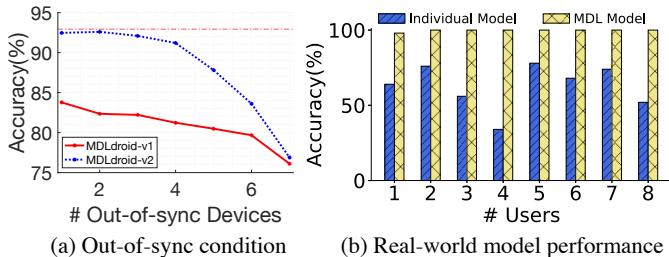


Fig. 13: Training accuracy performance in both out-of-sync and real-world use conditions.

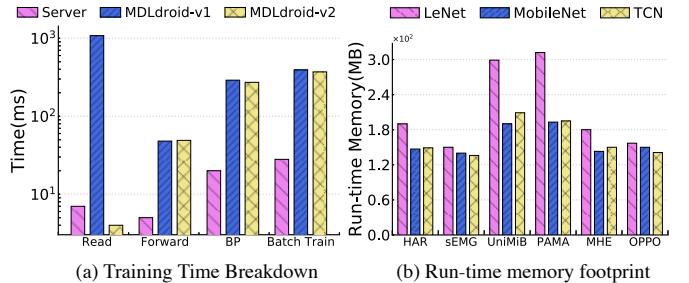


Fig. 14: Time reduction and run-time memory comparison.

training time breakdown for every single batch training using LeNet on HAR. Comparing to MDLdroid-v1, the reading time on average is significantly reduced by 270x, even it is slightly faster than reading on a server, *e.g.*, reading a batch with 16 rows of HAR data takes only 4ms on Pixel 2XL. Besides, both on-device Forward (*i.e.*, Forward Pass) and BP (*i.e.*, Backpropagation) remain the same time with MDLdroid-v1, but are slower by 9.6x and 13.6x, respectively, than on a server. The results shows that MDLdroid-v2 greatly speeds up training on device.

5) Communication Resource Reduction: We now evaluate the communication resource reduction (*i.e.*, time and battery) in MDLdroid-v2. In particular, we use three different DL models with a different size depending on datasets to fully evaluate the communication resource difference between v1 (*i.e.*, using BS) and v2. Both Fig. 15a and 15b show that v2 outperforms v1 with less communication time and battery on all DL models, *e.g.*, the time and battery of v2 using LeNet on HAR are reduced by 8.1x and 5.2x on average, respectively, over that of v1. Interestingly, the time of v2 using both MobileNet and TCN shows a minor reduction. This is because the model size of both DL models are much smaller than that of LeNet (*e.g.*, the size of MobileNet on HAR is 210K while LeNet is 3.5MB), the communication time hence has no major difference. However, the battery using both lightweight models in v2 are still largely reduced compared to that of v1. The result shows that applying WD in MDLdroid-v2 can practically reduce the communication resource, and using lightweight DL models on device can achieve less resource usage.

6) Memory Footprint of DL Models: Memory footprint on device is a critical resource index. We now evaluate the performance of run-time memory footprint (*i.e.*, including the initial memory footprint) based on three different DL models. Fig. 14b presents that the run-time memory footprint of LeNet on different datasets is notably larger than that of MobileNet and TCN, *e.g.*, the LeNet on PAMA2 is larger by 1.6x than

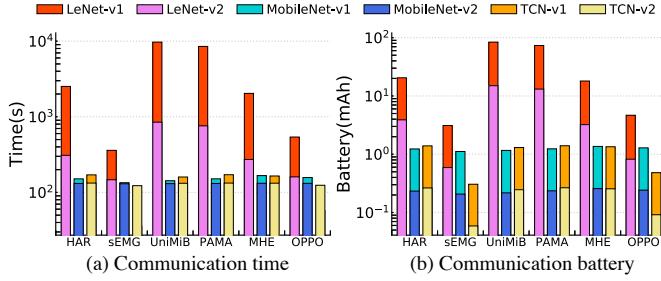


Fig. 15: Communication resource reduction comparison.

TABLE IV: Training Accuracy Comparison

Datasets	State-of-the-art	Server	FL-Best	MDL-Best	MDL-Trade-off	MDL-Best-v2
HAR	96% [38]	93.8%	92.7%	92.5%	90.0%	94.64
PAMAP2	90%+ [40]	97.6%	94.7%	95.2%	90.7%	99.71
MHEALTH	90% [36]	92.3%	91.0%	90.2%	85.4%	96.49
UniMiB	85% [38]	96.1%	93.3%	93.6%	91.5%	99.75
sEMG	88% [35]	89.2%	86.3%	85.8%	84.6%	91.74
OPPO	85% [39]	88.6%	87.5%	86.9%	84.7%	90.91

MobileNet on PAMA2. The result shows that using lightweight DL models on device can achieve a smaller run-time memory footprint.

7) **Resource-accuracy Performance in MDLdroid-v2:** In this experiment, we present a comprehensive result to highlight the improved resource-accuracy performance in MDLdroid-v2 in terms of training accuracy, battery consumption and training time using different DL models on all datasets in a network size up to 9 devices.

Training Accuracy Fig. 16 indicates that the training accuracy in v1 decreases with the network size, while v2 has a stable accuracy. Interestingly, the LeNet on HAR, UniMiB and PAMA2 of v2 in the multi-device condition even achieves a higher accuracy than training on a single smartphone. Overall, both MobileNet and TCN outperforms LeNet with a higher training accuracy. In conclusion, MDLdroid-v2 guarantees the training accuracy on device and achieves a higher accuracy when the network scales up.

Battery Consumption and Time Both Fig. 17 and Fig. 18 demonstrate that the overall battery consumption and training time are massively reduced in MDLdroid-v2 as the speed of both training and communication is notably increased. Specifically, compared to v1, the end-to-end training time in v2 using LeNet, MobileNet and TCN is significantly reduced by 6x, 5.5x and 4.4x on average, respectively. Also, the battery consumption of each smartphone using the three models is largely reduced by 3x, 3.6x and 2.9x on average, respectively. In particular, applying MobileNet on different datasets with different network size achieves less battery cost and time spent among all DL models on smartphone.

As a result, Table IV summarizes the comparison result of training accuracy, and MDLdroid achieves the best results. In particular, MDLdroid-v2 not only achieves higher accuracy over the baselines, but also significantly reduces end-to-end latency in terms of training task, model aggregation communication, and scheduling learning in practice.

8) **Real-world Performance of MDLdroid:** To evaluate the end-to-end performance of MDLdroid-v2 in reality, we conduct a real-world case study based on FindroidHR in a workplace (§IV). Since FindroidHR employs personal heart rate and motion sensing data, each user's data have to be kept on smartphone for privacy-preserving. Also, the data

distribution based on different users naturally represents as a non-IID scenario. In addition, due to the costly data collection, each user's data may not be sufficient to ensure the model performance.

In this study, we invite 8 subjects and assign a smartphone and a smartwatch shown in Table III to each of them, and Fig. 7 shows their specific locations. Besides, each of them is asked to collect and label a small amount of gesture data (*i.e.*, 36 per class) using the smartwatch, in which the specification is listed in Table I, then transmitting individual data to each smartphone for training. We employ MobileNet in Table II as the default DL model. We train both individual model (*i.e.*, training only with the small amount of individual data on device) and MDLdroid model via local collaborative learning. Fig. 13a demonstrates that they achieve an average accuracy of 62.75% due to insufficient individual data, while the MDLdroid model achieves a high accuracy of 99.75% on average. Besides, the actual resource cost for training the MDLdroid model is summarized as: the end-to-end time is 154.8s less than 3mins, the battery consumption of each smartphone on average is only around 1.18 mAh, and the run-time memory of training component on average is 187MB.

VI. DISCUSSION

Model Structure and Complexity MDLdroid aims for a universal on-device collaborative learning framework, hence practically it works for any DL models. Since different DL models result in a large difference of resource cost through our evaluation, the model complexity may strongly affect the actual resource usage on device. In practice, the latest large-sized models (*i.e.*, with heavy model parameters and large number of layers) can work for high accuracy, but will take a large amount of resources, *e.g.*, LeNet on HAR has 17x more model parameters and 24.5x more computational complexity than MobileNet on HAR, which performs notably less efficient for on-device training in our evaluation. Since resource efficiency is critical in the sensing domain, applying lightweight models or on-device model structure optimization [46] may work more efficiently to avoid overfitting. We plan to further optimize the model structure in our future work.

Efficient Compression Communication Applying the latest quantization or compression techniques for large-sized gradient parameters updates can significantly reduce communication latency and efficiently save resources in a network [47]. Since we essentially employ lightweight models for training resource efficiency, the existing compressed techniques may not work efficiently on lightweight models and will introduce more processing overhead, but can be applied to further deal with the scalability of MDLdroid.

Security Improvement Since security in a decentralized network is critical, we apply a number of improvements for model attack defense in MDLdroid (§III-B), *e.g.*, the model poisoning check, encrypted messages and models, aiming to offer a secure mobile collaborative DL framework without privacy considerations. Study [16] reports a security concern in FL that the personal data may be simulated by the latest generative adversarial learning techniques if the target model

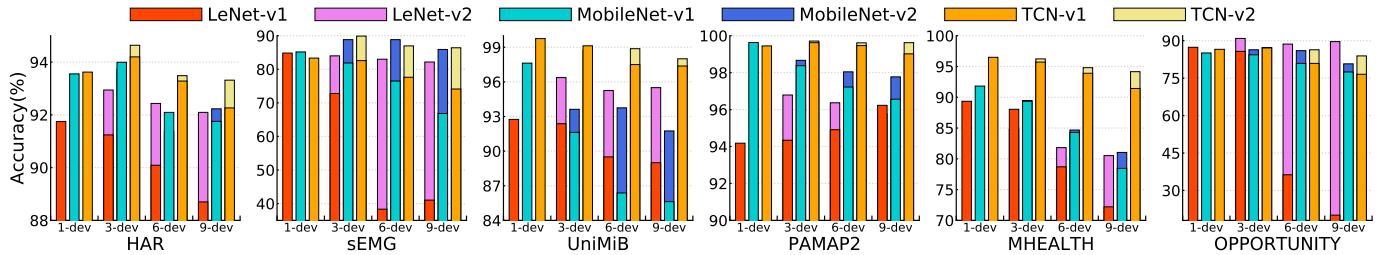


Fig. 16: Training accuracy based on different models.

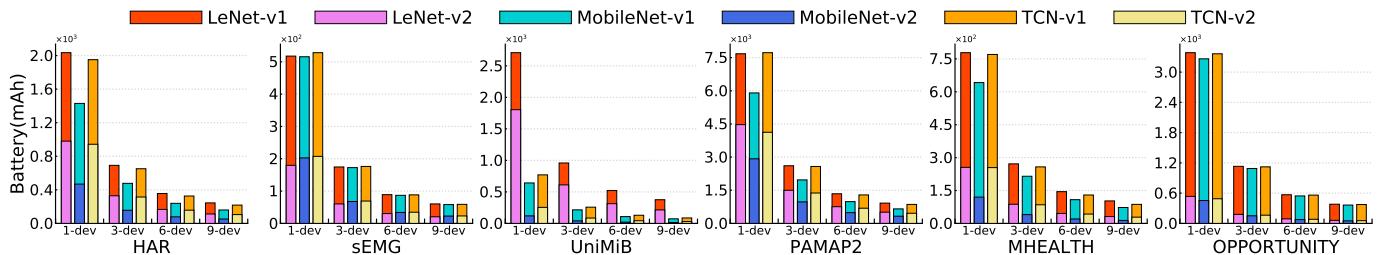


Fig. 17: Training battery consumption based on different models.

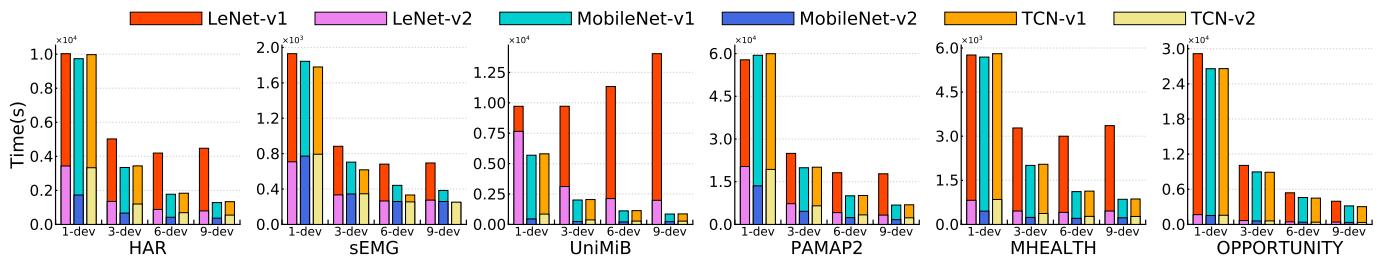


Fig. 18: Training time based on different models.

on server is leaking by malicious attacks. The leakage may be difficult to track if the central server cannot identify the deliberate attacker. In MDLdroid, as the peers are validated and exposed to each other based on a decentralized network, identifying the attacker may be more practical. We also plan to further improve the system security in our future work.

VII. RELATED WORK

Decentralized Deep Learning For decentralized framework, the existing work [29] proposes a theoretical model based on a *fixed directed* graph to offer a decentralized SGD algorithm to exchange model gradient parameters with its *one-hop* neighbors. However, if the relationship between the device and its *one-hop* neighbor is one-to-many, the device still suffers huge resource overhead which is similar to the master device case. On the other hand, as the underlying topology is a fixed graph, it cannot properly be performed in a real-time condition with resource dynamicity. By contrast, MDLdroid presents a dynamic *chain-directed* SGD algorithm based on a mesh network with a Chain-scheduler that enables a resource-aware model aggregation process to minimize training latency and reduce training resource overhead.

Resource-aware Mobile Deep Learning Most of existing works about resource-aware mobile DL mainly focus on

inference tasks. NestDNN [48] proposes a multi-tenant framework that can enable a resource-aware on-device to efficiently execute inference tasks for mobile vision applications. Besides, MCDNN [48] presents a framework that can execute multiple mobile vision applications based on cloud-based inference solution. In MDLdroid, we fully implement and execute both DL training and inference tasks on devices.

Resource-aware Task Scheduling The latest works [21] [23] propose to use a MARL based approach to solve task scheduling based on distributed network, which achieves fair performance. However, due to on-device resource limitation, the MARL implementation cannot well perform with training task on device. In contrast, MDLdroid applies a single agent-based DQN approach to deal with resource-aware task scheduling.

VIII. CONCLUSION

Towards pushing DL on devices, in this paper, we present MDLdroid, a novel decentralized mobile DL framework to enable resource-aware on-device collaborative learning for personal mobile sensing applications. MDLdroid achieves a reliable state-of-the-art model training accuracy on multiple off-the-shelf mobile devices. The key advantages of MDLdroid include on-device mobile DL, high training accuracy,

low resource overhead, low latency for model inference and update, and fair scalability.

ACKNOWLEDGMENT

This work is supported by Australian Research Council (ARC) Discovery Project grants DP180103932 and DP190101888.

REFERENCES

- [1] E. L. Murnane *et al.*, “Mobile manifestations of alertness: Connecting biological rhythms with patterns of smartphone app use,” in *MobileHCI ’16*, 2016.
- [2] J. V. Jeyakumar *et al.*, “Sensehar: A robust virtual activity sensor for smartphones and wearables,” in *Sensys ’19*, 2019.
- [3] R. Miotti *et al.*, “Deep learning for healthcare: review, opportunities and challenges,” *Briefings in bioinformatics*, 2017.
- [4] J. Wang *et al.*, “Deep learning towards mobile applications,” in *ICDCS’18*, 2018.
- [5] N. D. Lane and P. Georgiev, “Can deep learning revolutionize mobile sensing?” in *HotMobile ’15*, 2015.
- [6] Y. Tu *et al.*, “Network-aware optimization of distributed learning for fog computing,” ser. INFOCOM’20, 2020.
- [7] E. Li, Z. Zhou, and X. Chen, “Edge intelligence: On-demand deep learning model co-inference with device-edge synergy,” in *MECOMM’18*, 2018.
- [8] J. L. Kröger, P. R., and T. R. B., “Privacy implications of accelerometer data: A review of possible inferences,” in *ICCP’19*, 2019.
- [9] Y. Zhang *et al.*, “Findroidhr: Smartwatch gesture input with optical heartrate monitor,” *IMWUT ’18*, 2018.
- [10] S. Dhar *et al.*, “On-device machine learning: An algorithms and learning theory perspective,” 2020.
- [11] L. Zhang, “Transfer adaptation learning: A decade survey,” *CoRR*, 2019.
- [12] W.-Y. Chen *et al.*, “A closer look at few-shot classification,” in *ICLR’19*, 2019.
- [13] K. A. Bonawitz *et al.*, “Towards federated learning at scale: System design,” in *SysML ’19*, 2019.
- [14] D. Zhang *et al.*, “A survey on collaborative deep learning and privacy-preserving,” in *DSC’18*, 2018.
- [15] W. Y. B. Lim *et al.*, “Federated learning in mobile edge networks: A comprehensive survey,” *IEEE Communications Surveys & Tutorials*, 2020.
- [16] Z. Wang *et al.*, “Beyond inferring class representatives: User-level privacy leakage from federated learning,” ser. INFOCOM’19, 2019.
- [17] A. N. Bhagoji *et al.*, “Analyzing federated learning through an adversarial lens,” in *ICML’19*, 2019.
- [18] X. Lian *et al.*, “Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent,” in *NIPS’17*, 2017.
- [19] M. Li *et al.*, “Scaling distributed machine learning with the parameter server,” in *OSDI’14*, 2014.
- [20] A. Aral, M. Erol-Kantarci, and I. Brandić, “Staleness control for edge data analytics,” *Proc. ACM Meas. Anal. Comput. Syst.*, 2020.
- [21] D. ben noureddine, A. Gharbi, and S. Ahmed, “Multi-agent deep reinforcement learning for task allocation in dynamic environment,” in *ICSOFT’17*, 2017.
- [22] J. Yang, H. Xu, and P. Jia, “Task scheduling for heterogeneous computing based on bayesian optimization algorithm,” in *CIS’09*, 2009.
- [23] T. T. Nguyen, N. D. Nguyen, and S. Nahavandi, “Deep reinforcement learning for multi-agent systems: A review of challenges, solutions and applications,” *CoRR*, 2018.
- [24] P. H. Jin *et al.*, “How to scale distributed deep learning?” *CoRR*, 2016.
- [25] K. Persand, A. Anderson, and D. Gregg, “Composition of saliency metrics for channel pruning with a myopic oracle,” 2020.
- [26] Y. Zhang, T. Gu, and X. Zhang, “Mdldroid: a chainsgd-reduce approach to mobile deep learning for personal mobile sensing,” in *IPSN ’20*, 2020.
- [27] S. Gupta, W. Zhang, and F. Wang, “Model accuracy and runtime tradeoff in distributed deep learning: A systematic study,” in *ICDM’16*, 2016.
- [28] K. Yu *et al.*, “Layered SGD: A decentralized and synchronous SGD algorithm for scalable deep neural network training,” *CoRR*, 2019.
- [29] Z. Jiang *et al.*, “Collaborative deep learning in fixed topology networks,” in *NIPS’17*, 2017.
- [30] S. Ruder, “An overview of gradient descent optimization algorithms,” *CoRR*, 2016.
- [31] T. Hoefer, C. Siebert, and W. Rehm, “A practically constant-time mpi broadcast algorithm for large-scale infiniband clusters with multicast,” in *IPDPS’07*, 2007.
- [32] V. Mnih *et al.*, “Human-level control through deep reinforcement learning,” *Nature*, 2015.
- [33] M. Tokic, “Adaptive ϵ -greedy exploration in reinforcement learning based on value differences,” in *KI 2010: Advances in Artificial Intelligence*, 2010.
- [34] M. Blot *et al.*, “Gossip training for deep learning,” ser. NIPS’16, 2016.
- [35] S. Lobov *et al.*, “Latent factors limiting the performance of semg-interfaces,” *Sensors*, 2018.
- [36] O. Banos *et al.*, “mhealthdroid: A novel framework for agile development of mobile health applications,” in *Ambient Assisted Living and Daily Activities*, 2014.
- [37] D. Micucci, M. Mobilio, and P. Napoletano, “Unimib shar: a new dataset for human activity recognition using acceleration data from smartphones,” *CoRR*, 2017.
- [38] D. Anguita *et al.*, “A public domain dataset for human activity recognition using smartphones,” in *ESANN’13*, 2013.
- [39] D. Roggen *et al.*, “Collecting complex activity datasets in highly rich networked sensor environments,” in *INSS’10*, 2010.
- [40] A. Reiss and D. Stricker, “Introducing a new benchmarked dataset for activity monitoring,” in *ISWC’12*, 2012.
- [41] Y. Lecun *et al.*, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, 1998.
- [42] A. G. Howard *et al.*, “Mobilens: Efficient convolutional neural networks for mobile vision applications,” *CoRR*, 2017.
- [43] S. Bai, J. Z. Kolter, and V. Koltun, “An empirical evaluation of generic convolutional and recurrent networks for sequence modeling,” *CoRR*, 2018.
- [44] S. Shi *et al.*, “A distributed synchronous SGD algorithm with global top-k sparsification for low bandwidth networks,” *CoRR*, 2019.
- [45] J. Wang and G. Joshi, “Adaptive communication strategies to achieve the best error-runtime trade-off in local-update sgd,” ser. MLSys’19, 2019.
- [46] Y. Zhang, T. Gu, and X. Zhang, “Mdldroidlite: a release-and-inhibit control approach to resource-efficient deep neural networks on mobile devices,” in *Sensys’20*, 2020.
- [47] A. Koloskova, S. U. Stich, and M. Jaggi, “Decentralized stochastic optimization and gossip algorithms with compressed communication,” ser. ICML’19, 2019.
- [48] B. Fang, X. Zeng, and M. Zhang, “Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision,” in *MobiCom’18*, 2018.



Yu Zhang received the B.S. degree in Software Engineering from Southwest University for Nationalities, China. He is currently pursuing the Ph.D. degree in computer science at RMIT University, Australia. His research interests include mobile computing, on-device machine learning, wireless sensor network, embedded systems, Internet of Things and big data analytics.



Tao Gu is currently a Professor in Department of Computing at Macquarie University, Australia. His research interests include Internet of Things, ubiquitous computing, mobile computing, embedded AI, wireless sensor networks, and big data analytics. He is currently serving as an Editor of IMWUT, an Associate Editor of TMC and IoT-J. Please find out more information at <https://taogu.site>.



Xi Zhang received the B.S. degree from the Beijing Jiaotong University Haibin College of Computer Science, China in 2014, the M.S. degree from Monash University of Information Technology, Australia in 2018, and He is currently pursuing the Ph.D. degree of Computer Science in RMIT University, Australia. His research interests include Internet of Things, mobile computing and machine learning.