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

Yu Zhang
RMIT University
zac.lhjzyzzoo@gmail.com

Tao Gu
RMIT University
tao.gu@rmit.edu.au

Xi Zhang
RMIT University
zaibuer@gmail.com

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 multiple user data. 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 constraint and serious failure by attacks. 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 than the master-slave approach.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; **Distributed computing methodologies**; *Reinforcement learning*; *Mobile agents*; • **Networks** → **Network resources allocation**; *Network protocol design.*

KEYWORDS

Mobile deep learning, Neural networks, Distribute computing, Resource allocation, Reinforcement learning

1 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 [31], recognizing activities in ambient assisted living areas [17], and monitoring personal health in smart home or hospital [32]. 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-modality sensing data [40]. It is advocated that DL will be the next key enabler for advanced personal mobile sensing [24].

Personal Sensing Requirements 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 transmit the data from local to cloud, offloading the heavy training workloads to the cloud. Edge server can also be used to efficiently offload the workloads [27]. However, real-world personal sensing applications pose several requirements in which the server-based approaches may not fit. Firstly, personal sensing data are significantly privacy-sensitive as the data contain a variety of human motion and biological contexts. Studies [39] [21] show that leaking motion data can cause a series of violations concerning user location, health condition, emotional state, and identification information. Even governments across the worlds are reinforcing laws to protect personal data privacy, for example the General Data Protection Regulation (GDPR) [44]. Server-based approaches have serious *privacy* concerns. Secondly, due to dynamic external nature effects and internal characteristics of each individual, personal sensing data are strongly affected by specific personal activities, social abilities and surrounding environment conditions over time [25] [17], and features may be updated at any time. Simply deploying a pre-trained global model on device may not *continually* work well to adapt the *local* features which have been changed. Continually training a local model with new data is a fundamental requirement, and applications usually require low-latency responses for both model inference and update. In practice, continually transmitting sensing data to the server and downloading model updates for *training* can incur fast battery drain and considerable latency for mobile devices especially when the network connection is unstable or broken. Different from server-based approaches, mobile DL may be an ideal solution to effectively preserve sensing data privacy without being transmitted over the public network, and enable quick local model inference and update response for continuous on-device training [11] [40]. Thus, mobile DL is a promising direction for personal sensing applications.

Mobile Deep Learning Challenges Existing work [40] reveal that mobile DL poses two challenges. Firstly, pushing DL with both model training and inference to a resource-constraint mobile device in practice is extremely constricted by its capability. Secondly, data collection with various conditions is usually costly in reality, and the model performance and robustness can be seriously affected due to insufficient data. Collaborative deep learning is proposed to ensure

model training efficiency and robustness based on multiple user data [46]. Following this direction, Google's Federated Learning [19] has been proposed to provide a mobile collaborative DL framework. This framework not only preserves data privacy by transferring the gradient parameters of a DL model without exposing local data, but also improves the global model performance and robustness by a large number of user models. However, Federated Learning relies on a central server for intensive global model aggregations, which may not work well for personal sensing scenarios due to the low-latency requirement for continuous local model update. Scaling down the framework to a central edge server [41] or even a master device may efficiently reduce the latency. However, Federated Learning deploys a master-slave structure which is less attack-resistant [48]. A recent study [6] demonstrates that the model poisoning attacks can cause a serious negative impact to the central training process. In addition, the central edge server or master device may quickly become the bottleneck due to substantial communication and intensive model aggregation, resulting in huge resource overhead [23]. Moreover, the system fault-tolerant may be limited in practice if the central node goes down.

In contrast, applying a decentralized structure for mobile DL can theoretically offer high attack-resistant and reliable fault-tolerant [48]. Existing approaches propose a theoretical decentralized structure based on a *fixed directed* graph network such as Ring-Allreduce [28]. 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 low level of resources available. Furthermore, studies [15] [17] report that using opportunistic user context from *multiple* people can enrich the sensing data distribution to make model robust to the *local* variations. Therefore, moving towards a *local resource-aware decentralized collaborative* mobile DL approach without central server support for personal sensing applications is strongly motivated in order to mitigate resource overhead and reduce latency for training.

Decentralized Mobile Scheduling Deploying a decentralized DL framework on multiple devices can be extremely difficult due to the design of resource-efficiency task scheduling for resource-constrained devices. A generic optimization algorithm [42] can be applied to perform task scheduling. However, the recent *Multi-agents Reinforcement Learning* (MARL) approach [33] may work better in such a *non-stationarity* and multi-device scenario. Agent-based approaches have a significant advantage of estimating the future scheduling order because agents inherently learn features from the past experiences based on Reinforcement Learning (RL), while the generic optimization approaches cannot estimate a future possible state and will be limited to control a complex environment due to lacking of such learning mechanism [5]. Thus, using agent-based approaches can theoretically make scheduling more reliable for complex environments.

Our Approach Aiming to push DL to mobile devices, we present MDLdroid, a novel decentralized **M**obile **D**eep **L**earning framework to enable resource-aware on-device collaborative learning for personal mobile sensing applications. MDLdroid targets to fully operate on multiple off-the-shelf An**droid** smartphones connected in a mesh network, and achieve high training accuracy and reliable execution of the state-of-the-art DL models.

Our challenge is two-fold. To address the first 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 [9] 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 communication and memory) of resource overhead for each device. Different from the existing Ring-Allreduce approach [28], each pair of devices is dynamically scheduled to perform model aggregation based on their resource condition for latency reduction. In practice, we leverage on a mesh graph to aggregate multiple descendant model parameters into global model parameters to complete each training iteration. Once the *Epoch* reaches the given iterations, the entire training process is completed. The global model generated can then be deployed on each device for inference.

Secondly, designing a resource-efficiency task scheduler in a decentralized DL framework is challenging because resources on device are constrained strictly and changed dynamically (i.e., the *Non-stationarity* problem). A device may be dynamically switched to *busy* state due to other high priority tasks, leading to training being suspended. In addition, the model aggregation task needs to be distributed to multiple devices in a fair manner to battery consumption across the network. To address this challenge, we propose a single agent-based scheduler based on Reinforcement Learning to self organize scheduling tasks using dynamic information of on-device resource, named *Chain-scheduler*. Chain-scheduler includes an effective reward function design to map each perceived scheduling action to a reward, aiming to optimize the given constraints. Through our analysis in Section (§3.3), Chain-scheduler can achieve the best latency-reduced scheduling close to *Tree-scheduler* in the Tree-Allreduce approach [37], and it can also achieve the best energy balance close to *Ring-scheduler* in the Ring-Allreduce approach [28]. To tackle the *Non-stationarity* problem, we apply a continuous environment learning strategy to repeat learning if the scheduling environment is changed. In this way, scheduling can be more adaptive to on-device resource changes. In addition, we further enhance the performance of the reward function by designing a Threshold-based Decaying Greedy-Exploration (TDGE) strategy to save on-device resources used for scheduling tasks.

We fully implement MDLdroid on off-the-shelf smartphones based on modified DL4J libraries. To evaluate MDLdroid, we use two standard Convolutional Neural Network (CNN) models (LeNet [26] and AlexNet [20]) since CNN can be used to effectively process multi-channel sensing data [43]. We evaluate the training performance using 6 public datasets in Table 1, containing diverse personal mobile sensing data. Results show that MDLdroid achieves high training accuracy which is comparable to the state-of-the-art accuracy reported in 3, speeds up training by 2x to 3.5x compared to the single-device baseline and 1.5x compared to Federated Learning. In addition, MDLdroid reduces latency overhead due to busy condition by 23% and 53% compared to *Tree-scheduler* and *Ring-scheduler*,

respectively. Moreover, MDLdroid reduces the variance in battery consumption among devices by 40% compared to *Tree-scheduler*.

Summary of Contributions Key contributions of this paper are summarized as follows:

- We present MDLdroid, a novel decentralized mobile DL framework based on a mesh network to enable resource-aware on-device collaborative learning for personal mobile sensing applications (§3).
- We propose a ChainSGD-reduce approach, in particular a *chain-directed* S-SGD algorithm, to minimize the resource overhead of model aggregation tasks in a decentralized framework (§3.2).
- 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 (§3.3).
- We evaluate MDLdroid on off-the-shelf Android smartphones with two standard DL models using 6 public personal mobile sensing datasets (§4). Results indicate that MDLdroid accelerates training effectively, outperforming the state-of-the-arts.

To the best of our knowledge, this is the first time that collaborative DL is fully implemented and functioned on off-the-shelf mobile devices with both model training and inference without central server support. MDLdroid takes mobile DL to the next step, and opens up new possibilities for building secure and adaptive mobile sensing applications.

Implication MDLdroid defines two implication models. The *individual* model is used for an individual who has sufficient personal data and multiple mobile devices to offer shared resources. The personal sensing data can only be safely distributed to the given mobile devices verified by the same identity (e.g., Google account). MDLdroid can accelerate on-device learning, and alleviate the resource burden from a single device.

The *non-individual* model is mainly applied for a group of people to explore specific local sensing features. The data will be strictly kept on device to preserve data privacy, and only the model gradient parameters of each individual will be exchanged to improve model robustness. With the nature benefits of a decentralized approach, model poisoning check [6] can be easily deployed in each device to prevent attacks. Even if a few of devices are down due to attacks, MDLdroid can isolate them and keep others working safely to avoid catastrophic failure. Moreover, MDLdroid can be potentially used in many multi-user sensing scenarios [4], such as specific family behaviour recognition in a smart home, patient-specific health condition monitoring in a hospital, and healthcare tracking for older adults in different aged care facilities.

2 MOTIVATION

To investigate the limitations of deploying mobile DL on device in current solutions, we conduct preliminary studies to discover the bottleneck of on-device training.

Setup We select three well-known public datasets with different scales, i.e., sEMG (25MB) [22], HAR (160MB) [1], and OPPO (500MB) [1] in Table 1. We use an off-the-shelf Android smartphone, i.e., Google Pixel 2XL running Android 8.1. In each experiment, the smartphone runs complete training with a dataset using DL4J libraries [10].
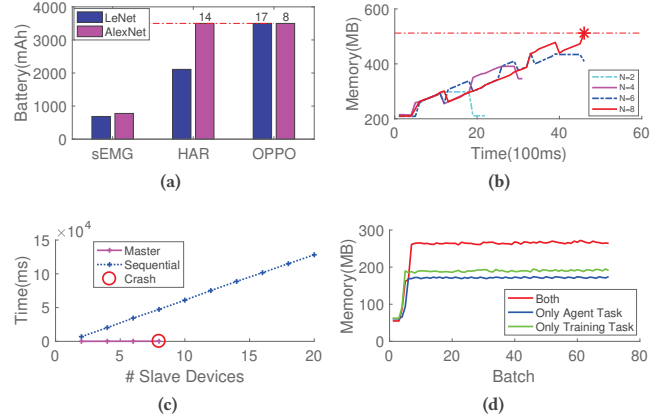

Figure 1: Preliminary results. (a) Battery consumption on single-device training; (b) Memory crash on CNN-LSTM; (c) Time cost of Master vs. Sequential model aggregation; (d) Memory conflict when running multi-agents.

Bottleneck of Single-device Training The encouraging result is that single-device training with LeNet for both sEMG and HAR achieves a fair accuracy with no failure. As we raise the memory threshold, the run-time memory usage of training is moderately increased with no major issues observed. However, from Figure 1a, we observe that battery overhead dramatically rises with the growth of data size and model complexity. In particular, with a maximum battery capacity of 3500 mAh on the smartphone, the training of HAR with AlexNet and the training of OPPO with both LeNet and AlexNet fail due to massive battery drain. Thus, single-device training on off-the-shelf smartphone is not currently feasible.

Memory Overhead of Master Model Aggregation To have quantitative measurements, we implement Federated Learning [7] on device without central server support, and run the master model aggregation process (i.e., memory usage is $O(N)$) on a master device to evaluate its resource utilization. Each experiment runs on the same smartphone receiving N copies of model gradient parameters, where N denotes the number of other slave devices. We select PAMA2 dataset from Table 1 to train a CNN-LSTM hybrid model [47] (i.e., 46.4MB model file and 11.61M model parameters). Figure 1b illustrates its memory usage since memory is more critical than battery due to model aggregation. We observe that memory usage is rapidly increased with N. When $N = 8$, it ends up with *OutOfMemory* (i.e., maximum 512MB). Therefore, multi-device training in a centralized framework has severe memory limitations for model aggregation.

Cost of Sequential Model Aggregation To solve the memory limitations, a naive sequential model aggregation approach (i.e., memory usage is $O(2)$) is implemented by sequentially loading model files into memory on the master device. We use the same CNN-LSTM hybrid model to evaluate the time cost. We simulate N is up to 20 to increase the scalability of model aggregation. Figure 1c indicates the time usage of sequential aggregation is way higher than using master aggregation (e.g., 47307ms vs. 160ms respectively when $N = 8$). Besides, the time usage of sequential aggregation
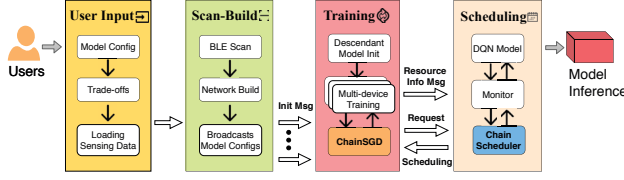
Figure 2: MDLdroid Architecture consists of three-stage processes: 1) user model configuration input; 2) network scan & build; 3) model training & task scheduling.

presents a linear increase with N, which is also critical in practice. Moreover, since two copies of model files are aggregated in memory space for each loading, the NeighborSGD in Eq. (2) is used in this implementation. However, a severe accuracy degradation issue is revealed as shown in Figures 3a and 3a (§3.2.2). Thus, although sequential model aggregation simply minimizes the memory usage, it can still cause considerable extra training time cost and accuracy degradation.

Resource Conflict in MARL Our preliminary experiment reveals that severe resource conflict exists in MARL. We implement a simplified MARL [5] on the same smartphone using DL4J libraries. Figure 1d indicates that running training and agent tasks concurrently on-device can dominate the main memory of the device, resulting in 0.6x higher than that of running any single task. We thus turn our intention to a single agent-based RL scheduling approach, which motivates our proposal.

3 MDLDROID FRAMEWORK

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

3.1 System Architecture

We first give an architecture of MDLdroid in Figure 2. Since MDLdroid is designed to operate full-scale DL on Android based on a mesh network, we employ both Bluetooth Low Energy (BLE) and Bluetooth Socket (BS) to build the mesh network due to accessibility and low energy consumption. In principle, any on-device mesh-based protocol can be applied (§3.2).

In the **first** stage, users input different model configurations based on their demand to train models. MDLdroid uses two combined network topology in the **second** stage. The BS-based mesh topology is applied to perform the decentralized model aggregation between devices in the training stage, while the BLE-based tree topology is used for the centralized resource condition monitoring in the task scheduling stage. In the **third** stage, each device is required to continually report resource condition to a mobile agent (MA). Once all training tasks request the model aggregation for each iteration, the Chain-scheduler on the MA can manage the resource-efficiency scheduling paths as a *chain-directed* graph. When model aggregation of each iteration are completed, a copy of the aggregated model parameters will be resource-aware broadcasted (§3.2.3) to all devices for the next iteration. **Finally**, once

training is completed, a global model will be distributed to all devices via broadcasting for model inference Especially, MDLdroid can also reload a pre-trained model to continually train with new local sensing data.

3.2 ChainSGD-reduce Approach

Collaborative learning relies on the distributed *stochastic gradient descent* (SGD) algorithms [19]. 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 a one-to-many broadcasts [13]:

$$\Delta w^{(k)}(t) = \frac{1}{N} \sum_{i=1}^{N} \Delta w_i^k$$
$$w^{(k)}(t+1) = w^k(t) - \eta \Delta w^k(t) \tag{1}$$

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

3.2.1 Asynchronous SGD vs. Synchronous SGD.

The distributed SGD algorithms are mainly classified into two categories—Asynchronous SGD (A-SGD) and S-SGD [9]. A-SGD can be more communication efficient and runs with no strong dependency among machines. However, study [45] points out that A-SGD suffers from an uncertain training accuracy degradation issue due to its delay model updating mechanism. S-SGD representing in Eq. (1), on the other hand, runs quite stable without this issue, but the drawback of S-SGD is that some machines may be slowed down in run-time due to dynamic low resources, and hence the overall training time depends on the slowest machine. By given limited 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.

3.2.2 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 [18] 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, $V = \{1, 2, ..., N\}$ 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.

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

However, although the model aggregation is divided by j descendant neighbor aggregations, the single device j still requires to
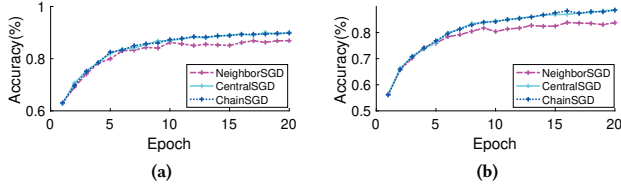
Figure 3: Training accuracy degradation and restore when the number of device increases (a) $N = 6$; (b) $N = 9$.

Figure 4: Model aggregation structure comparison: CentralSGD vs. NeighborSGD vs. ChainSGD

concurrently perform m model aggregations, which may cause significant memory overhead revealed by 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* in memory and communication 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 efficiency way.

Compared with centralized graph and decentralized neighbor graph, Figure 4 demonstrates that the major differences of ChainSGD-reduce are twofold: 1) the model aggregation is managed only with one of neighbors at a time; 2) the order of the aggregation tasks is dynamically scheduled depending on the real-time resource condition of device.

When ChainSGD-reduce is applied to NeighborSGD in Eq. 2, we reveal an accuracy degradation issue. Figure 3a shows that the training accuracy of NeighborSGD is lower than that of CentralSGD by 3% ($N = 6$). Besides, Figure 3b shows that with N increases, the training accuracy gap between NeighborSGD and CentralSGD becomes larger by roughly 5% less ($N = 9$). It is because that the global gradient parameters $\Delta w^{(k)}(t)$ decompose through the model aggregation process. We thus redefine ChainSGD as the following *pair aggregation* function $W(j, i)$ aiming to restore training accuracy.

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

where $W(j, i)$ represents the aggregated gradient parameters sending from a remote device j to device i as a *pair aggregation*. The global gradient parameters of each iteration are denoted as $\Delta w^{(k)}(t) = (N-1)W(j, i)$. Since ChainSGD offers a dynamic model aggregation process, multiple *pair aggregation* can be performed simultaneously based on resource condition of devices within one round to reduce latency showing in Figure 4. Especially, θ is a reversal parameter to enable that the gradient parameters of $W(j, i)$ can be restored close to CentralSGD, and θ starts from 1. Once a *pair aggregation* is done, the local θ_i will be updated with a remote θ_j as θ_i' for sending to next round. Our approach requires that the BS message for model aggregation contains Δw^k and θ as $(\Delta w^k, \theta)$. Both Figures 3a and
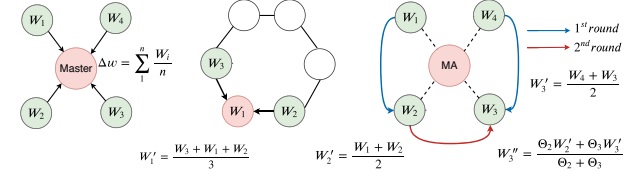
3a show that the performance of ChainSGD is consistent with that of CentralSGD with no accuracy degradation.

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

ALGORITHM 1: ChainSGD-reduce Algorithm

Initialize parameters w_0, number of iteration t, number of devices N, training dataset D_{train};
for *All devices $i \in N$* **do**
    for *t Epochs* **do**
        Train model on local $d_{train}^t \in D_{train}$;
        while *true* **do**
            if *Get an incoming BS message $(W(j, i), \theta)$* **then**
                Pair aggregation Δw_i^k in Eq. (3);
                Update θ;
            end
            else
                Get a remote neighbor j from MA;
                Send $(\Delta w_i^k, \theta)$ to neighbor j, Break;
            end
        end
        Update local $w_i^{(k)}(t+1) = w_i^k(t) - \eta \Delta w^k(t)$;
    end
end

3.2.3 *Resource-aware Broadcasting*.

In ChainSGD-reduce, the MA device handles the global model parameters broadcasting after each training iteration. If the MA device sequentially broadcasts by BS messages, it may suffer from heavy communication overhead when the number of devices N is large. We employ a binomial-tree broadcast algorithm [14] with a message forwarding function to address this issue. Basically, our implementation enables the broadcast list sorted by devices' resource condition, and the devices with more resources will be assigned with a list of forwarding tasks to mitigate the overhead on MA. Thus, the number of broadcast round by MA can be reduced from N to $\lceil \log_2 N \rceil$.

3.2.4 *Fault-tolerant*.

MDLdroid 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 preformed. Then MA 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.

3.3 Resource-aware Chain-scheduler

We propose Chain-scheduler using an agent-based RL model to dynamically schedule model aggregation tasks. To mitigate the memory overhead analyzed in our preliminary experiment (§2), we design a single agent-based RL mechanism. 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. In MDLdroid, two optimal goals are defined for Chain-scheduler to achieve resource-efficiency. One is to reduce training latency due to the resource condition of some devices dynamically turning *busy* in real-time. Another is to balance communication overhead and battery consumption across the network.

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

$$\arg\min N(T(t)) + N(E(t))$$
$$\text{s.t. } M \le M_{max}, \ B \le B_{max} \tag{4}$$

where T and E denote *training time* and *energy balance* 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 represents the current training iteration. We denote M_{max} and B_{max} as the maximum memory and maximum battery offered by the target device, respectively.

Training Time T The training time for each iteration t is defined as:

$$T(t) = T_{tr} + f(T_a, T_b) \tag{5}$$

where T_{tr} denotes the training time of each individual device, T_a denotes the overall model aggregation time for iteration t, and T_b represents the overall latency time caused by the *busy* state of devices. $f(x)$ is the schedule function. Especially, due to concurrent processes, $f(x)$ can schedule T_a and T_b to reduce iteration latency. The worse case is $T_a + T_b$ in which the scheduled sequence is linear and no concurrent overlap, while the ideal case is $\max(T_a, T_b)$ in which training latency is minimized as the concurrent overlap.

Energy Balance E The energy balance for each iteration t can be modeled as:

$$E(t) = \frac{1}{N} \sum_{i=1}^{N} (E_c^i - \mu)^2 \tag{6}$$

where μ is the mean of $\{E_1, E_2...E_N\}$, and E_c^i denotes the energy consumption of device i. We employ a variance function to represent the balance performance of Chain-scheduler.

The best estimated scheduling sequence represents as a reverse tree structure of *Tree-scheduler* [37] to reduce latency, which requires $\lceil \log_2 N \rceil$ communication rounds. While the worse case represents as a ring structure of *Ring-scheduler* [28] with N communication rounds. However, *Ring-scheduler* can perform the best
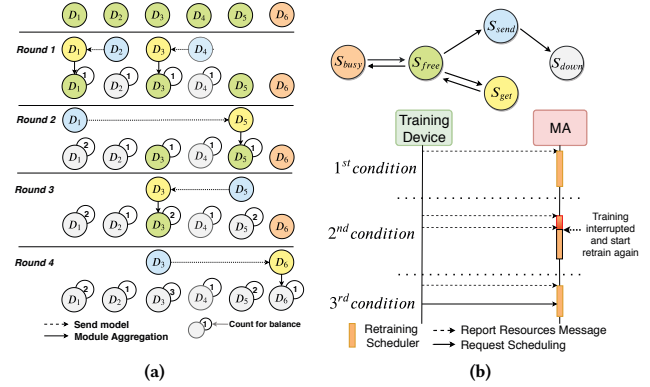

Figure 5: (a) The process of scheduling by busy condition; (b) State diagram and Re-training mechanism

energy balance as each device only requires to aggregate model once. Thus, Chain-scheduler is aiming to achieve optimal trade-off between *Tree-scheduler* and *Ring-scheduler*.

3.3.1 *Multi-goal Reward Function*.

We employ a DQN model [33] in MA to learn the scheduling environment based on the optimization function in Eq. (4). 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. $s = \{s_t, i = 1, 2, ..., T\}$ 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. Figure 5b illustrates the relationship of these states.
- **Action**: We define $a = \{a_i, i = 1, 2, ..., N\}$, where N denotes the number of devices. It represents which device is selected by the scheduler to act in the environment.
- **Reward**: $r = \{r_t, t = 1, 2, ..., T\}$ is defined by the reward function $r(s_t, a, s_{t+1})$ in Eq. (7).

Reward Model To solve the optimization function in Eq. (4), we design a three-stage mechanism as follows to offer a fast learning process. **Firstly**, to reduce training latency, we design a decaying penalty function $\rho + t\rho/2(N-1)$ aiming to set S_{busy} as the lowest priority. Since our protocol requires that a pair (i, j) should be selected by each scheduling, it takes two steps in the learning process. Then, $2(N-1)$ represents the *best learning steps* at one *Epoch*. With N increases, the penalty reward of S_{busy} decreases. Thus, the later S_{busy} chooses, the smaller penalty reward the model gets. From concurrent perspective, the training latency can be reduced
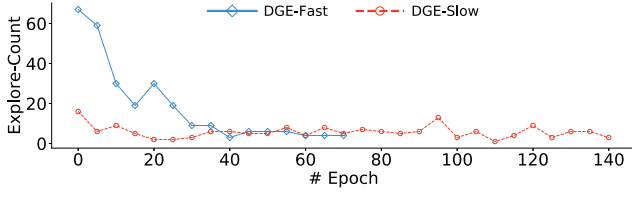
78

Figure 6: RL training speed comparison by using different exploration rate

as the process of model aggregation can be done with S_{free} in advance. **Secondly**, to balance energy consumption, if the count of $S_{free} \rightarrow S_{get}$ for each device is close to the average, the energy variance of all devices in Eq. (6) can reach the minimal. Hence, we design a penalty function $\alpha - \beta n_{agg}$, where n_{agg} records the count of $S_{free} \rightarrow S_{get}$. With n_{agg} increases, the penalty reward also increases. In addition, we design an incentive function $\alpha + \beta n_{agg}$ to speed up the model learning. With n_{agg} increases, the model can get more rewards if $S_{free} \rightarrow S_{send}$ is done. **Thirdly**, to achieve fast learning, we design an efficient *termination* function. If the selected action is a repeat as an *invalid* action or the learning step is larger than the *best learning steps*, the model gets a severe penalty reward and the current learning step is terminated. In summary, we present our **reward function** as follows.

$$r(t) = \begin{cases} \alpha - \beta n_{agg} & S_{free} \rightarrow S_{get} \\ \alpha + \beta n_{agg} & S_{free} \rightarrow S_{send} \\ \rho + t\rho/2(N-1) & S_{busy} \\ -1 & Exceeds\ limits \\ -1 & Select\ invalid\ action \\ 1 & Completed \end{cases} \tag{7}$$

where α, β and ρ denote initial reward, initial busy penalty and battery cost per aggregation, respectively. n_{agg} denotes the number of model aggregations. By default, we set α to -0.04, β to 0.1 and ρ to -0.8.

3.3.2 *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 [38]. The decaying greedy-exploration (DGE) strategy has been used as a default option with a fair performance. However, Figure 6 shows DGE-Fast (using 2x exploration rate) completes earlier than DGE-Slow, i.e., each stops at 73 *Epoch* and 143 *Epoch*, respectively. A key observation from our empirical study is that more sufficient exploration achieves more efficient learning time reduction. Therefore, in the proposed TDGE strategy, we aim to ensure sufficient exploration in the beginning to accelerate the learning process. The interpretation is to ensure the model to fully explore until the reward reaches the given threshold, we then switch the exploration to start epsilon decaying from the predefined value. Specifically, the threshold is approximately defined by achieving optimal latency and battery balance at the same time in Eq.(8) since the reward function is designed to optimize both features by Eq. (4). The estimation of optimal latency is defined as reward value by

Eq. (10), where the estimation of optimal battery balance is defined in Eq. (9).

We define threshold $T(N)$ by separating the calculation of the optimal reward into two parts in Eq. (8), which imitate architecture in defining optimization function of Chain-scheduler. The first part represents the maximal reward value of battery balance by Eq. (9). The second part denotes the optimal reward value of latency by Eq. (10).

$$T(N) = f(N) + g(N) - \Phi \tag{8}$$

where Φ denotes an offset value obtained from our experiment.

$$f(N) = \begin{cases} \beta(N\ mod\ 2) + f(\lfloor N/2 \rfloor) & N > 1 \\ 0 & N = 0\ \&\ 1 \end{cases} \tag{9}$$

$$g(N) = \beta \log_2 N \tag{10}$$

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

ALGORITHM 2: Chain-scheduler Algorithm

Initialize target network weights, epsilon $\epsilon = 1.0$, threshold TR in Eq. (8), state s;

while $Epoch < maxEpoch$ **do**
  for $t \leftarrow 1$ **to** $maxStep$ **do**
    if random($0,1$) $< \epsilon$ **then**
      | randomly explore an action from validAction();
    else
      take action a with ϵ-greedy policy based on Q-value function $Q(s_t, a_t)$;
    end
    receive s_{t+1} and r_t in Eq. (7);
    $s_t \leftarrow s_{t+1}$;
    if s_{t+1} is terminated **then**
      if *reward* $> TR$ **then**
      | $\epsilon \leftarrow \epsilon_{new}$
      end
      break;
    end
  end
  update ϵ by decaying;
end

3.3.3 *Continuous Environment Learning*.

As analyzed in Section (§2), *Non-Stationarity* scenario has a negative impact on the performance of the RL model. However, since MA can continuously monitor device resource condition, this impact can be mitigated by a repeating environment learning mechanism [2]. If the current resource condition is not equal to the previous record, MA can restart the RL learning process to update the model. Figure 5b indicates three *re-learning* conditions. In the first two conditions, the RL learning process can be restarted with a small learning slot by the resource condition changes of environment, and it finishes before the scheduling actions. For the third condition, if scheduling actions require to perform during the slot of *re-learning*, Chain-scheduler can output a fair scheduling based on the history experience of the RL model.
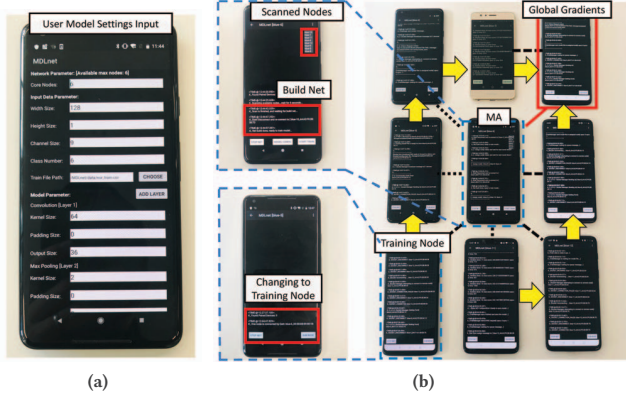
79

Figure 7: Experiment screenshot. (a) User model input; (b) Training execution screenshot

Table 1: Dataset Specifications

Datasets	Type	Task	#-S	#-C	#-SA	#-R	#-H	#-TR	#-TE
sEMG [22]	EMG	GR	37	6	14695	50	8	25	6
MHEALTH [3]	IMU	HBM	10	9	3255	50	23	100	30
UniMiB [30]	IMU	FDR	30	8	8430	50	1	116	15
HAR [1]	IMU	ADLs	30	6	10299	50	9	160	60
OPPORTUNITY [35]	IMU	ADLs	12	11	16837	50	77	500	50
PAMAP2 [34]	IMU	ADLs	9	12	12397	100	9	900	90

4 EVALUATION

4.1 MDLdroid 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. We also tailor our implementation for execution on Android smartphone. With minor model configurations, MDLdroid is fully compatible with a range of DL models without scaling down the model. We employ 9 off-the-shelf Android smartphones and Table 2 gives their specifications. Figure 7a plots a screenshot in which user customizes model configuration such as the parameters for certain datasets, customized hidden layer structures, and the required number of training devices. Figure 7b plots a screenshot during an execution of training on 9 smartphones using MDLdroid. The MA device scans all nearby devices, and build a BLE mesh network. The black dash lines represent the BLE connections between MA and training devices for resource condition monitoring. The yellow lines indicates a particular *chain-directed* model aggregation process via BS.

4.2 Datasets

To evaluate MDLdroid, we select 6 public personal mobile sensing datasets with a training scale ranged from 25M to 900M. These datasets are typically used for building a variety of personal mobile applications, e.g., gesture recognition (GR), recognition of activity for daily living (ADLs), fall detection recognition (FDR), and health behavior monitoring (HBM). The specification of each dataset is listed in Table 1. #-S, #-C, #-SA, #-R, #-H, #-TR, #-TE separately denote the number of subjects, the number of classes, the number

Table 2: Smartphone specifications

Device	HD	RAM	CPU	Battery(mAh)
OnePlus 6	128GB	8GB	Snapdragon 845	3300
Pixel 2 XL	64GB	4GB	Snapdragon 835	3520
Huawei Honor 8	32GB	4GB	HiSilicon Kirin 950	3000

of samples, sampling rate (Hz), the number of channels, the size of training data (MB), and the size of testing data (MB),

4.3 Evaluation Methodology

In our evaluation, all the participating smartphones are placed in proximity with a range from 1m to 5m for any twos. To evaluate actual battery consumption, we set all smartphones as discharging. Before each experiment, we reset the battery of smartphones to be full. For static dataset allocation, we divide each dataset equally among all the training devices since they have similar resource capacity and pre-load a sub-dataset to each device in advance to simplify our evaluation. We first evaluate the performance of Chain-scheduler. We then compare MDLdroid to Federated Learning. Finally, we conduct a series of experiments to discover optimized parameters to trade-off between resource used, training accuracy and scalability.

4.4 Performance of Chain-scheduler

To evaluate Chain-scheduler, we select two resource-agnostic schedulers (i.e., *Tree-scheduler* and *Ring-scheduler*) as the baseline. We also use both DGE and the threshold-based greedy-exploration (TGE) (i.e., the exploration only relies on the given threshold without decaying) approaches (§3.3.2) as the baseline to compare with our TDGE approach and evaluate the performance of the exploration strategy when training Chain-scheduler.

4.4.1 Experimental Setup.

For a fair benchmark comparison, we simulate the resource dynamicity scenarios 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 Figure 5b. We run each experiment 50 times and report the performance and resource usage presented in the following sections.

4.4.2 Exploration Strategy of Chain-scheduler.

In this experiment, we first compare TDGE with both DGE and TGE in term of training time. Figure 8a shows the average training time for each exploration strategy in a network size ranged from 3 to 9 devices. Result shows that TDGE outperforms TGE and DGE in training time by 1.2x and 1.3x less, with a standard deviation of 1.9x and 2.1x less, respectively. In addition, Figure 8b shows the convergence of cumulative reward in each strategy in a network size of 8 devices. The triangle markers indicate convergence points. We observe that TDGE converges 1.2x and 1.4x faster than TGD and DGE, respectively. As a result, TDGE can accelerate the process of Chain-scheduler training.
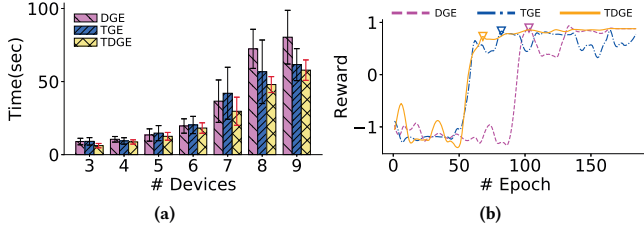
Authorized licensed use limited to: RMIT University Library. Downloaded on October 28,2020 at 01:37:29 UTC from IEEE Xplore. Restrictions apply.

Figure 8: Chain-scheduler training comparison by different exploration strategies. (a) Training time; (b) Convergence of cumulative reward
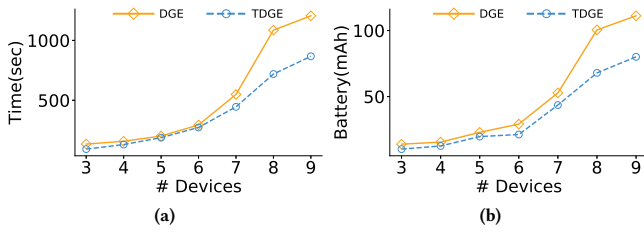

Figure 9: Resources reduction in re-learning conditions. (a) Time and (b) Battery

4.4.3 Performance in Re-learning.

In this experiment, we evaluate TDGE under a re-learning scenario in terms of training time and battery consumption. Figure 9a shows that TDGE outperforms DGE with less training time, especially in more-device scenarios (e.g., 1.5x less when the number of devices is larger than 7). Figure 9b shows that on average TDGE consumes 1.3x less battery consumption than DGE. Similarly, more battery savings are observed in TDGE with more devices.

4.4.4 Scheduling Performance.

In this experiment, we compare Chain-scheduler with Tree-scheduler and Ring-scheduler in terms of training time and energy balance using the HAR dataset in Table 1. Figure 10a shows that on average ChainSGD-scheduler outperforms Tree-scheduler and Ring-scheduler by 23% and 53% less training time, respectively. In addition, more time savings are observed in ChainSGD-scheduler when the number of devices increases. Figure 10b shows that Ring-scheduler achieves the minimal energy variance among three, and ChainSGD-scheduler reduces energy variance by 40% on average compared to Tree-scheduler. This experiment demonstrates that ChainSGD-scheduler achieves the best trade-off between training time and energy variance compared to Tree-scheduler and Ring-scheduler.

4.5 Performance of MDLdroid

To give a comprehensive evaluation for the performance of MDLdroid, we compare MDLdroid with the Federated Learning (FL) [7] from training accuracy and resource used perspectives. Besides, we
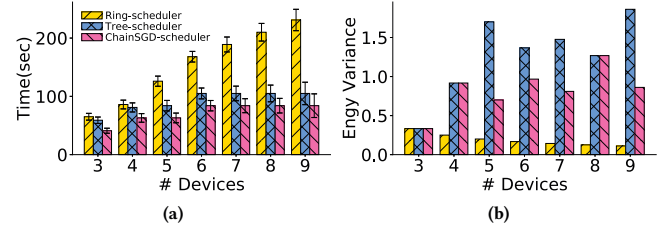

Figure 10: Scheduling performance comparison. (a) Latency; (b) Energy balance

choose a server-based approach to further ensure the training accuracy to be reliable. we next explore the optimized resource-accuracy trade-off options and limitations of MDLdroid.

4.5.1 Experimental Setup.

For performance benchmarking, we use FL as our baseline, which is implemented based on a master-slave structure, i.e., the same system architecture, to keep training computational and communication costs identical for fair comparison. In addition, we also compare the training accuracy of MDLdroid vs. the server-based approach by running training with the same model configurations on a desktop computer using all the 6 datasets. To measure battery consumption, we use Java refection to access the system instance of *BatteryStatsImpl* after rooting the smartphones. To monitor real-time CPUs and memory consumption on smartphone, we use Android Debug Bridge (ADB) commands.

4.5.2 MDLdroid vs. FL.

We compare the overall performance of MDLdroid vs. FL from three resource perspectives—peak-memory overhead, training time, and network energy balance.

Peak-Memory Figure 11a plots the peak-memory value for each approach with a network size ranged from 1 to 9 using the PAMA2 dataset based on LeNet. Result shows that the peak-memory overhead of the master device in FL increases linearly with the number of devices, while MDLdroid remains stable at low overhead due to only a pair of devices is involved during each model aggregation task.

Training Time Figure 11b shows that the training time in MDLdroid is effectively saved by 1.5x on average compared to FL. The training time in MDLdroid decreases with the network size increased. While the training time in FL presents a U-shaped curve as the communication time for model aggregations gradually increases. This is due to the efficient design of both model aggregation scheduling (§3.3) and resource-aware broadcasts (§3.2.3) in MDLdroid.

Network Energy Balance Figure 11c shows that the energy consumption of each device for model aggregation in MDLdriod is much less than that of the master device in FL, especially in more device scenarios. The result shows a training device in MDLdriod achieves 5.8x energy consumption reduction for communication on average compared to the master device in FL. In addition, another key observation is that MDLdroid achieves better energy balance among devices as MDLdroid evenly distribute model aggregation
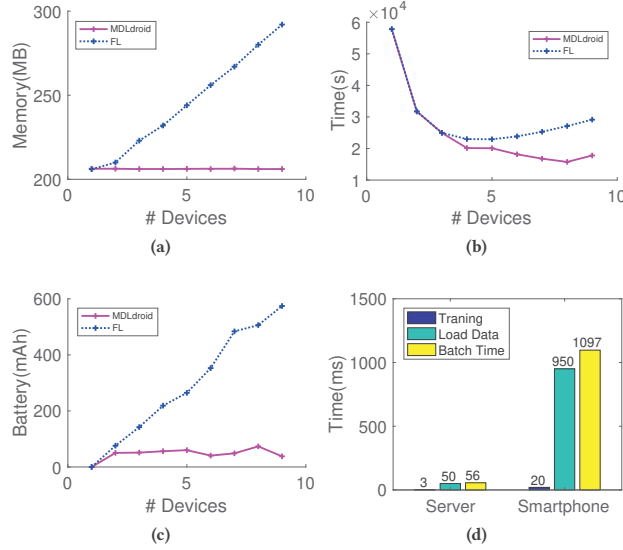
Figure 11: MDLdroid vs. FL performance comparison. (a) Memory; (b) Time; (c) Battery consumption; (d) One batch training time breakdown comparison

tasks to each device during scheduling. Since MDLdroid requires a training device to periodically report its resource condition via a tiny BLE message, the actual battery consumption to send model parameters is much smaller than that using BS, e.g., fully training PAMA2 by 20 *Epoch* on a single device with roughly 15 mAh out of 3600 mAh cost.

4.5.3   *Trade-off between Resource and Accuracy.*

    Both model aggregation frequency and training iteration *Epoch* can significantly impact the balance between resource and accuracy. In this experiment, we evaluate the trade-off between resource and accuracy in three aspects—training accuracy by a given threshold, battery consumption by maximum battery, and training time. In addition, we design 5 trade-off parameters, i.e., 1-E20, 2-E20, 10-E20, 1-E30, and 1-E40, where each of them represents the number of model aggregation rounds per *Epoch*—the number of training iteration *Epoch*.

    Scalability We observe that the training accuracy decreases after applying trade-offs, as shown in Figure 12. With the network size increases, the training data size for each device decreases, resulting that the learning convergence rate of a DL model becomes slow. However, Figure 12 indicates the accuracy can be improved if we enlarge the *Epoch* from 1-E20 to 1-E40, while the training time and battery cost increase as shown in Figure 14 and 13, respectively. Therefore, the scalability of network may be limited by the trade-off between resource and accuracy.

    Battery Saving To explore the battery limitations by different trade-offs, we continually charge smartphones during the experiment. Specifically, the battery consumption can be reduced by 2x to 8.5x compared to single-device training, as shown in Figure 13.

Table 3: Training Accuracy Comparison

Datasets	State-of-art	Server	FL-B	Chain-B	Chain-Off
HAR	96% [1]	93.8%	92.7%	92.5%	90.0%
PAMAP2	90%+ [34]	97.6%	94.7%	95.2%	90.7%
MHEALTH	90% [3]	92.3%	91.0%	90.2%	85.4%
UniMiB	85% [1]	96.1%	93.3%	93.6%	91.5%
sEMG	88% [22]	89.2%	86.3%	85.8%	84.6%
OPPO	85% [35]	88.6%	87.5%	86.9%	84.7%

With the number of devices increases, the battery consumption can be effectively shared among multiple devices.

    Training Time Reduction Figure 14 shows that the training time of MDLdroid is reduced by 2x on average compared to single-device training. Besides, the training time is significantly increased if the model aggregation frequency increases since sending model gradient parameters via BS is time-consuming depending on the model size of dataset.

    Training Speed Limitation Figure 11d reveals that loading one batch data from file to memory has a huge latency on device using DL4J libraries. The same result can be found on the server. As DL4J requires to convert data into an *INDArray*, the speed of this step depends on the performance of hardware (i.e., CPUs). Since DL4J uses only CPUs on Android, the latency is much larger than that on the server. Using accelerators available on the mobile device [8] may largely improve the training process.

    In summary, MDLdroid accelerates training by 2x to 3.5x as shown in Figure 14 and reduces battery consumption by 2x to 8.5x, compared to single device as shown in Figure 13. With the number of training *Epoch* increases, the training accuracy is increased by 2.5% on average as shown in Figure 12, but much more battery and time are consumed. While increasing model aggregation frequency can sightly improve training accuracy with a small impact on battery usage. However, the training time is increased by up to 2x due to heavy communication cost , as shown in Figure 14. Therefore, we choose 1-E20 as the optimized resource-accuracy option to achieve the best performance. Finally, Table 3 summarizes the comparison result of training accuracy, and MDLdroid achieves reliable state-of-the-art results. In the table, FL-B denotes the best accuracy on FL, Chain-B denotes the best accuracy on MDLdroid and Chain-Off denotes the accuracy on MDLdroid after trade-off.

5   DISCUSSION AND LIMITATION

In this section, we discuss several limitations in our current prototype implementation.

    Training Time The long training time in Figure 14 may not be very practical for end users in reality. However, the training speed of the prototype implementation depends on several factors: 1) the speed of reading training data from a CSV file on mobile device is low as shown in Figure 11d. 2) the training performance on single device is strongly limited by DL4J libraries; 3) the used standard model structures are large for mobile training, and advanced lightweight models should be applied in practice, such as MobileNet [16]. Further improving training time will leave for our future works.

    Bluetooth Limitation As an initial prototype system, we use Bluetooth Low Energy for building a mesh network to simplify our implementation. However, when the model complexity increases,
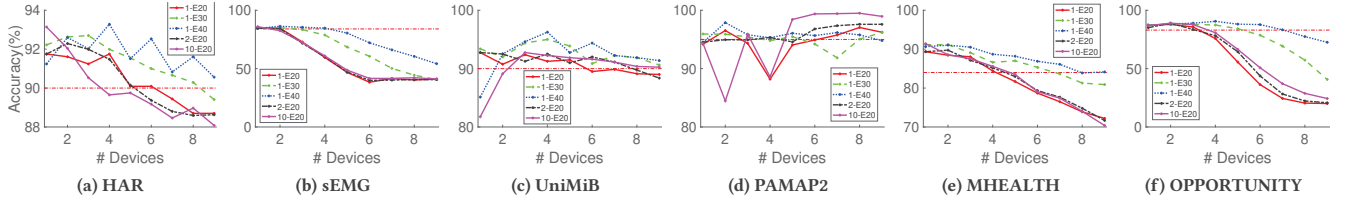
82

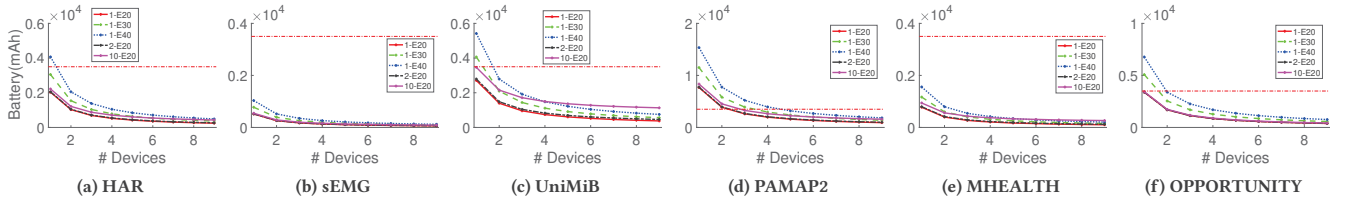**Figure 12: Training accuracy by trade-off options**

Figure 13: Training battery consumption by trade-off options
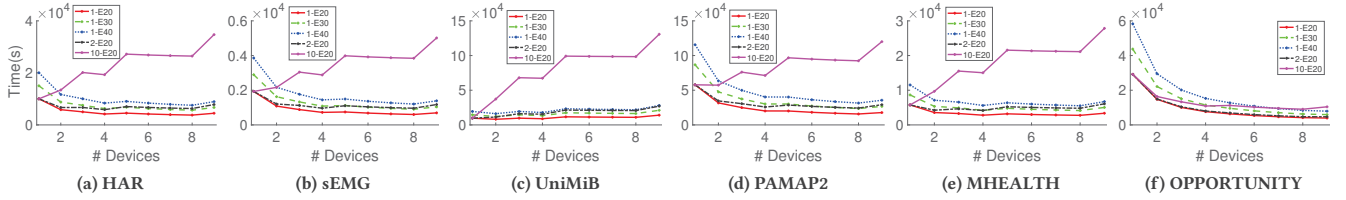

Figure 14: Training time by trade-off options

sending a large number of model parameters via BS may suffer long latency due to low bandwidth available in Bluetooth, and hence will affect the overall training performance. Since Wi-Fi Direct [36] has been widely available on smartphones and it offers much higher bandwidth, a hybrid solution can be implemented to use Wi-Fi Direct in the model aggregation process to reduce the communication cost between devices. Furthermore, proper model compressing techniques [29] can be applied for effective communication, which we leave for our future work.

Future User Inference and Applications MDLdroid primarily targets personal sensing applications which are privacy-sensitive with low-latency response requirement for continually model inference and update, but the framework can be applied to a wide range of DL-based low-latency applications with a moderate model size, such as real-time surveillance, image recognition, and natural language processing. The prototype user inference of MDLdroid is mainly used for experiments. Since end users may not need to manually set complex parameters for training, we plan to embed the MDLdroid into mobile OS to offer automatic background training, and develop a wider variety of applications to fully explore the capability of MDLdroid in our future work.

6 RELATED WORK

Decentralized Deep Learning For decentralized framework, the existing work [18] 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 [12] proposes a multi-tenant framework that can enable a resource-aware on-device to efficiently execute inference tasks for mobile vision applications. Besides, MCDNN [12] 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 [5] [33] 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.

7 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.

ACKNOWLEDGMENTS

REFERENCES

[1] Davide A., Alessandro G., Luca O., Xavier P., and J L Reyes-Ortiz. 2013. A Public Domain Dataset for Human Activity Recognition using Smartphones. In *ESANN'13*.
[2] S. Abdallah and M. Kaisers. 2016. Addressing Environment Non-Stationarity by Repeating Q-learning Updates. *Journal of Machine Learning Research* (2016).
[3] Oresti B., Rafael G., Juan A. H., Miguel D., Hector P., Ignacio R., Alejandro S., and Claudia V. 2014. mHealthDroid: A Novel Framework for Agile Development of Mobile Health Applications. In *Ambient Assisted Living and Daily Activities*.
[4] Amin B. Abkenar, Seng Loke, Arkady Zaslavsky, and Wenny Rahayu. 2019. GARSAaaS: group activity recognition and situation analysis as a service. *JISA'19* (2019).
[5] D. b. noureddine, Atef G., and Samir A. 2017. Multi-agent Deep Reinforcement Learning for Task Allocation in Dynamic Environment. In *ICSOFT'17*.
[6] Arjun Nitin Bhagoji, Supriyo Chakraborty, Prateek Mittal, and Seraphin Calo. 2019. Analyzing Federated Learning through an Adversarial Lens. In *ICML'19*.
[7] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, Chloé Kiddon, Jakub Konecný, S. Mazzocchi, H. B. McMahan, T. V. Overveldt, D. Petrou, D. Ramage, and J. Roselander. 2019. Towards Federated Learning at Scale: System Design. *CoRR* (2019).
[8] Yitao Chen, Saman Biookaghazadeh, and Ming Zhao. 2018. Exploring the Capabilities of Mobile Devices Supporting Deep Learning. In *HPDC '18*.
[9] Jeffrey D., Greg S. C., Rajat M., Kai C., Matthieu D., Quoc V. L., Mark Z. M., Marc'Aurelio R., Andrew S., Paul T., Ke Y., and Andrew Y. N. 2012. Large Scale Distributed Deep Networks. In *NIPS'12*.
[10] deeplearning4j. 2019. NDArrays: How Are They Stored in Memory? *Deeplearning4j: Open-source, distributed deep learning for the JVM* (2019).
[11] Yunbin Deng. 2019. Deep Learning on Mobile Devices - A Review. *CoRR* (2019).
[12] Biyi F., Xiao Z., and Mi Z. 2018. NestDNN: Resource-Aware Multi-Tenant On-Device Deep Learning for Continuous Mobile Vision. In *MobiCom '18*.
[13] S. Gupta, W. Zhang, and F. Wang. 2016. Model Accuracy and Runtime Tradeoff in Distributed Deep Learning: A Systematic Study. In *ICDM'16*.
[14] T. Hoefler, C. Siebert, and W. Rehm. 2007. A practically constant-time MPI Broadcast Algorithm for large-scale InfiniBand Clusters with Multicast. In *IPDPS'07*.
[15] Seyed Amir Hoseini-Tabatabaei, Alexander Gluhak, and Rahim Tafazolli. 2013. A Survey on Smartphone-Based Systems for Opportunistic User Context Recognition. *CSUR'13* (2013).
[16] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* (2017).
[17] Jeya Vikranth Jeyakumar, Liangzhen Lai, Naveen Suda, and Mani Srivastava. 2019. SenseHAR: A Robust Virtual Activity Sensor for Smartphones and Wearables. In *SenSys'19*.
[18] Zhanhong Jiang, Aditya Balu, Chinmay Hegde, and Soumik Sarkar. 2017. Collaborative Deep Learning in Fixed Topology Networks. In *NIPS'17*.
[19] Jakub Konecný, H. Brendan McMahan, Daniel Ramage, and Peter Richtárik. 2016. Federated Optimization: Distributed Machine Learning for On-Device Intelligence. *CoRR* (2016).
[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS'12*.
[21] J. Leon Kröger, Philip R., and T. Rahman B. 2019. Privacy Implications of Accelerometer Data: A Review of Possible Inferences. In *ICCSP '19*.
[22] Sergey L., Nadia K., Innokentiy K., Victor K., and Valeri A. M. 2018. Latent Factors Limiting the Performance of sEMG-Interfaces. *Sensors* (2018).
[23] Xiangru L., Ce Z., Huan Z., Cho-Jui H., Wei Z., and Ji L. 2017. Can Decentralized Algorithms Outperform Centralized Algorithms? A Case Study for Decentralized Parallel Stochastic Gradient Descent. In *NIPS'17*.
[24] Nicholas D. Lane and Petko Georgiev. 2015. Can Deep Learning Revolutionize Mobile Sensing?. In *HotMobile '15*.
[25] Francisco Laport-López, Emilio Serrano, Javier Bajo, and Andrew T. Campbell. 2019. A review of mobile sensing systems, applications, and opportunities. *KAIS'19* (2019).
[26] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* (1998).
[27] En Li, Zhi Zhou, and Xu Chen. 2018. Edge Intelligence: On-Demand Deep Learning Model Co-Inference with Device-Edge Synergy. In *MECOMM'18*.
[28] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *OSDI'14*.
[29] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J. Dally. 2017. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. *CoRR* (2017).
[30] Daniela M., Marco M., and Paolo N. 2017. UniMiB SHAR: a new dataset for human activity recognition using acceleration data from smartphones. *CoRR* (2017).
[31] E. L. M., Saeed A., Mark M., Matthew K., Julie A. K., Tanzeem C., Geri G., and Dan C. 2016. Mobile Manifestations of Alertness: Connecting Biological Rhythms with Patterns of Smartphone App Use. In *MobileHCI '16*.
[32] Riccardo Miotto, Fei Wang, Shuang Wang, and Xiaoqian Jiang. 2017. Deep learning for healthcare: review, opportunities and challenges. *Briefings in bioinformatics* (2017).
[33] T. T. Nguyen, N. D. Nguyen, and S. Nahavandi. 2018. Deep Reinforcement Learning for Multi-Agent Systems: A Review of Challenges, Solutions and Applications. *CoRR* (2018).
[34] A. Reiss and D. Stricker. 2012. Introducing a New Benchmarked Dataset for Activity Monitoring. In *ISWC'12*.
[35] D. Roggen, A. Calatroni, M. Rossi, T. Holleczek, K. Förster, G. Tröster, P. Lukowicz, D. Bannach, G. Pirkl, A. Ferscha, J. Doppler, C. Holzmann, M. Kurz, G. Holl, R. Chavarriaga, H. Sagha, H. Bayati, M. Creatura, and J. d. R. Millàn. 2010. Collecting complex activity datasets in highly rich networked sensor environments. In *INSS'10*.
[36] A. A. Shahin and M. Younis. 2014. A framework for P2P networking of smart devices using Wi-Fi direct. In *PIMRC'14*.
[37] Shaohuai Shi, Qiang Wang, Kaiyong Zhao, Zhenheng Tang, Yuxin Wang, Xiang Huang, and Xiaowen Chu. 2019. A Distributed Synchronous SGD Algorithm with Global Top-k Sparsification for Low Bandwidth Networks. *CoRR* (2019).
[38] Michel Tokic. 2010. Adaptive ϵ-Greedy Exploration in Reinforcement Learning Based on Value Differences. In *KI 2010: Advances in Artificial Intelligence*.
[39] He Wang, Ted Tsung-Te Lai, and Romit Roy Choudhury. 2015. MoLe: Motion Leaks Through Smartwatch Sensors. In *MobiCom '15*.
[40] J. Wang, B. Cao, P. Yu, L. Sun, W. Bao, and X. Zhu. 2018. Deep Learning towards Mobile Applications. In *ICDCS'18*.
[41] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, and K. Chan. 2019. Adaptive Federated Learning in Resource Constrained Edge Computing Systems. *J-SAC'19* (2019).
[42] J. Yang, H. Xu, and P. Jia. 2009. Task Scheduling for Heterogeneous Computing Based on Bayesian Optimization Algorithm. In *CIS'09*.
[43] Jian Bo Yang, Minh Nhut Nguyen, Phyo Phyo San, Xiao Li Li, and Shonali Krishnaswamy. 2015. Deep Convolutional Neural Networks on Multichannel Time Series for Human Activity Recognition. In *IJCAI'15*.
[44] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. 2019. Federated Machine Learning: Concept and Applications. *TIST'19* (2019).
[45] Kwangmin Yu, Thomas Flynn, Shinjae Yoo, and Nicholas D'Imperio. 2019. Layered SGD: A Decentralized and Synchronous SGD Algorithm for Scalable Deep Neural Network Training. *CoRR* (2019).
[46] D. Zhang, X. Chen, D. Wang, and J. Shi. 2018. A Survey on Collaborative Deep Learning and Privacy-Preserving. In *DSC'18*.
[47] Xiang Zhang, Lina Yao, Chaoran Huang, Sen Wang, Mingkui Tan, Guodong Long, and Can Wang. 2018. Multi-modality Sensor Data Classification with Selective Attention. In *IJCAI-18*.
[48] G. Zyskind, O. Nathan, and A. '. Pentland. 2015. Decentralizing Privacy: Using Blockchain to Protect Personal Data. In *SPW'15*.