ABSTRACT

FEDERATED LEARNING SYSTEMS FOR MOBILE SENSING DATA

by
Xiaopeng Jiang

Federated Learning (FL) has emerged as a new distributed Deep Learning (DL) paradigm that enables privacy-aware training and inference on mobile devices with help from the cloud. This dissertation presents a comprehensive exploration of FL with mobile sensing data, covering systems, applications, and optimizations.

First, a mobile-cloud FL system, FLSys, is designed to balance model performance with resource consumption, tolerate communication failures, and achieve scalability. In FLSys, different DL models with different FL aggregation methods can be trained and accessed concurrently by different apps. In addition, FLSys provides advanced privacy-preserving mechanisms and a common API for third-party app developers to access FL models. FLSys adopts a modular design and is implemented in Android and AWS cloud. Extended from FLSys, ZoneFL exploits a mobile-edge-cloud architecture to adapt models to user behaviors in different geographical zones to further improve scalability and model utility. Both FLSys and ZoneFL are evaluated with real-world deployments to showcase the superior model performance, scalability, and fault-tolerance.

Second, Federated Meta-Location Learning (FMLL) is proposed on smart phones for fine-grained location prediction, based on GPS traces collected on the phones. FMLL has three components: a meta-location generation module, a prediction model, and a FL framework. The meta-location generation module represents the user location data as relative points in an abstract 2-Dimensional (2D) space, which enables learning across different physical spaces. The model fuses Bidirectional Long Short-Term Memory (BiLSTM) and Convolutional Neural Networks (CNN) layers, where BiLSTM learns the speed and direction of the mobile

users, and CNN learns information such as user movement preferences. FMLL uses federated learning to protect user privacy and reduce bandwidth consumption.

Third, Complement Sparsification (CS) is presented as an FL pruning mechanism that achieves low bidirectional communication overhead between the server and the clients, low computation overhead at the clients, and good model accuracy. CS uses a complementary and collaborative pruning at the server and the clients. At each round, CS creates a global sparse model that contains the weights that capture the general data distribution of all clients, while the clients create local sparse models with the weights pruned from the global model to capture the local trends. For improved model performance, these two types of complementary sparse models are aggregated into a dense model in each round, which is subsequently pruned in an iterative process.

Fourth, Federated Continual Learning (FCL) is explored as a more intricate FL scenario wherein data accumulates over time and undergoes distributional changes. A framework, Concept Matching (CM), is introduced for efficient FCL. The CM framework groups client models into model clusters, and then uses novel CM algorithms to build different global models for different concepts in FL over time. In each round, the server sends the global concept models to the clients. To avoid catastrophic forgetting, each client selects the concept model best-matching the implicit concept of the current data for fine-tuning. To avoid interference among client models with different concepts, the server clusters the models representing the same concept, aggregates the model weights in each cluster, and updates each global concept model with a cluster model of the same concept. Since the server does not know the concepts captured by the aggregated cluster models, a theoretical grounded server CM algorithm is proposed to effectively update a global concept model with a matching cluster model.

FEDERATED LEARNING SYSTEMS FOR MOBILE SENSING DATA

by
Xiaopeng Jiang

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science

August 2024

APPROVAL PAGE

FEDERATED LEARNING SYSTEMS FOR MOBILE SENSING DATA

Xiaopeng Jiang

Dr. Cristian Borcea, Dissertation Advisor — Date
Professor, Computer Science Department, NJIT

Dr. Yi Chen, Committee Member — Date
Professor, Martin Tuchman School of Management and Computer Science Department, NJIT

Dr. Xiaoning Ding, Committee Member — Date
Associate Professor, Computer Science Department, NJIT

Dr. NhatHai Phan, Committee Member — Date
Associate Professor, Data Science Department, NJIT

Dr. Guy Jacobson, Committee Member — Date
Principal Inventive Scientist, AT&T Labs, Bedminster, NJ

BIOGRAPHICAL SKETCH

Author: Xiaopeng Jiang

Degree: Doctor of Philosophy

Date: August 2024

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science
  New Jersey Institute of Technology, Newark, NJ, US, 2024
- Master of Science in Computer Science
  New Jersey Institute of Technology, Newark, NJ, US, 2016
- Master of Science in Occupational & Health Engineering
  New Jersey Institute of Technology, Newark, NJ, US, 2011
- Bachelor of Engineering in Safety Engineering
  Northeastern University, Shenyang, Liaoning, China, 2009

Major: Computer Science

Presentations and Publications:

X. Jiang and C. Borcea, "Federated Continual Learning Using Concept Matching", the 38th Annual Conference on Neural Information Processing Systems (Neurips 2024), (Under submission 2024).

P. Lai, **X. Jiang**, V. D. Mayyuri, A. Chen, C. Borcea, N. Phan, R. Jin, "FedX: Adaptive Model Decomposition and Quantization for IoT Federated Learning", the 25th International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing (ACM MobiHoc 2024), (Under submission 2024).

X. Jiang, H. Hu, V. D. Mayyuri, A. Chen, D. M. Shila, A. Larmuseau, R. Jin, C. Borcea, N. Phan, "FLSys: Toward an Open Ecosystem for Federated Learning Mobile Apps", IEEE Transactions on Mobile Computing (TMC), Janurary 2024

P. Sen, **X. Jiang**, Q. Wu, M. Talasila, W. Hsu, C. Borcea, "GoPlaces: An App for Personalized Indoor Place Prediction", the 20th IEEE International Conference on Mobile Ad-Hoc and Smart Systems (IEEE MASS 2023), September 2023.

X. Jiang, T. Oh, N. Phan, H. Mohammadi, V. D. Mayyuri, A. Chen, R. Jin, C. Borcea, "Zone-based Federated Learning for Mobile Sensing Data", the 21st IEEE International Conference on Pervasive Computing and Communications (IEEE PerCom 2023), March 2023.

X. Jiang and C. Borcea, "Complement Sparsification: Low-Overhead Model Pruning for Federated Learning", the 37th AAAI Conference on Artificial Intelligence (AAAI 2023), February 2023.

P. Sen, **X. Jiang**, Q. Wu, M. Talasila, W. Hsu, C. Borcea, "Indoor Place Prediction on Smart Phones (Demo Abstract)", the 20th ACM Conference on Embedded Networked Sensor Systems (ACM SenSys 2022), November 2022.

X. Jiang, S. Zhao, G. Jacobson, R. Jana, W. Hsu, M. Talasila, S. A. Aftab, Y. Chen, C. Borcea, "Federated Meta-Location Learning for Fine-Grained Location Prediction", IEEE International Conference on Big Data (IEEE Big Data 2021), December 2021.

S. Zhao, **X. Jiang**, G. Jacobson, R. Jana, W. Hsu, R. Rustamov, M. Talasila, S. A. Aftab, Y. Chen, and C. Borcea. "Cellular Network Traffic Prediction Incorporating Handover: A Graph Convolutional Approach", the 17th Annual IEEE International Conference on Sensing, Communication and Networking (IEEE SECON 2020), June 2020.

To Those Whom I Love And Those Who Love Me

ACKNOWLEDGMENTS

I would like to acknowledge and give my warmest thanks to my dissertation advisor Dr. Cristian Borcea who made this work possible. Dr. Borcea has been a constant source of knowledge, inspiration, and constructive feedback. He cares deeply about us, his PhD students, and fosters an encouraging, welcoming, and productive research atmosphere. His high standards in research teach us every detail, from motivation to presentation. His guidance and advice carried me through all the stages of my PhD study, not only in my research, but also in all areas of my personal life.

Next, I would like to extend my heartfelt gratitude to every member of my Committee: Dr. Yi Chen, Dr. Xiaoning Ding, Dr. Hai Phan, and Dr. Guy Jacobson for their valuable feedback and suggestions that vastly helped in improving the dissertation and papers.

In addition, I would like to thank Dr. Manoop Talasila, Dr. Wen-Ling Hsu, Dr. Rittwik Jana, and Syed Anwar Aftab from AT&T Research Labs, and Dr. An Chen and Vijaja D. Mayyuri from Qualcomm for being my industry collaborators. Their comments and suggestions have helped me improve the dissertation greatly.

More specifically, I would like to thank my fellow lab-mates: Dr. Shuai Zhao, Dr. Han Hu, Dr. Phung Lai, and Pritam Sen, for their valuable insights and support during my research. Collaborating on research and writing papers with them has been a pleasant experience.

Most importantly, I would not have achieved a fraction of this without the unconditional love and support of my family members. Although they are overseas, they have striven to show me their constant support and have helped me tremendously when I was in need. My family has supported my entire journey, and endlessly helped me, and for that, I am forever thankful.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

LIST OF TABLES
(Continued)

Table **Page**

LIST OF FIGURES

LIST OF FIGURES
(Continued)

Figure **Page**

LIST OF FIGURES
(Continued)

Figure **Page**

CHAPTER 1

INTRODUCTION

Federated Learning (FL) [1] has the potential to bring deep learning (DL) on mobile devices, while preserving user privacy during model training. FL balances model performance and user privacy through three design features. First, each device trains a local model on its raw data. Second, the gradients of the local models from multiple users are sent to a server for aggregation to compute a global model that is more accurate than individual local models. Third, the server shares the global model with all users. During this federated training, the raw data from individual users never leave their devices. A wide range of mobile apps, e.g., predicting or classifying health conditions based on mobile sensing data, can benefit from running DL models on smart phones using FL, which offers privacy-preserving global training that incentivizes user participation.

Despite the growing interest in FL to preserve user privacy, the lack of a publicly available FL system has precluded the widespread adoption of FL models on smart phones. This has also limited our understanding of how real-world applications can benefit from FL. Furthermore, there are many open problems in FL, such as how to cope with non Independent and Identically Distributed (IID) data issue in FL, how to train models efficiently and collaboratively from resource restrained mobile devices, how FL system to be adapted in a mobile-edge-cloud computing architecture, how to tackle the ever-changing data distribution in the real world. This dissertation is to firstly design and implement an efficient end-to-end FL system driven by real-life mobile applications, enhance it in terms of scalability and mobile user mobility awareness, study some FL mobile sensing applications, and then to tackle some open

problems in FL, such as the communication and computation overhead, and Federated Continual Learning (FCL) scenario.

The rest of this chapter presents an overview of FL systems for mobile devices in Section 1.1, and discusses two FL applications with mobile sensing data in Section 1.2. Section 1.3 proposes two mechanisms to optimize FL: one to reduce communication and computation overhead, and the other to address the ever-changing data distribution in FL. The contributions of this dissertation proposal are presented in Section 1.4. Section 1.5 acknowledges the contributors to this dissertation. Finally, Section 1.6 details the structure of this dissertation.

1.1 Federated Learning Systems for Mobile Devices

1.1.1 FL System Driven by Real-life Mobile Applications

Despite progress on theoretical aspects and algorithm/model design for FL [2–6], the lack of a publicly available FL system targeting mobile devices has precluded the widespread adoption of FL models on smart phones, even though such models can enable novel mobile apps that apply DL on mobile data (many times collected from sensors on the phones) in a privacy-preserving manner. Furthermore, this has also limited our understanding of how real-world applications can benefit from FL. Most of existing FL systems are either unavailable for the research and practice communities (e.g., Google [1], FedVision [7]), under development [8], or do not support mobile devices [9]. Well-developed open systems enabling on-device training [10, 11] do not provide support for third-party app development and do not consider the constraints of mobile devices. Most of the existing FL studies are based on simulations [2–6, 12], which may lead to an oversimplified view of the applicability of FL models in real-world. In the meantime, although demonstrated in several scenarios such as keyboard typing prediction [13], FL lacks real-world applications, which can drive the design

of FL systems. Indeed, real-world benchmarks for FL are pivotal to help shape the developments of FL systems [14].

In this dissertation, we take a unique application-system co-design approach to design, build, and evaluate an FL system. Our system design is informed by a critical mobile app, which illustrates a large category of apps that use DL on mobile sensing data: human activity recognition (HAR) on smart phones. In addition to HAR, we analyzed other real-life applications [1, 7, 13, 15, 16] to inform the system design. A list of important questions emerges, and many of them are not addressed in existing FL system designs [1, 8, 13, 16] that largely ignore the constraints of mobile devices: How can we balance FL model performance with resource constraints on the phones? How can we ensure the training conducted on phones is completed on time, despite limited resources, i.e., computation power and battery life? How can the server achieve seamless scalability and accurate model aggregation in the presence of large and variable numbers of users who typically train different models and how can the system simultaneously cope with potential communication failures (e.g., connectivity lost on the phone)? After a global model is shared with the phones, how can a third-party DL app utilize this model? How does the system support different types of advanced privacy preserving mechanisms?

Aiming to answer the aforementioned research questions, this dissertation presents the design, implementation, and evaluation of **FLSys**, a mobile-cloud federated learning (FL) system that supports deep learning models for mobile apps. FLSys is a key component toward creating an open ecosystem of FL models and apps that use these models.

1.1.2 Zone-based FL System

To further improve the accuracy of a mobile sensing model with FL, the system needs to adapt to user behavior, which is location-dependent. For example, people's

lifestyles depend on their living areas. Living in dense areas of the city with fewer recreational facilities prevents people from doing enough exercise. Similarly, health problems may be related to the level of pollution in different parts of the city.

We propose **Zone-based Federated Learning (ZoneFL)**, a novel federated learning (FL) architecture that builds and manages different models for different geographical zones. By design, ZoneFL satisfies the privacy-preserving requirement because FL [17] learns from data collected by many users, while protecting the user data privacy during training. In FL, the models are trained on mobile devices with their local data, and the server aggregates the models received from mobile devices. The users' privacy-sensitive data never leave the mobile devices.

We give vehicular traffic prediction and heart health notification as two concrete motivating examples. For traffic prediction, the traffic patterns in shopping districts and business districts are different because of different zone-dependent user behavior. A heart health notification app sends alerts about the level of cardiovascular risk associated with users' current activity based on the altitude and climate of a geographical zone. Using ZoneFL will outperform a global model in such applications, and we enjoy privacy prescerving and scalability of ZoneFL as well.

ZoneFL divides the physical space into geographical zones mapped to a mobile-edge-cloud system architecture for good model accuracy and scalability. Each zone has a federated training model, called a zone model, which adapts well to data and behaviors of users in that zone. Benefiting from the FL design, the user data privacy is protected during the ZoneFL training. We propose two novel zone-based federated training algorithms to optimize zone models to user mobility behavior: Zone Merge and Split (ZMS) and Zone Gradient Diffusion (ZGD). ZMS optimizes zone models by adapting the zone geographical partitions through merging of neighboring zones or splitting of large zones into smaller ones. Different from ZMS, ZGD maintains fixed zones and optimizes a zone model by incorporating the gradients derived from

neighboring zones' data. ZGD uses a self-attention mechanism to dynamically control the impact of one zone on its neighbors. Extensive analysis and experimental results demonstrate that ZoneFL significantly outperforms traditional FL in two models for heart rate prediction and human activity recognition. In addition, we developed a ZoneFL system using Android phones and AWS cloud. The system was used in a heart rate prediction field study with 63 users for 4 months, and we demonstrated the feasibility of ZoneFL in real-life.

1.2 Federated Learning Applications with Mobile Sensing Data

1.2.1 Human Activity Recognition

FLSys design is largely informed by a critical mobile app: human activity recognition (HAR) on the phones, which is important for industry, public health, and research. Simply speaking, mobile apps using HAR can harness recognized human physical activities using data collected from phone sensors. HAR is a representative FL app on smart phones that needs privacy-sensitive mobile sensing data collected in the wild in order to work effectively. From an industry point of view, accurate HAR can help the smart phone manufacturers to be smart about allocating resources and extending battery life. The Covid-19 pandemic highlights the public health importance of understanding individual & population behaviors under government orders and (health) emergencies [18]; furthermore, combining user activities with mental wellness surveys and prediction has the potential to develop personalized interventions to help individuals to better cope with anxiety, stress, and substance abuse, and other important societal issues [19]. Current research on HAR models uses centralized learning on data collected in controlled lab environments on standardized devices and controlled activities [20–26], which do not work well in real-world. Furthermore, they do not consider the inter-play between concurrent data collection, training, and inference on model utility and resource consumption on the phones.

Instead, we use HAR in the wild (open environments, where the user mobility, activities, or application usage are not controlled in any way). Users' behaviors, revealed by HAR data collected over long periods of time, may be privacy-sensitive, especially when location data is collected in addition to inertial measurement unit (IMU) data. Furthermore, collecting user data at a central server for training may violate recent privacy regulations (e.g., GDPR). In general, the privacy-sensitive nature of mobile sensing data, which may also include photos and videos, makes HAR ideal for studying the design of FL systems.

To study how HAR can be supported by FLSys in the wild, we collected data from 100+ college students in two areas during April - August 2020. The students used their own Android phones, and their daily-life activities were not constrained in any way by our experiment. Data collected on mobile devices is non-IID, which affects FL-trained models [15]. We have evaluated a variety of HAR models in both centralized and federated training, and designed **HAR-Wild**, a Convolution Neural Network (CNN) model with a data augmentation mechanism to mitigate the non-IID problem.

1.2.2 Fine-Grained Location Prediction

Another critical application of FL with mobile sensing data is fine-grained user location prediction. A system that achieves high accuracy for fine-grained user location prediction on smart phones can be used by the OS and the apps to improve system or app performance [27]. For example, accurate location prediction can be utilized in 5G networks at every time scale and across all layers of the protocol stack [28]. Since 5G performance is sensitive to small changes in location, the phone could use a map showing location-based quality of wireless network service to adapt video quality as a function of the predicted user locations. Augmented reality apps are delay-sensitive and can benefit from fine-grained location prediction to speed up

content rendering. Yet another example is context-aware apps that need to adapt in advance based on where the user will move next, such as location-based gaming or advertising. For instance, location-based gaming could adapt in real-time based on the predicted user locations to be more interesting or challenging.

Existing location prediction systems cannot be used in such scenarios. Most location prediction research focuses on Place ID prediction. They work either at large spatial scales, or at large time scales. For example, works for destination prediction [29–32], place-label prediction [33–35], and Place of Interest (POI) prediction [36–43] have poor spatial accuracy (e.g., hundreds of meters). We are aware of one study predicting location at small time-scale (e.g., predict where the user will be in several minutes), but the location error is in the order of hundreds of meters [44]. There are additional works that focus on small time-scale check-in POI prediction [45, 46], but they do not work for fine-grained locations or for every location in a road network.

A location prediction system would have limited usability if it could predict only places that have been visited previously by the user. The system can be improved by training the prediction model with data collected by all the users who adopt the system. While sharing location data across users will improve prediction accuracy, a naive method using GPS traces directly in centralized training is unlikely to be accepted by the users due to privacy concerns [47]. If users' location traces are disclosed, the identities of the users can be inferred even if pseudonyms are used [48, 49]. This is due to the fact that location can contain identity information [50]. Thus, the system needs to also provide location privacy protection.

We propose Federated Meta-Location Learning (FMLL) for fine-grained location prediction from GPS traces that works on the users' phones while preserving users' location privacy. In our work, the term fine-grained refers to both spatial and temporal scales. FMLL uses FL framework with two additional components: a

meta-location generation module and a prediction model. The framework runs on the phones of the users and also on a server that coordinates learning from all users in the system. The meta-location generation module represents the user location data as relative points in an abstract 2D space, which enables learning across different physical spaces. The model fuses Bidirectional Long Short-Term Memory (BiLSTM) and Convolutional Neural Networks (CNN), where BiLSTM learns the speed and direction of the mobile users, and CNN learns information such as user movement preferences. FMLL uses federated learning to protect user privacy and reduce bandwidth consumption. Our experimental results, using a dataset with over 600,000 users, demonstrate that FMLL outperforms baseline models in terms of prediction accuracy. We also demonstrate that FMLL works well in conjunction with transfer learning, which enables model reusability. Finally, benchmark results on Android phones demonstrate FMLL's feasibility in real life.

1.3 Federated Learning Optimizations

1.3.1 Low-Overhead Model Pruning for FL

Traditionally, FL uses dense and over-parameterized DL models. Empirical evidence suggests that such models are easier to train with stochastic gradient descent (SGD) than more compact representations [51]. However, the over-parameterization comes at the cost of significant memory, computation, and communication overhead. This is a problem for resource-constrained mobile and Internet of Things (IoT) devices [52], a major target for FL, which need to perform not only inference but also training. Therefore, reducing the computation and communication overhead in FL, while maintaining good model performance, is essential to ensure widespread FL deployment on mobile and IoT devices.

One potential solution to this problem is model pruning/sparsification, which aims to produce sparse neural networks without sacrificing model performance [53].

Sparse models result in significantly reduced memory and computation costs compared to their dense counterparts, while performing better than small dense models of the same size [54]. Sparse models lead to a better generalization of the networks [55] and are more robust against adversarial attacks [56, 57]. Pruning/sparsification can be used in FL, where the server and the clients can collaboratively optimize sparse neural networks to reduce the computation and communication overhead of training.

Despite the benefits of sparse models, it is challenging to design a communication-computation efficient model pruning for FL. A typical pruning mechanism has three stages: training (a dense model), removing weights, and fine-tuning [58]. Since a model with some of the weights removed has to recover the performance loss through additional fine-tuning in the back-propagation, the fine-tuning together with weights removal represents the computation overhead of the mechanism. In FL, this overhead cannot be placed only on the server because the server does not have access to the raw training data for fine-tuning. Therefore, pruning has to be done collaboratively between the server and the clients, and a significant computation overhead will be placed on the clients. Since FL exchanges model updates between the clients and the server every training round, smaller pruned models will lead to lower communication overhead. However, low communication overhead comes at the expense of computation overhead for pruning.

We propose Complement Sparsification (CS), a pruning mechanism that satisfies all these requirements through a complementary and collaborative pruning done at the server and the clients. At each round, CS creates a global sparse model that contains the weights that capture the general data distribution of all clients, while the clients create local sparse models with the weights pruned from the global model to capture the local trends. For improved model performance, these two types of complementary sparse models are aggregated into a dense model in each round, which is subsequently

pruned in an iterative process. CS requires little computation overhead on the top of vanilla FL for both the server and the clients. We demonstrate that CS is an approximation of vanilla FL and, thus, its models perform well. We evaluate CS experimentally with two popular FL benchmark datasets. CS achieves substantial reduction in bidirectional communication, while achieving performance comparable with vanilla FL. In addition, CS outperforms baseline pruning mechanisms for FL.

1.3.2 Federated Continual Learning Using Concept Matching

Most of the current FL research assumes the data have been collected before training, and the data at clients do not change over the training rounds. In many applications, this is not the case, as data accumulate over time and change its distribution. The data distribution change, also referred to as concept drift, makes prediction models obsolete over time. For example, a user sleep quality prediction model trained on data collected during routine life will not work well when the users experience changes in their sleep patterns due to stress, illness, or travel. For a traffic prediction model, the opening of a new highway, seasonal variations in traffic, and changes in public transportation routes can alter the data distribution over time, and impair the prediction performance. In addition, on mobile/IoT devices, such as such as smart watches and smart cameras, it is difficult to train with the entire dataset on-device at every round due to their resource constraints. This effect of dynamic data is being actively studied by the Continual Learning (CL) community in centralized settings. However, CL research in FL settings is still in its infancy.

Federated Continual Learning (FCL) performs FL under the CL dynamic data scenarios. There are two main challenges in FCL. One, inherited from CL, is catastrophic forgetting [59]. Due to concept drift, the model forgets previously learned knowledge as it learns new information over time. A concept infers a function from training examples of its inputs and outputs [60]. For example, in human activity

recognition (HAR) [61], the concepts can be the subsets of activities, the locations of the activities, or the health status of the user. FCL imposes privacy constraints on the top of CL, which escalates this challenge. Even if the clients are aware of concept drift (e.g., sedentary vs. active lifestyle in HAR), they may not want to reveal it to the FL server due to privacy concerns. The second challenge is that the FL clients with different data concepts may potentially interfere with each other, because the data in FL is typically non independently or identically distributed (non-iid). The interference will sabotage the efforts of clients' training during aggregation and lead to underperforming global models. CL amplifies this interference in FL, because the union of the clients data may also be distributed differently over time.

An efficient FCL framework shall tackle these challenges to achieve good model performance. So far, no existing system has achieved this goal under realistic assumptions. While several works [62–70] have recently targeted FCL, their applicability is limited due to unrealistic assumptions (e.g., the server knows the concept drift from the clients or the classes to learn do not change over time), or the interference among the clients is not handled.

We propose Concept Matching (CM), an FCL framework to address these challenges. The CM framework groups client models into model clusters, and then uses novel CM algorithms to builds different global models for different concepts in FL over time. In each round, the server sends the global concept models to the clients. To avoid catastrophic forgetting, each client selects the concept model best-matching the implicit concept of the current data for fine-tuning. To avoid interference among client models with different concepts, the server clusters the models representing the same concept, aggregates the model weights in each cluster, and updates each global concept model with a cluster model of the same concept. Since the server does not know the concepts captured by the aggregated cluster models, we propose a novel server CM algorithm that effectively updates a global concept model with

a matching cluster model. We formulate and prove the theoretical ground of the server CM algorithm, which guarantees to update the concept models in the right gradient descent direction. In addition, the CM framework provides flexibility to use different clustering, aggregation, and concept matching algorithms. The evaluation over several datasets demonstrates that CM outperforms state-of-the-art systems and scales well with the number of clients and the model size.

1.4 Contributions of Dissertation

1.4.1 FL System Driven by Real-life Mobile Applications

This dissertation presents **FLSys**, the first FL system in the literature created using an application-system co-design approach for smart phones to address the aforementioned research questions in Subsection 1.1.1. We provide a comprehensive description of the design, implementation, and evaluation of **FLSys**. The two main challenges for an FL system on phones are concurrent management of multiple FL activities under resource constraints and frequent disconnections due to networking and battery issues. These two challenges are not considered by any existing FL system. To solve them, we propose an innovative system architecture that provides (1) a unified system to manage resources on the phone in the presence of multiple models, third-party apps using these models, and data collectors for these models; and (2) an asynchronous protocol to manage the FL process in the presence of disconnections. The FLSys components on smart phones manage training, inference, data collection/preprocessing, and privacy to balance model utility with resource consumption, while tolerating disconnections.

Furthermore, the engineering of an effective and efficient FLSys prototype on Android and AWS and its evaluation with data collected in the wild is also a major novel contribution of this article. No such system is currently available to the research

community. While implemented in Android and AWS, FLSys has a general system design and API that can be extended to other mobile OSs and cloud platforms.

At a more specific level, there are four novel contributions in the system architecture that combine solutions in machine learning, fault-tolerance, software engineering, and cloud systems. First, FLSys balances model performance, privacy and resource consumption on-demand through data collection and training configurations, such as sampling rate, model structure, hyper-parameters, and differential privacy (DP) mechanisms. Second, FLSys uses an asynchronous protocol between the server and the phones to handle phone failures to participate in training due to resource constraints or disconnections, while maintaining good model performance. This protocol allows the devices to self-select for training when they have enough data and resources and allows the sever to operate correctly in the presence of communication failures with the phones. Third, FLSys enables an ecosystem of third-party apps and models, as well as the ability to use different aggregators, data collectors/preprocessors, and DP-based privacy mechanisms through its modular design. FLSys provides a common API for third-party apps to retrieve inference results from different DL models, while efficiently managing resource consumption and contention. FLSys also flexibly supports different types of DP mechanisms, both on the mobile devices and in the cloud to protect user privacy against an honest-but-curious server. Fourth, in FLSys, different aggregation algorithms and training policies can be deployed selectively as modules in the cloud using function as a service (FaaS) support, which makes operating FL more cost-efficient. We also leverage FaaS and cloud storage solutions to engineer a scalable FL server.

Another novel contribution of this article is the HAR model that we designed and built to test FLSys, which is tailored to work efficiently on resource-constrained phones with non independent and identically distributed (non-IID) data. For HAR experiments on FLSys, we collected data from 100+ college students in two

areas during a 4-month period. The students used their own Android phones, and their daily-life activities were not constrained in any way by our experiment. Data collected on mobile devices are non-IID, which affects FL-trained models [15]. We have evaluated a variety of HAR models with both centralized and federated training, and designed **HAR-Wild**, a Convolution Neural Network (CNN) model with a data augmentation mechanism to mitigate the non-IID problem. HAR-Wild was also designed to have a small memory footprint, which is appropriate for resource-constrained devices. To showcase the ability of FLSys to work with different FL models, we also built and evaluated a natural language sentiment analysis (SA) model on a dataset with 46,000+ tweets from 436 users.

We carried out a comprehensive evaluation of FLSys together with HAR-Wild and SA to quantify the model utility and the system feasibility in real life conditions. This article is the first in the literature to share an extensive FL evaluation on smart phones, using an end-to-end mobile-cloud FL system and mobile data collected in the wild. We conducted a comprehensive evaluation across three distinct training settings: 1) centralized training, 2) simulated FL with advanced privacy preserving mechanisms, and 3) Android FL. Centralized training provides an upper bound on model accuracy and is used to compare our HAR-Wild model with baseline approaches. The results demonstrate that HAR-Wild outperforms the baseline models in terms of accuracy. Furthermore, the federated HAR-Wild performance using simulations (TensorFlow and DL4J [1]), Android emulations, and Android phone experiments is close to the upper bound performance achieved by the centralized model. The results on smart phones demonstrate that FLSys can perform communication and training tasks within the allocated time and resource limits, while the FL server is able to handle a variable number of users. Finally, micro-benchmarks

[1]https://deeplearning4j.org/

on Android phones show FLSys with HAR-Wild and SA are practical in terms of training and inference time, as well as memory and battery consumption.

1.4.2 Zone-based FL System

To build an effective DL system for mobile sensing data that works efficiently on smart phones, the following requirements shall be satisfied: (i) *Privacy-preserving*: learn from data provided by many users, while protecting user data privacy; (ii) *Mobility-awareness*: achieve good model accuracy by adapting to user mobility behavior, and (iii) *Scalability*: scale well as the number of users increases. We propose **Zone-based Federated Learning (ZoneFL)**, a novel federated learning (FL) architecture that builds and manages different models for different geographical zones, to satisfy these requirements.

The main novel contribution of ZoneFL is its zone-based approach to satisfy requirements for mobility-awareness and scalability. To adapt DL models to user mobility for higher accuracy and to achieve good scalability, ZoneFL divides the physical space into geographically non-overlapping zones mapped to a mobile-edge-cloud architecture. Each zone trains its own zone model, which adapts to the data and behaviors of the users who spend time in that zone. As users move from one zone to another, collect data, and participate the training of different zones. For inference, their mobile devices switch from one zone model to another. Thus, zone models achieve higher accuracy than globally trained FL models, satisfying the mobility-awareness requirement. In ZoneFL, edge nodes manage the FL training within their zones and host the latest models for their zones. Mobile devices can download these models when they enter a new zone. The cloud collaborates with the edge nodes to dynamically maintain the zone partitions for the entire space, but it is not involved in training. Compared to traditional FL mobile-cloud architecture, the mobile-edge-cloud architecture of ZoneFL is more scalable because model aggregation is done

distributedly at the edge (satisfying the scalability requirement), has lower latency for mobile users who interact with the edge instead of the cloud, and results in less bandwidth consumption in the network core [71, 72].

A major challenge in ZoneFL is how to ensure the zone models adapt to user mobility behavior changes over time. To solve this challenge, we propose two novel zone-based federated training algorithms: Zone Merge and Split (ZMS) and Zone Gradient Diffusion (ZGD). ZMS optimizes zone models by adapting the zone geographical partitions through merging of neighboring zones or splitting of large zones back to previously merged smaller zones. The algorithm ensures that merging and splitting results in better model accuracy in each new zone. ZMS can be used when the initial zone partitions are suboptimal, and the zone partitions will be gradually improved as ZMS proceeds. Different from ZMS, ZGD maintains fixed zones and optimizes a zone model by leveraging concepts from graph neural networks to incorporate the gradients derived from neighboring zones' data. ZGD uses a self-attention mechanism to dynamically control the impact of one zone on its neighbors. ZGD can be used to further optimize zone models when the zone partitions are relatively stable according to ZMS.

ZoneFL was evaluated in terms of model accuracy and system performance using two models and two real-world datasets: Human Activity Prediction (HAR) with mobile sensing data collected in the wild, and Heart Rate Prediction (HRP) with the FitRec dataset [73]. The results demonstrate that models using ZoneFL without optimization performed by ZMS and ZGD significantly outperform their counterpart models using traditional FL for zones that have enough training data. ZoneFL with ZGD and ZMS further imporve the model performance, with ZMS improving the performance in the initial rounds and ZGD after that.

We implemented a ZoneFL system using Android phones and AWS cloud. The system was tested with the HRP model in a field study in the wild with 63 users for

4 months. The results show that ZoneFL achieves low training and inference latency, as well as low memory and battery consumption on the phones. ZoneFL scales better, because a zone edge server only handles only 34.98% to 37.26% of the communication and computation load handled by a global FL server. We also observed multiple zone merges and splits in the field study, when the model utility improved significantly. Compared with global FL, ZoneFL has a slightly higher training time on the mobile phones when the users participate in training for several zones. This overhead is an acceptable cost for the benefits provided by ZoneFL. Overall, the system results demonstrate the feasibility of ZoneFL in a real-life deployment.

1.4.3 Federated Meta-Location Learning

We build a novel location prediction system that is meticulously designed to meet the following requirements: (R1) achieves high prediction accuracy at a fine-grained spatio-temporal scale; (R2) works well for pedestrians and bicyclists; (R3) works in places that have not been visited before by the owners of the smart phones invoking the prediction there; and (R4) protects user location privacy. To the best of our knowledge, there is no existing work that satisfies all these requirements.

This dissertation presents Federated Meta-Location Learning (FMLL) that satisfies all these requirements. FMLL uses FL framework with two main components: a meta-location generation module and a prediction model. The meta-location generation module represents the user location data as relative points in an abstract 2D space, which is a grid with fixed-size cells. Meta-locations enables training on data received from all users even from different physical locations. This meta-location also scales all data to the same range and avoids the bias introduced by data with high longitude and latitude values, which weighs more during the deep learning optimization. Our novel prediction model is trained on input derived from meta-location. The model uses Bidirectional Long Short-Term Memory (BiLSTM)

and Convolutional Neural Networks (CNN), where BiLSTM learns the speed and direction of the mobile users, and CNN learns information such as user movement preferences. These two components are fused into a dense network with softmax activation. The federated learning framework allows the system to train on data from all users, while protecting user privacy.

Both meta-location and the prediction model contribute to the superior prediction accuracy for pedestrians and bicyclists (requirements R1 and R2). The learning framework runs on the smart phones of the users and on a server that coordinates learning from all users in the system. It helps to satisfy R3 because a user can benefit from the learning on other users' smart phones with locations not visited by this user. Meta-location also contributes to R3 because it can extract repeated patterns, even when the physical locations are different. In FMLL, privacy is protected by combining federated learning (FL) [74] with our meta-location generation (R4). FL trains the models locally on each phone and then computes a global model at the server by aggregating the gradients of the local models. In this way, the server never gets access to the raw data. However, the gradients of the local models may still leak private location information if the FL model uses physical location data [75]. This problem is substantially mitigated by using meta-locations, because similar meta-locations may be generated from different physical locations, making the identification of physical locations at the server more difficult.

Our experimental results, using a dataset with over 600,000 users, demonstrate that FMLL outperforms baseline models in terms of prediction accuracy for pedestrians and bicyclists respectively. We also demonstrate model reusability on another dataset, using FMLL with transfer learning [76]. We benchmarked the model on Android phones, and the results demonstrate that both training and inference are feasible in terms of execution time and battery consumption.

1.4.4 Complement Sparsification to Reduce Overhead

To design a communication-computation efficient model pruning mechanism for FL, four requirements must be satisfied: (R1) reduce the size of the local updates from the clients to the server; (R2) reduce the size of the global model transferred from the server to the clients; (R3) reduce the pruning computation overhead at the clients; (R4) achieve comparable model performance with dense models in vanilla FL. All these requirements must be satisfied under the assumption that the server does not have access to raw data due to privacy concerns. None of the existing works on FL pruning [77–81] can satisfy these requirements simultaneously. They either impose substantial computation overhead on the clients or only reduce the communication overhead from the clients to the server, but not vice versa. The main unsolved problem is the apparent contradictory nature of the requirements.

We propose **Complement Sparsification (CS)**, a pruning mechanism for FL that fulfills all the requirements. The main idea is that the server and the clients generate and exchange sparse models complementarily, without any additional fine-tuning effort. The initial round starts from vanilla FL, where the clients train a dense model for the server to aggregate. The server prunes the aggregated model by removing low magnitude weights and transfers the global sparse model to the clients. In the following rounds, each client trains from the sparse model received from the server, and only sends back its locally computed sparse model. The client sparse model contains only the weights that were originally zero in the global sparse model, thus complementing the global model. Then, the server produces a new dense model by aggregating the client sparse models with the global sparse model from the previous round. As in the initial round, the server removes the weights with low magnitude and transfers the new global sparse model to the clients. The new model has a different subset of non-zero weights because the client model weights are

amplified with a given aggregation ratio to outgrow other weights. In this way, all the weights in the model get updated to learn over time.

In CS, both the server and the clients transfer sparse models to save communication overhead bidirectionally (R1 and R2). Without deliberate fine-tuning, the computation overhead imposed on the system is minimized (R3). In CS, the pruning at the server preserves a global model that captures the overall data distribution, while the newly learnt client data distribution resides on the complementary weights (i.e., the zero weights of the global sparse model). Practically, the clients' training recovers the model performance loss during pruning without additional fine-tuning. Iteratively, the performance of the global model improves over time. Eventually, the clients can use the converged global sparse model for inference. This process can achieve comparable model performance with dense models in vanilla FL (R4).

We demonstrate that CS is an approximation of vanilla FL, and evaluate CS with two popular benchmark datasets [82] for Twitter sentiment analysis and image classification (FEMNIST). We measure model sparsity to quantify communication overhead. Specifically, CS achieves good model accuracy with server model sparsity between 50% and 80%. This sparsity represents the overhead reduction in the server-to-clients communication. The clients produce model updates with sparsity between 81.2% and 93.2%. The client sparsity represents the overhead reduction for client-to-server communication. CS reduces the computation overhead by 29.1% to 49.3% floating-point operations (FLOPs), compared with vanilla FL. We also demonstrate through experiments and a qualitative analysis that CS performs better than baseline model pruning mechanisms in FL [78, 79] in terms of model accuracy and overhead.

1.4.5 Concept Matching for FCL

Concept Matching (CM) is the first framework for FCL to tackle catastrophic forgetting due to concept drift over time in CL, overcome the interference among

clients in FL, and achieve good model performance. Intuitively, if we can separate the client models based on the data concepts, and train different models specifically to learn each concept iteratively, catastrophic forgetting and the interference among clients can be greatly diminished. This process has to be performed under the FL assumption that the server cannot access any raw data. The CM framework achieves these goals through **clustering and concept matching** in FL. At every training round, to avoid interference among the clients, the server clusters the client models representing the same concept and aggregates them. To mitigate catastrophic forgetting, different concept models are trained for each concept through concept matching which occurs differently at the server and the clients. The server concept matching is to match and update the concept model of the previous round with a cluster model. We propose a novel distance-based concept matching algorithm for the server concept matching. This algorithm matches a cluster model with a concept model close in distance, and aligns them to update the concept model in the appropriate gradient descent direction. The client concept matching is to test the concept models from the previous round on the current local data, and to select the one with the lowest loss as the best match. The CM framework does not require the clients to have any knowledge about the concepts. Furthermore, the server does not need any additional information when compared to vanilla FL (i.e., it only requires the model weights from the clients). The CM framework provides flexibility to use a variety of clustering, aggregation, and concept matching algorithms. The framework can evolve as new algorithms are proposed for different applications and models. Our server concept matching algorithm achieves up to 100% effectiveness. This result is grounded in a theorem, in which we proved that with each iteration of gradient descent, the distance between the current model and the previous one decreases. Our algorithm ensures this condition for each concept model by updating it with a matching cluster model. Furthermore, using several datasets, we experimentally

demonstrated the superior performance of CM over the state-of-the-art solutions, its algorithms effectiveness to match concepts of data to model, its resilience when configured with different numbers of concepts, and its feasibility and low overhead on a real IoT device.

1.5 Contributors to this Dissertation

The prototype of FLSys was designed and implemented collaboratively with my colleague Han Hu. The author designed and implemented the mobile-side components, the deep learning model, the communication protocol, and the FL emulation process. Han Hu's contributions are the design of the training protocol, the implementation of the FL simulation process, and the implementation of cloud-side components in FLSys. The high-level design and the data pre-processing steps were designed and implemented collaboratively by the author and Han Hu. In addition, Thinh On and Phung Lai's contribution is evaluating FLSys with differential privacy.

For ZoneFL, the author designed ZoneFL system and ZMS algorithm, implemented the mobile-side training app, improved the server-side components from FLSys, and analyzed the field study. Thinh On designed and implemented ZGD algorithm for ZoneFL, and ran simulations to evaluate ZoneFL. Hessam Mohammadi participated in the initial design of ZoneFL, and some ideas from the initial design was used in ZoneFL. Khang Dang implemented the data collector for ZoneFL.

For a better understanding of the two systems and my contribution, the whole systems are presented in this dissertation, including the parts of my colleagues.

1.6 Structure of the Dissertation

The remainder of this proposal is organized as follows. Chapter 2 provides a review of the literature related to this dissertation. Chapter 3 presents FLSys - an open

ecosystem for FL mobile apps. Chapter 4 presents ZoneFL. Chapter 5 shows Federated Meta-Location Learning for fine-gained location prediction. Chapter 6 presents Complement Sparsification: low-overhead model pruning for FL. Chapter 7 presents FCL using Concept Matching. Chapter 8 concludes the dissertation and discusses the future directions.

CHAPTER 2

LITERATURE REVIEW

This chapter reviews Federated Learning (FL) background, two FL applications with mobile sensing data, and works enhancing FL.

2.1 Federated Learning Background

2.1.1 FL Preliminaries

FL is a multi-round communication protocol between a coordination server and a set of N clients to jointly train a learning model f_θ, where θ is a vector of model parameters (also called weights). The training proceeds in rounds. At each round t the server sends the latest model weights θ_t to a randomly sampled subset of clients S_t. Upon receiving θ_t, each client $u \in S_t$ uses θ_t to train its local model and generates model weights θ_t^u. Client u computes its local gradient $\nabla\theta_t^u = \theta_t^u - \theta_t$, and sends it back to the server. After receiving the local gradients from all the clients in S_t, the server updates the model weights by aggregating all the received local gradients using an aggregation function $\mathcal{G} : \mathbb{R}^{|S_t| \times n} \rightarrow \mathbb{R}^n$, where n is the size of $\nabla\theta_t^u$. The aggregated gradient will be added to θ_t: $\theta_{t+1} = \theta_t + \lambda\mathcal{G}(\{\nabla\theta_t^i\}_{i \in S_t})$, where λ is the server's learning rate. A typical and widely applied aggregation function $\mathcal{G}$ is the weighted averaging, called Federated Averaging (FedAvg) [15].

By joining the FL protocol, clients minimize the average of their loss functions as follows: $\theta^* = \arg\min_\theta \frac{1}{N}\sum_{u=1}^{N} \mathcal{L}_u(\theta)$, where $\mathcal{L}_u$ is the loss function of client u on their local training dataset D_u. $\mathcal{L}_u$ is defined as $\mathcal{L}_u(\theta) = \frac{1}{|D_u|}\sum_{x \in D_u} \mathcal{L}\big(f_\theta(x), y\big)$, where $|D_u|$ denotes the number of data samples in D_u, and $\mathcal{L}$ is a loss function (e.g., cross-entropy) penalizing the mismatch between the predicted values $f_\theta(x)$ of an input x and its associated ground-truth label y.

2.1.2 FL Systems

FL can be categorized into Horizontal FL, Vertical FL, and Federated Transfer Learning (FTL) [16]. In Horizontal FL, data are partitioned by device user Ids, such that users share the same feature space [16]. In Vertical FL, different organizations have a large overlapping user space with different feature spaces. These organizations aim at jointly training a model to predict the same model outcomes, without sharing their data. In FTL, the datasets of these organizations differ in both the user space and the feature space. In Vertical FL and FTL, different organizations need to align their common users and exchange intermediate results by applying encryption techniques [83]. The server cannot just average the gradients, but it needs to minimize a joint loss. At inference stage, the organizations may have to send their individual intermediate results to the server to compute a final result. The systems of these two categories rely on cryptography and their interactions are more complex. Our FLSys focuses on Horizontal FL, with an option for extension to Vertical FL and FTL in the future. For simplicity, we will use FL to indicate Horizontal FL in the rest of our paper.

Table 2.1 shows the comparison between FLSys and other FL systems/frameworks across several features required for an efficient and effective FL system. FLSys is the only system that supports all these features, and it is also the only one that supports third-party apps and efficient mobile sensing data collection. Specifically, FLSys addresses unanswered questions on concurrent training of multiple models for different apps and APIs for third party app developers. Furthermore, unlike all the other systems, FLSys enables models that work with data collected from the phones' sensors, which adds challenges related to efficient and effective data collection.

Among the comparison systems, the FL work done at Google is the best known. However, despite work [1] that describes the conceptual design of a scalable FL system

for mobile devices, Google has not published the implementation and evaluation of an end-to-end FL system to address the features in Table 2.1. Recently, its TensorFlow Lite [84] framework started to support on-device training, but this framework does not attempt to provide any other type of system support required by FL.

Systems such as FATE [85] and FedVision [7], introduce FL architectures based on web-services. They focus on either institutional collaboration or a target application, and they do not have any support for mobile devices. Similarly, Nvidia's FLARE [86] is a domain-agnostic, open-source, and extensible SDK for FL, but it does not support mobile device training. Among the systems supporting mobile devices, Syft [10] offers KotlinSyft for on-device training and provides an FL server, PyGrid, with a web-UI. However, Syft does not address scalability or provides advanced privacy preserving mechanism. FedML [8] shares some goals with FLSys. However, this open source system is still under construction. In addition, FedML focuses more on software engineering aspects, rather than on system aspects such as efficient sensor data collection or scalability. The closest FL system to ours is Flower [87], which provides a high-level FL programming library, employs TensorFlow Lite for on-device training, and evaluates scalablity with a number of embedded edge computing devices. However, this system does not focus on mobile devices and does not provide a solution to support third-party apps or mobile sensing data collection. The evaluation is conducted on embedded edge computing devices instead of real mobile devices. Last but not least, FLSys is the only system designed to provide modular deployment. The policies, algorithms, and functions are implemented at fine granularity. The system can be deployed as interchangeable modules with serverless cloud resources, instead of an always-on server. This makes it easy to both upgrade the system and achieve cost-efficiency when scaling up.

Table 2.1 Comparison of Different FL Frameworks

	TF-Lite	Syft	FLARE	FATE	FedML	Flower	FLSys
On-device training	✓	✓			*	✓	✓
Scalability						✓	✓
Fault-tolerance		✓				✓	✓
Client heterogeneity		✓			✓	✓	✓
Advanced privacy preserving		*	✓	✓		✓	✓
Concurrent third-party app support							✓
Efficient sensor data collection							✓
Modular deployment				✓			✓

(* denotes planned feature)

2.2 Federated Learning Applications with Mobile Sensing Data

2.2.1 Human Activity Recognition

Our HAR model focuses on sensing and classification of physical activities through smart phone sensors. Recent works show that deep learning models are effective in HAR tasks. For example, Ignatov [20] proposed a CNN based model to classify activities with raw 3-axis accelerometer data and statistical features computed from the data. Several works [21,22,26] proposed LSTM-based models and achieved similar performances.

Most research on HAR models uses centralized learning on data collected in controlled lab environments with standardized devices and controlled activities, in which the participants only focus on collecting sensor data with a usually high and fixed sampling rate frequency, i.e., 50Hz or higher. Although there are good publicly available HAR datasets, e.g., WISDM [23], UCI HAR [24], and Opportunity [25], they are not representative for real-life situations. Different from existing works, this paper shows that HAR-Wild over FLSys performs well on the data collected in the wild, which are subject to fluctuating sample rates and non-IID data distribution.

2.2.2 Location Prediction

Early exploration of location prediction adapted Markov Chains and Hidden Markov Models [27]. Conventional machine learning (ML) methods, such as Bayesian networks [33], Support Vector Machines (SVM) [34], and tree-based models [31], were also applied for location prediction. Due to the limited information extraction capability of these models, the performance suffered.

More recently, researchers have started to exploit deep learning (DL) techniques for location prediction by treating it as a time series prediction problem. In a taxi destination prediction competition, de Brébisson et al. [29] tested several DL models, including MLP, LSTM, Bidirectional-RNN and Memory Network. Overall, the best model was Bidirectional-RNN with a time window covering five successive GPS points. In our case, given the need for fine-grained temporal scale, RNN-based methods alone cannot work well because they do not capture information such as road network characteristics and user preferences. Another obstacle to directly adopting RNN-based methods is that the transition from one location to another cannot happen between any two locations [32]. We overcome this by defining a reachable region centered at the current location and bounded by the traveling speed. We also differ in terms of privacy requirements. FMLL uses meta-locations instead of physical locations. In addition to improved privacy, this allows FMLL to easily scale uniformly among all users without losing the speed information, which is difficult when using physical locations.

The trajectory of movement on a map can be naturally processed with CNN-based methods. Lv et al. [30] proposed T-CONV, and beat the performance of Bidirectional-RNN [29] in the taxi destination prediction problem. The method uses trajectory data to mark the visited cells in a grid-like space, but does not incorporate the visit frequencies at specific locations. The CNN component of FMLL, on the other hand, incorporates visit frequency, which helps to improve prediction accuracy. Zhang

et al. [88] treated crowd inflow and outflow of grid cells in a city as a two-channel image-like matrix, and used CNNs for crowd flow prediction. This is a different problem from ours, but we share the ideas of visit frequencies for grid cells, and further extend the idea to represent the output as reachable grid cells.

Recent research [36,37] applied state-of-art DL methods on POI IDs prediction. These works seem close to ours in terms of predicting human mobility. However, their problem definition is completely different, and their models use mechanisms that cannot work well for our problem. In Section 5.5, we adapted them for fine-grained location prediction and evaluated their performance. Section 5.5 will further discuss the reasons for their inferior performance in fine-grained location prediction.

None of the studies discussed so far attempted to provide location privacy. Current privacy-preserving techniques in ML, such as differential privacy (DP), FL, and cryptographic methods could be applied for our problem, but have limitations. DP requires that computations be insensitive to changes in any particular individual's record, thereby restricting data leaks through the results. However, recent studies show record-level DP fails to address information leakage attacks [89]. A study by Graepel et al. [90] demonstrated machine learning on encrypted data using homomorphic encryption, but there are trade-offs regarding computational complexity and prediction accuracy. FL enables learning on the mobile devices without sending the raw data/features to the server. However, recent studies [75] showed user-level privacy leakage against FL by a malicious server, which can exploit the parameters received from the users. FMLL uses FL, but mitigates such attacks by using meta-location, which makes it difficult to identify physical locations.

2.3 Enhancing Federated Learning

2.3.1 Coping with Non-IID Data in FL

A well-reported issue restricting the performance of models trained by FL is non-IID data distribution across users [4, 91]. Different from centralized learning, the datasets among different users may follow different distributions in FL, because of the heterogeneous devices, imbalanced class distribution, different user behaviors, etc. As a result, DL models trained in FL algorithms usually suffer from inferior performance when compared with centralized models [91].

To mitigate the non-IID issue, several algorithms have been proposed [2–6]. In FedProx [2], a regularization is introduced to mitigate the gradient distortion from each device. Sarkar et al. [3] presented a cross-entropy loss to downweigh easy-to-classify examples and focus training on hard-to-classify examples. Verma et al. [6] proposed to estimate the global objective function by averaging different objective functions given a common region of features among users, and keep different objective functions estimated from local users' data in different regions of the feature space. Data augmentation approaches have been proposed [4], including a global data distribution based data augmentation [5]. The federated training of our HAR-Wild and SA models use a uniform data augmentation method, similar to these techniques.

2.3.2 FL Incorporating Differential Privacy

Differential privacy (DP) [92–95] offers a state-of-the-art metric for quantifying privacy when sensitive data are involved, and it is currently deployed by organizations such as Apple, Google, Microsoft, Facebook, and US Census Bureau [96]. An algorithm satisfies DP when adding, removing, or changing one record does not alter its output. The definition of DP was formalized by [93] as follows:

Definition 1 (Differential Privacy [93])**.** A randomized mechanism $\mathcal{M} : \mathcal{D} \rightarrow \mathcal{R}$ with domain $\mathcal{D}$ and range $\mathcal{R}$ fulfills (ε, δ)-differential privacy if for any two neighboring

datasets $(d, d') \in \mathcal{D}$ that differ exactly in a single data sample and for any subset of outputs $S \in \mathcal{R}$, the following condition holds:

$$Pr[\mathcal{M}(d) = S] \leq e^{\varepsilon} Pr[\mathcal{M}(d') = S] + \delta \tag{2.1}$$

where Pr stands for probability, ε is the privacy budget and δ is the probability that ε-differential privacy is broken. The privacy budget ε controls the amount of difference between the probability distributions generated by d and d'. The smaller value of ε, the stronger privacy guarantee.

In this article, we tested FLSys with two well-known DP mechanisms for FL: user-level DP (User-DP) [97] and sample level local DP (LDP) [98–101].

User-DP guarantees to protect clients' participation (membership) information in training the global model. User-DP is implemented by clipping local gradients [102] derived from clients' local training data. Then, DP-preserving noise in added into the aggregation of the clipped local gradients (using federated averaging algorithm [103]).

Definition 2 (User-level Differential Privacy [97])**.** A randomized mechanism $\mathcal{M} : \mathcal{D} \rightarrow \mathcal{R}$ with domain $\mathcal{D}$ and range $\mathcal{R}$ fulfills (ε, δ)-differential privacy at user level if for any two neighboring sets of users (u, u') that differ in exactly one user, $(\mathcal{D}_u, \mathcal{D}_{u'}) \in \mathcal{D}$ and for any subset of outputs $S \in \mathcal{R}$, the following condition holds:

$$Pr[\mathcal{M}(\mathcal{D}_u) = S] \leq e^{\varepsilon} Pr[\mathcal{M}(\mathcal{D}_{u'}) = S] + \delta \tag{2.2}$$

In LDP, one focuses on protecting the legitimate value of a training sample of an individual user. The definition of LDP is as follows:

Definition 3 (Local Differential Privacy)**.** A randomized mechanism $\mathcal{M}$ satisfies (ε, δ)-LDP if for any two inputs (x, x') and for any subset of outputs $S \in Range(\mathcal{M})$, the following condition holds:

$$Pr[\mathcal{M}(x) = S] \leq e^{\varepsilon} Pr[\mathcal{M}(x') = S] + \delta \tag{2.3}$$

2.3.3 Location Embedding in FL

To adapt to user mobility behavior, a naive approach in FL could be to incorporate the user location in the model input [104–106]. However, compared with a model without location input, such an approach increases both the model size and the computation overhead, which leads to extra resource consumption on the mobiles. Different from these approaches, ZoneFL balances the trade-offs between model utility and system scalability by developing novel federated training algorithms seamlessly integrated into a scalable mobile-edge-cloud system architecture. Furthermore, potential attacks by an honest-but-curious server in an FL system that embeds user locations may be able to infer user mobility traces from the model weights. In ZoneFL, such a location privacy breach is more difficult because the fine-grained user location is not embedded in the models.

2.3.4 Clustering and Personalization in FL

As FL being adapted in pervasive computing [107, 108], user clustering has been proposed to improve the model accuracy of traditional FL. Clustering in FL [109, 110] groups clients by the similarity of their local updates and trains the clusters independently. MLMG [111] uses a Multi-Local and Multi-Global model aggregation to train the non-IID user data with clustering methods. Clustered FL [112] performs clustering with geometric properties of the FL loss surface. However, these works have the same scalability issue as traditional FL because they require a central server

to cluster users. Khan et al. [113] propose an FL scheme with a clustering algorithm based on social awareness, which selects cluster heads to avoid a centralized server. In OPS [114], users share their model parameters with a group of trusted friends. One problem with these solutions is that utilizing social relationships to create clusters carries privacy risks.

Although ZoneFL shares the idea of training models over groups of users with clustering approaches, there is no efficient clustering method to group users by their mobility behavior without violating users' location privacy. ZoneFL optimizes models to user mobility behavior and does not require centralized model updates or privacy-sensitive user information. The edge managers do not have access to users' locations; they just know that the user has been in a possibly large zone. Furthermore, ZoneFL provides a solution that can be naturally deployed at the edge for better scalability, which is a further advantage compared to clustering approaches.

Personalized FL can also improve the FL model performance by mitigating the issue of non-independent and identically distributed (non-IID) data, which leads to lower performance in FL compared to centralized learning. Its key idea is to learn a personalized model per user [115]. There are different methods for adapting global models for individual users [116], including adding user context, transfer learning, using personalized layers, knowledge distillation, etc. Ditto [117] leverages global-regularized multi-task learning to provide fairness and robustness through personalization in FL. In the adaptive personalized FL [118], each user trains a local model incorporating certain mixed weights in the global model. Ozkara et al. [119] use quantization and distillation for personalized compression in FL. Although effective, these solutions demand extra computation on mobiles, which may negatively affect their resource consumption. ZoneFL is orthogonal to personalized FL, which can be leveraged in ZoneFL to produce personalized models for each user in each zone.

2.3.5 Model Pruning in FL

Model pruning can be categorized as structured pruning and unstructured pruning. There is a large body of literature on model pruning designed for centralized learning [120–122]. There methods are computationally demanding and require a dataset representing the global data distribution. Therefore, they are not practical in FL, which does not share raw data with the server, and are difficult to use on resource-constrained mobile and IoT devices. Our CS model pruning, on the other hand, is designed for FL on resource-constrained devices. It does not requires a centralized dataset and eliminates explicit fine-tuning for computation efficiency. CS applies unstructured pruning in FL, due to its freedom to update different significant weights over FL training rounds and, thus, achieves better performance.

The recent literature contains several works on model pruning for FL. An online learning approach [123] determines the near-optimal communication and computation trade-off by gradient sparsity. Liu et al. [124] apply model pruning and maximize the convergence rate. PruneFL [78] adapts the model size to find the optimal set of model parameters that learns the "fastest". FL-PQSU [79] is composed of a 3-stage pipeline: structured pruning, weight quantization, and selective updating. Yu et al. [77] present an adaptive pruning scheme, which applies dataset-aware dynamic pruning for inference acceleration. In SubFedAvg [125], the clients use a small subnetwork through pruning. Although most of these works achieve comparable model accuracy with vanilla FL and save some communication when the clients send the local updates to the server, they all impose substantial computation overhead on clients for additional optimizations or recovering the performance loss from pruning. In CS, pruning has very low computation overhead at the clients, as their only task is to remove the weights that were previously non-zero in the global sparse model. This low overhead makes CS practical for resource-constrained devices.

While the works mentioned so far prune model weights, other works choose to remove neurons from the model at the clients. To cope with device heterogeneity, Ordered Dropout (OD) [80] lets the clients train subnetworks of the original network in an ordered fashion. However, OD cannot save any communication from the server to the clients. In FedDrop [81], subnets are randomly generated from the global model at the server using dropout with heterogeneous dropout rates, and the clients only train and transmit the subnets to the server. This work saves communication bidirectionally, but suffers from inferior model accuracy compared to vanilla FL. CS not only reduces the bidirectional communication overhead, but also achieves comparable performance with vanilla FL.

In addition to pruning, there are other methods targeting the overhead in FL. Some works [126, 127] optimize the communication frequency. LotteryFL [128] communicates the personalized lottery networks learnt by the clients. Ozkara et al. [119] use quantization and distillation for personalized compression in FL by manipulating the loss function at the clients. In DGC [129], the clients only send large gradients for aggregation and leave small gradients to accumulate locally until they become large enough. These works cannot enjoy all the benefits of using a sparse model, such as better generalization [55] for a model to maintain good performance on unseen data, and higher robustness to adversarial attacks [56, 57]. Since these methods belong to different classes of model compression, we do not compare them with CS.

2.3.6 Federated Continual Learning

Most of the works on generic FL focus on system design [11, 61], model performance [130–133], privacy [134, 135], and communication and computation overhead [136, 137]. Some works [112, 138–140] cluster the client models in FL. These clustering approaches in FL assume the number of client groups is a constant,

and cannot be applied directly in CL scenarios. All the works mentioned here assume the training data for the clients do not change over time, which limits the applicability in FCL. Our work, on the other hand, focuses on making FL work well in the presence of dynamic changes of the concepts in data.

CL allows learning continuously over time from a stream of data, while avoiding catastrophic forgetting. Recent works addressing CL can be categorized into three families [141]: replay [142,143], regularization [144,145] and parameter isolation [146, 147]. These techniques do not address additional challenges from FL. In addition to its distributed nature, FL also introduces privacy restrictions. For example, FL clients shall not share their task IDs with the server. In addition, even if the clients can learn new concepts well without forgetting the previous ones, the aggregation may sabotage the efforts of the clients when their learning paths diverge due to non-iid data. This phenomenon has been demonstrated experimentally with image data in a recent work [63]. Expanding CL to FL, our work adheres to the FL requirement that the server only accesses the client model weights, and it handles the interference among the clients in FL.

FCL is a new research area that combines FL and CL. FedWeIT [63] and CFeD [70] need to share the task IDs with the server. CDA-FedAvg [64] is not proven to work with concept drift caused by different sets of classes. FedViT [148] is not compatible with models other than ViT. FedPC [66] focuses on P2P FL instead of server-client FL. TARGET [62] is under an impractical assumption that all clients train the same set of classes incrementally over time. Other works [65,67–69,149] do not address the interference among the clients. Unlike prior works, our work tackles catastrophic forgetting and the interference among the clients under more realistic assumptions, such as the clients do not share any additional information with the server beyond the model weights, and the classes can change arbitrarily over time.

2.4 Chapter Summary

This chapter discussed the existing studies related to FL systems, applications, and optimizations. First, we reviewed FL preliminaries, and discussed the available FL systems in literature. Next we discussed the state-of-art works in human activity recognition and location prediction. Finally, we presented related works in literature to enhance FL, including mitigating non-IID, incorporating DP, using location embedding in FL, clustering and personalization in FL, model pruning in FL, and tacking FCL.

CHAPTER 3

FLSYS: TOWARD AN OPEN ECOSYSTEM FOR FEDERATED LEARNING MOBILE APPS

This chapter presents our experience of designing, implementing, and evaluating an end-to-end FL system (FLsys). FLSys is co-designed with a human activity recognition (HAR) model to specifically operate on smart phones that utilize mobile sensing data. The two main challenges for an FL system on phones are concurrent management of multiple FL activities under resource constraints and frequent disconnections due to networking and battery issues. These two challenges are not considered by any existing FL system. To solve them, we propose a novel system architecture that provides (1) a unified system to manage resources on the phone in the presence of multiple models, third-party apps using these models, and data collectors for these models; and (2) an asynchronous protocol to manage the FL process in the presence of disconnections. The FLSys components on smart phones manage training, inference, data collection/preprocessing, and privacy to balance model utility with resource consumption, while tolerating disconnections.

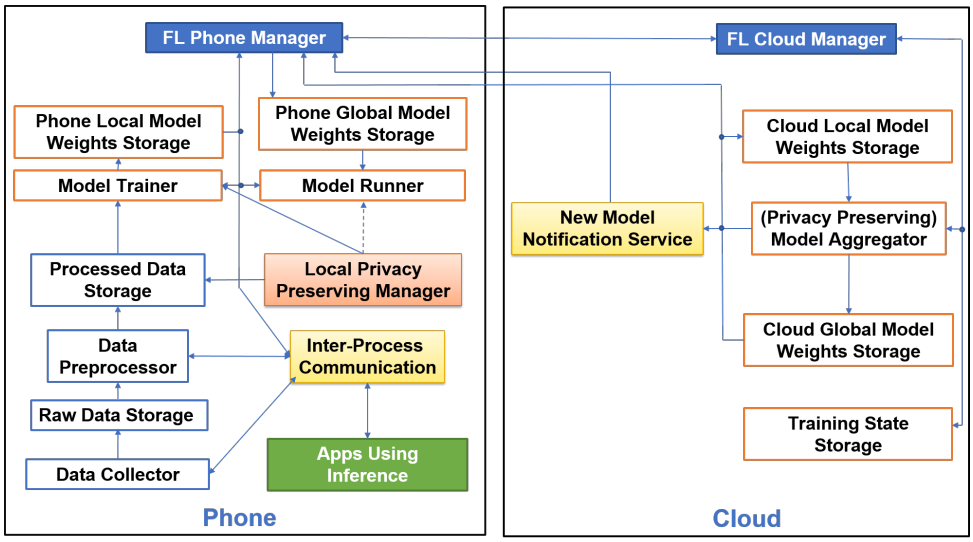
In this chapter, Section 3.1 explains the design of FLSys. Section 3.2 describes its prototype implementation. Section 3.3 presents the HAR model and data. Section 3.4 shows the experimental results. The chapter is summarized in Section 3.5.

3.1 FLSys Design

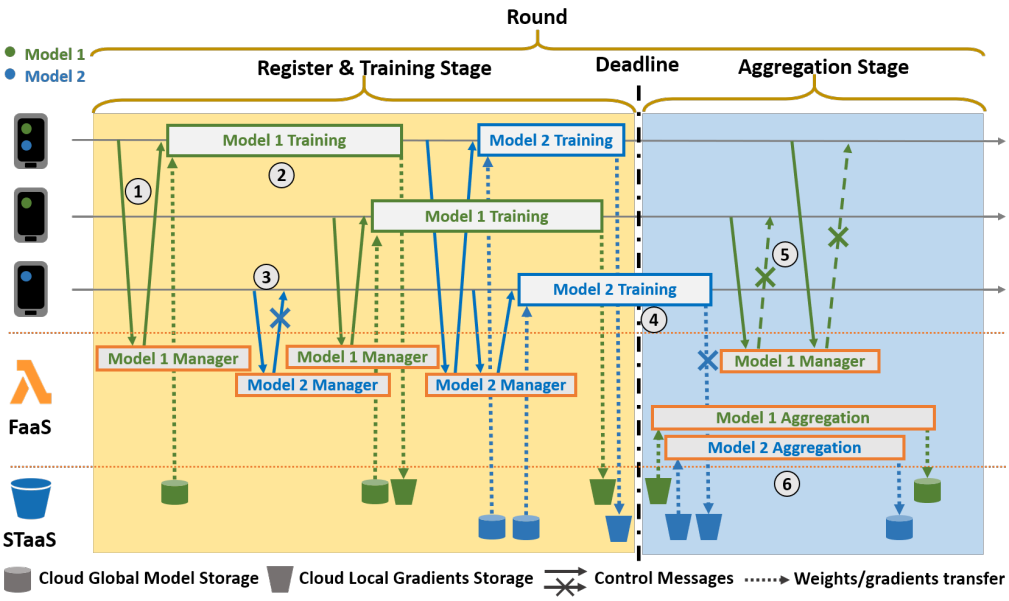
This section presents the design of FLSys. Specifically, it describes the system requirements derived from an application-system co-design, the novel FLSys architecture that addresses these requirements, along with the four operation phases of FLSys, namely data collection and processing, privacy protection, federated training, and inference at the phones.

3.1.1 System Requirements

Our aim is to design and build an FL system that addresses the questions mentioned in Subsection 1.1.1. We use the HAR model, detailed in Section 3.3, to illustrate an entire category of FL models based on mobile sensing data collected in the wild. We extract seven key requirements derived from this model and from other real-world FL applications, such as next word prediction, on-device search query suggestion [13], on-device robotic navigation [150], on-device item ranking [1], object recognition [7], sentiment analysis, etc., and utilize them to guide our FLSys design: (*R1*) Effective data collection: The data collection on the phone must balance resource consumption (e.g., battery) with sampling rates required by different models; (*R2*) Support for advanced privacy preserving mechanisms: Even though FL is privacy-preserving by design, there are still potential privacy issues (e.g., learn user information from the gradients) [151, 152]. Therefore, the system must provide a plugin interface for advanced privacy protection mechanisms, such as local differential privacy; (*R3*) Tolerate phone unavailability during training: Since the phones may sometimes be disconnected from the network or choose not to communicate to save battery power, the interaction between the phones and the cloud must tolerate such unavailability during federated training; (*R4*) Scalability: The cloud-based FL server of our system must be able to scale to large numbers of users in terms of both computation and storage; (*R5*) Model flexibility: The system must support different DL models for different application scenarios and different aggregation functions in the cloud; (*R6*) Support for third-party apps: The system must provide programming support for third party apps to concurrently access different models on the phones, while efficiently managing resource consumption and contention; and (*R7*) Modularity: The system shall not be heavy to deploy, and its policies, algorithms, and functions shall be designed and implemented as interchangeable modules for simple, cost-effective deployment and scalability.



(a) FLSys architecture.



(b) Asynchronous protocol with phone self-selection and multiple models.

Figure 3.1 FLSys architecture and asynchronous protocol.
Typical operations: ① Phone Manager of Client #1 registers with the Cloud Manager of Model 1, which grants registration based on training settings. ② Phone Manager of Client #1 fetches up-to-date global model from a designated storage, trains it with local data, and uploads local gradients to a designated storage. ③ Phone Manager of Client #2 tries to register, but is denied. ④ Phone Manager of Client #2 successfully registers at a later time, but the training misses the deadline, thus its gradients upload is denied. ⑤ Clients #1 and #2 try to register during server aggregation and are denied. ⑥ Each model's Aggregator loads the gradient updates, aggregates them, and saves the aggregated model.

3.1.2 FLSys Overview

FLSys addresses requirements $R1 - R7$ synergistically in a novel system architecture. For some requirements, we propose novel solutions, as no current FL system addresses them, while for others we customize existing solutions for our needs in order to provide a complete design and implementation. Figure 3.1a shows the system architecture, and Figure 3.1b shows the overall process of one training round. These figures emphasize five novel contributions made in FLSys, compared with existing FL systems: **(1)** FLSys allows the phones to self-select for training when they have enough data and resources; **(2)** FLSys has an asynchronous design (Figure 3.1b), in which the server in the cloud tolerates client failures/disconnections and allows clients to join training at any time. **(3)** FLSys supports multiple DL models that can be used concurrently by multiple apps; each phone trains and uses only the models for which it has subscribed; **(4)** FLSys acts as a "central hub" on the phone to manage the training, updating, and access control of FL models used by different apps; and **(5)** FLSys allows apps/models to use different privacy mechanisms that trade model accuracy for privacy guarantees.

These features balance model utility with mobile device constraints and privacy, and can help create an ecosystem of FL models and associated apps. FLSys allows different developers to build FL models/apps and provides a simple way for users to take advantage of these apps, as it offers a unifying system for the development and deployment of FL models and apps that use these models. FLSys acts as a common middleware layer for all these apps and models. The users just need to download/install the apps, and FLSys will take care of downloading/installing the FL models used by the apps, will perform FL training as needed, and will run FL inference on behalf of the apps.

3.1.3 System Architecture

The architecture (Figure 3.1a) has two main components: *(1) FL Phone Manager*, which coordinates the FL activities on the phone; and *(2) FL Cloud Manager*, which coordinates the FL activities in the cloud. These two components work together to support the four phases of the FL operation: data collection and preprocessing, privacy protection, model training and aggregation, and mobile apps using inference. In the following, we describe each phase and explain how the system architecture satisfies the seven system requirements.

Data Collection and Preprocessing. The FL Phone Manager controls the data collection using one or multiple *Data Collectors*. A basic Data Collector is tasked with collecting data from one sensor at a given sampling rate. Such basic Data Collectors could be embedded in more complex ones to collect different types of data at the same time. It is important to have one app that coordinates data collection because having multiple apps collecting overlapping sets of data multiple times is inefficient. Having the FL Phone Manager to coordinate the data collection also simplifies sensor access control.

To satisfy requirement *R1*, FLSys supports on-demand configuration of sensor types, sampling rates, and the period to flush data from memory to storage. Each model informs the FL Phone Manager of the type of data and sampling rate it needs. In this way, the FL Phone Manager knows which Data Collectors to invoke and which sampling rates are needed. The FL Phone Manager balances sensing accuracy (i.e., high sampling rate) with resource consumption.

To regulate and keep such balance aligned with the user experience, the FLSys has three features: (1) include several built-in sampling rate settings, with empirical values from our experience; and (2) collect key statistics of the data collection (e.g., CPU time consumed, battery life impact, etc.) and show them to the user, upon

request; and (3) provide global level controls for the user to adjust the data collection behaviors, should the user feel that their experience is impacted by data collection.

The Data Collectors store the sensed data in the *Raw Data Storage* and inform the FL Phone Manager each time new data is added to the Raw Data Storage. For efficiency, the Data Collectors can buffer a certain amount of sensed data in memory before committing it to the storage. The FL Phone Manager can dynamically reconfigure the data flushing period that defines when the data is written to storage. Data Collectors set this data flushing period. Some models may use the raw data directly, while others may require additional processing. The FL Phone Manager decides when to invoke the model-specific *Data Processors*, which will store the data in the *Processed Data Storage*. This is a matter of policy and can be done any time new data is available in the Raw Storage Data or at a regular interval. The only constraint is to have all the data preprocessed before a new local model training operation.

To deal with the problem of non-IID data distribution, described in Subsection 2.3.1, the *Data Preprocessor* can augment the data collected locally on the device with data received from the cloud. The augmentation dataset is model-specific and mitigates the distortion the data classes by providing data samples for classes with not enough data. When the users join FLSys for a new model, their phones receive an augmentation dataset from the *FL Cloud Manager* for models that use data augmentation techniques.

Privacy Threat Model and Protection. To satisfy requirement *R2*, the *Local Privacy Preserving Manager* delivers advanced privacy protection mechanisms on the phone component of FLSys. It is designed to work with different privacy mechanisms, which are available on a per-model basis.

Threat Model. In this study, we focus on defending against privacy inference attacks from an honest-but-curious server, which can attempt to infer clients' local

training data. Note that the server knows the identity of the clients to coordinate the FL training. The server may try to extract the clients' local data or infer membership information of specific clients' training data samples by using training data extraction attacks [153] and membership inference attacks [154–156], respectively, via observing the clients' local gradients. Third-party apps on the phones, which may or may not use FLSys, may act maliciously by trying to access the model data or performing inference attacks, etc. There are many OS-based, programming language-based, and networking-based approaches that can prevent or alleviate these issues. All these solutions can be applied outside of FLSys.

Defenses. An effective way to protect clients' local training data against an honest-but-curious server is to use local differential privacy (LDP) [101], specifically to preserve ϵ-LDP in FL [157]. LDP provides rigorous privacy protection, without computational overhead, compared with other techniques such as secure multi-party computation [158] and homomorphic encryption [96]. Meanwhile, anonymizers (shuffler [159], faking source IP, VPN, Proxy, mixnets, etc. [157]) could be compromised or could collude with the server to extract sensitive information from local gradients [160]. This introduces extra privacy risks for the clients' local training data. In addition, it is challenging for dimension reduction-based privacy-preserving techniques to achieve good utility under rigorous privacy guarantees with complex models and tasks [161].

Therefore, our system supports existing LDP-preserving approaches in FL, which are currently the most suitable solutions. Existing LDP-preserving approaches in FL can be divided into two categories: **(1)** Clients add noise to local gradients to protect the values of the local gradients [162]; and **(2)** Clients add noise to each training sample to protect the value of each training sample [163], and then they use these perturbed samples to derive local gradients. For both approaches, the clients send the LDP-preserved local gradients to the server for model updates. Our system

further supports *User-level DP* (*User-DP*) [97] to protect the membership information of clients against inference attacks.

These supported mechanisms [97] use DP to provide different levels of privacy protection. Within a DP budget allocated to a given privacy mechanism, the global model converges without an undue cost of model utility. In *User-DP*, the aggregated gradients at the FL Cloud Manager are perturbed to protect clients' participation (membership) information in training the global model (Definition 2, Subsection 2.3.2). In *LDP*, every training sample is perturbed under LDP ensuring that the legitimate value of the training sample is protected against being inferred by the server through observing local gradients (Definition 3, Subsection 2.3.2). As these two mechanisms illustrate, the *Model Aggregator* in the cloud may (e.g., User-DP) or may not (e.g., LDP) apply privacy preserving mechanisms. Generally speaking, privacy mechanisms in FLSys are handled by the *Local Privacy Preserving Manager* on the phones, with potential collaboration from the *Model Aggregator* in the cloud.

Federated Training. To satisfy requirement *R3*, we make two design decisions. First, FLSys allows the phones to self-select for training when they have enough data and resources. This is different from traditional FL architectures [1], where the server selects the phones to participate in training, which may not be available or may not have enough data or resources for training. Second, in FLSys, the communication between the phones and the cloud is asynchronous to cope with phone disconnections. The software at the cloud side is designed to tolerate missing messages from the phones. Overall, FLSys reduces communication overhead and increases client utility, at the expense of less control in the client sampling process, compared to [1].

In order to use a given model on the phone, the FL Phone Manager first registers the phone with the *FL Cloud Manager*. If the phone model and mobile OS are known to work with the model, the FL Cloud Manager registers the phone with the *New Model Notification Service*, which works as a Publish-Subscribe cloud service, and

returns the subscription to the phone. This subscription allows the phone to receive asynchronous notifications when a new global model is available for download. The FL Phone Manager downloads the model at a time determined based on the model usage frequency and power settings.

The training for each model is done in rounds. The FL Cloud Manager decides the duration of a round, based on preferences associated with each model. For example, the server may start a new aggregation (i.e., by invoking the *Model Aggregator* for a certain model) when a given time interval has passed or when a certain number of local training updates have been received from the phones. The FL Phone Manager decides when to participate in training. This decision is done based on local policies that attempt to balance inference accuracy, the amount of input data for training, and the resources consumed during training. The intention to participate in training for a given model is conveyed by a message sent to the FL Cloud Manager. Based on the model preferences (e.g., amount of data, and the number of users in a training round), the server may decide to ask the phone to train for the model and to provide the FL Phone Manager with a URL to upload the results in the *Cloud Local Gradients Storage*. If there is a deadline for participation in the round, the FL Cloud Manager lets the FL Phone Manager know about it.

The FL Phone Manager invokes the *Model Trainer* for the given model and passes as parameter the location of the data in the Processed Data Storage. After the training is done, the Model Trainer stores the newly computed gradients in the *Phone Local Gradients Storage*. The FL Phone Manager decides when to upload these gradients to the Cloud Local Gradients Storage. The FL Cloud Manager will invoke the Model Aggregator for the model when the duration for the round expires or when enough updates have been uploaded. The Model Aggregator reads the updates from the Cloud Local Gradients Storage, computes the aggregated weights, and stores them in the *Cloud Global Model Weights Storage*. The intermediate training state is stored

in the *Training State Storage* to provide lower I/O latency compared with the other types of cloud storage in our design. This is because FLSys needs frequent access to these data during training. Then, the Model Aggregator sends a notification via the New Model Notification Service to let the phones know that a new model version is available.

The cloud-side system satisfies requirement *R4*, as it can scale to large numbers of users due to its modular design that decouples computation, communication, storage, and notification services. The cloud elasticity features of each service allow different services to scale up or down according to the workload.

As we observe from the architecture, each model is managed individually by FLSys, and multiple models can co-exist both at the phones and the cloud. In the cloud, different models use independent cloud resources, which can be scaled independently. On the phone, independent model trainers and inference runners are responsible for different applications. The cloud contains all the models in the system, while each phone contains only the models for which it has subscribed. This modular design allows our system to satisfy requirement *R5*.

Mobile Apps Using Inference. We decouple mobile apps that need inference on the phones from the models that provide the inference. This allows an app to use multiple models, while the same model can be used by multiple apps. FLSys provides an API and a library that can be used by third-party app developers to perform inference using DL models on the phone. In this way, the system architecture satisfies requirement *R6*. When an app needs inference from a model, it sends a request to the FL Phone Manager using one of the OS IPC mechanisms. The FL Phone Manager then generates the input for the inference from the data stored in the Processed Data Storage or the Raw Data Storage, and then invokes the *Model Runner* with this input. The Model Runner sends the result to the App using IPC. When possible, the FL

Phone Manager re-uses preprocessed data to reduce resource consumption or performs one inference for several applications that invoke the same model concurrently.

Model Concurrency. Given the design of FLSys, both the FL Phone Manager and the FL Cloud Manager are able to handle multiple models concurrently. However, the meanings of concurrency are slightly different for each side. FL Cloud Manager needs to handle the aggregation of all models that are registered with it. Also there is the need to communicate to a potentially large number of clients for each model at the same time. FLSys handles this concurrency through services provided by the underlying cloud platform, which support concurrency by design. FLSys just needs to orchestrate the invocation of these services. The FL Phone Manager needs to handle concurrent training and inference. Our preliminary experiments on smart phones show parallel training of multiple models is very slow due to resource contention. It also affects the user experience on the phones. Therefore, we decided to train models sequentially. The FL Phone Manager can request to participate in training rounds for multiple models concurrently, but it locally decides a sequential order in which to train these models, based on parameters such as frequency of model usage by apps, the training round deadlines, and historical training latency for each model. Finally, the inference requests from the apps are executed as soon as they are received to maintain good user experience.

System Modularity. FLSys components are designed and implemented at fine granularity as interchangeable modules for different policies and algorithms to satisfy requirement *R7*. This design makes it easy to deploy different data collection modules, DP-based privacy preserving mechanisms, model trainers at the clients (with different optimizers or loss functions), and aggregation functions at the server. Furthermore, new models can be added on-demand, based on the apps that need them. This modular design can be readily deployed in a serverless manner in the cloud, which

leads to improved scalability (i.e., scale up only the components that are overloaded) and cost-efficiency (i.e., no need to run always-on servers).

3.2 Prototype Implementation

We implemented an end-to-end FLSys prototype in Android and AWS cloud, which have been chosen because they are the market leaders for mobile OSs and cloud platforms, respectively. However, the FLSys design is general and it can be implemented in other mobile OSs and cloud platforms. The prototype implements all of the components described in the system architecture (Figure 3.1a). This section reviews the implementation technologies, the reasons for selecting them, and then focuses on the Android implementation and the AWS implementation of FLSys.

3.2.1 Implementation Technologies

Deep Learning Framework. We chose Deep Learning for Java (DL4J) as the underlying framework for the on-device DL-related operations (i.e., training and model execution) because it was the only mature framework that supported model training on Android devices until very recently, when TensorFlow Lite [84] and KotlinSyft [10] became available for on-device training. While the Model Aggregator in the cloud could be implemented using other DL technologies, for consistency, we implement it in DL4J as well. The models are stored as zipped JSON and bin files in folders on the phone and in AWS S3 buckets in the cloud.

On-device Communication. For IPC among Android apps/services, we use Android Bound Service and Android Intent. A bound service can efficiently serve another application component because it does not run in the background indefinitely. Through IPC, the FL Phone Manager can provide third-party apps with an interface to request inference results without revealing the model or the data. Furthermore, it can communicate with the Data Collector.

Cloud Platform and Services. We opt to utilize the Function-as-a-service (FaaS) architecture for our cloud computation. The core cloud components of FLSys are implemented and deployed as AWS Lambda functions [164]. We decided to choose FaaS for our implementation for five reasons. First, it matches our asynchronous, event-based design, as Lambda functions are triggered by events. Second, it provides fine-grained scalability at the function level; therefore leading to less resource consumption in the cloud. Furthermore, computation and storage are scaled automatically and independently by the cloud platform. Third, unlike other cloud platforms, it does not require running virtual machines when no computation is necessary; this saves additional resources and reduces cost. Fourth, FaaS simplifies the development and deployment of our prototype because it does not require software installation, system configuration, etc. Fifth, different functions can be implemented in different programming languages making the implementation even more flexible.

Lambda functions are triggered in different ways in our prototype. We use the AWS API Gateway to define and deploy HTTP and REST APIs. For instance, we create a REST API to relay clients' requests to participate in the FL training to the Lambda function that handles these requests. We also use the AWS EventBridge to define rules to trigger and filter events for Lambda functions.

FLSys uses a number of cloud services for storage, authentication, and publish-subscribe communication. For model storage, validation datasets, and FL Cloud Manager configuration files, we use AWS S3, which offers a reliable and cost-effective solution for data accessed infrequently. More importantly, AWS S3 buckets can be accessed directly by phones, which simplifies the asynchronous communication in FLSys. To authenticate clients and allow them to upload and download models from the AWS S3, FLSys uses Identity Pool in AWS Cognito. To store data that is accessed frequently, such as training round states and model states, we use AWS DynamoDB, a reliable NoSQL database. AWS SNS is utilized in conjunction with the Google

FCM to notify clients when newly trained models are ready. The use of a Google Cloud service in our AWS implementation was necessary in order to push notifications directly to apps on the phones when a new global model is ready in the cloud.

3.2.2 Phone Implementation

The phone implementation (left-side of Figure 3.1a) consists of three apps: a FL Phone Manager, a HAR Data Collector, and a Testing App used to test model inference.

Data Collector. We implemented a HAR Data Collector app designed for long-term and battery efficient data collection. This Data Collector was implemented as an app that can be used independent of FLSys, but for better efficiency, the Data Collectors can be implemented as modules of the FL Phone Manager. To that end, sensor values are not collected at an enforced fixed high frequency, but are instead collected independently through Android listeners whose actual frequency is variable, determined by the underlying OS. This is appropriate for data collection in the wild. In our experience, this tends to be much friendlier to the performance and battery life of the user devices, lowering the risk that a user abandons FLSys prematurely due to concerns about how it is affecting their device resources. Furthermore, users are given the option to pause or stop data collection of all or a subset of sensors in case they have resource consumption or privacy concerns. For simplicity, the raw data and the processed data are stored as files.

FL Phone Manager. The FL Phone Manager app decides to initiate an on-device training round based on evaluating a Ready To Config policy (RTCp). We implemented a simple policy to check if the phone is charging and is connected to the network before declaring its availability for training. If yes, it sends a Ready To Config message (RTCm) to the FL Cloud Manager. RTCm is implemented as an HTTP request with JSON payload and is sent to a REST API URL in AWS. The

FL Cloud Manger either accepts or denies the phone's participation in this training round, based on a simple Accept/Deny for Training policy (A/DFTp) that checks the phone model and client identity.

The phone is accepted for a round of training when it receives an Accept For Training message (AFTm). AFTm contains the information of the AWS S3 locations from where to download the latest global model weights and where to upload the local gradients. The message also contains the deadline for this training round's completion. The FL Phone Manager evaluates a Start To Train policy (STTp) based on the available device resources and the round's deadline to determine whether to actually perform the on-device training for this round or not.

The FL Phone Manager will create the corresponding Model Trainer if it decides to train. The Model Trainer is implemented with Android native *AsyncTask* class to ensure the trainer is not terminated by Android, even when the app is idle. *AsyncTask* also enables multiple trainers to train in the background. Once the training is complete, the Model Trainer uploads the local gradients to the corresponding AWS S3 location.

Model inference is implemented as a background service with Android Interface Definition Language (AIDL), and it gets inference requests from third-party apps. When such a request is received, the FL Phone Manager uses the current sensor data from the Data Collector as input for the model, runs the inference, and responds to the third-party apps with the inference results.

Testing App. We implemented a simple testing App to test model inference. The App uses *AidlConnection* to interface with the FL Phone Manager. Let us note that the App itself does not access any data or model.

3.2.3 Cloud Implementation

The cloud implementation (right-side of Figure 3.1a) consists of two main components: FL Cloud Manager and Model Aggregator.

FL Cloud Manager. The FL Cloud Manager is implemented as a series of Lambda Functions (FaaS service in AWS). When starting a training round, it reads a configuration file and determines the deadline for the round (i.e., the time when the round must finish). During the period between the start time and the deadline, the FL Cloud Manager accepts or denies clients' requests for training (RTCm). When the deadline is reached, the FL Cloud Manager executes the Model Aggregator according to the Start for Aggregation policy (SFAp). The current policy checks if enough clients have submitted their local gradients in the AWS S3 (a configurable parameter). Then, the Lambda function implementing the FL CLoud Manager schedules an event for itself to perform the next training round and terminate. The training process stops when the pre-defined number of rounds is achieved, or the desired performance (model accuracy) is achieved, if the model developers provided a validation dataset.

Model Aggregator. For implementation simplicity, the Model Aggregator uses the federated average technique [165], with the assumption that each client contributes equally to the global model in each training round. When it is invoked, it loads the uploaded local gradients, and aggregates their gradients to the global model of this round. Once the global model is updated, the Model Aggregator invokes AWS SNS to notify clients that they can download the newly aggregated model. Note that the Model Aggregator is called dynamically through reflection, such that different aggregation functions can be dynamically swapped.

3.2.4 Asynchronous Federate Averaging Implementation

Algorithm 1 shows the pseudo-code of our asynchronous federated averaging process. The algorithm consist of three procedures, which execute asynchronously.

Algorithm 1 AsyncFedAveraging

```
1: procedure ClientLoop
2:     while true do
3:         readyToConfig ← evaluateReadyToConfigPolicy(powerState, wifiState,...)
4:         if readyToConfig then
5:             response ← sendRTCm( )
6:             if response == "AFT" then
7:                 B ← sampling(D_L)
8:                 θ_l ← θ^t
9:                 for batch b ∈ B do
10:                    θ_l ← θ_l − η∇L(θ_l; b)
11:                Δ_l ← θ_l − θ^t
12:                uploadClientGradients(Δ_l)
13: procedure ServerRTCmHandler(RTCm)
14:    if evaluateAcceptForTrainingPolicy(RTCm) then
15:        returnResponse("AFT")
16:    else
17:        returnResponse("DFT")
18: procedure ServerLoop
19:    deadlineTriggered ← false
20:    setupDeadline( ) (deadlineTriggered ← true when triggered)
21:    while true do
22:        if deadlineTriggered then
23:            if evaluateStartForAggregationPolicy( ) then
24:                {Δ_1, ...Δ_k} ← loadClientGradients( )
25:                Δ^t = (Σ_k Δ_k)/k
26:                θ^{t+1} ← θ^t + γΔ^t
27:                if isRoundAcceptable( ) then
28:                    acceptRound(θ^{t+1})
29:                else
30:                    abortRound( )
31:            else
32:                abortRound( )
33:            if evaluateStartNewRoundPolicy( ) then
34:                startNewRound( )
35:                deadlineTriggered ← false
36:                setupDeadline( )
37:            else
38:                stopTraining( )
39:        else
40:            wait( )
```

"ClientLoop" (lines 1-12) runs at clients and executes a round of training (lines 7-12), if the phone self-selects for training and the cloud accepts it (lines 1-6). "ServerRTCmHandler" (lines 13-17) is a part of the FL Cloud Manager and decides whether a phone is accepted for training. "ServerLoop" (lines 18-40) also runs at the

FL Cloud Manager. It performs the aggregation of local gradients and controls the progression of training. The clients participating in a training round must submit their local gradients before the deadline for the round expires. When the deadline comes, the procedure first evaluates the Start for Aggregation policy, which checks whether there are enough local gradient updates in order to preform aggregation. If yes, the aggregation is preformed (line 24-26); if not, this round is aborted, but the uploaded gradient updates will be carried to the next round. After aggregation, the procedure may check against pre-defined conditions to decide whether this aggregation outcome should be accepted or not (lines 27-30). Finally, the procedure checks if a new round should be started by evaluating the Start New Round policy. If a new round is to be started, a new deadline will be set (lines 33-36). Otherwise, the procedure terminates.

3.2.5 FLSys Setup Workflow

By design, FLSys acts as a service provider that handles multiple FL models with minimum input from the users. The setup procedures for FLSys are divided into two stages. The first stage involves the FL Cloud Manager and the app developers, without user involvement. The second stage involves the FL Phone Manager and the mobile apps that use FL models, and it requires minimum user involvement. The FL Cloud Manager is deployed before the first stage, and the FL Phone Manager should be installed on the user's device before the second stage. To illustrate these stages, let us briefly explain the setup workflow using the HAR app as an example.

In the first stage, the developers of the HAR app need to register the model with the FL Cloud Manager. The app developers need to provide the FL model to be trained and the training plan (e.g., training frequency, number of rounds, number of participants in a round, etc.) to register the app. The model can be developed by the app developers or by a third party. After registration, a unique key for the

authentication between the app and the FL Phone Manager in the second stage will be provided.

The second stage is typically triggered during the installation process of the HAR app on the user's device. The app will communicate with the FL Phone Manager and authenticate itself using the aforementioned unique key. Once the app is successfully authenticated, the FL Phone Manager will perform a series of operations and eventually become ready to serve the FL model for the app. These operations include: (1) Register the phone with the FL Cloud Manager; (2) Set up communication channels with the app; (3) If the model does not exist on the phone, download the model specified by the app and the training plan from the FL Cloud Manager; If the model already exists on the phone, establish the connection between the app and that model; and (4) Set up the local training schedule and notify the user. After the second stage, the FL model that the HAR app needs is installed on the phone, ready for inference and training. The training plan can be adjusted by the developers through the FL Cloud Manager. User-experience related parameters can be adjusted by the user through the FL Phone Manager.

3.3 HAR-Wild: Data, Model, and Training

Human Activity Recognition (HAR) generally refers to the task of identifying and classifying people's physical activities and behaviors, through various types of sensors. We focus on HAR through smart phone sensors. Recent works show that deep learning models are effective in HAR tasks. For example, Ignatov [20] proposed a CNN-based model to classify activities with raw 3-axis accelerometer data and associated statistical features. Several works [21,22,26] proposed LSTM-based models and achieved similar performance. Most research on HAR models uses centralized learning on data collected in controlled lab environments with standardized devices and controlled activities [23–25]. Our FLSys with HAR-Wild performs well on the

data from the wild which are subject to fluctuated sampling rate and non-IID data distribution. Different from existing works, our goal is to have an end-to-end study on HAR under FL. We collected sensor data in the wild, built a HAR-Wild model, and implemented an FL training algorithm that works well for data collected in the wild. Most importantly, we used this application to derive the requirements for FLSys, described in Subsection 3.1.1.

We co-designed FLSys with a HAR model, which was used to extract the main requirements for FLSys and, then, to demonstrate the efficiency and effectiveness of FLSys. To show that FLSys works with different concurrent models, we also implemented and evaluated a sentiment analysis (SA) model, as described in Section 3.4. In this section, we describe the HAR dataset, our HAR-Wild model, and its training algorithm using data augmentation to deal with non-IID data in the wild.

3.3.1 Data Collection

Although there are good HAR datasets publicly available, e.g., WISDM [23], UCI HAR [24], they are not representative for real-life situations because they were collected in rigorously controlled environments on standardized devices and controlled activities, in which the participants only focused on collecting sensor data with a usually high and fixed sampling rate frequency, i.e., 50Hz or higher. Thus, given our goal to test FLSys with data collected in the wild, we collect smartphone sensor data (Table 3.1) "in the wild" from university students through our Data Collector app. as subjects for the following reasons: **(1)** University students have relatively good access to the smartphones and related technologies; **(2)** University students should be more credible and easier to be motivated than other sources (e.g., recruiting test subjects on crowd-sourcing websites); and **(3)** It will be easier for our team to recruit and distribute rewards to students. we launched two sensor data collection runs at two

universities for 4 months. We asked the participants to allow the app to collect data in the background and label some data themselves. A total of 116 participants with valid data were recorded after the two data collection runs. The types of data that the app can collect are listed in Table 3.1. In Table 3.2, we show the total number of minutes of data collected from the four types of sensors that collects continuous (relatively continuous compared with other sensors that collect data points at a much lower rate or at certain events) data sessions. We can see that only the Accelerometer collected a meaningful amount of data at the sampling rate higher than 5Hz. Table 3.3 shows the number of data points collected from the rest of the sensors. Note that the Pressure sensor and the Light sensor collect data points at regular (but low) rates, while other sensors at certain events.

Table 3.1 Types of Sensor Data Collected

Sensor	Type	Sampling rate	Availability	Description
Accelerometer	Motion	5Hz ∼50Hz	114/116	Measures the linear acceleration (including gravity)
Gyroscope	Motion	5Hz	114/116	Measures the angular speed
Rotation	Motion	5Hz	114/116	Provides the rotation vector component
GPS	Position	Variable	110/116	Typical GPS data
Magnetometer	Position	5Hz	113/116	Measures the geomagnetic field strength
Pressure	Environment	5Hz	83/116	Measures ambient air pressure
Light	Environment	Variable	110/116	Measures ambient light level (illumination)
Battery log	Phone state	Variable	116/116	Measures the smartphone's battery level
Proximity	Phone state	Variable	109/116	Measures the distance between the screen and an object
Power state	Phone state	Variable	115/116	Different values indicating different state/event of the smartphone
Call log	User data		98/116	Logs the calls that the user had
SMS log	User data		98/116	Logs the short messages (SMS) that the user had

*The sampling rate is estimated from observation, actual values vary
**The availability means that out of the 116 participants, how many of them provided this data type

Table 3.2 Total Time of Sensor Sessions (Continuous Sensors)

Sensor	Total_time	60hz_up	50hz	40hz	20hz	5hz	The_rest
Accelerometer	2278050.17	120056.98	241932.44	642.59	69136.15	1012894.06	833387.95
Gyroscope	2273397.28	110149.8	15448.17	7.39	31354.51	1795766.05	320671.36
Rotation	2290372.31	23073.62	71003.64	441.01	33341.33	2061158.39	101354.32
Magnetometer	2287446.51	870.32	4058.21	0	23.01	2193983.48	88511.49

*In a number of minutes
**Only count data sessions at least 30 seconds long

Table 3.3 Total Number of Data Samples

Sensor	Total number of data samples
gps	1251970
pressure	971782334
light	380064951
batteryLog	7721722
proximity	3280332
powerState	794895
callLog	11362
textsLog	204728

The data collection was approved by the IRBs at both universities. Regarding labels of activities, we provide a simple interface in our Data Collector app to quickly select the appropriate labels. We summarize the labels into three categories: **(1)** Physical activity; **(2)** Phone position; and **(3)** User location. The reported activities cover a vast majority of daily activities that are of research interests. Each category has several pre-defined sub-types for users to choose from. The details of all labels can be seen in Table 3.4. Users can choose the labeling to last either for a defined time length (e.g., 5 minutes, 10 minutes, or 20 minutes) or for an unlimited time (i.e., in this case, users need to remember to turn off the label when the activity is done). The phones were naturally heterogeneous, and the daily-life activities were not constrained by our experiments.

Therefore, we collected a novel HAR dataset in the wild that is different from the existing datasets in the following three aspects: **(1)** The sensors' sampling rates vary from time to time and from user to user, due to battery constrains, device heterogeneity, and usage differences; **(2)** The same basic activity will generate different signals since different users will have different habits of carrying smart phones; **(3)** Label distributions are not just biased, but vary significantly among users.

Table 3.4 Labels and the Total Number of Minutes Collected for Each Label

Label	Total minutes	Label	Total minutes
sitting	862544.50	table	864904.00
walking	158087.40	mounted	49440.29
driving	38013.98	strap	2485.22
lying	488596.70	vehicle_car	72121.55
cycling	2589.50	vehicle_train/subway	355.24
workout_gym	3649.47	vehicle_plane	0.00
workout_running	2212.69	vehicle_bus/shuttle	0.00
workout_others	16500.13	vehicle_motorcycle	88.84
workout_recreational	0.00	home	1171968.00
palm	511092.20	outside	41886.80
bag	6446.19	travel	14094.12
pocket	99557.67	elevator	924.09
		office	17562.08

3.3.2 Data Processing

For the purpose of model design and evaluation, we selected a subset of representative labels and divided them into five activity classes: walking, sitting (not in vehicle), in-vehicle, cycling, and workout/running.

Although data from multiple sensors was collected (Table 3.1), we found that by using accelerometer data our HAR-Wild model can achieve a comparable results with several baseline approaches. Following common practice [20–22,26], the accelerometer data are processed into data segments of shape $[3, 100]$, indicating 100 data points of 3 axis: x, y, and z.

To cope with inconsistent labels caused by the sampling rates in our Data Collector, we design the data processing with three objectives in mind: **(1)** Fix or mitigate the errors and imperfections in the raw sensor data, including gaps, repetitive samples and fluctuating sampling rates; **(2)** Identify and remove inconsistency in the labels; and **(3)** Convert the raw data into short segments of data points with shape $[3, 100]$, that are the basic input data units for DL models, where each data segment consists of 100 3-axis data points of Accelerometer data. The processing steps are as follows:

(1) Any duplicated data points are merged by taking the average of their sensor values;

(2) Using 300 milliseconds as the threshold, continuous data sessions are identified and separated by breaking up the data sequences at any gap that is larger than the threshold;

(3) Data sessions that have unstable or unsuitable sampling rates are filtered out. We keep the data sessions that have a stable sampling rate of 5Hz, 10Hz, 20Hz, or 50Hz;

(4) The label sessions that are associated with each data session (if any) are identified from the raw labels. Note that the label sessions are also filtered with the following two criteria to ensure good quality: (a) The first 10 seconds and the last 10 seconds of each label session are trimmed, due to the fact that users were likely operating the phone during these time periods; (b) Any label session longer than 30 minutes is trimmed down to 30 minutes, in order to mitigate the potential inaccurate labels due to users' negligence (forgot to turn off labeling);

(5) We sample data segments at the size of 100 data points with sliding windows. Different overlapping percentages were used for different classes and different sampling rates. The majority classes have 25% overlapping to reduce the number of data segments, while the minority classes have up to 90% overlapping to increase the available data segments. We sample 15% of data for testing, while the rest are used for training (Table 3.5);

(6) All the sensor values are normalized with z-score using the mean and standard deviation computed from the training dataset. This step is a common practice that helps to stabilize the gradients during training;

(7) Finally, we compute the mean-variance on each data segment with a rolling window (window size = 20), then filter out any data segment that has a variance value less than the threshold of 0.001. This step filters out the "flat" data segments that are labeling errors.

Data Normalization. In our models, the accelerometer data is normalized as $x \in [-1, 1]^3$ to achieve better model utility. However, unlike image data which is naturally bounded to a fixed range, the acceleration values are not explicitly bounded. Thus, the values must be clipped and the trade-off between keeping details in the data (extreme values) and having a suitable range our models to work needs to be made. We normalize the accelerometer data with the following steps: We compute the mean and variance of each axis (i.e., X, Y, and Z) using only training data to avoid information leakage from the training phase to the testing phase. Then, both training and testing data are normalized with z-score, based on the mean and variance computed from training data. Based on these results, we choose to clip the values in between $[min, max] = [-2, 2]$ for each axis, which covers at least 90% of possible

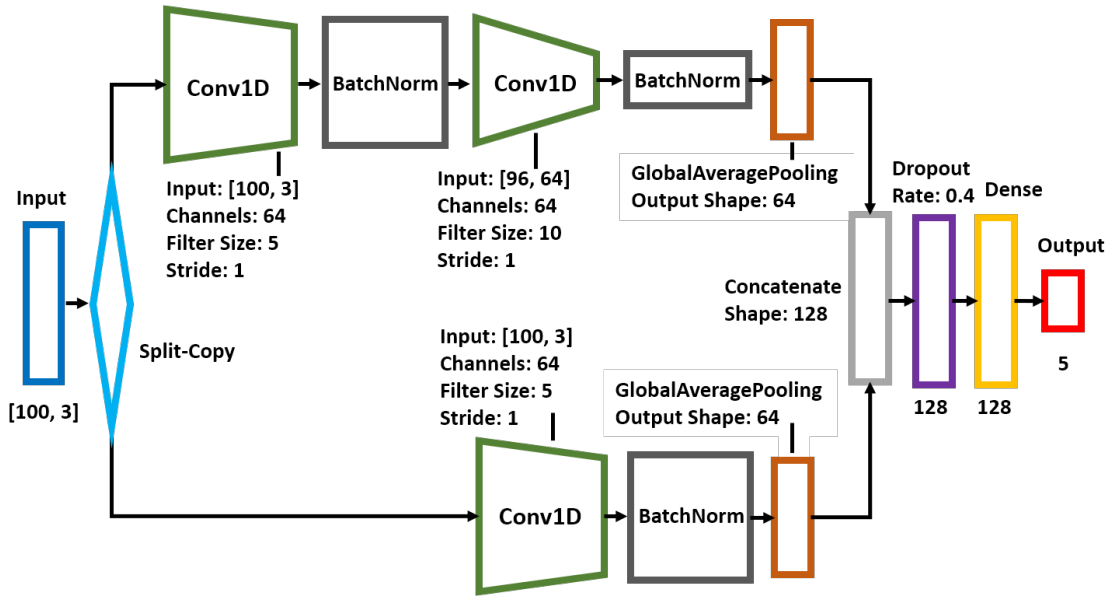


Figure 3.2 HAR-Wild model architecture.

data values. Finally, all values are linearly scaled to $[-1, 1]$ to finish the normalization process:

$$x = 2 \times [\frac{x - min}{max - min} - 1/2] \tag{3.1}$$

3.3.3 Model Design

The design of our HAR-Wild model has two requirements: low computational complexity and small memory footprint. Satisfying these requirements ensures the model can work efficiently on resource-constrained phones. Figure 3.2 shows our model architecture. For low computation complexity, HAR-Wild is based on CNN (instead of RNN, e.g., LSTM) and tailored to work well on mobile devices. In addition, instead of using data from multiple sensors, HAR-Wild can achieve comparable results with several baseline approaches by using only accelerometer data, which makes the training faster.

The accelerometer data are processed into data segments of shape $[3, 100]$, indicating 100 data points of 3 axis: X, Y, and Z. We leverage the recipe of ResNet

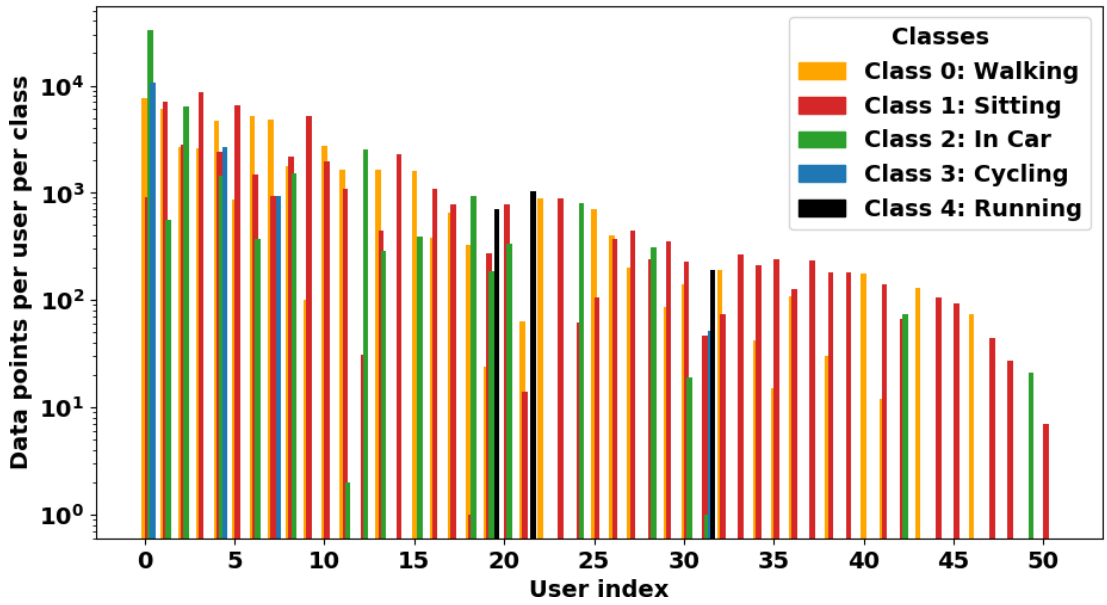


Figure 3.3 Number of data points of each class for each user.

model [166] into a small-size model, by using the processed accelerometer data as input of (1) a sequence of a 1D-CNN - a Batch Norm - a 1D-CNN - a Batch Norm - a Flatten layer, and (2) a sequence of a 1D-CNN - a Batch Norm - a Flatten layer. The two flatten layers are concatenated before feeding them into a sequence of a Drop Out layer - a Dense layer - and an Output layer. By doing so, HAR-Wild can memorize and transfer the low level latent features learned from the very first 1D-CNN, directly derived from the input data, to the output layer for better classification. We use Global Average Pooling [167] given its small memory footprint, instead of the popular Local Max/Average Pooling [168]. In addition to being appropriate for resource-constrained phones, a small-size model such as HAR-Wild is expected to perform better on data collected in the wild, since the data will likely have more distribution drift, increasing the chance of model overfitting on large-size models.

3.3.4 HAR-Wild Async Augmented Training

The performance of FL models is negatively affected by non-IID data distribution [4, 15, 91], and we observed this to be true for HAR-Wild as well. Figure 3.3 shows the

distribution of the dataset we collected for HAR-Wild. To address this problem, we leverage data augmentation training [169] and tailor it to mitigate the distortion in computing gradients at client-side by balancing the client data with a small number of augmentation data samples without an undue computational cost.

The pseudo-code for HAR-Wild's asynchronous augmented learning is shown in Algorithm 2. This algorithm is integrated in Algorithm 1 by replacing lines 7-12 from Algorithm 1 with the AUGMENTEDGRADIENTS procedure in Algorithm 2. Before the whole training process starts, the FL Cloud Manager executes the procedure INIT (lines 1-3, Algorithm 2), which first collects a small pool of random samples for each class that will be used for data augmentation (line 2). These data can be collected from a small number of volunteers or controlled users who share IID data with the FL Cloud Manager in FLSys. The augmentation data pool could also come from publicly available datasets. Then, the augmentation data pool $\mathcal{A}$ is delivered to each client (line 3). In each training round, each client (i.e., phone) randomly samples the augmentation data (line 8). Then, the sampled augmentation data $\mathcal{D}_{\mathcal{A}}$ will be combined with the local data $\mathcal{D}_{\mathcal{L}}$ (line 10, CONCATENATE($\mathcal{D}_{\mathcal{A}}$, $\mathcal{D}_{\mathcal{L}}$)) to compute the local gradients (lines 11-13, LOCALTRAINING). The local gradients are then sent to the cloud for the asynchronous average aggregation and model update (line 14).

In order to deliver the augmentation data to the clients (line 3), we consider two objectives: **(i)** privacy protection, and **(ii)** communication efficiency. One naive approach is to send data to augment the missing classes at the clients in each training round, since the local missing data can change over time. In this approach, the FL Cloud Manager needs to know which classes are missing for each client in each training round. This could increase the communication cost and significantly increase data privacy risk, since the cloud learns certain aspects of the user behavior based on the classes that miss data over time. To achieve both privacy protection and communication efficiency, the approach implemented in FLSys (Algorithm 2) first

Algorithm 2 HAR-Wild Asynchronous Augmented Learning

procedure INIT(*clients*)
    augmentation pool $\mathcal{A} \leftarrow$ SAMPLEAUGMENTDATA(*clients*)
    DELIVERAUGMENTPOOL($\mathcal{A}$, *clients*)
procedure AUGMENTEDGRADIENTS(Round t, Client i)
    Augmentation data pool $\mathcal{A}$
    Local data pool $\mathcal{L}_i$
    $\theta_l \leftarrow \theta^t$
    augmentation data $\mathcal{D}_{\mathcal{A}}$ = SAMPLEAUGMENTDATA($\mathcal{A}$)
    local data $\mathcal{D}_{\mathcal{L}}$ = SAMPLEDATA($\mathcal{L}_i$)
    training data $\mathcal{D}_{\mathcal{T}}$ = CONCATENATE($\mathcal{D}_{\mathcal{A}}$, $\mathcal{D}_{\mathcal{L}}$)
    for batch $b \in \mathcal{D}_{\mathcal{T}}$ **do**
        $\theta_l \leftarrow \theta_l - \eta \nabla \mathcal{L}(\theta_l; b)$
    $\Delta_l \leftarrow \theta_l - \theta^t$
    UPLOADCLIENTGRADIENTS(Δ_i)

delivers the entire augmentation data to every client only once at the beginning of the training process. Then, the clients use only the data necessary to augment their missing data. The clients check the missing classes when they receive the data, and re-check every time they accumulate enough new data (the amount of new data is a model-specific configuration parameter).

3.4 Evaluation

The evaluation has two main goals: **(i)** Analyze the performance of the two FL models, HAR-Wild and sentiment analysis (SA) with different aggregators and DP mechanisms; **(ii)** Quantify the system performance of FLSys with HAR-Wild and SA on Android and AWS. In terms of system performance, we investigate energy efficiency and memory consumption on the phone, system tolerance to phones that do not upload local gradients, and FL aggregation scalability in the cloud. We also study the overall response time for third party apps that use FLSys on the phone. For model evaluation, we use Accuracy, Precision, Recall, and F1-score metrics. For system performance, we report execution time and memory consumption for both the phones and the cloud, and battery consumption on the phones.

Table 3.5 Number of Samples in the Dataset for 51 Users

Type	Class 0 Walking	Class 1 Sitting	Class 2 In Car	Class 3 Cycling	Class 4 Running
Training	48855	51499	49185	14281	1920
Testing	8514	8828	8595	2514	319

Table 3.6 Model Settings of HAR-W and Baselines

Model	Optimizer	Other key parameters
HAR-Wild (centralized)	Adam	LR=0.0005, dropout_rate=0.4, batch_size=1024 Sampling: Same as class distribution
HAR-Wild (sim-FL)	Adam	client_LR=0.005, server_LR=1.0, dropout_rate=0.4, batch_size=128, Sampling: $[50, 100]$ samples per class, $[15, 30]$ augment samples per class
HAR-Wild (sim-FL with additional aggregators)	Adam	client_LR=0.005, server_LR=1.0, dropout_rate=0.4, batch_size=128 degree_of_adaptivity = 1, decay_parameters = 0.1, 0.9 Sampling: $[50, 100]$ samples per class, $[15, 30]$ augment samples per class
HAR-Wild (sim-FL with DP)	Adam	client_LR=0.005, server_LR=1.0, dropout_rate=0.4, batch_size=256 Sampling: $[50, 100]$ samples per class, $[15, 30]$ augment samples per class
HAR-Wild (FLSys)	Adam	client_LR=0.005, server_LR=1.0, dropout_rate=0.4, batch_size=64 Sampling: $[50, 100]$ samples per class, $[15, 30]$ augment samples per class
CNN-Ig (centralized)	Adam	LR=0.0005, dropout_rate=0.05, batch_size=1024 Sampling: Same as class distribution
BiLSTM (centralized)	Adam	LR=0.0005, dropout_rate=0.2, batch_size=1024 Sampling: Same as class distribution

Most of the evaluation is done with HAR-Wild, which illustrates a typical FL model based on mobile sensing data. To demonstrate that FL works for different models, we also show results for the SA model. The rest of the section is organized as follows: Subsection 3.4.1 compares HAR-Wild against baseline models and evaluates the effect of data augmentation, different aggregators, and advanced privacy mechanisms on HAR-Wild's performance. Subsection 3.4.2 describes the sentiment analysis (SA) model, used to demonstrate FLSys's support for different models, and shows its performance. Subsection 3.4.3 shows the HAR-Wild performance over the FLSys prototype, in terms of model accuracy, fault tolerance, and scalability. Since we did not have enough phones for larger-scale experiments, we show these results using Android/Linux emulators to replay each user's data. Finally, Subsection 3.4.4 presents results for HAR-Wild and SA over FLSys on two types of Android phones.

Table 3.7 HAR-Wild Using Centralized and FL Training vs. Baselines: Macro-Model Performance

Model	Accuracy	Precision	Recall	F1-score
HAR-W-32-centralized	0.8186	0.8486	0.8360	0.8409
HAR-W-64-centralized	0.8249	0.8512	0.8354	0.8428
HAR-W-128-centralized	0.8262	0.8529	0.8449	0.8484
BiLSTM	0.7868	0.8074	0.7831	0.7941
CNN-Ig	0.7639	0.7970	0.7715	0.7834
CNN-Ig_featureless	0.7708	0.8004	0.7779	0.7878
HAR-W-64-fed-stock	0.5368	0.3828	0.3569	0.3190
HAR-W-64-fed-uniform	0.7181	0.7464	0.7419	0.7378
HAR-W-64-fed-yogi	0.7107	0.6865	0.7731	0.7130
HAR-W-64-fed-adam	0.7072	0.6829	0.7592	0.7058
HAR-W-64-fed-adagrad	0.6691	0.6030	0.7429	0.6358

3.4.1 HAR-Wild Model Evaluation

Table 3.5 shows the basic information of our collected dataset used for all HAR-Wild experiments. Some users have very limited numbers of labeled activities; thus, we select data from 51 users who labeled a reasonable amount of samples.

Comparison with Baseline Approaches. We perform centralized evaluation to assess HAR-Wild's utility compared to several baselines. Centralized training works as an upper bound performance for FL models. In addition, it allows us to fine-tune the model's hyper parameters. The evaluation includes three variants of HAR-Wild: *HAR-W-32*, *HAR-W-64*, and *HAR-W-128*, which have the numbers of convolution-channels set to 32, 64, and 128. For comparison, we consider two baseline models: (1) *Bidirectional LSTM* with 3-axial accelerometer data as input. This is a typical model for time-series data, and we fine-tune it based on grid-search of hyperparameters; and (2) The CNN-based models proposed by Ignatov [20], with(*CNN-Ig*) and without(*CNN-Ig_featureless*) additional features using the author's recommended settings by Ignatov [20]. For a fair comparison, we used TensorFlow implementations for all models. Table 6.1 shows all the hyper-parameters and model configurations.

Figure 3.4 shows that HAR-Wild models outperform the baseline approaches. While the experiments run for up to 10,000 epochs to determine the performance

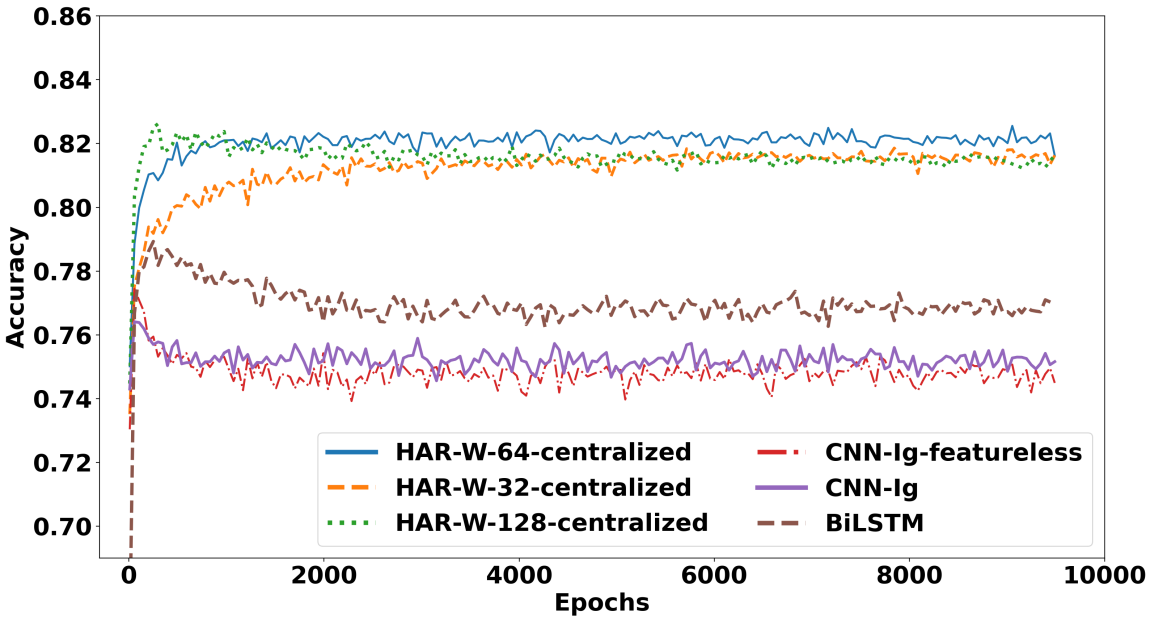


Figure 3.4 Centralized training evaluation.

upper bound, we observe the accuracy achieves acceptable performance after 1,000 epochs. On average, HAR-W-64 performs best and achieves 82.49% accuracy compared with 78.68%, 76.39%, and 77.08% of the BiLSTM, CNN-Ig and CNN-Ig-featureless. The results in Table 3.7 demonstrate that our HAR-Wild models also achieve the best performance in all the other metrics. Let us note that the absolute performance results may appear low when compared to HAR models run on data collected in controlled environment. This is because the data collected in the wild is noisier and non-IID. Overall, HAR-W-64 (60,613 trainable weights) has the best trade-off among model accuracy, convergence speed, and model size, and we use it in all the following experiments for HAR-Wild.

Comparison of Different FL Versions of HAR-Wild. We also perform FL simulations to compare HAR-Wild's performance across three dimensions: (1) with and without data augmentation (2) with different aggregators (3) with and without advanced privacy mechanisms. Since the simulations are in TensorFlow, we can also compare the FL results with the centralized training results. In the simulated FL, we replay the data collected in the wild for each user.

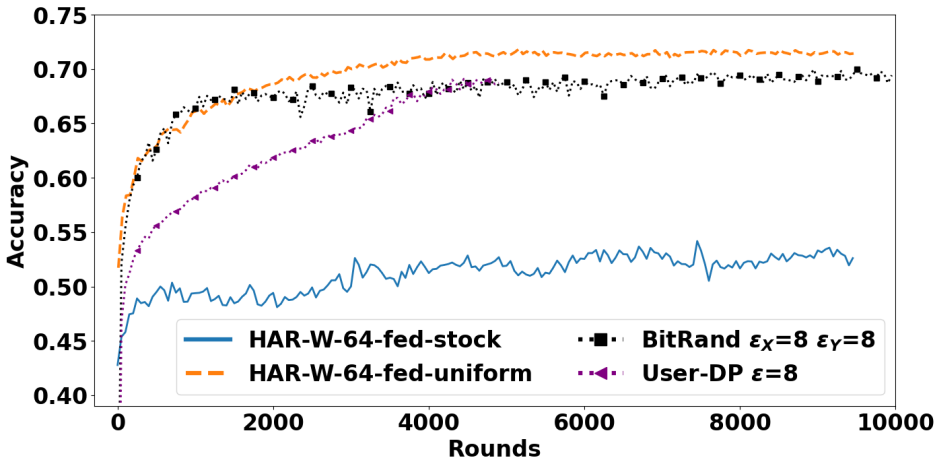


Figure 3.5 Comparison of FL HAR-Wild versions, w/ and w/o data augmentation, and w/ and w/o privacy protection.

In the following results, the basic FL HAR-Wild model without data augmentation and without privacy mechanisms is called *HAR-W-64-stock*. The model with data augmentation, but without privacy mechanisms, is called *HAR-W-64-uniform*. The augmentation data, consisting of 640 samples of each class, is fixed and shared with all clients.

The modular design of FLSys supports different FL aggregators. In addition to the standard FedAvg, we train the HAR-Wild model in FL with three aggregators designed to handle non-IID data [170]: FedYogi, FedAdam and FedAdagrad. To evaluate privacy protection in HAR-Wild, we apply the two types of privacy-preserving mechanisms available in FLSys (described in Section 3.1): *User-level DP* (*User-DP*) and *Local DP* (*LDP*). We experiment with one User-DP mechanism proposed by [97] and five LDP mechanisms: *BitRand* [171], *Duchi* [172], *Piecewise* [173], *Hybrid* [173], *Three-Outputs* [174]. All the hyperparameters are provided in Table 6.1.

Table 3.7 shows the results for different FL versions of HAR-Wild. HAR-W-64-fed-uniform (FedAvg with data augmentation) achieves 71.8% accuracy, which is

Table 3.8 Macro-model Performance for HAR-W-64-fed-uniform for Different Types of Privacy Protection Mechanisms and Different Parameters

DP Mechanism	Privacy Budget	Accuracy	Precision	Recall	F1-score
Non-DP	$\varepsilon \to \infty$	0.7181	0.7464	0.7419	0.7378
User-DP	$\varepsilon = 2$	0.5399	0.5264	0.5797	0.5259
User-DP	$\varepsilon = 4$	0.5973	0.5603	0.6297	0.5502
User-DP	$\varepsilon = 8$	0.6970	0.6333	0.7264	0.6523
BitRand	$\varepsilon_X = \varepsilon_Y = 2$	0.4251	0.3667	0.3715	0.3277
BitRand	$\varepsilon_X = \varepsilon_Y = 4$	0.5193	0.4607	0.5110	0.4416
BitRand	$\varepsilon_X = \varepsilon_Y = 8$	0.6943	0.6885	0.7359	0.7031
Duchi	$\varepsilon = 2$	0.4846	0.4286	0.5233	0.4201
Duchi	$\varepsilon = 4$	0.5122	0.4307	0.4998	0.4360
Piecewise	$\varepsilon = 2$	0.4857	0.4086	0.4267	0.3944
Piecewise	$\varepsilon = 4$	0.5065	0.4245	0.4686	0.4222
Hybrid	$\varepsilon = 2$	0.4791	0.3961	0.3714	0.3714
Hybrid	$\varepsilon = 4$	0.5353	0.4521	0.4508	0.4431
Three-Outputs	$\varepsilon = 2$	0.2906	0.2662	0.2348	0.0192
Three-Outputs	$\varepsilon = 4$	0.2946	0.3288	0.2424	0.2386

about 10% less than the accuracy of the centralized-trained HAR-Wild. This is the cost of privacy-protection provided by FL.

We tested FedYogi, FedAdam and FedAdagrad with and without data augmentation, and in both case they achieve comparable accuracy with FedAvg. Table 3.7 shows the results with data augmentation. Surprisingly, due to the noisy nature of HAR sensor data, the aggregators designed to handle non-IID data do not guarantee better performance than FedAvg. Therefore, the rest of the experiments will use FedAvg, which is the prevailing aggregator in FL.

For privacy protection mechanisms, we train the HAR-W-64-fed-uniform model with the aforementioned DP mechanisms. Then, we evaluated the trade-offs between model utility and privacy budget for different versions of HAR-Wild with privacy mechanisms, as shown in Table 3.8. As expected, the model utility decreases as privacy budget ε tightens. From this table, we select the best User-DP model (i.e., the one with $\varepsilon = 8$) and the best LDP model (i.e., BitRand with $\varepsilon_X = \varepsilon_Y = 8$) in terms of accuracy, and compare them with the models with and without augmentation in Figure 3.5. The results show that HAR-Wild with User-DP achieves a model accuracy of 69.70%, which is just 2.11% lower than the model without privacy protection.

Table 3.9 SA Model Performance Per Class for Centralized and Federated Learning

	Class	Accuracy	Precision	Recall	F1-score	Support
CL	negative	0.81	0.75	0.69	0.72	3159
	positive		0.84	0.88	0.86	5746
FL	negative	0.79	0.73	0.64	0.68	3159
	positive		0.81	0.87	0.84	5746

HAR-Wild with LDP (BitRand) achieves an accuracy of 69.43%, which is just 2.38% lower than the noiseless model. Note that our defense successfully prevents the server to reconstruct recognizable sensor signals and infer its associated ground-truth labels. One of the reasons is that it is more challenging to infer whether a time series of sensor signals belongs to a particular client than other domain applications. When using a tighter privacy budget, e.g., $\varepsilon_X = \varepsilon_Y = 4$ or 2, the gap between BitRand and Non-DP model becomes bigger. This is due to the fact that BitRand has not been designed for imbalanced data and cannot work well with significantly imbalanced data as our HAR dataset, especially when reducing the privacy budget ε_Y for protecting the labels. Let us also emphasize that both privacy protection mechanisms offer rigorous privacy guarantees in FLSys without significant computational overhead.

The different aggregators and privacy preserving mechanisms also showcase how the modularity of FLSys can be used to easily exchange different implementations of a module.

3.4.2 Sentiment Analysis (SA) Model Evaluation

FLSys is designed and implemented to be flexible, in the sense that training and inference of multiple models can run concurrently. On the server, different applications use independent AWS resources. On the phone, independent model trainers and inference runners are responsible for different applications. This subsection showcases the training performance of the SA model, a text analysis model that interprets and classifies the emotions (positive or negative) from text

data. For example, with the inferred emotions of mobile users' private text data, a smart keyboard may automatically generate emoji to enrich the text before sending.

We build the SA model for tweet data. We use the FL benchmark dataset Sentiment140 [1], which consists of 1,600,498 tweets from 660,120 users. We select the users with at least 70 tweets, and this sub-dataset contains 46,000+ samples from 436 users. Our SA model first extracts a feature vector of size 768 from each tweet with DistilBERT [175]. Then, it applies two fully connected layers with ReLU and Softmax activation, respectively, to classify the feature vector into positive or negative. The number of hidden states of the first fully connected layer is set to 128 to balance the convergence speed and model size. In the FL version of the model, 5% of the users are used for data augmentation, and the rest of the users follow 4:1 train-test split.

While the reference implementation associated with this benchmark dataset reached 70% accuracy [82] using 100 users with stacked LSTM in FL simulation, our SA model achieves superior performance, as shown in Table 3.9. Centralized learning achieves 81% accuracy, while FL achieves 79% accuracy (an acceptable drop). The FL version of this SA model will be further evaluated while running over FLSys on Android phones in Subsection 3.4.4.

3.4.3 HAR-Wild over FLSys Emulation Performance

To evaluate the performance of HAR-Wild over the FLSys prototype at scale, we use Android emulation because we did not have enough phones for these experiments. Furthermore, since Android emulation is slow and costly, we run several larger-scale experiments with the same DL4J algorithms and functions in Linux, which is much faster. We train the model in these experiments for only 1,000 rounds because the simulation results showed that the accuracy is acceptable starting with this number of rounds.

[1] http://help.sentiment140.com/home

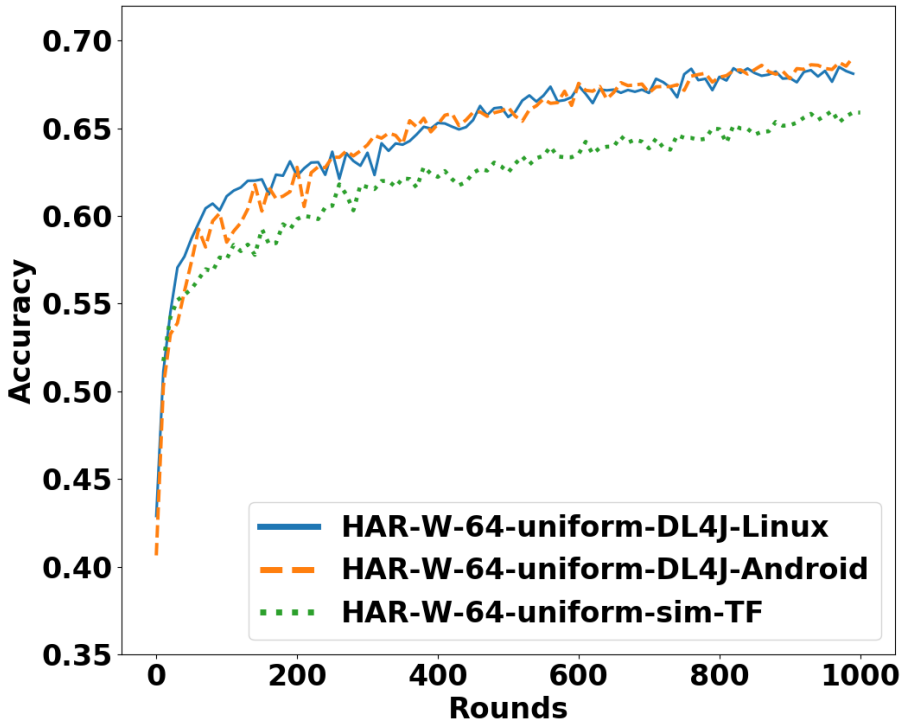


Figure 3.6 HAR-Wild over FLSys using Android/Linux emulation.

Table 3.10 Performance Per Class of HAR-Wild over FLSys Using Android Emulation

Class	Accuracy	Precision	Recall	F1-score
0	0.6907	0.7003	0.6628	0.6810
1		0.5922	0.8655	0.7032
2		0.8606	0.5443	0.6668
3		0.8324	0.6450	0.7268
4		0.6682	0.9028	0.7680

All the phone components of the prototype, except for Data Collector and Data Preprocessor, run in the emulators. The cloud part of the prototype runs in AWS. The Android emulators run on top of virtual machines (VMs) in Google Cloud, as AWS does not support nested virtualization. We run 10 VMs in Google Cloud, and each VM has 16 vCPUs and 60GB memory. On each instance, we run 4 Android v10 emulators from AVD manager in Android Studio. Each emulator is loaded with 3 users' data files, and each file is sampled twice as different clients. In each round, each Android emulator participates in training on behalf of a few clients. We set the deadline for the round in the FL Cloud Manager to 6 minutes.

Accuracy. Figure 3.6 shows that HAR-Wild with 64 clients emulation in both Android and Linux on FLSys achieve comparable accuracy with the simulated FL with TensorFlow, i.e., 69.07%, 68.50%, and 66.00%. Table 3.10 shows HAR-Wild's performance per class using FLSys and Android emulation. Although our data collected in the wild are inevitably unbalanced (Table 3.5), every class performs reasonably well with F1-scores between 66.7% and 76.8%. Figure 3.7 shows the results of HAR-Wild with higher number of clients (up to 960) using Linux emulations. The client data was over-sampled from the original 51 users. HAR-Wild model achieves up to 69.17% accuracy, and more clients help the model converge quicker with better performance.

Fault Tolerance. In daily life, some clients may fail to upload a trained model to the FL Cloud Manager due to network or computation issues. This set of experiments verifies the fault tolerance of FLSys in terms of model performance as a given percentage of clients drop out randomly in each round. Figure 3.7 shows the accuracy of HAR-Wild with up to 50% clients dropping out randomly from 480 clients in each round. With 1,000 rounds of training, the accuracy is reduced by at most 3.11%. This is a promising result showing that FLSys can tolerate reasonably large dropout rates during training.

Scalability. As discussed in Section 3.2, computation and storage scale independently in the cloud for FLSys. This set of experiments verifies the scalability of FLSys across training rounds. The only FL function that may be computationally intensive in the cloud is the Model Aggregator. Figure 3.9 shows the Model Aggregator in AWS scales linearly with the number of participating clients. We also observe that the aggregation of 960 clients generally finishes in less than 4 minutes. By interpolating these results and given the current 15 minutes execution time limit of an AWS Lambda process [176], the FLSys prototype (with single-threaded aggregator) can handle up to 3,600 clients, which is a sufficient number of clients, per

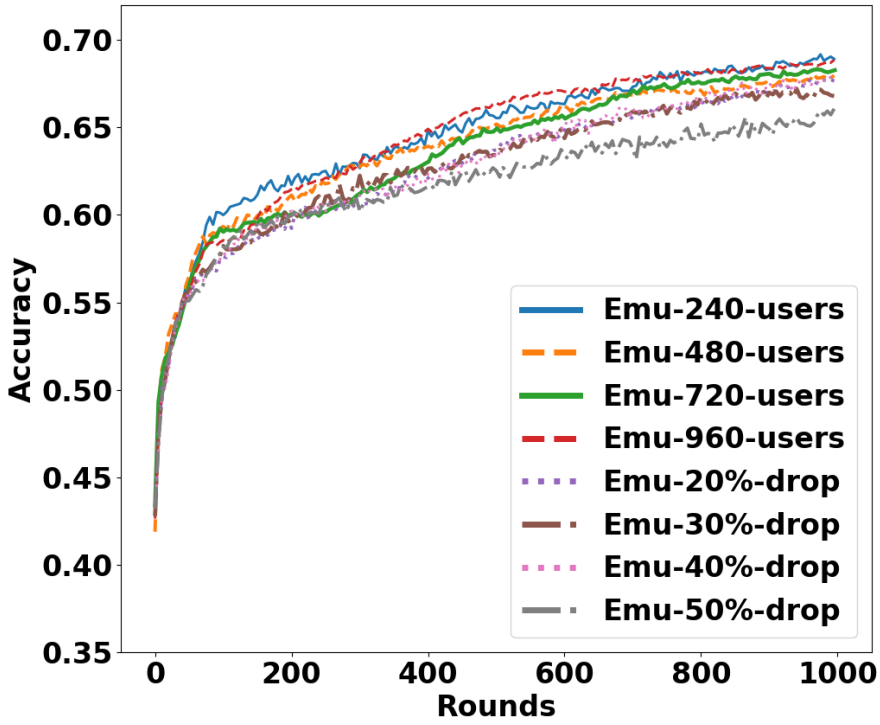


Figure 3.7 Linux emulation of HAR-Wild over FLSys, while varying total number of users and number of users dropping from training.

training round. This number can be multiplied substantially by implementing both thread-level and process-level parallelization to handle real-world traffic volume.

Overall, the results for accuracy, fault-tolerance, and scalability demonstrate that FLSys and HAR-Wild can work well in real-life, where they are deployed on Android phones and the AWS cloud.

3.4.4 FLSys Performance on Smart Phones

We benchmarked FLSys with HAR-Wild and SA on Android phones using a testing app to evaluate training and inference performance. We also assessed the resource consumption on the phones. We used three phones with different specs (Nexus 6P, Google Pixels 3 and 3a). The results demonstrate the on-device feasibility of FLSys, even for a low-end Nexus 6P phone, unveiled in 2015 and running Android 7. Since

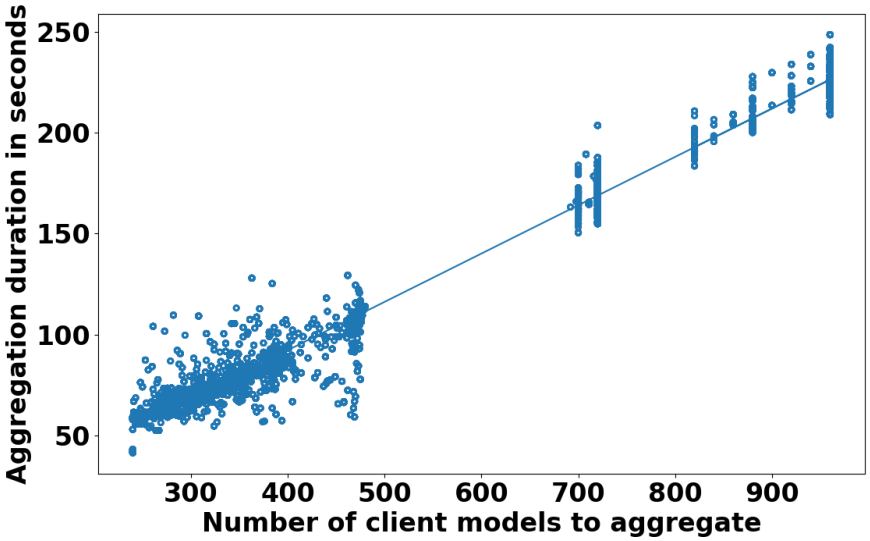


Figure 3.8 Aggregation time and participating clients.

FLSys works well on such a low-end phone and people change their Android phones every 2-3 years on average [2], we expect FLSys to work well on most of today's phones.

Training Performance. Table 5.7 shows the training time and the resource consumption on the phones. The training time is recorded by training 650 samples for 5 epochs for HAR-Wild, and 100 samples for 5 epochs for SA, which are the optimum scenarios determined in Subsection 3.4.3. Foreground training is done while leaving the screen on, and it uses the full single core capacity. It provides a lower bound for the training time. However, in reality, we expect training to be done in the background, either on battery or on charger. As in practice, other apps or system processes working in background may interfere with training. We take 10 measurements for each benchmark, and report the mean and standard deviation.

Training for one round is fast on the phones. The foreground training time on the more powerful phone, Pixel 3, is just 0.7 min for HAR-Wild, and 0.22 min for SA. The background training time on charger, which is the expected situation for FL training, is good for any practical situation. The phones experience a higher training time compared with the foreground case (completed one training round in less than 4

[2]https://www.statista.com/statistics/619788/average-smartphone-life/

Table 3.11 Training on Android Phones: Resource Consumption and Latency

Model	Phone	Max RAM Usage (MB)	Foreground Training Time Mean/SD (min)	Background Training Time on Charger Mean/SD (min)	Background Training Time on Battery Mean/SD (min)	Battery Consumption per Round (mAh)	Number of Training Rounds for Full Battery
HAR	Nexus 6P	219	4.95/0.94	39.10/26.10	45.34/24.31	35.10	98
	Google Pixel 3a	156	1.23/0.01	3.94/0.04	85.82/33.07	9.72	308
	Google Pixel 3	165	0.70/0.06	3.58/0.10	79.96/36.82	3.79	769
SA	Nexus 6P	139	1.62/0.08	5.04/0.13	29.79/17.13	7.94	435
	Google Pixel 3a	128	0.33/0.005	0.84/0.006	25.42/5.72	2.02	1481
	Google Pixel 3	136	0.22/0.002	0.76/0.02	24.19/8.12	0.76	3846

Table 3.12 Inference on Android Phones: Resource Consumption and Latency

Model	Phone	Max RAM Usage (MB)	Foreground Inference Time Mean/SD (millisecond)	Background Inference Time on Charger Mean/SD (millisecond)	Background Inference Time on Battery Mean/SD (millisecond)	Battery Consumption per prediction (μAh)	Millions of inferences for Full Battery
HAR	Nexus 6P	161	54.65/16.36	1963.04/1540.29	7646.73/16349.49	4.49	0.77
	Google Pixel 3a	158	38.48/10.07	99.73/19.76	100.11/19.69	4.12	0.73
	Google Pixel 3	177	36.59/6.43	99.60/33.69	100.11/21.45	1.94	1.50
SA	Nexus 6P	114	19.66/6.06	20.10/20.04	20.25/28.11	3.35	1.03
	Google Pixel 3a	108	11.90/3.71	20.65/4.45	19.58/3.93	2.3	1.30
	Google Pixel 3	129	10.11/2.88	15.59/5.89	17.42/5.69	0.17	17.63

minutes). The background training time on battery is notably longer, since Android attempts to balance computation with battery saving.

The results show training is also feasible in terms of resource consumption. The maximum RAM usage of the app is less than 165MB, and modern phones are equipped with sufficient RAM to handle it. While we did not perform experiments for battery consumption in the foreground (as this test was used just for a lower bound on computation time), we measured battery consumption for background training on battery. The phones could easily perform hundreds of rounds of training on a fully charged battery. It is worth noting that, typically, one round of training per day is enough, as the users need enough time to collect new data.

Inference Performance. The results in Table 5.6 demonstrate that FLSys can be used efficiently by third-party apps. The inference time is measured within the third party testing app. Let us note that the inference is performed locally by the FL Phone Manager, without any network communication. Thus, the measured

time consists of the inference computation time and the inter-process communication time. We continuously perform predictions for 30 minutes and report the average values. The inference time for the three scenarios on the third-party app, foreground, background on charger, and background on battery, follows a similar trend as training. FLSys and HAR-Wild/SA have reasonable resource consumption, which make them effective in practice.

In addition to HAR and SA, many other applications may benefit from FLSys. For example, FL models are appropriate for privacy-sensitive image and video data collected on mobile devices. Existing research confirms that such models are feasible on resource-constrained mobile devices. For training, Mathur et al. [177] demonstrated that training a 2-layer DNN classifier on top of a pre-trained MobileNet [178] on Android clients for the Office-31 dataset takes about 30 minutes to converge. For inference, we tested the inference time of MobileNet on 224*224 images, and it takes about 120ms for a single CPU thread. These numbers are comparable with our results on HAR and confirm that such models could run over FLSys.

3.4.5 FLSys Performance in the Cloud

As discussed in Section 3.2, computation and storage scale independently in the cloud for FLSys. This set of experiments verifies the scalability of FLSys in the cloud for the Android emulation scenario presented in Subsection 3.4.3 across training rounds.

Results. The only FL function that may be computationally intensive in the cloud is the Model Aggregator. Figure 3.9 shows the Model Aggregator in AWS scales linearly with the number of participating clients. We also observe that the aggregation finishes in general in less than 10 seconds. By interpolating these results and given the current 15 minutes execution time limit of AWS Lambda [164], the FLSys prototype can handle up to 60,000 clients per training round. This number can be increased substantially by allocating more resources to the Lambda function used for model aggregation. AWS Lambda supports up to 6 vCPUs and 10,240MB

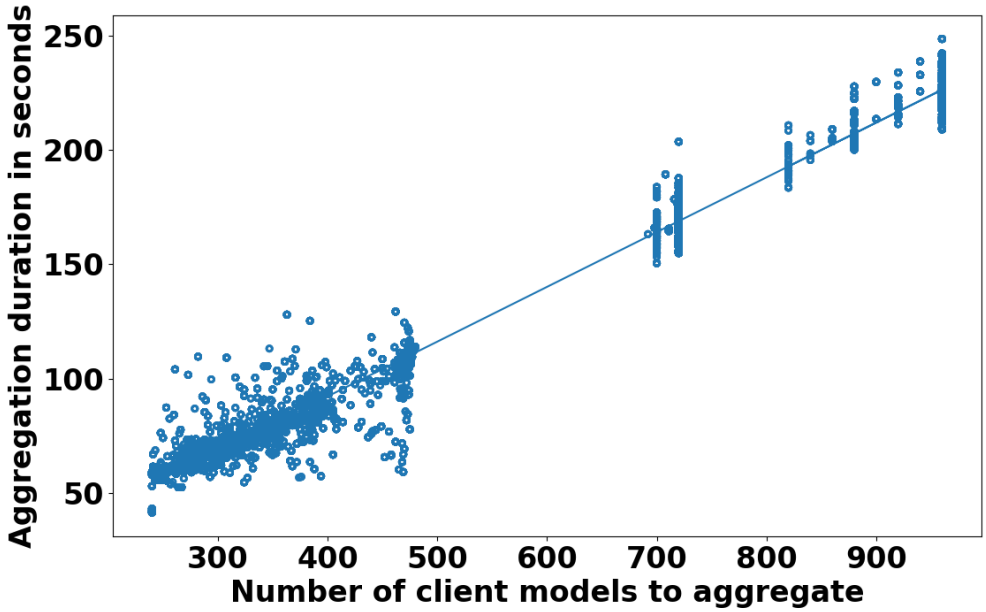


Figure 3.9 FLSys aggregation execution time against the number of models.

of memory, while our current allocation is 1 vCPU and 2,048MB of memory. Parallel execution of multiple Lambda functions, using in-memory processing, can further increase the number of clients supported by FLSys per round. Given that AWS Lambda has default concurrency limit of 1,000 (and can be increased further), there is no doubt the system may be scaled to handle real-world traffic volume.

3.5 Chapter Summary and Lessons Learned

This paper presented our experience with designing, building, and evaluating FLSys, an end-to-end federated learning system. FLSys was designed based on requirements derived from real-life applications that learn from mobile user data collected in the wild, such as human activity recognition (HAR). Compared with existing FL systems, FLSys balances model utility with resource consumption on the phones, tolerates client failures/disconnections and allows clients to join training at any time, supports multiple DL models that can be used concurrently by multiple apps, provides support for advanced privacy protection mechanisms, and acts as a "central hub" on the phone to manage the training, updating, and access control of FL models used by different

apps. We built a complete prototype of FLSys in Android and AWS, and used this prototype to demonstrate that FLSys is effective and efficient in practice in terms of model performance, privacy protection, resource usage, and latency. We believe FLSys can open the path toward creating an FL ecosystem of models and apps for privacy-preserving deep learning on mobile sensing data. In terms of actual deployment of FLSys in practice, we believe it can be offered as FL as a Service (FLaaS) by cloud providers.

Next, we report lessons learned and future work. The lessons learned are based on our experience with running FLSys on data collected in the wild from 50+ users over a 4-month period. Larger scales and longer periods are necessary for additional insights into system scalability and robustness, as well as model performance at scale.

Build mechanisms to cope with non-IID data. Since our data collection happened during the Covid-19 pandemic, we expected to see somewhat similar data from users who mostly stayed indoors. However, the data was non-IID, strengthening the idea that data collected in the wild will almost always be non-IID. A future work in FLSys is to provide support for model and data-specific augmentation and other approaches to cope with non-IID data.

Beware the simulation pitfalls. One common practice in FL simulations is to use the same instances/placeholders in memory for the different clients. Such simulations must carefully reset the instances for different clients to avoid any information leakage among clients, which can never happen in a real system. Our initial experiments showed unexpectedly different results between simulations and Android emulators with DL4J for the same settings. The first problem we discovered was that Batch Normalization (BN) is not supported in DL4J for specific data shapes. We implemented our own BN in DL4J, but the simulation results still did not match the experimental results. Finally, we realized that BN does not work well for FL (consistent with [179]), but it does work in the simulations due to shared instances

among the simulated clients. Thus, the FL models used in the reported experiments do not use BN. The second problem we noticed was that the Adam optimizer worked well for simulation, but not for the Android emulator experiments. This was also caused by shared instances accessed by all clients in the simulation. This should not happen in practice given privacy leakage through the shared instances. The lesson learned was that simulation may show better results than experiments with real systems for FL. Since most of FL papers in the literature are based on simulations, their results may suffer from similar problems with the ones described here. We believe FLSys offers an opportunity to test such FL models in real-life conditions.

Balance mobile resources and model accuracy. In the current FL literature, there are no results to show the FL models work well on mobile devices, while consuming a limited amount of resources on these devices (e.g., battery power, memory). A lesson that we understood early on is that FLSys will need to balance resource usage on mobiles with model accuracy. Therefore, FLSys used an asynchronous design in which policies on the mobile devices are evaluated to decide when it makes sense for the device to participate in training and consume resources. Our results show that good model accuracy can be achieved even when a significant number of mobile devices do not participate in training in order to save resources. Let us also note that real systems cannot expect to run the same number of rounds that we observe in simulations. For example, it is common to see 10,000 rounds in simulations. However, in real life, mobile devices may not train more than once a day due to both resource consumption and lack of enough new data. In such a situation, running 10,000 rounds will take over 27 years. Thus, models must be optimized for a realistic number of rounds.

Design for flexibility. FLSys was designed for model flexibility on the phones from the beginning (i.e., allow apps to use multiple interchangeable models). Nevertheless, we did not originally design for flexiblity in the cloud. At first, we used

virtual machines in the cloud and durable cloud storage for all FL operations. When we analyzed scalability and performance issues, we realized that an FaaS solution and different types of storage are necessary. Therefore, we changed the design of the FLSys in the cloud to allow for different types of cloud platforms and storage options. Thus, FLSys can easily be ported to other cloud platforms beyond AWS.

CHAPTER 4

ZONE-BASED FEDERATED LEARNING

In this chapter, we introduce *ZoneFL* to adapt to user mobility behavior in mobile sensing data and improve the system scalability in FL. ZoneFL employs a novel approach by dividing the physical space into geographical zones, which are then mapped to a mobile-edge-cloud system architecture. This design enables achieving both high model accuracy and scalability. Each zone corresponds to a federated training model known as a zone model, which adapts well to the data and behavior patterns of users within that specific zone. The FL nature of ZoneFL ensures that user data privacy is protected during the training process.

The chapter is organized as follows. Section 4.1 presents the ZoneFL training and the algorithms to dynamically adapt to user mobility. Section 4.2 describes the design and implementation of the ZoneFL system. Section 4.3 shows the experimental results and analysis. The paper concludes and discusses future work in Section 4.4.

4.1 ZoneFL Training

This section presents zone partition, an overview of the ZoneFL training, and then describes our two federated training algorithms that allow ZoneFL to adapt to changes in user mobility.

4.1.1 Zone Partition

The physical space (e.g., a city) is partitioned into non-overlapping zones, based on administrative boundaries or other knowledge about their characteristics (e.g., shopping district, park, etc.). The zones are model-specific. For example, a heart rate prediction model has different zones compared with a vehicular traffic prediction model. In this way, ZoneFL can achieve better model performance by targeting

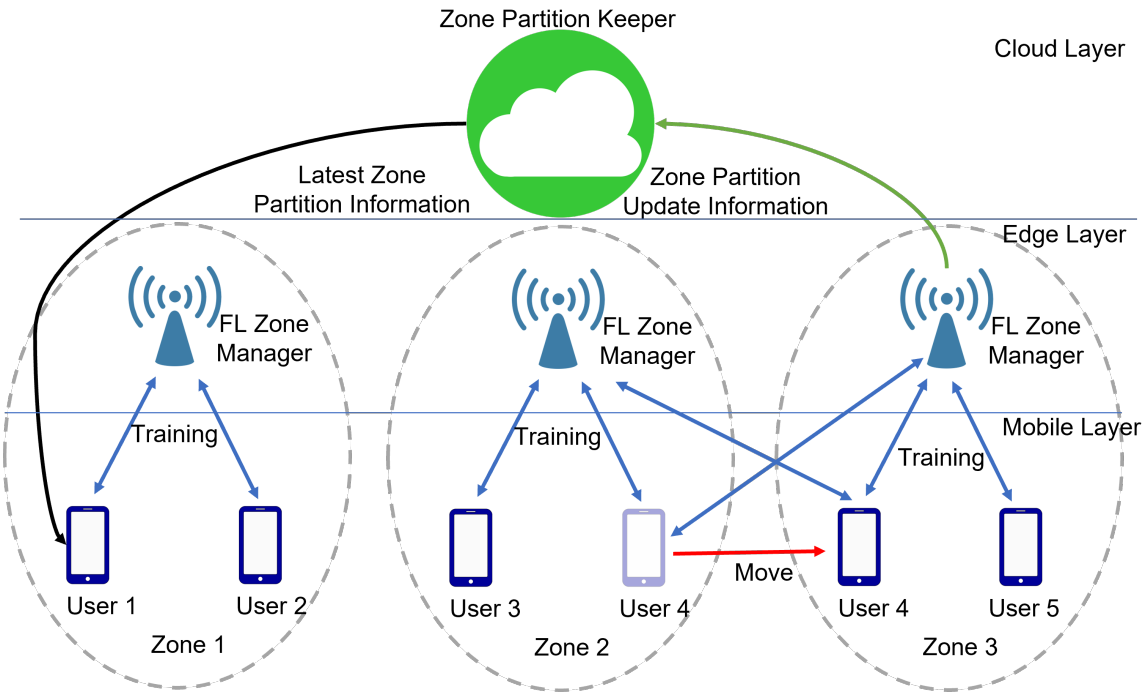


Figure 4.1 ZoneFL training architecture.

training to zones in which the user behavior is more homogeneous for a given type of mobile sensing data. For example, the user mobility behavior in a park (e.g., exercising) is different from the behavior in a shopping districts (e.g., leisurely walking).

The granularity of zones can be defined based on the target application and the size of the user pool, i.e., each zone shall be small enough for behavior differences, while big enough to have sufficient users for better scalability-utility balance. The zone topology is a graph defined by neighboring relations of zones. By default the neighboring relation is adjacency (i.e., two zones are neighbors if their borders touch each other), but this could be modified, for example to define that two zones geographically closer than a given threshold are neighbors.

4.1.2 ZoneFL Training Overview

ZoneFL is designed to use a mobile-edge-cloud architecture, and its main goal is to train separate zone models (i.e., separate instances of the same base model) on mobile

sensing data collected in each zone. Figure 4.1 shows the ZoneFL training and its high-level architecture. Each zone is managed by an FL Zone Manager at the edge, which maintains the latest models for its zone. A mobile is not tied to a single zone, but collects data from all the zones visited by the user and engages in training for each of these zones. For example, Figure 4.1 shows how *User 4* moves from *Zone 2* to *Zone 3* and collects data in both zones. For each zone, mobiles that collected data in that zone train the zone models jointly with the edge zone manager. Mobile devices download the updated models their apps need from the edge managers when they need inference in a new zone (e.g., *User 4* in *Zone 3*). In this process, the FL Zone Manager at the edge does not know when and where the user was in a zone. It only knows the user has collected data in a zone, and need performance inference. Therefore, the potential privacy information that the edge can infer is very limited. The cloud collaborates with the edge nodes to dynamically maintain the zone partition information for the entire space, as the geographical coordinates of the zones may change over time, but it is not involved in training. The mobile devices download the zone partition information and the identifiers of the edge managers from the cloud every time new zone configuration information is available. The zone partition information is used to associate data with different zones, perform local training, and send the weights to the corresponding zone edge manager, which will aggregate the zone model.

The logical architecture of ZoneFL allows for the FL Zone Managers to be located in the cloud or at the edge. This decoupling of the software component from the hardware is useful until edge nodes will become widespread. Currently, the mobile-edge-cloud architecture is available only in certain major cities, etc. Nevertheless, the mobile-edge-cloud architecture provides better scalability than a mobile-cloud architecture because edge nodes in ZoneFL have a lower communication and computation load than the cloud server in traditional FL. Furthermore, the edge

allows for faster interaction with the mobiles and for less bandwidth consumption in the network core. Finally, let us note that we assume only one edge node per zone. If there are multiple edge nodes in a zone, they can act as relays between the mobile devices and the node that runs the FL Zone Manager.

A major question in ZoneFL training is how to adapt the zone models to changes in user mobility behavior over time. We present two federated training algorithms that address this questions in different ways. First, Zone Merge Split (ZMS) dynamically adapts the zone partitions (i.e., the zone geographic coordinates) by either 1) merging two neighboring zones into a larger zone, whose model performs better than each of the individual zone models, or 2) splitting a larger zone back into previously merged smaller zones, whose individual models perform better than the model of the larger zone. Second, Zone Gradient Diffusion (ZGD) improves a zone model by aggregating contextual information derived from local gradients of neighboring zones. In ZGD, the zones do not change, but the user mobility behavior change is captured through the diffusion of information from neighboring zones. A self-attention mechanism is applied in ZGD to dynamically quantify the impact of each zone on its neighbors. Different deployments of ZoneFL may use either ZMS or ZGD or a combination of both based on trade-offs between model utility, scalability, and user mobility behavior.

4.1.3 Zone Merge and Split (ZMS)

ZMS is a dynamic zone management protocol that optimizes model utility across zones. We use novel greedy algorithms for the two operations.

Zone Merging. Given a set of N non-overlapping zones $Z = \{Z_i\}_{i\in[0,N]}$ and its complete set of possible combinations of zones Θ, merging a zone Z_i with its neighboring zones in Z is to find the smallest set of non-overlapping and merged zones $\mathcal{Z} = \{\mathcal{Z}_j\}_{j\in[0,|\mathcal{Z}|]}$ where $\mathcal{Z} \in \Theta$, $|\mathcal{Z}|$ is the number of non-overlapping and merged zones in $\mathcal{Z}$, and $\cup_j \mathcal{Z}_j = \cup_i Z_i$ so that: **(1)** The model utility across merged

zones $\sum_{\mathcal{Z}_j \in \mathcal{Z}} \mathcal{L}(\boldsymbol{\theta}_j, \mathcal{Z}_j)$ is optimized (Equation 4.1); and **(2)** Every zone Z_i achieves better model utility after merging (Equation 4.2). Note that $\mathcal{L}(\boldsymbol{\theta}_j, \mathcal{Z}_j)$ is the loss function of a zone $\mathcal{Z}_j$ with the model parameters $\boldsymbol{\theta}_j$.

$$\mathcal{Z}^*, \{\boldsymbol{\theta}_j^*\} = \underset{\{\boldsymbol{\theta}_j\}, \mathcal{Z} \in \Theta}{\arg\min} \sum_{\mathcal{Z}_j \in \mathcal{Z}} \mathcal{L}(\boldsymbol{\theta}_j, \mathcal{Z}_j) \tag{4.1}$$

$$\text{s.t. } \forall Z_i \in \mathcal{Z}_j : \mathcal{L}(\boldsymbol{\theta}_j^*, Z_i) \leq \mathcal{L}(\boldsymbol{\theta}_i^*, Z_i) \tag{4.2}$$

where $\mathcal{L}$ is a loss function, and the loss of a zone $\mathcal{Z}_j$ is an average loss over all the users' local data in that zone: $\mathcal{L}(\boldsymbol{\theta}_j, \mathcal{Z}_j) = \frac{1}{|U_j|} \sum_{u \in U_j} \mathcal{L}(\boldsymbol{\theta}_j, u)$ where $|U_j|$ is the number of users in the zone $\mathcal{Z}_j$.

Zone Splitting. Splitting a large zone into a set of smaller sub-zones is the reverse process of merging zones. Given a large zone $Z = \cup_{i \in [0,N]} Z_i$ formed by merging smaller sub-zones $\{Z_i\}_{i \in [0,N]}$ and $\boldsymbol{\Theta}$ is the set of all possible combinations of sub-zones $\{Z_i\}_{i \in [0,N]}$, splitting Z is to find a set of sub-zones $\mathcal{S} \in \boldsymbol{\Theta}$, such that: **(1)** The model utility across sub-zones is optimized; and **(2)** Every sub-zone Z_i achieves better model utility after the zone splitting.

$$\mathcal{S}^*, \{\boldsymbol{\theta}_j^*\} = \underset{\mathcal{S} \in \boldsymbol{\Theta}, \{\boldsymbol{\theta}_j^*\}}{\arg\max} \frac{1}{|\mathcal{S}|} \sum_{\mathcal{Z}_j \in \mathcal{S}} [\mathcal{L}(\boldsymbol{\theta}_Z^*, Z) - \mathcal{L}(\boldsymbol{\theta}_j^*, \mathcal{Z}_j)] \tag{4.3}$$

Equation 4.3 indicates S^* is the set of zones which has the maximal utility gain from the federated training of the original zone Z, i.e., $1/|\mathcal{S}| \sum_{\mathcal{Z}_j \in \mathcal{S}} [\mathcal{L}(\boldsymbol{\theta}_Z^*, Z) - \mathcal{L}(\boldsymbol{\theta}_j^*, \mathcal{Z}_j)]$.

Zone Merge and Split (ZMS) Algorithm. We propose ZMS, a greedy algorithm to dynamically adapt the zone models to changes in the user mobility behavior over time. In simple terms, ZMS merges two zones when the model

Algorithm 3 Zone Merging Algorithm

Input: Zone Z_i

1: $C \leftarrow \emptyset$ # initialize a list of zone merging candidates
2: $\mathcal{N} \leftarrow \textbf{\textit{getNeighbors}}(Z_i)$ # get neighboring zones of Z_i
3: **for** each neighboring zone $Z_n \in \mathcal{N}$ **do**
4: $\quad \boldsymbol{\theta}_{in}^{t} \leftarrow (\boldsymbol{\theta}_{i}^{t} + \boldsymbol{\theta}_{n}^{t})/2$ # average of two zone models
5: $\quad \boldsymbol{\theta}_{in}^{t+1} \leftarrow \arg\min_{\boldsymbol{\theta}_{in}} \mathcal{L}(\boldsymbol{\theta}_{in}^{t}, Z_i \cup Z_n)$ # Equation 4.1
6: $\quad$ **if** $\mathcal{L}(\boldsymbol{\theta}_{in}^{t+1}, Z_i) < \mathcal{L}(\boldsymbol{\theta}_{i}^{t+1}, Z_i)$ and $\mathcal{L}(\boldsymbol{\theta}_{in}^{t+1}, Z_n) < \mathcal{L}(\boldsymbol{\theta}_{n}^{t+1}, Z_n)$ # satisfying Equation 4.2 **then**
7: $\quad\quad C \leftarrow C \cup Z_n$ # add Z_n into a list of candidates
8: **if** $C \neq \emptyset$ **then**
9: $\quad Z_n^* \leftarrow \arg\max_{Z_n \in C} \left[\mathcal{L}\left(\boldsymbol{\theta}_{in}^{t+1}, Z_i\right) - \mathcal{L}(\boldsymbol{\theta}_{i}^{t+1}, Z_i)\right] + \left[\mathcal{L}\left(\boldsymbol{\theta}_{in}^{t+1}, Z_n\right) - \mathcal{L}(\boldsymbol{\theta}_{n}^{t+1}, Z_n)\right]$ # get the best neighboring zone
10: $\quad \textbf{\textit{Merge}}(Z_i, Z_n^*)$

performance of the merged zone is better than the performance of each of the models of the individual zones (i.e., to be merged). Each Zone Manager makes its own decisions regarding when to run the zone merging or zone splitting, as this decision depends on the conditions of each zone. For instance, the users in some zones may collect more data than the users in other zones, which may result in more frequent training. Also, the user behavior may change in some zones, while remaining similar in others. While running the zone merging and splitting in every training round may result in the best zone partitioning, such a solution results in too much overhead for both mobile users and Zone Managers. Therefore, we need to balance the trade-offs between zone partitioning efficiency and the computation and communication overhead.

Instead of checking all possible zone merges, ZMS randomly selects a zone Z_i to check for possible zone merging at every round t. In the merging Algorithm 3), ZMS merges Z_i with its best neighboring zone Z_n^*, optimizing the zone merging objectives in Equation 4.1 and 4.2 (Algorithm 3, Lines 2-7). The number of neighbors in line 2 is typically a small constant in practice, and lines 4-7 repeat over it. The additional round of training in line 5 trades computation cost for better performance improvement guarantee. It can be omitted, and $\boldsymbol{\theta}^{t+1}$ becomes $\boldsymbol{\theta}^{t}$ in lines 6 and 9. The best neighboring zone Z_n^* is the zone that provides the maximal utility gain after

the zone merging among all potential merges (Algorithm 3, Line 9). To compute the utility gain, at the next training round $t+1$, we quantify the improvement of the loss in zones Z_i and Z_n using the zone models $\boldsymbol{\theta}_i^{t+1}$ and $\boldsymbol{\theta}_n^{t+1}$ trained respectively on Z_i and Z_n compared with using the zone model trained on the merged zone $Z_i \cup Z_n$.

The zone models are trained and validated in the background by the phones in their respective zones. Mobile phones retain a small validation dataset to validate the zone models, and send the validation results to their zone manager to be used in merge decisions. Thus, these operations do not incur latency during merges. The only operation that needs to be done specifically for a merge is the validation of the model over the two zones. To reduce the overhead, the zone manager to select only a percentage p of the phones in its zone to perform training and validation in this case.

Merging in ZMS also handles the case when the original zones set during bootstrapping do not have enough data for adequate training. In this situation, ZMS will merge such zones with neighboring zones, therefore improving performance.

Algorithm 4 Zone Splitting Algorithm

Input: Zone $\mathcal{Z}_j = \cup_i Z_i$, level l

1: $C \leftarrow$ GETCANDIDATES$(\mathcal{Z}_j, l)$
2: **for** each zone $Z_c \in$ top-$k(C)$ **do**
3: $\quad \boldsymbol{\theta}_c^{t+1} \leftarrow \arg\min_{\boldsymbol{\theta}_c} \mathcal{L}(\boldsymbol{\theta}_j^t, Z_c)$
4: $\quad$ **if** $\mathcal{L}(\boldsymbol{\theta}_c^{t+1}, Z_c) < \mathcal{L}(\boldsymbol{\theta}_j^{t+1}, Z_c)$ **then**
5: $\quad\quad$ ***split***$(\mathcal{Z}_j, Z_c)$ # split the sub-zone Z_c from the merged zone $\mathcal{Z}_j$
6: $\quad\quad$ **break**
7: **function** GETCANDIDATES$(\mathcal{Z}_j, l)$:
8: $\quad C \leftarrow \emptyset$ # initialize a list of worst sub-zones
9: $\quad$ **for** $Z_c \in$ ***subZones***$(\mathcal{Z}_j, l)$ **do**
10: $\quad\quad$ **if** $\mathcal{L}(\boldsymbol{\theta}_j^t, Z_c) > \mathcal{L}(\boldsymbol{\theta}_j^t, \mathcal{Z}_j)$ **then**
11: $\quad\quad\quad C \leftarrow C \cup Z_c$
12: $\quad$ **return** $sorted(C)$ # descending $\mathcal{L}(\boldsymbol{\theta}_j^t, Z_c)$

ZMS repeats this zone merging process across federated training rounds to create a set of merged zones, denoted $\mathcal{Z}$. However, over time, in response to user mobility behavior changes, some of the merged zones may need to be split.

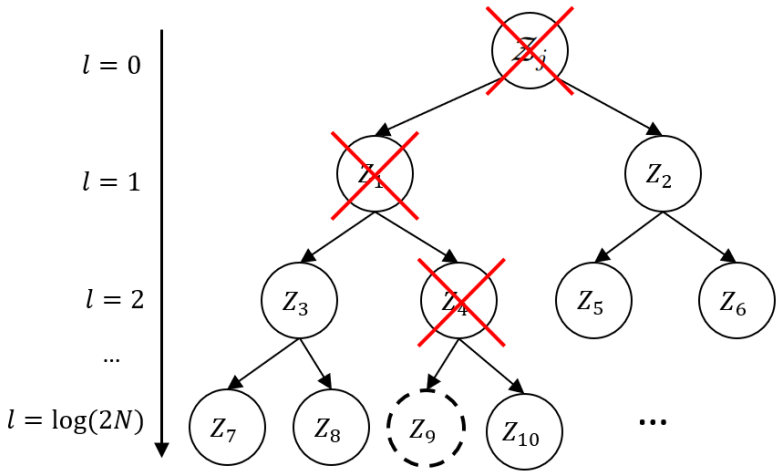


Figure 4.2 Binary tree and zone splitting.

The key idea of zone splitting is to identify the zone that performs worst in terms of model utility and split it from an original merged zone so that the zones after splitting perform better than the original zone. Specifically, ZMS recursively split sub-zones of a merged zone, which have the worst model utility, such that the splitting optimizes model utility across all sub-zones. Each of the merged zones $\mathcal{Z}_j \in \mathcal{Z}$ is a set of sub-zones $\{Z_i\}_{i\in[1,N]}$ represented by a binary tree of zone merging history, as illustrated in Figure 4.2. Each internal node in the tree represents a merged zone from its two sub-zones (child nodes). Each leaf node is an indivisible zone.

At each training round, ZMS randomly selects a binary tree representing a merged zone $\mathcal{Z}_j$ to check for a potential zone splitting. ZMS considers all the internal nodes up to level l as potential sub-zones to split. For instance, if $l = 2$ in Figure 4.2, we consider $\{Z_i\}_{i\in[1,6]}$ as candidates for the zone splitting (Algorithm 4, Line 1). We select top-k sub-zones having inferior model utility (i.e., higher losses compared with the merged zone $\mathcal{Z}_j$) (Algorithm 4, Lines 7-12). If a candidate zone Z_c trained independently achieves better model utility, i.e., $\mathcal{L}(\boldsymbol{\theta}_c^{t+1}, Z_c) < \mathcal{L}(\boldsymbol{\theta}_j^{t+1}, Z_c)$ where $\boldsymbol{\theta}_c^{t+1}$ is the zone model trained on Z_c and $\boldsymbol{\theta}_j^{t+1}$ is the model trained on the merged zone $\mathcal{Z}_j$ (Algorithm 4, Line 4), then ZMS splits Z_c from the merged zone $\mathcal{Z}_j$ (Algorithm 4, Line 5). In a training round, ZMS permits at most one zone splitting (Algorithm 4, Line 6)

to minimize the overhead and avoid distributed consistency problems. All ancestor nodes of Z_c are removed, creating a set of new merged-zones and their associated binary trees. For instance, in Figure 4.2, if we split zone Z_9, we create a set of new merged zones, including zones $Z_3 = Z_7 \cup Z_8$, $Z_2 = Z_5 \cup Z_6$, Z_9, and Z_{10}. By doing this, we focus on keeping the best merges after a zone splitting; thus approximating the zone splitting objective (Equation 4.3) without affecting the zone merging objectives (Equation 4.1 and 4.2). The training and validation at the phones for split is done in a similar way with the ones for merge.

Algorithm 5 Zone Gradient Diffusion with Self-Attention

Input: Zone Z_i

1: $\mathcal{N}_i \leftarrow$ ***getNeighbors***(Z_i)
2: **for** $Z_n \in \mathcal{N}_i$ **do**
3: $\quad e_{in} \leftarrow \sigma\big(\boldsymbol{\nabla}(\boldsymbol{\theta}_i^t, Z_i) \bullet \boldsymbol{\nabla}(\boldsymbol{\theta}_i^t, Z_n)\big)$ # where "$\bullet$" is an inner product
4: $\forall Z_n \in \mathcal{N}_i : \beta_{in} \leftarrow \dfrac{\exp(e_{in})}{\sum_{Z_j \in \mathcal{N}_i} \exp(e_{ij})}$ # computing coefficients
5: $\boldsymbol{\theta}_i^{t+1} \leftarrow \boldsymbol{\theta}_i^t + \boldsymbol{\nabla}(\boldsymbol{\theta}_i^t, Z_i) + \sum_{Z_n \in \mathcal{N}_i} \beta_{in} \boldsymbol{\nabla}(\boldsymbol{\theta}_i^t, Z_n)$ # aggregating gradients from neighboring zones

4.1.4 Zone Gradient Diffusion (ZGD)

In addition to ZMS, we propose ZGD, an algorithm that keeps the zones fixed but adapts the model by aggregating contextual information derived from local gradients of neighboring zones (Algorithm 5). We found that contextual information captures changes in mobility patterns and significantly improves the utility of zone models. In ZGD, at round t, the neighboring zones Z_n of a zone Z_i derive their local gradients using the model parameters $\boldsymbol{\theta}_i^t$ from the zone Z_i by using local data D_u from their users u, as follows: $\boldsymbol{\nabla}(\boldsymbol{\theta}_i^t, Z_n) = 1/|U_n| \sum_{u \in U_n} \boldsymbol{\nabla}(\boldsymbol{\theta}_i^t, D_u)$. Note that users u compute the gradients $\boldsymbol{\nabla}(\boldsymbol{\theta}_i^t, D_u)$ and send the gradients to the zone manager Z_n for data privacy protection.

Intuitively, the more similar the gradients of a zone ($\boldsymbol{\nabla}(\boldsymbol{\theta}_i^t Z_i)$) are with the gradients of a neighboring zone ($\boldsymbol{\nabla}(\boldsymbol{\theta}_i^t, Z_n)$), the higher the impact of the neighboring

zone Z_n on Z_i will be. We quantify this impact through self-attention coefficients β_{in} by normalizing the inner product of the local gradients of the zone Z_i and its neighboring zones $Z_n \in \mathcal{N}_i$:

$$\forall Z_n \in \mathcal{N}_i : \beta_{in} \leftarrow \frac{\exp(e_{in})}{\sum_{Z_j \in \mathcal{N}_i} \exp(e_{ij})} \tag{4.4}$$

where $e_{in} = \sigma\big(\boldsymbol{\nabla}(\boldsymbol{\theta}_i^t, Z_i) \bullet \boldsymbol{\nabla}(\boldsymbol{\theta}_i^t, Z_n)\big)$, σ is the sigmoid function, and "$\bullet$" is an inner product.

Finally, we aggregate the gradients from neighboring zones to update the zone model $\boldsymbol{\theta}_i^t$ at round t:

$$\boldsymbol{\theta}_i^{t+1} \leftarrow \boldsymbol{\theta}_i^t + \boldsymbol{\nabla}\big(\boldsymbol{\theta}_i^t, Z_i\big) + \sum_{Z_n \in \mathcal{N}_i} \beta_{in} \boldsymbol{\nabla}\big(\boldsymbol{\theta}_i^t, Z_n\big) \tag{4.5}$$

By doing so, ZGD updates the zone models to diffuse contextual information from one zone to all the remaining zones across training rounds. This operation significantly enriches the information used to optimize zone models in ZoneFL, compared with existing FL algorithms.

4.2 System Design and Implementation

4.2.1 System Architecture

The ZoneFL architecture has three main components, as shown in Figure 4.3: *(1) FL Phone Manager* coordinates the ZoneFL activities on the phone; *(2) FL Zone Manager* coordinates the ZoneFL activities at the edge; and *(3) Zone Partition Keeper* maintains and provisions the latest zone partition information in the cloud. The edge software components of the architecture can be mapped either to edge nodes

or to servers in the cloud. For example, some FL Zone Managers could be deployed at the edge nodes where edge is available, while others can be hosted in the cloud where edge is not available yet. The FL Zone Manager can be migrated between the cloud and the edge nodes.

The software components work together to support the six phases of ZoneFL: data collection and preprocessing, privacy protection, model training and aggregation, mobile apps using models for inference, zone partition maintenance, and zone partition adaptation to user mobility changes. The first four phases follow traditional FL. The Data Collector stores the sensed data in the Raw Data Storage and informs the FL Phone Manager each time new data is added to the Raw Data Storage. The FL Phone Manager decides invokes the model-specific Data Processors and stores the data in the Processed Data Storage. The Local Privacy Preserving Manager uses differential privacy techniques to further preserve user privacy. The Model Trainer performs local training on the phone, and the Model Aggregator aggregates the gradients at the edge. A Publish-Subscribe edge service, New Model/Zone Partition Notification Service, allows the phone to receive asynchronous notifications when a new zone model is available. When an app needs inference from a model, it sends a request to the FL Phone Manager using the OS IPC mechanisms. In response, the FL Phone Manager generates the input for the inference from the data stored in the Processed Data Storage, and then it invokes the Model Runner with this input. The Model Runner sends the result to the App using IPC. Next, we explain the two phases that are specific to ZoneFL.

Zone Partition Maintenance. The Zone Partition Keeper maintains the latest zone partition information in the system, which is represented as a graph. Each non-overlapping zone is a vertex, and each edge connects two neighboring zones. The initial zone partition information is bootstrapped by the administrator of the system based on administrative divisions of a region. The Zone Partition Keeper is also

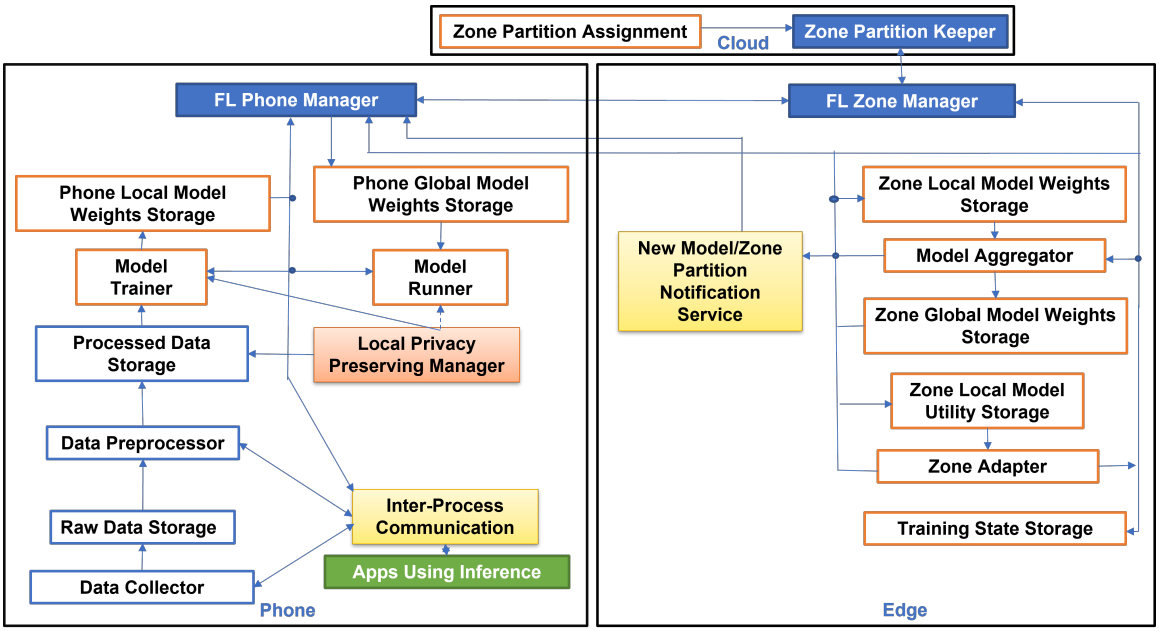


Figure 4.3 System architecture.

responsible for maintaining information about the identity (e.g., IP addresses) of the FL Zone Managers at the edge.

Initially, a phone receives the zone partition information from the Zone Partition Keeper. Then, it maps its data to different zones, based on the geographic locations where the data were collected. This determines the list of zones to which the phone subscribes for training. The phone communicates with the FL Zone Managers of these zones to jointly train the zone models. For inference, a phone may use a zone model even if the phone did not participate in the training of the given zone. This allows new users to quickly benefit from ZoneFL.

Zone Partition Adaptation. The Zone Adapter of each edge node is responsible for dynamic adaptation of zone partitions in ZMS. In order to perform merge and split, as described in Subsection 4.1.3, the system needs to perform zone level model validation. This operation is done through the cooperation of the phones and the edge manager. The FL Zone Manager maintains a Zone Local Model Utility Storage for phones to report the model utility computed on their validation datasets, and periodically aggregates the validation results. This process involves additional

communication between phones and the FL Zone Manager, but it mitigates potential privacy issues, since data never leaves the phone.

4.2.2 ZoneFL Prototype Implementation

We implemented an end-to-end ZoneFL prototype on Android phones and AWS cloud. This prototype, with ZMS for dynamic adaptation, was used in our field study, described in Section 4.3. AWS offers AWS Local Zones [180] as its edge computing service. However, it is not available yet in the area of our field study, and therefore the edge components of ZoneFL are deployed in the AWS cloud. We chose Deep Learning for Java (DL4J) as the underlying framework for DL-related operations, because it is a mature framework that supports model training on Android devices.

Deployment and Operation Scripts. The system administrator prepares the initial zone partition information as a geojson file, which defines the zones' geometry as polygon coordinates. We implement Python scripts to deploy and operate the system. These deployment script reads the geojson file provided by the system administrator to create an independent FL Zone Manager for each zone in AWS. The operation scrips are used to collect performance and reliability data.

FL Zone Manager. The core computing components of the FL Zone Manager are implemented and deployed as AWS Lambda functions [164] for low overhead and fast start time. We create a REST API to relay clients' requests to participate in the FL training to the Lambda function that handles these requests. We also use the AWS EventBridge to define rules to trigger and filter events for Lambda functions. For model storage, model utility storage, validation datasets, and configuration files, we use AWS S3. To store data that is accessed frequently, such as training round states and model states, we use AWS DynamoDB. AWS SNS is utilized in conjunction with the Google FCM to notify clients when newly trained models are ready. Most FL Zone Manager components interact only with components within their zone. The

only exception is the Zone Adapter, which communicates with its counterparts in neighboring zones to implement ZMS. As public cloud providers are racing to deploy edge computing infrastructure [180,181], we expect these cloud services or their edge-based variants will soon be available at the edge.

Zone Partition Keeper. We use an AWS S3 bucket as the Zone Partition Keeper of all zones. This is the only shared AWS resource in the system. All the other AWS resources are independent among different zones. In this way, once edge computing becomes more widespread, the FL Zone Manager can be migrated from the cloud to the edge. The latest zone partition information is made available to phones for download. The previous partition information is also stored for the Zone Adapter to help with the split operation in ZMS.

Android implementation. The Android phone implementation consists of three apps: FL Phone Manager, Data Collector, and Testing App (used to test model inference). The Data Collector was implemented starting from ExtraSensory [182]. This app collects heart rate (HR) sensing data from a Polar HR tracking wrist band [183], which connects to the phone over Bluetooth. In the FL Phone Manager, the Data Preprocessor uses the geojson file with Android Google Map API to check the zone to where each data point belongs to. Then, the Data Preprocessor generates the model input for training. The Model Trainer is implemented with the Android native *AsyncTask* class to ensure the trainer is not terminated by Android, even when the app is idle. The Model Trainer communicates with the FL Zone Manager of each zone to train the models sequentially. Model inference is implemented as a background service with Android Interface Definition Language (AIDL), and it gets inference requests from the Testing App. This app uses *AidlConnection* to interface with the FL Phone Manager for the inference results.

4.3 Evaluation

The evaluation presents results for both model utility and system performance. The model utility experiments have two goals: **(i)** Compare the performance of ZoneFL with Global FL (i.e., traditional FL trained with all users globally); **(ii)** Quantify the benefit of ZGD and ZMS. The system experiments have four goals: **(i)** Demonstrate the feasibility of ZoneFL on smart phones; **(ii)** Investigate ZoneFL scalability; and **(iii)** Quantify the ZoneFL phone training time overhead.

4.3.1 Datasets, Models, and Metrics

We use two datasets collected in the wild to evaluate two types of ZoneFL models: **(1)** A human activity recognition dataset [184]; and **(2)** A heart rate dataset [185]. We choose these two datasets because we observe the advantages of ZoneFL with these two real-world mobile sensing applications we have data. The attributes, other than zone, that affect the prediction are handled by the model design.

Human Activity Recognition (HAR). The dataset has data from 51 users, moving in a region larger than 20,000 km^2. Each user provided mobile accelerometer data, GPS coordinates, and labeled their daily activities on their personal Android phones. The labels used in the experiments are "Walking," "Sitting," "In Car," "Cycling," and "Running." In the experiments, we start with 9 non-overlapping zones over the region covered by the dataset, based on GPS coordinates. The zones are diverse and include a university campus zone, several suburban residential zones, a riverside urban zone, a metro zone, etc. On average, each user have 1,995 data samples for each zone. The preprocessing and the CNN-based model architecture follow the work associated with the dataset [184]. For this classification task, we use accuracy as the main metric for model performance.

Heart Rate Prediction (HRP). The dataset contains 167,373 workout records for 956 users in 33 countries. The data collected by the users using their

mobile/wearable devices include multiple sources of sequential sensor data such as heart rate, speed, GPS, sport type, user gender, and weather conditions. We filter the data such that users with at least 10 workouts are included in training and inference (4:1 split). We exclude users having less than 10 workouts because those data points are not significant. To evaluate ZoneFL, we assign the initial zones of each country to its principal (largest) administrative divisions so that we can have a manageable number of zones. Among the countries, we select the top 6 countries having at least 10 zones with a reasonable number of average data samples per zone to effectively assess ZoneFL's performance. We use an LSTM-based model [185] to predict the heart rate given input features consisting of altitude, distance, and time elapsed (or speed) of the workouts. For this prediction task, we use the root mean squared error (RMSE) metric. We only use HRP to evaluate zone dynamic adaptation because HRP dataset has sufficient number of zones and users.

4.3.2 Model Utility Results

ZoneFL vs. Global FL. Table 4.1 shows the performance comparison between ZoneFL and Global FL. In this experiment, ZoneFL works only with the zones defined at the beginning of the experiment (called Static ZoneFL), without employing ZMS or ZGD to adapt the models to the user mobility behavior over time. Thus, it provides a lower bound on ZoneFL's performance, which is expected to improve with ZMS and ZGD. Global FL trains with all the users in the datasets. Zone FL trains a different model for each zone in the respective dataset. Some users have data and participate in training in more than one zone. The metrics are computed per user in the test data set and then averaged. ZoneFL models outperform the Global FL models by 6.67% for HAR and by 6.74% for HRP. This performance gain is significant given that it is very challenging for DL models to achieve 1% improvement in HAR and

Table 4.1 ZoneFL vs. Global FL

Application	Metrics	Global FL	Static ZoneFL	Improvement Gain
HAR	Accuracy (%)	65.27	69.63	6.67%
HRP	RMSE	21.20	19.86	6.74%

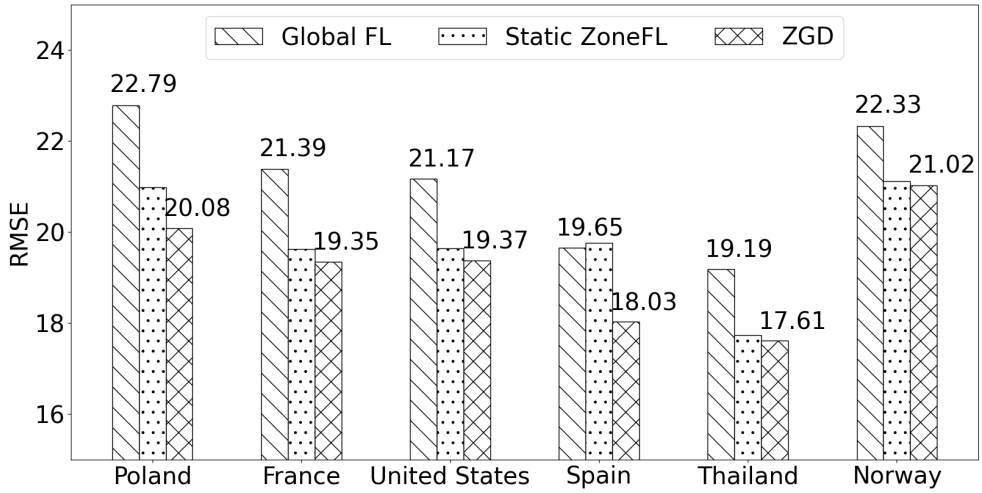


Figure 4.4 Simulation results of global FL and ZoneFL algorithms.

HRP tasks as illustrated in recent studies [73,186]. As shown next, we observe further improvement with the dynamic adaptation algorithms.

ZGD Performance. Although ZGD and ZMS adapt zone models to user mobility behavior changes, they serve slightly different purposes. Hence, we present the performance of ZGD and ZMS separately. ZGD is designed to work with fixed zones that have enough data for training. ZMS is designed to adapt the zone partitions until all of them achieve reasonable model performance. In practical terms, ZMS is generally used for the beginning rounds of ZoneFL, while ZGD is used once the zone model performance is relatively stable. For both algorithms, we show just the results for HRP because its dataset is more suitable for dynamic adaptation by having more zones.

Figure 4.4 shows the performance of ZGD for the top-6 countries in the HRP dataset. ZoneFL with ZGD performs better than Static ZoneFL for each country, and it clearly outperforms Global FL (by as much as 11.89% for Poland). We also observe that Static ZoneFL performs better than Global FL for 5 countries, and slightly worse

Table 4.2 ZMS Improvement

	Before (RMSE)	After (RMSE)	Improvement Gain (%) Mean / SD	Occurrence Per 100 Rounds
Merge	23.79	21.44	9.87 / 3.11	4
Split	23.04	20.71	11.10 / 3.63	3

for one country. The reason for the worse performance for Spain is that the static zones do not capture well the changes in user mobility behavior. ZoneFL with ZGD is able to alleviate this problem and result in better performance than Global FL.

ZMS Performance. Table 4.2 shows the average model performance improvement for (zone) merge and split in HRP. In merge, the improvement gain is calculated as follows: $\frac{L_1+L_2}{2} - L_{12}$, where L_1 and L_2 are RMSE losses evaluated on the two constituent zones, and L_{12} is RMSE loss computed on the merged zone. The reverse formula is used for splitting a larger zone in two sub-zones. The results demonstrate that ZMS can significantly improve the model performance. On average, 4 merges and 3 splits occur every 100 rounds of training, which shows that dynamic adaptation needs to happen about once a month in a scenario where users train once a day.

4.3.3 System Results

To showcase the feasibility and advantages of ZoneFL over Global FL in a real-life deployment, we conducted an HRP field study with 63 users for 4 months. Along with smart phone sensor data such as accelerometer, gyroscope, etc., the users were tasked to collect heart rate data from a Bluetooth-connected heart rate tracking wrist band for their daily activities. The region of the field study is larger than 20,000 km^2, and it was originally divided in 9 zones. The study ran the prototype of ZoneFL with ZMS, described in Subsection 4.2.2. In the field study the ZMS split operation is performed for only one level ($l = 1$, Subsection 4.1.3). The prototype worked reliably throughout the duration of the field study. Next, we present experimental results for our prototype.

Table 4.3 Training on Phones: Resource Consumption and Latency

Application	Phone	Max RAM Usage (MB)	Foreground Training Time Mean/SD (min)	Background Training Time Mean/SD (min)	Battery Consumption per Round (mAh)	Number of Training Rounds for Full Battery
HAR	Nexus 6P	232	15.21/2.89	59.99/4.06	53.86	64
	Google Pixel 3	228	2.13/0.24	9.32/0.09	9.91	294
HRP	Nexus 6P	266	3.09/0.39	10.97/1.08	33.18	104
	Google Pixel 3	230	0.40/0.10	5.07/0.37	4.66	625

ZoneFL Feasibility on Smart Phones We benchmarked ZoneFL with HAR and HRP on Android phones using a testing app to evaluate training and inference performance. We also assessed the resource consumption on the phones, with different specs (Nexus 6P and Google Pixels 3). The results demonstrate the on-device feasibility of ZoneFL, even for the Nexus 6P phone, unveiled in 2015 and running Android 7. Since ZoneFL works well on such a low-end phone, we expect ZoneFL to work well on most of today's phones.

Training Performance. Table 5.7 shows the ZoneFL training time and resource consumption on the phones. The training time is recorded by training 1995 samples and 86 samples (i.e., the average numbers of samples per zone per user) in 5 epochs for HAR and HRP. Foreground training (screen turned on) provides a lower bound for the training time by using the full single core capacity. In reality, we expect training to be done in the background, while the phone is being charged. We take 10 measurements for each benchmark and report the mean and standard deviation since other apps or system processes working in background may interfere with the training.

Training for one round is fast on the phones. The foreground training time on Pixel 3 is just 2.13 min for HAR, and 0.4 min for HRP. The background training time is also good for any practical situation. The background training time is notably longer compared with foreground training, since Android attempts to balance computation with battery savings.

Table 4.4 Inference on Phones: Resource Consumption and Latency

Application	Phone	Max RAM Usage (MB)	Foreground Inference Time Mean/SD (millisecond)	Background Inference Time Mean/SD (millisecond)	Battery Consumption per prediction (μAh)	Millions of inferences for Full Battery
HAR	Nexus 6P	161	54.65/16.36	1963.04/1540.29	4.49	0.77
	Google Pixel 3	177	36.59/6.43	99.60/33.69	1.94	1.50
HRP	Nexus 6P	232	528.93/53.53	1809.71/700.96	45.47	0.08
	Google Pixel 3	229	167.71/6.83	669.88/112.01	5.74	0.51

The results also show training is feasible in terms of resource consumption. The maximum RAM usage of the app is less than 266MB, and modern phones are equipped with sufficient RAM to handle it. The phones could easily perform hundreds of rounds of training on a fully charged battery. It is worth noting that, typically, one round of training per day is enough, as the users need enough time to collect new data.

Inference Performance. The results in Table 5.6 demonstrate that ZoneFL can be used efficiently by third-party apps working in real-time. The inference time is measured within the third party testing app. Let us note that the inference is performed locally by the FL Phone Manager, without any network communication. Thus, the measured time consists of the inference computation time and the inter-process communication time. We continuously perform predictions/classifications for 30 minutes and report the average values. The inference time for the two scenarios on the third-party app, foreground and background, follows a similar trend as training.

Scalability ZoneFL utilizes multiple FL Zone Managers to receive and aggregate the gradients from the users. Compared with a single server in Global FL, the communication and computation load in ZoneFL is distributed among multiple zone servers. Considering a user may send gradients to multiple zone servers, Table 4.5 computes the average ZoneFL server load savings based on the user percentage distribution over the number of zones in Figure 4.5. The results demonstrate ZoneFL

Table 4.5 Server Load in ZoneFL over Global FL

Application	HAR	HRP
ZoneFL server load	37.26%	34.98%

Table 4.6 ZMS in The Field Study

Merge Time	Zone X/ RMSE	Zone Y/ RMSE	Merged Zone RMSE
2022-04-09 13:57	A/13.96	B/18.40	12.56
2022-05-29 12:53	C/44.53	D/11.86	10.84
2022-06-05 13:07	E/18.48	A/15.28	13.30
2022-07-29 21:56	F/17.40	G/39.23	14.78

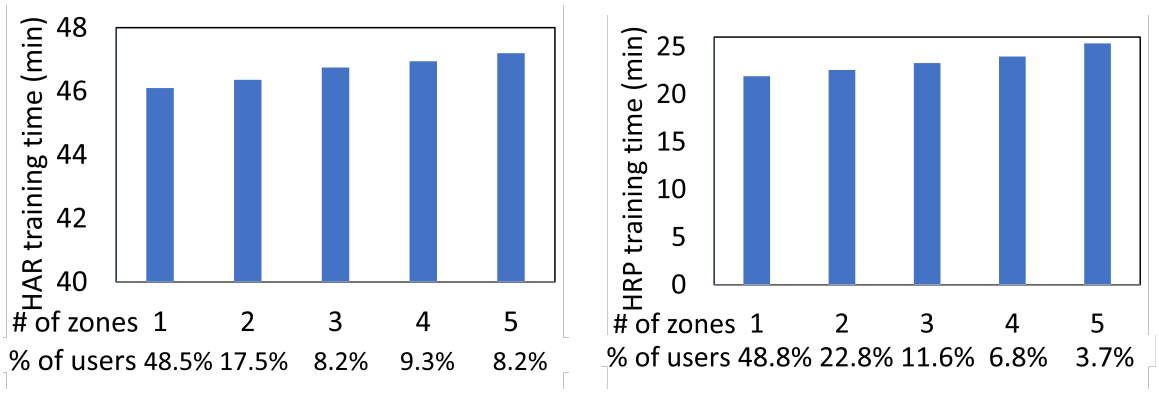


Figure 4.5 User training time vs. number of zones in the user data.

scales better than Global FL because the server load is 34.98% to 37.26% of the one in Global FL.

ZMS Performance in the Field Study Table 4.6 depicts zone merge time and model utility gains in the field study. At the end of the field study, the number of the zones was changed from 9 to 7 after several merges and splits. In ZMS, a merge occurs when the merged model performs better than both individual zone models. The highest model utility gain observed is to improve RMSE from 44.53 to 10.84. This is because the original zones did not have enough users and data. We also observed two splits happened during the filed study. The highest RMSE improvement for split is from 16.38 to 11.20. These observations showcase the ZMS improvements in our ZoneFL prototype deployed in real-life.

ZoneFL User Training Time Overhead In ZoneFL, the phones may have data from and may train in multiple zones, which may introduce a certain level of overhead compared with Global FL. For every round in Global FL, a phone trains once for all its data. In ZoneFL, a phone may train multiple times (once per zone from where it has data), but for a smaller fraction of data. Figure 4.5 illustrates the background training time in Android when the phone trains the same amount of data, while varying the number of zones the data are distributed to. The percentage of users shown under the X axis represents the fraction of users that have data in [1, 5] zones (e.g., 8.2% of users have data in 5 zones). The number of samples trained per zone follows the average reported in Subsection 4.3.3. For the 49% of the users that have data in a single zone, there is no overhead compared to Global FL (i.e., train once with all the data). For the rest of the users, we observe a small overhead, which increases with the number of zones. However, the training time overhead never exceeds 3.5 minutes. Considering that the training occurs in the background, this is an acceptable overhead for the benefits of ZoneFL in terms of model utility and server scalability.

4.4 Chapter Summary

This paper proposed ZoneFL, a mobile-edge-cloud FL system, that distributes training across geographical zones to improve model utility and scalability compared with traditional FL. We augmented ZoneFL with two training algorithms, ZMS and ZGD, enabling zone models to adapt to changes in user mobility behavior. ZMS and ZGD can work complementary during FL training rounds, with ZMS improving model utility in the initial rounds and ZGD further improving the utility after that. Using two different models, including human activity recognition and heart rate prediction, and mobile sensing datasets collected in the wild, we showed that ZoneFL outperforms traditional FL in terms of model utility and server scalability. We implemented an

Android/AWS prototype of ZoneFL with ZMS and demonstrated the feasibility of ZoneFL in real-life conditions.

CHAPTER 5

FEDERATED META-LOCATION LEARNING FOR FINE-GRAINED LOCATION PREDICTION

The goal of this chapter is to design a system for fine-grained location prediction from GPS traces that works on the users' phones. In our work, the term fine-grained refers to both spatial and temporal scales. Specifically, we aim to achieve high prediction accuracy, with prediction errors within the range of GPS errors. Furthermore, we want our system to be able to predict any potential locations of the users, not just important places identified by Place IDs as it is done by existing works. We focus on pedestrians and bicyclists, instead of users in cars or in public transportation systems, because their less predictable behavior makes the problem more difficult. In addition, their lower speeds and ability to stop whenever they want are expected to enable more applications of predictions. We also want to be able to predict at minute-scale (e.g., predict with a temporal step of one minute for $1, 2, ..., n$ minutes ahead). For example, we want to predict where a pedestrian will be in 5 minutes with a 10m spatial error.

We propose *Federated Meta-Location Learning (FMLL)* on smartphones for fine-grained location prediction using GPS traces collected on the devices. FMLL consists of three main components: a meta-location generation module, a prediction model, and an FL framework. The meta-location generation module represents user location data as relative points in a 2D space, enabling learning across diverse physical environments. The prediction model combines Bidirectional Long Short-Term Memory (BiLSTM) and CNN. BiLSTM captures mobile users' speed and direction, while CNN learns additional information such as user movement preferences. The framework operates on both user devices and a central server that coordinates learning across all participants in the system. FL is employed in FMLL to protect user privacy and reduce bandwidth consumption.

This chapter presents the meta-location generation in Section 5.1. Section 5.2 details the prediction model. Section 5.4 describes the datasets and the meta-location preprocessing. Section 5.3 presents FMLL framework. Section 5.5 shows the experimental evaluation. Section 5.6 discusses real-life deployment aspects and additional use cases for our model. The chapter is summarized in Section 5.7.

5.1 Meta-Location Generation

The fundamental information to predict location is travel direction and speed. The user movement preferences and road characteristics also help the prediction. The GPS trajectories of each user contain this information. FMLL on the phones process the raw location data to generate the meta-location, which represents trajectories as relative points in an abstract 2-Dimensional (2D) space. This section presents the process of meta-location generation and its benefits.

5.1.1 Raw Location Data

The raw location data is recorded by each phone using the embedded GPS sensor. Let $\mathbf{L_t} = \langle \mathbf{lat_t}, \mathbf{lon_t} \rangle$ denote the latitude and longitude of a user at time t. FMLL performs learning based on the transportation mode, such as walking or bicycling. Therefore, only the data specific to the desired transportation mode is selected for further processing. In real world, if the transportation mode is not explicitly known, it can be inferred from accelerometer data on mobile devices [187].

5.1.2 Meta-Location Input for Prediction Model

The raw location data of each user is processed on their smart phone to produce meta-location as two types of inputs for the prediction model: *fixed-length sequences of relative points* and *historic region occupancy matrices* of the space considered for prediction. The input sequences contain the speed and direction information of the user trajectories. The occupancy matrices record frequently visited places and the

most likely trajectories between these places. The inputs are computed offline (e.g., when the phones are charging) and can be updated over time based on new data to enable re-training.

To generate the input sequences, FMLL splits the user trajectories into fixed-length sub-trajectories. The length in time of the trajectories is determined experimentally. Each sub-trajectory is transformed into a sequence of relative points in an abstract 2D space. The X and Y coordinates of relative points at time t are determined based on their offsets from the location at previous time step t-1. The location of the very first point in a trajectory session is excluded. A location offset is denoted as $\mathbf{\Delta L_t} = \langle \mathbf{lat_t} - \mathbf{lat_{t-1}}, \mathbf{lon_t} - \mathbf{lon_{t-1}} \rangle$. An input sequence of at time t that looks back k steps is denoted as $\mathbf{S_t} = (\mathbf{\Delta L_{t-k+1}}, \mathbf{\Delta L_{t-k+2}}, ..., \mathbf{\Delta L_{t-1}}, \mathbf{\Delta L_t})$. In its training, FMLL considers all possible k-length sequences, including overlapping sequences.

The historic region occupancy matrices are extracted from a historic occupancy matrix of the entire space (e.g., a city). FMLL divides the entire space into a grid of fixed-size cells, and each cell corresponds to an element in the historic occupancy matrix. Each element represents the number of visits of the user in its corresponding cell. The matrix represents the occupancy of a bounded region $\mathbf{R_t}$ with area A, which is centered at the physical location $\mathbf{L_t}$ at time t. $\mathbf{R_t}$ is divided into $M \times M$ fixed-size grid-cells, where A and M are predefined constants based on the maximum speed of users and the desired spatial granularity for the prediction. Each historic region occupancy matrix $\mathbf{H_t}$ is a $M \times M$ matrix, and it is extracted from the historic occupancy matrix for the entire space. Once extracted, this matrix is a meta-location input that does not maintain any relation with the physical locations that it represents. A matrix can implicitly tell if a road exists in a given cell (i.e., non-zero value for the corresponding matrix element) and can also tell if adjacent cells form routes taken frequently by the user.

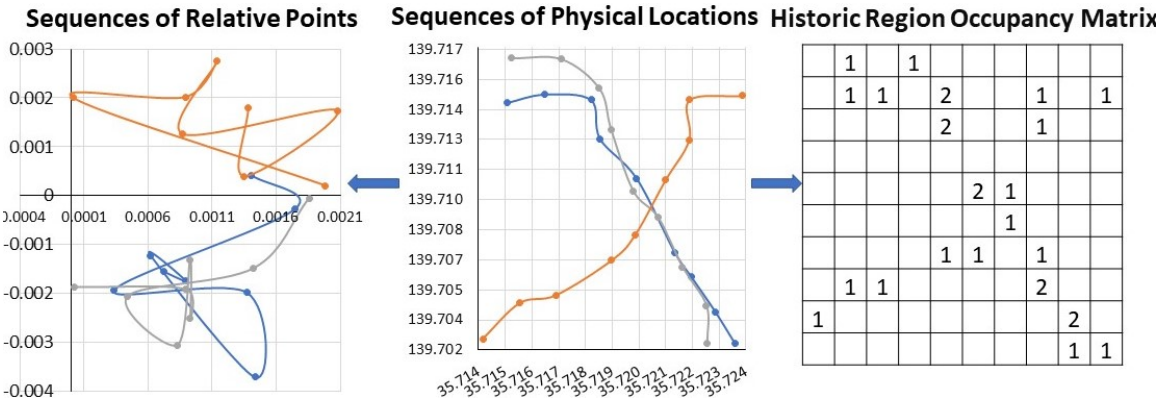


Figure 5.1 Illustration of meta-location generation.

Figure 5.1 illustrates how the meta-location input is generated from the sequences of physical locations. Let us note that in real world, the historic region occupancy matrix is generated from the numbers of visits to all grid cells which do not have to be temporal sequences of physical locations. We observe that the sequences of relative points do not resemble the physical sequences, which helps with location privacy protection. Overall, different physical locations can be mapped to the same meta-locations. This not only helps repeated patterns to be extracted from different physical locations, but also make it difficult for adversaries (i.e., the server in FL) who attempt to infer the physical locations.

5.1.3 Meta-Location Output for Prediction Model

The location to be predicted $\mathbf{L_{t+i}}$ is mapped into the region $\mathbf{R}$. FMLL builds a prediction matrix $\mathbf{Y_{t+i}}$ (Equation 5.1).

$$\mathbf{y_{i,j,t+i}} = \begin{cases} 1, & if\ \mathbf{L_{t+i}} \in \mathbf{R_{i,j}} \\ 0, & otherwise \end{cases} \tag{5.1}$$

where $\mathbf{y_{i,j,t+i}}$ is an element of $\mathbf{Y_{t+i}}$, $\mathbf{R_{i,j}}$ $(\mathbf{1} \leq \mathbf{i}, \mathbf{j} \leq \mathbf{M})$ is a cell in region $\mathbf{R}$. The meta-location output is formulated as a categorical class rather than a numerical

value, so that we can set the spatial granularity of the prediction as a constant. Another reason for using categories is that the historic region occupancy matrix does not contain information to predict with spatial granularity beyond the grid-cell size. Overall, the output is a relative grid-cell, which is translated into a physical grid-cell on the user's phone.

5.1.4 Meta-Location Benefits

Using meta-location has four benefits. First, different meta-location sequences have the same magnitude, which is required for DL data. DL algorithms minimize the distance between two data points as a loss function. During this minimization, high-magnitude data weighs more than low-magnitude data, and it can lead to bias. For example, if physical location sequences are used directly, the training will focus on minimizing the loss for the data with high latitude and high longitude values. One way to avoid this problem is to scale every sequence to the same range [188]. However, scaling the location sequence will remove the traveling speed, which is necessary for location prediction. Since speed cannot be assumed constant for accurate prediction, there may not be an efficient mechanism to preserve it. With our meta-location generation, all meta-location sequences are in the same range and can be used directly in DL. This is especially important for FL training across all users.

The second benefit is the ability to change configuration parameters in data representation to perform prediction at different spatial granularity. For example, the grid-cell size can be 10m $\times$ 10m for pedestrians, and 40m $\times$ 40m for bicyclists. Instead of predicting the coordinates as arbitrary numerical values without a target granularity, with meta-location, we can formulate the output of the model categorically with specified granularity, and use accuracy to quantify the model performance. This also improves the model utility when an application requires certain spatial and temporal granularity from the model.

The third benefit is location privacy protection. This is achieved in conjunction with FL, which shares only the model gradients with the server. The data do not leave the phones because the learning happens on the phones. However, the gradients of the local models may still leak private location information if the FL model uses physical location data [75]. This problem is substantially mitigated by the use of meta-location. The meta-location input contains the essential information for location prediction, including speed, direction, and user movement preference, while not disclosing the physical location ($\mathbf{L_t} = \langle \mathbf{lat_t}, \mathbf{lon_t} \rangle$) of the user to DL model.

The fourth benefit is the extraction of repeated patterns across different physical locations. When learning directly from different physical locations, the DL models encounter entirely different samples. However, because meta-location uses the same abstract 2D space, it may become similar for different physical locations. This speeds up learning because there will be more similar meta-location samples.

5.2 FMLL Model

This section presents the formal problem definition for our model, and the description of the model architecture and its components.

5.2.1 Problem Definition

The problem is defined based on the meta-location input and output, defined in Section 5.1. Let $\mathbf{S_t} \in \mathbf{R}^{2k}$ be the size-k sequence of relative points at time t for a given user. Let $\mathbf{H_t} \in \mathbf{Z}^{+M \times M}$ be the historic regional occupancy matrix of the same user, which is a square matrix of order M centered at the user location at time t. Our goal is to predict the relative location of this user $\hat{\mathbf{Y}}_{\mathbf{t+i}} \in {\mathbf{Z_2}}^{M \times M}$ for the future i^{th} timestamp.

$$\hat{\mathbf{Y}}_{\mathbf{t+i}} = \mathbf{F}(\mathbf{S_t}; \mathbf{H_t}) \tag{5.2}$$

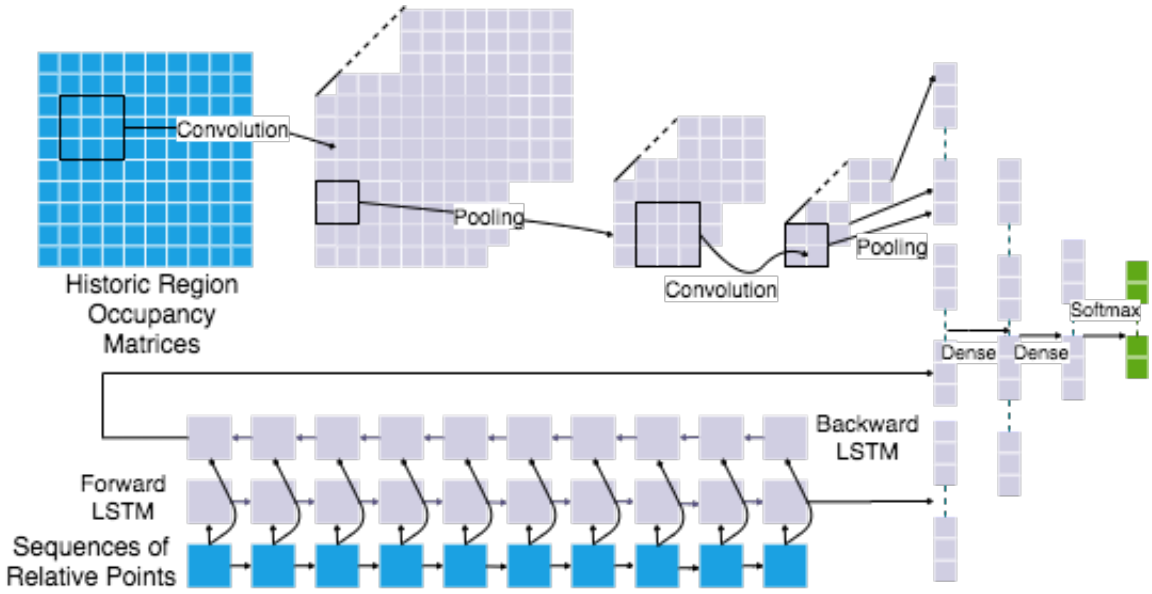


Figure 5.2 Model architecture.

where **F** is our DL model for location prediction. The predicted location is a cell in the $M \times M$ grid representation of the space surrounding the current location of the user.

5.2.2 Model Architecture

Meta-location provides a novel input and output formulation for the location prediction problem. Existing models cannot used directly for three reasons. First, they [29, 30, 32, 36, 37] are designed to use either sequence input or matrix input, instead of both, so they are not able to take advantage of the benefits offered by the complementary meta-location inputs. Second, some problems, such as POI check-in prediction [36, 37], are naturally different from fine-grained location prediction. The user may check-in anywhere and anytime, and consequently their problem formulation does not bound the input and output spatially or temporarily. Their output candidates can be any POI IDs in the dataset, while fine-grained location prediction output can only be in the neighboring range of the current location. Third, the heuristics of designing some location prediction models is different from designing a fine-grained location prediction model. For example, the check-in location

prediction [36, 37] should be designed to learn the popularity of POIs, while the fine-grained location prediction should learn the speed and direction of movement, road characteristics, etc.

To learn effectively for fine-grained location prediction, we propose FMLL model. As shown in Figure 5.2, the model fuses BiLSTM and CNN, where BiLSTM learns the speed and direction of the user mobility from the sequences of relative points, and CNN learns user movement preferences and likely user routes from the historic region occupancy matrix. BiLSTM and CNN work in parallel. A densenet-type connection is used to fuse BiLSTM and CNN, and then softmax activation is adapted to output which grid-cell the user will be in. Batch normalization and dropout layers are also added in both BiLSTM and CNN to avoid over-fitting, but for simplicity they are not shown in the figure. This architecture is designed to capture as much user-level information as possible.

For training, the phones use the meta-location input derived from physical locations, which can be pre-computed. For prediction, the sequence input is simple and can be generated in real-time, based on the last k recorded GPS locations. The historic region occupancy matrix, centered at the current location, is extracted in real-time from the pre-computed historic occupancy matrix for the entire space.

BiLSTM. Sequences of relative points contain the information of travel speed and direction. FMLL uses BiLSTM to learn them with a targeted spatial and temporal granularity. An LSTM unit is composed of a cell, an input gate, an output gate, and a forget gate. The cell remembers values over arbitrary time intervals, and the three gates regulate the flow of information into and out of the cell. In BiLSTM, one LSTM reads the relative location sequence forward, while a second LSTM reads it backward. The final two layers of hidden states are then concatenated, and the concatenation of these layers captures the speed and direction of users. To avoid overfitting, we perform both regular dropout and recurrent dropout. FMLL adopts

BiLSTM for three reasons: (1) LSTM works well for sequence modeling. (2) BiLSTM augments data by using backward sequences in training. While backward sequences are not part of the dataset, they are real sequences that could occur. (3) Unlike unidirectional LSTM which leads to a final internal state containing more information about the last points of a sequence (while the information about the first points is forgotten) [29], BiLSTM preserves the sequence information equally across the relative points.

CNN. Intuitively, knowing the exact speed and direction can determine the next location. However, the predicted speed and direction has to be tuned with other information for better learning. FMLL uses CNN on the historic region occupancy matrices, associated with the sequences fed into BiLSTM, to capture spatial features such as user movement preferences or the likelier route followed by a user between two points. CNN can learn this type of information because the historic region occupancy matrices contain information reflecting both occupancy frequency (explicit) and movement trajectory (implicit). Our CNN consists of batch normalization, convolution, max pooling, RELU activation, and dropout. With help from convolution and pooling, CNN is able to capture local connectivity and shift invariant. In our model, local connectivity can be the direction to which a user prefers to turn at a given intersection. The road characteristics are shift-invariant because the road network in a city usually follows the same urban design, and is similar in different areas.

Fusion. Although sequences of relative points and historic region occupancy matrices can be fit into next location by BiLSTM and CNN respectively, fusing the complementary information learnt from them can significantly improve the model performance. FMLL fuses the output layers from BiLSTM and CNN by concatenation, which allows for different-length outputs from BiLSTM and CNN. Then the concatenated output is fed into fully connected densenets, and the final

output is computed by *softmax* activation, as shown in Equation 5.3, where $\mathbf{k}$ corresponds to the $\mathbf{k_{th}}$ grid-cell, n = M × M is the total number of grid-cells, $\hat{\mathbf{y}}_\mathbf{k}$ is the $\mathbf{k_{th}}$ element of the output $\hat{\mathbf{Y}}$, and $\phi_\mathbf{k}$ is the $\mathbf{k_{th}}$ element of the final hidden layer before activation. The dense layers can gradually extract features of our desired length, and softmax converts them into probabilities. The output $\hat{\mathbf{Y}}$ contains the predicted probabilities of the future user location in each grid cell.

$$\hat{\mathbf{y}}_\mathbf{k} = \frac{\exp(\phi_\mathbf{k})}{\sum_{\mathbf{i=1}}^{\mathbf{n}} \exp(\phi_\mathbf{i})} \tag{5.3}$$

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} -\frac{1}{n} \sum_{i=1}^{n} \mathbf{y_i} \cdot \log(\hat{\mathbf{y}}_\mathbf{i}) \tag{5.4}$$

FMLL uses cross entropy loss for optimization, which is a standard function in multi-class classification. Therefore, the model learns the parameter $\mathbf{w}$ by minimizing the cross entropy loss measurement (Equation 5.4). $\mathbf{y_i}$ is the $\mathbf{i_{th}}$ element of the ground truth $\mathbf{Y}$, where the grid cell of the user's future location is set to 1, and all others are 0.

5.3 FMLL Learning Framework

This section describes the FMLL framework that enables training and inference, while preserving the privacy of the user location. The section presents the system architecture for the framework, the operation stages of FMLL, and an enhanced training method.

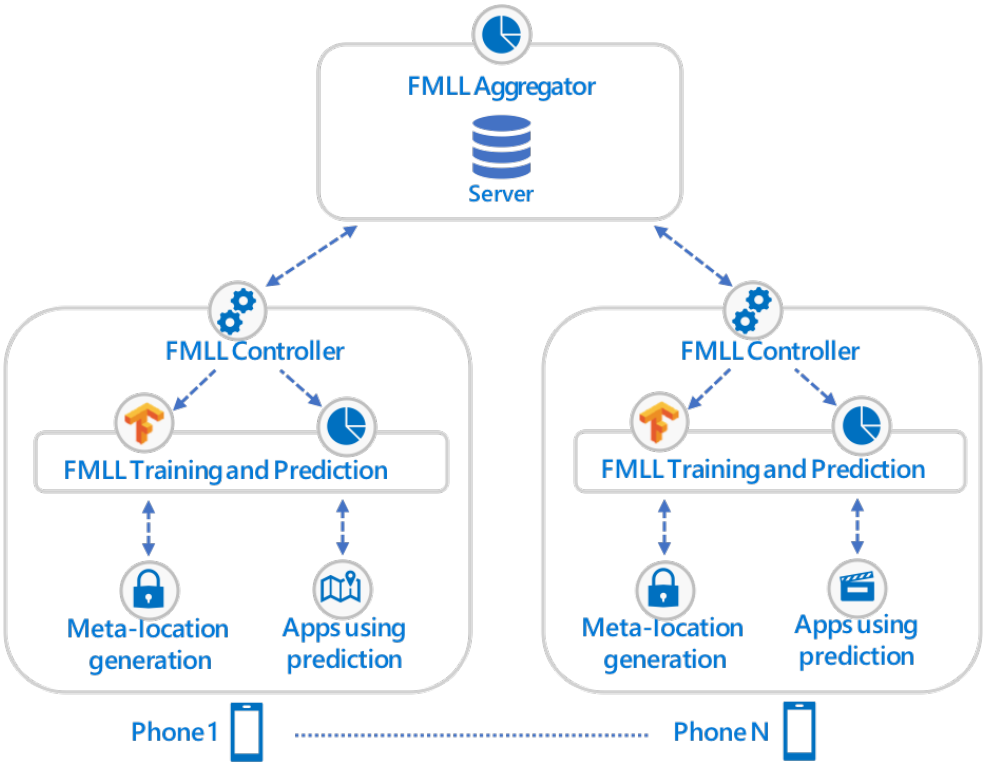


Figure 5.3 FMLL aystem architecture.

5.3.1 System Architecture

The system architecture of our framework is shown in Figure 5.3. The framework software runs on a server and on the phones of the users, and it uses federated learning (FL) [74] for training across all users. The FMLL Controller on the phones mediates the communication between the server and the phones. The Meta-Location Generation module on the phones processes the physical location data and generates meta-location for training. The FMLL Training and Prediction module runs on the phones. This module performs local model training on the phones and then submits the model gradients to the server through the Controller. The FMLL Aggregator module at the server aggregates the gradients of the local models into a global model, and then distributes this model to the phones. When the OS or apps need a prediction, the Training and Prediction module is invoked. The output of the prediction is a meta-location, which is then converted into a physical location, with help from Meta-Location Generation module.

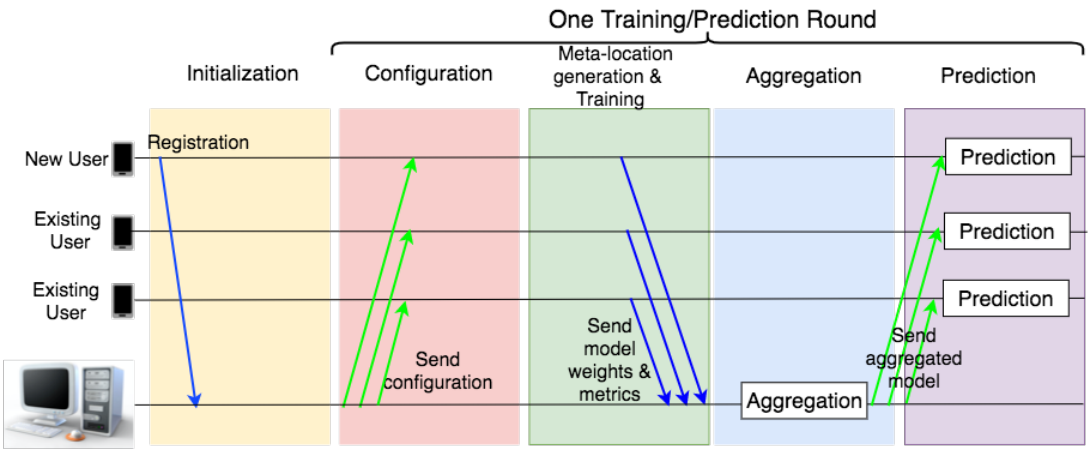


Figure 5.4 Federated Learning operation of FMLL.

5.3.2 Operation Stages

To deploy the model in real-world and evolve it while the users collect location data over time, the FMLL learning framework has five computation and communication stages, illustrated in Figure 5.4. During these stages, the phones and the server jointly contribute to the model. The stages are executed periodically in rounds, similar to Google's FL framework [74]. In each round, the model is fine-tuned by re-training from the existing model. In the following, we detail each stage.

(1) *Initialization.* Newly participating phones are required to register with the server to ensure that the server knows when model gradients uploaded at different times come from the same user. This could further allow the server to remove potential malicious users who may inject fake data into the model. [1]

(2) *Configuration.* A training/prediction round starts with the configuration stage. The server informs the phones of the deadline to participate in training. This deadline is the end of stage 3. The server can select a subset of the connected phones based on the optimal number of participating phones in each round and the availability of training data. The server sends configuration parameters to the phones on how to generate meta-location for training. Parameters, such as sequence length, matrix size, grid size, may vary according to the desired spatial accuracy of the prediction. The server also sends the current global model parameters to each phone that did not participate in the previous training round, along with a training plan, such as gradient computation settings.

(3) *Meta-location Generation and Training.* Based on the configuration from the server, the phone performs meta-location generation. Then, it uses the global model received from the server to compute gradients based on its processed data. Finally, the phone sends the gradients back to the server.

[1] Protection against such malicious users is outside the scope of the study.

(4) *Aggregation.* This is the main server computation stage. The server waits for the phones to report gradient updates, aggregates them using federated averaging, and updates its global model weights with aggregated gradients. Then, it deploys the model to the phones to ensure they have the latest model because they may not participate in a new round for a while.

(5) *Prediction.* The software on the phone can invoke the new FMLL model for predictions. Up to this stage, they use the old model from the previous round. This stage has a much larger duration than all the others. For example, training can be done once a day for a few minutes, while predictions could be performed at any time.

5.3.3 Training with Data Augmentation

A well-reported issue that restricts the performance of FL models is non Independent and Identically Distributed (IID) data distribution. FL trains on the dataset of each individual user. The datasets among different users may follow different distributions, due to user behavior differences, imbalanced class distribution, etc. While DL training can efficiently converge models with IID data, models trained in FL settings usually suffer from inferior performance [15]. To mitigate the non-IID issue, we leverage an advanced data augmentation mechanism inspired by Zhao et al. [4].

In this enhanced FL training, the data from a small percentage of users (e.g., less than 5%) are allocated as an augmentation dataset and made available to the aggregation server, and the server can sample and share it with the other users. This is usually the case when a small amount of users are willing to share their data with the server [189].

Training with data augmentation has three phases. First, the FMLL model is trained with the augmentation dataset at the server. This model is then distributed to the phones that will participate in FL training. Second, each phone selected in every round randomly picks a certain number of samples from the augmentation dataset and concatenates them with its own dataset. Third, on-device training is conducted by initializing the model received from the server (trained with the augmentation dataset) and further training it with the augmented local data. The rest of the FL procedures are the same as in the basic FL technique. Thus, the local non-IID data

of each user are augmented with IID data from the augmentation dataset, and the non-IID issue is mitigated.

5.4 Dataset and Meta-location Preprocessing

This section describes the two real-world public datasets that we use in our evaluation, as well as the meta-location preprocessing to extract the inputs and outputs expected by our system.

5.4.1 Dataset Description

We use two datasets: Open PFLOW [190] and Geolife [191]. Most experiments will use Open PFLOW, which is much larger. Geolife is used to demonstrate model reusability. Open PFLOW contains GPS trajectories that cover the Greater Tokyo area. It includes GPS data for 617,040 users, sampled every minute. The dataset covers typical movement patterns of people in the metropolitan area for one day, and it includes five transportation modes: stay, walk, vehicle, train, and bicycle. We select the data labeled "walk" and "bicyle", because their lower speeds and ability to stop whenever they want are expected to enable more applications of predictions. In real world, the transportation mode can be inferred from accelerometer data on mobile devices [187]. Unless specified otherwise, the experiments are for pedestrian mobility. Bicycle data are used to demonstrate model reusability by using the pedestrian model to predict on bicycling data directly. Geolife [191] contains GPS trajectory data for 182 users over a five-year period. From this dataset, we selected the users (73) who labeled their trajectories with walking.

5.4.2 Meta-location Preprocessing

We choose a 200m $\times$ 200m region for both historic region occupancy matrices input and grid output, assuming a user can walk up to 100 meters in a minute. Let us recall that the region is centered at the current location of the user. For each experiment,

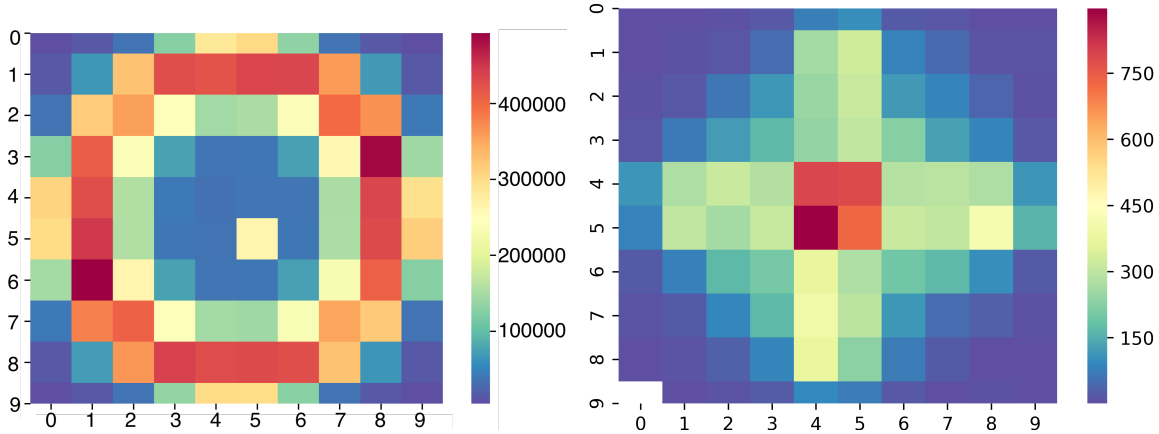


Figure 5.5 Heatmap of possible next minute location in Open PFLOW (left) and Geolife (right) for 20m × 20m grid cells.

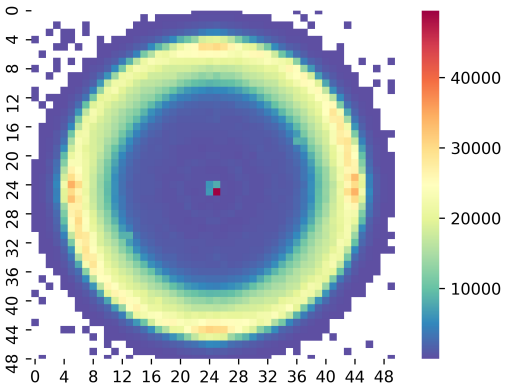


Figure 5.6 Open PFLOW 5th min prediction heatmap of 20m × 20m grid cells.

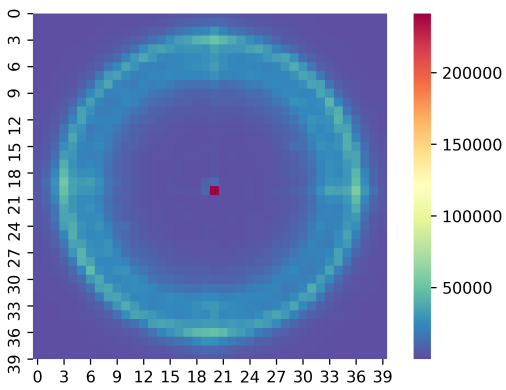


Figure 5.7 Open PFLOW next minute prediction heatmap of 5m × 5m grid cells.

we divide the region based on the accuracy we want to obtain. For example, we can divide the region into 100 cells of size 20m × 20m, and predict the location in one of these cells. The historic occupancy matrix is computed once and saved on mobile devices. The historic region occupancy matrix, centered at the current location, is extracted in real time from the overall occupancy matrix.

For the fixed-length sequences of relative points, we choose a length of 9 (i.e., 9 locations, sampled one minute apart). Any walking session less than 9 minutes is dropped from the input. The sequence length should contain enough information to capture the direction and speed of movement, but it should not be too long, delaying the time when the first prediction can happen. We selected $length = 9$ as a good

trade-off. In addition, most people in our datasets walk fewer than 15 minutes at once, so our length is practical from this point of view. To achieve quicker convergence, the relative points are scaled between -1 and 1.

Both datasets contain a large number of standing-still datapoints. We consider an input sequence to be standing-still if the user does not move at all in 9 minutes (the sequence length). Even though there is a "stay" transportation mode, about 50% of the walk data is standing-still. Our experiments presented in Section 5.5 are without standing-still data. However, we also ran experiments with standing-still data (not included in the paper for the sake of brevity). These experiments proved the imbalance introduced by standing-still data is not a problem for large amounts of data.

Figure 5.5 shows the heatmap of the next minute possible location in Open PFLOW and Geolife after removal of standing-still data. The heatmap is generated by counting the number of samples in the dataset leading to each grid cell output. Since walking speed is similar for most people, in Open PFLOW, the location of next minute is most likely in the red/orange circle. In Geolife, because of fewer samples (three orders of magnitude fewer than in Open PFLOW), the circular pattern from typical walking speeds becomes less evident. The heatmap also illustrates the average number of samples per class in Open PFLOW is sufficient (i.e., 250,000). However, in Geolife, it is only 450 on average, and a lot of classes have less than 100 samples, which may be inadequate for training. Nevertheless, we keep this dataset to test if our model can be reused by leveraging transfer learning.

An accurate location prediction model requires large amounts of data. As we increase the region size to predict further into the future (Figure 5.6) or decrease the grid cell size for higher accuracy (Figure 5.7), the number of grid cells increases quadratically. Therefore, the number of cells with insufficient samples increases correspondingly, and the prediction accuracy for both may suffer.

5.5 Experimental Evaluation

Our evaluation has three goals. First, we assess the model's performance without FL. This allows a fair comparison between our model and the baseline models, which do not include privacy protection techniques such as FL. Second, we measure the performance of the federated solution of FMLL. Third, we test the feasibility of running FMLL on phones.

5.5.1 Model Performance Without FL

Implementation FMLL is implemented using the Keras library. The output space of BiLSTM has dimensionality of 512 in each direction. Regular and recurrent dropout rates are set to 0.2 in BiLSTM. Two blocks of CNN are used sequentially with output space of 256 and 512, respectively. Both blocks include a convolutional layer with a 3×3 convolution window, a stride of 1, RELU activation, maximum pooling layer with size 2×2, batch normalization, and a dropout rate of 0.2. The outputs of BiLSTM and CNN are concatenated and fed into four densenet layers with output spaces of 1024, 512, 256, and 128, respectively. A dropout of 0.2 is added between the four dense layers. The final output is fitted by a densenet layer with softmax activation. We utilize the ADAM optimizer with learning rate of 0.001. The loss function is chosen as categorical cross entropy, which is commonly-used for multi-class classification.

Metrics We choose three metrics for prediction performance: cross entropy loss, categorical accuracy, and weighted F1 score (i.e., the weighted average of F1 scores of each class divided by the number of samples in each class). A high F1 score indicates that both precision and recall are high.

Baseline models Because the problem formulation of FMLL is unique and FMLL is designed to learn from meta-location instead of actual physical location, there are few comparable baseline candidates. We choose the baselines that can be

adapted to learn from meta-location: Bidirectional RNN [29], T-CONV [30], and HO. Bidirectional RNN [29] is the best model in a taxi destination prediction competition. The state-of-the-art T-CONV model, used by Lv et al. [30], has recently outperformed the Bidirectional RNN in the taxi destination prediction. Highest Occupancy Model (HO) is a simple statistical prediction model. In HO, the location in the next minute is predicted as the most occupied grid cell in the historic region occupancy matrix. If there are multiple grid-cells with the same highest occupancy, HO will randomly select one of them.

To adapt the baseline models to be comparable with FMLL, we use the same meta-location input as FMLL, which are fixed-length sequences of relative points and historic region occupancy matrices. The dimensions of the hidden states and window sizes are chosen to be same as in FMLL, and eventually fed into the same output layers. In this way, the baselines can also demonstrate the contributions of each individual component of FMLL, and the benefits of fusing them.

We considered and tested two other state-of-the-art POI ID prediction models for fine-grained location prediction, ST-RNN [36] and DeepMove [37]. These models use recall@k as metrics, which is the the proportion of correct predictions found in the top-k recommendations. The values of recall@10 when applying these models to fine-grained location prediction are less than 5%.

The performance of these models is so poor in our settings because they are very different from FMLL in their problem formulation and design. First, these models use physical locations, instead of meta-locations. With physical locations, there are few similar trajectory samples across users. Without repeated patterns, the deep learning models cannot learn. Second, to predict the POI ID in a given time frame, these models formulate the problem as a recommender system that predicts a ranking of possible POI IDs a user will check in. On the other hand, our problem is formulated as a precise binary prediction of whether the user will be at a location

cell in the grid or not. Third, these models do not consider spatial granularity, because a POI ID, such as a mall, typically covers a large area. They also do not consider temporal granularity, as they predict the next POI IDs without knowing when the user will visit that POI. Unlike these models, FMLL bounds the spatial and temporal granularity in the prediction and learns features such as speed and direction of travel, user preferences, and road characteristics. FMLL is unique in its fine-grained prediction across both spatial and temporal scales.

Experimental Settings The experiments are conducted on a Ubuntu Linux cluster (CPU: Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz with 512GB memory, GPU: 4 NVIDIA P100-SXM2 with 64GB total memory), and the training and testing run with GPU acceleration. The ratio of training, validation, and testing datasets is 4:1:1. We choose a batch size of 256. Early stopping is used to stop the training earlier if the accuracy has not improved in the last 10 epochs for the validation dataset. The users are simulated in this system by replaying their location traces.

Results Unless specified otherwise, the experiments are for pedestrian mobility, using Open PFLOW, and predicting one minute ahead for 20m $\times$ 20m grid-cells. As discussed in Subsection 1.2.2, such short-term prediction can have significant benefits in real life.

Comparison with baseline models. Table 5.1 shows the prediction metrics for FMLL without FL and the baseline models. FMLL outperforms the baselines in both loss and accuracy. The weighted F1 score indicates that both precision and recall are high in FMLL. We notice that neither BiRNN nor T-CONV performs well. However, by fusing BiLSTM with CNN, FMLL leads to substantially better performance. This is because each of them captures a different type of information: speed and direction of traveling (BiLSTM) and movement trajectory and preferences (CNN).

Table 5.1 Performance of FMLL w/o FL and Baselines

Model	Metrics		
	Loss	Accuracy	Weighted F1
HO	NA	0.329	0.382
BiRNN	2.859	0.222	0.188
T-CONV	0.900	0.501	0.469
FMLL	**0.157**	**0.955**	**0.955**

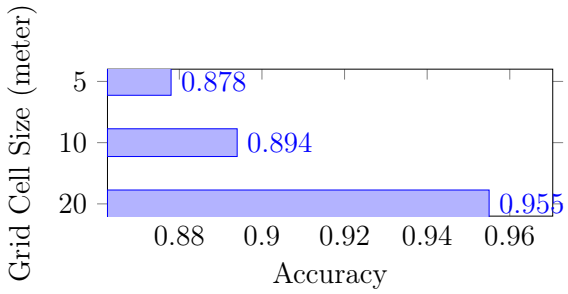


Figure 5.8 Prediction accuracy as a function of grid-cell size.

Performance vs. spatial scale. Figure 5.8 shows how FMLL's accuracy varies with spatial scale (i.e., grid-cell size). Even though the accuracy decreases as the size of the cell decreases, FMLL can still achieve good performance (i.e., 87.8% accuracy) for 5m $\times$ 5m cells.

Performance vs. time scale. Figure 5.9 shows how FMLL's accuracy varies over time (i.e., predict x minutes ahead). The performance decreases over time, but we believe it is acceptable up to 5 minutes ahead. In addition to loss accumulation, there are two reasons for the decrease in accuracy over time. First, the user data is just for one day, and we cannot capture many recurring mobility patterns. Second, pedestrian mobility is inherently difficult to predict. Nevertheless, short-term prediction can be beneficial to many applications. For example, adjusting bit rate video streaming based on 5G coverage prediction can allow buffering up enough video while coverage is still good to avoid quality degradation due to predicted poor coverage in the next few minutes.

Model Reusability. A model is reusable when it can be used directly or in conjunction with transfer learning (TL) on another dataset. Currently, this property

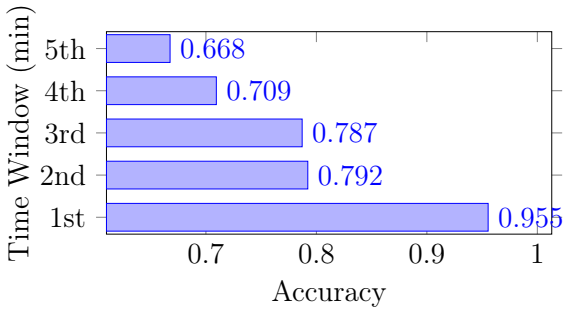


Figure 5.9 Prediction accuracy as a function of time windows.

Table 5.2 FMLL w/o FL Pre-trained on Pedestrian Data and Used to Predict on Bicycling Data w/o TL

Model	Metrics		
	Loss	Accuracy	Weighted F1
Walking	0.157	0.955	0.955
Bicycling	0.955	0.853	0.849

exists mainly in image recognition and natural language processing because few models can satisfy it.

We demonstrate that our model pre-trained on the pedestrian data from Open PFLOW works for bicyclist data from the same dataset, even without TL. We also demonstrate that TL works well in conjunction with FMLL when applied to the Geolife dataset. We believe that the main reason for FMLL's reusability is its meta-location, which makes it less location-specific.

Table 5.2 shows the performance obtained when testing the pre-trained FMLL (i.e., trained only with pedestrian data) on bicycling data. For comparison, we also show the accuracy when testing on pedestrian data. Because bicycling speed is approximately four times walking speed, the sequence input is scaled down four times and the matrix input is scaled up four times to match the magnitude of pedestrian data. The results show good performance, with an accuracy of 85.3%, which demonstrates the model's reusability.

Table 5.3 shows the performance of FMLL over the Geolife dataset in two cases: a model trained directly on Geolife, and a TL model that pre-trained with Open PFLOW and fine-tuned with Geolife. The results demonstrate the reusability of

Table 5.3 FMLL Performance on Geolife dataset: Geolife alone vs. TL from Open PFLOW to Geolife

Model	Metrics		
	Loss	Accuracy	Weighted F1
Geolife alone	2.291	0.400	0.390
TL from Open PFLOW to Geolife	2.190	0.448	0.441

Table 5.4 FMLL with FL Performance

Training method	Metrics		
	Loss	Accuracy	Weighted F1
FMLL w/ FL without data augmentation	3.016	0.664	0.631
FMLL w/o FL on augmentation dataset alone	2.652	0.792	0.815
FMLL w/ FL with data augmentation	**2.432**	**0.842**	**0.853**

FMLL because the TL model from Open PFLOW to Geolife achieves 12.2% higher accuracy than the model trained on the Geolife dataset alone. This result is surprising, but it is explained by the fact that Open PFLOW is a much larger dataset than Geolife. We also observe, as expected, that training on Geolife leads to low accuracy. While the accuracy of the model pre-trained on Open PFLOW is significantly better, it is still not good in absolute terms. The reason is that the two datasets cover very different road networks. There are two types of urban planning for road networks, either as grid or circles, and these two types are quite different. Therefore, a model trained on one type cannot work very well on the other type.

5.5.2 Model Performance with FL

Implementation and Settings FMLL with FL is implemented using TensorFlow Federated (TFF). For simplicity, we used the default federated averaging algorithm in TFF [74], instead of more sophisticated averaging mechanisms that may help FL's accuracy [192]. The same cluster is used for the experiments but we could not use GPU acceleration because current version of TFF (0.9.0) is not optimized for GPUs.

To simulate user participation in one FL round, we randomly sample 320 users every round for training. Before the training, we set aside some random users' data for testing, and these users are not selected for training. Because the number of samples per user is very limited (from 1 to 150, with an average of 60 in Open

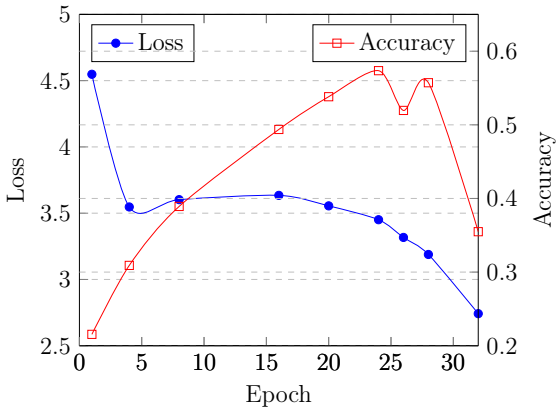


Figure 5.10 Loss and accuracy over epochs with 40 rounds of training.

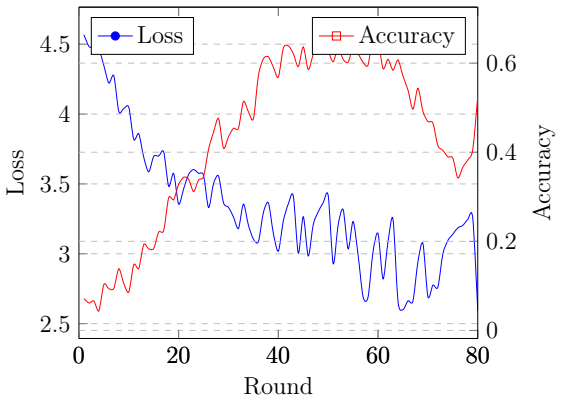


Figure 5.11 Loss and accuracy over rounds with 24 epochs of training.

PFLOW), the batch size is set to 32. The number of users per round and the batch size are set experimentally for the best performance. All the other model settings are the same as in FMLL without FL.

Results Due to the lack of a centralized validation dataset in FL, it is difficult to know when underfitting or overfitting occurs. Also, the lack of validation makes it impossible to determine the optimum number of epochs by early stopping. Therefore, in this experimental evaluation, we first show the model performance as the number of training epochs increases. Next, with the optimum number of epochs to train FMLL, we train the model with an increasing number of rounds for the best performance possible.

Performance over number of epochs. Figure 5.10 shows the testing performance of the models trained for 40 rounds and varying the number of epochs. Similar to the training for the model without FL, the loss decreases as the number of training epochs increase because the deep learning iteration process aims at minimizing the loss. Initially the accuracy increases as the loss decreases. However, overfitting happens as the number of epoch keeps increasing, and accuracy decreases. The figure shows that the the best number of epochs is 24.

Performance over number of rounds. Figure 5.11 shows the testing performance of FMLL with FL models over rounds, with he number of training epochs fixed at 24. Unlike training over epochs, the performance fluctuates over training rounds. This is because every round is a sampling process based on the available phones, and there is no guarantee that the data quality from the sampled phones meets the demand to further improve the model. The fluctuation is minor at the beginning, but it increases over time. Similar with training over epochs, the loss decreases as the number of rounds increase due to re-training and the larger amount of sampled data. However, accuracy may not be further improved after a certain number of rounds because the model becomes overfitted. We observe that round 59 produces the model with highest prediction accuracy.

Performance with data augmentation. Table 5.4 shows the performance of FMLL with FL. This experiment uses the training with data augmentation described in Subsection 5.3.3. During training, for each user selected in every round, we randomly pick 100 samples from the augmentation dataset, which are concatenated with the user dataset. The best performance is achieved in the 4th round while training 20 epochs per user. FMLL with FL using data augmentation improves the accuracy by 20% over the original FMLL with FL. These results demonstrates that our data augmentation mechanism can effectively improve the performance of the FGFL with FL model.

5.5.3 Model Benchmarks on Smart Phones

Inference Performance

Implementation and Settings. The prototype model used so far was developed in TensorFlow. While TensorFlow and TensorFlow Lite training are not supported in Android, we can download the server-trained model on the phone to perform prediction. Since the TensorFlow model is heavier than versions developed

Table 5.5 Smart Phone Specs

Phone Model	Android Version	Battery Capacity (mAh)	CPU (GHz)	RAM (GB)
ZTE Blade V8 Pro	6	2213	8x2.0	3
Huawei Nexus 6P	7	2986	4x1.55; 4x2.0	3
Google Pixel 3XL	10	3082	4x2.5; 4x1.6	4

in TensorFlow Lite or DL4J, the performance and resource consumption of such a model act as upper bounds for lighter model implementations.

We converted the server-trained model to a TensorFlow protocol buffer (pb) file, and the model takes 67MB on the phone. We developed a benchmark app on Android to load the model and perform predictions for a given input.

The maximum memory usage is verified using the Android Profiler in Android Studio. The running time per prediction is calculated by averaging 100 predictions. After leaving the app running in background for 2 hours to perform predictions in a loop, the battery consumption is computed from the percentage of power usage measured by Battery Historian [2]. The battery statistics are reset after switching the app to the background to make sure the app foreground usage is not included.

Results. We test the model on three phones with different specs, as shown in Table 5.5. The results in Table 5.6 demonstrate low resource consumption and good prediction latency. The maximum RAM usage of FMLL is less than 200MB. The Google Pixel 3XL, a mid-range phone in 2020, can make a prediction in 46.41ms, which is fast enough for all practical purposes. On the same phone, the battery consumption per prediction is 0.0238μAh, and 129 million predictions can be executed with a full battery. With further optimization, such as model pruning and compression, the resource consumption can be even lower. These results demonstrate that the model can be deployed effectively in real life.

Training Performance

[2]https://github.com/google/battery-historian

Table 5.6 Inference Resource Consumption and Latency

Phone Model	Maximum RAM Usage (MB)	Running Time per Prediction (ms)	Battery Consumption per Prediction (μAh)	Millions of Predictions for Full Battery
ZTE Blade V8 Pro	187	207.47	1.87	1.18
Huawei Nexus 6P	192	79.68	1.11	2.69
Google Pixel 3XL	167	46.41	0.238	12.93

Implementation and Settings. Since TensorFlow and TensorFlow Lite do not support on-device training currently, we implement FMLL in DL4J. The model takes 50MB in a zipped bin and json format. We develop a benchmark app on Android to load the initial model received from the server and perform one round of training from pre-processed data. We implement the training with Android native AsyncTask class, which will not be terminated by Android even when the app is idle. The same tools and methods as in the previous experiment are used to measure resource consumption.

The training time is recorded by training 200 samples for 20 epochs, which is the optimum scenario determined in Subsection 5.5.2. Foreground training is done while leaving the screen on, and it uses the full single core capacity. It provides a lower bound for the training time.

As in real world, other apps or system processes may interfere with training. We take 10 measurements for each benchmark, and report the mean and standard deviation.

Results. Table 5.7 shows the training time and the resource consumption on the phones. The results demonstrate that training is feasible in real-life, even without optimizing the models. The maximum RAM usage of FMLL is less than 753 MB, and current phones are equipped with sufficient RAM to handle it. We observe quite different training times among different phones. The foreground training time demonstrates the full computation capacity of a single core, and the most powerful

Table 5.7 Training Resource Consumption and Latency

Phone Model	Maximum RAM Usage (MB)	Foreground Training Time Mean/SD (min)	Background Training Time on Charger Mean/SD (min)	Background Training Time on Battery Mean/SD (min)	Battery Consumption per Round (mAh)	Number of Training Rounds for Full Battery
ZTE Blade V8 Pro	743	6.84/0.30	6.81/0.04	7.54/1.09	87.52	25
Huawei Nexus 6P	753	9.94/0.94	8.94/0.32	44.90/13.52	114.63	26
Google Pixel 3XL	592	2.65/0.34	14.34/0.10	375.49/126.57	15.16	203

phone, Pixel 3XL, has the shortest time. We do not report battery consumption on foreground, as the screen consumes a much larger amount of energy that training.

The background training time on charger, which is the expected situation for FL training, is reasonable in practice. On two phones, the training time is similar with the foreground time; only one phone experiences a significantly higher training time. The explanation for the newer phone model performing worst is related to the newer Android version it uses (10 vs. 6 and 7 on the other two phones). According to Google, from Android 9, to ensure that system resources are given to the apps that need them the most, the system limits apps' access to device resources like the CPU or battery based on the user's usage patterns.

We also performed experiments for background training on battery. The results on the older phones are reasonable in terms of both training time and battery consumption. However, the training time on the new phone is not acceptable for the same reason discussed above. We also observe that the training time for foreground and background on charger varies much less than background training time on battery. This is because once the task goes to background, other apps or system processes are more likely to interfere with the training process over a longer period of time.

To conclude, the results show that FMLL with FL is feasible in practice. Typically, the model would not have to be trained more than once a day because it needs enough new data to participate in a new round of training. Our results show that all phones can perform this training in the background while charging in less than 15 min.

5.6 Discussion

This section discusses issues related to real-life deployment of FMLL and expands on the use cases presented so far.

So far, all our use cases involved software running on the phones only. However, location prediction could also be used by Internet services accessed from the phones. An example is represented by enhanced location-based services (LBSs) that use predicted user locations instead of current user locations. Such an LBS could increase the revenue associated with location-based ads by delivering only relevant ads on the user's predicted path. The advantage of FMLL for such a use case is that users do not have to share location traces with enhanced LBSs, but just predictions computed on the phones.

In FMLL, there is a trade-off between prediction accuracy and privacy. Specifically, the accuracy is higher when FMLL does not use FL. In such a case, the relative input data is submitted to the server, which performs training and prediction. The output of the prediction is an abstract grid-cell, which is sent back to the phone. The phone can translate this abstract grid-cell into a physical grid-cell. In addition to higher accuracy, this use of FMLL reduces the computation and battery power needed on phones when compared with FMLL using FL. Furthermore, without additional external information, the server cannot learn the physical location of the user because the server handles only relative data.

Nevertheless, more sophisticated location inference attacks are possible at the server if FL is not used. If the attacker learns one physical location in a sequence by physically observing the user, it is easy to determine the whole sequence. Then, based on the sequence, the attacker may infer the user identity [48, 49] and link it to the visited locations.

The FMLL model can also be used for service providers that have access to user location traces. Two examples are cellular network operators or map services

that provide turn-by-turn routing to destinations. While the abstract representation of data in FMLL is not necessary for privacy protection in this use case because the providers already know the physical locations of the users, FMLL is expected to lead to better prediction performance than models using physical data. This is because its abstract data representation eliminates the training bias introduced by physical location data.

Network operators can use accurate location prediction in 5G networks at every time scale and across all layers of the protocol stack [28]. Minute-scale location prediction can benefit many applications, including proactive caching, load balancing, scheduling, synchronization, topology, power control, resource allocation, and handover [28]. At the physical layer, by tracking and predicting the change in mobile user movements, the weights of antenna elements can be dynamically optimized for best signal coverage while incurring minimum interference from other users. Furthermore, our model can easily be configured for different spatial and temporal scales, which may allow providers to use differently configured models for different cities or different days of the week.

Premium services provided by network operators can take advantage of accurate location predictions to free up resources at crowded target locations ahead of time to ensure the best network performance for premium customers when they reach these destinations. The Open Radio Access Networks (O-RAN) [193] propose to optimize RAN resources to provide better network connectivity to smart phone users. One such optimization can be to intelligently take advantage of accurate location prediction of users.

5.7 Chapter Summary

This chapter propose Federated Meta-Location Learning (FMLL) on smart phones for fine-grained location prediction, based on GPS traces collected on the phones. FMLL

uses FL framework with two additional components: a meta-location generation module, a prediction model. The meta-location generation module represents the user location data as relative points in an abstract 2D space, which enables learning across different physical spaces. The model fuses BiLSTM and CNN, where BiLSTM learns the speed and direction of the mobile users, and CNN learns information such as user movement preferences. FMLL uses federated learning to protect user privacy and reduce bandwidth consumption. Our experimental results, using a dataset with over 600,000 users, demonstrate that FMLL outperforms baseline models in terms of prediction accuracy. We also demonstrate that FMLL works well in conjunction with transfer learning, which enables model reusability. Finally, benchmark results on Android phones demonstrate FMLL's feasibility in real life.

CHAPTER 6

COMPLEMENT SPARSIFICATION: LOW-OVERHEAD MODEL PRUNING FOR FEDERATED LEARNING

In this chapter, we propose *Complement Sparsification (CS)*, a novel pruning mechanism through collaborative pruning performed at both the server and the clients. CS simultaneously meets the requirements of low bidirectional communication overhead between the server and clients, low computation overhead at the clients, and maintaining good model accuracy, under the FL assumption that the server does not access any raw data. At each round, CS creates a global sparse model that contains the weights that capture the general data distribution of all clients, while the clients create local sparse models with the weights pruned from the global model to capture the local trends. For improved model performance, these two types of complementary sparse models are aggregated into a dense model in each round, which is subsequently pruned in an iterative process.

The chapter is organized as follows. Section 6.1 presents CS process, its technical insights, and algorithm analysis. Section 6.2 shows the experimental results. The chapter concludes in Section 6.3.

6.1 Complement Sparsification in FL

Complement Sparsification (CS) aims to reduce the bidirectional communication overhead between the server and the clients, impose minimum computation overhead on the system, and achieve good model performance. Figure 6.1 shows its overview. In the initial round, the clients train from random weights and send their dense models to the server. After aggregation, the server prunes a percentage of model weights with low magnitude and sends the global sparse model to the clients. A pruning mask is also sent to the clients to mark the pruned weights. A 0 in the mask means the

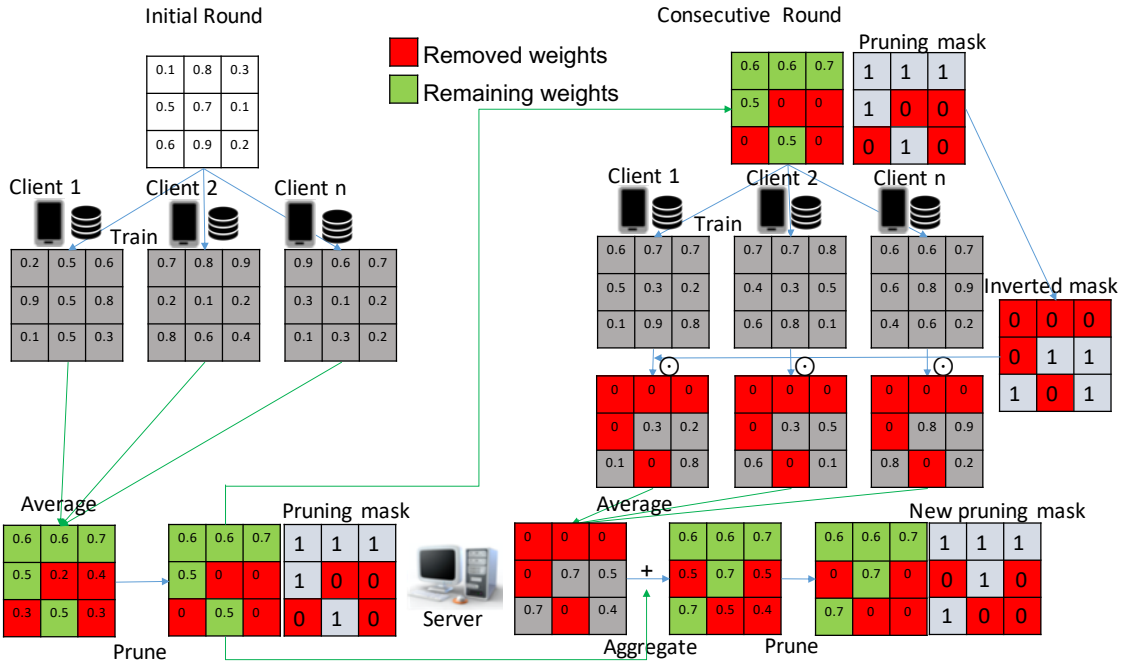


Figure 6.1 Overview of complement sparsification in FL.

weight is removed, while a 1 means the weight remains. In the following rounds, after training, the clients apply the inverted mask of the global sparse model and send their sparse models back. The server aggregates the client models with the global sparse model from the previous round. Because the inverted mask keeps the weights of the client models that were originally zero in the global sparse model, a full dense model is produced by the aggregation. In the new dense model, the weights with low magnitude are pruned away, and a new global sparse model is produced with a new pruning mask different from the one in the previous round. The new model has a different subset of non-zero weights because the client model weights are amplified with a given aggregation ratio to outgrow other weights.

The accuracy of the model improves over time, as all the model weights get eventually updated. Unlike pruning methods that require fine-tuning, the computation overhead of CS is merely removing some weights. The bidirectional communication overhead is also substantially reduced because both the server and the clients transfer sparse models.

6.1.1 Preliminaries

In order to formulate CS, we start with the formulation of FL, which is a distributed DL system that finds the model weights w that minimize the global empirical loss $F(w)$:

$$\min_{w} F(w) := \sum_{n=1}^{N} \frac{|x_n|}{|x|} F_n(w) \tag{6.1}$$

$$F_n(w) := \frac{1}{|x_n|} \sum_{i \in x_n} f_i(w) \tag{6.2}$$

where $F_n(w)$ is the local empirical loss for each client $n \in \{1, 2, .., N\}$, x_n is the local dataset of client n, $|x_n|$ is the dataset size of client n, $|x| = \sum_{n=1}^{N} |x_n|$ is the dataset size of all clients, and $f_i(w)$ is the loss function of a given client for a given data sample i in its dataset.

Each client n trains on its local data in every round.

$$\theta_{t+1,n} = w_t - \eta g_n \tag{6.3}$$

where θ is the current local model, w_t is the global model of previous round, η is the learning rate, and $g_n = \nabla F_n(w_t)$ is the average gradient of w_t on its local data. This step may iterate multiple times with different batches of data, and repeat over the whole dataset.

Without loss of generality, we assume that every client participates in aggregation in every round. The server aggregates the learning outcomes from the clients

as shown in Equation either (6.4) or (6.5).

$$w_{t+1} = w_t - \eta \sum_{n=1}^{N} \frac{|x_n|}{|x|} g_n \tag{6.4}$$

$$w_{t+1} = \sum_{n=1}^{N} \frac{|x_n|}{|x|} \theta_{t+1,n} \tag{6.5}$$

Equations (6.4) and (6.5) are equivalent because of (6.3). In (6.4), the server can use a different learning rate η from the client learning rate η in (6.3).

6.1.2 CS Workflow

Initial Round. CS starts from vanilla FL. The aggregated weights w_{t+1} are pruned by the server, with a pruning function $(w'_{t+1}, mask) = Prune(w_{t+1})$. The pruning function in CS removes the weights with low magnitude without any deliberate fine-tuning. We choose it because of its low overhead. The pruned model w'_{t+1} and the pruning mask $mask$ are sent to the clients for the following rounds. The pruning $mask$ is a binary tensor indicating where w'_{t+1} has weights set to 0.

Consecutive Rounds. In a new round, each client n receives the pruned model w'_t from the server, trains it on the local data x_n , and produces a new local model $\theta_{t+1,n}$:

$$\theta_{t+1,n} = w'_t - \eta g_n \tag{6.6}$$

Next, the clients compute the inverted bit-wise $\neg mask$ and apply the element-wise product $\odot$ between $\neg mask$ and $\theta_{t+1,n}$ (Equation (6.7)). If we want to save

communication overhead and not send the $mask$ from the server to the clients, the clients can derive $\neg mask$ directly from w'_t, at the expense of a trivial computation overhead.

$$\theta'_{t+1,n} = \theta_{t+1,n} \odot \neg mask \tag{6.7}$$

The server receives the complement-sparsified weights $\theta'_{t+1,n}$ from clients and aggregates them with w'_t and an aggregation ratio η', as shown in Equation (6.8).

$$w_{t+1} = w'_t + \eta' \sum_{n=1}^{N} \frac{|x_n|}{|x|} \theta'_{t+1,n} \tag{6.8}$$

Then, the server repeats the protocol from the previous rounds, and the CS workflow continues iteratively.

6.1.3 Algorithmic Description

Algorithm 6 shows the pseudo-code of CS. CS executes as a multi-round, iterative FL cycle (line 4-14), involving local model updates done by the clients with batches of data (lines 16-20), complement sparsifying the local models (line 21), server aggregation (lines 9-12), and the global model pruning (line 13). To prune the global model, we remove the weights with low magnitude (lines 26-29) and generate a binary tensor masking the zero weights (lines 30-33).

Algorithm 6 Complement Sparsification Pseudo-code

procedure SERVEREXECUTE:
  require CS aggregation ratio η' and server model sparsity $p\%$
  initialize $t = 0$, w_0 **randomly**, and tensor $mask$ to **zero**
  while !converged **do**
    // Update Done at Clients and Returned to Server
    for each client n **do** // In Parallel
      $(\theta_n, |x_n|) =$ n.CLIENTUPDATE$(w_t, mask)$
    $|x| = \sum_n |x_n|$
    if $t == 0$ **then**
      $w_{t+1} \leftarrow \sum_{n=1}^{N} \frac{|x_n|}{|x|} \theta_n$
    else
      $w_{t+1} \leftarrow w_t + \eta' \sum_{n=1}^{N} \frac{|x_n|}{|x|} \theta_n$
    $(w_{t+1}, mask) \leftarrow$ PRUNE(w_{t+1}, p)
    t++

procedure CLIENTUPDATE$(w, mask)$
  // Executed at Clients
  require step size hyperparameter η
  $x_n \leftarrow$ local data divided into minibatches
  for each batch $b \in x_n$ **do**
    $\theta_n = w - \eta \nabla F_n(w; b)$
  $\theta_n \leftarrow \theta_n \odot \neg mask$
  // Results Returned to Server
  return $(\theta_n, |x_n|)$

procedure PRUNE(w, p)
  // Executed at Server
  $th \leftarrow p^{th}$ percentile in w
  for each element $e \in w$ **do**
    if $e < th$ **then**
      $e \leftarrow 0$
  $mask \leftarrow w$
  for each element $e \in mask$ **do**
    if $e! = 0$ **then**
      $e \leftarrow 1$
  return $(w, mask)$

6.1.4 Technical Insights

In FL, the clients produce models that fit the local data, while the server's aggregation averages out the noise in the client models and produces a global model that fits the global data. In other words, the clients and the server are in a complementary relationship. In every round, the clients perturb the global model to follow their local data distribution better, and the server conciliates the client models to capture the global data distribution. CS draws from these insights when it creates complementary

sparse models at the server and the clients, respectively. In this way, it can reduce the computation and communication overhead, while achieving good model performance.

In CS, the server extracts a sparse model from the aggregated dense model. This sparse model preserves the global data distribution. Although the server does not fine-tune the sparse model, the clients perform implicit fine-tuning. They learn the local data distribution and create client sparse models that reflect shifts between the local and the global distribution. The updates are more easily reflected in the complement set of the global sparse model weights (i.e., the weights that were previously 0). Therefore, the clients complement-sparsify the models as in Equation (6.7), and only send the important model updates to the server with low communication overhead. This process also avoids overfitting the non-zero weights of the global sparse model by the clients' local data. The computation overhead is mostly imposed on the server, as the clients merely apply the inverted pruning mask.

Because we want all the weights to get updated over time for an accurate model, in every round, CS needs to produce a full dense model and generate a pruning mask different from the previous round. This is achieved by aggregating the complementary weights of the client models at round $t+1$ with the global model weights at round t as in Equation (6.8). More specifically, the new aggregated model weights are calculated by adding the global sparse model weights and the weighted sum of the client weights. The server uses a constant aggregation ratio $\eta' > 1$ to ensure that the pruned weights from the previous round outgrow the other weights, thus, will be less likely to be pruned in the current round. If some client updates are always small and are consequently removed by the server, the training can use a higher η', but η' shall not be higher than $1/\eta$ to avoid gradient explosion (see Subsection 6.1.5).

6.1.5 Algorithm Analysis

To show that in terms of performance CS is indeed an approximation of vanilla FL, we derive the aggregation function of vanilla FL (6.4) from CS (6.8), as follows.

$$w_{t+1} = w'_t + \eta' \sum_{n=1}^{N} \frac{|x_n|}{|x|} \theta'_{t+1,n} \qquad \text{(6.8 revisited)}$$

$$\approx w'_t + \eta' \sum_{n=1}^{N} \frac{|x_n|}{|x|} (\theta_{t+1,n} - w'_t) \qquad (6.9)$$

$$= w'_t - \eta'\eta \sum_{n=1}^{N} \frac{|x_n|}{|x|} \frac{w'_t - \theta_{t+1,n}}{\eta} \qquad (6.10)$$

$$= w'_t - \eta'\eta \sum_{n=1}^{N} \frac{|x_n|}{|x|} g_n \qquad (6.11)$$

$$\approx w_t - \eta'\eta \sum_{n=1}^{N} \frac{|x_n|}{|x|} g_n \qquad (6.12)$$

Equation (6.9) is from $\theta'_{t+1,n} \approx \theta_{t+1,n} - w'_t$. This is because the locally trained client model $\theta_{t+1,n}$ differs from the previous global sparse model w'_t mostly on the zero weights of w'_t. $\theta_{t+1,n} - w'_t$ sets the non-zero weights in w'_t to 0, similar with $\theta_{t+1,n} \odot \neg mask$ in (6.7). Equation (6.10) is derived by taking $-\eta$ out of the sum. Equation (6.11) is derived by using (6.6) in (6.10). The final result in Equation (6.12) is because the pruned weights w'_t approximate the weights before pruning w_t, as they only differ in the small magnitude weights. Comparing (6.12) with (6.4), the server applies $\eta'\eta$ as its learning rate. The aggregation ratio η' is essentially the server-client learning rate ratio used to adjust the server learning rate over the client learning rate. In practice, because learning rate is typically chosen between 0 and 1, η' shall be chosen between 1 and $1/\eta$ to ensure $\theta'_{t+1,n}$ outgrows w'_t without exploding w_{t+1}.

6.2 Evaluation

The evaluation has six goals: (i) Compare the learning progress of CS and vanilla FL; (ii) Compare the learning progress of CS and FL pruning baselines; (iii) Investigate the effectiveness of low overhead pruning in CS; (iv) Quantify the communication savings in CS; (v) Quantify the computation savings in CS; (vi) Investigate the trade-off between model sparsity and model performance in CS.

6.2.1 Datasets

CS is evaluated with two benchmark datasets in LEAF [194]: Twitter and FEMNIST. Twitter consists of 1,600,498 tweets from 660,120 users. We select the users with at least 70 tweets, and this sub-dataset contains 46,000+ samples from 436 users. FEMNIST consists of 80,5263 images from 3,597 users. The images are 28 by 28 pixels and represent 62 different handwritten characters (10 digits, 26 lowercase, 26 uppercase). We choose these two datasets because they represent important types of data in DL, text and image, and also allow us to observe how CS behaves with different scales of user pools and datasets.

In LEAF, we can choose IID or non-IID sampling scenarios. To evaluate CS under more realistic conditions, we choose non-IID for both datasets and make sure the underlying distribution of data for each user is consistent with the raw data. The training dataset is constructed with 80% of data from each user, and the rest of the data are for testing.

6.2.2 Models

We use a sentiment analysis (SA) model for the Twitter dataset, which classifies the emotions as positive, negative, or neutral. For example, with the inferred emotions of mobile users' text data, a smart keyboard may automatically generate emoji to enrich the text before sending. Our SA model first extracts a feature vector of size 768 from each tweet with pre-trained DistilBERT [175]. Then, it applies two dense

layers with ReLU and Softmax activation, respectively, to classify the feature vector. The number of hidden states of the two dense layers are 32 and 3, respectively.

We use a CNN-based image classification (IC) model for the FEMNIST dataset. This model uses three convolutional layers and two dense layers to classify an image into one of the 62 characters. The three convolutional layers have 32, 64, and 64 channels, respectively, with 3 by 3 filters, stride of 1, and ReLU activateion. A max pooling follows the first and the third convolutional layers. Then, the flattened tensor is fed into two dense layers of 100 and 62 neurons, with ReLU and Softmax activation, respectively.

The SA model has 24,707 trainable parameters, while the IC model has 164,506 trainable parameters. We choose these DL models also because we want to observe whether CS performs differently on different model sizes.

6.2.3 Experimental Settings

Table 6.1 Training Hyper-parameters for SA and IC Models

Model	Optimizer	Weight initializer	Client LR	Aggregation ratio	Batch size	Epoch
SA	Adam	he_uniform	0.01	1.5	64	5
IC	Adam	he_uniform	0.01	1.5	64	5

CS is implemented with Flower [11] and Tensorflow. The experiments are conducted on a Ubuntu Linux cluster (Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz with 512GB memory, 4 NVIDIA P100-SXM2 GPUs with 64GB total memory). We tested CS with different hyper-parameters, and only present the convergence progress with the hyper-parameters that led to the best results. Table 6.1 shows the training hyper-parameters for the two models. We set the aggregation ratio (η' in equation 6.8) to 1.5 to avoid clients' training outcomes being pruned away if they are too small. We set the server model sparsity to 50%, unless otherwise specified.

6.2.4 Baselines

We compare CS with two recently published baselines: PQSU [79], and PruneFL [78]. PQSU is composed of structured pruning, weight quantization, and selective updating. PruneFL includes initial pruning at a selected client, further pruning during FL, and adapts the model size to minimize the estimated training time. As CS, both PQSU and PruneFL aim to reduce communication and computation overhead in FL, and assume the server has no access to any raw data.

We run PruneFL from its GitHub repository. Since PQSU's orginal source code can not run continuous FL, we implement PQSU with Flower and Tensorflow, similar to CS. To make them comparable, we use the same data, model structures, model sparsity, and hyper-parameters.

6.2.5 Results

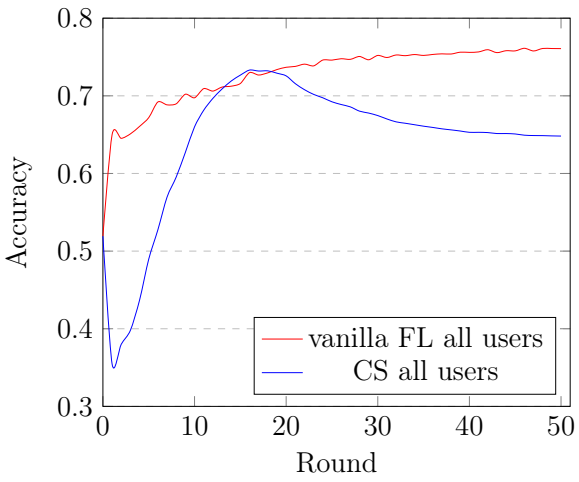


Figure 6.2 Test set accuracy vs. communication rounds for SA trained with all users in every round.

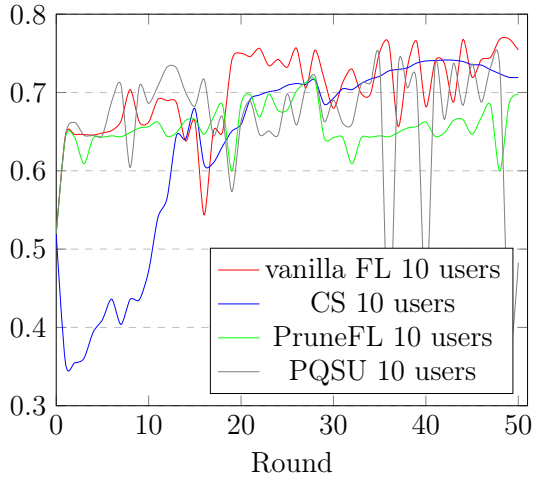


Figure 6.3 Test set accuracy vs. communication rounds for SA trained with 10 random users in each round.

Comparison with vanilla FL. Figure 6.2 shows the SA accuracy over training rounds when all users participate in every training round. In terms of best performance, the accuracy of CS is comparable with vanilla FL (73.3% vs. 76.1%). The less than 3% difference is the cost of the significant overhead reduction in CS,

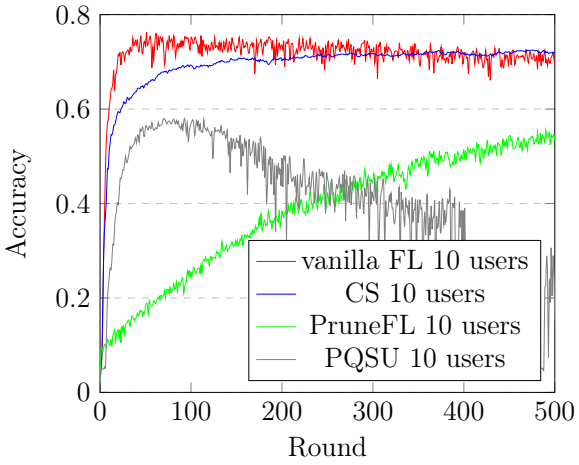


Figure 6.4 Test set accuracy vs. communication rounds for IC trained with 10 random users in each round.

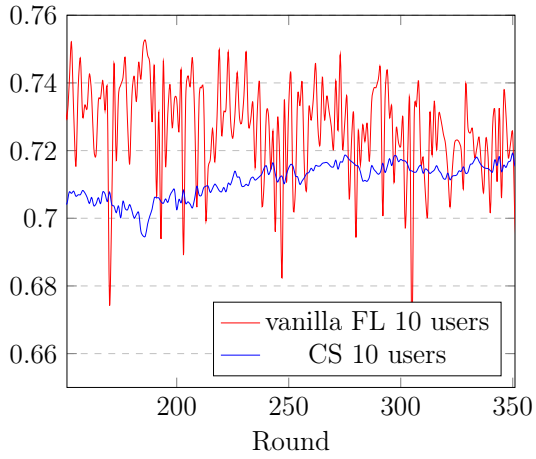


Figure 6.5 Zoom in of rounds 150-300 from Figure 6.4.

which will be shown later in this section. In the initial rounds, there is a gap in accuracy due to pruning and the fact that clients did not have yet time to recover the performance loss through local training. However, as FL proceeds, CS allows clients to implicitly fine-tune the pruned model. The accuracy gap between CS and vanilla FL gradually decreases, until overfitting occurs for CS. Nevertheless, the system can use the best model for inference.

In FL on mobile and Internet of Things (IoT) devices, however, it is more realistic that only a small portion of users participate in each training round due to resource constraints on the devices. Figure 6.3 shows the SA accuracy over communication rounds when 10 randomly selected users participate in each training round. In terms of best accuracy, CS (74.3%) competes with vanilla FL (76.9%). An advantage of CS is that its learning curve fluctuates significantly less than vanilla FL. This is because the effect of non-IID data is alleviated by runing in CS, while it is fully observed in vanilla FL. This phenomenon is further confirmed by the IC model.

Figure 6.4 shows the IC model accuracy over communication rounds when 10 randomly selected users participate in each round of training, which is a more realistic case than all users participating in every round. Overall, the learning curves between

CS and vanilla FL are close, with best accuracy of 72.5% and 76.3%, respectively. FEMNIST is a much larger dataset with more users than Twitter, and IC is a more complex model, with more possible output classes than SA. Therefore, it takes IC more rounds (up to 500) to converge. Let us note that the overfitting in Figure 6.2 does not appear in Figure 6.4. This is because a larger model is less likely to be overfitted by smaller amounts of information (partial participation of clients every round). Figure 6.5 shows a zoomed in portion of the graph in Figure 6.4. The results demonstrate that vanilla FL fluctuates more abruptly than CS during the training. This is an important advantage for CS in practice. In real-world FL over mobile or IoT devices, the data gradually accumulate as FL proceeds, but the system can not wait hundreds of rounds for a final best model or does not have a fully representative test dataset to select the best model for users. In CS, it is safe to distribute the latest model to users, while the latest model for vanilla FL may suffer from inferior accuracy.

Comparison with baselines. Figures 6.3 and 6.4 also show the model accuracy comparison between CS and the baselines. For SA, PruneFL and PQSU reach best model accuracy of 71.5% and 73.4%, which are 2.8% and 0.9% lower than CS, respectively. On the larger FEMNIST dataset, the results of IC show a clearer advantage for CS. For IC, PruneFL and PQSU reach best model accuracy of 55.5% and 57.9%, which are 17% and 15% lower than CS, respectively.

The original PruneFL paper [79] show comparable performance with vanilla FL on the FEMNIST dataset. However, the experiments used only the data of 193 users (out of 3597), the 193 users were further mixed and treated as 10 “super-users", and all these users participated in every round of training. We believe our experiments represent a more realistic scenario because we use all users and do not mix multiple users into a “super-user". For PQSU, the over-optimization on clients overfits the global model quickly. Thus, PQSU cannot benefit from additional data and training

rounds. To conclude, the baselines suffer from poor performance in realistic conditions for large datasets.

Next, we present a qualitative discussion to explain that the baselines have higher overhead. For communication overhead reduction, PruneFL uses an adaptive process in which the model not only shrinks, but also grows to reach the final targeted model sparsity. During the communications rounds with a grown model, PruneFL has higher communication overhead than CS. PQSU, on the other hand, can only save communication overhead in one direction, when clients transfer the sparse model to server. When PQUS transfers the model from the server to the clients, the communication overhead is higher than CS. Therefore, for the same targeted model sparsity, both PruneFL and PQSU have higher communication overhead than CS. For computation overhead, PruneFL imposes additional computation overhead including importance measure, importance aggregation, and model reconfiguration, while PQSU requires clients to further fine-tune their sparse models after the training. Thus, they also suffer from higher computation overhead than CS, because CS only needs to remove weights from the dense model.

Global sparse model vs. aggregated dense model. Figures 6.6 and 6.7 show the comparison between the global sparse model and the aggregated dense model (i.e., the model before sparsification in each round) in CS for SA and IC models. Overall, the global sparse model exhibits a smooth learning curve and outperforms the aggregated model. This demonstrates the effectiveness of the low-overhead model pruning in CS, which reduces communication overhead and maintains good model performance by removing weights in low magnitude. In CS, the aggregated model not only captures the global distribution, but also gets polluted by the noisy distribution shift induced from the clients data. In each round, simply removing the weights with low magnitudes from the newly aggregated model can effectively eliminate the noisy

distribution shift, and the global sparse model can steadily learn the global data distribution.

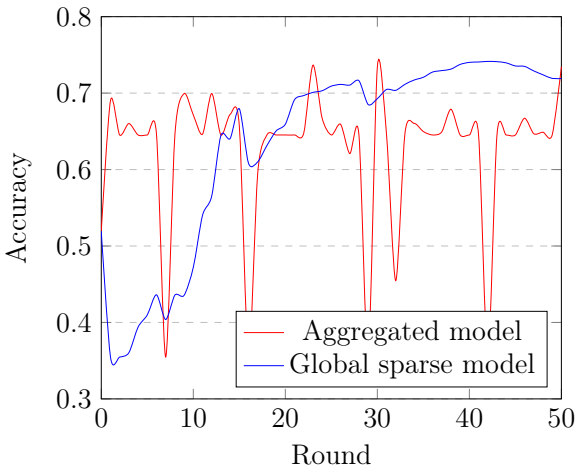


Figure 6.6 Global sparse model vs. aggregated dense model accuracy for SA with 10 random users every round.

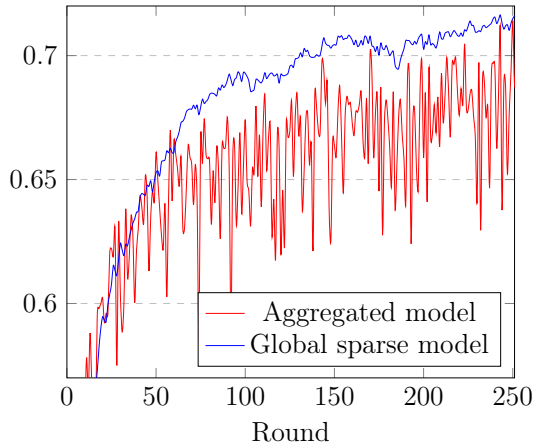


Figure 6.7 Global sparse model vs. aggregated dense model accuracy for IC with 10 random users every round.

Client model sparsity. Sparsity is the percentage of zero weights in the model. A model with high sparsity can save both computation and communication cost in FL. In CS, the client model applies the inverted pruning mask, but in practice the client model sparsity is much higher than the complementary percentage of the server model sparsity. This is because when a client trains the global sparse model, only a portion of the zero weights in the global sparse model gets updated. Tables 6.2 and 6.3 show the client model sparsity of SA and IC averaged over the number of rounds until they converge, while varying the server model sparsity. Let us note that we do not include the mask in the communication overhead, due to its small size. The server model sparsity indicates the communication cost saving from the server to the clients, while the client model sparsity represents the saving from the clients to the server. In general, the client model becomes sparser when the server model is denser. The results also show that the layers with more parameters benefit more from CS, as they are sparser than the small layers. The results demonstrate a substantial reduction in the communication overhead. For example, in Table 6.2,

when the reduction in the communication from the server to the clients is 80% (i.e., server model sparsity), for SA, the reduction in the communication from the clients to server is 81.2%. We observe similar results for IC (Table 6.3).

Table 6.2 Client Sparsity vs. Server Sparsity for SA

	Model layer	Server model sparsity			
		0.5	0.6	0.7	0.8
Client model sparsity	Dense (768×32)	0.933	0.885	0.841	0.812
	Output (32×3)	0.887	0.851	0.833	0.810
	Full model	**0.932**	**0.884**	**0.841**	**0.812**

Table 6.3 Client Sparsity vs. Server Sparsity for IC

	Model layer	Server model sparsity			
		0.5	0.6	0.7	0.8
Client model sparsity	Conv2D $(3 \times 3 \times 32)$	0.569	0.528	0.587	0.788
	Conv2D $(32 \times 3 \times 3 \times 64)$	0.842	0.788	0.800	0.900
	Conv2D $(64 \times 3 \times 3 \times 64)$	0.917	0.837	0.791	0.868
	Dense $(64 \times 16 \times 100)$	0.920	0.863	0.853	0.909
	Output (100×62)	0.756	0.722	0.721	0.698
	Full model	**0.904**	**0.843**	**0.828**	**0.891**

Table 6.4 CS Training FLOPs Saving vs. Server Sparsity for SA

	Model layer/ FLOPs	Server model sparsity			
		0.5	0.6	0.7	0.8
FLOPs saved (%)	Dense/147744	31.1	36.1	41.3	47.0
	Output/585	29.1	34.5	40.5	46.3
	Full model/148329	**31.1**	**36.1**	**41.3**	**47.0**

Training FLOPs savings. To evaluate the reduction in the computation overhead at the clients, we compute the training FLOPs savings based on the server and client model sparsity. We consider the number of multiply-accumulate (MAC) operations performed by each layer for both the forward and the backward pass during the training. In the forward pass, the clients perform FLOPs on the non-zero weights received from the server. In the backward pass, the MAC operations are counted for both the hidden state and the derivative. The hidden state MAC operations are fully counted as FLOPs. For the derivative, only the MAC operations on weights with

Table 6.5 CS Training FLOPs Saving vs. Server Sparsity for IC

	Model layer/ FLOPs	Server model sparsity			
		0.5	0.6	0.7	0.8
FLOPs saved (%)	Conv2D/1168224	19.0	24.3	32.9	46.3
	Conv2D/13381824	28.1	32.9	40.0	50.0
	Conv2D/17916096	30.6	34.6	39.7	48.9
	Dense/614700	30.7	35.4	41.7	50.3
	Output/37386	25.1	30.6	37.2	43.1
	Full model/33118230	**29.1**	**33.6**	**39.6**	**49.3**

non-zero values are counted as FLOPs. Here, the non-zero weights include both the non-zero set of weights received from the server and the zero weights that are updated to non-zero by the client. Let us note that the inverted pruning mask is applied after clients training, and therefore it does not help with FLOPs savings.

Tables 6.4 and 6.5 show the CS training FLOPs savings for both SA and IC, as a percentage of the FLOPs needed by vanilla FL. The values displayed in the second column of the tables are the training FLOPs of a single sample in vanilla FL. We compare them with CS training FLOPs under different server model sparsity. We observe that CS can save up to 49.3% training FLOPs, and the savings increase as the server model sparsity becomes higher. Similar with the communication savings, the layers with more parameters save a higher percentage of FLOPs.

Server model sparsity vs. model accuracy. Figures 6.8 and 6.9 show how the model accuracy varies with the server model sparsity for SA and IC. Since the server model sparsity is a parameter that can be set to different values for different models, it allows the system operators to achieve the desired trade-off between the model accuracy and the reduction in communication/computation overhead. In general, the model performs better when the server model sparsity is low. The results show that for SA, even a sparsity of 90% can lead to good performance (an accuracy deterioration of merely 2% compared to 50% sparsity). However, for IC, the sparsity should be kept to at most 70% to achieve acceptable performance.

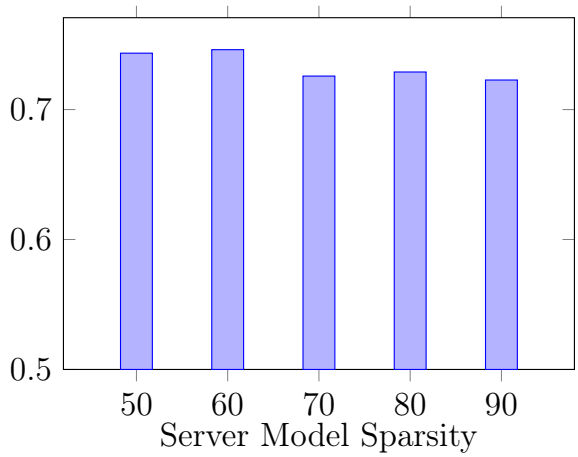


Figure 6.8 Accuracy as a function of server sparsity for SA.

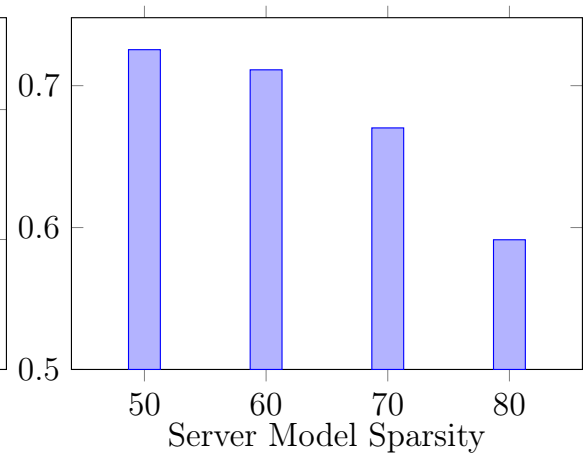


Figure 6.9 Accuracy as a function of server sparsity for IC.

6.3 Chapter Summary

This chapter proposed Complement Sparsification (CS), a practical model pruning for FL that can help the adoption of FL on resource-constrained devices. In CS, the server and the clients create and exchange sparse and complementary subsets of the dense model in order to reduce the overhead, while building a good accuracy model. CS performs an implicit fine-tuning of the pruned model through the collaboration between the clients and the server. The sparse models are produced with little computational effort. We demonstrate that CS is an approximation of vanilla FL. Experimentally, we evaluate CS with two popular benchmark datasets for both text and image applications. CS achieves up to 93.2% communication reduction and 49.3% computation reduction with comparable performance with vanilla FL. CS also performs better than baseline models in terms of model accuracy and overhead.

CHAPTER 7

FEDERATED CONTINUAL LEARNING USING CONCEPT MATCHING

This chapter proposes an Federated Continual Learning (FCL) framework: *Concept Matching (CM)*. The CM framework operates by grouping client models into clusters based on shared concepts and then constructing global models that capture different concepts in FL over time. In each round of training, the server sends the global concept models to the clients. To prevent catastrophic forgetting, each client selects the concept model that best matches the concept of the current data and fine-tunes it accordingly. To mitigate interference among client models with different concepts, the server clusters the models that represent the same concept, aggregates the model weights within each cluster, and updates the global concept model with the cluster model that corresponds to the same concept. Since the server is unaware of the concepts captured by the aggregated cluster models, we propose a novel server concept matching algorithm that effectively updates a concept model with a matching cluster model.

The chapter is organized as follows. Section 7.1 presents CM frame. Section 7.2 describes CM algorithms. Section 7.3 shows the evaluation results. The chapter concludes in Section 7.4.

7.1 CM Framework

7.1.1 Motivating Application Scenarios

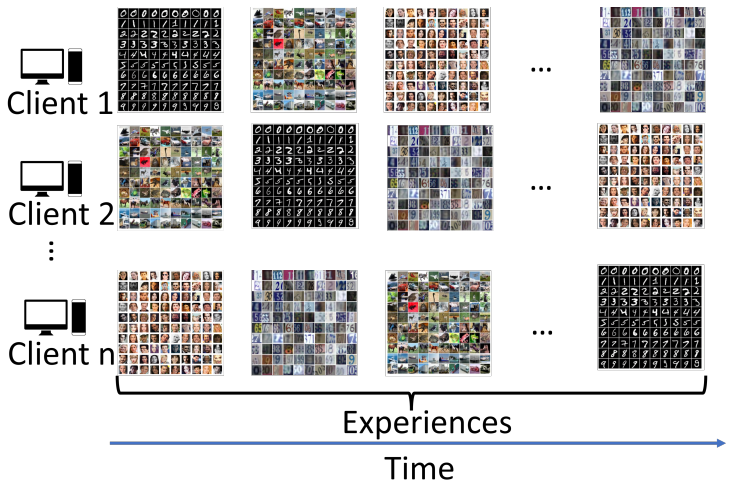
CM is an effective learning framework for FCL, where each client encounters a stream of data with different concepts over time. Figure 7.1a shows an example that different sets of image classes as different concepts over time. For the photos a client takes in daily life, the concepts can be different types of places of interest visited (e.g., museum

vs. national park), or different types of meals (i.e., breakfast, lunch, dinner). The clients may or may not be aware of the concepts of the data. In FCL, it is infeasible for clients to store all the data (e.g., limited storage on mobile/IoT devices) or the system cannot wait for all data to be accumulated before the training starts [195]. As an example, a smart camera captures video footage over time, encompassing different concepts such as day and night, or seasonal changes. However, it does not have the capacity to store all the videos. The system has to consume the data promptly to train one or multiple models working well for every concept. As in regular FL, the server in FCL only receives and aggregates the model weights from the clients, without accessing any additional information.

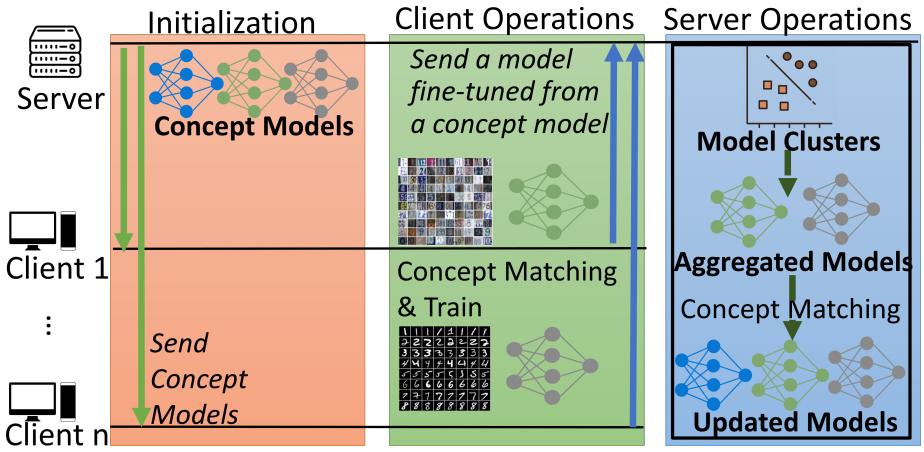
The number of concepts is typically a small constant estimated from the semantics of the application. In CM framework, the system administrator can select its value using domain knowledge. For example, a HAR model can use the locations of the user activities as different concepts, such as home, workplace, park, etc. Image data, on the other hand, can use the number of categories as the number of concepts. Our evaluation demonstrates the resilience of CM, when configured with numbers of concepts that are different from the ground truth.

7.1.2 Problem Definition

For each client $n \in \{1, 2, ..., N\}$, the data arrives in a streaming fashion as a (possible infinite) sequence of learning experiences $\mathcal{S}_n = e_n^1, e_n^2, ..., e_n^t$. Without loss of generality, each experience e_n^t consists a batch of samples $\mathcal{D}_n^t$, where the i-th sample is a tuple $\langle x_i, y_i \rangle_n^t$ of input and target respectively. Let $\mathcal{C} = \{C_1, C_2, ..., C_k\}$ be the set of K concepts hidden in entire dataset $\mathcal{D}$. Each concept C_k is associated with a probability distribution $P_k(X, Y)$, where X denotes the input space and Y denotes the label space. A batch of client samples follows one of the distributions $\mathcal{D}_n^t \sim P_k(X, Y)$, which may or may not be explicitly known by the client.



(a) FCL motivating example.



(b) CM training overview.

Figure 7.1 FCL using CM.

The goal is to learn a set of models $\{w_k\}_{k=1}^{K}$, and each model w_k can perform well for its corresponding concept C_k. The problem can be formulated as the Equation 7.1, where L is the loss function, and $\mathcal{D}_n$ is the entire stream of data on client n.

$$\arg \min_{\{w_k\}_{k=1}^{K}} \sum_{k=1}^{K} \sum_{n=1}^{N} L(w_k, \mathcal{D}_n) \tag{7.1}$$

7.1.3 Learning Framework for FCL

Figure 7.1b shows the CM training in an FCL round. To initialize the learning process (**orange box**), the system admin at the server side determines the number of concepts K and designs the model. The server initializes the weights of K global concept models randomly and sends them to the clients.

In the client operation phase (**green box**) of every round, each client n receives the weights of the global concept models $W^{t-1} = (w_1, w_2, ..., w_K)^{t-1}$ from the server. To avoid catastrophic forgetting caused by training a model with the data of different concepts, the clients perform concept matching with the local data of current round to select the best-matching global concept model as Equation 7.2. Next, the client fine-tunes the best-matching global concept model weights w_{k^*} with the local data, produces a new local model with weights θ_n^t, and sends it to the server.

$$k_n^* = ClientConceptMatch(W^{t-1}, \mathcal{D}_n^t) \tag{7.2}$$

In the server operation phase (**blue box**), the server receives the client models with weights $\{\theta_n^t\}_{n=1}^N$. To avoid interference among client models with different concepts, the server clusters the client models into a set of clusters of size J, denoted as Ω^t (Equation 7.3). Since the union of clients data per round may not cover all concepts, J and K are usually different. Then, the server produces aggregated cluster models with weights $W'^t = (w'_1, w'_2, ..., w'_J)^t$ (Equation 7.4). The server does not know the concept in a aggregated cluster model or which global concept model to update. Therefore, it needs to match the aggregated cluster models Θ^t with the global concept models W^{t-1}, and only update the global concept models with data encountered in this round as Equation 7.5.

$$\Omega^t = Cluster(\{\theta_n^t\}_{n=1}^N) \tag{7.3}$$

$$W'^t = Aggregate(\Omega^t) \tag{7.4}$$

$$W^t = ServerConceptMatch(W'^t, W^{t-1}) \tag{7.5}$$

Algorithm 7 shows the pseudo-code of the CM framework with the local training on clients. CM executes as a multi-round, iterative FL cycle (lines 3-9). At each round (line 3), each client operates in parallel (line 5), including clients concept matching (line 13), local model updates with batches of data (line 14-16), and returning the client model weights to the server (line 18). Then, server performs clustering (line 7), aggregation (line 8), and server concept matching to update the global concept models with the aggregated cluster models (line 9).

Algorithm 7 Concept Matching Framework Pseudo-code

```
1: procedure ServerExecute:
2:     initialize W^0 = (w_1, w_2, ..., w_K)^0 randomly, total number of rounds T
3:     for t = 1 to T do
4:         // Update Done at Clients and Returned to Server
5:         for each client n do // In Parallel
6:             θ_n = n.ClientUpdate(W^{t-1})
7:         Ω^t = Cluster({θ_n^t}_{n=1}^N)
8:         W'^t = Aggregate(Ω^t)
9:         W^t = ServerConceptMatch(W'^t, W^{t-1})

10: procedure ClientUpdate(W)
11:     // Executed at Clients
12:     require step size hyperparameter η, local dataset of current round D
13:     k* = ClientConceptMatch(W, D)
14:     x_n ← D divided into minibatches
15:     for each batch b ∈ x_n do
16:         θ_n = w_{k*} − η∇L(w_{k*}, b)
17:     // Results Returned to Server
18:     return θ_n
```

7.1.4 Design Discussion

The CM framework provides the flexibility to use different clustering, aggregation, and concept matching algorithms. It can evolve as new algorithms are proposed for different applications and models. The aggregation algorithms in FL are orthogonal to the CM framework, and any of them can be employed to further mitigate the challenges of non-iid data. We evaluate the CM framework with multiple clustering algorithms, and demonstrates that CM works well with classic clustering algorithms, such as kmean, agglomerative, and BIRCH. Some clustering algorithms, such as DBSCAN [196] and OPTICS [197], do not require the number of clusters to be known. As a future work, the CM framework may be extended to work in such a scenario. In addition, dimension reduction techniques [198] can also be applied in conjunction with clustering algorithms to mitigate the curse of dimensionality.

Regarding privacy, the CM framework is the same as vanilla FL (i.e., clients send only their model weights to server). An adversarial server might seek to extract information from the client model using the same techniques from vanilla FL, which is beyond the scope of this paper. Regarding communication, the clients send a single model trained with the local data in the same way as vanilla FL, but the server sends multiple concept models to the clients. The number of concepts is usually a small constant under the control of the system administrator, and the concept models designed in CM can be smaller than the single model in vanilla FL, because each model only learns a single concept. Nevertheless, it is essential to balance the trade-off between model performance and communication overhead, by taking into account the available system resources. To further improve privacy protection and communication efficiency, CM can use existing privacy protection [199] or communication reduction [200] techniques for FL.

The two learning scenarios [201] in FCL are class-incremental and task-incremental. Class-incremental requires the system to learn different classes over

time. The changes of the classes in data causes concept drift, but the system is not given any additional information to identify the concept drift. Task-incremental focuses on learning different tasks over time, such as different languages in speech recognition. This scenario can also consider the tasks to be different classes. In this case, the difference from class-incremental is that the tasks (e.g., sedentary activities vs. physical exercises in HAR) are known, and the concept drift is recognized by the model. In centralized CL, the task IDs can be used to help both training and inference. However, in FCL, the clients can utilize the task IDs, but they cannot share them with the server due to privacy concerns.

The CM framework is designed to be compatible with both task-incremental and class-incremental scenarios, because the clients do not need to possess any prior understanding of concept drift in data, including task IDs. In both scenarios, the server does not know the concepts of clients data, and it shall perform clustering and concept matching every round. In the task-incremental scenario where the clients know the task IDs, the clients only need to perform concept matching at the initial rounds. The clients can maintain a mapping between the global concept model ID and the task ID. After the client encounters all tasks, it can use the mapping to match the data of a task with the global concept model. During inference, the clients can pick the corresponding global concept model to perform prediction with their input data. For class-incremental, the clients have to perform concept matching every round. At inference, the clients can apply an ensemble model method [202] to produce an output from all the global concept models.

7.2 Concept Matching Algorithms

The concept matching algorithms at the client and the server collaboratively and iteratively update each global concept model with the information learnt from the data of a matching concept, and achieve up to 100% effectiveness. The two algorithms

are connected through client training, server clustering and aggregation. The main novelty of this distributed approach lies in the server concept matching algorithm, which ensures the model updates in the correct gradient descent direction.

Algorithm 8 Server Concept Matching Pseudo-code

```
1: procedure ServerConceptMatch(W', W)
2:     // Executed at Server
3:     require distRecord of size K as the global record of distance between each
       global concept model and the corresponding previous global concept model
4:     for each aggregated cluster model w'_j ∈ W' do
5:         candidate ← null
6:         candidateDist ← ∞
7:         for each global concept model w_k ∈ W do
8:             tmpDist ← Distance(w'_j, w_k)
9:             if tmpDist < distRecord[k] and tmpDist < candidateDist then
10:                candidate ← k
11:                candidateDist ← tmpDist
12:        W[candidate] ← w'_j
13:        distRecord[candidate] ← candidateDist
14:    return W
```

Server Concept Matching. After clustering, the groups of the client models fine-tuned with the data of different concepts are unordered, and the number of clusters may not be the same as the total number of concepts because the clients' data union in the current round may not cover all concepts. The server does not know how to update the global concept models without the matching between the aggregated cluster models and the global concept models from previous round.

To resolve this challenge, we propose a novel distance-based server concept matching algorithm. This algorithm not only updates a global concept model with a cluster model close in distance, but also ensures the update in the correct gradient descent direction. Our algorithm can use different distance metrics. For a normal size neural network such as LeNet, Manhattan or Euclidean distance can be used for their low computational complexity. For larger neural networks, dimension reduction techniques can be incorporated to mitigate the curse of dimensionality.

The pseudo-code of server concept matching algorithm is shown in Algorithm 8. The algorithm requires a global record of the distance between each global concept model and the corresponding previous global concept model (line 3). For each aggregated cluster model (line 4), the algorithm tracks its best-matching candidate (line 5) and its distance from the best-matching candidate (line 6). Each aggregated cluster model is compared with each global concept model (line 7) by computing their distance (line 8). If their distance is smaller than both the global distance record of the corresponding concept k and the distance from previous matching candidate (line 9), we consider them a better match (line 10) and update the distance between the cluster model and its matching candidate (line 11). After checking all the global concept models, the algorithm updates the best-matching global concept model with the aggregated cluster model (line 12), and also updates the global distance record with the distance between the best-matching pair (line 13).

This algorithm is theoretically grounded. Intuitively, in a gradient descent learning algorithm, as the learning curve becomes flatter over iterations, the learning slows down. Therefore, the distance between the current model and the model of the previous iteration becomes smaller. Theorem 1 formulates this intuition, and Algorithm 8 utilizes this theorem. By tracking the distance record, Algorithm 8 updates each concept model with a matching cluster model only when their distance becomes smaller than the current distance. The proof of Theorem 1 demonstrates theoretically that Algorithm 8 updates the global concept model with a matching cluster model in the correct gradient descent path. This allows the concept models to learn over time without interference from other concepts.

Assumption 1. *Differentiability: The loss function $L(w)$, used to optimize a neural network, is differentiable with respect to the model parameters w.*

Assumption 2. *Lipschitz continuity: The gradient of the loss function $\nabla L(w)$ is Lipschitz continuous with a positive constant L. By Lipschitz continuity definition,*

for any two points w^1 and w^2, the following inequality holds $\|\nabla L(w^1) - \nabla L(w^2)\| \leq L\|w^1 - w^2\|$, where $\|.\|$ denotes the norm.

Lemma 1. *Given a loss function $L(w)$ under assumption 1 and 2, w is updated with gradient descent $w^{t+1} = w^t - \eta\nabla L(w^t)$, where t is the iteration number, η is the learning rate, and $\nabla L(w^t)$ is the gradient of the loss function with respect to w^t, the following inequality holds $\|\nabla L(w^{t+1})\| < \|\nabla L(w^t)\|$.*

Proof. To prove $\|\nabla L(w^{t+1})\| < \|\nabla L(w^t)\|$, we can use the assumption 2. Let L be the Lipschitz constant. Then, we have $\|\nabla L(w^{t+1}) - \nabla L(w^t)\| \leq \mathrm{L}\|w^{t+1} - w^t\|$.

Now, using the reverse triangle inequality, we can write $\|\nabla L(w^{t+1})\| - \|\nabla L(w^t)\| \leq \|\nabla L(w^{t+1}) - \nabla L(w^t)\|$. Substituting the previous inequality, we get $\|\nabla L(w^{t+1})\| - \|\nabla L(w^t)\| \leq \mathrm{L}\|w^{t+1} - w^t\|$.

Using the gradient descent update $w^{t+1} = w^t - \eta\nabla L(w^t)$, we can write $\|\nabla L(w^{t+1})\| - \|\nabla L(w^t)\| \leq \mathrm{L}\eta\|\nabla L(w^t)\|$. Rearranging the terms, we get $\|\nabla L(w^{t+1})\| \leq (1 - \mathrm{L}\eta)\|\nabla L(w^t)\|$.

Since L and η are both positive, we have $1 - \mathrm{L}\eta < 1$. Therefore, we can conclude that $\|\nabla L(w^{t+1})\| < \|\nabla L(w^t)\|$. This completes the proof. □

Theorem 1. *Given a loss function $L(w)$ under assumptions 1 and 2, w is updated with gradient descent $w^{t+1} = w^t - \eta\nabla L(w^t)$, where t is the iteration number, η is the learning rate, and $\nabla L(w^t)$ is the gradient of the loss function with respect to w^t, the following inequality holds $\|w^{t+1} - w^t\| < \|w^t - w^{t-1}\|$.*

Proof. From the inequality in lemma 1, $\|\nabla L(w^t)\| < \|\nabla L(w^{t-1})\|$, using the gradient descent update $w^{t+1} = w^t - \eta\nabla L(w^t)$, we write $\|w^{t+1} - w^t\| < \|w^t - w^{t-1}\|$. This completes the proof. □

Theorem 1 is under Assumption 1 Differentiability and Assumption 2 Lipschitz continuity. Without requiring strong assumptions, such as convexity, Assumption 1 and 2 can be applied to most loss functions for neural networks. In the theorem,

gradient descent is also the prevalent algorithm to update neural networks. Therefore, this theorem can be applied to the general optimization process of most neural networks.

Client Concept Matching. The clients receive the global concept models from the server every round, and each model shall learn the data distribution of each concept. The clients need to select one of the global concept models, and fine-tune it with the data of the current round. Since the clients may or may not be aware of the concepts, they shall perform client concept matching to match the concept of the current data with a global concept model.

At round t, a client n can test the global concept models of the previous round $\{w_k^{t-1}\}_1^K$ on its current local data $\mathcal{D}_n^t$, and select the k^*-th concept model with the smallest loss for further fine-tuning as Equation 7.6. Since the data do not accumulate over a given limit in FCL, testing the models is an effective method to select the best-matching global concept model without significant overhead.

$$k^* = \arg\min_k L(w_k^{t-1}, \mathcal{D}_n^t) \tag{7.6}$$

7.3 Evaluation

The evaluation has following goals: (i) investigate CM effectiveness in terms of model accuracy and learning curve volatility; (ii) compare CM with state-of-the-art FCL solutions; (iii) quantify CM algorithms effectiveness to match concepts of data to models; (iv) investigate CM scalabilitiy in terms of number of clients and model size; (v) understand CM resilience when configured with numbers of concepts that are different from ground truth; (vi) quantify CM feasibility and overhead in terms of operation time on a real IoT device; (vii) performance of different clustering algorithms; (viii) model accuracy for class-incremental vs. task-incremental scenarios;

(ix) CM performance under diverse experimental settings, including datasets, concept configuration, number of clients, and model size.

7.3.1 Experimental Setup

Datasets. We evaluate CM with two models over two "super" datasets, each consisting of multiple datasets (i.e., one model per "super" dataset). The training, test, and validation split of the data follows 7:2:1. The first "super" dataset consists of six frequently used image datasets: SVHN, FaceScrub, MNIST, Fashion-MNIST, Not-MNIST, and TrafficSigns. To simulate different concepts, the "super" dataset is splitted into five concepts. As shown in Table 7.1, the data in the five concepts differ across a wide spectrum of classes and number of samples. The original FaceScrub dataset has 100 classes. In order to stress test CM, it is splitted into Concept 2 and 3 with 50 different classes each, as we aim to verify whether CM can differentiate them successfully. Since MNIST datasets are easy to learn, we mix the three MNIST datasets together to make it more difficult.

Table 7.1 SVHN, FaceScrub, MNIST, Fashion-MNIST, Not-MNIST, and TrafficSigns "Super" Dataset Details for Each Concept

Concept	1	2	3	4	5
Dataset	SVHN	FaceScrub0	FaceScrub1	MNIST, Fashion-MNIST, Not-MNIST	TrafficSigns
No. Classes	10	50	50	30	43
No. Samples	88300	9898	9899	138197	45956

The second "super" dataset consists of Cifar100 and TinyImagenet. They have 100 and 200 classes respectively. The classes in Cifar100 and TinyImagenet are further splitted into three distinct concepts each, resulting in a total of six concepts. Table 7.2 shows the number of class and samples per concept.

Table 7.2 Cifar100 and TinyImagenet "Super" Dataset Details for Each Concept

Concept	1	2	3	4	5	6
Dataset	TinyImagenet0	TinyImagenet1	TinyImagenet2	Cifar100_0	Cifar100_1	Cifar100_2
No. Classes	66	67	67	33	33	34
No. Samples	33000	33500	33500	19800	19800	20400

Models. To compare with the baseline fairly, we use the same CNN-based image classification model as [63] for the first "super" dataset. We believe this model is ideal in size to learn from the dataset. This model uses two convolutional layers and three dense layers to classify an image input (32*32*3) into one of the 183 classes. The two convolutional layers have 20 and 50 channels, with 5 by 5 filters, stride of 1, and ReLU activation. A 3 by 3 max pooling with stride of 2 follows them. Then, the flattened tensor is fed into three dense layers of 800, 500 and 183 neurons respectively, with ReLU and Softmax activation. Unless noted otherwise, the results are based on this model with the first "super" dataset.

For the "super" dataset of Cifar100 and TinyImagenet. A simplified EfficientNet is implementated with a multi-stage CNN that uses a balanced scaling strategy to achieve high accuracy with fewer parameters. It starts with an initial convolutional layer that uses a 3x3 kernel with a stride of 2 to increase the number of channels while reducing spatial dimensions. After batch normalization and a ReLU activation, the model includes a series of MBConv (Mobile Inverted Bottleneck Convolution) blocks, which consist of three key phases: expansion, depthwise convolution, and projection. The expansion phase uses a 1x1 convolution to increase the number of channels, followed by batch normalization and ReLU activation. The depthwise convolution phase employs a 3x3 depthwise convolution and optional squeeze-and-excitation (SE) mechanism, where global average pooling is applied, and the output is scaled to emphasize important features. The projection phase reduces the channel count back to a desired size with another 1x1 convolution. The MBConv blocks also support residual connections if the input and output have the same channel count and strides are 1. After the MBConv blocks, the model uses global average pooling to create a single vector representation of the input, followed by a dropout rate of 0.2 to reduce overfitting. The classification layer at the end is a dense layer with softmax

activation, providing a probability distribution over the 300 classes of Cifar100 and TinyImageNet.

The two models are compiled with the Adam optimizer and categorical cross-entropy loss for multi-class classification, with accuracy as the primary metric. As it is common practice in class-incremental CL, the models follow the single-head evaluation setup [203–205], where it has one output head to classify all labels. This setup is ideal for clients with constrained resource capacity in FL, because the clients do not have to spend computation resources on expanding or selecting the output head. Let us note that using the total number of labels as the model output size does not mean we have to know the entire label space or even the label space size, because we can use any output size not smaller than the upper bound of the number of labels. It makes no difference in the training and testing accuracy when experimenting with a given dataset, because the weights associated with unencountered output neurons will not be updated in the backward pass.

Comparison solutions. We compare CM with vanilla FCL, vanilla FL, and three state-of-the-art FCL solutions. Vanilla FCL follows the same dynamic data scenario above. Vanilla FL represents the static FL scenario, which is less challenging than FCL. In our case, the "super" datasets are distributed to the clients without being split into concepts. Every round, the clients train with the entire local dataset containing all concepts. For the three state-of-the-art FCL solutions, we select FedWeIT, TARGET [62], and EWC [59]. FedWeIT and TARGET represent parameter isolation and replay approaches for FCL, respectively. EWC is a commonly employed approach in CL. We apply it on the clients' training. For all comparisons, we test the models with the test sets for model accuracy.

Experimental Settings. We implement CM with TensorFlow and scikit-learn. The experiments are conducted on a Ubuntu Linux cluster (Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz with 512GB memory, 2 NVIDIA P100-SXM2 GPUs with

16GB total memory). Non-overlapping chunks from the five concept datasets are further distributed randomly to the clients. At every round, the local data across clients are non-IID. Each client encounters one of the five local concept datasets randomly, and uses a cyclic sliding window of 320 samples in the encountered concept dataset. Unless otherwise specified, CM is tested with 20 clients (all clients participate in each training round), kmean clustering algorithm, FedAvg aggregation algorithm, and Manhattan distance for the server CM algorithm. For the local training, we use the Adam optimizer with learning rate of 0.001, weight initializer of HeUniform, and batch size of 64. Each client maintains and uses a single Adam optimizer throughout the training for all concepts. We train 15 epochs every round, and use early stopping with the patience value as 3. We test CM with different hyper-parameters, and only present the results with the hyper-parameters that lead to the best results. The system runs 100 rounds of training for each experiment.

To quantify the overall concept matching effectiveness, we evaluate under the class-incremental scenario, and assume the clients are not aware of the concepts and perform the client concept matching every round. In task-incremental scenarios, the difference is that the clients do not have to perform concept matching every round, because they know the task IDs and the concept drift due to the transition of different tasks. For the same experimental setting, task-incremental training would achieve the same model performance (Figure 7.4) with lower computation overhead at the clients.

Parameters for Clustering Algorithms. Kmean, agglomerative, and BIRCH require the number of clusters as a parameter. Unless otherwise specified, we set this parameter to be 5. DBSCAN and OPTICS require some threshold values tuned for the data as parameters instead of the number of clusters. We adhere to the convention when selecting their parameters. For DBSCAN, there are two main parameters: $min\_samples$ and ϵ. $min\_samples$ is the fewest number of points required to form a cluster. We adjust it to be 3, which is a lower value than the

average number of clients per concept (20/5). ϵ is the maximum distance between two points while the two points can still belong to the same cluster. To choose ϵ , we firstly calculate the average distance between each point (the model weights of a client) and its 3 ($min\_samples$) nearest neighbors, and then we sort distance values in the ascending order and plot them. We choose ϵ to be 20 as the point of maximum curvature in the plot. Similarly, we set $min\_samples$ parameter to be 3 for OPTICS. The other parameters for these clustering algorithms are the default values in scikit-learn.

Experiment Statistical Significance. The experiment statistical significance is reported as ± 1 Standard Deviation (SD) of the test set accuracy and forgetting rate in the comparison results with the state-of-the-art solutions. The factors of variability may include train/test split, initialization, and the stochastic nature of DL model optimization. We assume normally distributed errors, and run each experiment five times for SD.

IoT Device Setup. CM is evaluated on a Qualcomm QCS605 IoT device. This device is equipped with Snapdragon™ QCS605 64-bit ARM v8-compliant octa-core CPU up to 2.5 GHz, Adreno 615 GPU, 8G RAM, and 16 GB eMMC 5.1 onboard storage. We choose it because its specifications are ideal for AI cameras and image-based applications. The device is connected to Internet through WiFi with a bandwidth of 300 Mbps. We re-use CM simulation Python code on the device. Since the device does not support native Linux and its the operating system is rooted Android 8.1, we need to run a Linux distribution on the Linux kernel of Android for easy package management and better support. We achieve this goal with two open source projects: termux-app [1] and ubuntu-in-termux [2]. Termux-app is an Android application for terminal application and Linux environment. It provides some basic Linux commands and packages, but is not on par with a mature Linux distribution,

[1]https://github.com/termux/termux-app
[2]https://github.com/MFDGaming/ubuntu-in-termux

such as Ubuntu. Ubuntu-in-termux bridges the gap. Through it, we are able to install well-maintained Python environments and libraries to execute training on-device. The Python training process can be observed directly under adb shell, which does not include the overhead of Termux-app or Ubuntu-in-termux.

7.3.2 Results

Effectiveness of CM. Figure 7.2 and 7.3 demonstrate that CM effectively achieves better model accuracy than vanilla FCL and vanilla FL. Compared with vanilla FCL, the concept model accuracy improvement is up to 26.4%. The superior performance of CM is more apparent for difficult concepts (RBG images in Concepts 2 and 3 of Figure 7.2 and all Concepts of Figure 7.3) than for easy concepts (BW images in Concept 4 of Figure 7.2). Specifically, the most significant difference is observed in the concept 3 of Figure 7.3, despite the observed fluctuations caused by the challenging nature of the dataset. However, CM exhibits a smoother learning progress than vanilla FCL in Figure 7.2 when the concepts are more distinct. These results indicate CM can effectively mitigate catastrophic forgetting and the potential interference among clients. Vanilla FL exhibits the same smooth learning progress due to the unchanging concepts. Surprisingly, CM also outperforms vanilla FL by up to 13.5%. This is because CM employ different concept models for each concept, which is better than a single global model even under the static data distribution. Unless otherwise specified, the rest of the results in this section are based on the "super" dataset combining SVHN, FaceScrub, MNIST, Fashion-MNIST, Not-MNIST, and TrafficSigns.

Model accuracy comparison with FCL state-of-the-art solutions. Table 7.3 shows the model accuracy comparison. CM outperforms the FCL state-of-the art solutions and achieves 90.3($\pm$0.07)% accuracy (weighted average over the number of samples per concept). While EWC and TARGET performs reasonably well (86.7% and 87.0% respectively), FedWeIT does not perform well under more

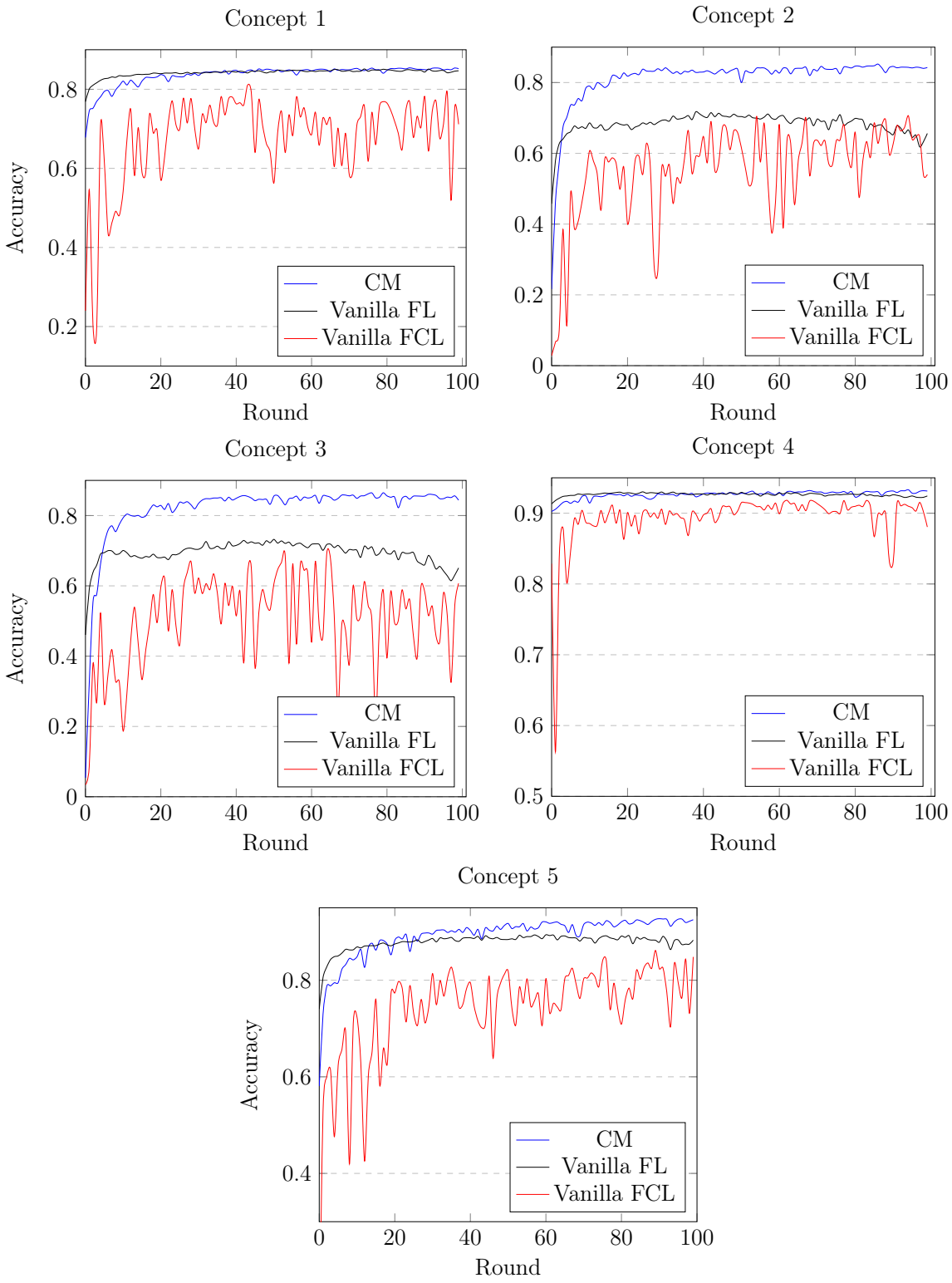


Figure 7.2 CM Effectiveness with SVHN, FaceScrub, MNIST, Fashion-MNIST, Not-MNIST, and TrafficSigns: test set accuracy over training rounds.

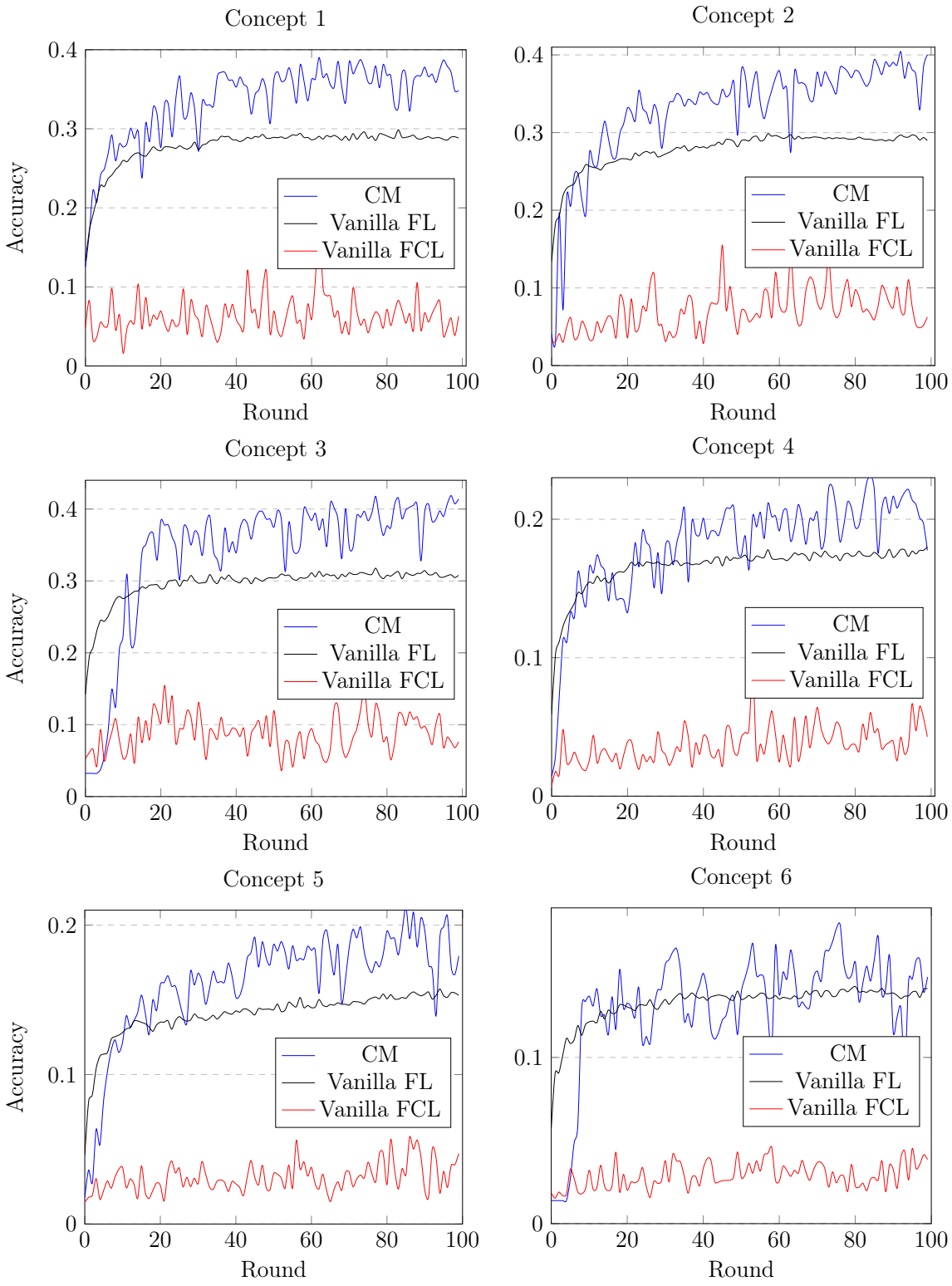


Figure 7.3 CM effectiveness with Cifar100 and TinyImagenet: test set accuracy over training rounds.

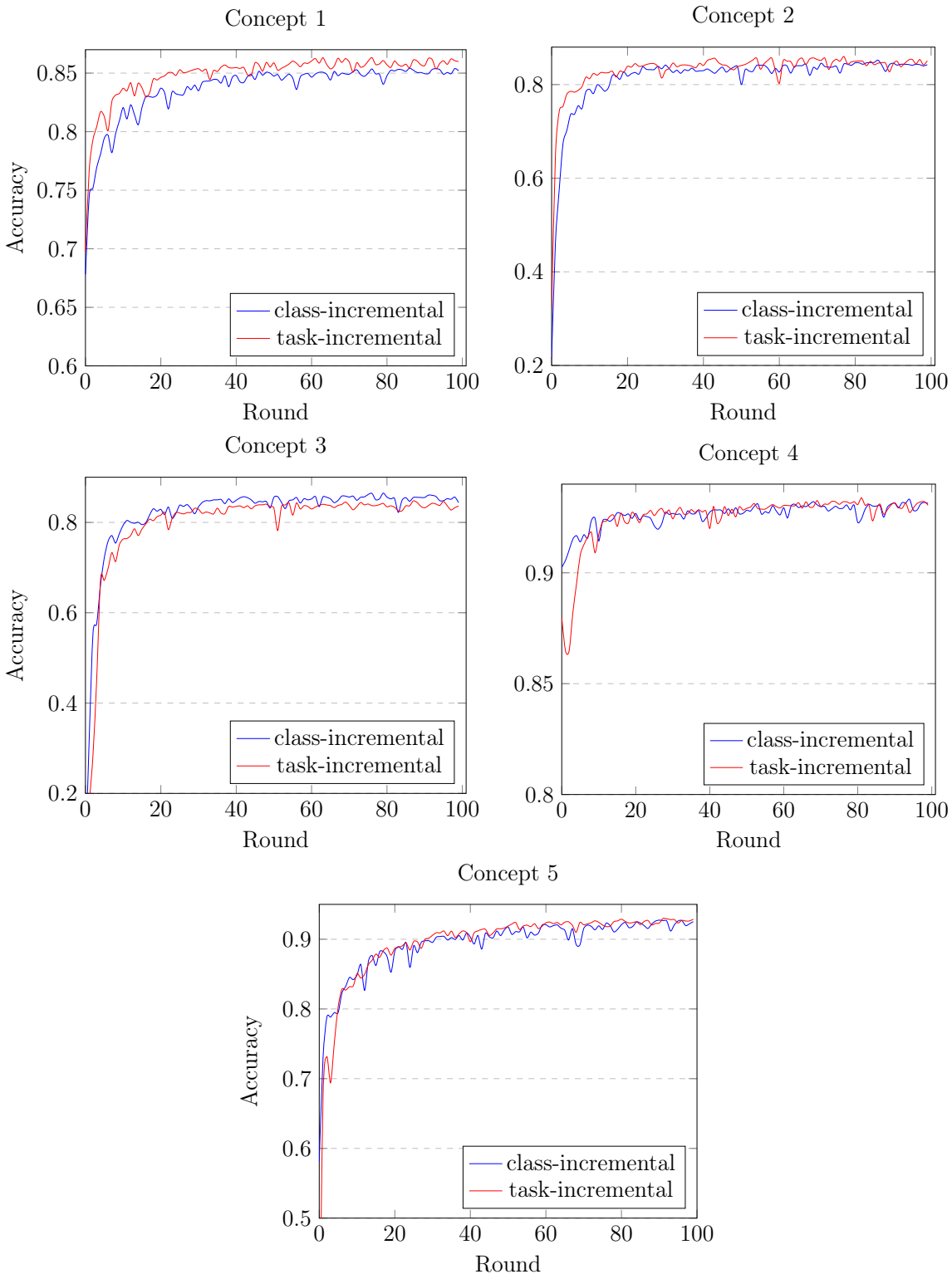


Figure 7.4 Class-incremental vs. task-incremental: SVHN, FaceScrub, MNIST, Fashion-MNIST, Not-MNIST, TrafficSigns test set accuracy over communication rounds.

realistic assumptions and in a larger scale experiment than its original evaluation (i.e., 5 clients per round, 5 classes to train per client, and non-overlapping classes over clients). Since FedWeIT applies a completely different design when learning information across tasks or concepts, its inferior performance may be partially due to the sparse parameters employed, which fail to separate different concepts and fully capture the complex information (i.e. up to 50 classes) in the concepts.

Forgetting rate comparison with FCL state-of-the-art solutions. Throughout FCL training, the model accuracy experiences declines in certain rounds, and the extent of these accuracy drops can be viewed as an indicator of forgetting. The forgetting rate is determined by summing the accuracy drops across these performance-deteriorating rounds and subsequently averaging the values over the entire span of 100 training rounds. Table 7.4 shows the results of the forgetting rate comparison. CM has only 0.17(±0.01)% forgetting rate. Compared with CM, the other baselines exhibit higher forgetting rates ranging from 9.3 to 18.8 times. These results illustrate that CM can significantly mitigate the effect of catastrophic forgetting. The mean and SD of model accuracy and forgetting rate, shown in Table 7.3 and 7.4, also demonstrate statistical significance of the superior performance of CM, because the improvement is at least 47 times of the SD.

Table 7.3 Model Accuracy (%) Comparison (Mean and SD) with SOTA

Concept	1	2	3	4	5	avg
FedWeIT	61.0(±0.31)	62.0(±0.35)	66.8(±0.31)	72.6(±0.24)	70.7(±0.26)	68.3(±0.27)
EWC	82.3(±0.23)	74.5(±0.21)	72.3(±0.2)	92.1(±0.03)	85.9(±0.09)	86.7(±0.11)
TARGET	82.2(±0.20)	72.4(±0.19)	73.0(±0.23)	92.0(±0.02)	87.3(±0.06)	87.0(±0.09)
CM	85.4(±0.15)	85.2(±0.16)	86.5(±0.20)	93.3(±0.02)	92.4(±0.04)	90.3(±0.07)

Table 7.4 Forgetting Rate (%) Comparison (Mean and SD) with SOTA

Concept	1	2	3	4	5	avg
FedWeIT	4.76(±0.02)	4.16(±0.02)	5.89(±0.02)	1.71(±0.01)	3.68(±0.01)	3.15(±0.01)
EWC	3.36(±0.02)	3.17(±0.02)	4.30(±0.01)	1.04(±0.01)	2.65(±0.01)	2.17(±0.01)
TARGET	2.47(±0.01)	2.84(±0.01)	2.44(±0.02)	0.70(±0.01)	2.11(±0.01)	1.58(±0.01)
CM	0.16(±0.01)	0.29(±0.01)	0.32(±0.01)	0.09(±0.004)	0.34(±0.01)	0.17(±0.01)

CM algorithms effectiveness to match concepts of data to models. The matching effectiveness is defined as the percentage of correct concept matching over the entire training process (i.e., the collaborative client/server concept matching from the data to the models). Table 7.5 shows the concept matching effectiveness with a variety of clustering algorithms and distance metrics. To quantify the concept matching effectiveness separately with minimum influence from the clustering errors, we use the cluster ID that has the most number of clients with the given data concept, instead of the actual cluster ID for the mapping of a given client data concept. The results demonstrate that CM achieves up to 100% concept matching effectiveness. Table 7.5 demonstrate that CM provides the flexibility to use different clustering algorithms and distance metrics, as it performs well with all of them. As neural networks grow in size and complexity, the curse of dimensionality may manifest itself and requires advanced clustering algorithms and distance metrics.

Concept matching scalability. Table 7.6 shows CM performs well as the number of clients increases. With 80 clients, both the clustering and the CM algorithms perform perfectly. This is because the clustering algorithms generally perform better with larger number of samples. CM enjoys this benefit and achieves better model performance (up to 95.4%) with a larger number of clients. We further test CM with 20% increase or decrease in the size of CNN layer channels and dense layer neurons. A larger model can further stress-test CM, and a smaller model can reduce the communication overhead for CM. To avoid overfitting the model under the given dataset, we could not further downsize the model or split the data into more clients. Table 7.7 shows the performance metrics of CM, and there is a small improvement (90.4%) in model accuracy when using a larger model. Tables 7.6 and 7.7 show that both the clustering and the concept matching achieve near flawless performance.

Table 7.5 Matching Effectiveness (%) with 100 Rounds

	Kmean	Agglomerative	BIRCH	DBSCAN	OPTICS
Manhattan	100	100	100	97.4	93.4
Euclidean	100	99.8	100	90.4	94.9
Chebyshev	98.6	99.9	100	97.7	93.6

Table 7.6 Performance vs. # of Clients

	Matching effectiveness %	Model accuracy %
20	100	90.3
40	99.7	95.3
80	100	95.4

Resilience to number of concepts configured differently from ground truth. CM requires an estimated number of concepts configured in the initialization phrase. To understand its resiliency in case the system administrator fails to estimate correctly, we vary the number of concepts from 3 to 7, with 5 being the ground truth. As shown in Figure 7.5, when the configured number of concepts is higher (6 and 7) than the ground truth, 5 concept models (out of 6 or 7) learn the corresponding concepts smoothly. The extra concept models do not effect the smooth learning progress, and the average model accuracy achieves 90.5% and 90.0% respectively. When the number of concepts is smaller (4 and 3) than the ground truth, the system treats similar concepts (e.g., two FaceScrub concepts) as one. Although the model accuracy on the affected concepts (2, 3, and 5) exhibit minor fluctuations, the smooth learning progress for the other concepts (1 and 4) is not affected. Nevertheless, the average model accuracy achieves 89.5% and 88.9% respectively, and beats the state-of-the art solutions (87.0%). These results demonstrate CM has good resilience in terms of the estimated number of concepts configured. Moreover, the results suggest it is better if system administrators over-estimate the number of concepts, as performance remains strong in such instances.

Client operation overhead. Designed for mobile or IoT devices, CM is evaluated on a real IoT device in terms of the client end-to-end operation time.

Table 7.7 Performance vs. Model Size

	Matching effectiveness %	Model accuracy %
-20%	100	90.0
original	100	90.3
+20%	100	90.4

Table 7.8 Client Operation Time (Second) on Real IoT Device for SVHN, FaceScrub, MNIST, Fashion-MNIST, Not-MNIST, TrafficSigns "Super" Dataset

	Receiving model	Concept Matching	Training	Sending model	Total
Vanilla FL	0.67	N/A	85.27	0.67	86.61
CM	3.35	8.52	85.27	0.67	97.81

Compared with vanilla FL, the client operation overhead comes from receiving multiple concept models from the server and the client concept matching. Table 7.8 shows the breakdown of client operation time on the Qualcomm QCS605 IoT device in one round. We assume the worst-case scenario that the clients do not know the concepts, and perform concept matching every round. Compared with a multi-epoch training process over the entire experienced data, the concept matching can be achieved by testing only a portion of the data. The communication time is calculated as sending or receiving the model(s) of size 25.2MB over 300 Mbps WiFi network. The total client end-to-end operation time in one round is 97.81 seconds, which is feasible for a real-world deployment. Overall, the total CM operation time has a low overhead (11%) over vanilla FL operations on the IoT device. We believe the improvement in performance achieved by CM is worth this overhead cost.

Performance of clustering algorithms. Table 7.9 shows the results for 5 clustering algorithms whose parameters are detailed in Subsection 7.3.1. A perfect clustering can group all client models correctly with the same concept. Adjusted Rand Index (ARI) is a commonly used metric for clustering algorithms: 1.0 stands for perfect matching. Table 7.9 also shows the minimum ARI, as the worst clustering performance over 100 rounds. The results show that CM with BIRCH performs best, as it achieves up to 96 rounds of perfect clustering out of 100 and 0.994 average ARI.

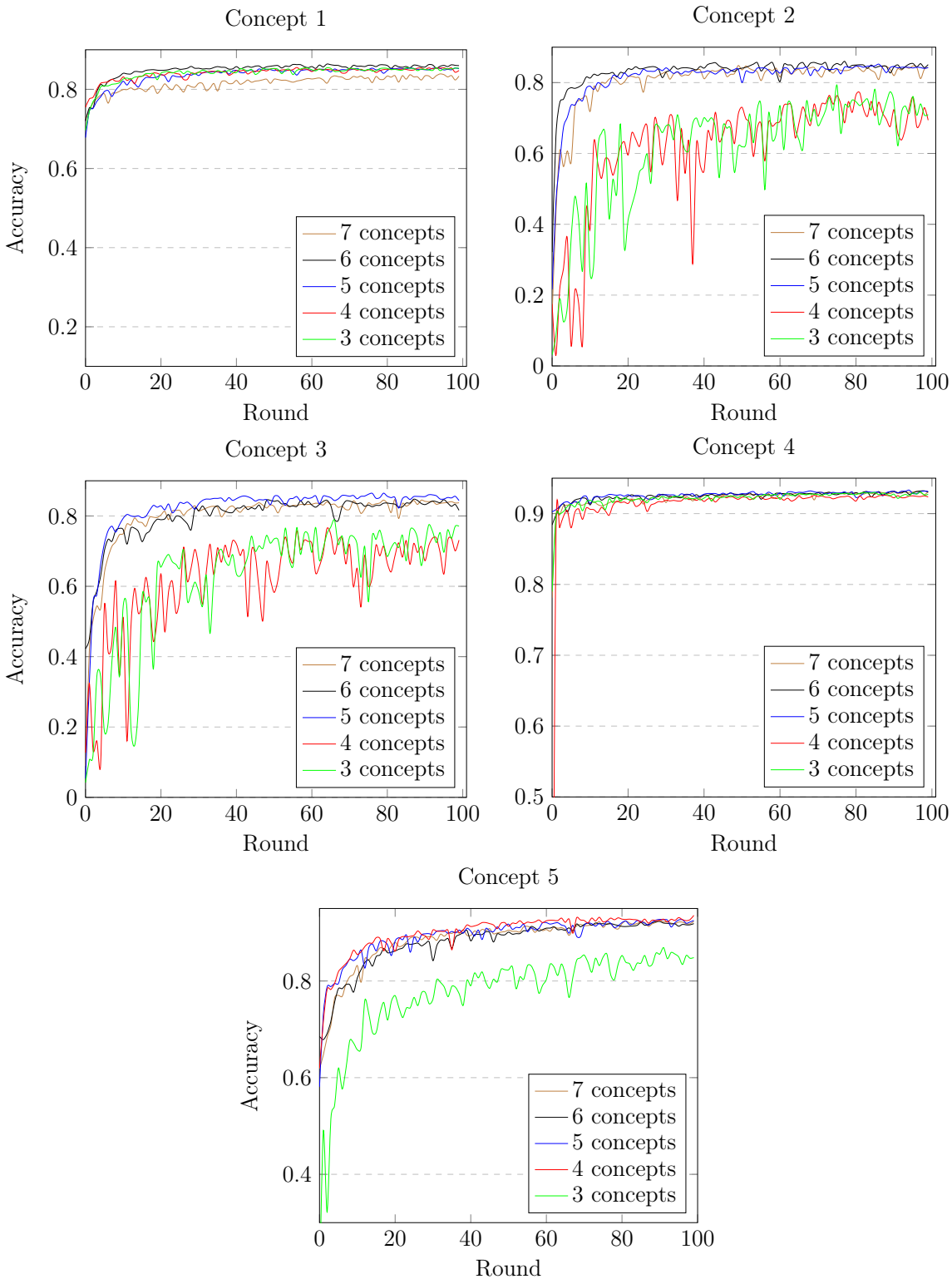


Figure 7.5 Model accuracy over communication rounds with different number of concepts configured.

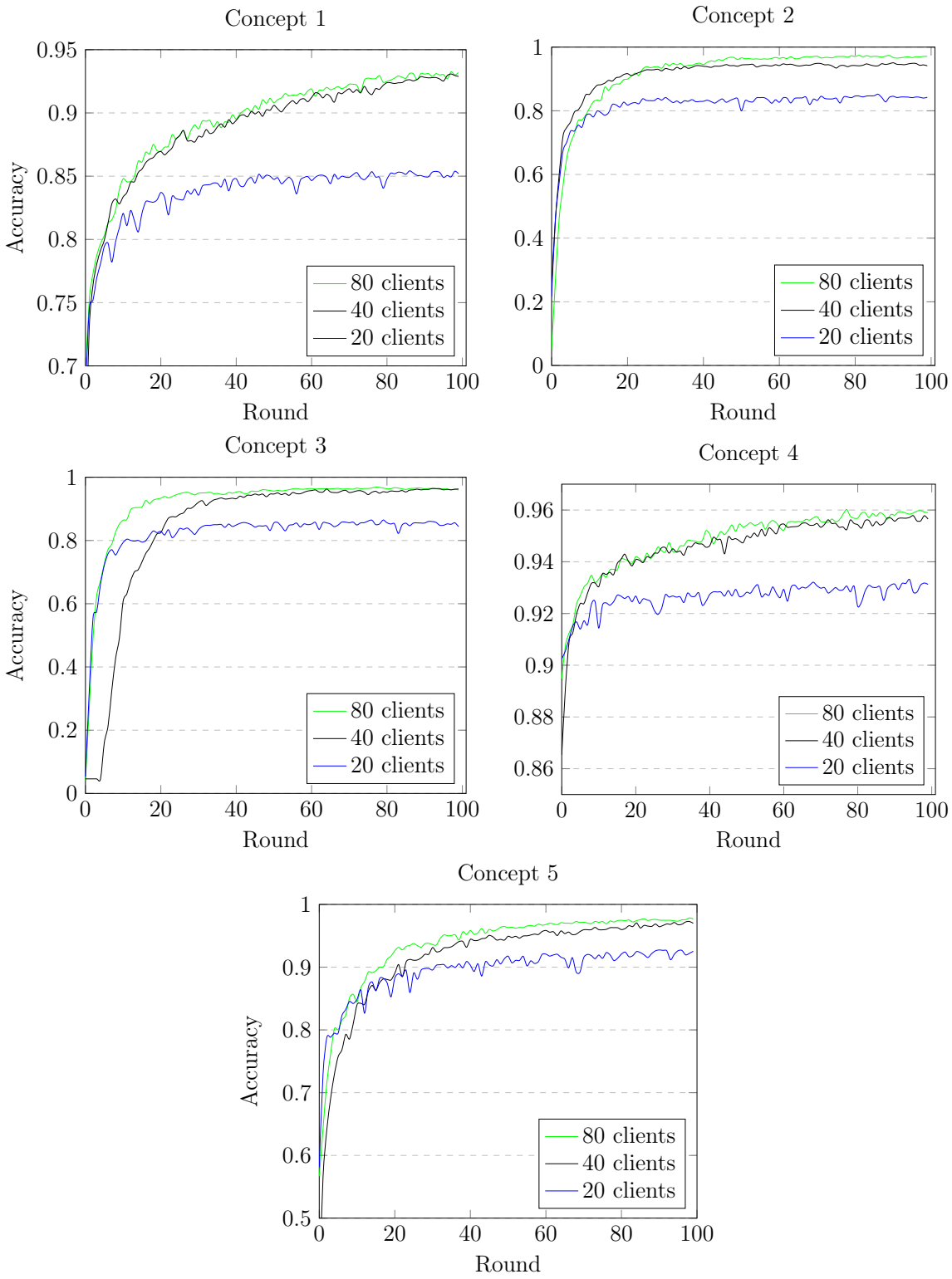


Figure 7.6 SVHN, FaceScrub, MNIST, Fashion-MNIST, Not-MNIST, TrafficSigns test set accuracy vs. communication rounds as number of clients increasing.

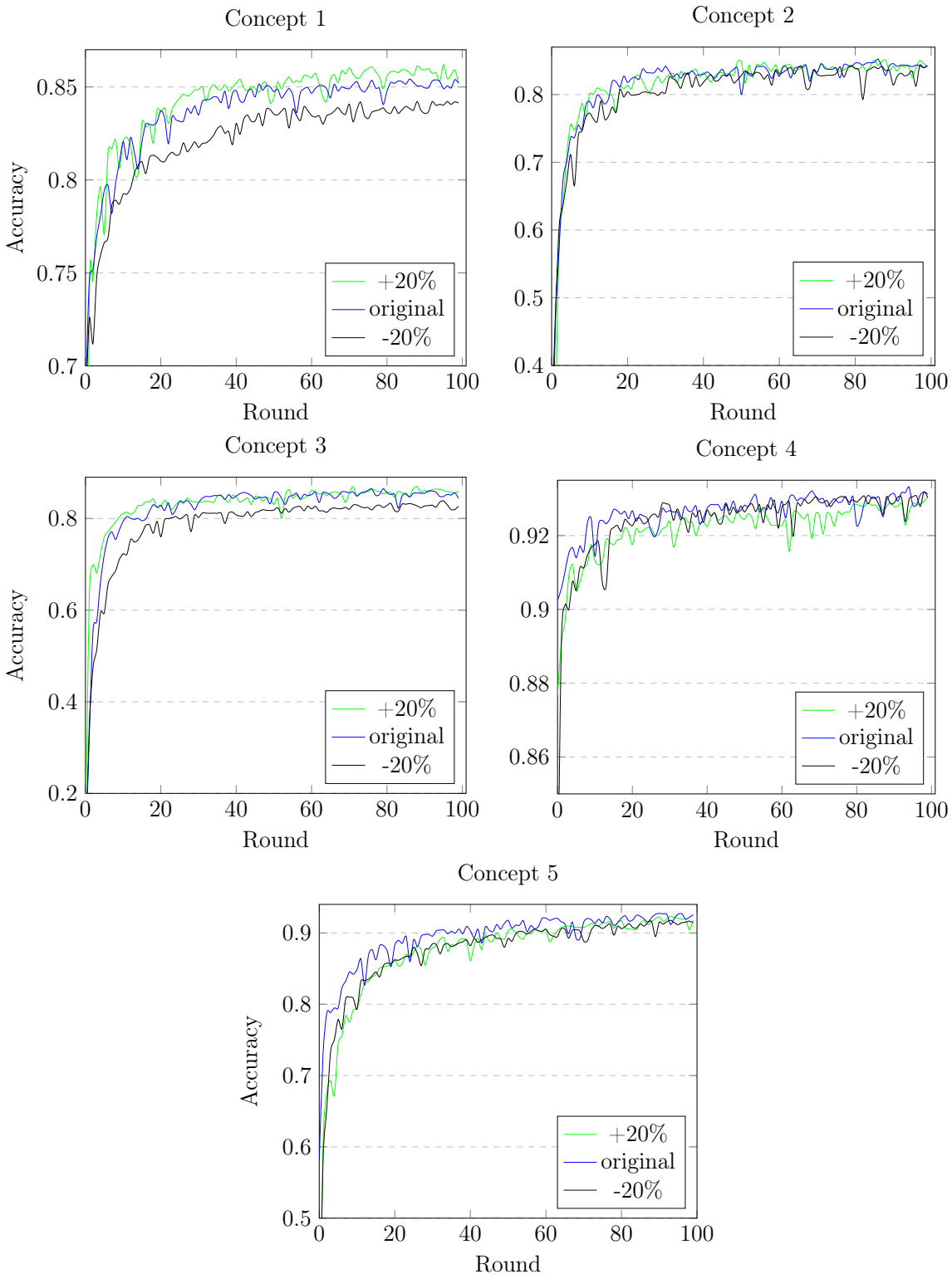


Figure 7.7 SVHN, FaceScrub, MNIST, Fashion-MNIST, Not-MNIST, TrafficSigns test set accuracy vs. communication rounds for training 20 clients with different model size.

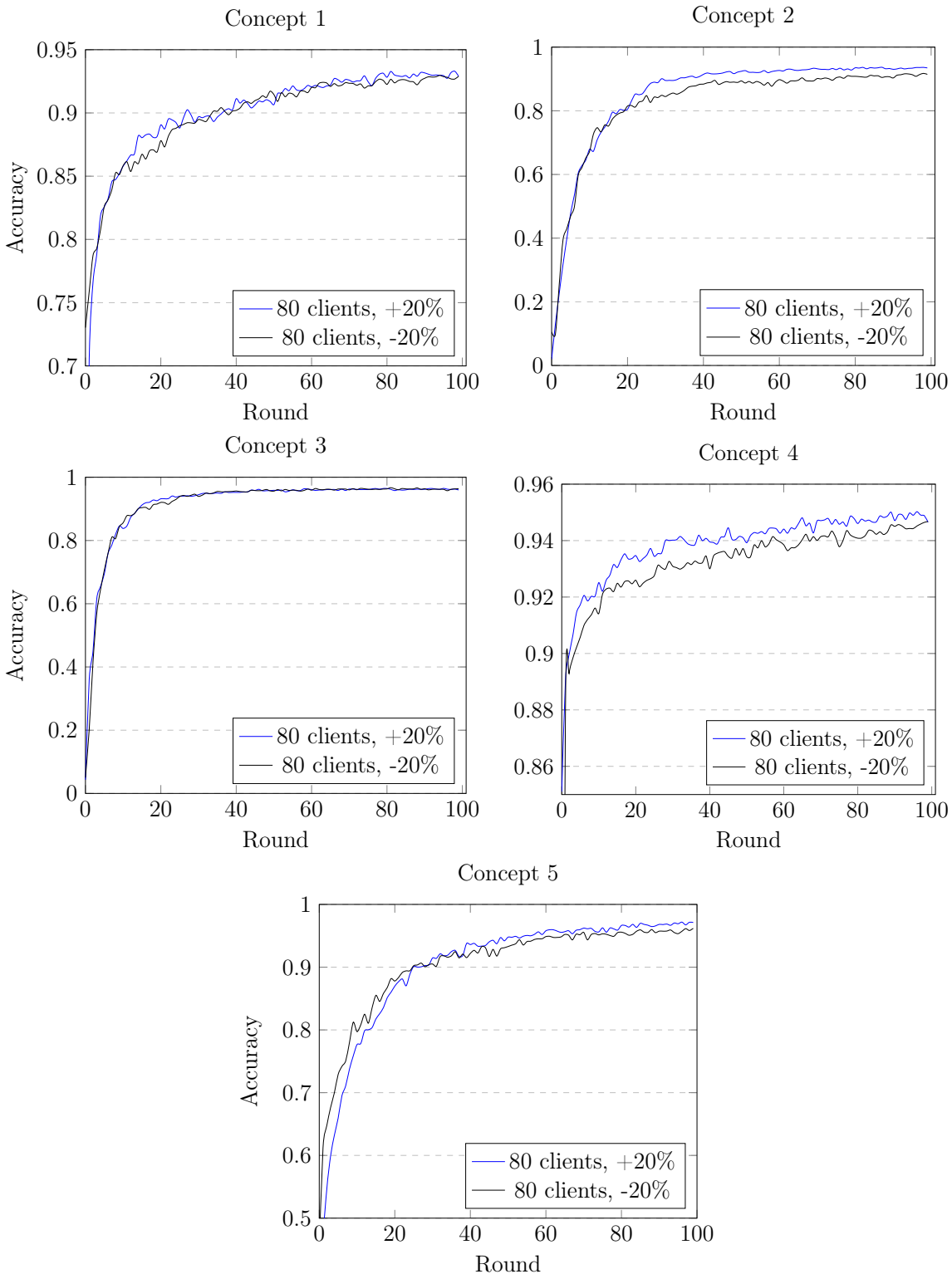


Figure 7.8 SVHN, FaceScrub, MNIST, Fashion-MNIST, Not-MNIST, TrafficSigns test set accuracy vs. communication rounds for training 80 clients with different model size.

Table 7.9 Clustering Performance with 100 Rounds Training for SVHN, FaceScrub, MNIST, Fashion-MNIST, Not-MNIST, TrafficSigns "Super" Dataset

	Kmean	Agglomerative	BIRCH	DBSCAN	OPTICS
Rounds with perfect clustering	91	93	96	66	50
ARI (average)	0.988	0.989	0.994	0.972	0.888
ARI (minimum)	0.713	0.704	0.771	0.700	0.478
Model accuracy % (average)	90.3	90.1	90.4	88.5	88.4

Furthermore, all algorithms perform reasonably well and achieve over 88.4% average model accuracy.

Model accuracy using CM under SVHN, FaceScrub, MNIST, Fashion-MNIST, Not-MNIST, TrafficSigns "super" dataset with different number of clients and model size. Figure 7.6 and 7.7 show the learning curves using CM, under SVHN, FaceScrub, MNIST, Fashion-MNIST, Not-MNIST, TrafficSigns "super" dataset with different number of clients and model size. The results show that CM can always learn smoothly under, and achieves up to 95.4% model accuracy (weighted average over the number of samples per concept). To stress test CM, we further investigate the model performance when training 80 clients with the network size increased and decreased 20%. The learning curves in Figure 7.8 demonstrate CM's smooth learning progress, as it achieves 94.3% and 94.9% average accuracy, respectively. These results illustrates CM scales well in terms of the number of clients and the moodel size.

Model accuracy using CM under TinyImagenet and Cifar100 "super" dataset with different number of clients and model size. Figure 7.9 and 7.10 show the learning curves using CM, under TinyImagenet and Cifar100 "super" dataset with different number of clients and model size. The results show that CM can learn each concepts well as number of clients or model size increasing. Because this "super" dataset is more difficult to learn, we observe fluctuations in the learning process.

CM vs. vanilla FL with each original dataset as a concept under SVHN, FaceScrub, MNIST, Fashion-MNIST, Not-MNIST, TrafficSigns

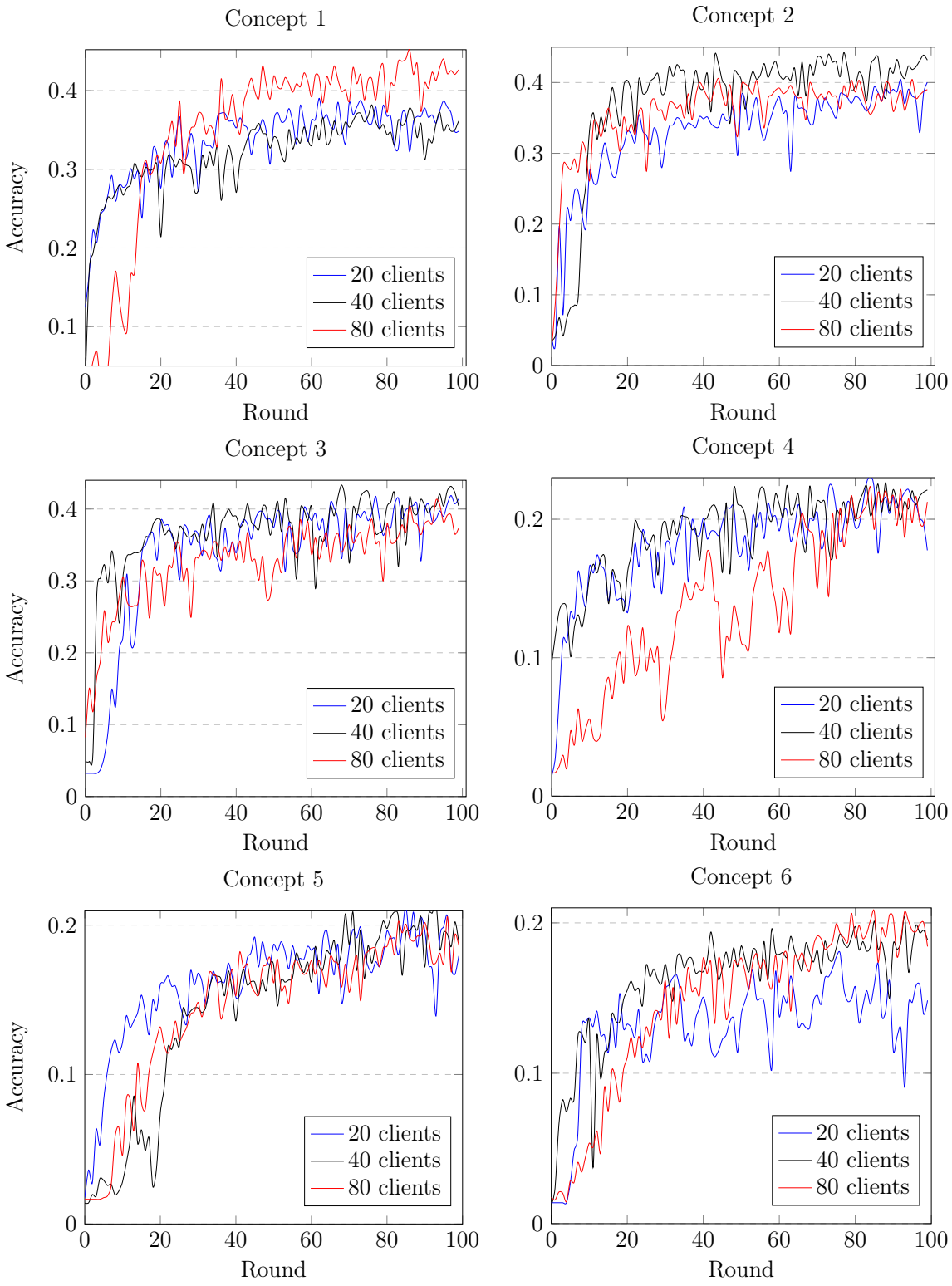


Figure 7.9 TinyImagenet and Cifar100 test set accuracy vs. communication rounds as number of clients increases.

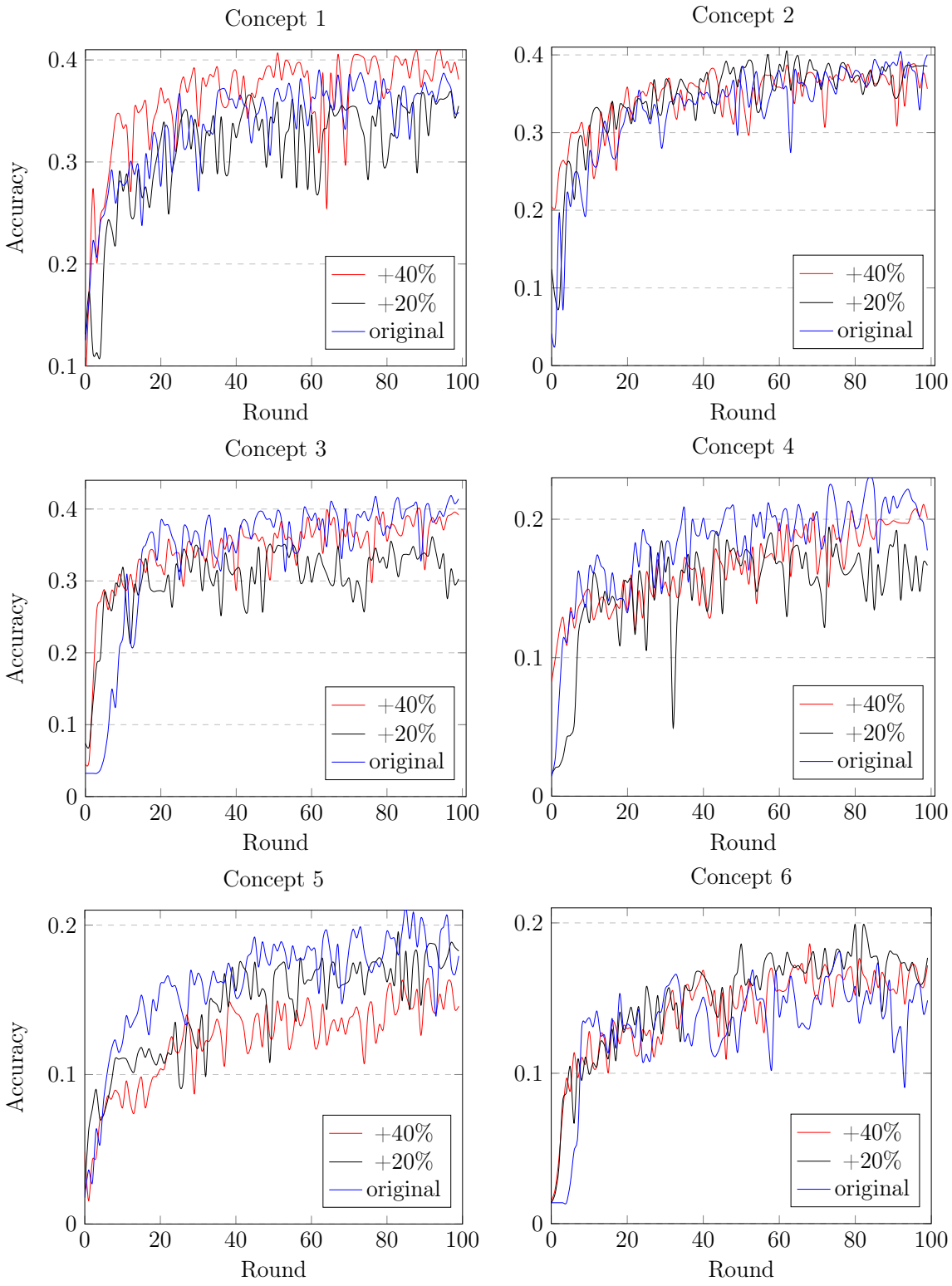


Figure 7.10 TinyImagenet and Cifar100 test set accuracy vs. communication rounds for training 20 clients with increasing model size.

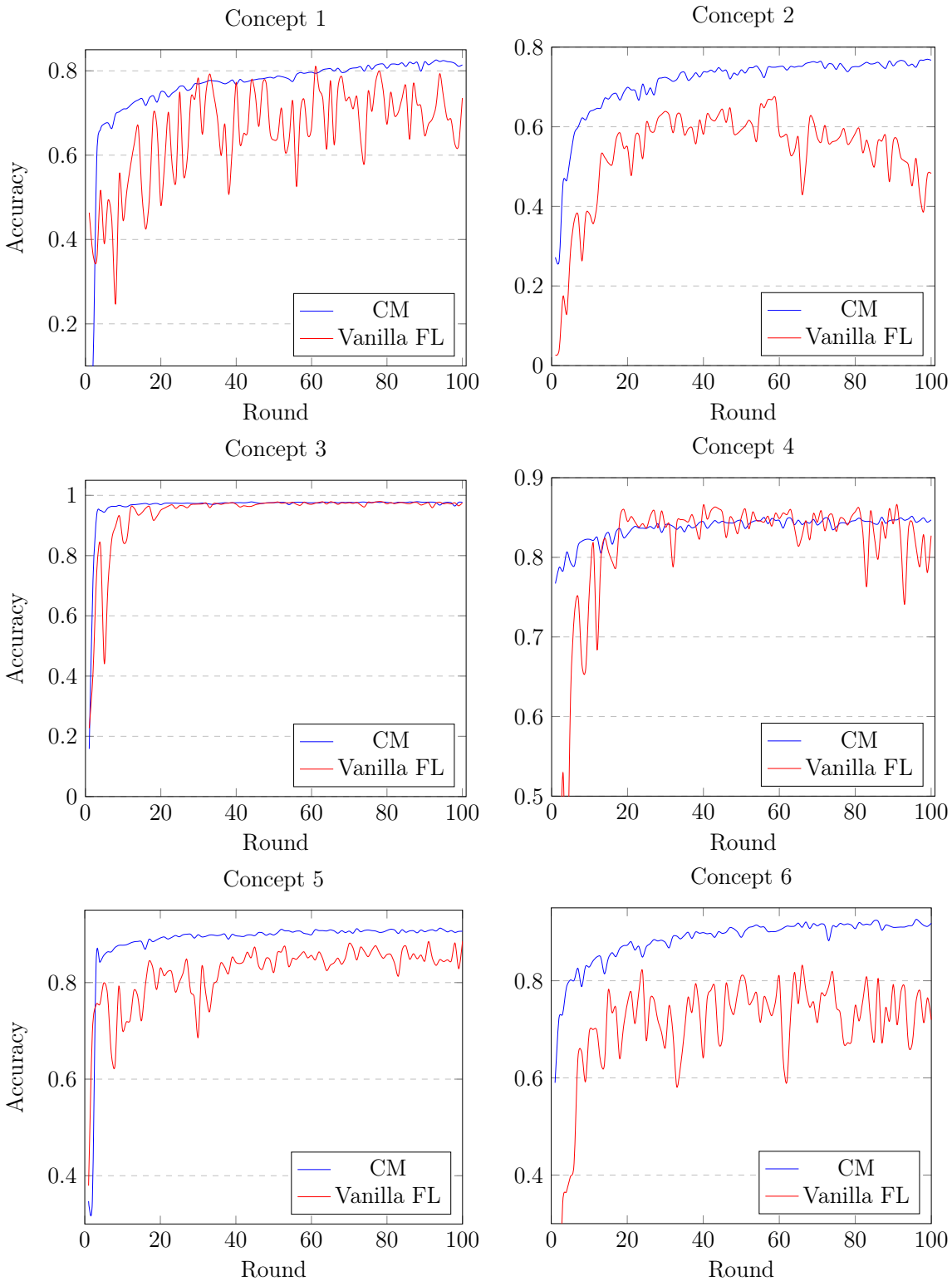


Figure 7.11 CM vs. vanilla FL with each original dataset as a concept: SVHN, FaceScrub, MNIST, Fashion-MNIST, Not-MNIST, TrafficSigns test set accuracy over communication rounds.

"super" dataset. We further diversify the experimental settings by treating each original dataset of SVHN, FaceScrub, MNIST, Fashion-MNIST, Not-MNIST, TrafficSigns as a concept without mixing or splitting the datasets. To evaluate CM comprehensively, we compare the test set accuracy between vanilla FL and CM. Similar to the original experimental settings, Figure 7.11 illustrates the smooth learning progress for CM over vanilla FL for all concepts. Compared with vanilla FL, the weighted average accuracy over the number of samples for all the concepts is improved from 85.5% to 88.0% as well.

7.4 Chapter Summary

Concept Matching (CM) is a novel FCL framework to alleviate catastrophic forgetting and interference among clients by training different models for different concepts concealed in the data. To avoid interference among clients, CM uses a clustering algorithm to group the client models with the same concept. To mitigate catastrophic forgetting, the server and the clients run concept matching algorithms that collaboratively train and update each concept model with the matching data of the same concept. Also, the server concept matching algorithm ensures the updating of the concept model in the correct gradient descent direction. CM achieves higher model accuracy than state-of-the-art systems, and works regardless of whether the clients are aware of the concepts or not. Our extensive evaluation also demonstrates that CM performs well with a variety of clustering algorithms and distance metrics, and scales well with the number of clients and the model size.

CHAPTER 8

CONCLUSIONS AND FUTURE DIRECTIONS

This dissertation firstly presented our experience with designing, building, and evaluating FLSys, an end-to-end federated learning system. We believe FLSys can open the path toward creating an FL ecosystem of models and apps for privacy-preserving deep learning on mobile sensing data. In the future, FLSys can be offered as an OS service. Our long-term goals are further extending it with robust aggregators, robust predictions, certified guarantees, advanced privacy and security solutions. Building upon FLSys, zoneFL leverages a mobile-edge-cloud architecture, adapting to user behaviors in different geographical zones to enhance scalability and model utility. Both FLSys and zoneFL have been evaluated through real-world deployments, demonstrating superior model performance, scalability, and fault-tolerance. As future work for ZoneFL, we will investigate how ZGD and ZMS work together to further improve model performance, and whether similarity of data distribution among zones should be considered when defining the zone neighborhood relationship.

Secondly, as a FL application with mobile sensing data, we proposed FMLL, a novel system for fine-grained location prediction that protects user privacy. The main novelties of FMLL are its meta-location concept to represent physical data and its prediction model. Our experiments demonstrated good prediction accuracy, model reusability, and system feasibility on smart phones. While FMLL is designed for predictions on smart phones, its model can also be used by network/service providers in their data centers. In the future, FMLL could be incorporated directly into smart phone OSs to improve system and app performance using location prediction. In addition, its GPS-based location prediction can be fused with location prediction done

by wireless network providers based on 5G signal fingerprinting techniques to further improve the end-user's experience in real-world applications such as augmented reality, mobile gaming, and video streaming.

Thirdly, we introduce Complement Sparsification (CS) as an FL pruning mechanism that minimizes bidirectional communication overhead between the server and clients, reduces computation overhead at the clients, and maintains good model accuracy. CS employs a complementary and collaborative pruning approach at both the server and clients. We evaluate CS experimentally in two applications: image classification and sentiment analysis. We demonstrate both analytically and experimentally that CS is an approximation of vanilla FL, resulting in models that perform well. Additionally, CS outperforms baseline pruning mechanisms for FL. Since CS is a generic solution in FL, it can be further enhanced with advanced aggregators, personalized FL mechanism, etc.

Fourthly, Federated Continual Learning (FCL) is explored as a more complex FL scenario where data accumulates over time and undergoes distributional changes. We propose Concept Matching (CM) for efficient FCL. The CM framework groups client models into clusters and uses novel CM algorithms to build different global models for various concepts in FL over time. In addition, a theoretically grounded server CM algorithm is proposed to effectively update a global concept model with a matching cluster model. Evaluations across multiple datasets demonstrate that CM outperforms state-of-the-art systems, works well with different clustering algorithms, and scales effectively with the number of clients and model size. In the future, we will design and implement a real-world FCL system using CM to showcase its efficiency.

FL offers a path to leveraging pervasive computing for training DL models in a privacy-preserving manner. Currently, the size of the models that can be handled is limited by the least powerful devices in the system. However, as larger models, such as Large Language Model, demonstrate superior performance in AI tasks, there

is an increasing demand for efficient collaborative training mechanisms for such large models through pervasive computing. This demand aligns with the principles of resource efficiency and sustainability, aiming to fully utilize idle computational resources at our disposal. Looking ahead, FL or distributed learning in general, will be explored to train larger models that exceed the resource capacity of individual devices. These efficient training mechanisms and the resulting models can be further utilized in autonomous systems in the physical world, enhancing their ability to serve us more effectively.

REFERENCES

[1] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konev̌cnỳ, S. Mazzocchi, H. B. McMahan *et al.*, "Towards federated learning at scale: System design," *arXiv preprint arXiv:1902.01046*, 2019.

[2] T. Li, A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, and V. Smith, "Federated optimization in heterogeneous networks," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 429–450, 2020.

[3] D. Sarkar, A. Narang, and S. Rai, "Fed-focal loss for imbalanced data classification in federated learning," *arXiv preprint arXiv:2011.06283*, 2020.

[4] Y. Zhao, M. Li, L. Lai, N. Suda, D. Civin, and V. Chandra, "Federated learning with non-iid data," *arXiv preprint arXiv:1806.00582*, 2018.

[5] M. Duan, D. Liu, X. Chen, R. Liu, Y. Tan, and L. Liang, "Self-balancing federated learning with global imbalanced data in mobile systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 01, pp. 59–71, Jan 2021.

[6] D. C. Verma, G. White, S. Julier, S. Pasteris, S. Chakraborty, and G. Cirincione, "Approaches to address the data skew problem in federated learning," in *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications*, T. Pham, Ed., vol. 11006, International Society for Optics and Photonics. SPIE, 2019, pp. 542 – 557.

[7] Y. Liu, A. Huang, Y. Luo, H. Huang, Y. Liu, Y. Chen, L. Feng, T. Chen, H. Yu, and Q. Yang, "Fedvision: An online visual object detection platform powered by federated learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 08, 2020, pp. 13 172–13 179.

[8] C. He, S. Li, J. So, X. Zeng, M. Zhang, H. Wang, X. Wang, P. Vepakomma, A. Singh, H. Qiu, X. Zhu, J. Wang, L. Shen, P. Zhao, Y. Kang, Y. Liu, R. Raskar, Q. Yang, M. Annavaram, and S. Avestimehr, "Fedml: A research library and benchmark for federated machine learning," *arXiv preprint arXiv:2007.13518*, 2020.

[9] FATE, "An Industrial Grade Federated Learning Framework," https://fate.fedai.org/, Accessed on 06/28/2024.

[10] OpenMined, "PySyft," https://blog.openmined.org/tag/pysyft/, Accessed on 06/28/2024.

[11] D. J. Beutel, T. Topal, A. Mathur, X. Qiu, J. Fernandez-Marques, Y. Gao, L. Sani, K. H. Li, T. Parcollet, P. P. B. de Gusmão *et al.*, "Flower: A friendly federated learning research framework," *arXiv preprint arXiv:2007.14390*, 2020.

[12] V. Mugunthan, A. Peraire-Bueno, and L. Kagal, "Privacyfl: A simulator for privacy-preserving and secure federated learning," in *Proceedings of the 29th ACM International Conference on Information and Knowledge Management*, 2020, pp. 3085–3092.

[13] T. Yang, G. Andrew, H. Eichner, H. Sun, W. Li, N. Kong, D. Ramage, and F. Beaufays, "Applied federated learning: Improving google keyboard query suggestions," *arXiv preprint arXiv:1812.02903*, 2018.

[14] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated learning: Challenges, methods, and future directions," *IEEE Signal Processing Magazine*, vol. 37, no. 3, pp. 50–60, 2020.

[15] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings *et al.*, "Advances and open problems in federated learning," *Foundations and Trends® in Machine Learning*, vol. 14, no. 1–2, pp. 1–210, 2021.

[16] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine learning: Concept and applications," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 10, no. 2, pp. 1–19, 2019.

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

[18] G. D. Jacobsen and K. H. Jacobsen, "Statewide covid-19 stay-at-home orders and population mobility in the united states," *World Medical and Health Policy*, vol. 12, no. 4, pp. 347–356, 2020.

[19] K. R. Choi, M. V. Heilemann, A. Fauer, and M. Mead, "A second pandemic: mental health spillover from the novel coronavirus (covid-19)," *Journal of the American Psychiatric Nurses Association*, vol. 26, no. 4, pp. 340–343, 2020.

[20] A. Ignatov, "Real-time human activity recognition from accelerometer data using convolutional neural networks," *Applied Soft Computing*, vol. 62, pp. 915–922, 2018.

[21] A. Murad and J.-Y. Pyun, "Deep recurrent neural networks for human activity recognition," *Sensors*, vol. 17, no. 11, p. 2556, 2017.

[22] F. Hernández, L. F. Suárez, J. Villamizar, and M. Altuve, "Human activity recognition on smartphones using a bidirectional lstm network," in *2019 XXII Symposium on Image, Signal Processing and Artificial Vision (STSIVA)*. IEEE, 2019, pp. 1–5.

[23] J. R. Kwapisz, G. M. Weiss, and S. A. Moore, "Activity recognition using cell phone accelerometers," *ACM SigKDD Explorations Newsletter*, vol. 12, no. 2, pp. 74–82, 2011.

[24] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. L. Reyes-Ortiz, "A public domain dataset for human activity recognition using smartphones." in *Esann*, vol. 3, 2013, p. 3.

[25] R. Chavarriaga, H. Sagha, A. Calatroni, S. T. Digumarti, G. Tröster, J. d. R. Millán, and D. Roggen, "The opportunity challenge: A benchmark database for on-body sensor-based activity recognition," *Pattern Recognition Letters*, vol. 34, no. 15, pp. 2033–2042, 2013.

[26] Y. Chen, K. Zhong, J. Zhang, Q. Sun, and X. Zhao, "Lstm networks for mobile human activity recognition," in *2016 International conference on artificial intelligence: technologies and applications*. Atlantis Press, 2016, pp. 50–53.

[27] H. Zhang and L. Dai, "Mobility prediction: A survey on state-of-the-art schemes and future applications," *IEEE Access*, vol. 7, pp. 802–822, 2019.

[28] R. Di Taranto, S. Muppirisetty, R. Raulefs, D. Slock, T. Svensson, and H. Wymeersch, "Location-aware communications for 5g networks: How location information can improve scalability, latency, and robustness of 5g," *IEEE Signal Processing Magazine*, vol. 31, no. 6, pp. 102–112, Nov 2014.

[29] A. De Brébisson, E. Simon, A. Auvolat, P. Vincent, and Y. Bengio, "Artificial neural networks applied to taxi destination prediction," in *Proceedings of the 2015th International Conference on ECML PKDD Discovery Challenge - Volume 1526*, ser. ECMLPKDDDC'15, 2015, pp. 40–51.

[30] J. Lv, Q. Sun, Q. Li, and L. Moreira-Matias, "Multi-scale and multi-scope convolutional neural networks for destination prediction of trajectories," *IEEE Transactions on Intelligent Transportation Systems*, pp. 1–12, 2019.

[31] T. Hoch, "An ensemble learning approach for the kaggle taxi travel time prediction challenge." in *Proceedings of the 2015th International Conference on ECML PKDD Discovery Challenge*, 2015.

[32] H. Wu, Z. Chen, W. Sun, B. Zheng, and W. Wang, "Modeling trajectories with recurrent neural networks," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 2017, pp. 3083–3090.

[33] M. Dash, K. K. Koo, J. B. Gomes, S. P. Krishnaswamy, D. Rugeles, and A. Shi-Nash, "Next place prediction by understanding mobility patterns," in *2015 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*, March 2015, pp. 469–474.

[34] B. T. Nguyen, N. V. Nguyen, N. T. Nguyen, and M. H. T. Tran, "A potential approach for mobility prediction using GPS data," in *2017 Seventh International Conference on Information Science and Technology (ICIST)*, April 2017, pp. 45–50.

[35] Q. Wu, X. Chen, Z. Zhou, and L. Chen, "Mobile social data learning for user-centric location prediction with application in mobile edge service migration," *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 7737–7747, 2019.

[36] Q. Liu, S. Wu, L. Wang, and T. Tan, "Predicting the next location: A recurrent model with spatial and temporal contexts," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI'16. AAAI Press, 2016, p. 194–200.

[37] J. Feng, Y. Li, C. Zhang, F. Sun, F. Meng, A. Guo, and D. Jin, "Deepmove: Predicting human mobility with attentional recurrent networks," in *Proceedings of the 2018 World Wide Web Conference*, 2018, pp. 1459–1468.

[38] J. Feng, C. Rong, F. Sun, D. Guo, and Y. Li, "Pmf: A privacy-preserving human mobility prediction framework via federated learning," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 4, no. 1, Mar. 2020.

[39] P. Wang, H. Wang, H. Zhang, F. Lu, and S. Wu, "A hybrid markov and lstm model for indoor location prediction," *IEEE Access*, vol. 7, pp. 185 928–185 940, 2019.

[40] C. H. Liu, Y. Wang, C. Piao, Z. Dai, Y. Yuan, G. Wang, and D. Wu, "Time-aware location prediction by convolutional area-of-interest modeling and memory-augmented attentive lstm," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1, 2020.

[41] D. Kong and F. Wu, "Hst-lstm: A hierarchical spatial-temporal long-short term memory network for location prediction," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, 7 2018, pp. 2341–2347.

[42] R. Wu, G. Luo, Q. Yang, and J. Shao, "Learning individual moving preference and social interaction for location prediction," *IEEE Access*, vol. 6, pp. 10 675–10 687, 2018.

[43] Y. Wang, N. J. Yuan, D. Lian, L. Xu, X. Xie, E. Chen, and Y. Rui, "Regularity and conformity: Location prediction using heterogeneous mobility data," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '15, 2015, p. 1275–1284.

[44] R. Trasarti, R. Guidotti, A. Monreale, and F. Giannotti, "Myway: Location prediction via mobility profiling," *Information Systems*, vol. 64, pp. 350 – 367, 2017.

[45] S. Xu, J. Cao, P. Legg, B. Liu, and S. Li, "Venue2vec: An efficient embedding model for fine-grained user location prediction in geo-social networks," *IEEE Systems Journal*, vol. 14, no. 2, pp. 1740–1751, 2019.

[46] D. Zhang, Y. Zhang, Q. Li, and D. Wang, "Sparse user check-in venue prediction by exploring latent decision contexts from location-based social networks," *IEEE transactions on Big Data*, vol. 7, no. 5, pp. 859–872, 2019.

[47] J. Krumm, "A survey of computational location privacy," *Personal and Ubiquitous Computing*, vol. 13, no. 6, pp. 391–399, 2009.

[48] M. Gruteser and B. Hoh, "On the anonymity of periodic location samples," in *Security in Pervasive Computing*. Springer, 2005, pp. 179–192.

[49] T. Xu and Y. Cai, "Feeling-based location privacy protection for location-based services," in *Proceedings of the 16th ACM conference on Computer and Communications Security (CCS)*. ACM, 2009, pp. 348–357.

[50] R. Popa, A. Blumberg, H. Balakrishnan, and F. Li, "Privacy and accountability for location-based aggregate statistics," in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2011, pp. 653–666.

[51] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *arXiv preprint arXiv:2001.08361*, 2020.

[52] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, and F. Kawsar, "An early resource characterization of deep learning on wearables, smartphones and internet-of-things devices," in *Proceedings of the 2015 international workshop on internet of things towards applications*, 2015, pp. 7–12.

[53] M. C. Mozer and P. Smolensky, *Skeletonization: A Technique for Trimming the Fat from a Network via Relevance Assessment*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, p. 107–115.

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

[55] B. Bartoldson, A. Morcos, A. Barbu, and G. Erlebacher, "The generalization-stability tradeoff in neural network pruning," *Advances in Neural Information Processing Systems*, vol. 33, pp. 20 852–20 864, 2020.

[56] A. S. Rakin, Z. He, L. Yang, Y. Wang, L. Wang, and D. Fan, "Robust sparse regularization: Defending adversarial attacks via regularized sparse network," in *Proceedings of the 2020 on Great Lakes Symposium on VLSI*, ser. GLSVLSI '20, 2020, p. 125–130.

[57] C. Wu, X. Yang, S. Zhu, and P. Mitra, "Mitigating backdoor attacks in federated learning," *arXiv preprint arXiv:2011.01767*, 2020.

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

[59] R. M. French, "Catastrophic forgetting in connectionist networks," *Trends in Cognitive Sciences*, vol. 3, no. 4, pp. 128–135, 1999.

[60] T. M. Mitchell, *Machine learning*. New York, NY, USA: McGraw Hill, 1997.

[61] X. Jiang, H. Hu, T. On, P. Lai, V. D. Mayyuri, A. Chen, D. M. Shila, A. Larmuseau, R. Jin, C. Borcea *et al.*, "Flsys: Toward an open ecosystem for federated learning mobile apps," *IEEE Transactions on Mobile Computing*, vol. 23, no. 1, pp. 501–519, 2022.

[62] J. Zhang, C. Chen, W. Zhuang, and L. Lyu, "Target: Federated class-continual learning via exemplar-free distillation," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023, pp. 4782–4793.

[63] J. Yoon, W. Jeong, G. Lee, E. Yang, and S. J. Hwang, "Federated continual learning with weighted inter-client transfer," in *International Conference on Machine Learning*. PMLR, 2021, pp. 12 073–12 086.

[64] F. E. Casado, D. Lema, M. F. Criado, R. Iglesias, C. V. Regueiro, and S. Barro, "Concept drift detection and adaptation for federated and continual learning," *Multimedia Tools and Applications*, pp. 1–23, 2022.

[65] Z. Wang, Y. Zhang, X. Xu, Z. Fu, H. Yang, and W. Du, "Federated probability memory recall for federated continual learning," *Information Sciences*, vol. 629, pp. 551–565, 2023.

[66] L. Yuan, Y. Ma, L. Su, and Z. Wang, "Peer-to-peer federated continual learning for naturalistic driving action recognition," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023, pp. 5249–5258.

[67] J. Zhang, C. Chen, W. Zhuang, and L. Lv, "Addressing catastrophic forgetting in federated class-continual learning," *arXiv preprint arXiv:2303.06937*, 2023.

[68] D. Qi, H. Zhao, and S. Li, "Better generative replay for continual federated learning," in *The Eleventh International Conference on Learning Representations*, 2023.

[69] J. Dong, L. Wang, Z. Fang, G. Sun, S. Xu, X. Wang, and Q. Zhu, "Federated class-incremental learning," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 10 164–10 173.

[70] Y. Ma, Z. Xie, J. Wang, K. Chen, and L. Shou, "Continual federated learning based on knowledge distillation," in *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence*, vol. 3, 2022, pp. 2182–2188.

[71] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, "Mobile edge computing: A survey," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 450–465, 2017.

[72] M. Murshed, C. Murphy, D. Hou, N. Khan, G. Ananthanarayanan, and F. Hussain, "Machine learning at the network edge: A survey," *arXiv preprint arXiv:1908.00080*, 2019.

[73] J. Ni, L. Muhlstein, and J. McAuley, "Modeling heart rate and activity data for personalized fitness recommendation," in *The World Wide Web Conference*, 2019, pp. 1343–1353.

[74] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konecný, S. Mazzocchi, H. B. McMahan, T. V. Overveldt, D. Petrou, D. Ramage, and J. Roselander, "Towards federated learning at scale: System design," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 374–388, 2019.

[75] Z. Wang, M. Song, Z. Zhang, Y. Song, Q. Wang, and H. Qi, "Beyond inferring class representatives: User-level privacy leakage from federated learning," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, April 2019, pp. 2512–2520.

[76] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" in *Advances in Neural Information Processing Systems*, 2014, pp. 3320–3328.

[77] S. Yu, P. Nguyen, A. Anwar, and A. Jannesari, "Adaptive dynamic pruning for non-iid federated learning," *arXiv preprint arXiv:2106.06921*, 2021.

[78] Y. Jiang, S. Wang, V. Valls, B. J. Ko, W.-H. Lee, K. K. Leung, and L. Tassiulas, "Model pruning enables efficient federated learning on edge devices," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 34, no. 12, pp. 10 374–10 386, 2022.

[79] W. Xu, W. Fang, Y. Ding, M. Zou, and N. Xiong, "Accelerating federated learning for iot in big data analytics with pruning, quantization and selective updating," *IEEE Access*, vol. 9, pp. 38 457–38 466, 2021.

[80] S. Horvath, S. Laskaridis, M. Almeida, I. Leontiadis, S. Venieris, and N. Lane, "Fjord: Fair and accurate federated learning under heterogeneous targets with ordered dropout," *Advances in Neural Information Processing Systems*, vol. 34, pp. 12 876–12 889, 2021.

[81] D. Wen, K.-J. Jeon, and K. Huang, "Federated dropout—a simple approach for enabling federated learning on resource constrained devices," *IEEE Wireless Communications Letters*, vol. 11, no. 5, pp. 923–927, 2022.

[82] S. Caldas, S. M. K. Duddu, P. Wu, T. Li, J. Kone**v**cnỳ, H. B. McMahan, V. Smith, and A. Talwalkar, "Leaf: A benchmark for federated settings," *arXiv preprint arXiv:1812.01097*, 2018.

[83] Z. Feng, H. Xiong, C. Song, S. Yang, B. Zhao, L. Wang, Z. Chen, S. Yang, L. Liu, and J. Huan, "Securegbm: Secure multi-party gradient boosting," in *2019 IEEE International Conference on Big Data (Big Data)*, 2019, pp. 1312–1321.

[84] TensorFlow, "On-Device Training with TensorFlow Lite," https://www.tensorflow.org/lite/examples/on_device_training/overview, Accessed on 06/28/2024.

[85] D. Verma, G. White, and G. de Mel, "Federated ai for the enterprise: A web services based implementation," in *2019 IEEE International Conference on Web Services (ICWS)*, 2019, pp. 20–27.

[86] Nvidia, "Nvidia FLARE," https://nvidia.github.io/NVFlare/index.html, Accessed on 06/28/2024.

[87] D. J. Beutel, T. Topal, A. Mathur, X. Qiu, J. Fernandez-Marques, Y. Gao, L. Sani, K. H. Li, T. Parcollet, P. P. B. de Gusmão, and N. D. Lane, "Flower: A friendly federated learning research framework," *arXiv preprint arXiv:2007.14390*, 2020.

[88] J. Zhang, Y. Zheng, D. Qi, R. Li, and X. Yi, "Dnn-based prediction model for spatio-temporal data," in *Proceedings of the 24th ACM SIGSPACIAL International Conference on Advances in Geographic Information Systems*, ser. SIGSPACIAL '16, 2016, pp. 92:1–92:4.

[89] B. Hitaj, G. Ateniese, and F. Perez-Cruz, "Deep models under the gan: information leakage from collaborative deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 603–618.

[90] T. Graepel, K. Lauter, and M. Naehrig, "Ml confidential: Machine learning on encrypted data," in *International Conference on Information Security and Cryptology*. Springer, 2012, pp. 1–21.

[91] J. Kone**v**cný, H. B. McMahan, D. Ramage, and P. Richtárik, "Federated optimization: Distributed machine learning for on-device intelligence," in *NIPS Optimization for Machine Learning Workshop*, 2015.

[92] C. Dwork, "A firm foundation for private data analysis," *Communications of the ACM*, vol. 54, no. 1, pp. 86–95, 2011.

[93] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor, "Our data, ourselves: Privacy via distributed noise generation," in *Advances in Cryptology - EUROCRYPT 2006: 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2006, pp. 486–503.

[94] C. Dwork, F. McSherry, K. Nissim, and A. Smith, "Calibrating noise to sensitivity in private data analysis," in *Theory of Cryptography: Third Theory of Cryptography Conference*, 2006, pp. 265–284.

[95] C. Dwork and A. Roth, "The algorithmic foundations of differential privacy," *Foundations and Trends® in Theoretical Computer Science*, vol. 9, no. 3–4, pp. 211–407, 2014.

[96] S. Wagh, X. He, A. Machanavajjhala, and P. Mittal, "Dp-cryptography: marrying differential privacy and cryptography in emerging applications," *Communications of the ACM*, vol. 64, no. 2, pp. 84–93, 2021.

[97] H. B. McMahan, D. Ramage, K. Talwar, and L. Zhang, "Learning differentially private recurrent language models," in *International Conference on Learning Representations*, 2018.

[98] A. Evfimievski, J. Gehrke, and R. Srikant, "Limiting privacy breaches in privacy preserving data mining," in *Proceedings of the Twenty-Second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '03, 2003, p. 211–222.

[99] S. P. Kasiviswanathan, H. K. Lee, K. Nissim, S. Raskhodnikova, and A. Smith, "What can we learn privately?" *SIAM Journal on Computing*, vol. 40, no. 3, pp. 793–826, 2011.

[100] U. Erlingsson, V. Pihur, and A. Korolova, "Rappor: Randomized aggregatable privacy-preserving ordinal response," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14, 2014, p. 1054–1067.

[101] M. Kim, O. Günlü, and R. F. Schaefer, "Federated learning with local differential privacy: Trade-offs between privacy, utility, and communication," IEEE, pp. 2650–2654, 2021.

[102] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ser. ICML'13. JMLR.org, 2013, p. III–1310–III–1318.

[103] H. B. McMahan, E. Moore, D. Ramage, and B. A. y Arcas, "Federated learning of deep networks using model averaging," *arXiv preprint arXiv:1602.05629*, vol. 2, no. 2, 2016.

[104] X. Jiang, S. Zhao, G. Jacobson, R. Jana, W.-L. Hsu, M. Talasila, S. A. Aftab, Y. Chen, and C. Borcea, "Federated meta-location learning for fine-grained location prediction," in *2021 IEEE International Conference on Big Data (Big Data)*. IEEE, 2021, pp. 446–456.

[105] M. R. Sprague, A. Jalalirad, M. Scavuzzo, C. Capota, M. Neun, L. Do, and M. Kopp, "Asynchronous federated learning for geospatial applications," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2018, pp. 21–28.

[106] A. Li, S. Wang, W. Li, S. Liu, and S. Zhang, "Predicting human mobility with federated learning," in *Proceedings of the 28th International Conference on Advances in Geographic Information Systems*, 2020, pp. 441–444.

[107] S. EK, F. PORTET, P. LALANDA, and G. VEGA, "A federated learning aggregation algorithm for pervasive computing: Evaluation and comparison," in *2021 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2021, pp. 1–10.

[108] A. Usmanova, F. Portet, P. Lalanda, and G. Vega, "A distillation-based approach integrating continual learning and federated learning for pervasive services," *arXiv preprint arXiv:2109.04197*, 2021.

[109] C. Briggs, Z. Fan, and P. Andras, "Federated learning with hierarchical clustering of local updates to improve training on non-iid data," in *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2020, pp. 1–9.

[110] R. Presotto, G. Civitarese, and C. Bettini, "Fedclar: Federated clustering for personalized sensor-based human activity recognition," in *2022 IEEE International Conference on Pervasive Computing and Communications (PerCom)*. IEEE, 2022, pp. 227–236.

[111] Y. Qin and M. Kondo, "Mlmg: Multi-local and multi-global model aggregation for federated learning," in *2021 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, 2021, pp. 565–571.

[112] F. Sattler, K.-R. Müller, and W. Samek, "Clustered federated learning: Model-agnostic distributed multitask optimization under privacy constraints," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 8, pp. 3710–3722, 2020.

[113] L. U. Khan, M. Alsenwi, Z. Han, and C. S. Hong, "Self organizing federated learning over wireless networks: A socially aware clustering approach," in *2020 International Conference on Information Networking (ICOIN)*. IEEE, 2020, pp. 453–458.

[114] C. He, C. Tan, H. Tang, S. Qiu, and J. Liu, "Central server free federated learning over single-sided trust social networks," *arXiv preprint arXiv:1910.04956*, 2019.

[115] Y. Mansour, M. Mohri, J. Ro, and A. T. Suresh, "Three approaches for personalization with applications to federated learning," *arXiv preprint arXiv:2002.10619*, 2020.

[116] V. Kulkarni, M. Kulkarni, and A. Pant, "Survey of personalization techniques for federated learning," in *2020 Fourth World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4)*. IEEE, 2020, pp. 794–797.

[117] T. Li, S. Hu, A. Beirami, and V. Smith, "Ditto: Fair and robust federated learning through personalization," in *International Conference on Machine Learning*. PMLR, 2021, pp. 6357–6368.

[118] Y. Deng, M. M. Kamani, and M. Mahdavi, "Adaptive personalized federated learning," *arXiv preprint arXiv:2003.13461*, 2020.

[119] K. Ozkara, N. Singh, D. Data, and S. Diggavi, "Quped: Quantized personalization via distillation with applications to federated learning," *Advances in Neural Information Processing Systems*, vol. 34, pp. 3622–3634, 2021.

[120] J. Guo, W. Ouyang, and D. Xu, "Multi-dimensional pruning: A unified framework for model compression," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 1508–1517.

[121] V. Sanh, T. Wolf, and A. Rush, "Movement pruning: Adaptive sparsity by fine-tuning," *Advances in Neural Information Processing Systems*, vol. 33, pp. 20 378–20 389, 2020.

[122] Y. Wang, X. Zhang, L. Xie, J. Zhou, H. Su, B. Zhang, and X. Hu, "Pruning from scratch," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 07, 2020, pp. 12 273–12 280.

[123] P. Han, S. Wang, and K. K. Leung, "Adaptive gradient sparsification for efficient federated learning: An online learning approach," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2020, pp. 300–310.

[124] S. Liu, G. Yu, R. Yin, and J. Yuan, "Adaptive network pruning for wireless federated learning," *IEEE Wireless Communications Letters*, vol. 10, no. 7, pp. 1572–1576, 2021.

[125] S. Vahidian, M. Morafah, and B. Lin, "Personalized federated learning by structured and unstructured pruning under data heterogeneity," in *2021 IEEE 41st International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 2021, pp. 27–34.

[126] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, and K. Chan, "Adaptive federated learning in resource constrained edge computing systems," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 6, pp. 1205–1221, 2019.

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

[128] A. Li, J. Sun, B. Wang, L. Duan, S. Li, Y. Chen, and H. Li, "Lotteryfl: Personalized and communication-efficient federated learning with lottery ticket hypothesis on non-iid datasets," *arXiv preprint arXiv:2008.03371*, 2020.

[129] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep Gradient Compression: Reducing the communication bandwidth for distributed training," in *The International Conference on Learning Representations*, 2018.

[130] L. Zhang, L. Shen, L. Ding, D. Tao, and L.-Y. Duan, "Fine-tuning global model via data-free knowledge distillation for non-iid federated learning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 10 174–10 183.

[131] L. Gao, H. Fu, L. Li, Y. Chen, M. Xu, and C.-Z. Xu, "Feddc: Federated learning with non-iid data via local drift decoupling and correction," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 10 112–10 121.

[132] X. Jiang, T. On, N. Phan, H. Mohammadi, V. D. Mayyuri, A. Chen, R. Jin, and C. Borcea, "Zone-based federated learning for mobile sensing data," in *2023 IEEE International Conference on Pervasive Computing and Communications (PerCom)*. IEEE, 2023, pp. 141–148.

[133] W. Huang, M. Ye, and B. Du, "Learn from others and be yourself in heterogeneous federated learning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 10 143–10 153.

[134] J. Geiping, H. Bauermeister, H. Dröge, and M. Moeller, "Inverting gradients-how easy is it to break privacy in federated learning?" *Advances in Neural Information Processing Systems*, vol. 33, pp. 16 937–16 947, 2020.

[135] K. Wei, J. Li, M. Ding, C. Ma, H. H. Yang, F. Farokhi, S. Jin, T. Q. Quek, and H. V. Poor, "Federated learning with differential privacy: Algorithms and performance analysis," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 3454–3469, 2020.

[136] X. Jiang and C. Borcea, "Complement sparsification: Low-overhead model pruning for federated learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 7, 2023, pp. 8087–8095.

[137] W. Wu, L. He, W. Lin, R. Mao, C. Maple, and S. Jarvis, "Safa: A semi-asynchronous protocol for fast federated learning with low overhead," *IEEE Transactions on Computers*, vol. 70, no. 5, pp. 655–668, 2020.

[138] X. Ouyang, Z. Xie, J. Zhou, G. Xing, and J. Huang, "Clusterfl: A clustering-based federated learning system for human activity recognition," *ACM Transactions on Sensor Networks*, vol. 19, no. 1, pp. 1–32, 2022.

[139] A. Ghosh, J. Chung, D. Yin, and K. Ramchandran, "An efficient framework for clustered federated learning," *Advances in Neural Information Processing Systems*, vol. 33, pp. 19 586–19 597, 2020.

[140] Y. Zhang, M. Duan, D. Liu, L. Li, A. Ren, X. Chen, Y. Tan, and C. Wang, "Csafl: A clustered semi-asynchronous federated learning framework," in *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2021, pp. 1–10.

[141] M. De Lange, R. Aljundi, M. Masana, S. Parisot, X. Jia, A. Leonardis, G. Slabaugh, and T. Tuytelaars, "A continual learning survey: Defying forgetting in classification tasks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 7, pp. 3366–3385, 2021.

[142] D. Rolnick, A. Ahuja, J. Schwarz, T. Lillicrap, and G. Wayne, "Experience replay for continual learning," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[143] D. Isele and A. Cosgun, "Selective experience replay for lifelong learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[144] H. Ahn, S. Cha, D. Lee, and T. Moon, "Uncertainty-based continual learning with adaptive regularization," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[145] R. Aljundi, K. Kelchtermans, and T. Tuytelaars, "Task-free continual learning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 11 254–11 263.

[146] A. Mallya and S. Lazebnik, "Packnet: Adding multiple tasks to a single network by iterative pruning," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2018, pp. 7765–7773.

[147] J. Serra, D. Suris, M. Miron, and A. Karatzoglou, "Overcoming catastrophic forgetting with hard attention to the task," in *International Conference on Machine Learning*. PMLR, 2018, pp. 4548–4557.

[148] X. Zuo, Y. Luopan, R. Han, Q. Zhang, C. H. Liu, G. Wang, and L. Y. Chen, "Fedvit: Federated continual learning of vision transformer at edge," *Future Generation Computer Systems*, 2023.

[149] Y. Guo, T. Lin, and X. Tang, "Towards federated learning on time-evolving heterogeneous data," *arXiv preprint arXiv:2112.13246*, 2021.

[150] B. Liu, L. Wang, and M. Liu, "Lifelong federated reinforcement learning: a learning architecture for navigation in cloud robotic systems," *IEEE Robotics and Automation Letters*, vol. 4, no. 4, pp. 4555–4562, 2019.

[151] L. T. Phong, Y. Aono, T. Hayashi, L. Wang, and S. Moriai, "Privacy-preserving deep learning via additively homomorphic encryption," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 5, pp. 1333–1345, 2018.

[152] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical secure aggregation for privacy-preserving machine learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, 2017, p. 1175–1191.

[153] N. Carlini, F. Tramer, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. Brown, D. Song, U. Erlingsson *et al.*, "Extracting training data from large language models," *arXiv preprint arXiv:2012.07805*, 2020.

[154] M. Nasr, R. Shokri, and A. Houmansadr, "Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 739–753.

[155] Z. He, T. Zhang, and R. B. Lee, "Model inversion attacks against collaborative inference," in *Proceedings of the 35th Annual Computer Security Applications Conference*, ser. ACSAC '19, 2019, p. 148–162.

[156] B. Hitaj, G. Ateniese, and F. Pérez-Cruz, "Deep models under the GAN: information leakage from collaborative deep learning," *CoRR*, vol. abs/1702.07464, 2017.

[157] L. Sun, J. Qian, and X. Chen, "LDP-FL: Practical private aggregation in federated learning with local differential privacy," *International Joint Conference on Artificial Intelligence*, 2021.

[158] Y. Tian, R. Wang, Y. Qiao, E. Panaousis, and K. Liang, "Flvoogd: Robust and privacy preserving federated learning," *arXiv preprint arXiv:2207.00428*, 2022.

[159] R. L., Y. C., H. C., R. G., and M. Y., "FLAME: differentially private federated learning in the shuffle model," vol. 35, no. 10, pp. 8688–8696, 2021.

[160] Ú. Erlingsson, V. Feldman, I. Mironov, A. Raghunathan, K. Talwar, and A. Thakurta, "Amplification by shuffling: From local to central differential privacy via anonymity," in *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2019, pp. 2468–2479.

[161] R. Liu, Y. Cao, M. Yoshikawa, and H. Chen, "Fedsel: Federated sgd under local differential privacy with top-k dimension selection," in *International Conference on Database Systems for Advanced Applications*, 2020, pp. 485–501.

[162] M. Malekzadeh, B. Hasircioglu, N. Mital, K. Katarya, M. E. Ozfatura, and D. Gündüz, "Dopamine: Differentially private federated learning on medical data," *arXiv preprint arXiv:2101.11693*, 2021.

[163] L. Lyu, Y. Li, X. He, and T. Xiao, "Towards differentially private text representations," in *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2020, pp. 1813–1816.

[164] Amazon Web Services, "Serverless Computing - AWS Lambda ," https://aws.amazon.com/lambda/, Accessed on 06/28/2024.

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

[166] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[167] M. Lin, Q. Chen, and S. Yan, "Network in network," *arXiv preprint arXiv:1312.4400*, 2013.

[168] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. Cambridge, United States: MIT press Cambridge, 2016, vol. 1, no. 2.

[169] E. Jeong, S. Oh, H. Kim, J. Park, M. Bennis, and S.-L. Kim, "Communication-efficient on-device machine learning: Federated distillation and augmentation under non-iid private data," *arXiv preprint arXiv:1811.11479*, 2018.

[170] S. J. Reddi, Z. Charles, M. Zaheer, Z. Garrett, K. Rush, J. Kone**v**cný, S. Kumar, and H. B. McMahan, "Adaptive federated optimization," in *International Conference on Learning Representations*, 2021.

[171] P. Lai, H. Phan, L. Xiong, K. P. Tran, M. Thai, T. Sun, F. Dernoncourt, J. Gu, N. Barmpalios, and R. Jain, "Bit-aware randomized response for local differential privacy in federated learning," https://openreview.net/forum?id=ZUXZKjfptc9, Accessed on 06/28/2024.

[172] J. C. Duchi, M. I. Jordan, and M. J. Wainwright, "Local privacy and statistical minimax rates," in *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, 2013, pp. 429–438.

[173] N. Wang, X. Xiao, Y. Yang, J. Zhao, S. C. Hui, H. Shin, J. Shin, and G. Yu, "Collecting and analyzing multidimensional data with local differential privacy," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, 2019, pp. 638–649.

[174] Y. Zhao, J. Zhao, M. Yang, T. Wang, N. Wang, L. Lyu, D. Niyato, and K.-Y. Lam, "Local differential privacy-based federated learning for internet of things," *IEEE Internet of Things Journal*, vol. 8, no. 11, pp. 8836–8853, 2021.

[175] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter," *arXiv preprint arXiv:1910.01108*, 2019.

[176] Amazon Web Services, "Lambda quotas," https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html, Accessed on 06/28/2024.

[177] A. Mathur, D. J. Beutel, P. P. B. de Gusmao, J. Fernandez-Marques, T. Topal, X. Qiu, T. Parcollet, Y. Gao, and N. D. Lane, "On-device federated learning with flower," *arXiv preprint arXiv:2104.03042*, 2021.

[178] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[179] X. Li, M. Jiang, X. Zhang, M. Kamp, and Q. Dou, "Fedbn: Federated learning on non-iid features via local batch normalization," *arXiv preprint arXiv:2102.07623*, 2021.

[180] "Deployment-aws local zones," https://aws.amazon.com/about-aws/global-infrastructure/localzones/, Accessed on 06/28/2024.

[181] Microsoft Azure, "Azure private multi-access edge compute (mec)," https://azure.microsoft.com/en-us/solutions/private-multi-access-edge-compute-mec/, Accessed on 06/28/2024.

[182] Y. Vaizman, "A mobile application for behavioral context recognition and data collection in the wild," http://extrasensory.ucsd.edu/ExtraSensoryApp/, Accessed on 06/28/2024.

[183] Polar, "SDK for Polar sensors," https://github.com/polarofficial/polar-ble-sdk, Accessed on 06/28/2024.

[184] H. Hu, X. Jiang, V. D. Mayyuri, A. Chen, D. M. Shila, A. Larmuseau, R. Jin, C. Borcea, and N. Phan, "Flsys: Toward an open ecosystem for federatedlearning mobile apps," *arXiv preprint arXiv:2111.09445*, 2021.

[185] J. Ni, L. Muhlstein, and J. McAuley, "Modeling heart rate and activity data for personalized fitness recommendation," in *The World Wide Web Conference*, ser. WWW '19, 2019, p. 1343–1353.

[186] K. Chen, D. Zhang, L. Yao, B. Guo, Z. Yu, and Y. Liu, "Deep learning for sensor-based human activity recognition: Overview, challenges, and opportunities," *ACM Computing Surveys (CSUR)*, vol. 54, no. 4, pp. 1–40, 2021.

[187] M. A. Shafique and E. Hato, "Use of acceleration data for transportation mode prediction," *Transportation*, vol. 42, no. 1, pp. 163–188, 2015.

[188] C. M. Bishop, *Neural networks for pattern recognition.* Oxford, United Kingdom: Oxford University Press, 1995.

[189] S. Zhao, R. Bharati, C. Borcea, and Y. Chen, "Privacy-aware federated learning for page recommendation," in *Proceedings of 2020 IEEE International Conference on Big Data*, 2020.

[190] T. Kashiyama, Y. Pang, and Y. Sekimoto, "Open pflow: Creation and evaluation of an open dataset for typical people mass movement in urban areas," *Transportation Research Part C: Emerging Technologies*, vol. 85, pp. 249 – 267, 2017.

[191] Y. Zheng, L. Wang, R. Zhang, X. Xie, and W. Ma, "Geolife: Managing and understanding your past life over maps," in *The Ninth International Conference on Mobile Data Management (MDM 2008)*, 2008, pp. 211–212.

[192] H. Wang, M. Yurochkin, Y. Sun, D. Papailiopoulos, and Y. Khazaeni, "Federated learning with matched averaging," *arXiv preprint arXiv:2002.06440*, 2020.

[193] O-RAN Alliance, "O-ran use cases and deployment scenarios," https://mediastorage.o-ran.org/white-papers/O-RAN.WG1.Use-Cases-and-Deployment-Scenarios-White-Paper-2020-02.pdf, Tech. Rep., Accessed on 06/28/2024.

[194] S. Caldas, S. M. K. Duddu, P. Wu, T. Li, J. Kone**v**cný, H. B. McMahan, V. Smith, and A. Talwalkar, "Leaf: A benchmark for federated settings," *arXiv preprint arXiv:1812.01097*, 2018. [Online]. Available: https://arxiv.org/abs/1812.01097

[195] V. Lomonaco and D. Maltoni, "Core50: a new dataset and benchmark for continuous object recognition," in *Conference on Robot Learning*. PMLR, 2017, pp. 17–26.

[196] M. Ester, H.-P. Kriegel, J. Sander, X. Xu *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise." in *KDD'96: Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, vol. 96, no. 34, 1996, pp. 226–231.

[197] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander, "Optics: Ordering points to identify the clustering structure," *ACM SIGMOD record*, vol. 28, no. 2, pp. 49–60, 1999.

[198] M. A. Carreira-Perpinán, "A review of dimension reduction techniques," *Department of Computer Science, University of Sheffield, Technical Report, CS-96-09*, vol. 9, pp. 1–69, 1997.

[199] V. Mothukuri, R. M. Parizi, S. Pouriyeh, Y. Huang, A. Dehghantanha, and G. Srivastava, "A survey on security and privacy of federated learning," *Future Generation Computer Systems*, vol. 115, pp. 619–640, 2021.

[200] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proceedings of the IEEE*, vol. 108, no. 4, pp. 485–532, 2020.

[201] D. Maltoni and V. Lomonaco, "Continuous learning in single-incremental-task scenarios," *Neural Networks*, vol. 116, pp. 56–73, 2019.

[202] T. G. Dietterich, "Ensemble methods in machine learning," in *Multiple Classifier Systems: First International Workshop*. Springer, 2000, pp. 1–15.

[203] D. Shim, Z. Mai, J. Jeong, S. Sanner, H. Kim, and J. Jang, "Online class-incremental continual learning with adversarial shapley value," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 11, 2021, pp. 9630–9638.

[204] Z. Mai, R. Li, H. Kim, and S. Sanner, "Supervised contrastive replay: Revisiting the nearest class mean classifier in online class-incremental continual learning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 3589–3599.

[205] Z. Mai, R. Li, J. Jeong, D. Quispe, H. Kim, and S. Sanner, "Online continual learning in image classification: An empirical survey," *Neurocomputing*, vol. 469, pp. 28–51, 2022.