Collaborative Learning between Cloud and End Devices: An Empirical Study on Location Prediction

Yan Lu[†‡], Yuanchao Shu[‡], Xu Tan[‡], Yunxin Liu[‡], Mengyu Zhou[‡], Qi Chen[‡], Dan Pei[*]
Microsoft Research[‡], New York University[†], Tsinghua University[*]

ABSTRACT

Over the years, numerous learning methods have been put forward to model and predict different user behaviors on end devices (*e.g.,* ads click, location change, app launch). While the learn-then-deploy approaches achieve promising results in many scenarios, data heterogeneity and variability throw impediment in the way of deploying pre-learned models to a large cluster of end devices. On the other hand, learning on devices like smartphones suffers from limited data, computing power and energy budget. This paper proposes Colla, a collaborative learning approach for behavior prediction that allows cloud and devices to learn collectively and continuously. Colla finds a middle ground to build tailored model for each device, leveraging local data and computation resources to update the model, while at the same time exploits cloud to aggregate and transfer device-learned knowledge across the network to solve the cold-start problem and prevent over-fitting. We fully implemented Colla with a multi-feature RNN model on both smartphones and in cloud, and applied it to predict user locations. Evaluation results based on large-scale real data show that compared with training using centralized data, Colla improves prediction accuracy by 21%. Our experiments also validate the efficiency of Colla, showing that one overnight training on a commodity smartphone can process one-year data from a typical smartphone, at the cost of 2000mWh and few hundreds KB communication overhead.

CCS CONCEPTS

• **Computer systems organization** → **Distributed architectures**; *Cloud computing*; • **Information systems** → Data analytics; • **Networks** → Network algorithms.

KEYWORDS

edge computing; collaborative learning; smartphone; DNN; location prediction; knowledge distillation

ACM Reference Format:
Yan Lu, Yuanchao Shu, Xu Tan, Yunxin Liu, Mengyu Zhou, Qi Chen, Dan Pei. 2019. Collaborative Learning between Cloud and Edge Devices: An Empirical Study on Location Prediction. In *SEC '19: ACM/IEEE Symposium on Edge Computing, November 7–9, 2019, Arlington, VA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3318216.3363304

1 INTRODUCTION

User behavior prediction on end devices (*e.g.,* PCs, laptops, smartphones) has long been a topic of interest in both system and machine learning community. Researches have been carried out to predict, for example, online user behaviors from weblogs [1], word input [2] and app usages [3] from OS system logs, and physical activities such as locations which user will visit [4] based on data from portable sensing devices. These prediction results benefit numerous third-party applications including personal digital assistant, recommendation systems, advertising etc.

Despite a broad range of applications, the vast majority of approaches used for user behavior prediction are data-driven - use data mining and machine learning techniques to learn behavior models from historical data [5–11]. More recently, Deep Neural Networks (DNNs) have also been widely applied in behavior modeling due to its great success in sequence prediction [4, 12, 13]. Albeit implementation differences, these approaches learn in a centralized way (*e.g.,* in the cloud) from labeled activities, and make predictions on each end device (*e.g.,* desktops and smartphones).

This classical learning paradigm has several downsides. First, most designs focus on building *one single model* that achieves the best performance on a given "mixed-user" dataset. However, when deployed to end devices, the global model may perform badly for some users due to the skewed data distributions. For instance, mobility patterns of kids, college students and company employees are distintive from each other. A natural solution is to train different models for different groups of users. However, it is non-trivial to determine the number of groups as well as the consequent methodology of training and deployment. Second, to achieve good performance on the "mixed-user" dataset, large model capacity is needed, eventually increasing the resource usage on end devices. Third, the learn-then-deploy paradigm does not take runtime model execution results as well as newly-generated data on end devices into account, failing to adapt to data drift over time.

This paper is driven by a simple question: can each end device (*e.g.,* smartphones) learn its own prediction model and improve it over time? We believe the answer is yes. Our key insight is that despite limited amount of data and skewed data distribution on each device, in user behavior prediction, there exist common patterns [14] cross users/devices due to the intrinsic correlations in the physical space, hence allowing cloud and (multiple) end devices to complement each other and *learn collectively*. For example, users may install very different apps on their smartphones. But all these apps come from the same app store, and Yelp, for instance, is more likely to be launched around the same time (*e.g.,* lunch time) on all devices.

Design of such a system, however, has to address several challenges. The key is to enable effective and efficient local training, allowing knowledge transfer from both cloud to device and device to device, while at the same time, accommodating the huge gap between the cloud and end devices in terms of compute power, data variation, and energy budget. To realize the benefits of cloud-device collaboration, we let the cloud side do the heavy lifting at beginning to train an initial cloud model. Devices then take over to perform *incremental* learning tasks using their local data, and build their own models (*a.k.a.,* client model) in a distributed way for local inference. In the meantime, cloud serves as a sink node that enables knowledge share across the network from time to time, expediting learning progress on each device and alleviating the impact of insufficient data and over-fitting.

Under this framework, we choose recurrent neural networks (RNN) as a template predictive model, and further propose diverse model architectures for cloud and end devices. Specifically, cloud hosts a heavy model with more layers and a larger hidden layer size, resulting in a larger capacity, while each device only maintains a lightweight model tailored to local data and computing power. To enable knowledge sharing between models, an update mechanism using a novel *dual knowledge distillation* approach is devised. At the client side, akin to classical knowledge distillation [15, 16], device fine-tunes its model using both local ground truth (*i.e.,* hard labels) as well as *soft labels* created by the cloud model, whereas on the cloud side, the cloud model is updated by distilling the knowledge from lightweight device models[1]. To deal with data variation, a model grouping mechanism is incorporated in the cloud. It dynamically classifies device models, and updates the cloud by generating multiple cloud models. This way, each client only pulls one cloud model and hence client model benefits from peer devices' training process on correlated data. The cloud model is also enhanced over time, providing a good baseline for new coming devices thus solving the cold-start problem. In what follows, we first talk about problem formulation and model design in §2, and then elaborate knowledge transfer and model update in §3.

To explore the feasibility of applying collaborative learning to user behavior prediction and quantify the benefits along multiple dimensions, we take smartphone as an example and conduct a case study on location prediction. Our empirically experiments over a large-scale dataset collected from real users reveal the following key observations. First, we found that, despite limited amount of data and computation resources, end devices can still develop knowledge of their mobility patterns promptly and efficiently by training a neural network model. The key is to customize device model architecture and leverage cloud to bootstrap the learning process. Second, by exploiting knowledge distilled from the crowd, the collaboratively-trained model achieves a higher prediction accuracy than both centralized-trained model based on the aggregated data and the client-trained models by individual devices, as well as state-of-the-art baseline prediction methods. Third, prediction accuracy increases with successive model updates with the help of model grouping. Nevertheless, device variations are observed and the gain from each update diminishes over time. Our main contributions in the paper are as follows.

[1]Note that raw data uploading is not required.

- We revisit the problem of user behavior prediction, and propose Colla, a collaborative learning framework that allows devices and the cloud to learn collectively. In contrast to the traditional learn-then-deploy paradigm, Colla allows local devices to play an active role in learning their own data, and wisely leverages the cloud and other devices for both data and computational resources.
- We study the feasibility of Colla by applying a multi-feature RNN network to the problems of smartphone-based location prediction. A novel dual distillation mechanism with model customization and grouping mechanisms is proposed, demonstrating superior prediction accuracy over the state-of-the-art methods.
- We fully implement Colla on Android and Microsoft Azure, and evaluate its performance on a large-scale real dataset. Key observations and insights are reported, shedding light on how collaborative learning would work for behavior prediction in practice, and how a better learning system can be designed with a swarm of networked devices.

2 USER BEHAVIOR PREDICTION: A NON-COLLABORATIVE PERSPECTIVE

We set the context by formulating behavior prediction problems from a non-collaborative perspective (§2.1). This represents a class of classical prediction algorithms that run on centralized data. Amidst these methods, RNN has demonstrated superior performance recently due to its ability on handling sequence dependence. Hence, in § 2.2, we first introduce an RNN with Long Short-Term Memory (LSTM) unit, and use it as our template model design. Note that although the design of RNN is not a key contribution of this paper, it lays a foundation on the design of Colla (§3), making it a generally applicable method for different behavior predictions.

2.1 Problem Formulation

Sequence data is one of the most widely collected data on end devices. The particular focus of this paper on sequence prediction is to decide with given time series patterns – (randomly sampled) data observations from the past – can we learn to make reasonable prediction on future data, such as the next location. Formally, we define the problem as follows.

Definition 2.1 (Sequence.). S^d is a sequence of samples $q_1^d q_2^d \ldots q_n^d$ on device d, where $q_i^d = (t_i, v_i)$, t_i and $v_i \in V$ are the timestamp and value of sample q_i^d. The time gap between t_i and t_{i+1} may vary between samples, and we mainly focus on the discrete sequence where v_i is a discrete value, such as location landmarks and application ID.

Definition 2.2 (Sequence Prediction Problem.). Given sequence data S^d, where d is from a set of end devices $D = \{d^0, \ldots d^k\}$, sequence prediction problem seeks to predict the next data point q_{n+1}^d for each device $d \in D$.

Note that q_{n+1}^d is a tuple of timestamp t_{n+1} and value v_{n+1}, which makes prediction a complicated multivariate prediction problem. In this study, the timestamp information is taken as the input of RNN and the value v_{n+1} is the prediction value. Therefore, we can vary input t_{n+1} to predict the next value v_{n+1} at any given time. As we assume discrete values of a sequence, the prediction can be regarded as a series of multiclass classification problem, with each class represents the possible value of v_{n+1}.

In § 2.2, we take location prediction as an example to describe our model design. In this situation, the sequence data consists of trajectory sequence which records the time and location ID of a device. Note that in many applications like keyboard, location and app launch prediction, ground truth values v_{n+1} at t_{n+1} can be naturally obtained so there is no need for local data labeling.

2.2 Prediction Methods

RNNs has demonstrated outstanding performance recently due to its advantages in sequence dependency modeling, generalization ability, high flexibility and scalability [4, 12, 13]. Considering contextual data such as time also provides valuable information for user-behavior prediction, we propose a Multiple Feature RNN (M-RNN) model. Figure 1 shows the structure of M-RNN. In mobility prediction, the M-RNN model takes multiple sequence data streams as input, and outputs a vector of multi-class probabilities with each entry representing the probability of each possible location.

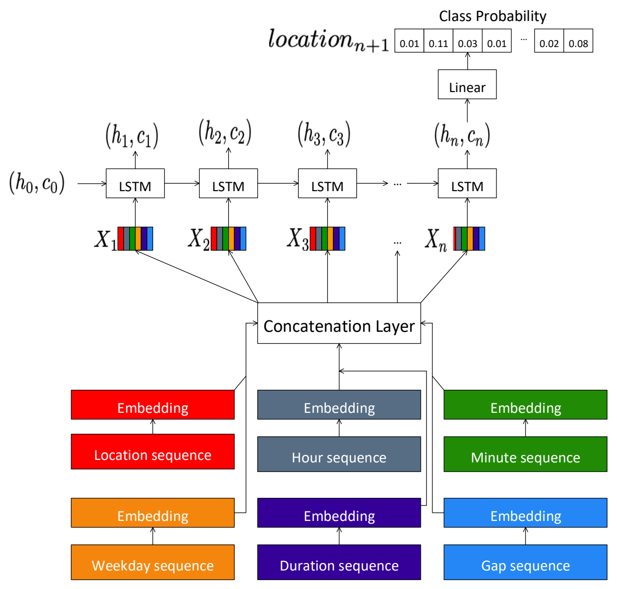


Figure 1: M-RNN architecture.

To leverage RNN, we first convert location ID and relevant time information in trajectory sequence into a vector that is friendly to neural networks. We borrow the idea of word embedding [17, 18], and convert each location ID into an embedding with the embedding lookup table. Five time-related features are extracted and converted to discrete values here. In specific, *Hour*, *Minute* and *Weekday* features represent the likelihood of seeing similar mobility patterns in the temporal domain. Using timestamps of consecutive data points that contain the same location ID, we extract *Duration*, which represents how long a user stays at one location. We also extract *Gap*, the time gap between two adjacent but different locations, to characterize *e.g.,* commute time. During inference, we take *Gap* as an input to predict future locations at any given time. Second, we transform the extracted information above into discrete *id* features. The numbers of discrete ID for *Hour*, *Minute* and *Weekday* are 24, 60 and 7, respectively. *Duration* and *Gap* are discretized into 144 values, with each one covering a time span of 10 minutes, and 144 in total covering 1440 minutes (*i.e.,* one day). It is the upper bound of the time span of a sequence we consider in location prediction. Finally, we convert the trajectory sequence into 6 discrete ID sequences, each representing the *Location sequence*, *Hour sequence*, *Minute sequence*, *Weekday sequence*, *Duration sequence* and *Gap sequence*.

After extraction, we get six sequences of features in total (location sequence plus five time-related sequences). We use location ID sequence in a 10-minutes time window (like a sentence in natural language processing) to pre-train the embedding model to convert the location ID into a dense vector. Similarly, at each timestamp, we train other embeddings end-to-end, and concatenate the embeddings of the six different *ids* as the input for the M-RNN model. We use X_n to represent the embedding concatenation at the n_{th} step.

The M-RNN model contains an LSTM layer. We feed the output hidden of the last step in the sequence to a fully connected layer f and then map feature vector into a $|V|$-dimensional vector, where $|V|$ is the number of location IDs. In LSTM, h_0 and c_0 are the initial hidden state and cell state, and

$$h_n, c_n = LSTM(X_n, (h_{n-1}, c_{n-1})), \tag{1}$$

which denotes the hidden state and cell state at n_{th} step. In the fully connected layer f, we use the $location_{n+1} = f(h_n)$ to represent the future location at the $n+1$ step. Since devices have different visit locations, in Colla, each device customizes its local model by setting an adaptive value of V (§ 3.3). For example, a device sets the size of its fully-connected layer to 25 when it has visited 25 unique locations. If it collects 5 new location IDs, it will expand the size to 30.

In training, we split a sequence which has n elements into $n-1$ training sequences. For instance, $[id_1, id_2, id_3]$ would be split into $[id_1]$->$[id_2]$ and $[id_1, id_2]$->$[id_3]$. Finally, softmax operation is performed to get output probability.

3 COLLA LEARNING FRAMEWORK

In this section, we describe the collaborative learning framework. It consists of cloud and a set of end devices such as smartphones. We illustrate the learning process using a star topology as an example where each device connects directly to the cloud. In Colla, cloud trains and maintains a large base model M whereas each device holds a small client model m customized for itself. We call the base model on the cloud and the client model on a device *cloud model* and *client model*, respectively.

3.1 Learning Flow

As shown in Figure 2, Colla learning process consists of the following four stages.

Stage 1: Bootstrapping. The cloud trains the very first cloud model from an initial dataset. This is done only once when the system is deployed, *e.g.*, when a keyboard application releases a new version

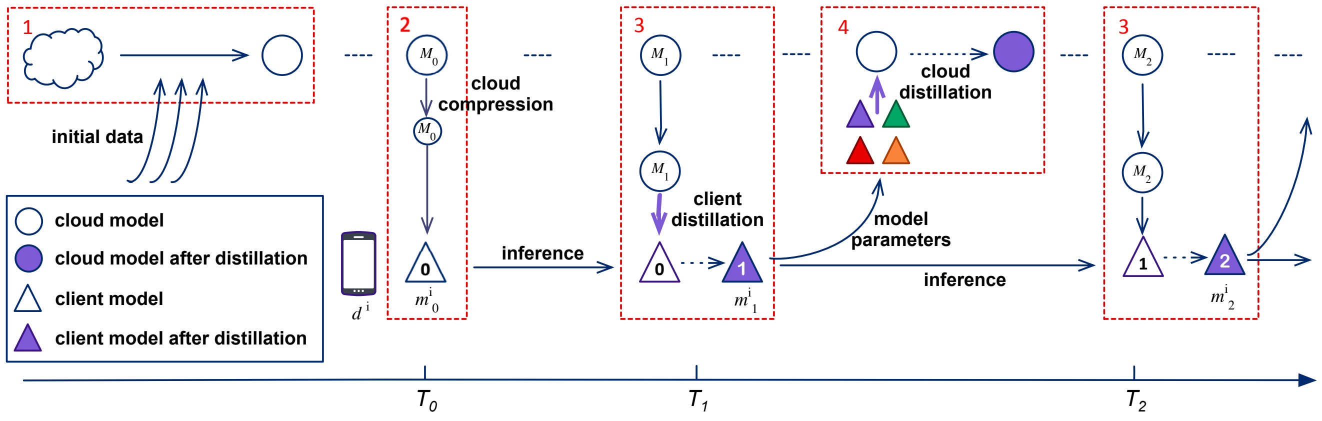


Figure 2: Colla learning flow. Number in dashed red box represents learning stage. Model grouping not included.

that can learn a model to predict next word given the prefix or previous word [19]. At this stage, it is reasonable to expect that the cloud owns some data (*e.g.*, anonymous user data or existing large public dataset like word corpus) to initiate the learning process.

Stage 2: First pulling. When a new device d^i joins the system at time T_0, it asks the cloud for the latest model. The cloud compresses (see § 3.2 for more details) its latest cloud model M_0 into a small one and sends it to the device. d^i uses the compressed small model as its first client model m_0^i to perform inference in the coming time period $T = T_1 - T_0$.

Stage 3: Client model update. After collecting a reasonable amount of data or simply after a fixed time period T, device d^i pulls the latest cloud model M_1 and merge it with the current client model m_0^i through knowledge distillation, resulting in a new client model m_1^i. In the simplest form of distillation, the heavy cloud model is used as the teacher model to fine-tune the client model (*i.e.,* the student model) over the local dataset (*a.k.a.,* the transfer set). Thus, client model is able to learn from both its local data and the cloud model. § 3.3 will describe our knowledge distillation process in detail. In § 5.4 and § 7, we also evaluate the cost of model training on commodity smartphones, and list works that can further expedite DNN execution on resource-constrained devices. After client model m_1^i being generated, its parameters are pushed to the cloud while the device uses the lightweight client model m_1^i to perform inference till time T_2.

Stage 4: Cloud model update. Once receiving model parameters from N of devices, cloud also updates its model. This is done again by knowledge distillation, but with multiple teacher models (*i.e., N* pushed models from end device d^i, $i \in [1, N]$). Here the transfer set includes all data available in the cloud. This stage results in a new cloud model. Besides updating the base cloud model, Colla also performs model grouping, dividing N device models into K groups. Each model group is used to teach the base cloud model into a new classified cloud model (not shown in Figure 2). Details are described in § 3.2.

Client model update (Stage 3) and cloud model update (Stage 4) happen repeatedly. For example, at time T_2, d^i pulls the latest cloud model M_2 (or M_2^k if d^i was classified into group k) and distills the knowledge into client model m_1^i using the data collected during time period $T_2 - T_1$, resulting a new client model m_2^i, and then pushes it to the cloud. Similarly, cloud updates its model from time to time using received models from devices. Since we perform distillation to transfer knowledge from multiple devices to cloud as well as from cloud to each device, we call the mechanism *dual knowledge distillation.* Note that model updates in cloud and device may work in an asynchronous way so as to adapt to various data collection rates and different device constraints (*e.g.*, energy).

3.2 Model Compression and Grouping

To reduce the overhead of running model inference on resource-limited end devices and consider the behavior diversity of different groups of users, Colla uses model compression and model grouping.

Model compression. As end devices usually have limited memory and computation resources, it is critical not to run a large model on end devices. However, the cloud needs to use a large model with enough capacity to combine the knowledge learned from all devices. We propose to use model compression to balance the needs of both the cloud and devices. To this end, the cloud compresses its large base model into a small one through knowledge distillation over the cloud dataset. As shown in Stage 2 in Figure 2, when a new device comes, the cloud sends the compressed small model to the device. The large model and the small model pair have different sizes of LSTM units and different embedding sizes. Specifically, for the large model, we set the size of LSTM unit to 128, and the embedding size for all six different ID spaces to 32. In total, the large M-RNN model contains 128, 796 parameters with a size of 503KB (128, 796 $*$ 4 Bytes). For the small model, we set the hidden size of LSTM unit to 16 and the embedding size to 4, resulting in a model with 11, 868 parameters and a size of 88KB (11, 868 $*$ 4 Bytes). In § 5.4, we show the impacts of different sizes of embedding and LSTM unit on prediction accuracy as well as execution cost on mobile devices.

Model grouping. With growing number of users in Colla, maintaining a single model in cloud may not achieve a satisfied prediction accuracy for all users as different groups of users may have very different behavior patterns. To solve this problem, instead of using a one-fits-all cloud model, Colla splits up the cloud model over time to serve different groups of users. Doing so requires dividing devices into different groups where users in the same group share similar behavior pattern. To this end, Colla conducts model grouping in Stage 4 in Figure 2. After receiving N small client models, cloud performs model inference using each of the N models over the cloud dataset. Based on N output feature vectors, we use the cross entropy between two feature vectors as the distance of two models, and cluster models into K groups using the Affinity Propagation [20]. In our current implementation, we take a brute force approach and choose K with the highest Silhouette Coefficient [21] from a set of {2, 3, 5, 10}.

Afterwards, cloud uses client models in each group to generate cloud model M^k, $k \in [1, K]$ per group, through cloud distillation (see § 3.3) using the cloud dataset. This happens at every cloud model update stage (*i.e.,* Stage 4 in Figure 2) and generates a new set of groups each time. The cloud records which client model (and thus the corresponding device) belongs to which group. As a result, when a device asks for the latest cloud model again (Stage 3 in Figure 2), the cloud will send back the classified cloud model of the device rather than the base cloud model. Note that the base cloud model is also updated using all the received client models and thus new devices may always get the latest cloud model to start with, before it goes to a group.

3.3 Model Update through Dual Distillation

Client and cloud model update are both critical in Colla which transfer knowledge between cloud and different client models. Knowledge distillation [15, 16] is widely used in machine learning to transfer the knowledge from a heavy model (*a.k.a.,* teacher model) to a cheap model (*a.k.a.,* student model) that is more suitable for deployment on edge devices [16, 22–26]. Taking classification problem as an example, for the training process without knowledge distillation, the model generates class probabilities using softmax function, and then matches them to one-hot ground truth labels to calculate loss for back propagation. The major difference after adding knowledge distillation is that, the student model not only matches the output probabilities p of the input x to the true label y, but also matches p to the class probabilities q predicted by the teacher model on the same input x. Mathematically, distillation loss is calculated as:

$$L = \lambda L(y, p) + (1 - \lambda) L(q, p), \tag{2}$$

where $L(*, *)$ is the loss function (*i.e.,* Cross-Entropy or Kullback Leibler (KL) Divergence), $L(y, p)$ is the original loss term and $L(q, p)$ is the distillation loss term, λ is a hyper-parameter to trade off the contribution of the two loss terms. It can be determined by hyper-parameter search during training.

With the popularity of distillation, much of the work [27, 28] has been proposed to allow two models teach each other. For instance, Deep Mutual Learning [27] trains two models on the same dataset simultaneously and makes them match the probability estimates of each other. BAN [28], on the other hand, enables one network to teach itself and generate new models via consecutive distillation. It also adopts ensemble learning to aggregates predictions of these models to generate a more reliable teacher model. Inspired by these works, we extend basic knowledge distillation to a dual distillation paradigm both in client and cloud. When cloud model teaches the client model, devices can utilize the general knowledge from cloud model to avoid overfitting despite the variations of local data. On the contrary, cloud model benefits from the ensemble learning by aggregating outputs from multiple client model.

Client Distillation. On the device side, knowledge is transferred from cloud model to client model. Therefore, we use cloud model as teacher model and conduct knowledge distillation on the data available on each device. Follow Equation 2, the loss function of client model i can be calculated as

$$L_i^d = \lambda_d L(y_i^d, p_i^d) + (1 - \lambda_d) L(q^c, p_i^d), \tag{3}$$

where superscript c and d denote the cloud side and device side. y_i^d represents the label of the data on device i, p_i^d and q^c represent the output probabilities of the client model (student) and cloud model (teacher) on device i, respectively. We use early stopping in client distillation to avoid over-fitting on limited local data. Here we assume local data is annotated (*i.e.,* with known labels). This might be impractical for tasks like object segmentation, but is trivial in many applications like sequence prediction, where word inputs, device locations, app launches are natural labels.

Cloud Distillation. Cloud distillation uses multiple client models to teach the cloud model by fine-tuning it on the cloud dataset. It aims to match the output probabilities of the cloud model to the average of the softmax output of each client model. Similarly, we have

$$L^c = \lambda_c L(y^c, p^c) + (1 - \lambda_c) L(\frac{1}{N} \sum_{i \in N} q_i^d, p^c), \tag{4}$$

where the superscript c and d denote the cloud side and device side respectively. y^c represents hard labels of the cloud data, p^c and q_i^d represent the class probabilities of the cloud data by the cloud model and client model i, respectively. This cloud distillation procedure is the same for both the base cloud model and the classified group models. The only difference is that the number of client models used in the distillation is different.

Note that in Colla, devices have different values of V (*i.e.,* customized FC layers) to cover diverse data classes on the client side. Usually, client models have a smaller label size $|V|$ than that of the cloud model. In order to facilitate knowledge distillation that requires the same label set between teacher and student model, we modify the output probabilities of teacher model. Specifically, in client distillation, only the output probabilities that matches to existing labels of the client model are picked and re-normalized as soft labels. In cloud distillation, the output probabilities of the device model are padded with zeros to match the feature vector size of the cloud model.

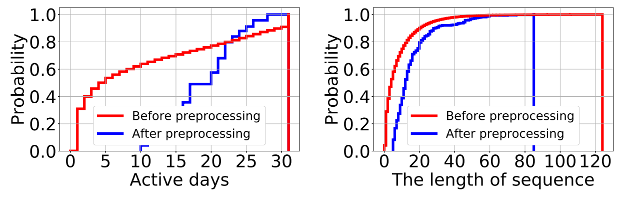
4 DATASET AND IMPLEMENTATION

To empirically evaluate Colla, we fully implement them on Microsoft Azure cloud and Android smartphones, and conduct trace-driven emulations using large-scale real data.

4.1 Dataset Description

Location prediction is conducted using WiFi AP scanning results collected on a large university campus for four months (from March to June in 2016). In total, there are 120, 624, 600 WiFi scanning records between 201, 041 devices and 2, 890 Cisco enterprise APs deployed in 116 buildings on the campus. At peak time, there are ∼20, 000 devices concurrently connected to the campus WLAN. The total number of unique devices is more than 60, 000 each day. On average, 4.89 APs are scanned by each device per day.

Based on the hypothesis that APs close to each other are more likely to be observed by the same mobile device, we group all 2, 890 APs into a smaller number of clusters ($|V| = 368$). Colla aims to figure out which AP cluster each device is more likely to visit given a future time t. To extract devices' trajectory sequences, we slice scanning records using a 10-minute time window. Device location is set to the cluster ID that contains the most visited AP in each time window. In data preprocessing, we filter out inactive devices that have less than 10 *active days* per month – a day is active only if it contains more than 5 unique location IDs. As a result, there are 12540 devices left. The distributions of the number of active days per month and the length of location sequences (*i.e.,* number of different locations) across all devices are shown in Figure 3.



(a) Distributions of the number of active days per month (b) Distributions of the length of location sequences

Figure 3: Location prediction data distribution.

4.2 Implementation

We implement the cloud part of Colla on Microsoft Azure cloud. In specific, we use an Azure virtual machine with Intel Xeon CPU@2.60GHz, 128GB memory and 3TB storage, 8 NVIDIA Tesla P100 GPU cards (CUDA 9.0.0) with 16GB GPU memory for experiments. The cloud part has three main components: data storage, model training engine and communication interface. Model structure is stored in a .json file and the weights are stored in a .npy file (numpy array). Initial training data is stored in HDFS format using Azure HDInsight. Model training is conducted using Keras (v1.0.0) and Theano (v1.0.0). RESTful API is provided for the communications between cloud and devices.

On the client side, we use Android smartphones and PyDroid 3[2], an IDE for Android featuring offline Python (v3.6) interpreter and built-in C and C++ compiler. We install Keras, numpy, scipy and Theano on Pydroid 3 using pip to enable model training with data stored in SD card. Due to the lack of programming model for mobile GPUs[3], we train the M-RNN model on smartphones using CPU. Execution cost is presented in § 5.4.

In terms of training configurations, we use cross entropy as loss function both for the true loss and distillation loss. Both λ_c and λ_d in Equation 4 and Equation 3 are set to 0.5 after a hyperparameter search. Batch size is set to 32 and we use Adam optimizer with a learning rate of 0.1. We adopt cross validation with the ratio of training/validation/test set setting to $16 : 4 : 5$. Early stop mechanism is adopted which terminates training if loss on the validation set doesn't increased in consecutive twenty epochs.

[2]https://play.google.com/store/apps/details?id=ru.iiec.pydroid3

5 EVALUATION

We present evaluation results of Colla in this section, using the dataset described in Section 4.1.

5.1 Experimental Settings

Colla: We use synchronized model update by default. Model update cycle is set to $T = 20$ days. It means all devices pull the cloud model at the end of each cycle, and upload (changed) parameters right after their local models have been updated using local data from the past period. Cloud periodically updates its model based on the model parameters from all clients.

Model: As described in Section 3, Colla adopts different M-RNN architectures in the cloud and on devices. The embedding layer of the heavy M-RNNs are sized at 32 with hidden layer size of 128, whereas the cheap model is sized at 4 with hidden layer size of 16.

Data: We select 10, 000 devices that contain at least 10 active days per month, and evaluate Colla across the entire four months (*i.e.,* March 2016 to June 2016). Particularly, we take a closer look at collaboration performance between 100 most active devices (*a.k.a.,* top-100). Data from the first month is used as initial training set for the cloud. Because we set $T = 20$ days and extract the first month as initial data, we split the data from mobility prediction (April 2016 to June 2016) into four parts. In evaluation, we use the current part as training set and the next part as evaluation set. Thus, there are three cycles in mobility prediction and two cycles in app launch prediction.

Metrics: Trained models are evaluated on the data from the subsequent cycle. We use following metrics to evaluate the performance of Colla:

prediction accuracy: the portion of correct prediction to the total predictions (*i.e.,* top-1 accuracy)

weighted-precision and weighted-recall: $Pr = w_i * \frac{\sum_{i=1}^{|V|} TP_i}{\sum_{i=1}^{|V|}(TP_i+FP_i)}$ and $Re = w_i * \frac{\sum_{i=1}^{|V|} TP_i}{\sum_{i=1}^{|V|}(TP_i+FN_i)}$, which are widely used in multi-class classification [29]. $w_i = \frac{\sum(i)}{\sum_{i=1}^{|V|}(i)}$ is the percentage of i in all data.

execution cost: local model inference time, training time and communication cost of smartphones.

Since there is no well-developed Reception Operating Characteristic (ROC) analysis for multi-class classification or prediction,

[3]Most smartphones feature GPUs from Qualcomm and Intel, which only focus on speeding up inference/prediction but not training. We expect computing platforms like CUDA will be released for smartphone GPU-based training in the near future.

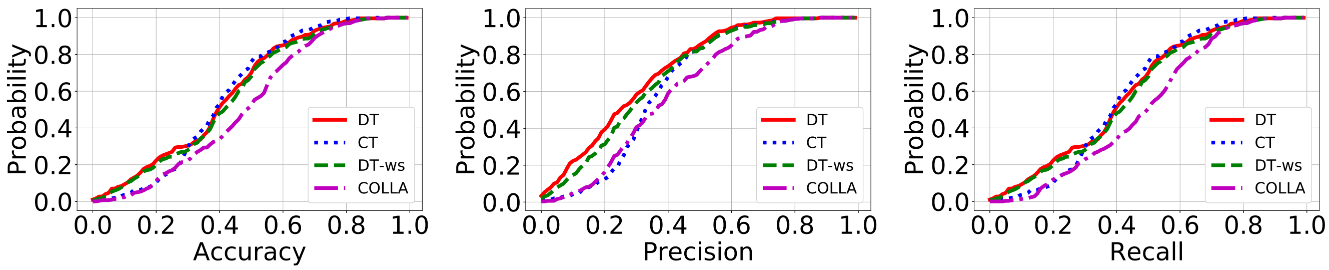


Figure 4: CDF of overall mobility prediction accuracy, precision and recall in three updates.

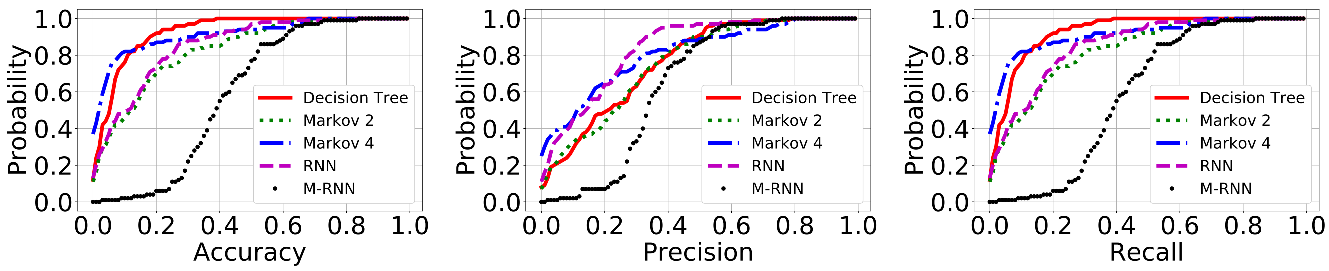


Figure 5: CDF of mobility prediction accuracy, precision and recall with baseline methods.

we do not include AUC (Area under the ROC Curve) in the list of measures.

5.2 End-to-end Prediction Performance

We first evaluate end-to-end performance between four different methods at the end of the evaluation period. These four methods are i) *device-training (DT)*, where each device trains its own model from scratch using its own data and never exchanges any information with cloud nor other devices. ii) *cloud-training (CT)*, where cloud collects data from all devices periodically and trains a global model (using heavy M-RNN) for prediction. i) and ii) are two extremes that are commonly used in legacy prediction approaches where clients are either treated independently or uniformly. iii) *device-training with warm-start (DT-ws)*, where each device pulls the very first trained model from cloud, and utilizes local data to fine-tune it at the beginning of each cycle. However, devices never exchange parameters with cloud nor other devices after the first pulling. This is the default approach to deploy models in transfer learning. iv) *Colla*, where collaborative learning is adopted.

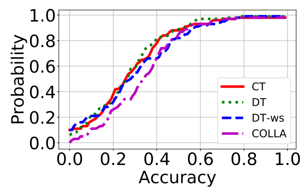
5.2.1 Top-100 devices with fixed size of initial dataset. Figure 4 shows the overall performances in three updates for mobility prediction. We see that CT performs slightly better than both DT and DT-ws, due to the extra data from other devices. In all three figures, Colla performs the best among these methods, demonstrating the effectiveness of collaborative learning. Some detailed accuracy numbers are listed in Table 1. It can be seen that the median accuracy of Colla is 0.51, 21.42% higher than the second best approach (0.42 of DT-ws), and Colla yields the best accuracy among all four methods on 68 devices (out of 100). Another interesting finding is that the performance of DT is nearly equal to DT-ws in Figure 4. Given DT-ws outperforms DT by 10% in terms of median accuracy after the first update, this comparison reveals that the help from warm start diminishes with successive client model updates.

Table 1: Mobility prediction performance. Best percentage means the percentage of the devices where the corresponding method performs the best.

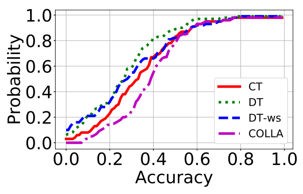
Method	Best pct.	Avg. acc.	Med. acc.
DT	0.0%	0.40	0.40
CT	28.00%	0.40	0.39
DT-ws	4.00%	0.41	0.42
Colla	68.00%	0.47	0.51

Figure 5 also compares Colla with multiple state-of-the-art methods including decision tree [6], Markov Chain model [7] and RNN [4] after the first update. As can be seen, Colla with M-RNN achieves a superior performance in all three metrics. For instance, M-RNN exceeds RNN 116% (0.39 vs. 0.18) in terms of median accuracy.

5.2.2 Random sampled devices with varying sizes of initial dataset. In a more realistic setting, we sample 100 devices randomly from all 100,00 devices, and extend the cloud initial dataset to top-10^4 devices.



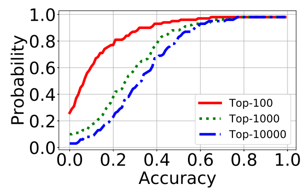
(a) Initial dataset with 10^3 devices



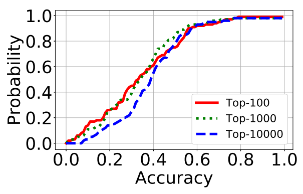
(b) Initial dataset with 10^4 devices

Figure 6: CDF of prediction accuracy on the random-100 devices in mobility prediction.

In Figure 6, we can see that model fine-tuning-based methods (*i.e.,* DT, DT-ws and COLLA) obtain better performance when the initial dataset is small. However, with a large initial dataset in the cloud, CT catches up and outperforms DT and DT-ws. In the best case, it achieves a median accuracy of 0.33 when the initial training set contains 10^4 devices. Note that the gain of CT comes at the cost of continuous local data uploading and heavy model exchange. Execution cost of local inference using a heavy model could also be prohibitively expensive (more experimental results in Section 5.4). Compared with CT and DT-ws, COLLA obtains the best prediction accuracy on different initial datasets. In addition, between these two figures, we can see it also benefits from the improvement of the pre-trained cloud model, bringing median accuracy from 0.32 to 0.39.



(a) CT



(b) COLLA

Figure 7: CDF of prediction accuracy using different methods in mobility prediction.

Figure 7 shows another perspective of Figure 6 by putting together models trained in the same way but with different initial data sizes. It is clearer that the larger data size we used for CT, the better generalization capability the model can achieve, hence the higher prediction accuracy.

5.3 Model Update Performance

In this section, we zoom in to examine gains from each model update.

Figure 8 shows mobility prediction performance from top-100 devices over four months (three updates). Compared with steady improvements of COLLA, DT-ws can hardly boost devices' prediction capabilities over time. Since DT-ws purely relies on each device's own data, over-fittings are more likely to happen. However, in COLLA, devices take advantages of the cloud model during subsequent local training, which can be seen as regularization to prevent over-fitting from happening.

We found the continuous improvement from each model update is largely brought by model grouping. From Figure 9(a), it can be seen that without grouping, although prediction results get better over time, the gain becomes marginal after three months. We also investigate the performance of the Affinity Propagation with Silhouette Coefficient to determine the number of groups K (algorithm in Section 3.2). To this end, during cloud model update, we cluster models into different numbers of groups K, and examine their end-to-end prediction performance. For the top-100 devices, we find clustering into two groups achieves the best overall accuracy (Figure 9(b)) after the second update, matching with the rank of Silhouette Coefficient (inside legend box in Figure 9(b)). It verifies the effectiveness of grouping using model inference results.

Another component that has impacts on local model update is the customization of FC layer. Here we compare Customized FC, where each device dynamically expands the size of FC layer, against Unified FC, where each device owns identical model architecture (cheap model with a same size of the FC layer), on top-100 devices in the first update. In Figure 10(a), we can see that Customized FC strictly outperforms Unified FC in terms of accuracy. This can be explained by the location class distribution of each user across cycles - new locations are more likely to be seen in the first few cycles (Figure 10(b)). Therefore, the gain from having a smaller FC layer, thus a higher prediction confidence, outweighs prediction errors from missing location classes in next cycle.

5.4 Execution Cost

COLLA obtains aforementioned accuracy improvements at the cost of periodical on-device model training and cloud model update. Due to orders of magnitude less computing power and energy budget, we focus on execution cost on smartphones. To obtain a comprehensive understanding of runtime cost, we compare three model update approaches. They are i) Same-Whole, where devices share the same architectures (*i.e.,* cheap M-RNN) and fine-tune the whole network during update; ii) Same-FC, where devices share the same architectures but only fine-tune the last fully connected layer; iii) Customization, where devices gradually expand (and fine-tune) the last FC layer with growing local observations. All other layers are frozen during training. We conducted experiments on multiple smartphones at different levels. Results from four M-RNNs with different sizes (*i.e.,* 4-16, 8-32, 16-64, 32-128) are presented, where, 4-16, for instance, means the embedding size is 4 and the hidden size is 16. Sizes of these models are shown in Figure 11(a), where $Customization(n)$ denotes the $n-th$ local update in customization update approach.

We firstly examine the number of parameters to evaluate the communication overhead. For down-link, all three model update approaches send the same cloud model to the client side. However, for the client-to-cloud communication, Same-FC and Customization only need to upload the parameters of the fully connected layer while Same-Whole needs to upload all parameters, since only modified model parameters are required for cloud model update. Figure 11(b) shows the communication overhead from top-100 devices. In our current implementation with client model sized

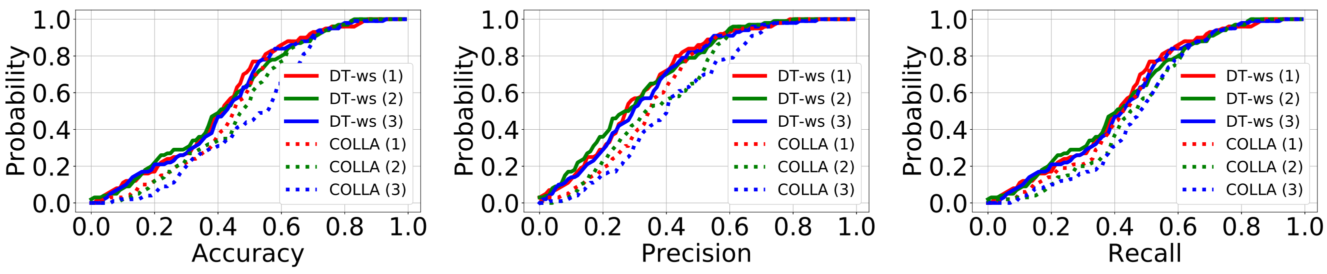
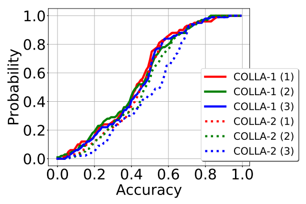
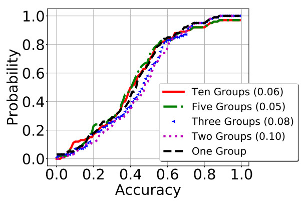


Figure 8: CDF of top-100 devices' performances during three updates in mobility prediction.

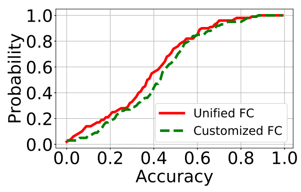


(a)

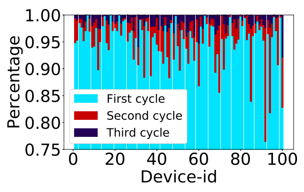


(b)

Figure 9: The accuracy without grouping (COLLA-1) and with grouping (COLLA-2) during three updates (Figure a). The accuracy and corresponding Silhouette Coefficient score (inside legend box) with different size of groups after the second update (Figure b).



(a) Prediction accuracy CDF



(b) Class distribution across cycles

Figure 10: Performance of local model customization.

to 4-16, communication cost of one model update is as low as $520KB(130000 * 4byte)$.

Another concern of running Colla on smartphones is the cost of local inference and client model training. As inference time can be affected by many factors (*i.e.,* different optimizations on matrix multiplication), we use FLOPS to measure the inference cost. In Figure 11(c), compared with Same-FC and Same-Whole (they have the same FLOPS due to the same model architecture), Customization nearly halved inference computation. Due to the smallest numbers of uploaded parameters, inference's FLOPS and model size, FC customization is proved to be the most efficient local model update approach in iterative training.

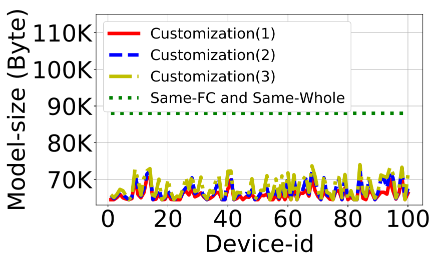
Next, we dig deep to measure model training/inference time as well as energy consumption. We randomly sampled a user from the top-100 list, and use the total 293 data samples from the first cycle (*i.e.,* 20 days) to fine-tune the local model, whereas 224 samples from the subsequent cycle are used for testing. Note that this user is at 23% percentile of all users in terms of the size of local samples. Figure 12(a) shows local model update time of four M-RNNs on five smartphones. We find that customization has the least training time among three updating approaches in all cases - it only takes 2.42 minutes on average to fine-tune the local model on a commodity smartphone, which is order of magnitude smaller than fine-tuning the whole model. Similar results are observed in Figure 12(b), where one prediction execution only needs 133.5 milliseconds. This makes it a feasible solution to turn location prediction as a component to third party applications on end devices. In terms of different model sizes, interestingly, we find model 8-32 needs the maximum time to fine-tune on all five phones. This is because although 8-32 ranks third in size, it demands much more training epochs to converge (as shown in Figure 13). We also used a Monsoon Power Monitor as a power supply for the smartphone, and tracks both runtime current and voltage to calculate energy consumption (Table 2). In summary, although training a large CNN model (e.g., ResNet-152) is still prohibitively expensive for smartphones, we find the cost of training a lightweight but effective RNN model is very viable in terms of both time, energy and network consumption. For applications like mobility prediction, one-hour training on a commodity smartphone can handle data collected over one year, at the cost of 2000 mWh energy consumption and few hundreds KB communication overhead. Note that training can be further accelerated using GPU, FPGA, and it can also be scheduled to run at night when charger plugged in.

Table 2: Time (s) and energy consumption (mWh).

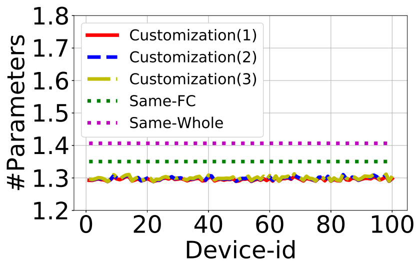
Model arch.	Huawei Mate 9 Pro		Huawei Mate 10	
	Time	Energy	Time	Energy
4 - 16	492.6	276.2	216.6	127.3

6 DISCUSSION

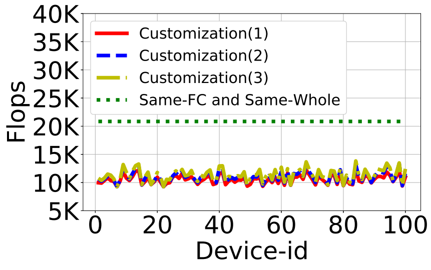
Despite promising results yielded by collaborative learning on mobility prediction, there are several practical issues and limitations of this study that warrant further investigation.



(a) Model size

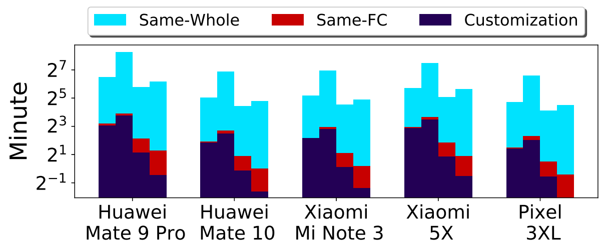


(b) Number of parameters (*1e5)

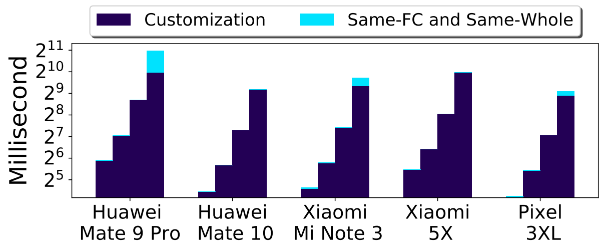


(c) Inference FLOPS

Figure 11: Runtime cost of Colla.



(a) Fine-tuning time



(b) Inference time. Note that Same-FC and Same-Whole are strictly higher than Customization but their values are very close in many cases.

Figure 12: Fine-tuning and inference time. Each bar represents one M-RNN architecture.

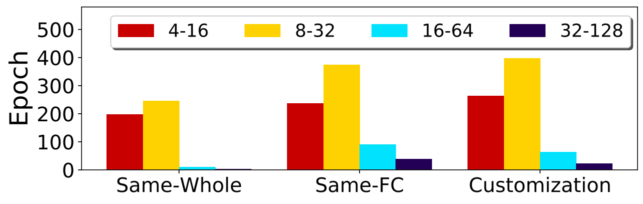


Figure 13: Fine-tuning epochs.

Beyond location prediction. Colla can naturally be adapted to user behavior predictions other than location changes. Taking app launch prediction as another example, we can replace trajectory sequence used in M-RNN (§ 2.2) with app launch sequence and predict the next app to be launched using features like time, previous app launch instances, battery level, CPU usage that might have strong correlations with app launch behaviors.

On an app launch prediction dataset collected from 27 volunteers using SherLock smartphone agent between January and March in 2016 [30], Colla achieved comparable results with Table 1, where a 31.8% improvement of median prediction accuracy is achieved by Colla over CT. The evaluation used same settings in Section 5.2 and we constructed app launch sequence from the resource utilization traces sampled every five seconds.

Table 3: App launch prediction performance.

Method	Best pct.	Avg. acc.	Med. acc.
DT	22.2%	0.52	0.48
CT	22.2%	0.40	0.44
DT-ws	0.0%	0.47	0.46
Colla	55.6%	0.55	0.58

Learning process. Open questions on learning process also remain. First, in our design, learning starts from a fixed amount of initial data. There are still insights to be gained on the quality (*e.g.*, diversity) of initial data that would be needed for a confident general model. It is also interesting to explore the gains from extra *data* uploading after bootstrapping. For instance, each new-coming device could push certain amount of data for one time, or some edge devices may be willing to upload data constantly but at a very low frequency. Besides, the fixed architectures of client model may be not suitable for incremental settings. When the capacity of client model cannot handle the change of all new data, Colla needs to tune the architecture of client model.

Selectively using device models for cloud model update could not only brings cloud model a better generalization ability, but also prevent cloud model from being corrupted by adversarial input. As shown in Figure 14, our preliminary results from filtering out 30% device-uploaded models with poor inference accuracy on the cloud dataset shows an end-to-end median accuracy increase of 3.46%. Extending Colla to CNNs is also an interesting subject to pursue.

With growing concern about data privacy, it is also worth investigating how Colla can be designed in a privacy-preserving

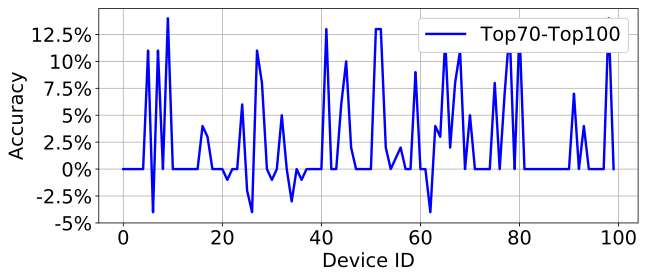


Figure 14: Location prediction accuracy gap between using Top-70 and Top-100.

way. This is challenging given that prior works [31] have shown that even the most innocuous aggregate, including the parameters of ML models, can reveal information about the training sets.

7 RELATED WORK

Distributed machine learning has been a hot topic in the machine learning community due to the emerging big data and big models. Model average [32–35] is a simple yet efficient technique which iteratively averages the locally trained models and performs parameter synchronization. Allreduce-based communication [36] is also used for the gradient communication in the data parallelism setting both for the single-machine multi-GPUs and multi-machines multi-GPUs. A more flexible architecture for distributed machine learning is parameter server [37–39], which uses dedicated servers to synchronize model parameters and implement other computation logic related to the optimization algorithms. However, most previous work on distributed machine learning treat each working node as a computing machine, receiving *i.i.d.* data and model parameters, generating gradients or updated parameters for synchronization. Endpoints in this work is not simply a computing device, but also consuming trained model by itself with local generated *non i.i.d.* training data. More recently, federate learning [19, 40, 41] attempts to leverages local collected data to train a global model. However, it fails to take into device diversity into consideration and no model customization is allowed.

Knowledge distillation was first proposed to transfer the knowledge from a cumbersome model (or ensemble of models) to a single small model more suitable for deployment [15]. It has been used since then in a wide range of tasks such as image classification [16], neural machine translation [22, 23] and speech recognition [16, 24]. The typical setting of knowledge distillation transfers knowledge from a teacher model to a student model, while there have been studies transferring knowledge cross all the models in a collaborative way, in order to boost task execution performance of all participants [25, 26]. Unlike CoDistillation [26] that uses the same dataset to train all the models, or Mutual Learning [25] that advocates an ensemble of students to learn collaboratively and teach each other, dual distillation in Colla features a completely different network where N different client models distill knowledge from their own data and share it between each other with the help of the cloud.

Researchers have been exploring various approaches to enable deep learning on mobile and edge devices that have limited computing power and energy budget. Those efforts including building smaller models without sacrificing too much accuracy [42–44], leveraging or building customized hardware for fast learning [45–48], model compression to reduce resource consumption [49–51], or system optimization to achieve a better resource-accuracy tradeoff [52–54]. For example, DeepEar [42] proposes a special model for audio sensing on smartphones in unconstrained acoustic environments. DianNao [45] designs a dedicated ASIC to accelerate ubiquitous machine learning. NestDNN [55] designs a dynamic framework to choose the most suitable model when the resource of application is changed. DeepX [49] leverages Runtime Layer Compression (RLC) and Deep Architecture Decomposition (DAD) to reduce resource usage. These learning practices on mobile edges all focus on individual devices and thus are complementary to our collaborative learning.

User behavior prediction has been studied for decades. It was indicated that the potential average predictability in, for example, human mobility can be as high as 93% [9, 56, 57]. Various methods have been proposed to profile user behaviors and make predictions on whereabouts, including Markov models [7, 8, 58–60], neural networks [12, 61], Bayesian networks [62], random forest [63], eigendecomposition [64] *etc.* Markov model, as well as its variations, model the probability of future movements by building a transition matrix between several locations based on past trajectories. Given its success in speech and NLP, RNN is also proposed for mobility prediction. For instance, Spatial Temporal Recurrent Neural Networks (ST-RNN) is designed to model temporal and spatial contexts [4]. Nevertheless, it only applies to continuous spatial prediction, and assumes distances between location points are known. More recently, DeepSense, a unified deep learning framework for mobile sensing data is proposed by integrating convolutional and recurrent neural network [65]. However, it focuses on accommodating diverse sensor noise patterns with a model trained remotely from uniform sampling data. Different from existing work, we demonstrate the advantages of applying a more general learning framework that combines intelligence from both the cloud and edges on location prediction problem.

8 CONCLUSION

We propose Colla, a learning framework designed for user behavior prediction by enabling end devices and a cloud to learn from sequence data in a collaborative way. A cloud-client collaboration mechanism is carefully designed to make the learning approach flexible and scalable. In particular, we propose a novel dual distillation method with model compression and model grouping to empower the cloud to aggregate the knowledge learned from end devices into a global cloud model and enable each device to distill knowledge from the cloud model and build a customized client model. We demonstrate the feasibility of the framework on mobility prediction using a multi-feature RNN. Experimental results on Azure and commodity smartphones with large-scale real data show that Colla establishes effective device models in terms of both prediction accuracy and execution cost.

REFERENCES

[1] P. G. Om Prakash and Dr. A. Jaya. Analyzing and Predicting User Behavior Pattern from Weblogs. *International Journal of Applied Engineering Research*, 11(0973-4562):62786283, 2016.

[2] Zhe Zeng and Matthias Roetting. A text entry interface using smooth pursuit movements and language model. In *ACM Symposium on Eye Tracking Research & Applications*, 2018.

[3] Chang Tan, Qi Liu, Enhong Chen, and Hui Xiong. Prediction for Mobile Application Usage Patterns. In *Nokia MDC Workshop*, 2012.

[4] Qiang Liu, Shu Wu, Liang Wang, and Tieniu Tan. Predicting the Next Location : A Recurrent Model with Spatial and Temporal Contexts. In *AAAI*, 2016.

[5] Joao Bartolo Gomes, Clifton Phua, and Shonali Krishnaswamy. Where Will You Go? Mobile Data Mining for Next Place Prediction. *Lecture Notes*, 8057 LNCS:146–158, 2013.

[6] Ivana Semanjski and Sidharta Gautama. Smart City Mobility Application Gradient Boosting Trees for Mobility Prediction and Analysis Based on Crowdsourced Data. *Sensors*, abs/15974.15987, 2015.

[7] Akinori Asahara, Kishiko Maruyama, Akiko Sato, and Kouichi Seto. Pedestrian-movement Prediction Based on Mixed Markov-chain Model. In *ACM SIGSPATIAL*, 2011.

[8] Sébastien Gambs, Marc-Olivier Killijian, and Miguel Núñez del Prado Cortez. Next Place Prediction Using Mobility Markov Chains. In *Proceedings of the First Workshop on Measurement, Privacy, and Mobility*, 2012.

[9] C. Song, Z. Qu, N. Blumm, and A.-L. Barabasi. Limits of Predictability in Human Mobility. *Science*, 327(5968):1018–1021, 2010.

[10] Fatma Somaa, Cédric Adjih, Inès El Korbi, and Leila Azouz Saidane. A Bayesian model for mobility prediction in wireless sensor networks. In *International Conference on Performance Evaluation and Modeling in Wired and Wireless Networks*, 2016.

[11] Nam T. Nguyen Binh T. Nguyen, Nhan V. Nguyen and My Huynh T. Tran. A Potential Approach for Mobility Prediction using GPS Data. In *ICIST*, 2017.

[12] Z Lin, M Yin, S Feygin, M Sheehan, and JF Paiement. Deep Generative Models of Urban Mobility. In *ACM KDD*, 2017.

[13] Qiang Liu, Shu Wu, Liang Wang, and Tieniu Tan. Predicting the Next Location: A Recurrent Model with Spatial and Temporal Contexts. In *AAAI*, 2016.

[14] Yoshua Bengio Jason Yosinski, Jeff Clune and Hod Lipson. How transferable are features in deep neural networks? In *NIPS*, 2014.

[15] Cristian Bucila, Rich Caruana, and Alexandru Niculescu-Mizil. Model compression. In *ACM KDD*, 2006.

[16] Geoffrey Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the Knowledge in a Neural Network. In *NIPS Deep Learning and Representation Learning Workshop*, 2015.

[17] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[18] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv*, 2013.

[19] Jakub Konecný, H. Brendan McMahan, Felix X. Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated Learning: Strategies for Improving Communication Efficiency. *CoRR*, abs/1610.05492, 2016.

[20] Renchu Guan, Xiaohu Shi, Maurizio Marchese, Chen Yang, and Yanchun Liang. Text Clustering with Seeds Affinity Propagation. *IEEE Transactions on Knowledge and Data Engineering*, 23(4):627–637, 2011.

[21] Peter J. ROUSSEEUW. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 87(21):0377–0427, 1986.

[22] Yoon Kim and Alexander M. Rush. Sequence-Level Knowledge Distillation. In *EMNLP*, 2016.

[23] Markus Freitag, Yaser Al-Onaizan, and Baskaran Sankaran. Ensemble distillation for neural machine translation. *CoRR*, abs/1702.01802, 2017.

[24] Liang Lu, Michelle Guo, and Steve Renals. Knowledge distillation for small-footprint highway networks. In *ICASSP*, pages 4820–4824. IEEE, 2017.

[25] Ying Zhang, Tao Xiang, Timothy M. Hospedales, and Huchuan Lu. Deep Mutual Learning. In *IEEE CVPR*, 2017.

[26] Rohan Anil, Gabriel Pereyra, Alexandre Passos, Robert Ormándi, George E. Dahl, and Geoffrey E. Hinton. Large scale distributed neural network training through online distillation. *CoRR*, abs/1804.03235, 2018.

[27] Ying Zhang, Tao Xiang, Timothy M. Hospedales, and Huchuan Lu. Deep Mutual Learning. In *Conference on Computer Vision and Pattern Recognition. CVPR-2018*, 2018.

[28] Tommaso Furlanello, Zachary C. Lipton, Michael Tschannen, Laurent Itti, and Anima Anandkumar. Born-Again Neural Networks. In *Thirty-fifth International Conference on Machine Learning. ICML-2018*, 2018.

[29] Marina Sokolova and Guy Lapalme. A Systematic Analysis of Performance Measures for Classification Tasks. *Inf. Process. Manage.*, 45(4):427–437, July 2009.

[30] Yisroel Mirsky, Asaf Shabtia, Lior Rokach, Bracha Shapira, and Yuval Elovici. SherLock vs Moriarty: A Smartphone Dataset for Cybersecurity Research. In *AISec with CCS*, 2016.

[31] Reza Shokri, Marco Stronati, and Vitaly Shmatikov. Membership Inference Attacks against Machine Learning Models. In *IEEE Symposium on Security and Privacy*, 2017.

[32] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized Stochastic Gradient Descent. In *NIPS*, 2010.

[33] Ryan McDonald, Keith Hall, and Gideon Mann. Distributed Training Strategies for the Structured Perceptron. In *ACL*, 2010.

[34] Daniel Povey, Xiaohui Zhang, and Sanjeev Khudanpur. Parallel Training of Deep Neural Networks with Natural Gradient and Parameter Averaging. *arXiv*, 2014.

[35] Sixin Zhang, Anna E Choromanska, and Yann LeCun. Deep Learning with Elastic Averaging SGD. In *NIPS*, 2015.

[36] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv*, 2017.

[37] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large Scale Distributed Deep Networks. In *NIPS*, 2012.

[38] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning with the Parameter Server. In *USENIX OSDI*, 2014.

[39] Eric P Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A New Platform for Distributed Machine Learning on Big Data. *IEEE Transactions on Big Data*, 1(2):49–67, 2015.

[40] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, 2017.

[41] Jakub Konecný, Brendan McMahan, and Daniel Ramage. Federated Optimization: Distributed Optimization Beyond the Datacenter. *CoRR*, abs/1511.03575, 2015.

[42] Nicholas D. Lane, Petko Georgiev, and Lorena Qendro. DeepEar: Robust Smartphone Audio Sensing in Unconstrained Acoustic Environments Using Deep Learning. In *ACM UbiComp*, 2015.

[43] Guoguo Chen, Carolina Parada, and Georg Heigold. Small-footprint Keyword Spotting Using Deep Neural Networks. In *IEEE ICASSP*, 2014.

[44] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv*, 2017.

[45] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: a Small-footprint High-throughput Accelerator for Ubiquitous Machine-Learning. In *ACM ASPLOS*, 2014.

[46] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *ACM/SIGDA FPGA*, 2015.

[47] Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *ACM/IEEE ISCA*, 2016.

[48] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *ACM/IEEE ISCA*, 2016.

[49] Nicholas D. Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. DeepX: A Software Accelerator for Low-power Deep Learning Inference on Mobile Devices. In *ACM/IEEE IPSN*, 2016.

[50] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized Convolutional Neural Networks for Mobile Devices. In *IEEE CVPR*, 2016.

[51] Emily L. Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation. In *NIPS*, 2014.

[52] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: Video analytics at scale via adaptive configurations and cross-camera correlations. In *ACM SIGCOMM*, 2018.

[53] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints. In *ACM MobiSys*, 2016.

[54] Ganesh Ananthanarayanan, Victor Bahl, Landon Cox, Alex Crown, Shadi Nogbahi, and Yuanchao Shu. Demo: Video Analytics - Killer App for Edge Computing. In *ACM MobiSys*, 2019.

[55] Biyi Fang, Xiao Zeng, and Mi Zhang. NestDNN: Resource-Aware Multi-Tenant On-Device Deep Learning for Continuous Mobile Vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking. ACM, 2018*, 2018.

[56] Xin Lu, Erik Wetter, Nita Bharti, Andrew J. Tatem, and Linus Bengtsson. Approaching the Limit of Predictability in Human Mobility. *Scientific Reports*, 3:1–9, 2013.

[57] Xin Lu, Linus Bengtsson, and Petter Holme. Predictability of Population Displacement after the 2010 Haiti Earthquake. *Proceedings of the National Academy of Sciences*, 109(29):11576–11581, 2012.

[58] George Liu and Gerald Maguire. A Class of Mobile Motion Prediction Algorithms for Wireless Mobile Computing and Communications. *Mobile Networks and Applications*, 1(2):113–121, 1996.

[59] Yu Zheng, Quannan Li, Yukun Chen, Xing Xie, and Wei-Ying Ma. Understanding Mobility Based on GPS Data. In *ACM UbiComp*, 2008.

[60] L. Song, D. Kotz, R. Jain, and Xiaoning He. Evaluating Location Predictors with Extensive Wi-Fi Mobility Data. In *IEEE INFOCOM*, 2004.

[61] Shiang-Chun Liou and Hsuan-Chia Lu. Applied Neural Network for Location Prediction and Resources Reservation Scheme in Wireless Networks. In *International Conference on Communication Technology Proceedings*, 2003.

[62] Sherif Akoush and Ahmed Sameh. Mobile User Movement Prediction Using Bayesian Learning for Neural Networks. In *International Conference on Wireless Communications and Mobile Computing*, 2007.

[63] Zidong Yang, Ji Hu, Yuanchao Shu, Peng Cheng, Jiming Chen, and Thomas Moscibroda. Mobility Modeling and Prediction in Bike-Sharing Systems. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2016.

[64] Nathan Eagle and Alex Sandy Pentland. Eigenbehaviors: Identifying Structure in Routine. *Behavioral Ecology and Sociobiology*, 63(7):1057–1066, 2009.

[65] Shuochao Yao, Shaohan Hu, Yiran Zhao, Aston Zhang, and Tarek Abdelzaher. DeepSense: A Unified Deep Learning Framework for Time-Series Mobile Sensing Data Processing. In *WWW*, 2017.