DeepAPP: A Deep Reinforcement Learning Framework for Mobile Application Usage Prediction

Zhihao Shen[*]
Xi'an Jiaotong University
Xi'an, China
szh1095738849@stu.xjtu.edu.cn

Kang Yang[*]
Xi'an Jiaotong University
Xi'an, China
yangkangyk@stu.xjtu.edu.cn

Wan Du
University of California, Merced
Merced, USA
wdu3@ucmerced.edu

Xi Zhao
Xi'an Jiaotong University
Xi'an, China
Zhaoxi1@mail.xjtu.edu.cn

Jianhua Zou
Xi'an Jiaotong University
Xi'an, China
jhzou@sei.xjtu.edu.cn

ABSTRACT

This paper aims to predict the apps a user will open on her mobile device next. Such an information is essential for many smartphone operations, e.g., app pre-loading and content pre-caching, to save mobile energy. However, it is hard to build an explicit model that accurately depicts the affecting factors and their affecting mechanism of time-varying app usage behavior. This paper presents a deep reinforcement learning framework, named as DeepAPP, which learns a model-free predictive neural network from historical app usage data. Meanwhile, an online updating strategy is designed to adapt the predictive network to the time-varying app usage behavior. To transform DeepAPP into a practical deep reinforcement learning system, several challenges are addressed by developing a context representation method for complex contextual environment, a general agent for overcoming data sparsity and a lightweight personalized agent for minimizing the prediction time. Extensive experiments on a large-scale anonymized app usage dataset reveal that DeepAPP provides high accuracy (precision 70.6% and recall of 62.4%) and reduces the prediction time of the state-of-the-art by 6.58×. A field experiment of 29 participants also demonstrates DeepAPP can effectively reduce time of loading apps.

CCS CONCEPTS

• **Human-centered computing** → **Ubiquitous and mobile computing systems and tools**; • **Computing methodologies** → **Reinforcement learning**.

KEYWORDS

Mobile Devices, App Usage Prediction, Deep Reinforcement Learning, Neural Networks

[*]Both student authors contributed equally to this research.

ACM Reference Format:
Zhihao Shen, Kang Yang, Wan Du, Xi Zhao, and Jianhua Zou. 2019. DeepAPP: A Deep Reinforcement Learning Framework for Mobile Application Usage Prediction. In *The 17th ACM Conference on Embedded Networked Sensor Systems (SenSys '19), November 10–13, 2019, New York, NY, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3356250.3360038

1 INTRODUCTION

Predicting the applications (apps) that a mobile user may use in next time slot can provide many benefits on smartphones, such as app pre-loading [1–3], content pre-fetching [4–6] and resource scheduling [7]. For instance, by knowing the apps a user may open in next 5 minutes, we can pre-load the apps in memory slightly in advance and improve user experience with minimized launch time. Traditional app prediction models are not designed for such a time-sensitive prediction, especially people may use multiple apps in the same time slot.

Most existing app prediction works [1, 8–16] normally predict the next app by modeling app usage transitions and exploiting contextual information using Markov model [4, 9, 13, 17] or Bayesian model [11]. They can only provide limited prediction accuracy due to two reasons. 1) conventional model-based methods assume app usages can be well modeled by Markov chain or Bayesian framework. However, app usages are determined by a variety of factors in the complex contextual environment. It is hard to explicitly capture the impact of all potential factors by a statistical model. As a consequence, most existing works [10, 11] only represent the context by a limited number of semantic labels (i.e., "Home", "Work place" and "On the way"). 2) the apps that people will use next have strong temporal-sequence dependency. It is necessary to consider the prediction of next apps successively. However, most existing works [1, 13, 16] predicts the next apps with maximum probabilities separately, hence ignores the effect of current app might bring to the future prediction.

To address the above limitations, we develop a Deep Reinforcement Learning (DRL) framework, named as *DeepAPP*, to learn a data-driven model-free neural network (also known as an agent in DRL), which takes the environment context as input and predicts the apps that will be opened next. We first train a deep neural network (DNN) agent using historical app usage data on a server and then run the trained DNN agent on either the server or each user's phone. The

DNN agent of DeepAPP makes prediction based on a neural network rather than an explicit model; therefore, it can take the complex environment context as input. Additionally, with reinforcement learning, DeepAPP can generate the predicted results according to the expected reward, which is determined by the future interactions between user and apps. To incorporate DRL into DeepAPP, we tackle a set of challenges and develop three novel techniques for app prediction, including a context-aware DRL input representation method, a lightweight agent and an agent enhancement scheme.

To enable more accurate app prediction, DeepAPP leverages more fine-grained representation of the environment context. Besides the time and currently-opened app [1, 13], DeepAPP leverages the distribution of surrounding Point of Interests (POIs) to capture the location features of the user. In addition, based on such a representation, the DNN-based agent can generalize the past experience to new locations. When a user goes to a new place, the DNN model can still make prediction according to similar known places.

In order to provide real-time inference, one essential requirement of app prediction is short inference latency. One successful implementation of DRL is Deep Q-Network (DQN), which has been applied in many applications, like Atari games [18] and mobile Convolutional Neural Network (CNN) model selection [19]. For one inference, DQN searches for the best action from all possible actions. It is efficient for small action spaces (e.g. 2 actions for Breakout in Atari game), but cannot be used for our app prediction due to the large action space. For example, if a user has installed 20 apps, the action space will be enormous ($C_{20}^0 + C_{20}^1 + ... + C_{20}^{20} = 2^{20} = 1,048,576$). DQN takes 2.04 seconds to perform one prediction in our implementation on a 2-core CPU, and it also has a converge problem during training. To handle this problem, we adopt a lightweight actor-critic based agent architecture [20] to avoid the heavy cost of evaluating all possible actions for one inference.

Ideally, we can train a specific DRL model for each individual user based on her own app usage data. However, it is difficult to obtain sufficient training data from each user. Additionally, users may install new apps. It is hard for a trained agent to cover these new apps during online inference. To solve the data sparsity problem, DeepAPP first trains a general agent with the data of all available users (e.g., 443 users in our dataset). We then use the trained agent for app prediction of every user. During online inference, we keep updating the agent to a personalized agent for each user based on her new app usage data. With reinforcement learning, we can update the parameters of the personalized agent incrementally by new data, without re-training the DNN model. At the same time, we also update the general agent periodically (e.g., one day in our implementation) using the data from all users. Once the general agent is updated, we also use it to further update each personalized agent by combining their DNN parameters. As each user has increasingly collected her own data to update her personalized agent, an adaptive coefficient is defined to gradually reduce the weight of the general agent in the update of each personalized agent.

We implement DeepAPP on TensorFlow [21]. We run the personalized agents of all users independently on a server that contains 2 CPUs. Experiment results demonstrate that two CPU cores are enough to make an inference within 0.31 seconds and perform one update of the personalized agent within 3.57 ms. Although such light cost of inference and agent update can be totally supported by current smartphones, we cannot run the TensorFlow version of DeepAPP on smartphones, since TensorFlow currently does not support mobile operating systems. We further implement DeepAPP on TensorFlow Lite [22] to perform inference directly on smartphones.

Table 1: Examples of app usage data.

UserID	Start Time	LAC	CID	AppName	Duration (s)
1B2A7	201805*080234	60*8	3*93	Wechat	5
5U2F1	201805*070821	64*2	2*83	Chrome	32

We first conduct trace-driven validations. Our dataset contains the app usage records of 21 days from 443 users in a big city. We use the 14-day data for training and the rest 7-day data for validation. Cross-validation tests are conducted. We train the general agent by the training data of all users, and update the personalized agent incrementally for each user using her testing data. The experiment results demonstrate that DeepAPP provides precision and recall of 70.6% and 62.4% respectively, corresponding to a performance gain of 8.62% and 15.56% over the state-of-the-art solution [16]. DeepAPP also provides a 6.58× inference time reduction compared with the DQN-based model.

We also recruit 29 volunteers and conduct field experiments over 55 days by a customized app. The experiment results reveal that DeepAPP provides precision and recall of 73.2% and 54.1% respectively in app prediction. With app pre-loading, DeepAPP can reduce the app loading time by 68.14% on average. More than 85% of the participants are satisfied with our app prediction system.

In summary, this paper makes following contributions.

- To the best of our knowledge, we are the first to leverage DRL in app prediction.
- We customize our DRL framework by considering unique challenges in app prediction, including a context representation method, a lightweight personalized agent and an agent enhancement technique by the data of all available users.
- We conduct extensive evaluations based on a large-scale app usage dataset and field experiments.

2 MOTIVATION

In this section, we first investigate the necessity for app prediction through questionnaires. We then introduce the data used in this work for app prediction system. Finally, we briefly introduce the key concepts of deep reinforcement learning.

2.1 Need for app prediction

We designed and released a questionnaire on a widely used online questionnaire survey platform, called WJX [23]. Questions are mainly about the necessity and urgency of an app prediction system. After 32-day collection, 238 enrolled participants returned their feedback. We filtered out invalid feedbacks and eventually we got 206 questionnaires. The participants include 65 females and 141 males, aged from 13 to 65. They have various occupations, such as company employees, civil servants, medical staff, college teachers, students, etc. The survey results indicate an urgent request for accurate app prediction. We have the following detailed analysis of our collected feedback.

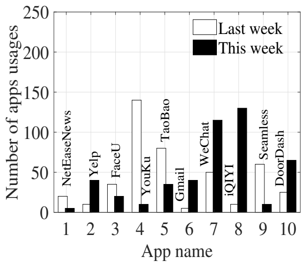


Figure 1: The number of app usages of different apps used by a user in two weeks.

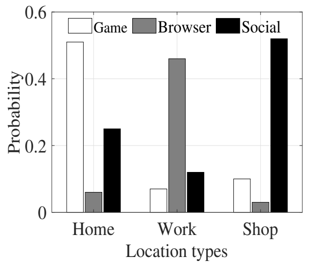


Figure 2: The relationship between app usages and environment context.

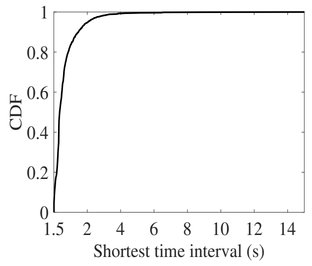


Figure 3: CDF of the shortest time interval between the transitions of app usages of users.

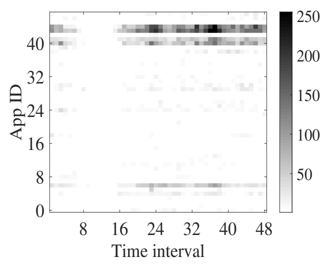


Figure 4: The distribution of the number of app usage records of different apps in time intervals.

- 76.63 % of them thought it takes a long time from clicking on an application icon to start using the application;
- 90.77 % of them are willing to use a software that can reduce waiting time of application loading.

2.2 Cellular data

In this paper, we use an anonymized cellular dataset collected by a mobile carrier of a city in China. The dataset contains 2,104,369 app usage records of 443 mobile users in 21 days. It covers 36,039 unique applications and 5,156 cell towers. When a user requests a network service from a mobile app, the request is sent to the corresponding server via cellular infrastructure. An app usage trace records the request information observed by the corresponding cell tower. Table 1 describes the format of our data record, which is composed of a set of fields, i.e., Anonymized ID (UserID), Start Time, LAC (Location Area Code), CID (Cell Tower ID), App ID and Duration. The duration denotes the time that the user has used a specific app. It is estimated by the start and end time of the record. Based on LAC and CID, we know that the user is within the coverage of the cell tower with which her smartphone is associated. By processing our data, we found the following observations.

Time-varying app usage preference. Figure 1 depicts the number of app usages of different apps that one user uses in two weeks. In the first week, she used MeiTuan a lot for online food ordering; whereas in the next week she turned to DianPing, another top online food purchase platform, maybe because DianPing provides more discount in that period. As a result, due to the time-variation of user preference on different apps, the DNN agent needs to be updated continuously. We leverage the reinforcement learning to solve the above problem by learning the app usage preference incrementally.

Context-related app usage. The environment context has an important impact on the apps that people use. We use our dataset to investigate the relation between app usages and the environment context where people use the apps on smartphones. From Figure 2, we can observe that people tend to use different apps in different environment context. We leverage the POI distributions nearby the location of the app usage to represent the environment context.

Real-time app prediction. It is critical to provide the real-time app prediction for users. If a user switches frequently to different apps in a short time, the DNN agent needs to update its predicted result before launching next apps. Figure 3 depicts the distribution of the shortest time interval between the transitions of different app usage data of users in a day. As shown, 94.7 % had made short-time switches less than 2 seconds. Therefore, the DNN agent is required to have the low time complexity, and thus we propose a lightweight actor-critic based personalized agent to reduce the prediction time.

Sparse app usage data. Adequate app usage data is also a key issue to achieve good prediction. However, it is difficult to obtain a large number of app usages for each single user. For new users, we even do not have any app usages from her. Figure 4 depicts the gray value distributions of the number of app usages of different apps of a user in time intervals in a week. The result reveals that app usages are scattered over the time intervals. If we always predict those apps with higher frequency, this sometimes affects the performance. We maintain a general agent to learn the general app usage behavior of all users for personalized prediction based on the historical app usages and continuously-collected app usages of all available users.

2.3 Deep reinforcement learning

Deep reinforcement learning (DRL) is a promising machine learning approach, which instructs an agent to accomplish a task by trail and error in the process of interacting with the environment. Four key elements are defined to describe the learning process of DRL, i.e., state, action, policy and reward.

The state $\boldsymbol{s}$ defines the input of an agent, referring to the environment representation. Different applications define different states. In app prediction, we define the state as the user's contextual information, including her current app, surrounding environment, and time.

The policy π is the core of the agent, which takes the state as input to generate an action. It learns a mapping from every possible state to an action according to the past experience. In DRL, the policy is implemented as a deep neural network (DNN).

The action $\boldsymbol{a}$ affects the environment. Every action gets a feedback from the environment. According to the feedback, we calculate a reward $r(\boldsymbol{s}, \boldsymbol{a})$, which indicates how good or bad an action $\boldsymbol{a}$ changes the environment given a specific state $\boldsymbol{s}$. Based on the reward, a value function $Q(\boldsymbol{s}, \boldsymbol{a})$ is defined to update the policy of the agent. The Q value reflects the long-term effect of an action, e.g., if an action has a high Q value, the parameters of the DNN agent will be updated to favor that action. As shown in Eq. 1, $Q(\boldsymbol{s}, \boldsymbol{a})$ is the long-term reward that an agent expects to obtain in the future, where r_t is the reward of step t, and λ is the discount factor.

$$Q(\boldsymbol{s}, \boldsymbol{a}) = E[\sum_{t=0}^{\infty} \lambda^t r_t | \boldsymbol{s}_0] \tag{1}$$

Table 2: Notations used in this paper.

Notation	Description
$\mathcal{S}$ $\boldsymbol{s}$	State space, state
$\mathcal{A}$ $\boldsymbol{a}$	Action space, action
A_u	The set of apps on a user's smartphone
r	Reward
$\boldsymbol{\theta}_\mu$	Parameters of actor network
$\boldsymbol{\theta}_Q$	Parameters of critic network
$\boldsymbol{B}$	Replay buffer
$\hat{\boldsymbol{a}}$	Proto-action
K	Number of nearest neighbors of $\hat{\boldsymbol{a}}$
$\boldsymbol{x}$	App feature
$\boldsymbol{l}$	Context feature
$\boldsymbol{t}$	Time feature
k	Prediction epoch
ω	The length of time slot
p	The decrease rate of the balance coefficient

Based on the above elements, the agent can learn to accomplish a specific task by training an agent with a specific policy, supposing we have enough transition samples $(\boldsymbol{s}_t, \boldsymbol{a}_t, r_t, \boldsymbol{s}_{t+1})$. The agent first perceives a state $\boldsymbol{s}$ and generates an action $\boldsymbol{a}$ by running the policy π. Then, the agent obtains a reward r given by the environment and updates the policy based on the estimate of $Q(\boldsymbol{s}, \boldsymbol{a})$. In this way, the agent and the environment interact with each other to modify the policy. After several iterations, the agent learns a stable policy. In addition, after each online inference, the agent can also use the above training process to update the policy of the DNN agent incrementally based on the new user data.

3 DESIGN OF DEEPAPP

In this section, we introduce an overview of DeepAPP and three key techniques developed in DeepAPP. Table 2 presents the notations frequently used in this study.

3.1 Overview

DeepAPP predicts the apps that will be opened by the user in the next time slot (5 minutes in our current implementation). We perform prediction at the start of each time slot or at the moment when the user closes an app (i.e. prediction epoch). Figure 5 depicts the architecture of our app prediction system, which consists of a back-end component and a front-end component.

3.1.1 The front-end component. It is implemented on smartphones, including two main modules, i.e., a context-sensing module and a background scheduler. The context-sensing module collects the context information (i.e. , currently-using app, location and time) and sends it to the back-end component. Based on the computation on the back-end component, the predicted result is transmitted back to the front-end component. The background scheduler performs a scheduling strategy to pre-load the predicted apps slightly before the next time slot.

3.1.2 The back-end component. It runs on a server and performs the training and inference of our DNN agents. It also updates the DNN agents online. The back-end component mainly consists of five modules as follows.

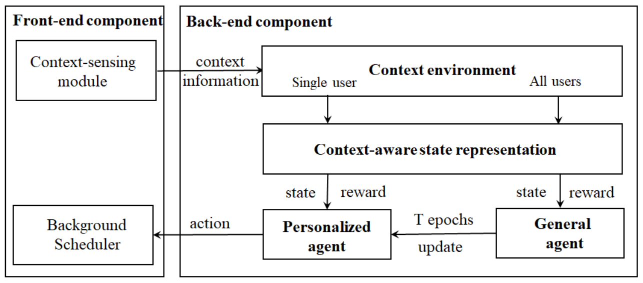


Figure 5: The architecture of app prediction system.

Context-aware state representation. To accurately describe a user's environment context in DeepAPP, we customize the context-aware state by a combined vector that consists of three key features, including app feature, context feature and time feature (see details in Section 3.2).

General agent. In DRL, the agent is used to interact with the environment. The state of the environment at one moment is represented by the above environment context. The objective of an agent is to learn an optimal policy to select an action given a specific state. Since we do not have sufficient app usage data for each user, we first train a general agent using the app usage data of all users.

Action space. Based on the perceived state, an agent predicts which apps a user will open in the next time slot. In particular, the action is denoted as a 0-1 vector $\boldsymbol{a}$, where $\boldsymbol{a}_i = 1$ indicates that app i will be opened in the next time slot. For a general agent, the action space $\mathcal{A}$ contains all feasible apps of all users. An actor-critic based agent is built to perform real-time inference in a large action space (see details in Section 3.3).

Reward function. For each action, a reward r is calculated to evaluate the prediction performance. Based on the reward, the agent updates its policy for better prediction by modifying its DNN parameters. As shown in Eq. 2, we define the reward function as the ratio between the number of correctly-predicted apps in the next time slot N_r (obtained from user feedback) and the number of predicted apps N_p (obtained from predicted result). If the number of predicted apps is 0 and the user does not use any apps in that time slot, we set the reward to 1. If the number of predicted apps is 0 or all predicted apps do not use in the predicted time slot, we set the reward to -5.

$$r = \begin{cases} 1, & N_r = 0 \wedge N_p = 0 \\ N_r/N_p, & N_r \neq 0 \wedge N_p \neq 0 \\ -5, & N_r = 0 \vee N_p = 0 \end{cases} \tag{2}$$

Personalized agent. During the online inference, we keep updating the general agent to a personalized agent for each user according to the real-time app usage data.

3.1.3 Two-step work flow. Based on the above customized modules, DeepAPP works in two steps, i.e., the offline training and the online inference. During the offline training, we train a general agent with enough app usage transition samples of all available users. During online inference, the personalized agent is step-wise updated by optimizing the DNN parameters based on personal app usages to adapt to the time-varying app usage preference. In order to learn app usage behaviors of new apps, the general agent is also updated by app usages of all available users. The updated general agent is further used to update the personalized agent periodically by a diminishing balance coefficient (see details in Section 3.4).

3.2 Context-aware state representation

At each prediction epoch k, DeepAPP quantifies the context-aware state as a combined vector to represent current environment context (i.e. currently-using app, location and time) of a specific user. Specifically, the state is measured as $s_k = [(x_k, l_k, t_k)]$, which consists of three key elements: the app feature x_k, the context feature $\boldsymbol{l}_k$ and the time feature $\boldsymbol{t}_k$.

App feature. To maintain the same dimension of input state, we construct the app feature by calculating transition times from one certain app to other apps. For a certain app i installed on the smartphone of a user, we denote the app feature of an app i at the prediction epoch k as $\boldsymbol{x}_k^i = [x_k^{i1}, x_k^{i2}, ..., x_k^{in}]$, where each $\boldsymbol{x}_k^{ij}$ is the normalized number of transition times of app i transits to app j.

Point of Interest. We adopt the POI information close to a certain location to represent the context of that area. In geographic information system, a POI can be a building, a shop, a scenic spot and so on. We crawled all the POIs of the city from AMap [24] (one of a leading online map provider), which provides APIs to find POIs on the map. All POIs are stored in the server-side database. We build indexes for fast query of POIs around a certain location. In all, our POI dataset contains over 300,000 POIs. They are classified into 23 main types, including restaurants, shopping, sports, business, etc.

Context feature. We calculate the context feature by the distribution of POIs. For a certain location i of the user, we denote the location at the prediction epoch k as a feature $\boldsymbol{l}_k^i = [l_k^{i1}, l_k^{i2}, ..., l_k^{im}]$ for m types of POI (23 in our implementation, which corresponds to the number of category of POI). Each $\boldsymbol{l}_k^{ij}$ is the number of POI category j within the radius of 500 meters. We also normalize the context feature $\boldsymbol{l}_k^i$ to represent the location at the prediction epoch k.

For training and data-driven validation, we use our cellular data and quantify the context feature by the POIs around the cell tower with which the user's phone is associated. For online inference, we obtain the user's location via her smartphone and take the POIs around her location into account. By doing so, we do not need the data or any support from mobile carriers when DeepAPP is running.

Time feature. We construct the time feature as a one-hot feature $\boldsymbol{t}_k$ with the dimension of $24 * 60/\omega$, where ω is the length of each time slot (unit in minutes). It is an effective way to discretize time information. We set the time slot of current app usage to 1, and other time slots are set to 0.

In our design, new features can be easily added to present the contextual information of users in more details, such as GPS locations [13], smartphone status [12] and Wi-Fi information [25, 26].

3.3 Actor-critic agent for app prediction

Deep Q-network (DQN) [18] has been proven to be effective for the design of policy of the agent in cases of the complex environment. However, the DQN-based method suffers from the high time complexity problem in the task with large action space [20]. It cannot be used for online app prediction, since users may switch between apps frequently less than 2 seconds (see the observations in Section 2.

Recently, some advanced techniques, such as Deterministic Policy Gradient (DPG) [27] and Deep Deterministic Policy Gradient (DDPG) [28], have been proposed to operate efficiently on the continuous space. They directly learn the mapping between the state space and the action space, and hence avoid to evaluate a large number of actions. Inspired by the above recent progresses in reinforcement learning, we propose an actor-critic based agent architecture for DeepAPP [20, 28].

Figure 6 depicts the design of our proposed actor-critic based architecture for the policy of both personalized and general agents. The basic idea is to allow the generalization over action space. We only need to evaluate a few actions that are close to the optimal action. By reducing the evaluation times, we minimize the computation time of one inference in DeepAPP. Specifically, the framework includes four main components, i.e., a continuous space, an actor network, a discretizer and a critic network. We first develop the continuous space which expands from the integer action space. Then, we leverage the actor network to output the predicted result (proto-action $\hat{\boldsymbol{a}}$) in the continuous space, which may not be in the original action space $\mathcal{A}$. Next, the predicted result is passed to the discretizer to find the most likely actions $\mathcal{A}_K$ in the action space, which are the actions close to the proto-action $\hat{\boldsymbol{a}}$. Finally, we adopt the critic network to select the action $\boldsymbol{a}$ with highest Q value in $\mathcal{A}_K$.

Continuous space. The conventional action space is defined by a binary vector, in which all bits are '0' except one '1', referring to as the specific app that people will be opened in the next time slot or not. The continuous space is a relaxed version of action space, which achieves generalization over actions. It maps similar actions into a close adjacent space. We then can find an approximate solution and evaluate adjacent actions around it to obtain the optimal predicted result. Specifically, we expand the action space to a continuous space, which is defined in the real field rather than the integer field. Each item in the vector can be a real number between 0 and 1.

Actor network μ^{θ}. We design the actor network as $\mu^{\theta}(s)$ that maps from the context-aware state space $\mathcal{S}$ to the action space $\mathcal{A}$, where μ^{θ} is the mapping function defined by parameters $\boldsymbol{\theta}_{\mu}$. Given the perceived state $\boldsymbol{s}$ of the environment, this actor network directly outputs an approximate predicted result, denoted as proto-action $\hat{\boldsymbol{a}}$. By evaluating results around $\hat{\boldsymbol{a}}$, we can avoid to search in the whole action space, and thus reduce the prediction time. However, proto-action $\hat{\boldsymbol{a}}$ may not be in the action space $\mathcal{A}$. Therefore, we use a discretizer to map from $\hat{\boldsymbol{a}}$ to an action $\boldsymbol{a} \in \mathcal{A}$.

Discretizer. Normally, the actions with lower Q values may occasionally fall near the proto-action $\hat{\boldsymbol{a}}$, which cause errors in the predicted result. Additionally, some actions close in the action space may have different long-term Q values. In the context of these circumstances, it is not advisable to simply select the closest action to $\hat{\boldsymbol{a}}$ as the final result. To avoid selecting an outlier action, we develop a discretizer, which maps from the continuous space to a set of adjacent actions $\mathcal{A}_K$. As shown in Eq. 3, we enumerate all actions in $\mathcal{A}$ to find K actions $\mathcal{A}_K$ that are close to the proto-action $\hat{\boldsymbol{a}}$.

$$\begin{aligned} & min_{\boldsymbol{a} \in \mathcal{A}} ||\boldsymbol{a} - \hat{\boldsymbol{a}}||_2 \\ & s.t. : a_i \in \{0, 1\}, \qquad a_i \in A_u \end{aligned} \tag{3}$$

With Eq. 3, we still have to go through all actions in $\mathcal{A}$. Although the number of calculations are the same as DQN, the time complexity of each calculation in Eq. 3 (i.e., addition) is much lower than that of DQN (i.e., $|\mathcal{A}|$ evaluations of the neural network).

Critic network Q^{θ}. We finally adopt a critic network $Q^{\theta}(\boldsymbol{s}, \boldsymbol{a})$ to find the action in $\mathcal{A}_K$ with the maximum Q as our result. Compared with the DQN-based method that evaluates all actions in $\mathcal{A}$ to find a proper action, we only evaluate a few actions in $\mathcal{A}_K$. Specifically, the critic network takes each action $\boldsymbol{a}$ in $\mathcal{A}_K$ and the state as input to find the action with the largest value function $Q(\boldsymbol{s}, \boldsymbol{a})$ as the final prediction result, as presented in Eq. 4 .

$$Q(\boldsymbol{s}, \boldsymbol{a}) = argmax_{\boldsymbol{a} \in \mathcal{A}_K} Q(\boldsymbol{s}, \boldsymbol{a}; \boldsymbol{\theta}_Q) \tag{4}$$

3.4 Online updating of the agents

Due to the data sparsity problem, we train a general agent with the app usage data of all users. We then use the general agent to perform app prediction for each individual user and gradually update it to a personalized agent using the personal app usage data of each user. As app usage data are collected from all available users, we also update the general agent by newly-collected data periodically, and further use the periodically-updated general agent to enhance the personalized agent.

3.4.1 Offline training of the general agent. During the offline training, DeepAPP trains a unified general agent with the app usage data of all users and uses it for online inference of each user. To ensure that the general agent can be used for the personalized agent, we maintain the same structure for these two agents. Their DNN networks have the same network topology with the same input and output. For input, we transform the feature of the contextual environment into a fixed length vector to represent the state. Regarding with output, the length of the output dimension is the same for all users' models, i.e., the action space is composed of all the possible apps of all users.

3.4.2 Online update of the general agent. During online inference, we update the general agent at regular time intervals (e.g., one day in our implementation). The update normally has two steps, i.e., the update of critic network and the update of actor network. First, according to the reward of current step, the agent optimizes the loss function L to update the critic network. The loss function L is defined as Eq. 5:

$$L = \frac{1}{N}\sum_{i=1}^{N}(Q_{tgt} - Q(\boldsymbol{s}_i, \boldsymbol{a}_i|\boldsymbol{\theta}_Q))^2 \tag{5}$$

where N is the number of app usage transitions from all users and the frozen Q value Q_{tgt} is learned by the target networks [18] as shown in Eq. 6.

$$Q_{tgt} = r_i + \lambda Q(s_{i+1}, \mu'(s_{i+1}|\boldsymbol{\theta}_{\mu'})|\boldsymbol{\theta}_Q) \tag{6}$$

where λ is the discount factor. With back propagation, the critic network can be easily updated according to the gradient of loss function $\nabla_{\theta_Q} L$.

We further update the actor network of the agent using Eq. 7:

$$\nabla_{\theta_\mu} J \approx \frac{1}{N}\sum_{i=1}^{N} \nabla_a Q(s, a|\boldsymbol{\theta}_Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta_\mu} \mu(s|\boldsymbol{\theta}_\mu)|s_i \tag{7}$$

where $\mu(s|\boldsymbol{\theta}_\mu)$ is the K predicted results of the actor network and $Q(\boldsymbol{s}, \boldsymbol{a}|\boldsymbol{\theta}_Q)$ is the evaluations of the K actions calculated by the critic network with Eq. 4. The gradient in Eq. 7 is calculated by the derivative rules of compound functions to optimize the parameters of actor network.

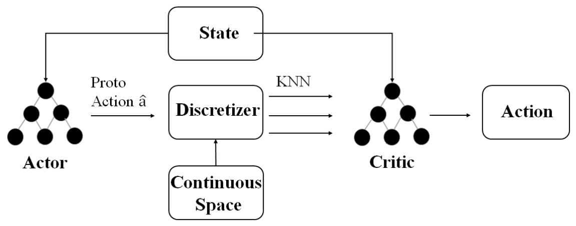


Figure 6: The architecture of the actor-critic agent.

3.4.3 Online update of the personalized agent. The personalized agent has the same updating algorithm as the general agent, as specified in Eqs. 5 - 7. The only difference is the updating frequency. We update the personalized agent more frequently at the start of each time slot (5 minutes in our current implementation) or when the user closes one app. High updating frequency makes the personalized agent rapidly adapt to the time-varying app usage preference.

3.4.4 Combination of the personalized agent and the general agent. We combine the parameters of the actor network $\boldsymbol{\theta}_\mu$ in the general agent and the parameters of the actor network $\boldsymbol{\theta}_{\mu_g}$ in the personalized agent as Eq. 8.

$$\boldsymbol{\theta}_\mu = \boldsymbol{\theta}_\mu + \eta \boldsymbol{\theta}_{\mu_g} \tag{8}$$

where a balance coefficient η is adopted to adjust the importance between the general agent and the personalized agent. As time goes on, we could obtain more individual app usage data and make good prediction based on the personalized agent. If we still keep a large weight of the general agent, the general app usage behaviors learned by the general agent may overwhelm the personal app usage behavior of the personalized agent. Therefore, with the increase of prediction epochs, we decrease η linearly with a decrease rate of p.

3.5 Work flow of DeepAPP

Based on the above designs, we present the work flow of DeepAPP, which mainly consists of three steps: initialization, offline training and online inference.

Initialization. At first, we randomly initialize the parameters $\boldsymbol{\theta}_\mu$ of the actor network μ^{θ} and the parameters $\boldsymbol{\theta}_Q$ of the critic network Q^{θ}, which denoted as main networks. If we directly evaluate the actor and critic networks to obtain the Q_{tgt} value in Eq. 5, the training process will be unstable [20]. To solve this problem, we create the copies of actor and critic networks (μ'^{θ} and Q'^{θ}) [18], which denoted as target networks. The target networks have the same initial parameters with the main networks. The target networks are updated slower (every 100 prediction epochs in our implementation) than those of the main networks. The parameters of the updated target networks will be used to calibrate the main networks during online stage later. We initialize a replay buffer $\boldsymbol{B}$ [18] to break the correlation between the app usage sequence.

Offline training. We leverage the offline app usage data from all available users in the replay buffer $\boldsymbol{B}$ to train networks of the general agent. We first convert all users' app usage data into transition samples (<$\boldsymbol{s}_t, \boldsymbol{a}_t, r_t, \boldsymbol{s}_{t+1}$>) and store them into the replay buffer $\boldsymbol{B}$. By randomly sample N app usage transitions from $\boldsymbol{B}$, we optimize the network parameters as defined in Eq. 5 and Eq. 7.

Online inference. During online inference, based on the trained general agent, the personalized agent performs prediction and updates its policy for better prediction. The parameters of the personalized agent is initialized with the parameters of the general agent.

At each prediction epoch k, DeepAPP first senses the context-aware state from the front-end component as the input of the actor network of personalized agent and derives a proto-action $\hat{\boldsymbol{a}}$by the actor network $\mu^{\boldsymbol{\theta}}$. In order to explore potential better actions, we also introduce a stochastic exploring mechanism by adding random noise into the action. Specifically, we add a random noise $\epsilon \boldsymbol{I}$ to the proto-action $\hat{a}$ [28], which has the similar idea as ϵ-greedy [29]. As ϵ decreases with the prediction epoch, more certain action will be taken with more training. $\boldsymbol{I}$ is a homotypic vector with action, which follows the standard uniform distribution ($U(0, 1)$).

Then, we find K nearest actions of the proto-action $\hat{\boldsymbol{a}}$by solving Eq. 3. The possible actions are passed to the critic network for evaluating the Q-value of each action. The action with highest Q-value is selected and passed to background controller at the front-end component of a specific user. According to the feedback from the user, the personalized agent calculates the reward to update the actor and critic networks. In order to limit the updating speed of the target networks, we adopt soft update technique [18] to stabilize the parameters of the target networks.

$$\begin{aligned} \boldsymbol{\theta}_{Q'} &:= \tau\boldsymbol{\theta}_Q + (1-\tau)\boldsymbol{\theta}_Q \\ \boldsymbol{\theta}_{\mu'} &:= \tau\boldsymbol{\theta}_{\boldsymbol{\mu}} + (1-\tau)\boldsymbol{\theta}_\mu \end{aligned} \tag{9}$$

Finally, at every combination cycle, we update the general agent and then combine it with the personalized agent as Eq. 8.

4 IMPLEMENTATION

In this section, we introduce implementation details of the back-end component and the front-end component respectively.

4.1 Back-end component

At the back-end side, DeepAPP trains a unified general agent for all users, and performs inference and update of the personalized agent for each user. All agents are implemented on TensorFlow [21] and share with the same network structure. They use a 2-layer fully-connected feedforward neural network to serve as the actor network, which has 1000 and 400 neurons in the first and second layer and use a 2-layer fully-connected neural network, with 400 and 200 neurons in the first and second layer for the critic network. To alleviate the over-fitting problem, we introduce an L_2 regularization term in the loss function [30]. Besides, there are a few hyper-parameters to set in both networks. We conducted a comprehensive empirical study to find best settings, as shown in Table 3.

The back-end component is implemented on a server, which contains 2 CPUs. Both CPUs have dual Intel(R) Xeon(R) CPU E5-2609 v4 @ 1.70GHz with 8 cores. Experiments demonstrate that a 2-core CPU is enough to support to make an inference and perform an update within 0.31 seconds and 3.57 ms respectively. We also use a GPU cluster with 2 nodes (12GB memory) to accelerate the offline training of the general agent, which can save 2.47 × training time compared with the training on CPUs.

Table 3: Hyper-parameter setting.

Hyper-parameter	Setting
Batch size N	32
Number of the offline training iterations	500,000
Future reward discount γ	0.90
The size of replay buffer $\boldsymbol{B}$	100,000
Learning rate of actor network $\nabla\mu$	0.0001
Learning rate of critic network ∇Q	0.001
Soft updating coefficient τ	0.01

4.2 Front-end component

The front-end is implemented as a customized app on smartphones (our current implementation runs on Android 9.0). The implementation of the app includes two modules, i.e., a context-sensing module and a background scheduler.

Context-sensing module. We use the context-sensing module to obtain the real-time context information of users (e.g. user feedback, location, time and currently-using app) for the next time prediction. Note that all sensitive information are acquired on a voluntary basis. Specifically, we obtain the location information through *Location-Manager* provided in Android SDK, and the current foreground app through *AccessibilityEventEvents* by accessibility services and smartphone status through Android logcat.

Background scheduler. We only pre-load apps that may be used in the next time slot to minimize the energy and memory cost for pre-loading apps on smartphones. To do so, we develop a background scheduler, which pre-loads the apps before next time slot according to the predicted result. In particular, we use *getLaunchIntentForPackage* in Android *PackageManager* to realize the pre-loading.

4.3 Data transmission

The data transmitted between the back-end component and front-end component are small in size. The data transmission can be supported either by WiFi or cellular networks. First, we need to transmit the context-sensing result from the front-end component to the back-end component. In each iteration of prediction, we only need to send about 480 bytes on average, including user feedback, location, time and currently-using app. The transmission delay is less than 30 ms on average if cellular networks are used. Since WiFi is faster than cellular networks, the transmission delay can be further reduced if WiFi networks are available. At the same time, we need to transmit the predicted result from the back-end component to the front-end component. The predicted result is composed of a string of app IDs. The information can be encapsulated in one packet within 120 bytes. The transmission delay is 25 ms on average.

4.4 Inference on smartphones

DeepAPP can also make inference on the smartphone of each user, without the need of a back-end component. This can improve the scalability of DeepAPP. We use TensorFlow Lite [22] as a solution to run DeepAPP on smartphones. TensorFlow Lite is a widely-used developing tool to deploy machine learning models on mobile devices with low latency and memory cost. We first use all users' app usage data to train a DNN agent. We export the DNN agent to a *tf.GraphDef* file. It ensures that the agent model can communicate with our DeepAPP app. Finally, we integrate the DNN agent into our DeepAPP app. Our Android DeepAPP app is written in Java,

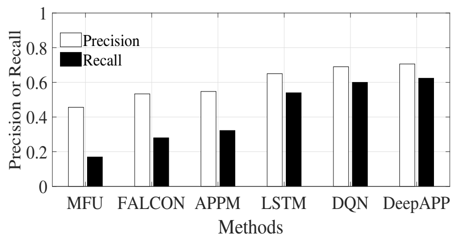


Figure 7: Prediction accuracy.

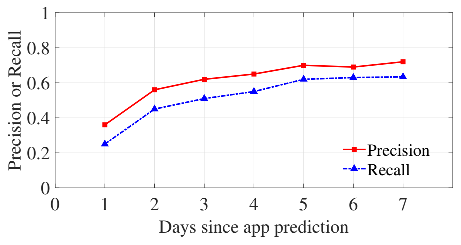


Figure 8: Evolution of prediction accuracy over time.

but TensorFlow Lite is implemented in C++. We use a JNI library provided by TensorFlow Lite to set up I/O interfaces.

Since TensorFlow Lite currently does not support training operation on mobile devices, we only run DeepAPP for inference, but do not update the personalized agent on smartphones. If a user chooses to run DeepAPP locally, she will not need to transmit her data to the back-end component. Her personalized agent will only be updated every day based on the general agent trained by all users' app usage data at the back-end side. We will evaluate the performance gain of the general agent in Section 5.1.5.

5 EVALUATION

In this section, we conduct extensive experiments to evaluate DeepAPP, including data-driven evaluation and a field study.

5.1 Data-driven evaluation

We conduct data-driven evaluations of DeepAPP on the dataset introduced in Section 2.2. It includes the app usage data of 443 active users, collected from a major mobile carrier of a big city for a period of 21 days (10 Apr. - 16 Apr., 2018 and 10 May. - 23 May., 2018). We divide the dataset into two parts, i.e., 14-day data for training and 7-day data for validation. Cross-validations, by repeating the experiments with different partitions of the training and validation data, have been conducted.

Performance criteria. We use precision and recall to measure the app prediction accuracy of DeepAPP. Precision is defined as the average ratio between the number of correctly-predicted apps and the number of all predicted apps in the next time slot. Recall is the average ratio between the number of correctly predicted apps and the number of real used apps of all users in the next time slot. In addition, we also measure the average execution time to measure the efficiency of DeepAPP.

Benchmarks. We compare the prediction accuracy of DeepAPP with following 5 baselines. All the parameters of baselines are set to the optimal values according to the empirical experiments on our dataset.

- *MFU.* Intuitively, we can always predict the next app as one of the most frequently used (MFU) M apps, which can be found based on the number of app usage records of each user in our dataset. M is set to 5.
- *FALCON.* Yan *et al.* [1] provide an effective context-aware app prediction method by utilizing spatial and temporal features (i.e. location and time). We derive the location information ("Home", "Work place" and "On the way") by the method introduced in [31].
- *APPM.* Parate *et al.* [4] leverage Prediction by Partial Match (PPM) model [32] for app prediction, which uses the longest app sequence to compute the probability of the following app.
- *LSTM.* Xu *et al.* [16] formulate the app prediction problem as a multi-label classification problem and propose a LSTM-based prediction model. We incorporate our context-aware state representation into their model.
- *DQN.* We also implement a DQN-based app prediction scheme. It is a simple way to leverage deep reinforcement learning in app prediction. We also implement our other designs, like the context-aware state and online updating, in this DQN-based scheme.

5.1.1 *Prediction accuracy.*

Figure 7 depicts the average prediction accuracy on the validation data. From the experiment result, we can see that DeepAPP provides the best prediction accuracy among other baselines. The reasons are as follows. First, DeepAPP learns a data-driven model-free agent to make prediction rather than traditional explicit models. The model-free agent can take complex environment context as input. Second, with reinforcement learning, DeepAPP can model the future reward of the apps in the time slot while other methods cannot, which is unreasonable in the real scenario. We also find that DeepAPP has similar performance as the DQN-based scheme. The DNN agent of DeepAPP focuses on reducing the time complexity of make an inference, which will be studied in Section 5.1.4.

5.1.2 *Evolution of prediction accuracy over time.*

Figure 8 presents the precision and recall of 443 mobile users on each day during a 7-day test. The prediction performance improves over time, which means DeepAPP can adapt well to app usage dynamics by updating the personalized agent online. The result also confirms the effectiveness of deep reinforcement learning in solving the time-varying prediction problem, allowing rapid adaptation to the change of app usage preference.

5.1.3 *Performance gain of the context-aware state.*

To verify the effectiveness of our context-aware state representation, we implement another version of DeepAPP (denoted as ”DeepAPP w/o S”) by only vectorizing the semantic locations (i.e., ”Home”, ”Work place” and ”On the way”). As depicted in Figure 9, the precision and recall of DeepAPP is 7.6% and 7.3% higher than those of DeepAPP w/o S. This is because DeepAPP w/o S cannot learn the app usage pattern at some locations where users have not been to or do not have semantic information.

5.1.4 *Execution time.*

DeepAPP involves lightweight computation in the inference while making prediction. We use app usage data of all users to test the average prediction time of two DRL-based

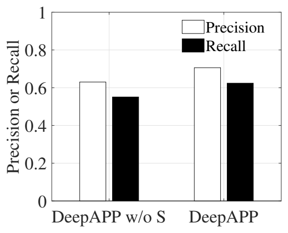


Figure 9: Context-aware state.

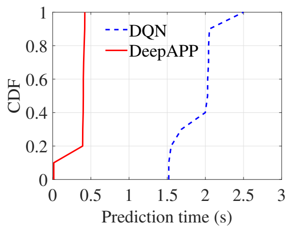


Figure 10: Time complexity.

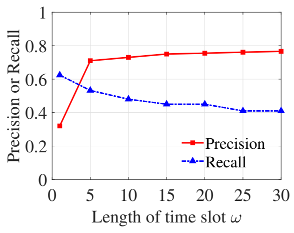


Figure 12: Effect of the length of time slot ω.

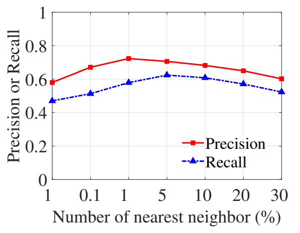


Figure 13: Effect of number of nearest neighbors K.

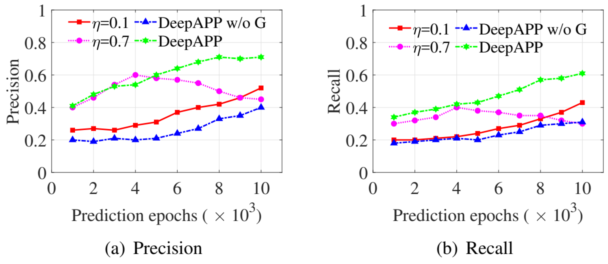


Figure 11: General agent.

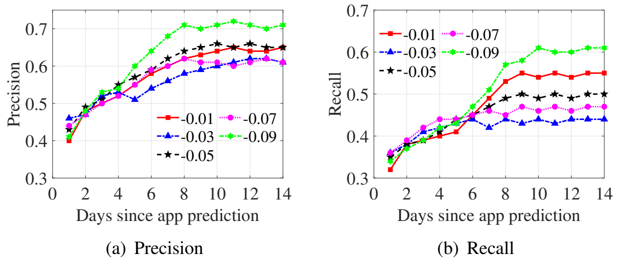


Figure 14: Effect of the decrease rate p.

Table 4: F-Score with different length of time slot ω.

ω (min)	1	5	10	15	20	25	30
F-Score	0.584	0.589	0.579	0.563	0.564	0.532	0.534

Table 5: F-Score with different K.

K (x% of $\|\mathcal{A}\|$)	1	0.1%	1%	5%	10%	20%	30%
F-Score	0.519	0.581	0.643	0.662	0.643	0.608	0.560

methods (i.e. DQN and DeepAPP). Figure 10 depicts the CDF of the prediction time. The average prediction time of DeepAPP (0.31 seconds) is far less than that of DQN (2.04 seconds). This indicates that our lightweight actor-critic based agent can effectively reduce the prediction time and enable real-time app prediction.

5.1.5 Performance gain of the general agent. We verify the effectiveness of the general agent in DeepAPP. We implement another two versions of DeepAPP. The first version discards the general agent, which denoted as "DeepAPP w/o G". The second version adopts two fixed balance coefficient η values to combine the personalized agent with general agent while online inference, which denoted as "η=0.1" and "η=0.7".

Figure 11 depicts prediction details of these three methods during online learning within 10,000 prediction epochs. With the increase of epochs, both the precision and recall increase. DeepAPP is consistently higher than the DeepAPP w/o G during online learning. The results confirm that the general agent succeeds in solving the data sparseness problem.

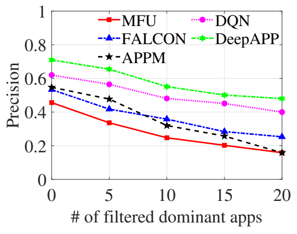
Besides, we find that the performance of DeepAPP with a linearly decreasing η value is superior to that under a fixed η value. For instance, under a larger η (i.e. 0.7), DeepAPP works well at the beginning, but weakens at the later stage. With a small η (0.1) and vice versa. A linearly decreasing η value can always maintain high precision and recall over time. We adopt a bigger η at the beginning, which addresses the data sparsity problem. As prediction epochs increase, we reduce the role of general agent and let the personalized agent dominated by the individual app usage data.

5.1.6 Parameter settings. We further test the choice of three parameters in DeepAPP, i.e., the length of time slot ω, the number of nearest neighbors of proto-action K and the decrease rate of the balance coefficient p.

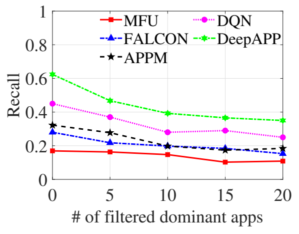
Length of time slot ω. Figure 12 depicts the performance of DeepAPP by varying the length of time slot ω from 1 minute to 30 minutes. As ω increases, the precision increases, but the recall gradually decreases. We select a proper ω by using F-Score [33], which achieves a balance between the precision and recall. As shown in Table 4, we can find F-Score reaches its maximum at ω =5, which is the default in the following experiments.

The number of nearest neighbors K. The motivation of the number of nearest neighbors of proto-action is to lower the impact of noisy actions which may occasionally fall near the proto-action. We conduct an experiment to select a proper K. Figure 13 shows the variation of accuracy by varying the number of nearest neighbors K from $K = 1$ to 30%. As we can see, when $K = 1$, the accuracy is worst, which proves the rationality of selecting K-nearest neighbor to find the optimal action. When $K > 1$, the technique can filter out noise actions which occasionally fall near the proto-action, resulting in the enhancement of the precision and recall of DeepAPP. As shown in Table 5, we also select a default $K = 5\%$ of $|\mathcal{A}|$ by using F-Score [33] as the default setting in the experiments.

The decrease rate p. In our design, we adopt an adaptive balance coefficient, which gradually reduces the weight of the general agent in the update of each personalized agent. We evaluate the performance of DeepAPP with different decrease rate of the balance coefficient from 0.09 to 0.01. Figure 14 depicts the performance of DeepAPP under various values of decrease rate with respect to the prediction epochs. Our method can maintain a high precision and recall when p is set to 0.09.

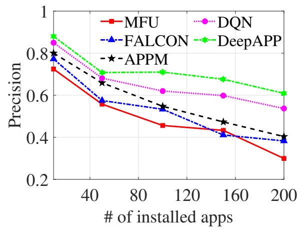


(a) Precision

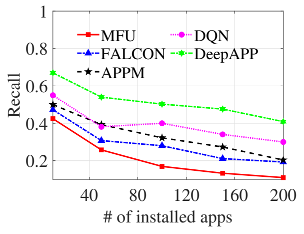


(b) Recall

Figure 15: Effect of dominant apps.

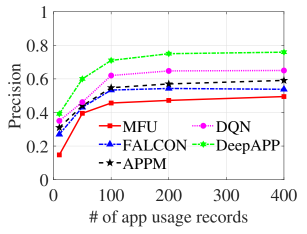


(a) Precision

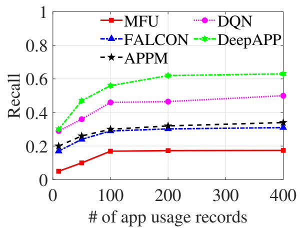


(b) Recall

Figure 16: Effect of the number of installed apps.



(a) Precision



(b) Recall

Figure 17: Effect of the number of app usage records.

5.1.7 *Performance under different scenarios.*

The above experiment results prove the effectiveness of DeepAPP for our dataset. We further study the performance of DeepAPP of different attributes, such as dominant apps, the number of installed apps and the number of app usage records.

Number of dominant apps. We study the impact of dominant apps (i.e., the most frequent apps) in the app prediction. We vary the number of dominant apps for an individual and then re-evaluate the prediction performance. As shown in Figure 15, the performance of DeepAPP is best among all benchmark methods, giving the precision of 65.5% and recall of 46.7%, compared with 41.8% and 21.8% in FALCON and 47.7% and 27.7% in APPM. The experiment results indicate that DeepAPP is also effective in predicting apps which are not used frequently.

Number of installed apps. When a user installs a large number of apps on smartphones, it will be more difficult to predict the next app. We explore how DeepAPP performs when the number of the installed apps (N) on smartphones is different. We first categorize the number of installed apps into 5 levels, i.e., $\{N < 10\}$, $\{N >= 10 \ \& \ N < 50\}$, $\{N >= 50 \ \& \ N < 100\}$, $\{N >= 100 \ \& \ N < 200\}$ and $\{N >= 200\}$. As shown in Figure 16, the precision and recall decrease as the number of installed apps increases. Especially, when the number of installed apps N is less than 10, the precision and recall are reached 88% and 58%, respectively. For larger N (i.e., $N >= 200$), the precision and recall are only about 60.1% and 40.9%. The experiment results demonstrate that the fewer the installed apps on smartphones, the easier for DeepAPP to predict the next apps.

Number of app usage records. The number of app usage records may have various impacts on the performance. We explore how DeepAPP performs when the number of app usage records (M) is different. We categorize the number of app usage records into 5 levels, i.e., $\{M < 50\}$, $\{M >= 50 \ \& \ M < 100\}$, $\{M >= 100 \ \& \ M < 200\}$, $\{M >= 200 \ \& \ M < 400\}$ and $\{M >= 400\}$. Figure 17 depicts the performance on different number of app usage records. With the increase of app usage records, the precision and recall are also improving, because the personalized agent can learn more app usage pattern when a user has a larger number of app usage records.

5.2 Field study

We also test DeepAPP by field experiments from 17 Sep. to 10 Nov. 2018. Compared with data-driven evaluations, in the field experiment, we can not only measure the accuracy of DeepAPP, but also collect the real user experience on DeepAPP. We deploy a system as the architecture in Figure 5. We recruit 29 participants and collect app usage records as ground truth. Participants include 13 females and 16 males, aged from 19 to 49, which have various occupations such as company employees, college teachers, college students, etc. After participants agree to take part in the experiment, we first install the Android application introduced in Section 4.2 on smartphones and monitor their app usage traces. We also collect their smartphone status such as power consumption and memory usage for the analysis of system overhead. At last, all participants successfully completed the experiment, and in all we collected 76,021 pieces of app usage records during the 55-day field experiment.

5.2.1 *User survey.*

We ask participants to complete a weekly questionnaires to collect the feedback on the usability of DeepAPP. Questionnaires are designed in a Likert scale format [34], which require participants to rate a statement from "strongly disagree (1)" to "strongly agree (5)". The results show that 87.51% of users are satisfied with our app prediction system, which is an alternative proof that our predictive model is effective and 71.88% of participants agree that the app can save their time of launching apps by preloading our predicted apps into the memory.

5.2.2 *Performance analysis.*

We analyze the performance of our field experiment from 3 aspects, i.e., accuracy, latency improvement and end-to-end prediction time.

Accuracy. We use the app usage data of participants to evaluate the accuracy of DeepAPP. Figure 18 depicts the evolution of precision and recall over time during the field experiment. As expected, like data-driven evaluations, DeepAPP can also quickly adapt to the time-variation of user preference and achieve high accuracy.

Latency improvement. We use the average ratio of the saved loading time to the launch time of smartphones without deploying DeepAPP to evaluate the time reduction on participants' smartphones. We profile the launch time of all installed apps on participants' smartphones. Then, we could obtain the time reduction according to the correctly-predicted result of the participants. This

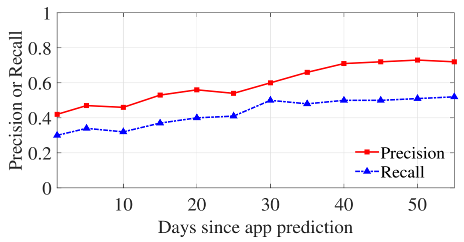


Figure 18: Accuracy during the field experiment.

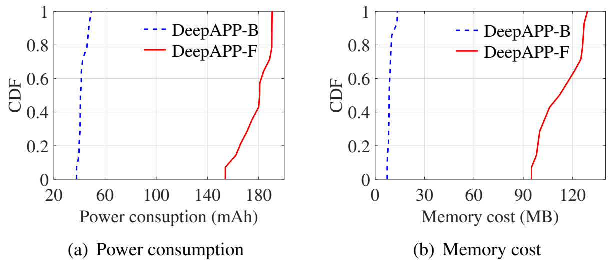


Figure 20: System overhead of DeepAPP prediction.

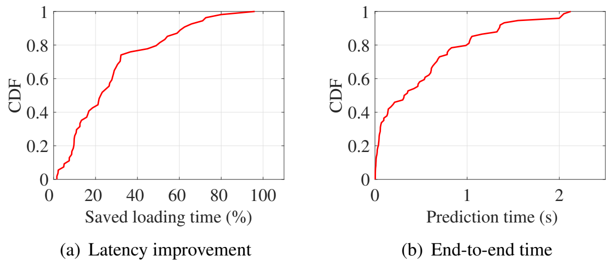


Figure 19: Performance analysis.

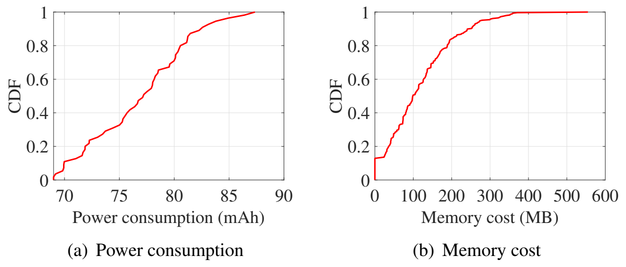


Figure 21: System overhead of app-preloading.

measurement ignores the launch time of apps if DeepAPP has pre-loaded the apps, which is neglectable in practice [2]. Figure 19(a) shows that our system can reduce the app loading time by 68.14% on average compared with no pre-loading.

End-to-end prediction time. The end-to-end prediction time is very important and directly related to user experience. We calculate the end-to-end prediction delay by the time difference between the start time of uploading the context information and the end time of receiving the predicted result, which can be easily obtained by the Android logcat from participants' smartphones. From Figure 19(b), we can see that prediction delay is negligible, i.e., less than 1 seconds of 80%, including both prediction computation and data transmission between the back-end component and the front-end component.

5.3 System overhead

DeepAPP may produce two types of overhead, i.e., 1) the power consumption and memory cost of running DeepAPP prediction and 2) the power consumption and memory cost caused by the apps pre-loaded by DeepAPP.

5.3.1 Overhead of DeepAPP prediction.

We test the overhead of DeepAPP prediction on 2 participants with the same model of smartphones (Honor 20 Pro). We implement two versions of DeepAPP of running app prediction, i.e., making inference on the back-end server (DeepAPP-B) and making inference on the front-end (DeepAPP-F).

Power consumption. In order to estimate the power consumption, we estimate the power consumption rate of each app by a power monitoring application (Accubattery [35]). As depicted in Figure 20(a), the extra cost of DeepAPP-B and DeepAPP-F are about 42.48 mAh and 178.87 mAh on average in a day, which can be almost ignored compared with total battery capacity of smartphones. At the same time, compared with DeepAPP-F, DeepAPP-B has less power consumption. This is because DeepAPP-B performs prediction inference and agent updating at the back-end server, saving the energy consumption of smartphones. The customized design of the context-aware module in DeepAPP does not cause additional energy consumption, compared with other systems [1, 11].

Memory cost. Figure 20(b) depicts that the memory cost and computation requirement of two versions of DeepAPP. The results reveal that the average memory cost of DeepAPP-B is less than 9.3 MB, and does not consume much extra memory (i.e. 113.6 MB) during making inference on the front-end. Current smartphones, like Samsung Galaxy S9 and HUAWEI Mate 10 Pro, have at least 4GB memory and 8-core CPU, which can totally support DeepAPP online inference without a back-end support.

5.3.2 Overhead of app pre-loading.

The overhead of app pre-loading is mainly in two aspects: power and memory. We test the overhead of pre-loading on 4 participants with the same model of smartphones (HUAWEI Mate 10 Pro).

Power consumption. As apps share hardware components, loading apps simultaneously will save more power than loading apps separately [36], and thus the power consumption of users actually consume is less than what we estimate. Figure 21(a) depicts the estimated average power consumption in different days. We find that the app consumes less than 2.18% of battery powers of participants' smartphones on average in a day, which is negligible for the total battery powers (4000 mAh). The reasons are as follows. First, DeepAPP does not pre-load unpredictable apps, which will not consume any additional power consumption. Second, DeepAPP only introduces the few additional power consumption by misprediction, which can be ignored by the higher precision of DeepAPP.

Memory cost. Due to app pre-loading will bring extra memory cost of smartphones, we further test the memory usage on users' smartphones. With the users' consent, we monitor the memory usage of participants and obtain a result in Figure 21(b). As shown, app pre-loading does not consume much memory on average, i.e. 190.6

MB of total memory, because the background scheduler only preloads apps that will be used in the next time slot. Besides, if the user does not use the predicted apps, we will immediately unload the apps in memory.

6 RELATED WORK

App prediction. Many app prediction methods [1, 8–16, 37] have been designed for personalized app prediction. Huang *et al.* [11] model the app usage transition by a first-order Markov model and use the contextual information, such as time, location and the latest used app. Natarajan *et al.* [9] model the app usage sequences using a cluster-level Markov model, which segments app usage behaviors cross multiple users into a number of clusters. Bayesian framework [11] improves the performance of app prediction by combining different features. PTAN [14] combines various explicit features such as location semantics (either home or work) and implicit features such as app usage information. Parate *et al.* [4] and Zhu *et al.* [10] transform the place into semantic location to improve the performance of app prediction on semantic location. Chen *et al.* [15] consider rich context by graph embedding techniques for personalized prediction. APPM [4] separately considers the prediction of a few specific apps with their launch time to prefetch in time on smartphone. However, most of them build an explicit model, which cannot capture the impact of all potential factors.

There are also some works that are orthogonal to our work. They benefit practical apps on smartphones from different perspectives. SmartIO [3] reduces the application loading delay by assigning priorities to reads and writes. HUSH [38] unloads background apps for energy saving automatically. CAS [7] develops a context-aware application scheduling system that unloads and pre-loads background applications in a timely manner. ShuffleDog [39] builds a resource manager to efficiently schedule system resources for reducing the user-perceived latency of apps.

Deep reinforcement learning. Mnih *et al.* solve the problem of stability and convergence in high-dimensional data input using Deep Q Network (DQN) [18]. Many technologies have been proposed to improve the performance of DQN. Prioritized experience replay [40] is put forward to improve the learning efficiency. Previous works have further extended deep reinforcement learning to continuous action space and large discrete action space. An actor-critic based on the policy gradient [28] is presented to solve the continuous control problem. Mnih et al. [41] propose asynchronous gradient descent for optimization of deep neural network and show successful applications on various domains. Arnold *et al.* [20] present an actor-critic architecture which can act in a large discrete action space efficiently. Based on this architecture, our work designs a new actor-critic based agent for app prediction.

Recently, deep reinforcement learning has been studied and applied in many domains [42–47]. DSDPS [43] applies DRL for the distributed stream data processing system based on the previous experience rather than solving the complicated model. AuTO [44] leverages a two-tier DRL model based on the long-tail distribution of data center services to solve the automatic decision-making of traffic optimization. DRL-TE [46] leverages an efficient DRL-based control framework to solve the traffic engineering problem in communication networks. This paper extends the application of DRL to the app prediction problem.

Cellular data. There are some studies using the same cellular network request data as our study [48–53]. SAMPLES [48] provides a framework to identify the application identity according to the network request by inspecting the HTTP header. CellSim [49] extracts similar trajectories from a large-scale cellular dataset. YU *et al.* [50] present a city-scale analysis of app usage data on smartphones. TU *et al.* [51] re-identify a user in the crowd by the apps she uses and quantify the uniqueness of app usage. Wang *et al.* [52] discover users' identifies in multiple cyberspace. However, the above studies do not leverage the app usage data for real-time app prediction.

7 DISCUSSION

Dataset limitation. The cellular data cannot capture the app usages that do not make any network requests or make requests through Wi-Fi networks. However, such a limitation does not impact the performance much. First, since app usages collect from a large number of users, DeepAPP can still learn the general app usage behaviors of different users by the general agent. Second, DeepAPP updates the personalized agent based on the online app usages, which can cover all the apps the user opens.

Deployment cost. The deployment cost of DeepAPP is mainly associated with the expense of back-end infrastructure placement. The back-end component consists of two modules, i.e. the context database and two agents. As the kernel of DeepAPP, agents provide lightweight prediction model for user, which requires adequate computing resources (e.g. CPU) for the running of DeepAPP. Besides, context database provides the reservation of transition samples and hence a reliable and effective storage system is available.

Privacy issues. In the data-driven evaluation, the data provider has anonymized the app usage data by replacing the user identification by a hash code. The app usage data only contain anonymized records of cell tower sequences, without any information relating to text messages, phone conversations or search contents. Besides, we randomly select from a large dataset for our dataset, which can also prevent leaking the mobile users' privacy.

In the field experiments, DeepAPP collects some private sensitive data (e.g. contextual information) from volunteers. To protect the privacy, we anonymize the user identifier in the database. In addition, since our context feature only need the POI distribution around the user, we do not need the exact location of the user.

8 CONCLUSION

This paper presents DeepAPP, a deep reinforcement learning framework for mobile app prediction, which predicts the next apps in the next time slot on her mobile device. By combining a context-aware state representation method, a personalized agent and a general agent together, DeepAPP can provide effective and efficient app prediction. Extensive data-driven evaluations and field experiments demonstrate high performance gain of DeepAPP.

ACKNOWLEDGEMENTS

We sincerely thank the anonymous shepherd and reviewers for their valuable comments. Zhihao Shen, Kang Yang, Xi Zhao and Jianhua Zou are supported by the National Natural Science Foundation of China Grant No. 91746111.

REFERENCES

[1] Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. Fast app launching for mobile devices using predictive user context. In *ACM MobiSys*, 2012.

[2] Xu Ye, Lin Mu, Lu Hong, Giuseppe Cardone, Nicholas Lane, Zhenyu Chen, Andrew Campbell, and Tanzeem Choudhury. Preference, context and communities:a multi-faceted approach to predicting smartphone app usage patterns. In *ISWC*, 2013.

[3] David T. Nguyen, Gang Zhou, Guoliang Xing, Xin Qi, Zijiang Hao, Ge Peng, and Qing Yang. Reducing smartphone application delay through read/write isolation. In *ACM MobiSys*, 2015.

[4] Abhinav Parate, Matthias BÃűhmer, David Chu, Deepak Ganesan, and Benjamin M. Marlin. Practical prediction and prefetch for faster access to applications on mobile phones. In *ACM Ubicomp*, 2013.

[5] Paul Baumann and Silvia Santini. Every byte counts: Selective prefetching for mobile applications. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 1(2):6, 2017.

[6] Yichuan Wang, Xin Liu, David Chu, and Yunxin Liu. Earlybird: Mobile prefetching of social network feeds via content preference mining and usage pattern analysis. In *ACM MobiHoc*, 2015.

[7] Joohyun Lee, Kyunghan Lee, Euijin Jeong, Jaemin Jo, and Ness B. Shroff. Context-aware application scheduling in mobile systems: what will users do and not do next? In *ACM Ubicomp*, 2016.

[8] Hong Cao and Miao Lin. Mining smartphone data for app usage prediction and recommendations: A survey. *Pervasive and Mobile Computing*, 37:1–22, 2017.

[9] Nagarajan Natarajan, Donghyuk Shin, and Inderjit S. Dhillon. Which app will you use next? collaborative filtering with interactional context. In *ACM RecSys*, 2013.

[10] Hengshu Zhu, Huanhuan Cao, Enhong Chen, Hui Xiong, and Jilei Tian. Exploiting enriched contextual information for mobile app classification. In *ACM CIKM*, 2012.

[11] Ke Huang, Chunhui Zhang, Xiaoxiao Ma, and Guanling Chen. Predicting mobile application usage using contextual information. In *ACM Ubicomp*, 2012.

[12] Choonsung Shin, Jin Hyuk Hong, and Anind K. Dey. Understanding and prediction of mobile application usage for smart phones. In *ACM Ubicomp*, 2012.

[13] Zhung Xun Liao, Shou Chung Li, Wen Chih Peng, Philip S. Yu, and Te Chuan Liu. On the feature discovery for app usage prediction in smartphones. In *IEEE ICDM*, 2014.

[14] Ricardo Baezayates, Di Jiang, Fabrizio Silvestri, and Beverly Harrison. Predicting the next app that you are going to use. In *ACM WSDM*, 2015.

[15] Xinlei Chen, Yu Wang, Jiayou He, Shijia Pan, Yong Li, and Pei Zhang. CAP: Context-aware app usage prediction with heterogeneous graph embedding. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 3(1):4, 2019.

[16] Shijian Xu, Wenzhong Li, Xiao Zhang, Songcheng Gao, Tong Zhan, Yongzhu Zhao, Wei-wei Zhu, and Tianzi Sun. Predicting smartphone app usage with recurrent neural networks. In *WASA*, 2018.

[17] Vassilis Kostakos, Denzil Ferreira, Jorge Goncalves, and Simo Hosio. Modelling smartphone usage: a markov state transition model. In *ACM UbiComp*, 2016.

[18] V Mnih, K Kavukcuoglu, D Silver, A. A. Rusu, J Veness, M. G. Bellemare, A Graves, M Riedmiller, A. K. Fidjeland, and G Ostrovski. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[19] Sicong Liu, Yingyan Lin, Zimu Zhou, Kaiming Nan, Hui Liu, and Junzhao Du. On-demand deep model compression for mobile devices: A usage-driven model selection framework. In *ACM MobiSys*, 2018.

[20] G. Dulac-Arnold, R. Evans, H. van Hasselt, P. Sunehag, T. Lillicrap, J. Hunt, T. Mann, T. Weber, T. Degris, and B. Coppin. Deep Reinforcement Learning in Large Discrete Action Spaces. *ArXiv e-prints*, 2015.

[21] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, and Michael Isard. TensorFlow: a system for large-scale machine learning. In *USENIX OSDI*, 2016.

[22] TensorFlow Lite. https://tensorflow.google.cn/lite/.

[23] WJX. www.wjx.cn.

[24] AMAP. https://www.amap.com.

[25] Chen Sun, Jun Zheng, Huiping Yao, Yang Wang, and D. Frank Hsu. Apprush: Using dynamic shortcuts to facilitate application launching on mobile devices. *Procedia Computer Science*, 19(19):445–452, 2013.

[26] Hengshu Zhu, Hui Xiong, Yong Ge, and Enhong Chen. Ranking fraud detection for mobile apps:a holistic view. In *ACM CIKM*, 2013.

[27] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *ICML*, 2014.

[28] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *Computer Science*, 8(6):A187, 2015.

[29] R. S. Sutton and A. G. Barto. Reinforcement learning: An introduction. *Machine Learning*, 8(3-4):225–227, 1992.

[30] Feiping Nie, Heng Huang, Xiao Cai, and Chris H Ding. Efficient and robust feature selection via joint âĎŞ2, 1-norms minimization. In *NIPS*, 2010.

[31] Sibren Isaacman, Richard Becker, RamÃşn CÃąceres, Stephen Kobourov, Margaret Martonosi, James Rowland, and Alexander Varshavsky. Identifying important places in people's lives from cellular network data. In *IEEE PerCom*, 2011.

[32] J. Cleary and I. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, 1984.

[33] Bjornar Larsen and Chinatsu Aone. Fast and effective text mining using linear-time document clustering. In *ACM SIGKDD*, 1999.

[34] Gerald Albaum. The likert scale revisited. *Market Research Society. Journal.*, 39(2):1–21, 1997.

[35] Accubattery. https://www.accubatteryapp.com/.

[36] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. AppScope: application energy metering framework for android smartphones using kernel activity monitoring. In *USENIX ATC*, 2012.

[37] Zhung Xun Liao, Yi Chin Pan, Wen Chih Peng, and Po Ruey Lei. On mining mobile apps usage behavior for predicting apps usage in smartphones. In *ACM CIKM*, 2013.

[38] Xiaomeng Chen, Abhilash Jindal, Ning Ding, Yu Charlie Hu, Maruti Gupta, and Rath Vannithamby. Smartphone background activities in the wild:origin, energy drain, and optimization. In *ACM MOBICOM*, 2015.

[39] Gang Huang, Mengwei Xu, Felix Xiaozhu Lin, Yunxin Liu, Yun Ma, Saumay Pushp, and Xuanzhe Liu. Shuffledog: Characterizing and adapting user-perceived latency of android apps. *IEEE Transactions on Mobile Computing*, 16(10):2913–2926, 2017.

[40] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized Experience Replay. *ArXiv e-prints*, 2015.

[41] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *ICML*, 2016.

[42] Guanjie Zheng, Fuzheng Zhang, Zihan Zheng, Yang Xiang, Nicholas Jing Yuan, Xing Xie, and Zhenhui Li. DRN: A deep reinforcement learning framework for news recommendation. In *WWW*, 2018.

[43] Teng Li, Zhiyuan Xu, Jian Tang, and Yanzhi Wang. Model-free control for distributed stream data processing using deep reinforcement learning. In *VLDB*, 2018.

[44] Chen Li, Lingys Justinas, Chen Kai, and Liu Feng. AuTO: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *ACM SIGCOMM*, 2018.

[45] Xianzhong Ding, Wan Du, and Alberto Cerpa. OCTOPUS: Deep reinforcement learning for holistic smart building control. In *ACM BuildSys*, 2019.

[46] Zhiyuan Xu, Jian Tang, Jingsong Meng, Weiyi Zhang, Yanzhi Wang, Chi Harold Liu, and Dejun Yang. Experience-driven networking: A deep reinforcement learning based approach. In *IEEE INFOCOM*, 2018.

[47] Zhenfeng Shao, Wenjing Wu, Zhongyuan Wang, Wan Du, and Chengyuan Li. Seaships: A large-scale precisely annotated dataset for ship detection. *IEEE Transactions on Multimedia*, 20(10):2593–2604, 2018.

[48] Hongyi Yao, Gyan Ranjan, Alok Tongaonkar, Yong Liao, and Zhuoqing Morley Mao. SAMPLES:self adaptive mining of persistent lexical snippets for classifying mobile application traffic. In *ACM MobiCom*, 2015.

[49] Zhihao Shen, Wan Du, Xi Zhao, and Jianhua Zou. Retrieving similar trajectories from cellular data at city scale. *arXiv preprint arXiv:1907.12371*, 2019.

[50] Donghan Yu, Yong Li, Fengli Xu, Pengyu Zhang, and Vassilis Kostakos. Smartphone app usage prediction using points of interest. In *ACM Ubicomp*, 2018.

[51] Zhen Tu, Runtong LI, Yong Li, Gang Wang, Di Wu, Pan Hui, Li Su, and Jin Depeng. Your apps give you away: Distinguishing mobile users by their app usage fingerprints. In *ACM Ubicomp*, 2018.

[52] Huandong Wang, Chen Gao, Yong Li, Zhili Zhang, and Depeng Jin. From fingerprint to footprint: Revealing physical world privacy leakage by cyberspace cookie logs. In *ACM CIKM*, 2017.

[53] Deren Li and Zhenfeng Shao. The new era for geo-information. *Science in China Series F: Information Sciences*, 52(7):1233–1242, 2009.