
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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SenSys ’19, November 10–13, 2019, New York, NY, USA  
© 2019 Association for Computing Machinery.  
ACM ISBN 978-1-4503-6950-3/19/11 . . $15.00  
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 (\(20^{20} = 1,048,576\)), 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. At the same time, we also train 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 personalization enhancement technique by the data of all available users. 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.

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

Table 1: Examples of app usage data.
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 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 trial 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 $s$ 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 $\pi$ 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 $a$ affects the environment. Every action gets a feedback from the environment. According to the feedback, we calculate a reward $r(s, a)$, which indicates how good or bad an action $a$ changes the environment given a specific state $s$. Based on the reward, a value function $Q(s, a)$ 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, $Q(s, a)$ is the long-term reward that an agent expects to obtain in the future, where $r_t$ is the reward of step $t$, and $\lambda$ is the discount factor.

$$Q(s, a) = E[\sum_{t=0}^{\infty} \lambda^tr_t|s_0]$$ (1)
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 \((s_t, a_t, r_t, s_{t+1})\). The agent first perceives a state \(s\) and generates an action \(a\) by running the policy \(\pi\). Then, the agent obtains a reward \(r\) given by the environment and updates the policy based on the estimate of \(Q(s, a)\). 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 DEEPPAPP

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.

**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 \(a\), where \(a_i = 1\) 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.

\[
 r = \begin{cases} 
 1, & N_r = 0 \land N_p = 0 \\
 N_r/N_p, & N_r \neq 0 \land N_p \neq 0 \\
 -5, & N_r = 0 \lor N_p = 0 
\end{cases}
\]  

**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

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S)</td>
<td>State space, state</td>
</tr>
<tr>
<td>(A)</td>
<td>Action space, action</td>
</tr>
<tr>
<td>(A_u)</td>
<td>The set of apps on a user’s smartphone</td>
</tr>
<tr>
<td>(r)</td>
<td>Reward</td>
</tr>
<tr>
<td>(\theta_\mu)</td>
<td>Parameters of actor network</td>
</tr>
<tr>
<td>(\theta_Q)</td>
<td>Parameters of critic network</td>
</tr>
<tr>
<td>(B)</td>
<td>Replay buffer</td>
</tr>
<tr>
<td>(\hat{a})</td>
<td>Proto-action</td>
</tr>
<tr>
<td>(K)</td>
<td>Number of nearest neighbors of (\hat{a})</td>
</tr>
<tr>
<td>(x)</td>
<td>App feature</td>
</tr>
<tr>
<td>(l)</td>
<td>Context feature</td>
</tr>
<tr>
<td>(t)</td>
<td>Time feature</td>
</tr>
<tr>
<td>(k)</td>
<td>Prediction epoch</td>
</tr>
<tr>
<td>(\omega)</td>
<td>The length of time slot</td>
</tr>
<tr>
<td>(p)</td>
<td>The decrease rate of the balance coefficient</td>
</tr>
</tbody>
</table>

**Table 2: Notations used in this paper.**

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.

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 \(t\), 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 \(s_t = (x_t, l_t, t_k)\), which consists of three key elements: the app feature \(x_t\), the context feature \(l_t\) and the time feature \(t_k\).

**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 \(x_{ik} = [x_{i1k}, x_{i2k}, ..., x_{inik}]\), where each \(x_{ijk}\) 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 providers), 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 \(l_{ik} = [l_{i1k}, l_{i2k}, ..., l_{ikon}]\) for \(m\) types of POI (23 in our implementation, which corresponds to the number of category of POI). Each \(l_{ijk}\) is the number of POI category \(j\) within the radius of 500 meters. We also normalize the context feature \(l_{ik}\) 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 \(t_k\) with the dimension of \(24 \times 60 / \omega\), where \(\omega\) 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 \(\hat{a}\)) in the continuous space, which may not be in the original action space \(A\). Next, the predicted result is passed to the discretizer to find the most likely actions \(A_K\) in the action space, which are the actions close to the proto-action \(\hat{a}\). Finally, we adopt the critic network to select the action \(a\) with highest Q value in \(A_K\).

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

**Discretizer.** Normally, the actions with lower Q values may occasionally fall near the proto-action \(\hat{a}\), 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 \(\hat{a}\) 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 \(A_K\). As shown in Eq. 3, we enumerate all actions in \(A\) to find \(K\) actions \(A_K\) that are close to the proto-action \(\hat{a}\).

\[
\min_{a \in A} || a - \hat{a} ||_2 \\
\text{s.t.: } a_i \in \{0, 1\}, \quad a_i \in A_u
\]  

With Eq. 3, we still have to go through all actions in \(A\). 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., \(|A|\) evaluations of the neural network).
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 the 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:

$$L = \frac{1}{N} \sum_{i=1}^{N} (Q_{tgt} - Q(s_i, a_i|\theta_Q))^2$$

(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.

$$Q_{tgt} = r_i + \lambda Q(s_{i+1}, a_{i+1}|\theta_{Q}^\mu)$$

(6)

where $\lambda$ 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:

$$\nabla_{\theta_{Q}^\mu} J = \frac{1}{N} \sum_{i=1}^{N} \nabla_a Q(s_i, a_i|\theta_Q)\nabla_{\theta_{Q}^\mu} Q(s_i, a_i|\theta_{Q}^\mu)$$

(7)

where $\mu(s_i|\theta_{Q}^\mu)$ is the $K$ predicted results of the actor network and $Q(s_i, a_i|\theta_{Q}^\mu)$ 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.
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 \(\hat{a}\) by the actor network \(\mu^\theta\). 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 \(\epsilon I\) to the proto-action \(\hat{a}\) [28], which has the similar idea as \(\epsilon\)-greedy [29]. As \(\epsilon\) decreases with the prediction epoch, more certain action will be taken with more training. \(I\) 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 \(\hat{a}\) 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.

\[
\begin{align*}
\theta_Q^\tau & : = \tau \theta_Q + (1 - \tau) \theta_Q^\text{target} \\
\theta_\mu^\tau & : = \tau \theta_\mu + (1 - \tau) \theta_\mu^\text{target}
\end{align*}
\] (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 \(2.47 \times \) training time compared with the training on CPUs.

#### 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.
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
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 sparsity 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 noisy 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 |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.
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 pre-loading 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
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.** DeepAPP-B performs inference without a back-end support. 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 pre-
loads 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 frame-
work [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 person-
alized 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. Mnh 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. Mnh 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 ap-
plied 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 net-
work 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’ identities 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 com-
puting 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 identifica-
tion 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 frame-
work 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


