# DinoDroid: Testing Android Apps Using Deep Q-Networks

Yu Zhao, *Member, IEEE,* Brent Harrison, *Member, IEEE,* and Tingting Yu, *Member, IEEE*

**Abstract**—The large demand of mobile devices creates significant concerns about the quality of mobile applications (apps). Developers need to guarantee the quality of mobile apps before it is released to the market. There have been many approaches using different strategies to test the GUI of mobile apps. However, they still need improvement due to their limited effectiveness. In this paper, we propose DinoDroid, an approach based on deep Q-networks to automate testing of Android apps. DinoDroid learns a behavior model from a set of existing apps and the learned model can be used to explore and generate tests for new apps. DinoDroid is able to capture the fine-grained details of GUI events (e.g., the content of GUI widgets) and use them as features that are fed into deep neural network, which acts as the agent to guide app exploration. DinoDroid automatically adapts the learned model during the exploration without the need of any modeling strategies or pre-defined rules. We conduct experiments on 64 open-source Android apps. The results showed that DinoDroid outperforms existing Android testing tools in terms of code coverage and bug detection.

**Index Terms**—Mobile Testing, Deep Q-Networks, Reinforcement Learning.

## 1 Introduction

Mobile applications (apps) have become extremely popular with about three million apps in Google Play's app store [1]. The increase in app complexity has created significant concerns about the quality of apps. Also, because of the rapid releasing cycle of apps and limited human resources, it is difficult for developers to manually construct test cases. Therefore, different automated mobile app testing techniques have been developed and applied [2].

Test cases for mobile apps are often represented by sequences of GUI events [1] to mimic the interactions between users and apps. The goal of an automated test generator is generating such event sequences to achieve high code coverage and/or detecting bugs. A successful test generator is able to exercise the correct GUI widget on the current app page, so that when exercising that widget, it can bring the app to a new page, leading to the exploration of new events. However, existing mobile app testing tools often explore a limited set of events because they have limited capability of understanding which GUI events would expand the exploration like humans do. This can lead to automated test generators performing unnecessary actions that are unlikely to lead to new coverage or detect new bugs.

Many automated GUI testing approaches for mobile apps have been proposed, such as random testing [3], [4] and model-based testing [5], [6], [7]. Random testing (e.g., Monkey [3]) is popular in testing mobile apps because of its simplicity and availability. It generates tests by sending thousands of GUI events per second to the app. While random testing can sometimes be effective, it is difficult to explore hard-to-reach events to drive the app to new pages because of the natural of randomness. Model-based testing [7], [8] can improve code coverage by employing pre-defined strategies or rules to guide the app exploration. For example, A3E [7] employs depth-first search (DFS) to explore the model of an app under test (AUT) based on event-flow across app pages. Stoat [8] utilizes a stochastic Finite State Machine model to describe the behavior of AUT and then utilizes Markov Chain Monte Carlo sampling [9] to guide the testing. However, model-based testing often relies on human-designed models and it is almost impossible to precisely model an app's behavior. Also, many techniques apply pre-defined rules to the model for improving testing. For example, Stoat [8] designed rules to assign each event an execution weight in order to speed up exploration. However, these pre-defined rules are often derived from limited observations and may not generalize to a wide categories of apps.

To summarize, the inherent limitation of the above techniques is that they do not *automatically* understand GUI layout and the content of the GUI elements, so it is difficult for them to exercise the most effective events that will bring the app into new states. Recently, machine learning techniques have been proposed to perform GUI testing in mobile apps [10], [11], [12], [13]. For example, Humanoid [12] uses deep learning to learn from human-generated interaction traces and uses the learned model to guide test generation as a human tester. However, this approach relies on human-generated datasets (i.e., interaction traces) to train a model and needs to combine the model with a set of pre-defined rules to guide testing.

Reinforcement learning (RL) can teach machine to decide

- *Yu Zhao was with the Department of Computer Science and Computer Science and Cybersecurity, University of Central Missouri, Warrensburg, MO, 64093.*
  *E-mail: yzhao@ucmo.edu*
- *Brent Harrison was with the Department of Computer Science, University of Kentucky, Lexington, KY, 40506.*
  *E-mail: harrison@cs.uky.edu.*
- *Tingting Yu (corresponding author) was with the Department of Electrical Engineering and Computer Science, University of Cincinnati, Cincinnati, OH, 45221.*
  *E-mail: yutt@ucmail.uc.edu*

[1] In our setting, an event refers to an executable GUI widget associated with an action type (e.g., click, scroll, edit, swipe, etc).

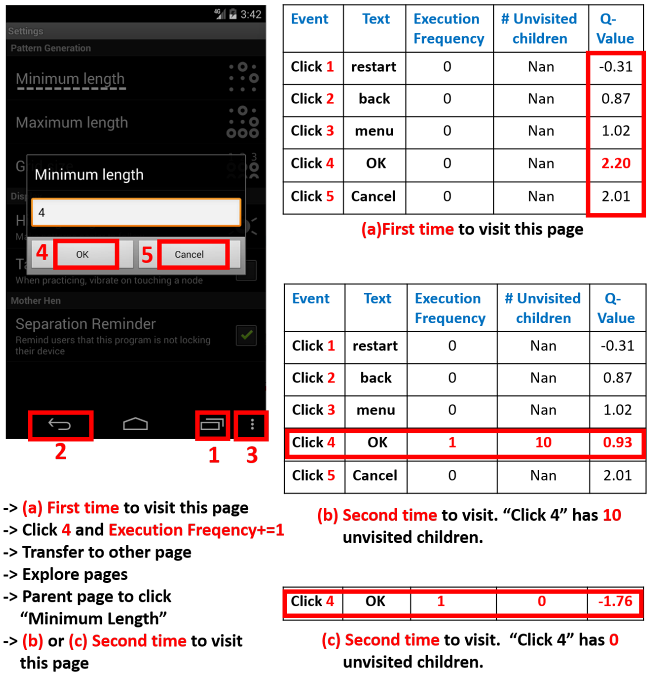
which events to explore rather than relying on pre-defined models or human-made strategies [14]. A Q-table is used to record the reward of each event and the information of previous testing knowledge. The reward function can be defined based on the differences between pages [11] or unique activities [10]. Reinforcement learners will learn to maximize cumulative reward with the goal of achieving higher code coverage or detecting more bugs.

While existing RL techniques have improved app testing, they focus on abstracting the information of app pages and then use the abstracted features to train behavior models for testing [15], [16]. For example, QBE [15], a Q-learning-based Android app testing tool, abstracts each app page into five categories based on the number of widgets (e.g., too-few, few, moderate, many, too-many). The five categories are used to decide which events to explore. However, existing RL techniques do not understand the fine-grained information of app pages like human testers normally do during testing, such as the execution frequencies and the content of GUI widgets. This is due to the limitation of the basic tabular setting of RL, which requires a finite number of states [17]. Therefore, the learned model may not capture the accurate behaviors of the app. Also, many RL-based techniques focus on training each app independently [18], [16], [19], [20], [11] and thus cannot transfer the model learned from one app to another.

To address the aforementioned challenges, we propose a novel approach, DinoDroid, based on deep Q-networks (DQN). DinoDroid learns a behavior model from a set of existing apps and the learned model can be used to explore and generate tests for new apps. During the training process, DinoDroid is able to understand and learn the details of app events by leveraging a deep neural network (DNN) model [21]. More precisely, we have developed a set of features taken as input by DinoDroid. The insight of these features represents what a human tester would do during the exploration. For example, a human tester may decide which widget to execute based on its content or how many times it was executed in the past. DinoDroid does not use any pre-defined rules or thresholds to tune the parameters of these features but let the DQN agent learn a behavior model based on the feature values (represented by vectors) automatically obtained during training and testing phases.

A key novel component of DinoDroid is a deep neural network (DNN) model that can process multiple complex features to predict Q value for each GUI event to guide Q-learning. With the DNN, DinoDroid can be easily extended to handle other types of features. Specifically, to test an app, DinoDroid first trains a set of existing apps to learn a behavior model. The DNN serves as an agent to compute Q values used to determine the action (i.e., which event to execute) at each iteration. In the meantime, DinoDroid maintains a special event flow graph (EFG) to record and update the feature vectors, which are used for DNN to compute Q values.

Because the features are often shared among different apps, DinoDroid is able to apply the model learned from existing apps to new AUTs. To do this, the agent continuously adapts the existing model to the new AUT by generating new actions to reach the desired testing goal (e.g., code coverage).



| Event | Text | Execution Frequency | # Unvisited children | Q-Value |
|---|---|---|---|---|
| Click 1 | restart | 0 | Nan | -0.31 |
| Click 2 | back | 0 | Nan | 0.87 |
| Click 3 | menu | 0 | Nan | 1.02 |
| Click 4 | OK | 0 | Nan | 2.20 |
| Click 5 | Cancel | 0 | Nan | 2.01 |

(a)First time to visit this page

| Event | Text | Execution Frequency | # Unvisited children | Q-Value |
|---|---|---|---|---|
| Click 1 | restart | 0 | Nan | -0.31 |
| Click 2 | back | 0 | Nan | 0.87 |
| Click 3 | menu | 0 | Nan | 1.02 |
| Click 4 | OK | 1 | 10 | 0.93 |
| Click 5 | Cancel | 0 | Nan | 2.01 |

(b) Second time to visit. "Click 4" has 10 unvisited children.

| | | | | |
|---|---|---|---|---|
| Click 4 | OK | 1 | 0 | -1.76 |

(c) Second time to visit. "Click 4" has 0 unvisited children.

-> (a) First time to visit this page
-> Click 4 and Execution Freqency+=1
-> Transfer to other page
-> Explore pages
-> Parent page to click "Minimum Length"
-> (b) or (c) Second time to visit this page

Fig. 1: A Motivating Example

In summary, our paper makes the following contributions:

- An approach to testing Android apps based on deep Q-learning.
- A novel and the first deep Q-learning model that can process complex features at a fine-grained level.
- An empirical study showing that the approach is effective at achieving higher code coverage and better bug detection than the state-of-the-art tools.
- The implementation of the approach as a publicly available tool, DinoDroid, along with all experiment data [22].

## 2 Motivation and Background

In this section, we first describe a motivating example of DinoDroid, followed by the background of deep Q-networks (DQN), the problem formulation, and the discussion of existing work.

### 2.1 A Motivating Example.

Fig. 1 shows an example of the app *lockpatterngenerator* [23]. This simple example demonstrates the ideas of DinoDroid, but the real testing process is much more complex. After clicking "Minimum length", a message box pops up with a textfield and two clickable buttons. Therefore, the current page of the app has a total of five events (i.e., "restart", "back", "menu", "OK", and "Cancel"). The home button is not considered because it is not specific to the app. When a human tester encounters this page, he/she needs to decide which event to execute based on his/her prior experience. For example, a tester is likely to execute the events that have never been executed before. The tester may also need to know the execution context of the current page (e.g., the layout of next page) to decide which widget to exercise.

In this example, suppose none of the five events on the current page have been executed before. Intuitively, the tester tends to select the "OK" event to execute because it is more likely to bring the app to a new page. "Cancel" is very possible to be the next event to consider because "restart", "back", and "menu" are general events, so the tester may have already had experience in executing them when testing other apps. In summary, to decide whether an event has a higher priority to be executed, the tester may need to consider its "features", such as how many times it was executed (i.e., execution frequency) and the content of the widget. DinoDroid is able to automatically learn a behavior model from a set of existing apps based on these features and the learned model can be used to test new apps.

Tab.(a)-Tab.(c) in Fig. 1 are used to illustrate DinoDroid. In this example, DinoDroid dynamically records the *feature values* for each event, including the execution frequency, the number of events not executed on the next page (i.e., child page), and the text on the event associated with the event. Next, DinoDroid employs a deep neural network to predict the accumulative reward (i.e., Q value) of each event on the current page based on the aforementioned features and selects the event that has the largest Q value to execute.

Tab.(a) shows the feature values and Q values when the page appears the first time. Since "OK" has the largest Q value, it is selected for execution. DinoDroid continues exploring the events on the new page and updating the Q value. When the second time this page appears, the Q value of the event on executing "OK" button decreases because it is already executed. As a result, "Cancel" has the largest Q value and is selected for execution. In this case (Tab.(b)), the child page of "OK" contains 10 unexecuted events. However, suppose the child page contains zero unexecuted events (Tab.(c)), the Q value becomes much smaller. This is because DinoDroid tends to select the event whose child page contains more unexecuted events.

The underlying assumption of our approach is that the application under test should follow the principle of least surprise (PLS). If an app does not meet the PLS, e.g., an "OK" textual widget is incorrectly associated with the functionally of "Cancel", it would mislead DinoDroid when finding the right events to execute. Specifically, DinoDroid exploits the learned knowledge to execute correct events that result in higher code coverage or triggering bugs.

## 2.2 Background

### 2.2.1 Q-Learning

Q-learning [24] is a model-free reinforcement learning method which seeks to learn a behavior policy for any finite Markov decision process (FMDP), Q-learning finds an optimal policy, $\pi$, that maximizes expected cumulative reward over a sequence of actions taken. Q-learning is based on trial-and-error learning in which an agent interacts with its environment and assigns utility estimates known as Q values to each state.

As shown in Fig. 2 the agent iteratively interacts with the outside environment. At each iteration $t$, the agent selects an action $a_t \in A$ based on the current state $s_t \in S$ and executes it on the outside environment. After exercising the action, there is a new state $s_{t+1} \in S$, which can be observed by the agent. In the meantime, an immediate reward $r_t \in R$ is received. Then the agent will update the Q values using the Bellman equation [25] as follows:

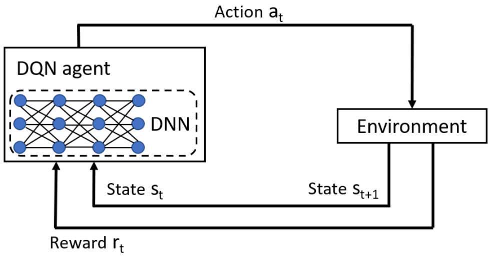


**Fig. 2: Deep Q-Networks**

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t)+\alpha*(r_t+\gamma*\max_a Q(s_{t+1},a)-Q(s_t,a_t))$$

In this equation, $\alpha$ is a learning rate between 0 and 1, $\gamma$ is a discount factor between 0 and 1, $s_t$ is the state at time $t$, and $a_t$ is the action taken at time $t$. Once learned, these Q values can be used to determine optimal behavior in each state by selecting action $a_t = \arg\max_a Q(s_t, a)$.

### 2.2.2 Deep Q-Networks

Deep Q-networks (DQN) are used to scale the classic Q-learning to more complex state and action spaces [26], [27]. For the classical Q-learning, $Q(s_t, a_t)$ are stored and visited in a Q table. It can only handle the fully observed, low-dimensional state and action space. As shown in Fig. 2, in DQN, a deep neural network (DNN), specifically involving convolutional neural networks (CNN) [28], is a multi-layered neural network that for a given state $s_t$ outputs Q values for each action $Q(s_t, a)$. Because a neural network can input and output high-dimensional state and action space, DQN has an ability to scale more complex state and action spaces. A neural network can also generalize Q values to unseen states, which is not possible when using a Q-table. It utilizes the follow loss function [27] to alter the network to minimize the temporal difference (TD) [29] error as a loss function $loss = r_t + \gamma * \max_a Q(s_{t+1}, a) - Q(s_t, a_t)$. The $\gamma$ is the discount factor which is between 0 and 1. In other word, with the input of $(s_t, a_t)$, the neural network is trained to predict the Q value as:

$$Q(s_t,a_t) = r_t + \gamma * \max_a Q(s_{t+1},a) \tag{1}$$

So in a training sample, the input is $(s_t, a_t)$ and output is the corresponding Q value which can be computed by $r_t + \gamma * \max_a Q(s_{t+1}, a)$.

## 2.3 Terminologies

A GUI *widget* is a graphical element of an app, such as a button, a text field, and a check box. An *event* is an executable GUI widget with a particular event type (e.g. click, long-click, swipe, edit), so a widget can be associated with one or more events. In our setting, a *state s* represents an app page (i.e., a set of widgets shown on the current screen. If the set of widgets is different, we have another page). We use $s_t$ to represent the current state and $s_{t+1}$ to represent the next state. A *reward r* is calculated based on the improvement of coverage. If code coverage increases, $r$ is assigned a positive number ($r$=5 by default); otherwise, $r$ is assigned a negative number ($r$=-2 by default). An *Agent* chooses which event to execute based on the accumulative rewards (i.e., Q values) on the current observed state. The chosen event is an *action*. For example, the agent performs an action by choosing to "click a button A" on the current page. A *Policy* is $\pi(a,s) = Pr(a_t = a|s_t = s)$, which maximizes the expected cumulative reward. Q-learning learns a policy to tell the agent what action to take.

### 2.4 Limitations of Existing Q-Learning Techniques

The techniques that are mostly related to DinoDroid are Q-learning-based Android app testing [18], [19], [20], [11]. These techniques are all based on a Q-table, which have several limitations. First, Q-table is not capable of handling high-dimensional features, such as text and image — adding a new feature space for a state to the Q-table will lead to its exponential growth. For example, a word embedding feature, represented as an M-dimension vector with $N$ possible values, would need $O(m^n)$ columns in a Q-table. Second, most existing techniques use the resource ID of an event to represent the state in a Q-table. However, different apps can have different ID assignments. Therefore, these techniques often focus on training and testing individual apps and cannot train a model from multiple apps to test new apps. The only work that uses Q-table to transfer knowledge among apps is QBE [10]. In order to limit the size of Q-table, QBE abstracts the state (i.e., an app page) into five categories in terms of the number of widgets on the current page and actions into seven general categories (i.e., menu, back, click, long-click, text, swipe, contextual). However, such abstractions may cause the learning to lose a lot of important information when making decisions on which events to execute.

Unlike traditional Q-learning, DinoDroid designs a novel deep neural network (DNN) model that can handle complex features with infinite feature space, such as word embedding with n-length vector and high-dimensional image matrix. DinoDroid is able to process these features at a fine-grained (ie., widget) level as opposed to abstraction [10]. DinoDroid's features are specifically designed to be general across apps, so the knowledge learned from existing apps can be transferred into new apps. For example, the feature regarding "execution frequency" applies to almost every app because exploring new events are always desirable. Also, when considering the "textual content" of events, an "OK" button in every app may have similar meaning across different apps.

## 3 DinoDroid Approach

Fig. 3 shows the overview of DinoDroid. An iteration $t$ begins with an app page. In the current state $s_t$, DinoDroid selects the event $e_t$ with the highest accumulative reward (i.e., Q value), performs the corresponding action $a_t$, and brings the app to a new state $s_{t+1}$. After performing $a_t$, DinoDroid employs a special event flow graph to generate feature values for each event in $s_{t+1}$. A reward $r_t$ is generated based on the observed code coverage and app crashes. A positive number is assigned to $r_t$ if code coverage increases or a unique bug is detected and a negative number is assigned if the coverage does not change. The deep Q-network agent uses a neural network model to compute and update the Q value for the event $e_t$ responding with $a_t$. The learning process continues iteratively from the apps provided as the training set. When testing a new app, DinoDroid uses the learned model as the initial model to guide the testing. The model is updated following the same process until a time limit is reached or the coverage reaches a plateau.

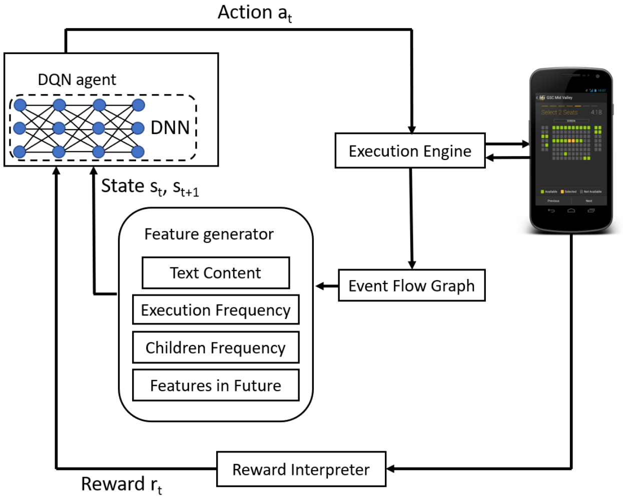


**Fig. 3: Approach Overview**

**Algorithm 1 DinoDroid's testing**

```
Require: App under test $AUT$, DQN's Model $M$ with before knowledge,
    execution time LIMIT
Ensure: updated new $M$
 1: if $M$ not exist then
 2:     $M \leftarrow$ buildNewModel() /*First time to run*/
 3: Memory ← ø /*Memory stores previous samples */
 4: G ← ø /*Initialize event flow graph */
 5: $p_0 \leftarrow$ execute(AUT) /*First lanuch to get the first page $p_0$*/
 6: G ← updateGraph($p_0$, G) /*update G with new page*/
 7: $s_0 \leftarrow$ featureGenerator($p_0$,G) /*Every event in $s_0$ contains 3 features*/
 8: while t < LIMIT do
 9:     if $t$ mod 10 equals 0 then
10:         sendSystemEvent() /*send random system event*/
11:     $a_t \leftarrow$ getActionEvent($s_t$, M) /*Event selection */
12:     $p_{t+1}$, codeCoverage, crash ← execute(AUT, $a_t$)
13:     $r_t \leftarrow$ rewardInterpreter(codeCoverage, crash)
14:     G ← updateGraph($p_{t+1}$, G) /*update G with new page*/
15:     $s_{t+1} \leftarrow$ featureGenerator($p_{t+1}$,G)
16:     $Q(s_t, a_t) = r_t + \gamma * \max_a Q(s_{t+1}, a)$ where $\gamma = 0.6$
17:     $Q \leftarrow Q(s_t, a_t)$
18:     batch ← extractTrainBatch(Memory) ∪ ($a_t$,$Q$)
19:     M ← updateModel(batch, M) /*Learning for DQN */
20:     Memory ← Memory ∪ ($a_t$,$Q$)
21:     $s_{t+1} \leftarrow s_t$
22: return M
```

### 3.1 DinoDroid's Algorithm

Alg. 1 shows the details of DinoDroid. When testing or training an app, DinoDroid first checks whether there exists a learned model, which saves all learned knowledge (weights on neural networks) of DinoDroid. If so, the model will be updated to adapt this app; otherwise, DinoDroid starts building a new model (Line 2). A memory is used to record the samples from earlier iterations (Line 3), where each sample contains the feature values of the executed event and the associated Q value. An event flow graph (EFG) is initialized for each app at the beginning of training or testing (Line 4). Note that the memory and the EFG will be abandoned after completing an iteration of the current app because the needed information has already been learned and saved in the model. The details of the EFG will be described in Section 3.2.2. Next, DinoDroid launches the app and reaches the first page. The initial state of DQN is obtained and the EFG is updated accordingly (Lines 5–6). The algorithm then takes the current page and the EFG to generate features for each GUI event on the current page (Line 7). DinoDroid considers three types of features as described in Section 3.2.

Now the iteration begins until a time limit is reached (Lines 8–22). To amplify the chance of bug detection, DinoDroid issues a random system event at every 10 iterations (Line 9–10) by using Stoat [8]'s system event generation library. At each iteration, DinoDroid uses a deep neural network model to select the event with the highest accumulative reward (i.e., Q value) and perform the corresponding action (Line 11). The details of the getActionEvent will be described in Section 3.3.2. After the execution, the algorithm obtains three kinds of information: the new page, the current code coverage, and the crash log (may be empty) (Line 12). The information is used to compute the reward $r_t$ (Line 13). Then, the event flow graph G is updated based on the new page $p_{t+1}$ (Line 14). With the updated G and the new page, the algorithm can generate features for each event in the state $s_{t+1}$ (Line 15). Based on $s_t$, $s_{t+1}$, $a_t$, and $r_t$, DinoDroid uses the equation (1) in Section 2.2.2 to compute the Q value of each event in $s_t$ (Line 16). To train the neural network, it uses a set of training samples, including both the current sample and history samples. Each sample uses $a_t$ as input and Q value as output (i.e., label). The history samples (obtained from earlier iterations) are recorded in a memory (Lines 17–20). In each iteration, the history sample size is set to be 4 due to cost-effectiveness. A larger history sample size can be set by users to retain more historical information for training, but it may also lead to a much higher cost.

## 3.2 Feature Generation

DinoDroid generates features for each event $e$. These features are fed into a neural network as shown in Fig. 6 so that the agent can choose an action (i.e., which event to exercise). DinoDroid currently supports three features, namely, *execution frequency of current events*, *execution frequency of children events*, and *textual content of events*. DinoDroid employs an event flow graph (EFG) at runtime to obtain the features and transforms them into numerical data format vectors provided as inputs to the neural network.

### 3.2.1 Types of Features

**Execution frequency of current events (FCR).** The insight behind this feature is that a human often avoids repeatedly executing an event that brings the app to the same page. Therefore, all GUI events should be given chances to execute. Instead of using pre-defined rules or thresholds to weigh different events to decide which ones should be assigned higher priorities to execute [8], DinoDroid automatically guides the selection of events based on the recorded FCR feature value. FCR encodes the number of times that each event was executed during the exploration. The feature value is updated by adding one when the event is executed.

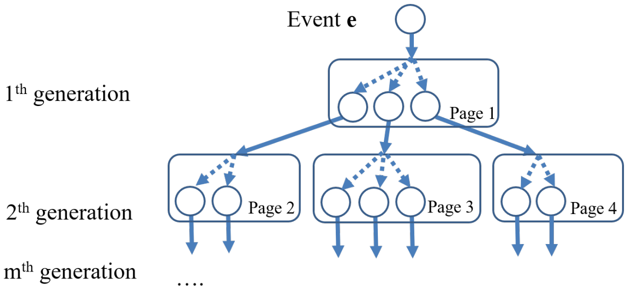


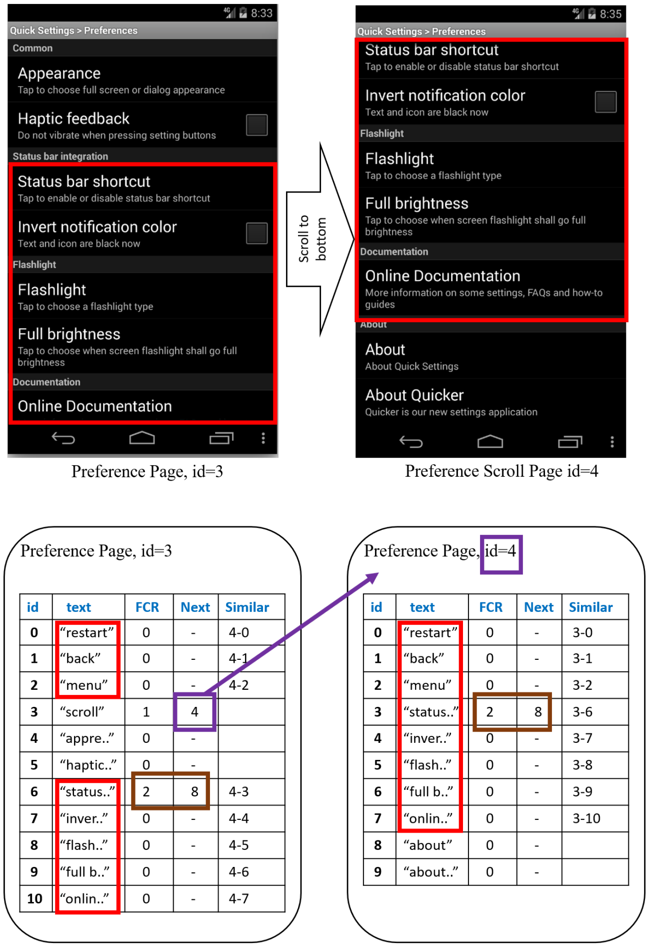
**Fig. 4: FCD extraction**

**Execution frequency of children events (FCD).** A human tester often makes a decision on which event to execute not only based on the events of current page, but also the succeeding pages (children pages) after executing an event. For example, if executing an event $e$ brings the app into a page containing many unexecuted events, $e$ is likely to be selected over the other events that bring the app into a page containing all events already being executed. Toward this end, DinoDroid uses the execution frequencies of events on children pages as a feature for each event of the current page. The $m^{th}$ ($m > 0$) generation of children pages are defined as the succeeding pages with distance $m$ from the current event in the event flow graph. As the example shown in Fig. 4 , the first generation of children pages contains only one page (i.e., $Page_1$), which results from executing the selected event (by the agent) on the current page. The second generation of children pages includes pages resulting from executing all events on the page ($Page_1$). DinoDroid's FCD feature encodes the execution frequencies of events in the $K^{th}$ generation of children pages. By default $K$ = 3, as we observed that if $K > 3$, it almost does not affect the selection of an event.

DinoDroid employs a fixed length feature vector for each generation of children pages. Intuitively, each element in the vector represents an event on the current generation and the value of the element records the execution frequency of the event. However, a generation of children pages may contain a number of events; encoding them all into a feature vector may lead to an unbearable size of vectors for a neural network to train. Instead, DinoDroid encodes each element in the vector as the number of events in the current generation that were executed $N$ times, where $N$ is equal to the index of the vector. By default, the length of the vector is set to 10. For example, if there is a total of 20 events in the current generation $m$, where 12 events were never executed and 8 events were executed twice, the feature vector of $m$ will be $[12, 0, 8, 0, 0, 0, 0, 0, 0, 0]$. Since $K$ is considered by DinoDroid, it will generate $K$ vectors for the FCD feature.

**Textual content of events (TXC).** The textual content of events may help the agent make the decision on which event to select. In the example of Fig. 1, the meaning of "OK" event suggests that it has a higher chance to bring the app to a new page than the other events. To obtain the TXC feature for each event, DinoDroid first employs Word2Vec [30] to convert each word in the event to a vector with a fixed length $L$. The Word2Vec model is trained from a public dataset text8 containing 16 million words and is provided along with the source code of Word2Vec [31].

The text on an event widget can have more than one words, such as "status bar shortcut" in Fig. 5. Therefore, we



8:33

Quick Settings > Preferences

Common

Appearance
Tap to choose full screen or dialog appearance

Haptic feedback
Do not vibrate when pressing setting buttons

Status bar integration

Status bar shortcut
Tap to enable or disable status bar shortcut

Invert notification color
Text and icon are black now

Flashlight

Flashlight
Tap to choose a flashlight type

Full brightness
Tap to choose when screen flashlight shall go full brightness

Documentation

Online Documentation

Preference Page, id=3

8:35

Quick Settings > Preferences

Status bar shortcut
Tap to enable or disable status bar shortcut

Invert notification color
Text and icon are black now

Flashlight

Flashlight
Tap to choose a flashlight type

Full brightness
Tap to choose when screen flashlight shall go full brightness

Documentation

Online Documentation
More information on some settings, FAQs and how-to guides

About

About
About Quick Settings

About Quicker
Quicker is our new settings application

Preference Scroll Page id=4

Preference Page, id=3

| id | text | FCR | Next | Similar |
|---|---|---|---|---|
| 0 | “restart” | 0 | - | 4-0 |
| 1 | “back” | 0 | - | 4-1 |
| 2 | “menu” | 0 | - | 4-2 |
| 3 | “scroll” | 1 | 4 | |
| 4 | “appre..” | 0 | - | |
| 5 | “haptic..” | 0 | - | |
| 6 | “status..” | 2 | 8 | 4-3 |
| 7 | “inver..” | 0 | - | 4-4 |
| 8 | “flash..” | 0 | - | 4-5 |
| 9 | “full b..” | 0 | - | 4-6 |
| 10 | “onlin..” | 0 | - | 4-7 |

Preference Page, id=4

| id | text | FCR | Next | Similar |
|---|---|---|---|---|
| 0 | “restart” | 0 | - | 3-0 |
| 1 | “back” | 0 | - | 3-1 |
| 2 | “menu” | 0 | - | 3-2 |
| 3 | “status..” | 2 | 8 | 3-6 |
| 4 | “inver..” | 0 | - | 3-7 |
| 5 | “flash..” | 0 | - | 3-8 |
| 6 | “full b..” | 0 | - | 3-9 |
| 7 | “onlin..” | 0 | - | 3-10 |
| 8 | “about” | 0 | - | |
| 9 | “about..” | 0 | - | |

**Fig. 5: Event Flow Graph Example**

use the first $N$ words to build the TXC feature. If the number of words on an widget is smaller than $N$, DinoDroid fills the vectors with all zeros to pad the words. As a result, the TXC feature is encoded as a matrix $L * N$, where $L$=400 and $N = 6$ by default. According to Pennington et al. [32], $L$ >300 has stable performance. We tried {300, 400} in our experiment and found no different results. We use L=400 because a larger size can better represent the text. We set N to be 6 because the word number on events is rarely up to 6 based on our observation.

Note that all configuration parameters (e.g., $K$, $L$, and $N$) can be adjusted to be larger values by users to achieve potentially better coverage, but at the cost of substantial amount of time in training and testing.

#### 3.2.2 Compacted Event Flow Graph

DinoDroid uses an event flow graph (EFG) to obtain features. The graph is represented by $G$ = ($V$, $E$). The set of vertices, $V$, represent events using their unique IDs. The set of edges, $E$, represent event transitions (i.e., from the selected event $e$ to all events in the next page) triggered by executing $e$. Each vertex records the three types of features described in Section 3.2.1. Fig. 5 shows an example, where each page is associated with a list of vertices and their features. The last label (“Similar”) will be discussed later in the section.

DinoDroid’s event flow graph is compacted in order to accommodate DQN. This is also the main difference from traditional EFGs [5] . In traditional EFG, whenever a new page is encountered, it will be added as a vertex to the EFG. However, if we use the vertices as states in DQN, it could cause the states to be huge or even unbounded [33], [34], [35], [36], [8]. Also, when encountering a similar page with a minor difference from an earlier page, if DinoDroid treats it as a new state, it could generate unbounded fake new events and thus waste exploration time. For example, in Fig. 5, there are many same events on the pages triggered by the “scroll” event. If we consider all of the events for each scroll triggered page, DinoDroid will waste time to execute the same events again and again without code coverage increase. Therefore, such similar pages should be combined to avoid this problem.

There have been techniques on compacting EFG for efficient exploration. For example, Stoat [8] ignores the details of ListView events by categorizing it into “empty” and “non-empty”, so it can merge two similar pages with only the ListView events are different into the same state. However, it may lose important information since some of the events triggered by the items under the ListView may be critical. In contrast, DinoDroid compacts the EFG by merging vertices instead of pages. Specifically, whenever a new page $P'$ is encountered, DinoDroid retrieves the pages with the same Android Activity ID [37] using UI Automator [38] from the existing EFG. It then compares the text of each vertex in $P'$ with the vertices of each of the retrieved pages. If the texts are the same, the two events are merged into a single vertex. The “Similar” tag in Fig. 5 records the information of the merged vertex (i.e., page id—event id). As such, when an event $e$ is executed, the feature vectors/matrices of both $e$’s vertex and its merged vertex are updated. Therefore, the same events will get an equal chance to be executed.

### 3.3 DinoDroid’s Deep Q-Networks

DinoDroid’s DQN agent examines the current state of the app and performs an action, i.e., selects the event with the largest Q value.

#### 3.3.1 DNN model

One of the key components in DinoDroid is a Deep Neural Network (DNN) model, which uses equation (1) to compute the Q value. The DNN takes the features of the event from last action $a_t$ as input and outputs the Q value, which is equal to $r_t + \gamma * \max_a Q(s_{t+1}, a)$. Comparing to regular RL using a Q-table, DNN is able to calculate Q values from any dimensions of features for each event. The DNN model involves feature handling and feature merging. The feature handling component employs a neural network algorithm to process each individual feature into a specific modality, represented by a one-dimension vector. The feature merging component combines the vectors of all features and passes them into the final fully connected layer [39] that is used to determine the event’s Q value. The purpose of the feature handling process is to help DNN make an accurate prediction.

**Feature handling.** DinoDroid employs specific DNN submodels to process individual features according to their types. For example, as shown in Fig. 6, the TXC feature is processed by a convolutional neural network (CNN) based

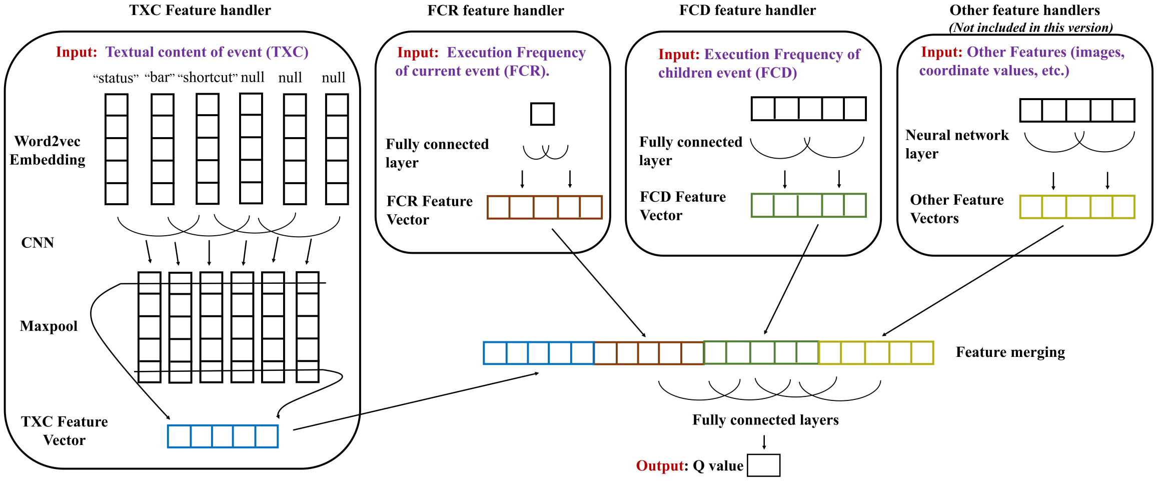


**Fig. 6: Deep Q-Network Model**

on natural language processing (NLP) and maxpool [40]. The FCR feature is handled by a fully connected layer [39], which is a common and efficient layer in neural network to process vectors. Other algorithms [41] could also be used but may involve additional cost.

DinoDroid's DNN model can be easily extended to process other features. For example, users can take image as another type of feature by leveraging existing image classification algorithms [42]. The DNN model accepts the formats of single value, vector, and matrix. By default, single value and vector are handled by a fully connected layer and matrix is handled by 1-D CNN [28].

**Feature merging.** Since the DNN model predicts a Q value based on all features, it combines the vectors generated by the feature handling components into a large one-dimensional vector. All input features to the DNN share the unique loss function described in Section 2.2.2. We leverage Keras deep learning architecture, which is capable of accepting mixed multiple inputs [43]. It then utilizes a three-fully connected layers [44] to process the large vector and predicts a Q value. The last layer with a linear activation to output just one value to represent the Q value. We use a stochastic gradient descent algorithm, Adam [45] to optimize the model with a learning rate 0.0001. The Adam optimization algorithm is an extension of stochastic gradient descent, which has been widely adopted for deep learning applications in computer vision and natural language processing [46], [47].

### 3.3.2 *Action Selection*

DinoDroid's DQN agent performs an action by selecting an event to execute from the current page. For editable events (e.g. text fields), DinoDroid automatically generates a random string with digit, char, and punctuation. DinoDroid maintains a Q value for each event. At the first $T$ iterations, DinoDroid executes random events to construct the DQN model. Randomness is necessary for an agent navigating through a stochastic maze to learn the optimal policy [48]. By default, $T$=20. After that, DinoDroid starts to use DNN to select an event.

To determine which event to execute on the current page, DinoDroid uses the $\varepsilon$-greedy policy [49], a widely adopted policy in reinforcement learning, to select the next event. DinoDroid selects the event with the highest Q value with probability $1 - \varepsilon$ and a random event with probability $\varepsilon$. The Q value is computed by the neural network model (Section 2.2.2). The value of $\varepsilon$ can be adjusted by users. By default $\varepsilon = 0.2$, which is the same as the value used in Q-testing [11]. In order to amplify the chance of bug detection, DinoDroid also generates system-level events at every 10 iterations, such as screen rotation, volume control, and phone calls [8].

### 3.3.3 *Reward Function*

In reinforcement learning, a reward is used to guide training and testing. DinoDroid's reward function is based on code coverage and bug detection. Specifically, DinoDroid sets the reward to a positive number (r=5) when the code coverage increases or revealing a unique crash and a negative number (r=-2) when the coverage does not change. To determine if a unique crash is revealed, we resort to the error message. In the log of an app execution, each crash is associated an accurate error message. If the same error message has happened before, we ignore it. The absolute value of positive reward is larger than that of the negative reward is because the intention is letting the machine favor higher coverage or detecting bugs. The reward values are configurable, however, these default values work well on different types of apps, as shown in the experiments.

Other reward functions, such as measuring the app page state changes may also be used [11], [10], [13], [18], [19]. For example, Q-testing [11] calculates the difference between the current page state and the states of recorded pages.

## 4 Evaluation

To evaluate DinoDroid, we consider three research questions:

**RQ1:** How does DinoDroid compare with the state-of-the-art Android testing tools in terms of code coverage?

**RQ2:** How does DinoDroid compare with the state-of-the-art Android testing tools in terms of bug detection?

**RQ3:** Can DinoDroid understand the features and learn a correct model?

### 4.1 Datasets

We need to prepare datasets for evaluating our approach. Since Sapienz [50] and Stoat [8] are two of the baseline tools that contain publicly available datasets to compare with, we used the datasets from their papers [2]. The datasets contains a total of 68 apps that falls into 18 domains [51]. We removed four apps because they crash right after launch on our Intel Atom(x86) emulator. The executable lines of code in the apps range from 109 to 22,208, indicating that they represent apps with different levels of complexity.

### 4.2 Implementation

We conducted our experiment on a four-core 3.60 GHZ CPU (with no GPU acceleration) physical x86 machine running with Ubuntu 16.04. With this machine, DinoDroid can send an event in 1.43 seconds on average. The dynamic exploration component is implemented on UI Automator [38]. The system events issued during testing are obtained from Androguard [52]. DinoDroid uses Emma [53] to obtain statement coverage. Keras [54] is used to build and run the deep neural network. The DQN agent is implemented by ourselves using Python.

### 4.3 Study Operation

We performed a two-fold cross validation by randomly dividing the whole 64 apps data set into two sets with each set containing 32 apps. It took 128 hours finish the experiment. An added fold would take about three more days. Specifically, the process randomly picked 32 apps as the training set and used the remaining 32 apps as the testing set. In the next fold, the first 32 apps are used in the testing set and the other 32 apps in the training set. Therefore, each app has a chance to be involved in both the training set and the testing set. The testing time of DinoDroid is set to one hour for each app, which is consistent with many other Android app testing tools [50], [12], [11], [19]. Given that RL has the tendency to exhibit high variability across runs, we repeat the testing for three times. The results are averaged over the 3 executions of each tool.

### 4.4 Comparison with Existing Tools

We compared DinoDroid with five existing Android testing tools: Monkey [3] , Stoat [8], Sapienz [50], and QBE [10], Q-testing [11]. Monkey is a random testing tool. Stoat employs model-based testing and pre-defined heuristic methods to explore the less unexecuted actions. Sapienz employs evolutionary testing. Both QBE and Q-testing use traditional Q-learning and QBE can also train and test on different sets of apps. The details of the tools are discussed in Section 7.

The testing time for all tools is set to one hour. Like DinoDroid, we repeated the experiment three times. We followed previous work [2], [11] to set 200 milliseconds delay between events for Monkey to avoid abnormal behaviors. Stoat [8] allocates one hour for model construction phase and two hours for Gibbs sampling phase. To use Stoat in our one-hour testing, we followed the suggestion proposed in [11], in which each phase of Stoat is set to 30 minutes for the best performance.

Note that while the training time of DinoDroid is extensive, we believe the cost will be diluted as the number of apps increases. If new apps are added to the training set, the existing model is directly updated without training from scratch. For the existing non-learning-based tools (e.g., Monkey, Sapienz), updating pre-defined rules often needs huge professional human effort.

## 5 Results and Analysis

### 5.1 RQ1: Code Coverage

Table 1 shows the results of code coverage obtained from DinoDroid and the other five tools on 64 apps. For each tool, we compute the average of the coverage scores collected in three test runs. The test results and log files can be found in our experiment dataset [22].

On average, DinoDroid achieves 48.9% line coverage, which is 22.8%, 17.8% 13.7%, 16.1%, and 18.6% more effective than Monkey, Q-testing, QBE, Stoat, and Sapienz, respectively. Specifically, DinoDroid achieved the highest coverage in 33 apps, compared to 16 apps in Sapienz, 9 in Stoat, 10 in Q-testing, 8 in QBE, and 5 in Monkey. The results indicate *DinoDroid is effective in achieving high coverage.*

We analyzed the app code covered by different tools. Taking the app "Nectroid" [55] as an example, many events reside in deep levels of EFG, which require a number of steps to explore. For instance, two widgets are associated with important functionalities: "add a new site" and "delete a site". Exercising the first widget requires 7 steps (launched page → menu → settings → select a site → new site → fill blankets → OK) and exercising the second widget requires 6 steps (launched page → menu → settings → select a site → Nectarine(long click) → delete → OK). Monkey, QBE, and Sapienz failed to cover the two particular sequences because they were not able to navigate the app to the deep levels. Q-testing failed to reach "delete a site". It finished "add a new site", but was not able to fill in any text information of the new site. Stoat was very close to reach "add a site", but at the last step, it selected "Cancel" instead of "OK". On the other hand, DinoDroid exercised the "OK" widget in 90% of pages that contain "OK", "Cancel", and a few other functional widgets during the learning process. We conjecture that this is because DinoDroid is able to learn that clicking "OK" widget is more likely to increase coverage.

### 5.2 RQ2: Bug Detection

Table 1 shows the number of unique bugs detected by the six tools (on the left of "/"). We reported the number of unique bugs detected in all three repeated runs. Like existing tools [8], [50], [11], we consider a bug to be a crash or an exception. The results showed that DinoDroid detected the largest number of bugs (269) compared to Monkey (73), Q-testing (54), QBE (59), Stoat (189), and Sapienz (63)). Specifically, DinoDroid detected most bugs in 38 apps, which is more effective than Monkey (5), Q-testing (0), QBE (3), Stoat (20), and Sapnize (2).

DinoDroid and Stoat detected a significantly larger number of bugs because they both use androguard [52] to issue system-level events to amplify the chance of bug detection (Section 3.3.2). When disabling system-level events, DinoDroid still detected the largest number of bugs (130), compared to Monkey (72), Q-testing (47), QBE (52), Stoat (78), and Sapienz (59). The numbers are reported in the right side of "/" in Table 1. Specifically, DinoDroid detected most number of bugs in 26 apps, which is more effective than Monkey (13), Q-testing (4), QBE (6), Stoat (11), and Sapnize (9). The above results suggest that *DinoDroid is effective in detecting bugs.*

Figure 7 shows the pairwise comparison of bug detection results between tools (with system-level events) in a single run. The black bars indicate the number of bugs detected in common and the grey bars indicate the number of bugs detected uniquely in each tool. For example, Stoat and DinoDroid detected 50 bugs in common. However, DinoDroid detected 37 bugs not detected by Stoat and Stoat detected 12 bugs not detected by DinoDroid.

## 5.3 RQ3: Understanding the Learned Model

RQ3 is used to understand whether the DQN agent can correctly learn the app behaviors based on the provided features. To do this, we analyzed the logs generated by the DQN agent for the 64 apps.

### *5.3.1 Understanding the Behaviors of Individual Features*

**FCR feature**. We first would like to understand the behavior of the FCR feature. Specifically, we would like to know whether DinoDroid will assign a higher priority to unexecuted events over executed events in the current page. We hereby computed the percentage of pages performing expected actions (i.e., triggering unexecuted events) among all pages. We consider only unexecuted and executed events instead of execution frequency because we need to silent the influence of children pages (i.e., the FCD feature) as unexecuted events do not invoke children pages. In total, there are 16,169 pages containing both executed and unexecuted events, with a total of 197,279 events; 85.2% of the pages performed expected actions. We then computed the probability of randomly selecting an event from each page and found that only 51.6% of the pages performed expected actions. The results show that DinoDroid indeed behaves better than a random approach.

**FCD feature**. We next would like to understand the behavior of the FCD feature. Specifically, we would like to know whether DinoDroid tends to select events with unexecuted children events over those with executed children events. To control the variation of event execution frequencies (FCR), we selected only pages that contain events with the same execution frequency from the traces. As a result, 495 pages with a total of 3,182 events were selected and 81.2 % of the pages performed expected actions (i.e., selecting events with unexecuted children events). When using a random selection, only 33.6% pages performed expected actions.

**TXC feature**. We also examined if DinoDroid is able to understand the content of events. To do this, we control the variations of FCR and FCD features by selecting pages that do not contain any executed events from the traces. A total of 3,060 pages were selected. We hypothesize that if Q values of these events are different (i.e., their textual content is different), DinoDroid is able to recognize the content of the widgets (i.e., perform expect actions). The results show that 91% of pages performed expected actions with the difference between highest Q value and lowest Q value greater than 10%.

The above results suggest that *the behaviors of the app features learned by DinoDroid are mostly expected.*

### *5.3.2 Understanding the Effectiveness of Individual Features*

We next explore how each of the three features affects the coverage effectiveness of DinoDroid. To achieve this, we repeated the experiment but disabled the FCR feature, FCD feature, and TXC feature separately. The results suggest that DinoDroid's average code coverage is higher than the coverage of the absence of FCR, FCD, and TXC by 0.7%, 2.4%, 3%, respectively.

We found the results surprising. Prior to the experiment, we expected the FCR feature to be the most crucial feature during testing because using event frequency to guide the exploration is intuitive and has been a common strategy used in existing work [8]. However, the actual result suggests that the textual feature (TXT) contributes more than the other features. Therefore, it is very important for a tester or a tool to understand the meaning of text.

### *5.3.3 The Whole DQN Model Behaviors*

The above experiment suggests how individual features affect the learning process of DinoDroid. We next conducted a deeper analysis to understand the behaviors of DinoDroid under the combination of the three features. To do this, we randomly selected a model from the two models learned from the 64 apps (by two cross-fold validation). We used this model to process the first page of the 32 apps, which were not used to train the model. We then manually set the values of FCR and TXC features while keeping the TXC feature as default and see how the model behaves.

As shown in Table 2, "FCR" indicates the execution frequency of the current event is executed and "FCD" indicates the three generations of children pages with the number of unexecuted events and the number of events being executed once (i.e., the first two elements in the feature vector). Each number in the table is the Q value averaged across all events with the same feature setting. For example, $<$(6#1); (1#1); (1#1)$>$ + "1" (the second row + the third column) indicates that when an event is executed once, the first generation of its children page contains six unexecuted events and one event is executed once, and the second and third generation of its children pages have one unexecuted events and one event is executed once.

DinoDroid learned the following behaviors as shown in Table 2. First, the Q value of (6#1);(1#1);(1#1) is larger than that of (1#6);(1#1);(1#1). This indicates that an event with more unexecuted children events in the first generation is more likely to be selected. Second, the Q value of (1#1);(6#1);(1#1) is larger than that of (1#1);(1#6);(1#1) and the Q value of (1#1); (1#1); (6#1) is larger than that of (1#1); (1#1); (1#6). This indicates the second and third generation (or perhaps the subsequence generations) of children events can also guide the exploration like the first generation. Third, the unexecuted event on the current page has the highest priority to be selected since its Q value is

**TABLE 1: Testing Result for Comparison**

| #APP. | # LOC | Code Coverage | | | | | | # Fault Triggered | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Mon. | Qt. | QBE. | St. | Sap. | DD. | Mon. | Qt. | QBE. | St. | Sap. | DD. |
| soundboard | 109 | 31 | 42 | 42 | 42 | 56 | 42 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 |
| gestures | 121 | 26 | 32 | 32 | 32 | 51 | 32 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 |
| fileexplorer | 126 | 31 | 40 | 40 | 40 | 31 | 40 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 |
| adsdroid | 236 | 13 | 40 | 29 | 29 | 33 | 25 | 3/3 | 3/3 | 0/0 | 5/2 | 3/2 | 3/0 |
| MunchLife | 254 | 65 | 63 | 65 | 65 | 71 | 66 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 |
| Amazed | 340 | 66 | 58 | 69 | 58 | 71 | 73 | 6/6 | 0/0 | 0/0 | 0/0 | 3/3 | 1/1 |
| battery | 342 | 72 | 71 | 69 | 71 | 54 | 73 | 0/0 | 0/0 | 0/0 | 3/3 | 1/1 | 9/9 |
| manpages | 385 | 58 | 58 | 62 | 48 | 67 | 63 | 0/0 | 0/0 | 0/0 | 3/0 | 0/0 | 3/0 |
| RandomMusicPlayer | 400 | 52 | 70 | 65 | 74 | 51 | 60 | 0/0 | 0/0 | 0/0 | 3/0 | 0/0 | 3/0 |
| AnyCut | 436 | 61 | 60 | 62 | 61 | 65 | 62 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 |
| autoanswer | 479 | 11 | 13 | 13 | 26 | 16 | 24 | 0/0 | 0/0 | 0/0 | 4/1 | 0/0 | 5/2 |
| LNM | 492 | 54 | 58 | 56 | 62 | 55 | 60 | 0/0 | 0/0 | 0/0 | 6/0 | 1/1 | 3/0 |
| baterydog | 556 | 62 | 55 | 62 | 54 | 67 | 62 | 1/1 | 0/0 | 0/0 | 1/1 | 0/0 | 0/0 |
| yahtzee | 597 | 38 | 10 | 43 | 49 | 43 | 57 | 1/1 | 0/0 | 3/3 | 3/0 | 0/0 | 6/3 |
| LolcatBuilder | 646 | 25 | 25 | 20 | 25 | 27 | 52 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 |
| CounterdownTimer | 650 | 58 | 60 | 69 | 77 | 45 | 77 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 |
| lockpatterngenerator | 669 | 78 | 69 | 75 | 66 | 79 | 78 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 |
| whohasmystuff | 729 | 46 | 65 | 66 | 67 | 32 | 74 | 3/3 | 0/0 | 0/0 | 3/0 | 0/0 | 3/0 |
| Translate | 799 | 45 | 44 | 43 | 40 | 48 | 47 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 |
| wikipedia | 809 | 25 | 27 | 26 | 27 | 29 | 27 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 6/3 |
| DivideAndConquer | 814 | 83 | 53 | 79 | 52 | 80 | 58 | 0/0 | 0/0 | 0/0 | 0/0 | 3/3 | 0/0 |
| zooborns | 817 | 34 | 20 | 29 | 35 | 36 | 35 | 0/0 | 3/3 | 0/0 | 4/1 | 0/0 | 3/0 |
| multismssender | 828 | 49 | 29 | 48 | 49 | 60 | 68 | 0/0 | 0/0 | 0/0 | 3/0 | 0/0 | 3/0 |
| Mirrored | 862 | 44 | 45 | 45 | 45 | 45 | 45 | 0/0 | 0/0 | 0/0 | 3/0 | 0/0 | 3/0 |
| myLock | 885 | 26 | 29 | 27 | 44 | 30 | 42 | 0/0 | 0/0 | 0/0 | 4/1 | 0/0 | 6/3 |
| aLogCat | 901 | 66 | 65 | 67 | 70 | 40 | 79 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 |
| aGrep | 928 | 45 | 38 | 49 | 37 | 54 | 57 | 1/0 | 3/3 | 9/6 | 2/2 | 3/3 | 9/6 |
| dialer2 | 978 | 37 | 36 | 38 | 34 | 35 | 46 | 0/0 | 0/0 | 0/0 | 3/0 | 1/1 | 3/0 |
| hndroid | 1038 | 7 | 10 | 11 | 9 | 8 | 9 | 3/3 | 3/3 | 2/2 | 3/3 | 3/3 | 4/3 |
| Bites | 1060 | 32 | 36 | 28 | 42 | 39 | 50 | 3/3 | 3/3 | 1/1 | 11/8 | 3/3 | 14/11 |
| tippy | 1083 | 82 | 75 | 60 | 74 | 80 | 83 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 |
| weight−chart | 1116 | 38 | 40 | 61 | 41 | 55 | 75 | 2/2 | 2/2 | 1/1 | 3/3 | 1/1 | 3/3 |
| importcontacts | 1139 | 40 | 41 | 63 | 37 | 42 | 41 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 |
| worldclock | 1242 | 89 | 92 | 92 | 92 | 90 | 89 | 0/0 | 0/0 | 0/0 | 3/0 | 0/0 | 3/0 |
| blokish | 1245 | 41 | 36 | 50 | 36 | 44 | 50 | 3/3 | 0/0 | 3/3 | 0/0 | 1/1 | 3/3 |
| aka | 1307 | 53 | 78 | 80 | 61 | 80 | 64 | 2/2 | 2/2 | 3/3 | 0/0 | 3/3 | 9/9 |
| Photostream | 1375 | 20 | 22 | 14 | 24 | 27 | 21 | 3/3 | 3/3 | 3/3 | 9/6 | 3/3 | 6/3 |
| dalvik−explorer | 1375 | 39 | 67 | 69 | 69 | 70 | 70 | 3/3 | 3/0 | 4/3 | 3/0 | 4/3 | 6/3 |
| tomdroid | 1519 | 47 | 54 | 50 | 52 | 50 | 53 | 0/0 | 0/0 | 0/0 | 1/1 | 0/0 | 3/3 |
| PasswordMaker | 1535 | 57 | 61 | 53 | 56 | 33 | 56 | 1/1 | 4/4 | 8/8 | 12/9 | 3/3 | 11/8 |
| frozenbubble | 1706 | 81 | 55 | 58 | 55 | 78 | 74 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 |
| aarddict | 2197 | 13 | 31 | 28 | 30 | 14 | 18 | 0/0 | 0/0 | 0/0 | 6/6 | 0/0 | 0/0 |
| swiftp | 2214 | 13 | 13 | 12 | 13 | 13 | 13 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 3/0 |
| netcounter | 2454 | 44 | 70 | 67 | 61 | 44 | 76 | 0/0 | 1/0 | 0/0 | 4/1 | 0/0 | 3/0 |
| alarmclock | 2491 | 64 | 67 | 64 | 67 | 37 | 70 | 3/3 | 1/1 | 0/0 | 8/2 | 3/3 | 7/1 |
| Nectroid | 2536 | 34 | 64 | 31 | 58 | 57 | 70 | 1/1 | 1/1 | 0/0 | 3/0 | 1/1 | 5/2 |
| QuickSettings | 2934 | 52 | 38 | 41 | 38 | 48 | 48 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 3/3 |
| MyExpenses | 2935 | 48 | 38 | 42 | 42 | 20 | 61 | 0/0 | 0/0 | 0/0 | 4/1 | 0/0 | 3/1 |
| a2dp | 3523 | 43 | 32 | 38 | 41 | 30 | 45 | 0/0 | 0/0 | 0/0 | 1/0 | 0/0 | 9/3 |
| mnv | 3673 | 22 | 44 | 34 | 43 | 8 | 45 | 3/3 | 2/2 | 3/3 | 3/3 | 1/1 | 6/3 |
| hotdeath | 3902 | 62 | 52 | 64 | 49 | 64 | 75 | 1/1 | 1/1 | 2/2 | 0/0 | 0/0 | 2/2 |
| SyncMyPix | 4104 | 21 | 20 | 20 | 25 | 21 | 26 | 0/0 | 0/0 | 0/0 | 3/0 | 0/0 | 3/0 |
| jamendo | 4430 | 21 | 16 | 24 | 14 | 23 | 26 | 1/1 | 1/1 | 2/2 | 7/3 | 0/0 | 9/3 |
| mileage | 4628 | 19 | 37 | 32 | 31 | 36 | 60 | 4/4 | 1/1 | 1/1 | 10/4 | 3/3 | 12/5 |
| sanity | 4840 | 19 | 25 | 16 | 22 | 15 | 34 | 2/2 | 3/0 | 2/0 | 2/0 | 4/1 | 5/2 |
| fantastichmemo | 8419 | 24 | 39 | 32 | 22 | 28 | 43 | 3/3 | 4/4 | 1/1 | 5/2 | 1/1 | 9/3 |
| anymemo | 8428 | 30 | 41 | 30 | 30 | 27 | 43 | 1/1 | 2/2 | 0/0 | 4/1 | 1/1 | 10/6 |
| Book−Catalogue | 9857 | 34 | 37 | 25 | 11 | 24 | 32 | 3/3 | 1/1 | 4/3 | 3/0 | 1/1 | 10/7 |
| Wordpress | 10100 | 5 | 3 | 4 | 5 | 4 | 8 | 0/0 | 0/0 | 0/0 | 9/6 | 0/0 | 10/4 |
| passwordmanager | 10833 | 10 | 3 | 5 | 7 | 4 | 8 | 1/1 | 0/0 | 0/0 | 0/0 | 0/0 | 2/0 |
| aagtl | 11724 | 15 | 18 | 16 | 16 | 17 | 17 | 5/5 | 4/4 | 3/3 | 4/4 | 6/6 | 4/4 |
| morphoss | 17148 | 12 | 17 | 19 | 18 | 11 | 25 | 4/4 | 0/0 | 1/1 | 6/3 | 2/2 | 10/4 |
| addi | 19945 | 14 | 16 | 18 | 17 | 19 | 18 | 6/6 | 3/3 | 3/3 | 6/3 | 3/3 | 6/3 |
| k9mail | 22208 | 5 | 7 | 11 | 7 | 5 | 7 | 0/0 | 0/0 | 0/0 | 6/1 | 1/1 | 7/1 |
| Overall | | 39.8 | 41.5 | 43 | 42.1 | 41.2 | 48.9 | 73/72 | 54/47 | 59/52 | 189/78 | 63/59 | 269/130 |

DD.= DinoDroid. St.= Stoat Mon.= Monkey Sap.= Sapienz Qt.=Q-testing ”a/b” indicates a=#crashes for all and b=#crashes without system level events

**TABLE 2: DinoDroid's behavior on every feature combination**

| FCD \ FCR | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| <(6#1), (1#1), (1#1)> | - | 3.45 | 2.32 | 1.2 | -0.65 | -0.47 |
| <(1#6); (1#1); (1#1)> | - | 2.57 | 0.23 | -0.171 | -3.35 | -3.55 |
| <(1#1); (6#1); (1#1)> | - | 3.44 | 2.03 | 0.16 | 0.19 | 0.00 |
| <(1#1); (1#6); (1#1)> | - | 1.01 | 0.92 | -0.31 | -3.59 | -3.96 |
| <(1#1); (1#1); (6#1)> | - | 1.92 | 0.97 | 1.07 | -0.9 | -3.2 |
| <(1#1); (1#1); (1#6)> | - | 1.09 | 0.81 | 0.21 | -3.3 | -4.53 |
| <(1#1); (1#1); (1#1)> | - | 1.01 | 0.78 | -1.66 | -3.86 | -4.32 |
| <(0#0); (0#0); (0#0)> | 10.96 | -4.42 | -4.98 | -5.09 | -5.13 | -5.14 |

<(A#B); (C#D); (E#F)>: the first generation has $A$ unexecuted events and $B$ events are executed once; the second generation has $C$ unexecuted events and $D$ events are executed once; the third generation has $E$ unexecuted events and $F$ events are executed once.

significantly larger (i.e., FCR=0). Fourth, with the same FCD feature values, the event with less execution frequency on the current page has a higher priority to be selected since the Q value is decreasing as the value of FCR increases.

All of the above behaviors align with the intuition on *how human would test apps* with the three types of features and demonstrate that DinoDroid's DQN can *automatically learn these behaviors without the need to manually set heuristic rules.*

## 6 Limitations and Threats to Validity

DinoDroid has several limitations. First, DinoDroid currently considers three features. As part of the future work,

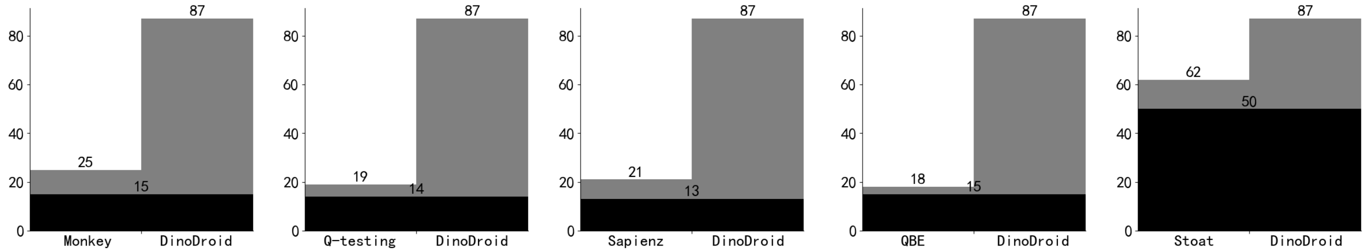


**Fig. 7: Comparison of tools in detecting crashes**

we will assess whether other features, such as images of widgets and the execution of sequences of actions that compose complete use cases, can improve the performance of DinoDroid. Second, DinoDroid handles feature values represented by matrices or vectors. However, some features may need more complex representations (e.g., multiple complex matrices) and processing them may take substantial amount of time. Third, in the current version, DinoDroid measures line coverage to train the apps, so we focus on open-source apps. DinoDroid can be extended to handle closed-source apps by collecting method-level coverage or using non-code-related rewards. Fourth, for a fair comparison, DinoDroid does not utilize extra advantageous features (e.g. login script, short message records, contact records) in its current version, because none of the state-of-the-art tools have such features by default. It is also very challenging to modify the tools to implement these features because we cannot control their installation of apps and do not know when to run the login scripts. Nevertheless, we believe that adding the extra functionality might increase the effectiveness of coverage and bug detection in DinoDroid and the existing tools. For example, after we added the login function in DinoDroid, k9 mail's code coverage was increased to 40% in one hour, compared with 7% coverage without the login function. Su et al. [56] implemented the login function and resolved some usability issues for the six tools in their study, but the apps in our benchmarks are emma instrumented and thus incompatible with their tools, which are jacoco instrumented. As part of future work, we will implement the extra advantageous features in DinoDroid, as well as in the state-of-the-art tools in our study. We will also compare DinoDroid with more existing tools, such as those studied in [56]. Fifth, the current version of our approach uses a fixed reward. Existing work (Q-testing [11]) has designed an adaptive reward function to exploit a neural network to calculated the difference between two states (app pages) to determine the reward value. We believe that our approach can work with the adaptive reward function by simply using Q-testing's trained model to determine the reward value.

The primary threat to the external validity of this work involves the representativeness of the apps used in our study. Other apps may exhibit different behaviors. We reduce this threat to some extent by using a set of apps developed and released by prior research work [2], which has been extensively used by existing Android testing research [57], [12]. These apps were also used by the baseline tools (i.e., Stoat [8], Sapienz [50]) in our study.

The primary threat to internal validity involves potential errors in the implementations of DinoDroid. We control this threat by testing our tools extensively and verifying their results against a small-scale program for which we can manually determine the correct results.

## 7 Related Work

There have been a number of techniques on automated Android GUI testing. Random testing [3] uses random strategies to generate events. Because of its simplicity and availability, it can send thousands of events per second to the apps and can thus get high code coverage. However, the generated events may be largely redundant and ineffective. DynoDroid [4] improved random testing by exploring the app in a manner that can avoid testing redundant widgets. But it may still be ineffective in reaching functionalities involving deep levels of the app due to randomness. Sapienz [50] uses multi-objective search-based testing to maximize coverage and fault revelation at the same time to optimize test sequences and minimize length. TimeMachine [58] enhances random testing with the ability to jump to a state in the past.

Model-based [5], [6], [7], [33], [59], [60], [61], [62] build and use a GUI model of the app to generate tests. The models are usually represented by finite state machines to model app states and their transitions. For example, Stoat [8] utilizes a stochastic Finite State Machine model to describe the behavior of app under test. Section 2.2 discusses Stoat. Unlike model-based testing, DinoDroid does not need to model the app behaviors. Instead, it can automatically learn app behaviors by deep Q-networks.

Systematic testing tools, such as symbolic execution [63], [64], aims to generate test cases to cover some code that is hard to reach. While symbolic execution may be able to exercise functionalities that are hard to reach, it is less scalable and not effective in code coverage or bug detection.

Machine learning techniques have also been used in testing Android apps. These techniques can be classified into two categories [11]. The techniques in the first category typically have an explicit training process to learn knowledge from existing apps then apply the learned experience on new apps [10], [15], [12], [65], [13]. For example, QBE [10] learns how to test android apps in a training set by a Q-learning. As discussed in Section 2.4, QBE is not able to capture fine-grained app behaviors due to the limitations of Q-learning.

In addition to QBE, Degott et al. [15] use learning to identify valid interactions for a GUI widget (e.g., whether a widget accepts interactions). It then uses this information to guide app exploration. Their following work [13] uses MBA reinforcement learning to guide the exploration based on the abstracted features: valid interactions and invalid interactions. However, like QBE, these techniques can not handle complex app features because of the high-level abstraction of state information.

Humanoid [12] employs deep learning to train a model from labeled human-generated interaction traces and uses the model together with a set of heuristic rules to guide the exploration of new apps. The result of deep learning is only used to select one event from the unexplored events on the current page. If all events on the current page are explored, the heuristic rules will be used to compute the shortest path in a graph to find a page with unexplored events. In contrast, DinoDroid does not need to manually build/label the training set or use any pre-defined search strategies or rules to guide testing. AppFlow [66] generates GUI tests from high-level, manually written test cases. It leverages machine learning to match screens and widgets to individual tests. Juha Eskonen et al. [67] employs deep reinforcement learning to perform UI testing for web applications. This work specifically focuses on the image feature and does not perform an an empirical study to assess its performance. In contrast, DinoDroid's feature merging component is able to handle multiple complex features. DinoDroid targets at Android apps and performs an extensive empirical study.

The techniques in the second category do not have explicit training process [18], [16], [19], [20], [11], [68]. Instead, they use Q-learning to guide the exploration of individual apps, where each app generates a unique behavior model. These techniques share the same limitations with Q-learning, where complex features cannot be maintained in the Q-table. For example, QDROID [16], DRIFT [69], and ARES [70] design the deep neural network agents to guide the exploration of Android apps. However, like QBE, the widgets are abstracted into several categories or boolean values. The abstraction of the states and actions may cause the loss of information in individual widgets. Wuji [71] combines reinforcement learning with evolutionary algorithms to test Android game apps. It designs a unique state vector for each game used as the position of the player character or the health points. Therefore, Wuji can not transfer the learned knowledge to new apps. Q-testing [11]'s main contribution is the design of a reward function to improve the performance of Q-testing by calculating the difference between the current state and the recorded states. Our work is orthogonal to Q-testing and one can improve the performance of DinoDroid by utilizing Q-testing's reward function.

## 8 Conclusions

We have presented DinoDroid, an automated approach to test Android application. It is a deep Q-learning-based approach, which can learn how to test android application rather than depending on heuristic rules. DinoDroid can take more complex features than existing learning-based methods as the input by using a Deep Q learning-based structure. With these features, DinoDroid can process complex features on the pages of an app. Based on these complex features, DinoDroid can learn a policy targeting at achieving high code coverage. We have evaluated DinoDroid on 64 apps from a widely used benchmark and showed that DinoDroid outperforms the state-of-the-art and state-of-practice Android GUI testing tools in both code coverage and bug detection. By analyzing the testing traces of the 64 apps, we are also able to tell that the machine really understands the features and provides a sensible strategy to generate tests.

## Acknowledgments

This research is supported in part by the NSF grant CCF-1652149.

## References

[1] "Google play data," https://en.wikipedia.org/wiki/Google_Play.

[2] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?" in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2015, pp. 429–440.

[3] "Ui/application exerciser monkey," https://developer.android.com/studio/test/monkey.html.

[4] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the Joint Meeting on Foundations of Software Engineering*, 2013, pp. 224–234.

[5] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the International Conference on Automated Software Engineering*, 2012, pp. 258–261.

[6] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "Mobiguitar: Automated model-based testing of mobile apps," *IEEE software*, vol. 32, no. 5, pp. 53–59, 2015.

[7] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Acm Sigplan Notices*, 2013, pp. 641–660.

[8] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the Joint Meeting on Foundations of Software Engineering*, 2017, pp. 245–256.

[9] S. Chib and E. Greenberg, "Understanding the metropolis-hastings algorithm," *The american statistician*, vol. 49, no. 4, pp. 327–335, 1995.

[10] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, and Y. Donmez, "QBE: Qlearning-based exploration of android applications," in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2018, pp. 105–115.

[11] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2020, pp. 153–164.

[12] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: a deep learning-based approach to automated black-box android app testing," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1070–1073.

[13] C. Degott, N. P. Borges Jr, and A. Zeller, "Learning user interface element interactions," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2019, pp. 296–306.

[14] M. M. Hing, A. van Harten, and P. Schuur, "Reinforcement learning versus heuristics for order acceptance on a single resource," *Journal of Heuristics*, vol. 13, no. 2, pp. 167–187, 2007.

[15] N. P. Borges, M. Gómez, and A. Zeller, "Guiding app testing with mined interaction models," in *Proceedings of the International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2018, pp. 133–143.

[16] T. A. T. Vuong and S. Takada, "Semantic analysis for deep q-network in android gui testing." in *SEKE*, 2019, pp. 123–170.

[17] C. Jin, Z. Allen-Zhu, S. Bubeck, and M. I. Jordan, "Is q-learning provably efficient?" *arXiv preprint arXiv:1807.03765*, 2018.

[18] D. Adamo, M. K. Khan, S. Koppula, and R. Bryce, "Reinforcement learning for android gui testing," in *Proceedings of the International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018, pp. 2–8.

[19] T. A. T. Vuong and S. Takada, "A reinforcement learning based approach to automated testing of android applications," in *Proceedings of the International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018, pp. 31–37.

[20] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro, "Autoblacktest: Automatic black-box testing of interactive applications," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2012, pp. 81–90.

[21] H. Larochelle, Y. Bengio, J. Louradour, and P. Lamblin, "Exploring strategies for training deep neural networks." *Journal of machine learning research*, vol. 10, no. 1, 2009.

[22] "Dinodroid," https://github.com/softwareTesting123/DinoDroid.

[23] "lockpatterngenerator," https://github.com/dharmik/lockpatterngenerator.

[24] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[25] R. Bellman, "On the theory of dynamic programming," *National Academy of Sciences of the United States of America*, vol. 38, no. 8, p. 716, 1952.

[26] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[27] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.

[28] "Conv1d layer," https://keras.io/api/layers/convolution_layers/convolution1d.

[29] P. Dayan, "Improving generalization for temporal difference learning: The successor representation," *Neural Computation*, vol. 5, no. 4, pp. 613–624, 1993.

[30] O. Levy, Y. Goldberg, and I. Dagan, "Improving distributional similarity with lessons learned from word embeddings," *Transactions of the Association for Computational Linguistics*, vol. 3, pp. 211–225, 2015.

[31] "word2vec," https://github.com/dav/word2vec.

[32] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[33] Y.-M. Baek and D.-H. Bae, "Automated model-based android gui testing using multi-level gui comparison criteria," in *Proceedings of the International Conference on Automated Software Engineering*, 2016, pp. 238–249.

[34] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," in *Acm Sigplan Notices*. ACM, 2013, pp. 623–640.

[35] A. Memon, I. Banerjee, and A. Nagarajan, "Gui ripping: Reverse engineering of graphical user interfaces for testing," in *Proceedings of the Working Conference on Reverse Engineering*, 2003, pp. 260–269.

[36] M. Pradel, P. Schuh, G. Necula, and K. Sen, "Eventbreak: Analyzing the responsiveness of user interfaces through performance-guided test generation," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 33–47, 2014.

[37] "Android activity," https://developer.android.com/reference/android/app/Activity.

[38] "Ui automator," https://developer.android.com/training/testing/ui-automator.

[39] L. Wan, M. Zeiler, S. Zhang, Y. Le Cun, and R. Fergus, "Regularization of neural networks using dropconnect," in *International conference on machine learning*. PMLR, 2013, pp. 1058–1066.

[40] Y. Kim, "Convolutional neural networks for sentence classification," *arXiv preprint arXiv:1408.5882*, 2014.

[41] D. Justus, J. Brennan, S. Bonner, and A. S. McGough, "Predicting the computational cost of deep learning models," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 3873–3882.

[42] J. Wang, Y. Yang, J. Mao, Z. Huang, C. Huang, and W. Xu, "Cnn-rnn: A unified framework for multi-label image classification," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2285–2294.

[43] Q. Gao, S. Lim, and X. Jia, "Hyperspectral image classification using convolutional neural networks and multiple feature learning," *Remote Sensing*, vol. 10, no. 2, p. 299, 2018.

[44] "Dense layer," https://keras.io/api/layers/core_layers/dense/.

[45] "adam," https://keras.io/api/optimizers/adam/.

[46] M. N. Y. Ali, M. G. Sarowar, M. L. Rahman, J. Chaki, N. Dey, and J. M. R. Tavares, "Adam deep learning with som for human sentiment classification," *International Journal of Ambient Computing and Intelligence*, vol. 10, no. 3, pp. 92–116, 2019.

[47] K. Gregor, I. Danihelka, A. Graves, D. Rezende, and D. Wierstra, "Draw: A recurrent neural network for image generation," in *Proceedings of the International Conference on Machine Learning*, 2015, pp. 1462–1471.

[48] A. D. Tijsma, M. M. Drugan, and M. A. Wiering, "Comparing exploration strategies for q-learning in random stochastic mazes," in *Proceedings of the Symposium Series on Computational Intelligence (SSCI)*, 2016, pp. 1–8.

[49] M. Tokic and G. Palm, "Value-difference based exploration: adaptive control between epsilon-greedy and softmax," in *Proceedings of the Annual Conference on Artificial Intelligence*, 2011, pp. 335–346.

[50] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2016, pp. 94–105.

[51] "Androtest data set," http://bear.cc.gatech.edu/ shauvik/androtest/.

[52] "Androguard," https://github.com/androguard/androguard.

[53] "Emma," http://emma.sourceforge.net/.

[54] "Keras," https://keras.io/.

[55] "nectroid," https://github.com/cknave/nectroid.

[56] T. Su, J. Wang, and Z. Su, "Benchmarking automated gui testing for android against real-world bugs," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 119–130.

[57] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically discovering, reporting and reproducing android application crashes," in *2016 IEEE international conference on software testing, verification and validation (icst)*. IEEE, 2016, pp. 33–44.

[58] Z. Dong, M. Bohme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *Proceedings of the International Conference on Software Engineering*. IEEE, 2020, pp. 481–492.

[59] T. Gu, C. Cao, T. Liu, C. Sun, J. Deng, X. Ma, and J. Lü, "Aimdroid: Activity-insulated multi-level automated testing for android applications," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 103–114.

[60] J. Yan, L. Pan, Y. Li, J. Yan, and J. Zhang, "Land: a user-friendly and customizable test generation tool for android apps," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2018, pp. 360–363.

[61] J. Yan, T. Wu, J. Yan, and J. Zhang, "Widget-sensitive and back-stack-aware gui exploration for testing android apps," in *Proceedings of the International Conference on Software Quality, Reliability and Security (QRS)*, 2017, pp. 42–53.

[62] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *International Conference on Fundamental Approaches to Software Engineering*, 2013, pp. 250–265.

[63] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of android apps," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2014, pp. 599–609.

[64] X. Gao, S. H. Tan, Z. Dong, and A. Roychoudhury, "Android testing via synthetic symbolic execution," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2018, pp. 419–429.

[65] J.-W. Lin, R. Jabbarvand, and S. Malek, "Test transfer across mobile apps through semantic mapping," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2019, pp. 42–53.

[66] G. Hu, L. Zhu, and J. Yang, "Appflow: using machine learning to synthesize robust, reusable ui tests," in *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 269–282.

[67] J. Eskonen, J. Kahles, and J. Reijonen, "Automating gui testing with image-based deep reinforcement learning," in *Proceedings of the International Conference on Autonomic Computing and Self-Organizing Systems*. IEEE, 2020, pp. 160–167.

[68] Y. Koroglu and A. Sen, "Functional test generation from ui test scenarios using reinforcement learning for android applications," *Software Testing, Verification and Reliability*, vol. 31, no. 3, p. e1752, 2021.

[69] L. Harries, R. S. Clarke, T. Chapman, S. V. Nallamalli, L. Ozgur, S. Jain, A. Leung, S. Lim, A. Dietrich, J. M. Hernández-Lobato *et al.*, “Drift: Deep reinforcement learning for functional software testing,” *arXiv preprint arXiv:2007.08220*, 2020.

[70] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella, “Deep reinforcement learning for black-box testing of android apps,” *arXiv preprint arXiv:2101.02636*, 2021.

[71] Y. Zheng, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, Y. Chen, and C. Fan, “Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning,” in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2019, pp. 772–784.