Semantic Analysis for Deep Q-Network in Android GUI Testing

Tuyet Vuong*, Shingo Takada*

*Dept. of Information and Computer Science, Keio University, Yokohama, Japan

{tuyet, michigan}@doi.ics.keio.ac.jp

***Abstract*— Since the big boom of smartphone and consequently of mobile applications, developers nowadays have many tools to help them create applications easier and faster. However, efficient automated testing tools are still missing, especially for GUI testing. We propose an automated GUI testing tool for Android applications using Deep Q-Network and semantic analysis of the GUI. We identify the semantic meanings of GUI elements and use them as an input to a neural network, which through training, approximates the behavioral model of the application under test. The neural network is trained using the Q-Learning algorithm of Reinforcement Learning. It guides the testing tool to explore more often functionalities that can only be accessed through a specific sequence of actions. The tool does not require access to the source code of the application under test. It obtains higher code coverage and is better at fault detection in comparison to state-of-the-art testing tools.**

Keywords: Automated Android Testing; GUI Testing; Reinforcement Learning, Deep Q-network

I. INTRODUCTION

In 2017, the number of monthly active Android devices reached a new milestone of 2 billion [1], accompanied by about 3 million applications in the Google Play Store [2]. The mobile application market is on-demand and so competitive that a small bug in the software can cause users to uninstall the product and opt for another. Assuring application quality is hence very important. Even though a lot of effort has been made to develop automated testing techniques, we still heavily rely on manual testing in practice, with only 3% of developers fully engaging in automation [3]. Developers also admitted in recent surveys that Graphical User Interface (GUI) testing is especially labor-intensive, time-consuming, and challenging to automate because GUI tests must be constantly rewritten when changes, even small, are made [3].

Android application's GUIs contain various types of *components* such as button, text input, slide bar, switch, etc. Moreover, users can interact with each type of component in multiple ways through *events*: click, long click, swipe, scroll, etc. This complexity makes GUI testing techniques like random testing inefficient because it attributes a uniform probability distribution to all combinations of components and events, while an efficient testing strategy should select and follow the specific paths that reveal application's functionalities.

DOI reference number: 10.18293/SEKE2019-080.

In our previous work [4], we presented an automated testing tool implementing the classic Q-Learning algorithm of reinforcement learning, which demonstrated positive improvements in code coverage. In this paper, we continue to follow the reinforcement learning approach and propose two main contributions:

- Improve the learning of the application's behavioral model by analyzing the semantic representations of GUI components.
- Use Deep Q-Network [5] to approximate the behavioral model of the application under test.

The remainder of this paper is organized as follows: Section II reviews related work in Android automated testing and the application of reinforcement learning in software testing. Section III provides a brief introduction to Deep Q-Network. Section IV elaborates our proposed approach and implementation details. Section V analyzes the evaluation results and finally section VI concludes the paper along with suggestions for future works.

II. RELATED WORK

In recent years, researchers have tried different approaches to test Android applications, among the most popular trends are random testing, model-based testing, and heuristic-based testing. Tools such as Puma [6] and A3E [7] build a model of the application under test then systematically explore the application based on this model. On the other hand, Evodroid [8] and Acteve [9] use special algorithms such as symbolic execution and evolutionary algorithms to test the application. As our paper proposes a black-box GUI testing method, we investigate further in black-box GUI testing tools.

Monkey [10] is a random event generator which is embedded in the Android Development Toolkit. It is therefore commonly used thanks to its simplicity and availability. It performs tests by sending thousands of events per second to the application and usually obtains high code coverage in comparison to other testing tools [11]. Despite the high code coverage, faults discovered by Monkey are hard to locate because tests are hard to reproduce and closer to stress tests than functionality tests. Dynodroid [12] improves random testing strategy by analyzing the context of the application then executes the most relevant event at each step (RandomBiasedStrategy). Developers can provide inputs such as authentication information beforehand to the tool to

unblock some steps. Dynodroid is also capable of generating system events by analyzing the listeners of the application.

Reinforcement Learning was used in software testing in the past and has shown its ability to improve random exploration strategy. Mariani et al. proposed a tool called AutoBlackTest [13], a black-box GUI testing tool for Java desktop software. The tool uses Q-Learning algorithm to build a behavioral model of the software, represented as a multi-directional graph. Using this behavioral model, the tool can plan its exploration route in order to get to hard-to-reach GUIs (which can only be reached through a specific sequence of actions). TESTAR [14] also uses Q-Learning to generate test sequences based on GUI. The Q-Learning algorithm was proven to be beneficial, provided that we choose an adequate set of parameters.

There are currently two main approaches when applying reinforcement learning to Android testing. The first approach is proposed by Koroglu, et al. [15], where they train a single matrix across multiple apps using random exploration then use it to test other applications. The Q-value distribution matrix is trained for two objectives: increasing activity coverage and crash detection. Overall, their tool obtained higher activity coverage and number of distinct crashes compared to other state-of-the-art testing tools. The second approach is presented in our previous work [4] and the work of D. Adamo et. al [16], where a unique set of Q-value is calculated for each application.

III. Deep Q-Network

Reinforcement Learning (RL) is a field in Artificial Intelligence where an agent learns to behave optimally in its environment through trial-and-error interactions [17], step by step. At each time step t, it observes the state s_t of the environment and takes an action a_t based on its policy π. The environment then transitions to a new state s_{t+1} based on s_t and a_t. It also outputs a scalar reward r_{t+1} as feedback that the agent then uses to update its knowledge. The goal of the agent is to learn a policy π^* that maximizes the expected cumulative reward of a sequence of actions in the environment.

The reinforcement learning problem can be formulated as a Markov decision process [17], defined by:

- A set of possible states: S
- A set of possible actions: A
- A reward function for the next state given a (state, action) pair: $R(s_t, a_t, s_{t+1})$
- A transition probability i.e distribution over the next state given a (state, action) pair: $T(s_{t+1}|s_t, a_t)$
- A discount factor $\gamma \in [0, 1]$, where lower γ emphasizes more on immediate rewards.

There are two major directions in solving RL problems: algorithms based on value functions and algorithms based on policy search [18]. Q-Learning falls into the first category. For each policy π, we define an action-value function or quality function (Q-function). The value $Q^\pi(s_t, a_t)$ is the expected cumulative reward that can be achieved by executing a sequence of actions that starts with an action a_t from a state s_t and then follows the policy π. The optimal Q-function Q^* is the maximum expected cumulative reward achievable for a given (state, action) pair, over all possible policies.

$$Q^*(s_t, a_t) = \max_\pi \sum_{t>0} (\gamma^t r_t | s = s_t, a = a_t, \pi) \quad (1)$$

Intuitively, if the optimal quality function Q^* is known, at each step s_t, the optimal strategy is to take the action that maximizes the sum: $r + \gamma Q^*(s_{t+1}, a_{t+1})$ where r is the immediate reward of the current step. This is known as the Bellman equation in dynamic programming [19]:

$$Q^*(s_t, a_t) = R(s_t, a_t) + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \quad (2)$$

The Q-learning algorithm uses equation (2) to estimate the value of Q^* iteratively. The Q-function is initialized with a default value. Every time the agent executes an action a_t from state s_t to reach state s_{t+1} and receives a reward r_{t+1}, the Q-function is updated as:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \quad (3)$$

In this formula, $\alpha \in [0, 1]$ is the learning rate and regulates the impact of a new observation on the estimated values. The Q-learning algorithm is guaranteed to converge to Q^* if applied to a Markovian environment, with a bounded immediate reward and with state-action pairs continually updated [20].

The main motivation for the birth of Deep Reinforcement Learning (DRL) is to scale the classic RL problems to more complex state and action spaces [5] [18]. In brief, DRL trains a neural network to approximate the optimal policy and/or optimal value functions. In the case of Q-Learning, the state-value function Q is estimated by a Deep Q-Network (DQN) with weight w:

$$Q(s_t, a_t, w) \approx Q^\pi(s_t, a_t) \quad (4)$$

The weight of the DQN is updated based on a loss function defined as:

$$L(w) = (r_t + \gamma \max_a Q(s_{t+1}, a, w) - Q(s_t, a_t, w))^2 \quad (5)$$

which leads to the following Q-Learning gradient:

$$(r_t + \gamma \max_a Q(s_{t+1}, a, w) - Q(s_t, a_t, w)) \frac{\partial Q(s_t, a_t, w)}{\partial w} \quad (6)$$

The stochastic gradient descent method can then be used to minimize the loss $L(w)$ and the Q-Network will gradually converge toward the optimal Q-function Q^*. In the next section, we explain how we use Deep Q-Network to incorporate the semantic representations of GUI states to the reinforcement learning algorithm.

IV. Proposed Approach: Semantic Analysis of GUI as an Input for Deep Q-Network in Android GUI Testing.

In our previous work [4] we used Q-Learning algorithm as an exploration strategy to test Android applications. We

dynamically built a behavioral model of the application under test while interacting with it. With classic Q-Learning, we calculated and constantly updated a dictionary holding the Q-value of each pair of (state, event). Even though the experiment demonstrated positive results, we noticed two main weaknesses in our model:

- The purpose of the reinforcement learning algorithm is to guide the exploration toward revealing and testing the application's hard-to-reach functionalities. However, a path that reveals the application's functionalities should consider the semantic meaning of the components upon which we take action. Take the example of an alarm clock application: the sequence of actions that reveals the functionality *Change timezone* should be described as *click on a menu component - scroll down the list - click on setting - click on change timezone button*, instead of simply *click - scroll - click - click*. The latter sequence of events would have very different outcomes when being executed on different sets of components.
- The algorithm cannot scale well when the number of states, GUI components in each state, and events increase. Mobile applications are becoming more and more complex and the Android OS can now support more and more gestures (events). Our testing tool should be able to handle large and complex GUIs.

The first point is addressed by a semantic analysis of GUI components that we elaborate in the following section. As for the second point, we propose using Deep Q-network to potentially solve the complexity problem of reinforcement learning.

A. Semantic classification of Android GUI components

GUI testing tools usually interact with the application under test by sending events to UI components but few of them have considered the meaning of the components they interact with. QBE [15] took some initiatives by separating the actions on hard buttons of the phone from the on-screen events (click, long-click, etc). However, they didn't consider any semantic meaning of the GUI components on the screen. In our approach, we use the semantic information of GUI components as an input to guide the exploration.

Recent works of T.F. Liu et. al presented a guideline to identify the semantic meaning of mobile app GUI [21]. They established a lexical database of 25 types of UI components, 197 text button concepts and 135 icon classes. We're interested in the classification of UI components in particular, where components are separated into groups such as Input, List Item, Toolbar, Background Image, etc. They employed a code-based heuristics approach: using a lexical database, they classified a component by examining its Android class and the classes of its parent components.

In our testing tool, we use the same method to classify components. However, instead of classifying all visible components, we only classify actionable components, which are either clickable, long-clickable, scrollable or checkable. If a parent component is actionable, then all of its children nodes are also actionable of the same type. We reduce the number of component types down to four main groups: Navigation, Input, Button, and List, as these group can cover the majority of actionable components [21]. The reference for classifying each group is presented in Table I, inspired by the one provided by T.F. Liu et. al.

TABLE I
CLASSIFICATION OF UI COMPONENTS

Group	Class name
Input	EditText, SearchBox, AutoCompleteTextView, AutoSuggestView, Field, Input, CheckBox, DatePicker, RadioButton, CheckedTextView, Switch, SeekBar
Navigation	Toolbar, TitleBar,ActionBar, Menu, Navigation, SideBar, Drawer, AppBar, TabWidget
List	ListItem, ListView, RecyclerView, ListPopUpWindow, GridView, GroupView
Button	Button, GlyphView, TextView, ImageView

We identify UI components in Navigation and List group by their parent nodes. In other words, once we identify a List or Navigation component by its class name, all of its children fall under the same group respectively. The Input group should be understood in a large sense, including all components that receive information from the user. Finally, only actionable TextView and actionable ImageView are classified as Button (Text Button and Image Button respectively).

B. Deep Q-Network as an Android Testing Tool

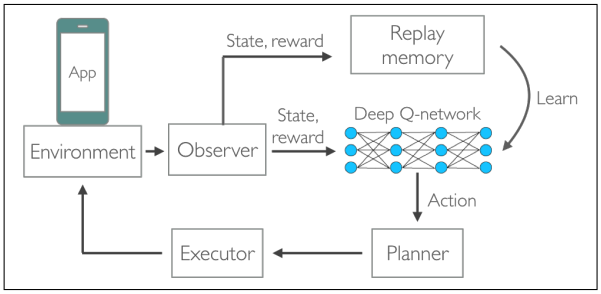

Fig. 1. The overall architecture

Our testing tool QDroid, whose architecture is presented in Figure 1, consists of 5 main modules: Environment, Observer, Deep Q-Network (including a replay memory), Planner, and Executor. QDroid interacts with the application under test step by step in order to estimate a behavioral model of the application, represented by the Deep Q-network. The goal of the testing tool is to generate test cases that test application's hard-to-reach states, cover the most code possible, and reveal faults, all in a limited amount of time.

A test case is created when the testing tool executes a sequence of actions in the application under test, this sequence is also known as an *episode* in reinforcement learning. In our implementation, an episode ends when one of the following conditions is met:

- It reaches the maximum number of 20 steps (transitions). After an empirical study where we vary the episode's length from 10 to 50, the value 20 was chosen because it gave the best average coverage across apps.
- Its last action leads to exiting the application.

- The screen is frozen (no changes in GUI) for the last 10 steps.

After an episode, the environment is reset, and the testing tool jumps to a random activity of the application. We reset the environment by killing all running processes in the emulator, then uninstalling and reinstalling the application. Each step of an episode proceeds as follows:

1) The *Observer* observes the application under test and builds the abstract representation of the GUI (the current *state*).
2) The *Observer* converts the current state into an input that we then use to feed to the Deep Q-Network.
3) The *Deep Q-Network* outputs a probability distribution over the next component that we should act on and passes it to the *Planner*.
4) The *Planner* chooses the next component to execute based on an $\epsilon - greedy$ policy.
5) The *Executor* executes an event on the chosen component.
6) The *Environment* transitions to a new state and returns the reward of the transition.
7) The transition, consisting of the old state, the new state, the executed component, and the reward, is added to the *Replay Memory*.
8) The *Deep Q-network* updates its weight by learning from a sampled batch of transitions from the *Replay Memory*.

This workflow is similar to the one in our previous work because both of them follow the Q-Learning algorithm. Nevertheless, the main difference with this architecture is concerned with the Deep Q-Network and the Observer, which also leads to changes in the Executor and the Planner. The implementation details of each module are given below:

1) Environment: The Environment is an interface that allows us to interact with the Android emulator and the application under test. Besides communicating with the Observer and the Executor, its most important role is to hold a function that calculates the reward value for each transition. We consider that a transition is better than another if it triggers more UI changes [13] [4]. Given two states s_1 and s_2, the reward function calculates the degree of change from s_1 to s_2 by counting the number of GUI events in s_2 but not in s_1, described as $|s_2 \backslash s_1|$. The relative change is then defined by the ratio $|s_2 \backslash s_1|/|s_2|$ where $|s_2|$ is the number of GUI events in $|s_2|$.

2) Observer: During run time, the Observer extracts the current screen's GUI hierarchy using Android UI Automator [22], obtaining the GUI tree. It then analyzes the GUI tree and classifies all actionable UI components into semantic categories as presented in Table I. It creates an abstract representation of the screen, also known as a $state$. A state consists of the activity's name and a set of GUI components. Each component is represented by a tuple containing the following information: semantic group, coordinates, class name, resource id, and four booleans indicating if the component is clickable, long-clickable, scrollable, checkable. The Observer only acknowledges components that belong strictly to the application under test in order to assure the accuracy of the model. However, it also takes into account two hard buttons of the phone: the menu button and the back button, which are important and necessary to navigate in the application. The two hard buttons are classified in the Navigation semantic group.

The Observer creates a vector of length 4 where the elements hold the number of components belonging to the semantic groups given in Table I. The vector is then passed as an input of the DQN.

3) Deep Q-Network and replay memory: Following Mnih, et. al's guidelines [8], we integrate a Deep Q-Network into our testing tool while simplifying its structure, because the dimension of our input is much smaller than in the case of learning from raw images. Our Deep Q-Network has two fully connected layers, each one is followed by ReLu activation and the loss function is defined as in section III. We implement a Replay Memory with a capacity of 500 transitions and a target Q-network which is updated every 20 learning steps. We train the Deep Q-Network every 5 transitions with a batch of 32 transitions sampled randomly from the Replay Memory.

4) Planner: The Planner is the principal actor that guides the testing tool to explore the application. It receives a probability distribution over the four GUI component groups and decides which component to act on next based on a $\epsilon - greedy$ policy. The policy selects a random component with the probability ϵ and follows the prediction of the Deep Q-Network with the probability $1-\epsilon$. At the beginning of the testing process, we want to encourage exploration behavior so that we can rapidly populate the state space. After a certain number of episodes when the Deep Q-network has collected enough transitions and has gained a certain knowledge about the application under test, we encourage the exploitation behavior: using the knowledge of the DQN to navigate in a more intelligent way. Hence, we decided to set $\epsilon = 1$ at the first episode and gradually decrease ϵ to 0.5 over the first 100 episodes (equivalent to about 2000 transitions). The value of ϵ is then maintained constant until the end of the testing process. When the Planner acts according to the Deep Q-Network prediction, it selects a component on the screen that has the highest probability value. For example, if the DQN outputs the probability distribution for (Input, Navigation, List, Button) respectively as (0.6, 0.2, 0.1, 0.1), the Planner will look for a component of group Input first. If there is not any component of group Input, it will look for a component of group Navigation and continue in the same manner until finding a component.

5) Executor: Unlike the executor in the testing tool of our previous work, which knows exactly which event to send to which component on screen, in this approach the executor only receives the information of which component on the screen that it needs to take action on. Based on the component's semantic group and clickable/long-clickable/checkable/scrollable information, the executor decides the type of event that it needs to send to the component.

For example, if the component is in the group Input and class EditText, the executor generates a random text to fill the input field. If the component is in the List group and scrollable, the executor executes a scrolling event. The testing tool currently supports 7 types of events: click, long click, scroll up, scroll down, swipe left, swipe right, text input.

V. Evaluation

A. Overview

We aim to answer three research questions:

- RQ1: Does QDroid achieve higher code coverage than our previous approach and state-of-the-art testing tools?
- RQ2: Does QDroid achieve high code coverage faster than other state-of-the-art testing tools?
- RQ3: Can the tool reveal faults during test? Is it better than other tools ?

We measure code coverage (line and method coverage) and count the number of distinct faults as metrics for evaluation. We compare the results of QDroid with our previous work (ClassicQ) [4] and state-of-the-art testing tools: Dynodroid (Dyno) [12], Puma [6], A3E [7] and GuiRipper (GuiR) [23].

We use AndroTest [11], a framework for comparing different automated testing tools to set up virtual machines and run each testing tool separately. Each virtual machine runs Ubuntu 32-bit, has 6114Mb of base memory and 2 processors. Each testing session is run for 2 hours, on a fresh Android emulator with all the data from the previous session removed. AndroTest also provides a set of instrumented open-source applications. Because a lot of apps provided are either too simple (containing only one activity with few GUI changes) or fail to start before the testing begins, we selected 12 apps which are stable for most of the testing tools.

After each test session on an application, QDroid provides developers with:

- The record of each action taken during each episode, which can be used to reconstruct test cases.
- The evolution of coverage during test.
- Android execution log, which is used to detect faults.
- The number of crashes occurred during the test.

B. Results

1) RQ1: Does QDroid achieve higher code coverage than our previous approach and state-of-the-art testing tools?: Table II gives details of the average method coverage obtained by each testing tool on each application. It also calculates the p-value of the hypothesis that QDroid obtains higher code coverage than each of the other testing tools in average. Fig. 2 shows the distribution of line and method coverage across target applications for QDroid, ClassicQ, Dynodroid, Puma, A3E and GuiRipper.

QDroid performs better than our previous tool ClassicQ, but it was not statistically significant ($p = 0.14$). As for other tools (Dynodroid, Puma, A3E and GuiRipper), QDroid obtained higher code coverage at a statistically significant level ($p < 0.05$). Its best and worst coverage is also higher than the best and worst coverage of all the other four tools.

TABLE II

Method coverage after two hours (%) and p-value for QDroid performs statistically significant better than each of the state-of-the-art testing tools

Application	Qdroid	ClassicQ	Dyno	Puma	A3E	GuiR
Any Memo	35.09	35.36	15.39	-	-	3.63
My Expenses	64.18	42.50	29.56	35.24	15.05	17.04
Who has my stuffs	89.01	88.09	58.93	67.48	51.15	29.92
Tippy Tipper	56.11	55.75	46.61	52.49	52.49	20.54
Munch Life	53.85	60.00	83.08	61.54	50.00	43.85
Mini Note Viewer	54.55	40.21	20.85	35.25	5.02	7.18
Mileage	35.29	34.73	26.85	34.06	2.99	16.38
Multi SMS sender	43.71	36.13	41.33	33.06	15.32	31.29
Hot Death	64.34	13.13	29.80	39.10	3.66	37.24
Random Music Player	58.73	58.73	58.73	58.73	3.17	41.27
Dalvik Explorer	83.51	83.40	23.82	70.87	41.08	5.39
Weight Chart	38.35	18.64	35.73	16.50	-	7.28
Mean	56.39	47.22	39.22	42.03	19.99	21.75
Standard deviation	16.64	22.02	18.80	20.23	20.98	13.96
p-value	-	0.14	0.02	0.047	0.0008	0.00009

One exception is with Munch Life where QDroid performs worse. In this case, we need to repeat the same action several times on the screen where the GUI skeleton doesn't change to be able to unlock a new state. Because QDroid favors the actions that trigger changes in UI and the test case is designed to end early when no GUI change is detected, QDroid isn't able to unlock the new state.

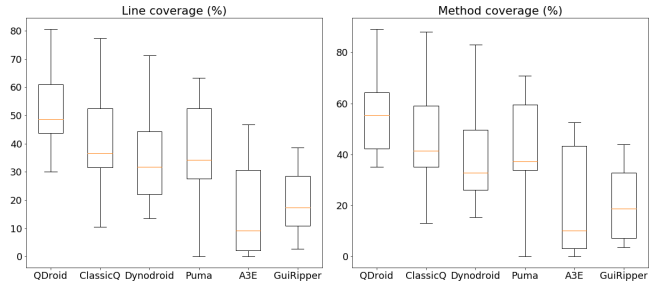

Fig. 2. Distribution of average coverage of each testing tool across apps

2) RQ2: Does QDroid achieve high code coverage faster than state-of-the-art testing tools?: The evolution of code coverage for QDroid, Dynodroid, Puma, A3E and GuiRipper (the average across all target applications and all runs) is presented in Fig. 3. On average, QDroid kick-starts with a high coverage and its performance improves over time. This proves that QDroid learns the behavioral model of the application gradually, and exploits this model more and more at the later phase of the testing process.

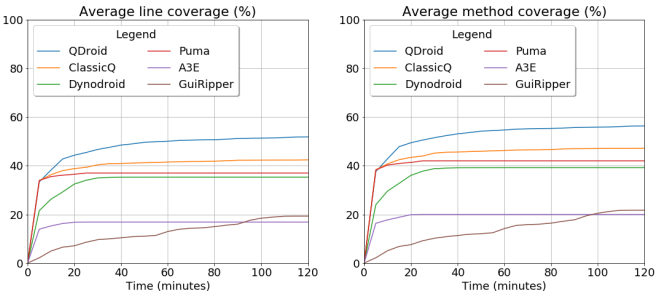

Fig. 3. Evolution of code coverage during test

TABLE III

NUMBER OF UNIQUE FAULTS DISCOVERED ACROSS ALL RUNS

App	Qdroid	Dyno	Puma	A3E	GuiR
Any Memo	11	9	0	0	0
My Expenses	0	0	0	0	0
Who has my stuffs	0	0	0	0	0
Tippy Tipper	0	0	0	0	0
Munch Life	0	0	0	0	0
Mini Note Viewer	1	1	1	0	0
Mileage	1	0	0	0	0
Multi SMS sender	0	0	0	0	0
Hot Death	2	0	0	0	0
Random Music Player	0	0	0	0	0
Dalvik Explorer	1	0	0	0	0
Weight Chart	1	1	0	0	0

3) RQ3: Fault detection ability: Table III gives the number of distinct faults discovered by each testing tool. The number of faults in each application before test is unknown. Distinct faults are identified by their unique error messages in the emulator's log. QDroid is able to detect faults during tests and outperforms the four other tools. The majority of the exceptions discovered belong to six main classes *java.lang*, *java.io*, *java.net*, *android.database*, *android.content* and *android.system*. Notice that even though different testing tools can have the same number of faults for one application, the faults are not necessarily identical. This is the case of Mini Note Viewer where the faults discovered by QDroid, Dynodroid and Puma are all different. For Any Memo and Weight Chart, the faults discovered by QDroid and Dynodroid have overlaps. Note that the framework we used, AndroTest, did not specifically state how many faults existed in each of the apps. Thus, we cannot determine the fault detection percentage of each tool. Nevertheless, QDroid performed the same or better compared to all other tools.

C. Threats to validity

The number of applications used in our evaluation is limited, hence raises a threat to external validity. Even though the target applications are chosen of different sizes and categories, it is unsure that the result can be generalized.

Training of a deep neural network can give very different outcome between different runs, which introduces a threat to internal validity. We ran each testing tool five times and took the average as the final result to reduce this threat.

VI. CONCLUSION

In this paper, we present a new approach using semantic analysis of GUI components and Deep Q-Network to conduct tests on Android applications. Our automated testing tool QDroid analyses the semantic information of GUI components and uses it as input to train a Deep Q-Network. The neural network, based on the principle of reinforcement learning, estimates a behavioral model of the application and exploits this model to guide the exploration inside the application. Evaluation has shown that QDroid presents an improvement in code coverage in comparison to state-of-the-art testing tools and it is effective in detecting faults.

As the integration of Deep Q-Network in the testing tool shows positive result, future work on this topic includes an evaluation of QDroid on applications having more complex GUI. Specifically, we plan on extending the action space (by combining event and component as one action) and the state space (for example, adding more information to distinguish similar states).

REFERENCES

[1] Google's announcement: https://twitter.com/Google/status/864890655906070529.

[2] Total Apps on Google Play: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.

[3] M. E. Joorabchi, A. Mesbah and P. Kruchten, "Real Challenges in Mobile App Development". Proc. of ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, 2013, pp. 15-24.

[4] Thi Anh Tuyet Vuong and Shingo Takada. 2018. "A reinforcement learning based approach to automated testing of Android applications". Proc. of A-TEST 2018, 31-37.

[5] Mnih et al. "Human-level control through deep reinforcement learning" Nature. 518. 529-33, 2005.

[6] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan.2014. "PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps". Proc. of MobiSys 2014, 204–217.

[7] Tanzirul Azim and Iulian Neamtiu. 2013. "A3E - Targeted and Depth-first Exploration for Systematic Testing of Android Apps". Proc. of OOPSLA 2013, 641–660.

[8] R. Mahmood, N. Mirzaei, and S. Malek. 2014. "EvoDroid: segmented evolutionary testing of Android apps" Proc. of FSE 2014, 599-609.

[9] S. Anand, M. Naik, M. J. Harrold, and H. Yang. 2012. "Automated concolic testing of smartphone apps". Proc. of FSE 2012, Article no. 59.

[10] Android Monkey: https://developer.android.com/studio/test/monkey.html

[11] S. R. Choudhary, A. Gorla and A. Orso, "Automated Test Input Generation for Android: Are We There Yet?", Proc. of ASE 2015, pp. 429-440.

[12] A. Machiry, R. Tahiliani, and M. Naik. 2013. "Dynodroid: an input generation system for Android apps", Proc. of ESEC/FSE 2013, 224-234.

[13] L. Mariani, M. Pezze, O. Riganelli and M. Santoro, "AutoBlackTest: Automatic Black-Box Testing of Interactive Applications", Proc. of 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, 2012, pp. 81-90.

[14] Anna I. Esparcia-Alcazar, Francisco Almenar, Urko Rueda Mirella Martinez, and Tanja E.J. Vos. 2016. "Q-learning strategies for action selection in the TESTAR automated testing tool". Proc. of META 2016. 174–180

[15] Y. Koroglu et al., "QBE: QLearning-Based Exploration of Android Applications", Proc. of ICST 2018, pp. 105-115.

[16] David Adamo, Md Khorrom Khan, Sreedevi Koppula, and Renée Bryce. 2018. "Reinforcement learning for Android GUI testing". Proc. of A-TEST 2018, 2-8.

[17] Richard S. Sutton and Andrew G. Barto (Eds.). 1998. "Reinforcement Learning An Introduction". MIT Press, Cambridge, MA.

[18] Arulkumaran, Kailash, Marc Peter Deisenroth, Miles Brundage and Anil A. Bharath. "Deep Reinforcement Learning: A Brief Survey." IEEE Signal Processing Magazine 34 (2017): 26-38.

[19] Richard Bellman. "On the Theory of Dynamic Programming". PNAS, 38(8): 716–719, 1952.

[20] Christopher J.C.H. Watkins and Peter Dayan. 1992. "Technical Note: Q-Learning". Machine Learning 8, 3-4 (May 1992), 279–292.

[21] Thomas F. Liu, Mark Craft, Jason Situ, Ersin Yumer, Radomir Mech, and Ranjitha Kumar. 2018. "Learning Design Semantics for Mobile Apps" Proc. of UIST 2018, 569-579.

[22] UI Automator for Python: https://github.com/xiaocong/uiautomator

[23] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. 2012. Using GUI Ripping for Automated Testing of Android Applications. Proc. of ASE 2012, 258–261.