Humanoid: A Deep Learning-based Approach to Automated Black-box Android App Testing

Yuanchun Li†, Ziyue Yang†, Yao Guo*, Xiangqun Chen
Key Laboratory of High Confidence Software Technologies (Ministry of Education),
Department of Computer Science, School of EECS, Peking University, Beijing, China
{*liyuanchun, ziyue.yang, yaoguo, cherry*}*@pku.edu.cn*

***Abstract*—Automated input generators must constantly choose which UI element to interact with and how to interact with it, in order to achieve high coverage with a limited time budget. Currently, most black-box input generators adopt pseudo-random or brute-force searching strategies, which may take very long to find the correct combination of inputs that can drive the app into new and important states. We propose Humanoid, an automated black-box Android app testing tool based on deep learning. The key technique behind Humanoid is a deep neural network model that can learn how human users choose actions based on an app's GUI from human interaction traces. The learned model can then be used to guide test input generation to achieve higher coverage. Experiments on both open-source apps and market apps demonstrate that Humanoid is able to reach higher coverage, and faster as well, than the state-of-the-art test input generators. Humanoid is open-sourced at https://github.com/yzygitzh/Humanoid and a demo video can be found at https://youtu.be/PDRxDrkyORs.**

***Index Terms*—Software testing, automated test input generation, graphical user interface, deep learning, mobile application, Android**

I. Introduction

Mobile applications (*apps* in short) have seen widespread adoption in recent years. These apps need to be adequately tested before being released. However, due to the rapid releasing cycle and limited human resources, it is difficult to manually construct test cases within a short time. As a result, automated test input generators for mobile apps have been studied extensively in both academia and industry.

The effectiveness of a test input generator is often measured by its test coverage. Thus the key to success for an automated test input generator is to choose the correct interactions for a given UI (the current UI during testing), such that the chosen interactions may reach new and important UI states, which in turn will lead to higher coverage in a limited time budget. Because it is hard for a machine to understand the GUI layout and the content within a GUI element, it is also difficult to determine which button to click or what should be inputted. As a result, most existing test generators [1]–[4] apply a random strategy to decide which GUI element to interact with and how. Although random strategies can also be further optimized, it has inherent limitations that make it difficult to choose the most efficient path to find the interactions that can drive the app into important states within a short time.

† co-primary authors, * corresponding author

Action Type	UI Element	Probability
touch		0.7
touch		0.15
touch	HAIDIAN QU	0.1
touch		0.02
touch		0.015
touch		0.002
touch	Instagram	0.001
touch	HAIDIAN QU	0.001
swipe_left		0.001
swipe_right		0.001
long_touch		0.0005
...	...	...

Fig. 1: An illustration of how Humanoid chooses test inputs. *The left side is a screenshot of the current UI of the AUT, and the right side enumerates the most possible interactions in the UI state. Humanoid computes a probability for each action based on a model learned from human interaction traces. The probability represents how likely the action will be chosen by Humanoid as the test input.*

In contrary to random input generators, human testers can easily identify the UI elements that are worth interacting with, even for a new app they have never seen before. The underlying reason is that human testers are themselves app users, so they have already gained some experience and knowledge about various mobile apps. Thus human testers know where to click and what to input, in order to achieve higher coverage, and taking less time as well.

Based on this observation, we propose Humanoid, an automated GUI test generator that is able to learn how humans interact with mobile apps and then use the learned model to guide test generation like a human tester. With the knowledge learned from human interaction traces, Humanoid can prioritize the possible interactions on a GUI page according to their importance and meaningfulness, as illustrated in Fig. 1. With the guidance of such a model, Humanoid is able to generate test inputs that can lead to important states faster than randomly generated inputs.

The core of Humanoid is a deep neural network model that predicts which UI elements are more likely to be interacted with by human users and how to interact with it. The input of

978-1-7281-2508-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ASE.2019.00104

IEEE computer society

the model is the current UI state as well as the most recent UI transitions, while the output is the predicted probability of each possible action on the UI page, which can be used to guide the test input generation process.

We implemented Humanoid and trained the interaction model with 304,976 human interactions extracted from a large-scale crowd-sourced UI interaction dataset Rico [5]. We compared Humanoid with six state-of-the-art test generators. The apps used for testing include 68 open-sourced apps obtained from the AndroTest [6] dataset and 200 popular apps from Google Play. The results showed that Humanoid was able to achieve 43.1% line coverage for open-source apps and 24.1% activity coverage for market apps, which was significantly higher than the best results (38.8% and 19.7%) achieved by other test generators using the same amount of time.

II. Tool Design

The core of Humanoid is an interaction model that learns the patterns about how humans interact with apps. Based on the model, the whole workflow can be separated into two phases, including an offline phase for training the model with human-generated interaction traces and an online phase in which the model is used to guide test input generation.

A. Model Training

End-users interact with an app based on what they see on the app's interface (i.e., GUI). Since different apps often share common UI design patterns, it is intuitive that the way how humans interact with GUI can generalize across different apps. The goal of our proposed interaction model is to capture such generalizable patterns.

We introduce a concept *UI context* to model what humans reference when they interact with an app. A *UI context* $context_i$ consists of the current UI state s_i and three latest UI transitions (s_{i-1}, a_{i-1}), (s_{i-2}, a_{i-2}), (s_{i-3}, a_{i-3}). The current UI state represents what the users see when they perform the action, while the latest UI transitions are used to model the users' underlying intention during the current interaction session.

Each UI state is represented as a two-channel UI skeleton image, in which the first channel renders the bounding box regions of text UI elements and the second channel renders the bounding box regions of non-text UI elements. Each action is represented by its action type and target location coordinates. The action type is encoded as a seven-dimensional vector, in which each dimension maps to one of the seven action types as described earlier. The action target location is encoded as a *heatmap*. Each pixel in the heatmap is the probability of the pixel being the action target location.

Thus, the representation of a *UI context*, i.e., the input for our interaction model, is a stack of images including one 2-channel image for the current UI state and three 3-channel images for three latest UI transitions (each transition include one 2-channel image for the UI state and one 1-channel image for the action). All images are scaled to the size of 180×320 pixels. For ease of learning, we also add one channel of zero padding for the current UI state. In the end, a *UI context* is represented as a $4 \times 180 \times 320 \times 3$ vector.

Given the *UI context* vector, the output of the interaction model are two conditional probability distributions:

1) $p_{type}(\ t \mid context_i)$, representing the probability distribution of t, the type of the next action, where $t \in \{touch, long\_touch, swipe\_up, ...\}$.
2) $p_{loc}(\ x, y \mid context_i)$, representing the probability distribution of screen coordinates x, y being the target of the next action, where $0 < x < screen\_width$ and $0 < y < screen\_height$.

The probability of each action on the current UI state can be calculated with:

$$p(action) = p_{type}(action.type) * \sum_{x,y \text{ in action.element}} p_{loc}(x, y)$$

Then, the action probabilities can be used to guide test input generation.

Fig. 2 shows the deep neural network model used to learn the two conditional probability distributions defined above. It accepts the representation of the current *UI context* $context_i$ as input, and outputs location and type distributions of a_i. The model uses convolutional layers to capture the GUI visual information and residual LSTM modules to capture the inter-action context information. The de-convolutional layers and the fully connected layers are used to generate the distribution of a_i's location and type respectively.

The dataset we used to train the interaction model is processed from Rico [5], a large crowd-sourced dataset of human interactions. We extracted 12,278 interaction flows belonging to 10,477 apps, and each interaction flow contained 24.8 actions on average. During training, the probability of the action taken by the human users is set to 1, while the probabilities of other actions are set to 0.

B. Guided Test Generation

Humanoid uses a GUI model to save the memory of transitions. The GUI model is represented as a UI transition graph (UTG), whose nodes are UI states and edges are the actions that lead to UI state transitions.

Humanoid generates two types of test inputs, including *exploration actions* that are used to discover the unseen behaviors in an app, and *navigation actions* that drive the app to known states that contain unexplored actions.

In each step, Humanoid checks whether there are unexplored actions in the current state. Humanoid chooses *exploration* if there are unexplored actions, and chooses *navigation* if the current state is fully explored. The navigation process is straightforward. In the exploration process, Humanoid gets the probabilities of the actions predicted by the interaction model, and makes a weighted choice based on the probabilities.

Since the actions that humans would take will be assigned higher probabilities, they get higher chances to be chosen by Humanoid as test inputs. Thus the inputs generated by Humanoid are more human-like than randomly chosen ones, which in turn will drive the app into important UI states faster and lead to higher test coverage.

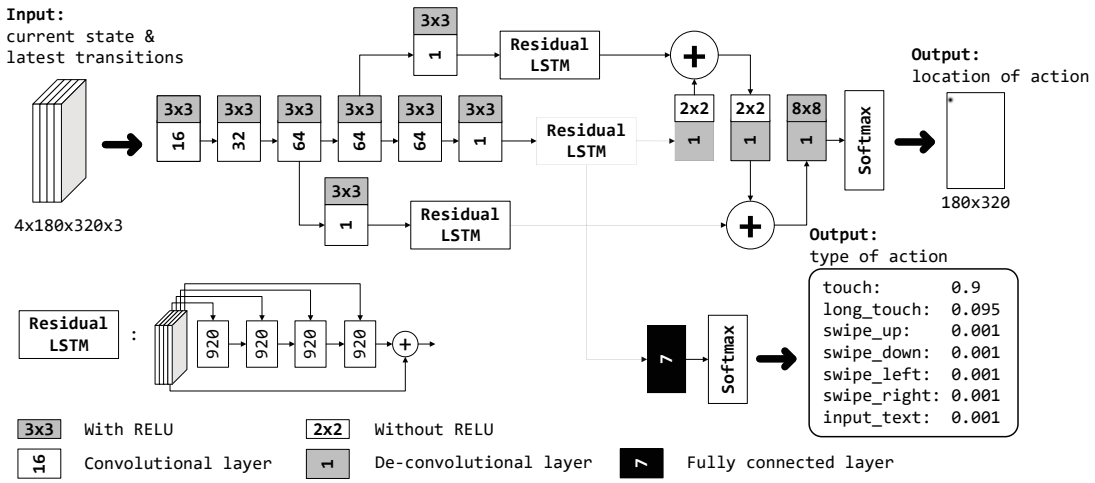


Fig. 2: The architecture of the interaction model in Humanoid.

III. Evaluation

We evaluated Humanoid by using it to conduct testing of two different sets of Android apps, including 68 open-source apps obtained from AndroTest [6] and 200 popular commercial apps downloaded from Google Play. We measured the test coverage and test progress of Humanoid and compared the results with six state-of-the-art testing tools, including Monkey [1], PUMA [7], Stoat [8], DroidMate [9], Sapienz [10] and DroidBot [11].

The machine we used to conduct the experiments is a workstation with two Intel Xeon E5-2620 CPUs, 64GB RAM and an NVidia GeForce GTX 1080 Ti GPU. Training the interaction model took about 66 hours. When applying the model, we ran 4 instances of Android emulators on the machine to test apps in parallel.

We used each testing tool (with their default configuration) to run each app for a fixed length of time on an Android emulator (1 hour for each open-source app and 3 hours for each market app because market apps are usually more complicated). In order to accommodate the recent market apps, most of the tools were evaluated on Android 6.0. However, as Sapienz is close-sourced and only supports Android 4.4, so it was evaluated on Android 4.4 instead. For each app and tool, we recorded the final coverage and the progressive coverage after each action was performed. We repeated this process three times and used the average as the final results.

When testing open-source apps, the test coverage achieved by each testing tool was almost converged in 1 hour. The overall comparison of the final line coverage is shown in Fig. 3. On average, Humanoid achieved a line coverage of 43.1%, which was the highest across all test input generators. Based on manual inspection of the test traces, the high coverage of Humanoid was mainly due to two reasons: First, Humanoid was able to identify and prioritize the critical UI elements when there were plenty of UI elements to choose from. Second, Humanoid had a higher chance to perform a meaningful sequence of actions, which can drive the app into

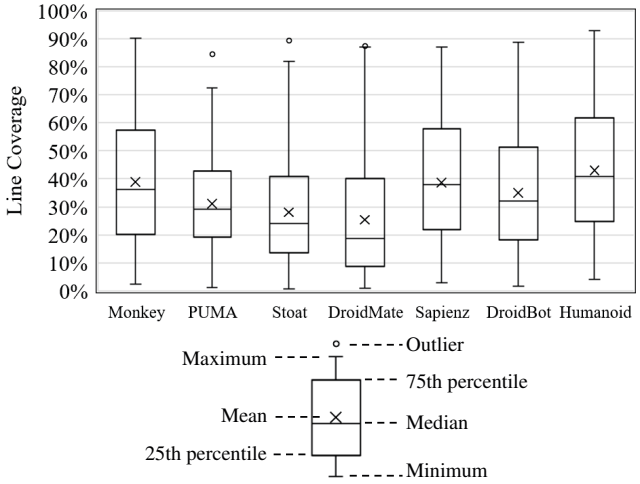


Fig. 3: Line coverage of different tools for open-source apps.

unexplored core functionalities.

It is interesting to see that Monkey, which adopts a random exploration strategy, achieved higher coverage than all other model-based testing tools except Humanoid. The fact that Monkey performs better than most other testing tools has also been confirmed by other researchers [6]. The main reason is that Monkey is able to generate much more inputs than other tools within the same amount of time.

We further conducted experiments on the market apps to see whether Humanoid is still more effective. Since the source code is not available for these market apps, we were unable to compute the line coverage, thus we used the activity coverage (percentage of reached activities) instead.

The final activity coverage achieved by the testing tools in 3 hours is shown in Fig. 4. Similar to open-source apps, Humanoid also achieved the highest coverage (24.1%) as compared with other tools.

Fig. 5 shows the progressive coverage *w.r.t.* the number of

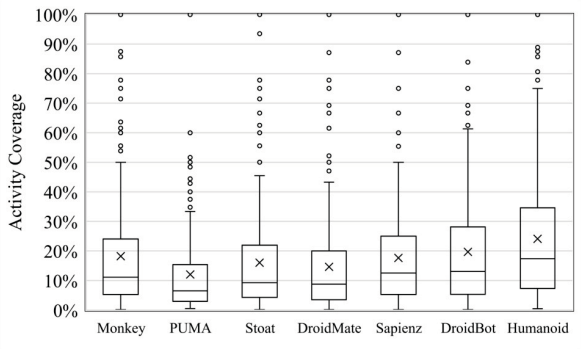
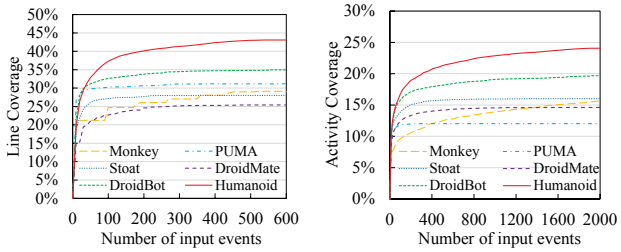


Fig. 4: Activity coverage of different tools for market apps.



(a) Open source apps (b) Market apps

Fig. 5: Progressive coverage of different tools.

input events sent by each testing tool. Note that we did not include Sapienz in the progressive coverage figures because it sends events too fast and we could not slow it down as it was close-sourced. In the first few steps, the line coverage of all testing tools increased rapidly, as the apps were just started and all UI states were new. Humanoid started to lead after about 50 events. That was because the easy-reachable code was already covered at that point, and the other states were hidden behind specific interactions that can hardly be produced by other testing tools. The coverage for some apps was not converged at the end of testing due to the complexity of these apps. However, we believe that Humanoid will keep the advantage even with longer testing time.

IV. Related Work

Automated GUI test generation has become an active research area since the prevalence of mobile apps. Most test generators adopt three types of strategies: random, model-based, and targeted.

A typical example using the *random strategy* is Monkey [1], which sends actions without any information from the app. DynoDroid [2] filters out unacceptable events based on the GUI layout. Sapienz [10] uses a genetic algorithm to search for the test sequences that can achieve higher coverage.

Several other testing tools build and use *a GUI model* of the app to generate test input [7], [11], [12]. Based on the GUI models, testing tools can generate inputs that can quickly navigate the app to unexplored states. Model-based strategies can also be optimized. For example, Stoat [8] can iteratively refine the test strategy based on existing explorations, and DroidMate [9] can infer acceptable actions for a UI element by mining from other apps.

The *targeted strategy* is designed to address the problem that some app behavior can only be revealed with specific test inputs. These testing tools [13], [14] usually use sophisticated techniques such as data flow analysis and symbolic execution to find the interactions that can lead to the target states.

Humanoid is different from these approaches as it utilizes the GUI visual information, which is an important reference when human users or testers are exploring an app.

V. Conclusion

This tool demonstration paper introduces Humanoid, a new GUI test generator for Android apps that is able to generate human-like test inputs through deep learning. Humanoid adopts a DNN model to learn how human users navigate through an app, from a large set of human-generated interaction traces. Experiments show that, with the guidance of the learned model, Humanoid is able to achieve higher test coverage, and faster, than six state-of-the-art testing tools.

Acknowledgment

This work was partly supported by the National Key Research and Development Program (2017YFB1001904) and the National Natural Science Foundation of China (61772042).

References

[1] "Android Developers, UI/Application Exerciser Monkey," 2012.
[2] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *FSE 2013*.
[3] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *ASE 2012*.
[4] Y.-M. Baek and D.-H. Bae, "Automated model-based android gui testing using multi-level gui comparison criteria," in *ASE 2016*.
[5] B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols, and R. Kumar, "Rico: A mobile app dataset for building data-driven design applications," in *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, 2017.
[6] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?" in *ASE 2015*.
[7] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps," in *MobiSys 2014*.
[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 *FSE 2017*.
[9] N. P. Borges Jr, M. Gómez, and A. Zeller, "Guiding app testing with mined interaction models," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, 2018.
[10] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016.
[11] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: A lightweight ui-guided test input generator for android," in *Proceedings of the 39th International Conference on Software Engineering Companion*, 2017.
[12] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie, "An empirical study of android test generation tools in industrial cases," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.
[13] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *FSE 2012*.
[14] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *OOPSLA 2013*, 2013.