MUBot: Learning to Test Large-Scale Commercial Android Apps like a Human

Chao Peng, Zhao Zhang, Zhengwei Lv, Ping Yang
ByteDance
Beijing, China
{pengchao.x, zhangzhao.a, lvzhengwei.m, yangping.cser}@bytedance.com

***Abstract*—Automated GUI testing has been playing a key role to uncover crashes to ensure the stability and robustness of Android apps. Recent research has proposed random, search-based and model-based testing techniques for GUI event generation. In industrial practices, different companies have developed various GUI exploration tools such as Facebook Sapienz, WeChat WeTest and ByteDance Fastbot to test their products. However, these tools are bound to their predefined GUI exploration strategies and lack of the ability to generate human-like actions to test meaningful scenarios. To address these challenges, Humanoid is the first Android testing tool that utilises deep learning to imitate human behaviours and achieves promising results over current model-based methods. However, we find some challenges when applying Humanoid to test our sophisticated commercial apps such as infinite loops and low test coverage. To this end, we performed the first case study on the performance of deep learning techniques using commercial apps to understand the underlying reason of the current weakness of this promising method. Based on our findings, we propose MUBot (Multi-modal User Bot) for human-like Android testing. Our empirical evaluation reveals that MUBot has better performance over Humanoid and Fastbot, our in-house testing tool on coverage achieved and bug-fixing rate on commercial apps.**

***Index Terms*—android testing, graphical user interface, deep learning**

I. Introduction

Mobile apps have been playing an important role in our daily life. Several testing techniques have been proposed to test mobile apps [1]–[6]. For instance, Android Monkey [7], is a random testing tool that comes with the Android software development kit and has been widely used in industrial practices owing to its low computation cost, ease of use and extensibility.

In industrial practises, ByteDance and Facebook, for example, have developed in-house Android GUI testing tools Fastbot [8] and Sapienz [9] respectively for stability and compatibility testing for their apps.

However, existing techniques omit an important factor - real users of these apps. Test inputs generated by these techniques are significantly different from real user interaction traces and cannot explore apps from a real user's perspective. As a result, although being effective in elevating test coverage and uncovering crashes, most of these crashes are classified to low priority for fixing by our developers.

To address this problem, Li et al. [10] propose Humanoid, aided by deep learning to learn from human interactions with mobile apps and generate test cases to imitate human behaviour and prioritise GUI interactions according to their importance from users' perspective. By contrast, it can take a very long time for other testing techniques to find a reasonable interaction sequence and reach important states. Furthermore, Humanoid outperforms other tools on many open-source apps in terms of test coverage achieved [10].

Although being attractive in generating meaningful test inputs and achieving an impressive performance on test effectiveness, it still has some weaknesses, especially when being applied to test large-scale and sophisticated commercial apps. In our empirical study, we find that the performance of Humanoid on our apps with more complex interaction logic and page-level features decreases significantly compared to its performance on open-source apps. The primary reason for such cases is that the inefficient GUI structure representation and single-modal model of Humanoid leads to a low test coverage.

According to these findings, we propose in this paper, two types of improvement to the deep learning algorithm and present a new tool, MUBot (Multi-modal User Bot), to improve test coverage and crash finding capabilities. Firstly, for the inefficient GUI representation, we introduce a thinner but more generic GUI representation instead of the original redundant high dimensional prototype with poor extensibility. Secondly, the model used in Humanoid is overly complex and the single-modal mechanism restricts the diversity of test cases, which means it can only predict one test case on one page view. Therefore, we redesign the test generation model into a multi-modal way to produce multiple test cases within the same page. Finally, we conduct experiments on large-scale and sophisticated commercial apps to measure the effectiveness on how our methods improve the test coverage, stability and reliability compared with Humanoid and Android Monkey.

In summary, the main contributions in this paper are:

- A case study on the test effectiveness of Humanoid on a commercial app and findings on its limitations (Section II).
- A general GUI representation suitable for deep learning algorithms as introduced in Section III-A.
- A new multi-modal model to learn from user interactions and guide automated test input generation, presented in Section III-B.
- An empirical evaluation (Section IV) comparing the performance of MUBot against Humanoid and Android Monkey. Specifically, in Section IV-D, we report the

industrial deployment of MUBot in our real daily testing environment to study hotspot activity coverage, crash-fixing rate and the importance of uncovered crashes from the users' perspective.

In addition, we report developers' observations and lessons learned in Section IV .

II. Background and Motivation

Before discussing our approach in detail, we present basic concepts in Android GUI testing that are essential in understanding our approach. We then provide a brief introduction to Humanoid and discuss its limitations on testing large-scale commercial apps by a case study as our research motivation.

A. Testing Android Apps

`Graphical user interface (GUI)` is the place where the user interacts with the Android app. In Android apps, an `Activity` is the implementation of a window or screen containing various GUI elements, such as buttons and textboxes. These GUI elements are usually referred to as `widgets` or `views` in the context of Android development and are responsible for user interaction handling. `User interactions` (as known as `actions` or `GUI events`) could be button clicks, text editions, screen long-touching, etc.

As Android apps are event-driven, apps can be considered as a combination of many `GUI states` and the transition between these states are triggered by GUI interactions. As a result, test inputs are normally in the form of `GUI interaction sequences`. Writing or recording test inputs manually can be time-consuming, which gives rise to the development of automated testing tools.

B. Humanoid Case Study

To the best of our knowledge, Humanoid is the only existing Android testing tool that learns and imitates real user behaviour. Unlike existing techniques that randomly select GUI actions from all available ones in the current GUI state, Humanoid prioritise inputs that are more likely to be performed by human based on knowledge learned from real user interactions. To understand and explore the app, Humanoid uses GUI representation construction similar to model-based testing techniques.

1) The Deep Learning Approach: The approach of Humanoid has two phases: offline learning and online testing. The core of Humanoid is the deep learning model which is trained during the first phase to learn the relation between the GUI contexts (the current GUI state of visual information and previous GUI transitions) and user-performed interactions (input event type such as clicking and the coordinates where this action takes place). After the offline learning, Humanoid is able to calculate the probability of each available actions on a given GUI state being interacted with by a real user and generate human-like test inputs during the online testing phase.

Similar to existing model-based test input generators, Humanoid uses a GUI representation to store the history of GUI transitions. It extracts the type of GUI elements to draw a segmented binary channel: text and image. The text segmentation channel contains GUI elements such as `TextView`, `TextButton` and other similar elements with text attributes.

2) Case Study: We apply Humanoid to test our daily news app, Toutiao, with 520 activities and compare activity coverage against Monkey which is the second best tool reported by the Humanoid literature [10].

a) Experiment Configuration: As there is a large fragmentation of the Android market, to improve the validity of our study , we use real devices from different vendors with various Android versions in contrast to Android emulators used by the experiment presented in the Humanoid paper [10]. For each tool, we perform 3-hour test runs on each device for three times and the activity coverage achieved is reported in the average number across the 3 test runs.

TABLE I
Activity Coverage Achieved by Monkey and Humanoid for 3 hours

#	Device Model	Android Version	Activity Coverage	
			Monkey	Humanoid
1	Huawei nova 5i Pro	9	3.7%	**4.7%**[1]
2	Huawei nova 5i Pro	10	**6.3%**	4.8%
3	Xiaomi Redmi Note 7	9	**4.9%**	3.8%
4	Xiaomi 9	10	**4.9%**	2.1%[1]
5	VIVO X21i	9	**4.7%**	-[2]
6	VIVO iQOO V1824BA	10	**7.2%**	-[2]
7	OPPO R11	8	**6.2%**	3.4%
8	OPPO Reno 3 5G	10	**6.5%**	4.4%[1]
9	Google Pixel 3	9	**6.6%**	5.0%
10	Google Pixel 3	10	**8.5%**	5.1%

[1] System out-of-memory error was observed during testing.
[2] We were completely not able to run Humanoid due to system out-of-memory.

b) Results.: As shown in Table I, Humanoid only achieved better activity coverage once (on Huawei nova 5i Pro, Android 9). For most test runs, Humanoid was outperformed by Monkey and more importantly, it caused system out-of-memory errors and we had to restart the computer when testing on VIVO devices running both Android 9 and 10 for all the 3 test runs. This situation also occurred once on Huawei and OPPO devices.

c) Findings: To study on what aspect we can improve the deep learning approach, we manually observe the execution of experiments and further investigate the underlying approach of Humanoid. We make the following findings.

Finding 1. The complicated GUI abstraction model hinders the effectiveness of GUI state abstraction and GUI action generation.

The first disadvantage of Humanoid is the high dimensionality of its GUI representation model. High dimensionality means that the more detailed element types the model has, the more segmentation channels the model will get. For instance, when there are only text and image widgets available, two segmentation channels are built. However, when it is applied to short video apps, for example, one more video widget channel needs to be added. In addition, it also overlooks

some layout information or other rare GUI element types, leading to inconsistent information extracted and encoded. However, the design of sophisticated industry apps often blurs the relationship between GUI element type and the interaction so that it can be hard for Humanoid to infer an efficient interactive action. For example, although there is a button widget type, the layout type is often used to design buttons in our apps, as it is easier to customise the appearance of layouts. In sum, to use Humanoid, the tester needs to specify the number of widget types in the app under test so that Humanoid will have enough segmentation channels to encode widgets. When the number of available channels is not enough for all widget types, Humanoid cannot encode the model and infer actions to test apps.

Finding 2. The complicated GUI abstraction model also makes GUI action generation slow and even sometimes causes system out of memory.

The second disadvantage is also introduced by this complicated GUI representation. During our experiment, we find that the stored representation model parameter file by Humanoid is around tens of megabytes and the average time used for generating one action is more than 5 seconds while other model-based tools take only less than 1 second. It is also observed that for 8 times during our experiment, the computer reported out-of-memory errors after several minutes' execution of Humanoid and we had to restart the machine.

Finding 3. The single-modal deep learning model tends to make repeated actions which may lead to infinite loops.

The last disadvantage is brought by the single-modal deep learning model. Given a GUI state, the single-modal model always generates the same probability distribution for widgets available in this GUI state. When the state is reached again during GUI exploration, the same action is likely to be selected again, resulting in infinite loops.

C. The RICO GUI Interaction Dataset

Our approach and Humanoid are both based on supervised deep learning using Rico, a large open-source dataset of human interactions [11]. The Rico dataset is consisted of GUI design information and user interaction data collected from more than 9,300 Android apps across 27 categories, which contains visual, textual, structural and interactive properties of more than 66,000 unique GUI screens and 3 million GUI elements.

Figure 1 shows an example of Rico trace data. Among all types of GUI data available in Rico, we focus on the `Interaction Traces` section which includes sequences of GUI elements and user actions connecting them. An action is a $< x, y >$ coordinate pair that represents the interaction position on the screen. The scroll-like action has more than one $< x, y >$ pairs to indicate the start and end position. GUI states in Rico are represented using a JSON-like format data, which contains properties of GUI elements in the state.

III. Our Approach

To address limitations of Humanoid as discussed in Section II, we present MUBot (Multi-modal User Bot) with a general GUI representation construction strategy and a multi-modal deep learning model. The workflow of MUBot is illustrated in Figure 2.

1) **Model Traning.** We train the deep learning model of MUBot using Rico.
2) **GUI State Abstraction.** During GUI exploration, the GUI state (app window) represented in a XML or JSON formatted GUI tree is encoded in a single-channel grey-scale map.
3) **GUI Feature Extraction.** The GUI state is then processed by the CNN (Convolutional Neural Network) to produce state-action sequences.
4) **Probability Distribution Calculation.** The state-action sequence is processed through the GRU (Gated Recurrent Units) and MDN (Mixture Density Network) modules to calculate the probability distribution.
5) **Final Prediction.** Sampling from the probability distribution, MUBot generates the final GUI action to be performed as a prediction of human behaviour.

The major improvements can be classified into 1) parsing GUI interaction data to the gray-scale map and 2) multi-modal model design and training strategy. We discuss these methods in detail in the rest of this section.

A. GUI Representation Construction

We propose a new general representation to build the GUI structure with lower data dimensions to improve the information volume. Practically, the primary interaction area and GUI elements in commercial apps are often located in nest layouts.For instance, in a video app, the primary interaction area is the layout of the video widget along with the nest layout of buttons around it, such as LIKE, FORWARD and COMMENT buttons. Therefore, if we highlight these areas and GUI elements, we can focus areas with intensive interaction hints. This method should be independent of the amount of GUI elements and the number of their types.

As shown as an example in Figure 3, we consider all GUI elements without distinguishing their types to draw a gray-scale GUI abstraction. The primary interaction area is aggregated higher pixel value and shown in brighter blocks.

With this grey-scaled abstraction, MUBot is able to encode all types of GUI elements in a single channel and is feasible to test apps with different features. However, in this example, Humanoid needs to keep three channels for images and icons, buttons and textboxes.

B. Multi-modal Deep Learning Model

To get a more comprehensive test coverage, we usually need to generate test inputs for deeper GUI states. This requires our test generation to predict several possible results rather than the best or an average result which is beyond the capability of the single-modal mechanism used by Humanoid. In addition to this concern, when single-modal is applied, the

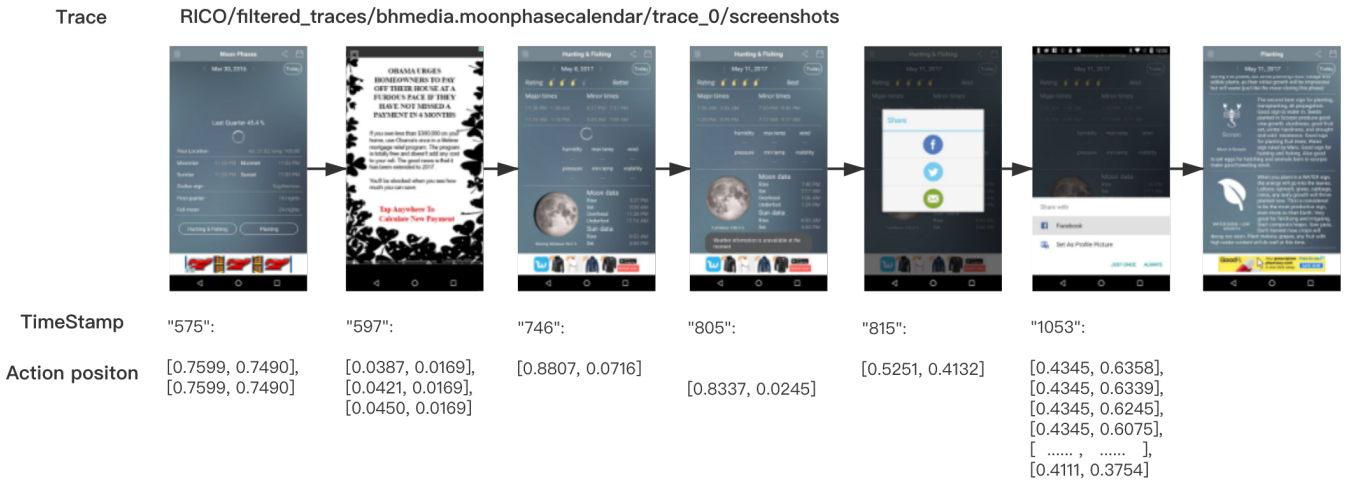

Fig. 1. An Example of Rico Trace Data

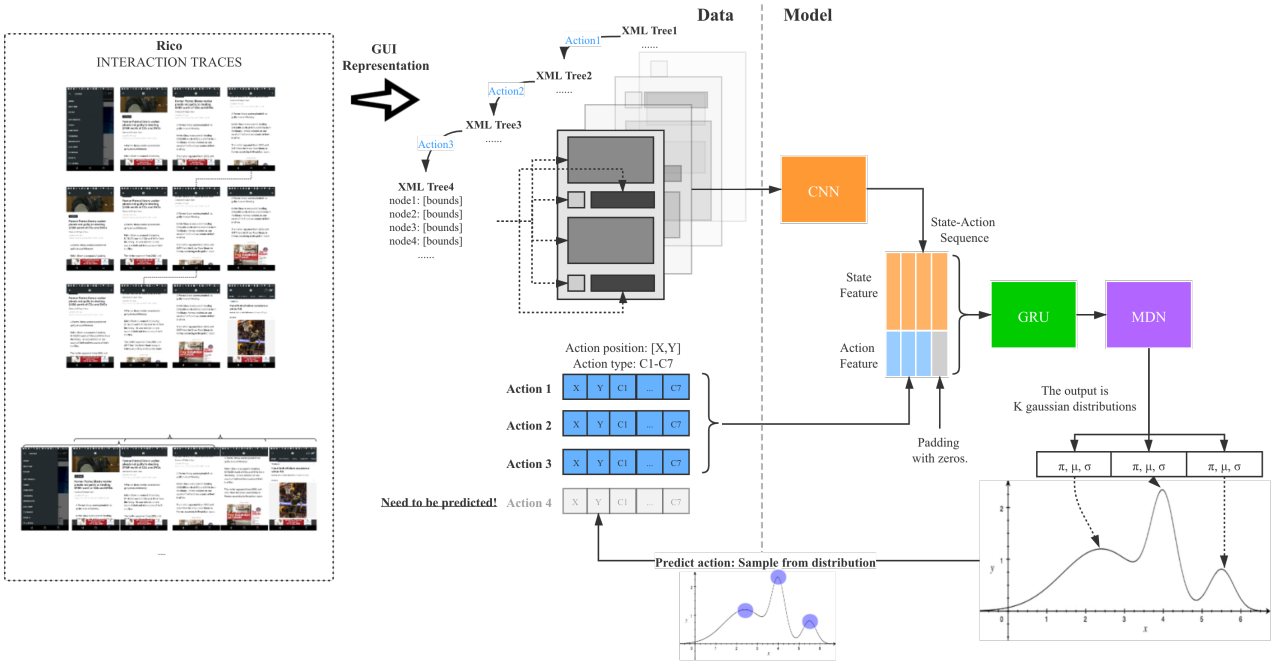

Fig. 2. MUBot Workflow Overview

same prediction is very likely to be made and the test process can easily fall into an infinite loop and never jumps out of it, as shown in Figure 4.

In a video app, a typical problem can be described as a repeated event sequence of the following:

Click a Feed → Click and Play Video → Click and Full Screen → BACK to Feed→ Click a Feed → [infinite loop]

By contrast, the multi-modal mechanism can predict several possible results and only one result will be chosen randomly. Therefore, it can rarely face such infinite loop problem.

In addition, the model used by Humanoid is quite complex and takes expensive computation cost. Given these insights, we propose a new slimmer but more efficient multi-modal model to generate test inputs.

1) Model Structure: Recalling the workflow of MUBot shown in Figure 2, the multi-modal model of MUBot has three parts, a CNN module, a GRU module, and a mixture density network. The input to the model is a stack of gray-scale maps with corresponding action vector code, except for the last vector element which is set to 0. The values of actions and states are normalized to (0, 1). The action position is indicated using an $< x, y >$ pair where x and y are continuous values within (0, 1) with the top-left corner as the relative origin $< 0, 0 >$. Action types are represented using one hot encoding. The output of MUBot is a mixed Gaussian distribution with K kernels. The predicted action is yielded from this distribution by sampling.

a) CNN Module.: Convolutional neural networks (CNN) are known as shift invariant or space invariant artificial neural networks, based on the shared-weight architecture of the convolution kernels or filters that slide along input features [12]. They are widely considered as a good way to extract image features to achieve some vision tasks like classification or segmentation. In our approach, we use a five-layer CNN

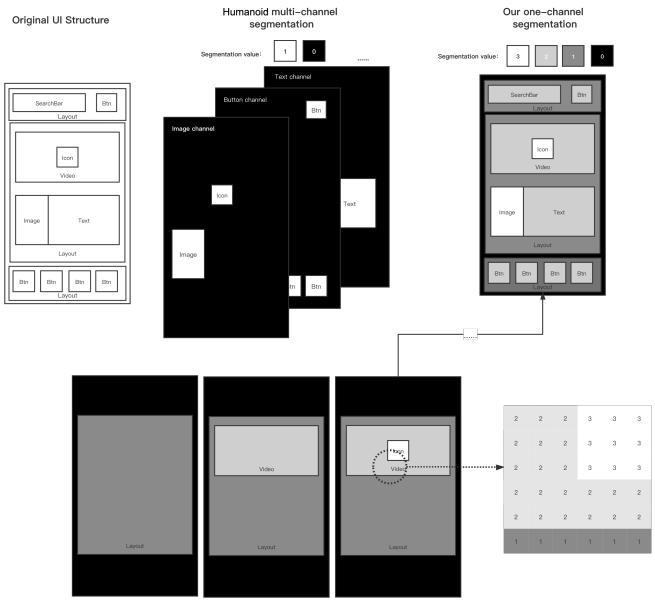

Fig. 3. Comparison of the GUI Representation Abstraction of Humanoid and MUBot

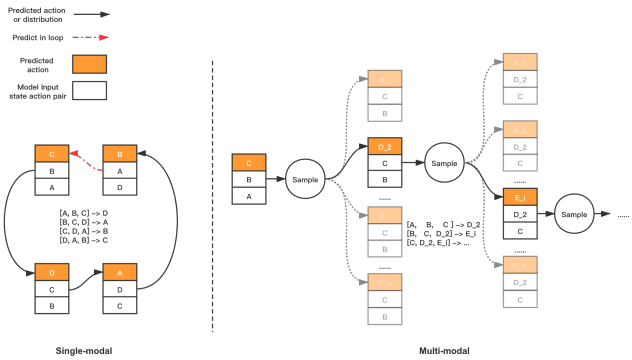

Fig. 4. An Example of an Infinite Loop

to extract features from gray-scale maps with ReLU as a non-linear activation function. Each layer is followed by an average pooling layer to shrink the spatial dimension and convert the gray-scale map into a vector eventually. Notably, the convolutional layers used in our model share parameters between the four gray-scale input, as the module is used to extract image features rather than calculate the correlation of the adjacent. The hyper-parameters of each CNN layer is presented in Table II. The final output is reshaped in to a 180-dimensional vector.

b) GRU Module.: Long short-term memory (LSTM) [13] is an artificial recurrent neural network (RNN) [14] architecture used for deep learning, with recognised performance in time series processing tasks. LSTM units include a memory cell that maintains information in memory for long periods, which is controlled by a set of gates. However, the limitation of LSTM is the complexity that costs many parameters and slows down the inference time. The inference time directly

TABLE II
THE HYPER-PARAMETERS OF CNN MODULE

layer	units	kernel	stride	padding
layer1	16	3	1	1
layer2	32	3	1	1
layer3	64	3	1	1
layer4	128	3	1	1
layer5	1	1	1	0

determines the number of test execution events and the execution time interval which is highly related to testing efficiency. To get a reasonable trade-off, we choose Gated Recurrent Units (GRU) [15] to process the sequence. Compared with LSTM, GRU has a more streamlined structure with a sound performance. In our scenario, as the input time series is short and fixed value, we use GRU instead of the more complicated LSTM to process the sequence data and keep the model slim and fast.

c) Mixture Density Network (MDN): For Android apps, there are many possible actions for users or testing tools to pick on the same page, such as scrolling up to browse more content, back to the previous page, or click on some clickable widgets. Based on this fact, we introduce MDN [16] in MUBot as the output layer to predict a multivariate mixture Gaussian distribution rather than a single result produced by the single-modal model. After producing multiple options of GUI actions, the final action picked on the current page is a sample drawn from this Gaussian distribution. The probability density of the predicted action can be modelled using a linear combination of Gaussian kernel functions as follows:

$$p\left(y|x\right) = \sum_{i=1}^{K} \pi_i\left(x\right) g_i\left(y|x\right)$$

We set π_i as a mixture coefficient and $g_i\left(y|x\right)$ represents a multivariate mixture Gaussian distribution given by x. K is the amount of sub-distributions. The value of K should be more than the number of possible actions in any given activity to capture and predict all possible actions. The parameters of the mixing coefficient and the Gaussian kernel are inferred by the neural network based on the historical sequence and the current state. The formulation of Gaussian distribution is shown in:

$$g\left(y|x\right) = \frac{1}{\left(2\pi\right)^{c/2}\sigma_i\left(x\right)} exp\{-\frac{||y-\mu_i\left(x\right)||^2}{2\sigma_i\left(x\right)^2}\}$$

where the vector $\mu_i\left(x\right)$ is the center of the i-th kernel. The parameters $\mu_i\left(x\right), \sigma_i\left(x\right)$, and mixing coefficient $\pi_i\left(x\right)$ of the Gaussian kernel are represented by three independent output layers. Each coefficient of the MDN output layer needs neurons, that is, when the number of Gaussian kernels is K and the dimension of the prediction output dimension is C, the number of parameters of the mixing coefficient π is K for each Gaussian distribution; the number of parameters of μ and σ are K * C. Furthermore, π and σ needs to meet specific constraints shown in the following equation. Summarization of π_i should be equal to 1, as the sum of the probability always

equals to 1. The σ should be a positive value because this represents the variance of the distribution.

$$\begin{cases} \sum_i^K \pi_i = 1, & 0 < \pi_i < 1 \\ \sigma_i >= 0, \end{cases}$$

Therefore, naively, the output of π_i uses *softmax* to meet the conditions, and σ uses an exponential activation function to ensure a non-negative value. However, in the actual training process, it is easy to cause the numerical explosion to eventually overflow. Based on these facts, we need to fix these constraints a little bit by normalizing output, π, to minus its maximum before applying to *softmax* function and use *nnelu* activation function to replace the normal e-exponential to reduce the numerical explosion in training. The *nnelu* can guarantee the σ always positive even when it is slightly greater than zero due to a tiny value *eps* which is helpful to prevent explosion. The *nnelu* is defined as follows:

$$nnelu(t) = \begin{cases} t + eps & \text{if } t > 0 \\ e^t - 1 + eps & \text{if } t < 0 \end{cases}$$

The mean value $\mu_i(x)$ can be used without any further changes. Finally, we can define the error in terms of the negative
log-likelihood.

$$\mathcal{L}_{MDN} = -ln\left(\sum_{i=1}^{K} \pi_i(x)\, g_i(y|x)\right)$$

According to the derivation of [17], we introduce a priori constraint to prevent the model from over sinking into minority high-frequency action types such as click. The final prior constraint is shown in the following equation:

$$\mathcal{L}_{prior} = -\sum_{j=1}^{N}\sum_{i=1}^{K}(\lambda_i - 1)\, ln\,(\pi_i(W, X_j))$$

The final loss function is calculated as the follows:

$$\mathcal{L} = \mathcal{L}_{MDN} + \mathcal{L}_{prior}$$

C. Model Training and Inference

1) Model Training: The CNN layers uses the *kaiming* initialization method [18], and the fully connected layer uses orthogonal parameter initialization [19]. We train this model with *batchsize* 512 and 12000 training steps. The learning rate is set to 2e-4 initially and decayed by 1/10 at steps of 3600 and 7200. As for optimiser, we choose *Adam* [20] with parameters *betas* to (0.5, 0.999) and we set *weight_decay* to 5e-4 to prevent overfitting. We set prior factor λ to 0.5 and set the number of Gaussian kernels to 8.

Due to the huge difference in the numbers of different action types present in Rico, as shown in Table III. We set a different weight for different actions respectively. We compute the weights using the following equation,

$$weight_{action} = \frac{totalActionNum}{actionNum} * \frac{1}{actionTypeNum}$$

TABLE III
THE PERCENTAGE OF EACH ACTION TYPE IN RICO

Action Types	Percentage
CLICK	89.98%
LONG_CLICK	1.74%
SCROLL_LEFT_RIGHT	0.24%
SCROLL_RIGHT_LEFT	0.99%
SCROLL_TOP_DOWN	5.93%
SCROLL_BOTTOM_UP	1.12%

In this equation, $weight_{action}$ is the weight of loss function for specific action types. $totalActionNum$ is total labelled action in dataset. $actionNum$ is total number of specific labelled actions. $actionTypeNum$ is the total number of types of actions predicted. The final loss function is multiplied by $weight_{action}$.

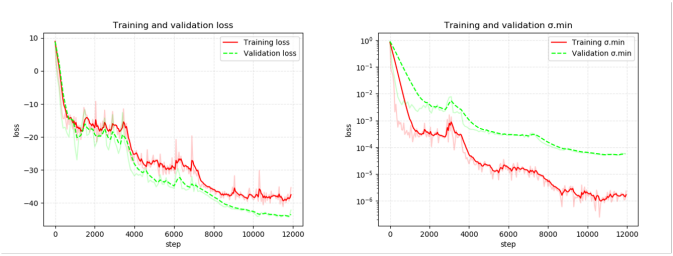

Fig. 5. The Training Loss Curve and Minimum Value of Output $\sigma.min$ curve

The training loss curve is shown in Figure 5 (left). We also record the minimum value of output σ during training. As shown in Figure 5 (right), $\sigma.min$ keeps converging, which means that the Gaussian peaks become narrow and reduce the randomness and uncertainty of actions. This is in line with our expectations that the model should concentrate on several hotspot actions instead of diverging randomly.

2) Migrate to the Real Test Environment: In this section, we describe how to deploy our model into a real automated testing environment. We set an actuator in the app to execute the action and send a request to the model deployed on GPU.

During the testing, as the model needs a history state sequence to predict the action on the current state, after the first three random executions we can then setup the testing model. By using the predicted action and current state as history states, it drives itself to explore the app automatically. The algorithm is shown in Algorithm 1.

For better illustration, we pick some output of the model generated for the daily news app, Toutiao. It is worth noting that in this app, some reasonable actions on the feed page are to scroll up to view more information cards, scroll left/right to switch content topics, and click to open a certain information card to read the content. We sample three times from the output distribution, hide the first three histories sequences and only show the current page. As shown in Figure 6, the model captures multiple possible behaviors.

IV. EMPIRICAL EVALUATION

In this section, we evaluate the test effectiveness and stability of MUBot on large and sophisticated commercial apps by comparing with Monkey and Humanoid. In addition, we

TABLE IV
SUMMARY OF SUBJECT APPS IN THE EXPERIMENT DATASET

App Name	Package Size	Num. of Activities	Description
Xigua[1]	45 Mb	335	Xigua Video is a short video platform that hosts a variety of video clips that are on average 2–5 minutes long.
Open Language[1]	62.9 Mb	145	Open Language is an language education app.
Tomato Novels[1]	28.8 Mb	246	Tomato Novels is a free novel reading app that provides novel reading, downloading, interest recommendation, video book, audio book, etc.
Facebook[2]	49.4 Mb	720	Facebook is a social app where users can post pictures, videos, stickers on the timeline or send to other users.
Instagram[2]	34.6 Mb	122	Instagram allows users to create and share photos and videos with friends and followers and discover other users with similar interests.
Youtube[2]	96.9 Mb	49	YouTube is an online video platform for users to upload and watch videos. Users can also rate, share and comment on videos.
Taobao[2]	186.3 Mb	545	Taobao is the largest online shopping platform in China. Users can browser products, make orders and track logistics.

[1] ByteDance apps with instrumented versions and line coverage measurement capabilities.
[2] Other commercial apps.

Algorithm 1 Testing Algorithm

1: Set $s = state$, $a = action$, $r = random(0, 1)$, $BACK_{PROB} = 0.05$
2: Operate app randomly to get first three state-action pairs and current state
$input = [< s_t, a_t >, < s_{t+1}, a_{t+1} >, < s_{t+2}, a_{t+2} >, < s_{t+3}, null >]$
3: **repeat**
4: Parse the state-action to gray-scale map and action code vector.
5: Model inference given $input$ and get $output$ (distribution parameters).
6: Sample from $output$ distribution and get predicted action a_{t+3}.
7: Execute action a_{t+3}
8: Get new state s_{t+4}.
9: $input \leftarrow [< s_{t+1}, a_{t+1} >, < s_{t+2}, a_{t+2} >, < s_{t+3}, a_{t+3} >, < s_{t+4}, null >]$
10: $t \leftarrow t + 1$
11: **until** Stop the testing

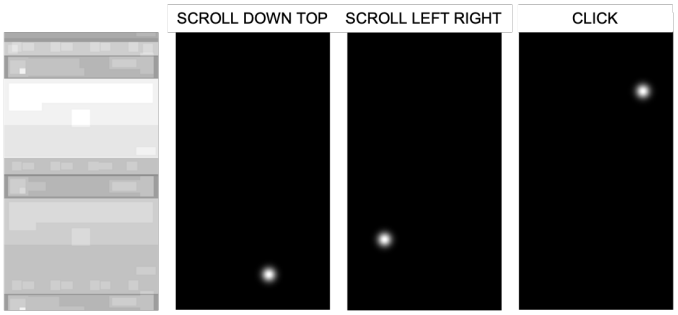

Fig. 6. Predicted Actions of the Feed Page in Toutiao. Left: Gray-scale structure of feed page. Right: Predicted actions position and type.

also report the industrial deployment of MUBot in the context of the daily testing of our apps with the help from their developers and provide developers' observations and lessons learned. We investigate the following research questions:

RQ1. Model Improvement: *Can the single-channel GUI representation and multi-modal deep learning model reduce the model size and time taken to generate actions?* To answer this question, we train models of Humanoid and MUBot using the same Rico dataset. We compare model parameters after training and record average time used to generate GUI actions during testing selected apps.

RQ2. Coverage Achieved: *Is MUBot able to achieve better code and activity coverage than Monkey and Humanoid?* Similar to the case study, for each subject app, we run Monkey, Humanoid and MUBot to generate GUI actions and explore GUI models. The experiment is set to one hour for all apps. We compare the rate of code lines and activities covered by these tools for our apps as we were able to instrument these apps with line coverage measurement capabilities. For the remaining commercial apps, we can only report activity coverage.

RQ3. Industrial Deployment: *Can MUBot visit more hotspot activities, uncover more crashes than our in-house model-based testing tool and achieve higher crash-fixing rate in daily test tasks?*
This question is investigated by first measuring the heat (number of hits) of activities using 100, 000 app traces recorded from internal manual test cases. We then report the rate of hotspot activity visits, the ratio of fixed crashes to all reported crashes and the importance of reported crashes from users' perspective. We say a bug is fixed when the developer submitted a merge request with the issue ID, passed the code review, and the bug is not revealed by replaying the GUI action sequences and does not appear in future MUBot and FastBot runs.

Subject Apps. In our experiment, we use popular commercial apps from different categories as shown in Table IV. We use our 3 commercial apps with source code so that we can report line coverage by code instrumentation. To add apps from more categories, we selected social, video, and shopping apps with billions of daily active users. We believe these apps are complicated and are able to serve as meaningful challenges

for Android testing in the context of real industrial practice and user experience. We do not include utility apps as their contents are more static and simpler compared with these apps.

Experiment Setup. We measure code line and activity coverage on different models of Android devices (OPPO, VIVO, Huawei and Google Pixel) with different Android versions (Android 8, 9 and 10) with the same account logged in. We set the minimum interval to generate actions to 800 milliseconds, in case that either tool can generate actions in noticeable short time, to avoid false positive crashes brought by running too fast, which is unlikely to appear in human-like test sequences. Each task runs three times without any human intervention. We implement the deep learning model using PyTorch and train the model on a Tesla V100 GPU for 5 hours.

A. RQ1. Model Improvement

As shown in Table V, MUBot took less parameter size and time to parse the GUI screen, infer the GUI model and generate the next action.

TABLE V
PARAMETER SIZE AND REASONING TIME

Tool (Model Configuration)	Million Parameters[1]	Reasoning Time (ms)
Humanoid	5.981	462.2
MUBot (K=8)[2]	0.947	52.28
MUBot (K=4)[2]	0.918	47.2

[1] The number of parameters each configuration took to train the model.
[2] The maximum number of possible actions.

After model training using the same dataset, model parameters generated by Humanoid is around 5 times more than MUBot. In addition, during the following experiment of testing apps, the average time used to generate one next action on the GUI model is 50 milliseconds by MUBot and 462 milliseconds for Humanoid, respectively.

Developers' Observation and Lessons Learned. The value of K (amount of sub-distributions) should be greater than possible actions at a given activity. By analysing hotspot activities and functionalities of our apps, the range is around 4 to 8. We do not choose a greater value because we want to capture most of the hotspot actions but avoid over-fitting users' long-sequence but low-frequency actions.

Although 462 milliseconds is an acceptable time to reason and generate one action when testing apps one at a time, in our production environment, GUI testing is deployed as a cloud service with hundreds or thousands real Android devices connected. The cloud service receives requests containing GUI screens from these devices and makes responses (GUI action for each device). However, deploying Humanoid in this cloud service is not feasible as it can only handle queries from no more than two devices during an one-second period.

B. RQ2.1 Line Coverage Achieved

In this research question, we evaluate whether our new GUI representation and multi-modal model can improve line and activity coverage compared with Monkey and Humanoid.

To better illustrate the performance over time, we report code line coverage every 5 minutes in the one-hour experiment period for each subject app and plot an average line coverage curve of the 3 experiment runs, as shown in Figure 7. MUBot has the best coverage after 20 minutes for all subject apps.

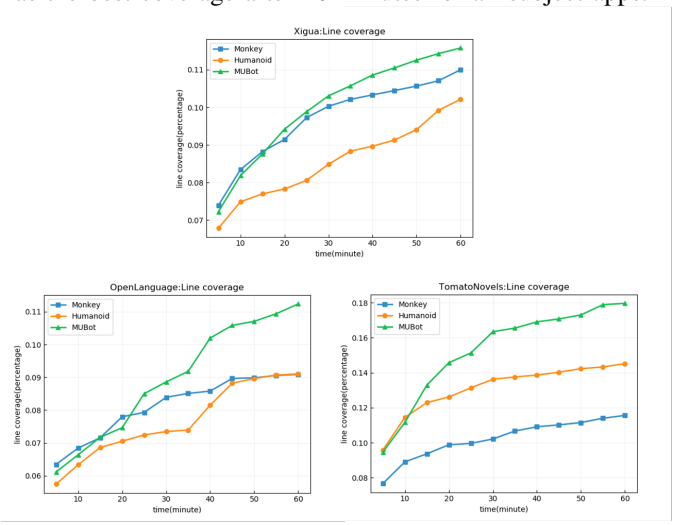

Fig. 7. Average Line Coverage

As these apps are complex with many interactive elements in the same page which is a catastrophe for the single-modal used by Humanoid, it achieves an even worse coverage than Monkey in some of the subject apps (Xigua Video and Open Language). he widget encoding strategy used by Humanoid maintains widget types in its model and produces uneven distribution of widget types when the complexity of apps increases, resulting in fewer code line coverage when priority is given to certain types of widgets. On the other hand, Monkey, as a random testing tool, is hard to explore some deeper pages and status. However, this situation often occurs in complex commercial apps. By contrast, the single-channel grey-scale GUI abstraction of MUBot avoids the uneven distribution without losing the accuracy of layouts and widgets of the GUI.

C. RQ2.2 Activity Coverage Achieved

We are able to measure activity coverage for all subject apps in this experiment. As shown in in Table VI, MUBot achieves the best activity coverage for 6 out of 7 apps. For Xigua on which Monkey has a better performance, our investigation reveals the underlying reason. In fact, large commercial apps have more interactions and logic code within the same activity. It means activity coverage is a rough statistic. In other words, real user interactions tend to stay in the primary activity to enjoy the main functionality provided by the subject app. Corner activities such as ABOUT THE APP, APP UPDATES, etc. are relatively rarely covered by daily usage.

Humanoid is stuck in some activities, resulting in a low activity coverage. By contrast, the randomness of Monkey gives itself the ability to explore rarer activities even though these activities are seldom visited in real user practice.

Developers' Observation and Lessons Learned. In stead of simply elevating activity coverage, developers are more con-

TABLE VI
ACTIVITY COVERAGE ACHIEVED FOR SUBJECT APPS

Apps	Monkey	Humanoid	**MUBot**
Xigua	**5.05**%	4.18%	4.38%
Open Language	5.17%	4.48%	**5.80**%
Tomato Novels	5.20%	4.84%	**6.51**%
Facebook	2.69%	2.30%	**3.48**%
Instagram	4.80%	4.55%	**6.28**%
Taobao	4.27%	4.64%	**5.03**%
YouTube	14.51%	Failed	**15.31**%

cerned with meaningful interactions within hotspot activities that have most important functionalities of these apps.

D. RQ3. Hotspot Activity Coverage and Crash-fixing Rate

To better understand the importance of human-like GUI interactions, we report the industrial deployment of MUBot to test the daily news app, Toutiao, in terms of hotspot activity coverage and crash fixing rate, compared with our existing model-based testing tool Fastbot [8]. The duration of the testing is set to three hours and is repeated three times for each tool. Fastbot supports running GUI exploration and action execution using multiple devices in parallel and provides two modes: independent mode (treating each device as stand-alone testing tasks) and collaborative mode (sharing explored data across all the devices). We use both mode of Fastbot using 50 Android devices and set the interval of test input generation to 800 milliseconds for both MUBot and FastBot.

We present our findings as follows:

1) Hotspot Activity Coverage: The frequency of visits to user hotspot activities achieved by MUBot, Fastbot independent mode and Fastbot collaborative mode is 53.1%, 25.7% and 21.2% respectively. It is reasonable that MUBot is able to prioritise the action execution that are likely to lead to user-preferred activities owing to the deep learning model.

2) Crash-fixing Rate: Crash-fixing rate is regarded as confidential information but we are able to report that the crash-fixing rate for bugs reported by MUBot is nearly double the rate of Fastbot.

3) The Value of Uncovered Crashes from Users' Perspective: The feedback from the quality assurance team reports that among all the end users that are already affected by crashes reported by these tools, 62% of the users are affected by crashes reported MUBot, while only 38% of them are affected by crashes uncovered by Fastbot. In addition, in terms of the frequency of users' triggering these reported crashes, 63% of the triggers that associated with the crashes revealed by MUBot while the remaining 37% are related to Fastbot.

V. RELATED WORK

In ths section, we summarise existing work on Android GUI testing, split into random, model-based, search-based and machine learning based techniques.

a) Random Android GUI Testing: Android Monkey [7] is a popular tool that randomly select actions available on the current GUI page.According to empirical evaluations on open-source apps [21] and commercial apps [22], Android Monkey is still the state-of-the-practice in many situations.

b) Android Model-based and Search-based GUI testing: Kong et al. [23] summarise that 63% of existing Android testing publications use model-based methods. DroidBot [24] and DroidMate [25] uses predefined GUI model representation strategy and guide action selection based on the GUI representation during testing. In contrast to the fixed model abstraction, APE [26] dynamically switches between coarser and finer levels of GUI abstraction on-the-fly. Stoat [1] has a two-phase approach including a finite state model construction using weighted GUI exploration and a Gibbs sampling through iteratively mutating the probabilistic model. In terms of search-based testing, Sapienz [9] uses a Pareto multi-objective approach via genetic evolution. To improve the evolution efficiency, TimeMachine [27] undertook test replay by dumping and loading status of a whole emulator at certain states that are evaluated as valuable in terms of code coverage.

c) Machine Learning based GUI testing: Existing machine learning based methods include deep learning and reinforcement learning techniques. As mentioned in Section II, Humanoid [10] proposes a deep learning approach by learning from human interactions but suffers from heavy model and low efficiency problems. Reinforcement learning techniques are also used to promote the effect of testing [28], [29] but is not suitable to learn from existing user actions. In addition, the deep Q-network which is the combination of deep and reinforcement learning is also explored in Android GUI testing to help explore functionalities that can only be accessed through a specific sequence of actions [30], [31].

However, these techniques focus on exploring more activities and GUI states, omitting the fact that hotspot activities and primary functionalities are also critical for testing.

VI. LIMITATIONS AND FUTURE WORK

Deep Learning. Although being effective to generate meaningful GUI action sequences and improve hotspot activities, we consider the followings as limitations of our deep learning approach and propose some feasible future work.

- **Interpretability** of machine learning models and underlying neural networks is an acknowledged difficult task by the machine learning community [32], [33] and lacks suitable metrics to assess the quality of explanations [34]. Therefore, the model only generates action sequences without giving the reason, which makes the analysis of these sequences to improve the model difficult.
- **Randomness** hinders the reproducibility. We design and use the multivariate mixture Gaussian distribution to predict multiple possible actions to avoid infinite loops and encourage different exploration paths. However, the random sampling from the Gaussian distribution makes test runs unreproducible.

GUI Representation Improvement. For GUI representation, the gray-scale map is an efficient way to represent a page. However, it still loses some information such as activates state, guided text and icons. We plan to add more information in the map and get a more accurate and detailed status description which can serve as a good ground for the following inference.

VII. Conclusion

We present MUBot based on deep learning for automated GUI testing on sophisticated commercial apps. For GUI representation, MUBot abstracts pages into a slim gray-scale maps to reflect GUI elements more efficiently. MUBot learns from interactive traces and imitates real users using its multi-modal deep learning algorithm. We empirically evaluated the test effectiveness of MUBot by comparing it with Monkey and Fastbot using 3 ByteDance apps and 4 other popular commercial apps. We also discussed developers' observations, lessons learned and the industrial deployment of MUBot. We made the following observation in our experiment:

1) Model parameter used by MUBot is 10 times less than Humanoid and the time used for generating a new action is reduced from 6 seconds to 300 milliseconds.
2) MUBot achieves the best line coverage for all 3 apps with source code instrumentations and best activity coverage for 6 out of 7 subject apps.
3) Bugs uncovered by MUBot are found to be more critical by the product development and quality assurance team.

References

[1] 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 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 245–256.

[2] Y.-M. Baek and D.-H. Bae, "Automated model-based android gui testing using multi-level gui comparison criteria," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 238–249.

[3] T. Takala, M. Katara, and J. Harty, "Experiences of system-level model-based gui testing of an android application," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2011, pp. 377–386.

[4] T. Su, "Fsmdroid: guided gui testing of android apps," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2016, pp. 689–691.

[5] O. Riganelli, S. P. Mottadelli, C. Rota, D. Micucci, and L. Mariani, "Data loss detector: automatically revealing data loss bugs in android apps," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 141–152.

[6] C. Peng, A. Rajan, and T. Cai, "Cat: Change-focused android gui testing," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 460–470.

[7] G. Developers, "Monkey," https://developer.android.com/studio/test/monkey, accessed: 2022-02-20.

[8] T. Cai, Z. Zhang, and P. Yang, "Fastbot: A multi-agent model-based test generation system beijing bytedance network technology co., ltd." in *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*, 2020, pp. 93–96.

[9] 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, pp. 94–105.

[10] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: A deep learning-based approach to automated black-box android app testing," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1070–1073.

[11] 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, pp. 845–854.

[12] M. V. Valueva, N. Nagornov, P. A. Lyakhov, G. V. Valuev, and N. I. Chervyakov, "Application of the residue number system to reduce hardware costs of the convolutional neural network implementation," *Mathematics and Computers in Simulation*, vol. 177, pp. 232–243, 2020.

[13] W. Zhu, C. Lan, J. Xing, W. Zeng, Y. Li, L. Shen, and X. Xie, "Co-occurrence feature learning for skeleton based action recognition using regularized deep lstm networks," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, no. 1, 2016.

[14] W. Zaremba, I. Sutskever, and O. Vinyals, "Recurrent neural network regularization," *arXiv preprint arXiv:1409.2329*, 2014.

[15] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[16] C. M. Bishop, "Mixture density networks," 1994.

[17] C. Li and G. H. Lee, "Generating multiple hypotheses for 3d human pose estimation with mixture density network," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 9887–9895.

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[19] W. Hu, L. Xiao, and J. Pennington, "Provable benefit of orthogonal initialization in optimizing deep linear networks," *arXiv preprint arXiv:2001.05992*, 2020.

[20] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[21] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?" in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 429–440.

[22] 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 *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 738–748.

[23] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein, "Automated testing of android apps: A systematic literature review," *IEEE Transactions on Reliability*, vol. 68, no. 1, pp. 45–66, 2018.

[24] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 23–26.

[25] N. P. Borges, J. Hotzkow, and A. Zeller, "Droidmate-2: a platform for android test generation," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 916–919.

[26] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical gui testing of android applications via model abstraction and refinement," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 269–280.

[27] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 481–492.

[28] D. Adamo, M. K. Khan, S. Koppula, and R. Bryce, "Reinforcement learning for android gui testing," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018, pp. 2–8.

[29] T. A. T. Vuong and S. Takada, "A reinforcement learning based approach to automated testing of android applications," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018, pp. 31–37.

[30] F. Y. B. Daragh and S. Malek, "Deep gui: Black-box gui input generation with deep learning," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 905–916.

[31] 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.

[32] L. H. Gilpin, D. Bau, B. Z. Yuan, A. Bajwa, M. Specter, and L. Kagal, "Explaining explanations: An overview of interpretability of machine learning," in *2018 IEEE 5th International Conference on data science and advanced analytics (DSAA)*. IEEE, 2018, pp. 80–89.

[33] Z. C. Lipton, "The mythos of model interpretability: In machine learning, the concept of interpretability is both important and slippery." *Queue*, vol. 16, no. 3, pp. 31–57, 2018.

[34] D. V. Carvalho, E. M. Pereira, and J. S. Cardoso, "Machine learning interpretability: A survey on methods and metrics," *Electronics*, vol. 8, no. 8, p. 832, 2019.