UNIVERSITY OF CALIFORNIA,
IRVINE

Deep-GUI: Towards Platform-Independent UI Input Generation with Deep Reinforcement Learning

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Software Engineering

by

Faraz YazdaniBanafsheDaragh

Thesis Committee:
Professor Sam Malek, Chair
Assistant Professor Joshua Garcia
Assistant Professor Iftekhar Ahmed

2020

TABLE OF CONTENTS

Page

LIST OF FIGURES **iv**

LIST OF TABLES **v**

ACKNOWLEDGMENTS **vi**

ABSTRACT OF THE THESIS **vii**

1 Introduction **1**

2 Method **5**
2.1 Data Collection 7
2.2 Model 8
2.2.1 Input and Output 9
2.2.2 UNet 9
2.2.3 Transfer Learning 10
2.2.4 Training 12
2.3 Inference 12
2.4 Monkey++ 14

3 Evaluation **16**
3.1 RQ1. Line Coverage 17
3.2 RQ2. Cross-Platform Ability 20
3.3 RQ3. Transfer Learning Effect 21

4 Background & Related Work **23**
4.1 Android Input Generation 23
4.1.1 Context-Blind 23
4.1.2 Context-Aware 24
4.2 Cross-Platform and Black-Box UI Input Generation 24

5 Discussion & Future Work **26**
5.1 Multi-Step Cross-Platform Input Generation 27
5.2 Smarter Processing of Information 27
5.3 Regression Testing and Test Transfer 28

Bibliography **29**

LIST OF FIGURES

Page

1.1 Example Screens and Heatmaps 3

2.1 Deep-GUI Overview 6
2.2 Neural Network Architecture 10
2.3 Clustering Readout Example 13

3.1 Monkey++ vs. Google Monkey on Android Applications 19
3.2 Deep-GUI vs. Random Agent on Websites 20
3.3 Transfer Learning Effect 22

LIST OF TABLES

Page

3.1 Monkey++ vs. Google Monkey on Android Applications 18
3.2 Deep-GUI vs. Random Agent on Websites 21

ACKNOWLEDGMENTS

This work was supported in part by awards CCF-1618132 and CNS-1823262 from the National Science Foundation, and Academic Research Credit Program from Google Cloud.

ABSTRACT OF THE THESIS

Deep-GUI: Towards Platform-Independent UI Input Generation with Deep Reinforcement Learning

By

Faraz YazdaniBanafsheDaragh

Master of Science in Software Engineering

University of California, Irvine, 2020

Professor Sam Malek, Chair

Although many Android input generation tools with different paradigms have been proposed, many of them fail to surpass even the simplest form of testing, i.e. random testing, in terms of coverage. Moreover, almost all these tools assume specific structures about the environment under the test. For instance, they require an XML encoding of the UI elements, or access to the source code for static analysis. This, however, is not always possible, e.g. when an application is simply a wrapper that uses a web-view to show content, or when the source code is not available. Moreover, these assumptions prevent these tools from applying to other platforms, such as web or iOS. In other words, unless a testing tool is truly black-box and platform-independent, its applicability is greatly compromised.

In this work, we propose Deep-GUI as the first effort towards fully cross-platform and black-box automated input generation. Using the power of deep learning, Deep-GUI learns the valid interactions given only the applications' screenshots, and therefore does not need any implementation-specific information about the application under test. Moreover, since the data collection, training, and inference processes are performed independently of the platform, Deep-GUI can be used in other platforms. We implement our extension of Google Monkey called Monkey++ that uses Deep-GUI, and show its effectiveness over Google Mon-

key in crawling Android applications. Furthermore, we provide evidence for the ability of Deep-GUI to operate across platforms without the need to re-train it, and explore future directions that can use the idea behind Deep-GUI to give rise to the next generation of automated input generation tools.

Chapter 1

Introduction

Automatic input generators for Android applications have been a hot topic for the past decade in software engineering community, simply because there are many applications to them. They can not only be used for automatically testing the applications, but also for crawling thousands of applications, which is useful in large scale studies as well as for app stores that need to ensure the security of the applications on their platform. Based on the exact usage in mind, input generator tools can be very generic, and simply crawl applications [10, 16, 18], or can be specifically looking for some certain criteria to be fulfilled, such as reaching activities with specific attributes [6]. Nevertheless, however these tools try to do this, they always use some pieces of information specific to the platform. For instance, many tools use static analysis to find the right combination of interactions with the application under test (AUT) [4, 6, 15, 30], while other tools depend on the GUI layout model that the platform provides to find and interact with the widgets [2, 19, 3, 8, 11, 5, 13, 28, 31, 7]. While this enables these tools with great potential for efficiently exploring apps in platforms that provide these types of information, it is also an obstacle for using these tools in new environments. For instance, almost none of the mentioned tools can be used for platforms other than Android without re-implementing the majority of their code, if possible at all.

This is simply because they rely on syntax and semantics of specific pieces of information they process. Moreover, in tools such as A^{3}E [4] , the tools systematically depend on the design choices of Android, which makes them very much limited to only this operating system, and might even stop working in the next updates of it. Another type of limitation is when the AUT has some non-native components to it, e.g. activities that are just wrappers for websites. In these situations, unless explicitly being taken care of, the tools are unable to perform fully operational.

Because of such limitations, it is reasonable to invest in black-box and platform-independent input generation tools. Google Monkey is the only tool that operates without structural dependency on the platform, although it is explicitly implemented for Android. While this tool is basically a random input generator, many studies suggest Google Monkey outperforms many of the existing white/gray box tools [9]. This essentially means that there is much potential in exploring black-box input generation options, in that it appears the information coming from the platform does not add much to the input generation ability. However, Google Monkey is the most basic form of black-box input generation. It blindly interacts with the screen without knowing if its action is a valid one. This might work quite well in many applications where the probability of randomly choosing an action that makes sense is high, but not in other applications. For instance, take Figure 1.1. In Figure 1.1a, since most of the screen contains buttons, almost all of the times that Google Monkey decides to touch, it touches on something valid and therefore tests a functionality. However, in Figure 1.1b, it is much less probable for Google Monkey to successfully touch the one button that exists on the screen, and therefore it takes much longer for it to test its functionality.

This article proposes Deep Generation of UI Inputs (Deep-GUI), a deep-learning-based tool that helps in such situations. Deep-GUI is able to filter out the parts of the screen that are irrelevant with respect to a specific action, such as touch, and therefore increases the probability of correctly interacting with the application. For example, given the screenshot

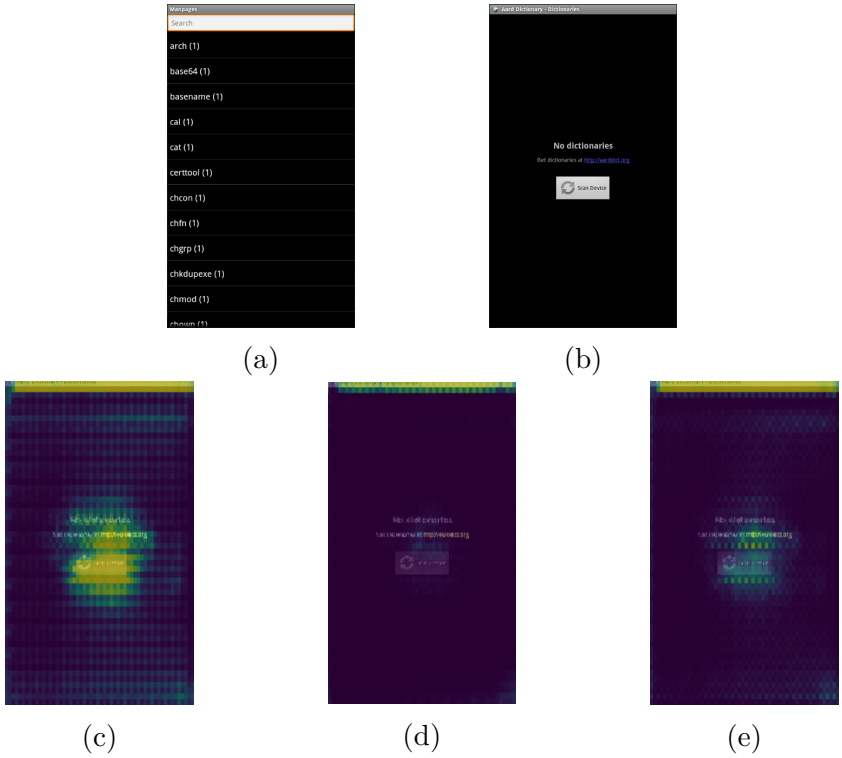


Figure 1.1: Two examples where it is respectively easy (a) and difficult (b) for Google Monkey to find a valid action, as well as the heatmaps generated by Deep-GUI associated with (b) for touch (c), scroll (d), and swipe (e) actions respectively. Note that in (c) the model correctly identifies both the button and the hyperlink –and not the plain text– as touchable.

shown in Figure 1.1b, Deep-GUI first produces the heatmap in Figure 1.1c, which shows for each pixel the probability of that pixel belonging to a touchable widget. Then it uses this heatmap to touch the pixels with higher values more often, hence increasing the chance of touching the button. In order to produce such heatmaps, Deep-GUI undertakes a deep learning approach. Moreover, what makes it unique is that Deep-GUI uses a completely black-box and cross-platform method to collect data, learn from it, and produce the mentioned heatmaps, and hence supports all situations, applications, and platforms. It also uses the power of transfer learning to make its training more data-efficient and faster. Our evaluations show that Deep-GUI is able to increase Google Monkey's performance on applications where Google Monkey struggles to find the valid actions. Also, we show that we can take a Deep-GUI model that is trained on Android, and use it on other platforms for efficient input generation. In summary, this article makes these contributions:

1. We propose Deep-GUI, a platform-independent method for automatically crawling an environment. To the best of our knowledge, this is the first tool that uses a completely black-box and cross-platform approach for data collection, training, and inference.

2. We provide Monkey++, an extension to Google Monkey that uses Deep-GUI to enhance the performance of Google Monkey in crawling Android applications.

3. We provide evaluations of Deep-GUI.

Chapter 2

Method

We here formally explain our definition of the problem of automatically generating inputs in an environment. Suppose that at each timestep t, the environment provides us with its state s_t. This can be as simple as the screenshot, or can be a more complicated content such as the UI tree. Also, suppose we define $A = \{\alpha_1, ...\alpha_N\}$ as the set of all possible actions that can be performed in the environment at all timesteps. For instance, in Figure 1.1b, all of the touch events associated with all pixels on the screen can be included in A. Note that these actions are not necessarily valid. We define a valid action as an action that results in triggering a functionality (like touching the send button) or changing the UI state (like scrolling down a list). Let us define $r_t = r(s_t, a_t)$ to be 1 if a_t is valid when performed on s_t, and 0 otherwise. Our goal is to come up with a function Q that, given s_t, produces the probability of validity for each possible action. That is, $Q(s_t, a_t)$ identifies how probable it is for a_t to be a valid action when performed on s_t. Therefore, Q is essentially a binary classifier (valid vs. non-valid) conditioned on s_t independently for each action in the set A. For simplicity, we also define $Q(s_t)$ as a function that, given an action α, returns $Q(s_t, \alpha)$. That is, $Q(s_t)(\alpha) = Q(s_t, \alpha)$

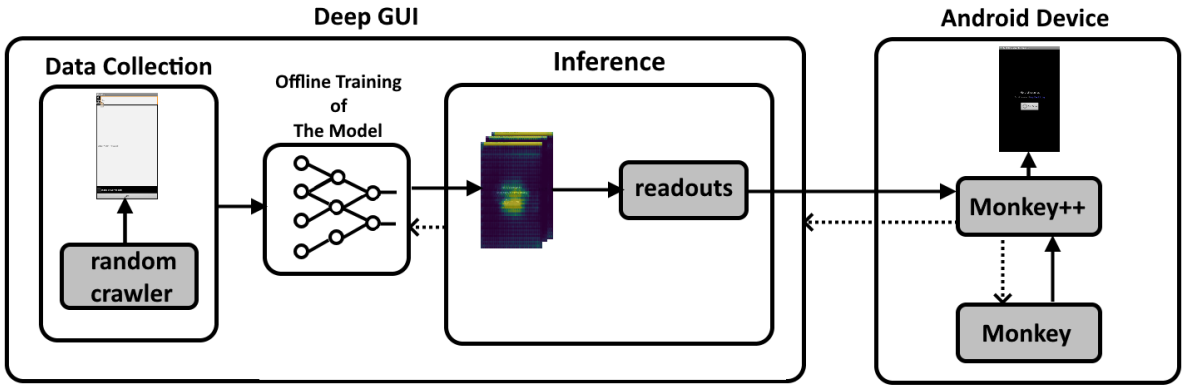


Figure 2.1: Overview of components comprising Deep-GUI & Monkey++

In Deep-GUI, we consider s_t to be the screenshot of AUT at each timestep. Set A consists of touch, up and down scroll, and right and left swipe events, on all of the pixels of the screen. We also define r_t as follows:

$$r(s_t, a_t) = \begin{cases} 0 & \text{if equals}(s_t, s_{t+1}) \\ 1 & \text{o.w.} \end{cases}$$

That is, if the screenshot undergoes a legitimate change after an action, we consider it to be a valid action in that screen. We define what a legitimate change means below. Note that we defined s_t, A, and r_t independent of the platform on which AUT operates. Therefore, this approach can be used in almost all existing environments.

This work consists of four components:

A. Data collection: This component helps in collecting necessary data to learn from.

B. Model: At the core of this component is a deep neural network that processes s_t and produces a heatmap $Q(s_t)$ for all possible actions a_t, such as the ones shown in Figure 1.1. The neural network is initialized with weights learned from large image classification tasks to provide faster training.

C. Inference: After training, and at the inference time, there are multiple readout mechanisms available for using the produced heatmaps and generating a single action. These mechanisms are used in a hybrid fashion to provide us with the advantages of all of them.

D. Monkey++: This is the only component that is specialized for Android, and its application is to fairly compare Deep-GUI with Google Monkey. Also, it provides a fast medium to use Deep-GUI in Android platforms as it can replace Google Monkey and be used almost the same way.

Figure 2.1 shows an overview of these four components and how they interact.

2.1 Data Collection

As we showed, we reduced the problem to a classification problem, therefore each datapoint in our dataset needs to be in the form of a three-way tuple (s_t, a_t, r_t), where our model tries to classify the pair (s_t, a_t) into one of the two values that r_t represents, i.e. whether performing the action a_t on the state s_t is valid or not. Since training a deep neural network requires a large amount of data, we cannot create this dataset manually. Therefore, we propose an automatic method to generate this dataset.

As defined above, we considered r_t to represent if the screen has a legitimate change after an action. We here define legitimate change as a change that does not involve an animated part of the screen. In other words, if specific parts of the screen change even in case of no interaction with the application, we filter those parts out when computing r_t. For instance, in Android, when focused on a textbox, a cursor keeps appearing and disappearing every second. But we filter out the pixels corresponding to the cursor.

For data collection, we first dedicate a set of applications to be crawled. Then, for each application, we randomly interact with the application with the actions in the set A and record the screenshot, the action, and whether the action resulted in a legitimate change. In order to filter out animated parts of the screen, before each action, we first record the screen for 5 seconds and consider all pixels that change during this period animated pixels. While this method does not fully filter all of illegitimate changes [1], as the results suggest, it is adequate.

A keen observer would realize that this method of data collection is a very natural choice to make. For instance, consider Android applications. For years, people have used Google Monkey to crawl Android applications for different purposes, but they never store the valuable data that it comes up with. Because of this, *even if a particular application has already been crawled by Google Monkey thousands of times before by other researchers, when a new researcher uses Google Monkey on that application, it still crawls randomly and makes all the mistakes that it has already seen hundreds of thousands of times.* The collection method described here is an attempt to share these experiences by training a model and making the model available to future researchers, as we discuss next.

2.2 Model

While, as discussed above, the problem is to classify the validity of a single action a_t when performed on s_t, it does not mean that each datapoint (s_t, a_t, r_t) cannot be informative about actions other than a_t. For instance, if touching a point results in a valid action, touching adjacent points may also result in a valid action with high probability. This can make our training process much faster and more data-efficient. Therefore, we need a model that can capture such logic.

[1] For instance, if an accumulative progress bar is being shown, this method does not work.

2.2.1 Input and Output

As the first step in order to do so, in our model, we define the input and output as follows. The input is a 3-channel image that represents s_t, the screenshot of the AUT at time t. For the output, we require our model to perform the classification task for all the actions, and not just a_t. While we do not directly use the prediction for other actions to generate gradients when training, this enables us to 1. use a more intuitive model 2. use the model at inference time by choosing the action that is most confidently classified to be valid. We use a T-channel heatmap to represent our output, T being the number of action types, i.e. touch, scroll, swipe. Note that we do not differentiate between up/down scroll or left/right swipe at this stage. Each channel is a heatmap for the action type it represents. For each action type, the value at (i, j) of the heatmap associated with that action type represents the probability that the model assigns to the validity of performing that action type on location (i, j). For instance, in Figure 1.1, the three heatmaps 1.1c, 1.1d, 1.1e show the model's confidence in performing touch, scroll, and swipe, respectively, at different locations of the screen.

2.2.2 UNet

We also would need a model that can intuitively relate the input and output, as defined above. We use a UNet architecture, as it has shown to be effective in applications such as image segmentation where the output is an altered version of the input image [23]. In this architecture, the input image is first processed in a sequence of convolutional layers known as the contracting path. Each of these layers reduces the dimensionality of the data while potentially encoding different parts of the information relevant to the task at hand. The contracting path is followed by the expansive path, where various pieces of information at different layers are combined using transposed convolutional layers [2] to expand

[2]In some references these are referred to as deconvolutional layers.

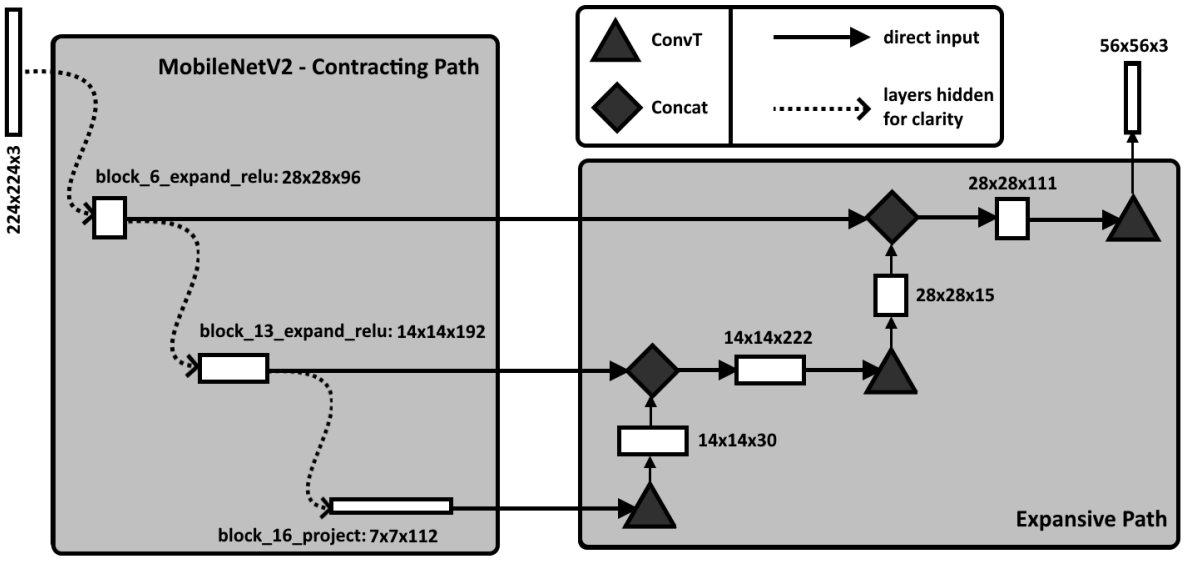


Figure 2.2: The deep neural network architecture used in Deep-GUI. The layers' names shown in MobileNetV2 are from Tensorflow [1] implementation of the architecture. `ConvT` is a transpose convolutional layer.

the dimensionality to the suitable format required by the problem. In our case, the output would be a 3-channel heatmap. In order for this heatmap to produce values between 0 and 1 (as explained above), it is processed by a sigmoid function in the last step of the model. As one can notice, because of the nature of convolutional and transposed convolutional layers, adjacent coordinate pairs are processed more similarly than other pairs. This makes it easier for the network to make deductions about all actions, and not just a_t. Moreover, the entire model seems to have an intuitive design: First, the relevant parts of information are extracted and grouped in different layers, and then combined to form the output. This is similar to how the UI elements are usually represented in software applications as a GUI tree.

2.2.3 Transfer Learning

While Google Monkey might struggle in finding valid actions when crawling an application, and while other tools might need to use other information such as GUI tree to detect such actions, humans find the logic behind a valid action pretty intuitive, and can learn it within

minutes of encountering a new environment. The reason behind this “intuition" lies in the much more elaborate visual experience that humans have that goes beyond the Android environments. Since birth, we see a myriad of objects in a myriad of contexts, and we learn to distinguish objects from their backgrounds. This information helps us a lot to distinguish a button in the background of an application, even if the background itself is a complicated image. Because of this, we humans do not need thousands of examples to learn to interact with an environment.

How can we use this fact to get the same training performance with fewer data in our tool? Research in machine learning has shown us that transfer learning can do this job [21]. In transfer learning, instead of a randomly initialized network, an existing model previously trained on a dataset for a potentially different but related problem is used as the starting point of all or some part of the network. This way, we “transfer" all the experience related to that dataset, without having invested time to actually process it. Therefore, training is more data-efficient. This is in particular important for us because, as discussed, the data collection process is very time-consuming given that the tool needs to monitor the screen for animations before collecting each datapoint.

The contracting path of the UNet seems like a perfect candidate for transfer learning because, unlike the expansive path, it is more related to how the network processes the input, rather than how it produces the output. This means that any trained model that exists for processing an image can be a candidate for us to use its weights.

In this work, as the contracting path, we used part of the network architecture MobileNetV2 [26] trained on the ImageNet dataset [25]. We chose MobileNetV2 because it is powerful and yet lightweight enough to be used inside mobile phones if necessary. Figure 2.2 shows how MobileNetV2 interacts with our expansive path to build the model used in Deep-GUI. Note that in order for the screenshot to be compatible with the already trained MobileNetV2

model, we first resize it to 224×224. Also, because of computational reasons, the produced output is 56×56, and is later upsampled linearly to the true screen size.

2.2.4 Training

At the training time, for each datapoint (s_t, a_t, r_t), the network first produces $Q(s_t)$ as the described heatmaps. Then, using the information about the performed action a_t, it indexes the network's prediction for the action to get $Q(s_t)(a_t) = Q(s_t, a_t)$. Finally, since this is a classification task, we use a binary crossentropy loss between r_t and $Q(s_t, a_t)$ to generate gradients and train the network. Please note that while we used an existing trained model as the initialization of the contracting path, we do train the weights on that path too.

2.3 Inference

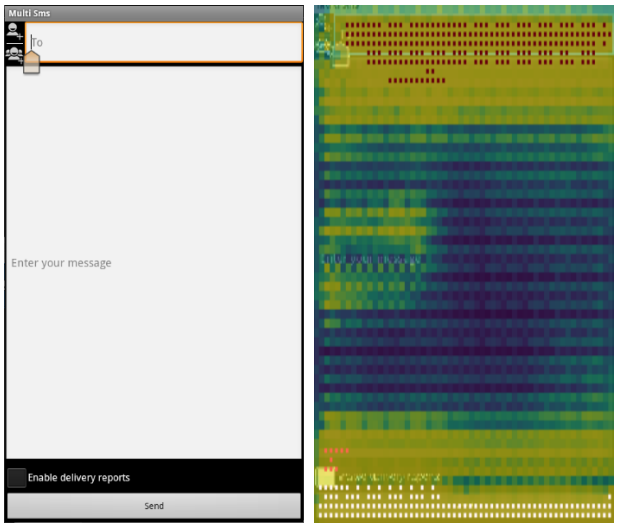
Once we have the trained model, we want to be able to use it to pick an action given a screenshot of an application at a specific state. Therefore, we require a readout function that can sample an action from the produced heatmaps. Here, we propose two readouts, and we explain how we use both in Deep-GUI.

The simplest possible readout is one that samples actions based on their relative prediction. That is, the more probable the network thinks it is for the action to be a valid one, the more probable it is for the action to be sampled. For this to happen, we need to normalize the heatmaps to a probability distribution over all actions of all types. Formally:

$$p(a_t = \alpha | s_t) = \frac{f(Q(s_t, \alpha))}{\sum_{\alpha' \in A} f(Q(s_t, \alpha'))}$$

where f identifies the kernel function. For instance if $f(x) = \exp(x)$, we have a softmax normalization. In our work, we chose to use the linear kernel $f(x) = x$. Using the probability distribution that the linear kernel produces, we then sample an action. We call this method the `weighted_sampling` readout.

However, humans usually interact with applications differently. We see widgets rather than pixels, and interact with those widgets as a whole. The `weighted_sampling` readout does not take this into account as it treats each pixel independently. Take Figure 2.3a as an example. The “Enable delivery reports" checkbox is potentially as important as the send button, because if it is checked a new functionality can be tested. However, because the button is larger than the checkbox, it takes the `weighted_sampling` readout longer to finally toggle the checkbox and test the new functionality.



(a) (b)

Figure 2.3: a: An example of a screen with equally important widgets of different sizes. b: The touch channel of the produced heatmap. The pixels belonging to different clusters that the `cluster_sampling` readout detects are colored with maroon, red, and white, depending on the cluster they belong to.

To address this issue, we use the `cluster_sampling` readout. In this approach, we first filter out all the actions α for which the predicted $Q(s_t, \alpha)$ is less than a certain threshold. This way, we make sure only the actions that are highly probable to be valid are considered. In Deep-GUI this threshold is 0.99. Then, for each channel in $Q(s_t)$, we use agglomerative clustering as implemented in python library `scikit-learn` [22] to cluster the pixels into widgets. Figure 2.3b shows the clustering result for the touch channel of the heatmap corresponding to Figure 2.3a. After detecting the clusters, we first randomly choose one of the action types, and

then randomly choose one of the clusters (i.e. widgets) in the channel associated with that action type. Finally, we choose a random pixel that belongs to that cluster and generate a_t.

While configurable, in our experiments we used a hybrid readout that uses `weighted_sampling` in 30% of the times, and `cluster_sampling` in 70% of the times. This way, we exploit the benefits that `cluster_sampling` offers, while we make sure we do not completely abandon certain valid actions because of the imperfections of the tool.

The discussed readouts identify the action type and the location of it on the screen. However, scroll and swipe also require other parameters such as direction or length. Deep-GUI chooses these parameters randomly. Also, because swipe and scroll are mostly used to discover other buttons, while touch is actually the action that triggers the functionality of the buttons, we configure the described readouts so that they are more biased towards choosing the touch action.[3]

2.4 Monkey++

While touch, swipe, and scroll are the most used action types when interacting with an environment, there are other actions that may affect the ability of a tool to crawl Android applications. In order to cover those actions as well, and also in order to be able to compare Google Monkey with our proposal fairly in Android environments, we introduce Monkey++, which is an extension to Google Monkey. Monkey++ consists of a server side, which responds to queries with Deep-GUI, and a client side, which is implemented inside Google Monkey.

Google Monkey works as follows. First, it randomly chooses an action type (based on the probabilities provided to it when starting it), and then randomly chooses the parameters

[3]In `weighted_sampling`, we multiply each heatmap belonging to touch, scroll, and swipe with 1, 0.3, and 0.1 respectively. In `cluster_sampling`, when randomly choosing an action type from the available ones, we use the same three numbers to bias the probability.

for those actions (such as the location to touch). Monkey++ works the same as Google Monkey with one exception. If the chosen action type is touch or gesture (which represents all types of movement, including scroll and swipe), instead of proceeding with the standard random procedure in Google Monkey, it sends a query to the server side. Using the inference procedure described above, Deep-GUI samples an action and returns to the client, which is then performed on the device. Algorithm 1 shows how Monkey++ works.

Algorithm 1: Monkey++ algorithm

```
while Google Monkey is running do
    get action type t from Google Monkey;
    if t is touch or gesture then
        get action a from Deep-GUI server
    else
        continue with Google Monkey and get action a
    end
    perform a
end
```

Chapter 3

Evaluation

We evaluated Deep-GUI with respect to the following research questions:

RQ1. What are the situations in which Monkey++ can surpass Google Monkey in terms of coverage?

RQ2. Is Deep-GUI actually cross-platform?

RQ3. How much is transfer learning helping Deep-GUI in learning better and faster?

We used the applications in the Androtest benchmark [9] as our pool of applications. Out of 66 applications available [1], we randomly chose 28 for training, 6 for validation, and 31 for testing purposes. We also eliminated one of the applications because of its incompatibility with our data collection procedure.[2]

To support a variety of screen sizes, we collected data from virtual devices of size 240×320 and also 480×854, and trained a single model that is used in the experiments explained

[1] Three applications caused crashes in the emulators and hence were not used.

[2] Application `org.jtb.alogcat` keeps updating the screen with new logs from the logcat regardless of the interactions with it, which highly deviates from the behavior of a normal Android application.

in sections RQ1 and RQ2. We collected an overall amount of 210,000 data points. Virtual devices, both for data collection and the Android experiments, were equipped with a 200MB virtual SD card, as well as 4GB of RAM. For data collection, training, and the experiments, we used an Ubuntu 18.04 LTS workstation with 24 Intel Xenon CPUs and 150GB RAM. We did not use GPU at any stage of this work. The entire source code for this work, the experiments, and the analysis is available at https://github.com/Feri73/deep-gui.

3.1 RQ1. Line Coverage

In order to test the ability of Monkey++ in exploring Android applications, we ran both Monkey++ and Google Monkey on each application in the test set for one hour, and monitored line coverage of the AUT every 60 seconds using Emma [24]. We ran 9 instances of this experiment in parallel, and calculated the average across different executions of each tool. Table 3.1 shows the final coverage for the applications in the test set. While in many applications Monkey++ and Google Monkey perform similarly, in some applications such as `com.kvance.Nectroid` Monkey++ outperforms Google Monkey significantly. We hypothesize that this is directly related to an attribute of applications we call non-trivial availability (NTA).

NTA identifies if an application offers anything to explore. Different factors can affect this value. For instance, if the majority of the application's code is executed at the startup there is not much available in the application to explore. As another example, consider applications that require signing in to an account to access their main functionality. Unless it is explicitly supported by the tools (which is not in this study), not much can be explored within the application.

Table 3.1: The applications in the test set as well as their final performances of Monkey++ and Google Monkey in them, sorted by NTA.

Application	NTA (bits)	Monkey++	G Monkey
es.senselesssolutions.gpl.weightchart	**2.8**	**%67**	**%65**
com.hectorone.multismssender	**2.6**	**%64**	**%67**
com.templaro.opsiz.aka	**2.4**	**%72**	**%66**
com.kvance.Nectroid	**2.3**	**%65**	**%50**
com.tum.yahtzee	**2.3**	**%67**	**%61**
in.shick.lockpatterngenerator	**2.2**	**%86**	**%84**
net.jaqpot.netcounter	**2.2**	**%71**	**%69**
org.waxworlds.edam.importcontacts	**2.0**	**%41**	**%34**
cri.sanity	**1.8**	**%25**	**%23**
com.chmod0.manpages	**1.7**	**%72**	**%63**
com.google.android.divideandconquer	1.5	%85	%88
com.example.android.musicplayer	1.3	%71	%71
ch.blinkenlights.battery	1.3	%91	%93
org.smerty.zooborns	1.2	%34	%33
com.android.spritemethodtest	1.2	%71	%87
com.android.keepass	1.1	%7	%8
org.dnaq.dialer2	1.0	%39	%39
hu.vsza.adsdroid	1.0	%24	%24
com.example.anycut	0.9	%71	%71
org.scoutant.blokish	0.9	%45	%46
org.beide.bomber	0.8	%89	%88
com.beust.android.translate	0.7	%48	%48
com.addi	0.6	%18	%18
org.wordpress.android	0.5	%5	%5
com.example.amazed	0.3	%82	%81
net.everythingandroid.timer	0.2	%65	%65
com.google.android.opengles.spritetext	0.1	%59	%59
aarddict.android	0.0	%14	%14
com.angrydoughnuts.android.alarmclock	0.0	%6	%6
com.everysoft.autoanswer	0.0	%9	%9
hiof.enigma.android.soundboard	0.0	%100	%100

com.tum.yahtzee: This is a dice game with fairly complicated logic and several buttons, each activating different scenarios over time.

org.waxworlds.edam.importcontacts: This application imports contacts from the SD card. There are multiple steps to reach to the final activity, and each contains multiple options that change the course of actions that the application finally takes.

hu.vsza.adsdroid: The only functionality of this application is to search for and list the data-sheets of electronic items. The search activity contains one drop-down list for search criteria, and a search button.

org.wordpress.android: This application is for management of WordPress websites. At the startup, it either requires a login or opens a web container, which does not affect the line coverage.

We hypothesize that Monkey++ outperforms Google Monkey in applications with high NTA. In order to test this, we define NTA as the uncertainty in coverage when randomly interacting with an application. That is, if random interactions with an application always result in a similar trace of coverage, it means that the available parts of the application are trivial to reach and will be executed with high certainty, and therefore, not much NTA is offered by the application. To compute uncertainty (and hence NTA) for an application, we take all line coverage information for that application in all timesteps of all experiments involving Google Monkey (as a random interaction tool), and calculate the entropy of the distribution of these coverage values. Table 3.1 shows the NTA (in binary bits) for each application, and discusses some examples of applications with high and low NTA. As one can notice, most of the applications in which Monkey++ achieves better coverage have higher NTA.

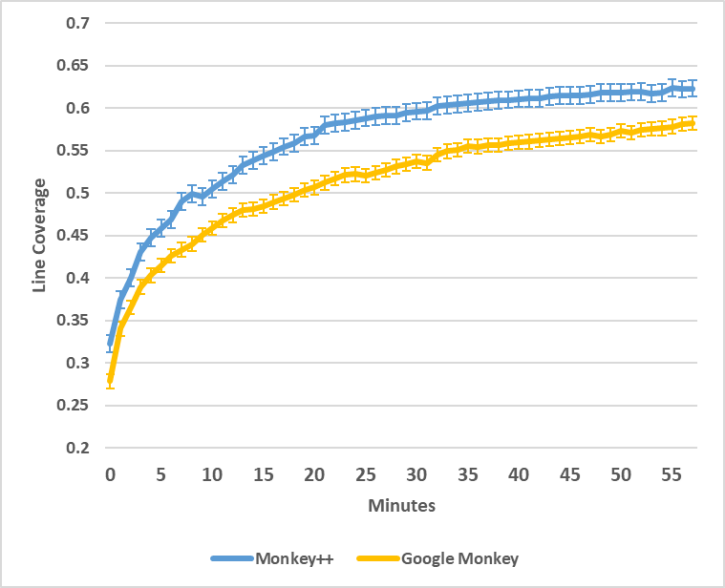


Figure 3.1: The progressive line coverage of Monkey++ and Google Monkey on the top 10 Android applications with regards to their NTA

To further demonstrate the ability of Monkey++ in crawling complex applications with high NTA, we analyzed the progressive coverage of the top 10 applications with regards to NTA. Figure 3.1 shows that Monkey++ achieves better results compared to Google Monkey, and does so faster. This superiority is statistically significant in all timesteps, as calculated by a one-tail Kolmogorov–Smirnov (KS) test (p-value < 0.05).[3]

[3]To calculate the error bars in Figure 3.1 and the p-value for KS-test, first for each application, the mean performance of Google Monkey on that application is subtracted from the performance of both Google Monkey and Monkey++, and then the error bars and the significance are computed with regards to this value across all applications.

3.2 RQ2. Cross-Platform Ability

Since the method we proposed is completely blind with regards to the application's implementation or the platform it runs on, we hypothesize it is applicable not only in Android but in other platforms such as web or iOS. Moreover, we claim that since UI design across different platforms is very similar (e.g. buttons are very similar in Android and web), we can take a model trained on one platform and use it in other platforms. This is particularly useful when developers want to test different implementations of the same application in different platforms.

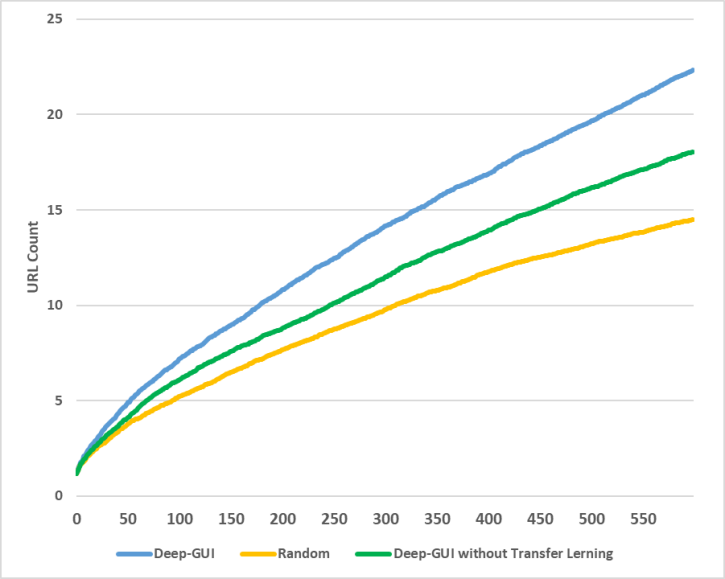


Figure 3.2: The progressive performance of Deep-GUI and random agent in web crawling. The difference between the three tools is statistically significant in all timesteps, as calculated by one tail KS-tests between all pairs (similar to the procedure described in footnote 3).

To test whether our approach is truly cross-platform, we implemented an interface to use Deep-GUI for interacting with Mozilla Firefox browser[4] using Selenium web driver [27], and compared it against a random agent[5]. Note that we did not re-train our model, and used the exact same hyper-parameters and weights we used for the experiments in RQ1, which are learned from Android applications.

For the web experiments, we used the top 15 websites in the US [12] as our test set, and ran each tool on each website 20 times, each time for 600 steps. To measure the performance,

[4]We used Responsive Design Mode in Mozilla Firefox with the resolution of 480×640

[5]The random agent uses the same bias for action types that is explained in footnote 3.

Table 3.2: The performance of Deep-GUI and random agent on each web site

Website	Deep-GUI	Random
google.com	17.4	12.9
youtube.com	94.3	12.1
amazon.com	13.2	15.2
yahoo.com	15.4	21.8
facebook.com	3.2	7.1
reddit.com	5.3	5.1
zoom.us	4.6	6.9
wikipedia.org	41.1	40.6
myshopify.com	3.6	6.0
ebay.com	13.4	11.4
netflix.com	5.1	4.8
bing.com	32.5	25.5
office.com	16.9	15
live.com	2.7	2.5
twitch.tv	65.6	30.1

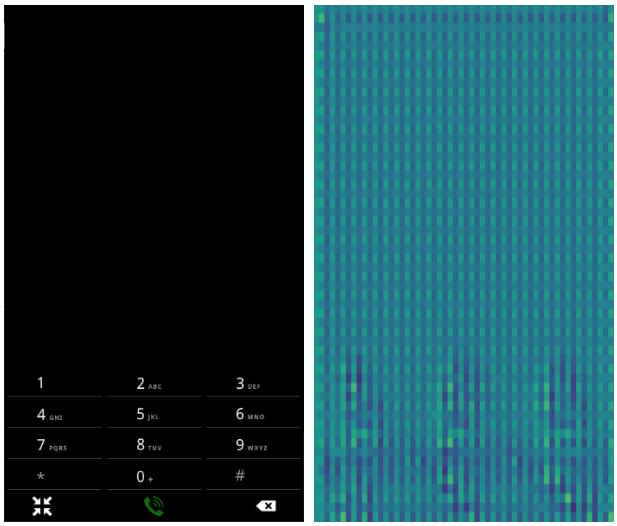
we counted the number of distinct URLs visited in each website, and averaged this value for each tool. Figure 3.2 and table 3.2 show that our model outperforms random agent, and confirms that our model has learned the rules of UI design, which is indeed independent of the platform.

3.3 RQ3. Transfer Learning Effect

As described, we used transfer learning to make the training process more data-efficient, i.e. we crawl fewer data and train faster. To study if using transfer learning was actually helpful, we repeated the web experiments, with the only difference that instead of using the model trained with transfer learning, we trained another model with random initial weights. Figure 3.2 shows that without transfer learning, the model's performance significantly decreases.

To see why this happens, take Figures 3.3b. This figure shows the initial output of the neural network on the screen in Figure 3.3a before training, when initialized with the ImageNet

weights. As one can see, even without training, the buttons stand out from the background in the heatmap, which gives the model a significant head-start compared to the randomly initialized model, and makes it possible for us to train it with a small amount of data.



(a) (b)

Figure 3.3: A screenshot and its corresponding heatmap generated by the model before training.

Chapter 4

Background & Related Work

4.1 Android Input Generation

Many different input generation techniques with different paradigms have been proposed in the past decade. However, they can be classified into two broad categories:

4.1.1 Context-Blind

The tools in this category process information in each action independent of other actions. That is, when choosing a new action, they do not consider the previous actions performed, and do not plan for future actions. Tools such as Google Monkey [10] and DynoDroid [16] lie in this category. These tools are fast and require very simple pre-processing, but they cannot guarantee to not miss activities or functionalities, as this requires more contextual information, such as maintaining the activities already visited.

4.1.2 Context-Aware

These tools integrate different aspects of local information in each activity to build a context, which is then used to plan for input generation. Most of the proposed input generation tools lie in this category. For instance, Sapienz [17] uses a genetic algorithm to learn a generic notion of context, which represents how certain sequences of actions can be more effective than the others. Tools that use different types of static analysis of the source code or GUI to model the information flow globally also lie in this category.

4.2 Cross-Platform and Black-Box UI Input Generation

Not many tools have explored black-box and/or cross-platform options for gathering information to be used for input generation, either with a context-aware or context-blind approach. Google Monkey is the only widely used tool that does not depend on any application-specific or platform-specific information. However, it is the most simple type of testing. Humanoid [14] is an effort towards becoming less platform-dependent while also generating more intelligent inputs. However, it is still largely dependent on the UI transition graph and the GUI tree extracted from the operating system. Also, since it depends on an existing dataset for Android, it would not be easy to train it for a new platform. The study of White et. al [29] is the most similar to this work. They study the effect of machine-learning-powered processing of screenshots in generating inputs with random strategy. However, because they generate artificial applications for training their model, their data collection method is limited in expressing the variety of screens that the tool might encounter, and is also still platform-dependent.

Deep-GUI uses deep learning to improve context-blind input generation, while also limiting the processed information to be black-box and platform-independent. This enables Deep-GUI to be as versatile as Google Monkey in the Android platform, while being more intelligent.

Chapter 5

Discussion & Future Work

Deep-GUI is the first attempt towards making a fully cross-platform test input generation tool. However, there are multiple areas in which this tool can be improved. The first limitation of the approach described here is the time-consuming nature of its data collection process which limits the number of collected data points and may compromise the dataset's expressiveness. By using transfer learning, we managed to mitigate this limitation to some degree. Also, the complex set of hyperparameters (such as hybrid readout probabilities) and the time-consuming nature of validating the model on applications makes it difficult to fine-tune all the hyperparameters systematically, which is required for optimizing the performance to its maximum potential.

Deep-GUI limits itself to context-blind information processing, in that it does not consider the previous interactions with AUT when generating new actions. However, it uses a paradigm that can easily be extended to take context into account as well. We believe this paradigm should be explored more in the future of the field of automated input generation.

Take our definition of the problem. If we call s_t the state of the environment, a_t the action performed on the environment in that state, r_t the reward that the environment provides

in response to that action, and $Q(s_t, a_t)$ the predictions of the model about the reward that the environment provides when performing a_t in s_t, then this is essentially a single-step reinforcement learning (RL) definition of the problem of test input generation, with a deep Q-Learning [20] solution to it. Looking at the problem this way enables researchers in the area of automatic input generation to benefit from the rich and active research in the reinforcement learning community, and explore different directions in the future such as the followings:

5.1 Multi-Step Cross-Platform Input Generation

Deep-GUI uses the RL definition of the problem in a context-blind manner. However, by re-defining s_t to include more context (such as previous screenshots, as tried in Humanoid) and expanding the definition of r_t to express a multi-step sense of reward, one can use the same idea to train models that not only limit their actions to only the valid ones (as this tool does), but also plan ahead and perform complex and meaningful sequences of actions.

5.2 Smarter Processing of Information

Even if a tool does not want to limit itself to only platform-independent information, it can still benefit from using a Q-Learning solution. For instance, one can define s_t to include the GUI tree or the memory content to provide the model with more information, but also use Q-Learning to process this information more intelligently.

5.3 Regression Testing and Test Transfer

While this work presents a trained model that targets all applications, it is not limited to this. Developers can take a Q-Learning model such as the one described in this work, collect data from the application (or a family of related applications) they are developing, and train the model extensively so that it learns what actions are valid, what sequences of actions are more probable to test an important functionality, etc. This way, when new updates of the application are available, or when the application becomes available in new platforms, developers can quickly test for any fault in that update without having to re-write the tests.

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 265–283, USA, 2016. USENIX Association.

[2] D. Amalfitano, A. R. Fasolino, and P. Tramontana. A gui crawling-based technique for android mobile application testing. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 252–261, 2011.

[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 *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 258–261, 2012.

[4] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, page 641–660, New York, NY, USA, 2013. Association for Computing Machinery.

[5] Y. Baek and D. Bae. Automated model-based android gui testing using multi-level gui comparison criteria. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 238–249, 2016.

[6] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall. Brahmastra: Driving apps to test the security of third-party components. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, page 1021–1036, USA, 2014. USENIX Association.

[7] N. P. Borges, 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*, MOBILESoft '18, page 133–143, New York, NY, USA, 2018. Association for Computing Machinery.

[8] W. Choi, G. Necula, and K. Sen. Guided gui testing of android apps with minimal restart and approximate learning. *SIGPLAN Not.*, 48(10):623–640, Oct. 2013.

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

[10] A. Developers. Ui/application exerciser monkey, 2012.

[11] 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 *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, page 204–217, New York, NY, USA, 2014. Association for Computing Machinery.

[12] A. Internet. Top sites in united states, 2020.

[13] K. Jamrozik and A. Zeller. Droidmate: A robust and extensible test generator for android. In *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 293–294, 2016.

[14] 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)*, pages 1070–1073, 2019.

[15] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk. Mining android app usages for generating actionable gui-based execution scenarios. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, page 111–122. IEEE Press, 2015.

[16] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 224–234, New York, NY, USA, 2013. Association for Computing Machinery.

[17] 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*, ISSTA 2016, page 94–105, New York, NY, USA, 2016. Association for Computing Machinery.

[18] K. Mao, M. Harman, and Y. Jia. Crowd intelligence enhances automated mobile testing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, page 16–26. IEEE Press, 2017.

[19] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro. Autoblacktest: Automatic black-box testing of interactive applications. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 81–90, 2012.

[20] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *ArXiv*, abs/1312.5602, 2013.

[21] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010.

[22] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[23] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham, 2015. Springer International Publishing.

[24] V. Roubtsov. Emma: a free java code coverage tool, 2006.

[25] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[26] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

[27] Selenium. The selenium browser automation project. `https://www.selenium.dev/`.

[28] 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*, ESEC/FSE 2017, page 245–256, New York, NY, USA, 2017. Association for Computing Machinery.

[29] T. D. White, G. Fraser, and G. J. Brown. Improving random gui testing with image-based widget detection. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 307–317, New York, NY, USA, 2019. Association for Computing Machinery.

[30] S. Yang, H. Wu, H. Zhang, Y. Wang, C. Swaminathan, D. Yan, and A. Rountev. Static window transition graphs for android. *Automated Software Engg.*, 25(4):833–873, Dec. 2018.

[31] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 23–26, 2017.