Textout: Detecting Text-layout Bugs in Mobile Apps via Visualization-oriented Learning

Yaohui Wang*§, Hui Xu†‡, Yangfan Zhou*§, Michael R. Lyu†‡, Xin Wang*§
* School of Computer Science, Fudan University, Shanghai, China
† Dept. of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China
‡ Shenzhen Research Institute, The Chinese University of Hong Kong, Shenzhen, China
§ Shanghai Key Laboratory of Intelligent Information Processing, Shanghai, China

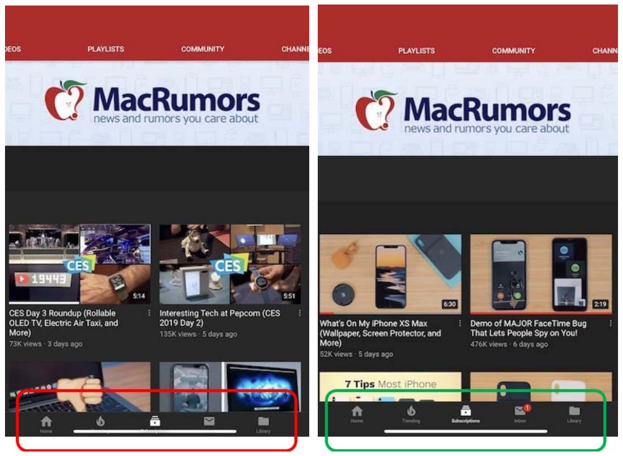
***Abstract*—Layout bugs commonly exist in mobile apps. Due to the fragmentation issues of smartphones, a layout bug may occur only on particular versions of smartphones. It is quite challenging to detect such bugs for state-of-the-art commercial automated testing platforms, although they can test an app with thousands of different smartphones in parallel. The main reason is that typical layout bugs neither crash an app nor generate any error messages. In this paper, we present our work for detecting text-layout bugs, which account for a large portion of layout bugs. We model text-layout bug detection as a classification problem. This then allows us to address it with sophisticated image processing and machine learning techniques. To this end, we propose an approach which we call Textout. Textout takes screenshots as its input and adopts a specifically-tailored text detection method and a convolutional neural network (CNN) classifier to perform automatic text-layout bug detection. We collect 33,102 text-region images as our training dataset and verify the effectiveness of our tool with 1,481 text-region images collected from real-world apps. Textout achieves an AUC (area under the curve) of 0.956 on the test dataset and shows an acceptable overhead. The dataset is open-source released for follow-up research.**

***Index Terms*—mobile application testing, GUI testing, GUI bug detection, text-layout bug detection, deep learning**

I. Introduction

Bug detection poses a continuing challenge for software development. Developers have to cope with a variety of bugs, such as crash bugs [1], performance bugs [2], [3], and layout bugs [4]. Traditional software testing concentrates on detecting crash and performance bugs, and the detection of layout bugs is a secondary concern. However, with the rapid growth in the number of smartphone users in recent years [5], it is highly desirable for mobile apps to provide more convenient and easy-to-use graphical user interfaces (GUI). Hence, layout bugs have become far more critical for mobile apps than they were previously.

Nevertheless, layout bugs are very common in real-world apps. There are many complaints in app stores [6] and internet media reporting the display issues of apps on particular devices. For example, Figure 1 demonstrates a recent layout bug of YouTube app when running on iOS (*i.e.*, the home bar is overlapped with the texts of several buttons). On the one hand, the translation from source code to GUI, although normally conducted automatically, is complex. When developing apps, developers generally define static layouts and some basic components in standalone files (*e.g.*, `*.xml`)



(a) The buggy layout where the home bar is overlapped with the texts of several buttons.

(b) The fixed layout where the home bar no longer overlaps the texts of the buttons.

Fig. 1. A real-world layout bug of the Youtube app. The original report can be found on the following website: https://www.macrumors.com/2019/01/14/youtube-app-2018-ipad-pro-support/.

while implementing their dynamic transitions and program logic in other files (*e.g.*, `*.java`). It is very difficult for developers to gauge the actual visual effect of a GUI with precision until it is shown on the screen. On the other hand, the host smartphones running apps suffer heavy fragmentation issues, *i.e.*, there are many hardware and system versions [7]. Consequently, the layout may be normal on one device but malformed on another, which increases the difficulty of finding such bugs among various devices during testing. State-of-the-art commercial mobile testing platforms (*e.g.*, Testin[1]) can handle the fragmentation issue by automated and parallel testing with thousands of different smartphones, but find it hard to detect malformed layouts. The main reason is that a malformed layout neither crashes a program nor generates any error messages. Therefore, there is a pressing need to improve current testing practices in detecting layout bugs.

Automatic detection of layout bugs is challenging. Existing approaches mainly rely on image differing or layout property

[1]https://www.testin.net/

978-1-7281-4982-0/19/$31.00 ©2019 IEEE
DOI 10.1109/ISSRE.2019.00032

validation [8]. A major drawback of these approaches is that they always require human input as the ground truth and cannot be automatically extended to support new app versions and devices. Thanks to the recent advancement of deep learning techniques in image processing [9], we now have the opportunity to tackle the problem more efficiently, by training a classifier which can detect malformed layouts in much the same way as humans do.

Intuitively, we can train a binary classifier to discriminate normal and abnormal layout images. However, there are several major challenges to overcome. Training a deep learning classifier generally requires thousands or even millions of images. While collecting normal-layout images is easy, collecting a large number of abnormal-layout images is difficult. Furthermore, the cases of layout bugs vary a lot among each other and they may occupy only a small region of the image. It is challenging to train a classifier that can effectively capture the buggy regions and achieve promising accuracy.

In this work, we focus on a popular type of layout bugs related to texts. Because text often gives important instructions to users, such bugs are more noticeable and irritating (*e.g.,* Figure 1a). To detect text-layout bugs, we propose a deep learning-based approach which we call Textout. Given an app screenshot, Textout firstly detects the textual regions of the image, and segments the image into several rectangular areas. It then applies a binary classifier to each sub-image and detects if it contains layout issues. We adopt convolutional neural networks (CNN) as the classifier and train it with segmented images. Because segmented images provide more accurate information about a bug, we are more likely to train an elegant model with high accuracy. Furthermore, as a screenshot can be segmented into tens of clips, the number of raw screenshot images required for training can be reduced. To evaluate the performance of our approach, we manually collected 1,481 text-region images segmented from 21 buggy screenshots and 38 normal screenshots of real-world apps. Experimental results show that Textout can detect most of the buggy images with a low false positive rate. It achieves an AUC (area under the curve) of 0.956.

To summarize, this paper makes the following main contributions.

- We present the first attempt to study text-layout bugs.
- We try to detect text-layout bugs using screenshots and model text-layout bug detection as an image classification problem.
- We build a large dataset that contains a wide range of normal and abnormal text-region images.
- We propose a deep learning-based approach, which we call Textout, and empirically show that it is an effective way of detecting text-layout bugs.
- We release our dataset as open-source to facilitate follow-up research[2].

The remainder of this paper is organized as follows: Section II presents the related work of GUI layout testing. Section III explains the different causes and types of text-layout bugs. Section IV describes our approach to detecting text-layout bugs. Section V provides a detailed account of how Textout operates. Section VI describes our experiments and evaluations. Section VII discusses the practicability and also the limitations of Textout. Section VIII concludes.

[2]https://github.com/DillionApple/Textout-dataset

II. Related Work

Our work focuses on detecting layout bugs. Those bugs may break the application's visual effect and make the displayed content hard to distinguish. Below, we discuss previous studies related to layout testing in order to set our work in context and bring out its significance.

Some layout bug detection approaches focus on web pages, taking advantage of specific HTML features. For example, Mahajan *et al.* [10] employed image comparison to find presentation failures in HTML. This can map GUI components back to the corresponding HTML elements and identify the faulty HTML code, but a target design image with a fixed resolution is required for every screen page. Fighting Layout Bugs [11] is a library developed by Google for detecting layout bugs automatically in web pages. It uses CSS injection to detect text and component borders on a web page. This information can help to determine whether a text line crosses the borders of other components, but can only be used to detect this specific kind of bug.

Some investigations focus on detecting GUI layout bugs on mobile applications. Hasselknippe *et al.* [8] proposed a tool which they called LBH (Layout Bug Hunter). It monitors the layout hierarchy, position, and size of the GUI components and uses several rules to judge whether a layout bug is present. Their work tried to classify GUI layout bugs, but the classification cannot cover all GUI layout bugs. And LBH only works for applications written by Fuse, which is an application developing framework, because it depends on this framework to gain the real-time GUI layout information. It may also misjudge some aesthetic design features as GUI layout bugs.

Some investigations have used a deep learning approach. Lu *et al.* [12] proposed a software GUI testing tool based on DNN (Deep Neural Network). Their tool accepts software screenshots as input and evaluates its layout as either "good" or "bad". But it only cares about the aesthetics of the layout design and cannot detect layout bugs. Wang *et al.* [13] also proposed a DNN-based method for layout testing. But they need to train one DNN model for each of the GUI components, and rely on the source code to generate abnormal layout samples.

Industrial tools have also been developed for GUI layout testing. Applitools [14] captures the visual difference between the real GUI image and the target design image in the testing process and generates a graphic report for users. Users can then either accept or reject the result. Android Studio, the official IDE for Android development, provides a layout debugging tool called Android Inspector [15]. It allows developers to debug their GUI layout at runtime and provides a user interface by which developers can check the detail of the layout tree. There is also a similar tool [16] in the iOS platform.

Our work is different from previous work. We aim to detect text-layout bugs, which occur very commonly in application development. We analyze screenshots and do not require other information. We model text-layout bug detection as an image classification problem and use text detection and CNN classifier to achieve the goal.

III. Preliminaries of Text-layout Bugs

In this section, we first discuss the role of text in mobile applications and some main causes of text-layout bugs. Then we try to categorize different types of text-layout bug.

A. Text Layouts for GUI Construction

Text is the most important way for users to gain information. In mobile applications, many GUI components have text displayed on them. These texts function in different ways. For example, they can supply the main content, navigate users around the application and prompt them to take desired actions. However, bad text layout may cause the GUI to be messy and the text information hard to distinguish. This significantly compromises the user's experience.

Text is usually displayed in a rectangular GUI component, named text view, which can be integrated with several other components. Below, we list some commonly used components that may contain text view. These concepts apply both to Android and iOS applications.

1) Button: Button is a common GUI component in mobile applications. It usually contains short text which indicates the related action bound with the button, such as `submit`, `accept`, `download` and `cancel`.

2) Navigation Bar: The navigation bar is used to indicate the user's current position in the application. It is usually displayed at the top of the screen. The name of the current screen is usually displayed in the center of the navigation bar. There may also be some buttons on its left side or right side which can guide the user to other screens.

3) Tab Bar: The tab bar is also called the bottom navigation bar. It usually contains several buttons aligned horizontally and positions at the bottom of the screen. It is used to separate different functions of an application into different screens. The icons and texts in the tab bar indicate the functions of the corresponding screens.

4) Table View: The table view is suitable for displaying list data, such as a contact list or music list. Each item in the list is displayed in a table cell. A table cell usually contains a title and sometimes a subtitle to indicate the data associated with it. Users can take further action by clicking the table cell.

B. Causes of Text-layout Bugs

There are many factors that may cause text-layout bugs. Below, we discuss several main causes of text-layout bugs.

1) Fragmentation: The host smartphones running apps suffer heavy fragmentation issues, *i.e.,* there are many hardware and system versions [7]. Text-layout bugs can be easily caused due to fragmentation. For example, different devices have different screen resolutions. When the screen width narrows, the width of the text view may narrow accordingly. As the content in the text view is not changed, the text view could be stretched vertically. If this is not handled appropriately, unexpected text overlapping issues may occur.

2) Unexpected Long Texts: Occasionally, the text content can be very long and cause display issues. Such cases typically occur when the text content is dynamically fetched from the Internet or obtained from user inputs. If developers have not considered the corner cases properly, the text rendered in a GUI component could be very long, often overflowing the expected region and overlapping with adjacent text.

3) Different System Settings: To increase the accessibility of smartphones, many manufacturers provide an interface that allows users to change the system font size. Older people or visually impaired people tend to increase the system font size on their smartphones. At the same time, the font size of some applications is not fixed, but changes dynamically according to the system font size settings. This may cause the text view on the GUI component to expand both horizontally and vertically. If developers have failed to anticipate this situation, the rendered GUI can be very messy.

4) Other Issues: Several other issues may also cause text-layout bugs. In particular, a layout problem of a text view could also be incurred by other adjacent GUI components. For example, a text view may be shaded by an adjacent button or an image.

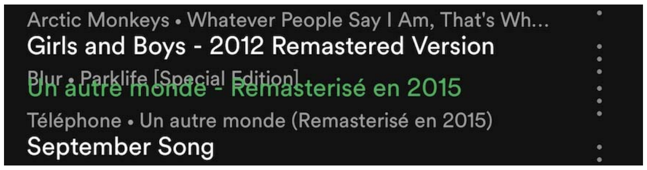
C. Categorization of Text-layout Bugs

We now discuss several popular types of text-layout bug according to their display issues. These types of bug all result in readability issues of textual information, leading to a messy GUI. We illustrate each type of bug as follows.

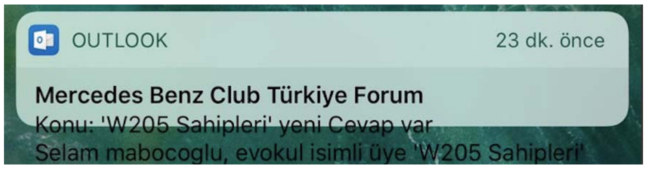
- *Overlapping texts*: Texts of two GUI components (*e.g.,* texts in two text views) may overlap with one another. Figure 2a shows a typical example of such overlapping.
- *Crossing-border texts*: Texts may exceed the corresponding containers' borders. For example, if the text of a button is too long, it will flow across the border of the button. A typical example is shown in Figure 2b.
- *Occluded texts*: Text may be occluded by other components, with the result that part of the text cannot be shown. Figure 2c shows a typical example.
- *Hybrid of the above*: Two or more types of bug may simultaneously affect some text in the GUI. Figure 2d shows an example of text compromised by more than one type of bug.

IV. Approach

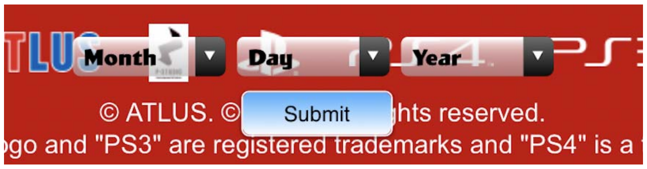
This section describes our deep learning-based approach of detecting text-layout bugs.


Arctic Monkeys • Whatever People Say I Am, That's Wh...
Girls and Boys - 2012 Remastered Version
Blur • Parklife [Special Edition]
Un autre monde - Remasterisé en 2015
Téléphone • Un autre monde (Remasterisé en 2015)
September Song

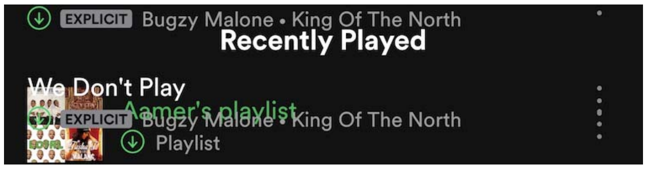
(a) Overlapping texts.


OUTLOOK
23 dk. önce
Mercedes Benz Club Türkiye Forum
Konu: 'W205 Sahipleri' yeni Cevap var
Selam mabocoglu, evokul isimli üye 'W205 Sahipleri'

(b) Crossing-border texts.



(c) Occluded texts.



(d) Hybrid.

Fig. 2. Real examples of different types of text-layout bug collected from the Internet.

The consequences of text-layout bugs are disagreeable visual issues. An obvious approach, therefore, would be to detect text-layout bugs based on the screenshots of an app during runtime. Intuitively, we can model this problem as an image classification problem. For each image, our classifier tries to discriminate it as "normal" or "abnormal" (*i.e.,* an image with layout bugs).

We mainly employ CNN for the image classification tasks. CNN is a type of Deep Neural Network(DNN), and has won a good reputation for its performance in image classification. In recent years, with the rising popularity of deep learning, a number of CNN models for image classification have been proposed. Some of them have already performed well on ImageNet [17], the well-known image classification competition. We therefore use CNN to do the classification for us.

But there are still many challenges to overcome. First, training a CNN generally requires thousands or even millions of images. It is easy to collect normal layout images but hard to collect a large volume of abnormal ones. This could lead to the problem of unbalanced data. Second, since a screenshot usually contains rich information and diversified data, distractions may compromise the performance of the CNN. It is also a complicated task to locate the bug areas on the screenshot precisely.

A. *Overview of Our Approach*

We address the challenges of text-layout bug detection as follows. To overcome the lack of abnormal data, we find a way to generate artificial abnormal data manually. We also use text detection to minimize the possibility of distraction on screenshots. This prompts the CNN to focus on text regions and reduces the risk of overfitting. It also helps us to locate the text-layout bug areas. Figure 3 shows the overview of our approach. It contains two phases: the training phase and the bug-detection phase.

Figure 3a shows the training phase. We first collect normal layout screenshots from applications on different smartphones. The text detection tool then detects text regions on them and segments them into pieces accordingly. These pieces are labeled as normal samples and then fed into the abnormal sample generator. The generator draws fake texts or components on them to produce abnormal samples. Finally, these normal and abnormal samples make up the training dataset which we use to train the binary CNN classifier.

Figure 3b shows the bug-detection phase. We use text detection to segment each input screenshot into pieces accordingly. The binary CNN classifier then gives each piece a predicted label: "normal" or "abnormal". Finally, we map samples labeled with "abnormal" back to their original screenshots and mark up the corresponding areas with red rectangles.

In the following paragraphs, we introduce the text detection technology and the binary CNN classifier, and then describe both the training phase and the bug-detection phase in detail.

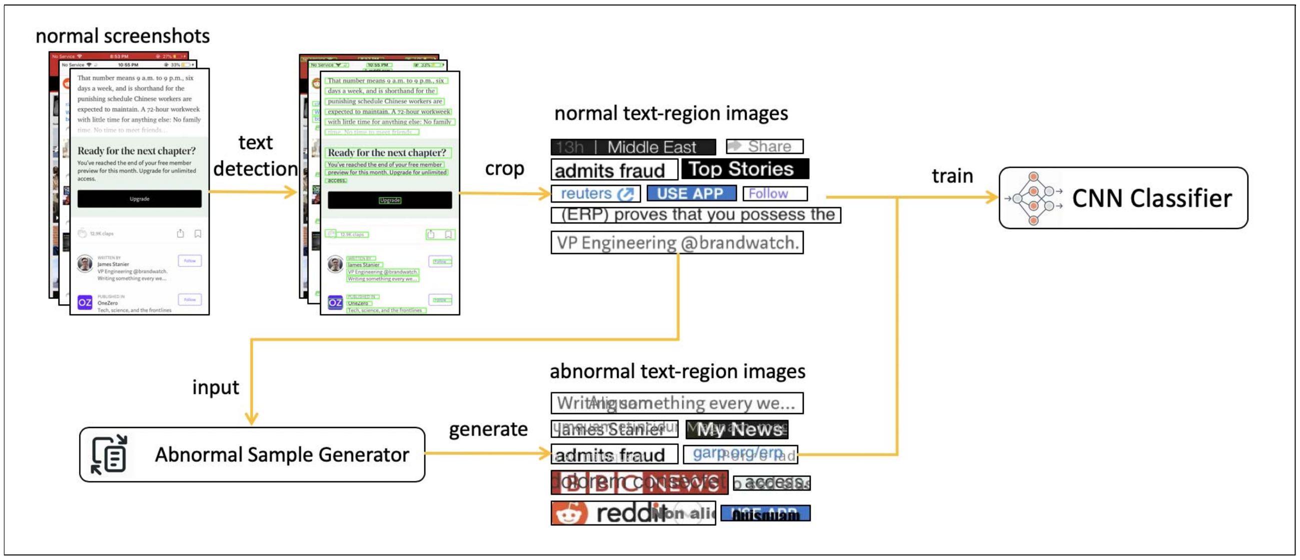
B. *Text Detection*

Text detection is used to detect and mark text regions on images. Several previous studies (*e.g.,* [18], [19]) have addressed the subject of text detection, and a range of sophisticated text detection tools are now available. The text detection tool scans the whole image with different-size windows. For each image clip segmented by the window, it gives a corresponding confidence score to indicate the likelihood that this area is filled with text.

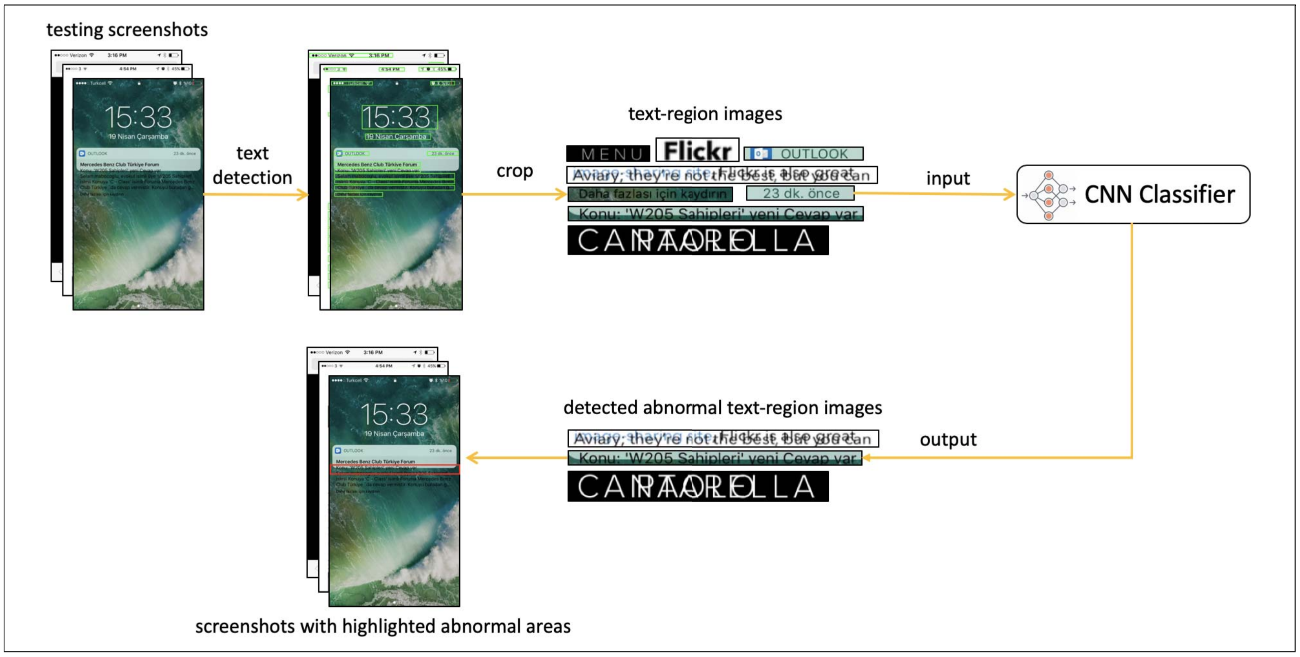
We use text detection to detect text regions on screenshots. In general, the texts displayed on mobile device screens are well printed, which enables text regions to be distinguished from screenshots with a high degree of precision.

C. *Binary CNN Classifier*

Building on the expertise developed in previous studies by others, we also customize a binary CNN classifier. This takes text-region images as input and gives a predicted label, "normal" or "abnormal", for each image. We train the CNN model using collected normal samples and generated abnormal samples.


normal screenshots
text detection
crop
normal text-region images
train
CNN Classifier
input
abnormal text-region images
generate
Abnormal Sample Generator

(a) The training phase of Textout.



(b) The bug-detection phase of Textout.

Fig. 3. The Textout framework, consisting of a training phase and a bug-detection phase

D. Training Phase

Figure 3a shows the overall training phase, including normal sample collection, abnormal sample generation, and CNN classifier training.

1) Normal Sample Collection: We collect screenshots from off-the-shelf applications on several different smartphones. We check each of them manually to ensure that no screenshot contains text-layout bugs. Then we use text detection to detect text regions on them and segment them into pieces accordingly. We consider these pieces as normal samples. They are further used to generate abnormal samples.

2) Abnormal sample generation: We generate abnormal samples from normal ones, referring to the categorization of text-layout bugs mentioned in section III. We produce buggy images of the four text-layout bugs by drawing fake texts or components on normal samples. We introduce considerable randomness in this process, to increase the diversity of the generated data and to reduce the risk of overfitting.

3) CNN Classifier Training: Before training, because color information is redundant for detecting text-layout bugs, we convert all samples to grayscale. We also adjust their sizes to fit the input size requirement of the CNN. They are then fed into the CNN model for training.

E. Bug-detection Phase

Figure 3b shows the sequence of the bug-detection phase. The input consists of the screenshots to be tested. After the execution, it highlights abnormal text-layout areas, if any, on the corresponding screenshots. We explain the process in detail below.

First, we use the text detection tool to detect text regions on the input screenshots and segment them into pieces accordingly. The tool also outputs the position and size of each text region. We use this information to map the text-region images back to their original screenshots.

The text-region images are then converted to grayscale and resized to meet the input shape requirement of the CNN. The CNN classifier we have trained takes these text-region images as input, and gives each of them a predicted value. We set a threshold to the predicted value to control the sensitivity of the CNN classifier. If the predicted value is higher than the threshold, then the corresponding sample is labeled "abnormal", or it is labeled "normal".

Finally, we map the samples labeled "abnormal" back to their original screenshots and highlight the corresponding bug areas on them.

V. Implementation

This section describes the implementation details of Textout. We first introduce the details of the two basic techniques we use in Textout, *i.e.,* the text detection tool and the customized binary CNN classifier. Then we introduce the details of the training data collection, the training phase, and the bug-detection phase.

A. Text Detection Tool

We use a popular open-source text detection tool[3] for this task, which is based on deep learning and provides a pre-trained model. It performs well on both normal text-layout screenshots and abnormal ones. Figure 4 shows several text detection results on screenshots with different text-layout bugs. The tool marks text lines with green rectangles. In Figure 4a, the text lines on the screen overlap. In Figure 4b, the texts in the bubble are too long and cross the border of the bubble. In Figure 4c, the copyright information on the bottom of the screen is occluded by the submit button. In Figure 4d, the titles of some table cells are too long and overlap the icons and texts below. As the results demonstrate, the text detection tool can help mark up all the concerned regions.

[3]https://github.com/eragonruan/text-detection-ctpn

B. Customized Binary CNN Classifier

ResNet [20], a kind of CNN, was proposed in 2016. It has won a good reputation for its high performance on ImageNet [17]. It uses shortcut links between convolutional layers to prevent vanishing/exploding gradients in deep neural networks.

We use a customized ResNet as the CNN classifier. Figure 5 shows its structure. This structure is based on that of ResNet50, a typical ResNet, but we modify its input layer and the last two layers to fit our needs.

Most text-region images are long in width and short in height. All input images are grayscale, *i.e.,* they only have one color channel. We therefore change the input size of the CNN to $200 \times 40 \times 1$, which means the input images should be 200 pixels in width, 40 pixels in height, and have a single color channel.

For the last two layers, we first change the 1000-dimensional fully connected layer in ResNet50 to a 1-dimensional fully connected layer (see "fc, 1" in Figure 5). The output of this layer is a single value. A sigmoid activation layer is then applied to normalize this value within the range 0 to 1. The sigmoid layer is used in the training phase, as the input samples are labeled with 0 or 1. In the bug-detection phase, we extract the output value of the "fc, 1" layer for prediction. We set a threshold to this value and can adjust the sensitivity of the CNN model by adjusting this threshold.

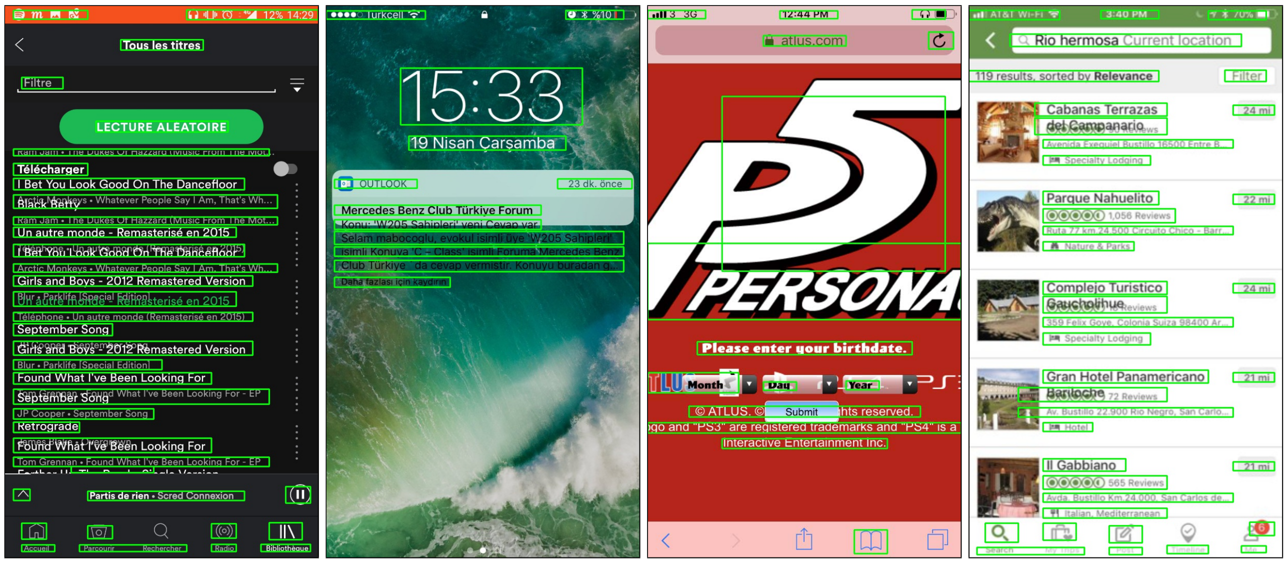
Keras is one of the most popular open-source deep learning libraries. It is capable of building complicated neural network structures, and can also extract the output of the neural network's intermediate layers. We use Keras to build this customized ResNet.

C. Training Data Collection

We collect 580 screenshots from 30 popular mobile applications on three different smartphones. The applications are selected from a variety of categories, such as social media, music, reading, multimedia, and shopping. The smartphones come from different platforms and have different resolutions. After collection, we check every screenshot manually to ensure that there are no layout bugs on them. We then use the text detection tool to detect text regions on them and segment them into text-region images accordingly. The segmentation results in 16,551 text-region images. We consider them as normal samples.

The normal samples are then used to generate abnormal samples. We draw fake texts or components on the normal samples to create different text-layout bugs. To prevent the CNN model from overfitting to the generated data, we need to increase the diversity of the data. We introduce a considerable degree of randomness in the generation process. We use a Python library called lorem[4] to generate random Latin sentences. We use random fonts, font sizes and font colors to create diversified texts. We use random icon images, photos and random colored blocks to simulate various GUI

[4]https://github.com/sfischer13/python-lorem



(a) Overlapping texts. Some text lines on the screen overlap each other.

(b) Crossing-border texts. The texts in the bubble are too long and cross the border of the bubble.

(c) Occluded texts. The copyright information on the bottom of the screen is occluded by the submit button.

(d) Hybrid. The titles of some table cells are too long and overlap the icons and texts below.

Fig. 4. Examples of text detection results on screenshots with different text-layout bugs. The detected text regions are marked with green rectangles. As the results demonstrate, the text detection tool can help mark up all the concerned regions.

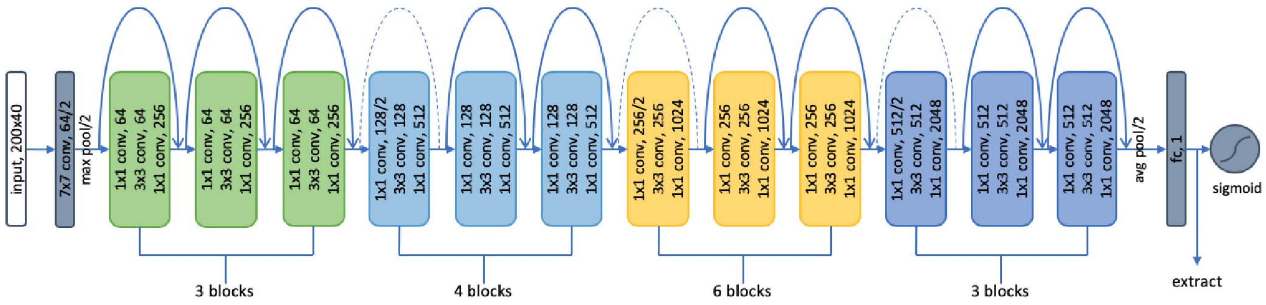


Fig. 5. The structure of the customized ResNet. This is based on the structure of ResNet50 and we modify its input layer and last two layers to fit our needs.

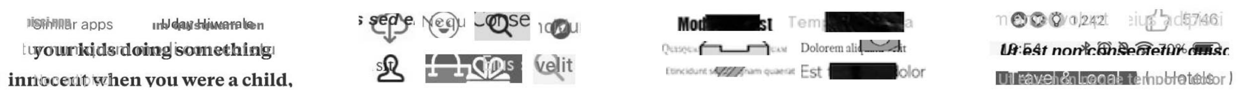
components. The positions of the texts and components are random within a reasonable range, to ensure that the generated samples have abnormal visual effects. Figure 6 shows some of the generated abnormal samples using this method. One normal sample produces one abnormal sample. The final training dataset contains a total of 33,102 samples, of which 16,551 are normal and the other 16,551 are abnormal.

D. Training Phase

After generating all the training data, we start the training phase. A Python script first reads the training images from a folder, converts them to grayscale and resizes them to fit the input size of the customized CNN. We label all normal samples as 0 and all abnormal samples as 1. In each training epoch, we randomly remove 10% of the samples from the training dataset for cross-validation purposes. We use binary cross-entropy as the loss function, and RMSprop as the optimizer. We train the model continuously until the loss function converges to a particular range.

E. Bug-detection Phase

We create a sequence so that the entire three-part bug-detection phase (text detection, abnormal sample detection, and marking) proceeds automatically. First, in the text detection part, screenshots are segmented into text-region images. The text-region images are then converted to grayscale and resized to fit the input shape requirement of the CNN. In the abnormal sample detection part, the CNN gives each of the text-region images a predicted label, either "normal" or "abnormal". Finally, in the marking part, we map the samples labeled with "abnormal" back to their original screenshots and highlight the corresponding areas on the screenshots.



(a) Overlapping texts. (b) Crossing-border texts. (c) Occluded texts. (d) Hybrid.

Fig. 6. Examples of generated abnormal samples of different types of text-layout bug. We generate these samples by drawing fake texts or components on the normal text-region images.

We implement the three parts mentioned above as three separate programs. They communicate through files in the file system, and the output of the previous program becomes the input of the next. We program a shell script to connect the three parts and automate the entire bug-detection process. Before execution, the user only needs to connect the input interface with the testing data.

VI. Experiments

In this section, we first describe the experiment settings. Then we describe the details of testing data collection and evaluate the performance of Textout on both text-region images and screenshots. We also evaluate the overhead of Textout at the end of the section.

A. Experiment Settings

Table I shows the main specifications of the smartphones we use. To increase the diversity of the collected screenshots, we choose three smartphones with different models, different operating systems, and different screen resolutions. We also choose 30 popular applications in different categories for the collection.

TABLE I
Details of smartphones

Model	System Version	Resolution
iPhone 8	iOS 12.2	1334 x 750
Nexus 6P	Android 6.0.1	2560 x 1440
Vivo X27	Android 9	2460 x 1080

Table II shows the hardware and software settings we use in the training phase and bug-detection phase. Note that the machine used in the bug-detection phase is an ordinary laptop without a GPU. We want to demonstrate that, although we need a powerful machine to train the model, an ordinary laptop can easily be used for bug detection.

TABLE II
Hardware and software settings in the experiments

	Model / Version	
	Training Phase	Bug-detection Phase
CPU	Intel® Core™ i9-7960X	Intel® Core™ i5-8259U
GPU	GeForce® GTX 1080 Ti	-
OS	Ubuntu 16.04.1	macOS 10.14.4
Python	3.7.3	
Keras	2.2.4	
TensorFlow	1.13.1	
CUDA	10.0	-

B. Performance

We evaluate the performance of Textout on screenshots of real-world applications. Since ours is the first study (to our knowledge) to detect text-layout bugs by analyzing only raw screenshot images, it is unlikely that a baseline for comparison has been developed in previous work.

1) Testing data collection: For normal samples, we take screenshots from several applications on the three experimental smartphones. We check the screenshots manually to ensure that they have no layout bugs. We collect 38 normal screenshots, and after text detection, we obtain 991 normal text-region images.

For abnormal samples, we find 4 screenshots with text-layout bugs from applications installed on the three experimental smartphones. We also collect another 17 screenshots searched from the Internet using keywords such as "text overlaps" and "GUI bugs". These 17 screenshots are from many different smartphones with different resolutions. The text detection tool then segments all the 21 screenshots into 490 text-region images, which are mixed with both normal and abnormal images. We manually check them one by one to separate them, and obtain 414 normal text-region images and 76 abnormal ones.

Finally, we combine these two parts of data. We obtain 59 screenshots, which contain 38 normal screenshots and 21 abnormal ones. And accordingly, we obtain 1,481 text-region images, which contain 1,405 normal text-region images and 76 abnormal ones.

2) Performance on text-region images: In the bug-detection phase, we extract the value before the sigmoid layer in the CNN model for classification. This is a single value, and we set a threshold to this value to change the sensitivity of the model. Table IIIa illustrates the different performance metrics of Textout on text-region images when different thresholds are set. When the threshold decreases, we obtain a higher true positive rate, *i.e.*, a higher recall value. As the data shows, Textout can detect most of the buggy text-region images with a low false positive rate. The orange line in Figure 7 shows the ROC curve on the text-region images. The corresponding AUC is 0.956, indicating that Textout has a high separability on text-region images.

3) Performance on screenshots: We map the bug detection results on the text-region images back to the screenshots to test the performance of Textout on screenshots. If the text-region images of a screenshot are all predicted as normal, then

TABLE III
THE PERFORMANCE OF TEXTOUT

(a) The performance of Textout on text-region images when setting different thresholds. In this experimental setting, there are 76 abnormal text-region images and 1,405 normal ones.

No.	Threshold	True Positive	True Negative	False Positive	False Negative	True Positive Rate	False Positive Rate
1	-112.984161	76	837	568	0	1.000000	0.404270
5	-59.562878	72	1130	275	4	0.947368	0.195730
9	-46.110771	68	1215	190	8	0.894737	0.135231
13	-38.659454	64	1259	146	12	0.842105	0.103915
17	-31.464548	60	1299	106	16	0.789474	0.075445
22	-22.044941	55	1337	68	21	0.723684	0.048399
26	-7.162227	51	1374	31	25	0.671053	0.022064
30	-0.142310	47	1383	22	29	0.618421	0.015658
34	15.635172	43	1394	11	33	0.565789	0.007829
39	34.562897	38	1398	7	38	0.500000	0.004982

(b) The performance of Textout on screenshots when setting different thresholds. In this experimental setting, there are 21 buggy screenshots and 38 normal ones.

No.	Threshold	True Positive	True Negative	False Positive	False Negative	True Positive Rate	False Positive Rate
1	-112.984131	21	0	38	0	1.000000	1.000000
2	-79.204926	20	4	34	1	0.952381	0.894737
3	-30.022625	19	18	20	2	0.904762	0.526316
4	-17.113678	18	27	11	3	0.857143	0.289474
5	-5.688166	17	31	7	4	0.809524	0.184211
6	-0.142313	16	32	6	5	0.761905	0.157895
7	5.921871	14	35	3	7	0.666667	0.078947
8	14.656797	13	35	3	8	0.619048	0.078947
9	17.760717	12	36	2	9	0.571429	0.052632
10	29.882370	10	36	2	11	0.476190	0.052632

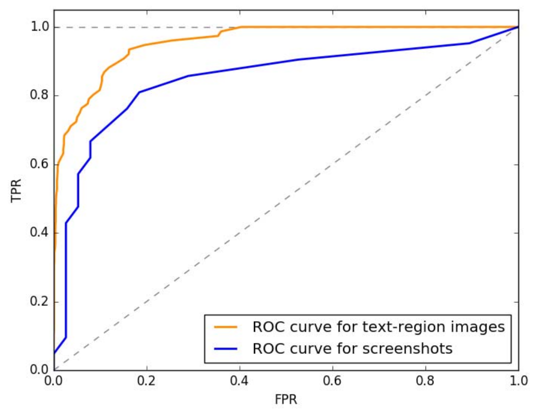


Fig. 7. The ROC curves for the testing data. The orange line is the ROC curve for text-region images, and its AUC is 0.956. The blue line is the ROC curve for screenshots, and its AUC is 0.848

we consider the screenshot as normal. If not, we consider it as abnormal. Table IIIb illustrates the different performance metrics of Textout on screenshots when different thresholds are set. As the data shows, Textout can detect most of the buggy screenshots with a low false positive rate. The blue line in Figure7 shows the ROC curve on screenshots. The corresponding AUC is 0.848, indicating that Textout also has a high separability on screenshots.

Because the CNN classifier is based on text-region images of various sizes, not the whole screenshot, our approach is in principle scalable to different screen resolutions of different smartphones. This claim is consistent with the experimental results above, where the screenshot images collected for testing are from many different smartphones with different resolutions.

C. *Overhead*

We measure the overhead of Textout on a regular laptop with no GPU to illustrate its practicability. We run the tool on 100 screenshots, which result in 2602 text-region images after text detection. The whole process takes 417 seconds. The average execution time is 4.17 seconds for each screenshot. The text detection part takes 3.37s on average, which accounts for about 81% of the total time. In average, the prediction part takes about 0.8 seconds per screenshot, *i.e.,* 0.03 seconds per text-region image.

VII. DISCUSSION

A. *Usability*

The screenshots are the only data Textout requires to detect text-layout bugs. The simplicity of our approach spares developers the need to write complex, hard-to-maintain GUI layout testing scripts. Also, Textout does not need prepared target design images, which are necessary for the methods based on image comparison.

Textout is easy to use. Users only need to connect the testing data with the input interface of Textout. The automatic execution can be triggered by one command line. The predicted abnormal text-layout regions are highlighted in the screenshots, so that it is very easy for users to check where the bugs occur.

The tool has a reasonable overhead. Its execution time is acceptable on an ordinary laptop without a GPU. The overhead can be much lower on a GPU-equipped computer.

In our approach, the screenshot images are collected manually. But since it is easy to take screenshots for smartphones in many automated testing tools (*e.g.*, Appium[5]), we can easily automate the collection process. Further, we can integrate Textout into the automated testing processes that developers currently use. For each screen in the testing process, they can take a screenshot and use Textout to detect whether there are text-layout bugs. The integration requires little programming effort and the overhead it brings to the testing process is reasonable.

B. Universality

Textout is cross-platform, cross-device and cross-application, because the CNN model we applied in Textout is based on text-region images of various sizes segmented from screenshots, not the whole screenshot. It works on both Android and iOS platforms and on devices with different resolutions.

Currently, the effectiveness of our approach has been examined only with the screenshot images of smartphones. In theory, this approach can also be easily applied to devices with wider screen sizes (*e.g.,* desktops and tablets) by collecting data from these devices and training a new CNN model. We intend to pursue the task of extending Textout to fit devices with wider screen sizes in our future research.

C. The Dataset

The size of our training dataset is relatively small for a CNN classification task, but the CNN model we have trained shows a good performance. We can further enhance the performance by extending the training dataset. The extension should be very easy, since the abnormal text-region images are generated from normal ones, and normal screenshots are easy to find in off-the-shelf apps.

The size of the test dataset in the experiment is relatively small. This is because collecting many real-world buggy screenshots from public resources is not easy. We made extensive efforts to search for them on the Internet and in app stores, and only 21 were found. Nevertheless, these 21 screenshots are representative, since they cover all the types of text-layout bug we define in Section III.

We intend to continuously extend the training dataset and test dataset in our future research.

D. Threats to Validity

In our experimental study, we have shown that Textout can detect all types of text-layout bug. However, its effectiveness will largely depend on the training data, *i.e.,* the samples of text-layout bugs. Therefore, it may not be effective for detecting other unknown text-layout bugs unless the corresponding training samples are provided.

Textout may also misclassify some normal screenshots as buggy images, especially when the application contains popup windows. Figure 8 shows two examples where the popup windows occlude part of the texts on the screens. These cases should not be considered as abnormal, but Textout may misidentify them as text-layout bugs. However, such images account for only a very small part of all images, and they can be easily filtered by developers in the testing process. We intend to pursue the detection (*e.g.,* via another classifier based on the whole screenshot) of such false positive images in our future work.

[5]https://appium.io/

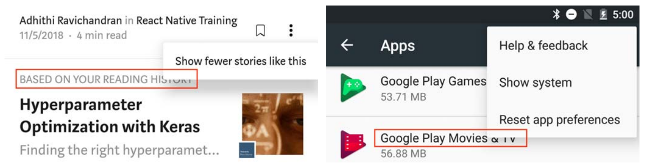

Adhithi Ravichandran in React Native Training
11/5/2018 · 4 min read
Show fewer stories like this
BASED ON YOUR READING HISTORY
Hyperparameter Optimization with Keras
Finding the right hyperparamet...
5:00
Apps
Help & feedback
Google Play Games
53.71 MB
Show system
Reset app preferences
Google Play Movies & TV
56.88 MB

Fig. 8. The text in the red rectangle is occluded by the popup window.

VIII. Conclusion

In this work, we focus on a popular type of layout bug related to texts. We propose a novel approach, Textout, which takes raw screenshot images as input and detects text-layout bugs on them. Textout uses text detection technology to detect text-regions on screenshots and uses CNN to detect buggy text regions. We collect 33,102 text-region images as our training dataset and verify the effectiveness of our approach with 1,481 text-region images collected from real-world apps. Textout achieves an AUC (area under the curve) of 0.956 and can detect most of the buggy clips with a low false positive rate. The overhead of Textout is acceptable on an ordinary laptop without a GPU. We release our dataset as open-source for further research.

In our future work, we will continuously extend our open-source dataset with more training and testing data. We will extend Textout to fit devices with wider screen sizes, such as desktops and tablets. We will also prepare to integrate Textout with other automated GUI testing tools.

IX. Acknowledgement

The authors are grateful to their shepherd Dr. S. Banerjee and anonymous reviewers for the detailed and valuable comments. This work is supported by the National Natural Science Foundation of China (Project Nos. 61672164 and 61571136). Lyu and Xu's work was supported by the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14210717 of the General Research Fund), and Microsoft Research Asia (2018 Microsoft Research Asia Collaborative Research Award). Yangfan Zhou is the corresponding author.

References

[1] A. Zhang, Y. He, and Y. Jiang, "Crashfuzzer: Detecting input processing related crash bugs in android applications," in *Proc. of the 2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2016, pp. 1–8.

[2] Y. Kang, Y. Zhou, H. Xu, and M. R. Lyu, "Diagdroid: Android performance diagnosis via anatomizing asynchronous executions," in *Proc. of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 410–421.

[3] Y. Kang, Y. Zhou, M. Gao, Y. Sun, and M. R. Lyu, "Experience report: Detecting poor-responsive ui in android applications," in *Proc. of the 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2016, pp. 490–501.

[4] S. Mahajan, A. Alameer, P. McMinn, and W. G. Halfond, "Automated repair of layout cross browser issues using search-based techniques," in *Proc. of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 249–260.

[5] StatCounter, http://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/worldwide.

[6] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What do mobile app users complain about?" *IEEE Software*, vol. 32, no. 3, pp. 70–77, 2014.

[7] L. Wei, Y. Liu, and S.-C. Cheung, "Taming android fragmentation: Characterizing and detecting compatibility issues for android apps," in *Proc. of the 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 226–237.

[8] K. F. Hasselknippe and J. Li, "A novel tool for automatic gui layout testing," in *Proc. of the 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2017, pp. 695–700.

[9] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[10] S. Mahajan and W. G. Halfond, "Finding html presentation failures using image comparison techniques," in *Proc. of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2014, pp. 91–96.

[11] Google, https://code.google.com/archive/p/fighting-layout-bugs.

[12] H. Lu, L. Wang, M. Ye, K. Yan, and Q. Jin, "DNN-based image classification for software gui testing," in *Proc. of the 2018 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation*. IEEE, 2018, pp. 1818–1823.

[13] H. Wang, S. Rath, and Y. Sharan, https://www.ebayinc.com/stories/blogs/tech/gui-testing-powered-by-deep-learning/.

[14] Applitools, https://applitools.com.

[15] Android Inspector, https://developer.android.com/studio/debug/layout-inspector.

[16] View Hierarchy Debugger, https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/debugging_with_xcode/chapters/special_debugging_workflows.html.

[17] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*, 2009.

[18] T. He, W. Huang, Y. Qiao, and J. Yao, "Text-attentional convolutional neural network for scene text detection," *IEEE Transactions on Image Processing*, vol. 25, no. 6, pp. 2529–2541, 2016.

[19] Z. Zhang, C. Zhang, W. Shen, C. Yao, W. Liu, and X. Bai, "Multi-oriented text detection with fully convolutional networks," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4159–4167.

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.