Discovering UI Display Issues with Visual Understanding

Zhe Liu
Institute of Software Chinese Academy of Sciences, University of Chinese Academy of Sciences

ABSTRACT

GUI complexity posts a great challenge to the GUI implementation. According to our pilot study of crowdtesting bug reports, display issues such as text overlap, blurred screen, missing image always occur during GUI rendering on difference devices due to the software or hardware compatibility. They negatively influence the app usability, resulting in poor user experience. To detect these issues, we propose a novel approach, `OwlEye`, based on deep learning for modelling visual information of the GUI screenshot.Therefore, `OwlEye` can detect GUIs with display issues and also locate the detailed region of the issue in the given GUI for guiding developers to fix the bug. We manually construct a large-scale labelled dataset with 4,470 GUI screenshots with UI display issues. We develop a heuristics-based data augmentation method and a GAN-based data augmentation method for boosting the performance of our `OwlEye`. At present, the evaluation demonstrates that our `OwlEye` can achieve 85% precision and 84% recall in detecting UI display issues, and 90% accuracy in localizing these issues.

ACM Reference Format:
Zhe Liu. 2020. Discovering UI Display Issues with Visual Understanding. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3324884.3418917

1 PROBLEM AND MOTIVATION

Graphic User Interface (GUI, also short for UI) is ubiquitous in almost all modern desktop software and mobile applications. A good GUI design makes an application easy, practical and efficient to use, which significantly affects the success of the application and the loyalty of its users [9].

However, the variety of visual effects in GUI design brings great challenges to developers. Consequently, many display issues as seen in Figure 1 always occur during the UI display process especially on different mobile devices.

Most of those UI display issues are caused by different system settings in different devices. Although the software can still run along with these bugs, they negatively influence the fluent usage with the app, resulting in the significantly bad user experience and corresponding loss of users. Therefore, this study is targeting at detecting those UI display issues.

In order to ensure the correctness of UI display, companies have to recruit many testers for app GUI testing or leverage the crowdtesting, which requires significant human effort. Some practical automated testing tools like Monkey [7, 23], Dynodroid [14] are also widely used in industry. However, these automated tools can only spot critical crash bugs, rather than UI display issues which cannot be captured by the system. In this work, we aim at detecting the UI display issues with the screenshots generated during automatic testing by visual understanding.

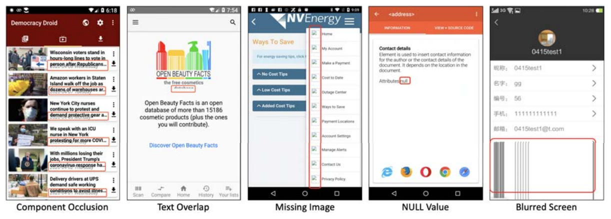

Figure 1: Examples of UI display issues

To understand the common UI rendering issues, we first carry out a pilot study on 10,330 non-duplicate screenshots from 562 mobile application crowdtesting tasks to observe display issues in these screenshots. Results show that a non-negligible portion (43.2%) of screenshots are of display issues which can seriously impact the user experience, and degrade the reputation of the applications. As seen in Figure 1, the common categories of UI display issues include *component occlusion, text overlap, missing image, null value* and *blurred screen*. Considering its popularity and lack of support in current practice of automatic UI testing, it would be valuable for classifying the screenshots with UI display issues from the plenty of screenshots generated during UI testing.

Inspired by the fact that display bugs can be easily spotted by human eyes, we propose an approach, `OwlEye`[1] to model the visual information by deep learning to automatically detect and localize UI display issues. Note that one strength of our approach over conventional program analysis is that it can be applied to any platform including Android, IOS, and it takes the screenshot as the input which is easy to be obtained in real-world practice.

2 RELATED WORK

GUI provides a visual bridge between applications and users. Therefore, many researchers are working on assisting developers or designers in the GUI search [4, 19] based on image features, GUI code generation [5, 16, 17] based on computer vision techniques. Different from these works, our works are focusing on GUI testing.

To ensure that GUI is working well, there are many static linting tools to flag programming errors, bugs, stylistic errors, and suspicious constructs. For example, Android Lint [1] reports over 260 different types of Android bugs, including correctness, performance,

[1]Our approach is named as `OwlEye` as it is like owl's eyes to effectively spot UI display issues. And our model (nocturnal like owl) can complement with conventional automated GUI testing (diurnal like eagle) for ensuring the robustness of the UI.

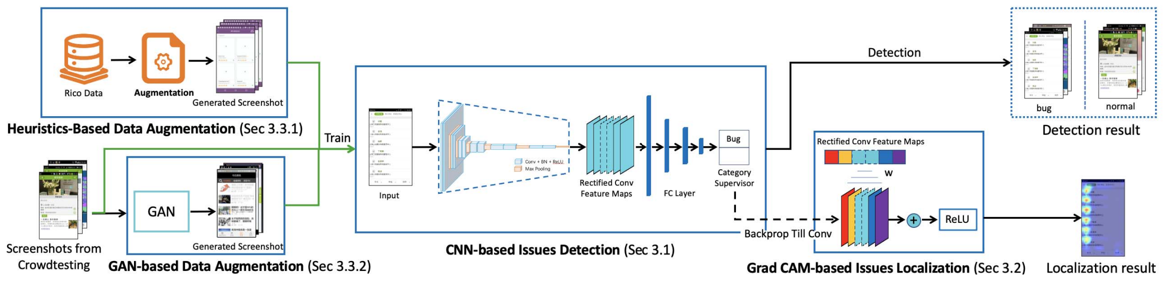
Rico Data
Augmentation
Generated Screenshot
Heuristics-Based Data Augmentation (Sec 3.3.1)
Train
Screenshots from Crowdtesting
GAN
Generated Screenshot
GAN-based Data Augmentation (Sec 3.3.2)
Input
Conv + BN + ReLU
Max Pooling
Rectified Conv Feature Maps
FC Layer
Bug
Category Supervisor
CNN-based Issues Detection (Sec 3.1)
Detection
bug
normal
Detection result
Rectified Conv Feature Maps
W
ReLU
Backprop Till Conv
Grad CAM-based Issues Localization (Sec 3.2)
Localization result

Figure 2: Overview of `OwlEye`

security, usability and accessibility. StyleLint [2] helps developers avoid errors and enforce conventions in styles. Different from static linting, automatic GUI testing [3, 15, 22] dynamically explores GUIs of an app. Several surveys [11, 24] compare different tools for GUI testing for Android apps. Some testing works focus on more specific UI issues such as UI rendering delays [8] and image loading [13].

3 APPROACH AND UNIQUENESS

3.1 CNN-based Issues Detection

As the UI display issues can only be spotted via the visual information, we adopt the convolutional neural network (CNN) [10, 12]. Our CNN model is composed of 12 convolutional layers with batch normalization, 6 pooling layers and 4 full connection layers for classifying UI screenshot with display issues. The size of convolutional kernel in convolutional layer is 3 * 3. We set up the number of convolutional kernels as 16 for convolutional layer 1-4, 32 for convolutional layer 5-6, 64 for convolutional layer 7-8, and 128 for convolutional layer 9-12. For the pooling layers, we use the most common-used max-pooling setting [21].

3.2 Grad CAM-based Issues Localization

After Issues detection, we adopt the feature visualization method to locate the detailed position of the issues for guiding developers to fix the bug. We apply Grad-CAM model for the localization of UI display issues. Grad-CAM is a technique for visualizing the regions of input that are “important” for predictions on CNN-based models [20] . First, a screenshot with UI display issue is input into the trained CNN model, and the category supervisor to which the image belongs is set to 1, while the rest is 0. Then the information is propagated back to the convolutional feature map of interest to obtain the Grad-CAM positioning. Through the feedback of global average pooling of the gradient, the weight of the importance of neurons is obtained. By performing the weighted combination of the forward activation graph, we can obtain the class-discriminative localization map. Finally, the point multiplication with the back propagation can obtain the result of UI display issues localization.

3.3 Data Augmentation Method

Training our model for visual understanding requires a large amount of input data. However, there is so far no such type of open dataset. Therefore, we develop a heuristic-based data augmentation method for generating UI screenshots with display issues from bug-free UI images. In order to ensure the diversity of generated images, we also develop a GAN-based data augmentation method.

3.3.1 Heuristic-based Data Augmentation. The data augmentation is based on the Rico [6] dataset which contains more than 66K screenshots, as well as their accompanied JSON file. With the input screenshot and its associated JSON file, the method first locates all the TextView and ImageView, then randomly chooses a TextView or ImageView depending on the augmented category. Based on the coordinates and size of the TextView/ImageView, the method then makes its copy and adjusts its location or size following specific rules to generate the screenshot with corresponding issues.

3.3.2 GAN-based Data Augmentation. The Generative Adversarial Networks(GAN) [18] has achieved good results in image generation. GAN uses adversarial process to evaluate generative models. The generative model is used to generate the screenshots with issues, and the discriminative model is used to judge whether there are issues in the generated results. The input to the model is a buggy screenshot in the real world. Both generative model and discriminative model adopt CNN with 4 layers of convolution layer. After 400 epochs, we can get the screenshot with issues generated by GAN.

4 RESULTS AND CONCLUSION

The experimental dataset comes from crowdtesting. For the 8,940 screenshots screenshots from 562 crowdtesting apps, we utilize the 1,600 screenshots (800 with UI display issues and 800 without) from 162 apps as testing set to evaluate the performance of `OwlEye`, and employ another 1,000 screenshots (half of them with issues) from another 50 apps as validation set to estimate how well the model has been trained and further tune the parameters. The 6,340 screenshots from the remaining 350 apps is utilized as training set.

We first present the issues detection performance of our proposed `OwlEye`, the precision is 0.85, and the recall is 0.84, indicating 84% (679/800) buggy screenshots can be found with `OwlEye`. `OwlEye` is much better than the baselines, i.e., 50% higher in precision with the best baseline (ORB-NB). We conduct a user study to evaluate the localization performance. We recruit 6 software developers online and used the 5-likert scale to answer whether they agreed with each localization result. They agree with the UI display issues localization results in an average of 90% screenshots.

This paper focuses on automatic detecting the UI display issues from the screenshots generated during automatic testing. `OwlEye` achieves more than 17% and 50% boost in recall and precision compared with the best baseline. As the first work of its kind, we also contribute to a systematical investigation of UI display issues in real-world mobile apps, as well as a large-scale dataset of app UIs with display issues for follow-up studies.

REFERENCES

[1] 2020. http://tools.android.com/tips/lint.

[2] 2020. https://github.com/stylelint/stylelint.

[3] Young-Min Baek and Doo-Hwan Bae. 2016. Automated model-based android gui testing using multi-level gui comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 238–249.

[4] Farnaz Behrang, Steven P Reiss, and Alessandro Orso. 2018. GUIfetch: supporting app design and development through GUI search. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. ACM, 236–246.

[5] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 665–676.

[6] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *Proceedings of the 30th Annual Symposium on User Interface Software and Technology (UIST '17)*.

[7] Android Developers. 2012. Ui/application exerciser monkey.

[8] Yi Gao, Yang Luo, Daqing Chen, Haocheng Huang, Wei Dong, Mingyuan Xia, Xue Liu, and Jiajun Bu. 2017. Every pixel counts: Fine-grained UI rendering analysis for mobile applications. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 1–9.

[9] Bernard J. Jansen. 1998. The Graphical User Interface. *SIGCHI Bull.* 30, 2 (April 1998), 22–26. https://doi.org/10.1145/279044.279051

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[11] Tomi Lämsä. 2017. Comparison of GUI testing tools for Android applications. (2017).

[12] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[13] Wenjie Li, Yanyan Jiang, Chang Xu, Yepang Liu, Xiaoxing Ma, and Jian Lu. 2019. Characterizing and Detecting Inefficient Image Displaying Issues in Android Apps. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, Xinyu Wang, David Lo, and Emad Shihab (Eds.). IEEE, 355–365. https://doi.org/10.1109/SANER.2019.8668030

[14] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 224–234. https://doi.org/10.1145/2491411.2491450

[15] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing combinatorics in GUI testing of android applications. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 559–570.

[16] Kevin Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. 2018. Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps. *arXiv preprint arXiv:1802.02312* (2018).

[17] Tuan Anh Nguyen and Christoph Csallner. 2015. Reverse engineering mobile application user interfaces with remaui (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 248–259.

[18] Alec Radford, Luke Metz, and Soumith Chintala. 2015. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434* (2015).

[19] Steven P Reiss, Yun Miao, and Qi Xin. 2018. Seeking the user interface. *Automated Software Engineering* 25, 1 (2018), 157–193.

[20] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. 2017. Grad-CAM: Visual Explanations From Deep Networks via Gradient-Based Localization. In *The IEEE International Conference on Computer Vision (ICCV)*.

[21] Patrice Y. Simard, Dave Steinkraus, and John C. Platt. 2003. Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis. In *Proceedings of the Seventh International Conference on Document Analysis and Recognition - Volume 2 (ICDAR '03)*. IEEE Computer Society, USA, 958.

[22] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 245–256.

[23] Thomas Wetzlmaier and Rudolf Ramler. 2017. Hybrid Monkey Testing: Enhancing Automated GUI Tests with Random Test Generation. In *Proceedings of the 8th ACM SIGSOFT International Workshop on Automated Software Testing* (Paderborn, Germany) *(A-TEST 2017)*. Association for Computing Machinery, New York, NY, USA, 5–10. https://doi.org/10.1145/3121245.3121247

[24] Samer Zein, Norsaremah Salleh, and John Grundy. 2016. A systematic mapping study of mobile application testing techniques. *Journal of Systems and Software* 117 (2016), 334–356.