Neat: Mobile App Layout Similarity Comparison Based on Graph Convolutional Networks

Zhu Tao
taozhu.1124@bytedance.com
Bytedance
Beijing, China

Yongqiang Gao
gaoyongqiang@bytedance.com
Bytedance
Shenzhen, China

Jiayi Qi
qijiayi@bytedance.com
Bytedance
Beijing, China

Chao Peng
pengchao.x@bytedance.com
Bytedance
Beijing, China

Qinyun Wu
wuqinyun@bytedance.com
Bytedance
Beijing, China

Xiang Chen
chenxiang.10101@bytedance.com
Bytedance
Beijing, China

Ping Yang
yangping.cser@bytedance.com
Bytedance
Beijing, China

ABSTRACT

A wide variety of device models, screen resolutions and operating systems have emerged with recent advances in mobile devices. As a result, the graphical user interface (GUI) layout in mobile apps has become increasingly complex due to this market fragmentation, with rapid iterations being the norm. Testing page layout issues under these circumstances hence becomes a resource-intensive task, requiring significant manpower and effort due to the vast number of device models and screen resolution adaptations. One of the most challenging issues to cover manually is multi-model and cross-version layout verification for the same GUI page. To address this issue, we propose Neat, a non-intrusive end-to-end mobile app layout similarity measurement tool that utilizes computer vision techniques for GUI element detection, layout feature extraction, and similarity metrics. Our empirical evaluation and industrial application have demonstrated that our approach is effective in improving the efficiency of layout assertion testing and ensuring application quality.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; • **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

Mobile App, Graphical User Interface, GCN, CNN, OCR, YOLOX

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FSE Companion '24, July 15–19, 2024, Porto de Galinhas, Brazil
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0658-5/24/07
https://doi.org/10.1145/3663529.3663832

ACM Reference Format:
Zhu Tao, Yongqiang Gao, Jiayi Qi, Chao Peng, Qinyun Wu, Xiang Chen, and Ping Yang. 2024. Neat: Mobile App Layout Similarity Comparison Based on Graph Convolutional Networks. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion '24), July 15–19, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3663529.3663832

1 INTRODUCTION

In today's mobile app landscape, the success of an app largely depends on its elegant graphical user interface (GUI) that provides an effective user experience (UX). However, as GUI layouts become increasingly complex, there is a higher likelihood of encountering unexpected layout anomalies on different device models, system versions and resolutions, such as misalignment, missing elements, squeezing, etc. In practice, the quality assurance team at ByteDance writes numerous GUI automation scripts for each app page to perform regression testing. Nevertheless, for the vast number of screenshot pairs captured during the execution of these test scripts, the cost of manually analyzing them is significant. As the number of models and resolutions increases, so does the amount of manpower required for this task. Therefore, automating the process of determining whether two pages are identical or not becomes necessary. Two common methods for this purpose are the analysis of page structure files (e.g., in XML format) and the analysis of page screenshot similarity.

Previous studies have proposed numerous approaches to calculating page screenshot similarity. In traditional image processing methods, the texture, color, and shape features of images are usually utilized for similarity calculation. One of the classic approaches is the structural similarity index measure (SSIM) [35], which compares the structural information of images to calculate similarity. However, the disadvantage of such methods is that they require the input two images to have the exact same resolution. To accommodate this limitation, we need to resize the images with different resolutions, but this process only works when the aspect ratio of

the two pages is the same. In our application scenario, the aspect ratio varies across different devices.

In addition to comparing screenshots directly, the structural information of Android app pages in XML format can be obtained using the Android Debug Bridge. Each entry of the XML page represents a GUI component, such as containers and widgets. When this structural information is available, the similarity of pages can be calculated using the pairing relationship of GUI components between the two pages, such as the longest common subsequence [37] method. However, in practice, such information is not always accessed and stored. In addition, the format of structured GUI information varies across different mobile operating systems.

To address the aforementioned issues, we propose NEAT (Non-intrusive End-to-end mobile App layout similarity measurement Tool)[1] based on graph convolutional network (GCN) [25]. We utilize computer vision techniques to extract GUI element locations and attributes, which are then encoded in a graph. The GCN is used to vectorize the whole page and calculate the similarity. In practical application, our tool offers the following two capabilities for two pages from different devices or different versions of the app with the same expected layout:

(1) Similarity calculation of the page layout based on the two given pages;
(2) GUI elements matching, including the number of matched GUI elements. This capability can be leveraged for element missing and misplaced element detection.

To more accurately evaluate the effectiveness of our tool, we conducted experiments on the page similarity comparison process and report industrial application results. The empirical evaluation demonstrates that the layout similarity comparison process is highly accurate and the tool is effective in practical scenarios, significantly enhancing developer efficiency and reducing the amount of human labor required for quality assurance.

The contributions of this paper are as follows:

- We introduce an automated page similarity comparison tool based on computer vision and graph convolutional networks to analyze the layout of mobile apps. The tool takes as input two pages of arbitrary resolution and outputs the score of layout similarity as well as the results of element matching.
- We propose a non-intrusive page layout information extraction process, and the experimental results demonstrate that the extracted layout information is effective in characterizing the page layout.
- We verify the effectiveness of our tool in industrial scenarios, demonstrating its ability in improving the efficiency and reducing the labor costs for mobile app testing.

2 MOTIVATION

In this section, we discuss the current state of mobile app page layout testing and major existing issues and challenges they face when applied in the industry, which motivated us to design and implement our approach.

[1]Non-intrusive means that the tool does not need to instrument the source code of the app under test.

(a) Page with Different Resolution

(b) GUI Components Missing

(c) GUI Components Misplaced

Figure 1: Examples of Page layout Issues

2.1 Non-intrusive Layout Similarity Comparison for Mobile Apps

During the layout testing process, a large number of pages need to be compared, each of which requires executing the same test script

on various resolutions and height-width ratios, as shown in Figure 1a. After these test scripts are executed, screenshots generated respectively are compared with baseline screenshots (typically verified by human as reference) to assess the page layout. This process can be labor-intensive and time-consuming when done completely manually, highlighting the need for an automated layout comparison tool.

Furthermore, obtaining page layout information is a prerequisite for layout comparison, while different operating systems and devices provide different methods of obtaining this information and the accessed GUI structure is in different formats. Maintaining scripts or tools for obtaining page layout information for various devices and systems can be challenging. Thus, there is a need for a device-and-OS-independent page layout comparison tool, or in other words, a non-intrusive layout similarity comparison tool. Such a tool can help reduce the burden of layout testing by automating the process of comparing page layouts while also providing reliable and accurate results.

2.2 GUI Components Matching

When test engineers identify differences in page layouts, they must then locate the problematic GUI components from the page pairs. However, GUI components are often not fixed in size or position, making this process challenging and time-consuming. It is also difficult to work with a group of pages that have anomalies when the type and location of the anomalies are unknown. Common types of layout issues include missing GUI components (Figure 1b) and misplaced components (Figure 1c). By leveraging the results of GUI component matching, we can analyze component missing and misplacement automatically. When the page layout is dissimilar, we can match the GUI components to identify possible issues, such as missing or misplaced components. By doing so, our non-intrusive layout similarity comparison tool can provide a more comprehensive and efficient approach to mobile app layout testing, reducing the amount of time and effort required while also improving the overall quality of the end product.

3 RELATED WORK

In this section, we discuss existing research on automated GUI testing and object detection.

Automated GUI Testing. Automated GUI testing is a critical component of mobile app development, as the design of the app's user interface directly affects the user experience. Layout problems that impact GUI components can cause significant issues, and as such, developers are increasingly turning to automated testing tools that leverage artificial intelligence techniques. Li et al. [17] found that the introduction of new types of interactions in GUIs presents new kinds of errors that are not targeted by current testing techniques. To address this, they proposed a new GUI fault model designed to identify and classify GUI faults, which can be reused by developers for benchmarking their GUI testing tools. Moran et al. [26] identified that transforming a mock-up of a graphical user interface into code is challenging and time-consuming. They presented an approach that automates this process by enabling accurate prototyping of GUIs via three tasks: detection, classification, and assembly. Zaeem et al. [41] observed that testing mobile apps is particularly expensive and tedious, often requiring substantial manual effort. To address this, they presented an approach to automatically generate test cases with test oracles for mobile apps.

In addition to mobile app pages, layout problems also occur on web pages, and researchers have proposed various approaches to addressing these issues. Yandrapally et al. [39, 40] propose techniques to detect duplicated web pages. Mahajan et al. [24] focus on web page layout failures and build a model that links observable changes in the website's appearance to faulty elements and styling properties. This model then predicts the elements and styling properties most likely to cause the observed failure for the page under test, and reports these to the developer. By doing so, their approach provides a more efficient and effective way of debugging web page user interfaces. Mahajan et al. [23] also propose a novel automated approach for debugging web page user interfaces. They use computer vision techniques to detect failures and can then identify HTML elements that are likely to be responsible for the failure. By leveraging computer vision techniques, they can more accurately and efficiently identify issues with web page layouts, helping to reduce the amount of time and effort required during the testing and development process.

In addition to GUI component distribution, testing tools also focus on other issues. For example, Google's Monkey [1] is a pure random testing tool that emits pseudo-random streams of GUI events and some system events to test the functionality of the app. Other similar testing tools [3, 11, 18, 21, 22, 27] are widely used in practical business. OwlEyes-Online [32] is an online GUI display problem detection tool that can automatically run the program, take screenshots, and construct the corresponding XML tree structure. It can then automatically generate a test report by analyzing the XML tree to locate any display problems. By doing so, this tool provides a comprehensive approach to testing page layouts, including GUI components distribution and other display problems, helping to improve the quality and efficiency of the testing and development process. WebEvo [31] adapts computer vision techniques to identify semantic structure changes and map changed elements to their original counterparts. By contrast, Neat compares layout similarity across two GUI pages, rather than comparing element contents.

In summary, these studies demonstrate the potential for automated approaches and computer vision techniques to improve the efficiency and accuracy of page layout testing, while also providing valuable insights into the root causes of layout failures.

Object Detection Model. We have also investigated algorithms used to locate and classify various GUI components on a page. Locating GUI components on app pages is similar to the task of object detection in computer vision, which involves finding the object on an image and providing its location and category. Traditional object detection methods, such as HOG [34] and DPM [30], have low accuracy and complicated implementation processes. Therefore, we investigate multiple object detection algorithms in the realm of deep learning [44]. DL based techniques includes the one-stage anchor-based SSD[19] YOLOv5 [15], anchor free YOLOX [9], and two-stage anchor-based algorithms such as Faster-RCNN[28]. By using a label tool such as LabelImage to label the GUI components on the page, these models can be trained to automatically extract the location and categories of GUI components.

App Page Descriptor Based on GCN. While convolutional neural networks (CNNs) are commonly used for image classification models in computer vision [2, 29, 33, 40], there are many non-image categories of data in the real world, such as social relationships between people and knowledge graphs. In recent years, research on GCN has matured, allowing for image or video data to be converted to graph data and then combined with CNN to extract features of the graph. GCN is a type of graph network that first generates graph data based on object data and then uses CNN to train a model with the graph data to extract features of the object. Research on GCN is comprehensively summarized in [42], with [45] providing a detailed explanation of current research and applications of graph networks. GCN can be widely used in visual data, as shown in applications such as [5, 14, 43]. Moreover, [25] proposes a network that learns a search embedding for layout similarity by encoding the properties of interface components and their spatial relationships via a graph, which serves as a reference in our approach.

4 OUR APPROACH

In this section, we describe our approach for similarity comparison of mobile app pages in detail, focusing on the three main phases of the process: GUI component detection (Section 4.1), layout feature extraction (Section 4.2), and similarity measurement (Section 4.3). Figure 2 provides an overview of the process of our proposed tool. Firstly, a trained deep learning object detection model is used to detect GUI components from mobile app pages. Secondly, graph convolutional neural network is used to extract descriptors of relations among close GUI components, and then a suitable similarity measurement is used to compute the distance between page layouts. Additionally, if further detection results are needed, a GUI components matching phase can match the same GUI components on both pages, as introduced in Section 4.4.

4.1 GUI Component Detection

To obtain the spatial information and types of GUI components on a page without code instrumentation or debugging bridge, each component is extracted using an object detection model, which is capable of identifying GUI components' location and type. This process is also referred to as detection in computer vision.

To train a more general GUI component detection model for a given mobile app page, we divide GUI components into seven categories: text, icon, image, pop-up, keyboard, and button, as shown in Figure 3. Optical Character Recognition (OCR) is used for text recognition. The YOLOX [9] framework is used to train the GUI component detection model for other types of GUI components. In other words, the combination of YOLOX and OCR modules can take the page screenshot as input and output the GUI component's location and type on the screenshot. This information is used for further layout graph construction.

4.2 Layout Graph Descriptors Extraction

4.2.1 Graph Data Constructing. The graph data can be described by the equation: $G = (V, E)$ where V stands for the corpus of all nodes and E for the corpus of all edges. Each node, i has feature x_i. The feature of the graph can be represented by a matrix X_{N*D}, where N denotes the number of nodes and D denotes the number of features of each node, which can also be described as the dimension of the feature vector. Graph is usually used to represent relations and connections. In our task, page layout can be abstracted as relations and connections between GUI components. Therefore we use graphs to represent GUI component distribution.

In our case, we quantify the distribution of GUI components on a page as a graph data. We describe the page layout by GUI components' properties and geometry features following the method introduced by [25]. For our graph data, the node vector(V) represents all GUI components on the page, the features of all nodes can be extracted through the YOLOX model, the edge vectors(E) represent the positional associations between components. In academia, the adjacency matrix is a common mathematical expression for the graph data. Although there is theoretically an edge between any two components, the adjacency matrix should reflect the actual connections with varying feature vectors, therefore, for the adjacency matrix A, the value is 1 only between two nodes connected by an edge, the rest are 0.

4.2.2 GCN-CNN Training. After constructing the adjacency matrix of the page and extracting GUI components' features through YOLOX model mentioned in Section 4.1, GCN is utilized to extract descriptor of all GUI elements of a page. The GCN network layer is implemented as Equation 1. The architecture of our GCN-CNN network is illustrated in Figure 4. We extract a feature descriptor of 25 dimensions based on experiment of [25] and apply a triplet loss [13] to make similar vectors closer and dissimilar ones farther.

$$\begin{aligned} H^{l+1} &= f(H^l, A) \\ f(H^l, A) &= \sigma(D^{-1/2} * A * D^{-1/2} * H^l * W^l) \end{aligned} \tag{1}$$

where H^l denotes the GUI feature matrix of the whole page for layer l, A is the adjacency matrix of the graph. σ denotes a nonlinear transform, D denotes the degree matrix of the graph, which only has values on the diagonal (the rest has the value of 0). These diagonal values represent the degree of the corresponding node. W^l denotes the weight parameters matrix of the GCN that has been initialized in the beginning of the training.

4.3 Similarity Comparison of Page Layout

Layout feature vectors of two pages are compared using the Euclidean distance for similarity measurement. If the similarity is lower than a pre-defined threshold S, these two page layouts are considered dissimilar. If the similarity is above the threshold S, we then use IoU (Intersection over Union) [20] to further determine the similarity of the two pages. The IoU values for all GUI categories are calculated and added up to compare with a pre-defined threshold T. If the total IoU value is higher than T, the two pages are considered to have a similar layout.

For long screenshots, we use a sliding window technique to crop the page into multiple smaller screenshots. The layout similarity of each cropped screenshot is calculated in parallel using multiple threads. If more than 75% of the cropped screenshots are determined to be similar, the layout of the two long screenshots is considered similar. This approach can significantly reduce the time required for layout similarity calculation.

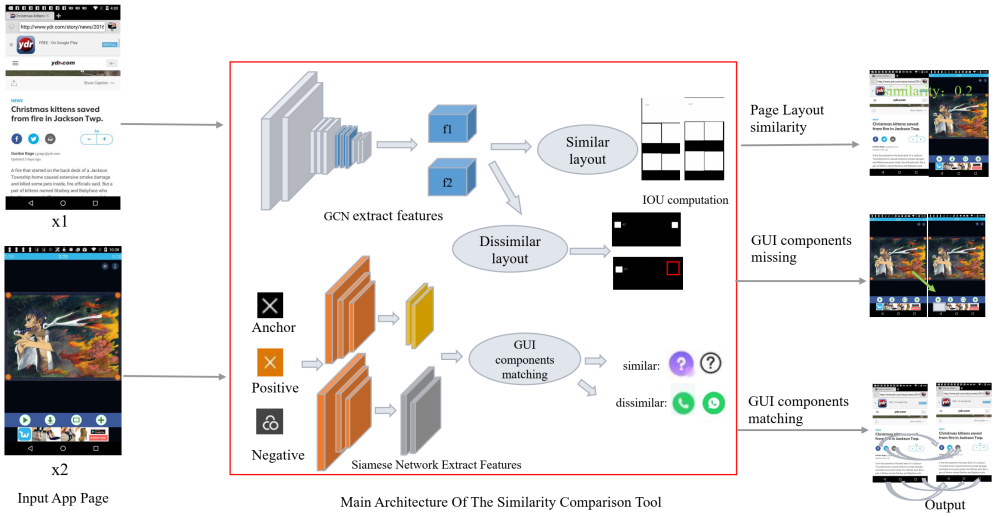


Figure 2: Neat Overview

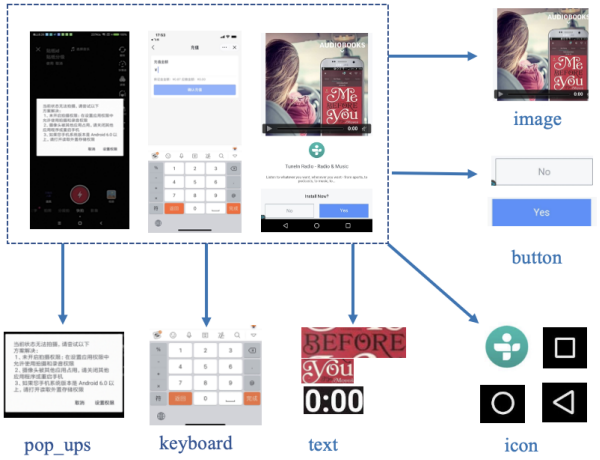


Figure 3: Examples of GUI Component Types

4.4 GUI Component Matching

GUI component matching is used to locate suspected abnormal components when dissimilar GUI pages are detected. Once the page layout similarity is obtained, it is possible to further compare the similarity of local areas of the page by checking whether the GUI components are similar or not. In particular, when dissimilar page layouts are encountered, it is necessary to find the specific GUI components that are not the same. After investigating existing contrastive learning frameworks [4, 6, 7, 10, 12], initial experiment did not show a significant difference of these networks' accuracy achieved for our task. Therefore, we decide to use SimCLR [6] as the backbone network due to its lightweight model structure and quick model inference, which is more suitable for the industrial environment with a high demand for a larger amount of query per second (QPS) The proposed GUI components comparison network is shown in Figure 5.

The process of GUI components similarity comparison and matching consists of the following steps:

(1) Detection of GUI components on the pages using the object detection model.
(2) Generation of positive and negative samples. Data augmentations such as flip and noise addition are used to form positive sample pairs, while negative samples are formed from different classes of GUI components.
(3) Adoption of a siamese network training framework [16] and fine-tuning on the SimCLR model with contrastive loss [16] to train a similarity comparison model of GUI components.
(4) Extraction of components' features through the trained model and calculation of similarity score using the Euclidean distance metric.

5 EVALUATION

Our layout comparison tool is designed for the improvement of layout comparison efficiency and layout issues discovery. Therefore, we investigate the following research questions (RQs):

RQ1. *What is the effectiveness of the similarity comparison capability of Neat?*
To answer this question, we use a collection of 10,000 app pages to evaluate the effectiveness of Neat in detecting and localizing GUI components, and comparing layout similarity, in terms of precision and recall rate. To the best of our knowledge, besides SSIM [36] and GCN-CNN [25], there are no more existing complete methods for page layout comparison that Neat can be compared with, among which [25] is mainly used to find the top K most similar pages, which is different from the goal of our proposed task. Therefore, we can only compare the effectiveness of Neat in GUI similarity camparison with SSIM for this research question.

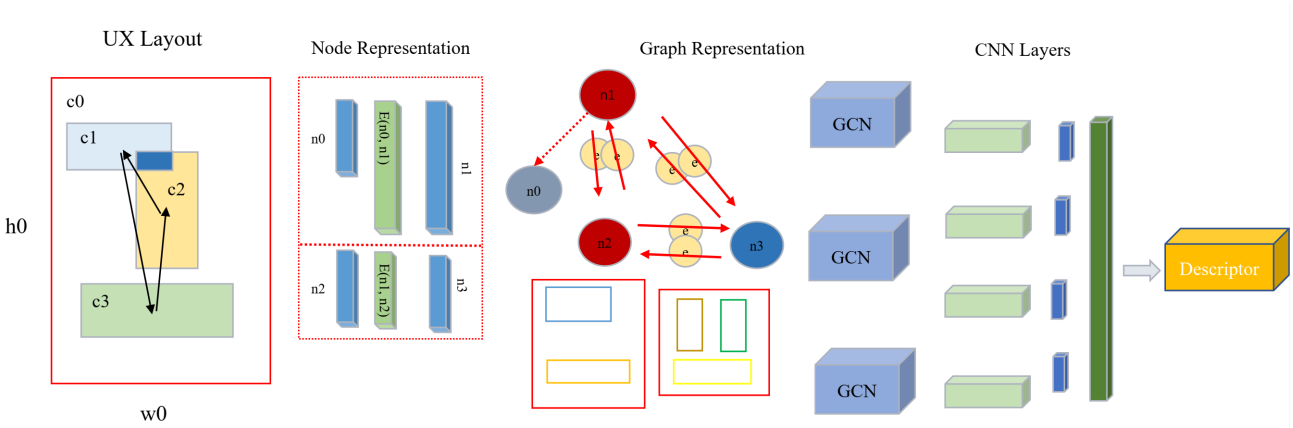


Figure 4: GCN Architecture Overview

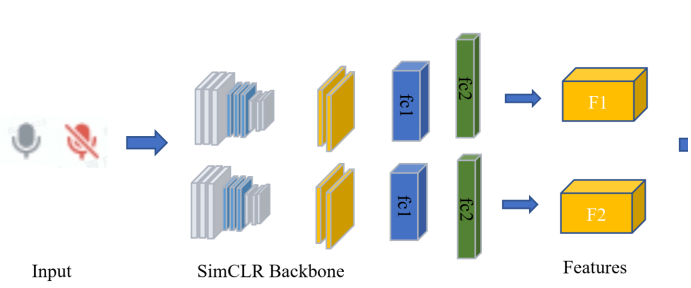
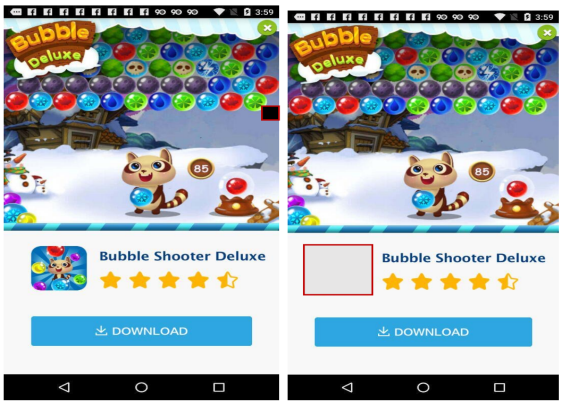
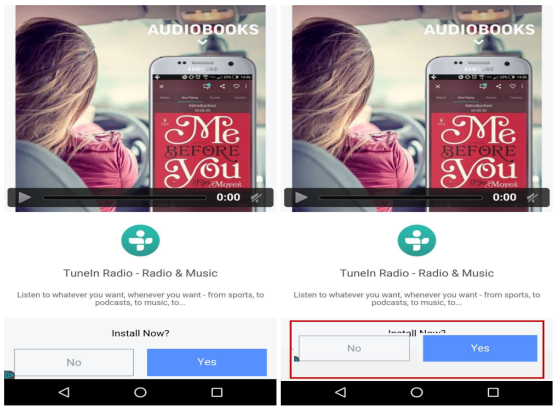


Figure 5: Similarity Comparison Network



(a) GUI Component Missing



(b) GUI Components Misplace

Figure 6: Examples of GUI Components Matching Results(Marked by Red Boxes)

RQ2. *Is the combination of a pre-trained model of contrast learning and the Siamese network effective on GUI components matching?*
We report precision and recall rate achieved by NEAT on 1,000 GUI component images from industrial apps after training the model using 9,000 images. The precision and recall rate curve when the distance threshold changes is also studied for this research question.

RQ3. *What is the performance of our similarity comparison tool for real-world industrial apps?*
For this question, we deploy NEAT in the testing pipeline of multiple app development teams at ByteDance and report the number of layout problems uncovered and human resource saved by NEAT.

5.1 Experiment Setup

Open-Source Dataset. RICO [8] is a large-scale open-source dataset of app layouts and is usually used in domains such as GUI components layout design, automated code generation for GUI components and user perception prediction. RICO is built from 9, 300 free Android apps from 27 different categories and involves 66, 000 unique UI screens in multiple domains, such as interactive, textual, structural display, etc.

Industrial Dataset. We collect 10,000 screenshots of our industrial mobile apps pages, these screenshots are from iOS or Android devices which contain different resolutions and scenes with distinct page layouts.

Configuration. We use 8 NVIDIA Tesla V100-SXM2-32GB GPUs for the training of the YOLOX object detection model, the GCN model and the contrast learning model.

5.2 RQ1. Effectiveness of Similarity Comparison of Page Layout

We illustrate the effectiveness of the tool from experimental results of GUI component detection and layout similarity comparison based on the GCN model, respectively.

5.2.1 *GUI Component Detection.*

Experiment Setup. We collected 10,000 app pages, 50% of these samples is from the RICO dataset, and the others are screenshots of our apps. 9,000 samples are used to train the detection model while another 1,000 are used for validation and testing. We compared the average precision (AP) of all pages based on Faster-RCNN, SSD, YOLOv5 and YOLOX models. The original models are for object detection and are trained on the RICO dataset for GUI component detection by us. The accuracy and speed of the detection model are considered together for the remaining study.

Evaluation Metrics. Mean Average Precision (mAP) [38] is a commonly used to study the effectiveness of object detection techniques and the formula is shown in Equations 2.

$$mAP = \frac{AP}{N} \tag{2}$$

where N represents the number of categories and AP represents the average precision rate achieved for the detection of GUI components from the same category. The average precision is calculated using Equation 3.

$$AP = \frac{p_i}{n_1} + \frac{p_m}{n_2} + \frac{p_b}{n_3} + \frac{p_k}{n_4} + \frac{p_u}{n_5} \tag{3}$$

where p_i, p_m, p_b, p_k and p_u represents the detection precision of each category and ni $(i \in 1, 2, ..., 5)$ represents the number of samples of each category. The larger the mAP, the more accurate the GUI component detection results for all categories.

Frame Per Second (FPS) is also included to evaluate the speed of prediction, as in industrial practice, Neat should test thousands of pages on dozens of different devices for each version update. The frequency of version updates and the time limit of the automated testing phase requires our method to complete each set of comparisons as quickly as possible with guaranteed results.

5.2.2 *Layout Similarity Comparison.*

Experiment Setup. We collected 5,000 real app page pairs as the training set. Another 1,000 pairs of pages are used as validation set for verifying the page similarity comparison effectiveness. All these page pairs are collect from the industrial apps datasets. The threshold S is set to 0.8 empirically. We evaluate the effectiveness on several different types of page layout.

Validation Metric. Precision and recall rates are used for the evaluation of the effectiveness of layout similarity comparison. The IoU threshold T is chosen during the experiment. Precision represents the accuracy of our method in determining whether the pages are similar, and recall represents how many dissimilar pages do our approach recall. Our ground truth data comes from manually confirmed data provided by the quality assurance team

5.2.3 *Results & Analysis.*

Detection Results. The average precision (AP) of GUI components detection based on different models is shown in Table 1. From the detection results, the Faster-RCNN model achieves 3 best AP out of 5 categories while YOLOX achieves 2 best out of 5 categories. Faster-RCNN achieves best mAP. But when taking FPS also into consideration, YOLOX achieves best FPS which is much faster than Faster-RCNN. Therefore, we select YOLOX as the GUI components detection model. The example of GUI components detection results of YOLOX is shown as Figure 7.

Similarity Comparison Results. As shown in Figure 8, with the IoU threshold increasing, precision gradually grows up, eventually reaching about 95%. And the recall decreases rapidly after the IoU threshold is greater than 0.6. When T is higher than 0.5, precision increases slower than before and recall decreases rapidly. Therefore, the T value is selected to 0.5.

To illustrate the higher accuracy of our proposed approach on app pages with different resolutions, we sampled 5,000 pairs of mobile app pages from industrial apps, 50% of the samples have the same resolution, and the others have different sizes, these app screenshots can be divided into multiple categories according to application scenarios, such as promotional pages, product pages, message pages, etc. Then we compare the accuracy of SSIM and our approach. As SSIM is only applicable when the two input images have the same resolution, we need to first scale both pages to the same size. The comparison results are shown as Table 2. As shown in this table, our approach is more accurate in comparison on all types of pages.

5.2.4 *Discussion.*

When layout difference is detected, there may be two types of problems: ① the same operation cannot reach the same page (functional problem), and ② the layout difference is too large (layout problem). Under different resolutions, pages without anomalies may also have subtle layout differences, and the detection has tolerance on this. Although this subtle anomaly might be a real problem, in practice, the severity is much smaller than the two above mentioned problems.

5.3 RQ2. Effectiveness of GUI Components Matching

After training our GUI components comparison model, we conducted experiments to assess its performance on multiple types of GUI components. The best feature distance threshold is determined through experiments, and the effectiveness of our method was demonstrated using precision and recall of components matching.

To train the contrasive learning model, we sampled 10,000 GUI component images from industrial app pages, and used 9,000 of them for training. The positive training pairs were generated from

Algorithm	Icon	Image	Button	Keyboard	Pop-up	mAP	FPS
Faster-RCNN	0.955	**0.92**	**0.96**	0.965	**0.93**	**0.946**	7
SSD	0.91	0.88	0.916	0.921	0.894	0.904	46
YOLOv5	0.95	0.885	0.94	0.959	0.91	0.928	**90.1**
YOLOX	**0.958**	0.906	0.951	**0.968**	0.923	0.941	**90.1**

Table 1: Detection Results

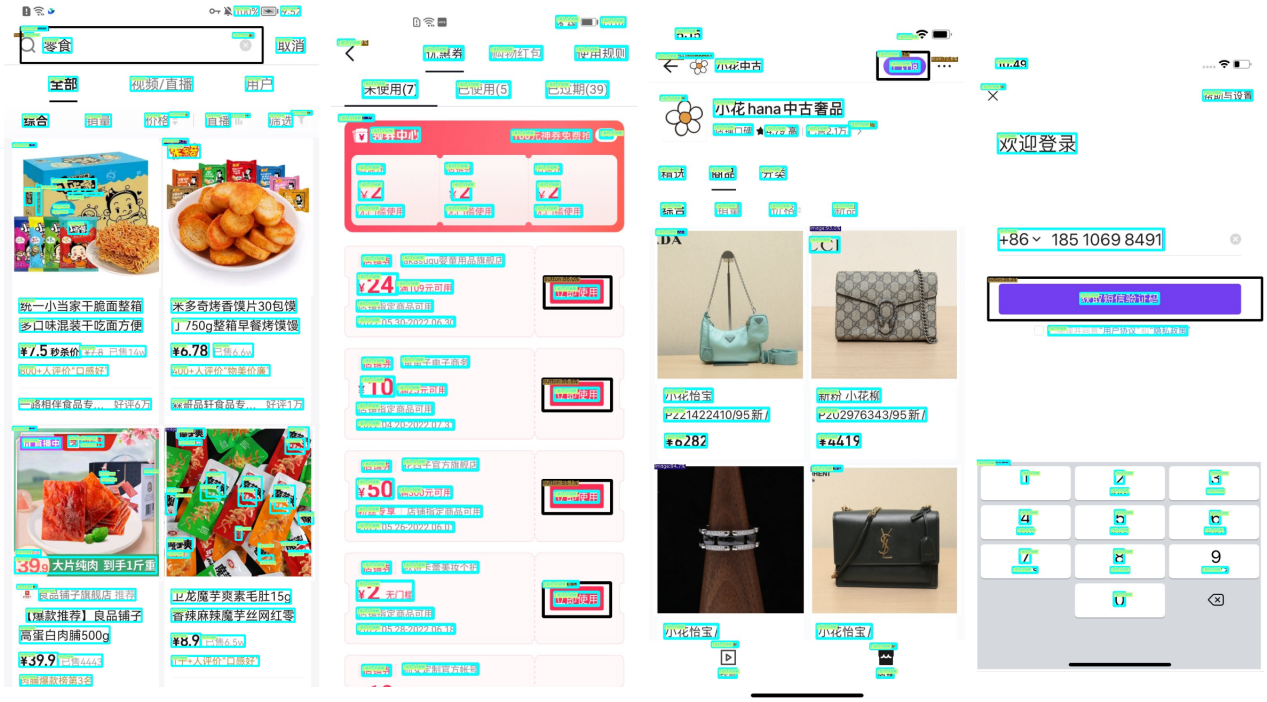


Figure 7: Example Detection Results of YOLOX

Algorithm	Commodity Page	Message Page	Order Page	Publicity Page	Waterfall Page	AP
SSIM	0.625	0.55	0.70	0.65	0.72	0.649
Our Approach	0.95	0.91	0.916	0.85	0.852	**0.895**

Table 2: Average Precision of Different types of Pages based on SSIM and Our Approach.

GUI components of the same type, where one sample and its augmentation result were used as a pair of positive samples. The training negative pairs were generated from GUI components of different types. In total, we had approximately 50,000 pairs for training, with a positive to negative sample ratio of about 6 : 1.

We validated our model on a sampled set of 1,000 GUI components, and determined that the distance threshold for successful component matching was 0.4. This threshold was determined based on the precision and recall curve of the test set, as shown in Figure 9. If the Euclidean distance of two GUI components' features was less than 0.4, we considered the two components to be a match. Otherwise, we output the specific location of the mismatch.

5.4 RQ3. Industrial Application Effectiveness

5.4.1 Industrial Deployment. To assess the effectiveness of our proposed page similarity comparison tool in real-world app development scenarios, we conducted an investigation of its practical application in industry. We evaluated the effectiveness and stability of Neat in the context of practical industrial deployment by providing the tool to multiple app production and development teams and continuously collecting results and feedbacks for half a year.

5.4.2 Diversity of Subject Apps. The app pages in the internal dataset originate from various product lines within the company, containing a wide array of scenarios under different categories of apps. For instance, there are shopping cart pages, product detail pages, and product waterfall display pages for e-commerce apps;

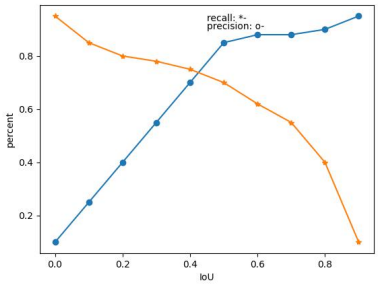


Figure 8: Precision and Recall Curves based on Different IoU Thresholds

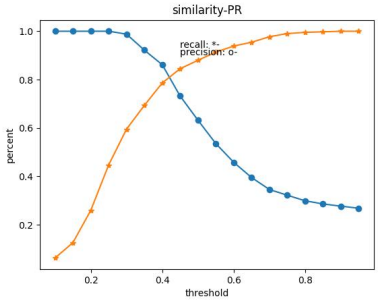


Figure 9: Precision and Recall Curves based on Different Thresholds

poster pages for advertising apps; playback pages and personal profile pages for video apps; text content pages for news apps, etc.

Due to varying device types, these pages come in various resolutions. Owing to differences in GUI design, these app pages each contain different categories of GUI components and distinct layouts.

During data collection, the training and test sets are sourced from different business apps. They may belong to the same category, but the specific content of the pages differs.

5.4.3 Result Analysis. The precision rate of similarity comparison of these industrial pages is shown in Figure 10. It achieves the best precision and recall rate for commodity pages while the worst for publicity pages. For commodity pages, the distribution of GUI components on the page is relatively uniform and follows certain rules. Therefore, the detection effect on such images is better. The more accurate the GUI detection result, the more accurate the subsequent layout similarity calculation will be. On the other hand, for publicity pages, most of them are irregularly designed pages, so the detection results are relatively less accurate, which leads to poorer judgment in similarity comparison.

We counted detailed data from the practical industrial for about half a year. During this period, our tool was called 83, 000 times per month, uncovered 120 app layout problems per month and saved

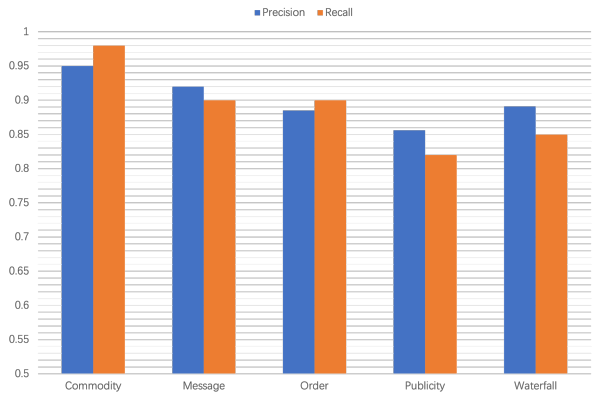


Figure 10: Similarity Comparison Precision and Recall Rate for Different Types of Page Layouts

13.8 man-days per month. The estimated time saving of 13.8 man-days[2] monthly was calculated based on the average time taken to manually identify and rectify such issues, as reported by the development teams. The issues ranged from misalignments and overlapping elements to responsiveness problems across different screen sizes. Importantly, developer feedback indicated that Neat's identification of problematic GUI components substantially streamlined their debugging process.

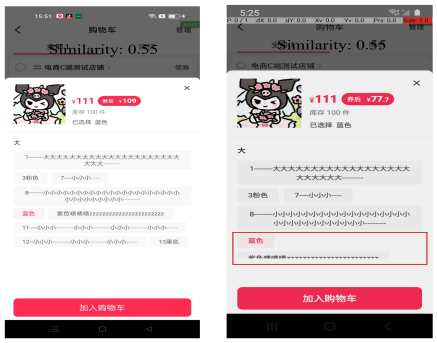
Figure 11 shows real GUI issues detected by Neat, where Figure 11a shows the text layout of the page different from that of the baseline layout, Figure 11b shows that there are more redundant images than the baseline layout in the same position and Figure 11c shows a page inconsistent to the baseline layout due to network or other display anomalies. In addition, some page transition problems were also found by comparing the expected page (that is, the same operation jumped to a different page). In practice, the tool is first used to determine the page similarity and when anomaly is detected, it identifies and marks mismatching components which can be used as helpful information for developers' reference.

Based on the accuracy metrics we gathered from actual business applications (Figure 10) the average precision across various types of pages is approximately 90%. Consequently, there is an approximate 10% incidence of false positives, where our tool incorrectly judges two page layouts to be similar when they are not, potentially leading to missed issues caused by page differences. However, with the current average accuracy rate, the product line considers this acceptable. We will also continue to collect bad cases that arise from actual applications and further optimize the entire automated judgment process.

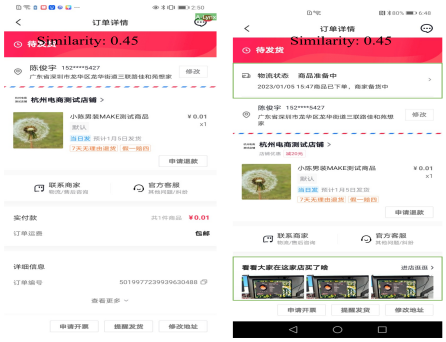
6 LESSONS LEARNED

In this section, we discuss practical experience gained from industrial deployment of our approach.

[2]When calculating the savings in labour costs, we first assume the amount of time it would take for a manual operation to determine whether two pages have similar layouts and GUIs. This assumption is used to estimate the labour cost savings that can be achieved by adopting an automated tool. Therefore, we only consider the cases where the judgment is correct and do not include the time spent on false positives in our calculations.



(a) Dissimilar Text of the Page Layout



(b) Dissimilar Image of the Page

(c) Dissimilarity Caused by Network Problem

Figure 11: Examples of Detected GUI Issues

Layout mismatch issues is a common issue in the app quality assurance practice. Feedback from the quality assurance team shows that, in the actual production environment, there are indeed many layout mismatch problems that impact the user experience and lead to potential user loss. Automated testing tools is helpful in saving human effort for uncovering this issue.

Detection of GUI Components for App Page. It is not hard to get app page screenshots from different scenarios. However, obtaining corresponding layout information for GUI components of app pages rendered on different operating systems can be challenging. For instance, Android uses XML to represent its GUI tree and provides this capability with its official software development kit while this is missing for iOS. To solve this problem, we abstract the GUI components location as an object detection problem, and then we can obtain both locations and category information of GUI components. The YOLOX model is selected as our GUI components detection model due to the fact that comparing with other object detection models, YOLOX has a better trade-off for the inferring time and overall accuracy. We classify the GUI components into 6 categories based on the common app page layout.

Descriptor Extraction of App Page. Although the information of GUI components on the page is known, it is insufficient to describe layout features of the entire page. To extract the layout descriptor of the page, we first attempted to vectorize the GUI components information through four steps

(1) top-down grouping based on the vertical coordinates of the GUI components,
(2) using a 100-dimensional vector to describe the GUI components in each group,
(3) extracting feature vectors of all groups on the page to obtain the page-level descriptor,
(4) comparing the descriptors of each group between two pages to determine the similarity of the page layout.

A GUI page can be divided into a 10 * 10 grid, using the type of component which has the largest proportion in the grid as the value of this grid. The 10 * 10 grid is then flattened into a 100-dimensional vector. For each vector element, 0 represents background, 1 for image, 2 for icon, and 3 for text.

This method performs well on pages with simple layouts, but is not effective for pages with complex layouts and numerous categories and quantities of components, leading to a high number of false positives. After investigating various approaches, we adopted a GCN-based method to extract the app page layout descriptors. In this method, each GUI component is considered a node, and the relationship between GUI components is considered an edge when generating graph data. The GCN network takes the overall structure information of the page as input, and experiments show that this approach provides more accurate similarity comparisons.

7 CONCLUSION

In this paper, we introduce Neat, an automated tool for comparing the layout similarity of mobile app pages, comprising three key components: GUI component detection, app page layout descriptor extraction, and GUI component matching. The practical deployment of Neat has demonstrated its ability to effectively and stably identify problems with mobile app pages, thereby significantly reducing the workload of quality assurance engineers during the auditing and automated testing phases. The tool has been successfully utilized by several mobile app development teams over the course of 6 months, detecting hundreds of real problems and saving approximately 13.8 man-days per month. By providing a more convenient and reliable means of testing GUI components, this tool is beneficial to both designers, developers and testers.

ACKNOWLEDGMENT

We would like to thank all the Reviewers for taking the time and effort necessary to review the paper. We sincerely appreciate all valuable comments and suggestions, which helped us to improve the quality of the manuscript.

REFERENCES

[1] Efthimios Alepis and Constantinos Patsakis. 2017. Monkey says, monkey does: security and privacy on voice assistants. *IEEE Access* 5 (2017), 17841–17851.

[2] Relja Arandjelovic, Petr Gronat, Akihiko Torii, Tomas Pajdla, and Josef Sivic. 2016. NetVLAD: CNN architecture for weakly supervised place recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 5297–5307.

[3] Tianqin Cai, Zhao Zhang, and Ping Yang. 2020. Fastbot: A Multi-Agent Model-Based Test Generation System Beijing Bytedance Network Technology Co., Ltd.. In *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*. 93–96.

[4] Mathilde Caron, Ishan Misra, Julien Mairal, Priya Goyal, Piotr Bojanowski, and Armand Joulin. 2020. Unsupervised learning of visual features by contrasting cluster assignments. *Advances in neural information processing systems* 33 (2020), 9912–9924.

[5] Ushasi Chaudhuri, Biplab Banerjee, and Avik Bhattacharya. 2019. Siamese graph convolutional network for content based remote sensing image retrieval. *Computer vision and image understanding* 184 (2019), 22–30.

[6] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*. PMLR, 1597–1607.

[7] Xinlei Chen and Kaiming He. 2021. Exploring simple siamese representation learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 15750–15758.

[8] 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 ACM Symposium on User Interface Software and Technology*. 845–854.

[9] Zheng Ge, Songtao Liu, Feng Wang, Zeming Li, and Jian Sun. 2021. Yolox: Exceeding yolo series in 2021. *arXiv preprint arXiv:2107.08430* (2021).

[10] Jean-Bastien Grill, Florian Strub, Florent Altché, Corentin Tallec, Pierre Richemond, Elena Buchatskaya, Carl Doersch, Bernardo Avila Pires, Zhaohan Guo, Mohammad Gheshlaghi Azar, et al. 2020. Bootstrap your own latent-a new approach to self-supervised learning. *Advances in neural information processing systems* 33 (2020), 21271–21284.

[11] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 269–280.

[12] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. 2020. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 9729–9738.

[13] Alexander Hermans, Lucas Beyer, and Bastian Leibe. 2017. In defense of the triplet loss for person re-identification. *arXiv preprint arXiv:1703.07737* (2017).

[14] Xiaowei Jia, Handong Zhao, Zhe Lin, Ajinkya Kale, and Vipin Kumar. 2020. Personalized image retrieval with sparse graph representation learning. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. 2735–2743.

[15] G Jocher et al. 2022. Ultralytics/yolov5: v7. 0—YOLOv5 SOTA Realtime Instance Segmentation.

[16] Gregory Koch, Richard Zemel, Ruslan Salakhutdinov, et al. 2015. Siamese neural networks for one-shot image recognition. In *ICML deep learning workshop*, Vol. 2. Lille, 0.

[17] Valéria Lelli, Arnaud Blouin, and Benoit Baudry. 2015. Classifying and qualifying GUI defects. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE, 1–10.

[18] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: A deep learning-based approach to automated black-box android app testing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1070–1073.

[19] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. 2016. Ssd: Single shot multibox detector. In *European conference on computer vision*. Springer, 21–37.

[20] Jeffri M Llerena, Luis Felipe Zeni, Lucas N Kristen, and Claudio Jung. 2021. Gaussian bounding boxes and probabilistic intersection-over-union for object detection. *arXiv preprint arXiv:2106.06072* (2021).

[21] Zhengwei Lv, Chao Peng, Zhao Zhang, Ting Su, Kai Liu, and Ping Yang. 2022. Fastbot2: Reusable Automated Model-based GUI Testing for Android Enhanced by Reinforcement Learning. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.

[22] 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*. 224–234.

[23] Sonal Mahajan and William GJ Halfond. 2015. Detection and localization of html presentation failures using computer vision-based techniques. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–10.

[24] Sonal Mahajan, Bailan Li, Pooyan Behnamghader, and William GJ Halfond. 2016. Using visual symptoms for debugging presentation failures in web applications. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 191–201.

[25] Dipu Manandhar, Dan Ruta, and John Collomosse. 2020. Learning structural similarity of user interface layouts using graph networks. In *European Conference on Computer Vision*. Springer, 730–746.

[26] Kevin Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. 2018. Machine learning-based prototyping of graphical user interfaces for mobile apps. *IEEE Transactions on Software Engineering* 46, 2 (2018), 196–221.

[27] Chao Peng, Zhao Zhang, Zhengwei Lv, and Ping Yang. 2022. MUBot: Learning to Test Large-Scale Commercial Android Apps like a Human. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 543–552.

[28] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems* 28 (2015).

[29] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4510–4520.

[30] Josep Sempau, Scott J Wilderman, and Alex F Bielajew. 2000. DPM, a fast, accurate Monte Carlo code optimized for photon and electron radiotherapy treatment planning dose calculations. *Physics in Medicine & Biology* 45, 8 (2000), 2263.

[31] Fei Shao, Rui Xu, Wasif Haque, Jingwei Xu, Ying Zhang, Wei Yang, Yanfang Ye, and Xusheng Xiao. 2021. WebEvo: taming web application evolution via detecting semantic structure changes. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 16–28.

[32] Yuhui Su, Zhe Liu, Chunyang Chen, Junjie Wang, and Qing Wang. 2021. OwlEyes-online: a fully automated platform for detecting and localizing UI display issues. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1500–1504.

[33] Fei Wang, Mengqing Jiang, Chen Qian, Shuo Yang, Cheng Li, Honggang Zhang, Xiaogang Wang, and Xiaoou Tang. 2017. Residual attention network for image classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3156–3164.

[34] Xiaoyu Wang, Tony X Han, and Shuicheng Yan. 2009. An HOG-LBP human detector with partial occlusion handling. In *2009 IEEE 12th international conference on computer vision*. IEEE, 32–39.

[35] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing* 13, 4 (2004), 600–612. https://doi.org/10.1109/TIP.2003.819861

[36] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing* 13, 4 (2004), 600–612.

[37] Wikipedia. 2024. *Longest common subsequence*. https://en.wikipedia.org/wiki/Longest_common_subsequence

[38] Wikipedia. 2024. *Mean Average Precision*. https://en.wikipedia.org/wiki/Evaluation_measures_(information_retrieval)

[39] Rahulkrishna Yandrapally, Andrea Stocco, and Ali Mesbah. 2020. Near-duplicate detection in web app model inference. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 186–197.

[40] Zheng-Wu Yuan and Jun Zhang. 2016. Feature extraction and image retrieval based on AlexNet. In *Eighth International Conference on Digital Image Processing (ICDIP 2016)*, Vol. 10033. SPIE, 65–69.

[41] Razieh Nokhbeh Zaeem, Mukul R Prasad, and Sarfraz Khurshid. 2014. Automated generation of oracles for testing user-interaction features of mobile apps. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 183–192.

[42] Si Zhang, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. 2019. Graph convolutional networks: a comprehensive review. *Computational Social Networks* 6, 1 (2019), 1–23.

[43] Zhaolong Zhang, Yuejie Zhang, Rui Feng, Tao Zhang, and Weiguo Fan. 2020. Zero-shot sketch-based image retrieval via graph convolution network. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 12943–12950.

[44] Zhong-Qiu Zhao, Peng Zheng, Shou-tao Xu, and Xindong Wu. 2019. Object detection with deep learning: A review. *IEEE transactions on neural networks and learning systems* 30, 11 (2019), 3212–3232.

[45] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph neural networks: A review of methods and applications. *AI Open* 1 (2020), 57–81.