Object Detection Techniques for Reverse-Engineering Large-Scale UI Design Resources

Linzhao Wu

A report submitted for the course
COMP8755 Computer Science Individual Projects
Supervised by: Zhenchang Xing
The Australian National University

October 2018

Except where otherwise indicated, this report is my own original work.

Linzhao Wu
25 October 2018

Acknowledgments

Much appreciate the supervision and support from my supervisor Dr Zhenchang Xing. Without his guidance and ideas, I would not be able to achieve the goals of this project. His great supports allow me to overcome numerous difficulties

I also need to thank Dr Chunyang Chen, who provided me many valuable suggestions which accelerated the progress of this project tremendously.

At last, I am very grateful to the mentally support of my family in every possible way.

Abstract

Graphical User Interface (GUI) design on mobile applications has always been a challenging work for GUI designers. They need to be able to design absorbing, easy-to-use GUI for users, as GUI provides the most direct impression to the users of an application. Considering the pivotal role of GUI to a mobile application, designers frequently need to be inspired by the work of other designers in the industry to invent more novel and pragmatic GUI for software developers to realise. However, most tools for designers are either with very limited number of UIs or lack of specific component categories of GUI.

In our project, we propose approaches to automatically collect a large-scale existing GUIs, and effectively generate specific categorised GUI components to provide inspirations for designers. We firstly collect GUI screenshots with components meta information using an automated GUI testing tool [Su et al., 2017], this annotated data is then used in the following machine learning steps for training neural-network based object detection models. In this step, a multitude of experiments are carried out to improve the accuracy of this neural network. After that, we apply an object detection deep-learning network, Faster R-CNN [Ren et al., 2015b], to be able to identify widgets from raw large-scale GUI screenshots which have no annotated information in it. Then, this large number of categorised widgets is clipped according to the results of object detection to form the large-scale widgets gallery for designers. At last, we developed a web application [mui] for accessing this gallery of GUI design components.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Mobile applications are playing an increasingly important role in our daily lives, they make our lives better by providing numerous functions. At the same time, the number of mobile applications has been huge, reaching 3.6 million in 2017. As a result, it is becoming more difficult for one application to be noticed by users, and even harder to make it stand out from its competitors. To make an application be a successful one, there are many factors need to be considered. Apart from providing various and practical functions, a refreshing and absorbing GUI is of vital importance. This is because it not only determines the direct experience the users have when they are using it, but also presents the most important first impression for them before they choose and download it.

Therefore, designing an outstanding and attractive GUI is a challenging work for designers, which requires a great amount of experience and talents to rely on. They need to be able to be innovative about their work while keeping up with the latest trend in the industry. Thus, the capability of being able to reference the work of other designers and get inspirations out of them is critical, which makes the resources with a large number of existing and successful mobile GUIs become highly useful and valuable. On one hand, these resources will speed up their work and make their work less time-consuming and rely less on experience. On the other hand, if an application has millions of followers, there must be something extraordinary in their UI design, so if designers can be inspired by their ideas, they are more likely to make their own work a successful one.

1.2 Related work

There are numerous websites available for mobile GUI designers. For instance, some of them have numerous comprehensive mobile app design templates collected from Internet, for example "Pinterest" [Pinterest]. but the drawback of these websites is that the design patterns are not specifically categorized. There are also websites for mobile UI designers which have detailed categories of UI design, such as "Inspired UI" [Pakhandrin], "Pttrns" [Labs] and so on. They provide many kinds of design

patterns of mobile UI and a variety of general ideas for designers of the whole screen of mobile, but have no independent categorised widgets provided directly. There is one type of websites that provide downloadable and editable widgets, a typical one is the website "Axure Widgets" [Oliver]; however, they are charged to use, so it is not suitable for inspiring design ideas for designers. Another type of websites, such as "Up Labs" [Up], have free widgets, but they are for personal use only. In comparison, our website is a gallery of a considerable amount of UI components for the sake of reference for designers when they develop their own ideas of UI and the UI widgets are categorised explicitly.

Neural network is applied to UI detection in the work of Chen [Chen et al., 2018a], in which widgets are automatically extracted from UI images and translated into GUI skeletons of Android. The approach of deep learning is used in that detection, which is a neural machine translator. Firstly, Convolutional Neural Networks (CNN) is applied to extracted features of input images. Then, they use Recurrent Neural Network (RNN) to encode the spatial features of those images. At last, RNN decoder is used to construct the GUI skeleton, which generates desired outcome. One similar work in this domain of mobile UI reverse engineering is the one done by Nguyn [Nguyen and Csallner, 2015], but the technique used is computer vision (CV) and optical character recognition (OCR). By contrast, the work of GUI detection and corresponding code generation done by Moran [Moran and Bernal-C'ardenas, 2018] uses combined approaches. They use CV techniques to detect components, and CNN is utilised to classify widgets within GUI. After that, K-nearest-neighbors (KNN) based algorithm is applied to assemble the hierarchy of components. The similarity of all the work above is that they detect widgets from the GUI of mobiles to generate code the hierarchy layers for the utilisation of programmers, which provides them ideas about the choices when they are developing the software supporting dedicated UI. Nonetheless, our work focusses on assisting designers to inspire them by large scale of existed widgets. Other than those work listed, there are a few loosely related works in the domain of detecting images from mobiles. For instance, in the work of Jo and Jung [Jo and Jung, 2016], logos of mobile phone applications are detected. The approach they use is a traditional one, they search websites and storing the logos and their corresponding website names into a database. In which way, they achieve the function of "smart learning". Other studies related to UI design focus on the researches of UI design patterns, such as the study of Meier, Heidmann and Thom [Meier et al., 2014], they find the relationship between location search patterns and user requirements.

1.3 Objective

Nowadays, there are two most widely used mobile platform, which are iOS and Android, both have a great number of GUI components in their application. Our work mainly focuses on Android apps; however, these approaches can also be applied to other platforms. The objective of this project is to develop a large-scale mobile GUI design resource using reverse-engineering method, which is used as the database of

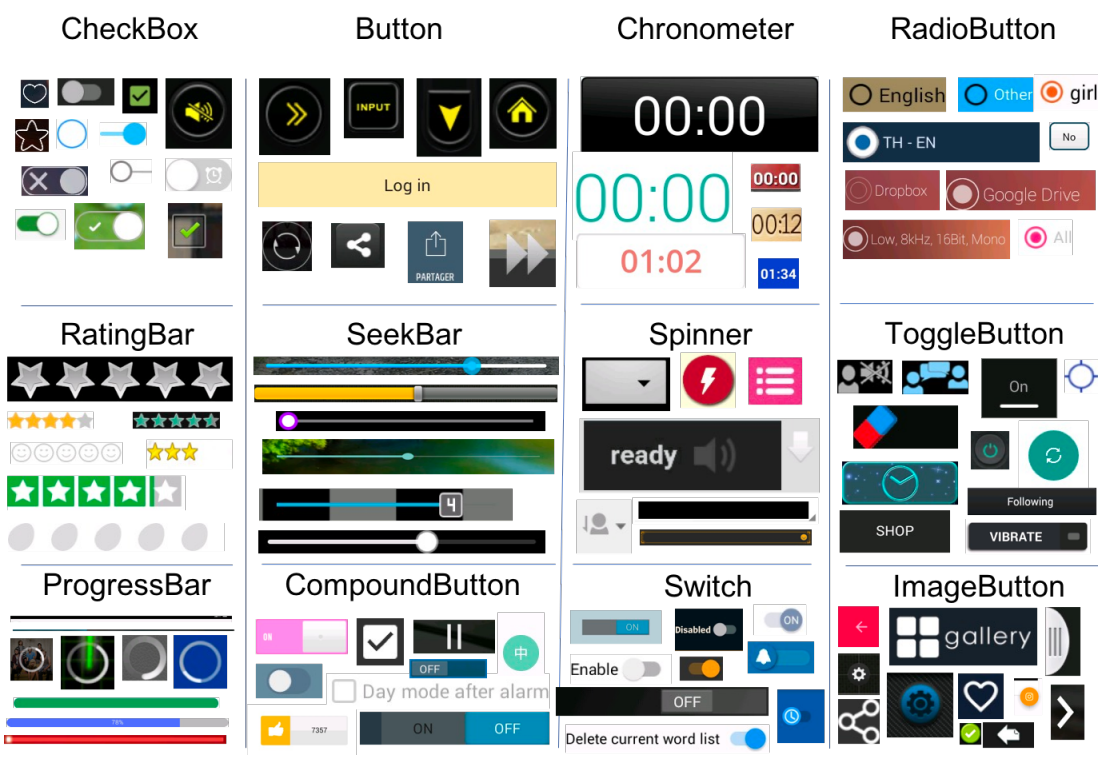


Figure 1.1: 12 types of Android GUI widgets

a UI gallery. The targets in our project are the 12 types of Android GUI widgets, that can be divided into 2 categories: buttons and information bars. More specifically, the category of buttons includes Button, Image button, Compound button, Radio button, Toggle button, Check box, and Switch. Information bars consist of Progress bar, Seek bar, Rating bar, Spinner, and Chronometer.

Figure 1.1 exhibits the examples of these widgets.As shown in this figure, the colours, shapes, sizes of widget in each type are essentially distinct, so providing a gallery for designers to access the design of widgets extracted from thousands of existed applications would make their work faster and more efficient. However, we ignore the text related widgets into consideration as their appearances are highly dependent on their content, which are irrelevant to GUI design.

Chapter 2

Methodology

There are four major steps in our approaches, GUI screen-shot collecting, processing, deep-learning model training, and large-scale widgets generating. The first section is about the collecting of the two data-sets of our projects, one is the screen-shot with meta-data which is used for training and testing, while the other one is pure screen-shot without annotations that is for evaluating the performance of our deep-learning module. The second section is the processing of these two data-sets we have been through for the purpose of better quality of the detection of widgets. The third section talks about the mechanism of our object-detection deep-learning module. The fourth is about the generating of the large-scale widgets that we used as the database of our UI-gallery for GUI designers.

2.1 Collecting GUI data-set

In the step of app screen-shot collecting, we need to collect two kinds of screen-shots. The first one is collected by running Android application test framework on downloaded Android apps, where these screen-shots are collected with meta-data (layout information), such as the type, location of the widgets. This data-set is for the purpose of training and testing the Faster R-CNN deep-learning model, which will be called Data-set A below for the sake of avoiding confusion. The other large-scale data-set is made by crawling the introduction screen-shots of applications in Google Play with no meta-data, which is called Data-set B below. The second data-set is firstly applied by the trained model to recognize the bounds of diverse types of widgets within the screen-shots. Then, these widgets are clipped and categorized into distinct types as the database of the GUI gallery for designers.

There are two reasons we choose to use another data-set instead of using the first data-set (with meta-data) to generate the database. The first is collecting screen-shot by the Android test framework Stoat is time-consuming (each application need to run for 45 minutes) that requires automatically exploring as many states in that application as possible to get screen-shot of each state; therefore, it is unsuitable for generating large-scale data-set. More importantly, the Android test framework just randomly capture the screen-shot, so it is difficult to maintain the design quality of these screen-shots. By contrast, most introduction screen-shots that are used

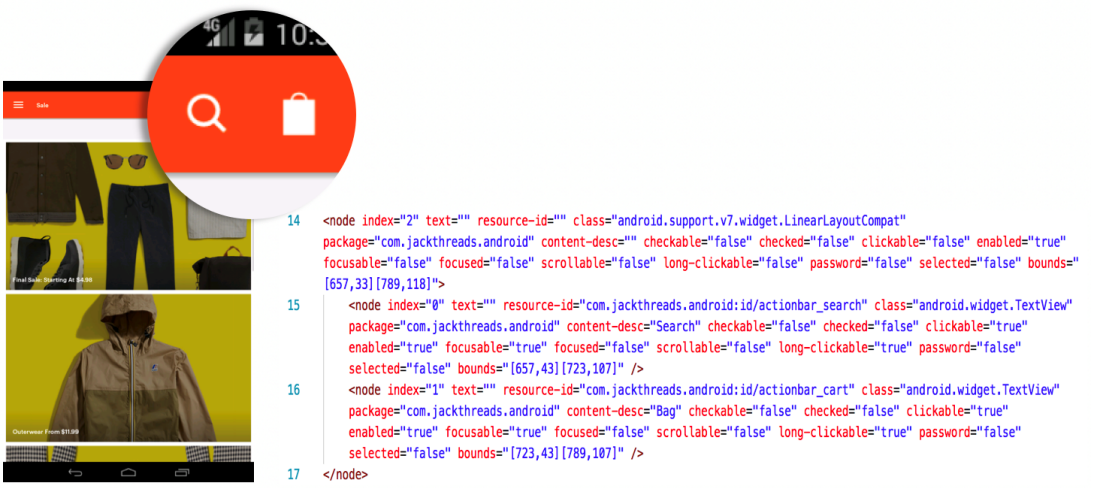


Figure 2.1: Example pair of UI screen-shot and its corresponding XML layout

in Google Play are elaborately selected by developers with their most typical GUI, thereby having better quality in terms of GUI design.

2.1.1 Collecting Android UI data-set with meta-data (Data-set A)

For the purpose of training the deep-learning model to recognize diverse classes of widgets and locate their position, we need a considerable number of pairs of UI images and their corresponding meta-data. The tool we use to collect the desired data is an Android test framework called Stoat [Su et al., 2017] which is for exploring states of Android applications. Due to the fact that Android applications are designed base on XML layout, we can use Stoat to navigate through the applications, take screen-shots and record their corresponding XML layout to collect the pairs of screen-shot images and their meta-data. Figure 2.1 illustrates the pair of widgets and their corresponding XML layout. We firstly crawl 7750 Android app packages from Google Play, then after applying Stoat on these applications, we manage to obtain 795817 pairs of GUI screen-shots and XML layout. However, there are great amount of duplication that need to be removed.

2.1.2 Collecting introduction screen-shots in Google Play (Data-set B)

In Google Play, every application has its own introduction information, the most conspicuous content is the screen-shots that gives potential users the typical appearance of that app. Normally, the developers put their satisfying design in it to attract users to download their app which makes it valuable for our purpose.

From 126297 Android apps in Google Play, we manage to crawl 469176 introduction screen-shots which have no meta-data attached. These data are used to be applied to the trained Faster-RCNN model [Ren et al., 2015a] to identify widgets in it, which is introduced in section 2.3. Even though it is possible that some of these

applications are the same as some of the ones that are used to collect the first data-set, it is unlikely that screen-shots in these two data-sets overlap as the maximum number of introduction screen-shots one app can have is just 8, and the number of apps that used in the data-set B is far larger than that used in data-set A. Note that data-set B has no duplication in it, thus can be used to clip into widgets directly without the necessity to remove duplicity.

2.2 Data processing on UI screen-shots with meta-data

Being the training data of the deep-learning model of Faster-RCNN, the data-set A is of vital importance, it needs to be in a high quality in the perspective of training a deep-learning module. It directly determines the training results so does the quality of the final database of GUI gallery. Therefore, we conduct many processes to refine the original data-set(data-set A) for the sake of improving the quality of it.

2.2.1 Constructing training data-set (Data-set A)

There are many kinds of data-set used nowadays for object detection networks, Pascal VOC data-set [Everingham et al., 2010] is one of them, which is provided for the PASCAL Visual Object Classes (VOC) challenge. This data-set is collected for the purpose of object recognition in 20 pre-specified object classes from realistic visual scenes, there are images and corresponding annotations provided for supervised learning problems. The implementation of Faster-RCNN [Chen and Gupta, 2017a] we choose also needs to be trained and tested using VOC data-set. Therefore, to train our Faster-RCNN module to detect classes of android UI widgets, we firstly need to translate our data-set A according to the format of VOC. The file structure of our generated VOC data-set is shown in Figure 2.2. The "PNGImages" directory includes the screen-shots in Data-set A, while the directory named "Annotations" stores the meta-data in "Json" format, corresponding to the files with same names in "PNGImages". The three most useful files in "ImageSets" are "trainval.txt", "train.txt" and "val.txt", and the other two are statistic information and the data collecting information about VOC generating. The "train.txt" consists of the corresponding names in "PNGImages", meaning that the files with these neames are in training data-set, while The "val.txt" means those files are in the testing data-set. The file "trainval.txt" is the union set of these two sets. We ignore unnecessary directories in the original structure of VOC (e.g. The SegmentationClass, SegmentationObject etc.), only reserve the ones that are useful for the training and testing of our Faster RCNN module.

We only preserved the XML nodes which are related to targeted widgets in the XML files (such as the leaf nodes and attributes) of the data-set A and exclude the irrelevant XML layout. In addition, as there is a great amount of duplication in the automatically collected data-set, we need to remove duplication while constructing the data-set. This procedure is introduced in section 2.2.2. At the same time, to make the data-set more similar to the crawled introduction screen-shot in data-set

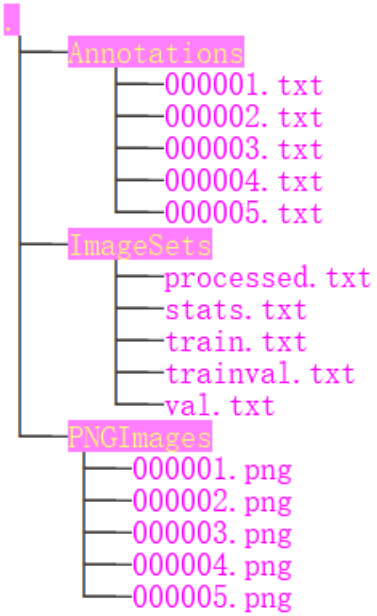


Figure 2.2: The example file structure of our generated VOC data-set

B, we applied the method of image augmentation, this step is described in below section 2.2.3. Another process on the data-set is balancing the number of widgets in the training data-set which is introduced in section 2.2.4.

After above processes, we split the whole data-set A into training data-set and testing data-set in a proportion of 14 to 1.

2.2.2 Removing the duplication

Most mobile applications have GUI design consistency in different user interfaces within one application. For instance, the design of buttons is the same in distinct interfaces or operation stages. One benefit of it is that makes users explore the application more easily. However, that makes many different UI having exactly the same widgets. At the same time, the way the Stoat explores an application is by emitting various UI operations, such as scrolling, clicking, editing, which also introduces a great amount of duplication. For example, when the Stoat goes into a preference setting screen, it tries every possible operation on that screen and catch screen-shot for each one of them. Even trivial operations such as a text input or a contact list scrolling are considered as operations and screen-shots are taken. These screen-shots are similar to each other in the perspective of UI design, as they contain same widgets, as shown in Figure 2.3 (a)(b), the content in the red rectangle are made by the trivial operations that make no differences when one focus on the design of widgets.

Therefore, we analyze the data-set A. The data density of screen-shots per app is shown in tableTable 2.1. After removing those apps that have no screen-shots, the

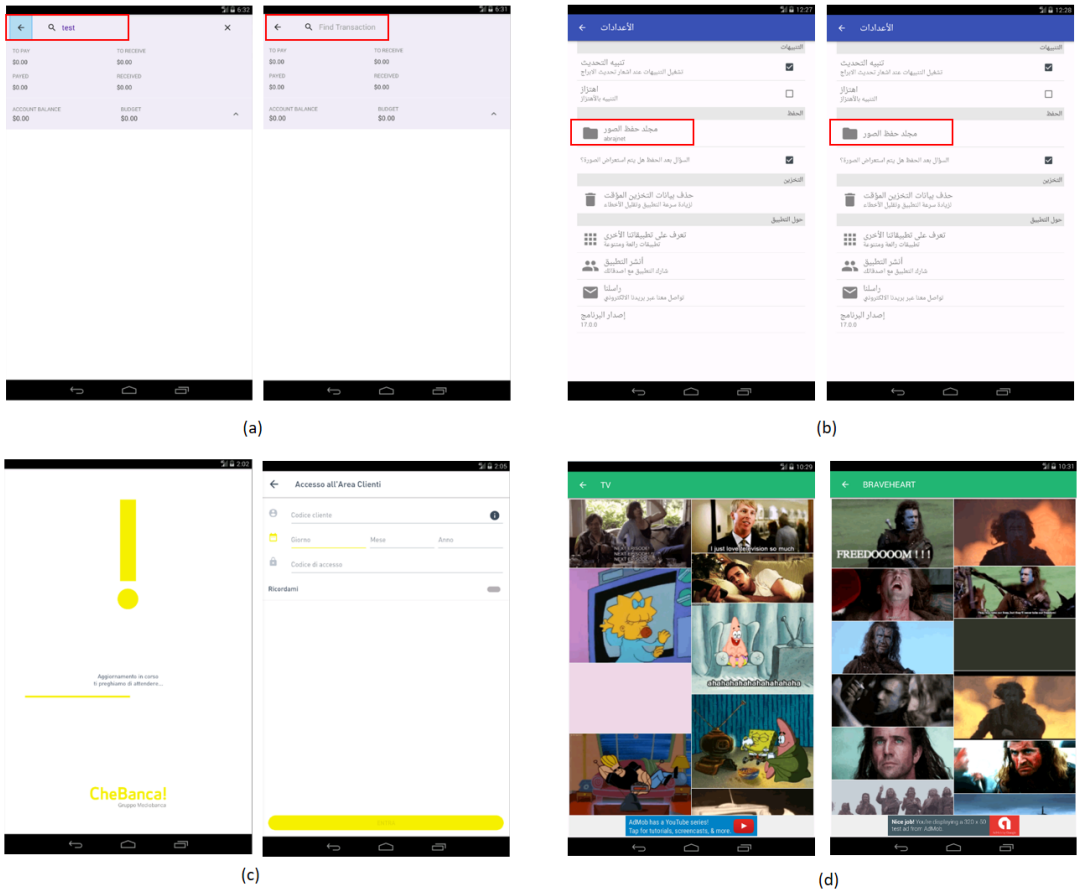


Figure 2.3: Example of duplication in data-set A

Table 2.1: The density distribution of containing screen-shots of apps in data-set A; the three middle rows are corresponding to the statistic data of the applications that have greater than specific number of screen-shots; for example, the second row is about the applications that have more than 1000 screen-shots

	App Number	App Proportion	Image number	Image proportion
The total number	7748	100%	795719	100%
> 1000 screen-shots	113	1.46%	200204	25.16%
> 500 screen-shots	337	4.35%	344771	43.33%
> 200 screen-shots	1123	14.49%	590294	74.18%
No screen-shots	2454	31.67%	0	0%

median number of screen-shots of each app is 47, while the average number is 142.7. From the analysis, we can see there is a considerable amount of unbalance in terms of number of screen-shots in each application. If one application allows the operation of rolling and text inputting, it can have a huge number of similar pictures in one application. Then, we manually went through the applications that have more than 1000 screen-shots, that around 99% of them are similar screen-shots. If we feed these similar screen-shots as they are into our deep learning module, there would cause serious over-fitting. Therefore, removing duplication is essential in the data-set A.

To remove duplication, we need to have a certain kind of measure to recognize the similarity between images. We use three approaches to check the similarity of screen-shots.

The first way to check similarity is by checking the XML attributes of the corresponding screen-shots. We firstly remove the XML nodes that have no widget information we are interested in. Then, we form a list that contains the string of the class and boundary of the corresponding widgets for each XML file. Then, we calculate the similarity of the strings in these two lists to show the similarity. This is measure is applied considering the fact that if two pictures are similar, they must have similar information of widgets(e.g. classes, boundaries). More specifically, if the shorter list of widgets has two thirds of its elements having the same information as the longer list, then the similarity of these two lists is calculated as 66.66%.

After the first round of eliminating duplication, we use the method of image histogram which uses the pixels intensity values of images to represent similarity between images. We first transform the two images into the same size and calculate the histogram value of these two. Then, we calculate the difference of these two arrays divided by the sum of them as the difference value of them. The approach of histogram needs to be used after the first one. There are two reasons. The first is that it only takes the density of colours into consideration, so when two pictures have similar colour density while distinct widgets layout, they are still similar in the point of view of histogram, as illustrated in Figure 2.3(c). The other reason is This measure is slower than the comparison of XML lists, as it need to scan every pixel in that screen-shot while the XML measure only need to compare two short string lists

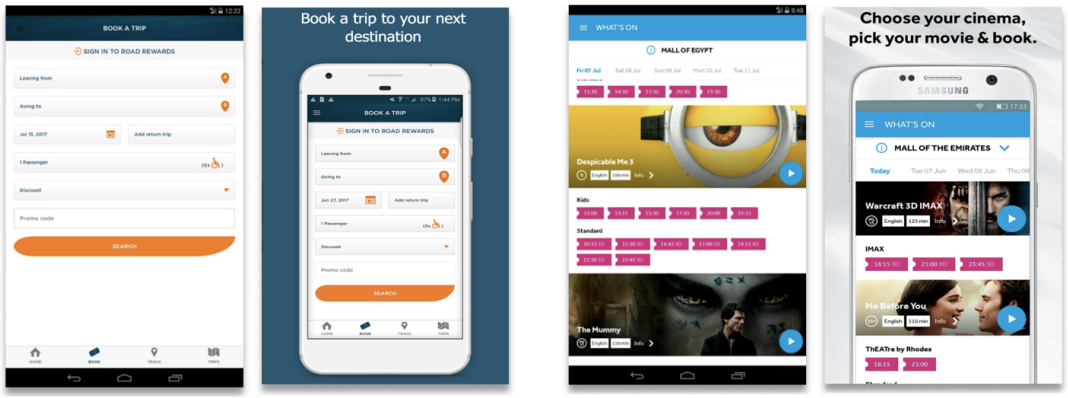


Figure 2.4: The comparison of screen-shots in data-set A and B

that contains useful information of widgets. This time gap makes huge difference when the number is in a large scale.

After that, there are still some pictures are the widely different in terms of XML layout checking and histogram checking, as shown in Figure 2.3(d). Although they are different from each other in terms of XMl similarity and image histogram, they are similar pictures in the perspective of picking out the widgets we want. The number of such kind pictures in one application can be hundreds, so we just randomly keep 15 of them to eliminate the unnecessary duplication in widgets.

2.2.3 Training data augmentation

As mentioned in section 2.1, the data-set A that is used to train our Faster-RCNN module is the screen-shot caught directly using the Android test tool, Stoat. This means that the content in the screen-shots of data-set A come only from the applications. However, the data-set B which is used to apply our model to not only contains application screen-shots but also has the demo view with the frames of mobile phone in it to present potential users an overall impression of using their application. Figure 2.4 illustrates the comparisons of two applications between the real screen-shots in application (as the ones we have in data-set A) and the introduction screen-shot in application store (as the ones we have in data-set B). These differences are wide enough to result in deficient performance of the model training.

Therefore, to minimize the differences of two data-sets, we adopt the method of image augmentation to make the data in these two data-sets resemble to each other as much as possible. Another benefit of conducting image augmentation is it increases the diversity of the training data and expand the quantity of training data (data-set A), which improves the accuracy of training [Wimmer et al., 2017]. Image augmentation applies small mutation to the original images, such as re-sizing, cropping, padding and so on. According to our observation, the introduction pictures in app store are shrinking screen-shots located in the middle with words around it, as shown in

Figure 2.4. Thus, to make the screen-shots look like the introduction pictures, we implement two kinds of image augmentation. In the first one, we randomly shrink the image in data-set A to 0.6-0.9 of their original size and also randomly move it up and down from the centre as they are in the introduction screen-shots in data-set B. Correspondingly, the meta data of components in the screens undergo the same mutation as that of screen-shots. Due to the variety of the colours around the screen-shots in data-set B, we also choose distinct colours for paddings. The top three pictures shown in Figure 2.5 (a) are the examples of the results of image augmentation. In the second kind of image augmentation, to make our processed screen-shots in data-set A more similar to the ones in data-set B, we choose 5 demo pictures of cellphone frame in data-set B, and replace the display part of that image with the shrunk and moved image of the one we choose in data-set A. The chances of choosing which one of these 5 frames for each image is equal. The bottom three images shown in Figure 2.5 (b) are the instances of this kind of image augmentation. We conduct image augmentation to all the screen-shots in our data-set A to expand our training data-set, while the probability of using the first approach of image augmentation is 80% and that of using second method is 20%.

2.2.4 Balancing components

Due to the nature of widgets and the habits of developers, some of widgets are widely used, such as Button and ImageButton, while some of widgets are seldom used such as Chronometer and CompoundButton. Therefore, there are a great amount of unbalance of widgets in the data-set A. The tableTable 2.2 shows the distribution of the widgets. The sum of Button and ImageButton account for 78.29% of the total widgets, on the other hand, the CompoundButton and Chronmeter only constitute 0.18%.

Although this unbalanced distribution does reflect the real possibility of the appearance of widgets in mobile applications, while the over low rate existence of CompoundButton and Chronmeters make them much less likely be recognized when they should be. Therefore, we choose to have more tolerance for those similar pictures if they have rare widgets, which means we set lower threshold when we remove the similarity for pictures. Additionally, since we split the data-set A into training data-set and testing data-set by the applications, it is highly likely that one application has more chances to use one type of widgets while another is uses another type of widgets more. So we need to manually to adjust the distributions of widgets if the unbalance distribution happens between training data-set and testing data-set.

2.3 Deep-learning based object detection module

Object detection is a kind of task that needs to detect objects in images, which consists of two parts, detecting the boundary of objects and identifying them. Intuitively, this can be considered as the problem of classification when we apply windows of all possible sizes in the image and classify them. However, there are 2 problems in

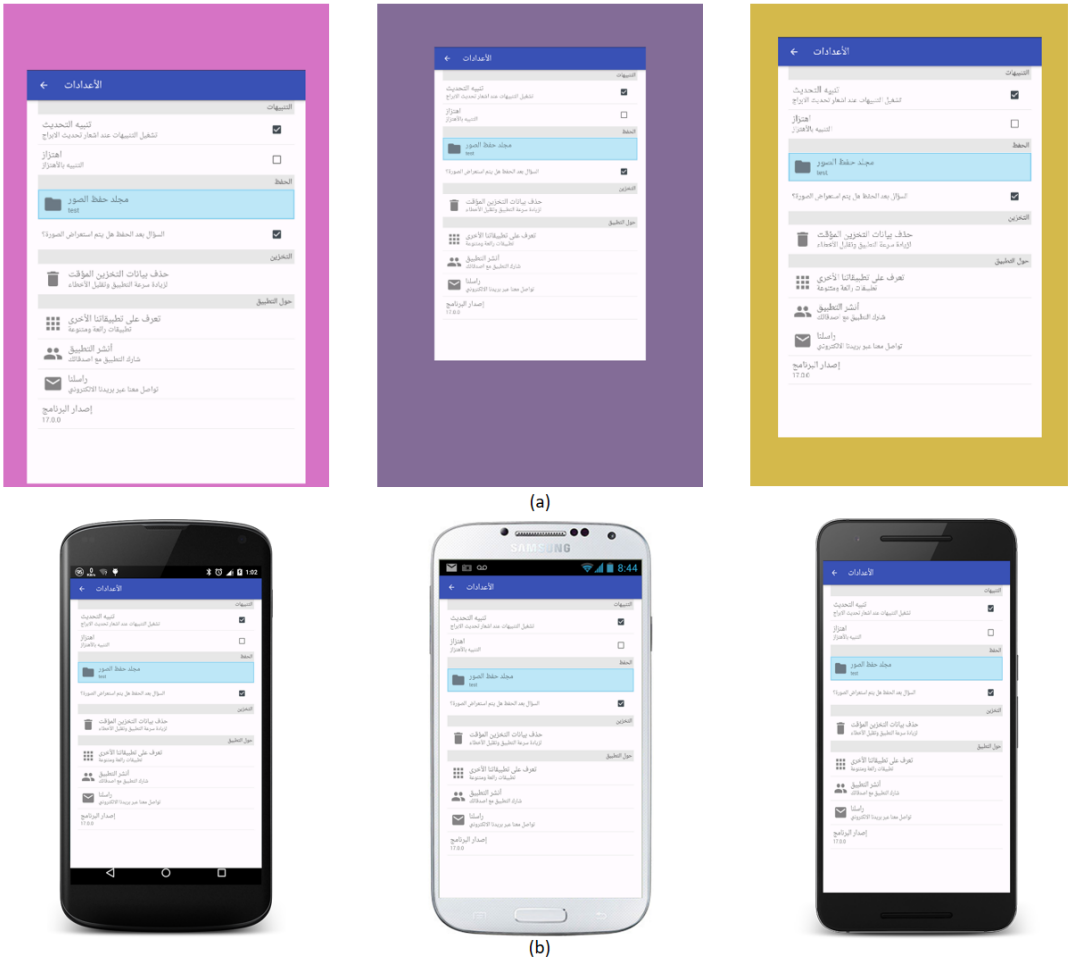


Figure 2.5: The examples of the two kinds of methods of image augmentation. The images in (a) are the image augmentation of the first method; The ones in (b) are the instances of the second approach

Table 2.2: The original distribution of the number of each type of widget in data-set A

	Number	Proportion
Button	171529	34.9%
ToggleButton	8028	1.63%
CheckBox	30647	6.23%
RadioButton	20831	4.24%
Switch	4648	0.95%
RatingBar	4006	0.81%
ImageButton	213281	43.39%
SeekBar	6692	1.36%
CompoundButton	778	0.16%
ProgressBar	20767	4.22%
Spinner	10268	2.09%
Chronometer	78	0.02%

it: we do not know the size of windows that have perfect boundary of objects and objects with different shapes have distinct ratios of the width and height of windows. These problems have been around for decades [Felzenszwalb et al., 2010] [Viola and Jones, 2001]. In recent years, with the development of deep learning network, the performance of the object detection has improved to a large extent [Girshick, 2015]. The deep-learning module we adopt is Faster-RCNN [Ren et al., 2015a] because of its high performance and fast speed, the implementation of Faster-RCNN we choose is a TensorFlow adaptation version [Chen and Gupta, 2017b]. For the initial convolutional layer, we choose ResNet-101 [He et al., 2016] instead of VGG-16 [Simonyan and Zisserman, 2014] because of its better accuracy and less computational complexity.

As shown in Figure 2.6, there are three major parts in the whole network, Convolutional Neural Network(CNN), Region Proposal Network(RPN) and objection detection network. Each of these parts are introduced into details below. The images are firstly fed through the CNN network to extract feature maps then they are processed by Region Proposal Network (RPN) to get proposals. Then, these proposals are applied to original feature maps, and go through Region of Interest (ROI) pooling to get fixed size proposals. At last, they are classified by two classification networks to detect objects. These steps are explained to details in the following sections.

2.3.1 Feature extraction using Convolutional Neural Network

Convolutional Neural Network (CNN) is a kind of feed-forward deep-learning neural network that has multiple layers of neurons. Most CNNs consists of convolutional layer, pooling layer, and full connection layer as shown in Figure 2.7. Typical usage of CNN is for classification of objects, while it can also be used in features extraction,

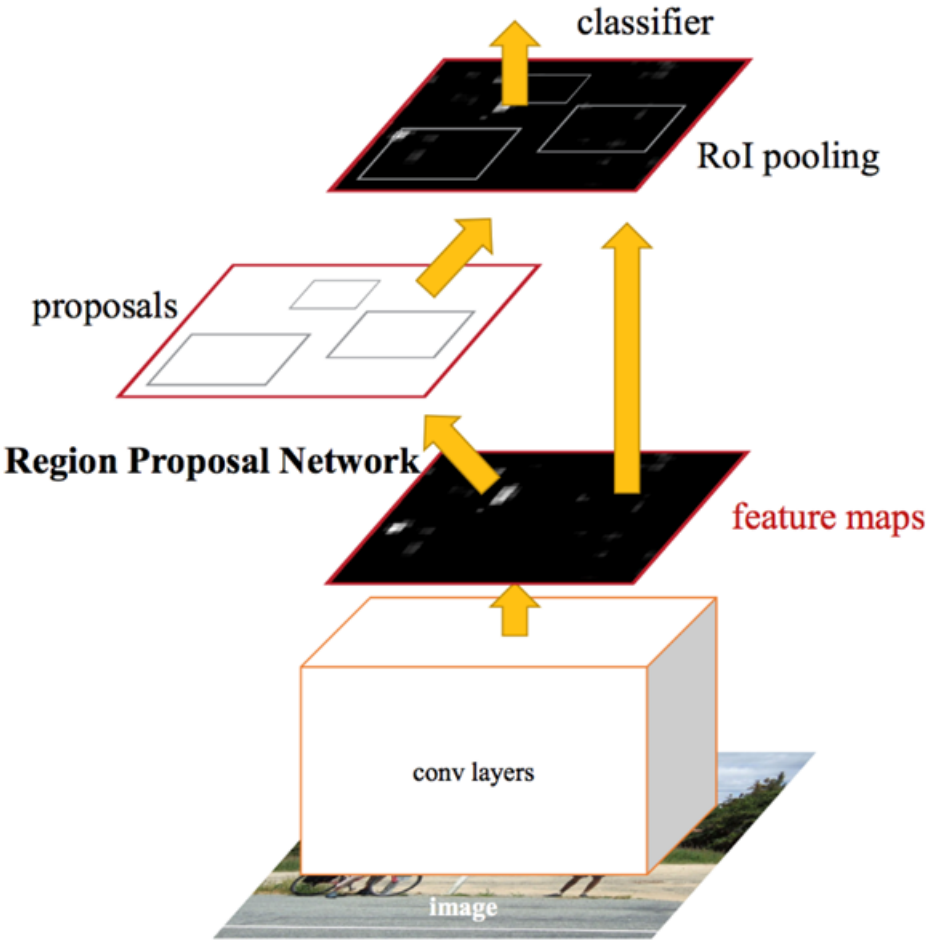


Figure 2.6: Faster R-CNN architecture; Reproduced from [Ren et al., 2015b]

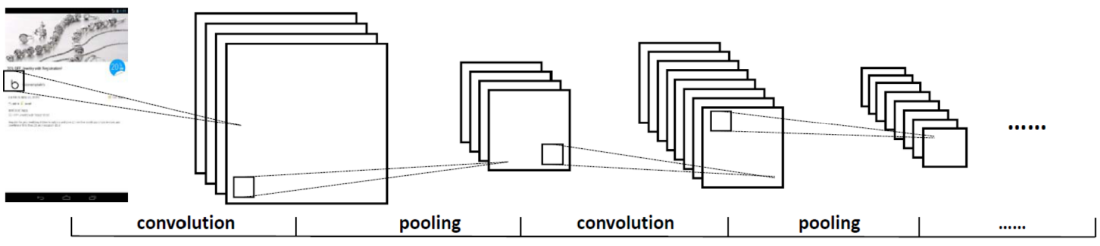


Figure 2.7: The layers of Convolutional Neural Network (CNN)

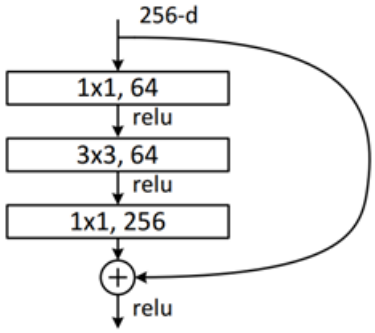


Figure 2.8: Three layers' skip connection; Reproduced from [He et al., 2016]

as it applied in the network of Faster-RCNN.

The convolutional layers are the ones that implements convolution operations to the pixels of images. The basic operator in convolution is on a small matrix so called kernel, the size of it in our network is 3×3. These kernels are convolved with input pixel data to produce feature different maps after they go through the whole image. The number of feature maps is determined by the number of filter kernels, each kernel learns to be activated when it sees a specific feature. The border of an image is padded to with 0 to keep the size. The output features maps are then put into the pooling layer, which works similarly to convolutional layer but has no overlap in it. Instead of convoluting, this layer finds the max value of pixels in that small window to represent that part of the image. After the sub-sampling in pooling layer, the number of weights to be calculated in network is reduced. In comparison, connection layer just connects all the neurons in last layer to next one which is used in the tasks of classification.

The CNN module we use in our task is Residual Neural Network (ResNet) [He et al., 2016], the major difference of its structure from others is it utilize skip connections to jump over some layers to avoid the problem of vanishing gradient [Hochreiter et al., 2001]. The ResNet we use is the one with 101 layers due to its high performance in classification [He et al., 2016] which has 3 layers deep ResNet block as shown in Figure 2.8.

2.3.2 Region Proposal Network

The RPN takes the output features maps from the last CNN as its input and generates rectangular object proposals with scores that measures the possibility of the anchor box containing objects. It tries 9 (K) anchor boxes with 3 kinds of scales (box areas of 128^2, 256^2, 512^2) and 3 kinds of aspect radios (1:1, 1:2, 2:1), which means for each location, there are 9 (K) different boxes centred around each location as shown in Figure 2.9.

After the anchors are generated, they need to go through the bounding-box regression. Each bounding box of the region is represented by an object foreground

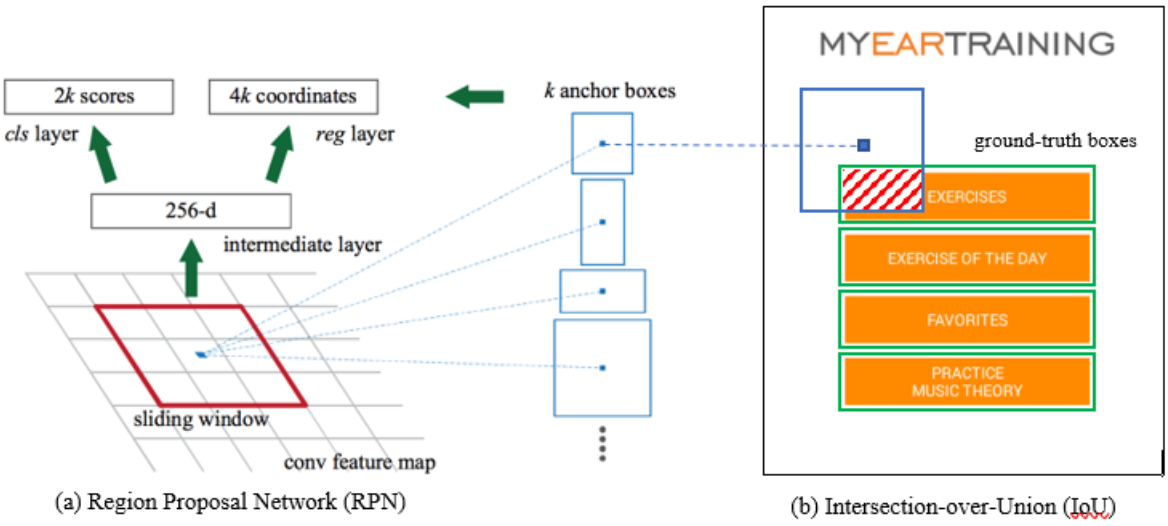


Figure 2.9: Region Proposal Network(RPN); Adopted from [Ren et al., 2015a]

Softmax [Nasrabadi, 2007] score and 4 coordinates. Subsequently, they need to be sorted by the score, and filter the top 6000 archers. After which, these archors are mapped to the original picture to find the ones that across the boundary. Then, do Non Maximum Suppression (NMS) and sort the left foreground archers again by foreground Softmax scores. At last, select the top 300 ones as the proposals. The possibility of an anchor box containing an object is measured by Intersection-over-Union (IoU) that overlaps with a ground-truth box. It is calculated by the intersection value of Archor and ground-truth divided by the union value of them, as shown below:

$$IoU = \frac{Area(Anchor \cap GroundTruth)}{Area(Anchor \cup GroundTruth)}$$

If the value of IoU has a highest value or is larger than a 0.7, the corresponding anchor is assigned a positive label.

The proposed regions generated by RPN are then related to the feature maps. The generated region is called Region of Interest (ROI). The ROI generates regions of various shapes and sizes, but the next classification network requires input feature maps with fixed size. Therefore, POI pooling is used for the transformation. The region proposals of different shapes and sizes are divided into M×N sub-areas, then, apply max pooling to them. For instance, for a ROI with size X×Y, each max pooling needs to use the largest value in the subarea of size $\frac{X}{M} \times \frac{Y}{N}$ to denote this area. Therefore, after the ROI pooling the ROIs are of the same size.

2.3.3 Object Classification

The final part of Faster-RCNN is object classification network, which consists of two fully connected layers. The first one use Softmax to classify the objects and

generate the possibility value of the correctness of it classification, which is the score of recognition. The second one uses the bounding box regression again to get more accurate regional proposals, which is the region of objects that are detected.

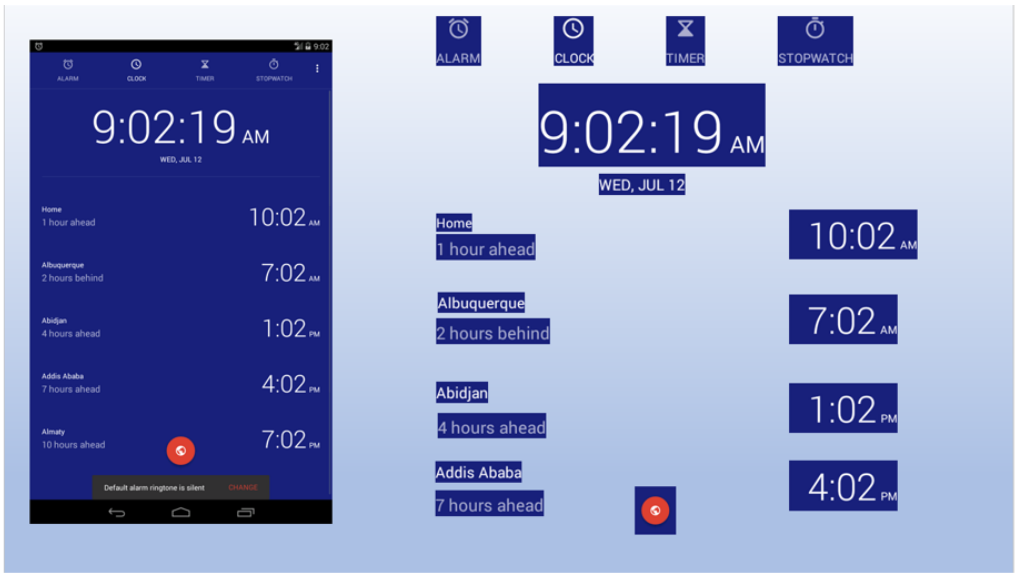
2.4 UI gallery database generating and applying

2.4.1 UI components recognizing

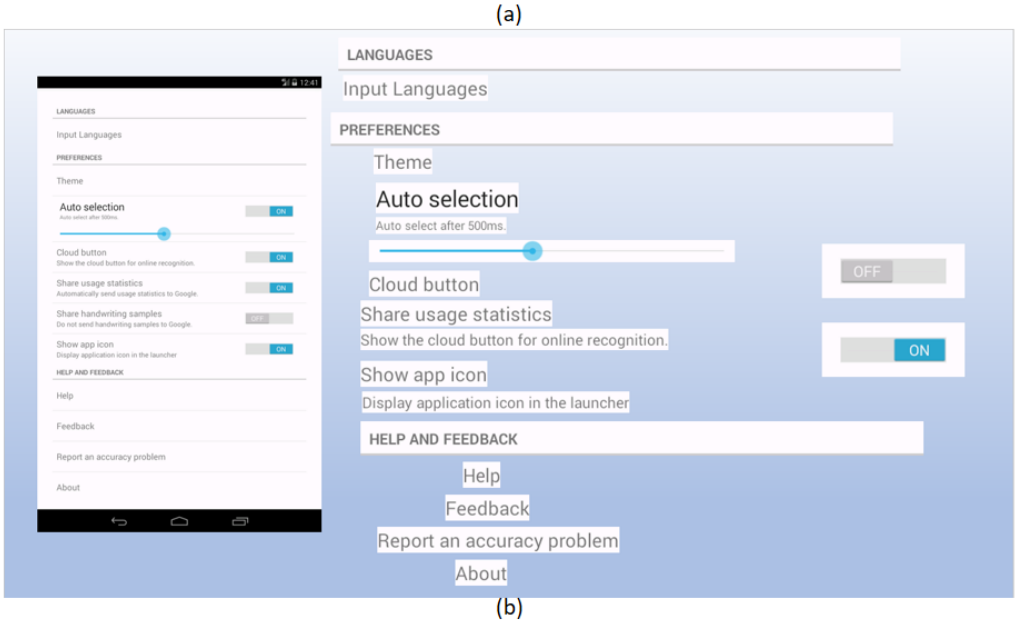
After the training of Faster RCNN, we apply the trained module to the large-scale data-set to detect the widgets from the introduction screen-shots scrawled from Google Play app store. To that end, we developed a tool to get the detection result of trained Faster RCNN module. It generates files in XML format with the detected information that corresponds to the detected screen-shots from dataset B, such as the class, boundary, score of the detected widgets. then, another clipping tool we developed is used to clip the widgets out of the screen-shots. At the same time, the information of these extracted widgets is generated, such as colour and size. Figure 2.10 is the example of extracting widgets from screen-shots.

2.4.2 UI gallery

These large-scale clipped widgets are served as the database of the UI gallery. Figure 2.11 illustrates the basic usages of the UI gallery we developed. Figure 2.11(a) is the home page that includes the instances of widgets of each class. There is also a button for customized searching in the home page. In the searching page where users are allowed to search their wanted widgets by colours, size and so on. Figure 2.11(b) shows the example of searching red buttons.



(a)



(b)

Figure 2.10: The examples of clipping widgets out from screen-shots

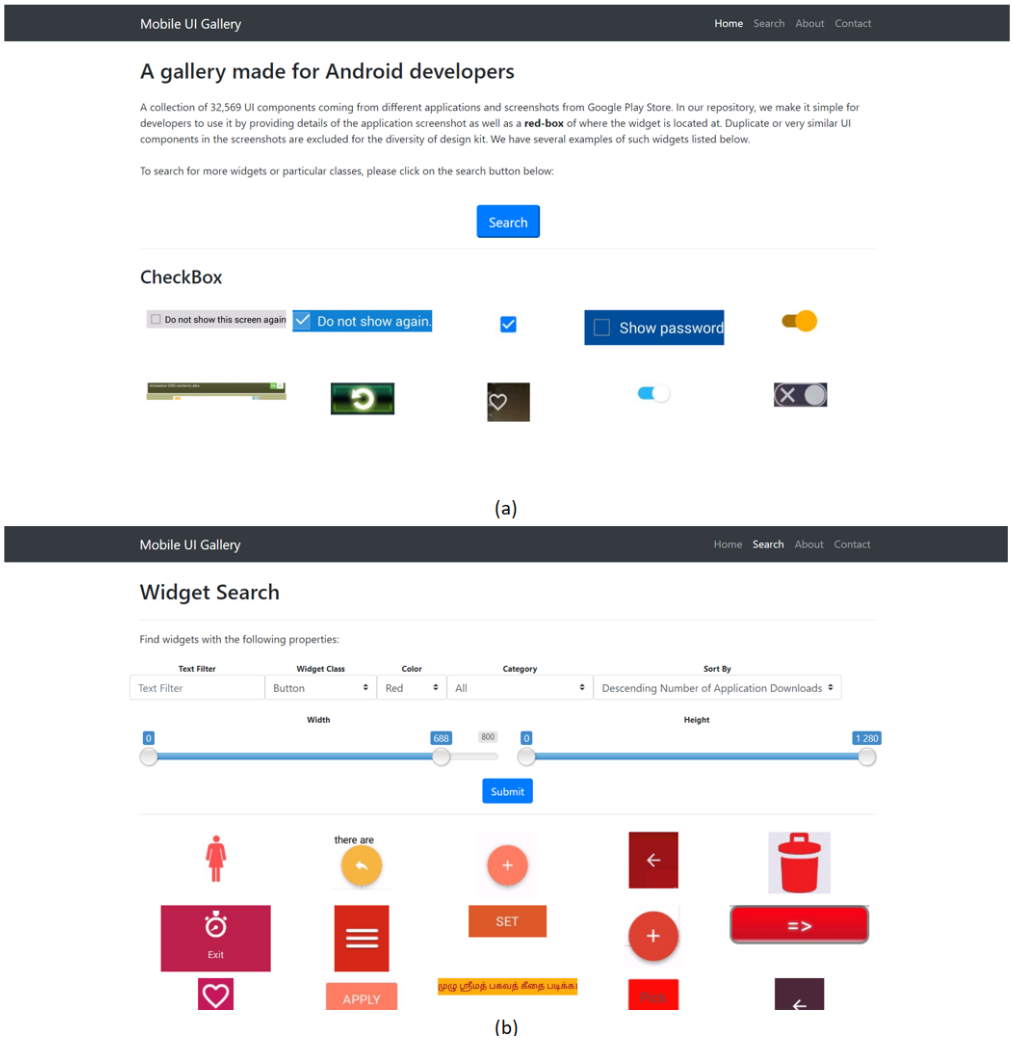


Figure 2.11: the website of UI gallery

Chapter 3

Experiments

3.1 The processing of data-set A

As mentioned in section 2.2, we apply the duplication elimination on the data-set A first in two thresholds. One is for the rare 3 widgets that are less than 1% the data-sets (RatingBar, CompoundButton and Chronometer). For these widgets, we choose the similarity threshold of XML being 94% in the first round of duplicity removal, which means the similarity of XML between two list need to be higher than 94% to be considered as duplicated pictures. Then, the threshold of image histogram for rare widgets is set to 98%. For other 9 widgets (Button, ImageButton, ProgressBar, SeekBar, CheckBox, RadioButton, Switch, ToggleButton and Spinner), we need to eliminate the similarity to a large extent. So, we apply the similarity threshold of XML to be 80% and the similarity threshold of histogram to be 95% to remove duplication. After removing duplication, the median number of screenshots of each app is 7, while the average number is 18.3. Then we conduct image augmentation on all the left images. The table table Table 3.1 shows the distribution after all the processing on data-set A. Compared to the original data-set, the processed data-set is more balanced, the proportion of Choronometer rises from 0.02% to 0.64% and that of CompoundButton from 0.16% to 1.83%. Due to the removal of duplication, the percentage of ImageButton and Button dip from 43.39% and 34.9% to 28.47% and 32.81% respectively. Therefore, the widgets are more balanced. Note that even though we can make it further balanced, the distribution of widgets in the training data-set should be consistent with that of the normal data-set.

3.2 The training and evaluating of object-detection module

3.2.1 Experimental environment setup

For the training of the Faster-RCNN module, we use the learning rate that starts with 0.001, the batch size is 256, and the module is trained for 140,000 iterations. The server we are running the module on has 12 CPUs (Intel(R) Core(TM) i7-7800X CPU @ 3.50GHz, 3484 MHz), a graphics card (VGA compatible controller: NVIDIA Corporation Device 1b06), and a memory of 64 Gigabytes. The Python version is

Table 3.1: The distribution of widgets of data-set A after duplication elimination

	Number	Proportion
Button	153820	32.81%
ToggleButton	17422	3.72%
CheckBox	57034	12.17%
RadioButton	17740	3.78%
Switch	6990	1.49%
RatingBar	2922	0.62%
ImageButton	133456	28.47%
SeekBar	9242	1.97%
CompoundButton	8576	1.83%
ProgressBar	12386	2.64%
Spinner	46224	9.86%
Chronometer	3012	0.64%

3.5.2 and the TensorFlow version is 1.10.1. To avoid the existence of data in both training set and testing set, we split the data-set by apps instead of by pictures, and the proportion is 14 to 1. That is to say, when we process the data-set A, for every 15 applications, 14 of them are put in training set and 1 of them are in testing set.

3.2.2 Evaluation metrics

There are three metrics used in our module for evaluating the performance, which are recall, precision and mean Average Precision (mAP). For each type of widgets, we calculate these three metrics. The recall of a certain class of widget is the fraction of this type widgets that have been predicted over the total amount of relevant widgets (true positives divided by the sum of true positives and false positives), it measures how accurate the prediction of class is. The precision of one type of widgets is the fraction of relevant class among the retrieved widgets of this class (true positives divided by the sum of true positives and false negatives), which measures the capability of finding all the positives. Average Precision (AP) for specific class of widget is calculated from the precision-recall curve, by varying the model score threshold that is considered as a positive detection of this class. The mAP is the average AP over all classes of widgets. The overall recall of our trained module is 0.6551, and the precision is 0.7441, while the mAP of the overall widgets is 0.6572. The results of individual classes of widgets get from our module are shown in table Table 3.2.

3.2.3 Detection capability analyzing

Apart from evaluation matrix, we manually analysed the detection capability of our module by randomly sampling 1200 pictures from the testing dataset of augmented

Table 3.2: The training result of each widget in terms of AP, Recall and Precision

	AP	Recall	Precision
Overall	0.6572	0.6551	0.7441
Button	0.6031	0.6589	0.6964
ImageButton	0.5890	0.6245	0.7758
CompoundButton	0.8621	0.8622	0.9899
ProgressBar	0.3036	0.3672	0.7055
SeekBar	0.7323	0.7717	0.8582
Chronometer	0.8637	0.8649	0.9796
CheckBox	0.6783	0.7306	0.7072
RadioButton	0.3985	0.4700	0.6458
Switch	0.7698	0.8476	0.7072
ToggleButton	0.7097	0.7199	0.9593
RatingBar	0.5869	0.7013	0.6279
Spinner	0.7901	0.8234	0.7689

Dataset A and Dataset B. We noticed that our module not only can recognise ordinary widgets successfully but is also able to extract those challenging widgets with complicated UI features.

Figure 3.1 illustrates the detection instances of Faster-RCNN. Figure 3.1(a) shows its capability of successfully detecting densely distributed widgets. Even though not all the widgets in that augmented screen-shot, it has detected most explicit ones of them. The point here is that the threshold of detection we set is 0.65, the higher threshold means only those widgets with higher confident scores are detected. An interesting observation is in Figure 3.1(b), although the and the red button and spinner in the background are occluded by a menu and in dark colour, our module still can estimate their boundaries correctly. However, this kind of detection is useless for our purpose of extracting data-set for GUI gallery, as they are not integrated widgets and their colours are not their original ones. In figure 3.1(c), Moreover, Faster-RCNN managed to detect those CompoundButton on the left top corner of images in that screen. Note the ones in first two images are implicit and almost merged into the background, which is hard to distinguish by eyes, showing the strong detection capability of our module on that scenario. The figure 3.1(d) illustrates its ability of extracting widgets which merge into complex backgrounds, the background of these widgets is a photograph.

In spite of the generating satisfactory results, the Faster RCNN module is still not perfect. There are occasions the module produces defective detection. For example, in Figure 3.2(a) a white circle is considered as a ProgressBar due to its similar shape and colour; An occluded off-state Switch also is detected as a ProgressBass in Figure 3.2(b) despite the correct boundary of Switch. In Figure 3.2(c), a SeekBar

Figure 3.1: The example of successful detection

like image successfully deceives the module and make it be a SeekBar. Figure 3.2(d) shows the failure of the module to distinguish the UML objects and Buttons. All the defective recognition above can be attributed to the similarity between widgets and other objects; however, there are still much enhancement can be made to increase the accuracy of detection.

Figure 3.2: The example of defective detections

Chapter 4

Conclusion and Future Works

4.1 Conclusion

In this project, we implemented several applications to develop a large-scale GUI resources (widgets in Android system) for the purpose of assisting mobile GUI designers, so that they can explore for novel design inspirations to integrate into their own work. Subsequently, these widgets were clipped from the introduction screen-shots of some widely used Android apps to present designers most trendy design of the components of GUI.

More specifically, there are for major steps to achieve our goal. The first step is data-set collecting. In this step, there are two data-sets. One consists of pairs of screen-shots and their corresponding meta-data (data-set A), which are gathered from many widely used Android applications on Google Play app store using an Android GUI testing tool, Stoat, this data-set is used for the training of detection module (Faster RCNN). Another data-set is a large-scale data-set with screen-shots with no annotations in it (data-set B) that are collected by scrawling from the introduction information of applications on Google Play, this data-set is for applying trained detection module and widgets clipping. After the first step, we applied some processes to the data-set A for the purpose of module training. These processes consist of constructing the training data-set, image augmentation, balancing widgets, removing the duplication. The purpose of them is to make the data-set A suitable for the training of Faster RCNN and maximize the effectiveness and accuracy of its detection. Then, the third step is training the module of Faster RCNN. In this step, we adopted a realization of Faster RCNN module with TensorFlow and modified the classes to the ones of Android widgets. Then, we train it using the data-set A, and evaluate the performance of our module with both the data-set A (as the way of testing the module) and the data-set B (manually evaluate the randomly picked subset of it). Our evaluation confirms the effectiveness of data processing methods. After the training of Faster RCNN, it is the fourth step. We applied the trained module to the data-set B to detect widgets and clip the detected widget as the large-scale database of a UI gallery that for the use of UI designers.

4.2 Future Work

For further improvement of the performance of our module, there are a few things can be done. Firstly, although the distribution of the classes of widgets do reflect the real one of widgets when developers and designers develop the application, it still is problematic to train a module with such an imbalance data. As revealed in the work of Masko [Masko and Hensman, 2015], the high imbalance of the classes in the training data-set can result in seriously negative impact on CNN performance. For example, one problem might be that only the classes of majority are classified repeatedly. Synthetic minority over-sampling technique [Han et al., 2005] and Re-sampling methods [Estabrooks et al., 2004] are promising methods for overcoming the adverse effect of an imbalanced data-set. Secondly, there are some works [Chen et al., 2018b] that are used to enhance the detection accuracy for small objects using the filter-amplifier network and loss boosting in the object detection network. These kinds of work could be beneficial to our work provided the existence of the large number of small widgets in the screen-shots. Thirdly, there are some inconsistency in the data-set A and data-set B, which can affect the performance of the detection widgets. For example, there are considerable setting interface in the data-set A, as shown in 2.3(a), while in data-set B there are few. On the other hand, in data-set B there are many cases the introduction screen-shots are complex and combined by several screen-shots with other additional pictures and words around them as shown in 2.3(b), while in data-set A most of the screen-shots are taken in applications. At last, due to the nature of Android widgets, there is no strict boundary between some types of widgets. For example, Button and ImageButton can be replaced with each other to a large extend, so weather using Button or ImageButton sometimes depends on the habits of developers. Therefore, in our future work, multi-label image classification [Boutell et al., 2004] can be a promising development.

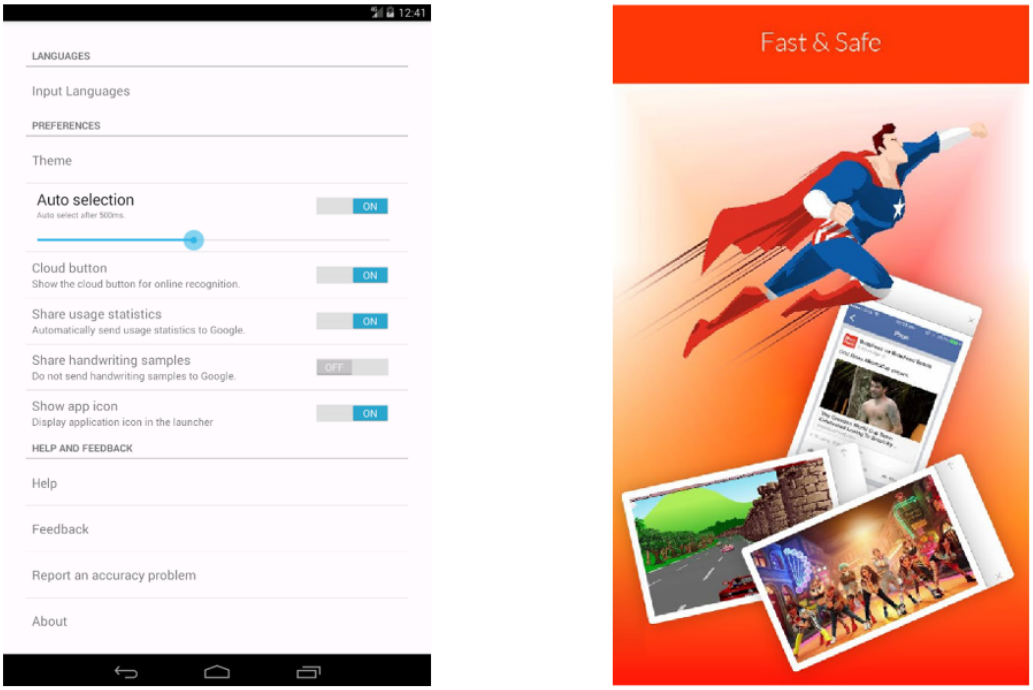


Figure 4.1: The difference of two data-sets

Bibliography

(cited on page 2)

mui-collection. http://mui-collection.herokuapp.com. Accessed 2018. (cited on page v)

Boutell, M. R.; Luo, J.; Shen, X.; and Brown, C. M., 2004. Learning multi-label scene classification. *Pattern recognition*, 37, 9 (2004), 1757–1771. (cited on page 28)

Chen, C.; Su, T.; Meng, G.; Xing, Z.; and Liu, Y., 2018a. From ui design image to gui skeleton: A neural machine translator to bootstrap mobile gui implementation. *The 40th International Conference on Software Engineering (ICSE)*, (2018). (cited on page 2)

Chen, X. and Gupta, A., 2017a. An implementation of faster RCNN with study for region sampling. *CoRR*, abs/1702.02138 (2017). http://arxiv.org/abs/1702.02138. (cited on page 7)

Chen, X. and Gupta, A., 2017b. An implementation of faster rcnn with study for region sampling. *arXiv preprint arXiv:1702.02138*, (2017). (cited on page 14)

Chen, Z.; Crandall, D.; and Templeman, R., 2018b. Detecting small, densely distributed objects with filter-amplifier networks and loss boosting. *arXiv preprint arXiv:1802.07845*, (2018). (cited on page 28)

Estabrooks, A.; Jo, T.; and Japkowicz, N., 2004. A multiple resampling method for learning from imbalanced data sets. *Computational intelligence*, 20, 1 (2004), 18–36. (cited on page 28)

Everingham, M.; Van Gool, L.; Williams, C. K.; Winn, J.; and Zisserman, A., 2010. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88, 2 (2010), 303–338. (cited on page 7)

Felzenszwalb, P. F.; Girshick, R. B.; McAllester, D.; and Ramanan, D., 2010. Object detection with discriminatively trained part-based models. *IEEE transactions on pattern analysis and machine intelligence*, 32, 9 (2010), 1627–1645. (cited on page 14)

Girshick, R., 2015. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, 1440–1448. (cited on page 14)

Han, H.; Wang, W.-Y.; and Mao, B.-H., 2005. Borderline-smote: a new over-sampling method in imbalanced data sets learning. In *International Conference on Intelligent Computing*, 878–887. Springer. (cited on page 28)

He, K.; Zhang, X.; Ren, S.; and Sun, J., 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778. (cited on pages ix, 14, and 16)

Hochreiter, S.; Bengio, Y.; Frasconi, P.; Schmidhuber, J.; et al., 2001. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. (cited on page 16)

Jo, I. and Jung, I. Y., 2016. Smart learning of logo detection for mobile phone applications. *Multimedia Tools and Applications; Dordrecht*, , 13211-13233 (2016). (cited on page 2)

Labs, B. Pttrns. https://pttrns.com/. Accessed 2018. (cited on page 1)

Masko, D. and Hensman, P., 2015. The impact of imbalanced training data for convolutional neural networks. (cited on page 28)

Meier, S.; Heidmann, F.; and Thom, A., 2014. A comparison of location search ui patterns on mobile devices. *MobileHCI '14*, , 465-470 (2014). (cited on page 2)

Moran, K. and Bernal-C'ardenas, C., 2018. Machine learning-based prototyping of. *SOFTWARE ENGINEERING JOURNAL,*, (2018). (cited on page 2)

Nasrabadi, N. M., 2007. Pattern recognition and machine learning. *Journal of electronic imaging*, 16, 4 (2007), 049901. (cited on page 17)

Nguyen, T. A. and Csallner, C., 2015. Reverse engineering mobile application. *International Conference on Automated Software Engineering*, (2015). (cited on page 2)

Oliver, M. Axure widgets. https://axurewidgets.com/download-axure-widgets/. Accessed 2018. (cited on page 2)

Pakhandrin, S. Inspired-ui. http://inspired-ui.com/. Accessed 2018. (cited on page 1)

Pinterest. Mobile app design. https://www.pinterest.com.au/timoa/mobile-ui-photos/. Accessed 2018. (cited on page 1)

Ren, S.; He, K.; Girshick, R.; and Sun, J., 2015a. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, 91–99. (cited on pages ix, 6, 14, and 17)

Ren, S.; He, K.; Girshick, R. B.; and Sun, J., 2015b. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497 (2015). http://arxiv.org/abs/1506.01497. (cited on pages v, ix, and 15)

Simonyan, K. and Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, (2014). (cited on page 14)

Su, T.; Meng, G.; Chen, Y.; Wu, K.; Yang, W.; Yao, Y.; Pu, G.; Liu, Y.; and Su, Z., 2017. Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017 (Paderborn, Germany, 2017), 245–256. ACM, New York, NY, USA. doi:10.1145/3106237.3106298. http://doi.acm.org/10.1145/3106237.3106298. (cited on pages v and 6)

Viola, P. and Jones, M., 2001. Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, vol. 1, I–I. IEEE. (cited on page 14)

Wimmer, G.; Uhl, A.; and Vecsei, A., 2017. Evaluation of domain specific data augmentation techniques for the classification of celiac disease using endoscopic imagery. In *Multimedia Signal Processing (MMSP), 2017 IEEE 19th International Workshop on*, 1–6. IEEE. (cited on page 11)