Code Runner: Solution for Recognition and Execution of Handwritten Code

Wenxiao Du
Department of Electrical Engineering, Stanford University
Stanford, CA
wxdu@stanford.edu

Abstract—This report introduces “Code Runner”, an Android application that can recognize and execute handwritten code by users. Current prevalent OCR-engines can recognize printed text with high accuracy, but can hardly handle handwritten text very well. The difficulty of hand-written text recognition is due to variation of characters and poor alignment of text line. Therefore, to achieve a workable solution for handwritten code recognition system, I first customize and train one popular OCR-engine, Tesseract, to make it able to deal with my own handwriting. Besides, several image processing methods and text-level post-processing algorithms have been adopted to enhance the system accuracy. Test results show that the system can have a reasonable accuracy on both characters set and realistic code text. This report will go through algorithms as well as implementation details of the entire project.

Keywords—handwriting recognition; Tesseract; Android; Image processing

I. INTRODUCTION

In technical interviews, interviewees are sometimes asked to write code on white board or paper. Interviewer can see the general logic of the code but cannot actually execute it and prove its correctness. On the other hand, type in code on computer may diminish the interaction between interviewers and interviewees. Code Runner is an Android application that provides a solution for such situation. Basically, it allows user take a photo of code by android phone. The app will then process the photo and display execution output of the code.

I choose C as the target language for it’s simple syntax. C’s vocabulary is quite small compared to other modern programming languages. On the other hand, C’s grammar is relatively strict. Missing declaration, missing semicolon or wrong variable type, etc., will all lead to compilation error. Therefore, it’s a good touchstone to test the accuracy of the recognition. Another concern is that indentation doesn’t matter in C, so the app can focus on the recognition of text only.

Many related works have been done on the similar topic. Iris is a system that can recognize and execute code written in ruby[1]. Team in Jadavpur University also has several papers about the hand-written text recognition using Tesseract-OCR Engine[2][3]. Besides, a previous EE368 student also developed a laptop-based application that can do code recognition. [4] Some suggestions and tricks are referred from these papers during the implementation of Code Runner.

In the following part of this report, I will first walk through the structure and pipeline of the entire project as an overview. Then details of each part, including preprocessing, Tesseract training, post-processing, will be discussed. After that, the accuracy of the recognition process will be evaluated on test set and result will be reported.

II. SYSTEM OVERVIEW

Since Tesseract OCR would require high computation complexity, running the entire project on mobile phone is infeasible. Therefore, the project is implemented based on client-server communication model. The client side, Android device, is in charge of capturing image and displaying result to user, while the server side is mainly responsible for most processing jobs.

The major stages in the pipeline includes:

1. Capture and upload Image: Using Android device to capture the image and upload it to the server.
2. Preprocessing: In this step, several image processing methods are applied to the uploaded image to make it ready for recognition. These methods include: Binarization, Small region removal and Morphological Opening. It would be discussed in section III.
3. Text recognition: In this step, we run the Tesseract-OCR engine to extract text from the preprocessed image. Tesseract cannot recognize handwritten text originally, and some training process is required to make it capable of doing so. Details would be discussed in section IV.
4. Post-processing: In this step, some heuristic algorithms are applied to the retrieved text. The general idea is to make the text looks more like a piece of C code. Details would be discussed in section V.
5. Manual adjustment: Even with all the previous steps, the system still cannot guarantee 100% correctness of recognition, and even 1 error would lead to compilation failure. Therefore, server will send the final text back to client and let user do the final check. This step should only involve slight modification.
6. Compile & execute & display: After manually modified by user, ideally everything should be correct now. The client then sends text back to the server. The server compiles and executes the code and passes the result to the client for display.

A flow chart for the pipeline is shown in Figure 1.
III. PREPROCESSING

Preprocessing is an essential step to obtain accurate text recognition results with the Tesseract OCR tool, because a code image captured by a hand held camera is far from ideal input for an OCR engine. 1) The illumination of the image is not uniform everywhere. 2) Small region caused by dust and noise might exist. 3) The boundary of text might be blurred due to low resolution. Our preprocessing workflow is designed to tackle these issues. And they are described in the following subsections.

A. Binarization

If input image is a RGB image, it will first be converted into a grayscale image. Locally Adaptive binarization is then applied to the image. I use 20x20 tiles as a window, and each tile is binarized separately using Otsu’s method. The neighboring tiles share overlapping area with width of 10 pixels to smooth the result.

B. Small region removal

Dust on camera or paper sometimes introduce noise on the image. Therefore, region labeling is conducted on the image and all regions with less than 12 pixels are removed. The threshold is selected so that small regions like punctuation won’t be falsely removed.

C. Morphological opening

The boundaries of characters sometimes are misclassified in the binarization step. I conduct morphological opening on the image to smooth the boundary and make characters looks more realistic.

An example of image before and after the preprocessing is shown in Figure 2.

IV. TESSERACT ENGINE[5]

Tesseract is an open source OCR engine that was developed at HP Labs between 1985 and 1995, and later improved extensively by Google. It was one of the top 3 engines in the 1995 UNLV Accuracy test. Tesseract is originally designed to recognize printed English text only. However, Efforts have been made to modify the engine and its training system to make them able to deal with other languages [6]. Therefore, if we define hand-written English as a new “language”, we can train Tesseract to recognize it.

A. Handwriting preparation

Rather than using any dataset from Internet, I produced training set manually by myself. I wrote 10 pages of codes with 2184 characters included. The distribution of characters is not strictly even, but I guarantee that for each character we have at least 20 training examples, which fulfills the requirement of Tesseract.

The reason why I didn’t use pages that contain all the characters separately is because Tesseract conducts layout analysis and word segmentation internally. It also recognizes words using language knowledge. Therefore, using realistic text (i.e. code here) should have better training effect.

After that, I scanned all the pages and conducted preprocessing exactly the same as described in section 3. A training example is shown in Figure 3.

B. Manual labeling

Tesseract requires a box file associates with each training image. Basically each box file contains all coordinates and corresponding characters in the image, one per each row. Box file is not generated automatically but need produced by manual labeling. I use an online tool [7] to do the label job. A screenshot of labeling process is shown in Figure 4.
After this step, we can follow the guideline in the document to generate training model.

C. Some tricks

To enhance the system performance, several tricks have been applied here:

1. Feed in ‘freq-dawg’ file while training. This is an optional but helpful file for the training process. The ‘freq-dawg’ contains frequent words in the target language. Since the vocabulary for C is quite small, it’s pretty easy to add such a file which would lead to a better accuracy of recognition.

2. Avoid ambiguity of characters. Ambiguity between characters is one of factors that cause misclassification. Therefore in this project, I modify the font of some character to avoid such ambiguity. You can see some of them from Figure 2. Quotations marks are linked together to differentiate from apostrophe. The point on the top of character ‘i’ and semicolon is made connect to the below part so that it won’t be misclassified as dot. Experiments show that such self-defined fonts help the performance.

V. POST-PROCESSING

The text returned by Tesseract contains mistakes in most case. Before pass it to human adjustment, I apply some heuristic text-level processing on it to avoid as much manual modification as possible.

A frequent word list is first generated just like what we did in IV-C. Then I tokenize the input text using all punctuation as delimiter. For each token, I calculate its edit distance between all the candidates in the frequent word list. If the distance is below some threshold, the token will be replaced by the candidate. The threshold is piecewise so that for the words with shorter length, the threshold is lower.

Such method does have drawbacks. For example, since ‘int’ is in frequent word list, if user has a variable named ‘ant’, it will be replaced. Overall the post-processing helps, and more sophisticated method can be tried in the future.

VI. RESULT AND EVALUATION

A. Code Runner screen shot

Figure 5 shows a screen shot of the application. This is the last step of the pipeline, in which user can manually adjust text sent from server.

B. Accuracy on test sets

In this step, we use 3 test images to evaluate Code Runner’s accuracy. 3 test images have different characteristics. Image 1 is a normal image, image 2 involves some perspective distortion, and image 3 is a rotation of image 1. 3 images are shown in Figure 6.
Fig. 6. Test images 1, 2, 3

All 3 images contain 216 characters each. The accuracy of recognition of each image is shown in Table 1:

<table>
<thead>
<tr>
<th>No.</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image 1</td>
<td>83.79%</td>
</tr>
<tr>
<td>Image 2</td>
<td>77.31%</td>
</tr>
<tr>
<td>Image 3</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 1. Accuracy on different test images

We can see that Code Runner has high accuracy on normal images with no perspective distortion and rotation. Perspective distortion doesn’t hurt performance a lot. For image 3, we can see Code Runner cannot handle rotation at all. It does give out 13 characters in the output text, but I can hardly tell which characters in image these outputs corresponds to, so I count it as 0% here.

C. Accuracy on test sets

I also did some experiments on a separate characters set. For each character, 10 examples are produced, so totally 560 characters are tested. The overall accuracy is 70.58%. Figure 7 shows the confusion matrix of the classification.

We can see that the accuracy drops sharply compared to experiments in Part B. This is because 1) With separate characters, the knowledge about frequent word list cannot help 2) Internally, Tesseract’s layout analysis and word segmentation cannot handle separate characters image very well.

Fig. 7. Confusion Matrix of character set experiment

VII. CONCLUSION AND FUTURE WORK

In general, the Code Runner is functional and can recognize most code. However, there are still several drawbacks that can be improved in the future.

First thing is its poor robustness against rotation. I’ve tried Hough transform to adjust skewness, but it doesn’t produce ideal result. I think one way to fix it is to detect and draw text line on image before apply Hough Transform. One possible way is first detect centroid of each character and use a straight line to fit them.

Another issue is that currently Code Runner cannot take in any input during running time, so its application is very limited. More interactive activity should be added to the app to deal with this case.

Meanwhile, the post-processing part can also be improved. For example, since a variable cannot be used before it’s declared in C, if we can incorporate knowledge like this into recognition, the accuracy would be further boosted.

The current recognition is limited to my own handwriting due to lack of training examples. If we want to generalize it to recognize handwriting from arbitrary people, more training example is needed.

The last issue is the generalization to other programming language. One possible challenge here is the recognition of indentation, because some language like Python relies on indentation for interpretation. The Tesseract alone cannot recognize indentation accurately, so some other method should be introduced to handle it.

REFERENCES


