Evaluating Mathematical Expressions with the Eyequation Android App

Nico Chaves and Noam Weinberger
Department of Electrical Engineering
Stanford University

I. INTRODUCTION

Optical character recognition (OCR) is an important technique for solving many technical problems. As a few examples, OCR has been used for digitizing books, identifying license plates, and assisting the vision impaired. In this report, we present an Android mobile application called Eyequation that uses OCR to recognize and evaluate arithmetic expressions. The app could be used to help students check their solutions to challenging problems quickly and easily.

While developing Eyequation, we found a related app called Photomath. The Photomath app requires the user to manually place a bounding box around the text of the equation, and we found the execution time to be somewhat slow (often a few seconds for a simple expression). Furthermore, Photomath can only solve a single equation at a time. Eyequation addresses each of these shortcomings. Currently, Eyequation can automatically detect multiple equations on a light background, display bounding boxes around each equation, and solve the equations independently. However, in our experience, Eyequation does not yet recognize text, particularly handwritten text, as accurately as Photomath. Furthermore, Eyequation is currently limited to arithmetic expressions involving integers 0-9 and the following characters: “+-()/x”. We plan to address these issues in our future work.

II. IMPLEMENTATION

The application’s pipeline can be broken up into six segments: keypoint detection, equation bounding, thresholding, character recognition, evaluation, and display.

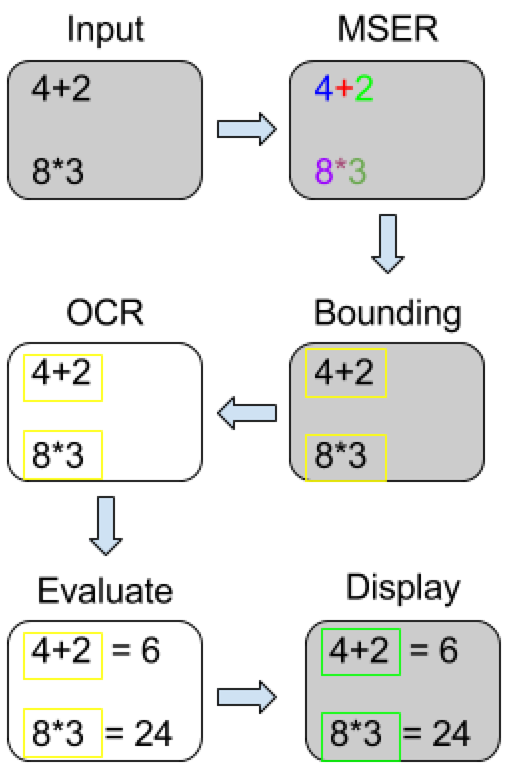

Figure 1: Pipeline Diagram

A. Keypoint Detection

The first step of the application is to find text in the image. We assume that the user has taken an image of dark text on a primarily light background, which may contain shadows. We first convert the image to grayscale and detect maximally stable extremal regions (MSER). MSER detection searches for regions of an image that remain relatively stable across many different thresholds. In other words, it finds regions that are darker than a certain value. As that value is increased, those regions will grow. Regions that do not grow while the value is significantly increased are those with much darker (or lighter) values than the region around them. MSERs are useful features for text detection because of the strong contrast between text and background and because of text's connectedness[1]. For this stage, we use the implementation of MSER keypoint detection available in the OpenCV[2] Android library.

In our future work, we plan to remove noise and shadow before applying MSER. This would reduce the number of spurious keypoints found by MSER, resulting in better bounding boxes around equations in the next step of the pipeline.

B. Equation Bounding

We use the keypoints detected by MSER to find image contours using OpenCV's implementation of Suzuki's Border Following Algorithm[3], and we construct bounding boxes around them. This provides us with a rectangle around the text. Depending on the scale of the image, each rectangle might bound an entire expression or, in the case of large font with large gaps in between, it might bound each character by itself. To deal with the latter case, we merge rectangles that are close together or overlapping. In particular, we merge a pair of rectangles if they are within a fixed vertical and horizontal offset (which we determined through trial-and-error experimentation) of each other. Note that we assume that each expression is written on a single line.

C. Thresholding

Though the equations have been isolated into bounding boxes, the boxes may contain shadows, noise, and other artifacts that could interfere with character recognition. To mitigate these artifacts, we perform locally adaptive thresholding over the entire image (see Figure 2 for an example). Locally adaptive thresholding is particularly useful for this application, because parts of the captured image are often under different amounts of shadow (for example, when the phone casts a shadow over part of the image). For this stage, we use OpenCV's implementation of locally adaptive thresholding. Alternatively, one may choose to apply thresholding to the interior of each bounding box individually. However, one still may need to apply locally adaptive thresholding to each bounding box, as some boxes may have non-uniform illumination. We chose our approach instead because it is

[1] Chen, Huizhong, et al. "Robust Text Detection in Natural Images with Edge-Enhanced Maximally Stable Extremal Regions." *Image Processing (ICIP), 18th IEEE International Conference*, 2011

[2] http://opencv.org

[3] Suzuki, S. and Abe, K., "Topological Structural Analysis of Digitized Binary Images by Border Following." *CVGIP 30 1*, 1985

simpler and it generates satisfactory binarized images of each expression in practice.

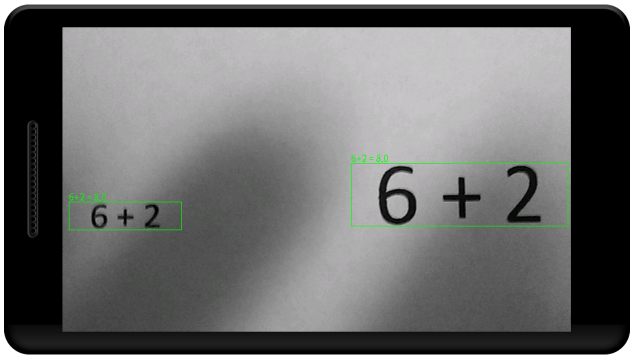

Figure 2a: Final Result under Shadow

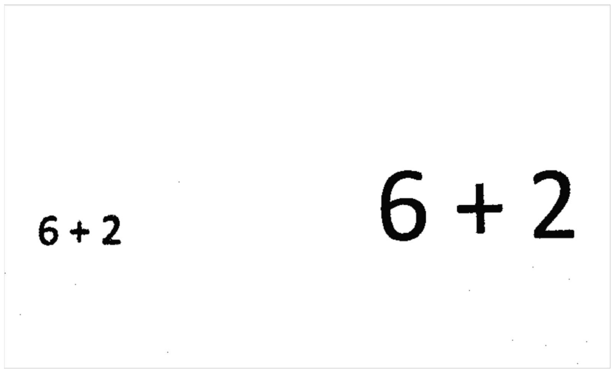

Figure 2b: Thresholded Image Generated from Image with Shadows

D. Character Recognition

The contents of each bounding box are then sent to the OCR engine. We use the Tesseract[4] OCR engine, which is an open source project currently supported by Google. The Tesseract classifier can be run in 2 different modes: (1) an "adaptive classifier" mode which manually generates a feature representation of the image to classify characters and (2) a convolutional neural network algorithm referred to as Cube. The neural network mode is slightly more accurate, but it runs more slowly.[5] We determined that the first mode provides satisfactory accuracy (as long as the input image has been processed as discussed previously). Furthermore, lower computational complexity is a high priority for mobile applications. To increase accuracy, we set Tesseract to match the text only to the symbols of interest, i.e. 0-9 and "+-()/x". We also set Tesseract to search only for single lines of text, which reduces its runtime.

E. Evaluation

The OCR engine then sends the mathematical expression to be evaluated. For simplicity, we do not support algebra, calculus, etc. For this stage, we use the lightweight Javaluator[6] library. First, we preprocess the text recognized by Tesseract (e.g. we convert artifacts like "--" to "-"). Javaluator then parses the string and, if it finds that the string is a valid mathematical expression, calculates the result.

F. Display

After obtaining the result of the expression, we overlay the result on the mobile device's screen next to the corresponding equation. Note that we display the captured image on the device throughout steps A-E as well. Moreover, we color-code the bounding boxes to indicate progress. When bounding boxes are found, we display yellow rectangles around them, as shown in Figure 3 below. As the contents of each bounding box is parsed and evaluated, we change the border color to green (if the evaluation is successful) or red (if

[4] https://github.com/tesseract-ocr/tesseract/

[5] Smith, Ray. *An overview of the Tesseract OCR engine*, 2007

[6] http://javaluator.sourceforge.net

the evaluation fails due to an OCR or mathematical syntax error). Note that we display the result in a sensible location based on the location of the bounding box itself. If the bounding box is at the very top of the screen and doesn't take up the entire screen, then we display the solved equation at the bottom of the box. However, if the bounding box takes up the entire screen, then we display the solved equation inside the bounding box. Displaying the full expression, along with the result, allows the user to be confident that the correct expression was indeed recognized.

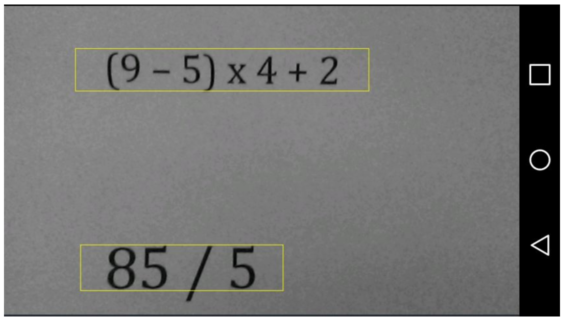

Figure 3: Yellow display shows that evaluation is still in progress

III. RESULTS

The application successfully reads, parses, and evaluates mathematical expressions accurately and rapidly. It succeeds at assessing multiple equations simultaneously, even when they are at different scales, as shown in Figure 4 below. However, the application does fail occasionally at several points along the pipeline.

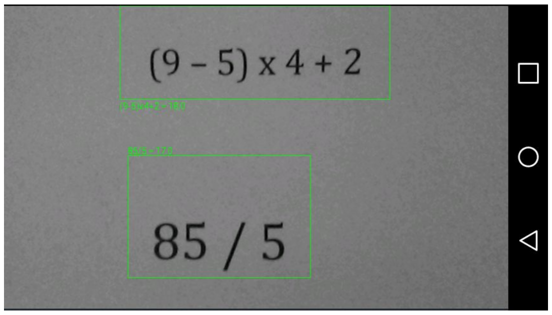

Figure 4: Successful result on multiple equations with different font size

At the beginning of the pipeline, the application may incorrectly label keypoints or, more commonly, may incorrectly group keypoints into bounding boxes. Often this is caused by poor scaling; when the text in the image takes up the entire field of view, the spaces between characters are so large that they are treated separately. The merging discussed above may not resolve this issue if the individual bounding boxes differ greatly along the vertical axis, which is interpreted as being on different lines of text. Other times, the opposite issue occurs and multiple equations are grouped together into a single bounding box. Again, this is affected by the scale of the text and of the image overall. Another error that can result in bounding box confusion is the detection of non-text keypoints. For example, if there is a line or border nearby, this may be grouped together with some text and corrupt the character recognition.

In all of the above cases, however, the error can be easily noticed and rectified by the user. For example, if a bounding box contains non-text or multiple lines of text that should be separate equations, this will cause the evaluator to fail and a red rectangle will display. It will be clear from the location and contents of the rectangle what has gone wrong and the user can then re-take the image slightly closer or farther away or shifted in order to correct the issue.

Another of the occasional sources of error is blur, which can be caused by unsteady hands when the image is taken. This causes the character recognition engine to misread the text, as shown in Figure 5 below. However, this error is also noticeable and fixable by the user. Either the evaluator will fail, causing the rectangle to turn red, or the evaluator will "succeed" but display an incorrect equation. Both cases are shown in Figure 5.

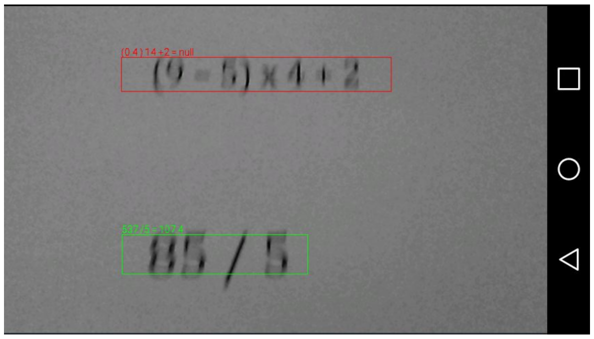

Figure 5: Errors due to blur

A minor and rare issue arises when results overlap when they are displayed. If an equation is at the top of the image, its result is displayed on the bottom of its bounding box. If another equation is below it, it is possible that the result displayed for that equation will overlap the display of the first. More generally, rectangles and result displays can collide, causing legibility difficulties.

Further, the application assumes that the image satisfies certain conditions; it is inaccurate when the image is heavily skewed, when the text is handwritten, or when the equation involves characters outside of the assumed alphabet. The heavy skew causes the characters of the equation to be bounded individually, as shown in Figure 6 below. We chose not to correct for skew since this would slow down the application and the user can easily rotate the device to avoid significant skewing.

Note that a real user will always introduce some amount of skew. For moderate amounts of skewing, the bounding box detection still succeeds. Furthermore, we found that in practice, Tesseract's internal de-skewing algorithm already corrects for a few degrees of skew. Therefore, the app can tolerate the moderate amounts of skew it is likely to encounter.

The app can successfully bound handwritten equations, but the OCR engine was not trained on handwriting--it was only trained on a set of approximately 8 "typical" typed fonts.

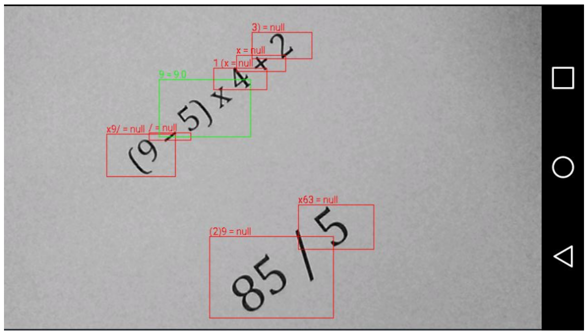

Figure 6: Error due to skew

When the image is reasonably steady, well-scaled, correctly oriented, and contains only typed arithmetic, however, the application works very well. It is quick, accurate, and robust against shadows and other noise.

IV. FUTURE WORK

There are several ways that we could expand upon this project:

We could experiment with different approaches to bounding equations with rectangles. For example, we could compare the performance of different keypoint detectors.

We could also design an evaluator to solve more complex equations, such as those found in algebra and calculus, and to train the character recognition engine to recognize advanced mathematical symbols. We could even connect the application with the Wolfram Alpha API[7], which would enable the user to get a result for even the most complex expressions.

Another possible improvement is to display not only the equation and results, but also intermediate steps in the solution. For example, explicitly showing carrying and borrowing in addition and subtraction. This would be a particularly helpful feature for educational uses.

We could also improve the character recognition by using an engine trained on handwriting, implementing many possible methods, such as those introduced by Zhang, et al.,[8] or by Ali, et al.[9]

V. ACKNOWLEDGEMENTS

We would like to thank Professor Gordon Wetzstein for his instruction in Digital Image Processing this quarter and Jean-Baptiste Boin for his mentorship and advice throughout this project.

[7] https://www.wolframalpha.com

[8] He Zhang, Jia Liu, Zhengyan Liu, Nan Zhang, Li Wang, Xinrong Lv, Peng Ren, "A fast handwritten numeral recognition framework based on peak densities." *Signal and Information Processing (ChinaSIP), IEEE China Summit and International Conference*, 2015

[9] Syed Salman Ali, Muhammad Usman Ghani, "Handwritten Digit Recognition Using DCT and HMMs." *Frontiers of Information Technology (FIT) 12th International Conference*, 2014