A Scalable Computer Vision Framework For Mobile Device Auto-Typing

Zongyi (Joe) Liu*, Rohit Kumar Gupta, Kun Chen, Bruce Ferry, Simon Lacasse
Department of Benchmarking
Amazon.com
Seattle, Washington, USA
joeliu*, gurohi, kuchen, bferry, lacasse@amazon.com

***Abstract*—In this paper, we present a computer vision framework that controls robots to auto type on a mobile device such as an Android phone or an iPad. The framework consists of three parts: (i) an image undistortion and segmentation algorithm that supports images captured by a top mounted camera or a side mounted camera, (ii) a deep neural network (DNN) algorithm that detects the keyboard region and recognizes isolated characters from an input image, and (iii) a grid click algorithm to construct data points to compute the image to robot coordinate translation matrix that is scalable to any touch device of different sizes. We have demonstrated in this paper the accuracy and scalability of the proposed system.**

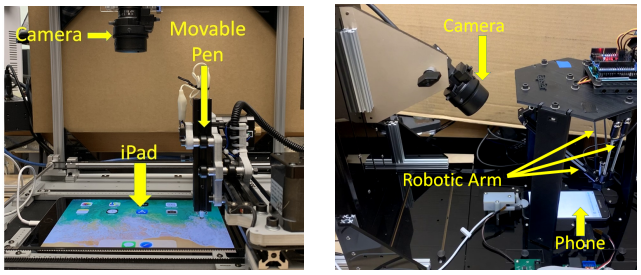
***Index Terms*—deep learning, image undistortion, isolated character recognition, grid pattern click**

I. Introduction

Building a robotic system to intelligently interact with a mobile device has many applications. For example, Liu *et. al* [1] built an automatic system to measure the video quality for online streaming mobile platforms such as Prime Video and Twitch. A smart robot is required to perform three types of actions: (i) *click action* where we need to locate an icon of interest and click the touch screen to open it, (ii) *swipe action* where we need to swipe the touch screen to take an action like closing an app, and (iii) *type action* where we need to input a group of texts from the digital keyboard of the device such as searching a movie title. With the rapid development of object detection algorithms [2]–[4] and text detection [5]–[9] technologies, it is not difficult to build an algorithm to locate an icon or a region of interest, and then build the click action and swipe action on top of it. However, the type action is more difficult because of a few challenges: first, a character usually has a much smaller touching area than an app icon so it requires higher location precision; second, most Optical Character Recognition (OCR) research is focused on scene text detection and recognition with challenging backgrounds, but only few pay attention to the isolated character recognition required for keyboard typing.

In this paper, we propose a computer vision framework that supports the typing action for different kinds of robots. The framework consists of three steps: (i) it uses an image preprocessing algorithm to undistort the captured images and segment out the display screen area, (ii) it uses a DNN based algorithm to detect keyboard region and recognize its isolated characters, and (iii) it runs a grid click algorithm to construct data points to compute image to robot coordinate translation matrix. This framework has been tested with high accuracy in our lab that is running 24/7 and has achieved good performance. The remaining of the paper is organized as follows: from Sec. II to Sec. IV we describe the framework in detail, in Sec. V we evaluate the overall performance of the framework including the accuracy and speed, and in Sec. VI we summarize our findings.

II. Image Artifact Correction And Display Screen Area Segmentation



(a) (b)

Fig. 1. The sample video capturing systems of (a) a Cartesian robot (Emile-3) with the camera mounted on the top, and (b) a Delta robot (Tapster) with the camera mounted on the side.

Generally speaking, a video capturing system for a mobile device has a camera mounted either on the top or on the side of a testing device (slant view) as Fig. 1 shows. So the captured frame contain two types of artifacts: the view angle artifact caused by camera view angles and the image distortion artifact caused by camera lens.

In our framework, the artifact correction is a semi-automated process. First, we created an image I_{orig} with white background and black grid circles that are evenly distributed horizontally and vertically. Given a mobile device, we have it display I_{orig} in the full screen mode, and use the camera to take a snapshot I_{cap}. Then we run the blob detect algorithm to detect the black circle locations L_{cir} from I_{cap} [10], which in turn are used to run the perspective transformation [11] to correct the slant view artifact and undistort transformation [11] to correct the distortion artifact. After the image correction, we run the display screen detection and segmentation algorithm

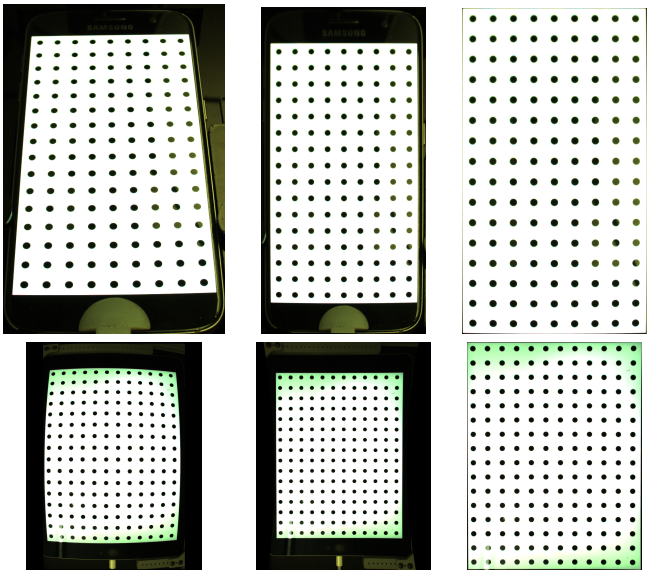


Fig. 2. The sample input and output images of the artifact correction and display screen segmentation algorithm, where the top row is captured from an Android phone by a side mounted camera as shown in Fig. 1 (b), and the bottom row is captured from an iPad device by a top mounted camera as shown in Fig. 1 (a).

in order to reduce background noises during the video analysis process. Briefly speaking, the detection algorithm runs contour detection on an artifacts corrected image, and then picks the contour with minimum area and covers all L_{cir}. Fig. 2 shows the input and output images of this process, where we saw that the input images have strong slant view artifact and image distortion artifact. But in the output images these artifacts have been successfully removed. It is worth noting that we only need to run this correction and segmentation process once during the initial calibration time, and re-use the generated matrices until the camera or testing unit shifts or moves forcing the calibration to be run again.

III. Keyboard Detector and Isolated Character Recognizer

The keyboard character recognizing algorithm has been described in detail in [12], and here we only briefly list its steps. The first step is the keyboard region detector built on the top of the popular Single Shot multibox Detection (SSD) [2] algorithm proposed by *Liu, et. al.* To train the SSD, we used 20,000 synthesized images: first, we downloaded 100+ different styles of keyboard images from the web, rendered it on the top of an image randomly selected from the COCO 2017 dataset [13], then added Gaussian noise. We picked the $300x300$ SSD model and modified it so that only one foreground class, the keyboard region, is defined in the classification layers. Similar to the original SSD model training process [2], we used the VGG-16 network [14] to initialize the parameters of the first five layers.

The second step is a DNN model (M_{char}) that locates and decodes the characters (case sensitive) in one pass. The M_{char} is based on the SSTD [5] [6] algorithm with a few modifications:

1) We changed the input data scaling dimension from $512x512$ to $512x256$, which not only better fits the aspect ratio of a keyboard image, but also runs much faster.
2) We changed the output foreground class number from one (text vs. non-text) to N_C where each character is one class. An upper case character will have different class label than its lower case.
3) Since the characters on a keyboard are separated, we added a strict non-maximum suppression (NMS) layer so that outputs with any overlapping bounding boxes are filtered based on their class confidence scores.
4) We removed the top two convolution layers (conv9_2 and conv10_2) to increase the speed.

To train M_{char}, we manually labeled the locations and class labels for characters of interest (COI) from 634 images collected from Android phones, iPhones and smart TVs. Similar to the keyboard area detector, we also synthesized the training images by adding Gaussian noises. For our system, the COI is set to have 68 characters that include letters 'a'-'z' and 'A'-'Z', numerical number '0'-'9', symbol '-', '+', and four control keys: '←' (backspace), '⇧' (caption), '↵' (return), '␣' (white space). For applications that need a different character set, the system can easily be re-trained.

The third step is a semi-supervised character correction algorithm. Specifically, we split the output characters of M_{char} into two sets: C_{high} and C_{low} that have high confidence classification scores and low confidence classification scores from M_{char}, respectively. Then we learned the hand-crafted features from C_{high} on the fly and then apply them to fix the errors. Specifically, we performed the following semi-supervised learning:

1) Geometry heuristics correction: we learned the geometric features such as width and height from C_{high}, and applied them to filter out the characters in C_{low} with significantly different values, i.e., we removed size outlier characters that are likely to be the background noises.
2) Upper-lower case correction: knowing the case of each character is important for applications such as typing in passwords to unlock screen. However, for some characters such as 'z' and 'o', it is difficult to make accurate estimation from an individual character. It is our observation that characters from a keyboard image are usually all in upper case or all in lower case, so that we estimated the keyboard's state by counting the number of upper case characters and lower case characters in C_{high}, then selecting the one with the majority vote as the keyboard case state and applying it to all the characters. So when we are performing a case sensitive typing such as inputting password, we will first estimate the state of the keyboard, then press first the '⇧' (caption) key when needed.

3) Dictionary correction: similar to most OCR algorithms that have a dictionary containing popular words, we also built a dictionary that contains the popular lines in keyboards, e.g., "1234567890" and "asdfghjkl". For an input image, we build lines from C_{high} by grouping characters that are horizontally aligned with each other. Then for line (L_{comp}) from C_{high}, we found the best match line (L_{dict}) from the dictionary using the *Longest Common Subsequence* (LCS) algorithm and then apply L_{dict} to correct L_{comp}.

IV. Grid Click Algorithm

After the keyboard character detection and recognition, we have obtained the image coordinates (XY_I) for each character in the input (corrected) image. However, robotic systems move in a world coordinate system and not image coordinate system, hence to enable a robotic system to click these characters, we still need to compute the translation matrix A that converts XY_I to the robot coordinate (XYZ_R). Here we assume that a mobile device is mounted on a fixed flat base such that the touch depth Z_R is a constant value that can be configured during device setup time. So the formula is simplified into the 2D space as defined in Eq. 1. To solve A, we need a set of robot coordinates $S(XY_R) = [XY_R(1), XY_R(2), ..., XY_R(n)]$ and their corresponding image coordinates $S(XY_I) = XY_I(1), XY_I(2), ..., XY_I(n)$ as defined in Eq. 2. To get $S(XY_R)$ and $S(XY_I)$, we have the mobile device under test (DUT) to run a in-house built java-script program: each time when the DUT's touch screen is clicked, the program draws a black circle centered at click location and clears the rest of the areas of the display screen to white background. So after a click action at some known robot coordinate $XY_R(i)$, we will capture an image and then run the blob detection algorithm [10] to locate the black circle's center in image coordinate $XY_I(i)$. We can use ($XY_R(i)$, $XY_I(i)$) as one data point for the transformation matrix A computation.

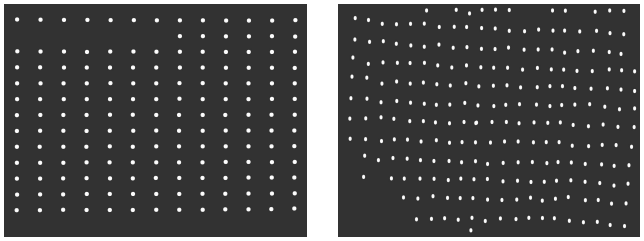
$$A \times XY_I = XY_R \tag{1}$$

$$A = S(XY_R) \times S(XY_I)^+ \tag{2}$$

where $S(XY_I)^+$ is the psuedo-inverse of $S(XY_I)$.

To get better precision, we need more data points for Eq. 2. To reduce over-fitting, these points should be sampled from different display screen area of a DUT. One solution is to manually move the robot arm around and then click, or manually configure the values for XY_R first. But neither is scalable. In our framework, we automate this process by building an algorithm to control the robot to perform grid pattern clicks on a DUT, as described in Algo. 1. Similar to the image artifact correction and display screen area segmentation process as described in II, we only need to run this grid click algorithm once during the initial calibration time, and re-use the generated matrices until the camera or testing units shift or move, forcing to the calibration algorithm again.

Fig. 3 plots the circle set constructed by running this algorithm on a Cartesian robot to click an iPad device (162 data points) and a Delta robot to click a Samsung phone (250 data points). We can see that although a few small areas are missed due to the miss touch problem (either shallow touch or too deep touch of a robotic pen), the data points have covered most of the display screen area.



(a) (b)

Fig. 3. The circle set constructed by running the grid click Algo. 1 on (a) a Cartesian robot with a top view camera as shown in Fig. 1 (a) to click an iPad device with $\Delta(X)$ set to 15mm and $\Delta(Y)$ set to 10mm, and (b) a Delta robot with a side view camera as shown in Fig 1 (b) to click a Samsung phone with both $\Delta(X)$ and $\Delta(Y)$ set to 5mm. The input images have been corrected and segmented using the algorithm described in Sec. II.

V. Experiments And Performance Evaluation

We first evaluated the performance of the keyboard detection and character recognition algorithm. Since we were not able to find a public keyboard database with character level ground truth labeling, we built a testing dataset using Bing and Google image search engine to download 140 images taken from smart phones that are commercially available. Then we manually labeled the bounding boxes and class labels for the 68 characters of interest (COI). Fig. 4 lists some sample output images where we can see that the algorithm achieves very good performance: it is able to recognize the occluded characters such as row 1 (a), the white texts and black texts in the same image such as row 1 (b), the numerical panels such as row 2 (b) and row 3 (a), or non touch-screen keyboard such as row 4 (a) and (b).

The M_{char} performs character detection and recognition of a keyboard region in one pass. The detection consists of recall and precision accuracy. For the recall performance, we define that a character is detected if the IOU (intersect area over union area) between its detected bounding box and the ground truth bounding box is over 0.2. For the precision performance, we defined an algorithm bounding box that had IOU less than 0.2 with any ground truth bounding box was a false positive. If there was more than one output bounding box mapping to the same ground truth bounding box, we picked the one with the highest class confidence score from M_{char} and set the rest as false positives. We list the performance of our algorithm in Table. I. For comparison, we also picked three text detection algorithms recently published: CTPN [15] in 2016, SSTD [5] in 2017 and RRD [16], and had them run on our testing dataset using the models publicly released by the authors. We saw that these algorithms worked poorly on the isolated characters. Of course, the performance could be improved if we re-train

Algorithm 1 The grid click algorithm to construct the map from image coordinate set $S(XY_I)$ to robot coordinate set $S(XY_R)$. Here ΔX and $\Delta(Y)$ are pre-configured values.

```
# Initialize
Move robot arm to XY_R(C) that is usually configured to be the center of the display screen
X_R, Y_R ← XY_R(C)
S(XY_I) ← []
S(XY_R) ← []

# Move the robot arm upwards
repeat
    count1 = process_row(X_R, Y_R, left)
    count2 = process_row(X_R, Y_R, right)
    Y_R = Y_R − Δ(Y)
until count1 + count2 = 0

# Move the robot arm downwards
repeat
    count1 = process_row(X_R, Y_R, left)
    count2 = process_row(X_R, Y_R, right)
    Y_R = Y_R + Δ(Y)
until count1 + count2 = 0

Function process_row(x0_R, y0_R, dir)
Count = 0
X_R = x0_R, Y_R = y0_R
while True do
    retry_count = 0
    Perform a click at X_R, Y_R
    if a circle centered at (X_I, Y_I) is detected on the display screen then
        S(XY_I) ← (X_I, Y_I)
        S(XY_R) ← (X_R, Y_R)
        if dir = left then
            X_R = X_R − Δ(X)
        else
            X_R = X_R + Δ(X)
        end if
        Count = Count + 1
    else if retry_count > 0 then
        # Since we have already retried,
        # it means the arm is out of the display screen area
        break
    else
        # Retry one more time in case this click is missed
        retry_count = retry_count + 1
    end if
end while
return Count
```

(a) (b)

Fig. 4. The sample output of our keyboard detection and character recognition algorithm on the testing dataset downloaded from Bing and Google image search engine, where the detected characters that have correct labels (case-sensitive) are boxed with green rectangles, and those have in-correct (including case error) labels are boxed with red rectangles. Here the characters of interest (COI) include 'a'-'z', 'A'-'Z', '0'-'9', '-', '+', '←' (BS), '⇧' (CL), '↵' (RT), '␣' (SP).

TABLE I
THE CHARACTER DETECTION ACCURACY BETWEEN OUR ALGORITHM AND OTHER STATE-OF-ART TEXT DETECTION ALGORITHMS. THE PERFORMANCE IS COMPUTED USING A TESTING DATASET WITH MANUALLY LABELED 4206 CHARACTERS FROM 140 IMAGES.

Algorithm	Recall Rate	Precision Rate
Ours	98.9%	98.6%
CTPN [15]	69.7%	N/A
SSTD [5]	21.9%	N/A
RRD [16]	11.3%	N/A

their models using our training data, but it also suggested that algorithms designed for scene text won't directly work for keyboard character images w/o finetuning.

For the recognition performance, we first mapped the algorithm output to the ground truth data using character bounding box IOU. Then we performed both case-sensitive and case-insensitive label matching. Table II lists the performance of our algorithm. For comparison, we also listed the result of Tesseract 4 [17], an OCR engine that does both detection and recognition and is popularly used in industry. Here we can see that our algorithm has achieved above 98% accuracy that is more than 40% improvement over Tesseract.

TABLE II

THE CHARACTER RECOGNITION ACCURACY BETWEEN OUR ALGORITHM AND THE TESSERACT 4 ALGORITHM. THE PERFORMANCE IS COMPUTED USING A TESTING DATASET WITH MANUALLY LABELED 4206 CHARACTERS FROM 140 IMAGES.

Algorithm	Case Sensitive	Case in-sensitive	Total Characters
Ours	98.2%	98.8%	4206
Tesseract 4 [17]	55.7%	58.4%	3771 [1]

TABLE III

THE PERFORMANCE OF THE END-END TEST USING A CARTESIAN ROBOT AS SHOWN IN FIG 1 (A) AND A DELTA ROBOT AS SHOWN IN FIG 1 (B). THE DUT IS A SAMSUNG S7 PHONE WITH $113mm \times 64mm$ DISPLAY SCREEN AND $6mm \times 6mm$ CHARACTERS IN THE DIGITAL KEYBOARD. NOTE THE CHARACTERS TYPED IN HAVE COVERED ALL 26 LETTERS

Robot Name	Robot Type	Number of Characters Typed in	Number of Incorrect Clicks
Tapster	Delta	1800	0
Emile-3	Cartesian	2800	0

In addition to accuracy, we also studied the speed: we tested our algorithm on a machine with ubuntu 16.04 OS, one NVIDIA GeForce Titan X Pascal GPU, and one Intel(R) Xeon(R) CPU @2.30GHz. The result showed that in average the SSD model takes about 30 milliseconds to process one image, the M_{char} model and the semi-supervised error corrector take about 80 and 8 milliseconds to process one keyboard region, respectively. So for applications that only need to click in one keyboard area, the algorithm can process ~ 8 frames per second which is fast enough to perform interactive typing in real-time.

We also designed a test to evaluate the end-end performance of our framework in terms of keyboard typing accuracy. The test consisted of three steps: (i) we picked a DUT and ran the image correction algorithm and the grid click algorithm as described in Sec. II and Sec. IV, respectively, (ii) we opened an app with the digital keyboard input, e.g., a notebook, and then ran the keyboard detection and recognition algorithm to locate each character on the display screen; and (iii) by using the information collected from step (i) and (ii), we controlled the robot to type in a word $W(in)$, and then compared it with the word $W(out)$ as displayed on the device. If $W(in)$ equals to $W(out)$, then we count it as a match, otherwise as a mismatch. To automate this process, after type in $W(in)$, we took a snapshot of the DUT and send it to a DNN based OCR engine to recognize $W(out)$. Since $W(out)$ has a simple background, the OCR usually had very high accuracy. To further minimize the OCR error, we picked $W(in)$ from a set of 393 words with each ten characters long as listed in [18] that covered all 26 letters. This is based on our observation that OCR has less errors in long English characters since they have built-in dictionary correction.

In our test, the DUT was a Samsung S7 phone because it has a small display screen with size $113mm \times 64mm$, and its digital keyboard has characters with size $6mm \times 6mm$. For mobile devices with larger display screen like an iPad, we expect the algorithm to achieve better performance. Table. III listed the results of the test using a Cartesian robot (Emile3) and a Delta robot (Tapster), where we can see that the algorithm achieved 100% accuracy on both robots. This suggests that by using the data points built by Alg. 1, we can compute the translation matrix A with very high precision.

[1] special characters such as '↵' and '␣' are excluded as they are not in the Tesseract dictionary.

VI. CONCLUSION

In this paper, we presented a computer vision framework for robot-mobile devices interaction and demonstrated a use case of keyboard typing. This is an end-to-end solution including an image artifact correction, display screen segmentation algorithm, a keyboard character recognition algorithm, and a grid click algorithm. For the image artifact correction and a display screen segmentation algorithm, we showed that it works for images captured by both top mounted and side mounted cameras. For the keyboard character recognition algorithm, we showed that it has over 98% accuracy when tested on 140 images with 4206 characters. For the grid click algorithm, we designed it as an automated process that can easily work on DUTs of various sizes, and evaluated it with a comprehensive test. Specifically, we used the framework to control a Cartesian robot and a Delta robot to type a list of words on a Samsung S7 mobile phone, and then compared what we typed with what was displayed on the DUT. The results showed that the algorithm has achieved 100% accuracy on both robots after typing in 2800 and 1800 characters (both covering all 26 letters), respectively.

In terms of future study, one direction is to improve its scalability for a moving camera. Today, our framework needs to re-run the image artifacts correction step and the grid correction algorithm each time when a camera is moved. For robots that move cameras frequently, we would need to improve these steps so that they could better handle view angle variations.

REFERENCES

[1] Z. Liu, B. Ferry, S. Lacasse, S. Fonte, R. Matthieu, and G. Larsen, "A scalable automated system to measure user experience on smart devices," in *Proceedings of International Conference on Consumer Electronics (ICCE)*, 2019.

[2] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," in *The IEEE European Conference on Computer Vision (ECCV*, 2016.

[3] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Advances in Neural Information Processing Systems (NIPS)*, 2015.

[4] K. He, G. Gkioxari, P. Dollár, and R. B. Girshick, "Mask R-CNN," *CoRR*, vol. abs/1703.06870, 2017. [Online]. Available: http://arxiv.org/abs/1703.06870

[5] P. He, W. Huang, T. He, Q. Zhu, Y. Qiao, and X. Li, "Single shot text detector with regional attention," in *Proceedings of Internatioanl Conference on Computer Vision (ICCV)*, 2017.

[6] P. He, W. Huang, Y. Qiao, C. C. Loy, and X. Tang, "Reading scene text in deep convolutional sequences," in *Proceedings of AAAI Conference on Artificial Intelligence, (AAAI)*, 2016.

[7] F. Wang, L. Zhao, X. Li, X. Wang, and D. Tao, "Geometry-aware scene text detection with instance transformation network," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[8] B. Shi, X. Bai, and C. Yao, "An end-to-end trainable neural network for image-based sequence recognition and its application to scene text recognition," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 11, pp. 2298–2304, 2017.

[9] J. Ma, W. Shao, H. Ye, L. Wang, H. Wang, Y. Zheng, and X. Xue, "Arbitrary-oriented scene text detection via rotation proposals," *IEEE Transactions on Multimedia*, vol. 20, no. 11, pp. 3111–3122, 2018.

[10] "Opencv simple blob detector." [Online]. Available: https://docs.opencv.org/3.3.0/d0/d7a/classcv_1_1SimpleBlobDetector.html

[11] "Opencv geometric image transforms." [Online]. Available: https://docs.opencv.org/3.4.4/da/d54/group__imgproc__transform.html

[12] Z. Liu, B. Ferry, and S. Lacasse, "A deep neural network to detect keyboard regions and recognize isolated characters," in *Proceedings of The 2nd International Workshop on Machine Learning at International Conference on Document Analysis and Recognition (ICDAR)*, 2019.

[13] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: common objects in context," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2014.

[14] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) workshop*, 2014.

[15] Z. Tian, W. Huang, T. He, P. He, and Y. Qiao, "Detecting text in natural image with connectionist text proposal network," in *The IEEE European Conference on Computer Vision (ECCV)*, 2016.

[16] M. Liao, Z. Zhu, B. Shi, G.-s. Xia, and X. Bai, "Rotation-sensitive regression for oriented scene text detection," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[17] "Tesseract-ocr 4." [Online]. Available: https://github.com/tesseract-ocr

[18] "What are 10-letter words with no repeating letters?" [Online]. Available: https://www.quora.com/What-are-10-letter-words-with-no-repeating-letters