A Deep Neural Network to Detect Keyboard Regions and Recognize Isolated Characters

anonymous

***Abstract*—Some robot systems need to interactively auto-type on touch screens using a digital camera as the input source. To do so, it is important to have an algorithm that reliably detects the keyboard region and locates and recognizes its characters from an image. However, today most research efforts in the optical character recognition (OCR) area is focused on scene text detection and recognition. Though these algorithms work well on images with strong noise impacts, they perform poorly on keyboard texts that have isolated characters. In this paper, we present a framework to solve this problem. The algorithm consists of three steps: (i) a Single Short Detector (SSD) to search the keyboard regions from an input image, (ii) a deep neural network model that locates and recognizes the individual characters (case sensitive) in one pass for a keyboard region, and (iii) a semi-supervised corrector that fixes the errors in step 2 using hand-crafted features. We will show that this algorithm is able to process different types of keyboards, and is robust to varying noise levels and text occlusions.**

***Index Terms*—OCR, isolated character recognition, keyboard detection, deep learning.**

I. Introduction

Interactively auto-typing on a keyboard or a touch screen is required in some robot systems. For example, we have built a Mobile eXperience Automation (MXA) system [1] that measures the streaming quality of services such as Amazon Prime Video on a mobile device like an IPhone or an IPad. It first captures the movie playing using a digital camera and then performs analysis. To work 24/7 without human input, a MXA system needs to intelligently click the touch screen of the device, in order to input a movie title or unlock the screen when needed.

The OCR technology has improved significantly using deep learning technology [2]–[9]. However, most research is focused on the area of scene text detection and recognition, and do not work well on keyboard images that have *isolated* characters instead of words. In this paper, we proposed a novel approach to solve this problem. Our algorithm consists of a keyboard region detector where we applied the *Single Shot Multibox Detector* (SSD) [10] to find all keyboard regions from an input image, an isolated character handler where we built a deep neural network (DNN) based on the *Single Shot Text Detector* (SSTD) [3], [11], and a semi-supervised corrector where we learned features from the high-confidence characters and then applied these features to fix errors such as character occlusion.

The main contributions of our work include: (i) separating keyboard region detection from character recognition so that characters with very small font size can be picked up, (ii) a DNN that is able to detect and recognize isolated characters in a single pass and is robust to background noises and mild to moderate variations in view orientation, and (iii) a semi-supervised character correction algorithm that can fix hard problems such as occluded characters.

This paper is organized as the follows: in Sec. II we described the algorithm in detail, in Sec. III we evaluated the algorithm with a testing data set and showed that it outperforms the state-of-art text detection algorithms, and in Sec. IV we concluded this paper and explored potential future works.

II. Algorithm Description

Our algorithm performs detection and recognition in three steps: a). it detects all keyboard regions from an input image, b). it crops out a sub-image for each region, scales it and locates its individual characters and decodes them, and c). it performs a semi-supervised correction using the features learned from high-confidence characters.

A. Keyboard Region Detection

Detecting text with very small font is challenging, particularly for isolated characters because they don't have neighboring texts such that their features are harder to differentiate from non-text noises. To address this problem, we designed our algorithm to first detect the keyboard regions that are relatively large in an input image, and then scaled the keyboard region to pick-up the small font characters. We employed the popular SSD [10] algorithm proposed by *Liu, et. al.*. To train the SSD, we used 20,000 synthesized images (I_{syn}): first, we downloaded 100+ different styles of keyboard images ($I_{keyboard}$) from the web, rendered it on the top of an image randomly selected from the COCO 2017 dataset [12], then added Gaussian noise (I_{noise}), as defined in Eq. 1.

$$I_{syn} = (\alpha I_{COCO} + (1 - \alpha) A I_{syn}) + I_{noise}(\sigma) \tag{1}$$

where α is the transparency factor, A is the affinity transformation matrix and σ is the standard deviation of Gaussian noise, with their values randomly generated. Fig. 1 lists two sample synthesized images that are used for training.

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 [10], we used the VGG-16 network [13] to initialize the parameters of the first five layers. Fig. 2 shows a sample output of the algorithm, where we can see that it correctly locates both keyboard regions with different background and foreground colors.

Fig. 1. Two synthesized sample images used for keyboard SSD training.

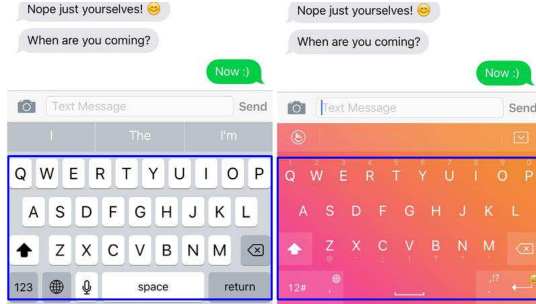

Fig. 2. One sample output image of the keyboard region detection algorithm, where the detected keyboard regions are boxed with blue rectangles.

B. Character Detection and Recognition

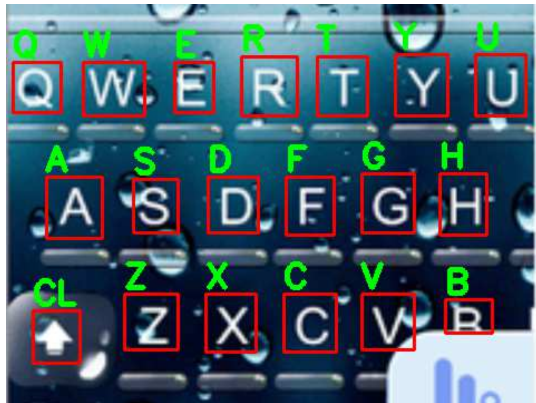

Fig. 3. The output of the character detection and recognition model (M_{char}) on a sample image with noisy background and small font height ($\sim$ 8 pixels). The character regional proposals are boxed with red rectangles, and the class labels are rendered on the top in green color. The image has been cropped and scaled-up for better visualization.

After a keyboard region is detected, using a dynamic thresholding algorithm such as the OTSU algorithm [14] to segment the characters seems to be an easy and quick solution. However, this approach has several problems: first, the input image may contain noise that a dynamic thresholding algorithm won't be able to process, e.g., the image as shown in Fig. 3. Second, a keyboard region may have both bright characters and dark characters, making the foreground selection a difficult task. And third, the dynamic thresholding can only do segmentation and we will still need a recognition algorithm. So in our approach, we chose a DNN model (M_{char}) that locates and

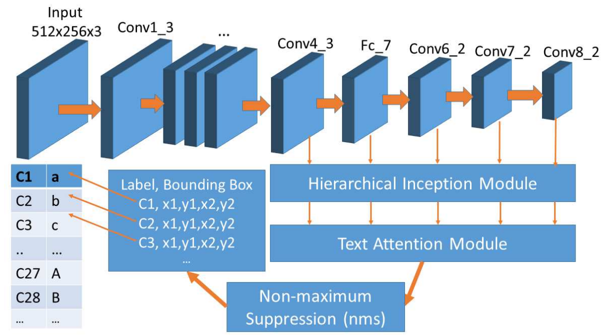

Fig. 4. The deep neural network structure of our character detection and recognition model.

decodes the characters (case sensitive) in one pass. As shown in Fig. 4, the M_{char} is based on the SSTD [3] [11] 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. And a upper case character will have different class label from 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 improve the speed.

To train M_{char}, we manually annotated the locations and class labels for characters of interest (COI) from 634 images collected from Android phone, IPhone and smart TVs such as a Samsung TV. Similar to the keyboard area SSD training data, we also synthesized images by adding Gaussian noises. For our video quality measurement MXA 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). And for applications that need a different character set, the system can easily be re-trained.

Fig. 3 shows the output of M_{char} on a sample image with noisy background and small fonts (8 pixels height). We can see that it is able to detect and recognize all characters correctly, including those with water drop (a noise source) near letters like such as ‘Q’ and ‘E’ in the first line, and the partially occluded ‘B’ in the bottom right.

For camera-based images, it is also important to handle mild to moderate variations in image orientation. To have an insight into how robust our algorithm is in this area, we picked one image with a near zero-degree keyboard region, rotated it from $-20°$ to $20°$, and then ran SSD and M_{char}. Fig. 5 shows the output, where we can see that the SSD works well with the view variations. And the M_{char} achieves 100% accuracy

without any rotation, only misses 2 characters (‘r’ and ‘m’) when rotated by $\pm 15°$, and has 80% accuracy when rotated by $\pm 20°$.

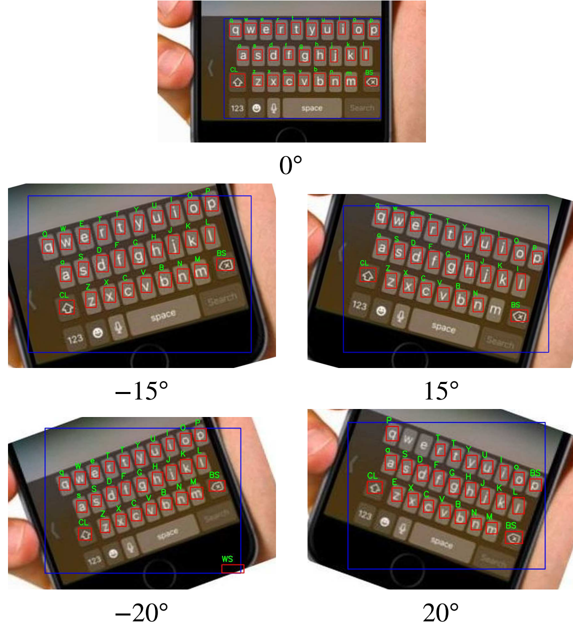

Fig. 5. The SSD and the character detection and recognition model M_{char} performance on a sample image with different variations in view orientation. The detected keyboard is boxed with blue rectangle, The character regional proposals are boxed with red rectangles, and the class labels are rendered on the top in green color. The images have been cropped and scaled-up for better visualization.

C. Semi-supervised Character Correction

We have shown that M_{char} has promising performance in the presence of noise and orientation artifacts. However, a few problems still exist: 1) decoding errors as some characters are difficult to differentiate even by humans, e.g., letter ’O’ vs. numerical ’0’, or letter ’l’ vs. numerical ’1’, 2) missing characters because they are occluded, and 3) false positives as some noises having similar shapes with characters. To further improve accuracy, we built a semi-supervised character correction algorithm. Specifically, we split the output characters into two sets: C_{high} and C_{low} that have high confidence classification scores and low confidence classification scores from M_{char}, respectively. And we learn the hand-crafted features from C_{high} on the fly and then apply them to fix the errors.

The first part of our corrector is the geometry heuristics correction, where it computes the character features F_W, F_H and $F_D(H)$ from C_{high} as defined in Eq. 2 and 3 and uses them to filter out the characters in C_{low} with significantly different values, i.e., it removes size outlier characters that are likely to be the background noises.

$$(F_W, F_H) = \forall_{char \in C_{high}} \text{median}(W_{char},\ H_{char}) \tag{2}$$

where W_{char} and H_{char} are the width and height of character.

$$F_D(H) = \forall_{(char_1, char_2) \in C_{high}} \text{ median}(X_1(char_2) - X_2(char_1)) \tag{3}$$

where $char_2$ is the immediate right neighbor of $char_1$, $X_1(char_2)$ is the left boundary of $char_2$ and $X_2(char_1)$ is the right boundary of $char_1$.

The second part of our algorithm is the upper-lower case correction like ‘z’<->‘Z’ or ‘o’<->‘O’, which is important for applications such as typing in passwords to unlock screen. Since the characters from a keyboard image are usually all in upper case or all in lower case, we can estimate the keyboard’s state by counting the number of upper case characters and lower case characters in C_{high}, then picking the one with the majority vote as the keyboard case state and applying it to all the characters.

The third part of our algorithm is 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. We skipped the control characters, e.g., ‘↵’ or ‘␣’, because they may be placed and aligned differently on each keyboard. Then for each computer generated 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 apply L_{dict} to correct L_{comp}. For character decoding error or false positives, we simply needed to update the class labels or discard the characters. For example, if L_{comp} is “ASOFGHJKL” and L_{dict} is “ASDFGHJKL”, then we will update the third character in L_{comp} from ‘O’ to ‘D’ and re-use its bounding box. For missing characters, on the other hand, we also needed to infer their location bounding boxes. In our algorithm, we constructed a bounding box using the geometry features learned from C_{high} as defined in Eq. 2 and 3. For example, if L_{comp} is “XCVB” and L_D is “ZXCVB”, then we will add a new character ‘Z’ and a bounding box (B_Z) with size $F_W \times F_H$, and place B_Z to the left of B_X with the distance of $F_D(H)$. To prevent over-correction, we added two constraints: 1) a match is set to true only when the length of $LCS(L_{dict}, L_{comp})$ is bigger than 40% of the L_{comp}, because an input keyboard may have a special layout that is not included in the dictionary, and 2) if a segment of (connected) characters to be inserted or deleted has a length of two or larger, then we skip the correction.

Fig. 6 lists an example of the correction output, where we can see that it correctly inserts two characters: ‘*e*’ and ‘*z*’, that are occluded by the hand-fingers in the original image. On the other hand, it skips insertion for the second line as the missing segment has two characters (‘*a*’ and ‘*s*’).

III. Experiments

To evaluate our algorithm, we obtained 140 images of commercially available smart phones, and manually labeled the bounding boxes and class labels for the 68 characters of

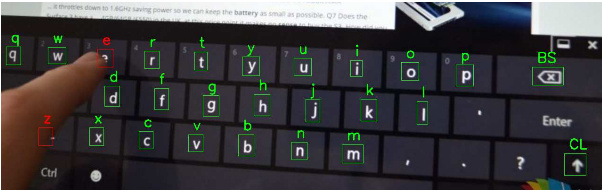

Fig. 6. The output of the semi-supervised correction algorithm, where the characters output by M_{char} are rendered in green color, and the corrected (inserted) characters are rendered in red color. The image has been cropped and scaled-up for better visualization. Here the characters of interest (COI) include 'a'-'z', 'A'-'Z', '0'-'9', '-', '+', '←' (BS), '⇧' (CL), '↵' (RT), '␣' (SP).

interest (COI). Fig. 7 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 (b), row 2 (c) and row 3 (b), the numerical panels such as row 1 (c) and row 3 (c), or non touch-screen keyboard such as row 2 (a) and (d).

Although our algorithm performs both character detection and recognition, we separated these two in the performance evaluation because it is easier to compare with the state-of-art algorithms that usually either do detection or recognition. For the detection performance, we defined 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 0.5. For others' algorithms that reported line/word level bounding boxes, we defined that a character is detected if more than 50% of its ground truth bounding box area is inside a detected text-line bounding box. Strictly speaking, this is the accuracy upper bound because at line/word level granularity, we won't be able to tell if any character in the middle is detected or not by an algorithm. To compute precision rate, we defined an algorithm bounding box that have IOU less than 50% with any ground truth bounding box as a false positive. If there is more than one algorithm bounding box mapping to the same ground truth bounding box, we will pick the one with the highest class confidence score from M_{char} and set the rest as false positives. Since we only labeled COI, we skipped the precision rate computation for other algorithms as they may detect non-COI texts/characters in the testing images, or report line/word line bounding boxes that each matches to multiple ground truth bounding boxes. Table I compares the detection accuracy between our algorithm and three other algorithms, where we can see that ours achieves the best recall rate (98.9%), and it is almost 30% better than the second one (CTPN). For the precision rate, our algorithm also achieves 98.6% accuracy.

To compute recognition performance, we first mapped the algorithm output to the ground truth data using character bounding box IOU as described in the previous paragraph. Then we performed both case-sensitive and case-insensitive label matching. For comparison, we also ran CRNN [15] proposed by Shi *et. al.* in 2017. Specifically, we first excluded the special characters such as '↵' and '␣' because they are not in the CRNN's dictionary. Then we cropped out a sub-image (I_{sub}) for each individual character using their ground truth bounding box and ran CRNN on I_{sub} so that it won't be impacted by the character detection error. Table II lists the algorithms' performance, where we can see that our algorithm achieves 98.2% and 98.8% accuracy for the case-sensitive and case-insensitive recognition, respectively. And it is a 17% improvement over CRNN.

Algorithm	Recall Rate	Precision Rate
Ours	98.9%	98.6%
CTPN [5]	69.7%	N/A
SSTD [3]	21.9%	N/A
RRD [4]	11.3%	N/A

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 4066 CHARACTERS FROM 140 IMAGES.

Algorithm	Case Sensitive	Case in-sensitive	Total Characters
Ours	98.2%	98.8%	4066
CRNN [15]	N/A [1]	83.0%	3750 [2]

TABLE II

THE CHARACTER RECOGNITION ACCURACY BETWEEN OUR ALGORITHM AND THE CRNN ALGORITHM. THE PERFORMANCE IS COMPUTED USING A TESTING DATASET WITH MANUALLY LABELED 4066 CHARACTERS FROM 140 IMAGES.

In addition to accuracy, we also studied the speed: we ran the algorithm on a machine running ubuntu 16.04 with one NVIDIA GeForce Titan X Pascal GPU and one Intel(R) Xeon(R) CPU @2.30GHz for the testing dataset. The result shows 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, e.g., our MXA system that auto plays movie or unlocks screens, the algorithm can process $\sim$ 8 frames per second which is fast enough to perform interactive typing in real-time.

IV. CONCLUSION

In this paper, we presented a system to detect a keyboard region and reliably locate and recognize its isolated characters. The algorithm consists of three parts: a SSD based keyboard detector, a DNN M_{char} model that locates and decodes isolated characters, and a semi-supervised algorithm to reduce errors. We evaluated the algorithm using a testing dataset with 4066 manually labeled characters from 140 images, and showed that the algorithm achieves significantly better performance than the state-of-art scene text algorithms in terms of both detection and recognition. In addition, the algorithm can process about 8 frames per second when running on a Linux machine with one NVIDIA GeForce Titan X Pascal GPU and one Intel(R) Xeon(R) CPU @2.30GHz, which is sufficient for interactive keyboard typing real-time applications.

[1] The CRNN only outputs lower case characters

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

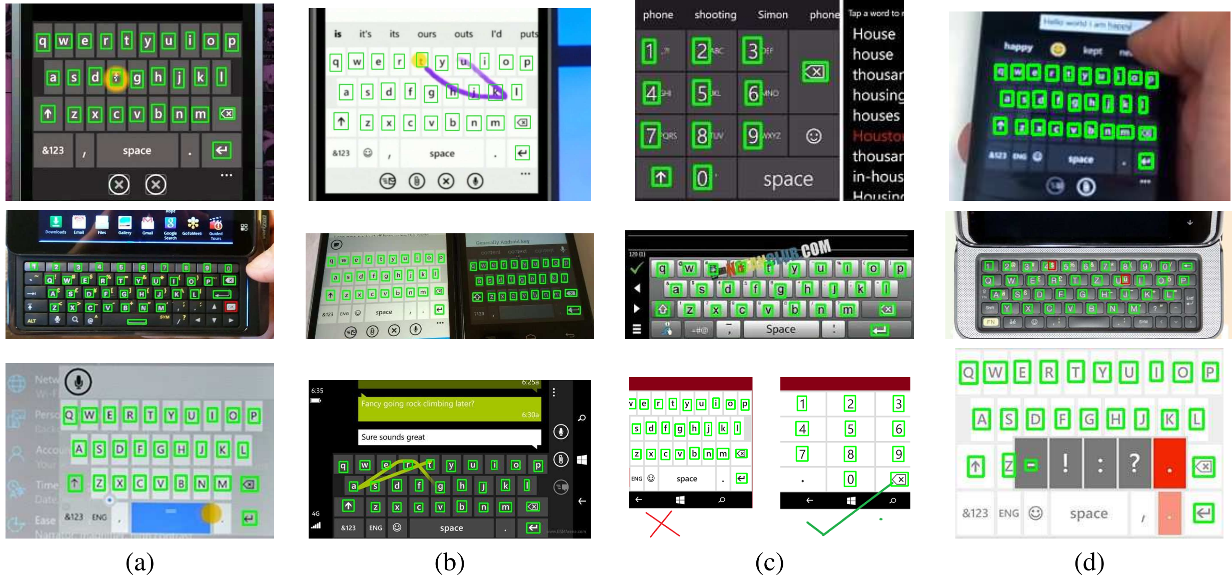

Fig. 7. The sample output of our algorithm on images of Windows Phone and Blackberry keyboards, where the detected characters that have correct case-sensitive labels are boxed with green rectangles, and those have in-correct (including case error) labels are boxed with red rectangles. Here the characters of interest (COI) include ‘a’-’z’, ‘A’-’Z’, ’0’-’9’, ’-’, ’+’, ‘←’ (BS), ‘⇧’ (CL), ‘↵’ (RT), ‘␣’ (SP).

The keyboards we tested in this paper are in the en_US style. To understand how well our algorithm works on other style keyboards, we ran it on an en_UK keyboard without any re-training. Fig. 8 shows the result where we can see that the algorithm captures most characters in COI correctly, though the letter case estimation was incorrect this time. Of course, to achieve best performance for a specific style particularly one with non-English characters, we should re-train the M_{char} model using the corresponding COI and update the text-line dictionary built in the semi-supervised algorithm.

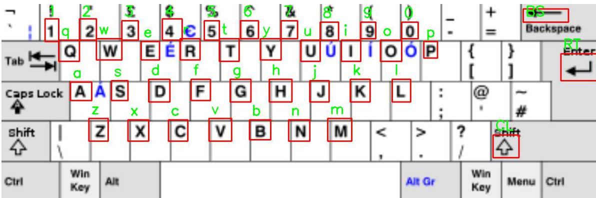

Fig. 8. The output of our algorithm on a *en_UK style* keyboard image w/ re-training (using the same COI as Fig. 7). The character regional proposals are boxed with red rectangles and the class labels are rendered on the top in green color. The image is cropped and scaled-up for better visualization.

In terms of future study, a few improvements can be made. First, we will research if we can replace the hand crafted features with a deep neural network in the semi-supervised algorithm step, so that it can be end-end trainable just like the other two parts of the system. Second, in Sec. II-B we have shown that the M_{char} model can handle about ±15° view orientation variation. Though this is sufficient for our MXA system, some applications may have input images with larger orientation changes. A simple solution is to add images with different view angles and re-train the M_{char} model. But we need to be careful because one character may change to another one after rotation, e.g., ‘b’ changes to ‘q’ after rotation by 180°. A better solution may be to modify the SSD model to output both the bounding box and the view orientation of a keyboard region, then perform affinity transformation before running the M_{char} model.

References

[1] “anonymous.”

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] X. Zhou, C. Yao, H. Wen, Y. Wang, S. Zhou, W. He, and J. Liang, “EAST: an efficient and accurate scene text detector,” in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[7] F. Bai, Z. Cheng, Y. Niu, S. Pu, and S. Zhou, “Edit probability for scene text recognition,” in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[8] Z. Cheng, Y. Xu, F. Bai, Y. Niu, S. Pu, and S. Zhou, “Aon: Towards arbitrarily-oriented text recognition,” in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[9] Z. Liu, G. Lin, S. Yang, J. Feng, W. Lin, and W. Ling Goh, “Learning markov clustering networks for scene text detection,” in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] N. Otsu, "A threshold selection method from gray-level histograms," *IEEE Transactions on Systems, Man, and Cybernetics*, 1979.

[15] 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.