STRING FINDING BASED ON APPLICATION CONNECTED TO SERVER

[1] *Po-Hsun, Huang* (黃柏勳),[2] *Yu – Chen, Chen* (陳昱成),[3] Chiou-Shann Fuh (傅楸善)

[1] Department of Computer Science and Information Engineering,
[2] Department of Electrical Engineering,
[3] Department of Computer Science and Information Engineering,
National Taiwan University, Taipei, Taiwan,
E-mail: haung12936@gmail.com ken7968684@yahoo.com.tw fuh@csie.ntu.edu.tw

ABSTRACT

For this project we propose and implement a reading tool on Andriod platform that can be used to identify keywords on paper-based media. We breakdown the image processing after receiving an image of the media into three steps. In the first step, we pre-process the image using binarization, deskew and de-noising. The second step involves using OCR (in particular, Tesseract OCR engine) to recognize the text and find the keywords. Finally, we highlight the keywords by circling it with bounding boxes.

1. INTRODUCTION

Have you ever read a long paper based article and find yourself unable to locate keywords? With the advancement of digital media, sometimes we take basic word search for granted. But the world is still full of media printed on paper and it would make our lives much simpler if we can apply this searching tool to old fashioned books. We propose a mobile application that will be able to find a word that the user specified through a phone's viewfinder. As soon as the phone detects the word it will highlight it in the viewfinder, saving the user a lot time manually searching for word. him/herself.

For example, we want to search for "right" in this paperbased Declaration of Independence. We only need to use our smart phone to scan over the paper. Whenever the word "right" appears on the phone screen, it will be immediately circled in red bounding boxes.

2. Pre-processing

2.1 Binarization

First we need to separate words from background, so we previously perform a global thresholding, which converts the image into a binary version using Otsu's method. Since our image is mostly black text on white background(or uni-colored text on uni-colored background). Otsu's method is used to automatically perform clustering-based image thresholding or, the reduction of a graylevel image to a binary image. The algorithm assumes that the image contains two classes of pixels following bi-modal histogram (foreground pixels and background pixels), it then calculates the optimum threshold separating the two classes so that their combined spread (intra-class variance) is minimal, or equivalently (because the sum of pairwise squared distances is constant), so that their inter-class variance is maximal. Consequently, Otsu's method is roughly a one-dimensional, discrete analog of Fisher's Discriminant Analysis. Otsu's method is also directly related to the Jenks optimization method. Here, the matter is straight forward. If pixel value is greater than a threshold value, it is assigned one value (may be white), else it is assigned another value (may be black). The function used is cv.threshold. First argument is the source image, which should be a grayscale image. Second argument is the threshold value which is used to classify the pixel values. Third argument is the maxVal which represents the value to be given if pixel value is more than (sometimes less than) the threshold value. It should work farily well in converting the image into a binary version.

The reason not to apply Otsu's method to every block is that if some blocks are background with a number of noise pixels, Ostu's method will keep the noise pixels while classifying the entire block as background will eliminate noise.

The second method is Maximally Stable Extremal regions (MSER)[3]. In computer vision, MSER is widely used as a method of blob detection. Like the SIFT detector, the MSER algorithm extracts from an image a number of co-variant regions, called MSERs. An MSER is a stable connected component of some level sets of the image. In this case, black words on the white paper are successfully detected by using vl-feat vl_mser function (MinDiversity=0.2, MaxVariation=0.7,

Delta=15, MaxArea=0.1). In all possible thresholdings of a gray-level image *I*, we refer to the pixels below a threshold as 'black' and to those above or equal as 'white'. If we were shown a movie of thresholded images I_t, with frame *t* corresponding to threshold *t*; we would see first a white image. Subsequently black spots corresponding to local intensity minima will appear and grow. At some point regions corresponding to two local minima will merge. Finally, the last image will be black. The set of all connected components of all frames of the movie is the set of all maximal regions; minimal regions could be obtained by inverting the intensity of *I* and running the same process. In many images, local binarization is stable over a large range of thresholds in certain regions. Such regions are of interest since they posses the following properties:

- Invariance to affine transformation of image intensities.
- Covariance to adjacency preserving (continuous) transformation on the image domain.
- Stability, since only extremal regions whose support is virtually unchanged over a range of thresholds is selected.
- Multi-scale detection. Since no smoothing is involved, both very fine and very large structure are detected.

The process produces a data structure storing the area of each connected component as a function of intensity. A merge of two components is viewed as termination of existence of the smaller component and an insertion of all pixels of the smaller component into the larger one. Finally, intensity levels that are local minima of the rate of change of the area function are selected as thresholds producing MSER. In the output, each MSER is represented by position of a local intensity minimum (or maximum) and a threshold. Although the set of extremal regions is covariant with any one-to-one continuous transformation of the image domain and thus covariant to projective transformation, the process of the selection of the maximally stable subset is affine-covariant. The MSERs are therefore only affine covariant.

The third method is locally adaptive thresholding. Unlike the global thresholding technique, local adaptive thresholding chooses different threshold values for every pixel in the image based on an analysis of its neighboring pixels. This is to allow images with varying contrast levels where a global thresholding technique will not work satisfactorily. The reason we choose locally adaptive thresholding instead of global thresholding is that the lighting/brightness of the image is not uniform, which will cause global thresholding to perform poorly in the extreme bright/dark regions.

The idea of locally adaptive thresholding is divide the image into blocks/windows. For each block, use grayscale variance to determine whether the block is uniform. If the block is non-uniform (high variance), apply Otsu's method/global thresholding to the block; if the block is uniform (low variance), classify the entire block as all black or all white based on the mean grayscale value.

Adaptive thresholding typically takes a grayscale or color image as input and, in the simplest implementation, outputs a binary image representing the segmentation. For each pixel in the image, a threshold has to be calculated. If the pixel value is below the threshold it is set to the background value, otherwise it assumes the foreground value.

There are two main approaches to finding the threshold: (i) the Chow and Kaneko approach and (ii) local thresholding. The assumption behind both methods is that smaller image regions are more likely to have approximately uniform illumination, thus being more suitable for thresholding. Chow and Kaneko divide an image into an array of overlapping subimages and then find the optimum threshold for each subimage by investigating its histogram. The threshold for each single pixel is found by interpolating the results of the subimages. The drawback of this method is that it is computational expensive and, therefore, is not appropriate for real-time applications.

An alternative approach to finding the local threshold is to statistically examine the intensity values of the local neighborhood of each pixel. The statistic which is most appropriate depends largely on the input image. Simple and fast functions include the mean of the local intensity distribution, the median value, or the mean of the minimum and maximum values. The size of the neighborhood has to be large enough to cover sufficient foreground and background pixels, otherwise a poor threshold is chosen. On the other hand, choosing regions which are too large can violate the assumption of approximately uniform illumination. This method is less computationally intensive than the Chow and Kaneko approach and produces good results for some applications. Like global thresholding, adaptive thresholding is used to separate desirable foreground image objects from the background based on the difference in pixel intensities of each region. Global thresholding uses a fixed threshold for all pixels in the image and therefore works only if the intensity histogram of the input image contains neatly separated peaks corresponding to the desired subjects and backgrounds. Hence, it cannot deal with images containing, for example, a strong illumination gradient. Local adaptive thresholding, on the other hand, selects an individual threshold for each pixel based on the range of intensity values in its local neighborhood. This allows for thresholding of an image whose global intensity histogram doesn't contain distinctive peaks.

OpenCV function adaptiveThreshold is applied for binarization. Through large amounts of expirements, we tried 15, 21, and 51 blockSize according to specific distance between cell phone and the paper which we want to scan. It is observed that a blockSize of 51 yields a good tradeoff between efficiency and thresholding effect.

2.2 De-skew

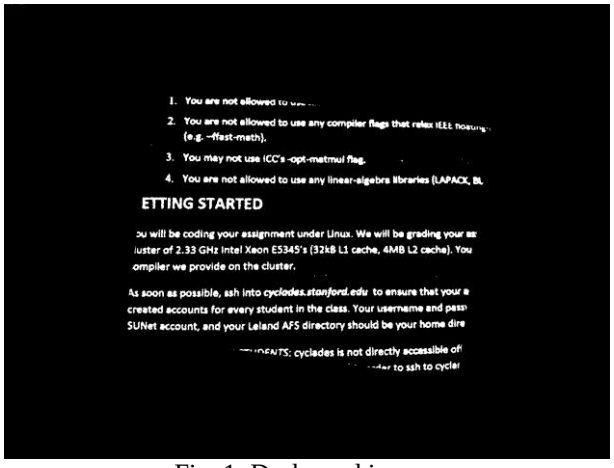


Fig. 1: Deskewed image.

We use the method described in Abuhaiba [2]. Which depends on finding a horizontal RLSA (Run-Length Smoothing Algorithm) image of the skewed document. The average skew of selected black connected components in the RLSA image is considered as the skew angle for the whole document. And then we use a single-pass skew detection and correction algorithum to de-skew the image.

However, after some experiment, we adapted another method. The input image from the user can be taken at an angle and the lines of text may not be horizontally aligned[4]. This may cause the following OCR stage to not perform as expected. Thus the image should be rectified first. Hough transform is used to detect (text) lines in the image. Then the binarized image is rotated by the mean of the rotation angles calculated from each text line. OpenCV function getRotationMatrix2D can be used to do it. In order to avoid cutting off corners in the rotation process, padding is introduced before rotating. See Fig. 1. The image after getRotationMatrix2D is larger than the previous images because of padding, but the padding will be chopped off as a last step to match the original image size. Observing that Tesseract does a decent job for images skewed by less than 5 degrees, we have included logic to bypass de-skew image rotation for such images. This improves accuracy because image rotation lowers image quality after interpolation. By skipping rotation, we essentially preserve more details in the image and therefore boost performance for images with a small skew. In our accuracy test in skew angles, 0-degree test cases have better performance than others. Even though theoretically 0-degree test cases should not be affected by image rotation if the skew is exactly 0 degree, we performed testing by hand-holding the device, which can introduce skew of a few degrees. Therefore skipping the rotation step can save us from the interpolation effect and boost performance.

2.3 De-noising

We also performed median filtering with a kernel size of 3 to eliminate any salt-and-pepper noise. The main idea of the median filter is to run through the signal entry by entry, replacing each entry with the median of neighboring entries. The pattern of neighbors is called the "window", which slides, entry by entry, over the entire signal. For 1D signals, the most obvious window is just the first few preceding and following entries, whereas for 2D (or higher-dimensional) signals such as images, more complex window patterns are possible (such as "box" or "cross" patterns). This improves the accuracy of the character size detection, because otherwise the size of noise spots will introduce error into character size calculation.

3. WORD RECOGNITION-OCR

We propose to use the Tesseract OCR [1] engine for this project, the latest version is 4.0.0, released on October 29, 2018. ln 2006, Tesseract was considered one of the most accurate open-source OCR engines then available. Tesseract 4 adds a new neural net (LSTM) based OCR engine which is focused on line recognition, but also still supports the legacy Tesseract OCR engine of Tesseract 3 which works by recognizing character patterns. It has many additional languages and scripts, bringing the total to 116 language, including right-to-left text such as Arabic or Hebrew, many Indic scripts as well as CJK quite well. Tesseract supports various output formats: plain text, hOCR (HTML), PDF, invisible-text-only PDF, TSV. The master branch also has experimental support for ALTO (XML) output.

Algorithm:

Step 1: the engine first detects component outlines, which are then grouped into blobs.

Step 2: Blobs are organized into lines and lines are analyzed for fixed pitch.

Step 3: The pitch is then used to separate out the words.

Step 4: During the first pass, each satisfactory word is entered into an adaptive classifier which will enable to the engine to more easily recognize the words further down the page.

Step 5: A second pass will be made to reanalyze the words at the beginning of the page since the classifier was not trained during the first pass.

Step6:A final phase resolves fuzzy spaces, and uses x-height normalization to detect lower case letters.

Tesseract's output will have very poor quality if the input images are not preprocessed to suit it: Images (especially screenshots) must be scaled up such that the text x-height is at least 20 pixels, low-frequency changes in brightness must be high-pass filtered, or Tesseract's binarization stage will destroy much of the page, and dark borders must be manually removed, or they will be misinterpreted as characters.

A. Segmentation by Letter

The first step of word recognition is to segment the image into rectangles each containing a letter. Because most letters consists of one connected component in the image, contours can be drawn (using OpenCV function findContours and drawContours) around each letter and then find a bounding box of each contour (OpenCV function boundingRect). Some letters (for instance, lower case "i", "j") consists of two connected components, and hence will have two bounding boxes, but it does not affect the algorithm since they will get combined in the next step as we combine letter bounding boxes into word bounding boxes.

B. Smart Segmentation by Word

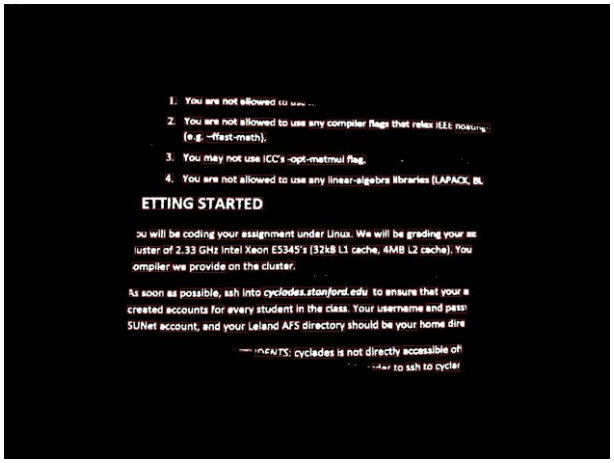


Fig. 2: Word's bounding box.

The motivation/assumption behind this method is that in a normal text document, line spacing, character size, word spacing, etc. scales linearly, i.e. large characters means large spacing, and vice versa [5]. We have to combine neighboring letter bounding boxes into a word bounding box, but there is no existing OpenCV functions that performs this task. Therefore the following functions are implemented in C++:

1) Find Character Size: We implemented function findCharSize to estimate the average height and width of each letter (or its bounding box). This is important because when deciding whether two letter-bounding boxes are neighbors, we need to look at the distance between them relative to the average size of bounding boxes. Making the decision based on absolute distance only is not accurate. The implementation was done by creating a map with keys being the areas of the bounding boxes, and the values being the occurrences (i.e. how many bounding boxes have areas equal to the key). Then the ten most frequent area values are selected, and their mean is used as the character size metric. The function calculates the values for a few important parameters including:
   - cArea: The mean calculated from the ten most frequent area values.
   - cHeight: An estimate of average height of letters, calculated from $\sqrt{cArea}$ multiplied by a constant factor to take into account that most letters have larger height than width.
   - cWidth: An estimate of average width of letters, calculated from $\sqrt{cArea}$ multiplied by a constant factor to take into account that most letters have larger width than height.
2) Find Neighbour: Function isNeighbour is implemented to determine whether two letters are next to each other in the same word. This is the key logic in determining which letter bounding boxes to merge together into a word bounding box. Two bounding boxes need to satisfy both conditions below in order to be called neighbours:

• The *x*-coordinate of the right edge of box 1 and the xcoordinate of the left edge of box 2 are off by at most 0.32 × cWidth (box 1 is to the left of box 2); or vice versa, the *x*-coordinate of the left edge of box 1 and the x-coordinate of the right edge of box 2 are off by at most 0.32 × cWidth (box 1 is to the right of box 2). The factor 0.45 is the parameter that gives the best results after several experiments.

• Because different letters have different heights, we decided to use the *y*-coordinate of the bottom edge to determine whether two boxes are on the same row. The *y*-coordinates of the bottom edges of the two boxes needs to be within 0.32 × cHeight of each other.

• In addition, we also want to combine the dot in lower case "i" and "j", therefore we allow the case where the *y*-coordinates of the top edges are within 0.28×cHeight. Segmentation By word factors are tuned so that the algorithm works for the majority of test cases. Take Fig. 2 for example, we draw a red bounding box through every word.

C. Search and Label Matches

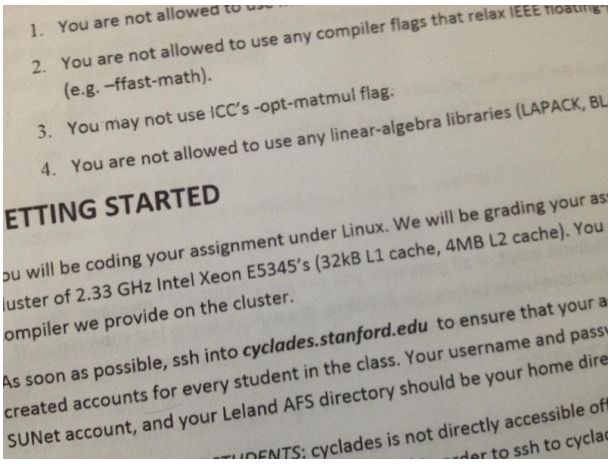


Fig. 3(a): Original image take from cell phone.

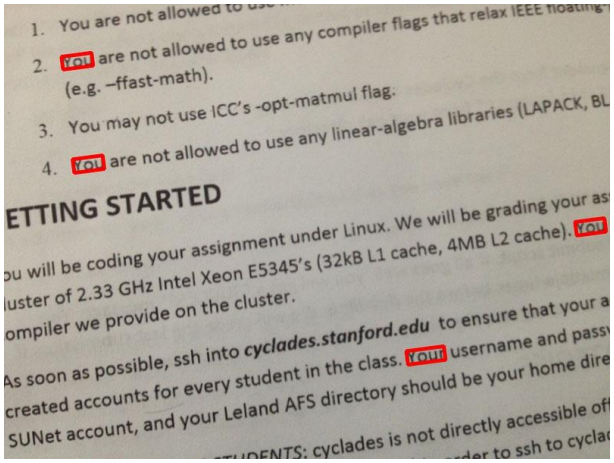


Fig. 3(b): Rotate back after recognizing and drawing bounding box where words we want at.

At last, bounding boxes (hopefully each containing exactly one word at this point) can be passed to the Tesseract OCR engine. See Fig. 3(b) for example. In order to improve efficiency, function withinRange is implemented to filter out boxes too wide or too narrow given the keyword length. Sometimes the image is out of focus or blurry due to vibration, therefore the result from Tesseract is not accurate. To cope with the imperfection, we label exact matches with red rectangles, non-exact matches with blue or green rectangles. Exact or non-exact matches are determined by the ratio between edit distance (or Levenshtein distance) and the length of the keyword. A ratio of exact zero (or edit distance equal to zero) means that there is an exact match. A ratio reasonably close to zero means that we have a non-exact match. Levenshtein distance is defined to be the minimum number of single-letter operations (insertion, deletion, or substitution) needed to transform one string into another. For example, to change "abcd" into "bcda", one can either change each letter (change the first letter "a" to "b", the second letter "b" to "c", the third letter "c" to "d", the fourth letter "d" to "a"), which has a total of four operations, or delete the "a" from the beginning of the string and add an "a" to the end of it, which has a total of two operations. Therefore the Levenshtein distance between the two strings is two. The distance can be calculated between any two strings using dynamic programming. Finally, we draw rectangular boxes of different colors (in order to differentiate between exact matches and close matches) at the coordinates where we find matches, and then overlay the rectangles onto the original image.

D. SERVER-CLIENT CONNECTION

OCR is an extremely computationally intensive operation. To perform this operation using the processors on an outdated smartphone would be impractical. Therefore, we decided to offload OCR and all other image processing operations onto the servers we configure. The server side is a PHP script that waits for an http connection. Once the connection is established, the PHP script receives the image uploaded by the phone along with other parameters and stores them on the afs disk. The server side needs to run OCR and OpenCV. Therefore, we configure OCR and OpenCV libraries, and then we implemented a backend Python script that is responsible for linking up with the PHP script and executing the OpenCV/OCR executable on our own server. Once the processing is complete, the PHP script picks up the output image and streams it back to the phone. Figure **3(c)** illustrates the connection from client side/mobile device to server side.

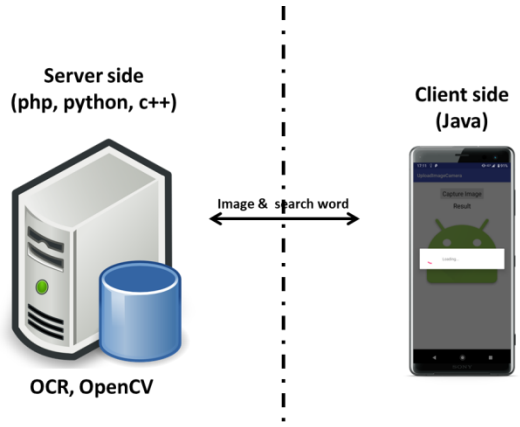


Fig. 3(c): Client-server connection.

On the client side, app takes a picture of the text when the user press the capture image button See Fig. 3(d). After the image was saved, It then sends the image to the server for processing. After the image is received from the server, the phone will display it on the screen until the user hits the back button to take a new picture. User could also clear the screen by hitting the clear button to take another picture. The whole process may take a few minutes, depends on your hardware device and network connection.

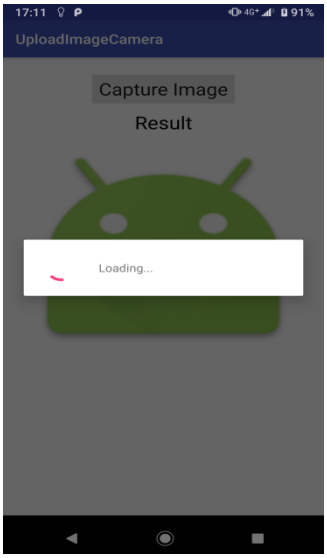
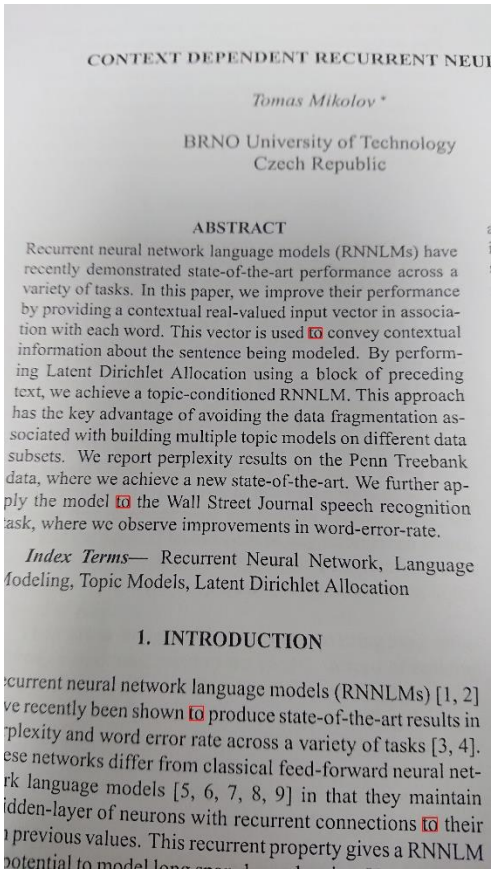


Fig. 3(d): User interface of App

5. EXPERIMENT RESULTS

Fig. 4(a): Server waits for image.

Fig. 4(b): Cell phone APP takes picture and uploads to Server and then waits for the result.



CONTEXT DEPENDENT RECURRENT NEUI

Tomas Mikolov *

BRNO University of Technology
Czech Republic

ABSTRACT

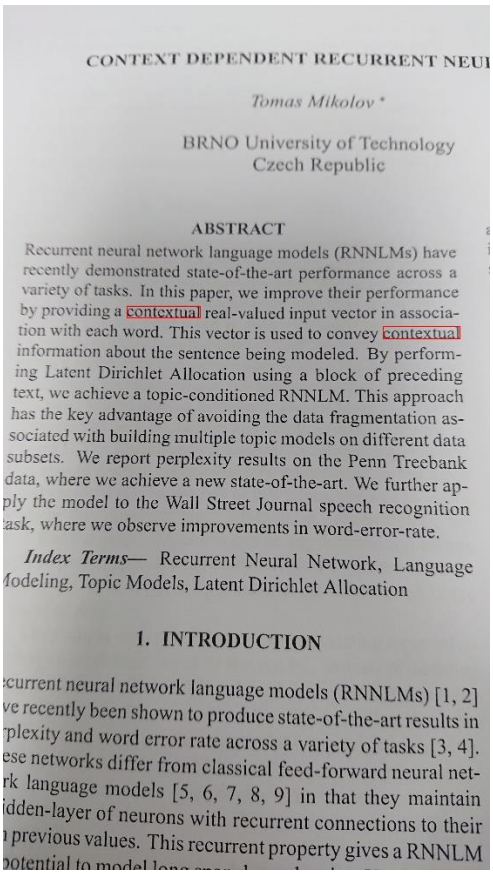
Recurrent neural network language models (RNNLMs) have recently demonstrated state-of-the-art performance across a variety of tasks. In this paper, we improve their performance by providing a contextual real-valued input vector in association with each word. This vector is used to convey contextual information about the sentence being modeled. By performing Latent Dirichlet Allocation using a block of preceding text, we achieve a topic-conditioned RNNLM. This approach has the key advantage of avoiding the data fragmentation associated with building multiple topic models on different data subsets. We report perplexity results on the Penn Treebank data, where we achieve a new state-of-the-art. We further apply the model to the Wall Street Journal speech recognition task, where we observe improvements in word-error-rate.

Index Terms— Recurrent Neural Network, Language Modeling, Topic Models, Latent Dirichlet Allocation

1. INTRODUCTION

current neural network language models (RNNLMs) [1, 2] ve recently been shown to produce state-of-the-art results in plexity and word error rate across a variety of tasks [3, 4]. ese networks differ from classical feed-forward neural net- rk language models [5, 6, 7, 8, 9] in that they maintain idden-layer of neurons with recurrent connections to their previous values. This recurrent property gives a RNNLM

Fig. 4(c): The result finding the word “to”.



CONTEXT DEPENDENT RECURRENT NEUI

Tomas Mikolov *

BRNO University of Technology
Czech Republic

ABSTRACT

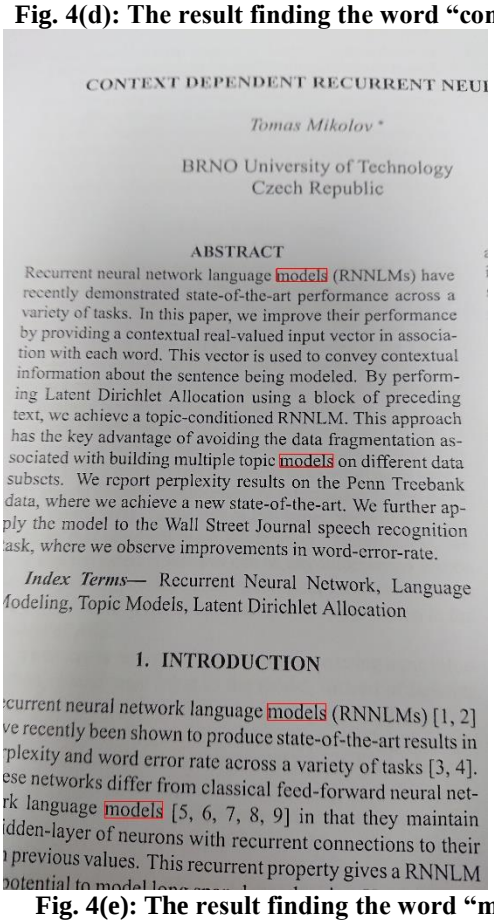
Recurrent neural network language models (RNNLMs) have recently demonstrated state-of-the-art performance across a variety of tasks. In this paper, we improve their performance by providing a contextual real-valued input vector in association with each word. This vector is used to convey contextual information about the sentence being modeled. By performing Latent Dirichlet Allocation using a block of preceding text, we achieve a topic-conditioned RNNLM. This approach has the key advantage of avoiding the data fragmentation associated with building multiple topic models on different data subsets. We report perplexity results on the Penn Treebank data, where we achieve a new state-of-the-art. We further apply the model to the Wall Street Journal speech recognition task, where we observe improvements in word-error-rate.

Index Terms— Recurrent Neural Network, Language Modeling, Topic Models, Latent Dirichlet Allocation

1. INTRODUCTION

current neural network language models (RNNLMs) [1, 2] ve recently been shown to produce state-of-the-art results in plexity and word error rate across a variety of tasks [3, 4]. ese networks differ from classical feed-forward neural net- rk language models [5, 6, 7, 8, 9] in that they maintain idden-layer of neurons with recurrent connections to their previous values. This recurrent property gives a RNNLM

Fig. 4(d): The result finding the word “contextual”.



CONTEXT DEPENDENT RECURRENT NEUI

Tomas Mikolov *

BRNO University of Technology
Czech Republic

ABSTRACT

Recurrent neural network language models (RNNLMs) have recently demonstrated state-of-the-art performance across a variety of tasks. In this paper, we improve their performance by providing a contextual real-valued input vector in association with each word. This vector is used to convey contextual information about the sentence being modeled. By performing Latent Dirichlet Allocation using a block of preceding text, we achieve a topic-conditioned RNNLM. This approach has the key advantage of avoiding the data fragmentation associated with building multiple topic models on different data subsets. We report perplexity results on the Penn Treebank data, where we achieve a new state-of-the-art. We further apply the model to the Wall Street Journal speech recognition task, where we observe improvements in word-error-rate.

Index Terms— Recurrent Neural Network, Language Modeling, Topic Models, Latent Dirichlet Allocation

1. INTRODUCTION

current neural network language models (RNNLMs) [1, 2] ve recently been shown to produce state-of-the-art results in plexity and word error rate across a variety of tasks [3, 4]. ese networks differ from classical feed-forward neural net- rk language models [5, 6, 7, 8, 9] in that they maintain idden-layer of neurons with recurrent connections to their previous values. This recurrent property gives a RNNLM

Fig. 4(e): The result finding the word “models”.

First, we use python3 to run up the server monitor. Server monitor will await our APP's connection. See Fig. 4(a). App's connection will be established after we take the photo. See Fig. 4(b). The APP will then upload the photo to the server and wait for the result returning from the server. Fig. 4(c), Fig. 4(d), and Fig. 4(e) are different experiment finding the word "to", "contextual", and "models". We can observe that the word we want to find is surrounded by red bounding box. Furthermore, We tested the keyword recognition rate with more than 100 data points. The sample space covers different skew angles (0 degree, 15 degrees and 30 degrees), and different word lengths, because we believe that those are the two main factors that can affect performance. The overall accuracy is 89.7%, and we have included a breakdown by word length and skew angle, See Fig. 4(f) and Fig. 4(g).

From the breakdown by keyword length, we can observe that although accuracy varies for different word lengths, there is no clear trend that performance deteriorates as keyword length increases/decreases, which is desirable. Variation does exist but it is largely due to the relatively small sample size. We would like to perform more testing if time permits.

From the breakdown by skew angle, notice that there is a slight deterioration in performance due to the skew. We have investigated this by looking at the intermediate steps/results on the algorithm and concluded that the de-skew algorithm works reasonably well, the deterioration is mostly due to the fact that image rotation lowers the image quality. We also noticed that there is very little change in performance from 15-degree skew to 30-degree skew, which is desirable.

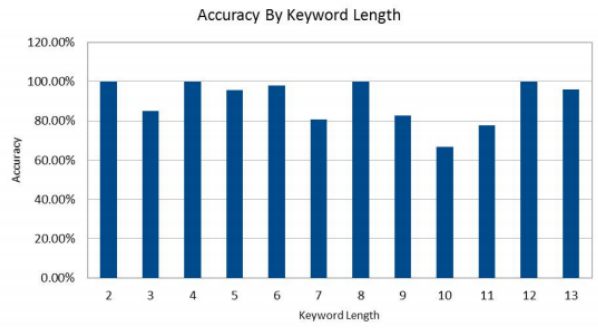
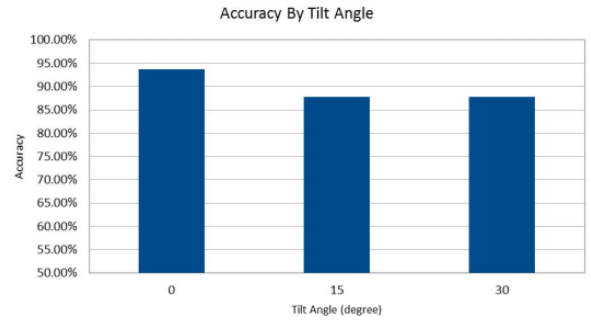


Fig. 4(f)



See Fig. 4(g)

By examining the test input and output images, we noticed that results may vary due to uneven line spacing, different fonts, and image quality. De-focus and blur due to hand shaking account for a large proportion of inaccuracies.

6. CONCLUSION

We successfully find the word we want without human finding. If the situation is that we have large amounts of paper and we want to search keyword in these paper. We tried a variety of methods and techniques to successfully identify a word from a corpus of words within an image. With local thresholding, deskewing and smart segmentation, we were able to successfully segment each word in a majority of test cases. Both preprocessing and postprocessing (editdistance based match analysis) were tailored to the powerful Tesseract OCR. Multi-platform (client, server) and multilanguage(java, C++, python, php) were introduced in this project to bypass numerous configuration difficulties. For the purpose of demonstration, we were able to accurately query any word using an Android phone. Ideally, our algorithm runs fast enough to support real-time word search. But due to the client-server transport delay, realtime scan-mode is impractical to use. To solve this problem, we could either move the computation to the client side, or use other feature detection (i.e. SURF) technique to reuse the previous detection results. Since consecutive frames can have large overlapping areas, we can increase efficiency by avoid re-computing those overlapping areas. Also, to improve user experience and make our app interesting, here are some future work to try. First Spell-Check Backed by a Lexicon**:** Since the correctness of words detected by our algorithm may be influenced by various factors, such as skewed view, unbalanced illumination, the spell-checking will also be included in our project. The basic idea is to improve OCR accuracy by constraining the word by a lexicon – a list of "legal" words that are allowed to occur in a document. The available lexicon such as Hunspell dictionary and UNIX shared dictionary will be helpful in our projects. Also, by leveraging the technique of dynamical programming to find the edit distance, spell-check could be done based on edit distance analysis. Second Word translator: Once the particular word is circled out, we could use Google Translate API to translate the word.

7. REFERENCES

[1] Akhil S, "An overview of Tesseract OCR Engine", A Seminar Report, 2016

[2] Ibrahim S. I. Abuhaiba, "Skew Correction of Textural Documents", J. King Saud Univ., Vol. 15, Comp. & Info. Sci., pp. 67-86 (A.H. 1423/2003)

[3] J. Matas, O. Chum, M. Urban, T. Pajdla, Robust Wide Baseline Stereo from Maximally Stable Extremal Regions
[4] Sam S. Tsai, Huizhong Chen, David Chen, Ramakrishna Vedantham, Radek Grzeszczuk and Bernd Girod, Mobile Visual Search Using Image and Text Features
[5] Vinay Raj Hampapur, Tahrina Rumu, Umit Yoruk, Keyword Guided Word Spotting In Printed Text