PureOCR

github.com/michael-s-teodor/pureocr

Mihnea S. Teodorescu & Moe Assaf
michael.s.teodor@gmail.com & mormoassaf@gmail.com

December 18, 2019

1 Problem description

A program is required that can learn to scan and process handwritten (or typed) text, recognising its characters and outputting them as plain text. Every ASCII character from 0x21 to 0x7E should be identifiable. This includes digits, lower and upper case letters and punctuation marks, adding up to a total of 94 characters.

2 Problem analysis

We interpret the problem as follows: A character exists in a given area of an image if and only if that area contains a cluster of non-white pixels. These areas can be easily identified recursively by an algorithm which maps each candidate cluster and sends it to the recognition stage. This interpretation is motivated by the following statements:

- Identifying candidate areas first provides a significant improvement in time complexity. This way, text recognition will not have to be conducted across the whole image, but rather in certain areas only.
- The problem can now be further simplified by applying a noise-removing filter function first. This reduces the number of required recursive steps exponentially.

Having mapped each cluster of pixels, it is now time to recognise each character. For this, we have set up a neural network of 400 inputs (for segments of 20×20 pixels) and 94 outputs – one per character. The network will be trained in accordance with each font to be recognised.

3 Design

In order to achieve the desired results, we have structured our algorithm into several stages:

- Pixel-continuity-oriented preprocessing approach for identifying letters and characters: our program goes through the image line by line and when we encounter a pixel we call a recursive functions which maps the character by going all directions and comparing the threshold value. Then we set the anchors of the letter and store them.
- Line recognition: After mapping all the characters we use a sorting algorithm to sort them by their y-axis coordinates. After that, we compare their coordinates consequently being able to recognise identify lines then we sort them by their x-coordinates using WHAM-sort and now we have achieved the right order of all letters.

- WHAM-sort: This sorting algorithm was developed by Wim Hesselink and Arnold Meijster, two researchers at the University of Groningen. Briefly, the algorithm can be described in four steps. Let's consider the following array of 'unsorted, but not-so-unsorted,' (as described by one of the authors) characters:

$$6, 13, 14, 12, 17, 42, 51, 3, 64, 12, 8, 93, 95, 96, 97$$

  1. Find longest ascending prefix: $6, 13, 14$
  2. Recursively sort the next segment with the same length as the prefix (in this example, the recursive step is easy, since this segmented is already sorted): $12, 17, 42$
  3. Merge the sorted segments (like merge-sort): $6, 12, 12, 14, 17, 42$
  4. Repeat the process (now for length 6, etc.)

  Similarly to merge-sort, WHAM-sort is also in $\mathcal{O}(n \log n)$. Why? Well, assume that worst-case the input is an alternating up-down sequence. Then WHAM-sort is actually a bottom-up merge-sort. Still WHAM-sort can be significantly faster than merge-sort. Merge-sort performs the same number of operations for any input of length n. The number of operations that WHAM-sort performs is dependent on the (number of inversions) in the input. The best case is actually the sorted input, on which no operations are performed at all.

- Merging neighbouring characters: There are occasions where we have letters like the letter i which has a dot and a line. This is one of the issues that we had because the dot or the line should not be recognised as one letter. This applies for question marks, semicolons, etc. We managed to fix this after implementing the line recognition because at allows us to compare their coordinates and merge the together.

- Neural network: we have the written a code for the neural network and it is able to feed forward the data and produce an output. However, we have not implemented the training algorithm yet.

- Postprocessing: Ultimately, the user will be provided with the option to apply a dictionary-based postprocessing algorithm. This aims to correct misspelled words, as well as incorrectly-recognised characters – should there be any.

4 Program code

main.py

```
#### Libraries
# Own libraries
from preprocessor import Preprocessor
from recogniser import Recogniser
from network import Network

#### Main
img = Preprocessor("samples/2.jpg")
recogniser = Recogniser()
img.process_image()
img.convert_to_binary()
img.preview_image()
```

preprocessor.py

```
#### Libraries
# Own libraries
from wham_sort import Wham_Sort
# Third-party libraries
import cv2

#### Class
class Preprocessor():

    def __init__(self, file):

        # Import a sample image also 0 is for grey scale and store its properties
        self.file = file
        self.original_img = cv2.imread(file, 0)
        self.processed_img = cv2.imread(file, 0)
        self.img = cv2.imread(file, 0)

        self.height, self.width = self.img.shape[0], self.img.shape[1]
        self.threshold = -1

        self.char_anchors = [] # [x_min, y_min, x_max, y_max]
        self.char_anchors_rows = [] # [x_min, y_min, x_max, y_max]
        self.data = [] # Contains copies of charachters row/char

    def process_image(self):
        self.threshold = get_threshold(self.height, self.width, self.img)
        self.char_anchors = find_letters(self.height, self.width, self.threshold, self.img
            )
        self.char_anchors = wham_sort(self.char_anchors, 1)
        self.char_anchors_rows = recognise_lines(self.char_anchors)
        self.char_anchors_rows = sort_letters_in_order(self.char_anchors_rows)
        self.char_anchors_rows = merge_neighbours(self.char_anchors_rows)

    def preview_image(self):
        draw_boxes_lines(self.char_anchors_rows, self.processed_img)
        while True:
            cv2.imshow("Preview (Press q to close)", self.processed_img)
            key = cv2.waitKey(0) & 0xFF

            # If the q key was pressed, break from the loop
            if key == ord("q"):
                break
            cv2.destroyAllWindows()

    def convert_to_binary(self):
        if (self.threshold == -1):
            self.threshold = get_threshold(self.height, self.width, self.img)
        self.processed_img = convert_to_binary_img(self.height, self.width, self.threshold
            , self.processed_img)
        return self.processed_img

#### Functions
def get_threshold(height, width, img):
    avg = 0
    for i in range(height):
        for j in range(width):
           avg += img[i][j]
```

```
    avg /= height*width
    return 0.75*avg

def convert_to_binary_img(height, width, threshold, img):
    for i in range(height):
        for j in range(width):
            if( img[i][j] > threshold):
                img[i][j] = 255
            else:
                img[i][j] = 0
    return img

def find_letters(height, width, threshold, img):
    anchors = []
    for i in range(height):
        for j in range(width):
            if (img[i][j] < threshold):

                # Map letter
                temp_arr = [j, i, j, i]
                temp_arr = map_letter(j, i, temp_arr, threshold, height, width, img)

                # Store anchors
                anchors.append(temp_arr)
    return anchors

def map_letter(x, y, temp_arr, threshold, height, width, img):
    if (x < 0 or x >= width or y < 0 or y >= height):
        return temp_arr

    if (img[y][x] < threshold):

        # Delete so that we dont encounter it again
        img[y][x] = -1

        # Find min x and y and max x and y
        if (x < temp_arr[0]):
            temp_arr[0] = x
        if (y < temp_arr[1]):
            temp_arr[1] = y
        if (x > temp_arr[2]):
            temp_arr[2] = x
        if (y > temp_arr[3]):
            temp_arr[3] = y

        for n in range(-1,2):
            for m in range(-1,2):
                if not (n == 0 and m == 0):
                    temp_arr = map_letter(x+n, y+m, temp_arr, threshold, height, width, img
                        )

    return temp_arr

def draw_boxes_lines(anchors_rows, img):

    # Draw boxes around letters in rows with the same colour and alternate
    for row in range(len(anchors_rows)):
        img = draw_boxes(anchors_rows[row], row%2*100,img)
```

```
113
114 def draw_boxes(arr, colour, img):
115     for letter in range(len(arr)):
116         min_x = arr[letter][0]
117         min_y = arr[letter][1]
118         max_x = arr[letter][2]
119         max_y = arr[letter][3]
120         for i in range(min_x, max_x+1):
121             img[min_y][i] = colour
122             img[max_y][i] = colour
123         for j in range(min_y, max_y+1):
124             img[j][min_x] = colour
125             img[j][max_x] = colour
126     return img
127
128 def recognise_lines(char_anchors):
129     char_anchors_rows = []
130     row = 0
131     total = (char_anchors[0][1] + char_anchors[0][3])/2
132     n = 1
133     char_anchors_rows.append([])
134
135     deviation = 20 # NOTE: needs to be adjusted so that its adaptive
136
137     for i in range(len(char_anchors)):
138         avg = total/n
139
140         # Calculate middle y
141         y = (char_anchors[i][1] + char_anchors[i][3])/2
142
143         if (y < avg + deviation and y > avg - deviation):
144             total += y
145             char_anchors_rows[row].append(char_anchors[i])
146         else:
147             row += 1
148             n = 0
149             total = y
150             char_anchors_rows.append([])
151             char_anchors_rows[row].append(char_anchors[i])
152         n+=1
153     return char_anchors_rows
154
155 def sort_letters_in_order(char_anchors_rows):
156     for row in range(len(char_anchors_rows)):
157
158         # Sort by x
159         char_anchors_rows[row] = wham_sort(char_anchors_rows[row], 0)
160     return char_anchors_rows
161
162 def merge_neighbours(char_anchors_rows):
163     for row in range(len(char_anchors_rows)):
164         length = len(char_anchors_rows[row])-1
165         for char in range(length):
166             if (char+1 < length):
167                 c1 = char_anchors_rows[row][char]
168                 c2 = char_anchors_rows[row][char+1]
169
170                 # Check if theyre on top of each other
```

```
                if (c1[1] >= c2[3] or c1[3] <= c2[1]):

                    # Check if their x's intersect
                    if( c2[0] <= c1[2] ):
                        char_anchors_rows[row][char][0] = min(c1[0],c2[0])
                        char_anchors_rows[row][char][1] = min(c1[1],c2[1])
                        char_anchors_rows[row][char][2] = max(c1[2],c2[2])
                        char_anchors_rows[row][char][3] = max(c1[3],c2[3])
                        char_anchors_rows[row].remove(c2)
                        length -=1
    return char_anchors_rows
```

network.py

```
#### Libraries
# Third-party libraries
import numpy as np

#### Class
class Network(object):

    def __init__(self, sizes):

        # sizes = number of neurons per layer [input, hidden, ..., hidden, output]
        self.num_layers = len(sizes)
        self.sizes = sizes

        # Each neuron has one bias
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]

        # Each neuron from layer n has sizes[n-1] weights
        self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes[1:])]

    def feedForward(self, a):

        # This function returns the output of the network when a is applied
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w,a) + b)
        return a

#### Helper functions
def sigmoid(z):
    return 1.0/(1.0 + np.exp(-z))

def sigmoid_derivative(z):
    return np.exp(z) / (np.exp(z) + 1)**2
```

recogniser.py

```
#### Libraries
# Own libraries
from network import Network

#### Class
net = Network([10000, 10, 10, 94])

class Recogniser():

    def __init__(self):
        pass

    def retrieve_char(self, img):

        # Compute the output of the neural network
        output = net.feed_forward(img)

        # Determine the most likely outcome and return its ASCII value
        max = 0
        char = 0
        for i in range(94):
            if (max < output[i]):
                max = output[i]
                char = i
        return chr(char + 33)

    def return_texts(self, data):
        texts = []
        for img in range(len(data)):
            texts.append(self.retrieve_char(img))
        return texts
```

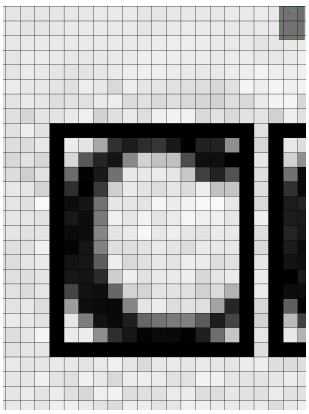
wham_sort.py

```
#### Class
class Wham_Sort():

    def wham_sort(length, arr):

        # Break recursive step
        if (length <= 1):
            return

        prefix = [None] * length
        suffix = [None] * length

        # Get prefix length
        p = 1
        while (p < length):
            if (arr[p-1] > arr[p]):
                break
            p += 1

        while (length - p):
```

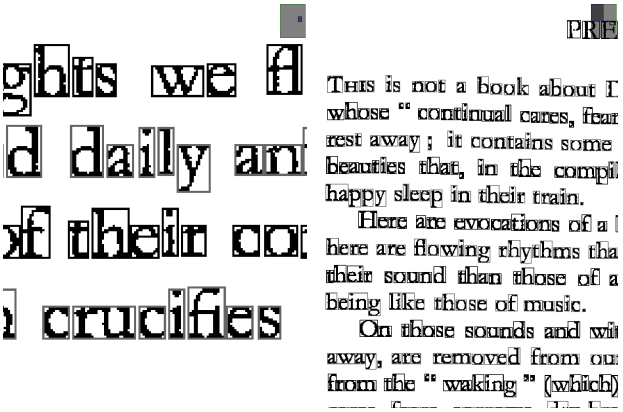
```
21
22              # Get prefix
23              prefix[0] = arr[0]
24              p = 1
25              while (p < length):
26                  if (arr[p-1] <= arr[p]):
27                      prefix[p] = arr[p]
28                  else:
29                      break
30                  p += 1
31
32              # Get suffix
33              s = 0
34              while (s < p):
35                  if (p+s >= length):
36                      break
37                  suffix[s] = arr[p+s]
38                  s += 1
39
40              # Next recursive step
41              wham_sort(s, suffix)
42
43              # Merge prefix and (now sorted) suffix into arr
44              l = p
45              r = s
46              p += s
47              idx = 0
48              left = 0
49              right = 0
50              while (left < l and right < r):
51                  if (prefix[left] < suffix[right]):
52                      arr[idx] = prefix[left]
53                      left += 1
54                  else:
55                      arr[idx] = suffix[right]
56                      right += 1
57                  idx += 1
58              if (left >= l):
59                  while (right < r):
60                      arr[idx] = suffix[right]
61                      idx += 1
62                      right += 1
63              else:
64                  while (left < l):
65                      arr[idx] = prefix[left]
66                      idx += 1
67                      left += 1
```

5 Test results

- This image shows how the threshold value is responsible for determining whether a pixel is part of a character or not.



- Here the image shows how the outcome turned out to be. As it shows in the picture, we draw a box around each character and it seems to be working perfectly. The image also shows how we alternate between grey and black to show how lines are being recognised. It also shows how the i's are merged together and this also applies for question marks, semicolon, question marks, etc.



6 Evaluation

- Threshold value: the way we determine the threshold value in our code depends on the average intensity of all pixels and we take a percentage out of that. However, in certain circumstances this can be quite deterministic and ambiguous. Thus, we need to implement a new way of find the threshold value and possibly make it dynamic in some parts of the image where there might be some shades which could cause the program to fail.

- Merged letters: there will be occasions where the letters will be connected together. Consequently, the preprocessor program will mark it as one single character making it hard for the neural network to understand causing an error. When the training part is implemented, the neural network will be trained enough to recognise merged letters and apply a different approach to those specific merged letters.

- Noise: there happens to be random chunks of letters which are not connected to the actual letter itself causing them to be recognised as characters in the preprocessor. We need to find a way of recognising them as errors and simply return nothing.
- Shades: shades can really mess up the threshold value and even cause the recursive function to reach its maximum depth due to python having a limit. We need to find a good procedure of either removing those shades from the image completely or making the threshold dynamic. Otherwise, the program would recognise shades as letters or just crash due to the recursion depth limit.
- Orientation and dimensions: we have not implemented any sort of procedure to the change the dimensions of the image so that texts can be aligned properly for the preprocessor to analyse.