PureOCR

github.com/michael-s-teodor/pureocr

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

December 14, 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 most 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 10,000 inputs (for segments of 100 × 100 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 and now we have achieved the right order of all letters.

- 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

```
from preprocessor import PreProcessor
from recogniser import Recogniser
from network import Network


img = PreProcessor("samples/5.jpg")
recogniser = Recogniser()
#======================================#
img.processImage()
img.convertToBinary()
img.previewImg()
#======================================#
```

preprocessor.py

```

import cv2 #import the image processing

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.charAnchors = [] # [x_min,y_min,x_max,y_max]
        self.charAnchorsRows = [] # [x_min,y_min,x_max,y_max]
        self.data = [] #contains copies of charachters row/char

    def processImage(self):
        self.threshold = getThreshold(self.height,self.width,self.img)
        self.charAnchors = findLetters(self.height,self.width,self.threshold,self.img)
        self.charAnchors = bubbleSort(self.charAnchors,1)
        self.charAnchorsRows = recogniseLines(self.charAnchors)
        self.charAnchorsRows = sortLettersInOrder(self.charAnchorsRows)
        self.charAnchorsRows = mergeNeighbours(self.charAnchorsRows)

    def previewImg(self):
        drawBoxesLines(self.charAnchorsRows,self.processed_img)
        while True:
            cv2.imshow("Preview (Press c to close)", self.processed_img)
            key = cv2.waitKey(0) & 0xFF
            # if the q key was pressed, break from the loop
            if key == ord("c"):
                break
            cv2.destroyAllWindows()

    def convertToBinary(self):
        if(self.threshold == -1):
            self.threshold = getThreshold(self.height,self.width,self.img)
        self.processed_img = convertToBinaryImg(self.height,self.width,self.threshold,self.processed_img)
        return self.processed_img

#==================================================================================================

def getThreshold(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 convertToBinaryImg(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 findLetters(height,width,threshold,img):
    anchors = []
    for i in range(height):
        for j in range(width):
            if( img[i][j] < threshold ):
                #map letter
                tempArr = [j,i,j,i]
                tempArr = mapLetter(j,i,tempArr,threshold,height,width,img)
                #store anchors
                anchors.append(tempArr)
    return anchors

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

    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 < tempArr[0] ):
            tempArr[0] = x
        if( y < tempArr[1] ):
            tempArr[1] = y
        if( x > tempArr[2] ):
            tempArr[2] = x
        if( y > tempArr[3] ):
            tempArr[3] = y

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

    return tempArr

def drawBoxesLines(anchorsRows,img):
    #draw boxes around letters in rows with the same colour and alternate
    for row in range(len(anchorsRows)):
        img = drawBoxes(anchorsRows[row], row%2*100,img)

def drawBoxes(arr,colour,img):
    for letter in range(len(arr)):
        minX = arr[letter][0]
        minY = arr[letter][1]
        maxX = arr[letter][2]
        maxY = arr[letter][3]
        for i in range(minX,maxX+1):
            img[minY][i] = colour
            img[maxY][i] = colour
        for j in range(minY,maxY+1):
            img[j][minX] = colour
            img[j][maxX] = colour
    return img

def recogniseLines(charAnchors):
    charAnchorsRows = []
    row = 0
    total = (charAnchors[0][1]+charAnchors[0][3])/2
    n = 1
    charAnchorsRows.append([])

    deviation = 20 #NOTE: needs to be adjusted so that its adaptive

    for i in range(len(charAnchors)):
        avg = total/n

        #calculate middle y
        y = (charAnchors[i][1] + charAnchors[i][3])/2

        if( y < avg + deviation and y > avg - deviation):
            total += y
            charAnchorsRows[row].append(charAnchors[i])
        else:
            row += 1
            n = 0
            total = y
            charAnchorsRows.append([])
            charAnchorsRows[row].append(charAnchors[i])

        n+=1
    return charAnchorsRows

def bubbleSort(arr,by):
    length = len(arr)
    for i in range(length-1):
        for j in range(length-1):
            if( arr[j][by] > arr[j+1][by] ):
                temp = arr[j+1]
                arr[j+1] = arr[j]
                arr[j] = temp
    return arr

def sortLettersInOrder(charAnchorsRows):
    for row in range(len(charAnchorsRows)):
        #Sort by x
        charAnchorsRows[row] = bubbleSort(charAnchorsRows[row], 0)
    return charAnchorsRows

def mergeNeighbours(charAnchorsRows):

    for row in range(len(charAnchorsRows)):
        length = len(charAnchorsRows[row])-1
        for char in range(length):
            if(char + 1 < length ):
                c1 = charAnchorsRows[row][char]
                c2 = charAnchorsRows[row][char+1]
                #check if theyre on top of each other
                if( c1[1] >= c2[3] or c1[3] <= c2[1] ):
                    #check if their x's are intersecting
                    if( c2[0] <= c1[2] ):

                        charAnchorsRows[row][char][0] = min(c1[0],c2[0])
                        charAnchorsRows[row][char][1] = min(c1[1],c2[1])
                        charAnchorsRows[row][char][2] = max(c1[2],c2[2])
                        charAnchorsRows[row][char][3] = max(c1[3],c2[3])

                        charAnchorsRows[row].remove(c2)
                        length -=1

    return charAnchorsRows
```

network.py

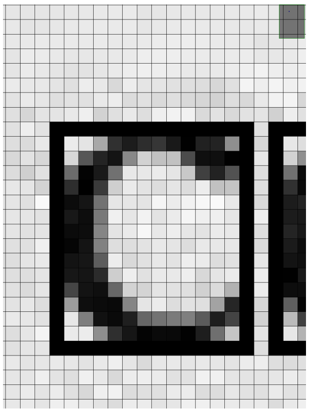
```python
#### Libraries
# Third-party library
import numpy as np

class Network(object):

    def __init__(self, sizes):
        # sizes = number of neurons per layer [input, hidden, ..., hidden, output]
        self.numLayers = len(sizes)
        self.sizes = sizes

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

        # Each neuron from layer n has sizes[n-1] weights
        # layer -> neuron -> random 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

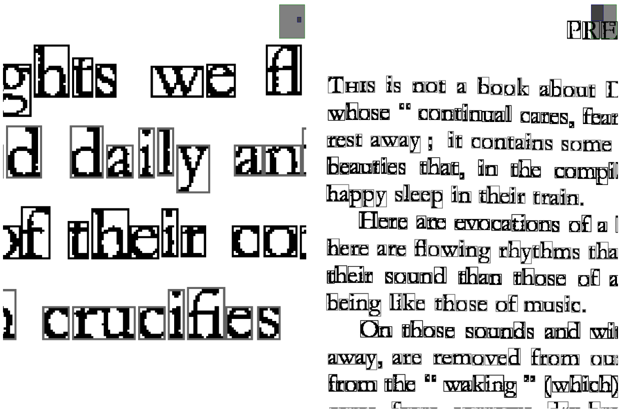
```python
from network import Network

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

class Recogniser():

    def __init__(self):
        pass

    def retrieveChar(self,img):
        output = net.feedForward(img)

        max = 0
        char = 0
        for i in range(94):
            if (max < output[i]):
                max = output[i]
                char = i
        return chr(char + 33 )

    def returnTexts(self,data):
        texts = []
        for img in range(len(data)):
            texts.append(self.retrieveChar(img))
        return texts


    # Compute the output of the neural network

    # Determine the most likely outcome and return its ASCII value
```

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.



ghts we fl
d daily an
f their co
crucifies

PREF

THIS is not a book about T
whose " continual cares, fear
rest away ; it contains some
beauties that, in the compil
happy sleep in their train.
Here are evocations of a
here are flowing rhythms tha
their sound than those of a
being like those of music.
On those sounds and wi
away, are removed from ou
from the " waking " (which)

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.