**Name- Shivansh Nautiyal**

**Prn-2107126086**

**Introduction**

Image segmentation is a vital task in computer vision, involving the division of an image into meaningful parts. It's crucial for object detection, image understanding, and quantitative analysis in various fields like autonomous driving, medical imaging, and quality control. By breaking down complex scenes, segmentation simplifies image analysis and facilitates precise measurements, making it an indispensable tool in computer vision applications.

**Propose Image Segmentation Project**

The objective of this project is to develop an image segmentation solution capable of accurately identifying and segmenting different braille characters within a braille sheet. The segmentation system should effectively differentiate between individual braille characters and accurately outline their boundaries within the image. The solution aims to assist in the digitization and interpretation of braille documents, thereby enhancing accessibility for individuals with visual impairments and facilitating automated braille recognition systems. The project will involve dataset collection, preprocessing, model selection, training, evaluation, and potential post-processing techniques to achieve robust and accurate segmentation of braille characters.

**A close up of a text

Description automatically generated A close up of a number

Description automatically generated**

**Braille Sheet Annotated Braille Sheet**

**Literature Review**

Traditional optical braille segmentation techniques typically rely on image processing algorithms to detect and segment braille characters from braille sheets. Here are some common techniques used in traditional optical braille segmentation:

**1. Thresholding -**

Thresholding techniques are used to separate braille characters from the background by setting a threshold value. Pixels with intensities above the threshold are considered part of the characters, while those below are considered background.

**2. Connected Component Analysis (CCA):**

CCA identifies connected regions in a binary image (typically obtained after thresholding) and labels them as individual components. In the context of braille segmentation, each braille character would ideally form a connected component.

**3. Edge Detection:**

Edge detection algorithms like Canny edge detector or Sobel operator can be employed to detect the edges of braille characters. Once edges are detected, contour analysis can be applied to extract individual characters.

**4. Morphological Operations:**

Morphological operations such as erosion and dilation can be used to refine the segmentation results. Erosion can help in removing noise and thinning characters, while dilation can help in closing gaps between characters.

**5. Template Matching:**

Template matching involves comparing a template of a braille character with different parts of the image to identify matches. If a match is found, the corresponding region can be segmented as a braille character.

**6. Feature-based Methods:**

Feature-based methods extract specific characteristics (e.g., size, shape, texture) of braille characters and use these features to segment them from the background.

**7. Machine Learning Approaches:**

While not traditional in the strict sense, machine learning techniques such as Support Vector Machines (SVM) or Random Forests can be trained to classify pixels or regions as either braille characters or background.

**Recent Advancements in deep learning.**

**BRAUNET-**

Optical Braille Recognition methods usually use many designed steps, such as image de-skewing, Braille dots detection, Braille cell grids construction and Braille character recognition, which are less robust for complex Braille scenes. This paper proposes an optimal semantic segmentation framework BraUNet to directly detect and recognize Braille characters in the whole original Braille images. BraUNet adds extra auxiliary learning strategy to UNet network, which uses long-range connections of feature maps between encoder and decoder to get more low-level features. And auxiliary learning strategy can combine multi-class Braille characters segmentation with Braille foreground extraction, which can improve the feature learning ability and the Braille segmentation performance. Then morphological post-processing is used on semantic segmentation results to get the final individual Braille character regions. Experimental results show the proposed framework is robust, effective and fast for Braille characters segmentation and recognition on both complex double sided Braille image dataset and handwritten Braille image dataset.

(<https://openaccess.thecvf.com/content_CVPRW_2020/papers/w34/Li_Optical_Braille_Recognition_Based_on_Semantic_Segmentation_Network_With_Auxiliary_CVPRW_2020_paper.pdf>)

import cv2

import numpy as np

import math

th2=cv2.imread('roi.jpg')

r,c,w=th2.shape

horizontalStructure = cv2.getStructuringElement(cv2.MORPH\_RECT, (c+2000,13))

horizontal = cv2.dilate(th2, horizontalStructure, (-1, -1))

cv2.imwrite("horizontal2.jpg", horizontal)

img = cv2.imread('horizontal2.jpg')

#defining the edges

edges = cv2.Canny(img,50,150,apertureSize = 3)

cv2.imwrite('edges.jpg',edges)

#finding the end points of the hough lines

#lines = cv2.HoughLines(edges,1,np.pi/180,200)

m=[]

minLineLength = 100

maxLineGap = 10

lines = cv2.HoughLinesP(edges,1,np.pi/180,15,minLineLength,maxLineGap)

for x in range(0, len(lines)):

for x1,y1,x2,y2 in lines[x]:

m.append(((x1,y1),(x2,y2)))

#print m

print "length of list m is:",len(m)

sorted\_m=sorted(m, key=lambda x: x[0][1])

print "length of list sorted\_m is:", len(sorted\_m)

sorted\_m.insert(0,((0,0),(c,0)))

#drawing line

for i in range (0,len(sorted\_m)):

cv2.line(th2,sorted\_m[i][0],sorted\_m[i][1],(0,0,255),3)

cv2.imwrite('hough\_lines.png',th2)

s=cv2.imread('hough\_lines.png')

p=[]

for i in range (0,len(sorted\_m)):

if i!=len(sorted\_m)-1:

p.append(th2[sorted\_m[i][0][1]:sorted\_m[i+1][0][1],sorted\_m[i][0][0]:sorted\_m[i][1][0]])

else:

p.append(th2[sorted\_m[len(lines)-2][0][1]:r, sorted\_m[len(lines)][0][0]:sorted\_m[len(lines)][1][0]])

pix=[]

print(len(p))

for x in range(len(p)):

def contains\_white(img):

gray\_image = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

ret, threshold = cv2.threshold(gray\_image,100,255,cv2.THRESH\_BINARY)

h,w,l=img.shape

for i in range(h):

for j in range(w):

if threshold[i][j]==255:

return True

result= contains\_white(p[x])

if result== True:

pix.append(p[x])

print(len(pix))

for i in range(len(pix)):

cv2.imwrite('part' +str(i)+'.jpg',pix[i])

cv2.waitKey(0)

cv2.destroyAllWindows()