Deterministic Object Detection And Counting Report

Ian Sinclair
ENCE 3620-1

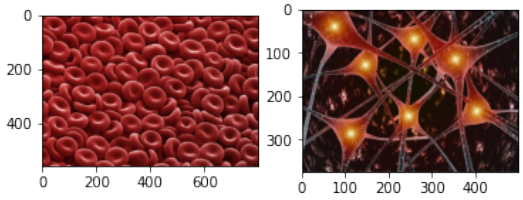
January 14, 2024
Dr. Mohammad H. Mahoor

Problem Statement

Use binary image processing techniques to count the number of objects (e.g. neurons or cells) in an image. You first need to convert the color image to gray scale and then binary using thresholding techniques. Then you should apply morphological operators to count to disconnect objects from each other and then apply connected component analysis to count the number of objects in the image. Also, find the center of detected neuron cells in image 1 and red cells in image 2. You are given two images for this problem. Work on both images and report the number of objects in each case. In your report, provide the output images after applying any operator or filter. Include your code in your report.

1 Implementation

Consider the following raw images.

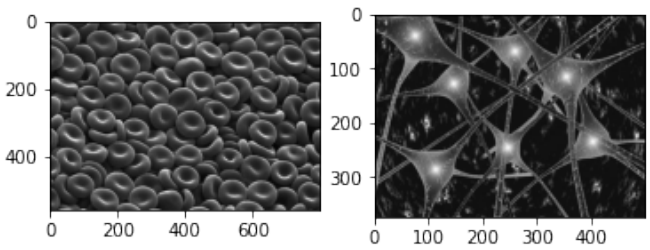


Here we want to devise a deterministic algorithm to count the number of unique objects in each image. Note, that the objects in each image both overlap and have non-trivial color distributions. Making is difficult to identify each unique object using traditional methods, i.e. BFS, or pixel value clustering.

The problem of color distribution can be solved by converting the image to grey scale. Note that the conversion method can affect intensity distribution in the 1-D image. here we use a weighted average method that is designed to better represent the original color distribution between the three color channels.

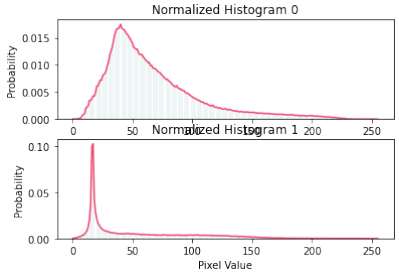
```
from traitlets.traitlets import List
def RGB_to_Greyscale( image ) :
  temp = image.copy()
  grey = np.zeros(( len(temp), len(temp[0]) ), dtype = int )

  for i in range(0 , len( image ) ) :
    for j in range(0 , len( image[i] ) ) :
      Filter =  0.299*temp[i][j][0] + 0.587*temp[i][j][1] + 0.114*temp[i][j][2]
      grey[i][j] = int( Filter )

  return grey
```

Next, as a means of separating objects from background, we define an object by a closed neighborhood of pixels with similar intensity values. By this definition, we want images that have a bimodal intensity distribution,



And generating the histograms of each image. Or rather have a district separation between high and low pixel intensity values.
To examine the distribution of our control images, consider the pixel intensity histogram.

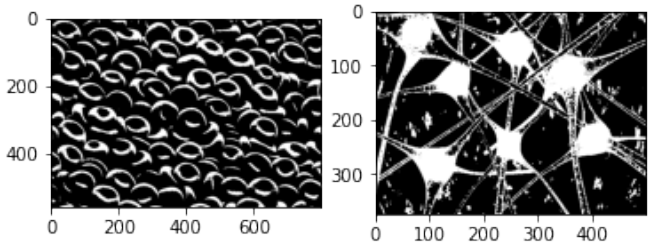
```
def generate_histogram( image ) :
  temp = image.copy()
  hist = []

  #Initialize an emply list on [0,255]
  for i in range( 0 , 256 ) :
    hist += [0]

  #Fill list with histogram values
  for i in range( 0 , len( image ) ) :
    for j in range( 0 , len( image[i] ) ) :
      intensity = int( image[i][j] )
      hist[intensity] += 1

  return hist

def generate_CDF( histogram ) :
  CDF = np.zeros( [256] )

  CDF[0] = histogram[0]

  for i in range( 1 , len( histogram ) ) :
    CDF[i] = CDF[i-1] + histogram[i]

  return CDF

def normalize_histogram( histogram ) :
  temp = histogram.copy()
  r = sum( temp )
  for i in range( 0 , len( temp ) ) :
    temp[i] = temp[i]/r

  return temp
```



We can improve the disparity between high intensity 'objects' and low intensity 'background' by converting the image to binary. Here we select a threshold and map all pixels below the threshold to 0 (minimum intensity) and all pixels above the threshold to 1 (maximum intensity). The choice of threshold affects the accuracy of the model's ability to filter out background. Here we use an otsu method to dynamically select a threshold based on the existing midpoint between bimodal peaks in the original grey-scale distribution. i.e. we select the midpoint between the perceived mean of high intensity pixels and the mean of low intensity pixels.

```
def Otsu( histogram ) :

```

```
  sig_max = 0
  t_best = 0
  q1 = generate_CDF( histogram )

  #mu = np.zeros(256,dtype = 'int')
  mu = 0
  for p in range( 0 , len( q1 ) ) :
    mu += p * histogram[p]

  mu1 = ((2)*histogram[2])/q1[2]
  mu2 = (q1[2]-mu1)/(1-q1[2])

  for t in range(1 , 255) :
    sig = q1[t]*(1-q1[t])*((mu1-mu2)**2)
    if sig > sig_max :
      t_best = t
      sig_max = sig

    mu1 = ((q1[t]*mu1) + (t+1)*histogram[t+1])/(q1[t+1])

    mu2 = (mu-(q1[t+1]*mu1))/(1-q1[t+1])


  return t_best

def Binary_Conversion( image, t : int) :
  binary = np.zeros(( len(image), len(image[0]) ), dtype = int )

  for i in range( 0 , len( image )) :
    for j in range( 0 , len( image[i] )) :
      if image[i][j] >= t :
        binary[i][j] = 1
      if image[i][j] < t :
        binary[i][j] = 0

  return binary
```



Notice, each object stands out against its background significant more. However, objects are still overlapping, and there are noisy high-intensity neighborhoods and that do not represent images.
As a result, we want to deploy a method to separate overlapping objects and evaporate false manifolds (high intensity neighborhoods that aren't objects).
And so consider the morphological operators:

Erosion: when convolved with an image, scrapes away at the edges of high intensity manifolds.

$$A \ominus B = \{z : (B)_z \in A\}$$

For manifolds A, B, and convolutional filter z

Dilation: when convolved with images, adds space to the edges along each manifold.

$$A \oplus B = \{z : (B)_z \cap A \neq \emptyset\}$$

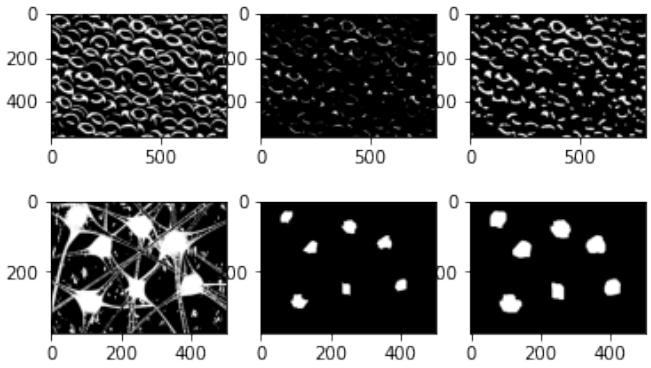
Next now the implementation of morphological operators,

```
def erosion( image ) :
  eroded_image = image.copy()

```

```
    mask = [(1,0), (-1,0), (0,1), (0,-1), (1,1), (-1,1), (-1,-1), (-1,1)]

    for i in range(0 , len( image ) ) :
      for j in range(0 , len( image[i] ) ) :
        eroded_image[i][j] = 1
        for k in mask :
          if 0 <= i+k[0] < len( image ) and 0 <= j+k[1] < len( image[i] ) :
            if image[i+k[0]][j+k[1]] != 1 :
              eroded_image[i][j] = 0
              break
          else :
            eroded_image[i][j] = 0
            break

    return eroded_image

def Dilation( image ) :
    Dilated_image = image.copy()

    temp_image = cv2.copyMakeBorder(image.copy(), 1, 1, 1, 1, cv2.BORDER_CONSTANT, value=0)

    mask = [(1,0), (-1,0), (0,1), (0,-1), (1,1), (-1,1), (-1,-1), (-1,1)]

    for i in range(0 , len( image ) ) :
      for j in range(0 , len( image[i] ) ) :
        Dilated_image[i][j] = 0
        for k in mask :
            if temp_image[i+k[0]][j+k[1]] == 1 :
              Dilated_image[i][j] = 1
              break

    return Dilated_image
```

```
neuron_image = erosion( binary_scale[1] )
for i in range( 0 , 10 ) :
    neuron_image = erosion( neuron_image )

neuron_image2 = Dilation( neuron_image )
for i in range( 0 , 5 ) :
    neuron_image2 = Dilation( neuron_image2 )

 blood_cells_image = erosion( binary_scale[0] )
for i in range( 0 , 2 ) :
    blood_cells_image = erosion( blood_cells_image )

blood_cells_image2 = Dilation( blood_cells_image )
for i in range( 0 , 2 ) :
    blood_cells_image2 = Dilation( blood_cells_image2 )
```

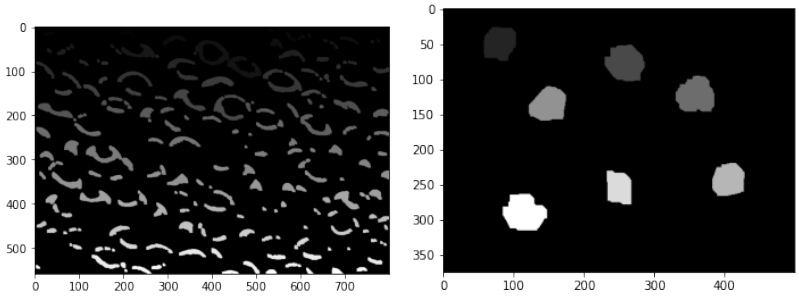
From here, opening is applied to both images. - First eroding each manifold with the goal of collapsing false manifolds, then dilating the remaining manifolds to reconstruct the pre-existing objects. Resulting in,



Finally, we can apply recursive connected component analysis to target the remain manifolds into object classes and count the number of unique classes.

```
def negate( image ) :
  label = np.zeros(( len(image), len(image[0]) ), dtype = int )

  for i in range( 0 , len( image )) :
    for j in range( 0 , len( image[i] )) :
      if image[i][j] != 0 :
        label[i][j] = -1

  return label

def neighbors(LB, L, P) :
  N = []
  if L+1 < len( LB ) :
    N += [(L+1, P)]
  if P+1 < len( LB[L] ) :
    N += [(L, P+1)]
  if L-1 >= 0 :
    N += [(L-1, P)]
  if P-1 >= 0 :
    N += [(L, P-1)]
  return N

def search( LB, label, L, P) :
  LB[L][P] = label
  Nset = neighbors(LB, L, P)

  for m in Nset :
    if LB[m[0],m[1]] == -1 :
      search(LB, label, m[0],m[1])

def find_components( LB, label ) :
  for L in range( 0 , len( LB )) :
    for P in range( 0 , len( LB[L] )) :
      if LB[L][P] == -1 :
        label += 1
        search( LB, label, L , P)

def recursive_connected_components( B, LB ) :
  LB = negate( B )
  label = 0
  find_components( LB, label )

  return LB


L = recursive_connected_components( blood_cells_image2, blood_cells_image2.copy() )
```

Here is an image detailing the intensity change between classes.



Which results in the number of classes from each image.

Number of Red Blood Cells	195
Number of Neurons	7

2 Conclusion

A deterministic algorithm pipeline is presented to count the number of distinct objects in an image. This pipeline is designed to solve the challenges of overlapping images with high noise and color distribution. It employs otsu thresholding and morphological operators as pre-processing to pull out objects and high-intensity manifolds and to collapse false manifolds or noise that could cause the algorithm to overestimate the number of images. Note: because of the use of morphological operations in the 'opening' process it is possible for objects to either collapse or split, resulting in an under or ever estimation respectively. Randomizing the morphological stack in the opening process over an ensemble of many trails, then taking the average of each ensemble may provide a better estimate / will further negate the effect of poor choices in the stack of morphological operators.