ECE 661 Homework 6

Michael Goldberg

October 16 2024

1 Theoretical Question

1.1 Otsu Vs. Watershed

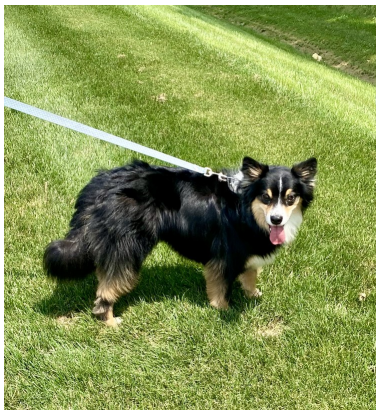
Otsu's algorithm and the Watershed algorithm both achieve image segmentation, however one may be more useful than the other in certain use cases. Otsu's advantage is that it is simple and computationally efficient, as you only need to calculate the histogram over the image once, and the thresholding happens automatically. However, Otsu's really only works well if the histogram is bimodal - that is, if there is a single global threshold that can separate two distinct peaks. If there are more complex lighting conditions complex backgrounds, or significant noise, Otsu's performance starts to dwindle.

Conversely, since the Watershed algorithm works based off of the flooding model operated on local gradients, it can handle complex images much better than Otsu's, as it does not require the same bimodal assumption that Otsu's does. However, this comes at the cost of higher computational complexity as the morphological operators used require more passes through the image than Otsu's. The Watershed algorithm can also lead to over-segmentation of the image if the original is noisy or fine-textured, so typically the image needs to be preprocessed to avoid this.

2 Programming Tasks

2.1 RGB Segmentation

The following task was to segment two images into foreground and background using Otsu's algorithm. Here are the two images to be segmented:



(a) Dog Image

(b) Flower Image

Otsu's algorithm aims to collect a histogram of the image's grayscale pixel value intensities and separate that histogram into its two most distinct parts using a threshold k. This k value is equal to the maximization of the Fisher Discriminant Function: $\lambda = \frac{\sigma_B^2}{\sigma_w^2}$, where σ_B^2 is the between-class variance and σ_w^2 is the within-class variance. Since the total variance of the set $\sigma_T = constant = \sigma_w^2 + \sigma_B^2$, then maximizing λ really means that we want to maximize σ_B^2.

The following is the calculation for σ_B^2 as well as intermediate values used, where L is the number of bins in the histogram, N is the total number of pixels in the image, and n_i is the number of pixels in each bin:

$$P_i = \frac{n_i}{N} \tag{1}$$

and

$$\omega_0 = \sum_{i=1}^{k} P_i \qquad\qquad \omega_1 = \sum_{i=k+1}^{L} P_i \tag{2}$$

Finally, mean can be calculated for class 0 and class 1, resulting in:

$$\mu_0 = \sum_{i=1}^{k} \frac{iP_i}{\omega_0} \qquad\qquad \mu_1 = \sum_{i=k+1}^{L} \frac{iP_i}{\omega_1} \tag{3}$$

and therefore:

$$\sigma_B^2 = \omega_0\omega_1(\mu_0 - \mu_1)^2 \tag{4}$$

We can then iterate through each possible value of k and calculate its corresponding σ_B^2, and collect the value where it is maximum. This is repeated for each channel and the masks are combined using boolean AND.

The segmentation must occur with a grayscale image, so there are two strategies we might employ. The first is to separate the image into its three RGB color channels, while the second would be to create three texture channels to feed into Otsu's.

2.1.1 RGB Segmentation

The results of the RGB segmentation can be seen below:

(a) Dog RGB Mask Red

(b) Dog RGB Mask Green

(c) Dog RGB Mask Blue

(d) Dog RGB Mask Combined

(a) Flower RGB Mask Red

(b) Flower RGB Mask Green

(c) Flower RGB Mask Blue

(d) Flower RGB Mask Combined

Parameters used: iterations = 4

It can be seen that Otsu's works relatively well for extracting the foreground with the RGB approach. However, with less complex backgrounds as is the case with the flower, it works way better than the dog image.

2.1.2 Texture Segmentation

This was performed by a sliding window approach, where each channel represents the results of a texture measure taken with three different sized sliding windows. The texture measure was determined through the following, where N is the window size:

$$texture_{i,j} = variance(window - \mu_{window}) \tag{5}$$

where the window is defined as the square region bounded by $i - int(\frac{N}{2})$ to $i + int(\frac{N}{2})$ in the x direction and $j - int(\frac{N}{2})$ to $j + int(\frac{N}{2})$ in the y direction. At the borders, the image was padded by $0s$ with a padding distance of $int(\frac{N}{2})$.

Once this texture image is established for three varying sizes of N, the same process is repeated as above when fed into Otsu's, and the masks are combined.

The results of the texture segmentation can be seen below:

(a) Dog Texture Mask N=11

(b) Dog Texture Mask N=17

(c) Dog Texture Mask N=23

(d) Dog Texture Mask Combined

(a) Flower Texture Mask N=5

(b) Flower Texture Mask N=7

(c) Flower Texture Mask N=9

(d) Flower Texture Mask Combined

Parameters used: Dog: 1 iteration, N = [11, 17, 23] Flower: 1 iteration, N = [5, 7, 9]

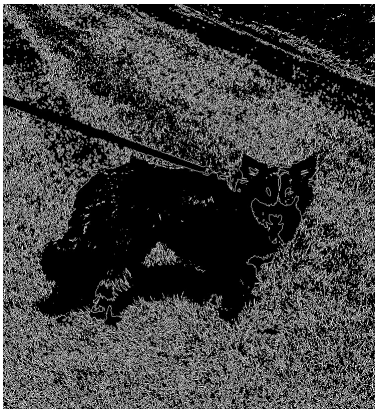
It can be seen that this method of texture finding really is best at extracting the edges of the foreground, which if morphologically closed, could be better performance than the RGB segmentation. However, as it stands, the RGB segmentation as a raw result is better than this segmentation.

2.1.3 Contour Finding

I decided to use a very simple contour finding algorithm to extract the contours from the final masks of both the RGB and the texture approaches from Otsu's.

The algorithm I chose stems from the concept of morphological gradients, which is essentially the erosion of an image subtracted from the dilation of an image. I decided to instead subtract the erosion of the mask from the original mask itself, leading to a single pixel tracing out the boundaries of the original mask (without overinflating the boundary which is what the morphological gradient would do due to the dilation). It is important to note that the structuring element used for this should be a 3×3 cross-shaped kernel, as this forces the boundary to be a single pixel wide without losing watertightness. The following is the result of this algorithm:

(a) Original RGB Dog Mask



(b) Contoured RGB Dog Mask

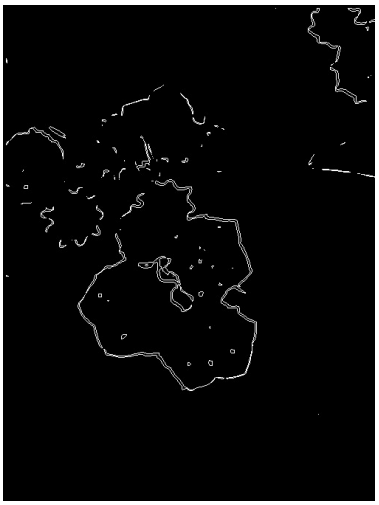
(c) Original Texture Dog Mask

(d) Contoured Texture Dog Mask

(a) Original RGB Flower Mask

(b) Contoured RGB Flower Mask

(c) Original Texture Flower Mask



(d) Contoured Texture Flower Mask

2.2 Otsu's on Custom Images

The above processes were repeated for my own images.

(a) Fox Image

(b) Whale Image

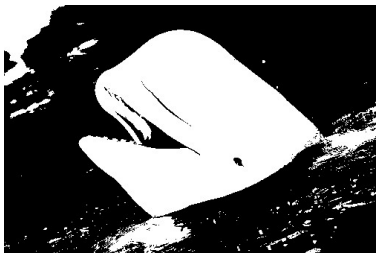
2.2.1 RGB Segmentation

(a) Fox RGB Mask Red

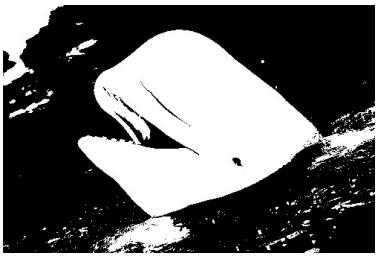
(b) Fox RGB Mask Green

(c) Fox RGB Mask Blue

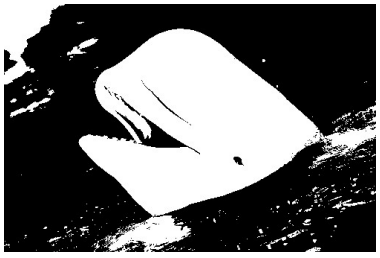
(d) Fox RGB Mask Combined



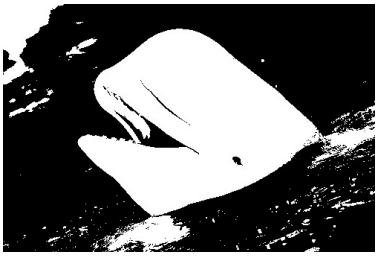
(a) Whale RGB Mask Red



(b) Whale RGB Mask Green

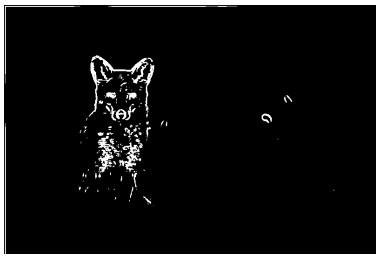


(c) Whale RGB Mask Blue

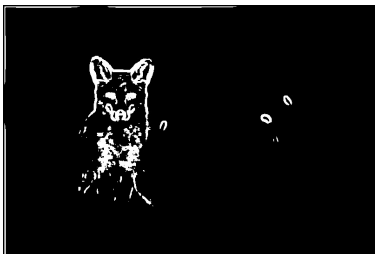


(d) Whale RGB Mask Combined

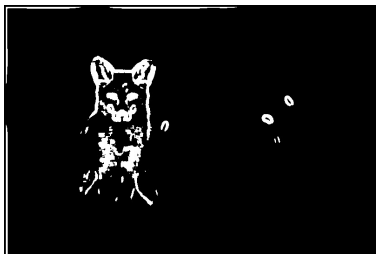
2.2.2 Texture Segmentation



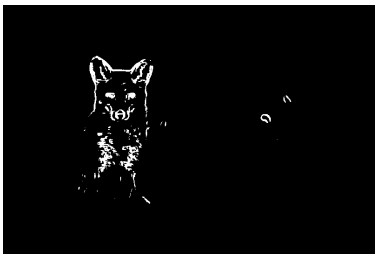
(a) Fox Texture Mask N=5



(b) Fox Texture Mask N=7



(c) Fox Texture Mask N=9



(d) Fox Texture Mask Combined

(a) Whale Texture Mask N=5

(b) Whale Texture Mask N=7

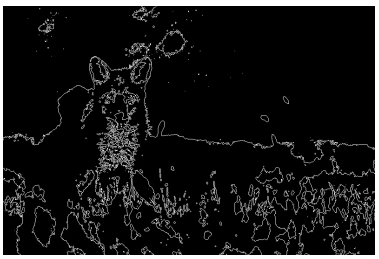
(c) Whale Texture Mask N=9

(d) Whale Texture Mask Combined

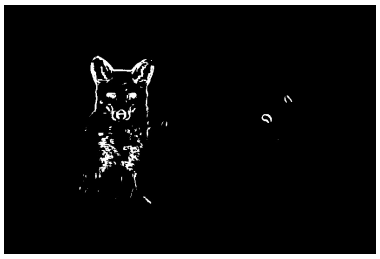
Parameters used: Fox: 1 iteration, N = [5, 7, 9] Whale: 1 iteration, N = [5, 7, 9]

2.2.3 Contour Finding

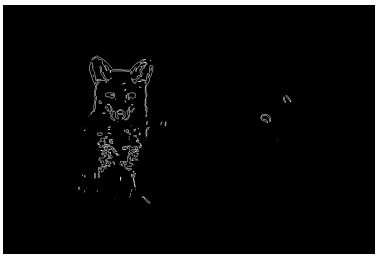
(a) Original RGB Fox Mask



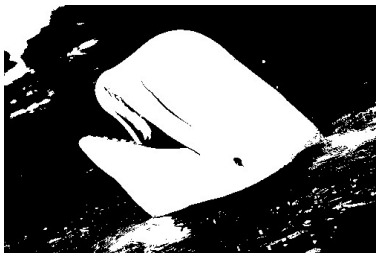
(b) Contoured RGB Fox Mask



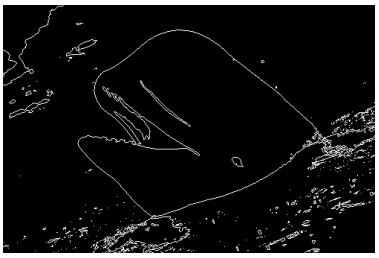
(c) Original Texture Fox Mask



(d) Contoured Texture Fox Mask



(a) Original RGB Whale Mask



(b) Contoured RGB Whale Mask

(c) Original Texture Whale Mask

(d) Contoured Texture Whale Mask

3 Code

```
import cv2
import numpy as np
import math
import matplotlib.pyplot as plt
from scipy.ndimage import uniform_filter

def compute_histogram(img):
 histogram = np.zeros(256, dtype=int)
 for pixel in img.ravel():
  histogram[int(pixel)] += 1

 return histogram

def find_otsu_threshold(histogram):

 max_variance = -np.inf
 w_0 = 0
 w_1 = 0
 sum_0 = 0
 sum_1 = 0

 threshold = 0

 total_sum = sum(histogram * np.arange(256))
```

```
 # Using 256 bins due to 0-255 being standard grayscale range
 for k in range(256):

  w_0 = sum(histogram[:k])
  w_1 = sum(histogram[k+1:])
  if(w_0 == 0 or w_1 == 0):
   continue

  sum_0 += (k * histogram[k])
  sum_1 = int(total_sum - sum_0)

  between_class_variance = w_0 * w_1 * (sum_0 / w_0 - sum_1 / w_1) ** 2

  if(between_class_variance > max_variance):
   max_variance = between_class_variance
   threshold = k


 return threshold



def otsu_rgb(img, max_iterations, namestring):
 mask = np.ones(img.shape[:2], dtype=np.uint8)

 title_list = ["red", "green", "blue"]

 cv2.imshow("channel blue", img[:, :, 0] * mask)
 cv2.imshow("channel green", img[:, :, 1] * mask)
 cv2.imshow("channel red", img[:, :, 2] * mask)

 cv2.waitKey(0)

 for iteration in range(max_iterations):
  channel_thresholds = []

  for i in range(3):
   channel = img[:, :, i] * mask

   histogram = compute_histogram(channel)

   threshold = find_otsu_threshold(histogram)

   channel_mask = np.where(channel >= threshold, 1, 0).astype(np.uint8)
   # mask[channel < threshold] = 0
```

```
   if(save_images and iteration == max_iterations - 1):
    save_channel_mask = channel_mask * 255
    cv2.imwrite('pics/' + namestring + '_rgb_mask_' + title_list[i] + '.jpg', save_channel_m

   channel_thresholds.append(channel_mask)

  # combining channel thresholds with boolean AND
  combined_mask = channel_thresholds[0]
  for i in range(len(channel_thresholds)):
   combined_mask = cv2.bitwise_and(combined_mask, channel_thresholds[i])

  mask = combined_mask.copy()


 return mask * 255

def otsu_texture(img, window_list, max_iterations, namestring):
 gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
 mask = np.ones(img.shape[:2], dtype=np.uint8)


 for iteration in range(max_iterations):
  channel_thresholds = []
  channels = []

  gray = gray * mask

  for window_size in window_list:
   pad_size = window_size // 2
   padded_img = np.pad(gray, pad_size, mode='constant', constant_values=0)
   variance_img = np.zeros(gray.shape)

   for v in range(img.shape[0]):
    for u in range(img.shape[1]):
     min_u = u - pad_size
     max_u = u + pad_size + 1
     min_v = v - pad_size
     max_v = v + pad_size + 1

     window = padded_img[min_v:max_v, min_u:max_u]

     if window.size > 0:
      mean_intensity = np.mean(window)
      variance_img[v, u] = np.var(window - mean_intensity)
     else:
```

```
      variance_img[v, u] = 0

   variance_img = cv2.normalize(variance_img, None, 0, 255, norm_type=cv2.NORM_MINMAX)
   # variance_img = (variance_img - np.min(variance_img)) / (np.max(variance_img) - np.min(v

   channels.append(variance_img)

  for i in range(3):
   channel = channels[i] * mask

   histogram = compute_histogram(channel)

   threshold = find_otsu_threshold(histogram)

   channel_mask = np.where(channel >= threshold, 1, 0).astype(np.uint8)

   if(save_images and iteration == max_iterations - 1):
    save_channel_mask = channel_mask * 255
    cv2.imwrite('pics/' + namestring + '_texture_mask_' + str(window_list[i]) + '.jpg', save

   channel_thresholds.append(channel_mask)

  # combining channel thresholds with boolean AND
  combined_mask = channel_thresholds[0]
  for i in range(len(channel_thresholds)):
   combined_mask = cv2.bitwise_and(combined_mask, channel_thresholds[i])

  mask = combined_mask.copy()

 return mask * 255

def find_contours(mask, namestring):
 kernel = np.array([[0, 1, 0], [1, 1, 1], [0, 1, 0]], dtype=np.uint8)
 subtraction_img = cv2.erode(mask, kernel)

 if(save_images):
  cv2.imwrite('pics/' + namestring + '_contours.jpg', mask - subtraction_img, [cv2.IMWRITE_J


save_images = True

img_dog = cv2.imread('pics/dog_small.jpg')
img_flower = cv2.resize(cv2.imread('pics/flower_small.jpg'), (504, 672))
img_fox = cv2.imread('pics/fox.jpg')
img_whale = cv2.resize(cv2.imread('pics/whale.jpg'), (501, 333))
```

```
foreground_rgb_dog = otsu_rgb(img=img_dog, max_iterations=4, namestring="dog")
foreground_texture_dog = otsu_texture(img=img_dog, window_list=[11, 17, 23], max_iterations=
if(save_images):
 cv2.imwrite('pics/dog_rgb_mask_combined.jpg', foreground_rgb_dog, [cv2.IMWRITE_JPEG_QUALITY
 cv2.imwrite('pics/dog_texture_mask_combined.jpg', foreground_texture_dog, [cv2.IMWRITE_JPEG

foreground_rgb_flower = otsu_rgb(img=img_flower, max_iterations=4, namestring="flower")
foreground_texture_flower = otsu_texture(img=img_flower, window_list=[5, 7, 9], max_iteratio
if(save_images):
 cv2.imwrite('pics/flower_rgb_mask_combined.jpg', foreground_rgb_flower, [cv2.IMWRITE_JPEG_Q
 cv2.imwrite('pics/flower_texture_mask_combined.jpg', foreground_texture_flower, [cv2.IMWRIT

foreground_rgb_fox = otsu_rgb(img=img_fox, max_iterations=4, namestring="fox")
foreground_texture_fox = otsu_texture(img=img_fox, window_list=[5, 7, 9], max_iterations=1,
if(save_images):
 cv2.imwrite('pics/fox_rgb_mask_combined.jpg', foreground_rgb_fox, [cv2.IMWRITE_JPEG_QUALITY
 cv2.imwrite('pics/fox_texture_mask_combined.jpg', foreground_texture_fox, [cv2.IMWRITE_JPEG

foreground_rgb_whale = otsu_rgb(img=img_whale, max_iterations=4, namestring="whale")
foreground_texture_whale = otsu_texture(img=img_whale, window_list=[5, 7, 9], max_iterations
if(save_images):
 cv2.imwrite('pics/whale_rgb_mask_combined.jpg', foreground_rgb_whale, [cv2.IMWRITE_JPEG_QUA
 cv2.imwrite('pics/whale_texture_mask_combined.jpg', foreground_texture_whale, [cv2.IMWRITE_

find_contours(foreground_rgb_dog, "dog_rgb")
find_contours(foreground_texture_dog, "dog_texture")

find_contours(foreground_rgb_flower, "flower_rgb")
find_contours(foreground_texture_flower, "flower_texture")

find_contours(foreground_rgb_fox, "fox_rgb")
find_contours(foreground_texture_fox, "fox_texture")

find_contours(foreground_rgb_whale, "whale_rgb")
find_contours(foreground_texture_whale, "whale_texture")
```