ECE 661: Homework 6

Xingguang Zhang

(Fall 2020)

Theory Question

Lecture 14 will present two very famous algorithms for image segmentation: The Otsu Algorithm and the Watershed Algorithm. These algorithms are as different as night and day. Present in your own words the strengths and the weaknesses of each.

- The Otsu algorithm takes use of the global statistic property of the image and the Watershed algorithm only consider the local gradient. The Otsu algorithm generate segmentation for the whole image at once with one parameter, and the Watershed algorithm requires to expand the marker iteratively.

-Otsu algorithm:

Strengths: Fast

Weaknesses: Details may not look good; can't label multiple segmented components at the same time.

-Watershed algorithm:

Strengths: Better details, able to label multiple connected components during segmentation.

Weaknesses: Slow to run.

Otsu Algorithm

Given a histogram of the pixel grayscales of a whole image, we want to find the optimal threshold of the grayscale which can separate the pixels in foreground and background apart.

Otsu algorithm uses Fisher's discriminant analysis to separate values in one distribution into 2 components whose distributions are the most linear distinguishable from each other.

The discriminant score λ is defined as the ratio between the between-class-variance(σ_B^2) and the within-class-variance(σ_W^2). We iterate over the bins of the grayscale histogram to find the fisher's discriminant score λ under every potential threshold k:

$$\lambda(k) = \frac{\sigma_B^2(k)}{\sigma_W^2(k)}$$

In Otsu algorithm we just optimize $\sigma_B^2(k)$, so the optimal threshold is

$$K = \underset{k}{\text{argmax}} \sigma_B^2(k) \tag{1}$$

We have two classes 0 and 1. The average grayscale of the entire image is μ, the average of the two classes are μ_0 and μ_1. The probability of grayscale i in the histogram is $p(i)$, the probability of a random gray belonging to the class 0 is ω_0 and the opposite is ω_1, the between class variance $\sigma_B^2(k)$ is defined as:

$$\sigma_B^2(k) = \omega_0(k)(\mu_0(k) - \mu)^2 + \omega_1(k)(\mu_1(k) - \mu)^2$$

If the smallest bin in the histogram is n and the largest is m, we have:

$$\omega_0(k) = \sum_{i=n}^{k} p(i)$$

$$\omega_1(k) = \sum_{i=k+1}^{m} p(i) = 1 - \omega_0(k)$$

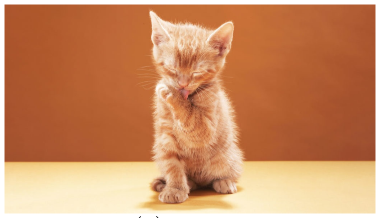
$$\mu_0(k) = \sum_{i=n}^{k} i \frac{p(i)}{\omega_0(k)}$$

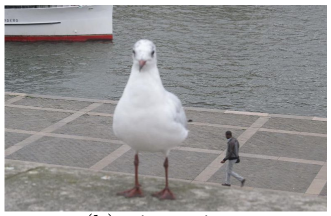
$$\mu_1(k) = \sum_{i=k+1}^{m} i \frac{p(i)}{\omega_1(k)}$$

Once we find threshold K by equation (1), we can mark all pixel value below K as background and others as foreground.

Programming Tasks

The task is to use different method to adopt Otsu algorithm to separate the foreground and background of the following images:


(a) cat.jpg


(b) pigeon.jpeg

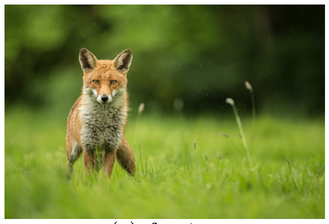

(c) fox.jpg

Figure 1: Input images

Image segmentation using RGB values

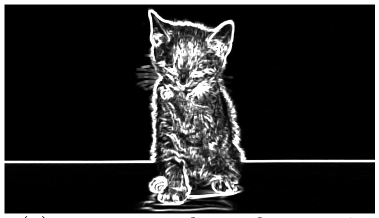
We first separate the color image into R, G, B channels. For each channel, use Otsu algorithm to segment the foreground and background parts.
In each channel, we adopt the Otsu algorithm iteratively. In the first iteration, we segment the histogram of the entire image, and we find the next optimal K which further separates one part of the former segmentation. We have a flag in the code to indicate which part (less or greater, less for the cat image and greater for the other two) we want to iterate over. We also have another flag to indicate if the pixels with values greater then the threshold belongs to the foreground.(All the 3 are yes) After the segmentation, we got an initial binary mask, then we adopt the erosion and dilation morphology process to refind the mask. For some images, since the remaining mask still contains some small parts of background, we further extract the largest connected component. The processing steps and parameters for the three images are shown in the following table 1.

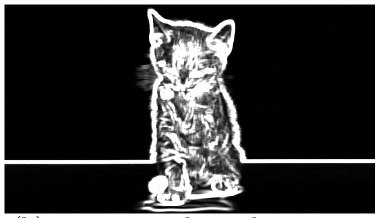
BGR image	iterations	thresholds	morphology kernel size	morphology steps
cat	[2, 1, 3]	[69, 155, 206]	5×5	$d \times 7,\ e \times 6$
pigeon	[1, 1, 1]	[159, 154, 153]	5×5	d, e, d, e, c
fox	[2, 1, 2]	[91, 108, 138]	3×3	$d \times 2,\ e,\ c$

Table 1: Segment steps, where d, e and c stands for dilation, erosion and extracting the largest connected component

Texture-based segmentation

I used the pixel variance inside a $N \times N$ neighborhood as the texture of the pixel. The steps for find the variance map is:
1. Convert a given RGB image to grayscale image.
2. Place a window of $N \times N$ at each pixel, subtract the mean intensity value and compute the intensity variance within each window as a texture measure at the center pixel.
3. Use the variance map of different window size N as the channels.
The variance maps are shown below:


(a) variance with window size 7


(b) variance with window size 11

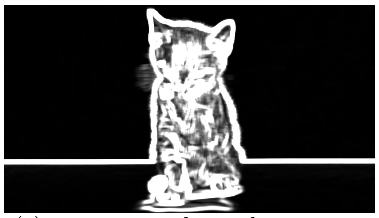
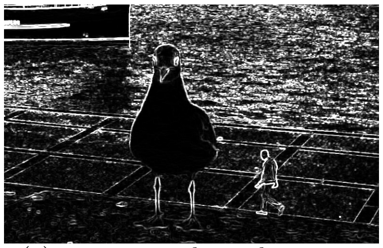
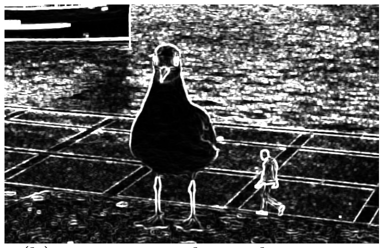

(c) variance with window size 15

Figure 2: Cat variance map with different window size


(a) variance with window size 3


(b) variance with window size 5

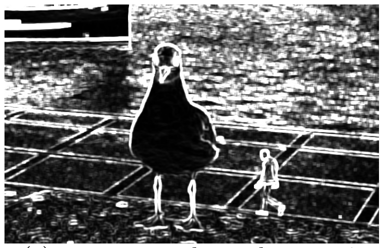
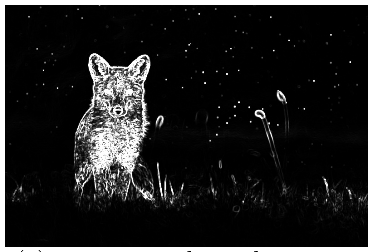
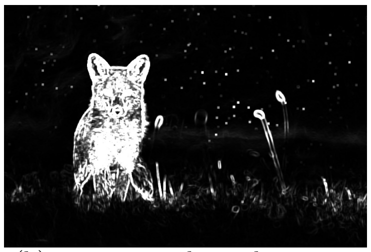

(c) variance with window size 7

Figure 3: Pigeon variance map with different window size


(a) variance with window size 3


(b) variance with window size 5

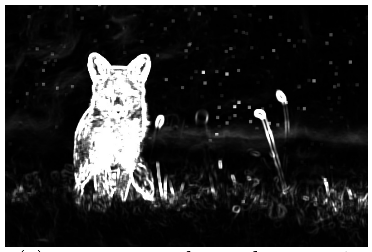

(c) variance with window size 7

Figure 4: fox variance map with different window size

In each channel, we adopt the Otsu algorithm iteratively. Like in the RGB based segmentation, we have a flag in the code to indicate which part (less or greater, we set less for all 3 images) we want to iterate over. We also have another flag to indicate if the pixels with values greater then the threshold belongs to the foreground.(Smaller for the pigeon image and greater for the other two). After the segmentation, we also used erosion, dilation and connected component to further refine the mask.
The processing steps and parameters for the three images are shown in the following table 2.

gray image	window size	iterations	thresholds	morphology kernel	morphology steps
cat	[7, 11, 15]	[5, 4, 4]	[4, 17, 20]	5×5	$d \times 2$, $e \times 3$
pigeon	[3, 5, 7]	[4, 3, 3]	[24, 77, 44]	3×3	e, d
fox	[3, 5, 7]	[4, 3, 3]	[8, 38, 43]	3×3	e, d, c, $d \times 2$, $e \times 2$

Table 2: Segment steps, where d, e and c stands for dilation, erosion and extracting the largest connected component

Contour Extraction

After we segment the foreground and background apart, the next step is contour extraction. Given a mask in which the foreground is marked as 1 and background is marked as 0. A pixel is on the contour if:
1. The pixel has value 1.
2. Not all its 8 neighbors is 1.

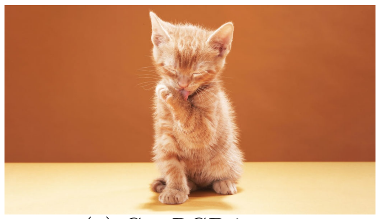
Notes and observation

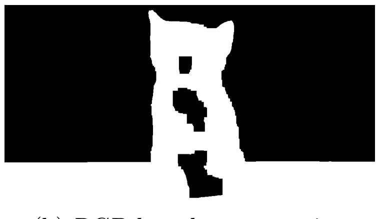
* To avoid the outside loops over k in the Otsu algorithm, we just calculate the elements in vectors $\mu_0(k)$, $\mu_1(k)$, $\omega_0(k)$, $\omega_1(k)$, $\sigma_B^2(k)$ in the same numpy call.

* To avoid loops over the image in the contour extraction, we make 9 copies of the original mask. We extent the input mask but 1 pixel in 4 directions and shift the original mask in 8 directions so that when we align the 9 copies, the pixel (i, j) and all its 8 neighbors can be placed at the same coordinate $(i + 1, j + 1)$ in different copies.

* Different methods perform differently on different images. For the cat and fox images, since the fur part has higher spatial frequency, the texture-based segmentation perform better, for the pigeon image, the RGB based method is better.

Segmentation results


(a) Cat RGB image


(b) RGB based segmentation

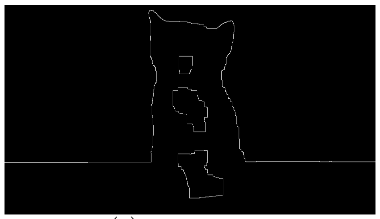
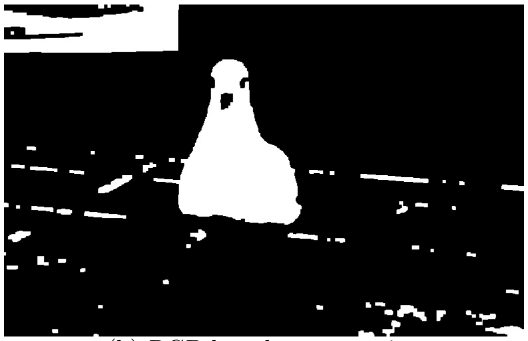
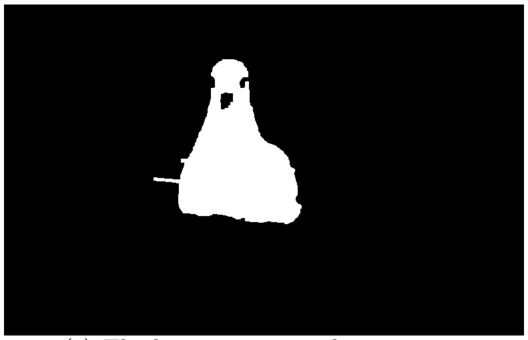

(c) cat contour

Figure 5: RGB Cat image segmentation

(a) pigeon RGB image


(b) RGB based segmentation


(c) The largest connected component

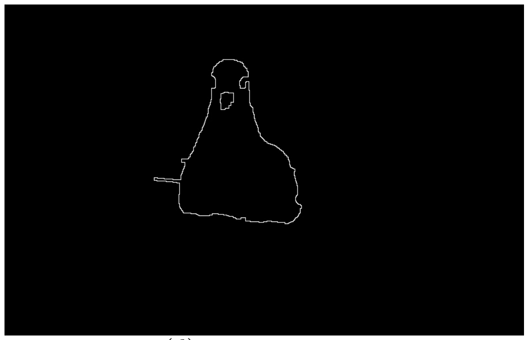
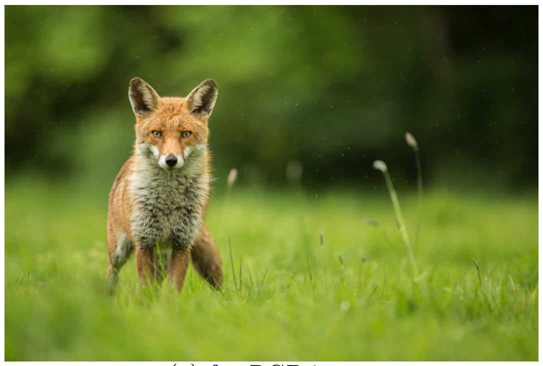
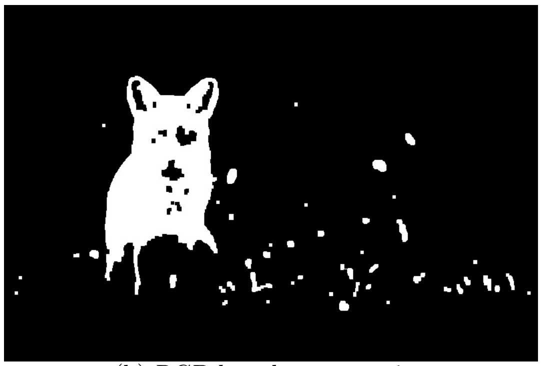
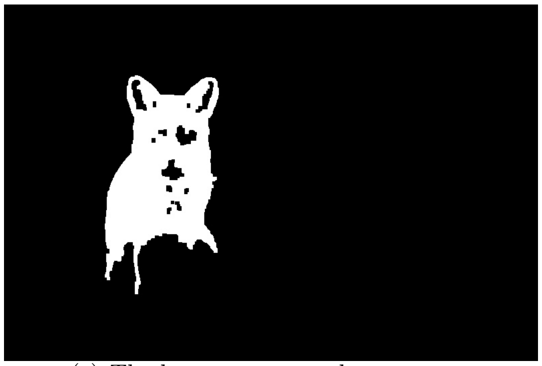

(d) pigeon contour

Figure 6: pigeon RGB image segmentation


(a) fox RGB image


(b) RGB based segmentation


(c) The largest connected component

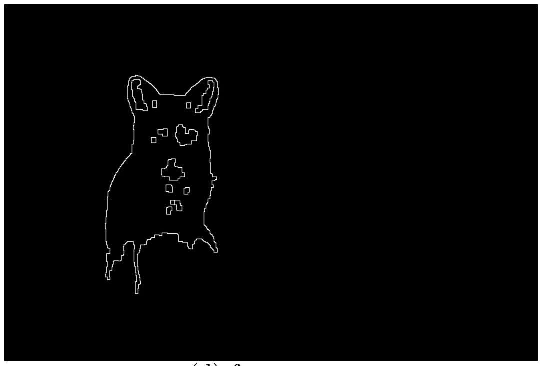
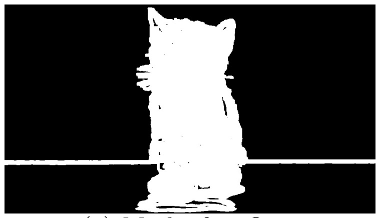
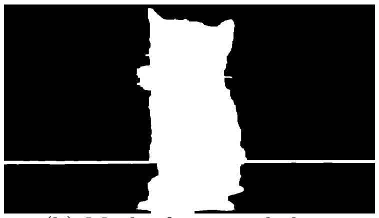

(d) fox contour

Figure 7: fox RGB image segmentation


(a) Mask after Otus


(b) Mask after morphology

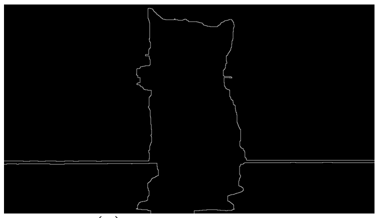
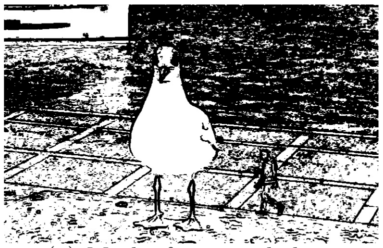
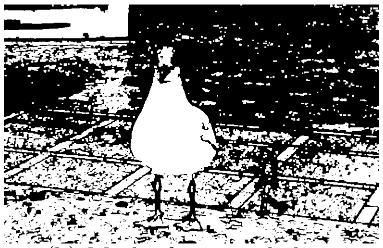

(c) contour image

Figure 8: Cat variance map segmentation


(a) Mask after morphology


(b) Mask after morphology

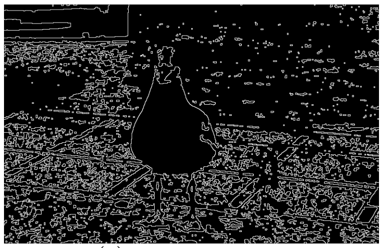
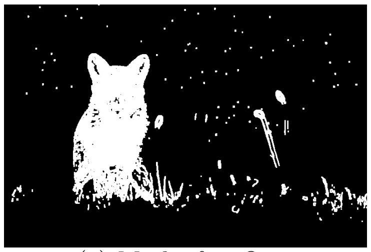
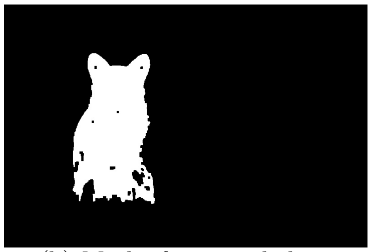

(c) contour image

Figure 9: Pigeon variance map segmentation


(a) Mask after Otus


(b) Mask after morphology

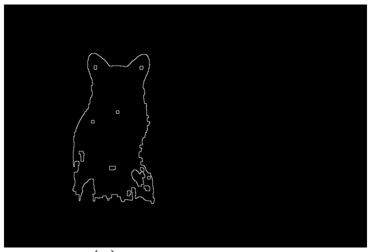

(c) contour image

Figure 10: fox variance map segmentation

Code

```
import numpy as np
import cv2
import os
import matplotlib.pyplot as plt


def Otsu(array):
    min_value = array.min().item()
    max_value = array.max().item()
    hist = cv2.calcHist([array],[0],None,[256],[min_value,max_value]).squeeze() /
   np.prod(array.shape)
    bins = np.linspace(min_value, max_value, 256)
    order1 = hist * bins

    omega_0 = np.cumsum(hist)
    omega_1 = 1 - omega_0
    cum_order = np.cumsum(order1)
```

```python
    cum_order_r = cum_order[-1] - cum_order

    mu_0 = np.zeros_like(omega_0)
    # avoid 0 omega_0 in the denominator, set the average of the first several 0
   valued bins as 0
    start_id = 0
    for i in range(256):
        if omega_0[i] == 0:
            start_id+=1
        else:
            break
    mu_0[start_id:] = (1 / omega_0[start_id:]) * cum_order[start_id:]
    mu_1 = (1 / omega_1) * cum_order_r

    sigmaB = omega_0 * omega_1 * (mu_0 - mu_1) * (mu_0 - mu_1)
    idx = np.argmax(sigmaB)
    K = bins[idx]
    return K


def iterOtsu(img, iter, l=False, reverse=False):
    '''
    if l is true, we select the left part of the segmented bins as our new segment
     target. This will make the threshold decrease
    if reverse is false, we assign the value above threshold as 1 in the mask
    '''
    array = img.ravel()
    for i in range(iter):
        k = Otsu(array)
        idx = np.where(array <= k) if l else np.where(array >= k)
        print(k, idx[0].shape)
        array = array[idx]

    mask = np.zeros_like(img, dtype=np.float32)
    if reverse:
        mask[np.where(img <= k)] = 1
    else:
        mask[np.where(img >= k)] = 1
    return mask, k


def BGRseg(img, iters, l=False, reverse=False):
    masks = []
    t = []
    for i in range(3):
        gray = img[:,:,i].squeeze()
        mask, threshold = iterOtsu(gray, iters[i], l, reverse)
        masks.append(mask)
        t.append(int(threshold))
    combined_mask = cv2.bitwise_and(masks[0], masks[1], masks[2])
    return combined_mask, t


def varmap(img, window_size):
    N = window_size
    b = int((N - 1) / 2)
    expand_im = cv2.copyMakeBorder(img, b, b, b, b, cv2.BORDER_REPLICATE)

    h, w = img.shape
    grids = np.zeros((h, w, N*N))
    for i in range(b, h+b):
        for j in range(b, w+b):
            grids[i-b, j-b] = expand_im[i-b:i+b+1, j-b:j+b+1].ravel()
    var_img = np.var(grids, axis=2, keepdims=False).astype(np.float32)
    return var_img


def contextseg(img, windows, iters, l=True, reverse=False):
    masks = []
    for i in range(len(iters)):
        var_img = varmap(img, windows[i])
        mask, threshold = iterOtsu(var_img, iter=iters[i], l=l, reverse=reverse)
        masks.append(mask)
    combined_mask = cv2.bitwise_and(masks[0], masks[1], masks[2])
    return combined_mask


def findContour(mask):
    '''
    Find the 8-neighbor contour in a parallel way.
    '''
    expanded_mask = cv2.copyMakeBorder(mask, 1, 1, 1, 1, cv2.BORDER_REPLICATE)
    neighbor_mask = np.zeros_like(expanded_mask)
    # cv2.copyMakeBorder(src, top, bottom, left, right, borderType, value)
    tl = cv2.copyMakeBorder(mask, 0, 2, 0, 2, cv2.BORDER_REPLICATE)
    t = cv2.copyMakeBorder(mask, 0, 2, 1, 1, cv2.BORDER_REPLICATE)
    tr = cv2.copyMakeBorder(mask, 0, 2, 2, 0, cv2.BORDER_REPLICATE)
    l = cv2.copyMakeBorder(mask, 1, 1, 0, 2, cv2.BORDER_REPLICATE)
    r = cv2.copyMakeBorder(mask, 1, 1, 2, 0, cv2.BORDER_REPLICATE)
    bl = cv2.copyMakeBorder(mask, 2, 0, 0, 2, cv2.BORDER_REPLICATE)
    b = cv2.copyMakeBorder(mask, 2, 0, 1, 1, cv2.BORDER_REPLICATE)
    br = cv2.copyMakeBorder(mask, 2, 0, 2, 0, cv2.BORDER_REPLICATE)
    accu = tl + t + tr + l + r + bl + b + br
    neighbor_mask[np.where(accu < 8)] = 1
    contour = cv2.bitwise_and(neighbor_mask, expanded_mask)
    return contour[1:-1, 1:-1]


def largestComponent(mask, connectivity=8):
    new_img = np.zeros_like(mask, dtype=np.float32)
    nb_components, output, stats, _ = cv2.connectedComponentsWithStats(mask.astype
    (np.uint8), connectivity)
    max_label, _ = max([(i, stats[i, cv2.CC_STAT_AREA]) for i in range(1,
    nb_components)], key=lambda x: x[1])
    new_img[np.where(output==max_label)] = 1
    return new_img


folder_task = '/home/xingguang/Documents/ECE661/hw6/images'
imcat = cv2.imread(os.path.join(folder_task, 'cat.jpg'))
impigeon = cv2.imread(os.path.join(folder_task, 'pigeon.jpeg'))
imfox = cv2.imread(os.path.join(folder_task, 'fox.jpg'))

graycat = cv2.cvtColor(imcat, cv2.COLOR_BGR2GRAY)
graypigeon = cv2.cvtColor(impigeon, cv2.COLOR_BGR2GRAY)
grayfox = cv2.cvtColor(imfox, cv2.COLOR_BGR2GRAY)

# ---------------------------cat image-------------------------------
```

```
mask, t = BGRseg(imcat, iters=[2, 1, 3], l=True, reverse=False)
print("BGR thresholds for cat image:", t)
kernel = np.ones((5,5),np.uint8)
dilation = cv2.dilate(mask,kernel,iterations = 7)
final_mask = cv2.erode(dilation,kernel,iterations = 6)

contour = findContour(final_mask)
cv2.imwrite("BGR_cat.jpeg", final_mask*255)
cv2.imwrite("BGR_cat_contour.jpeg", contour*255)

mask = contextseg(graycat, windows=[7, 11, 15], iters=[5, 4, 4], l=True)
cv2.imwrite("contextmask_cat.jpeg", mask*255)
kernel = np.ones((5,5),np.uint8)
dilation = cv2.dilate(mask,kernel,iterations = 2)
final_mask = cv2.erode(dilation,kernel,iterations = 3)
contour = findContour(final_mask)
cv2.imwrite("context_cat.jpeg", final_mask*255)
cv2.imwrite("context_cat_contour.jpeg", contour*255)

# ------------------------pigeon image-------------------------------
mask, t = BGRseg(impigeon, iters=[1, 1, 1], l=False, reverse=False)
print("BGR thresholds for ipgeon image:", t)

kernel = np.ones((5,5),np.uint8)
opened = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)
dilation = cv2.dilate(opened,kernel,iterations = 1)
final_mask = cv2.erode(dilation,kernel,iterations = 1)
largest = largestComponent(final_mask)
contour = findContour(largest)
cv2.imwrite("BGR_pigeon.jpeg", final_mask*255)
cv2.imwrite("BGR_pigeon_comp.jpeg", largest*255)
cv2.imwrite("BGR_pigeon_contour.jpeg", contour*255)

mask = contextseg(graypigeon, windows=[3, 5, 7], iters=[4, 3, 3], l=True, reverse=
    True)
cv2.imwrite("contextmask_pigeon.jpeg", mask*255)

kernel = np.ones((3,3),np.uint8)
erosion = cv2.erode(mask,kernel,iterations = 1)
final_mask = cv2.dilate(erosion,kernel,iterations = 1)
contour = findContour(final_mask)
cv2.imwrite("context_pigeon.jpeg", final_mask*255)
cv2.imwrite("context_pigeon_contour.jpeg", contour*255)

# -------------------------- fox image-------------------------------
mask, t = BGRseg(imfox, iters=[2, 1, 2], l=False, reverse=False)
print("BGR thresholds for fox image:", t)

kernel = np.ones((5,5),np.uint8)
dilation = cv2.dilate(mask,kernel,iterations = 2)
final_mask = cv2.erode(dilation,kernel,iterations = 1)
largest = largestComponent(final_mask)
contour = findContour(largest)
cv2.imwrite("BGR_fox.jpeg", final_mask*255)
cv2.imwrite("BGR_fox_comp.jpeg", largest*255)
cv2.imwrite("BGR_fox_contour.jpeg", contour*255)

mask = contextseg(grayfox, windows=[3, 5, 7], iters=[4, 3, 3], l=True, reverse=
    False)
```

```
cv2.imwrite("contextmask_fox.jpeg", mask*255)

kernel = np.ones((3,3),np.uint8)
erosion = cv2.erode(mask,kernel,iterations = 1)
final_mask = cv2.dilate(erosion,kernel,iterations = 1)
largest = largestComponent(final_mask)
largest = cv2.dilate(largest,kernel,iterations = 2)
largest = cv2.erode(largest,kernel,iterations = 2)
contour = findContour(largest)
cv2.imwrite("context_fox.jpeg", final_mask*255)
cv2.imwrite("context_fox_comp.jpeg", largest*255)
cv2.imwrite("context_fox_contour.jpeg", contour*255)
```