- Assignment 8 -

Signal and Image Processing

Philip Rajani Lassen & Maria Luise da Costa Zemsch

March 31, 2019

Image Segmentation

---

1 Histogram based segmentation

1.1 Cons for simple intensity-based segmentation

Global thresholding assumes that the intensity of the same features in the image are position invariant. However, in practice due to lighting conditions, i.e. shadows, features are defined by their intensities with respect to their pixel neighborhood. Another disadvantage with global thresholding is that it can struggle with noise. For example salt and pepper noise will lead to mis-classification due to the extreme nature of the pixel intensities Additionally global thresholding requires decent contrast between features and the background.

1.2

Pixel based thresholding can perform better when the contents of the image are predictable. For example MRI scans.

Edge based approaches can do better than intensity based approaches under varying light conditions and for images of higher complexity where intensity based thresholding stuggles.

1.3 Histogram based segmentation method

With histogram based segmentation we use a threshold to convert our grayscale image to a binary image. This threshold is chosen automatically by the method `find_threshold`. It searches for the nearest local minimum to the mean intensity value. By default every 10 neighbours are considered.

Listing 1: Optimal threshold

```
def find_threshold(img, order = 10):
  img_mean = np.mean(img)
  histo = np.histogram(img, bins  = 256)
  min_loc = argrelextrema(histo[0], np.less, order=order )
  min_val = np.argmin(np.absolute(min_loc - img_mean))
  return(min_loc[0][min_val])
```

The selection of the threshold is based on the idea, that the figures, which need to be detected, are with a higher intensity as the rest of the image. We assume that there is a local minimum between the background intensity and the intensity of the figures. Furthermore we assume that the whole background is darker than the figures. This assumption can lead to a wrong detection caused on overexposing or low contrast.
Good in this method is, the assumption that the local minimum is located next to the mean value, therefore we get a good differentiation of the background and structures in good exposed images. Negative is, that the chosen neighbours for the computation of the local minimum varies from image to image. If we set a special value, the output cannot be perfect.

Listing 2: Histogram based segmentation

```
def segmentation_histo(img, order = 10):
  threshold = find_threshold(img, order)
  img_thresh = cv.threshold(img,threshold,255,cv.THRESH_BINARY)
  return(threshold, img_thresh[1])
```

Regarding that threshold we can run our segmentation algorithm and get an output as seen in figure 1.

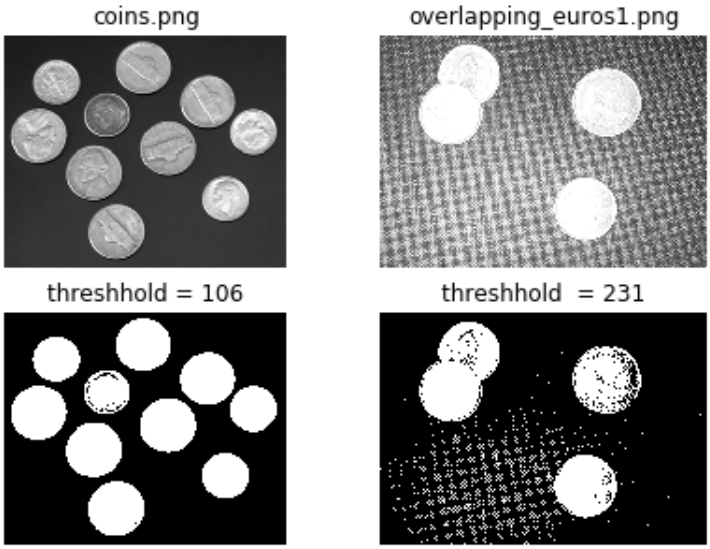

Figure 1: Intensity histogram based segmentation for the test images `coins.png` and `overlapping_euros1.png`. The optimal threshold was chosen by the method `find_threshold`.

As you can see the method works good for the `coins.png` image but not for `overlapping_euros1.png`. The reason can be explained including the intensity histograms on figure 2. As the second image is overexposed we get a high peak at the high intensity values especially 255. Theses values are not only located at the coins, but also in the background. Thus our chosen threshold (around the mean intensity value) cannot differ between coin or no coin. Due to overexposure, the threshold in this example should have been near to 255.

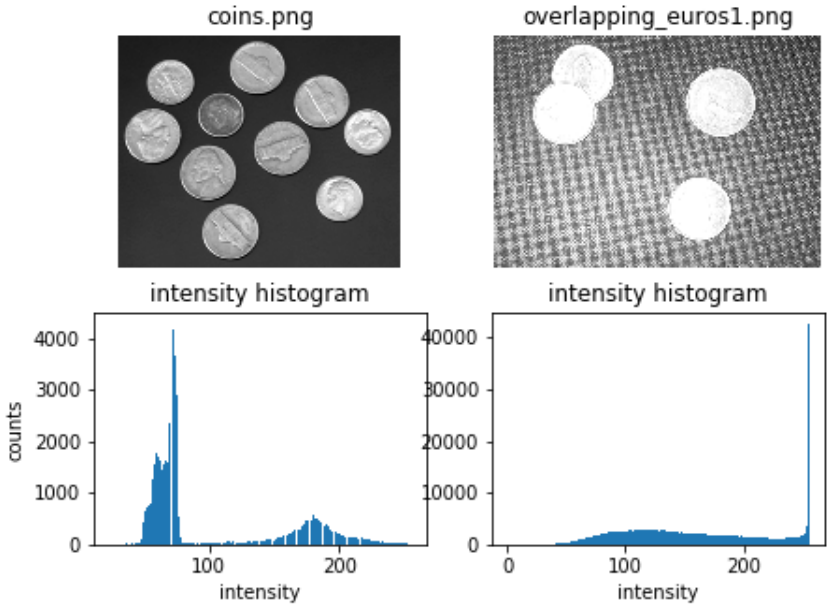

Figure 2: Intensity histograms of the test images `coins.png` and `overlapping_euros1.png`

2 The Hough Transform

2.1 Implementation of the straight line Hough Transform

With the straight line Hough Transform we try to find aligned points, which form a straight line. Therefore we need to transform our values from the x- and y-plane to the feature space. Considering a line in the x-/y-plane with the formula

$$y = a_i * x + b_i \tag{1}$$

This refers to a certain point (a_i, b_i) in the feature space. Instead of the parameters a and b, we can describe the line with the angle θ and the distance ρ from the origin.

$$\rho = x \cdot cos(\theta) + y \cdot sin(\theta) \tag{2}$$

If one point in the feature space appears several times, we have a defined line in the x-/y-plane[1].
This is the whole theory behind our algorithm. We are looking at θ-values in between -90 and 90 degrees. The maximal distance ρ_{max} is the diagonal of the image and can be computed by the Pythagoras. As we are looking at a binary image (edge detection), we can reduce the calculations by only looking at the non zero values at the image. For each of this points we

[1]Source: https://www.youtube.com/watch?v=4zHbI-fFIlI (25.03.19)

compute the right ρ's in the feature space for all θ-values with the function 2[2].

Listing 3: Straight line Hough Transform

```
def hough_trans_line(img):
  width, height = img.shape
  thetas = np.deg2rad(np.arange(-90., 90.)) #angle
  diag = np.ceil(np.sqrt(width**2 + height**2))
  rhos = np.arange(-diag, diag+1) #distance
  output = np.zeros((len(rhos), len(thetas)))
  edges_y, edges_x = np.nonzero(img)
  for i in range(len(edges_y)):
    x = edges_x[i]
    y = edges_y[i]
    for t in range(len(thetas)):
      theta = thetas[t]
      rho = x*np.cos(theta) + y*np.sin(theta)
      r = np.argmin(np.absolute(rhos-rho))
      output[r, t] += 1
  return output, thetas, rhos #same output structure as scikit-image.hough_line
```

Since the algorithm already only looks at the detected edges, the computational time was reduced drastically. Still, if the image has a lot of detected corners the runtime increases. Furthermore the algorithm works only on binary images. Therefore edge detection has to be applied before the first step.

2.2 Application of the algorithm

The result (see figure 3) is a grayscale image with high intensity values at the ρ and θ values, which resembles the found line-function in the original image. To get a better contrast in the Hough Transform image we log-transformed it[3].

[2]Source: https://alyssaq.github.io/2014/understanding-hough-transform/ (25.03.19)
[3]Source: http://scikit-image.org/docs/dev/auto_examples/edges/plot_line_hough_transform.html#sphx-glr-auto-examples-edges-plot-line-hough-transform-py (29.03.19)

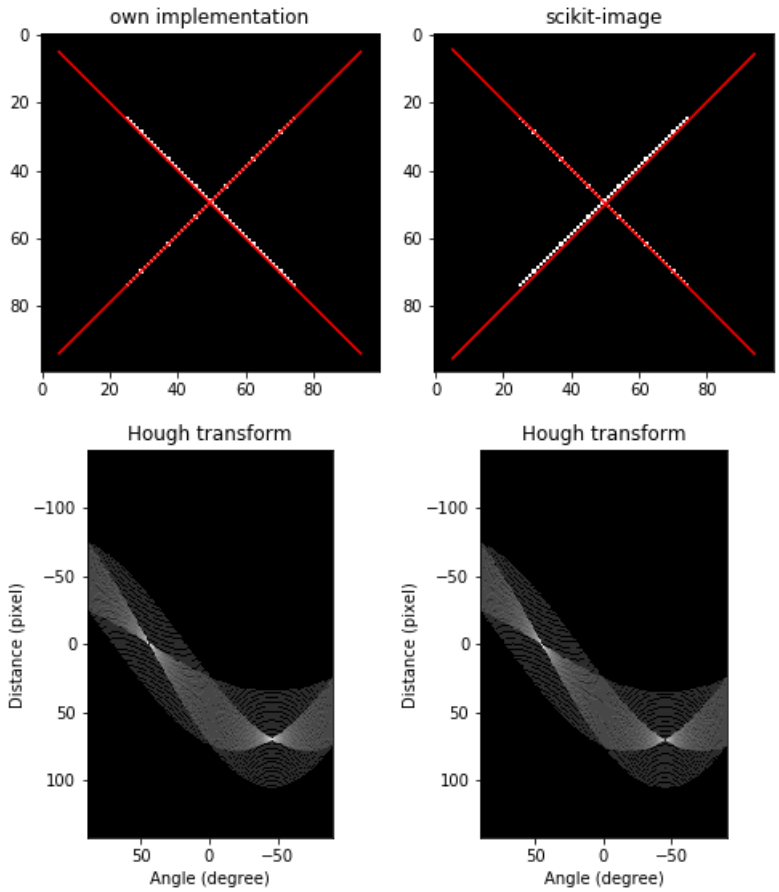

Figure 3: Straight line Hough transform for the `cross.png` image. The output of the self implemented method is plotted on the left. For comparison the output of the method `hough_line` from the package `scikit-image` was plottet to the right. The Hough Transform was log-transformed to higher the contrast in the image.

The function of the lines in the x-/y-plane can be reconstructed with the following function

$$y = \frac{\rho - x \cdot cos(\theta)}{sin(\theta)} \tag{3}$$

while ρ and θ are the max values in the feature space, which are found with the function `hough_trans_line_peak`.

Listing 4: Peaks in the straight line Hough Transform image

```
def hough_trans_line_peak(hough, angles, dis, k):
  lmc = peak_local_max(hough, min_distance=k)
  angele_ind = np.array([angles[lmc[0][1]], angles[lmc[1][1]]])
  dis_ind = np.array([dis[lmc[0][0]], dis[lmc[1][0]]])
  return angele_ind, dis_ind
```

If you compare the Hough Transform of our method with the method from the package

scikit-image, you almost noticed no difference. The lower max-peak is lightly lighter at the Hough Transform of the scikit-image package, thus one of the reconstructed lines does not perfectly fit the cross. Therefore our method gives a better output in this simple example.

2.3 Segmentation based on circular Hough Transform

With circular Hough Transform we could construct the algorithm segmentation_hough, which perform segmentation on images with circular structures. The first step is to detect the edges at the image. This binary image is used to compute the Hough Transform (see figure 5, right), which indicates the centers and the right radii of each circular structure.

Listing 5: Segmentation based on the circular Hough Transform

```
def segmentation_hough(img, sigma = 3.5, radii_lim = [20, 50], num_peaks = 12):
  edges = feature.canny(img, sigma=sigma)
  radii = np.arange(radii_lim[0], radii_lim[1], 2)
  hough = hough_circle(edges, radii)
  hough_par, x_par, y_par, rad_par = hough_circle_peaks(hough, radii,
                                                        total_num_peaks=num_peaks)
  img_seg = make_circle(img.shape, x_par, y_par, rad_par)
  return img_seg, np.max(hough, axis=0)
```

The output image is created by the method make_circle. It's based on the algorithms of assignment 3.

Listing 6: Binary image with circles

```
def make_circle(size, x_cord, y_cord, rad):
  mask = np.zeros(size)
  Y = size[0]
  X = size[1]
  for i in range(Y):
      for j in range(X):
        for c in range(len(x_cord)):
          r_ref = np.linalg.norm([i-y_cord[c], j-x_cord[c]])
          if r_ref < rad[c]:
              mask[i,j] = 1
  return mask
```

The example result can be seen in figure 5, where the segmentation algorithm was used on the image coin.png. The binary segmentation image represents very well the location and the size of the coins.

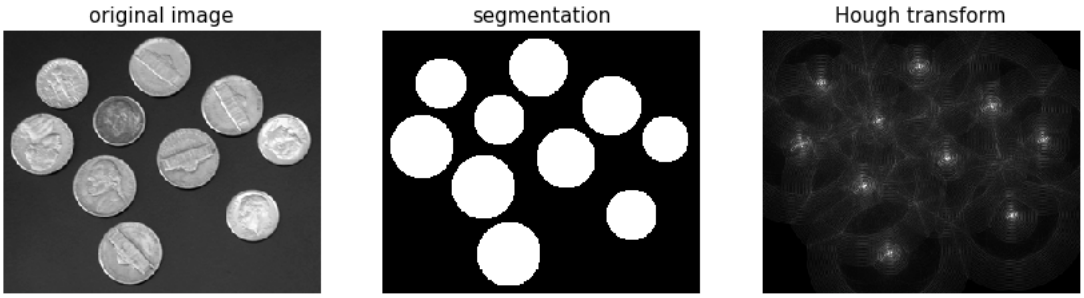

Figure 4: Segmentation based on the circular Hough transform for the `coin.png` image. The Hough Transform image shows the maximal values for every angle. Therefore it illustrates only the location of the coins.

This implementation of a segmentation algorithm is better then the histogram based method created in section 1.3, thus we can eliminate disruptive factors as background noise.

3 Machine learning based segmentation

3.1

In patch based machine learning we take subsections of images also known as patches and use machine learning to predict how we would like to classify this patch of the image. The training data are images and patches, and the test data is patches. Some of the advantages of patch based machine learning is that we can use more complex models for segmentation based on the machine learning classifier we use such as Neural Networks. Similarly we can leverage large amounts of data for better segmentation. Some of the disadvantages are that we need large amounts of data, and we aren't leveraging any domain specific information into our models. However we can encode some of the domain specific information as an additional input to our machine learning model, which can improve performance.

3.2

Using the evaluate function from the keras library, we get a test accuracy of approximately 0.9718. This means that for 0.9718 of the image patches we correctly labeled which one of the 135 classes the center pixel in the image patch belongs to.

3.3

Listing 7: Patch Image

```
def pred(f):
  (rows, cols) = f.shape
  result = np.zeros(f.shape)
  f = np.pad(f, (14, 14), 'constant')
  f = f / 255
  for i in range(int(rows)):
    for j in range(int(cols)):
      temp = f[i:i + 29, j:j + 29]
      temp = temp.reshape((1, 1, 29, 29))
      val = model.predict(x = temp)
      result[i, j] = (np.argmax(val))
  return result
```

The code simply goes through every patch in the image and classifies the center pixels output based on the neural network classifier, and return a new image.

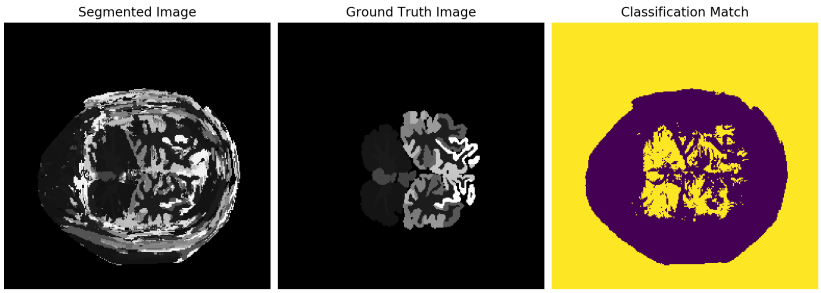

Figure 5: Segmented image from the prediction function on the left. The ground truth image in the middle. On the right is an image of matching classification labels for the pixels.

The image shows the result of the prediction function on one of the test images. The image on the right of the figure displays where the classifications match. The yellow pixels display matching classifications. We can see that the segmentation does fairly well. There are some misclassifications though in a ring around the brain and some other misclassifications in the center.

3.4

Listing 8: Dice Coefficent

```
def dice(f, g, c):
  num =  2 * np.sum(np.logical_and((f == c), (g == c)))
  denom = (np.sum(f == c) + np.sum(g == c))
  return num / denom
```

Above is a helper function the computes the dice coefficient for a single class.

Listing 9: Dice Coefficent for all classes

```
def dice(f, g):
  result = []
  for c in range(135):
    val = dice(f, g, c)
    result += [val]
  return result
```

Finally we implement a function that computes the dice coefficient for each class. It goes through each of the 135 classes and computes the dice coefficent and return an array of 135.