Thresholding

Objectives:

- To explore and implement various thresholding techniques, including simple, adaptive (mean and Gaussian), and automated methods (Otsu and Riddler-Calvard), to convert grayscale images into binary images and analyze the effectiveness of each method in different scenarios.

- To evaluate how the choice of threshold values and methods affects the segmentation of images, particularly in distinguishing objects from the background, and to determine the most appropriate thresholding technique for images with varying intensity distributions.

Thresholding is the binarization of an image. In general, we seek to convert a grayscale image to a binary image, where the pixels are either 0 or 255.

A simple thresholding example would be selecting a pixel value *p*, then setting all pixel intensities less than *p* to zero, and all pixel values greater than *p* to 255. In this way, we can create a binary representation of the image.

simple thresholding

Applying simple thresholding methods requires human intervention. We must specify a threshold value *T*. All pixel intensities below *T* are set to 0. All pixel intensities greater than *T* are set to 255.

We can also apply the inverse of this binarization by setting all pixels below *T* to 255 and all pixel intensities greater than *T* to 0.

simple_thresholding.py

```
1  import numpy as np
2  import argparse
3  import cv2
4
5  ap = argparse.ArgumentParser()
6  ap.add_argument("-i", "--image", required = True,
7      help = "Path to the image")
8  args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11  image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
12 blurred = cv2.GaussianBlur(image, (5, 5), 0)
13 cv2.imshow("Image", image)
```

On **Lines 1-10** we import our packages, parse our argu- ments, and load our image. From there, we convert the image from the RGB color space to grayscale on **Line 11**.

At this point, we apply Gaussian blurring on **Line 12** with a $\sigma = 5$ radius. Applying Gaussian blurring helps re- move some of the high frequency edges in the image that we are not concerned with.

Listing 2: simple_thresholding.py

```
14 (T, thresh) = cv2.threshold(blurred, 155, 255, cv2.THRESH_BINARY)
15 cv2.imshow("Threshold Binary", thresh)
16
17 (T, threshInv) = cv2.threshold(blurred, 155, 255, cv2.
       THRESH_BINARY_INV)
```

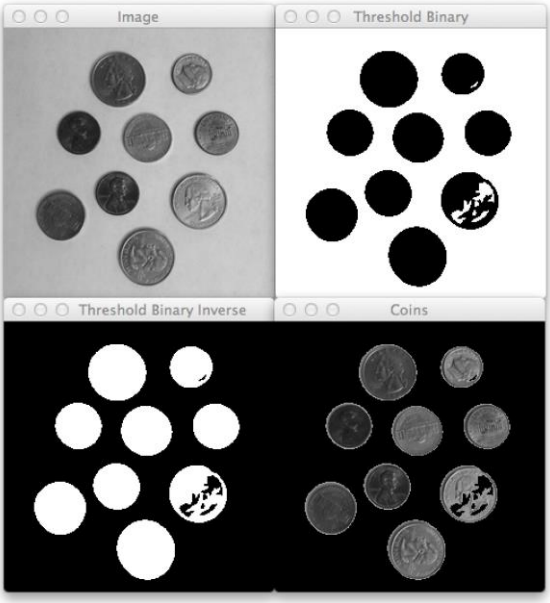

Figure 1: *Top-Left:* The original coins image in grayscale. *Top-Right:* Applying simple binary thresholding. The coins are shown in black, and the background is in white. *Bottom-Left:* Applying inverse binary thresholding. The coins are now white, and the background is black. *Bottom-Right:* Applying the inverse binary threshold as a mask to the grayscale image. We are now focused on only the coins in the image.

```
18 cv2.imshow("Threshold Binary Inverse", threshInv)
19
20 cv2.imshow("Coins", cv2.bitwise_and(image, image, mask = threshInv))
21 cv2.waitKey(0)
```

After the image is blurred, we compute the thresholded image on **Line 14** using the cv2.threshold function. This method requires four arguments. The first is the grayscale image that we wish to threshold. We supply our blurred image here.

Then, we manually supply our *T* threshold value. We use a value of $T = 155$.

Our third argument is our maximum value applied during thresholding. Any pixel intensity *p* greater than *T*, is set to this value. In our example, any pixel value that is greater than 155 is set to 255. Any value that is less than 155 is set to zero.

Finally, we must provide a thresholding method. We use the cv2.THRESH_BINARY method, which indicates that pixel values *p* greater than *T* are set to the maximum value (the third argument).

The cv2.threshold function returns two values. The first is *T*, the value we manually specified for thresholding. The second is our actual thresholded image.

We then show our thresholded image in Figure 1, *Top- Right*. We can see that our coins are now black pixels and the white pixels are the background.

On **Line 17** we apply inverse thresholding rather than normal thresholding by using cv2.THRESH_BINARY_INV as our thresholding method. As shown in Figure 1, *Bottom-Left*, our coins are now white, and the background is black. This is convenient as we will see in a second.

The last task we are going to perform is to reveal the coins in the image and hide everything else.

On **Line 20** we perform masking by using the cv2.bitwise_ and function. We supply our original coin image as the first two arguments, and then our inverted thresholded image as our mask. Remember, a mask only considers pixels in the original image where the mask is greater than zero. Since our inverted thresholded image on **Line 17** does a good job at approximating the areas the coins are contained in, we can use this inverted thresholded image as our mask.

Figure 1, *Bottom-Right*, shows the result of applying our mask – the coins are revealed while the rest of the image is hidden.

adaptive thresholding

One of the downsides of using simple thresholding methods is that we need to manually supply our threshold value, T. Not only does finding a good value of T require a lot of manual experiments and parameter tunings, but it's also not very helpful if the image exhibits a lot of range in pixel intensities.

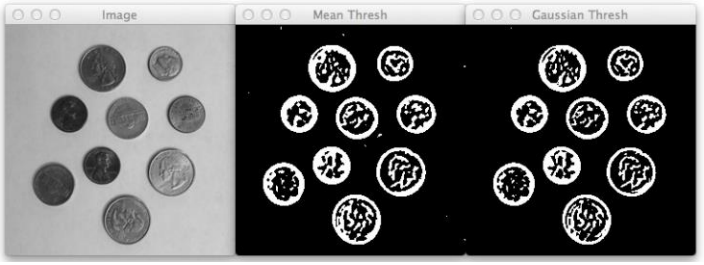

Figure 2: *Left:* The grayscale coins image. *Mid-level:* Applying adaptive thresholding using mean neighborhood values. *Right:* Applying adaptive thresholding using Gaussian neighborhood values.

Simply put, having just one value of *T* might not suffice.

To overcome this problem, we can use adaptive thresholding, which considers small neighbors of pixels and then finds an optimal threshold value of T for each neighbor. This method allows us to handle cases where there may be dramatic ranges of pixel intensities, and the optimal value of *T* may change for different parts of the image.

Listing 3: adaptive_thresholding.py

```
1  import numpy as np
2  import argparse
3  import cv2
4
5  ap = argparse.ArgumentParser()
6  ap.add_argument("-i", "--image", required = True,
7      help = "Path to the image")
8  args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11  image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
12 blurred = cv2.GaussianBlur(image, (5, 5), 0)
13 cv2.imshow("Image", image)
14
15  thresh = cv2.adaptiveThreshold(blurred, 255,
16      cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY_INV, 11, 4)
17 cv2.imshow("Mean Thresh", thresh)
18
19  thresh = cv2.adaptiveThreshold(blurred, 255,
20      cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY_INV, 15, 3)
21  cv2.imshow("Gaussian Thresh", thresh)
22 cv2.waitKey(0)
```

Lines 1-10 once again handle setting up our example. We import our packages, construct our argument parser, and load the image. Just as in our simple thresholding example above, we then convert the image to grayscale and blur it slightly on **Lines 11 and 12**.

We then apply adaptive thresholding to our blurred im- age using the cv2.adaptiveThreshold function on **Line 15**. The first parameter we supply is the image we want to threshold. Then, we supply our maximum value of 255, similar to the simple thresholding mentioned above.

The third argument is our method to compute the threshold for the current neighborhood of pixels. By supplying cv2.ADAPTIVE_THRESH_MEAN_C, we indicate that we want to compute the mean of the neighborhood of pixels and treat it as our T value.

Next, we need our thresholding method. Again, the description of this parameter is identical to the simple thresholding method mentioned above. We use cv2.THRESH_BINAR Y_INV to

indicate that any pixel intensity greater than *T* in the neighborhood should be set to 255. Otherwise, it should be set to 0.

The next parameter is our neighborhood size. This integer value must be odd and indicates how large our neighborhood of pixels will be. We supply a value of 11, indicating that we will examine 11 × 11 pixels re- regions of the image instead of trying to threshold the image globally, as in simple thresholding methods.

Finally, we supply a parameter simply called *C*. This value is an integer subtracted from the mean, allowing us to fine-tune our thresholding. We use $C = 4$ in this example.

The results of applying mean-weighted adaptive thresholding can be seen in the *middle* image of Figure 2.

Besides applying standard mean thresholding, we can also apply Gaussian (weighted mean) thresholding, as we do on **Line 19**. The order of the parameters is identical to **Line 15**, but now we are tuning a few of the values.

Instead of supplying a value of cv2.ADAPTIVE_THRESH_ MEAN_C, we instead use cv2.ADAPTIVE_THRESH_GAUSSIAN_C to indicate we want to use the weighted mean. We are also using a 15 × 15-pixel neighborhood size rather than an 11 × 11 neighborhood size as in the previous example. We also slightly alter our C value (the value we subtract from the mean) and use 3 rather than 4.

The results of applying Gaussian adaptive thresholding can be seen in the *right* image of Figure 2. There is little difference between the two images.

In general, choosing between mean adaptive thresholding and Gaussian adaptive thresholding requires a few ex- experiments on your end. The most important parameters to vary are the neighborhood size and *C*, the value you subtract from the mean.

By experimenting with this value, you can dramatically change the results of your thresholding.

otsu and riddler-calvard

Another way we can automatically compute the threshold value of *T* is to use Otsu's method.

Otsu's method assumes two peaks in the image's grayscale histogram. It then tries to find an optimal value to separate these two peaks – thus our value of *T*.

While OpenCV supports Otsu's method, we'll use the implementation by Luis Pedro Coelho in the Mahotas package since it is more Pythonic.

Let's jump into some sample code:

Listing 4: otsu_and_riddler.py

```
1  from __future__ import print_function
2  import numpy as np
3  import argparse
4  import mahotas
5  import cv2
6
7  ap = argparse.ArgumentParser()
8  ap.add_argument("-i", "--image", required = True,
9      help = "Path to the image")
10 args = vars(ap.parse_args())
11
12 image = cv2.imread(args["image"])
13  image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
14 blurred = cv2.GaussianBlur(image, (5, 5), 0)
15 cv2.imshow("Image", image)
16
17  T = mahotas.thresholding.otsu(blurred)
18 print("Otsu's threshold: {}".format(T))
```

On **Lines 1-5** we import the packages we will utilize. We have seen numpy, argparse, and cv2 before. We are now introducing Mahotas, another image-processing package.

Lines 7-12 then handle our standard practice of parsing arguments and loading our image.

As in previous thresholding examples, we convert the image to grayscale and then blur it slightly.

To compute our optimal value of *T*, we use the otsu function in the mahotas thresholding package. As our output will later show us, Otsu's method finds a value of $T = 137$ that we will use for thresholding.

Listing 5: otsu_and_riddler.py

```
19 thresh = image.copy()
20 thresh[thresh > T] = 255
21 thresh[thresh < 255] = 0
22 thresh = cv2.bitwise_not(thresh)
23 cv2.imshow("Otsu", thresh)
24
25 T = mahotas.thresholding.rc(blurred)
26 print("Riddler-Calvard: {}".format(T))
27 thresh = image.copy()
28 thresh[thresh > T] = 255
29 thresh[thresh < 255] = 0
30 thresh = cv2.bitwise_not(thresh)
31 cv2.imshow("Riddler-Calvard", thresh)
32 cv2.waitKey(0)
```

Applying the thresholding is accomplished on **Lines 19-22**. First, we make a copy of our grayscale image so that we have an image to threshold. **Line 20** then makes any values greater than *T* white, whereas **Line 21** makes all remaining pixels that are not white into black pixels. We then invert our threshold by using cv2.bitwise_not. This is equivalent to applying a cv2.THRESH_BINARY_INV thresholding type as in previous examples in this chapter.

The results of Otsu's method can be seen in the *middle* image of Figure 3. We can see that the coins in the image have been highlighted.

Another method to keep in mind when finding optimal values for *T* is the Riddler-Calvard method. Just as in Otsu's method, the

Riddler-Calvard method also computes an optimal value of 137 for *T*. We apply this method on **Line 25** using the RC function in Mahotas.thresholding. Finally, the actual thresholding of the image takes place on lines **27-30**, as in the previous example. Given that the values of *T* are identical for Otsu and Riddler-Calvard, the thresholded image in Figure 3 (*right*) is identical to the thresholded image in the center.

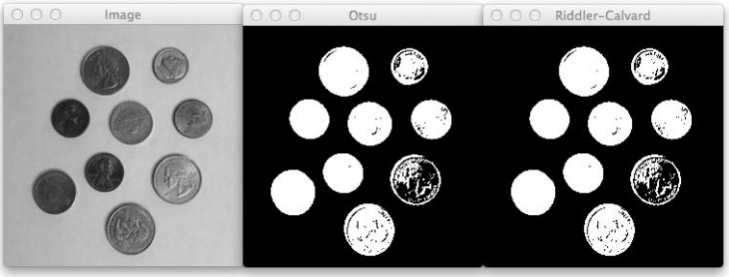

Figure 3: *Left:* The original grayscale coins image. *Middle:* Applying Otsu's method to find an optimal value of *T*. *Right:* Applying the Riddler-Calvard method to find an optimal value of *T*.

Listing 6: otsu_and_riddler.py

```
Otsu's threshold: 137
Riddler-Calvard: 137
```

Part I. Simulations (all examples) Individual

Part II. Group Activity

Task 1: Conduct an experiment by applying simple thresholding with varying threshold values (T) on a grayscale image, using only simple thresholding method. Observe and document how changes in the threshold value affect the segmentation results, particularly focusing on the clarity and accuracy of object boundaries.
Question: How does altering the threshold value in simple thresholding impact the segmentation of objects in the image, and what challenges arise when selecting an optimal threshold for images with diverse intensity levels?

Task 2: Write a Python script to apply different thresholding methods (simple, adaptive mean, adaptive Gaussian, Otsu, and Riddler-Calvard) to a grayscale image. Compare the results by visually inspecting how each method segments the image and handles varying pixel intensities.
Question: How do the results of different thresholding methods compare in terms of accurately segmenting the objects from the background, and which method performs best for images with uneven lighting or complex backgrounds?

Task 3: A company wants to develop an automated document scanner that can quickly digitize printed documents. They need to convert these images into binary (black and white) form, where the text appears clearly, and the background is removed. Your task is to use image thresholding techniques to accomplish this.

The system should be able to take a grayscale image of a printed document and transform it into a clear, binarized image. The document contains text printed on slightly uneven paper with some noise (e.g., shadows, creases). This makes it challenging to use a simple fixed threshold. Otsu's and Riddler-Calvard methods will allow you to calculate an optimal threshold value to segment the image efficiently.

Provide two (2) images of printed documents that contain some noise. Apply Otsu's and Riddler-Calvard and compare the results.

Discussion: Compare and contrast the thresholding methods: Simple, Adaptive, Otsu, and Riddler-Calvard.

Task 4: You are working as a data scientist in a healthcare startup that specializes in medical imaging. Your team is developing a system to automatically detect tumors in MRI scans. The first step is to segment the image to isolate regions of interest, such as tumors, from the background. Tumors usually have different intensities than the surrounding healthy tissue, so image thresholding can be an effective technique for segmentation.

Instructions:

1. **Understand the Problem:** Brain MRI images often contain varying intensities, where the tumor regions may appear brighter or darker than the surrounding tissue. Your task is to apply Otsu's and Riddler-Calvard thresholding techniques to detect these tumors.
2. **Get the Image:** You will be provided with a grayscale MRI scan of a brain with a possible tumor.
3. **Implement Thresholding Techniques:** Apply Otsu's method and Riddler-Calvard thresholding to segment the potential tumor from the MRI image. Use Python, OpenCV, and NumPy for this activity.

Part III. Individual Activity (Assignment)

Choose a diverse set of images for analysis, including:
Images with simple, well-lit objects against a uniform background.
Images with complex backgrounds, varying lighting conditions, and overlapping objects.
Images with high noise levels, such as salt-and-pepper noise.
Implementation:

Write a Python program using OpenCV to apply each of the following thresholding techniques:

- Simple Thresholding with manually selected thresholds.
- Adaptive Thresholding (both mean and Gaussian).
- Otsu's method.

Apply each thresholding method to all selected images.

Create visual comparisons by displaying the original images alongside the thresholded images for each method. Analyze the effectiveness of each method in segmenting the objects from the background, preserving details, and handling noise.
Document the results with detailed observations, noting where each method succeeds or fails, especially in challenging conditions.

Question:

Which thresholding method provides the most reliable and accurate segmentation across a variety of image conditions, and how do preprocessing steps like blurring influence the effectiveness of each method?

References:

Rosebrock, A. (2015). ***Practical Python and OpenCV+ Case Studies: An Introductory, Example Driven Guide to Image Processing and Computer Vision*****. Adrian Rosebrock, PyImageSearch.**
Rosebrock, A. (2017). ***Deep learning for computer vision with python: Starter bundle*****. PyImageSearch.**
Rosebrock, A. (2017). ***Deep learning for computer vision with python: ImageNet Bundle*****. PyImageSearch.**
Joshi, P. (2015). ***OpenCV with Python by example*****. Packt Publishing Ltd.**
Iyer, B., Rajurkar, A. M., & Gudivada, V. (2020). ***Applied computer vision and image processing*****. Springer Singapore.**