Coding Challenge - Phase 2

Pierre-Louis Cedoz
plcedoz@stanford.edu

1 Part 1: Parse the o-contours

Question: Using your code from Phase 1, add the parsing of the o-contours into your pipeline. Note that there are only half the number of o-contour files as there are i-contour files, so not every i-contour will have a corresponding o-contour. After building the pipeline, please discuss any changes that you made to the pipeline you built in Phase 1, and why you made those changes.

I did some modifications to the code to make it more clear and efficient:

- **utils.data:** I updated the function **utils.data** so that it takes into account the o-contours. The main difference with the code from part 1 is that I retain only the image slides that have both an inner contour and an outer contour. The **labels** variable is now a dictionary that associates for every **image_file** a list of the file of the inner-contour and the file of the outer contour:
$$labels[image\_file] = [i\_contour\_file, o\_contour\_file] \ for \ all \ image\_file \tag{1}$$
Since not every i-contour will have a corresponding o-contour, half of the samples are lost during this operation. However, if we wanted to retain the i-contours that don't have a corresponding o-contour, it would create an imbalance in the batches of data: if a batch of data contains less outer contours than inner-contours, it is impossible to create a target array. One solution to that would be to generate the missing o-contour file. To do so, an interpolation method could be implemented to generate the missing labels from their nearest neighbors (k-nearest neighbors) or a method similar to the one developed in part 2 where the outer contour would be generated starting from the inner contour.

- **utils.generator** In the first phase of the coding challenge, I had implemented a pipeline using the **tf.data** API that improve the performance of feeding data into a neural network using queues and operations in the back-end. However, since the phase 2 doesn't include building a neural network, I reimplemented the data pipeline using a generator, which is more straightforward. The generator class (**utils.generator**) takes as input the filenames and labels generated in **utils.data** and yields a batch of data at every training step. The samples are also sampled at every step.

  The main difference with part 1 is that we know have 2 labels: inner contours and outer contours. The **parse_function** is the same but the generator yields a list of 2 labels: **an array of inner contours and an array of outer contours.**

- **Pre-processing:** A normalization step was implemented while generating the images so that all intensities are between 0 and 255. This will be useful in part 2 for the inner contour prediction.

- **Model class**: The **model** class was deleted because I won't be training any Machine Learning model.

I implemented a function called **plot_generator** in **utils.plots** to check that everything was working and the output is shown in Figure 1:

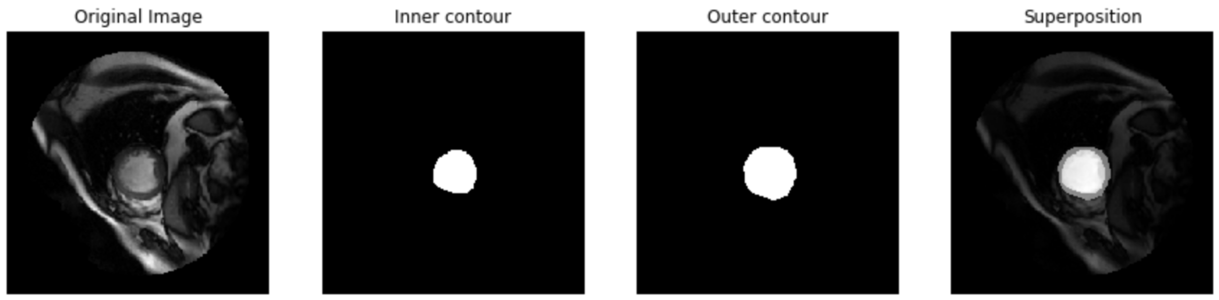

Figure 1: Illustration of the inner-contour and outer-contour

I inspected all the examples and realized that everything seems to work except for all the outer contours of **SCD0000501** samples. An example of such anomaly can be found in Figure 2. I decided to remove the examples from **SCD0000501** in subsequent analysis since part 2 relies on these outer contours.

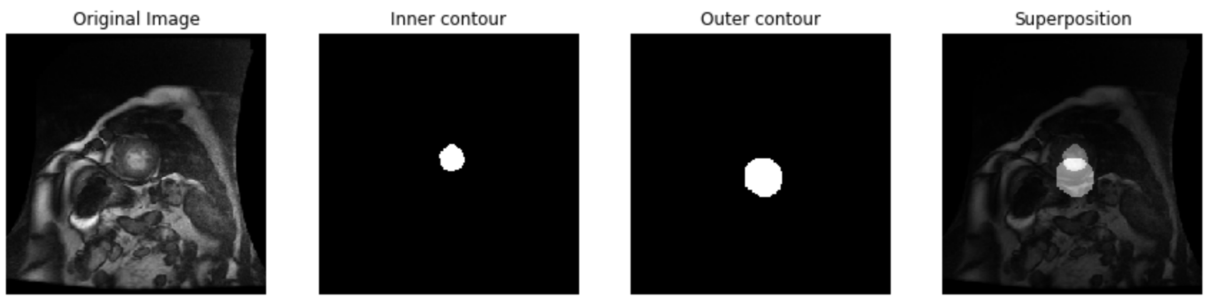

Figure 2: Illustration of the inner-contour and outer-contour for SCD0000501 sample

All figures can be found in the **output** folder.

2 Part 2: Heuristic Left Ventricle Segmentation approaches

Most of the code for this part is in **segmentation.py**.

2.1 Q2.1 - Intensity-based prediction

Question: Lets assume that you want to create a system to outline the boundary of the blood pool (i-contours), and you already know the outer border of the heart muscle (o-contours). Compare the differences in pixel intensities inside the blood pool (inside the i-contour) to those inside the heart muscle (between the i-contours and o-contours); could you use a simple thresholding scheme to automatically create the i-contours, given the o-contours? Why or why not? Show figures that help justify your answer.

A lot of the inspiration for this work came from these 2 webpages:

- *http : //homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/MORSE/threshold.pdf*
- *http : //scikit − image.org/docs/0.12.x/auto_examples/xx_applications/plot_coins_segmentation.html*

Segmentation involves separating an image into regions (or their contours) corresponding to objects. We usually try to segment regions by identifying common properties. Or, similarly, we identify contours by identifying differences between regions (edges). The simplest property that pixels in a region can share is intensity. So, a natural way to segment such regions is through **thresholding**, the separation of light and dark regions. The idea proposed in this project is to look at the distribution of pixel intensities inside the blood pool and compare it to the distribution of pixels in the heart muscle. **Figure 3** shows a visualization of these distributions:

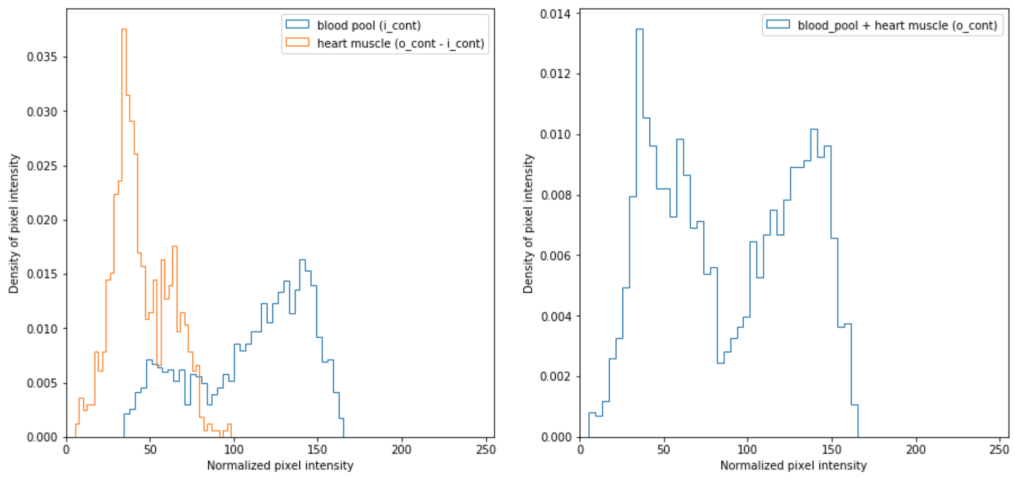

Figure 3: Distribution of pixel intensities (i-cont=inner contour, o-cont=outer contour)

There are two groups of pixels (blood pool and heart muscle), with different range of values. What makes thresholding difficult is that these ranges usually **overlap** (see **Figure 3**). In order to minimize the error of classifying a blood pool pixel as a heart muscle one or vice verso, I tried to minimize the area under the histogram for one region that lies on the other regions side of the threshold. Given the outer contour, I have access to the intensity histogram for the combined regions: blood pool and heart muscle (right of **Figure 3**). From this density histogram, the goal is to extract the histograms for each region: blood pool and heart muscle (left of **Figure 3**). To extract the 2 distributions I used 2 techniques:

- **Kmeans clustering:** The first approach is to consider the values in the two regions as two clusters. We want to find a threshold by clustering the histogram. The idea is to pick a threshold such that each pixel on each side of the threshold is closer in intensity to the mean of all pixels on that side of the threshold than the mean of all pixels on the other side of the threshold.
- **Mixture Modeling:** Another way to minimize the classification error in the threshold is to suppose that each group is Gaussian-distributed. Each of the distributions has a mean and a standard deviation independent of the threshold we choose.

In both cases, the method will classify each pixel as belonging to one cluster (or gaussian) depending on a threshold. The thresholds computed are given in Figure 4. As we can see in Figure 4, these algorithms are able to identify good clusters and their results are very similar. However in some cases (right case in Figure 4), the 2 distribution are not very distinct so the model might have some limitations. However, most cases in this dataset are well separated in 2 clusters.

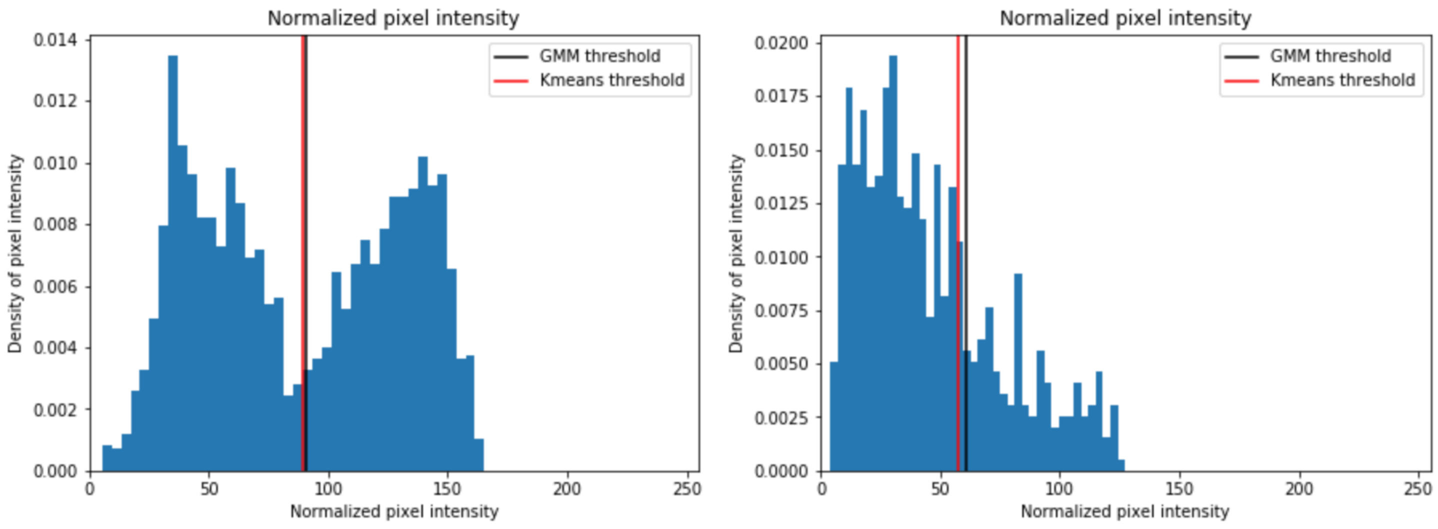

Figure 4: Threshold determination for kmeans and Gaussian Mixture Model

I was also thinking of finding the local minimum in the histogram for the threshold. However there are 2 main issues with this method:

- The place of minimum overlap (the place where the misclassified areas of the distributions are equal) is not is not necessarily where the valley occurs in the combined histogram. This occurs, for example, when one cluster has a wide distribution and the other a narrow one.
- The sum of two separate distributions, each with their own mode, may not produce a distribution with two distinct modes (see **Figure 4**).

Results of intensity-based segmentation: Once the threshold are find using one of the 2 methods, the next step is to compute the inner-contour. To do that, I restrained the image to the pixels within the outer contour and I applied the threshold on them. The results are shown on **Figure 5**. All results can be found in the **output** folder.

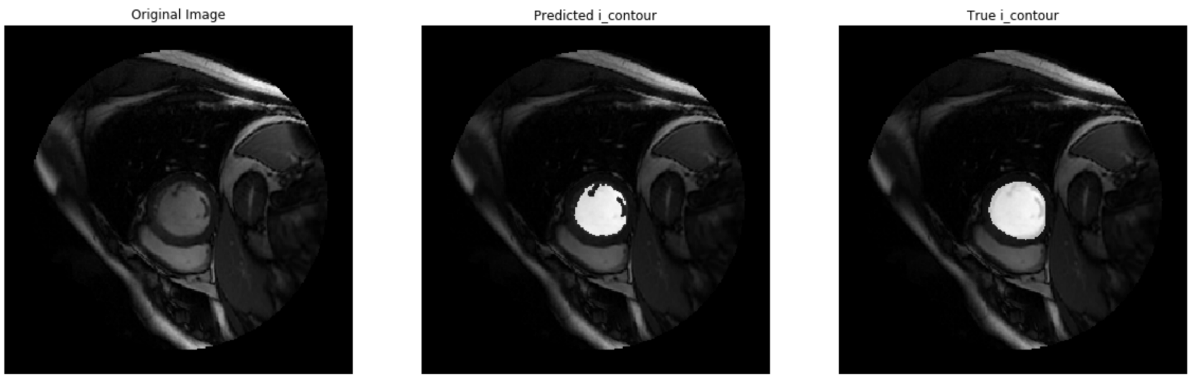

Figure 5: Output of the Intensity-based prediction model

Post processing: To improve the performance of the algorithm I also applied some post-processing steps from the **scipy** package that are mathematical morphology operation:

- **binary_dilation:** Operation that uses a structuring element for expanding the shapes in an image.
- **binary_fill_holes:** Operation that fills the holes in binary objects.

The results are shown in **Figure 6** for the same example as in **Figure 5** where we can see that the small anomalies are corrected.

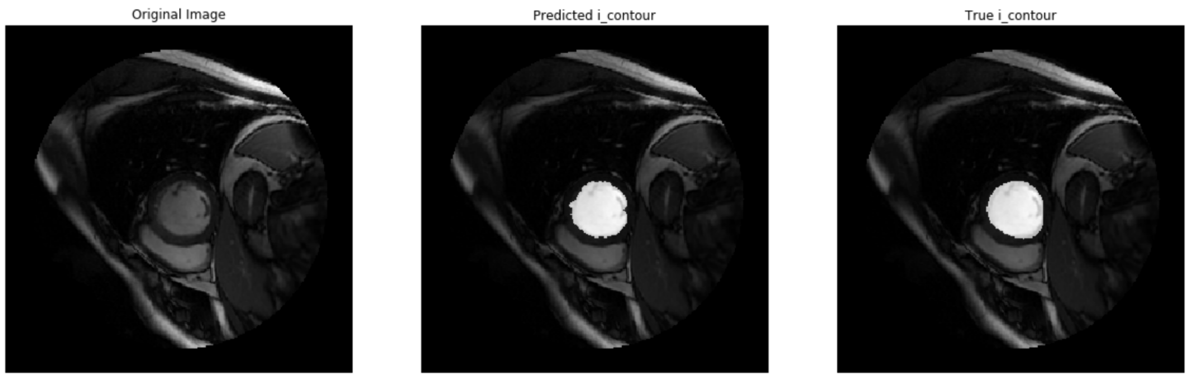

Figure 6: Post-processing of the segmentation model

To find the outer contour from the segmentation, I used the function **find_contours** from **skimage.measure**.

Issues with Intensity-base prediction: The major problem with intensity-based thresholding is that we consider only the intensity, not any relationships between the pixels. There is no guarantee that the pixels identified by the thresholding process are contiguous. We can easily include extraneous pixels that arent part of the desired region, and we can just as easily miss isolated pixels within the region (especially near the boundaries of the region). These effects get worse as the noise gets worse, simply because its more likely that a pixels intensity doesnt represent the normal intensity in the region. When we use thresholding, we typically have to play with it, sometimes losing too much of the region and sometimes getting too many extraneous background pixels.

2.2 Q2.2: Other heuristic-based methods

Question: Do you think that any other heuristic (non-machine learning) based approaches, besides simple thresholding, would work in this case? Explain.

As we saw in the intensity-based model, there are some limitations to only using the intensity to extract the inner contour.

2.2.1 Edge-based segmentation

Another idea would be to delineate the inner contour using edge-based segmentation. To do this, we get the edges using the **Canny edge-detector** from the **skimage.feature** package. The Process of Canny edge detection algorithm was introduced by John F. Canny [1] and it can be broken down to 5 different steps:

- Apply Gaussian filter to smooth the image in order to remove the noise
- Find the intensity gradients of the image
- Apply non-maximum suppression to get rid of spurious response to edge detection
- Apply double threshold to determine potential edges
- Track edge by hysteresis: Finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong edges.

Therefore, this method involves gradient calculation so as opposed to the intensity-based method, it includes the relationships between the pixels. The contours are then filled using mathematical morphology like in the previous part: **binary_dilation** and **binary_fill_holes**. The edge-based method is very performant on this dataset (sometimes even better than the ground truth) as we can see in **Figure 7**:

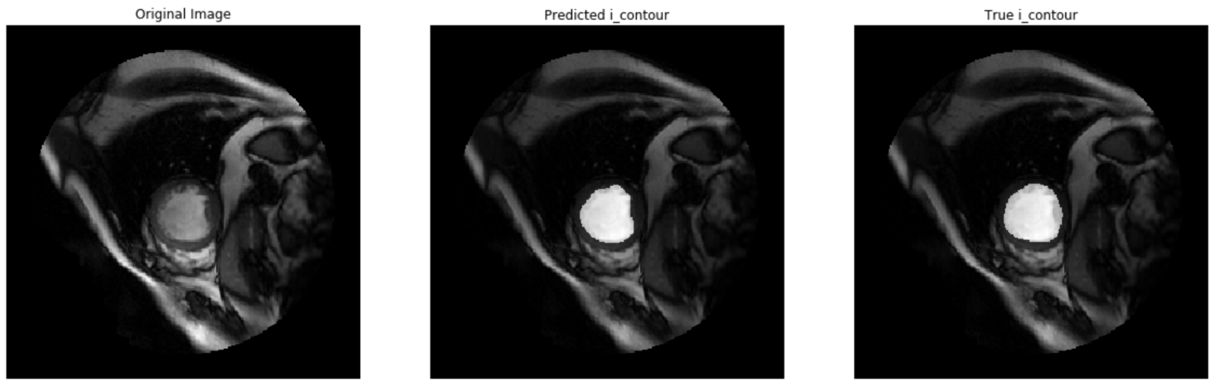

Figure 7: Canny-edge segmentation

Limitations : However, the edge-based method fails when contours are not perfectly closed and are not filled correctly (see **Figure 8**). The post-processing methods can limit this issue but there is always a trade-off because the post-processing methods alter the prediction of the algorithm.

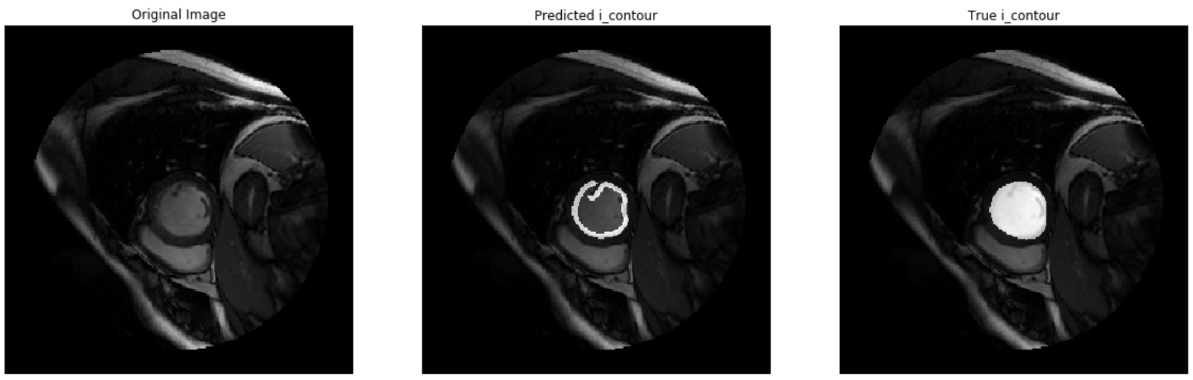

Figure 8: Failing Canny-edge segmentation

All results for the canny-edge segmentation can be found in the **output** folder.

2.2.2 Combining the 2 methods: Thresholding Along Boundaries

We could think of combining the intensity-based method and edge-based method. We can first apply some boundary-finding method (edge-based methods) and then sample the pixels for the thresholding only where the boundary probability is high.

Thus, our threshold method based on pixels near boundaries will cause separations of the pixels in ways that tend to preserve the boundaries. Other scattered distributions within the object or the background are of no relevance. I didn't implement this method because of the time constraints.

2.2.3 Local Thresholding

Another problem with global thresholding (intensity-based methods) is that changes in illumination across the scene may cause some parts to be brighter (in the light) and some parts darker (in shadow) in ways that have nothing to do with the objects in the image. We can deal, at least in part, with such uneven illumination by determining thresholds locally. That is, instead of having a single global threshold, we allow the threshold itself to smoothly vary across the image.

However, I didn't implement local thresholding here because the image was small (only the outer contour) and contains roughly only one subject (the blood pool).

2.2.4 Deep Learning

As described in [2], there exist plenty of other methods for the segmentation of the left ventricle. In particular, **Convolutional neural networks** have become a preferred tool for image segmentation. A straightforward way for LV segmentation is to train the deep network on ground truth contours to classify pixels. The pipeline built in Phase 1 of the coding challenge could have been used in such a way.

References

[1] John Canny. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, (6):679–698, 1986.

[2] Arseny Krasnobaev and Andrey Sozykin. An overview of techniques for cardiac left ventricle segmentation on short-axis mri. 8:01003, 01 2016.