Task 1: Contour Normalization Algorithm for Segmentation Masks

Created by Nguyen Quoc Thai

Date: 28/04/2024

Table of Contents

1. Algorithm explanation

Input:

- *Raw image*: to draw the contours for visualization.
- *Binary mask*: the mask for the object segmentation.
- *point_number*: The number of point on the contour is needed.

Output:

- Each contour consistently has a predetermined number of points.

Approach:

Since I used the binary mask for the car in the image with the object segment's color as white and the background as black then when loading the mask I need to do the bitwise_not to convert the color (from white to black and opposite) as the cv2 requirement. Then, I use the cv2. findContours to find the contour in the mask. Each mask can contain many object segments and each of them, has its own contour. For each contour found by the function, I will compare them with the point_number to decide whether to reduce or increase the points in the contour.

Pseudo code:

- <u>*Point reduction:*</u>

Step	**Explanation**
Split the points in the contours into 2 equal parts. With: - First half from 0 to int(length/2). - The second half is the rest.	Since the Ramer-Douglas-Peucker algorithm picks the start and the endpoint, if we input as the findContours return, the start point and end point are near together. It can lead to the skewed deletion of one side of the object's contour. So perform the algorithm on each part separately then combine.
number_of_retain_point_of_each_part = point_number/2	The reduction points will be divided equally for each part to ensure the balance of the contour.
For each part, reduce_contours_half function does: - Calculate the weight (distance) from each point between (start point, end point).	Using recursive algorithm to design the Ramer-Douglas-Peucker algorithm. By this method we can calculate the weight (epsilon) of each point.

- Store the furthest point into a weights dictionary with key: weight (distance), value: coordinate (x,y). - Do the same with the recursive algorithm for 2 small parts (start point, furthest point) and (furthest point, end point) until end == start. - Save the start and end points to the result list. - Sort weights dictionary by key descending - Save the first (Number of retained points for each part – 2) point to the result list. - Return result belong to the first/second half	Store in a dictionary helps to reserve mem by some points really close to the line (weight = 0), so only need to store one of them since we do not need to care about them. With Ramer-Douglas-Peucker only the point has the weight* (calculation method below)> predefined threshold (impact on the shape of contour), so to keep the important point, I sorted weights dictionary by descending will move the point with high weight (maybe they are the corner that defines the shape of the contour) to the top of the dictionary and base on predefine number of retain point to save the top (number_of_retain_point_of_each_part – 2) since we need to store start and end point first.

result = []
result.extend(first_half)
result.extend(second_half)
 	
Convert result to numpy array and expand dim from (Point_number, 2) to (Point_number, 1, 2).
	Combine the first and second part then convert to np array and expand the dimension as the cv2.drawContours requirement.

Use cv2.drawContours to draw point of contour on the image	

Calculation method:

"The algorithm recursively divides the line. Initially it is given all the points between the first and last point. It automatically marks the first and last point to be kept. It then finds the point that is farthest from the line segment with the first and last points as end points; this point is obviously farthest on the curve from the approximating line segment between the end points" – wikipedia.

- *Point increasement:*

Step	Explanation
Split the contour into one of three options by divide the number of points: - Half left – half right - Half top – Haft bottom - 4 equal parts divided by clockwise	Since the contour has various and complicated shapes it is hard for the point interpolation. By dividing it into smaller it will be easy for the algorithm to interpolate.

Calculate the need point to add:
point_need = point_number – len(contour)
point_need_of_each_part = point_need/2 if divide half-half, point_need/4 if divide 4 part
	The additional points will be divided equally for each part to ensure the balance of the contour.

For each part, generate_contour_points function does:	
- Sort the points in this part by x coordinate ascending.	Since the function provided by scipy.interpolate needs the x coordinate sorted by ascending.
- Split x, y to 2 numpy arrays.	
- Input to one of three line interpolation: linear interpolation, cubic spline, and 1d interpolation to get the function.	Since I tried three types of point interpolation, with the small test set they provided a similar result so I included all. For the 1d: x and y are arrays of values used to approximate some function f: y = f(x). For cubic spline: Interpolate data with a piecewise cubic polynomial which is twice continuously differentiable and it was proved to work well with complicated contours. For linear: Returns the one-dimensional piecewise linear interpolant to a function with given discrete data points (xp, fp), evaluated at x.
- From x array → x_max, x_min - Number of x interpolation: num = x_max – x_min, step = 1 - x_interpolation = start + np.arange(0, num) * step	Since we do not know how much interpolation point is enough and which is vital to retain, I choose to interpolate all of the point in the min-max x coordinate range.
- while len(x_ interpolation) < point_need_of_each_part:	Check to ensure add enough points to the reduced algorithm later.

x_interp = np.vstack((x_interp, x_interp)) - Input x_interpolation to function abow to get y_interpolation - Input x_interpolation, y_interpolation, start and end point of this contour to function retain_increasement_contours_part: - Calculate the weight (distance) from each point between (start point, end point) to get furthest point. - While furthest point's distance is available in weights: furthest point's distance -= 0.1 - Store this point into a weights dictionary with key: weight (distance), value: coordinate (x,y). - Do the same with the recursive algorithm for 2 small parts (start point, furthest point) and (furthest point, end point) until end == start. - Save the points of the provided contour to result list. - Sort weights dictionary by key ascending. - Save the first (point_need_of_each_part) point to the result list. - Return result.	Perform the same task as Ramer-Douglas-Peucker algorithm at first. If the weight already existed, minus it by 0.1 to store all the points into the weights dictionary for the later choice. Instead of retaining which have the high weight, I sorted by ascending and stored the points that have small weights (which will be close to the start and end point provided) so they may be the perfect interpolation points.

result = []
# if divide by half - half 	
result.extend(first_half)
	Combine parts then convert to np array and expand the dimension as the cv2.drawContours requirement.

result.extend(second_half)
 	
# if divide by 4 part 	
result.extend(p1)
result.extend(p2)
result.extend(p3)
result.extend(p4)
 	
Convert result to numpy array and expand dim from (Point_number, 2) to (Point_number, 1, 2).	

Use cv2.drawContours to draw point of contour on the image	

2. Challenges encountered and solution

Challenges	**Solution**
For the Ramer-Douglas-Peucker algorithm if the weight < threshold → remove this point, so it is hard to control the number of retain point.	Sort weight by descending and pick top highest weight as the point_number requirement.
If input the whole contour, the start point, and end point are near to each other. Therefore whenever apply the Ramer-Douglas-Peucker algorithm, it is easy to erase a point that is skewed in a certain part.	Divide the contour into the smaller part as I tested, it is now better with half-half.
With Ramer-Douglas-Peucker algorithm only care about high weight.	Use dictionary to store → reduce memory need.
Increase points: do not know which point in the interpolation point is vital to retain.	Interpolate more than need and modify the Ramer-Douglas-Peucker algorithm to retain only low weight point.
The points interpolated are skewed to one side of the contour.	Divide the contour into smaller part and perform the point interpolation for each part. The result that I tested show that the smaller divided part, the better result.
Do not know how to split the contour to be suitable.	Perform the test again and again. Test result image are saved in img folder with format: "interpolation_algorithm+split_type". Ex: 1d_left_right.jpeg
Some interpolated points are in the middle of the image	Test with three type of interpolation algorithm and split strategy.

	→ There is a little bit difference between them. It require more provided points at first to perform the interpolation better and this condition is not decided by us. → still a pain point.

3. Potential improvements

For both reduce and increase points, the algorithm works well will simple contour so dividing them into smaller parts but not too small (since it takes more time to perform a task) will be better.

To reduce points, we can apply some Clustering Algorithms to identify clusters of points that are close together and keep only a few representative points for each cluster. The remaining points can be discarded. Or we need to reduce but retain the shape of the contour especially the corner of the contour so I propose an approach with Machine learning. By collecting a large amount of data, for example, maybe 3 points representing a corner of the contour. We train the ML model to classify whether a group of 3 points is a corner or not to decide to keep or remove.

For the increase points, from my experience, I also suggest using the RNN or Transformer model by teaching them the pattern of the past so they can provide the next data point. Another approach is GAN by feeding them with their own data with the provided pattern, then they can provide the Synthetic Data and we have to do some post-processing to decide if the generation data point is suitable or not.

---Thank you for reading--