COMS4036A & COMS7050A
Computer Vision

Nathan Michlo, Devon Jarvis, Dr Richard Klein

Lab 6: Contours, Shape & Texture Models

Overview

In the previous lab, Quarantino asked you to help design a method to construct the ground truth dataset of puzzle piece masks from multiple labelled images. Since then Tino has been trying to solve the puzzle by hand but has found it is going to take longer than expected to go through all $4^{48} \cdot 48! \approx 9,83531722 \cdot 10^{89}$ possible permutations of his 48 piece puzzle. He suspects that the shapes of puzzle pieces obtained from the masks are unique enough to match the corresponding edges for fitting pieces together, or at least figure out which pieces and edges won't fit together. For Tino's sake help him figure out a way to reduce this search space so he doesn't have to keep trying different permutations of puzzle pieces until all the protons in the universe have inconveniently decayed[1] [2].

Instructions

The goal of this lab is to implement a system to extract the shape information from puzzle piece masks. We first extract the contours, from these contours we then build a shape model for each of the four sides, and finally use these side models to naïvely match them together using k-nearest neighbours.

The idea is that if two puzzle pieces can fit together, the shape or outline of the connecting sides will match, or rather the contours of those sides will be equivalent.

- Various python packages are used throughout code snippets in this handout, you can install them all with the following command. Python 3.6+ was used when testing the lab snippets.

```
pip3 install natsort numpy imageio scikit-image opencv-python scikit-learn \
             networkx matplotlib
```

- You have been given a new pre-processed dataset `"puzzle_corners_1024x768.zip"`. Unzip the dataset and load all 48 images and mask pairs. Do not resize the images.

```
import numpy as np
import imageio
from glob import glob
from skimage import img_as_float32
from natsort import natsorted

path_pairs = list(zip(
    natsorted(glob('./puzzle_corners_1024x768/images-1024x768/*.png')),
    natsorted(glob('./puzzle_corners_1024x768/masks-1024x768/*.png')),
))
```

[1] Graphical timeline from the Big Bang to the Heat Death of the universe.

[2] XKCD 2315: Eventual Consistency

```
imgs = np.array([img_as_float32(imageio.imread(ipath)) for ipath, _ in path_pairs])
msks = np.array([img_as_float32(imageio.imread(mpath)) for _, mpath in path_pairs])
```

1 Find Contours

The first step is to find the contours of all the puzzle pieces using our masks which can then be further processed in the next steps to extract the shape models for the 4 individual sides.

Hint: A contour in this lab is a 2-dimensional data structure defined as an ordered array of points. In python we usually define points as $(y,\ x)$, however, with `cv2` this is $(x,\ y)$. Line segments are defined by pairs of consecutive points in these contour arrays. A contour can be open or closed, a closed contour is effectively a polygon, while an open contour is just a set of line segments. A closed contour thus has a line segment joining the last point $[-1]$ to the first point $[0]$ in the array, some `cv2` functions can take the argument `closed=True`.

Contours returned by `cv2` have 3 dimensions, however, for this lab, the 2nd dimension can be ignored. Thus, you may want to reshape these contour arrays from $(-1, 1, 2)$ to $(-1, 2)$ and vice-versa. `cv2` functions often take in or return lists of contours, make sure to check the docs for the different functions.

1.1 Write a function called `get_puzzle_contour(mask)` that takes in a binary mask of a puzzle piece and returns a single contour array of that puzzle piece. A list of contours in a mask can be found using `cv2.findContours`[3]. Since multiple contours are returned, assume that the correct contour corresponding to the boundary of the puzzle piece is closed and the one with the largest area.

Hint: Pass `cv2.RETR_LIST` and `cv2.CHAIN_APPROX_SIMPLE` to `cv2.findContours`, check the docs for their meaning. You can sort all the contours by their area using `cv2.contourArea`.

1.2 To ensure we can tell if the side of a puzzle piece is sunken or protruding, later on, we need the points in the contour to be ordered in a clockwise direction. Code a new function called `get_clockwise_contour(contour)` that returns a copy of the contour, but reversed if points in the contour are ordered anti-clockwise.

Hint: Call `cv2.contourArea` with the named parameter `oriented=True`[4] to check the orientation of a contour. If the area returned is negative you know the ordering of points needs to be reversed. A useful way to reverse lists, especially numpy arrays, is indexing an axis with `rev = arr[::-1]`

1.3 Tino doesn't like the idea of using libraries if we don't know what they are actually doing under the hood. Do some research on how `cv2.findContours` works as well as how the `oriented=True` version of `cv2.contourArea` detects orientation. Give a short (maximum one paragraph) explanation for each.

1.4 Finally plot the result of any three reoriented boundaries (contours) overlaid on top of their corresponding original RGB puzzle piece images. Do not use the same puzzle pieces as the ones used in figure 1 and 2.

[3] Getting started with OpenCV contours.

[4] Docs for cv2.contourArea

2 Shape Models

With our normalised boundaries for each puzzle piece, we can now find the non-closed contours for the sides of the pieces and use this information to detect what type of puzzle pieces we are working with: corners, edges or interior pieces. Finally, we can normalise these side contours for matching with one another in the next step.

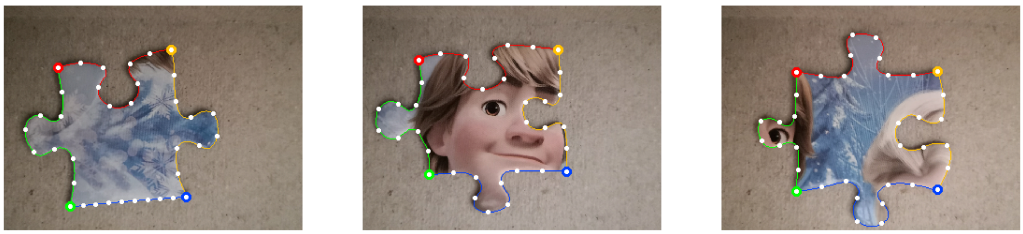

Figure 1: *Visualisations of contours for three puzzle pieces that fit together. Original side contours are colourised (like 2.1.2), the first point of each side contour is marked with the large dot of the same colour (like 2.2.1), interpolated/simplified side contours are marked in white dots (like 2.2.2).*

2.1 Extract Sides

The easiest way to extract the contours for all four sides of a puzzle piece is to use corner information. However, most simple methods for finding the corners can be unreliable in practice. Tino has graciously given up some of his precious time[5] to hand-label all the corners of the puzzle pieces using the python package `label-studio`.

2.1.1 Load the JSON corner data for each puzzle piece using the python `json` package. The format of the data is two lists with corresponding elements, the first containing names of the original RGB images, the second containing float arrays of shape $(4, 2)$ that correspond to the 4 corner points. The points in the corner arrays indicate coordinates (x, y) like the contours from `cv2`, not (y, x) as usual in python. Each x and y coordinate is actually the ratio along that axis of an image, thus, to obtain the original pixels of the corners, multiply each ratio respectively by the width and the height of the target image.

```
import json
with open("corners.json", mode="r") as f:
    names, corner_ratios = json.load(f)
```

2.1.2 Next we need to extract the four side contours for each puzzle piece using its contour and four ground truth corners. Code a function called `extract_sides(contour, corners)` that returns a list of side contours after the following steps:

- Find the index of the nearest point in the contour to each corner point using euclidean distance. Call this list of four indices `corner_indices`. Ask yourself why using the approximation mechanism `cv2.CHAIN_APPROX_SIMPLE` in question 1.1 might help here?

  Hint: `corner_dists = np.linalg.norm(contour - corner, axis=-1)`

- Sort `corner_indices` in case the 4 indices are not consecutive due to matching, as we need them to correspond to a valid closed contour.

[5] See footnote 1 on the first page.

- Construct the 4 non-closed contours for each side of the puzzle piece. The first and last points in these side contours can be obtained using paired indices from `corner_indices`, with all the points along the contour between these corner points included. Make sure to preserve the clockwise nature of the points along the contour from question 1.2. The list of sides should have the property that indexing for side n should mean that side $n+2$ opposite it, and $n+1$ is to the right or clockwise.

  Hint: Slicing for these side contours is non-trivial due to the last case going from corner -1 to 0 in `corner_indices`. The easiest way to handle all cases is with modulus to get consecutive corner index pairs `(i, j) = corner_indices[[a, (a+1)%4]]`. Since $i < j$ due to earlier sorting, shift the index of point i in the contour to index 0 using `np.roll` along `axis=0` and finally slice a segment of length `[0:j-i+1]`. It is possible for `j-i+1` to be negative.

2.1.3 As you did for question 1.4 with the same puzzle pieces, plot the quad formed by the corners with the colour `(255,255,255)`, overlaid by the points of the 4 sides extracted, colouring the different sides with red `(255,0,0)`, green `(0,255,0)`, blue `(0,64,255)`, yellow `(255,192,0)`, do not use the same puzzle pieces as in figure 1 and 2.

2.2 Normalise Sides

With our sides extracted, we now need to normalise them for matching using k-nearest neighbours by creating a shape model. This consists of two steps: transform by translating, scaling, and rotating, followed by interpolation to get the same number of points for each side. These interpolated side contours can be flattened into our feature vectors.

2.2.1 Code a function called `transform_puzzle_side(contour)` that takes in a single side contour and returns the translated, scaled and rotated version to be used for matching.

Imagine there is an arbitrary line running from the first point 0 to the last point -1 in the contour, we want to find the transformations to translate this line so that it is zero centred, then scale it so that it is of length 2, then rotate it so the line is on the x-axis with the first point pointing along the negative x-axis or towards the left.

After applying these transformations[6] to the entire side contour, the first point at index 0 should have the coordinate `(x, y) == (-1, 0)`, and the last point at index -1 should have the coordinate `(x, y) == (1, 0)`. Sunken sides should point up while protruding sides should point down.

Note that we don't try rotate sunken sides by $180°$ so that they correspond to protruding sides, as this separation between sunken and protruding sides is very useful for matching with knn! If you look at matching sides of puzzle pieces in figure 2, you can verify this final transform. Note too that we would also need to reverse the order of points in the sunken array if we wanted them to correspond. Ask yourself why this is.

Hint: Don't forget about the numpy functions `np.arctan2` and `np.linalg.norm`, as well as the python operator `@` for matrix multiplication which may be useful for rotations. Make sure to scale the x and y values by the same ratio as we want proportions to remain the same.

[6] Wikipedia page on 2D transforms.

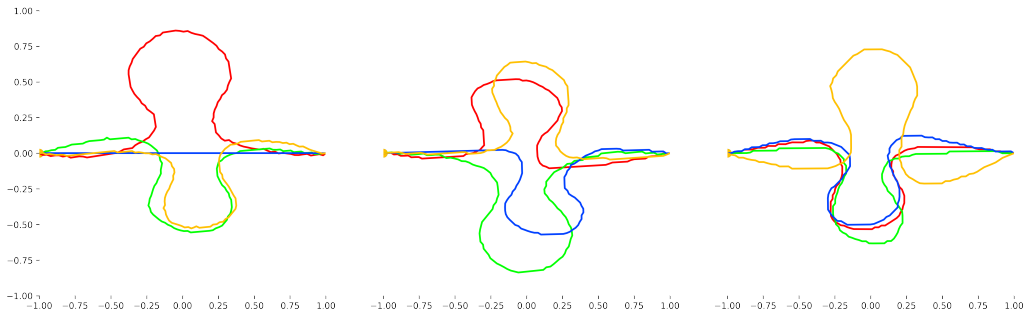

Figure 2: *Visualisations of the normalised sides for the same three puzzle pieces in figure 1. (See question 2.2.1 and question 3.3.*

2.2.2 Finally when normalising the sides for matching, we need an equal number of evenly spaced points along the side to be flattened into a feature vector. More advanced interpolation might work better, but for now, we just use linear interpolation over the entire length of a non-closed contour. Code a new function called `even_spaced_contour(contour, num_points=64)` that takes in a contour and the number of points that should be contained in the returned contour, then performing the following steps:

- To perform the linear interpolation we can use $B_{new} =$ `np.interp`(A_{new}, A, B), for clarity our argument names differ from the docs. `np.interp` works by looking at corresponding points $(a_i,\ b_i)$ where $a_i \in A,\ b_i \in B$ and finding the new points $(a'_j,\ b'_j)$ where $a'_j \in A_{new}$ and b'_j is the linearly interpolated value.
- To find A we need to calculate the cumulative sum of the length of all the line segments in the contour, divided by the total length of the contour. Note that the length of this array should be equal to the number of points, insert 0 at the front of the array returned by `np.cumsum`, since the number of segments is 1 less than the number of points.
- Once we have these cumulative ratios A for each point along the length of the line, we generate $A_{new} =$ `np.linspace(0, 1, num_points)` as the evenly spaced values between 0 and 1.
- Apply `np.interp` to all x values then all y values, with $B = (x_0, ..., x_n)$ and then $B = (y_0, ..., y_n)$ obtained from our contour points. recombine the new x and y values into your new evenly spaced contour points.

  Hint: Remember that the first $[0]$ and last points $[-1]$ of the new interpolated contour should be the same as the original contour.

2.2.3 As you did for question 2.1.3 with the same puzzle pieces, plot the new normalised sides with the same colours centred on a blank image as in figure 2. Do not use the same puzzle pieces as in figure 1 and 2.

Overlay this plot with points obtained from the simplified versions of those same normalised side contours after applying `even_spaced_contour` with `num_points=10` (remember to use 64 or more for the rest of the lab). Make sure these points are clearly visible as in figure 1.

3 Match Shape Models

With our 4 normalised side contours for each puzzle piece, we can now help Tino match the pieces together!

We can do this by treating each normalised side contour of a puzzle piece as a feature vector by flattening it into a one-dimensional array that can be used with k-nearest neighbours. However, before we do this, we need to be clever about which sides can be matched together to help avoid false positives! There are various checks we can perform if we know which pieces are corners or edges and which pieces belong in the interior.

a) Flat sides cannot be matched with any other sides.

b) The sides from each piece can only be matched to sides from other pieces, i.e. a piece can't match with itself.

c) A sunken side can only be matched to a protruding side and vice-versa.

d) The side clockwise of the flat side of an edge piece can only be matched to the side anti-clockwise from the flat side of another edge piece (or the sides from a corner) and vice-versa. Similarly, the side opposite the flat side of an edge piece can only be matched to sides from interior pieces.

To keep things simple we will only describe rules (a) and (c), the remainder you can try to implement if you wish.

3.1 Before we can apply these rules, we need to know which sides of pieces are flat so we can determine if they should be matched or not, and so that we can determine if a piece is an interior piece (no flat sides), edge piece (1 flat side), corner piece (2 flat sides) or invalid piece (3 or 4 flat sides).

Note that with side information, you could solve for possible the widths and heights of the puzzle in terms of pieces. Taking into account the number of pieces along the perimeter and the total area.

Code `is_flat_side(contour, min_ratio=0.9)` that checks if the distance `cont_dist` between the two endpoints of a non-closed contour is approximately equal to the entire length `cont_len` of the contour. If `cont_dist/cont_len >= min_ratio` then the side should be considered flat.

Hint: `np.linalg.norm` and `cv2.arcLength`[7] called with `closed=False` may come in handy.

3.2 Using the function from the previous step construct an array of all the non-flat normalised sides that can be matched together, thus allowing us to satisfy (a). Construct another array that corresponds to this one that contains the tuples with the index of the puzzle piece and the index of the side in that puzzle piece, so that if we have an index from this new array we can map back to the original puzzle piece and side. This will be useful after obtaining the output from k-nearest neighbours.

[7] arcLength opencv docs.

3.3 Remember in question 2.2.1 how we did not rotate the normalised sides if they were not protruding. We can easily satisfy rule (c) if we produce a training dataset from the array of normalised sides in the previous step except rotating by $180°$ and reversing the order of points in those sides.

Since the sides are normalised, we can rotate by multiplying the x and y coordinates of all the points in the contours by -1, which will convert protruding sides to sunken sides and vice-versa. However, remember how the first point of the side contour should point to the left down the $-$ve x-axis, we have just violated this property by performing the rotation, thus we need to reverse the order of our rotated sides so they correspond once again.

If we now find the k-nearest neighbours from the original non-rotated normalised sides, we minimise the distance between sunken and protruding points and maximise the distance if they are incompatible, see figure 2. Use `sklearn` to perform k-nearest neighbours, don't forget to flatten your sides into feature vectors, an incomplete example is given below:

```
from sklearn.neighbors import NearestNeighbors

# assuming sides has a shape of (-1, num_points, 2)
rotated_sides = (sides * (-1, -1))[:, ::-1, :]
# TODO: flatten both arrays of side contours into arrays of features
knn = NearestNeighbors(n_neighbors=1, algorithm="brute")
knn = knn.fit(rotated_side_features)
distances, indices = knn.kneighbors(side_features)
```

3.4 Finally let's plot a directed graph of all the connections between puzzle pieces to see how well our method has done using the package `networkx`[8]. If nodes/vertices in a graph are given by $V = \{0, ..., n\}$ and directed edges between nodes are defined as a list of ordered pairs $E \subseteq \{(a, b) \mid (a, b) \in V^2 \text{ and } a \neq b\}$

```
import networkx as nx
import numpy as np
import random
import matplotlib.pyplot as plt

def plot_graph(V, E, seed=42):
    random.seed(seed) # graphs are randomly plotted
    np.random.seed(seed) # graphs are randomly plotted
    G = nx.DiGraph()
    G.add_nodes_from(V)
    G.add_edges_from(E)
    nx.draw_kamada_kawai(G, with_labels=True)
    plt.show()
```

The graphing code is given above, use the result from the previous question to construct your list of nodes and list of edges to pass to this function and plot the result after matching together all puzzle pieces. Make sure the indices in your list of nodes V correspond to their file names.

You may adjust the `seed` so that the plot is more clear, for this and following questions.

3.5 Remember the edges are directed in the previous question, so if side a matches to side b, side b might not match back to side a. Plot a second graph that only includes edges where sides match back to themselves.

3.6 Like the previous question, remove sides that don't match back to themselves, but this time only construct a graph from the edge pieces. Our goal is to hopefully obtain a ring.

[8] Directed graph documentation for networkx.

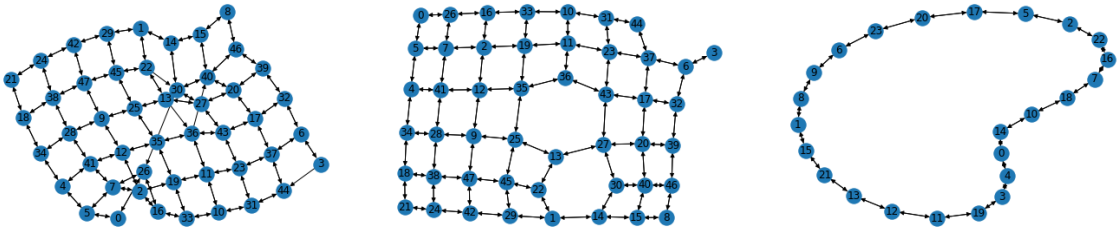

Figure 3: *Example graphs for questions 3.4, 3.5 and 3.6. Your results may differ slightly.*

3.7 Identify at least one potential issue with the detection of side contours. Think about the differences that can arise from human labeling bias after matching protruding and sunken puzzle piece sides. How could this issue be remedied?

Hint: Try plotting some `rotated_side_features` against some matching `side_features`.

3.8 Comment on any potential issues either with the data or the pipeline in generating the shape models, and how well this pipeline was able to solve the puzzle. Also comment on how well the pipeline would scale with the number of puzzle pieces.

4 Submission

Submit both a Jupyter Notebook and a PDF copy of the notebook containing the plots, code, results and answers from the tasks above.

Comment your code where necessary and make sure to **clearly** label the different questions using markdown headings of the full question numbers eg. `## Question 2.1.3`. Give some distinction between code for the questions and your own helper code.