002_data_segmentation

September 3, 2022

1 Data segmentation

[ ]:
```
#%cd ..

import numpy as np
import cv2
import imutils
import matplotlib.pyplot as plt
import pickle
import matplotlib.patches as patches
from skimage.segmentation import morphological_chan_vese

array_segmented_images = np.load(r"./data/segmentation.npy")
array_data_cropped = np.load('./temp/array_data_cropped.npy')

with open('./temp/array_times.pickle', 'rb') as handle:
    array_times = pickle.load(handle)


# Change this to recalculate segmentation. Pay attention as it could take time.
redo_segmentation = False
```

[ ]:
```
import ipywidgets as widgets
# Prettier plots
import seaborn as sns
sns.set(#font='Palatino',
        rc={
 'axes.axisbelow': False,
 'axes.edgecolor': 'k',
 'axes.facecolor': 'None',
 'axes.grid' : False,
 'axes.spines.right': False,
 'axes.spines.top': False,
 'figure.facecolor': 'white',
 'lines.solid_capstyle': 'round',
 'patch.edgecolor': 'w',
 'patch.force_edgecolor': True,
```

```
 'xtick.bottom': True,
 'xtick.direction': 'out',
 'xtick.top': False,
 'ytick.direction': 'out',
 'ytick.left': True,
 'ytick.right': False})

# Vectorial plot
import matplotlib_inline.backend_inline as backend_inline
backend_inline.set_matplotlib_formats('svg')

## Testing parallel loading of ZARR
from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor

def paral(func, lista, N, threads=True, processes=False):
    if processes:
        with ProcessPoolExecutor(max_workers=N) as executor:
            results = executor.map(func, lista)
        return list(results)
    elif threads:
        with ThreadPoolExecutor(max_workers=N) as executor:
            results = executor.map(func, lista)
        return list(results)

## Testing parallel loading of ZARR
def loadindex(index):
    try:
        return img[index][:]
    except Exception as e:
        print(e)

## Visualization method
def visualize_data(array_data, array_segments = None, array_times = None,␣
↪cmap='crest'):
    # Widget slider to browse the data
    index = widgets.IntSlider(
        value=5, min=0, max=array_data.shape[0] - 1, step=1, description="Index"
    )

    # Other widget slider to browse the channels
    channel = widgets.IntSlider(
        value=5, min=0, max=array_data.shape[3] - 1, step=1,␣
↪description="Channel"
    )

    # Checkbox to display RGB (override the channel)
    display_RGB = widgets.Checkbox(description="Display RGB", value=False)
```

```
    ui = widgets.HBox([index, channel, display_RGB])

    # Widget interaction function
    def anim(index_value, channel_value, display_RGB_value):
        fig = plt.figure(figsize=(10,8))
        if display_RGB_value:
            plt.imshow( array_data[index_value, :, :, (3,2,1)].swapaxes(0,1).
↪swapaxes(1, 2))
        else:
            plt.imshow(array_data[index_value, :, :, channel_value], cmap =␣
↪cmap)
        if array_segments is not None:
            if np.sum(array_segments[index_value])>0:
                plt.contour(array_segments[index_value], [0.5], colors='r')
        if array_times is not None:
            plt.title('Acquisition time: ' + str(array_times[index_value]))
        else:
            plt.title('Acquisition time: ' +␣
↪str(df['beginposition'][index_value]))
        plt.axis('off')
        return

    # Link widget and function
    out = widgets.interactive_output(anim, {"index_value": index,␣
↪'channel_value': channel, 'display_RGB_value': display_RGB})

    # Display result
    return ui, out
```

1.1 Dependencies and helper functions

[ ]:
```
def apply_segmentation(image):
    image_normalized = image/np.max(image)
    seg = morphological_chan_vese(image_normalized , num_iter= 200,␣
↪init_level_set='disk', smoothing=2, lambda1 = 10., lambda2 = 1.)
    return seg
```

1.2 Loading of the data

[ ]:
```
def calc_segmentation():
    # Segment all images in the dataset on channel 9 (# ! Takes ~1h to run)␣
↪(B03 - B08) / (B03 + B08)
    l_segmented_images = paral(apply_segmentation, array_data_cropped[:,:,:,9],␣
↪10) #array_data_cropped[:,:,:,9] #np.squeeze(ndwi_array)+1
    array_segmented_images = np.array(l_segmented_images)
```

```
    del l_segmented_images

#     # Store result as it's pretty heavy to compute
#     with open(r"./data/segmentation.npy", 'wb') as f:
#         np.save(f, array_segmented_images)

if redo_segmentation == True:
    calc_segmentation()
```

1.3 Filter out bad-segments based on size

[ ]:
```
# Filter out segments that are 80% smaller than the main segment
for index, segment in enumerate(array_segmented_images):
    try:
        cnts = cv2.findContours(segment.astype('uint8'), cv2.RETR_EXTERNAL, cv2.
 ↪CHAIN_APPROX_SIMPLE)
        cnts = imutils.grab_contours(cnts)
        cnts = sorted(cnts, key=cv2.contourArea, reverse=True)
        rect_areas = []
        for c in cnts:
            (x, y, w, h) = cv2.boundingRect(c)
            rect_areas.append(w * h)
        max_area = np.max(rect_areas)
        for c in cnts:
            (x, y, w, h) = cv2.boundingRect(c)
            cnt_area = w * h
            if cnt_area < 0.2 * max_area:
                segment[y:y + h, x:x + w] = 0
        array_segmented_images[index] = segment

    except Exception as e:
        #print(index, e)
        pass
```

[ ]:
```
# Filter out segments out segments that are outside of the largest segment when␣
 ↪the lake is the
# fullest (among the first images, as it becomes empty afterwards)
biggest_segment_index = np.argmax([np.sum(x) for x in array_segmented_images[:
 ↪20]])
biggest_segment = array_segmented_images[biggest_segment_index]
for index, segment in enumerate(array_segmented_images):
    segment_diff = biggest_segment - segment
    segment[segment_diff < 0] = 0
    array_segmented_images[index] = segment
```

1.4 Filtering out based on water content

[ ]:
```
# Display distribution of intensity difference with 'pure' lake
l_diff_mean_segments = [np.mean(array_data_cropped[index,:,:,9][segment==1])-np.
 ↪mean(array_data_cropped[biggest_segment_index,:,:,9][biggest_segment==1]) if␣
 ↪np.sum(segment)>0 else np.nan for index, segment in␣
 ↪enumerate(array_segmented_images)]
tresh = 1450

def plot_segment_differences():
    fig, ax = plt.subplots(1, figsize = (10,5))
    fig.patch.set_facecolor('white')
    plt.hist(l_diff_mean_segments, bins=100)
    plt.ylim(0, 8)
    plt.xlim(-100, 1600)
    plt.xlabel("Segment mean intensity value difference w.r.t clean segment")
    plt.ylabel("Frequency")


    # Create one rectangle patch and add it to the plot
    rect = patches.Rectangle((tresh, 0), 1600-tresh, 8, alpha = 0.3,␣
 ↪facecolor="red")
    ax.add_patch(rect)
    plt.title("Distribution of segment differences of intensity value w.r.t␣
 ↪cleanest segment")
    plt.show()
```

[ ]:
```
# Filter out segments that have an intensity which is significantly different␣
 ↪from when the lake is the purest
array_segmented_images = np.array([segment if (l_diff_mean_segments[index]<␣
 ↪tresh and not np.isnan(l_diff_mean_segments[index])) else np.
 ↪zeros_like(segment) for index, segment in enumerate(array_segmented_images)])
```

[ ]:
```
def visualize_segmentation():
    ui, out = visualize_data(array_data_cropped,  array_segments =␣
 ↪array_segmented_images, array_times = array_times)
    display(ui, out)
```

[ ]:
```
np.save('./temp/array_segmented_images.npy', array_segmented_images)

# with open('./temp/array_segmented_times.pickle', 'wb') as handle:
#     pickle.dump(array_segmented_times, handle, protocol=pickle.
 ↪HIGHEST_PROTOCOL)
```