hw3

September 7, 2021

1 Homework 3

This assignment covers the Harris corner detector, RANSAC and the HOG descriptor for panorama stitching.

[ ]:
```
from __future__ import print_function

# Setup
import numpy as np
from skimage import filters
from skimage.feature import corner_peaks
from skimage.io import imread
import matplotlib.pyplot as plt
from time import time

%matplotlib inline
plt.rcParams['figure.figsize'] = (15.0, 12.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading extenrnal modules
%load_ext autoreload
%autoreload 2
```

1.1 Introduction: Panorama Stitching

Panorama stitching is an early success of computer vision. Matthew Brown and David G. Lowe published a famous panoramic image stitching paper in 2007. Since then, automatic panorama stitching technology has been widely adopted in many applications such as Google Street View, panorama photos on smartphones, and stitching software such as Photosynth and AutoStitch.

In this assignment, we will detect and match keypoints from multiple images to build a single panoramic image. This will involve several tasks: 1. Use Harris corner detector to find keypoints. 2. Build a descriptor to describe each point in an image. Compare two sets of descriptors coming from two different images and find matching keypoints. 3. Given a list of matching keypoints, use least-squares method to find the affine transformation matrix that maps points in one image to another. 4. Use RANSAC to give a more robust estimate of affine transformation matrix. Given the transformation matrix, use it to transform the second image and overlay it on the first image, forming a panorama. 5. Implement a different descriptor (HOG descriptor) and get another stitching result.

There's a lot of material to get hands-on with so we recommend starting early!

1.2 Notes on Running This Notebook

Make sure to run each Part from it's begining to ensure that you compute all of the dependencies of your current question and don't crossover variables with the same name from other questions. For example, don't run parts 4 and 5 and then return to run only the last cell of part 3; your panorama won't be using the right transformed images! So long as you run each Part from it's beginning, you can run the Parts in any order.

When assembling your PDF, we recommend running all cells in order from the top of the notebook to prevent any of these discontinuity errors.

1.3 Part 1 Harris Corner Detector (20 points)

In this section, you are going to implement Harris corner detector for keypoint localization. Review the lecture slides on Harris corner detector to understand how it works. The Harris detection algorithm can be divide into the following steps: 1. Compute x and y derivatives (I_x, I_y) of an image 2. Compute products of derivatives (I_x^2, I_y^2, I_{xy}) at each pixel 3. Compute matrix M at each pixel, where

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

4. Compute corner response $R = Det(M) - k(Trace(M)^2)$ at each pixel 5. Output corner response map $R(x,y)$

Step 1 is already done for you in the function **`harris_corners`** in `panorama.py`. We used the Sobel Operator, which computes smoothed gradients at each pixel in the x and y direction. See skimage documentation for sobel_v and sobel_h for more information on the sobel kernels and operators.

For step 3, we've created a uniform window function w for you in the starter code. You can assume that the window size will be odd.

Complete the function implementation of **`harris_corners`** and run the code below.

1.3.1 Hint: There are two ways to solve this problem

Vectorized: If you want to be really efficient, you can use the function `scipy.ndimage.filters.convolve`, which is already imported in `panorama.py`, and compute the response map R at every pixel all at once. If you're clever with your convolutions and determinant and trace calculations, you can compute the windowed gradients in M ($\sum_{x,y} w(x,y) \cdot I_x^2$, and $\sum_{x,y} w(x,y) \cdot I_y^2$, and $\sum_{x,y} w(x,y) \cdot I_{xy}$), and then compute the response map without any for loops!

Iterative: The more intuitive solution is to iterate through each pixel of the image, compute M based on the surrounding neighborhood of pixel gradients in I_x, I_y, and I_{xy}, and then compute the response map pixel $R(x,y)$. You may find your implementations of conv_nested and conv_fast from HW 1 to be useful references!

Note that you'll want to explictly specify zero-padding to match the Harris response map definition, but we'll accept the default behavior of `scipy.ndimage.filters.convolve` as well. If you use zero-padding, both the vectorized and for-loop implementations will get you to the same answer!

The 'Alternate Accepted Harris Corner Solution' image presents the result of `scipy.ndimage.filters.convolve`'s default reflection padding, while the 'Harris Corner Solution' image presents the zero-padding solution. Similarly, 'Alternate Accepted Detected Corners Solution' image presents the result of `scipy.ndimage.filters.convolve`'s default reflection padding, while the 'Detected Corners Solution' image presents the zero-padding solution. **Both are accepted solutions!**

[ ]:
```
from panorama import harris_corners

img = imread('sudoku.png', as_gray=True)

# Compute Harris corner response
response = harris_corners(img)

# Display corner response
plt.subplot(1,3,1)
plt.imshow(response)
plt.axis('off')
plt.title('Harris Corner Response')

plt.subplot(1,3,2)
plt.imshow(imread('solution_harris.png', as_gray=True))
plt.axis('off')
plt.title('Harris Corner Solution')

plt.subplot(1,3,3)
plt.imshow(imread('solution_alternate_harris.png', as_gray=True))
plt.axis('off')
plt.title('Alternate Accepted Harris Corner Solution')

plt.show()
```

Once you implement the Harris detector correctly, you will be able to see small bright blobs around the corners of the sudoku grids and letters in the output corner response image. The function `corner_peaks` from `skimage.feature` performs non-maximum suppression to take local maxima of the response map and localize keypoints.

[ ]:
```
# Perform non-maximum suppression in response map
# and output corner coordinates
corners = corner_peaks(response, threshold_rel=0.01)

# Display detected corners
plt.subplot(1,3,1)
plt.imshow(img)
```

```
plt.scatter(corners[:,1], corners[:,0], marker='x')
plt.axis('off')
plt.title('Detected Corners')

plt.subplot(1,3,2)
plt.imshow(imread('solution_detected_corners.png'))
plt.axis('off')
plt.title('Detected Corners Solution')

plt.subplot(1,3,3)
plt.imshow(imread('solution_alternate_detected_corners.png'))
plt.axis('off')
plt.title('Alternate Accepted Detected Corners Solution')

plt.show()
```

1.4 Part 2 Describing and Matching Keypoints (20 points)

We are now able to localize keypoints in two images by running the Harris corner detector independently on them. Next question is, how do we determine which pair of keypoints come from corresponding locations in those two images? In order to *match* the detected keypoints, we must come up with a way to *describe* the keypoints based on their local appearance. Generally, each region around detected keypoint locations is converted into a fixed-size vectors called *descriptors*.

1.4.1 Part 2.1 Creating Descriptors (10 points)

In this section, you are going to implement the **simple_descriptor** function, where each keypoint is described by the normalized intensity of a small patch around it.

[ ]:
```
from panorama import harris_corners

img1 = imread('uttower1.jpg', as_gray=True)
img2 = imread('uttower2.jpg', as_gray=True)

# Detect keypoints in two images
keypoints1 = corner_peaks(harris_corners(img1, window_size=3),
                          threshold_rel=0.05,
                          exclude_border=8)
keypoints2 = corner_peaks(harris_corners(img2, window_size=3),
                          threshold_rel=0.05,
                          exclude_border=8)

print("Keypoints 1 shape = ", keypoints1.shape)
print("Keypoints 2 shape = ", keypoints2.shape)

# Display detected keypoints
plt.subplot(1,2,1)
plt.imshow(img1)
```

```python
plt.scatter(keypoints1[:,1], keypoints1[:,0], marker='x')
plt.axis('off')
plt.title('Detected Keypoints for Image 1')

plt.subplot(1,2,2)
plt.imshow(img2)
plt.scatter(keypoints2[:,1], keypoints2[:,0], marker='x')
plt.axis('off')
plt.title('Detected Keypoints for Image 2')
plt.show()
```

1.4.2 Part 2.2 Matching Descriptors (10 points)

Next, implement the **match_descriptors** function to find good matches in two sets of descriptors. First, calculate Euclidean distance between all pairs of descriptors from image 1 and image 2. Then use this to determine if there is a good match: for each descriptor in image 1, if the distance to the closest descriptor in image 2 is significantly (by a given factor) smaller than the distance to the second-closest, we call it a match. The output of the function is an array where each row holds the indices of one pair of matching descriptors.

Checking your answer: you should see an identical matching of keypoints as the solution, but the precise colors of each line will change with every run of keypoint matching so colors do not need to match.

Optional ungraded food for thought: Think about why this method of keypoint matching is not commutative.

[ ]:
```python
from panorama import simple_descriptor, match_descriptors, describe_keypoints
from utils import plot_matches

# Set seed to compare output against solution
np.random.seed(131)

patch_size = 5

# Extract features from the corners
desc1 = describe_keypoints(img1, keypoints1,
                           desc_func=simple_descriptor,
                           patch_size=patch_size)
desc2 = describe_keypoints(img2, keypoints2,
                           desc_func=simple_descriptor,
                           patch_size=patch_size)

print("Desc1 shape = ", desc1.shape)
print("Desc2 shape = ", desc2.shape)

# Match descriptors in image1 to those in image2
matches = match_descriptors(desc1, desc2, 0.7)
```

```python
# Plot matches
fig, ax = plt.subplots(1, 1, figsize=(15, 12))
ax.axis('off')
plt.title('Matched Simple Descriptor')
plot_matches(ax, img1, img2, keypoints1, keypoints2, matches)
plt.show()

plt.imshow(imread('solution_simple_descriptor.png'))
plt.axis('off')
plt.title('Matched Simple Descriptor Solution')
plt.show()
```

1.5 Part 3 Transformation Estimation (20 points)

We now have a list of matched keypoints across the two images. We will use this to find a transformation matrix that maps points in the second image to the corresponding coordinates in the first image. In other words, if the point $p_1 = [y_1, x_1]$ in image 1 matches with $p_2 = [y_2, x_2]$ in image 2, we need to find an affine transformation matrix H such that

$$\tilde{p_2} H = \tilde{p_1},$$

where $\tilde{p_1}$ and $\tilde{p_2}$ are homogenous coordinates of p_1 and p_2.

Note that it may be impossible to find the transformation H that maps every point in image 2 exactly to the corresponding point in image 1. However, we can estimate the transformation matrix with least squares. Given N matched keypoint pairs, let X_1 and X_2 be $N \times 3$ matrices whose rows are homogenous coordinates of corresponding keypoints in image 1 and image 2 respectively. Then, we can estimate H by solving the least squares problem,

$$X_2 H = X_1$$

Implement **`fit_affine_matrix`** in `panorama.py`

-Hint: read the documentation about np.linalg.lstsq

[ ]:
```python
from panorama import fit_affine_matrix

# Sanity check for fit_affine_matrix

# Test inputs
a = np.array([[0.5, 0.1], [0.4, 0.2], [0.8, 0.2]])
b = np.array([[0.3, -0.2], [-0.4, -0.9], [0.1, 0.1]])

H = fit_affine_matrix(b, a)

# Target output
sol = np.array(
```

```
    [[1.25, 2.5, 0.0],
     [-5.75, -4.5, 0.0],
     [0.25, -1.0, 1.0]]
)

error = np.sum((H - sol) ** 2)

if error < 1e-20:
    print('Implementation correct!')
else:
    print('There is something wrong.')
```

After checking that your `fit_affine_matrix` function is running correctly, run the following code to apply it to images. Images will be warped and image 2 will be mapped to image 1.

[ ]:
```
from utils import get_output_space, warp_image

# Extract matched keypoints
p1 = keypoints1[matches[:,0]]
p2 = keypoints2[matches[:,1]]

# Find affine transformation matrix H that maps p2 to p1
H = fit_affine_matrix(p1, p2)

output_shape, offset = get_output_space(img1, [img2], [H])
print("Output shape:", output_shape)
print("Offset:", offset)


# Warp images into output sapce
img1_warped = warp_image(img1, np.eye(3), output_shape, offset)
img1_mask = (img1_warped != -1) # Mask == 1 inside the image
img1_warped[~img1_mask] = 0     # Return background values to 0

img2_warped = warp_image(img2, H, output_shape, offset)
img2_mask = (img2_warped != -1) # Mask == 1 inside the image
img2_warped[~img2_mask] = 0     # Return background values to 0

# Plot warped images
plt.subplot(1,2,1)
plt.imshow(img1_warped)
plt.title('Image 1 Warped')
plt.axis('off')

plt.subplot(1,2,2)
plt.imshow(img2_warped)
plt.title('Image 2 Warped')
```

```
plt.axis('off')

plt.show()
```

Next, the two warped images are merged to get a panorama. Your panorama may not look good at this point, but we will later use other techniques to get a better result.

[ ]:
```
# Merge the two images
merged = img1_warped + img2_warped

# Track the overlap by adding the masks together
overlap = (img1_mask * 1.0 +  # Multiply by 1.0 for bool -> float conversion
           img2_mask)

# Normalize through division by `overlap` - but ensure the minimum is 1
normalized = merged / np.maximum(overlap, 1)

plt.imshow(normalized)
plt.axis('off')
plt.title('Fit-Affine Panorama')
plt.show()

plt.imshow(imread('solution_fit_affine_panorama.png'))
plt.axis('off')
plt.title('Fit-Affine Panorama Solution')
plt.show()
```

1.6 Part 4 RANSAC (20 points)

Rather than directly feeding all our keypoint matches into `fit_affine_matrix` function, we can instead use RANSAC ("RANdom SAmple Consensus") to select only "inliers" to use for computing the transformation matrix.

The steps of RANSAC are: 1. Select random set of matches 2. Compute affine transformation matrix 3. Find inliers using the given threshold 4. Repeat and keep the largest set of inliers 5. Re-compute least-squares estimate on all of the inliers

In this case, use Euclidean distance between matched points as a measure of inliers vs outliers.

Implement **ransac** in `panorama.py`, run through the following code to get a panorama. You can see the difference from the result we get without RANSAC.

[ ]:
```
from panorama import ransac

# Set seed to compare output against solution image
np.random.seed(131)

H, robust_matches = ransac(keypoints1, keypoints2, matches, threshold=1)
print("Robust matches shape = ", robust_matches.shape)
```

```python
print("H = \n", H)

# Visualize robust matches
fig, ax = plt.subplots(1, 1, figsize=(15, 12))
plot_matches(ax, img1, img2, keypoints1, keypoints2, robust_matches)
plt.axis('off')
plt.title('RANSAC Robust Matches')
plt.show()

plt.imshow(imread('solution_ransac.png'))
plt.axis('off')
plt.title('RANSAC Robust Matches Solution')
plt.show()
```

We can now use the tranformation matrix H computed using the robust matches to warp our images and create a better-looking panorama.

[ ]:
```python
output_shape, offset = get_output_space(img1, [img2], [H])

# Warp images into output sapce
img1_warped = warp_image(img1, np.eye(3), output_shape, offset)
img1_mask = (img1_warped != -1) # Mask == 1 inside the image
img1_warped[~img1_mask] = 0     # Return background values to 0

img2_warped = warp_image(img2, H, output_shape, offset)
img2_mask = (img2_warped != -1) # Mask == 1 inside the image
img2_warped[~img2_mask] = 0     # Return background values to 0

# Plot warped images
plt.subplot(1,2,1)
plt.imshow(img1_warped)
plt.title('Image 1 warped')
plt.axis('off')

plt.subplot(1,2,2)
plt.imshow(img2_warped)
plt.title('Image 2 warped')
plt.axis('off')

plt.show()
```

[ ]:
```python
# Merge the two images
merged = img1_warped + img2_warped

# Track the overlap by adding the masks together
overlap = (img1_mask * 1.0 +  # Multiply by 1.0 for bool -> float conversion
           img2_mask)
```

```
# Normalize through division by `overlap` - but ensure the minimum is 1
normalized = merged / np.maximum(overlap, 1)
plt.imshow(normalized)
plt.axis('off')
plt.title('RANSAC Robust Panorama')
plt.show()

plt.imshow(imread('solution_ransac_panorama.png'))
plt.axis('off')
plt.title('RANSAC Robust Panorama Solution')
plt.show()
```

1.7 Part 5 Histogram of Oriented Gradients (HOG) (20 points)

In the above code, you are using the **simple_descriptor**, and in this section, you are going to implement a simplified version of HOG descriptor. HOG stands for Histogram of Oriented Gradients. In HOG descriptor, the distribution (histograms) of the directions of gradients (oriented gradients) are used as features. Gradients (x and y derivatives) of an image are useful because the magnitude of a gradient is large around edges and corners (regions of abrupt intensity changes) and we know that edges and corners pack in a lot more information about object shape than flat regions. The steps of HOG are: 1. Compute the gradient image in x and y directions * Use the sobel filter provided by skimage.filters 2. Compute gradient histograms * Divide image into cells, and calculate histogram of gradients in each cell 3. Flatten block of histograms into feature vector 4. Normalize flattened block by L2 norm

Implement **hog_descriptor** in `panorama.py` and run through the following code to get a panorama image.

[ ]:
```
from panorama import hog_descriptor

img1 = imread('uttower1.jpg', as_gray=True)
img2 = imread('uttower2.jpg', as_gray=True)

# Detect keypoints in both images
hog_keypoints1 = corner_peaks(harris_corners(img1, window_size=3),
                              threshold_rel=0.05,
                              exclude_border=8)
hog_keypoints2 = corner_peaks(harris_corners(img2, window_size=3),
                              threshold_rel=0.05,
                              exclude_border=8)

print("HoG keypoints1 shape = ", hog_keypoints1.shape)
print("HoG keypoints2 shape = ", hog_keypoints2.shape)
```

[ ]:
```
from panorama import simple_descriptor, match_descriptors, describe_keypoints

# Set seed to compare output against solution
np.random.seed(131)
```

```
# Extract features from the corners
hog_desc1 = describe_keypoints(img1, hog_keypoints1,
                               desc_func=hog_descriptor,
                               patch_size=16)
hog_desc2 = describe_keypoints(img2, hog_keypoints2,
                               desc_func=hog_descriptor,
                               patch_size=16)

print("HoG desc1 shape = ", hog_desc1.shape)
print("HoG desc2 shape = ", hog_desc2.shape)

# Match descriptors in image1 to those in image2
hog_matches = match_descriptors(hog_desc1, hog_desc2, 0.7)

# Plot matches
fig, ax = plt.subplots(1, 1, figsize=(15, 12))
ax.axis('off')
plt.title('Matched HOG Descriptor')
plot_matches(ax, img1, img2, hog_keypoints1, hog_keypoints2, hog_matches)
plt.show()

plt.imshow(imread('solution_hog.png'))
plt.axis('off')
plt.title('Matched HOG Descriptor Solution')
plt.show()
```

Once we've described our keypoints with the HOG descriptor and have found matches between these keypoints, we can use RANSAC to select robust matches for computing the tranasformtion matrix.

[ ]:
```
from panorama import ransac

# Set seed to compare output against solution image
np.random.seed(131)

H, robust_matches = ransac(hog_keypoints1, hog_keypoints2, hog_matches,␣
 ↪threshold=1)
print("Robust matches shape = ", robust_matches.shape)
print("H = \n", H)

# Plot matches
fig, ax = plt.subplots(1, 1, figsize=(15, 12))
plot_matches(ax, img1, img2, hog_keypoints1, hog_keypoints2, robust_matches)
plt.axis('off')
plt.title('Robust Matched HOG descriptor + RANSAC')
plt.show()
```

```
plt.imshow(imread('solution_hog_ransac.png'))
plt.axis('off')
plt.title('Robust Matched HOG descriptor + RANSAC Solution')
plt.show()
```

Now we use the computed transformation matrix H to warp our images and produce our panorama.

[ ]:
```
output_shape, offset = get_output_space(img1, [img2], [H])

# Warp images into output sapce
img1_warped = warp_image(img1, np.eye(3), output_shape, offset)
img1_mask = (img1_warped != -1) # Mask == 1 inside the image
img1_warped[~img1_mask] = 0     # Return background values to 0

img2_warped = warp_image(img2, H, output_shape, offset)
img2_mask = (img2_warped != -1) # Mask == 1 inside the image
img2_warped[~img2_mask] = 0     # Return background values to 0

# Plot warped images
plt.subplot(1,2,1)
plt.imshow(img1_warped)
plt.title('Image 1 warped')
plt.axis('off')

plt.subplot(1,2,2)
plt.imshow(img2_warped)
plt.title('Image 2 warped')
plt.axis('off')

plt.show()
```

[ ]:
```
# Merge the two images
merged = img1_warped + img2_warped

# Track the overlap by adding the masks together
overlap = (img1_mask * 1.0 +  # Multiply by 1.0 for bool -> float conversion
           img2_mask)

# Normalize through division by `overlap` - but ensure the minimum is 1
normalized = merged / np.maximum(overlap, 1)
plt.imshow(normalized)
plt.axis('off')
plt.title('HOG Descriptor + RANSAC Robust Panorama')
plt.show()

plt.imshow(imread('solution_hog_panorama.png'))
plt.axis('off')
```

```python
plt.title('HOG Descriptor + RANSAC Robust Panorama Solution')
plt.show()
```