CS231A: Computer Vision, From 3D Reconstruction to Recognition Homework #3

(Winter 2021) Due: **Wednesday, March 3**

(Extra Credit) Matching Homographies for Image Rectification (20 points)

Building off the Fundamental Matrix Estimation From Point Correspondences in PSET 2, this problem seeks to rectify a pair of images given a few matching points. The main task in image rectification is generating two homographies H_1, H_2 that transform the images in a way that the epipolar lines are parallel to the horizontal axis of the image. A detailed description of the standard homography generation for image rectification can be found in Course Notes 3. You will implement the methods in `p1.py` (note that you need to copy the implementation of `lls_eight_point_alg` and `normalized_eight_point_alg` from PSET 2 first) and complete the following:

(a) The first step in rectifying an image is to determine the epipoles. Complete the function `compute_epipole()`. Submit the epipoles you calculated and a copy of your code. **[5 points]**

Tip: If you want to check your answer, recall that the epipole is where all the epipolar lines intersect.

(b) Finally, we can determine the homographies H_1 and H_2. We first compute H_2 as the homography that takes the second epipole e_2 to a point at infinity $(f, 0, 0)$. The matching homography H_1 is computed by solving a least-squares problem. Complete the function `compute_matching_homographies()`. Submit the homographies computed for the sample image pair a copy of your code. **[10 points]**

(c) Submit a copy of your rectified images plotted using `plot_epipolar_lines_on_images()`. **[5 points]**

Space Carving (45 points)

Dense 3D reconstruction is a difficult problem, as tackling it from the Structure from Motion framework (as seen in the previous problem set) requires dense correspondences. Another solution to dense 3D reconstruction is space carving[1], which takes the idea of volume intersection and iteratively refines the estimated 3D structure. In this problem, you implement significant portions of the space carving framework. In the starter code, you will be modifying the `p2.py` file inside the `p2` directory.

(a) The first step in space carving is to generate the initial voxel grid that we will carve into. Complete the function `form_initial_voxels()`. Submit an image of the generated voxel grid and a copy of your code. **[10 points]**

[1] http://www.cs.toronto.edu/~kyros/pubs/00.ijcv.carve.pdf

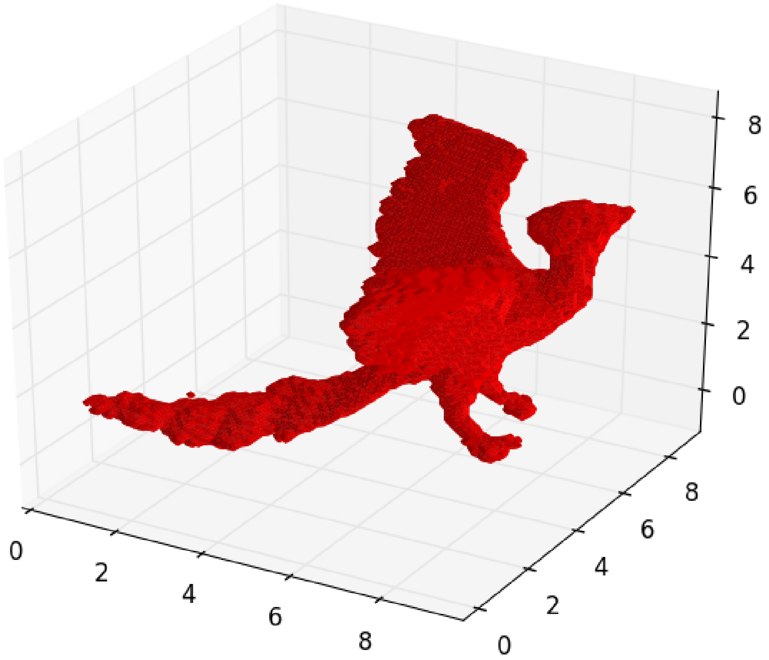

Figure 1: Our final carving using the true silhouette values

(b) Now, the key step is to implement the carving for one camera. To carve, we need the camera frame and the silhouette associated with that camera. Then, we carve the silhouette from our voxel grid. Implement this carving process in `carve()`. Submit your code and a picture of what it looks like after one iteration of the carving. [**15 points**]

(c) The last step in the pipeline is to carve out multiple views. Submit the final output after all carvings have been completed, using `num_voxels` $\approx 6,000,000$. [**5 points**]

(d) Notice that the reconstruction is not really that exceptional. This is because a lot of space is wasted when we set the initial bounds of where we carve. Currently, we initialize the bounds of the voxel grid to be the locations of the cameras. However, we can do better than this by completing a quick carve on a much lower resolution voxel grid (we use `num_voxels` $=$ 4000) to estimate how big the object is and retrieve tighter bounds. Complete the method `get_voxel_bounds()` and change the variable `estimate_better_bounds` in the `main()` function to `True`. Submit your new carving, which should be more detailed, and your code. [**10 points**]

(e) Finally, let's have a fun experiment. Notice that in the first three steps, we used perfect silhouettes to carve our object. Look at the `estimate_silhouette()` function implemented and its output. Notice that this simple method does not produce a really accurate silhouette. However, when we run space carving on it, the result still looks decent!

  (i) Why is this the case? [**2 points**]

  (ii) What happens if you reduce the number of views? [**1 points**]

  (iii) What if the estimated silhouettes weren't conservative, meaning that one or a few views had parts of the object missing? [**2 points**]

Representation Learning (15 points)

In this problem, you'll implement a method for image representation learning.

Unlike the previous problems, for this one you will have to work with Google Colaboratory. In Google Drive, follow these steps to make sure you have the ability to work on this problem:

a. Click the wheel in the top right corner and select Settings.

b. Click on the Manage Apps tab.

c. At the top, select Connect more apps which should bring up a GSuite Marketplace window.

d. Search for Colab then click Add.

Now, upload the contents of the 'code/represenation_learning' to a location of your choosing on Drive. Then, navigate to this folder and open the file "RepresentationLearning.ipynb" with Colaboratory. The rest of the instructions are provided in that document. Note that there is no autograder for this problem, as you should be able to confirm whether your implementation works from the images and plots. Include the following in your writeup:

a. Once you finish the section "PyTorch MNIST Data Preparation", include the 3 by 3 grid visualization of the MNIST data (5 points).

b. Once you finish the section "Training for MNIST Digit Prediction", include the graphs of training progress over 10 epochs (5 points).

c. Once you finish the section "Fine-Tuning for MNIST digit classification", include all 3 sets of graphs from this section (5 points).

Fun fact: the method you just implemented was discovered fairly recently, in the ICLR2018 paper "Unsupervised Representation Learning by Predicting Image Rotations"!

(Extra Credit) Monocular Depth Estimation (15 points)

Now that you've had some experience with representation learning on a small dataset, in this problem you will see how learned representations will help with estimating 3D structure in images by implementing training of monocular depth estimation on a real dataset (the NYU Depth Dataset V2). We will be implementing the approach taken in "High Quality Monocular Depth Estimation via Transfer Learning", which showed that taking a bigger model pre-trained on object classification proved to help a lot with training that model to perform monocular depth estimation.

As with the previous problem, you'll be working with Colaboratory, so once again begin by uploading the contents of 'code/monocular_depth_estimation' to a location of your choosing on Google Drive. Then, open the file "MonocularDepthEstimation.ipynb" with Colaboratory. The rest of the instructions are provided in that document. Note that there is no autograder for this problem, as you should be able to confirm whether your implementation works from the images and plots. Include the following in your writeup:

a. Once you finish the section "Checking out the data", include the grid visualization of the NYU data (5 points).

b. Once you finish the section "Testing the trained model", include the visualization of model predictions (5 points).

c. Once you finish the section "Training without feature transfer", include a screenshot of the Tensorboard graph showing the comparison between training from scratch and fine-tuning (5 points).

NOTE: this problem requires fairly long training time on Colaboratory (on the order of hours) so be sure to plan accordingly! Checkpoints should be saved in your Drive after training, so you can pick up where you left off after different parts of the problem.

Tracking (40 points)

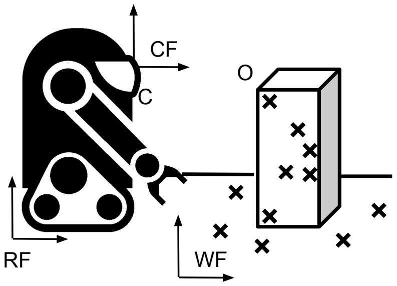

Figure 2

Lastly, a problem having to do with optical and scene flow. Consider the scenario depicted in Fig. 2, where an object O is being interacting by a person, while the robot observes the interaction with its RGB-D camera C. Files `rgb1.png`, `rgb2.png` and `rgb3.png` contain three RGB frames of the interaction as observed by the robot. Files `depth1.txt`, `depth2.txt` and `depth3.txt` contain the registered depth values for the same frames.

a. Detect and track the $N = 200$ best point features using Luca-Kanade point feature optical flow algorithm on the consecutive images, initialized with Tomasi-Shi corner features. Use the opencv implementation (look at this tutorial for more information: `https://docs.opencv.org/3.4/d4/dee/tutorial_optical_flow.html`). For the termination criteria use the number of iterations and the computed pixel residual from *Bouget*, 2001. Use only 2 pyramid levels and use minimum eigen values as an error measure. What is the lowest quality level of the features detected in the first frame? Provide a visualization of the flow for block sizes 15×15 and 75×75. Explain the differences observed. Is it always better a larger or a smaller value? What is the trade-off? **[15 points]**

   Note: You will need the opencv-python package. Please make sure that you have this installed.

b. Use the registered depth maps and the camera intrinsics provided in the starter code to compute the sparse scene flow (i.e. 3D positions of the tracked features between frames). (Note that depth maps contain NaN values. Features that have NaN depth value in any of the frames should be excluded in the result.). **[25 points]**