

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

(Winter 2021)

Due: **Friday, February 25**

Overview

This PSET will involve concepts from lectures 8, 10, 11, 12, and 13. It will involve 3D reconstruction, representation learning, supervised and unsupervised monocular depth estimation, as well as optical and scene flow. Although there are 5 problems so this PSET may look daunting, most of the problems require comparatively little work. Some of these problems are new for this year, so please report any issues you have on Ed.

Submitting

Please put together a PDF with your answers for each problem, and submit it to the appropriate assignment on Gradescope. We recommend you to add these answers to the latex template files on our website, but you can also create a PDF in any other way you prefer. For the written report, in the case of problems that just involve implementing code you can include only the final output and in some cases a brief description if requested in the problem. There will be an additional coding assignment on Gradescope that has an autograder that is there to help you double check your code. Make sure you use the provided ".py" files to write your Python code. Submit to both the PDF and code assignment, as we will be grading the PDF submissions and using the coding assignment to check your code if needed.

For submitting to the autograder, just create a zip file containing the p1,p2,p3,p4, and p5 folders. Before zipping, if you have finished these problems problem 2 and problem 3 please download your .ipynb and replace the initial file with that, and make sure to include you finished code in p2/code and p3/code.

Space Carving (40 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 carving¹, 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. **[5 points]**
- (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

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

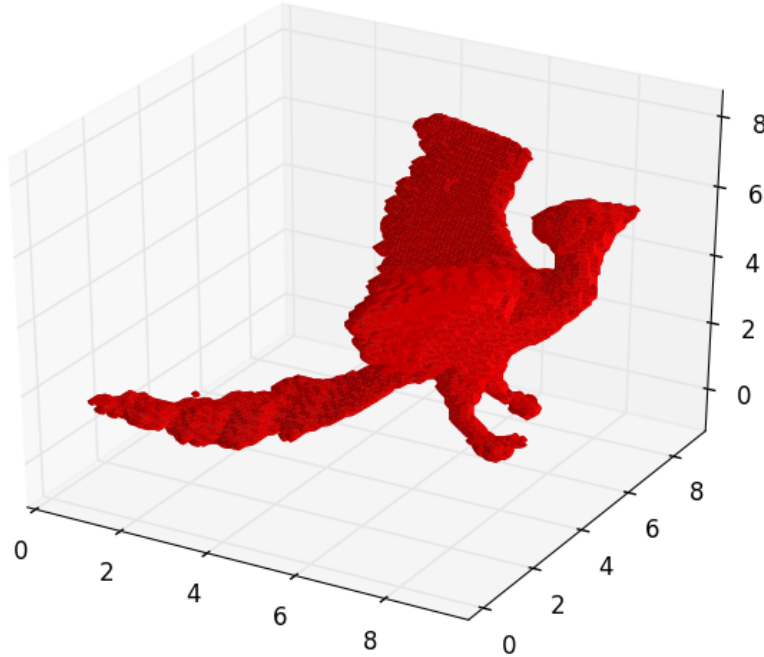


Figure 1: Our final carving using the true silhouette values

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 point]**
 - (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 (10 points)

In this problem, you'll implement a method for image representation learning for classifying clothing items from the Fashion MNIST dataset.

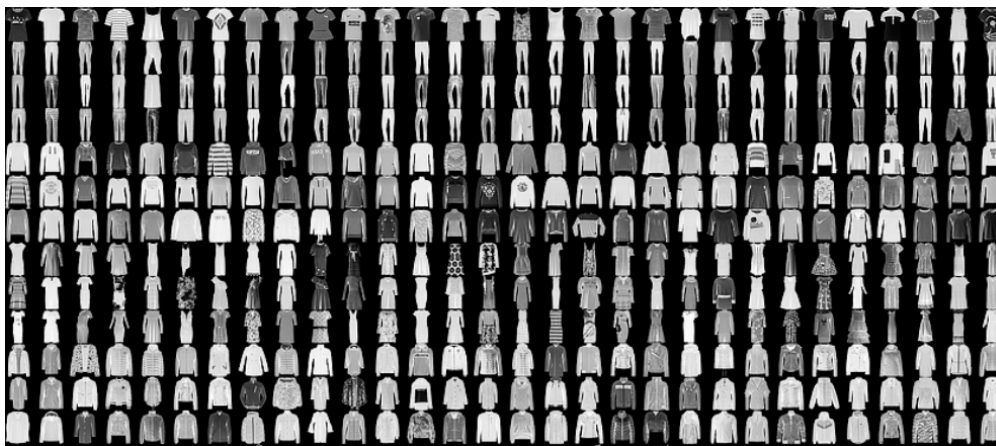


Figure 2: The Fashion MNIST dataset

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 “p3/RepresentationLearning.ipynb” and the contents inside ‘p3/code’ 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 “Fashion MNIST Data Preparation”, include the 3 by 3 grid visualization of the Fashion MNIST data [**1 point**].
- b. Once you finish the section “Training for Fashion MNIST Class Prediction”, include the two graphs of training progress over 10 epochs, as well as the test errors [**4 points**].
- c. Once you finish the section “Representation Learning via Rotation Classification”, include the 3 by 3 grid visualization and training plot, as well as the test error [**4 points**].
- d. Once you finish the section “Fine-Tuning for Fashion MNIST classification”, include all 3 sets of graphs from this section, as well as the test errors [**1 point**].

Fun fact: the method you just implemented is from the ICLR2018 paper “Unsupervised Representation Learning by Predicting Image Rotations”.

Monocular Depth Estimation (10 points)

Now that you’ve had some experience with representation learning on a small dataset, we’ll move on to the more complex task of training a larger model on the task of monocular depth estimation.

We will be implementing the approach taken in “High Quality Monocular Depth Estimation via Transfer Learning”, which showed that taking a big model pre-trained on classifying objects from

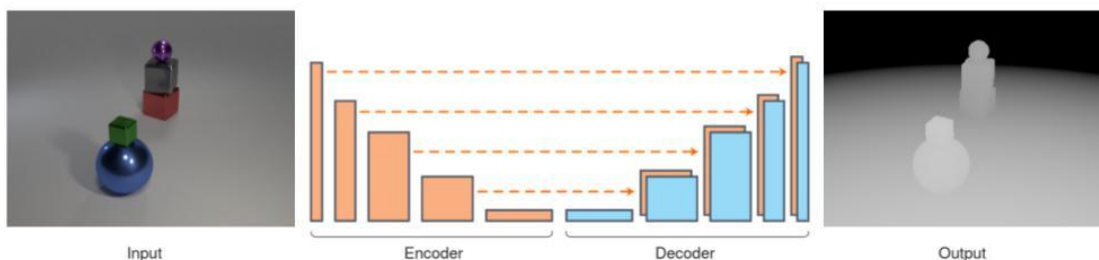


Figure 3: The task this problem will involve

the ImageNet dataset helps a lot with training that model to perform monocular depth estimation on the NYU Depth v2 dataset. We created a smaller and simpler dataset that is a variation on the CLEVR dataset for this problem that we call CLEVR-D, so we'll actually not be using the pre-trained features, but otherwise the approach is the same.

As with the previous problem, you'll be working with Colaboratory, so once again begin by uploading the contents of 'p3/code' 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:

- Once you finish the section "Checking out the data", include the grid visualization of the CLEVR-D data [5 points].
- Once you finish the section "Training the model", include a screenshot of your train and test losses from Tensorboard as well as the final outputs of the network [5 points].

Extra Credit As described at the bottom of the Colab notebook, you may optionally try to do representation learning by using an autoencoder, and see if that helps with training for monocular depth prediction. If you decide to do this, include a several sentence summary of how you went about it as well as sentence or two about the results, plus plots from Tensorboard and visualizations of the network in action [20 points].

Unsupervised monocular depth estimation (20 points)

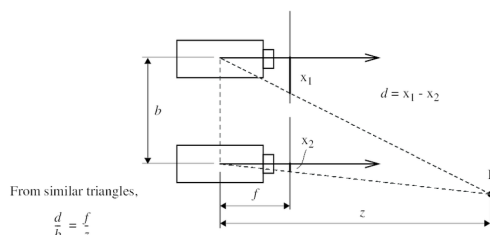


Figure 4

In this problem, we will take a step further to train monocular depth estimation networks without ground-truth training data. Although neural networks best train with large-scale training data, it is often challenging to collect ground-truth data for every domain of problem. For example, Microsoft Kinect, one of the most popular depth camera, uses infrared camera that does not

work outdoors, and training monocular depth estimation networks for outdoor scenes can be more challenging.

We instead will utilize the knowledge we learned about stereo computer vision in this course to train monocular depth estimation networks without ground-truth data. In this problem, we will train a network to predict disparity. As shown in Figure 4, disparity(d) is simply inverse proportional to depth(z), which still serves our purpose. Given a pair of left and right view of rectified images as inputs, we can synthesize right image by shifting left image toward right by the disparity and vice versa. We will utilize this trait to synthesize both left and right images and enforce them to look similar to the original left and right images.

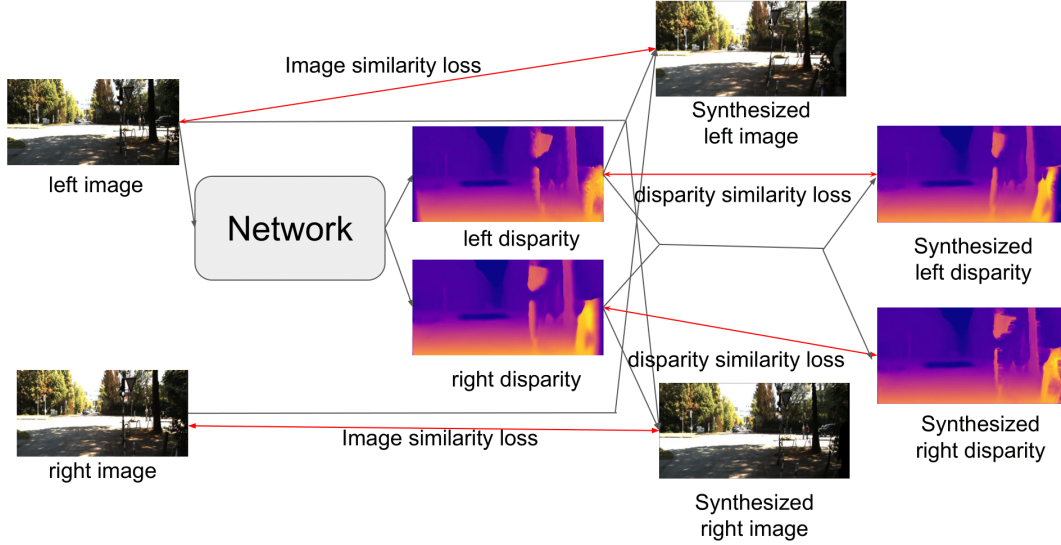


Figure 5

Figure 5 is a summary of how unsupervised monocular depth estimation works. This method is derived from the paper "Unsupervised Monocular Depth Estimation with Left-Right Consistency". The networks take left view of the stereo image img_l as input and outputs two disparity maps $disp_l$ (the disparity map of the left view that maps the right image to the left) and $disp_r$ (the disparity map of the right view that maps the left image to the right). Although the network only takes left image as an input, we train the network to predict disparity of both left and right sides. This design allows us to make monocular depth prediction possible (i.e. does not take stereo images as input) and enforce cycle consistency between the left and right view of the stereo images.

Then, assuming that the input images are rectified, we can generate left and right images from the predicted disparities. To be more concrete, using the left disparity $disp_l$, we synthesize left image and disparity as following: $img'_l = \text{generate_image_left}(img_r, disp_l)$ and $disp'_l = \text{generate_image_left}(disp_r, disp_l)$. Similarly, using the right disparity $disp_r$, we synthesize right image and disparity as following: $img'_r = \text{generate_image_right}(img_l, disp_r)$ and $disp'_r = \text{generate_image_right}(disp_l, disp_r)$. We will ask you to implement `generate_img_left` and `generate_img_right` in this pset.

In order to predict a reasonable disparity that can shift left image to right and vice versa, we compare the synthesized image with real image: $L_{img} = \text{compare}_i(img'_l, img_l) + \text{compare}_i(img'_r, img_r)$. For completeness, compare_i is L1 and SSIM.

In order to enforce cycle consistency between the left and right disparities, we compare the synthesized disparity with predicted disparity: $L_{disp} = \text{compare}_d(disp'_l, disp_l) + \text{compare}_d(disp'_r, disp_r)$.



Figure 6: Input and output (left/right disparity) of trained monocular depth estimation networks.

For completeness, compare_d is L1.

Please fill in the codes at `p4/problems.py` as outlined below. For running the code, you'll have to instal PyTorch and torchvision, and then you can test it out by running it locally `'python problems.py'`. Alternatively, you can use the ipython notebook `'problems.ipynb'` with Google Colab as in the previous two problems – for this problem you can just upload the files directly to the . Optionally, if you want to train the network yourself, you may follow the instruction on the README file. We will not ask you to train the model since it is too computation-intensive.

- a. Before we get started, we would like you to implement a data augmentation function for stereo images that randomly flips the given image horizontally. In neural networks, data augmentation takes a crucial role in better generalization of the problem. One of the most common data augmentation when using 2D images as input is to randomly flip the image horizontally. One interesting difference in our problem setup is that we take a pair of rectified stereo images as input. In order to maintain the stereo relationship after the horizontal flip, it requires a special attention. Please fill in the code to implement the data augmentation function. In your report include the images generated by this part of the code (no need to include the input images). **[5 points]**
- b. Implement a function `bilinear_sampler` which shifts the given horizontally given the disparity. The core idea of unsupervised monocular depth estimation is that we can generate left image from right and vice versa by sampling rectified images horizontally using the disparity. We will ask you to implement a function that simply samples image with horizontal displacement as given by the input disparity. In your report include the images generated by this part of the code (no need to include the input images). **[5 points]**
- c. Implement functions `generate_image_right` and `generate_image_left` which generates right view of the image from left image using the disparity and vice versa. This will be a simple one-liner that applies `bilinear_sampler`. In your report include the images generated by this part of the code (no need to include the input images). **[5 points]**
- d. In Figure 6, we visualize output of the networks trained with the losses you have implemented. You may notice that there are some boundary artifacts on the left side of the left disparity and right side of the right disparity. Briefly explain why. **[5 points]**

Tracking with Optical and Scene flow (20 points)

Lastly, a problem having to do with optical and scene flow. Consider the scenario depicted in Fig. 7, where an object O is being interacting by a person, while the robot observes the interaction with its RGB-D camera C . Files `rgb01.png`, `rgb02.png`, ..., `rgb10.png` in the `globe1` folder contain ten RGB frames of the interaction as observed by the robot. Files `depth01.txt`, `depth02.txt`, ..., `depth10.txt` contain the corresponding registered depth values for the same frames.

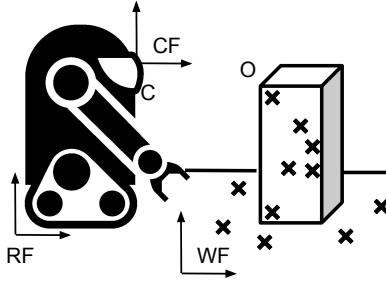


Figure 7

- 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? [5 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.). [5 points]
- c. Now let's switch to another scenario of a person opening a book. We have the same format of data as in part(a); the ten RGB frames and the corresponding depth files are available in the `book` folder. Run the algorithm you implemented for part(a) in this new setting and provide a visualization of the flow for block sizes 75×75 . How does the result look qualitatively, compared to the visualization from part(a)? How many features are we tracking in this new scenario and how does it compare to part(a)? Give a brief explanation why it is the case.[5 points]
- d. Next, we will be comparing dense optical flow tracking results from two different methods: [Gunnar Farnebeck's algorithm](#), and [FlowNet 2.0](#). First, run the provided `p5.py` script to get the dense optical flow for two frames from the set used in part b, and two frames from the [Flying Chairs dataset](#). The resulting dense optical flow images should be saved to a file titled `farnebeck-(imagename).png`; include this in your report.

Describe any artifacts you find between `farnebeck-globe.png` and `farnebeck-globe2.png`. Does one type of image appear have a better optical flow result? What differences between the image pairs might have caused one optical flow result to appear clearer than the other?[2 points]

- e. Next, go to [Google Colab](#) and upload the included `flownet.ipynb` notebook. Once loading the notebook, follow the included instructions to start a GPU runtime, and upload the respective images from the `p5_data` folder used in the last step (`p5_data/globe2/rgb01.png`,

p5_data/globe2/rgb02.png, p5_data/globe1/rgb04.png,
p5_data/globe1/rgb06.png, p5_data/chairs/frame_chairs_1.png,
p5_data/chairs/frame_chairs_2.png) to the notebook files tab. Run the cells in the notebook to produce the optical flow between the pairs of images, and include these flow images (and the reconstructed images) in your report. Note any qualitative differences you see between the optical flow results for the different pairs of images. Did one of the optical flow results and reconstructions from a specific pair of the images look less noisier than the others? Describe why you think this might be the case, and if there are any ways in which the performance on the other pairs of images could be improved with the FlowNet model. **[3 points]**