CS231A: Computer Vision, From 3D Perception to 3D Reconstruction and beyond Homework #4

(Spring 2024) Due: **Saturday, June 6**

Overview

This PSET will involve concepts from lectures 12, 13, 14, and 15. It will involve optical and scene flow, Kalman Filters with monocular and stereo vision, and a learned observation model. Although there are 4 problems so this PSET may look daunting, it is mostly instructions, and most of the problems require comparatively little work. You will **need** to use Colab only for Q1 (extra credit), and it is recommended for Q2-4.

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.

The assignment will require you to complete the provided python files and create a PDF for written answers. The instructions are **bolded** for parts of the problem set you should respond to with written answers. To create your PDF for the written answers, we recommend you add your answers to the latex template files on our website, but you can also create a PDF in any other way you prefer. To implement your code, make sure you modify the provided ".py" files in the code folder.

For the written report, in the case of problems that just involve implementing code, you will only need to include the final output (where it is requested) 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 all the files in the 'code' folder. Makes sure to add your finished code to the '.py' files after your code works in colab.

1 Extra Credit - Unsupervised Monocular Depth Estimation (25 points)

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 1, disparity(d) is simply inverse

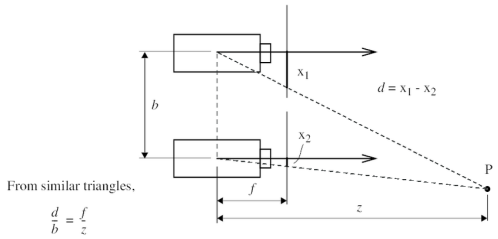

Figure 1

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.

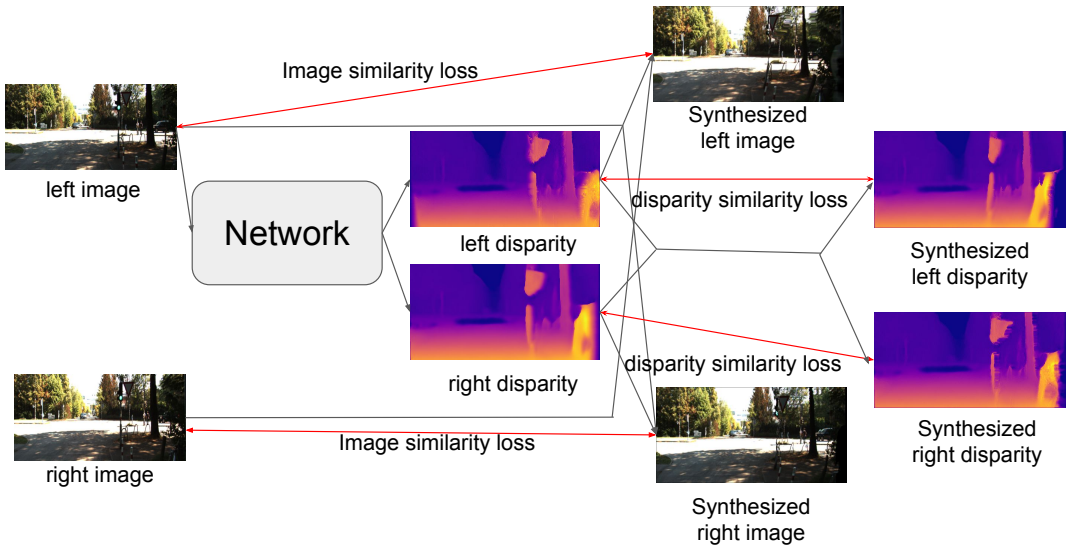

Figure 2

Figure 2 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

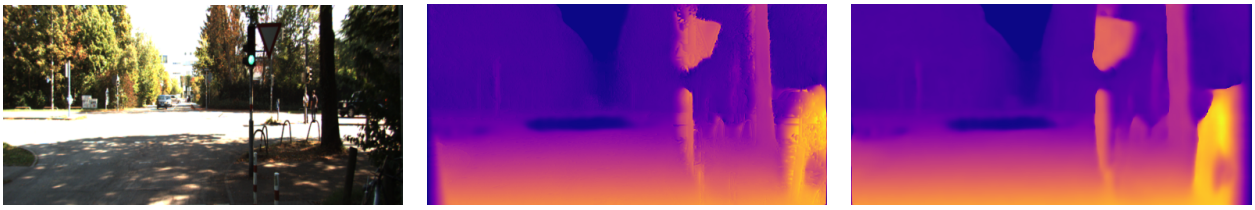

Figure 3: Input and output (left/right disparity) of trained monocular depth estimation networks.

generate_image_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 enforces cycle consistency between the left and right disparities, we compare the synthesized disparity with predicted disparity: $L_{disp} = \text{compare}_d(disp'_l, disp_l) + \text{compare}(disp'_r, disp_r)$. For completeness, compare_d is L1.

Please fill in the missing parts of the code in **p1/problems.py** as outlined below. For running the code, you'll have to install PyTorch and torchvision, and then you can test it out by running it locally 'python problems.py'. **Alternatively**, you can use the iPython notebook 'PSET4.ipynb' with Google Colab as for the rest of the problems. If you want to train the network yourself, you may follow the instruction . We will not ask you to train the model for this question however 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 for included plots]**

b. **Can you think of any other techniques discussed previously in the course that we can use to apply data augmentation for this task? [5 points for write-up]**

c. 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 for images]**

d. 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 for images]**

e. In Figure 3, 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 these artifacts may exist. [5 points]**

2 Extended Kalman Filter with a Nonlinear Observation Model (30 points)

Consider the scenario depicted in Figure 4 where a robot tries to catch a fly that it tracks visually with its cameras. To catch the fly, the robot needs to estimate the 3D position $\mathbf{p}_t \in \mathbb{R}^3$ and linear

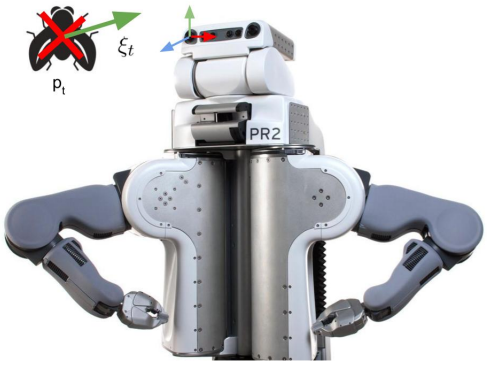

Figure 4

velocity $\xi_t \in \mathbb{R}^3$ of the fly with respect to its camera coordinate system. The fly is moving randomly in a way that can be modelled by a discrete time double integrator:

$$\mathbf{p}_{t+1} = \mathbf{p}_t + \Delta t \xi_t \tag{1a}$$
$$\xi_{t+1} = 0.8\xi_t + \Delta t \mathbf{a}_t \tag{1b}$$

where the constant velocity value describes the average velocity value over Δt and is just an approximation of the true process. Variations in the fly's linear velocity are caused by random, immeasurable accelerations $\mathbf{a}_t$. As the accelerations are not measurable, we treat it as the process noise, $\mathbf{w} = \Delta t \mathbf{a}_t$, and we model it as a realization of a normally-distributed white-noise random vector with zero mean and covariance Q: $\mathbf{w} \sim N(0, Q)$. The covariance is given by $Q = \text{diag}(0.05, 0.05, 0.05)$

The vision system of the robot consists of (unfortunately) only one camera. With the camera, the robot can observe the fly and receive noisy measurements $\mathbf{z} \in \mathbb{R}^2$ which are the pixel coordinates (u, v) of the projection of the fly onto the image. We model this projection mapping of the fly's 3D location to pixels as the observation model h:

$$\mathbf{z}_t = h(\mathbf{x}_t) + \mathbf{v}_t \tag{1c}$$

where $\mathbf{x} = (\mathbf{p}, \xi)^T$ and $\mathbf{v}$ is a realization of the normally-distributed, white-noise observation noise vector: $\mathbf{v} \sim N(0, R)$. The covariance of the measurement noise is assumed constant and of value, $R = \text{diag}(5, 5)$.

We assume a known 3x3 camera intrinsic matrix:

$$K = \begin{bmatrix} 500 & 0 & 320 & 0 \\ 0 & 500 & 240 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{1d}$$

a. Let $\Delta t = 0.1s$. Find the system matrix A for the process model, and implement the noise covariance functions (Implement your answer in the *system_matrix*, *process_noise_covariance*, and *observation_noise_covariance* functions in p2.py). **[5 points]**

b. Define the observation model h in terms of the camera parameters (Implement your answer in the *observation* function in p2.py). **[5 points for code]**

c. Initially, the fly is sitting on the fingertip of the robot when it is noticing it for the first time. Therefore, the robot knows the fly's initial position from forward kinematics to be at $\mathbf{p}_0 = (0.5, 0, 5.0)^T$ (resting velocity). Simulate in Python the 3D trajectory that the fly takes as well as the measurement process. This requires generating random acceleration noise and observation noise. Simulate for 100 time steps. **Attach a plot of the generated trajectories and the corresponding measurements. [5 points for plot]**

d. Find the Jacobian H of the observation model with respect to the fly's state $\mathbf{x}$. (Implement your answer of H in function *observation_state_jacobian* in p2.py.)**[5 points for code]**

e. Now let us run an Extended Kalman Filter to estimate the position and velocity of the fly relative to the camera. You can assume the aforementioned initial position and the following initial error covariance matrix: $P_0 = \text{diag}(0.1, 0.1, 0.1)$. The measurements can be found in `data/Q2E_measurement.npy`. **Plot the mean and error ellipse of the predicted measurements over the true measurements. Plot the means and error ellipsoids of the estimated positions over the true trajectory of the fly. The true states are in** `data/Q2E_state.npy` **[5 points for plots]**

f. **Discuss the difference in magnitude of uncertainty in the different dimensions of the state. [5 points for write-up]**

3 From Monocular to Stereo Vision (30 points)

Now let us assume that our robot got an upgrade: Someone installed a stereo camera and calibrated it. Let us assume that this stereo camera is perfectly manufactured, i.e., the two cameras are perfectly parallel with a baseline of $b = 0.2$. The camera intrinsics are the same as before in Question 1.

Now the robot receives as measurement $\mathbf{z}$ a pair of pixel coordinates in the left image (u^L, v^L) and right image (u^R, v^R) of the camera. Since our camera system is perfectly parallel, we will assume a measurement vector $\mathbf{z} = (u^L, v^L, d^L)$ where d^L is the disparity between the projection of the fly on the left and right image. We define the disparity to be positive. The fly's states are represented in the left camera's coordinate system.

a. Find the observation model h in terms of the camera parameters (Implement your answer in function *observation* in p3.py). **[5 points for code]**

b. Find the Jacobian H of the observation model with respect to the fly's state x. (Implement H in function *observation_state_jacobian* in p3.py) **[5 points for code]**

c. What is the new observation noise covariance matrix R? Assume the noise on (u^L, v^L), and (u^R, v^R) to be independent and to have the same distribution as the observation noise given in Question 1, respectively. (Implement R in function *observation_noise_covariance* in p3.py). **[5 points for write-up]**

d. Now let us run an Extended Kalman Filter to estimate the position and velocity of the fly relative to the left camera. You can assume the same initial position and the initial error covariance matrix as in the previous questions. **Plot the means and error ellipses of the predicted measurements over the true measurement trajectory in both the left and right images.** The measurements can be found in `data/Q3D_measurement.npy`. Plot the means and error ellipsoids of the estimated positions over the true trajectory of the fly. The true states are in `data/Q3D_state.npy` Include these plots here.**[5 points for images]**

e. In this Question, we are defining $\mathbf{z} = (u^L, v^L, d^L)^T$. Alternatively, we could reconstruct the 3D position $\mathbf{p}$ of the fly from its left and right projection (u^L, v^L, u^R, v^R) through triangulation and use $\mathbf{z} = (x, y, z)^T$ directly. **Discuss the pros and cons of using (u^L, v^L, d^L) over (x, y, z)! [5 points for write-up]**

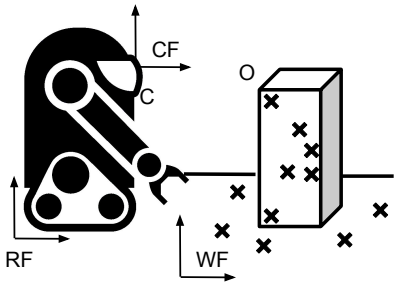

Figure 5

4 Linear Kalman Filter with a Learned Inverse Observation Model (20 points)

Now the robot is trying to catch a ball. So far, we assumed that there was some vision module that would detect the object in the image and thereby provide a noisy observation. In this part of the assignment, let us learn such a detector from annotated training data and treat the resulting detector as a sensor.

If we assume the same process as in the first task, but we have a measurement model that observes directly noisy 3D locations of the ball, we end up with a linear model whose state can be estimated with a Kalman filter. Note that since you are modifying code from previous parts and are implementing your own outlier detection for part C, there is no autograder for this problem - we will be grading based on your plots.

a. In the folder `data/Q4A_data` you will find a training set of 1000 images in the subfolder `training_set` and the file `Q4A_positions_train.npy` that contains the ground truth 3D position of the red ball in the image. We have provided you with the notebook LearnedObservationModel.ipynb that can be used to train a noisy observation model. As in PSET 3, use this notebook with Google Colab to do this – note that you'll need to upload the data directory onto a location of your choosing in Drive first. Alternatively, if you have an M1+ -chip Mac, you can use the commented line `device = torch.device('mps')` to run locally by uncommenting it (optional). **Report the training and test set mean squared error in your write-up. [5 points]**

b. In the folder `data/Q4B_data` you will find a set of 1000 images that show a new trajectory of the red ball. Run your linear Kalman Filter using this sequence of images as input, where your learned model provides the noisy measurements (the logic for this is provided in `PSET4.ipynb`). Now you can work on using the model by completing `p4.py`. Tune a constant measurement noise covariance appropriately, assuming it is a zero mean Gaussian and the covariance matrix is a diagonal matrix. Plot the resulting estimated trajectory from the images, along with the detections and the ground truth trajectory (the logic for this is provided in the starter code). **[5 points for plots]**

c. Because the images are quite noisy and the red ball may be partially or completely occluded, your detector is likely to produce some false detections. In the folder `data/Q4D_data` you will find a set of 1000 images that show a trajectory of the red ball where some images are blank (as if the ball is occluded by a white object). Discuss what happens if you do not reject these outliers but instead use them to update the state estimate. Like in the previous question, run your linear Kalman Filter using the sequence of images as input that are corrupted by occlusions (this is also provided in the notebook `LearnedObservationModel.ipynb`). **Plot the resulting estimated trajectory of the ball over the ground truth trajectory. Also plot the 3-D trajectory in 2-D (x vs. z) and (y vs. z) to better visualize what happens to your filter.** [**5 points**]

d. Design an outlier detector and use the data from `data/Q4D_data`. Provide the same plots as in part c with `filter_outliers=True`. **Explain how you implemented your outlier detector and add your code to the report. Hint: Your observation model predicts where your measurement is expected to occur and its uncertainty.** [**5 points**]