Project 4: Depth Estimation Using Stereo (part 1)

CS 6476

Spring 2021

Brief

- Due: Apr 2, 2021 11:59PM (with part 2)
- Project materials including report template: proj4_part1.zip
- Hand-in: through Gradescope (with part 2)
- Required files: `<your_gt_username>.zip`, `<your_gt_username>_proj4.pdf`

Overview

The goal of this project is to create stereo depth estimation algorithms, both classical and deep learning based. For classical stereo depth estimation algorithms, you will be using deterministic functions to compare patches and compute a disparity map. For deep learning based algorithms, you will be using a learning method to estimate the disparity map. There will be two parts in this project, the first of which is described in this handout. You will implement functions in `part1_*.py` to generate random patches, evaluate the similarity of those patches, and then compute the disparity map for several images. The corresponding notebook for this section is `part1_simple_stereo.ipynb`. Part 2 of this project (including its corresponding handout) will be released separately.

Setup

1. Install Miniconda. It doesn't matter whether you use Python 2 or 3 because we will create our own environment that uses python3 anyways.
2. Download and extract the part 1 starter code.
3. Create a conda environment using the appropriate command. On Windows, open the installed "Conda prompt" to run the command. On MacOS and Linux, you can just use a terminal window to run the command, Modify the command based on your OS (`linux`, `mac`, or `win`): `conda env create -f proj4_env_<OS>.yml`. If you're running into issues building the environment, try running `conda update --all` first.
4. This will create an environment named "cs6476_proj4". Activate it using the Windows command, `activate cs6476_proj4` or the MacOS / Linux command, `conda activate cs6476_proj4` or `source activate cs6476_proj4`
5. Install the project package, by running `pip install -e .` inside the repo folder. This might be unnecessary for every project, but is good practice when setting up a new `conda` environment that may have `pip` requirements.
6. Run the notebook using `jupyter notebook ./proj4_code/part1_simple_stereo.ipynb`

7. After implementing all functions, ensure that all sanity checks are passing by running `pytest proj4_unit_tests` inside the repo folder.
8. Complete part 2 (template and instructions to be released separately).

1 Simple stereo by matching patches

Introduction

We know that there is some encoding of depth when images are captured using a stereo rig, much like human eyes. You can try a simple experiment to see the stereo effect in action. Try seeing a scene with only your left eye. Then close your left eye and see using your right eye. Make the transition quickly. You should notice a *horizontal* shift in the image perceived. Can you comment on the difference in shift for different objects when you do this experiment? Is it related to the depth of the objects in some way?

In this section, we will generate a **disparity map**, which is the map of horizontal shifts estimated at each pixel. We will start working on a simple algorithm, which will then be improved to calculate more accurate disparity maps.

The notebook corresponding to this part is `part1_simple_stereo.ipynb`.

1.1 Random dot stereogram

It was once believed that in order to perceive depth, one must either match feature points (like SIFT) between left and right images, or rely upon clues such as shadows.

A random dot stereogram eliminates all other depth cues, and hence proves that a stereo setup is sufficient to get an idea of the depth of the scene. A random dot stereogram is generated by the following steps:

1. Create the left image with random dots at each pixel (0/1 values).
2. Create the right image as a copy of the left image.
3. Select a region in the right image and shift it horizontally.
4. Add a random pattern in the right image in the empty region created after the shift.

In `part1a_random_stereogram.py`, you will implement `generate_random_stereogram()` to generate a random dot stereogram for the given image size.

1.2 Similarity measure

To compare patches between left and right images, we will need two kinds of similarity functions:

1. Sum of squared differences (SSD):

$$SSD(A, B) = \sum_{i \in [0,H), j \in [0,W)} (A_{ij} - B_{ij})^2 \tag{1}$$

2. Sum of absolute differences (SAD):

$$SAD(A, B) = \sum_{i \in [0,H), j \in [0,W)} |A_{ij} - B_{ij}| \tag{2}$$

where A and B are two patches of height H and width W.

In `part1b_similarity_measures.py`, you will implement the following:

- `ssd_similarity_measure()`: Calculate SSD distance.
- `sad_similarity_measure()`: Calculate SAD distance.

1.3 Disparity maps

We are now ready to write code for a simple algorithm for stereo matching. You will need to follow the steps visualized in Figure 1:

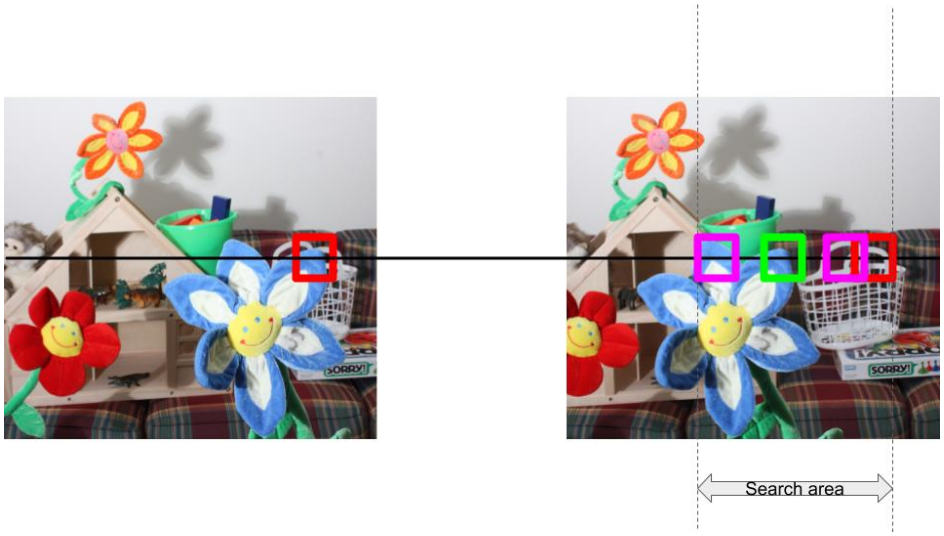
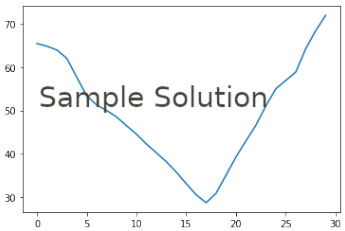


Figure 1: Example of a stereo algorithm.

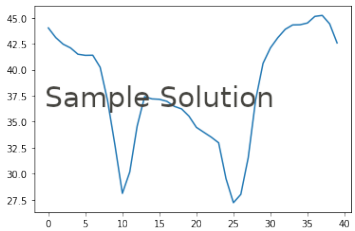
1. Pick a patch in the left image (red block), P_1.
2. Place the patch in the same (x, y) coordinates in the right image (red block). As this is binocular stereo, we will need to search for P_1 on the left side starting from this position. Make sure you understand this point well before proceeding further.
3. Slide the block of candidates to the left (indicated by the different pink blocks). The search area is restricted by the parameter `max_search_bound` in the code. The candidates will overlap.
4. We will pick the candidate patch with the minimum similarity error (green block). The horizontal shift from the red block to the green block in this image is the disparity value for the center of P_1 in the left image.

Note: the images have already been rectified, so we can search only a single horizontal scan line.

In `part1c_disparity_map.py`, you will implement `calculate_disparity_map()` (please read the documentation carefully!) to calculate the disparity value at each pixel by searching a small patch around a pixel from the left image in the right image.



(a) Convex profile.



(b) Non-convex profile.

Figure 2

1.4 Error profile analysis

Before computing the full disparity map, we will first analyze the similarity error distribution between patches. You will have to find two examples which display a close-to-convex error profile, and a highly non-convex profile, respectively. For reference, we provide the plots we obtained (see Figure 2). Based on your output visualizations and understanding of the process, answer the reflection questions in the report.

1.5 Real life stereo images

You will iterate through pairs of images from the dataset and calculate the disparity maps for images. The code is already given to you. You just need to compare the disparity maps and answer the reflection questions in the report.

1.6 Smoothing

One issue with the above results is that they aren't very smooth. Pixels next to each other on the same surface can have vastly different disparities, making the results look very noisy and patchy in some areas. Intuitively, pixels next to each other should have a smooth transition in disparity (unless at an object boundary or occlusion).

In this part, we try to improve our results. One way of doing this is through the use of a smoothing constraint. The smoothing method we use is called semi-global matching (SGM) or semi-global block matching. Before, we picked the disparity for a pixel based on the minimum matching cost of the block using some metric (SSD or SAD). Now, the basic idea of SGM is to penalize disparity computations which are very different than their pixel-wise neighbors by adding a penalty term on top of the matching cost term. SGM approximately minimizes the global (over the entire image) energy function.

$$E(D) \leq \sum_{p}(C(p, D_p) + \sum_{q} PT(|D_p - D_q|))$$

$C(p, D_p)$ is the matching cost for a pixel with disparity D_p, q is a neighboring pixel, and $PT(\cdot)$ is some penalty function penalizing the difference in disparities. You can read more about how this method works and is optimized here: Semi-Global Matching - Motivation, Developments, and Applications and Stereo Processing by Semi-Global Matching and Mutual Information.

You will not need to implement SGM by yourself. But to help understand SGM, you will implement a function which computes the **cost volume**. You have already written code to compute disparity map. Now you will extend that code to compute the cost volume. Instead of taking the argmin of the similarity error profile, we will store the tensor of error profiles at each pixel location along the third dimension. If we have an input image of dimension (H,W,C) and max search bound of D, the `cost_volume` will be a tensor of dimension (H,W,D). The cost volume at (i, j) pixel is the error profile obtained for the patch in the left image centered at (i, j).

In `part1c_disparity_map.py`, you will implement `calculate_cost_volume()` to calculate the disparity map.

Testing

We have provided a set of tests for you to evaluate your implementation. We have included tests for part 1 inside `part1_simple_stereo.ipynb` so you can check your progress as you implement each function. When you're done with the entire project, you can call additional tests by running `pytest proj4_unit_tests` inside the root directory of the project, as well as checking against the tests on Gradescope. *Your grade on the coding portion of the project will be further evaluated with a set of tests not provided to you.*

Rubric (part 1 only)

- +10 pts: `part1a_generate_random_stereogram()`
- +6 pts: `part1b_similarity_measures.py`
- +20 pts: `part1c_disparity_map.py`