Project 4: Depth Estimation Using Stereo (part 2)

CS 6476

Spring 2021

Brief

- Due: Apr 5, 2021 11:59PM
- Project materials (including code for both parts & report template): proj4.zip
- Hand-in: through Gradescope
- 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. In part 1, you implemented classical stereo depth estimation algorithms using a deterministic function to evaluate patches and then get disparity map. In part 2, you will implement deep learning based algorithms to estimate the disparity map. Specifically, you will 1) implement the part for generating patch and architecture of MC-CNN model in `part2_*.py` and go through `part2_disparity.ipynb`. Make sure you pass all the sanity checks for part 2 before starting training. 2) use `part2_mc_cnn.ipynb` to go through the training and visualize the results of your model.

Setup

You can skip steps 1-5 if you've already started part 1.

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 project 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`
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. Section 2:
    - Part 2.1: Run the notebook using `jupyter notebook ./proj4_code/part2_disparity.ipynb`

- Part 2.2: Run the notebook `part2_mc_cnn.ipynb` and upload the zipped (`semiglobalmatching` and `proj4_code`) as `proj4.zip` to Colab.

7. Once you are done executing the Colab notebook, and are satisfied with your visualization, run the final cell, which will generate a file called `final_model_data.pth`. Download it and make sure you save that file in your `proj4_code` folder while submitting, as we will evaluate your model's performance as a hidden Gradescope test.

8. Generate the zip folder for the code portion of your submission once you've finished the project using `python zip_submission.py --gt_username <your_gt_username>`

2 Learning-based stereo matching

In the previous section, you saw how we can use simple concepts like SAD and SSD to compute matching costs between two patches and produce disparity maps. Now let's try something different – instead of using SAD or SSD to measure similarity, we will train a neural network and learn from the data directly.

Introduction

You'll implement what has been proposed in the paper [Zbontar & LeCun, 2015], and evaluate how it performs compared to classical cost matching approaches. The paper proposes several network architectures, but what we will be using is the `accurate` architecture for the Middlebury stereo dataset. This dataset provides a ground truth disparity map for each stereo pair, which means we know exactly where the match is supposed to be on the epipolar line. This allows us to extract many such matches and train the network to identify what type of patches should be matches and what shouldn't. You should definitely read the paper in more details if you're curious about how it works.

You don't have to worry about the dataset – we provide images in a ready-to-use format (with rectification). In fact, you won't be doing much coding in this part. Rather, you should focus on experimenting and thinking about *why*. Your report will have a lot of weight in this part, so try to be as clear as possible.

Note: The network in Part 2.2.1 can take around 15-30 mins to train on Colab. We suggest you **start early and don't wait until the last minute**.

2.1 PyTorch functions on CPU

In this part, we will implement an MCNET network architecture as described in the paper (See Figure 1), generate patches for the training process, and calculate disparity for MCNET.

The corresponding notebook for this part is `part2_disparity.ipynb`.

2.1.1 Network architecture

MCNET

We will follow the description of the "accurate" network for Middlebury dataset. The inputs to the network are 2 image patches, coming from left and right images. Each will pass through a series of convolution + ReLU layers. The extracted features are then concatenated and passed through additional fully connected + ReLU layers. The output is a single real number between 0 and 1, indicating the similarity between the two input images [Zbontar & LeCun, 2015]. In this case, since training from scratch will take a really long time to converge, you'll train from our pre-trained network instead. In order to load up the pre-trained network, you must first implement the architecture exactly as described below:

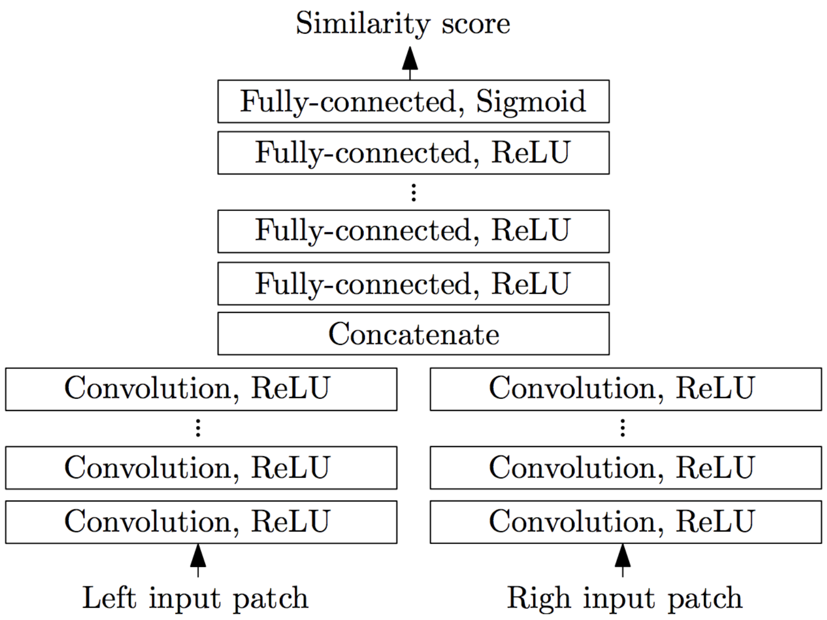

Figure 1: Visualization of network architecture.

Hyperparameters	#
input_patch_size	11×11
num_conv_layers	5
num_conv_feature_maps	112
conv_kernel_size	3
num_fc_layers	3
num_fc_units	384

For efficiency we will convolve both input images in the same batch (this means that the input to the network will be $2 \times batch\_size$). After the convolutional layers, we will then reshape them into $[batch\_size, conv\_out]$ where $conv\_out$ is the flattened output size of the convolutional layers. This will then be passed through a series of fully connected layers and finally a sigmoid layer to bound the output value to $[0, 1]$.

Here is an example of a network with $num\_conv\_layers = 1$ and $num\_fc\_layers = 2$:

```
conv_layers = nn.Sequential(
    nn.Conv2d(in_channel, num_feature_map, kernel_size=kernel_size, stride=1, padding=(
        kernel_size // 2)),
    nn.ReLU(),
    )

fully_connected_layers = nn.Sequential(
    nn.Linear(conv_out, num_hidden_units),
    nn.ReLU(),
    nn.Linear(num_hidden_units, 1),
    nn.Sigmoid()
)

conv_feature_batch = conv_layers(input_batch)
conv_feature_batch.reshape((batch_size, conv_out)
output_batch = fully_connected_layers(conv_feature_batch)
```

In `part2a_network.py`, you will implement the following network architecture:

- `MCNET`: Implement the network architecture as described in the paper.

2.1.2 Patch generation

In `part2b_patch.py`, you will implement `gen_patch()` to extract a patch from an image.

2.1.3 Disparity map calculation with MCNET

The core logic for calculating disparity for MCNET will remain the same, but we will have to do a few things differently. It will take around 1-2 mins to generate the disparity map if implemented correctly.

The steps required here are as follows:

1. We will operate on convolutional features instead of raw pixels. Pass the images through the convolutional block of MCNET to obtain the features.
2. Pick a patch in the left image features, P_1.
3. Calculate the search-space of corresponding patches in the right image features:
    (a) As before, place the patch in the corresponding location in the right image features, and slide it to obtain a sequence of window patches.
    (b) Concatenate these patches at the 0th dimension to form a batch of patches.
4. Compute the similarity values over the entire window using the similarity function provided to you. All the similarity values over the window will be present as a $(k \times 1)$ tensor.
5. Pick the patch with the minimum similarity error.

Note: It is important that the similarity calculation happens in parallel over the entire search window. Otherwise, the disparity calculation will take a really long time in the subsequent part.

In `part2c_disparity.py`, you will implement `mc_cnn_similarity()` and `calculate_mccnn_cost_volume()` to calculate the disparity value at each pixel using MCNET.

Note: Before proceeding to the next part, you need to ensure that **all sanity checks for this part are passing** by running `part2_disparity.ipynb` with jupyter notebook, and running `pytest proj4_unit_tests`

2.2 Train and evaluation on Google Colab

In this part, we will train the MCNET architecture and evaluate the overall performance.

Setup

We will be using Google Colab, which is a cloud-based Jupyter notebook environment. You can choose to run this section locally as well, especially if you have a good GPU, but the assignment is designed to run on Colab with GPU (this project is doable without a GPU, but a GPU makes the process much faster and frustration free.). These are the steps we follow:

1. Upload `part2_mc_cnn.ipynb` to Google Colab
2. Zip `semiglobalmatching` and `proj4_code` into `proj4.zip` and upload them to the Colab runtime.
3. Unzip the uploaded zip using `!unzip -qq uploaded\_file.zip -d ./`
4. In Colab, make sure you select "GPU" in the menu ("Runtime" → "Change runtime time" → "Hardware accelerator").

You will need to follow the instructions in `Setup`, `Compute Requirements`, and `DataLoader` in the notebook to download the necessary data and set up the environment in Google Colab.

2.2.1 Train MCNET

In this part, we will train a neural network that learns how to classify 2 patches as positive vs negative match. Your task is to train a best network by experimenting with the learning parameters. The following shows the experiments you need to complete for this part:

- Experiment with the learning rate: try using large (> 1) vs. small ($< 1e-5$) values. Based on your output visualizations, answer the reflection questions in the report.
- Experiment with the window size: In the previous part, we use window size of 11 as suggested in the paper, meaning that the input to the network will be patches of size 11x11. This corresponds to the block size that will be used when perform stereo matching later on. You can experiment with other window size, namely 5x5, 9x9, and 15x15 and compare the performance.
- Tune the training parameters and pick the best combination of hyperparameters with the best disparity map visualization. You should show the training loss plot in the report and answer the reflection questions in the report. **Typically, models with average error of around 20 tend to pass the Gradescope tests**.

2.2.2 Evaluate stereo matching

In this part, we will again generate the disparity map but this time from our newly trained matching cost network. We will use `calculate_mc_cnn_disparity` from `part2c_disparity.py` for this.

Note that all the required functions in Part 2.1 need to be implemented correctly before starting this part.

Hint: You don't have to re-train the network every time you want to evaluate, as long as your saved model is in Colab file system. Don't forget to change `load_path` to your best model.

Then we will evaluate your trained network as a stereo matching cost with the metrics used in the Middlebury leaderboard for stereo matching. For the bicycle image, you should see the improvement in using the trained network vs. SAD cost matching.

- ***avgerr***: average absolute error in pixels (lower is better)
- ***bad1***: percentage of bad pixels whose error is > 1 (lower is better)
- ***bad2***: percentage of bad pixels whose error is > 2 (lower is better)
- ***bad4***: percentage of bad pixels whose error is > 4 (lower is better)

Evaluate stereo matching with SGM

You will use the semi-global matching module in part1 and the `calculate_mccnn_cost_volume` in `part2c_disparity.py` to evaluate the disparity map generated by SAD method and MC-CNN model.

Based on your outputs, answer the reflection questions in the report.

3 Writeup

For this project (and all other projects), you must do a project report using the template slides provided to you. Do ***not*** change the order of the slides or remove any slides, as this will affect the grading process on Gradescope and you will be deducted points. In the report you will describe your algorithm and any decisions you made to write your algorithm a particular way. Then you will show and discuss the results of your algorithm. The template slides provide guidance for what you should include in your report. A good writeup doesn't just show results–it tries to draw some conclusions from the experiments. You must convert the slide deck into a PDF for your submission.

Testing

We have provided a set of tests for you to evaluate your implementation. We have included tests inside `part1_simple_stereo.ipynb` and `part2_disparity.ipynb` so you can check your progress as you implement each section. *Your grade on the coding portion of the project will be further evaluated with a set of tests not provided to you.*

Bells & whistles (extra points)

Also note that while we're closely following the proposed matching cost network from the paper, we're still skipping several bells and whistles post-processing components used in the paper, so the results are still far from perfect. You are free to add any components you think would be useful (doesn't have to be from the paper). There is a maximum of 10 pts extra credit for any interesting experiments beyond the outline we have. Be creative, and be clear and concise about what extra things you did in the report. Here are some starting points:

- Data augmentation for training: Rather than cropping patches directly, you can augment the data with slight rotation/affine transformations to make it more robust to noise and perspective change. Be careful not to transform too much that we lose the precision of the match. To earn extra credit, you must explain the augmentation you do, and show the improvement with adding it.
- Experiment with training from scratch: Get great performance by training from scratch using datasets like KITTI2015.

If you choose to do anything extra, **include your code implementation in `proj4_code/extra_credit.py`**, and add slides *after the slides given in the template deck* to describe your implementation, results, and analysis. Adding slides in between the report template will cause issues with Gradescope, and you will be deducted points. You will not receive full credit for your extra credit implementations if they are not described adequately in your writeup.

Rubric

- +10 pts: `part1a_random_stereogram.py`
- +6 pts: `part1b_similarity_measures.py`
- +20 pts: `part1c_disparity_map.py`
- +10 pts: `part2a_network.py`
- +4 pts: `part2b_patch.py`
- +15 pts: `part2c_disparity.py`
- +35 pts: Report
- -5*n pts: Lose 5 points for every time you do not follow the instructions for the hand-in format

Submission format

This is very important as you will lose 5 points for every time you do not follow the instructions. You will submit two items to Gradescope:

- `<your_gt_username>.zip` containing:
  - `proj4_code/` - directory containing all your code for this assignment

  - `additional_data/` - (optional) if you use any data other than the images we provide you, please include them here

- `<your_gt_usernamme>_proj4.pdf` - your report

Do ***not*** install any additional packages inside the conda environment. The TAs will use the same environment as defined in the config files we provide you, so anything that's not in there by default will probably cause your code to break during grading. Do ***not*** use absolute paths in your code or your code will break. Use relative paths like the starter code already does. Failure to follow any of these instructions will lead to point deductions. Create the zip file using `python zip_submission.py --gt_username <your_gt_username>` (it will zip up the appropriate directories/files for you!) and hand it in with your report PDF through Gradescope (please remember to mark which parts of your report correspond to each part of the rubric).

Credits

Assignment developed by James Hays, Frank Dellaert, Cusuh Ham, Ayush Baid, Jonathan Balloch, Patsorn Sangkloy, Vijay Upadhya, Jing Wu, Esther Gu, Anant Joshi and John Lambert. The dataset was obtained from the Middlebury stereo datasets. Smoothing code was obtained from here.