Report: Model Fitting & Multi-View Stereo

Computer Vision - Assignment 4

Ian SCHOLL
ischoll@student.ethz.ch

1. Introduction

1.1. Environment Setup

I used Miniconda on Linux Ubuntu 20.04.3 LTS. My CPU is Core™ i7-1185G7 @ 3.00GHz × 8. Training took me roughly 20 hours for MVS.

1.2. Hand In

See together with this report:

- The modified code file for line fitting: *line_fitting.py*
- The modified code files for MVS: *module.py & data_io.py*
- The trained model: *model_000003.ckpt*
- Some images of the results in the folder *images*
- Some more files & folders as in the sample folder structure in the assignment slides

2. Model Fitting

2.1. Line Fitting

2.1.1 Least-squares solution

One simply needs to apply the *np.linalg.lstsq* method correctly for this part.

2.1.2 RANSAC

We are interested in estimating a 2D line through (x,y) points. Therefore, my first intuition was to compute the distances as the *perpendicular* distance to the line, using the following formula:

$$d = \sqrt{\left(\frac{x_0 + my_0 - mk}{m^2+1} - x_0\right)^2 + \left(m\frac{x_0 + my_0 - mk}{m^2+1} + k - y_0\right)^2} = \frac{|k + mx_0 - y_0|}{\sqrt{1+m^2}}$$

Figure 1: Formula for perpendicular distance between point (x_0, y_0) and a line. For us: m = k and k = b [1]

However, there's also a second way to compute the distance of the points to the line, and that is by looking just at the vertical distance in y, as often done in practice for simplicity:

$$d = |y_0 - (k*x_0 + b)|$$

Both these distances can then be compared with the given threshold to classify inliers.

2.1.3 Results

Using a vectorized form to make that comparison and to compute the inlier mask, I received the following results in very short computation time (some milliseconds):

	Ground truth	**Linear regression**	**RANSAC, *perpendicular* dist.**	**RANSAC, *vertical* dist. in y**
k	1	0.6159656578755459	0.9643894946568977	0.9987449792570882
b	10	8.96172714144364	9.982921294966836	9.997009758791009

Indeed we can observe that linear regression doesn't quite get close to the ground truth, as the outlier are affecting its results. RANSAC pretty much matches the correct result as expected for both versions. We also see that for the given setup and random seeds, using the vertical distance gives the best performance. This gets very clear when looking at the following plot:

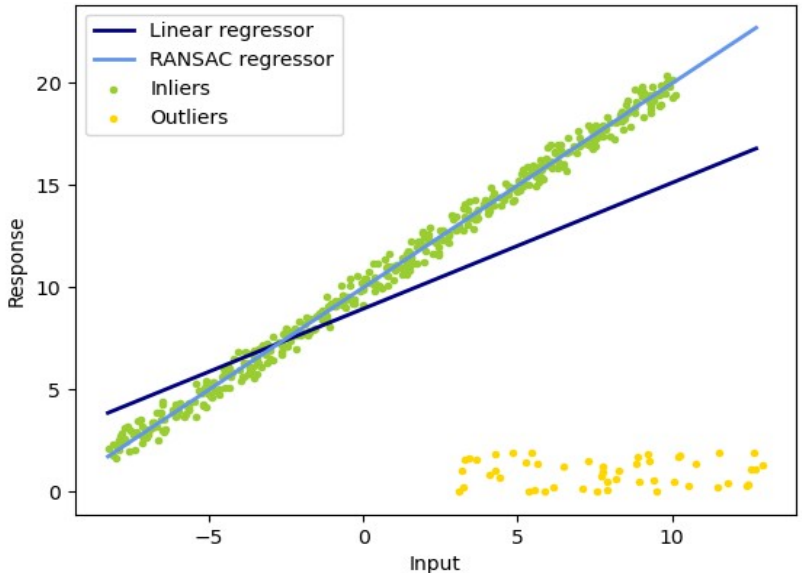

Figure 2: Comparison of linear regression and RANSAC line fitting (using vertical *distance in y)*

The line calculated with RANSAC closely follows the inlier data in green, whereas the linear regressor is biased by the outliers in yellow. This shows why RANSAC is such a useful and efficient method.

3. Mulit-View Stereo

3.1. Dataset

3.1.1 RGB Images

Using the Image methods of PIL, it is straightforward to import an RGB image. All its channels, currently with values in the range [0,255], simply have to be divided by 255 to obtain the wished for range of [0,1].

3.1.2 Camera Parameters

A .txt file can be read in line by line using the built in ‘open’ method. I then simply convert those strings into numpy arrays, combining the right lines and values to assign the E-matrix, K-matrix and depth range values.

3.2. Network

3.2.1 Feature Extraction

This network can be defined pretty straightforwardly with a single *torch.nn.Sequential* instance. I only had to pay attention on how to set the *paddings* in order for the outputs to have the desired dimensions. Additionally, I had to change the *dtype* of x in the forward call to *float()*, such that its dtype is in sync with the one of the *Sequential*.

3.2.2 Differentiable Warping

The differentiable warping clearly was the trickiest part of this assignment. The idea is to give the points in the image plane sort of a third dimension/variable, which is affected by all the possible depth values. In order to do this, I had to add a 1 to all the pixels, namely to homogenize them to $\mathbf{p}_i = [x_i\ y_i\ 1]^T$. Then we can use the already given projection to project these points from the reference frame to the source view frame using:

$$\mathbf{p}_{i,j} := \mathbf{p}_i(d_j) = (\mathbf{R}_{0,i}\mathbf{p}_i)*d_j + \mathbf{t}_{0,i}$$

However, we then ideally want to bring those projected pixels $\mathbf{p}_{i,j}$ back into a homogeneous form, which is why I divide through the 3rd element. Finally, I normalize these new x and y coordinates using the width and height respectively, as this is a crucial step in stabilizing the performance (see also question 1 in chapter 3.5). I have tried to train my model both with and without this normalization, and the effect was significant, as the loss only was able to reach low levels and produce good absolute depth error when using such a normalization.
Stacking all these projected, normalized x- and y-coordinates into a grid then allows to bi-linearly interpolate the source features (*src_fea*) using the suggested *torch.nn.functional.grid_sample* method.

3.2.3 Similarity Computation and Regularization

1. Here we first need to reshape the reference features and the warped source features. In order to do so, all that is necessary is to divide the channel dimension by the desired G (amount of groups), while simultaneously introducing a new dimension of size G in order to 'absorb' this structural change. It is then along that divided channel dimension that we can compute the dot product and take the mean as given in *formula (3.1)* of the assignment.

2. The *SimilarityRegNet* network I had to divide into several *torch.nn.Sequential* instances and layers, as some intermediate outputs are used together as inputs for later layers. Again, the main challenge consisted in using the right *paddings* such that on one hand the output dimension would correspond to what was asked for, as well as for the intermediate results to have the same dimensions to be summed up as inputs for later layers.

   Furthermore, the initial input x (namely the similarity S) had to be reshaped to 4 dimensions in order to perform the forward pass through the network (and again transformed to *float()*, such that its *dtype* is in sync with the one of the *Sequentials)*. Again, I have tried several ways of reshaping S. All of them resulted in similar end results and absolute depth errors. However, I was able to find the best performance and especially the fastest decrease in loss using first a transpose of dimensions 2 & 3 (1 & 2 in python syntax), and then multiplying the dimension D, along which we don't want to aggregate information on the image, with the batch dimension B. Like this, none of the elements get shuffled when reshaping the output later.

3.2.4 Depth Regression

This part was pretty straightforward, needing basically only one line to compute. As given in the formula of the assignment, all I had to do was summing over the product of the probability volume with all the depth values, whose dimensions I had to reshape in order to be compatible with the probability volume dimensions.

3.2.5 Loss Function

The L1 loss can easily be computed using the *torch.nn.functional.l1_loss* method, constantly adding up while looping over the batch dimension. Again I have tried two different versions here: for the first try I simply overlaid the given mask on the ground truth and estimated depth, letting the *l1_loss* average over all the values (therefore also the values put to 0 by the mask). This scales the loss by some factor. The second method was to *eliminate* all the ground truth and estimated depths with a 0 in the mask. The mean loss is then calculated only over the actual depths admitted by the mask, which shifts the loss by a factor.

However, such a scalar shift does not change the interpretation of every iteration, as the loss will still behave the same proportionally over the training. The first method was computationally less expensive and faster (which can make easily some hours of difference for the training in our case), and therefore I stuck with it; it also performed just as good as the second method looking at the absolute depth errors obtained in training.

3.2.6 Photometric Confidence

(Already implemented in the skeleton code given for the assignment)

3.3. Training

3.3.1 Results

The training overall took around 20 hours on my laptop. I *split it in 2 parts*, resuming and training 2 epochs each. After the 4th epoch of training and validation I have achieved the following results:

loss	abs_depth_error	thres_2mm_error	thres_4mm_error	thres_8mm_error
5.751222557646193	4.382861307689121	0.402429191086104	0.195002902288304	0.093362899784965

The loss (and absolute depth error) during training and validation has converged as follows:

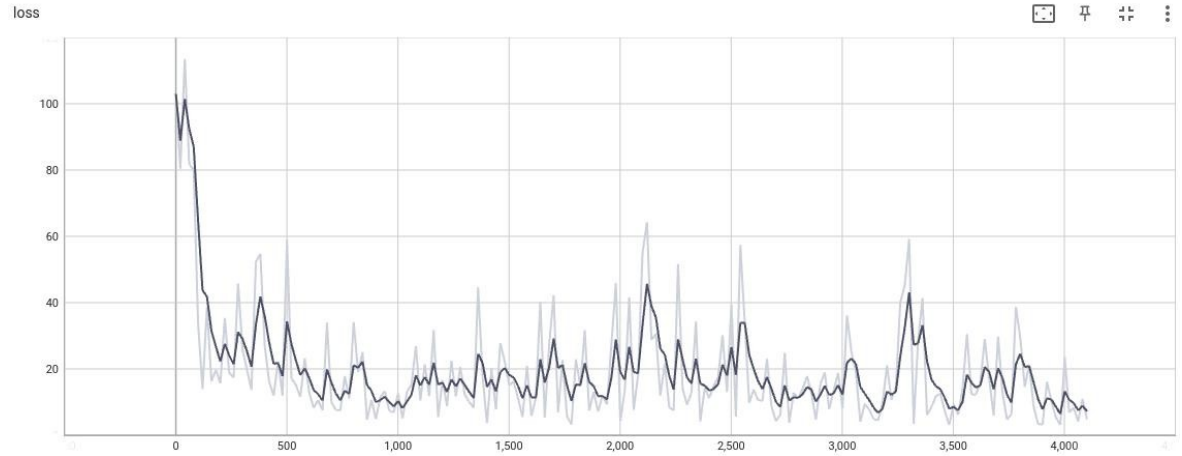

Figure 3: Convergence of the loss on the training dataset over the 4 epochs of training

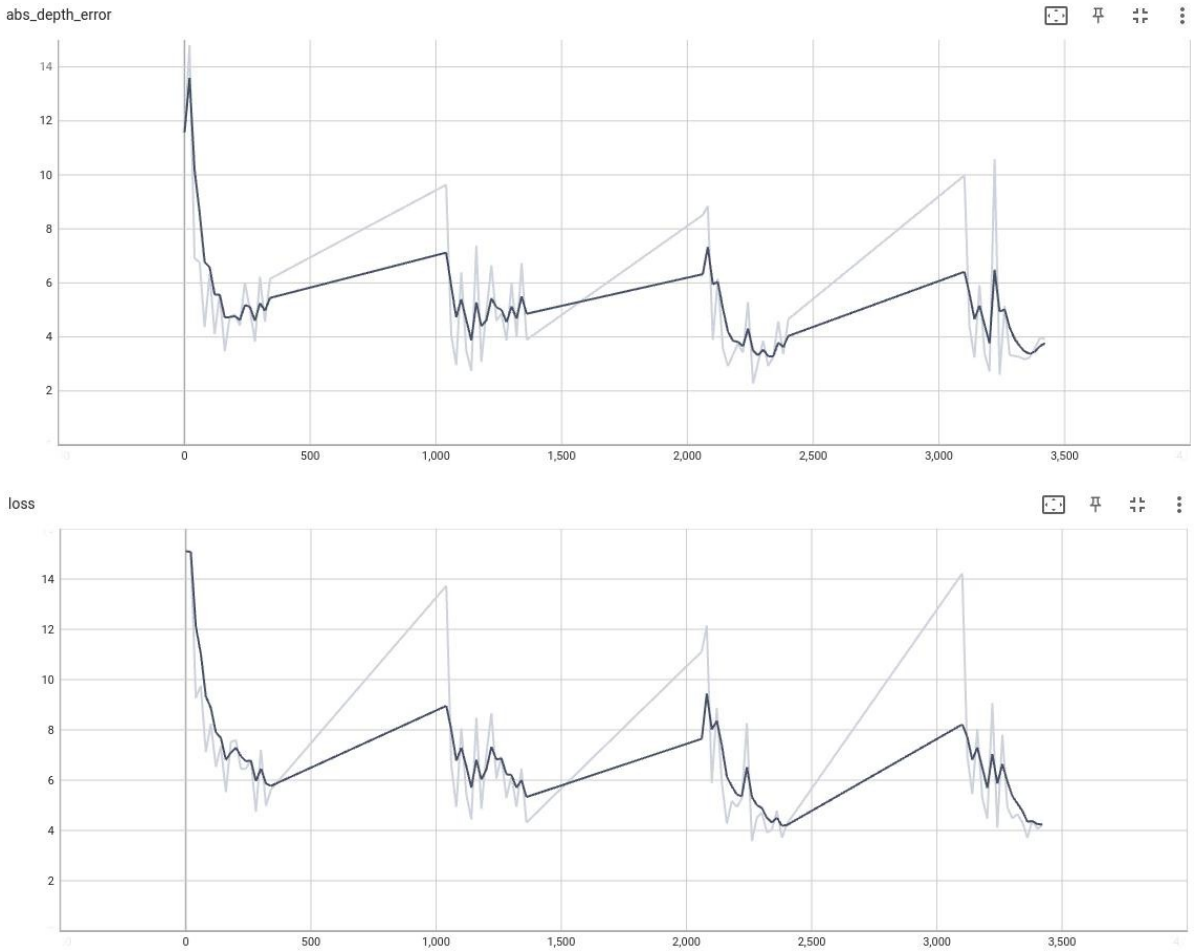

Figure 4: Convergence of the absolute depth error (up) and the loss (down) on the validation dataset over the 4 epochs of training

This shows, that especially in the first epoch, the loss converges very quickly and then flattens down for the higher number of iterations. Also all the further iterations only manage to slightly minimize the loss and the absolute depth error more.

We can also compare the depth estimation with the ground truth for some scan (007), to see that they are very close indeed:

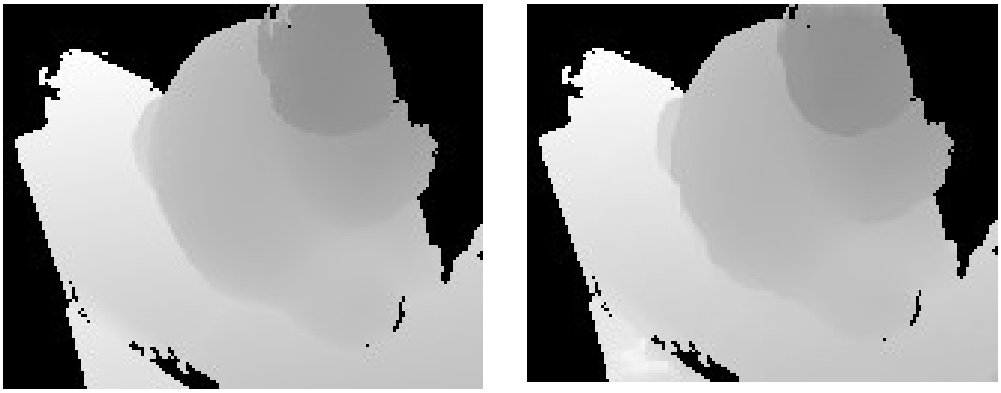

Figure 5: Comparison of depth ground truth (left) and estimation (right) for scan 007

One can therefore conclude, that the training results are very satisfactory and able to reproduce the ground truth pretty accurately, suggesting that the following evaluation and 3D modeling should also go down well.

3.4. Test

1. *Explain what geometric consistency filtering is doing in the report.*

This filtering implemented in method *check_geometric_consistency* of *eval.py* creates a mask to eliminate all points that don't fulfill the following 2 conditions:

a) $\sqrt{((x_{repr} - x_{ref})^2 + (y_{repr} - y_{ref})^2)} < 1$

b) $|d_{repr} - d_{ref}| / d_{ref} < 0.01$

In other words, we define thresholds on how small the reprojection error of the x,y-coordinates has to be and also on the relative depth error. This allows us to get rid of any points that didn't perform well and geometrically make little sense, and therefore could disturb our final results as outliers.

2. *For all the scenes, visualize (visualize_ply.py) and take screenshots of the point clouds in Open3D.*

In the following, I show images of the point clouds obtained after evaluation of the model, for the preset scans 001 & 009:

Figure 6: Colored 3D point cloud after evaluation of the model on scan 001

Figure 7: Colored 3D point cloud after evaluation of the model on scan 009

For all the further scenes, see the results in the annex (chapter 5). All of them seem very accurate, with only very little outlier points. The training and evaluation therefore was successful and had the expected results.

3.5. Questions:

1. *In our method, we sample depth values, $\{d_j\}^D_{j=1}$, that are uniformly distributed in the range [DEPTH MIN, DEPTH MAX]. We can also sample depth values that are uniformly distributed in the* inverse *range [1/DEPTH MAX, 1/DEPTH MIN]. Which do you think is more suitable for large-scale scenes?*

As can be found in various sources ([2], [3]), the *inverse* range is very suitable for large-scale scenes. This is because we can achieve higher numerical stability using the inverses. We get better (sub-pixel) estimates and using the inverses the error in position can be expressed as a Gaussian. This will further make it easier to filter or use other methods on the model, which might require normalization. Still it has to be mentioned that using the inverses also increases the memory needed, which can be a caveat for some setups for training.

2. *In our method, we take the average while integrating the matching similarity from several source views. Do you think it is robust to some challenging situations such as occlusions?*

This will generally depend on the setup. This method will not be very robust if we choose only a small number of source views and also if we almost always choose the same ones. Then it is likely that occlusions or other similar challenges will effect the result greatly, as they will have more weight in the average. On the other hand, we can also try to choose enough source views out of a bigger set, and also in a random manner at every iteration. In that case for example most occlusions that only occur on a single or few source views should not really have an effect on our total model and we have therefore a robust method.

4. Sources

[1] *Distance from a point to a line*, Wikipedia: https://en.wikipedia.org/wiki/Distance_from_a_point_to_a_line, last consulted on 01.12.2021

[2] Qingshan Xu and Wenbing Tao. L*earning inverse depth regression for multi-view stereo with correlation cost volume*. In *AAAI*, 2020.

[3] *Inverse depth parametrization*, Wikipedia: https://en.wikipedia.org/wiki/Inverse_depth_parametrization, last consulted on 02.12.2021

5. Annex

In the following I attach the resulting 3D point clouds of the model when evaluated on all further scans but 001 & 009:

5.1. Scan 002

5.2. Scan 003

5.3. Scan 005

5.4. Scan 006

5.5. Scan 007

5.6. Scan 008

5.7. Scan 014

5.8. Scan 016