Computer Vision Assignment 4

Nicolas Wicki

December 2021

1 Introduction

1.1 Environment Setup

No problems encountered.

1.2 Hand in

2 Model Fitting

2.1 Line Fitting

2.1.1 Least-squares Solution

The implementation is mostly straightforward. However, one needs to remember that we need an additional column besides x for the coefficient matrix to represent b of the equation (using column of ones).

2.1.2 RANSAC

`random.sample` makes our life much easier and easily delivers random indices for our x and y according to `num_subset`. Using the line estimation from the least-squares solution, we can determine outliers considering the whole data set according to the provided threshold. We simply repeat this process a certain number of times (300 in our case) and pick the line giving us the least number of outliers (or the highest number of inliers).

2.1.3 Results

Running `line_fitting.py` delivers the following results:

- Ground Truth: (k, b) = (1, 10)
- Least-squares Estimation: (k, b) = (0.616, 8.962)
- RANSAC: (k, b) = (0.999, 9.997)

A visualization comparing the linear regressor with the RANSAC regressor can be seen in Figure 1.

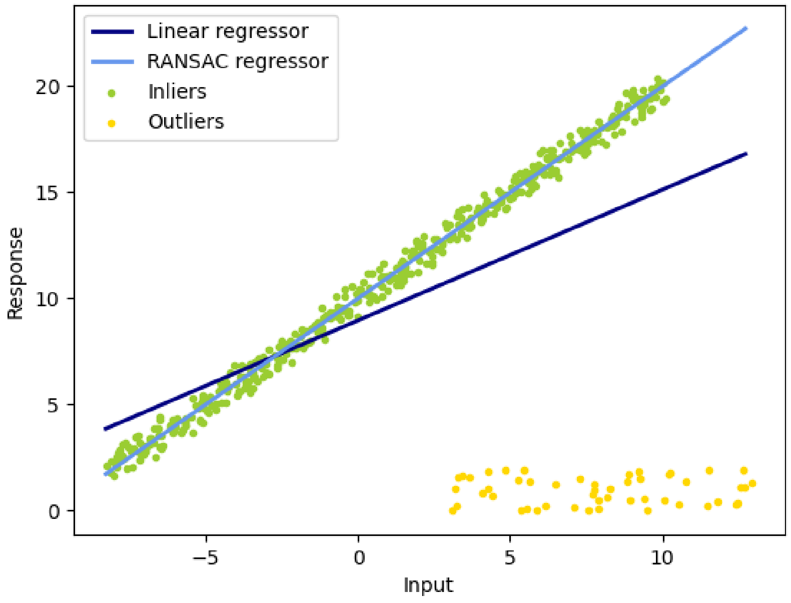

Figure 1: Result from `line_fitting.py`

3 Multi-View Stereo

3.1 Dataset

3.1.1 RGB Images

Intensities are indicated with values from 0 to 255. Dividing by 255 gives values between 0 and 1. To read the image, I used the `open` function provided by the `PIL` library.

3.1.2 Camera Parameters

I extracted the relevant parameters from the camera-parameter file using hard-coded index slices according to the mentioned data format.

3.2 Network

3.2.1 Feature Extraction

Setting up the architecture would be rather tedious as we have quite a lot of layers. However, using list comprehension and a lambda function to adjust to the slightly differing parameters makes it much more readable. To ensure the same image size of the stride-one convolutions, we use a convenient argument, which is simply `'same'` [1]. To halve the image size exactly using the stride-two convolutions, we need to use a padding of two[2].

3.2.2 Differentiable Warping

1. As explained in Wang, Galliani, Vogel, *et al.* [1], the corresponding pixel $p_{i,j}$ of source view i for a pixel p of reference view 0 with depth value d_j is computed as follows: $p_{i,j} = K_i \cdot (R_{0,i} \cdot (K_0^{-1} \cdot p \cdot d_j) + t_{0,i})$ where K_i is the intrinsic matrix of the source view i, K_0 is the intrinsic matrix of the reference view 0, $R_{0,i}$ is the rotation matrix part of the relative transformation from reference view 0 to source view i, and $t_{0,i}$ is the translation also part of the relative transformation from reference view 0 to source view i.

2. To implement the differentiable warping, we use rather tedious tensor multiplications where we need to rearrange the dimensions to allow for proper scaling by the depth values and matrix multiplication by the rotation matrix. We normalize the homogeneous points to get points in the image plane from where we sample the warped source features using the by `PyTorch` provided `grid_sample` function.

[1]For kernel size three, this is the same as a padding of one

[2]This can be computed using the width and height computation formula provided in the PyTorch documentation. The kernel size does also factor into the equation.

3.2.3 Similarity Computation and Regularization

1. To get the dot product, we multiply the reference features with the warped source features, and broadcast the reference features along the depth dimension of the warped source features. Taking the average along the reduced channel dimension gives the desired scaled dot product.

2. To set up the architecture for the SimilarityRegNet we can follow a similar approach to the feature extraction network. Special cases here include the 2D transposed convolution where to get the appropriate image size, we need to follow a different formula to compute the size and adjust padding and output padding provided by the `PyTorch` function `ConvTranspose2D`. [3] To feed the data into the network, we need to adjust its size. The group dimension will be used as channels, so we need to switch its position with the depth dimension, which we do not account for directly during training. This way, we can combine batch and depth dimensions, which gives us the appropriate data format for neural networks (as defined by many `PyTorch` networks).

3.2.4 Depth Regression

The depth regression can easily be computed by scaling the probabilities of each depth with its depth value. Then, we sum along the depth dimension and get the desired result.

3.2.5 Loss Function

We use the `l1_loss` to train our network, which is easily implemented with corresponding function from `PyTorch`. Note, however, that we need to mask each input to account for pixels not being at ground truth depth.

3.2.6 Photometric Confidence

Nothing to be additionally implemented here.

3.3 Training

The convergence of the loss on the training dataset and the validation dataset can be seen in Figure 2 respectively in Figure 3.

3.4 Test

3.5 Questions

1. Photometric consistency works at coarser scales. So, for rather homogeneous areas with low texture details, it matches the task. However, it breaks down at finer scales, where we instead use geometric consistency filtering to

[3] Its documentation provides the corresponding formula.

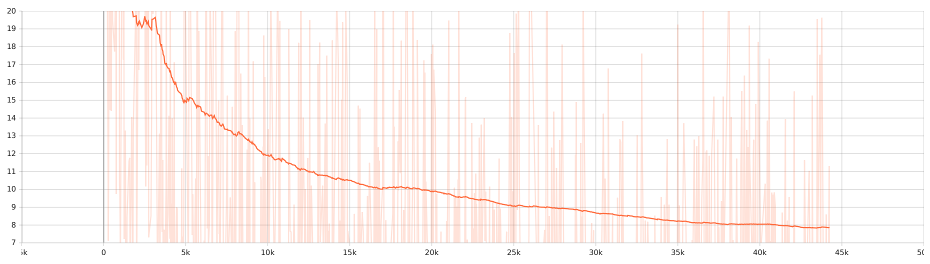

Figure 2: Training loss of the MVS network

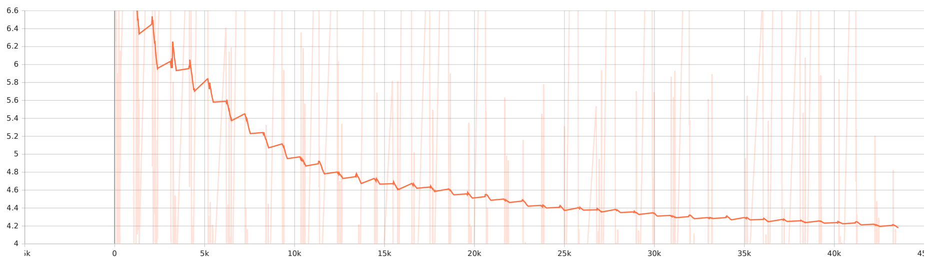

Figure 3: Validation loss of the MVS network

optimize the estimates. To enforce the geometric consistency, we compute a forward-backward reprojection error between reference and source image, which acts as a threshold for unreliable estimates.

2. The results are visible in Figure 4 and Figure 5

3.6 Questions

1. The inverse range. During differentiable warping, we scale the points by the depth. If we get really high numbers for the coordinates, and we basically squared the magnitude with the depth, numerical errors are much more likely. Using the inverse acts as a normalizer and keeps the magnitude in check.

2. I think the average is not fit to handle occlusions. Since we divide by the amount of pixels compared, we might get a much lower similarity score than expected. Detecting those outliers (the occluding pixels) and excluding them from the similarity score might increase robustness.

References

[1] F. Wang, S. Galliani, C. Vogel, P. Speciale, and M. Pollefeys, “Patchmatchnet: Learned multi-view patchmatch stereo,” in *Proceedings of the*

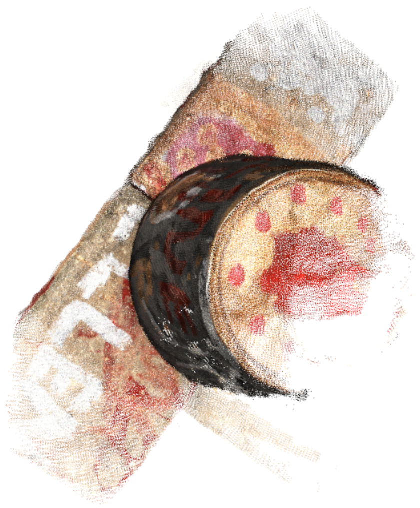
Figure 4: Result of scan 1

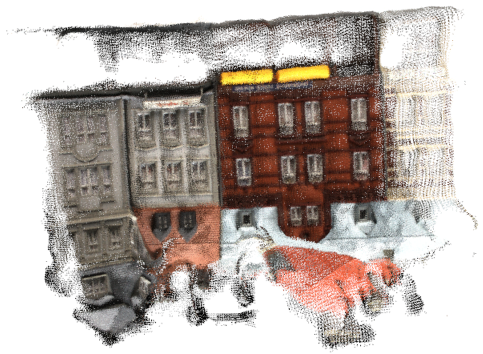
Figure 5: Result of scan 9

IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2021, pp. 14 194–14 203.