Estimating a dense depth map from a sparse depth map using a multi layer perceptron and object segmentation.

1. Introduction

The rising number of autonomous machines operating in real-world environments are depended on sensors that can create reliable maps of the world. Autonomous machines, like self-driving cars, often use a lot of different sensors to achieve this goal. The processed data from each of these sensors should not only be reliable on their own, but also be complementary with the other sensors, meaning each of them have distinct advantages and weaknesses.

One of the most used sensors for world interpretation today are stereo cameras, and the dense depth maps created from these images. Traditional methods for creating these dense depth maps present several challenges. Standard pixel matching methods with small descriptor patches tend to produce salt and pepper noise, while larger descriptor areas or other low-pass filtering methods remove interesting depth data.

This project is intended as a proof of concept for a new algorithm using image segmentation, sparse stereo maps and a Multi Layer Perceptron (MLP) to create dense depth maps. In this project we have prioritized minimizing the average depth error. Therefore we have done little to no optimization regarding the speed. Instead we have focused on writing modular code which simplifies experimentation with parameters. All of our code is written in python with heavy use of Opencv and Tensorflow.

2. Algorithm

The first phase of our algorithm consists of segmentation and creating a sparse depth map. The segmentation detects distinct objects in the image using classical segmentation algorithms. These objects are then encoded using a «one-hot» encoding scheme. The sparse depth map is created using classical feature matching and stereo geometry algorithms..

Finally, the algorithm uses an MLP with the pixel coordinates and the «one-hot» segmentation encoding as input. The sparse stereo map is used as training data for the MLP. After training, each pixel of the original image is run through the MLP to create the final dense depth map.

The MLPs main strength is in describing smooth functions. A dense depth map can be viewed as a combination of piecewise smooth functions, where each separate object is relatively smooth. We believe that increasing the input dimensionality, by adding the object segmentation, will provide the MLP with the necessary information to also describe the discontinuities between objects.

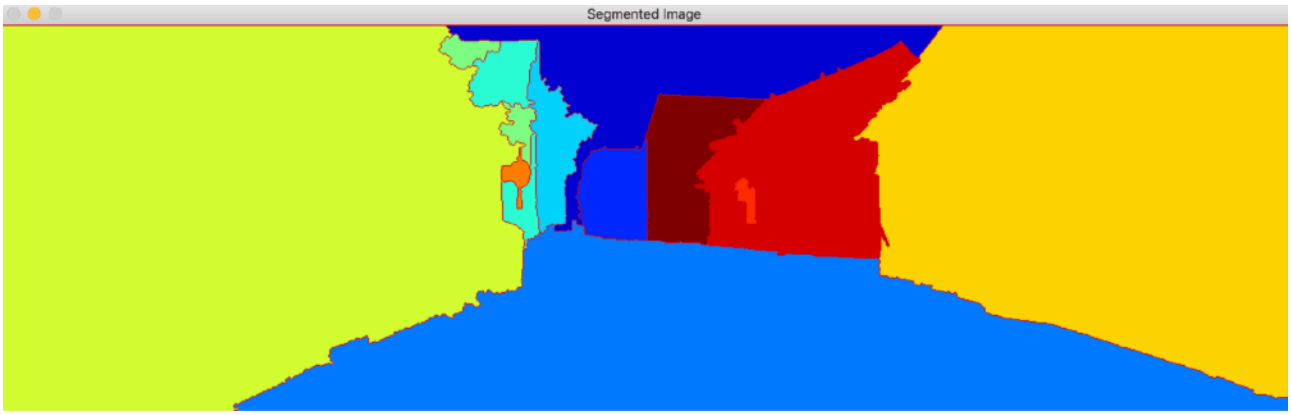
2.1 Segmentation

We want to segment an image into coherent objects. Thus we have chosen to use opencv's version of watershed combined with manual selection of marker points through a GUI setup. Manual marker selection is chosen for two reasons. Segmentation is only a small part of the curriculum in the course UNIK4690, so we didn't want to spend to much time on implementing automatic segmentation. Secondly, it opens up for experimentation of how well an image is segmented, and how this affects the MLP.

The Watershed algorithm works by first selecting one or more initial markers for each object. The next step consists of calculating the gradient of the image. The low gradient pixels in the neighbourhood of each marker are then «filled» to belong to the corresponding object. This process continues with higher and higher gradients until different objects meet and form a «barrier». These barriers are then considered as borders between objects.

The segmentation is done using opencv functionality and the code can be viewed in the Watershed.py module. The marker selection is done using the module PixelSelectGUI.py.

Watershed marker selection using our GUI



Segmented Image

2.2 Sparse depth map using stereo geometry

To detect feature points and calculate descriptors we decided to use opencv's implementation of the SIFT algorithm, mainly because SIFT is a robust method resulting in many good feature points. It's drawbacks are that it can be slow relative to other methods, such as SURF or binary descriptors. Since this is a proof of concept we prioritize having many good matches rather than speed. We run the SIFT algorithm with a lower contrast threshold (0.003) than usually suggested. By reducing the contrast threshold we reduce the quality of the feature points, but because of stereo geometry we can filter out bad matches instead of bad feature points. We kept the edge threshold at the suggested 20.

For feature matching we started out using opencv's BFMatcher, a brute force algorithm. The advantage of the BFMatcher is that it will always find the best matches, since it checks every combination. After some initial testing we decided that the bf algorithm wasn't optimal for our purpose, since it doesn't take advantage of the stereo Geometry. Therefor we wrote our own version of the BFMatcher. Our algorithm consists of three phases:

1. Initial matching phase:

The purpose of this stage is to create a huge set of matches of varying quality. For each keypoint the algorithm scans along the epipolar line within a relatively high tolerance, for matches. The distances are calculated using the Euclidean distance as suggested by Lowe[3]. If matches are found, with distance smaller than a threshold, the matches are saved. Only scanning along the epipolar lines will not only increase the quality of the matches, but also massively improve the speed of the algorithm. For this stage we used an epipolar tolerance of 5 pixels and a distance threshold of 200.

2. Local threshold phase:

The purpose of this stage is to keep matches in areas of the image with few matches and remove matches in areas of the image with many matches. Matches below a distance threshold, are always kept. Matches above the distance threshold are only kept if they are among the best matches in a neighborhood. We used a 15x15 neighborhood and kept the matches if they were among the 4 best matches in this neighborhood. The distance threshold for always keeping matches is set to 100.

3. Refining phase:

The purpose of this stage is to remove erroneous matches. First matches with negative disparity are excluded. Then the ratio test is performed. Afterwards we perform a new epipolar test with lower tolerance. The ratio test threshold is set to 0.8 and the epipolar tolerance is set to 0.7 in this stage.

Now we take advantage of the angle and size of the keypoints. With stereo cameras mounted close to each other these two parameters should be approximately equal in the left and right image. Although this isn't necessarily true for points in the image very close to the camera, these assumptions drastically improved the quality of the matches. The angle difference threshold is set to 10 degrees and the size difference threshold is 1.

Although most of this phase could be directly implemented in phase 1, we achieved better results implementing them as a separate phase. The reason for this is mainly the ratio test. If phase 1, especially the epipolar line test, is to strict, most of the matches won't get a second best match to use for ratio testing. We considered removing the ratio test but achieved better results by keeping it.

After matching we use the cameras baseline, focal length and basic Stereo geometry to calculate the sparse depth map.The sparse depth map is implemented using opencv functionality and the code can be viewed in the Matcher.py and StereoDepth.py modules.

2.3 MLP

This project uses a fully connected neural network (Multi-Layer Perceptron or MLP) to estimate a dense depth map from a sparse depth map. The input data is formatted as **x=*[x,y,s1,s2,s3…,sn]*** where x, y are the pixel coordinates and **s1,s2,…,sn** is a «one-hot» encoding of the corresponding segment.

The MLP uses the sparse stereo map from the StereoMatcher module as training data. Before training we remove depth points from the sparse set that are in a 3x3 neighborhood of segment edges, since these points might be misplaced in the wrong segment.

The MLP uses a stochastic k-fold validation scheme where 20% of the training data is randomly selected and used as a validation set between each epoch. The MLP uses the Adam Optimizer with 4 step-lengths from 0.1 to 0.0001. If 3 epochs in a row do not show improved results the next step length is selected. After all step lengths are used early stopping will engage. Each epoch consist of 75 training iterations.

After some testing we decided to use the absolute error as the loss function for the MLP. This loss function seemed slightly better in not getting stuck in local minima, but the mean squared error function had similar results.

To get the best result from using the MLP, the input and the target data has to be normalized before training. The input data is normalized by dividing x coordinates by the image height and y coordinates by the image width. The target data is normalized by **Tnorm = (T - Tmin*0.8) / (TMax*1.2 - Tmin*0.8)**

Multiplication of Tmin by 0.8 and Tmax by 1.2 is done so that the whole range (0,1) is not used by the training data. The reason for this is that the MLP output will always be a number between (0,1) since we use the sigmoid activation function on the output. If we don't multiply Tmin and Tmax the MLP can never achieve depths larger than Tmax or smaller than Tmin.

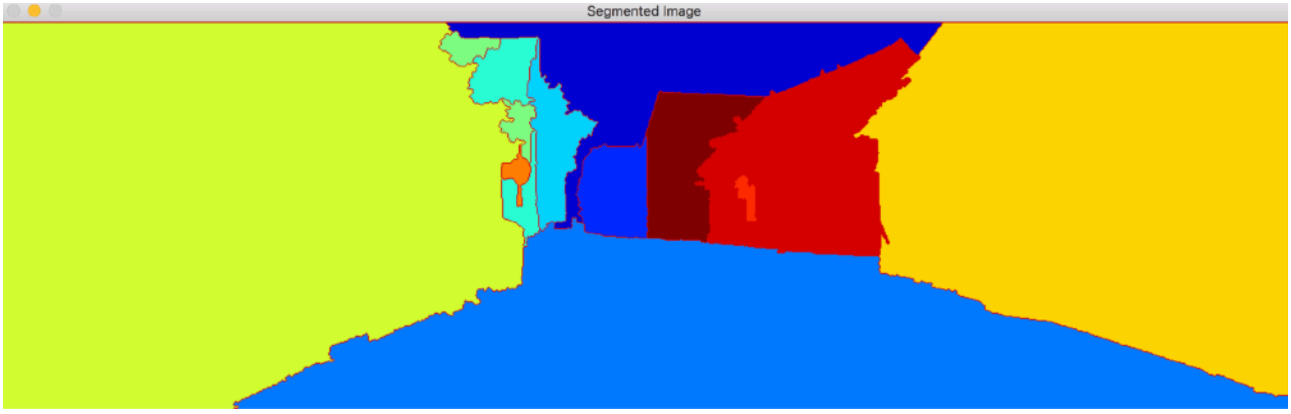
The depth of segment edges are not currently estimated by the MLP, since the Watershed algorithm does not assign edge pixels to any segment. Instead we interpolate these depths using the minimum in a small neighborhood testing on the middlebury image as shown below. This gave relatively good results indicating that segment edges usually appear on foreground objects, not background.

The MLP is implemented using Tensorflow and all the code can be found in the MLP.py module. MLPDataProsessor.py is a module used to process and format input and output data used by MLP.py. Lastly, the MLPDataInterpreter.py is used for statistic runs and data visualization.

3. Setting MLP parameters/ experiments

We tested several different hidden layer topologies and activation functions by running each MLP configuration 30 times. Before each run we removed 10% of the data randomly and used this to create a test set. All runs used the same sparse depth map and segmentation, as shown in the pictures below:

Sparse depth map used for statistics run



The segments used for statistics run

After each run the average distance difference of the calculated test-set points and the real depths as calculated by the stereo geometry algorithm were stored. We measured 5 different statistics:

1. Average distance error range(ADER):
   The range of the average distance errors in each run

2. Average distance error(ADE):
   Average of the average distance error in each run

3. Standard deviation range(SDR)
   The range of the standard deviation in each run

4. Average standard deviation(ASD):
   The average of the standard deviation in each run

5. Average distance errors standard deviation(ADESD):
   The standard deviation of the average errors over all runs

The first two values ADER and ADE measure average errors of each run. The next two values, SDR and ASD measure the standard deviation of the errors of each run. The last value ADESD measures the standard deviation of the average error over all runs. If consistently good results over several runs is important, this is an important factor.

We started by testing several different hidden layers topologies while using the sigmoid function as activation function in all layers. Several different topologies achieved good results, but we decided to continue with the [15,15] topology since it both had good average error and good standard deviation across all runs, meaning it consistently had good results. Also visual inspection revealed that the less complicated networks did better in areas with little training and test data.

After testing different hidden layer topologies we tested different combinations of sigmoid and Relu activation functions. The pure sigmoid configuration had better results.

All measurements in the below table are made with only sigmoid activation functions.

Layers:	ADER	ADE	SDR	ASD	ADESD
[20,10]	0.2708, 0.6035	0.4562	0.4853, 1.3170	0.9039	0.0787
[40,20]	0.2528, 0.6480	0.4449	0.4973, 1.3045	0.8486	0.1087
[15,15]	0.2806, 0.5790	0.4362	0.4904, 1.1956	0.8512	0.0756
[30,30]	0.2599, 12.3431	1.2958	0.4745, 13.0047	1.7952	2.6093
[10,20]	0.2811, 0.7447	0.5028	0.6182, 2.2433	1.0789	0.1087
[20,40]	0.2677, 56.7298	7.9375	0.5570, 14.5297	6.2609	12.5004
[5,10,5]	0.3201, 0.9649	0.5205	0.5088, 1.6068	0.9645	0.1470
[10,15,10]	0.2695, 0.5508	0.3782	0.4236, 1.6244	0.7555	0.0664
[15,20,15]	0.2235, 0.5536	0.3867	0.4288, 1.3305	0.8381	0.0850
[20,25,20]	0.2532, 0.6858	0.3774	0.4318, 1.4005	0.8511	0.0954
[10,5,5]	0.2865, 0.8924	0.5216	0.4597, 1.6045	1.0322	0.1387
[15,10,10]	0.2628, 0.5591	0.4057	0.4904, 1.4499	0.8608	0.0803
[5,5,10]	0.3536, 1.0730	0.5650	0.5379, 1.7586	1.1284	0.1651
[10,10,15]	0.2269, 0.6614	0.4315	0.4482, 1.9043	1.0537	0.1031
[10,10,10]	0.2688, 0.5589	0.4362	0.5237, 1.6750	0.9823	0.0795
[15,15,15]	0.2478, 0.5397	0.3771	0.4598, 1.2256	0.8452	0.0700

All measurements in the below table are made with the [15,15] configuration

Layers	ADER	ADE	SDR	ASD	ADESD
[relu, sigmoid, sigmoid]	0.3178, 0.7361	0.4576	0.5929, 1.5051	0.9835	0.1097
[sigmoid, relu, sigmoid]	0.4098, 57.2231	8.6290	0.6543, 14.0138	6.9398	12.4954

Layers	ADER	ADE	SDR	ASD	ADESD
[sigmoid, sigmoid, relu]	9.4935, 13.5008	11.9285	9.7202, 15.0250	13.0192	1.1883
[relu, relu, sigmoid]	0.4874, 13.1777	6.0441	1.0594, 15.9017	7.0840	5.2498

4. Results

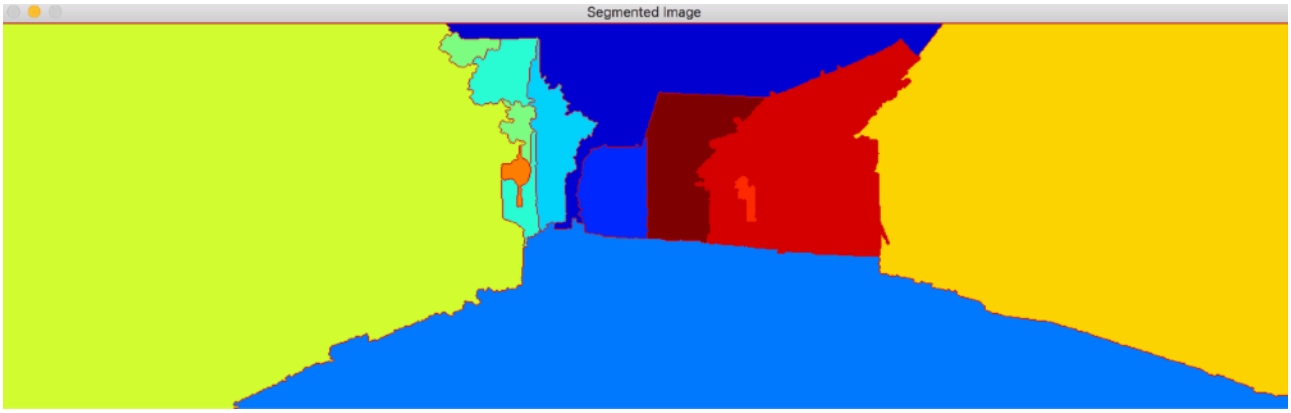
After tuning the MLP we tested it with 3 different images, two of the images are from the Kitty residential set and the last one is from the Middlebury stereo dataset. The Kitty images used don't have ground truth so we evaluated these by extracting a test set from the sparse map before training and considering this as ground truth. The results from these images are of course not comparable to images with a dense ground truth set, but will give a good indication of the performance. We have used depth zero for the sky in all images. The resulting dense depth map image is shown with a logarithmic depth map. The numbers in the dense depth map image are the difference between the actual test set depths and the MLP depths in meters. The numbers on the color bar is the estimated depth in meters with corresponding color indicated by the dot before the number.

4.1 First Kitty image:

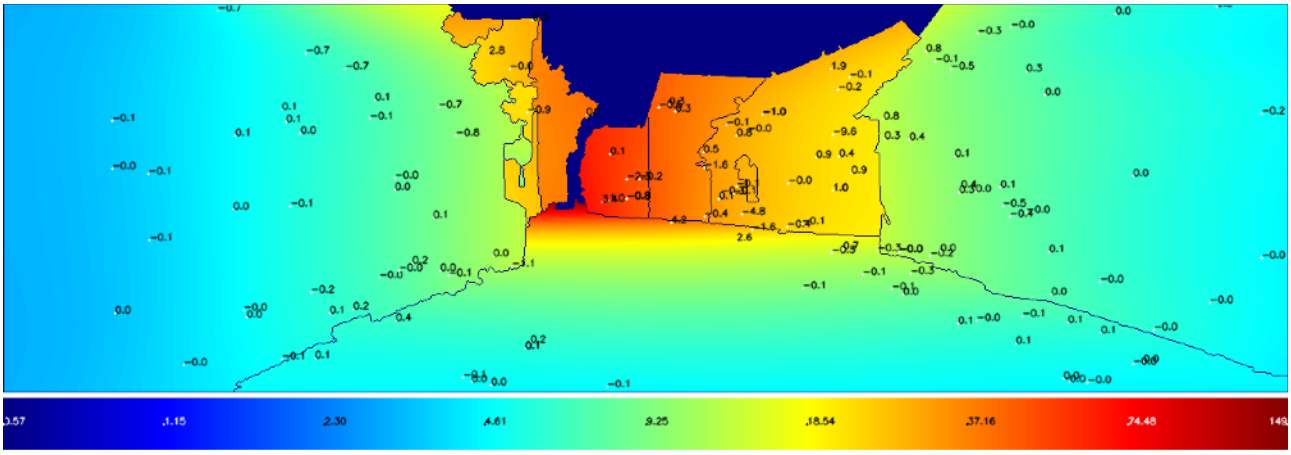
This image is considered a best case scenario for our algorithm. The image contains a lot of planar surfaces and is easy to segment.

The original image from the left camera

The sparse depth map used for training



The segmentation used



Resulting depth map (logaritmic colormap), average test set error = 0.480 meters

We see that the errors are generally low, especially close to the camera. One challenge is the stop sign seen in the middle of the image. The sign should be closer than the house behind it, this isn't indicated in the depth map. The reason for this is that the sign segment also contains part of the window of the house behind it and uses the depths from this window.

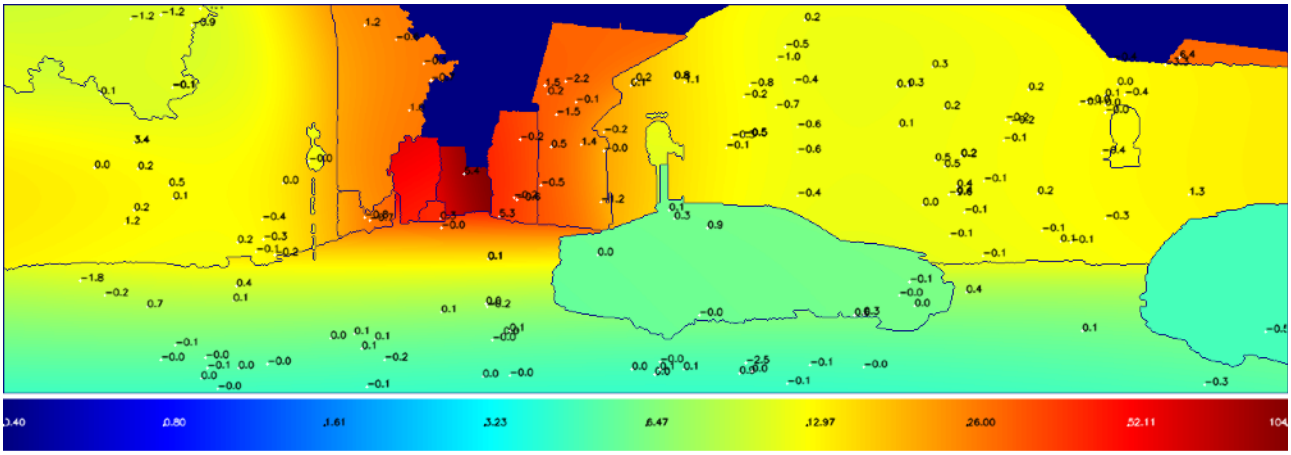
4.2 Second Kitty Image

This image should be somewhat harder than the first one. Our segmentation isn't as good as the for the first image and we use more segments (18 in this one vs 12 in the first one).

The original image from the left camera

The sparse depth map used for training

The segmentation used



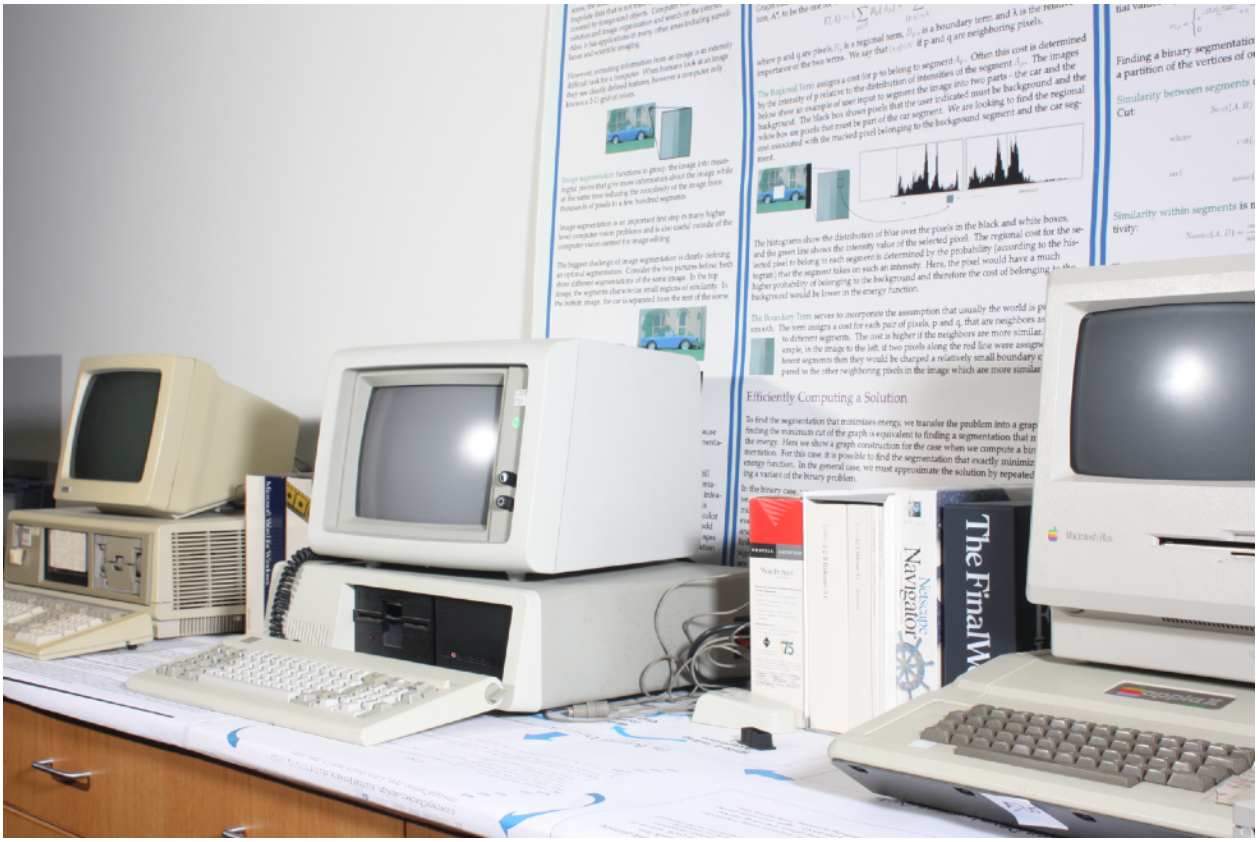
Resulting depth map (logaritmic colormap), average test set error = 0.569 meters

The results are still quite good, although the average error is somewhat higher. The tree in the top left corner is especially problematic. The segmentation isn't perfect and points that should be on the tree are on the wall instead. This affects the depth of the wall.

4.3 The Middlebury image

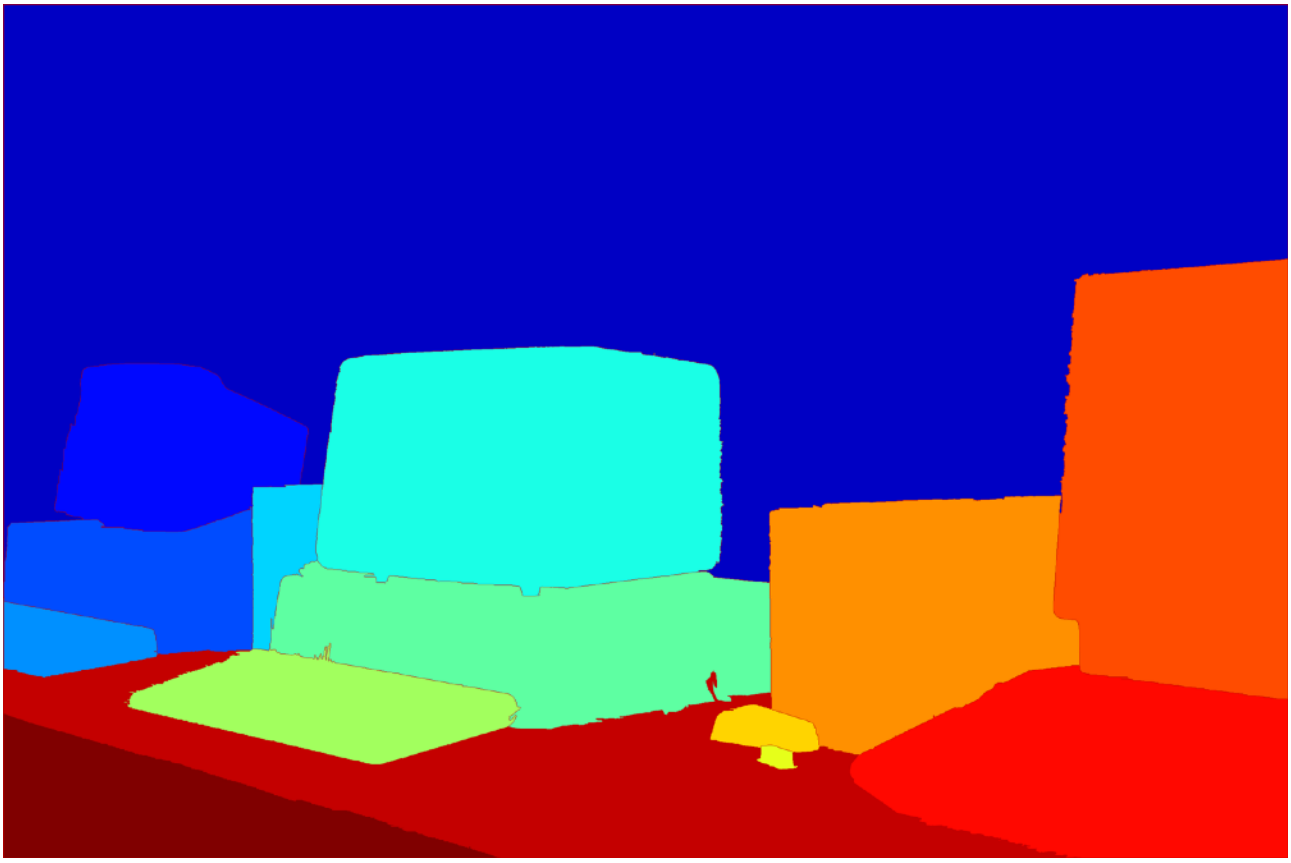
This image will be very challenging for our algorithm. It contains a lot of details and objects are composed of several planar surfaces. The area in the top left has little texture and we will get few feature points here. Since this image is almost four times the size of the last images we adjusted some parameters. The neighborhood for local thresholding in the matching algorithm is increased to from 15x15 to 30x30 and a match is kept if its among the 16 best in this neighborhood. Points in the sparse set are now removed if they are within 4 pixels of an segment edge. We also

adjusted the threshold in the matchers local threshold phase up to 300 to get more matches in difficult areas

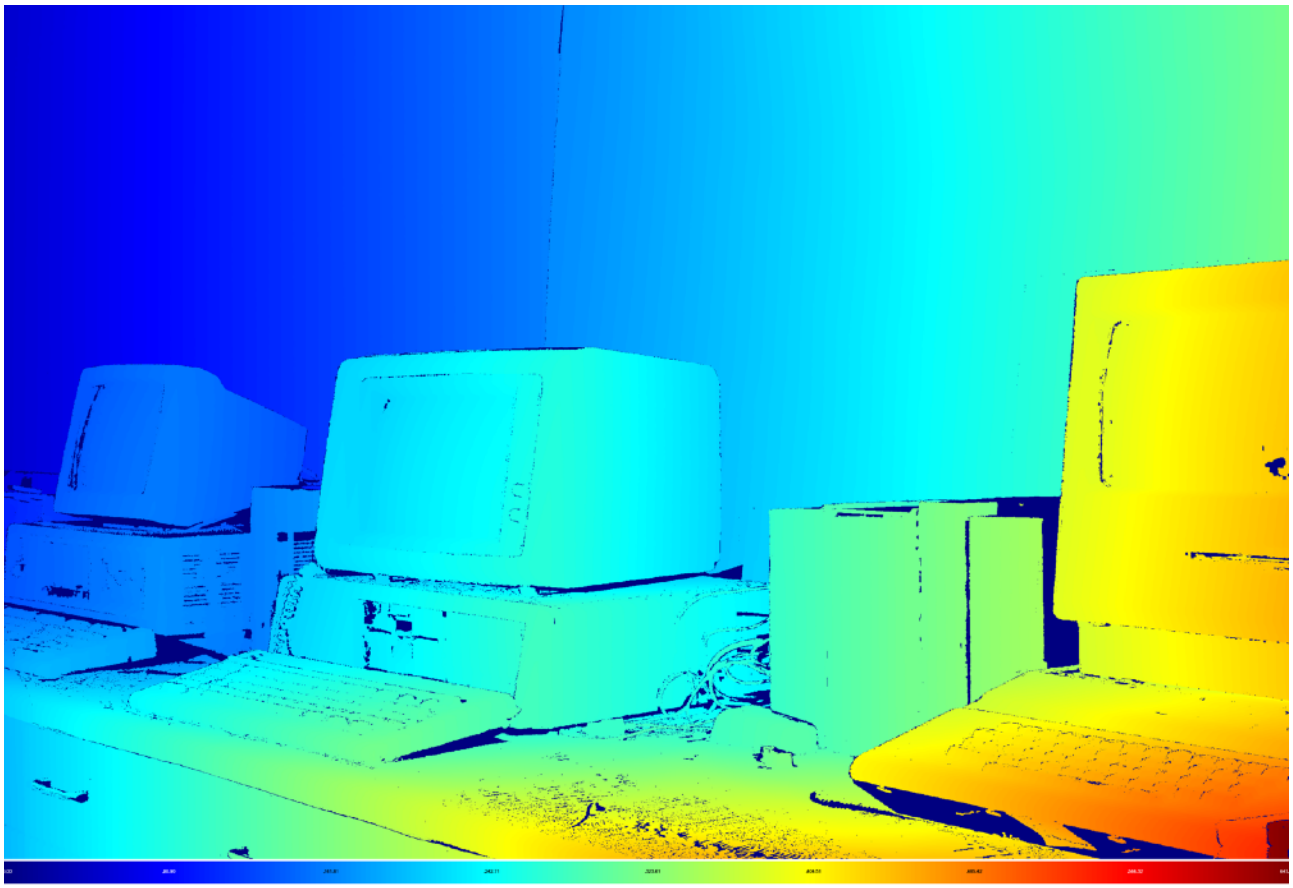


The original image from the left camera

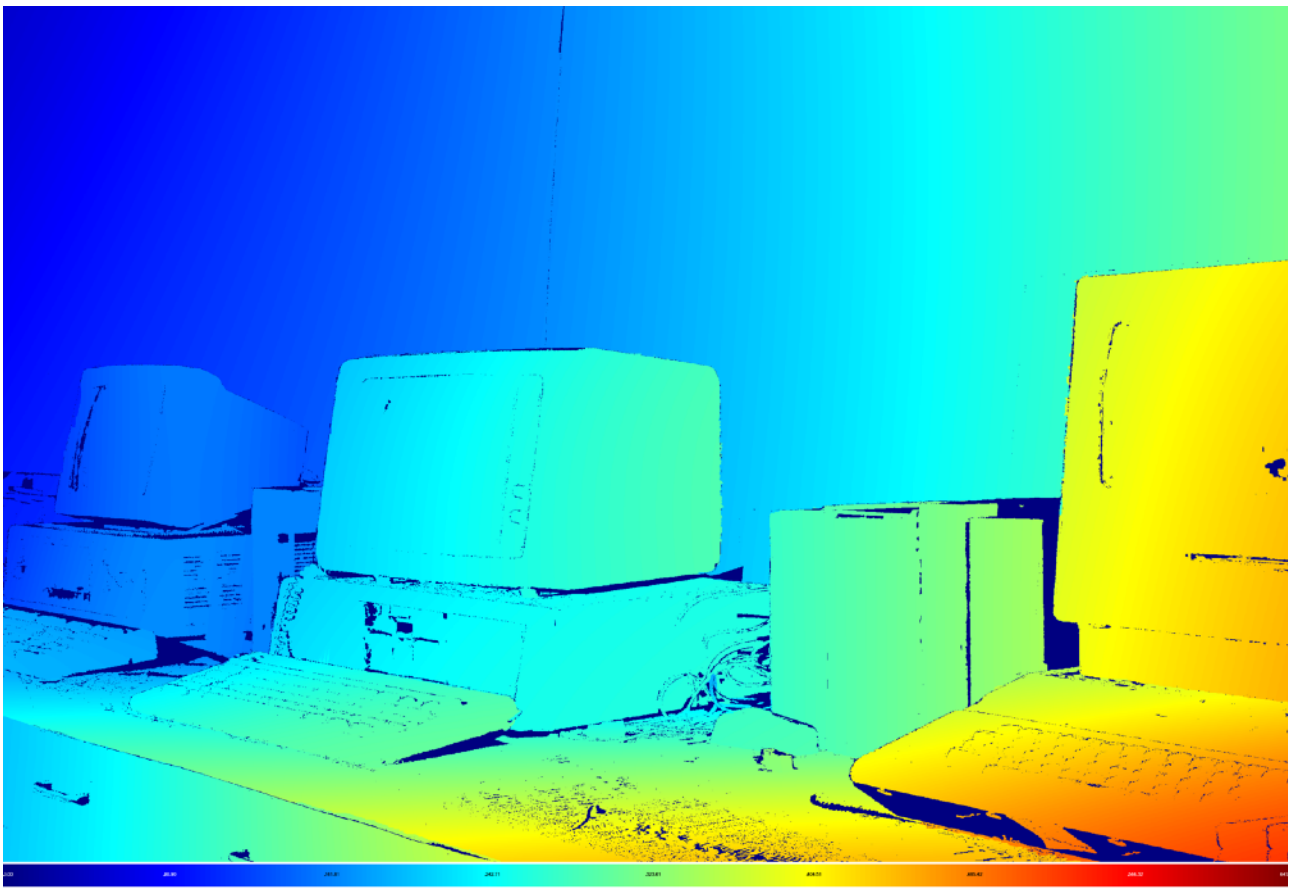
The sparse depth map used for training



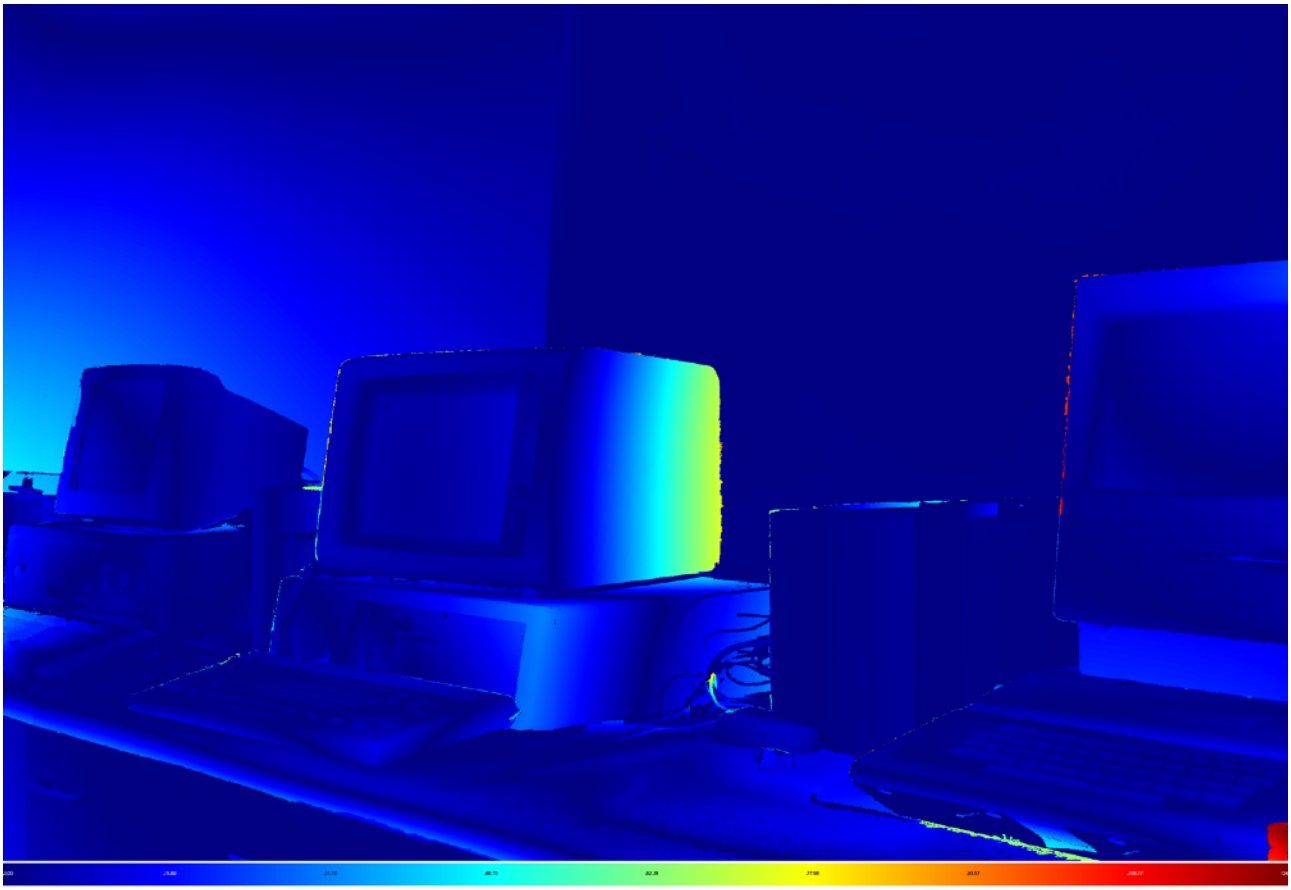
The segmentation used



Ground truth disparities



Calculated disparities



The disparity difference

The result image measures disparities instead of depth, since the ground truth is given in disparities. We clearly see that this image is more challenging for our algorithm. The result on the top left wall and the side of the middle computer screen are especially problematic.

The Middlebury datasets contains a ranking based on several statistics, and we have calculated some of these metrics for our image as well. These results aren't directly comparable since we haven't used the Middlebury software, our segmentation is partially manual, and we only tested one image, but it should serve as a good indicator. The metrics are:

Avgerr:
The average disparity error.

Bad0.5, bad1, bad2, bad4:
The amount of pixels with more than respectively 0.5, 1, 2 and 4 disparity difference from the ground truth.

All of our runs are compared with the unmasked vintage image in the dense training set.

Results for this run:

Avgerr: 6.04 (Rank 15 out of 80)

bad0.5: 73.98% (Rank 46 out of 80)
bad1 : 60.17% (Rank 48 out of 80)
bad2 : 50.42% (Rank 58 out of 80)
bad4 : 37.70% (Rank 61 out of 80)

As we see our algorithm does much better on the average rating than the bad pixels ratings. This is to be expect since MLP is reducing the average error, and not trying to match each single pixel.

General result:

The average error was usually from 6 to 13 pixels, which correspond to rankings between 15 and 43 out of 80 on the official Middlebury ranking[1]. Outliers have been observed from 4.7(Rank 10) to the high 14's (Rank 50). The high variation between different runs seemed to mostly depend on how well the featureless wall in the top left corner was estimated.

5. Augmented reality (AR):

We have implemented an example program for use of the dense depth map, namely AR. This program uses our dense depth map estimation of a scene, and optionally dense depth map estimation of an object. It is then presented in a GUI such that one can play around with putting the object in the scene at different depths in the image.

The program calculates a dense depth map of the scene image using our algorithm. If stereo images are available for the object image, a depth map can be calculated also for the object, else the object is given «flat» depth. Depth, scale, rotation and position for the object is then entered through the GUI and the object is placed in the scene.

6. Possibilities not addressed in this project

Our algorithm does not achieve acceptable results in parts of the scene where the sparse depth map has very few data-points. This challenge could be addressed by creating a confidence metric based on the number of data-points in the neighborhood of each pixel and using alternative algorithms in areas with low confidence.

---

[1] http://vision.middlebury.edu/stereo/eval3/

This project hasn't really taken advantage of the stereo geometry in feature detection and description either. Since we remove matches not along the same epipolar line, horizontal changes should weigh more than vertical changes during feature extraction. We believe that a specialized algorithm could improve the estimated depth map further.

Our project hasn't focused on achieving fully automated segmentation. We believe that a Watershed variant without manual marker selection could be viable, but other segmentation methods should also be tested.

Although the [15, 15] MLP topology was quite robust to changes in number of segments, number of features and image size, it could probably be improved to some degree depending on these variables.

7. Conclusion

This project shows that combining segmentation, stereo geometry and the multilayer perceptron can be used to create a dense depth map. Although a lot of work remains in optimizing the speed and the accuracy of the algorithm, this prototype has already achieved very good results in typical city scenes. Lastly, our algorithm does not exhibit the salt and pepper noise often seen in dense depth maps, but instead creates a relatively smooth depth map within each segment.

8. The code:

All of our modules except PixelSelect.py have a if __name__=='__main__' method with an example run. PixelSelect is used by the Watershed module and not meant to be run directly.

The most relevant example runs are in the MLPDataProsessor.py which does a full run of the algorithm and the ARGUI.py which starts the AR program. More information about the code can be found in the readme.md or in each of the individual file documentations.

8. Citations

[1]. D. Scharstein, H. Hirschmüller, Y. Kitajima, G. Krathwohl, N. Nesic, X. Wang, and P. Westling.
High-resolution stereo datasets with subpixel-accurate ground truth.
In German Conference on Pattern Recognition (GCPR 2014), Münster, Germany, September 2014.

[2]. Andreas Geiger, Philip Lenz, Christoph Stiller and Raquel Urtasun.
Vission meets robotics: The KITTI Dataset. International Journal of Robotics Research. 2013

[3] David G. Lowe, Distinctive Image Features from Scale-Invariant Keypoints.
International Journal of Computer Vision, 2004.