Machine Learning Engineer Nanodegree
Capstone Project Report

May 11th, 2018
Christoph Soehnel

Dense Disparity Estimation from Monocular Image Cues

I. Definition

Project Overview

In Computer Vision not only the pixel content, i. e. the visual appearance, of an image is important but also the meaning of a pixel with respect to a specified reference frame. This allows us to measure distances like depths in the actual world what is useful for many applications from computer-aided inspection to robotics and self-driving cars.

While there are others, depth estimation in the pure image domain can be categorized in two main paradigms:

1. Stereo Vision
2. Structure from Motion

Since the first approach requires at least two cameras that record time-synchronized images, it is somewhat costly to build what is an issue for industries that want to offer computer vision solutions in a low-cost high-volume market. The second approach only utilizes one (moving) camera but has drawbacks with stationary scenes and moving objects within the scenes what makes it less robust compared to Stereo Vision. Both approaches have in common, that depth information is derived from different locations a scene object occupies within multiple images.

While technically more challenging, there has also been research to the problem of monocular depth estimation, especially recently with the raise of Deep Learning, e. g. Godard et al.[1] These approaches only need one image to infer the depth of a scene during run-time, but they eventually need multiple images of the same scene during training.

Problem Statement

Within this project, I focused on the problem of dense depth estimation using only monocular image cues. As depth is a function of disparity, i. e. the translation between corresponding pixels in two images, the actual task was dense disparity estimation without any second frame, neither from a second camera nor from the same moving camera.

Disparity however, is only defined for at least two images. Therefor I tried to estimate the disparities with respect to a virtual second image with known displacement to the first image. Since an estimate for every single pixel of the input image and not only for a subset of them was generated, the result is called dense, not sparse.

[1] http://visual.cs.ucl.ac.uk/pubs/monoDepth/

I used Machine learning to solve the problem, incorporating a data-set, where each sample consists of a pair of stereo images and a matching ground-truth disparity map. By using only one of the stereo images as input and the ground-truth disparity map as the target during training, the second stereo image became the virtual image mentioned above.

Metrics

To be consistent with Mayer et al.[2], I used the endpoint error (EPE)[3] as a metric to quantify the performance of my solution and to compare it against the benchmark model:

$$E_{EPE_{disp}} = \frac{1}{n} \sum_{i \in I} |d_e^i - d_g^i|$$

This is the mean of the absolute errors over all pixels in the image, comparing the estimated disparity value of a pixel against its ground truth disparity value.

II. Analysis

Data Exploration

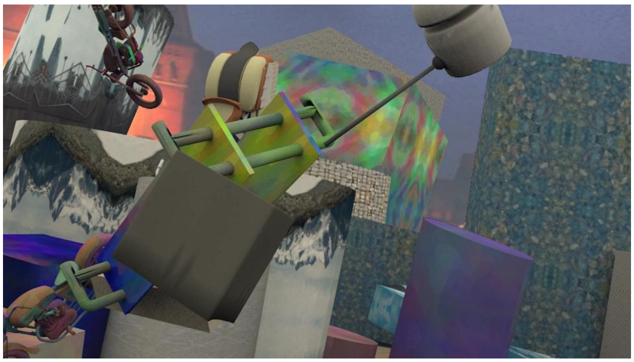
There are several data-sets for disparity estimation available. Most of them lack either dense ground truth data, because the disparity reference is generated through other imperfect algorithms[4], or contain not enough samples[5] for most Deep Learning approaches. Mayer et al. introduced a new data-set to overcome these issues.

Throughout this project the data-set FlyingThings3D was used. It consists of 22390 training images and 4370 testing images. Each image has a spatial resolution of 960 x 540 px, an 8-Bit RGBA color-coding and comes with a corresponding dense ground truth disparity map of the same resolution and 4-Byte float values. Both, the training/testing[6] images and the disparity[7] maps are available on the web[8]. Please note that the cleanpass version of the data-set will be used, which avoids motion and defocus blur in the generated images and is also the variant Mayer et al. picked for comparisons. All images were used in their webp format which gave a good compromise between quality and compression. Regarding disparity, the left disparity maps, i. e. containing the translations from the left to the right images, were used in their pfm[9] format.

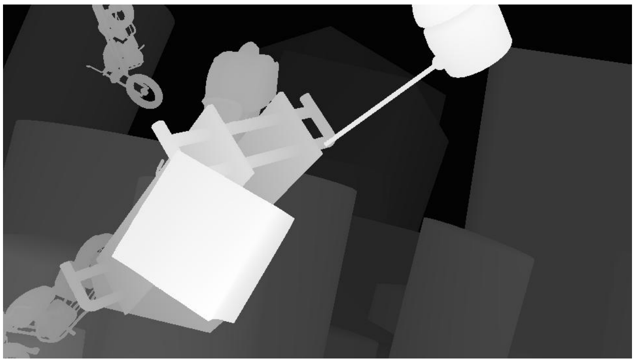
Exploratory Visualization

As the following sample from the test-set shows, the synthetic images of the data-set can contain highly structured scenes, with different textures, occlusions and lighting conditions. The ground-truth disparities are a perfect representation of the scenes depth structure.

[2] https://arxiv.org/pdf/1512.02134.pdf
[3] https://arxiv.org/pdf/1612.02590.pdf
[4] https://www.cityscapes-dataset.com/
[5] http://sintel.is.tue.mpg.de/
[6] https://lmb.informatik.uni-freiburg.de/data/SceneFlowDatasets_CVPR16/Release_april16/data/FlyingThings3D/raw_data/flyingthings3d__frames_cleanpass_webp.tar
[7] https://lmb.informatik.uni-freiburg.de/data/SceneFlowDatasets_CVPR16/Release_april16/data/FlyingThings3D/derived_data/flyingthings3d__disparity.tar.bz2
[8] https://lmb.informatik.uni-freiburg.de/resources/datasets/SceneFlowDatasets.en.html
[9] https://lmb.informatik.uni-freiburg.de/resources/datasets/SceneFlow/assets/code/python_pfm.py



Sample image from the test-set

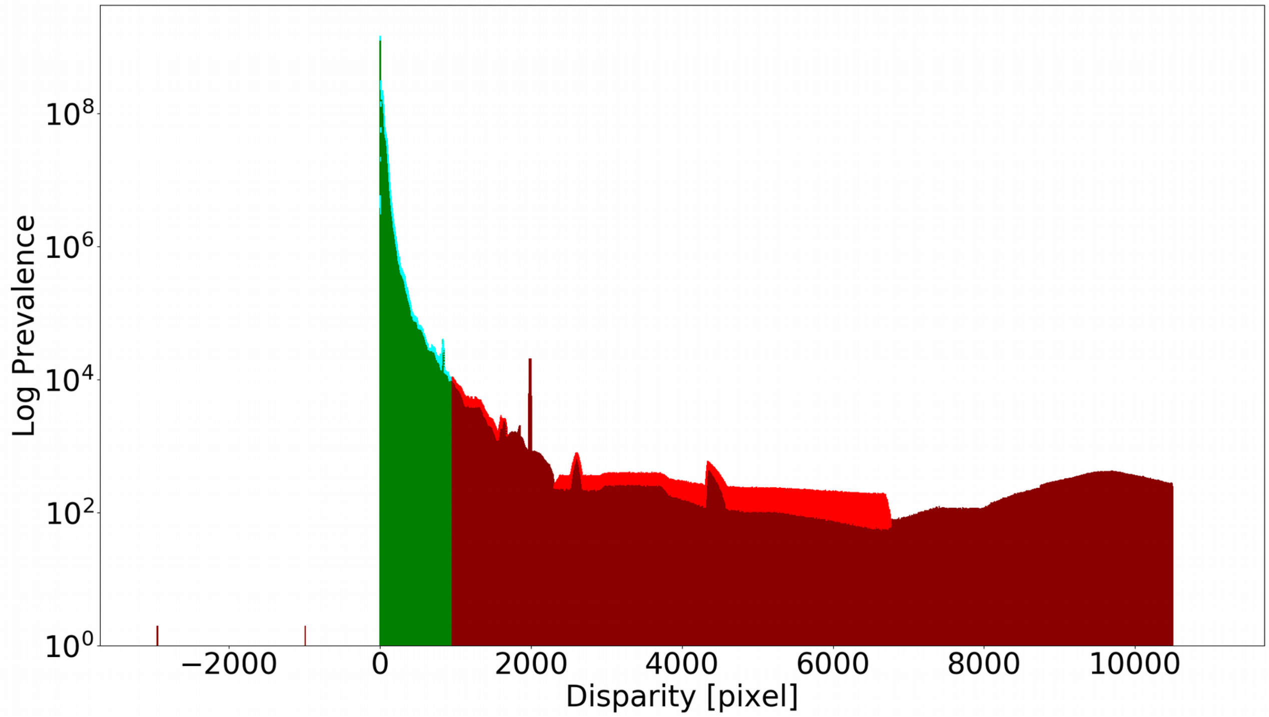


Corresponding ground truth disparity map

Depending on the scene, the disparities may vary significantly, meaning that it is hard to expect or assume any special distribution within the data. Nevertheless, we can assume at least two rules for valid disparities:

1. Disparities must always be greater than zero
2. Disparities must always be smaller than image width, i. e. 960 px

Applying these rules on the data-set show that 40 images of the training-set (0.18 %) and 6 images of the test-set (0.14 %) contain invalid disparities.



Disparity distributions of training (bottom) and test (top) data-sets. Green/cyan colors show valid ranges, dark-red and red colors show invalid ranges

Algorithms and Techniques

After reading the publication of Isola et al.[10] proposing an image-to-image transformation framework based on Conditional Adversarial Networks (pix2pix), I was very impressed about the visual quality of the output their algorithm can achieve. It was not that I believed using Conditional

[10] https://phillipi.github.io/pix2pix/

Adversarial Networks would be much superior in solving the problem than other strategies, but their approach made me curious of its abilities to solve my problem. On the one hand, my problem perfectly fits in the domain of their framework. On the other hand, it was also a decision for the purpose of learning something new, something that had not been covered explicitly during the Nanodegree, like Generative or Conditional Adversarial Networks.

Inspired by the publication of Isola et al., I decided to apply a Conditional Adversarial Network to transform a single input image into its corresponding disparity map. This was done by letting a Generator network learn to generate a disparity map from an input image and a noise vector. A Discriminator network learned to distinguish these generated disparity maps from real disparity maps. In theory, the training converges, when the *Generator* produces disparity maps of such high quality that the Discriminator is not able to do its job. The following scheme is copied from Isola et al.:

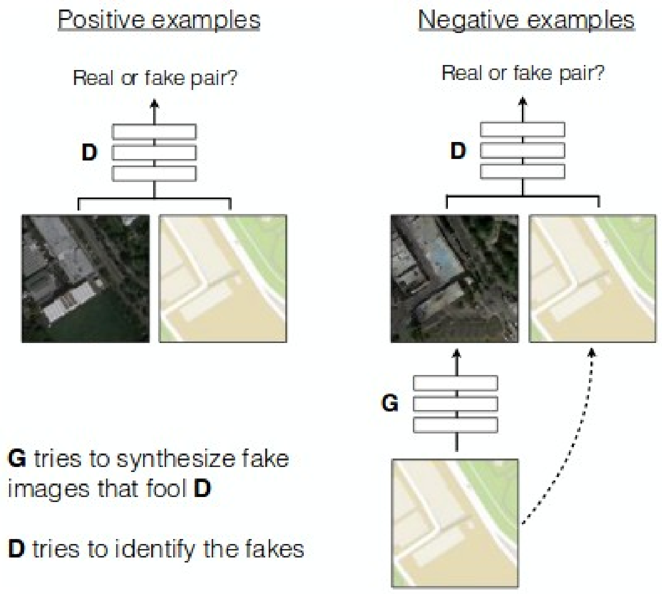

Positive examples
Negative examples
Real or fake pair?
D
Real or fake pair?
D
G
G tries to synthesize fake images that fool **D**
D tries to identify the fakes

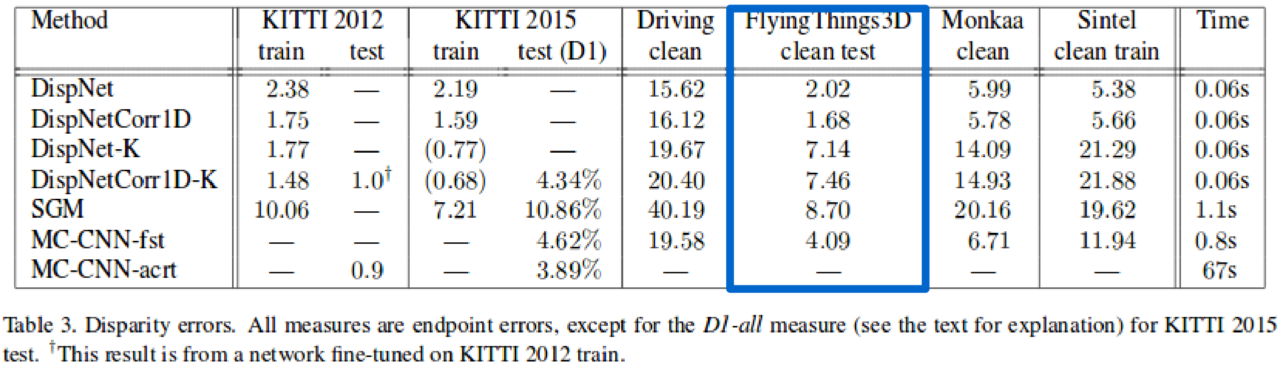
Figure 2: Training a conditional GAN to predict aerial photos from maps. The discriminator, D, learns to classify between real and synthesized pairs. The generator learns to fool the discriminator. Unlike an unconditional GAN, both the generator and discriminator observe an input image.

Scheme of a conditional GAN (copied from Isola et al.)

The beauty of this approach is, that the structured loss function, capturing the content deviations of two images or disparity maps, is learned implicitly during training of the networks and would be very difficult to formulate by hand. As far as I know, Conditional Adversarial Networks had not been used for depth/disparity estimation from single images until I started the project but the impressive examples shown in Isola et al. made me confident that the approach can handle this task as well.

Benchmark

Mayer et al. give benchmark results for their approach DispNet on their data-set and also compare it to a traditional Stereo Vision approach SGM. Within this capstone project I will compare my model to both of the mentioned models by calculating the error to the given ground-truth data in the same fashion. With this method I will be able to show how my model performs in contrast to the baseline.



Method	KITTI 2012		KITTI 2015		Driving	FlyingThings3D	Monkaa	Sintel	Time
	train	test	train	test (D1)	clean	clean test	clean	clean train	
DispNet	2.38	—	2.19	—	15.62	2.02	5.99	5.38	0.06s
DispNetCorr1D	1.75	—	1.59	—	16.12	1.68	5.78	5.66	0.06s
DispNet-K	1.77	—	(0.77)	—	19.67	7.14	14.09	21.29	0.06s
DispNetCorr1D-K	1.48	1.0†	(0.68)	4.34%	20.40	7.46	14.93	21.88	0.06s
SGM	10.06	—	7.21	10.86%	40.19	8.70	20.16	19.62	1.1s
MC-CNN-fst	—	—	—	4.62%	19.58	4.09	6.71	11.94	0.8s
MC-CNN-acrt	—	0.9	—	3.89%	—	—	—	—	67s

Table 3. Disparity errors. All measures are endpoint errors, except for the *D1-all* measure (see the text for explanation) for KITTI 2015 test. †This result is from a network fine-tuned on KITTI 2012 train.

Bechmark results (copied from Mayer et al.)

Methodology

Data Preprocessing

To keep the training with the chosen data-set as close to the original publication as possible, I modified the input data the following way:

- Drop the alpha channel of the RGBA images
- Resize the data to 256 x 256 px

To avoid negative effects on the input scaling and the training process itself, I decided to remove the outliers, i. e. the samples with invalid disparities as discussed in section II. After this, normalization was done by scaling the input with the global minimum and maximum values of the data-set:

$$X' = \frac{X - X_{min}}{X_{max} - X_{min}}$$

Finally, the training-set was shuffled and split into 90 % training-set and 10 % validation-set, giving the following segmentation:

	Training-set	**Test-set**
Samples	22390	4370
Samples w/o outliers	22350	4364
Training split	20115	
Validation split	2235	
Test Split		4364

Layout of the used data-set

Of course the test-set was held back during implementation and fine-tuning. It was only used to evaluate the final quality of the proposed approach.

Implementation

I picked Python 3 as programming language and Keras with Tensorflow back-end as Deep Learning framework to implement my solution. To get started, I tried to build on existing Keras implementations[11][12] of the original pix2pix framework by reimplementing and revalidating the original paper on the edges2shoes*[13]* data-set.

It turned out, that implementing the architecture within Keras is not straightforward, see appendix for details. While building the required residual connections within the U-Net Generator network requires usage of Keras' more advanced functional API, the definition of the overall training procedure is more involved as well.

To train two networks interleavingly within each iteration, I used the same algorithm as seen in the existing implementations mentioned above.

As these implementations break with the boundaries of some convenient Keras wrappers, I was not able to painlessly use all of Keras' smart data handling and feeding routines anymore. To avoid possible slow-downs in training data feeding, I implemented a batch queue mechanism that runs in a parallel thread. However, for performance speed-up the whole validation-set was preloaded into RAM, since the validation error was calculated at the end of every epoch.

On the algorithm side it is important to note, that the output of the Generator network is non-deterministic by design. According to the original paper by Isola et al. the noise within the pix2pix framwork, i. e. the conditional GAN, is introduced only using Dropout, even within inference time. Hence, reported quantities can vary in a certain range when a test is repeated.

During the first training runs it turned out, that - depending on the batch size - between approx. 14 and 100 hours per 100 epochs on a NVIDIA GTX 1080 Ti GPU, doing parameter optimization involving cross-validation is not feasible in a reasonable amount of time. Hence, I decided to simply optimize on a changing validation-set.

Refinement

After the initial implementation was generating similar results on the edges2shoes data-set to the ones produced by the used templates, I modified both architecture and parameters to optimize the performance on the FlyingThings3D data-set.

1. **Gray instead of color:** Since disparity maps can be considered as one-channel images, there was no need to feed a three-channel disparity map into the discriminator or for the generator to produce a three-channel output. Removing these unneeded channels results in fewer parameters to train. However converting the actual camera images into gray-scale images further reduces the trainable parameters, but throws away useful information for estimating the disparities. Hence, the RGB channels of the input images have been kept.

2. **Global normalization instead of sample normalization:** During several trials I found out that global normalization, as described above yields better results than sample normalization or sample standardization. In global normalization the minimum and maximum values are calculated from the whole data-set, whereas in sample normalization they are calculated for

[11] https://github.com/tjwei/GANotebooks
[12] https://github.com/tdeboissiere/DeepLearningImplementations/tree/master/pix2pix
[13] https://github.com/phillipi/pix2pix/blob/master/datasets/bibtex/shoes.tex

every sample from this sample. In sample standardization the mean of the sample is subtracted and the result is divided by its standard deviation . The removal of the outliers as described earlier, was also a result of several trials with different normalizations.

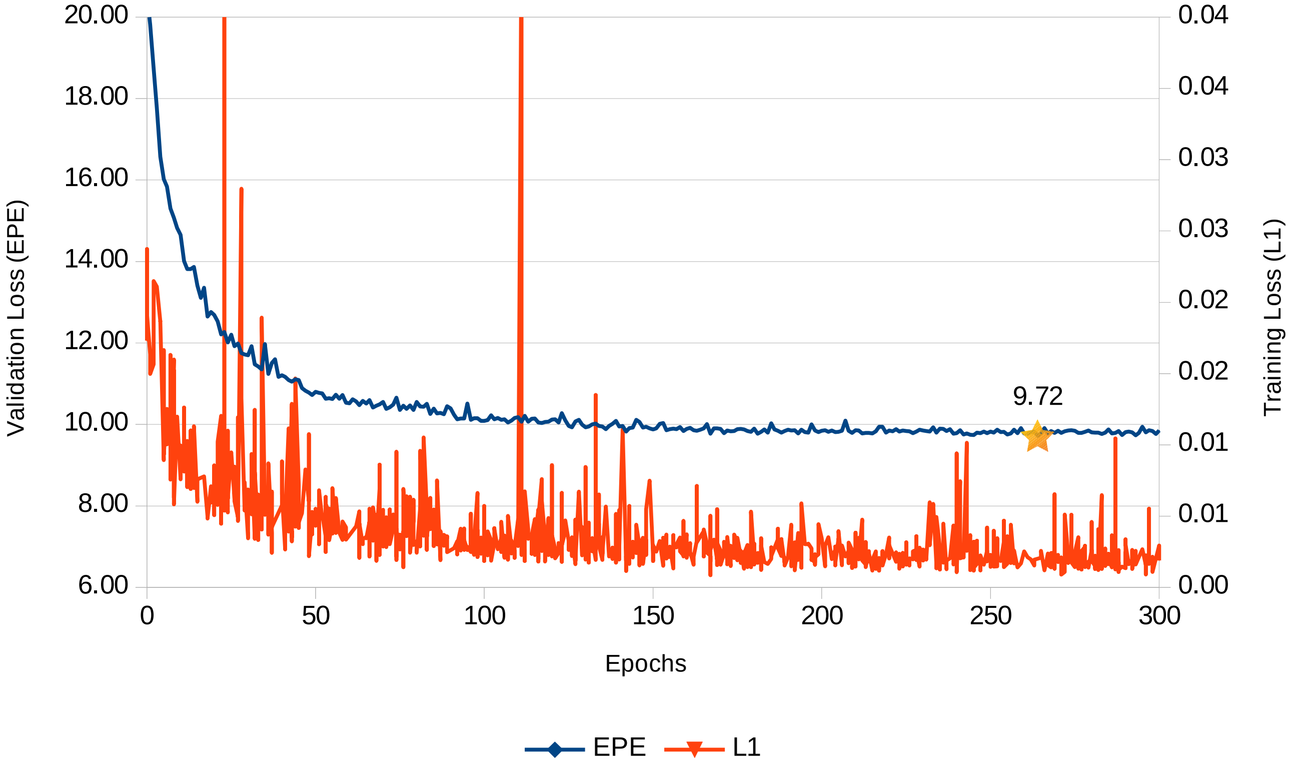
3. **Parameter optimization:** Different variations of the original parametrization have been tried. This includes changes of the learning rates, the network initializations, the optimizer types, the batch size, the spatial resolutions of input and output, the number of epochs and the receptive field size of the discriminator. After multiple training runs, I found it optimum for my problem and data-set by setting the Discriminator layout to PixelGAN (minimal receptive field[14]), the learning rate for both networks to 2e-5 (no decay for Adam optimizer suggested), and the batch size to 1. I trained over 300 epochs and kept the weights from that epoch yielding the smallest validation loss.

III. Results

Model Evaluation and Validation

The following chart plots the training and the validation losses. It clearly can be observed, that both curves start decreasing. As expected, the curve of the training loss is very noisy. This is because of the small batch size forces the optimizer to take rather sub-optimal steps during gradient descent than with bigger batch sizes.

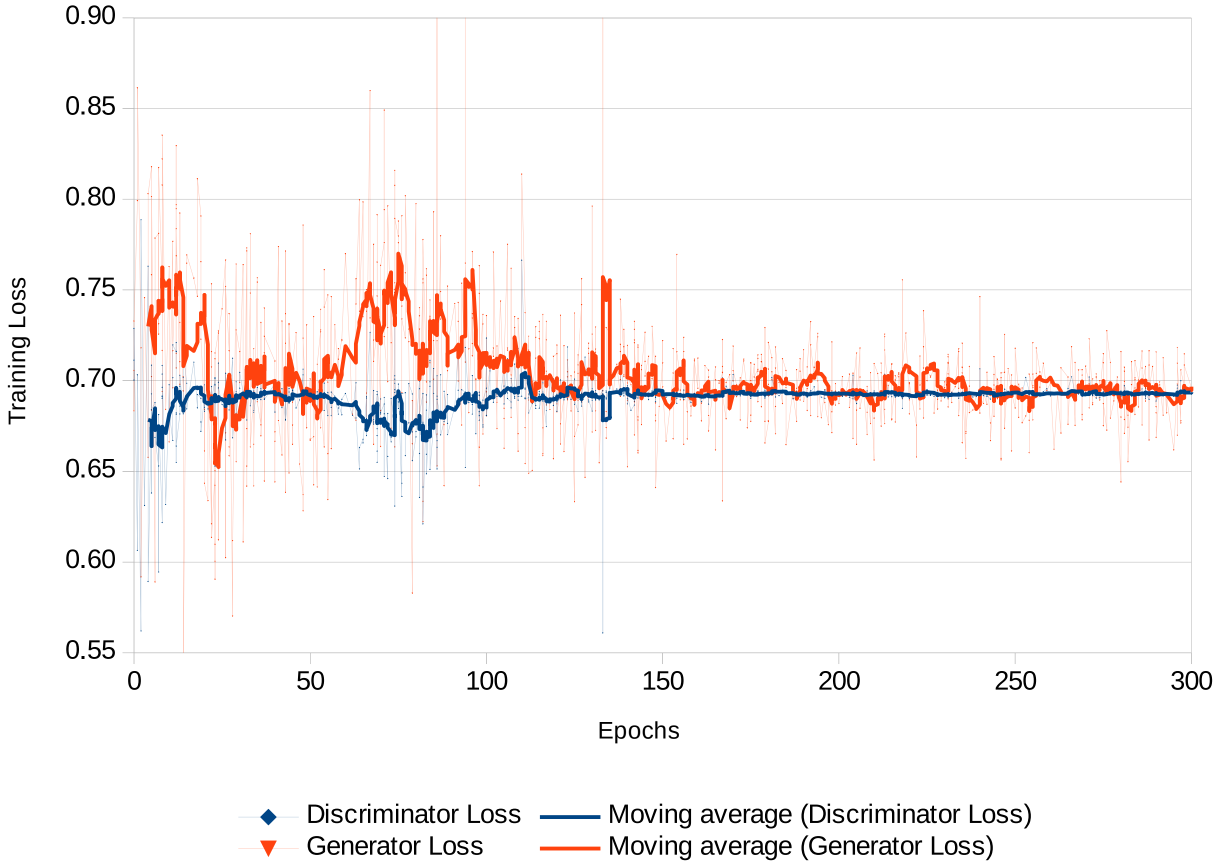
Around epoch 165 the validation loss (EPE) starts to plateau, reaching its minimum at epoch 265. Please note, that the values of the training and the validation losses are in separate domains since the validation loss has been corrected for the input scaling while the training loss has not.



Loss chart: training vs. validation

[14] https://github.com/rbgirshick/rcnn/blob/master/utils/receptive_field_sizes.m

Comparing the losses of the Discriminator and the Generator we can observe some clues of the competition during training between these networks. Both plots are very noisy, hence I added smoothed overlays to the chart. While the Generator loss generally seems to have a higher standard deviation than the Discriminator, one could see some hints that the spikes in the loss of one network is inverted by the other. That gives some evidence, that training progressed as it should. However, the magnitudes of the inversions are not identical and the Discriminator seems to be more “stable” than the Generator.

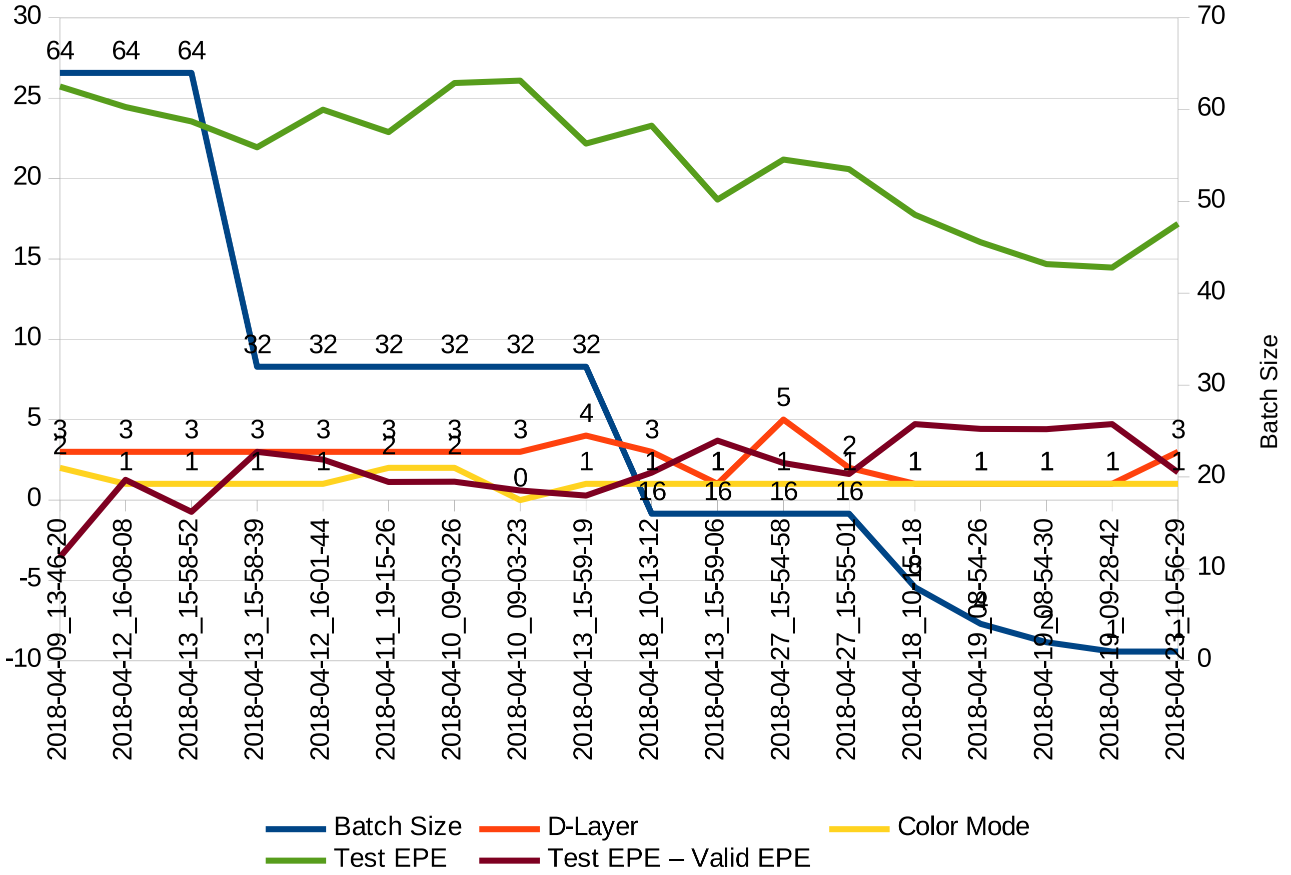


Loss chart: Discriminator vs. Generator

I conducted approx. 25 – 30 training runs for optimizing the parameters towards the lowest validation error (EPE). Even the chosen parameters yield the lowest validation loss, it is neither the training run with the most comprehensive loss charts nor with the visually most beautiful disparity map output. In addition to this, it has a very high discrepancy between validation error and test error.

After finishing the last 18 training runs with slightly different parameterizations I calculated their EPEs on the test-set. This is allowed, because I did not optimize the parameters afterwards, i. e. the test data is only used to report the actual quality of the last 18 trials once.

In the following chart I visualized for all of these runs the test EPE, the difference between test EPE and lowest validation EPE and with respect to parameterization: batch size, receptive field of the Discriminator (number of layers) and the applied color mode of every run.



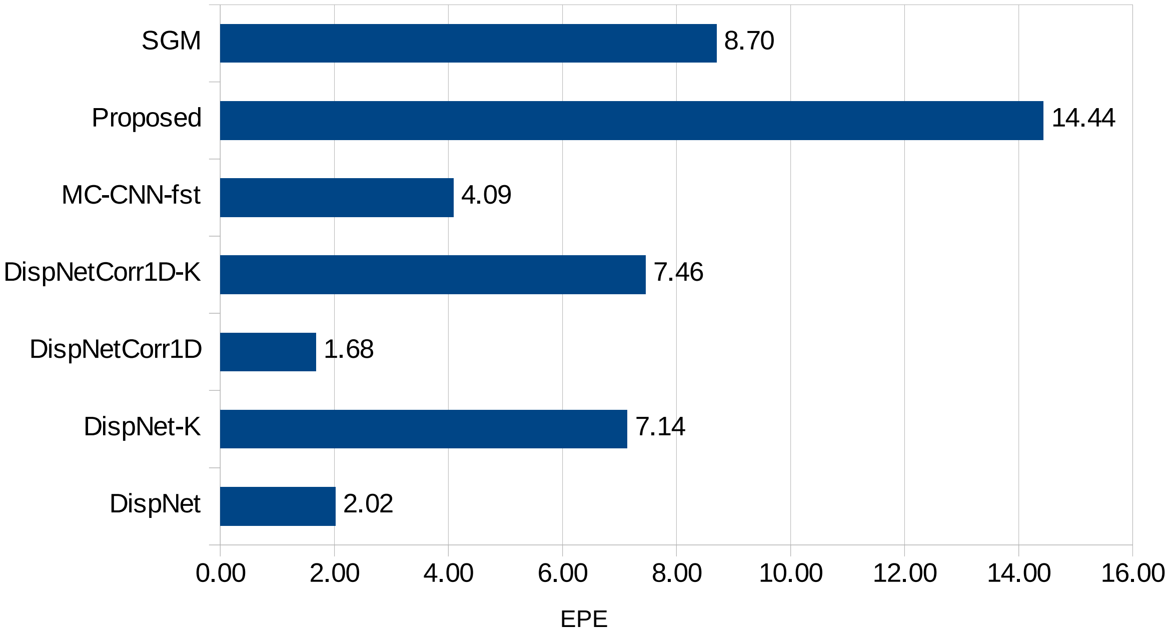
Comparing Test EPEs for training runs with different parameterizations

Interestingly it turned out, that lower batch sizes and simpler Discriminators yields lower validation errors. However, with simpler Discriminators the difference between validation error and test error seem to raise. There is also some clue that color mode 1, i. e. image 3-channel RGB and disparity map 1-channel gray, performs better than using greyscale images or estimating a 3-channel disparity map.

Without any proof of correctness, one explanation I could think of is, that simpler Discriminators tend to consider fewer pixels and the structural information of the image or disparity map has less influence. Since EPE is a per-pixel metric it is maybe beneficial if the Discriminator is not focused on structures but pixels. Or to put it differently: The training tries to get the single values right before it gets the structure right. Regarding batch size, a low batch size introduces more sub-optimal gradient-descent steps into the training process what maybe helps to overcome local minima. With a batch size of 1 the batch normalization applied within the model degrades to an instance normalization. Interestingly this seem to be beneficial in this set-up. However, the observation that the difference between validation and test errors is higher with simpler Discriminators is something that I have to leave for future analyses.

Justification

With the chosen approach I was able to generate disparity maps of plausible visual appearance, i. e. more distant areas have smaller values resulting in darker colors and closer objects have greater values resulting in brighter colors. Hence, it could be stated that my solution using a modified pix2pix framework with its Conditional Adversarial Network is able to solve the problem. However, with the state-of-the-art conventional SGM algorithm being around 1.6 times as accurate and the best Deep Learning based approach DispNet being around 8 times as accurate as the proposed approach I must admit that my solution might not be the best option to solve the disparity estimation problem. But out of the compared algorithms it is the only one using monocular image cues during inference and therefor it has a harder job than the benchmark approaches.

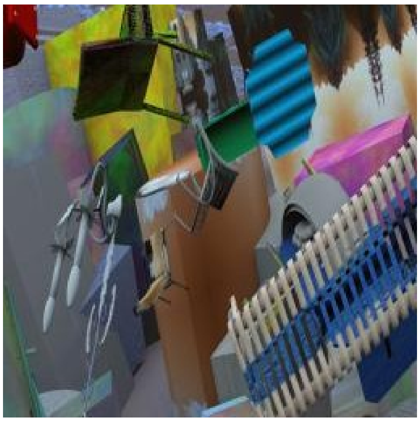


Comparison with the benchmark

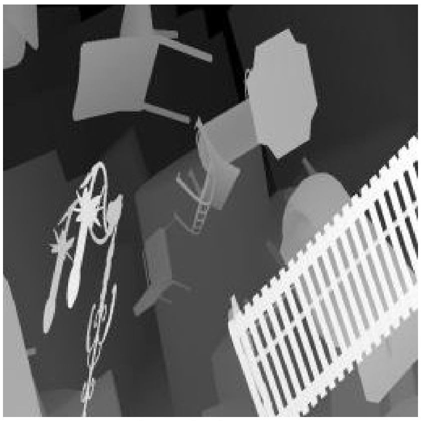
IV. Conclusion

Free-Form Visualization

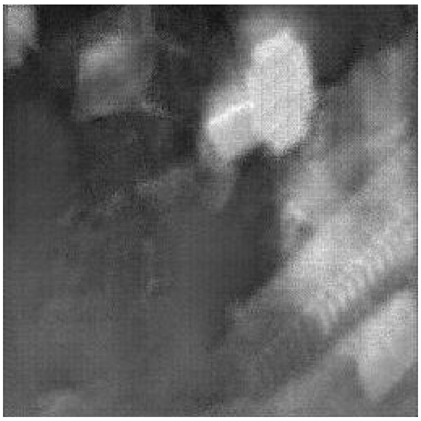
The following table shows a sample image from the validation-set, its corresponding ground-truth disparity map and the predictions of the generator network for this sample image after different training epochs. It can be seen, that beginning with epoch 165 the generations capture the depth structure of the scene quite okay, but lack the details and are somewhat blurry. This is most likely a symptom of the applied simple Discriminator.



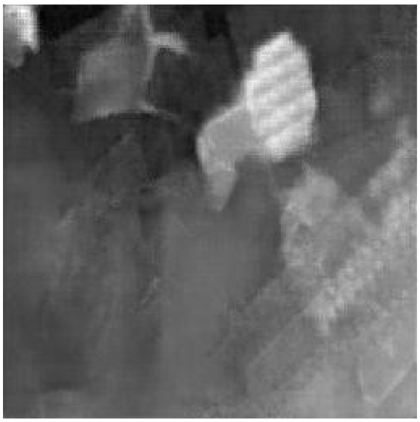
RGB image

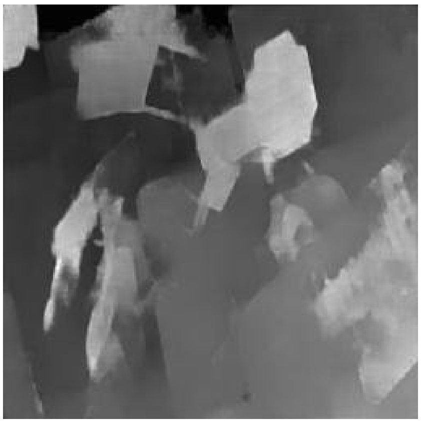


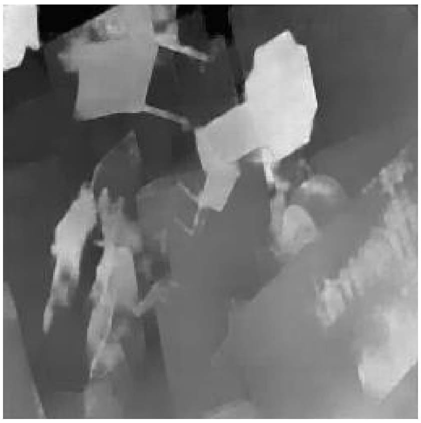
Ground-truth disparity map



Generator output (epoch 1)


Generator output (epoch 10)


Generator output (epoch 165)


Generator output (epoch 265)

Reflection

Within my project I applied the pix2pix framework, i. e. Conditional Adversarial Networks, to the problem of disparity estimation. I chose the FlyingThings3D data-set for this purpose, because it is big enough for Deep Learning based approaches and due to its synthetic generation, the ground truth disparity data is completely dense and accurate.

After implementing the core algorithm based on a publicly available solution, I validated its functionality based on the reference data-set before extending the framework and switching to my problem domain and data-set.

I modified especially the data handling and the evaluation routines and within dozens of training runs I optimized the parameters to the lowest validation loss. Finally, I did come up with a working solution but not with a new benchmark. Since my algorithm only uses one image per inference whereas the considered benchmark algorithms use two, I judge my experiments a success.

Improvement

As new techniques within Deep Learning seem to be published almost daily, I am quite sure that there are a lot of optimizations available or it is even possible that Conditional Adversarial Networks are not the very best architecture to tackle the problem at all.

First of all, one should revalidate and analyze my observations regarding the interaction of batch size, Discriminator complexity and test/validation EPE difference.

One improvement I could think of is to train on multiple data-sets using depth instead of disparity. When ground-truth disparity maps are converted to ground-truth depth maps using the corresponding base lines and focal lengths than the network could maybe learn a more robust depth estimation from a variety of data-sets followed by fine-tuning on the actual data-set in the end.

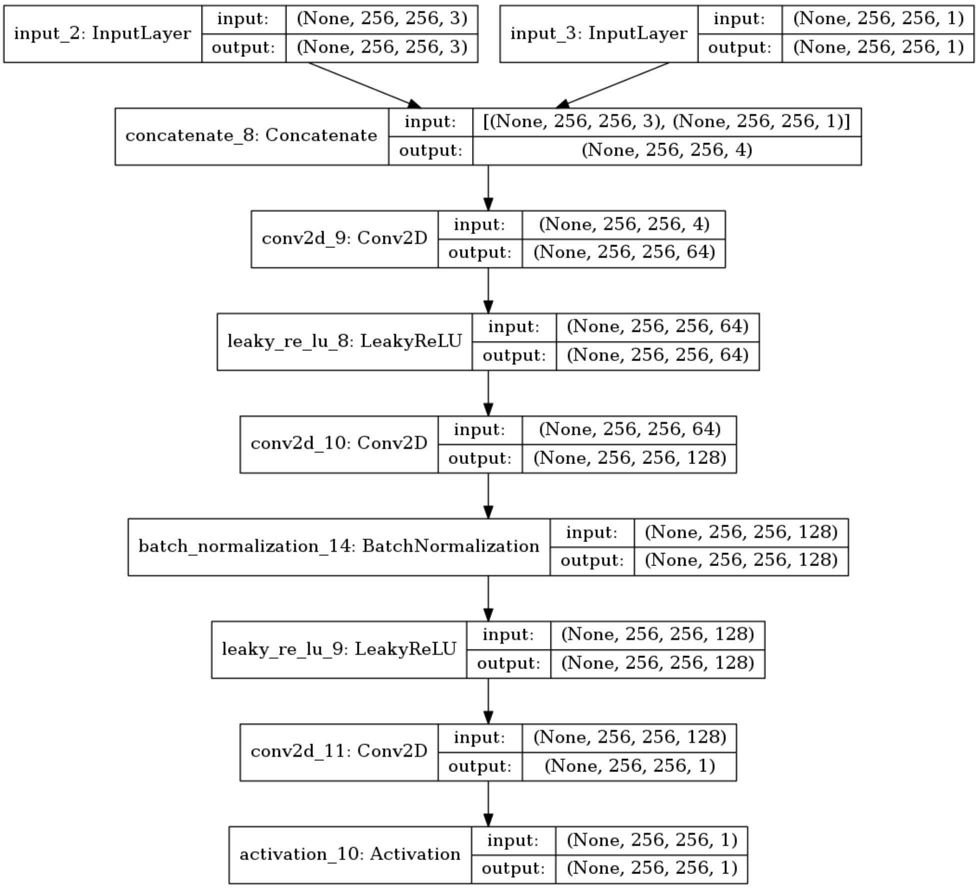
Recent technologies like Group Normalization[15] could replace Batch Normalization and maybe further improve the results. Then, one could apply a smarter algorithm than trial and error regarding hyper-parameter tuning to find the best parameter-set given the current architecture or just alter the architecture even more.

[15] https://arxiv.org/abs/1803.08494

V. Appendix

Discriminator Network

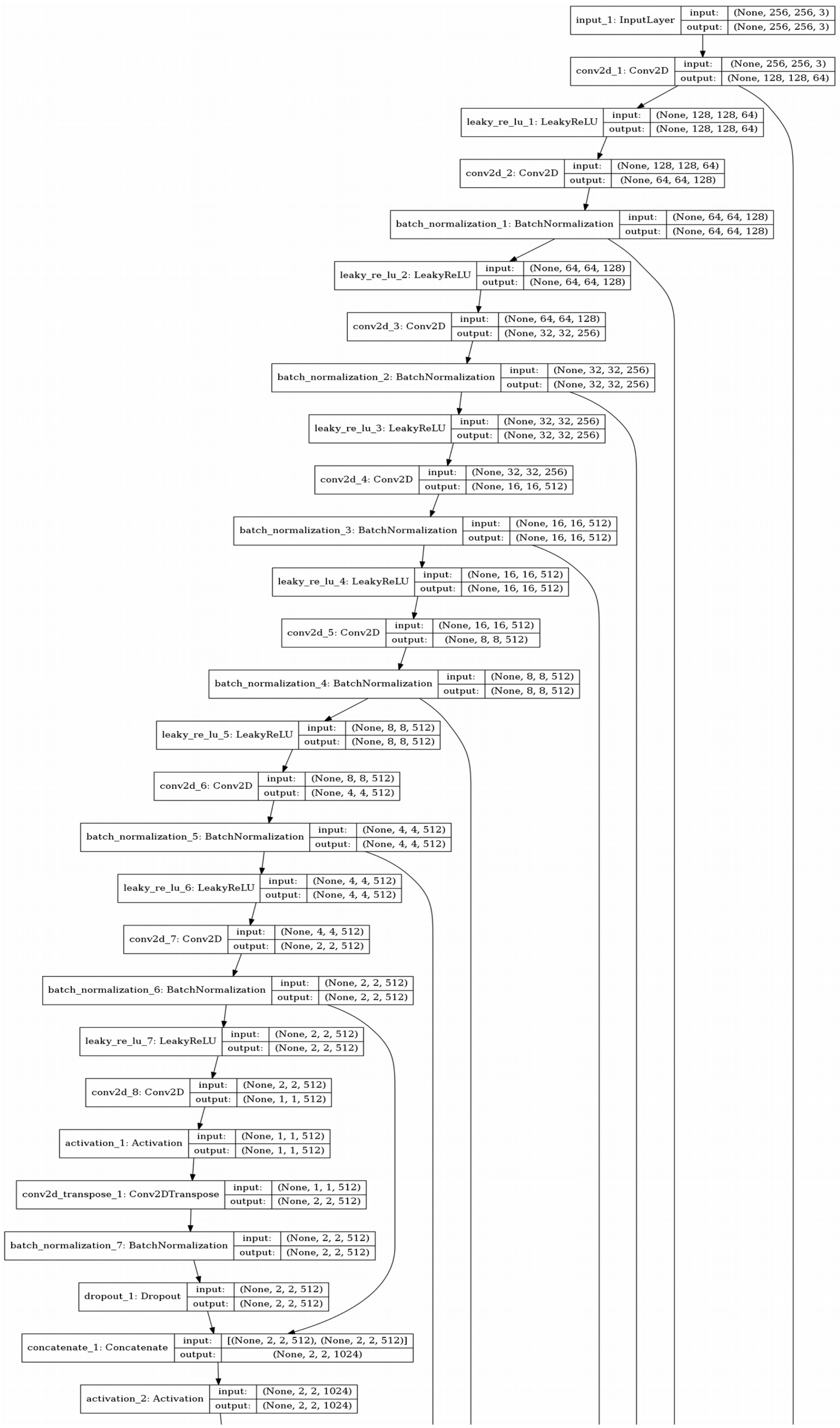
The following scheme outlines the final architecture I used for the discriminator network. Within the pix2pix framework, the discriminator has been set-up as a PixelGAN instead of a PatchGAN.

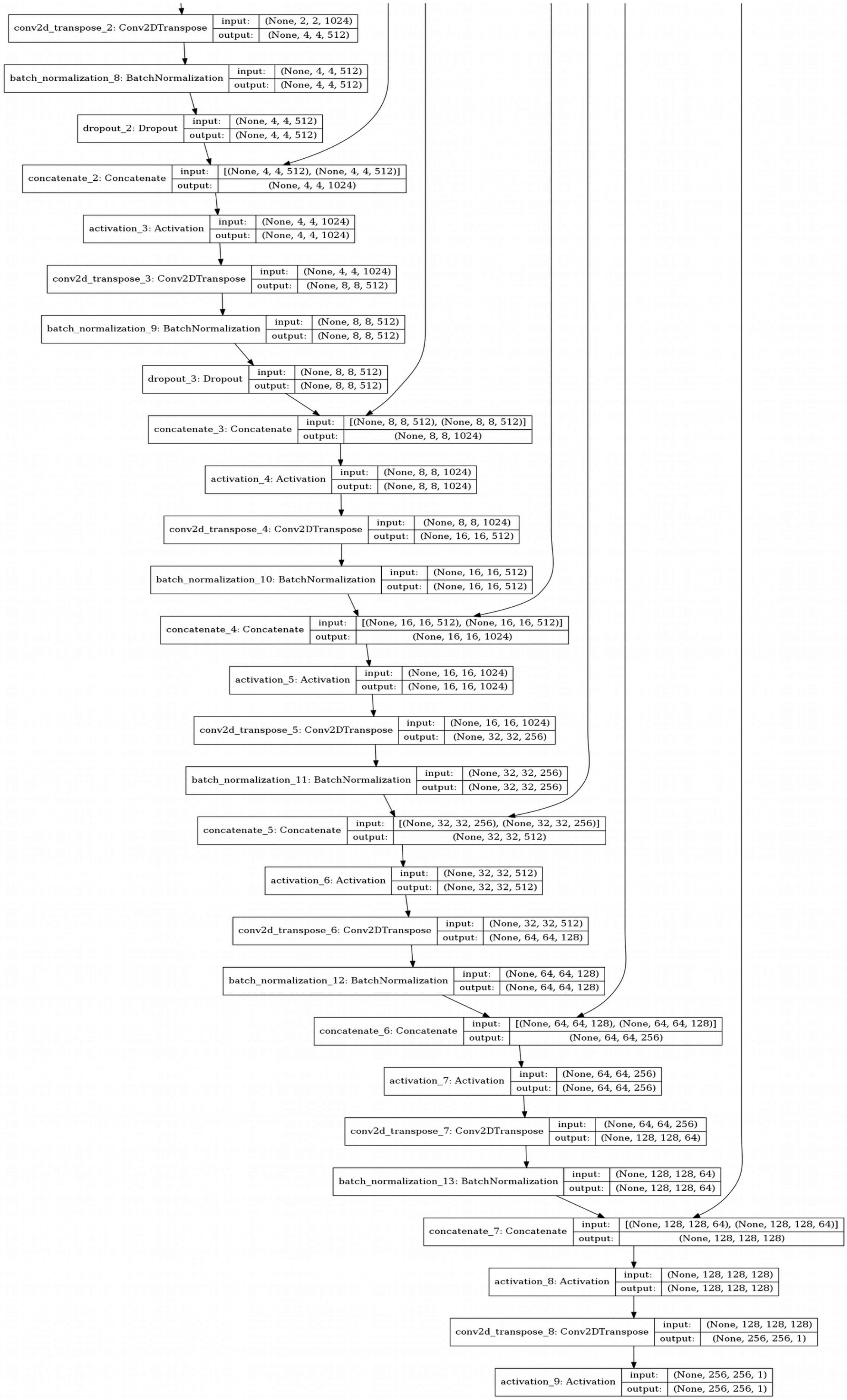

input_2: InputLayer
input: (None, 256, 256, 3)
output: (None, 256, 256, 3)
input_3: InputLayer
input: (None, 256, 256, 1)
output: (None, 256, 256, 1)
concatenate_8: Concatenate
input: [(None, 256, 256, 3), (None, 256, 256, 1)]
output: (None, 256, 256, 4)
conv2d_9: Conv2D
input: (None, 256, 256, 4)
output: (None, 256, 256, 64)
leaky_re_lu_8: LeakyReLU
input: (None, 256, 256, 64)
output: (None, 256, 256, 64)
conv2d_10: Conv2D
input: (None, 256, 256, 64)
output: (None, 256, 256, 128)
batch_normalization_14: BatchNormalization
input: (None, 256, 256, 128)
output: (None, 256, 256, 128)
leaky_re_lu_9: LeakyReLU
input: (None, 256, 256, 128)
output: (None, 256, 256, 128)
conv2d_11: Conv2D
input: (None, 256, 256, 128)
output: (None, 256, 256, 1)
activation_10: Activation
input: (None, 256, 256, 1)
output: (None, 256, 256, 1)

This model has 9025 trainable and 256 non-trainable parameters giving 9281 parameters total.

Generator Network

The next scheme visualizes the generator network as used in my solution. Within the pix2pix framework, this architecture is called U-Net, because of its involved residual connections. This network has 54425217 parameters total with 54415361 being trainable and 9856 being non-trainable.


input_1: InputLayer | input: (None, 256, 256, 3) | output: (None, 256, 256, 3)
conv2d_1: Conv2D | input: (None, 256, 256, 3) | output: (None, 128, 128, 64)
leaky_re_lu_1: LeakyReLU | input: (None, 128, 128, 64) | output: (None, 128, 128, 64)
conv2d_2: Conv2D | input: (None, 128, 128, 64) | output: (None, 64, 64, 128)
batch_normalization_1: BatchNormalization | input: (None, 64, 64, 128) | output: (None, 64, 64, 128)
leaky_re_lu_2: LeakyReLU | input: (None, 64, 64, 128) | output: (None, 64, 64, 128)
conv2d_3: Conv2D | input: (None, 64, 64, 128) | output: (None, 32, 32, 256)
batch_normalization_2: BatchNormalization | input: (None, 32, 32, 256) | output: (None, 32, 32, 256)
leaky_re_lu_3: LeakyReLU | input: (None, 32, 32, 256) | output: (None, 32, 32, 256)
conv2d_4: Conv2D | input: (None, 32, 32, 256) | output: (None, 16, 16, 512)
batch_normalization_3: BatchNormalization | input: (None, 16, 16, 512) | output: (None, 16, 16, 512)
leaky_re_lu_4: LeakyReLU | input: (None, 16, 16, 512) | output: (None, 16, 16, 512)
conv2d_5: Conv2D | input: (None, 16, 16, 512) | output: (None, 8, 8, 512)
batch_normalization_4: BatchNormalization | input: (None, 8, 8, 512) | output: (None, 8, 8, 512)
leaky_re_lu_5: LeakyReLU | input: (None, 8, 8, 512) | output: (None, 8, 8, 512)
conv2d_6: Conv2D | input: (None, 8, 8, 512) | output: (None, 4, 4, 512)
batch_normalization_5: BatchNormalization | input: (None, 4, 4, 512) | output: (None, 4, 4, 512)
leaky_re_lu_6: LeakyReLU | input: (None, 4, 4, 512) | output: (None, 4, 4, 512)
conv2d_7: Conv2D | input: (None, 4, 4, 512) | output: (None, 2, 2, 512)
batch_normalization_6: BatchNormalization | input: (None, 2, 2, 512) | output: (None, 2, 2, 512)
leaky_re_lu_7: LeakyReLU | input: (None, 2, 2, 512) | output: (None, 2, 2, 512)
conv2d_8: Conv2D | input: (None, 2, 2, 512) | output: (None, 1, 1, 512)
activation_1: Activation | input: (None, 1, 1, 512) | output: (None, 1, 1, 512)
conv2d_transpose_1: Conv2DTranspose | input: (None, 1, 1, 512) | output: (None, 2, 2, 512)
batch_normalization_7: BatchNormalization | input: (None, 2, 2, 512) | output: (None, 2, 2, 512)
dropout_1: Dropout | input: (None, 2, 2, 512) | output: (None, 2, 2, 512)
concatenate_1: Concatenate | input: [(None, 2, 2, 512), (None, 2, 2, 512)] | output: (None, 2, 2, 1024)
activation_2: Activation | input: (None, 2, 2, 1024) | output: (None, 2, 2, 1024)


conv2d_transpose_2: Conv2DTranspose
input: (None, 2, 2, 1024)
output: (None, 4, 4, 512)
batch_normalization_8: BatchNormalization
input: (None, 4, 4, 512)
output: (None, 4, 4, 512)
dropout_2: Dropout
input: (None, 4, 4, 512)
output: (None, 4, 4, 512)
concatenate_2: Concatenate
input: [(None, 4, 4, 512), (None, 4, 4, 512)]
output: (None, 4, 4, 1024)
activation_3: Activation
input: (None, 4, 4, 1024)
output: (None, 4, 4, 1024)
conv2d_transpose_3: Conv2DTranspose
input: (None, 4, 4, 1024)
output: (None, 8, 8, 512)
batch_normalization_9: BatchNormalization
input: (None, 8, 8, 512)
output: (None, 8, 8, 512)
dropout_3: Dropout
input: (None, 8, 8, 512)
output: (None, 8, 8, 512)
concatenate_3: Concatenate
input: [(None, 8, 8, 512), (None, 8, 8, 512)]
output: (None, 8, 8, 1024)
activation_4: Activation
input: (None, 8, 8, 1024)
output: (None, 8, 8, 1024)
conv2d_transpose_4: Conv2DTranspose
input: (None, 8, 8, 1024)
output: (None, 16, 16, 512)
batch_normalization_10: BatchNormalization
input: (None, 16, 16, 512)
output: (None, 16, 16, 512)
concatenate_4: Concatenate
input: [(None, 16, 16, 512), (None, 16, 16, 512)]
output: (None, 16, 16, 1024)
activation_5: Activation
input: (None, 16, 16, 1024)
output: (None, 16, 16, 1024)
conv2d_transpose_5: Conv2DTranspose
input: (None, 16, 16, 1024)
output: (None, 32, 32, 256)
batch_normalization_11: BatchNormalization
input: (None, 32, 32, 256)
output: (None, 32, 32, 256)
concatenate_5: Concatenate
input: [(None, 32, 32, 256), (None, 32, 32, 256)]
output: (None, 32, 32, 512)
activation_6: Activation
input: (None, 32, 32, 512)
output: (None, 32, 32, 512)
conv2d_transpose_6: Conv2DTranspose
input: (None, 32, 32, 512)
output: (None, 64, 64, 128)
batch_normalization_12: BatchNormalization
input: (None, 64, 64, 128)
output: (None, 64, 64, 128)
concatenate_6: Concatenate
input: [(None, 64, 64, 128), (None, 64, 64, 128)]
output: (None, 64, 64, 256)
activation_7: Activation
input: (None, 64, 64, 256)
output: (None, 64, 64, 256)
conv2d_transpose_7: Conv2DTranspose
input: (None, 64, 64, 256)
output: (None, 128, 128, 64)
batch_normalization_13: BatchNormalization
input: (None, 128, 128, 64)
output: (None, 128, 128, 64)
concatenate_7: Concatenate
input: [(None, 128, 128, 64), (None, 128, 128, 64)]
output: (None, 128, 128, 128)
activation_8: Activation
input: (None, 128, 128, 128)
output: (None, 128, 128, 128)
conv2d_transpose_8: Conv2DTranspose
input: (None, 128, 128, 128)
output: (None, 256, 256, 1)
activation_9: Activation
input: (None, 256, 256, 1)
output: (None, 256, 256, 1)