DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Games Engineering

Deep Learning for Video Depth Estimation from Defocus

Kevin Galim

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Games Engineering

Deep Learning for Video Depth Estimation from Defocus

Deep Learning für Video Depth Estimation aus der Bildunschärfe

Author:	Kevin Galim
Supervisor:	Prof. Dr. Laura Leal-Taixé
Advisor:	Maxim Maximov
Submission Date:	29.08.2019

I confirm that this master's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, 29.08.2019 Kevin Galim

Abstract

Depth estimation from RGB images is a challenging problem with many applications like 3D vision for robots or refocusing images after they have been recorded. Hardware solutions for recording depth like RGB-D cameras are often not applicable or it is desired to produce depth maps after the capture process. A way of predicting depth from RGB images is to use the amount of present defocus in a frame. Since the defocus blur degree is dependent on the object distance and the camera focus distance, it is possible to make assumptions on how far objects are away from the camera. To distinguish between blur and object texturing, multiple frames with different focus are used as input. Common approaches usually require the captured scene to be completely static making it impractical for dynamic video sequences. Our goal is thus to allow camera or object movement between the frames and eventually predict depth maps for arbitrary, continuous video sequences where the focus distance is always changed each frame. For this purpose, we are presenting several end-to-end trainable convolutional neural networks which can use the out-of-focus blur for making pixel-wise depth predictions for the displayed scene. Among the proposed networks, the recurrent autoencoder produces the best quality results. This type of network is a CNN which behaves similar to an LSTM but can handle multi-channel 2D images as input. Training CNNs typically demands a large dataset which has to fulfill special characteristics in our case. Since we are not aware of any existing dataset we create a labeled real-world dataset for training using an RGB-D camera and an Android smartphone. For that, we show the necessary calibration procedure and registration process to get aligned color and depth frames. Furthermore, we develop suitable Android applications which automatically change the focus every frame in a way that meets exactly the requirements for our CNN architectures. However, since manual recording is very time consuming and neural networks are known for needing a lot of data for training, we demonstrate how to efficiently create a synthetic dataset using the 3D graphics program Blender. Applying our approach, we can fully automatically produce large training datasets for our architectures. Trained on artificial data, our final models are nevertheless able to successfully generalize to real-world situations. Our solution, in the end, produces plausible depth map predictions even for challenging scenarios e.g. for settings under the influence of fast camera movement.

Contents

1 Introduction

1.1 Depth from Defocus

Depth prediction from images is a challenging problem with a long research history [Bho19]. There are several different ways to estimate depth from single images. One way to do so is to use the amount of out-of-focus blur which is caused by objects not being in the lens focus [KRT16]. By using the fact, that the degree of blur depends on how far the respective object is off the lens focus, it is possible to make conclusions about its distance to the camera. This process can be further improved by using multiple captures with the same camera position but a different focus in each frame. In this way, out-of-focus blur can be better distinguished from mere object texturing. The term for a collection of frames recorded from the (typically) same camera pose but with a different focus is called a focus stack. The process of analyzing the defocus blur works well in static scenes which, apart from the focus change, do not contain any other change like object movement. During the recording of a whole image stack though it is more likely that certain components are moving. Our goal in this work is thus to successfully predict depth maps from focus stacks under the influence of camera and object movement. This aim is more challenging since correspondences of dynamic objects have to be now solved as well which is not the case when dealing with sole static focus stacks.

Figure 1.1 visualizes the idea of this project. We have a focus stack with several images recorded at different focus as input. The number of frames thereby can be arbitrary. Also, movement during the capture process is allowed. As in the graphics, the frame at the bottom has a close focus while the frame on top was recorded with a far focus. During the recording time of the sequence, the focus distance was increased which is visualized by the sine-like plot in the figure. Furthermore, the camera moved to the left which caused the scene to shift. The desired output is then a prediction of a depth map which contains an estimate for the distance to the camera for each pixel. We are inputting multiple images whereas the final prediction is a single depth map referring to the last frame in the input stack.

Knowing the depth of RGB images serves many applications. For instance, having depth knowledge in Augmented Reality (AR) systems is a key to plausibly place virtual objects in the real world [Ber+14]. Also, in the field of robotics, robots need to have an

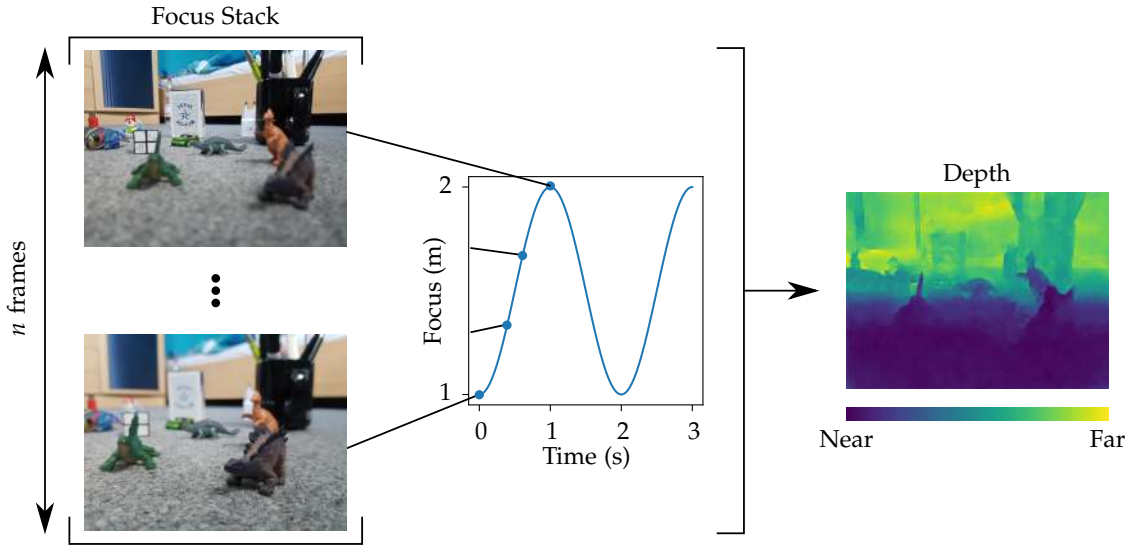


Figure 1.1: Project concept.

estimate of how far they are away from objects in the world to properly interact with them [Cas+18]. A more artistic use case is the refocusing of images after they have been recorded to naturally emphasize certain regions in the picture [KRT16]. Related to this application is also the computation of an image where all objects are in perfect focus. This aim is impossible to achieve by recording since technically only objects which are exactly positioned at the focus distance appear in complete focus. On the other hand, it is often not needed to have mathematically perfect focus since there is some tolerance for the human eye to consider objects as sharp. The threshold where objects are considered as sufficiently sharp is also referred to as the Depth of Field (DoF). Nonetheless, if objects in the scene are very far apart from each other, one typically has to decide to focus on only one of them while the others are likely to blur in the recording. However, having an estimating for the image depth, one can utilize the dependency between the degree of blur and the object distance and thereby reverse the out-of-focus blur to produce an image where all objects appear in focus.

1.2 Deep Learning

Deep Learning [GBC16] has shown great impact in common computer vision tasks such as image classification [KSH12], semantic segmentation [RFB15], super resolution [WCH19] and face recognition [Bal19]. In many disciplines, deep learning methods are able to outperform conventional approaches. When dealing with images, the Convo-

lutional Neural Network (CNN) is predominant. CNNs make use of the convolutional operation which is well suited for 2D multi-channel image processing, unlike classical fully connected neural networks which can merely input them as stacked 1D vectors. One of the most influential works thereby is the AlexNet [KSH12] which won the ImageNet Large Scale Visual Recognition Challenge in 2012, with a top-5 error which was lower by more than 10% compared to all other approaches. The authors used a GPU implementation of a deep CNN architecture which made it feasible to train. The deep structure of the network thereby led to its great success. Furthermore, CNNs are getting more and more attention in the field of depth prediction. We thus also choose to use deep learning architectures for realizing our goal. Their flexibility and ability to learn arbitrary complex relations makes it the most promising method to tackle our desired depth from defocus project which is especially challenging in terms of additionally handling scene movement. We think that CNNs have the greatest potential for our purpose compared to other, more conventional methods.

1.3 Circle of Confusion

As previously mentioned, there is a relationship between the out-of-focus blur strength and the camera distance. If a single point in the scene is in perfect camera focus it will also appear as a single point on the image. Otherwise, the point will blur and appear as a circle with a certain radius on the picture. The diameter of this circle is also called the Circle of Confusion (CoC) [PC82]. Therefore the CoC is a measure of how much in or out of focus a point in the scene is. Figure 1.2 shows an overview of a camera lens. In the figure, there are three different pairs of light rays visualized as blue, orange and

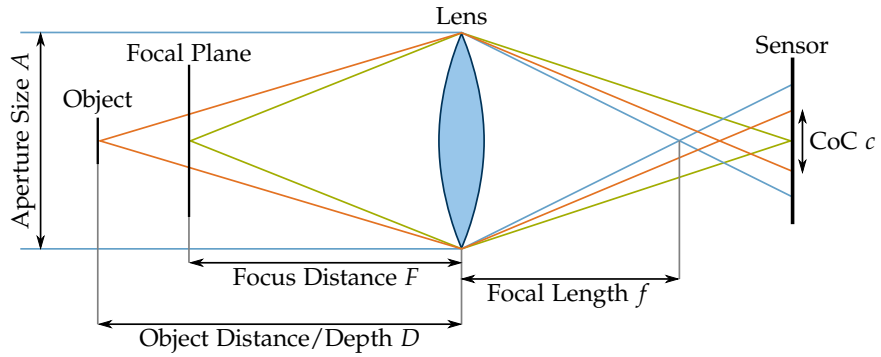


Figure 1.2: Lens model.

green lines which travel from the scene (left) to the camera sensor (right) where the image is finally synthesized. On their way, they pass the camera lens which alters the

direction of the rays towards the image sensor. The plane where are all points appear completely in focus in the final image is called the focal plane. The distance between the focal plane and the camera lens is called the focus distance. In this figure, there is one point on the focal plane which emits the green light rays. It can be observed that these light rays meet exactly in one point on the image sensor which makes it appear in focus. The diameter of the circle of confusion is equal to zero. Now we assume that we have a point on an object which lies behind the focal plane, which emits the orange rays in the figure. In this case, we can see that the rays do not meet at a single point on the sensor but are apart from each other. The distance between the entry points between the two rays is given by the circle of confusion, the diameter of the circle on the image caused by the point in the scene. The last case, the blue rays, are parallel light rays. This would occur if a point would be infinitely far away from the camera which is not possible in the real world. The distance between the lens and the ray intersection point is thereby called the focal length.

In a more mathematic fashion the circle of confusion c can be calculated as

$$c = \frac{Af|D - F|}{D(F - f)}. \tag{1.1}$$

where A is the aperture size/diameter, f the focal length, D the object distance or depth which we want to predict later and F the camera focus distance. Here we can see the direct interplay of the circle of confusion and the depth. The more the depth D and the focus distance F vary from each other, the larger the circle of confusion c will be. Instead of expressing this equation with the aperture size A, we can also write it using the f-number N which is the ratio between the focal length f and the aperture size A

$$N = \frac{f}{A}. \tag{1.2}$$

Thus we can rewrite the CoC as

$$c = \frac{f^2|D - F|}{ND(F - f)}. \tag{1.3}$$

1.4 Summary of Contributions

In short, our goal can be summarized as solving the challenging problem of predicting accurate depth maps for dynamic RGB video sequences based on the changing defocus blur induced by altering the focus distance. For that, we use the power of deep convolutional neural networks which are ideal for image processing while being able to learn complex data relations. Our contribution thereby is composed of the following points:

- We propose several architectures which successfully make use of the out-of-focus blur to produce plausible depth map predictions for diverse real-world scenarios. We show that our approach functions well even in challenging circumstances such as fast camera movement and reflective objects by testing on several real-world test sets.
- We describe a method to collect a labeled real-world dataset using an Android smartphone camera and an RGB-D sensor. We develop suitable Android applications which automatically change the focus distance during the recording process and show how to synchronize and align the captured color and depth images.
- We implement a fully automatic mechanism to create arbitrary large datasets synthetically. We thereby use the 3D graphics software Blender [Com17] to create and render focus sweeping scenes which are composed of random objects and textures. Using this technique we can produce a large amount of training data at little expense. We later show that the training on synthetic data nevertheless leads to very good generalization on real-world datasets.

2 Related Work

There are already several approaches on how to tackle the depth estimation problem from a focus stack of images. Two of them are Deep Depth From Focus [Haz+18] and Video Depth-From-Defocus [KRT16]. Additionally, since we are processing video data, we also refer to previous work on CNNs which infer from image sequences and have properties of interest for our needs. Specifically, we are going to analyze the two techniques from [AD18] and [Cha+17].

2.1 Depth Prediction

2.1.1 Deep Depth From Focus (DDFF)

In the paper "Deep Depth From Focus" [Haz+18], the goal is to predict a depth map for a static focus stack of images. The authors of this paper report the best result by using a stack of ten frames where the focus distance increases in every subsequent frame. To predict a depth map from the input they suggest to use a deep learning approach which in their case is a modified VGG16 [SZ14] encoder-decoder architecture. Their contribution lies in additionally implementing batch normalization layers [IS15] and adding dropout to the CNN. They compare their approach with other similar work and claim to reach the best performance. Besides the depth prediction approach, they also describe how to collect an appropriate real-world dataset for training and testing. Their dataset is recorded by using a light-field camera which allows accurate refocusing of images after the image has already been taken. On top of the light-field camera, they attach an RGB-D sensor which can measure the depth like a camera can measure the color. These depth maps are then used as ground truth to train the neural net. A challenge thereby is to calibrate the light-field camera and the RGB-D sensor with each other to later be able to align the produced depth maps to match the color images. Without proper calibration, the RGB and the depth recordings would not fit since the lenses are shifted.

We notice that in practice the DDFF struggles to generalize to scenarios which have not been present in the training set. Nevertheless, we also make use of the original DDFF and additionally extend it by an LSTM which is placed in between the encoder and the decoder. Thus the network can better use the temporal information which

improves the performance and generalization power of the DDFF. We thereby also show how to transfer the skip connections properly from the encoder to the decoder in the presence of the LSTM. Similar to their method, we record a real-world dataset using a calibrated setup consisting of an Android phone and an RGB-D depth sensor. We cannot use a light field camera since these devices are unable to record full video sequences which is critical in our case.

2.1.2 Video Depth-From-Defocus

Video Depth-From-Defocus [KRT16] is another approach for depth estimation. But unlike DDFF, their method works with videos which means that slight movement can happen between the frames which is similar to our aim. They use a DSLR camera to capture their scenes. However, the way they record differs from the typical way. During the filming process, they manually change the lens focus continuously. The focus distance is first increased to a certain maximum and then decreased again to its initial value. This procedure is then periodically repeated until the end of the video. To estimate the depth later they do not use a deep learning approach but a multi-step variational approach. The first step thereby is the alignment of the video frames to each other, meaning reversing the movement that happened between the frames. This is done by back warping each frame to the first by using an optical flow estimation. A difficulty here is to only reverse the movement and not alter the original focus of the frame since the focus is later required to estimate the CoC and the depth. The authors suggest PatchMatch [Bar+09] to compute the flow which seems to be relatively resistant to focus changes. The next steps of the variational approach are depth estimation, computing an all-in-focus image and refining the focus. As a result, they are not only able to compute a depth map but also an image where all parts are in focus.

We adapt the concept of focus sweeping in videos for our approach. We will thereby use an Android phone instead of a DSLR where the focus distance can be modified via software resulting in less camera shake. Furthermore, we are not modifying the focus distance periodically but merely ramp from the minimum to the maximum focus and then restart again from the minimum. This reduces the complexity of the data and is doable with a typical Android camera lens running on a specifically designed application. Their approach is limited regarding the amount of motion which can be compensated whereas our architectures manage to successfully produce decent depth maps even for scenes that involve a large amount of movement.

The Video Depth-From-Defocus authors also test their approach on the data provided by Suwajanakorn et. al [SHS15]. Suwajanakorn et. al predict depth from focal stacks as well, however under the assumption that the input focus stack is perfectly static. They, therefore, provide a dataset which is perfectly aligned and contains no movement. We

as well make use of the dataset later to test and compare our network architectures. The captures of static scenes are an ideal starting point to get a first impression of the performance of our models.

2.2 Deep Learning Approaches for Image Sequence Processing

2.2.1 Burst Image Deblurring Using Permutation Invariant Convolutional Neural Networks

In [AD18] the aim is to create a sharp, noise-free image from a set of blurry and noisy captures. This is useful for smartphone cameras which are usually influenced by severe camera shake due to their small lens optics. By recording multiple low-quality frames in sequence and feeding them to their proposed neural network architecture, they are able to produce one high-quality image as a result. For this, they introduce the Permutation Invariant Convolutional Neural Network which can input an arbitrary number of input images in an order-independent manner to produce one final prediction. The concept is similar to a regular CNN which processes each frame independently. Their contribution lies in additionally inserting global maximum pooling layers between regular CNN layers which collect information from each input stream and forward it to the next layers in the form of maximum feature maps. This information exchange enables the network to consider the relations between the images rather than processing each frame independently. They further compare their method to other state-of-the-art image restoration techniques and show that their CNN produces the best results.

We decide to modify and adapt this network type for our needs since its capability of learning from moving images is very suitable for us. We thereby lower the original large feature channel count which is not necessary for the depth prediction process and introduce batch normalization layers which stabilize the training. Also, we omit the global maximum pooling layers in the decoder fraction of the network to process the encoding output independently. The idea is that the encoder should already detect the relationship and change of the CoC and store this information in the encoding whereas the decoder should only use this information to make an independent estimate of the depth for each frame which will later be combined and scored by a final global pooling layer and a small CNN.

2.2.2 Interactive Reconstruction of Monte Carlo Image Sequences using a Recurrent Denoising Autoencoder

Another video image processing deep learning technique was introduced by [Cha+17] for denoising render sequences. Realistic rendering approaches often tend to produce

output which is contaminated by noise which significantly reduces the visual experience. Thus there is a great need of denoising techniques which task is to remove the image noise in the render output as an additional post-processing step. The authors thereby develop a CNN named the recurrent autoencoder which is successfully able to produce aesthetic noise-free images. The input thereby is not only composed by a single image but by a whole image sequence of noisy rendering outputs which are sequentially given to the model. By using specifically designed recurrent skip connections the network can learn temporal information to improve the output quality. These recurrent skip connections thereby provide information about the previous frame output to the currently processed frame. Like an LSTM [HS97] the CNN keeps a hidden state which updates every time a new frame is input. However, LSTMs are usually limited to only being able to input a small vector whereas the recurrent autoencoder can input fully-sized 2D images.

The recurrent structure and ability to process complete images make it also interesting for our task so we come up with a derived structure. We add batch normalization and increase the number of feature channels which are very few originally since denoisers are expected to operate very performantly. Also, we replace the static nearest neighbor upsampling layers with learnable transposed convolutions to give the model more flexibility.

3 Approach

3.1 Convolutional Neural Networks (CNNs)

To predict the depth map for our focus ramp video sequence we want to make use of convolutional neural networks (CNNs) [KSH12]. For images, CNNs are generally more suitable than traditional, fully connected neural networks since they have far less learnable parameters and are translational invariant. CNNs typically input 2D multi-channel images and use the convolutional layer as the main component which is applying a discrete convolution mask on each pixel. For the continuous case, the 2D convolution for the weight function w and the image function f at the coordinate (x, y) is defined as

$$(w * f)(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} w(x', y') f(x - x', y - y')\, dx'\, dy'. \tag{3.1}$$

The convolutional layer uses a discrete convolution though where the weights w are tensors with limited dimensions and are learned during training. An image usually has three input channels to store RGB information which is why $w \in \mathbb{R}^{C \times K \times K}$ is usually a 3D tensor with the number of input channels C (e.g. $C = 3$ for RGB) and the kernel size K (commonly $K = 3$). The output of the layer is called a feature map or an activation map. One convolutional layer usually also has a whole set of these weight tensors w resulting in the output of multi-channel feature maps. Since there are no pixels in the neighborhood around the image border the convolution does not get applied to those. This shrinks the resolution of the output slightly ($K - 1$ for both width and height). To prevent this, one can additionally pad the image with $K - 1$ zero values at the border before applying the convolution to preserve the original resolution. Padding the image so that the input and output resolution match is also called same-padding. Using no padding at all is called valid-padding.

The convolutional layer is optionally followed by a batch normalization [IS15] layer which has the purpose to make the training process of CNNs more stable. The batch normalization layer takes a batch of feature maps x as input and normalizes it by subtracting its mean value μ_x and dividing by its standard deviation σ_x.

$$x' = \frac{x - \mu_x}{\sigma_x}. \tag{3.2}$$

To also allow the model to change the range of the output x', the layer includes two trainable parameters γ and β which modify x' as

$$y = \gamma x' + \beta. \tag{3.3}$$

The final result y is then forwarded to the next layer which is usually a non-linear activation function. For its simplicity and effectiveness one of the common activation functions is the Rectified Linear Unit (ReLU). It is defined as

$$ReLU(x) = \max(x, 0) \tag{3.4}$$

and simply clamps the negative part of x to zero.

The last important component of the convolutional neural network is the pooling layer. The pooling layer typically reduces the image dimensions by applying a maximum or average function over a neighborhood of pixels (usually neighborhood of 2 pixels). The dimension reduction is achieved by not applying the function to every pixel but every n-th pixel. The stride hyperparameter controls thereby the "n" which is usually set to 2 for pooling. This parameter also exists for the convolutional layer alike. The max-pooling is more common than the average pooling since the gradient is easier to compute which makes the training process faster. The task of the pooling is to select the strongest activations from the feature maps which were returned by the convolutional layers. It is common practice to increase the channel dimension of the feature maps with convolutions while decreasing their resolution with pool layers. The idea is to extract features using convolutions and select the most important features by pooling. With increasing network depth we get a large number of low-resolution feature maps.

3.2 Encoder-Decoder Architectures

Encoder-decoder networks [BKC15] are a type of neural networks and are especially useful when needing pixel-wise predictions as such in the field of semantic segmentation. Also in our scenario, we want to make pixel-wise depth estimations which is why we are going to work with encoder-decoder type networks. An encoder-decoder system consists of an encoder and a decoder. In our case, the encoder and decoder are both CNNs. The idea of the encoder-decoder system is to first extract all features of the input image which are relevant for the prediction task and then reassemble the final output based on the extracted features. The decoder thereby reinterprets the encoder output and synthesizes the final prediction. The task of the encoder, on the other hand, is to produce the feature maps. It has a similar structure compared to the CNN described in the previous section. It repetitively applies convolutions and pooling layers to extract and select features. As a result, we get a compressed version of

our original input, a collection of many low-resolution feature maps. This compressed version is also called the encoding. It should describe the properties of interest of our image in a lower-dimensional space. For our purpose, we would like the encoding to include all relevant information concerning depth so mostly information about the circle of confusion. The decoder then receives this representation as input and restores the original image resolution while also reducing the number of channels to match the desired target dimension.

The decoder usually has a similar structure to the encoder. For each feature channel increasing convolution in the encoder, we have a convolution in the decoder reversing this increase. Thus, instead of increasing the feature channel dimension like a typical CNN the decoder decreases it. In this way, we can gain the original input channel dimension back. The decoder also needs to produce the original image resolution out of the smaller encoding. Thus for every pooling layer in the encoder, the decoder has an upsampling layer which increases the resolution again. The upsampling layer can either be fixed, like a bilinear upsampling or also trainable. A trainable upsampling layer is usually realized as a transposed convolution [DV16] layer with kernel size and stride set to 2. The transposed convolution performs a similar task compared to the standard convolution but in the backward direction. Since the convolution is a linear function, it can be formulated as one large convolution matrix to be multiplied by a flattened input image vector. The transposed convolution acts the same way but the matrix here has transposed dimensions. Thus it reverses the feature channel and resolution change done by a convolution. Applying a convolution and then a transposed convolution with the same hyperparameters to an input will result in an equally shaped output. A stride of 2 would usually half the input resolution when using a normal convolution whereas a transposed convolution will double it. As a result, we can counter each resolution bisecting max-pooling layer in the encoder by a stride 2 transposed convolution.

Another important concept of the encoder-decoder architecture is the concatenation skip connection [HLW16]. Due to the decrease of the input dimension by the encoder we loose information of the original image we might need for the decoder to produce the desired network output. A concatenation connection tries to solve this problem by forwarding an intermediate output of one layer not only to its successor but also to a far later layer in the network. We thus create a connection between layers which do not directly follow each other in the network structure. Skip connections allow outputs to be forwarded without further processing to a later layer meaning they do not additionally get shrunk by the encoder. Therefore we can counteract the loss of information which is happing as the input passes the network. The mechanism of a concatenation connection is visualized in Figure 3.1. In the graphic, the input x_{n-k} from Layer_{n-k} gets not only forwarded to the successive layer but also to the later layer Layer_{n+1} in form of concatenation with x_n from Layer_n. As a result, the Layer_{n+1} gets

more information and the ability to improve the performance of the network.

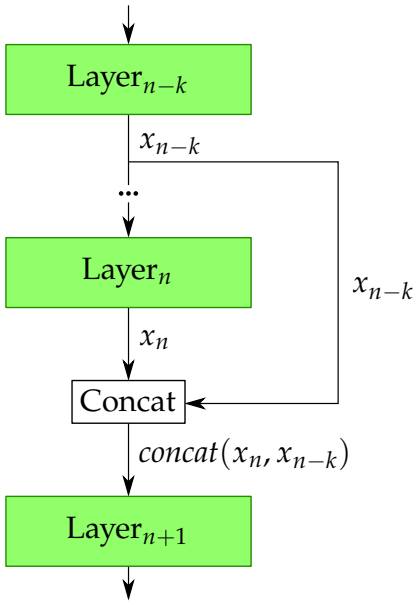
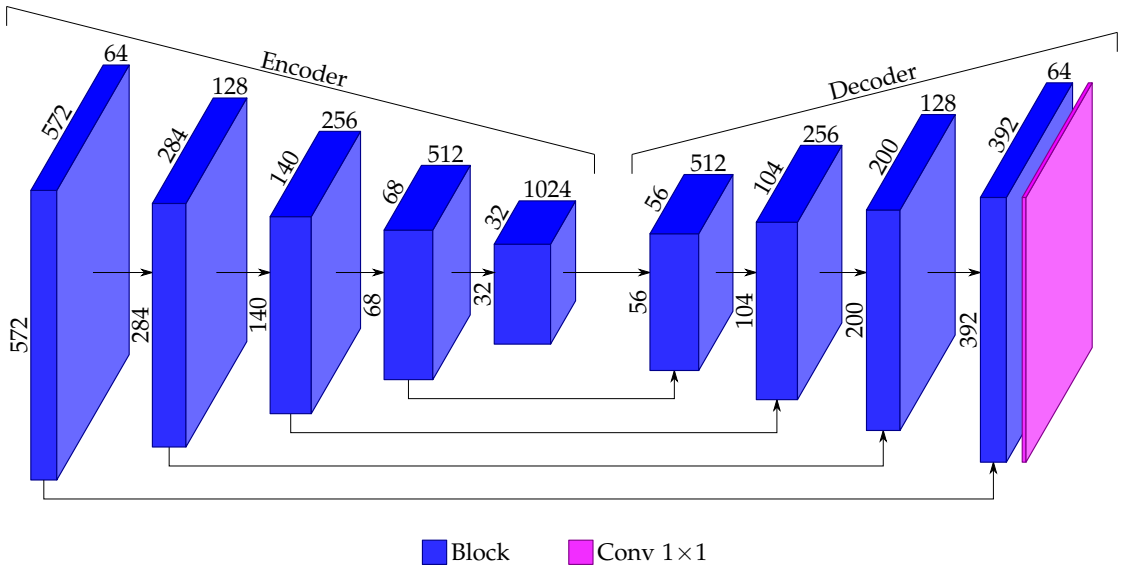


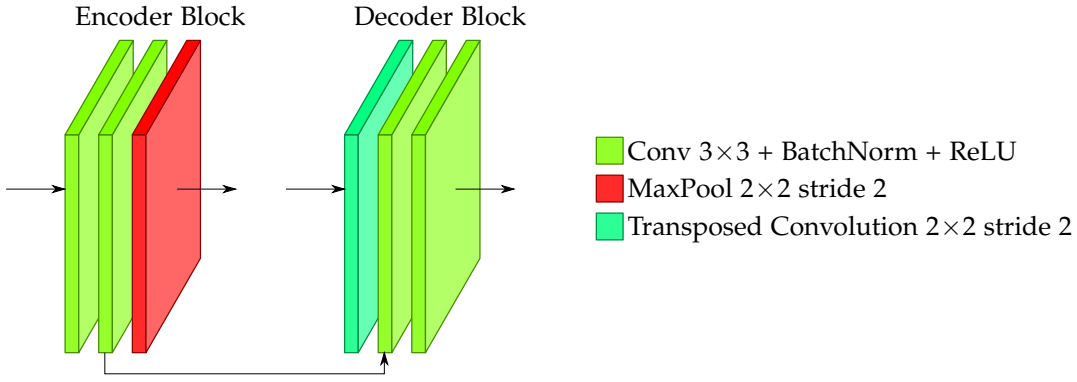
Figure 3.1: Concatenation skip connection.

When we talk about concatenation of two tensor batches $x_1 \in \mathbb{R}^{B \times C_1 \times H \times W}, x_2 \in \mathbb{R}^{B \times C_2 \times H \times W}$ we mean the creation of a new tensor $x_{cc} \in \mathbb{R}^{B \times (C_1 + C_2) \times H \times W}$ which is the result of concatenating the second axis of x_1 and x_2, the channel axis. B is hereby the batch size, C the number of channels, H the height and W the width of the feature map tensor. If the width or the height of two tensors does not match, it is common to either pad the smaller tensor or crop the larger tensor.

A popular, well-known encoder-decoder architecture is the U-Net [RFB15] which is used for image segmentation. It has been a basis for several subsequent CNNs. The architecture of the network is displayed in Figure 3.2a. The blue boxes in the image represent blocks of the encoder and decoder which themselves consist of several 3×3 convolutions, batch normalizations, and ReLU layers. Additionally, every encoder block except the last finishes with a maximum pooling and every decoder block starts with a transposed convolution. The number on the top of each box indicates how many feature channels the respective block produces. It is equal to the number of feature channels which are output by the last convolution in the block. The other two labels give the width and height of the feature maps inside the block. We thereby always refer to the resolution before a pool operation in the encoder block or after a transposed convolution in the decoder block. Blocks of the encoder and blocks of the decoder operating on a similar feature level are connected by concatenation connections,



(a) U-Net architecture.



(b) U-Net block structure.

Figure 3.2: U-Net.

indicated by the arrows beneath each block. When concatenating, the larger tensor from the encoder is cropped accordingly. Originally, the network inputs 572×572 grayscale images and yields a corresponding foreground-background segmentation estimation given by two channels of the resolution 392×392. Thereby the final layer of the CNN is a 1×1 convolution which reduces the feature channels to two channels. Since the convolutions inside the network do not use padding, some resolution is always lost after each convolution. As a consequence, the output resolution of the CNN is smaller than the input resolution. Figure 3.2b additionally shows how an encoder

and a decoder block is composed in detail.

3.3 Video Depth Estimation Architectures

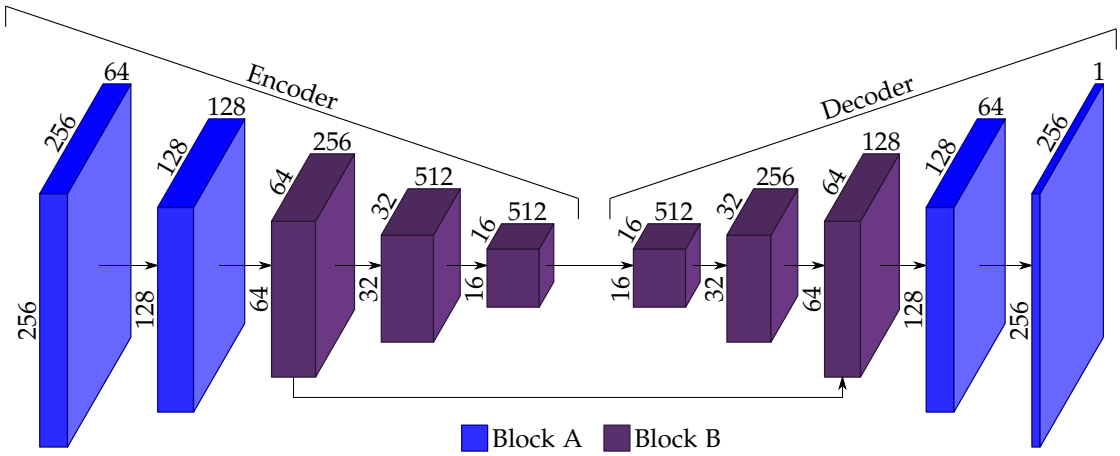
The above described U-Net inputs single images and outputs a pixel-wise segmentation of it. We also want to use CNN encoder-decoders to make pixel-wise predictions but for a whole stack of images. Thus we need to adjust the architecture to be able to process multiple images at once and output a single depth frame. Our goal is to use the varying defocus blur present in the input stacks as the basis for depth prediction. Additionally, the network should be able to deal with movement during the image sequence. In the following, we propose several different architectures which have the potential to execute our desired task.

3.3.1 LSTM-DDFFNet

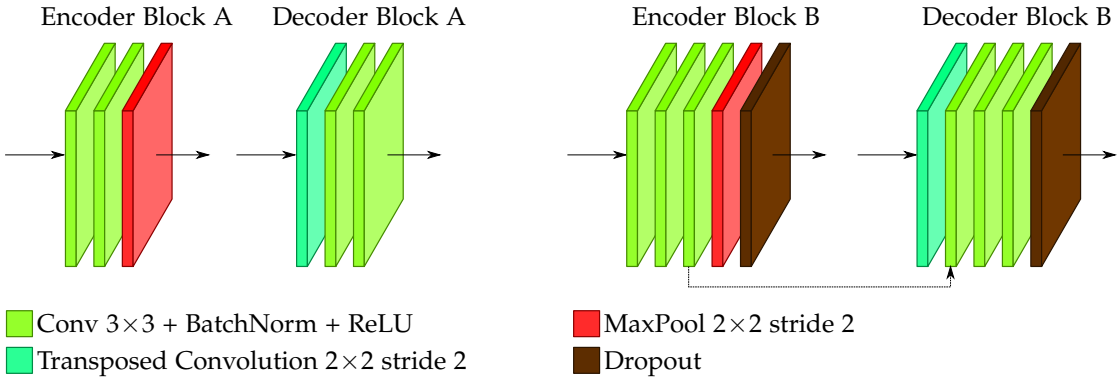
The first architecture we are going to use is based on the DDFFNet [Haz+18]. The original DDFFNet is an encoder-decoder system which uses a pre-trained VGG16 [SZ14] as encoder and a respective mirrored VGG16 as decoder. VGG16 is an image classifier CNN consisting of a fully convolution part followed by several fully connected layers returning a final 1000-component vector containing the computed class probabilities. The DDFF encoder uses only the fully convolutional fraction of the VGG. The network is already trained on the large ImageNet dataset. By using these weights as initialization for new, different prediction tasks it has been shown that CNNs can more easily generalize to other tasks while requiring significantly fewer data. This technique is also called transfer learning [Tan+18]. DDFF thus makes use of the fact that VGG16 is already able to reliably detect objects and object borders and uses that as a starting point for depth prediction.

An overview of the DDFFNet architecture can be seen in Figure 3.3. The network (Figure 3.3a) consists of two different kinds of blocks which are displayed in detail in Figure 3.3b. They differ in the fact that the second block type has one additional convolutional layer. The DDFF, as well as the VGG, use 3×3 convolutions, maximum pooling layers and ReLU activation functions. Differing from the original VGG, the DDFF uses batch normalization after each convolution as well as dropout layers at the end of the last three encoder blocks and the first three decoder blocks. Dropout [Hin+12] is a way of adding regularization to the training process. It randomly leaves out a portion of the output from the previous layer at a certain probability (usually 50%). Dropout can reduce overfitting and improve the model's generalization power. The pre-trained VGG is originally focused on object shapes which are not sufficient to infer depth. Since the goal is later to move away from mere shape memorizing, the

use of dropout is a reasonable choice. The decoder blocks are similar to the encoder blocks except instead of a final pooling layer they have a transposed convolution at the beginning. The CNN uses one skip connection from the third encoder block to the third decoder block.



(a) DDFFNet architecture.



(b) DDFFNet blocks.

Figure 3.3: DDFFNet.

This network itself again is only able to input a single image. The way the DDFF approach handles focus stacks of multiple images is displayed in Figure 3.4. In particular, each frame of the stack if first processed independently by the encoder-decoder. The output for each frame is then concatenated in the channel dimension (which is only a single channel) and finally fed to a scoring 1×1 convolution. The convolution outputs only a single channel which contains the depth map predictions for each pixel. This means that at first, for each image in the stack, the DDFF encoder-

decoder generates a preliminary depth prediction. These single predictions are then scored using the final convolution producing the final output. Consequently, the final convolutional layer has a high amount of influence since it is the only instance where the images are not treated independently. Another fact about this structure is that all input images are processed with the same importance whereby the temporal information in the video may not be optimally used.

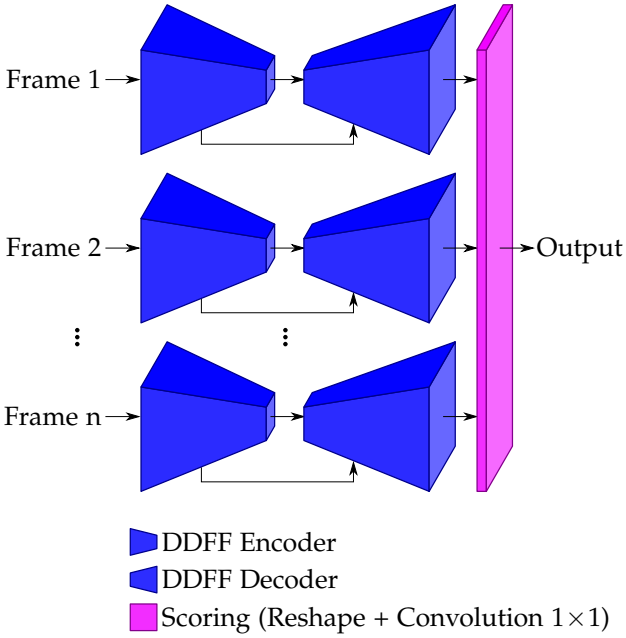


Figure 3.4: DDFF multi frame input mechanism.

We want to improve the DDFF architecture to also consider the temporal ordering of the video frames. A common way of learning temporal information is the use of a recurrent neural network (RNN). In particular, we are going to use a more advanced version of the RNN, the Long Short Term Memory (LSTM) [HS97]. LSTMs can memorize data which was previously given to them and use this knowledge in joint with the current input to produce the output. Every time a new input is processed by an LSTM, its state is updated. The memory state of the LSTM is also called its hidden state. LSTMs are thus useful when dealing with sequential data where a prediction should always depend on previous input.

However, unlike convolutional layers, LSTMs can only input 1D vectors with a limited dimension. It is thus also not possible to simply stack each row of an image since the total dimension would be way too large. We, therefore, propose to use the LSTM as a module between the encoder and the decoder of the DDFFNet and only give

the encoding to the LSTM which has a way lower total size. The LSTM will thereby be successively fed with each image encoding. The decoder finally is only applied once when the last frame is input and the LSTM already saw all input. Since there is only one decoder output, no scoring convolution is required anymore. An overview of our enhanced architecture can be seen in Figure 3.5a. In this setup, even though the encoding dimension is already pretty low, it is still not sufficiently small for an LSTM. The scale of the preferred input sizes usually is limited to a few thousand. The flattened embedding would thus still be too large for the LSTM. To solve this issue we come up with two methods. One is to first flatten the embedding, use a fully connected layer to further reduce the dimension and then feed it to the LSTM module. To format the output of the LSTM properly for the decoder we again apply a fully connected layer to restore the original dimension and then unflatten it to match the original feature map shape. Figure 3.5b shows this procedure when inputting an encoding of 512 channel with 8×8 feature maps. The visualization only shows the processing of one input. The LSTM procedure though is the same for every input. The second option is to only flatten the height and width dimension of the feature maps and keep the channel dimension unchanged. Thus we have a vector of flattened feature maps. We then give each of these channels to a low dimensional LSTM. We do this alike for each input image and feed all to the same LSTM. The output then will finally be unflattened again to match the original feature channel dimensions. Since we are inputting a very long sequence to the LSTM (every feature map channel for every frame) we decide to use a double-layered LSTM. The idea behind this is related to the use of a double-layered LSTM when learning from texts. There the first layer should analyze the relationship between single words while the second one should analyze the interplay of whole sentences. If we transfer this to our problem statement, the first LSTM layer should be responsible to analyze the pixels of each image and the second one should infer from the complete images in the sequence. Figure 3.5c displays the idea of the second approach by again inputting an encoding like in Figure 3.5b.

The original DDFFNet has one skip concatenation connection from the encoder to the decoder. In our modified LSTM-DDFF version we have multiple encoder outputs but only one decoder output. Thus we need to somehow map the feature maps produced by the encoder to match the dimensions expected by the decoder. We again found two different options to do so. The most simple one is to just transfer the feature maps from the last encoder only and ignore the previous encoders. Another way though is to concatenate all feature maps for the skip connection and score them using a convolutional layer to get the desired shape. This scoring mechanism is similar to the 1×1 convolution scoring layer at the end of the original DDFF (Figure 3.4).

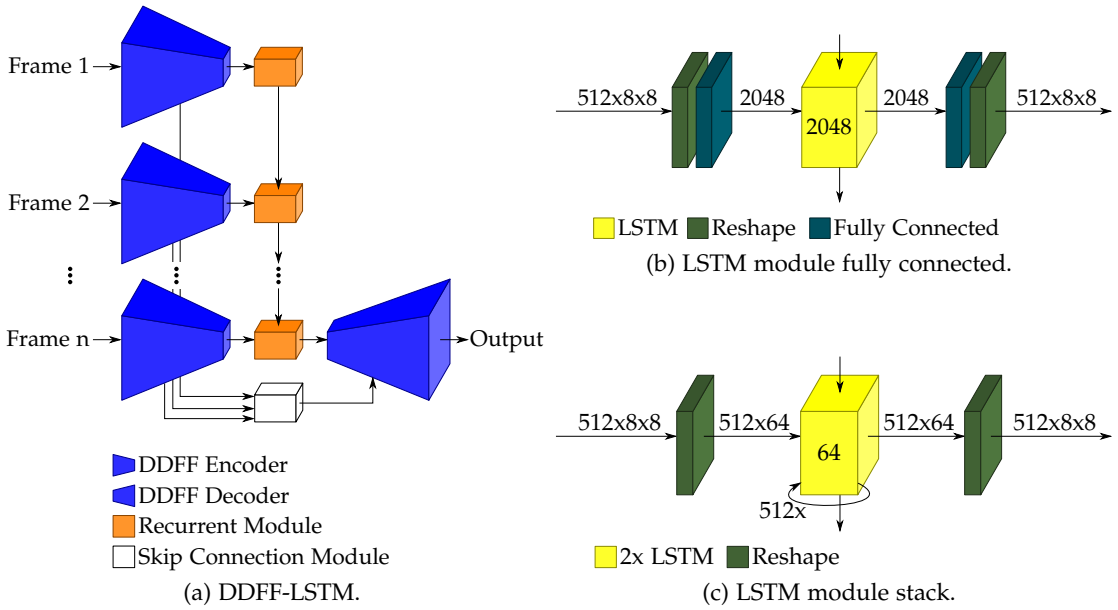


Figure 3.5: DDFF-LSTM.

3.3.2 PoolNet

Another possible architecture to predict depth from varying blur video sequences are Permutation Invariant Convolutional Neural Networks [AD18]. Their original purpose is to produce sharp and noise-free images from a sequence of blurry and noisy images. The practical application is to synthesize a high-quality picture even though the recording device was influenced by camera shake which is an issue especially with small lenses like in mobile phone cameras. This seems to be a promising architecture for our needs since we too are dealing with objects which might have moved during the video recording and want to associate them to each other to estimate their amount of blur. The neural network can input an arbitrary number of images while not distinguishing in what order the images are presented to the network. This is reasonable for their aim of synthesizing one image from several recordings since the sequential order is not a factor for the reconstruction process.

The architecture itself is similar to a classical encoder-decoder architecture like the U-Net and the DDFFNet. There is an encoder and a decoder processing the inputs. Though unlike the DDFFNet which processes the inputs mostly independently (except the last layer), this CNN has a way to exchange information between the encoder-decoder input streams. In particular, it uses global max-pooling layers which are placed

between the encoder and decoder blocks. The max-pooling layers thereby input the feature maps from all streams and compute a set of global maximum feature maps. The next step for each stream is the concatenation of its individual feature maps and the global maximum feature maps which is then finally forwarded to the next layer. Because of its characteristic global max-pooling layers, we will refer to this network type as PoolNet.

In Figure 3.6 you can see an example of a PoolNet system for two inputs. In this model, the first layer of both input streams output a three-channel feature map where x_1 refer to the maps for the first stream and x_2 for the second stream. These are then forwarded to a global pool layer as well as to a concatenation node placed before the next layer. The pooling layer computes the maximum of the two input feature maps $\max(x_1, x_2)$ and also forwards it to the concatenation node of both streams. Each stream, therefore, receives global information in the form of the maximum feature maps. At the concatenation node, the two inputs are finally concatenated in the channel dimension and transferred to the next layer.

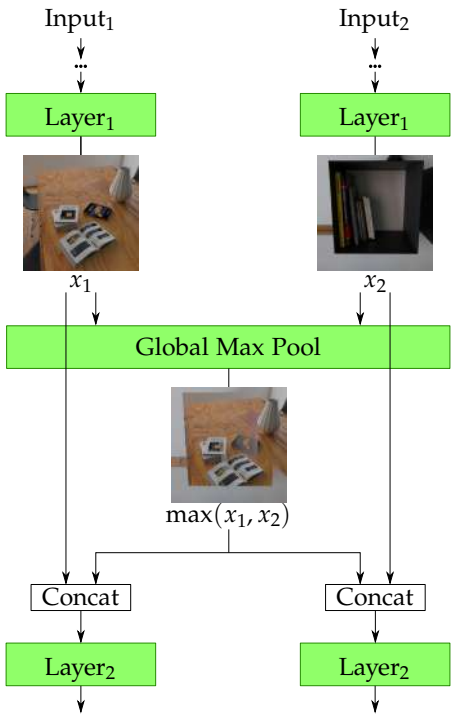
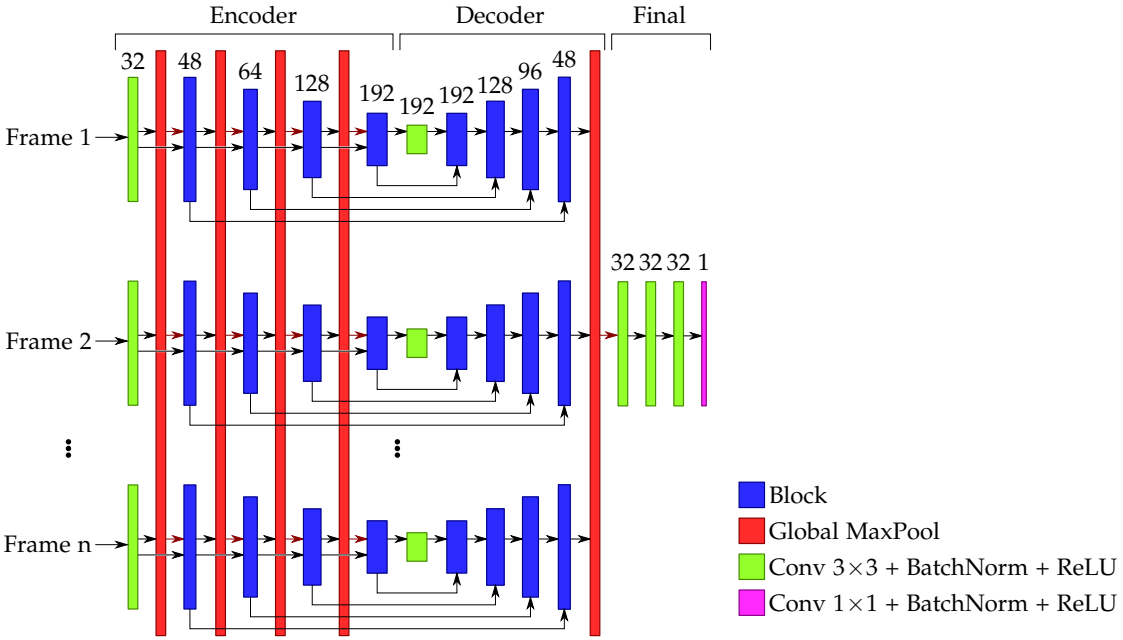


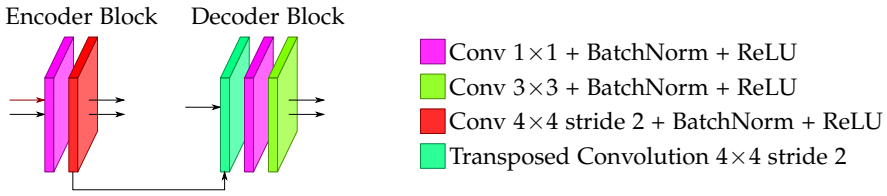
Figure 3.6: PoolNet global max pool.

The global pool layer thus enables the individual streams to exchange information. Since the maximum function is commutative, the order of the inputs itself is irrelevant.

Derived from the original PoolNet, we present our network structure in Figure 3.7a. The large layers between the blocks hereby indicate a global maximum pool layer. The numbers above the blocks and layers represent the number of feature channels at this stage. The detailed structure of one block can be seen in Figure 3.7b. As usual,



(a) PoolNet architecture.



(b) PoolNet block.

Figure 3.7: PoolNet.

we have skip connections from the encoder to the decoder. However, different from typical encoder-decoder architectures, the PoolNet does not use regular max-pooling layers to reduce the feature map dimensions but a 4×4 convolution with stride 2 reducing the resolution alike. Also, after the decoder, there is a global pooling layer reducing the result of all input streams to one. If this last pooling layer would be omitted, we would receive an output for each input. Additionally, a final small CNN is giving the model more control over the final output. Compared to the original

PoolNet we use fewer feature channels which is sufficient for depth estimation and also introduce a batch normalization layer after each convolution to improve convergence. Furthermore, we do not use global pooling layers in the decoder, except for the final pooling since we demand a single output. The decoder should thereby process the encoding mostly independently whereas the task of the encoder is to determine the CoC change relations among the frames. Thus the encoding is already required to contain all CoC information which finally gets expanded independently by the decoder.

3.3.3 Recurrent Autoencoder

The last architecture we are introducing is the recurrent autoencoder [Cha+17]. Strictly speaking, this network is not an autoencoder [GBC16] but an encoder-decoder system, which can be seen as an autoencoder-like structure. The original use case of the network is to denoise rendered image sequences. Therefore, like the PoolNet, this architecture is also able to natively input several images. A major design aspect of the network is that it is very lightweight with a low number of trainable parameters making inference very fast. This is necessary to compete with other denoising methods in the field of rendering where performance is always a key factor.

The special aspect of the recurrent autoencoder is that it also keeps a hidden state storing previously input information like an LSTM while being able to input multi-channel 2D images at the same time. The network is though still fully convolutional meaning no fully connected layers are used. Also, no image flatting is performed by the CNN. This feature is implemented by using recurrent concatenation skip connections from one block to itself. The general idea can be seen in Figure 3.8. The feature maps which were output from one specific block layer (layer_3) for the previous input_{n-1} are stored and then reused for the next input_n. In particular, for the subsequent input_n at a stage (layer_2) in the same block, these previous feature maps are concatenated with the current feature maps. The hidden state of the recurrent autoencoder is thus composed of the previously output feature maps from all recurrent blocks. For the first input, this state will be initialized with zeros.

The original recurrent autoencoder architecture implements the recurrent connections exactly as in Figure 3.8. The feature maps from the third layer of one encoder block are stored and then, for the next input, concatenated with the feature maps of the first block layer. Thus the second block layer receives both information from the past as well as from the current input in the form of feature maps. This enables the network to memorize previous input data while not implying any restrictions for the length of the input sequence. Therefore, in theory, arbitrary long sequences could be memorized by the model. In practice though too long input sequences are likely to be forgotten and the last inputs contribute more to the final output than the first inputs. The last

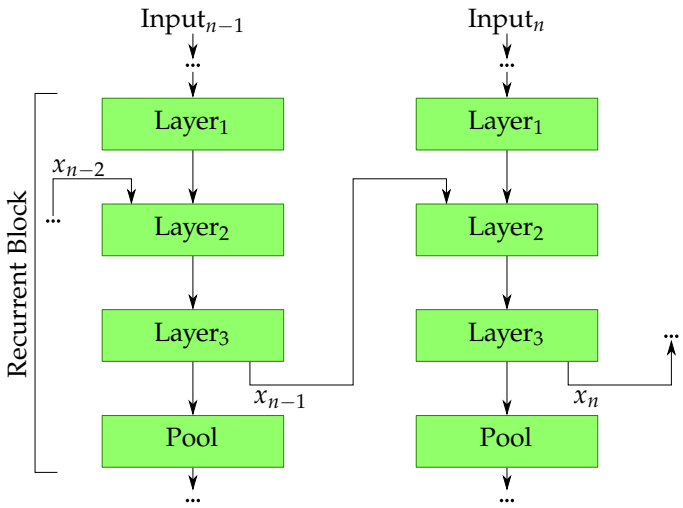
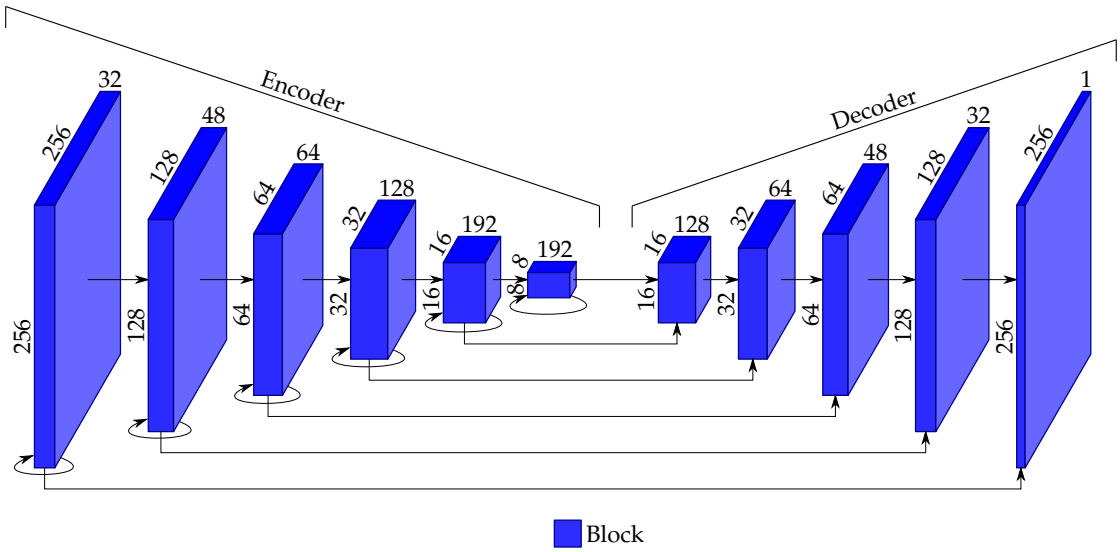


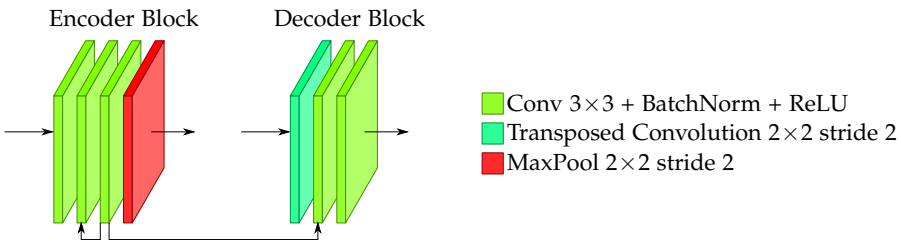
Figure 3.8: Recurrent concatenation connection.

input is going to have the largest contribution. Since the recurrent skip connections only connect blocks for direct successive inputs, the recurrent autoencoder is also more likely focused on comparing successively fed image pairs with each other than comparing frames having a long temporal distance between each other. This might be advantageous in our case since consecutive frames have more moderate focus distance and object position changes than frames which are wider apart. Similar properties hold for our LSTM-DDFF structure. The PoolNet structure though, for instance, treats all images equally and does not distinguish how far frames are apart in terms of temporal distance. The LSTM-DDFFNet and the recurrent autoencoder consider the order in which the inputs are presented to the model which is suitable for video sequences. The recurrent autoencoder additionally does not need to reduce the encoding dimension to a very low amount which is necessary for the LSTM-DDFFNet structure. We can thus preserve more information while still having the recurrent network properties of a classical RNN.

Our derived recurrent autoencoder CNN structure can be seen in Figure 3.9a and its block architecture in Figure 3.9b. We also only use recurrent skip connections for the encoder blocks like the original. These are indicated by the circular arrows in the graphic. We choose to use more feature channels than its original since inference speed is not that relevant for us and we want to focus more on the output accuracy. Also, we will use batch normalization after each convolution like in our previous neural networks. The original recurrent autoencoder uses simple nearest neighbor upsampling layers to gain back resolution in the decoder blocks. We will, however, use transposed convolutions instead to make the upscaling process itself also learnable.



(a) Recurrent AE architecture.



(b) Recurrent AE block.

Figure 3.9: Recurrent autoencoder.

3.3.4 Architecture Extensions

Finally, we are going to introduce several general concepts to further extend our already existing architectures. One way we will modify our networks is to incorporate the role of the circle of confusion for the depth prediction process explicitly to provide additional supervision and control. Furthermore, we want to support our neural networks with the known focus distance for each image as an extra parameter to better generalize to different camera types.

3.3.4.1 Two-Decoder Architectures and Consecutive Architectures

As our goal is to predict depth based on the defocus blur we would like to incorporate the circle of confusion as an explicit component in our approach. For this purpose,

we adjust our previous networks to not only produce a depth map but also a circle of confusion map. The CoC map will thereby contain an estimate of the CoC diameter for each pixel. Our loss function will then also include supervision on this CoC map as well as on the depth map.

We come up with two ways of explicitly making use of the CoC, namely the use of a Two-Decoder architecture and the use of two consecutive networks. Our Two-Decoder approach extends an already existing encoder-decoder system by an additional decoder with its own weights. The output of the second decoder will then be supervised accordingly to output CoC map predictions. Figure 3.10 illustrates our idea. The

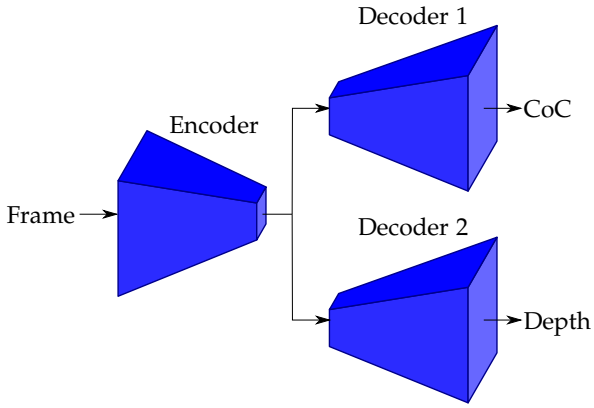


Figure 3.10: Two-Decoder architecture.

encoding output of the encoder is simply forwarded to two different decoders instead of only forwarding it to a single decoder. One decoder will be responsible for the CoC and one for the depth map. Skip connections are created for both decoders alike.

The second option is the use of two neural networks in sequence (Figure 3.11). The first CNN thereby should predict a CoC map for each input image. We can use the

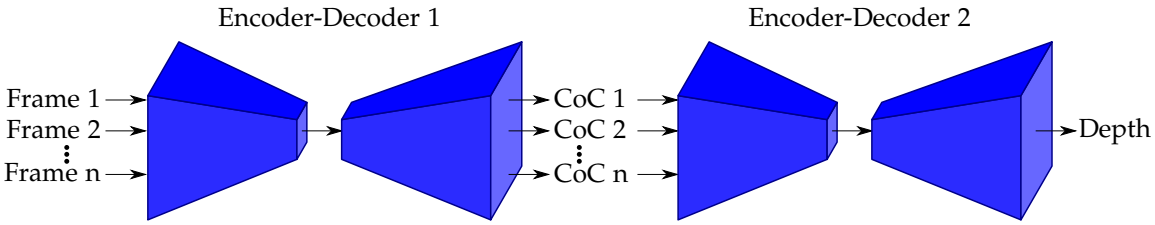


Figure 3.11: Consecutive architecture.

PoolNet architecture and omit the last global pooling layer after the decoder for this purpose. Finally, the second CNN should input the previously predicted CoC maps

and produce one estimate for the depth using the estimated CoC. The task of the second network is thus to solve Equation 1.1 (or Equation 1.3) for the depth D given the predicted CoC c.

3.3.4.2 Passing the Focus Distance

We also want to experiment with giving the focus distance F as an additional camera parameter to our neural networks so that the model does not need to estimate it implicitly. We thus want to relieve the model from the responsibility to also detect with which camera parameters an image was captured. The focus distance F seems to be a good choice since it is only a single scalar value which varies for each picture in the focal stack. To give a single scalar to our model we add an additional constant channel per input frame which is filled only by this single value. By passing this extra parameter in combination with our architecture extensions involving the prediction of a CoC map as another step (section 3.3.4.1), the neural network should be additionally assisted in retrieving D from Equation 1.1 since an additional component (parameter F) is known. Another advantage of giving the focus distance for each image is to notify our order invariant architectures such as the PoolNet about the capture order. The focus distance gives these models a way to identify and distinguish the input images by their recording order.

4 Dataset

For training and testing, we require a dataset consisting of video frames with moving focus and corresponding ground truth depth (at least for training). However, there does not seem to be an already existing dataset which would fulfill all our requirements. Thus we decide to create suitable datasets ourself. We thereby use two different techniques to do so. The first is to acquire data from the real world using a normal camera for the color images and an RGB-D sensor for ground truth depth. The second one is to create a dataset synthetically by using the 3D graphics program Blender [Com17]. With this software, we can render arbitrary scenes with arbitrary lens settings and generate exact depth ground truth.

4.1 Real-World Dataset

Creating a real-world dataset to fulfill our purposes is a complex process. We need video frames with constantly changing lens focus along with aligned and time synced depth ground truth. For this aim, we developed two Android applications which automatically change the lens focus while capturing pictures. We then use an Android mobile phone with an RGB-D sensor mounted on top of it to record data with ground truth depth.

4.1.1 Recording RGB Focus Stacks

4.1.1.1 Android and DSLR Camera Comparison

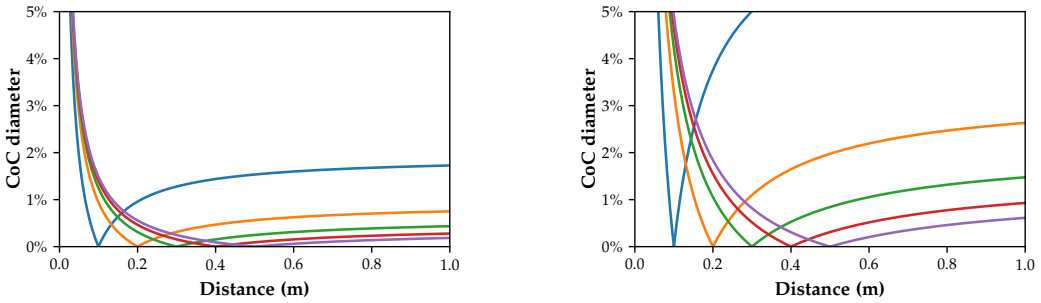
The advantage of using an Android phone when recording the data is that the focus distance can be set via software and the lens will automatically adjust itself. If we would use a typical DSLR camera we would have to manually turn the lens to change the focus which introduces small camera shake and thus may yield worse image results. Controlling the lens focus by software on the other hand, as it is possible with Android phones, should not cause any shake. Another point is that it is easily controllable at what focus distance we want to shoot at a certain time when using an Android application. Plus, the exact focus distance, at which a frame was captured, can be retrieved from the camera. By using a DSLR camera one would have to manually alter

the focus every time while the camera is filming which does not yield reliable focus distances. Also, when recording several videos it is hard to achieve the same focus distances in all recordings. One would need a lot of manual skill to always rotate the lens at the same speed. Having inconsistent focus distances for each sweep may introduce additional difficulties for our depth estimation methods.

A drawback however when using a smartphone camera for capturing the dataset is that these cameras usually possess small lenses with a fixed focal length. This leads to a large depth of field, meaning the range where objects will appear almost sharp is very large. The depth of field gets larger the greater the focus distance is. Meaning if we only consider focusing on small objects close to the camera we can mitigate this effect to a certain degree. DSLRs however usually have separate and configurable lenses which are a lot larger than in smartphone cameras. Thus DSLR cameras with an appropriate lens can also achieve a shallow depth of field on even larger focus distances.

We can see this effect by looking at Figure 4.1. There are two plots showing the relationship between the distance and the circle of confusion according to Equation 1.1, one for the Samsung Galaxy S7 camera and one for a typical DSLR camera. We plot five graphs for focus distances $F \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$ in meters. There are five graphs per plot each having CoC zero in their respective focus distance (blue, orange, green, red, violet). Instead of plotting the absolute CoC value in meters we divide it by the sensor size to get a relative CoC value giving how large the circle of confusion is compared to the image size. For the S7 we can see that the CoC is lower in general compared to the DSLR. Also, for the larger focus distances (e.g. the violet graph for 0.5) we can see that the graph is very flat and close to zero CoC after the root meaning that even if objects lie behind the focus distance they appear sufficiently sharp on the image and it might be hard for our approach to distinguish the CoC in that case. In general with the S7 the graphs are very flat after 0.5 meters resulting in potentially indistinguishable distances. With the DSLR however, this effect is present in a way lower degree which shows in the form of steeper graphs resulting in larger and better distinguishable CoC.

In the end, however, we choose to go with the Android phone instead of a DSLR. Automatic refocusing without shake seems more crucial than the ability to record scenes with decent blur for larger distances. We will focus on recording datasets where objects are very close to the lens so we can mitigate the large depth of field of the mobile phone camera. Eventually, we developed two different Android applications for recording videos while automatically altering the focus distance where one has the purpose to record longer video sequences at a high FPS rate while the other one only shoots a limited number of frames in a burst manner. Figure 4.2 thereby shows a screenshot for both programs. The first application is based on the Camera2Video [Goo19b] example code and can record videos while adjusting the focus distance. The current focus distance to capture is thereby given by a linear function based on the current recording



(a) Samsung Galaxy S7 camera CoC.

(b) DSLR camera CoC.

Figure 4.1: Comparison of the circle of confusion when using a mobile phone camera and a DSLR camera.

time. The start and the end of the desired focus ramp can thereby be set arbitrary. Since we have a small lens device we want to ramp from 0.1 m to 0.5 m for instance. However, since the lens is mechanical in the end and needs a certain time to change its focus it is not guaranteed that the lens is in the correct position when shooting frames. The video recorder does not wait for the lens to adjust its focus while recording frames, it just shoots at a certain frame rate, e.g. 30 FPS. After the video is recorded it will be saved as MP4 file along with a JSON file containing information about the frames such as focus distances and timestamps.

Our second application is derived from the Camera2Basic [Goo19a] example project and shoots a small number of single pictures in a burst manner. Unlike the other program, this one requires the lens to be in position to shoot the next frame. As a result, we get frames which were captured exactly at the desired focus distance. In the end, this application is more accurate in terms of achieving the exact desired focus stack but yields fewer frames than the video recording program at a lower frame rate. The sequence hereby is stored as JPG images again including a JSON file for storing information about the frames.

The video recording application can capture sequences at a higher frame rate resulting in less potential movement between the frames. When capturing bursts, on the other hand, the capture delay between two consecutive images is higher and depends on the image format specified for the camera. The Samsung Galaxy S7 camera internally operates on the YUV image format. Thus to achieve the highest possible frame rate we would have to record all images as YUV images and also store them in the YUV color space. Converting them to RGB would have to be done separately after the video is recorded. By doing so the camera can capture one frame every 40 milliseconds in burst

(a) Video recording application.

(b) Burst recording application.

Figure 4.2: Android applications for focus stack recording.

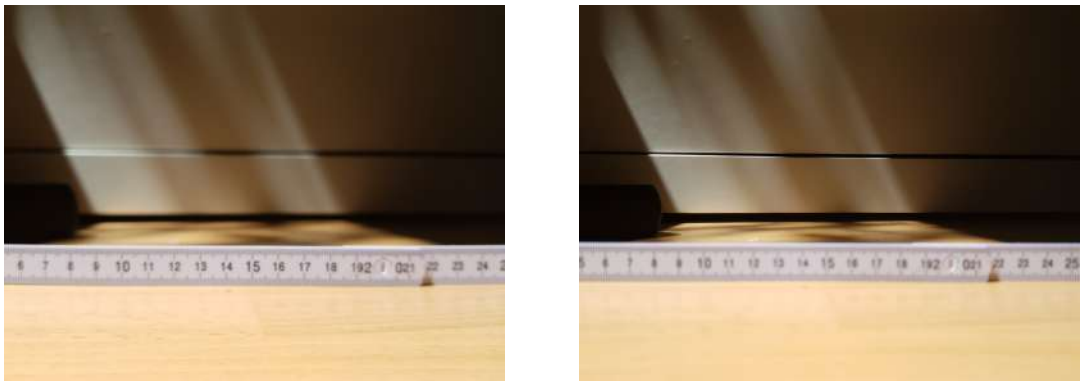
mode which corresponds to an effective frame rate of 25 frames per second which is almost as fast as recording a standard video. When recording sequences in RGB mode, however, we experience that the first two images have also only a similar delay of 40 ms, though the third frame has a way higher delay of 120 to 200 ms. The reason for this might be that the camera is busy with converting the captured images in the buffer to RGB and has to wait for this process to finish to store new captures in the buffer queue.

4.1.1.2 Focus Breathing

Focus breathing or lens breathing is the slight change of the field of view when changing the focus distance. This effect is more severe when changing the focus distance by a very large distance, e.g. by focusing on a very close object and then focus to infinity. As a result, we have a slight magnification effect when decreasing the focus distance. In Figure 4.3 we can see two pictures taken with a Samsung Galaxy S7 at the same camera position of the same scene. The only thing that is changed is the focus distance which is closer (0.1 meter) on the left image than on the right (0.45 meter). As we can see, even though we only increase the focus distance slightly, the field of view is also altered. E.g. the 25 m mark of the ruler is cut off on the left whereas in the right capture it is fully visible due to the slight increase of the field of view.

In what amount focus breathing is noticeable depends on the specific lens. Cheaper

cameras, like smartphone cameras, tend to have a large magnification effect when changing the focus. However even with more expensive DSLR cameras, the effect is usually clearly noticeable. When shooting still images, focus breathing is usually irrelevant for the photographer. When recording videos on the other hand this issue plays a more important role since it is usually desirable to change the focus while recording without affecting the field of view. Also for our approach, this might interfere with our depth estimation process since this distortion introduces additional change in the frames. However, later tests show that the CNNs are nevertheless able to produce proper predictions from focus-breathing contaminated data. Thus a special mechanism to remove the focus breathing is not necessary.



(a) Scene captured at focus 0.1 m.

(b) Scene captured at focus 0.45 m.

Figure 4.3: Focus breathing.

We also notice another problem related to focus breathing. When capturing bursts at the maximum speed of 40 ms with our Android application, we experience a kind of motion blur effect (Figure 4.4) even though the phone was fixed on the ground. We reduce this problem to the fact that the high capture speed leads to situations where the lens of the camera is still moving to match the requested focus distance while the camera shutter is already open capturing the next frame. We notice that this effect is especially occurring in the first few frames and most noticeable in the second captured frame where the CoC changes the most. This issue is very undesirable since it distorts the whole image result. As a workaround, we capture every frame twice per focus distance and then discard the first shot frame and only save the second frame. This doubles the capture delay but eliminates the motion blur caused by focus breathing on our final images. In particular, using this workaround gives the camera lens more time to be in the right focus distance position before the shutter opens to capture the next frame. As a result, the motion blur effect disappears on our captures.

Figure 4.4: Motion blur due to the field of view change caused by focus breathing. The frame was recorded with a Samsung Galaxy S7.

4.1.2 RGB-D Sensor for Depth Ground Truth

To use the captured RGB sequences as training data we need to first retrieve ground truth depth data for each frame. The key here is to acquire depth maps that are as exact as possible since our models will later use this data for learning how to estimate depth themselves. One way to capture depth from the real world is to use an RGB-D sensor [GVS18]. RGB-D sensors capture not only RGB image sequences like regular cameras but also additionally produce matching depth maps. This is why they are called RGB-D cameras, where the D is short for depth. The first popular RGB-D sensor is the Microsoft Kinect which was released for the gaming console Xbox 360 allowing games to use human motion as input source e.g. for character movement. The Kinect was groundbreaking in the field of depth perception and very popular among users but it also has its downsides like its large size and the requirement for an additional AC-DC power supply. A less known but more compact camera, which only requires a USB connection, was later released by Asus, the Asus Xtion Live Pro. Its technological basis is very similar to the Kinect. This camera is additionally easily accessible on a variety of operating systems with the help of the OpenNI driver allowing developers to acquire the color and depth stream of the camera and use e.g. OpenCV [Bra00] to further process the data. Later on, there was a follow-up model for the Kinect released by Microsoft which uses new techniques to capture more accurate depth. At this time also Intel released a series of RGB-D cameras under the RealSense (RS) technology which came with their own open-source driver.

We think that RGB-D cameras are a convenient and promising way to acquire appropriate ground truth depth which we can later use for supervised deep learning. As mentioned above these devices are also capable of recording RGB pictures as well so

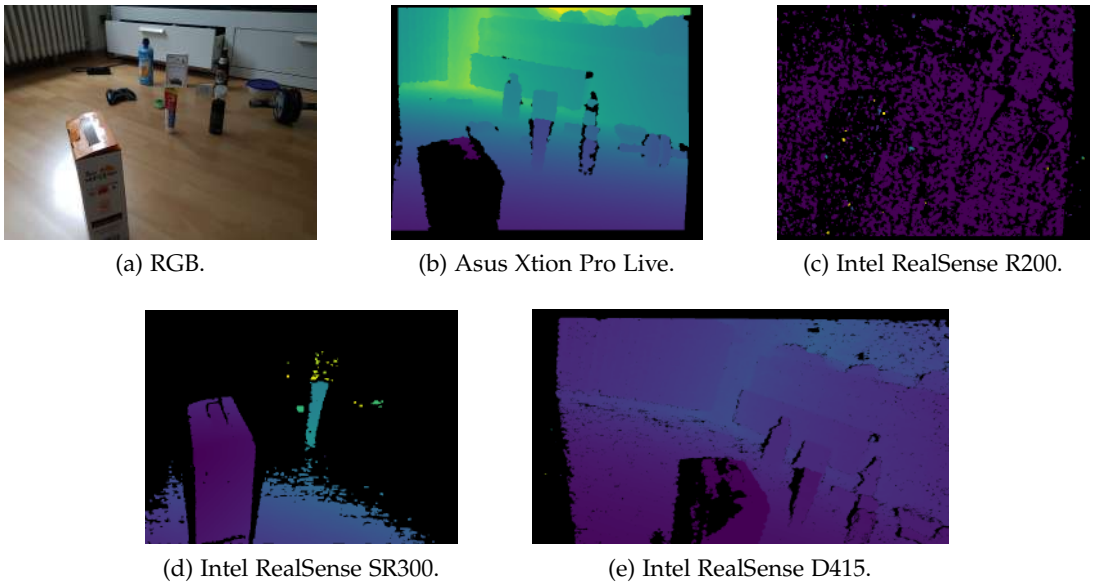
one may ask why we need an Android phone to record pictures instead of only using the RGB-D devices to capture both depth and color. The problem, however, is that these devices usually use a very cheap, low resolution and low-quality color sensor which additionally has a fixed focus meaning it is not possible to use it with our approach. That is why we use the Android camera to record the color information and the RGB-D sensor to only record the matching ground truth depth information.

Nevertheless, RGB-D cameras still have appropriate properties to fulfill our needs of retrieving accurate depth frames which leads us to experiment with a whole variety of depth cameras to determine the best option. The RGB-D sensors which are accessible to us are the Asus Xtion Live Pro and the Intel RealSense models R200, SR300 and D415. In Table 4.1 you can see a comparison of the specifications of the various models. We leave out the color sensor specifications since it is irrelevant for our purpose and only list the depth sensor specifications.

Table 4.1: A comparison of different RGB-D sensors.

	Asus Xtion	RS R200	RS SR300	RS D415
Resolution	640×480	480×360	640×480	1280×720
FPS	30	60	60	Up to 90
Distance	0.8 - 3.5 m	0.5 - 3.5 m	0.2 - 1.2 m	0.3 - 10 m
Driver	OpenNI	librealsense	librealsense	librealsense 2.0
Release	2011	2015	2016	2018

In Figure 4.5 you can see some captures taken with different RGB-D sensors. The depth is normalized and visualized via a colormap where blue indicates near objects and yellow far ones. RGB-D sensors usually are not able to acquire depth for all pixels because of distance restrictions, occlusions or reflections on objects. These pixels are marked as black. We can observe that even though the Asus Xtion was released before all others, it still produces the best depth map. The only drawback is that it does not work for near objects like the box in the front in our example. Apart from that, the depth map is very smooth and mostly all pixels are assigned a depth value. The R200 produces very bad results in our test and the SR300 only works for very close distances. The Intel D415 provides the highest resolution output though the depth map contains lots of small unknown depth spots. Also, it is very unstable and was flickering a lot during the recording. Additionally, sometimes the D415 produces very large values for a pixel without indicating it as unknown. In the end, we choose the Asus Xtion since it has the overall best performance in our opinion.



(a) RGB. (b) Asus Xtion Pro Live. (c) Intel RealSense R200.

(d) Intel RealSense SR300. (e) Intel RealSense D415.

Figure 4.5: Depth maps captured with different RGB-D cameras.

4.1.3 Synchronization and Registration of RGB and Depth Images

Now we can collect the necessary data needed for training the neural networks. However, one open issue is still that when shooting sequences with the Android camera and the RGB-D depth sensor, the output frames will not match each other exactly since the camera pose will be slightly different. As there is always some displacement between both sensors we get two frames shot from a slightly different position and angle. The closer we can physically align both lenses to each other the better the frames will match in the end. However, there will always be some gap resulting in imperfect data. This is very undesirable since we need the input data and its ground truth to be as exact as possible. Especially, the outline of each object on the color frame should match with that on the depth frame. Otherwise, the network may not learn that object borders are potential clues for depth changes. Overall the network will learn to predict depth which is shifted like the original offset of the two sensors.

Thus it is necessary to align one image to another. To avoid distortion of the RGB image we will leave the RGB image untouched and modify the depth information only. The process of aligning images to each other in stereo vision is called image registration. Specifically, since we are dealing with depth maps here, aligning a depth map to a color map is called depth registration [Tan+16].

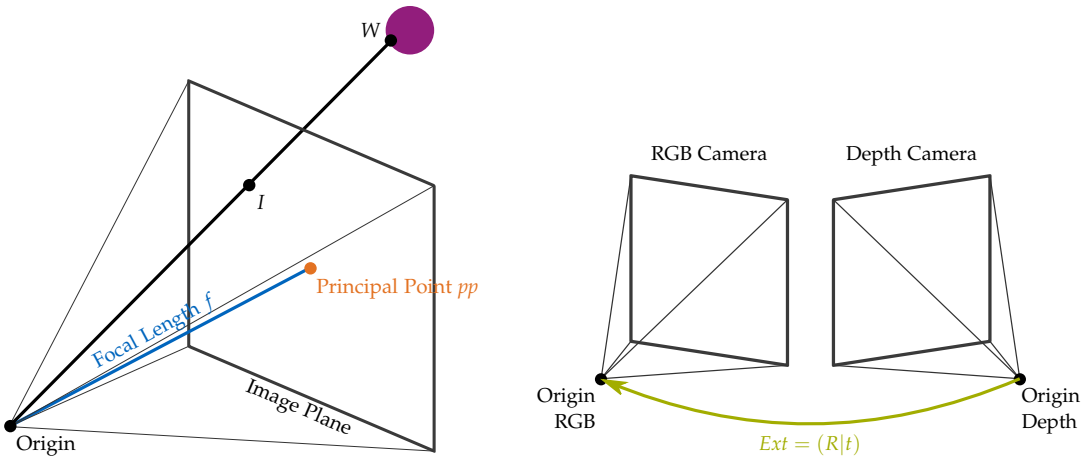
4.1.3.1 Calibration

A prerequisite to perform proper registration is a suitable calibration [Reh+17] for both involved lenses. By calibration, we can retrieve the camera parameters which describe the properties of the single lenses as well as the displacement between both lenses. The calibration process is thus split into two parts, the intrinsic calibration which is done for each lens individually (mono) and the extrinsic calibration which is used to describe the relationship between the lenses (stereo). Since we have two lenses, the color, and the depth sensor, we need to perform two intrinsic calibrations in total for each lens and one extrinsic calibration.

In Figure 4.6 you can see an overview of both calibration types with their involved parameters. For the intrinsic calibration, we want to determine the focal length f and the principal point pp for the lens. The principal point contains the x and y coordinate of the image center in pixel units. In the perfect case, this value should be half of the pixel grid resolution of the camera. The focal length, on the other hand, is the distance between the image plane, where the 3D world is projected onto to capture a 2D image, and the origin which is the point where all camera projection rays originate from. It is also expressed in pixel units like the principal point pp. Due to small inaccuracies during the manufacturing process of lenses, these values need to be estimated for each lens individually since they will always defer more or less from the specific or expected value. This also leads to the fact that there are actually two focal lengths f_x and f_y for both image plane directions which both need to be determined by calibration. Since the focal length is given in pixel/sensor units and a sensor unit may not be exactly quadratic but slightly rectangular these two values defer by the actual aspect ratio of the sensor units. In a perfect lens with perfectly quadratic units, the two focal lengths would be identical. Figure 4.6a visualizes the focal length (blue) and the principal point (orange) of a camera. Additionally, it shows how the 3D point W on the violet ball gets projected through its camera ray on the respective point I on the image plane.

The extrinsic calibration is used to determine the displacement of two stereo cameras. As displayed in Figure 4.6b the extrinsic parameters consist of the rotation matrix $R \in \mathbb{R}^{3\times3}$ and translation $t \in \mathbb{R}^3$ which are the transformations necessary to map the origin of the depth camera to the origin of the RGB camera. They both describe by what amount one camera is shifted and rotated compared to the other camera. Using these parameters we can later map the depth image to the color image. Since we are dealing with points in the 3D world the rotation matrix and the translation vector are three-dimensional. It holds that

$$origin_{rgb} = R \cdot origin_{depth} + t. \tag{4.1}$$



(a) Intrinsic calibration.

(b) Extrinsic calibration.

Figure 4.6: Intrinsic and extrinsic camera calibration.

4.1.3.2 Depth Registration

If we have successfully retrieved the necessary intrinsic and extrinsic parameters we can start aligning the captured depth maps to their respective color frames [Reh+17]. Depth registration is a three-step process. It first deprojects the depth pixel, then transforms its 3D coordinates to the RGB camera coordinate system and finally projects them on the RGB camera image plane.

Now consider we want to align a depth pixel with coordinates (x^d, y^d) to its respective color pixel coordinate (x^{rgb}, y^{rgb}) (Figure 4.7). For the deprojection step we first express the pixel coordinates (x^d, y^d) relative to the depth camera origin as the 3D point I^d which is lying on the image plane of the depth camera. This is just the pixel coordinates minus the depth camera principal point with the depth camera focal length on the z-coordinate.

$$I^d = \begin{pmatrix} x^d \\ y^d \\ f^d \end{pmatrix} - \begin{pmatrix} pp_x^d \\ pp_y^d \\ 0 \end{pmatrix} \tag{4.2}$$

Then we deproject I^d by diving the result by the focal length of the depth camera f^d and multiplying by the given depth at the original coordinates (x^d, y^d). We retrieve

$$W^d = \frac{I^d}{f^d} D(x^d, y^d). \tag{4.3}$$

W^d is the deprojected world coordinate point corresponding to I^d, given relative to

the origin of the depth sensor. Now we need to convert this point from the depth camera coordinate system to the RGB camera coordinate system. This is done as in Equation 4.1 using the extrinsic calibration data $Ext = (R|t)$ to get the point W^{rgb}.

$$W^{rgb} = R \cdot W^{d} + t \tag{4.4}$$

The last step is to finally project the transformed point on the image plane of the RGB camera to get I^{rgb}. In order to achieve that, we divide W^{rgb} by its z coordinate and multiply by the RGB focal length.

$$I^{rgb} = \frac{W^{rgb}}{W_z^{rgb}} f^{rgb} \tag{4.5}$$

The desired RGB camera coordinate pair (x^{rgb}, y^{rgb}) is finally given by adding the principal point shift to I^{rgb}.

$$\begin{pmatrix} x^{rgb} \\ y^{rgb} \\ f^{rgb} \end{pmatrix} = I^{rgb} + \begin{pmatrix} pp_x^{rgb} \\ pp_y^{rgb} \\ 0 \end{pmatrix} \tag{4.6}$$

This three step depth registration process is illustrated in Figure 4.7 as the registration of a coordinate pair which is lying on the picture of a violet ball. Since the cameras are shifted they see the ball from different angles and the position on the output image does not match each other. By applying the technique from above though one is able to successfully find the matching depth information for each color pixel.

As explained before, there are actually two focal lengths f_x and f_y for each axis involved in the intrinsic calibration. For simplicity we neglected this fact in the pixel projection and deprojection step. However eventually, we need to incorporate both focal lengths for each axis and adapt the deprojection as

$$W^{d} = \begin{pmatrix} \frac{x^d - pp_x^d}{f_x^d} \\ \frac{y^d - pp_y^d}{f_y^d} \\ 1 \end{pmatrix} D(x^d, y^d) \tag{4.7}$$

and the projection as

$$\begin{pmatrix} x^{rgb} \\ y^{rgb} \end{pmatrix} = \begin{pmatrix} \frac{W_x^{rgb}}{W_z^{rgb}} f_x^{rgb} + pp_x^{rgb} \\ \frac{W_y^{rgb}}{W_z^{rgb}} f_y^{rgb} + pp_y^{rgb} \end{pmatrix}. \tag{4.8}$$

The coordinate transformation step is not affected and is the same as in the simplified approach with the single focal length f.

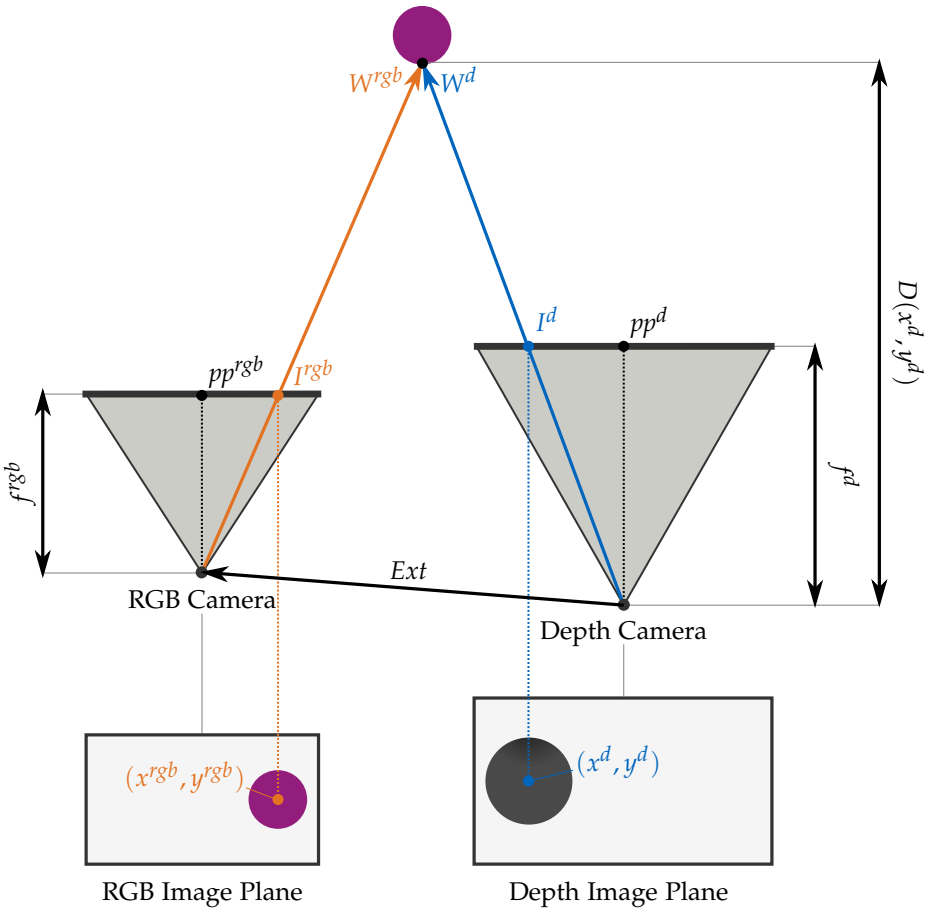


Figure 4.7: Depth registration.

4.1.3.3 RGB-Depth Calibration

Performing calibration and retrieving the necessary parameters for image registration is a well-studied field concerning classical RGB camera pairs. This usually involves the recording of several pairs of checkerboard captures, each from a different angle. A program is then used later where apart from having to select the edges of the checkerboard, everything is computed automatically. The intrinsic calibration is done for each lens first involving the described checkerboard process and then by using this intrinsic calibration result, the extrinsic calibration is computed. There are a lot of programs available for this task like OpenCV or the Camera Calibration Toolbox for Matlab [Bou01] which builds on OpenCV.

However, when it comes to calibrating an RGB sensor with a depth sensor this topic

gets more complicated. Depth sensors cannot see the structure of the checkerboard, they will only see a plane which has similar depth. Therefore the calibration of depth sensors to find its focal length and principal point is a more challenging field. RGB-D cameras also come with both an RGB sensor and a depth sensor with an already manufacture provided calibration. Thus the need for depth sensor calibration is less relevant since the existing calibration is already quite accurate. Nevertheless, some people found the manufacture calibration of those devices not sufficient which led to some research to perform RGB-D calibration in different ways [BMP17]. Most authors though focus explicitly on certain RGB-D devices, mostly the Kinect and also release a toolkit which only can handle this type of camera. Our custom architecture of using an Android phone as an RGB lens and the depth sensor of the RGB-D camera as the depth source is usually not considered by those approaches.

This leads us to a different approach to get our setup calibrated. By using the fact that the RGB and depth sensor of the Asus Xtion are already calibrated, meaning the driver can produce depth frames which are aligned to the Asus RGB frames, we perform the calibration between the RGB sensor of the depth camera and the Android camera. Thereby we circumvent the depth sensor calibration problem and fall back to the well studied RGB stereo calibration. Doing so, we register every depth frame twice in the end. Once to RGB-D color sensor, which is handled by the driver already, and once to the Android phone which is done by us using our calibration. In theory, one could also extract the extrinsic calibration of the RGB-D device and combine it with the extrinsics of the Android RGB and Asus Xtion RGB sensor. This would be a simple combination of the coordinate transform step in the depth registration method. In this way, we could deproject from the depth sensor, do the combined coordinate transform using both extrinsics and project again to the Galaxy S7 smartphone. The result though should be approximately the same with when doing two consecutive registrations with additional projecting on the RGB-D color sensor first before registering to the Android camera. The latter also avoids the expense of extracting the manufacture calibration of the Asus Xtion.

4.1.3.4 Time-Syncing RGB Sequences with Depth Sequences

Another issue we face when recording video sequences with both the Android smartphone and the RGB-D camera is how to properly determine which frames belong together in the end. Since there will always be a slight delay between the recording start of each device and also the frame rate will vary slightly during the capturing process, we need a reliable way to correctly associate the color and depth frame pairs which are taken at the same time. Since the recording of the RGB-D camera is managed by a PC and the Android system is also fully programmable we can save the recording

time stamp for each frame. However since the system clocks of the PC and the Android phone will always be slightly off, we cannot conclude that similar timestamps result in frames taken at the same time. Even if the clocks of the two devices only differ by several hundred milliseconds, the color and depth sequence will be severely out of sync. For our dataset, we need perfect matching color and depth information so it is necessary to fix this problem as good as possible.

A common way to sync the system clocks of multiple systems is the Network Time Protocol (NTP) [Mil91]. Using the NTP all systems to be synced contact a common time server which will send its common clock time back to the systems. Thus all devices will receive the same common time from one server and now have a common clock to operate on. The NTP is necessary for coherent time stamping between different systems. We also experimented with NTP to connect from both the laptop and the Android phone to the same time server to agree on a common clock. As a result, we got more similar clock times on both devices, though it was still not accurate enough. The delay by contacting the timeserver made the approach too inaccurate and unreliable.

So we come up with another way to get similar system clocks on both devices. For that, we use the fact that both the phone and the RGB-D sensor are connected to the same laptop. Furthermore, it is possible to establish a socket connection between the laptop and the Android phone via the Android Debug Bridge (adb). Therefore we create a socket between the RGB-D recording Python script and the Android application which is responsible for recording the focus sweeps. Now we can exchange messages between those two programs. We want to send the current system clock time of the laptop to the Android phone and compute all timestamps based on this common clock. To do so we send a message from the Python program to the Android application containing its current system time. This message will be forwarded via the respective USB port where the Android phone is connected. After the Android phone received the message it will retrieve the time from it and from now on operate on this clock. Even though the USB connection is pretty fast and the delay of the sent message is very low it may still be enough to result in slightly differing clocks. Also, the USB connection speed is unreliable meaning that sometimes we get a higher delay than usual.

A way to address this problem and to notify the Android phone about the delay we request the phone to send a message back to the PC after the initial message is received. When the Python script receives this response we can calculate the round trip time which is the difference between the time when the message to the Android phone was sent and the time when the response from the phone was received. By assuming the USB connection operates on about the same speed in both transfer directions, we divide the calculated round trip time by two. Now we have an estimate of the delay which happened when the messages were exchanged. We again send a second message to the phone containing the calculated delay estimate. Now the phone can correct the

received server time by the received delay and is thus now able to accurately translate its own system time to the PC system time.

Figure 4.8 displays how the message transfer works. The PC sends a message at time t_1^{PC} containing his timestamp. The Android phone receives that message at time t_1^{And} and saves the received PC time. It will respond with an empty message after that. As the PC receives the response it calculates the round trip time RTT_1 of the message exchange by subtracting t_1^{PC} from the time when the answer was received. Then the PC will send a final message containing the half round trip time as content to the phone. Using the received values the Android phone can calculate the clock difference of both

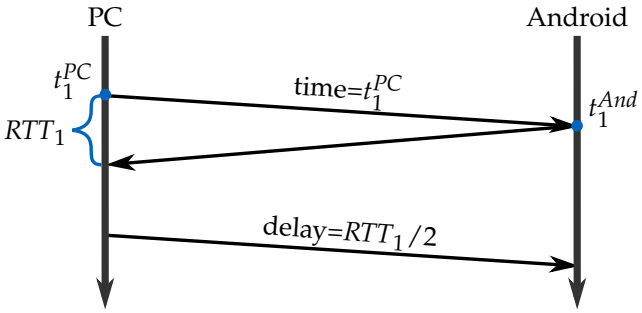


Figure 4.8: Time-Sync message exchange.

systems Δt and use it from now on for inferring the PC time from its own system clock time. This can be described by the following equations

$$\begin{aligned} \Delta t &= t_1^{PC} + RTT_1/2 - t_1^{And} \\ t^{PC} &= t^{And} + \Delta t. \end{aligned} \tag{4.9}$$

Experiments show that this time synchronization method is pretty accurate. Also, we found out that the delay at which the depth images are transferred to the PC over USB is neglectable. Unlike the connection to the Android phone which has lots of overhead due to adb and the socket logic, a pure byte stream is used which is pretty fast in practice. Besides, there is no way to find out the true delay from the RGB-D camera since it has no system clock or it is at least not accessible through the provided driver. Now that we have similar clocks we can match image pairs by the closest timestamps. The time-syncing issue is thus solved when using an Android phone as a camera. With a DSLR this would be much harder since these devices are usually not programmable and one does not have access to the system clock of the device.

4.1.3.5 Calibration and Registration Experiments

Before we start the calibration and recording we immovably fix both cameras on a box. The Asus device is standing on top of the block while the phone is fixed in front of it bellow the Asus camera (Figure 4.9). Since the screen of the Android phone is

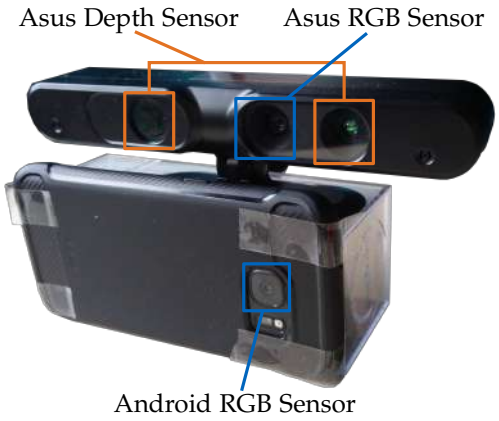
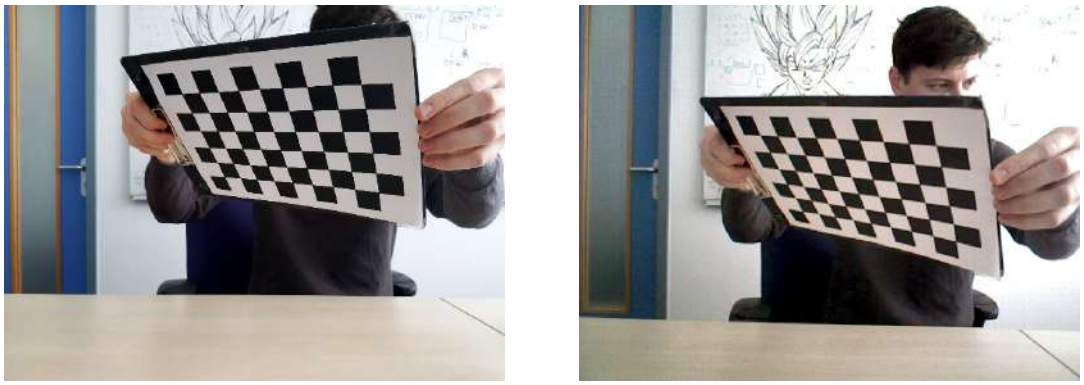


Figure 4.9: The recording setup. The Asus Xtion uses two sensors for depth map recording.

not accessible anymore, we connect the phone to a laptop and control it remotely via the application Vysor [Clo19]. The Asus Xtion is also connected to the laptop and we use OpenCV along with the OpenNI driver to capture frames. Furthermore, we capture 30 differently positioned checkerboard images with both cameras to use for our calibration. An example of such an image pair can be seen in Figure 4.10. For simplification purposes, we set the resolution of the S7 to 640×480 to match the Asus resolution. After the capture process, we use the Camera Calibration Toolbox for Matlab to calculate the intrinsic and extrinsic camera parameters. The program can input the captured checkerboard pictures and, by the help of some manual effort, calculate the desired parameters. It, however, requires the stereo setup to be a left-right setup while we have a top-bottom setup. By rotating all calibration images by $90°$ clockwise beforehand though we convert our setup to one where the Android phone is on the left and the Asus Xtion is on the right.

Using the 30 calibration images per sensor, we now perform the intrinsic calibration for each lens individually using the Matlab tool. For each image, we are requested to select the four corners of the checkerboard. Thereby it is important to always select the same first corner during the Android and the RGB-D calibration. The firstly selected corner will be treated as the origin and it is crucial for the extrinsic calibration later to have matching origins for each image pair. After the intrinsic calibrations are done we

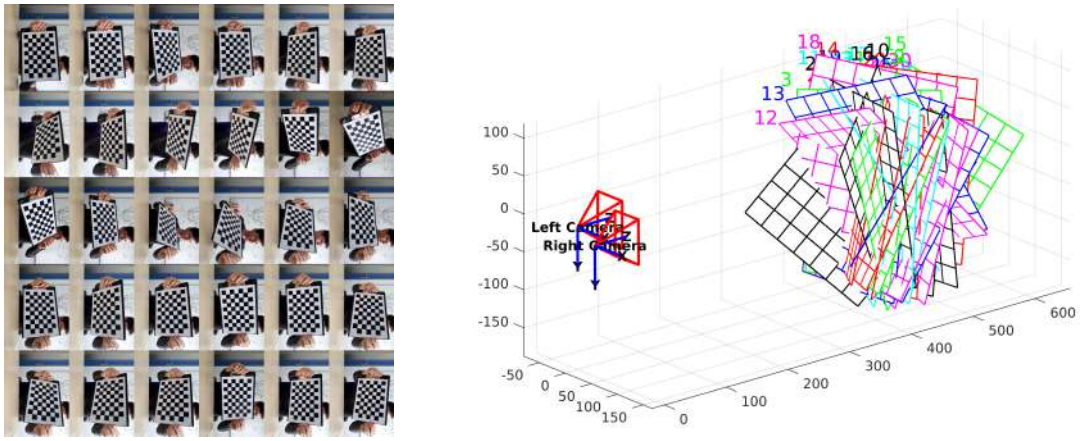


(a) Samsung Galaxy S7 capture.

(b) Asus Xtion capture.

Figure 4.10: Checkerboard captured with a Samsung Galaxy S7 camera (left) and an Asus Xtion RGB camera (right). The Asus Xtion was placed on top of the S7 for this capture and thus sees the checkerboard from a slightly different angle.

can continue with the extrinsic calibration which is fully automatic now. As a result, we get all calibration parameters saved to Matlab file. The tool also shows a visualization of both cameras in the world space including all checkerboard planes (see Figure 4.11).



(a) Samsung Galaxy S7 captures.

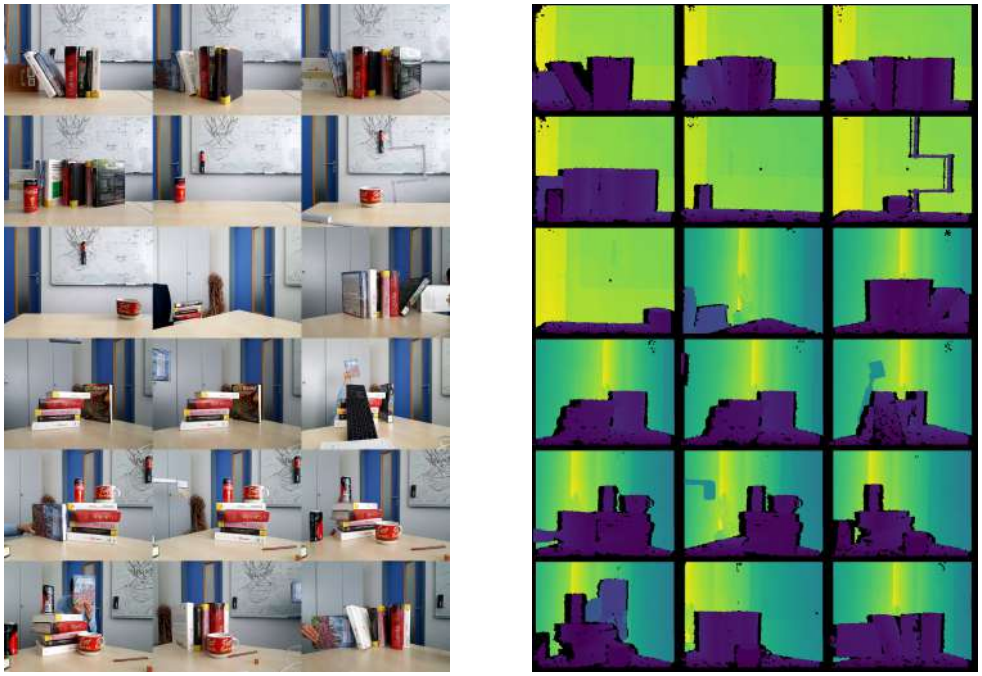
(b) Matlab toolbox extrinsics view.

Figure 4.11: Calibration results.

After the calibration is completed we can start recording the dataset. We use the first Android application for recording videos for this dataset. We set the camera to a high

resolution (2880 × 2160) to get the best blur quality. Furthermore, we set the frame rate for both devices to 30 frames per second. The depth camera can only capture at a low resolution so we will upscale the depth frames later to match the color resolution. Since the depth camera is limited by its minimum distance (about 0.5 m in practice) we set the minimum focus distance to 0.4 m and the maximum distance to 1.8 m even though we would prefer closer ranges since the depth of field of the Samsung Galaxy S7 for larger distances is very large. Every 6 seconds we do a full linear focus ramp from the minimum to the maximum and back. Each video sequence takes about 20 seconds. We shoot in a static room while moving one object at a time and keeping the cameras fixed. In total, we recorded 18 clips.

In Figure 4.12 you can see an example frame of all 18 videos. For each example,



(a) RGB frames. (b) Depth frames.

Figure 4.12: One example frame for each of our 18 clips along with its unregistered depth frame. Depth frames are normalized for visualization.

we show its color frame along with its yet unregistered depth. The depth maps are visualized using a color map where purple to blue indicates close objects and green to yellow more distant ones. Black pixels indicate that the RGB-D sensor failed to determine a depth at this position. As you can see all sequences have been recorded

in a similar setting where mostly books or cups were involved. As mentioned above, the Android application for recording videos does not wait until the lens is configured for the requested focus distance resulting in different actual focus distances. We query the Android camera API to gather the actual focus distance for each taken frame and visualize them in Figure 4.13. As we can see, the true focus distances reflect our

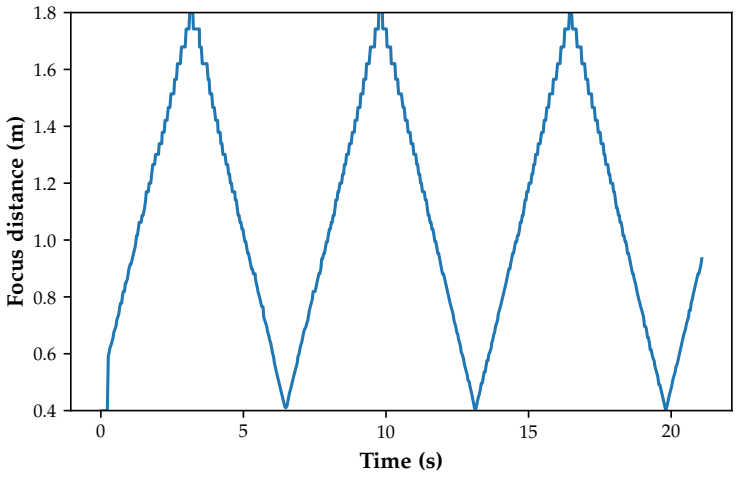


Figure 4.13: Actual focus distances for an example sequence from our dataset.

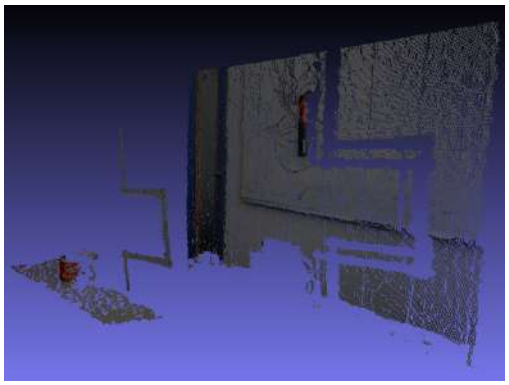
intention pretty well. About every 6 seconds one full ramp from the minimum to the maximum distance (0.4 m to 1.8 m) is performed.

Now after the dataset is recorded we can match color-depth pairs by using the frame timestamps. After that, we need to register each depth image using the intrinsic and extrinsic calibration results. In Figure 4.14 you can see an overview of the registration process for one image pair. Figure 4.14a and Figure 4.14c hereby show the color and depth frame received from the Android phone and the Asus Xtion. The respective registered version of the depth image can be seen in Figure 4.14d. To estimate how well the registration works we blend the RGB image with the depth image and observe if the outlines of the objects match with the outlines of the depth map. Figure 4.14e thereby shows the unregistered depth frame blended with the color image. As expected the object shapes do not match at all since both lenses see the scene from a different position and no alignment effort has been done. Figure 4.14f on the other hand shows the registered depth blended with the color picture. Here we can observe that the calibration and registration work well as the areas seem to correctly match each other. As a side effect though the number of unknown pixels increases a lot since there are some areas which are hidden from the depth camera due to its pose during recording.

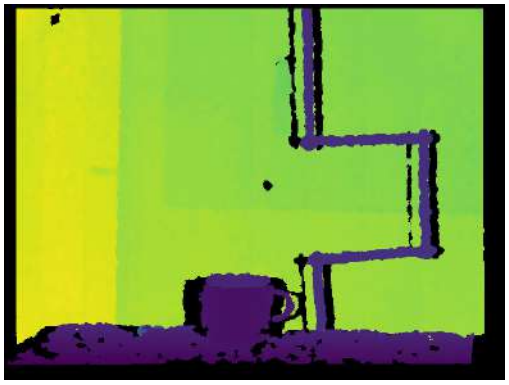
E.g. the ruler is hiding a part of the wall for the depth camera so we have no depth for this part of the wall (indicated by the ruler shaped black part). Especially at the border of the image, we get a lot of missing information. Furthermore, because of some mapping inaccuracies, we get some small black unknown areas in the registered depth.

The dataset results seem to pretty promising for training later. Though there are some undesirable problems. First of all, for a lot of pixels depth is missing meaning we would have to ignore these pixels later. This reduces our effective data amount. For some pixels, depth is not even marked as unknown but is not correct. E.g. the table in Figure 4.14 should have purple-colored depth thoroughly but also shows some yellow areas which should indicate objects being far in the background. This is caused by reflections in our setting which confuse the RGB-D sensor and result in incorrect depth estimation. There is no proper way to identify such wrongly classified areas meaning we have no other option to just feed this erroneous parts of the data to network as well. Second, we were forced due to hardware restrictions of the depth sensor to use pretty far focus distances. The S7 camera has a very large depth of field at far focus distances. As a result, defocus blur is hardly noticeable in the pictures. As a consequence, the networks later may also fail to recognize the blur which is not what we intend. Third, even though the depth registration yields good results, there are still inaccuracies around the border of objects. The last and biggest disadvantage of recording a real-world dataset is that it is very time-consuming. Even though we only recorded 18 sequences we needed several hours in the end. Neural networks are known for needing a lot of data to be properly trainable. At least 1000 sequences would be a desirable data amount. Collecting this huge amount of data is unfeasible for us which is why we eventually decide to not use this data later for our experiments. Nevertheless, for a bigger team with more resources, collecting a large real dataset, as we did at a small scale, might still be a viable option.

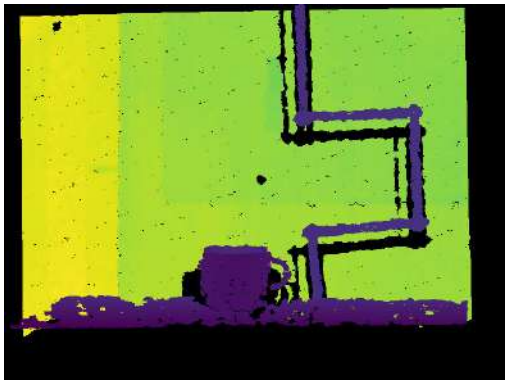
(a) Android RGB image.



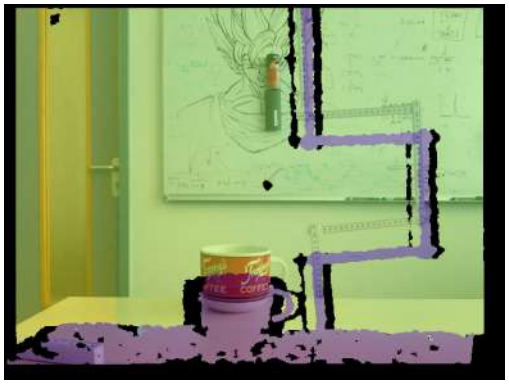
(b) Point cloud created using RGB and depth.



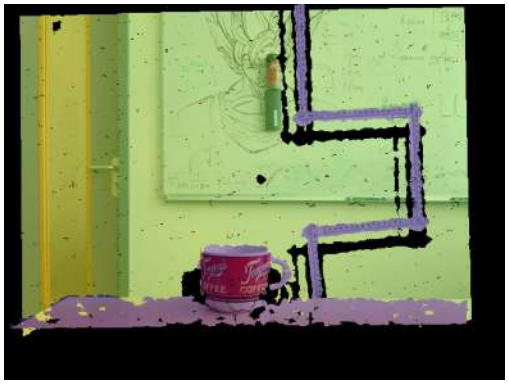
(c) Unregistered depth map.



(d) Registered depth map.



(e) RGB blended with unregistered depth map.



(f) RGB blended with registered depth map.

Figure 4.14: Depth registration results shown on a dataset example.

4.2 Synthetic Blender Dataset

To create datasets efficiently we are going to to use the 3D computer graphics software Blender to render virtual/synthetic scenes for us. We want Blender to automatically create scenes and render focus sweep sequences for our dataset. For that, we use Blender's render engine Cycles which can render scenes with realistic lighting and also implements a lens model similar to the real world with configurable parameters such as f-number, focal length, and sensor size. It is thus possible to simulate arbitrary lenses making it very convenient to configure the amount of defocus blur we desire. Another advantage when creating data with Blender is that the program can output a mathematically accurate depth map for each render. No calibration, registration or time-syncing is thus required as we needed for our real-world dataset. Also, we can create scenes with any 3D models and materials/textures that are available. The light setting can be chosen arbitrarily as well and the objects can be animated freely to move or rotate in space. The only issue with synthetic data might be the question if the neural networks later can generalize to real-world data even though they are trained on virtual Blender renders. Synthetic data output has generally no camera noise, camera shake or focus breathing as our real-world data has. Also, in the end, the lighting of the Cycles engine might appear realistic, however, it is nevertheless just an approximation up to a limited number of light bounces. Furthermore, 3D objects and textures are usually more perfect and clean in the virtual world than they are in the real world which may also affect the generalization power of our neural networks later. Nevertheless, since our goal is to predict depth based on defocus and the fact that the circle of confusion does not dependent on specific object shapes or textures this should not pose a serious issue for the networks. Under the assumption that the CoC is domain-independent, training with synthetic datasets is as suitable as training on real-world data.

4.2.1 Sources for 3D Models and Textures

Regardless of the potential drawbacks, creating a huge amount of data fully automatically makes the use of Blender seem like a very good alternative to our previous recorded dataset. To assemble a large variety of scenes we need a lot of 3D models and textures. For the 3D models, we use the Thingi10K [ZJ16] 3D model dataset which includes 10000 3D models which are originally used for 3D printing tests. Almost all objects are stored as STL files which can be natively imported into Blender. The dataset is freely accessible on the Internet. Some models are quite large leading to long render times which is why we will only include models of this dataset which have a file size less than 1 MB. This effectively leaves around 6400 models. For the textures, we are going to make use of multiple sources. The first of them is Texture

Haven [Tuy19] consisting of about 120 high-quality textures. Each of them is not only provided with a diffuse map but also specular, normal and roughness textures are supplied. We will only use diffuse and specular textures for our Blender scenes here for simplicity. Also, normal maps are meant to define additional, more detailed 3D structure of objects by defining how the light should be reflected at each position. This modification though is not reflected in the depth maps produced by Blender meaning we would introduce discrepancies between our color and depth images. Our second source is Texture Ninja [Joo19], also offering high-resolution textures. The quality may be slightly worse than the one from Texture Haven though this website features a much larger pool of over 5000 textures. The last texture database we use is the one from GRSites [GRS19]. Including about 5700 textures, the quality is way less than the other two sets. Though this database is still useful since a lot of textures are just simple colors or pattern. In the real world, there are also lots of simple colored objects meaning we can simulate this aspect of the world by this texture source. Additionally, usually intended for web design, this dataset features textures of almost every color. To vary the illumination in our scene we will also use environment maps which will affect the light color. HDRI Haven [Zaa19] offers about 200 high-quality maps for this purpose. We crawl all available textures on those sites with the help of a Python script. The 3D models, on the other hand, can be simply downloaded as one package on the website.

4.2.2 Random Scene Generation

Equipped with a large pool of 3D objects and textures we can now start to develop a script for automatic scene assembly. Blender uses the Python programming language internally and is able to execute any given Python script. We can fully access the functionality of Blender via Python which is very convenient for realizing our idea. First of all, we will position the camera at the origin of the scene and make it face the positive y-direction to simplify the calculations for the object placement later. You can see the conceptional setup of our scene in Figure 4.15. We will keep the camera always fixed and just animate the objects. Additionally, we position a box acting as a wall at the very back of the scene which fills the complete field of view. Thereby we avoid infinite depth values which occur when a pixel sees no object. Now for each scene we want to randomly select a portion of 3D models (e.g. 100 objects per scene) and textures and place them randomly in the field of view of the camera. To sample spawn points in the camera field of view we firstly calculate the field of view angles FOV_x and FOV_y in both the x- and y-axis of the camera image plane. FOV_x can be calculated by querying the configured sensor width S and focal length f in Blender and then solving

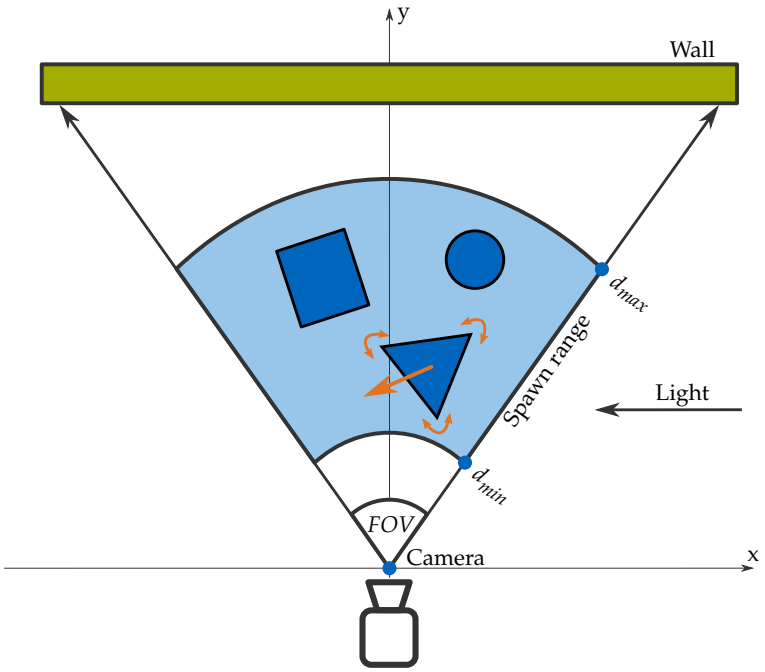


Figure 4.15: Blender random scene concept.

the equation

$$FOV_x = 2 \arctan\left(\frac{S}{2f}\right). \tag{4.10}$$

FOV_y is then given by FOV_x and the aspect ratio of the render resolution as

$$FOV_y = \frac{FOV_x}{\frac{resolution_x}{resolution_y}}. \tag{4.11}$$

Next we also want to set a minimum distance d_{min} and maximum distance d_{max} to specify how close or far away objects should be spawned from the camera. To now select a random position in our desired spawn area we choose random spherical coordinates fulfilling

$$\begin{aligned} r &\in [d_{min}, d_{max}] \\ \theta &\in \left[-\frac{FOV_y}{2}, \frac{FOV_y}{2}\right] \\ \varphi &\in \left[-\frac{FOV_x}{2}, \frac{FOV_x}{2}\right]. \end{aligned} \tag{4.12}$$

To account for our camera being rotated to face the positive y-direction in Blender we need to additionally rotate by 90° in both spherical directions so our coordinates lie in

the field of view of the camera later. The distance is unaffected though. Our adjusted spherical coordinates are

$$\begin{aligned} r' &= r \\ \theta' &= \theta + \frac{\pi}{2} \\ \varphi' &= \varphi + \frac{\pi}{2}. \end{aligned} \tag{4.13}$$

Finally we convert the spherical coordinates to Blender conform Cartesian coordinates using the equations

$$\begin{aligned} x &= r' \sin\theta' \cos\varphi' \\ y &= r' \sin\theta' \sin\varphi' \\ z &= r' \cos\theta'. \end{aligned} \tag{4.14}$$

Also we choose a random object rotation and a random object scale for additional scene variety. Since the 3D objects have varying dimensions we first normalize them. We also want objects in the front to be smaller than objects in the back so they do not occlude the scene too much and to make objects in the back as visible as closer objects. For this purpose we multiply the scale of each object with its distance to the camera. For different illumination settings we randomize the light rotation around the origin and offset it afterwards randomly from the origin. Also the light strength is randomized. For varying the light color, we either select a uniform color or an environment map from the HDRI Haven dataset by chance.

In order to create some object movement we randomly generate two 3D vectors per object v_P and v_R to act as translation and rotation velocity. We will animate each spawned object by offsetting its initial position P_0 and rotation R_0 for each frame $t \in [0, T-1]$ as

$$\begin{aligned} P_t &= P_0 + t v_P \\ R_t &= R_0 + t v_R. \end{aligned} \tag{4.15}$$

These poses will be inserted as keyframes in Blender. Later we will render the whole animation to get the desired sequence. We have no handling of object collision which does not matter later on. We also experiment to handle collision and prevent object intersection via the Blender physics system but eventually experience it as very unstable and unreliable. Also, we neglect that objects can move out of the camera view during the animation but since we will not have really fast movements it is not necessary to take care of that issue.

We also want to vary the object textures and materials. Besides textured objects, our scene should also contain untextured materials to represent objects with a simple color in the real world. Nevertheless, for each object, a material is created. Based on a certain probability we choose a texture from our texture pool as diffuse and

specular (if available) color or a uniform random color. The diffuse and specular ratio is also randomly determined. Additionally, instead of choosing a texture from our dataset we will also use a random Blender noise texture based on a probability. Blender noise textures are textures filled with random colors and thus are also a good way for randomization.

4.2.3 Rendering Experiments

Now we can automatically render concrete datasets without any user interaction. Since we want to test our networks on real-world datasets taken from a Samsung Galaxy S7 we set the f-number, focal length and sensor width to match the phone's camera. The render resolution is set to 640×480 since our neural networks will later only input low-resolution crops anyway and it is also a natural camera resolution option. For each frame, we need to perform two render passes, one for the color frame and one for the depth frame. If we would render both frames in the same pass, the depth frame would also have defocus blur as the color frame has. Depth frames should be perfectly sharp though. It is already mentioned above that we want to use Blender's high-quality renderer called Cycles. Cycles is a lot slower than Blender's more simple renderer named Blender Render but produces superior realistic results. However, to speed up render times for the dataset we will use Blender Render to render the depth frame. We do not need any lighting or other realistic effects when computing the depth making the use of either Blender Render or Cycles equivalent when creating depth maps. Blender Render takes only about a fraction of the time Cycles does. E.g. in our experiments later, Blender Render could produce frames in about 1-2 seconds while Cycles needed about 45 seconds. Initially, with default settings, though Cycles needs two minutes per frame which is too long. We make some adjustments to the Cycles render engine to get a good balance between quality and performance. First of all, we set the samples to 512 and the tile size equal to the render resolution. Also, we limit the number of light bounces from 0 to 2 and configure the glossy bounce to 1. The transparent and transmission bounces are all set to 0. Additionally, we disable both types of caustics. The last setting we make is to clamp the direct light at 0.5 and the indirect at 0. These changes, in the end, are sufficient to improve the render performance from 2 minutes to about 30-45 seconds depending on the objects in the scene. For the Blender Render, we do not need to make any adjustments. Since defocus blur is also not handled by Blender Render as it is in Cycles the output depth frames will automatically be perfectly sharp.

Before starting the render process we split the 3D model database and texture database we have in two sets for training and testing. Thus we will be able to also test our model performance in the virtual world later. This is useful when we want

to test how our network performs in cases which are hard to model in the real world. E.g. how our models would behave if it would receive perfectly sharp images without defocus blur. Such pictures are hard (theoretically impossible) to record with a real camera. We will use 80 percent of the objects and textures for our training set and the remaining 20 percent for testing.

Since we are not limited by the minimum distance like it was the case with the Asus Xtion, we will choose closer focus distances now for our synthetic dataset. We set the range between 0.1 and 0.45 meter. Also, we decide to record far fewer frames per focus sweep resulting in more focus change compared to the movement of the objects. The neural networks do not need that many frames later as input and it saves render time. So we choose to render 5 frames per sweep. Additionally, we will not sweep from the maximum to the minimum focus anymore but just start again from the minimum focus distance. This removes the difficulty for the networks later to recognize if the focus distances were increasing or decreasing. The Android phone camera is also fast enough to jump from the maximum to the minimum focus distance without any delay. We are going to render 5 sweeps per random scene. In Figure 4.16 you can see a plot of the focus distances given the frame indices. We also configure Blender to output additional optical flow via Blender's vector pass option for some further testing later on.

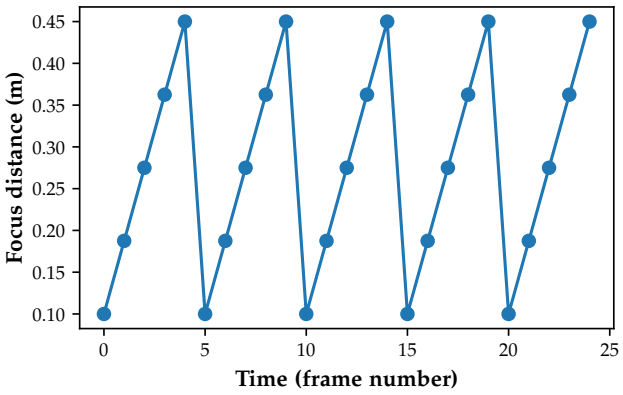


Figure 4.16: Blender focus distances.

We decide to render two different datasets to represent different aspects of the real world. One with mostly textured, small and many objects and one with more untextured, larger and far fewer objects. Per dataset, we render about 2500 sequences of randomly generated scenes consisting of 5 consecutive focus sweeps resulting in 2500×25 frames in total (color only). Figure 4.17 shows an example random scene in Blender. We increased the size of the objects a bit compared to our actual setup for visualization purposes. In Figure 4.18 you can see one render example from each set.

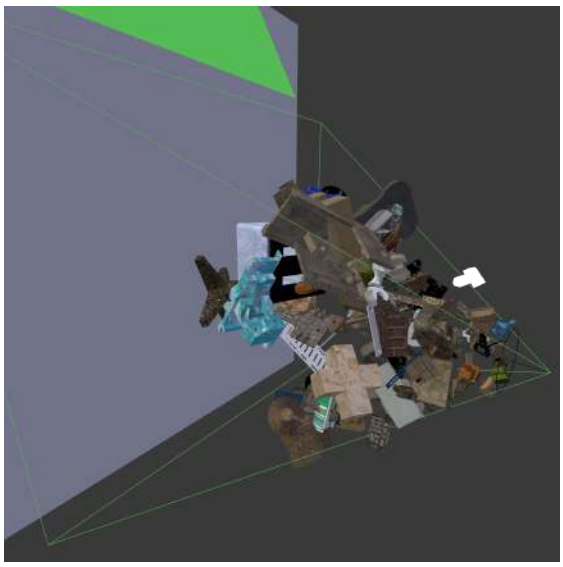
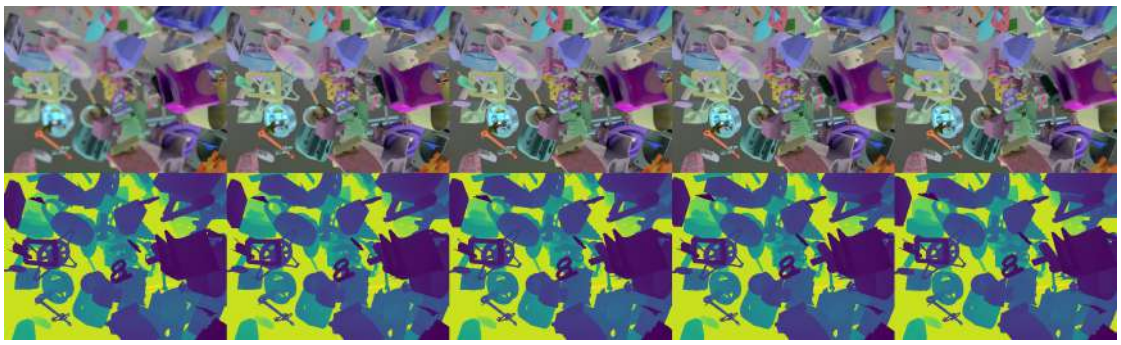
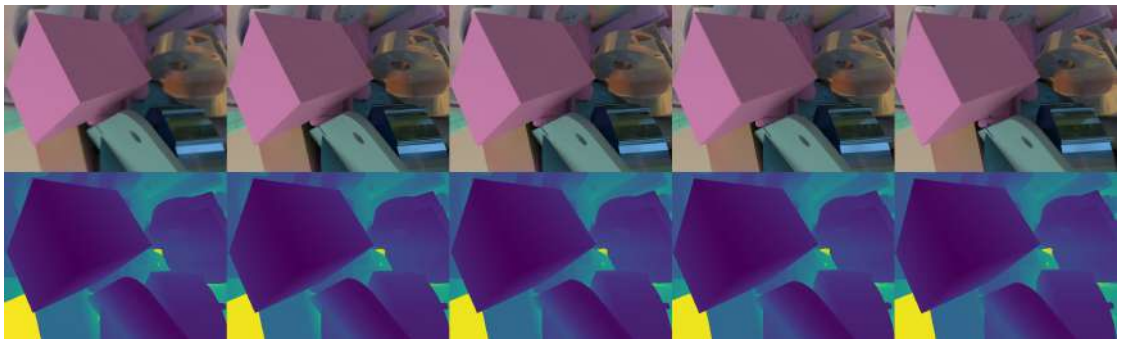


Figure 4.17: Blender random scene example. Objects have been increased for better visibility.

From the left to right the focus distances are increasing and one should be able to see that objects in the back are blurry in the beginning but get sharper each frame. The opposite holds for the objects in front.



(a) Small object dataset.



(b) Large object dataset.

Figure 4.18: An example for each of our two rendered datasets. We show one focus sweep consisting of 5 frames. The top row shows the color image whereas the bottom line shows the corresponding depth visualized using a color map (blue means near and yellow far). During the sequence, the focus distance is increasing and the objects are moving. The temporal order is from left to right.

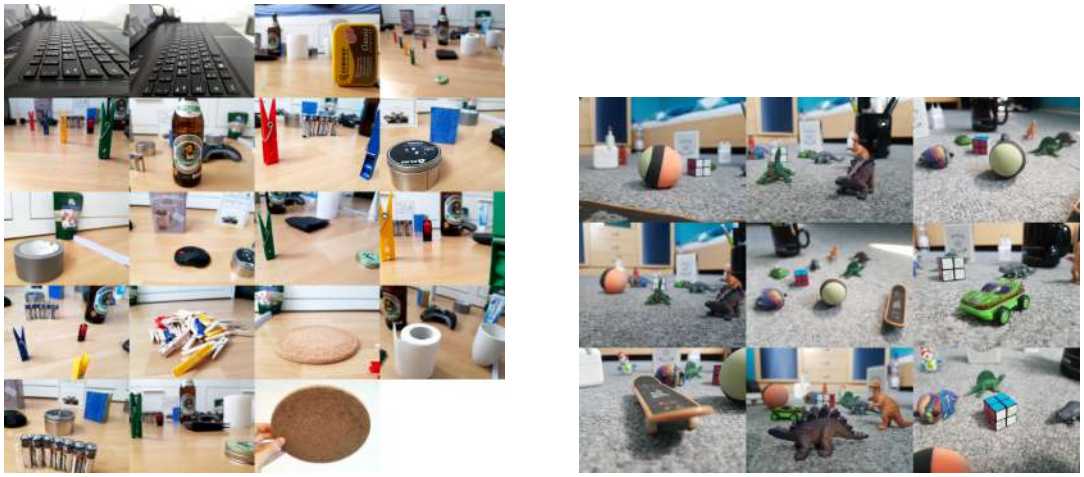
4.3 Datasets for Testing

So far we focused on creating datasets mostly for training. However later we also need to evaluate the performance of each model which is why we need independent real-world test data. The testing, in the end, should show how well our models would behave in real-world scenarios. Though having depth ground truth for the testing data is desirable for calculating various metrics it is not mandatory since we can also evaluate how plausible the output pictures look. We already experienced that it is quite tough and challenging to create depth maps from real-world scenes due to issues like various unknown regions and the minimum distance restrictions. Thus we decide to

omit the depth information for our testing sets to be more flexible in the scenarios we want to present to the networks.

In the end, we create two test sets filmed in two different rooms. Following our training set, we use the Samsung Galaxy S7 and the Android applications for recording and set the resolution to 640×480. We also mimic the focus distances to range from 0.1 to 0.45 meter and only ramp from the minimum to the maximum. For convenient recording, we will keep the objects fixed and only move the camera to create the desired scene movement.

For the first dataset though we will keep the camera as fixed as possible and only hold the phone freely so camera shake will affect the recording. We use our first video recording application which records a full video at 30 FPS from which we will extract frames later. We film a total of 19 clips. An example frame of each can be seen in Figure 4.19a. Since this scene is recorded in a room with a wooden floor we call it the wooden floor dataset. In the second scene we will actively move the camera. We will use the burst recording application this time and record four frames at the focus distances $F \in [0.1, 0.217, 0.33, 0.45]$. We do five focus ramps in a row and thus get 20 frames per clip. You can see extracts from this dataset in Figure 4.19b. We refer to it as the carpet dataset. Additionally, we will test on the dataset from Suwajanakorn et al. [SHS15] which is used in their work. They record several scenes consisting of around 10-30 frames with a mobile phone and a DSLR camera. While recording they are decreasing the focus distance from its maximum to its minimum value. They additionally process their dataset to remove any movement in the scene including focus breathing. Every scene is completely static and every frame matches its predecessor exactly except the change of defocus blur. To fit our setup, we will reverse each clip and sample four evenly distributed images spanning the complete sequence to get the maximum CoC change.



(a) Dataset wooden floor.

(b) Dataset carpet.

Figure 4.19: Real-world test datasets.

Figure 4.20: An example sequence from our carpet dataset. The camera is moving while the focus distance is increasing.

5 Training and Evaluation

5.1 Training

Now that we defined our CNN architectures and created a proper training set we can start training the models. We will use the standard Mean Squared Error (MSE) as loss function. However we will leave out pixels, which are lying on the plain white background wall in our Blender dataset. Otherwise the network may learn that any scene always has a kind of white wall as background which is not desired. We thus will use a mask set $\mathcal{M}$ which only includes pixels that have a smaller depth value than a certain threshold. The threshold will thereby be set to the distance between the camera and the wall meaning all objects lying in front of the wall will be included. We can formulate our mask as

$$\mathcal{M} = \{p \in P \mid D(p) < thresh\} \tag{5.1}$$

where P indicates the set of all pixels. The masked MSE loss for the ground truth y and the prediction $\hat{y}$ is then defined as

$$MSE_{\mathcal{M}}(y, \hat{y}) = \frac{1}{|\mathcal{M}|} \sum_{p \in \mathcal{M}} (y(p) - \hat{y}(p))^2 . \tag{5.2}$$

We will use this MSE loss for both training and validation purposes. Along with the MSE, we will also provide the root mean square (RMS) which is simply defined as

$$RMS_{\mathcal{M}}(y, \hat{y}) = \sqrt{MSE_{\mathcal{M}}(y, \hat{y})}. \tag{5.3}$$

As another validation metric, we introduce the masked accuracy loss [Haz+18] as

$$Acc_{\mathcal{M}}(y, \hat{y}) = \frac{1}{|\mathcal{M}|} \sum_{p \in \mathcal{M}} \mathbb{1} \left(\max \left(\frac{\hat{y}(p)}{y(p)}, \frac{y(p)}{\hat{y}(p)} \right) < 1.25 \right). \tag{5.4}$$

If the target and the predicted pixel value differ less than 25%, the pixel will be counted as correctly predicted. The ratio of correctly predicted pixels to the total pixel count in the mask is then the accuracy. The validation metrics will only be evaluated on the depth predictions and not on the CoC output. The train loss for our architectures which do not predict the CoC but the depth D only is the simple MSE loss

$$\mathcal{L}_D = MSE_{\mathcal{M}}(D, \hat{D}). \tag{5.5}$$

If there is a CoC component C we sum up the MSE of the depth maps and of the CoC maps using a weight λ_C which we set to 1. This gives

$$\mathcal{L}_{D,C} = MSE_{\mathcal{M}}(D, \hat{D}) + \lambda_C MSE_{\mathcal{M}}(C, \hat{C}). \tag{5.6}$$

We implement our approach using PyTorch [Pas+17] and train the CNNs with a learning rate of 1×10^{-4} and a batch size of 4. As optimizer we choose Adam [KB14] which is common choice for training neural networks. We are going to train on NVIDIA TITAN X 12 GB GPUs for our single CNNs and on NVIDIA Quadro P6000 24 GB GPUs for our larger consecutive architectures. After 200 epochs we stop the training process. The network is always going to receive one focus ramp reaching from the minimum to the maximum focus distance. From the five focus ramps per sequence, we will select one randomly and feed it to the network. Even though one ramp consists of five subsequent frames we will mostly only use four frames if not explicitly specified since this leads to the best performance. When using four frames we will leave out the fourth frame in the sequence and thus forward frames with the focus distances $F \in [0.1, 0.1875, 0.275, 0.45]$. By default, we will also only train on the first dataset containing the smaller, highly textured objects. We will explicitly state later when we incorporate also the part with the larger objects as an addition to the smaller object dataset. 20% of the train data are going to be used for validation only. Our images in the dataset have a quite large resolution of 640×480 so we are going to randomly crop a 256×256 part out of them before handing them to the CNNs. For additional data augmentation, we horizontally flip the frames of each sequence with a probability of 50%.

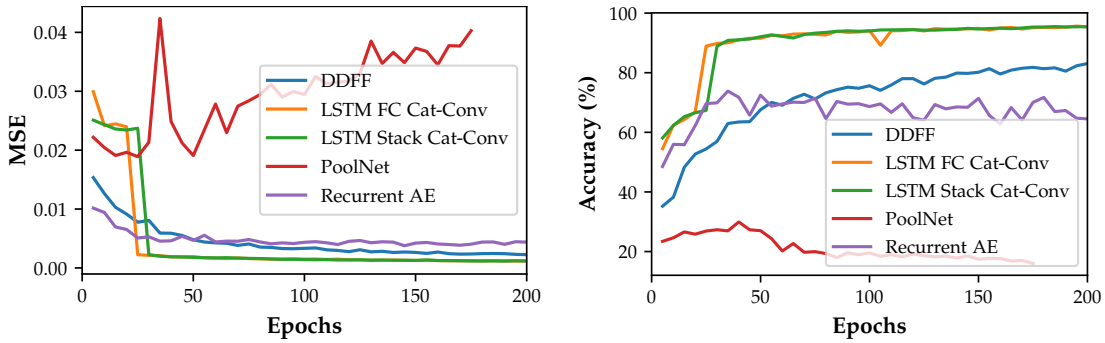
We have depth ground data for each frame as a result of our Blender render output which we will use to supervise our architectures. However, we will always only predict depth for the last frame in the focus sweep so we will only provide the last depth map to the loss function. The last frame thereby has the largest focus distance (the peaks in Figure 4.16). The same accounts for the CoC maps. A special case is our consecutive architecture which first predicts the circle of confusion for every frame and then a single depth frame. In that case, we will use all CoC maps. We do not have CoC maps stored on the drive but create them on the fly by inserting the values from the depth maps and the camera parameters in Equation 1.1. Afterwards, we normalize both depth and CoC maps by diving by the maximum depth and CoC value which appears in the dataset excluding the wall. As a result, we get depth and CoC maps having values between 0 and 1 which commonly leads to more stable training.

5.2 Evaluation and Results

We are now presenting our training and testing results. The training dataset was additionally split in a train and validation set at an 80:20 ratio. We will thereby provide tables showing the quantitative results obtained by evaluating on our synthetic Blender test set. The quantitative results are composed of the average metric values. Furthermore to determine the performance of the architectures in real-world scenarios, we will test on the Suwajanakorn data as well as on our two real-world datasets. Since our real-world test sets are not labeled we will provide the predicted depth maps instead of metric values. As a first instance to determine the CNN candidates which perform the best on real-world data we will use the Suwajanakorn dataset since it is perfectly aligned and has no movement. Afterwards, we will further evaluate the best networks on the more challenging carpet and wooden floor data. The carpet data contains lots of camera motion whereas the wooden floor data has a reflective floor making it challenging for the architectures to determine the correct depth values. Since there is no depth ground truth data for the real-world test sets, we will rely on our perception to rate the performance of our models. For visualizing our depth and CoC maps we will use the colormap in Figure 5.3. Dark blue therefore signalizes close objects whereas yellow indicates very far ones. Regarding the CoC, dark blue stands for perfectly sharp (zero CoC) and yellow for strong defocus. While evaluating we do not focus on perfect absolute depth values but rather on relative depth relations. That is why we will only show normalized depth maps meaning we scale them accordingly to always span the whole color range of the colormap.

Figure 5.1 shows an overview of the training process of our proposed architectures by showing the MSE and accuracy for the validation set for each training epoch. The models thereby are in their most simple configuration without incorporating the large object dataset or using CoC supervision. DDFF is thereby the unmodified version by Hazirbas et al. which we will use as a reference to compare our derived architectures. LSTM FC Cat-Conv and LSTM Stack Cat-Conv, on the other hand, represent the DDFF-LSTM using fully connected layers and the version which stacks flattened feature maps for a long sequence LSTM respectively. Both are using a convolution for the skip connection aggregation. We can see that except the PoolNet every model is converging to an appropriately low MSE. The DDFF-LSTM versions hereby seem to yield the best results followed by the DDFF and the recurrent autoencoder. Since the PoolNet fails to converge during training we will not further evaluate its performance on the test sets. We experiment with further extensions as the use of CoC supervision, providing the focus distance as an additional channel and using two PoolNets in sequence but did not manage to achieve good results. We thus conclude that the PoolNet is not suited for our aim and focus on our remaining architectures which have a promising low

validation loss.



(a) MSE.

(b) Accuracy.

Figure 5.1: Validation metrics retrieved during the training process for the DDFF and the proposed architectures.

5.2.1 LSTM-DDFFNet

For the first experiments on our test sets, we will use the standard DDFF and the LSTM-DDFF variations. We evaluate both the LSTM-DDFF version which flattens all feature maps to one large vector and reduces the input using fully connected layers (FC) and the version which flattens each feature map individually and feeds it stacked as a long sequence to a double-layered LSTM (Stack). We also have two variations of how to accumulate the skip connections from the multiple encoders to match the dimensions of our single decoder. We test the scheme where we only consider the output of the last encoder (Cat-Last) against scoring the outputs from all encoders using a convolution (Cat-Conv). All networks are trained on the small textured objects set only. Since all LSTM variations are using fully connected layers or a fixed size LSTM they can only input 256×256 image crops. We thus subdivide the higher resolution test frames in appropriate crops and then reassemble the output again to get one final depth map. Table 5.1 shows the quantitative results obtained by the models at the final epoch 200. The results are calculated using the synthetic Blender test set which we rendered along with the training set. Therefore the test set only incorporates unseen objects and textures and is thus a good way to figure out if the models can infer depth based on the given CoC and not based on the known object shapes and textures. The table shows that most of the LSTM-DDFF versions have a rather high MSE and a low accuracy compared to the standard DDFF. However, the LSTM Stack Cat-Conv shows very good test results and an accuracy of about 95% which is also way higher than the

standard DDFF.

Table 5.1: Quantitative results for the DDFF and LSTM-DDFF variations.

Model	MSE	RMS	Accuracy
DDFF	1.8×10^{-3}	0.041	86.78
LSTM FC Cat-Last	2.0×10^{-2}	0.140	23.90
LSTM FC Cat-Conv	1.8×10^{-2}	0.132	30.18
LSTM Stack Cat-Last	1.9×10^{-2}	0.139	23.27
LSTM Stack Cat-Conv	1.1×10^{-3}	0.033	94.98

We now use the Suwajanakorn dataset to check if the test losses for the synthetic dataset match with the performance on real-world data. Figure 5.4 shows the output of the model types for four different clips from this dataset. As you can see in the figure, the standard DDFF performs very poorly on all input images. It can detect the nearest points but in general mixes up close and far objects which is not desirable. All object shapes are well detected though probably because the VGG is pre-trained on performing classification which is also used for semantic segmentation [LSD14]. But the depth values are mostly far off e.g. in the picture with the red ball the front part of the couch and the second distant red ball are classified as equally distant. Also in the last clip with the books, the most distant part of the image according to the DDFF is the area below the second book (marked as yellow). Compared to the DDFF, the LSTM-DDFF versions perform a lot better. The Cat-Last variations which only consider the skip connection from the last encoder have some problems with properly detecting the near regions in the frames e.g. the front keyboard keys. However, apart from that, the depth maps seem to be quite accurate except for some noise distortions. The best performing network, in the end, is the LSTM-DDFF Stack Cat-Conv version. Here also near objects are well detected and there is less noise than in its FC Cat-Conv counterpart. An issue which all LSTM versions have though is that there are visible seams due to the performed input cropping. E.g. if you compare the right part of the keyboard depth maps with the rest of the prediction you notice that it does not perfectly match. Apart from that, the last LSTM-DDFF Stack Cat-Conv version produces plausible results.

Comparing these results with the test loss results from Table 5.1 we can see that both agree that the LSTM Stack Cat-Conv has superior performance compared to the other networks. However the test metrics rate the DDFF as the second best performing network which is not the case in the Suwajanakorn tests. The DDFF seems to overfit during training by merely memorizing object shapes which is, in the end, the original purpose of its VGG16 base. This overfitting still leads to a good performance on the less challenging synthetic dataset but eventually fails in the real-world Suwajanakorn

case. The other three LSTM versions, on the other hand, have a way higher test MSE but their generalization power is superior which is the main point in the end to achieve our goal.

Nonetheless of the decent results of the LSTM-DDFF Stack Cat-Conv, we want to test if we can avoid the undesired seams in our depth map results by downscaling the input image instead of cropping it. After some experimenting we figure out that nearest neighbor downscaling behaves best compared to other algorithms. Figure 5.5 compares both cropping and downscaling for the LSTM-DDFF Stack Cat-Conv. It can be observed that the seams are naturally absent in the downscaling approach but the general quality and level of detail is reduced. E.g. the depth of the front part of the keyboard is way less plausible than in its counterpart obtained by cropping. Also, veils with incorrect depth values around objects appear as around the red ball in the front of the first clip and around the books in the back of the last clip. We believe that the downscaling corrupts and reduces the circle of confusion leading to worse depth results.

We further experiment with two extensions of the LSTM-DDFF Stack Cat-Conv. One is to also incorporate our big untextured object dataset and use it together with the dataset with the smaller objects. We refer to it as Larger Data since the training set is now a combination of two datasets. The other one is the use of an additional decoder predicting the CoC as another form of supervision (CoC-Dec). A further test is to combine both these extensions. The results for our synthetic test set can be seen in Table 5.2. We observe that all versions yield very similar high test scores. The lowest MSE is thereby achieved by the network which uses CoC supervision.

Table 5.2: Quantitative results for the LSTM-DDFF Stack Cat-Conv variations.

Model	MSE	RMS	Accuracy
Standard	1.1×10^{-3}	0.033	94.98
CoC-Dec	9.9×10^{-4}	0.031	95.58
Larger Data	1.2×10^{-3}	0.035	94.10
Larger Data & CoC-Dec	1.4×10^{-3}	0.037	92.96

We again confirm these results with the real-world Suwajanakorn test set whereby Figure 5.6 displays the predictions of the models. The first thing we observe is that the models which were trained on the larger data perform very poor compared to the ones which were trained without it. The near area of the keyboard and the ball scene are wrongly classified as distant. The additional CoC-Decoder seems to not change the outcome that much. This finding is not reflected well in the test loss values which assigns each network a similar MSE and accuracy. In the end, we do not manage to

improve the standard LSTM-DDFF Stack Cat-Conv network by our adjustments.

5.2.2 Recurrent Autoencoder

The final tests evaluate our recurrent autoencoder networks in detail. Similar to the tests for the LSTM-DDFF we train variations involving the larger dataset and an additional decoder for the CoC. The respective metrics for the labeled synthetic test set are shown in Table 5.3. All variations manage to achieve a similar decently low MSE. In terms of accuracy, the CoC-supervision model has the best value. Comparing with Table 5.1 the metric values are worse than the standard DDFF and the LSTM-DDFF Stack Cat-Conv but better than the other three LSTM-DDFF variations. Also all models in Table 5.2 seem to outperform the recurrent autoencoders in terms of the test metrics. Nevertheless the achieved test scores seem to be sufficient for performing further tests on our real-world datasets.

Table 5.3: Quantitative results for the recurrent autoencoder.

Model	MSE	RMS	Accuracy
Standard	3.4×10^{-3}	0.058	69.02
CoC-Dec	3.2×10^{-3}	0.056	82.83
Larger Data	3.2×10^{-3}	0.056	70.77
Larger Data & CoC-Dec	3.4×10^{-3}	0.058	69.94

In Figure 5.7 you can see the results for our typical four input sequences. All four variations produce pretty good depth maps, even better than the ones from our best LSTM-DDFF network. The standard version seems to have some issues with classifying the background of the last example with the books. The background behind the two books should be filled consistently with a yellowish color. There is some noise though on the left side of the background. However, all other versions, especially the ones trained on the large dataset, do not produce or weaken this issue. Generally, the incorporation of the large object dataset seems to produce smoother gradients than the versions without. If you look at the surface of the couch in the first example or the table in the third example you can see a smoother change of depth. The CoC decoder versions introduce some inaccuracies in the close regions as with the first row of the keyboard keys. Overall we find that the larger dataset variation without the CoC supervision produces the best results. The test metrics in Table 5.3 are supporting the good performance of the models although they would rate the LSTM-DDFF Stack Cat-Conv as superior. Also, the higher accuracy of the CoC supervision version is not represented in the images.

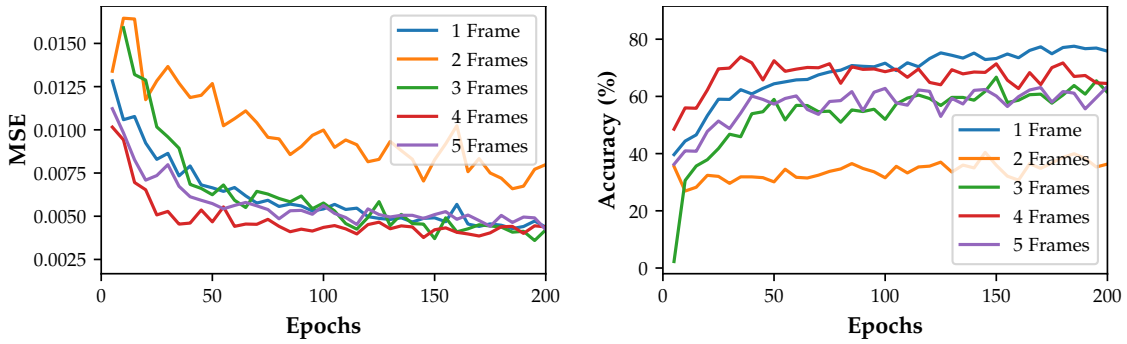
In Figure 5.8 you can see the CoC output of the two versions using the CoC decoder. The focus of the last frame is always equal to our maximum focus distance so the focus is on the background. Objects in the front are lying before the focal plane and should thus be blurred. This phenomenon can be observed very well in our CoC output. Yellow thereby indicates maximum defocus whereas dark blue indicates perfect focus. We can see that near objects are indeed recognized as out of focus while the background is recognized as being in focus. The CoC supervision is thus functioning as expected.

We additionally test these models on other samples from the Suwajanakorn dataset. This time we choose the portrait photos which are taken by a mobile phone. The previous examples are recorded with a DLSR. To get decent defocus blur the pictures contain objects which are very close to the camera. We show the result for all six sequences in Figure 5.9. As you can see, incorporating the larger dataset in the training set reduces the quality quite a lot compared to the upper two models not using it. We omit the model which is trained on the large data and uses a CoC decoder because of space reasons and the fact that it performs even worse than only incorporating the larger data without the CoC decoder. The models trained on the larger object database return wrong depth values for very near areas which can be observed in each of the examples. The CoC decoder also introduces some inaccuracies for near areas as the bottle or the kitchen scene in the third sample. However, it seems to have superior performance on the background like the room behind the green plant. Nevertheless, we think that the standard version without larger data and without CoC decoder performs the best since the foreground quality is a lot better than in the other models.

We want to further analyze the recurrent autoencoder structures by testing them on our own, more challenging datasets. The depth results for the carpet dataset, which contains a lot of movement, can be seen in Figure 5.10 and the respective CoC output in Figure 5.11. In general, the produced depth maps look very plausible meaning that the recurrent autoencoder is indeed able to handle sequences with severe object movement, or in our case, severe camera movement. The model trained on the larger dataset without CoC supervision thereby seems to produce the most smooth and accurate results. However, the depth map for the skateboard scene contains wrong depth predictions in the front center area. This is the case though with all four model variations. We can also observe this misbehavior in the CoC output. The tests for the wooden floor dataset are displayed in Figure 5.12 (depth) and in Figure 5.13 (CoC). The models incorporating the larger dataset predict again smoother depth maps but struggle to correctly identify the close area in the front. The models without can more reliably detect near areas but are noisier. The usage of CoC supervision does not affect the final output that much. According to Figure 5.13 the CoC output is better without using larger data. As another test, we evaluate the depth consistency when inputting multiple subsequent ramps from the same video. We use the five consecutive ramps of

one example from the carpet dataset and show the result for each ramp in Figure 5.14. You can see that the output of all models is overall consistent. The depth maps do not start to diverge even though the camera is turning clockwise around the scene at a quite fast rate.

The final test will examine how the number of frames in the focus stack affect the performance of the recurrent autoencoder. We will use the standard version trained on the small data without CoC. Figure 5.2 shows the training progress when training on a one up to a five-frame focus stack. The one frame extreme case achieves the highest



(a) MSE.

(b) Accuracy.

Figure 5.2: Recurrent AE validation metrics for training with different focus stack sizes.

accuracy thereby. Considering the test metrics in Table 5.4 the model which uses four frames has way better values than the other models which do not yield good test results. When testing, we give the same amount of frames as the models are trained on.

Table 5.4: Quantitative results for the recurrent autoencoder trained and tested with different focus stack sizes.

Model	MSE	RMS	Accuracy
1 Frame	4.8×10^{-2}	0.219	10.99
2 Frames	3.4×10^{-2}	0.183	13.12
3 Frames	5.2×10^{-2}	0.227	9.33
4 Frames	3.4×10^{-3}	0.058	69.02
5 Frames	2.7×10^{-2}	0.162	26.97

Further real-world tests on the Suwajanakorn dataset show too that the single frame model does not perform well (Figure 5.15). Up to four frames, the quality of the depth maps continuously increases. When giving five frames, we feel that the quality slightly

decreases compared to the four frame network. There are generally more and larger areas with incorrectly predicted depth. E.g. the background of the keyboard scene or the front part of the scene with the books are more off than the version with four frames. The test values from the table hereby correspond the Suwajanakorn results in the way that a four-frame focus stack seems to ideal for the recurrent autoencoder. Giving more than four input frames to the recurrent autoencoder thereby seems to exceed the maximum sequence length the architecture can memorize.

5.3 Discussion

In summary, considering the real-world depth predictions, the recurrent autoencoder architecture produces the most plausible results among all our CNNs. The PoolNet fails to generalize to the validation set in the first place. We believe that this network type cannot deal with the movement happening in our training set. Considering the LSTM-DDFF versions only the LSTM-DDFF which stacks the feature maps to a long sequence and uses a convolution to score the skip connections delivers good results. We think that stacking the encodings to a long sequence preserves more information than using a fully connected layer which severely reduces the data dimension. Additionally, using a convolution instead of only taking the output from the last encoder for the skip connection gives the model more flexibility resulting in better performance. In the end, however, we need to crop or downscale images since the LSTM requires a fixed resolution input. This leads to very undesirable artifacts in the output and reduces the quality in general. The recurrent autoencoder networks thus outperform all other models. The recurrent structure and the consideration of the input order make it very resistant to movement in the scene. However, it is not possible to determine which version of the model is the best in general. In some scenarios, the larger data shows better results whereas in other scenarios the smaller data is superior. The CoC supervision does not seem to affect the output a lot so it is not necessary to include a separate decoder. This saves parameters and makes the recurrent autoencoder a very lightweight model which is fast to train and fast to infer with. We also showed that a focus stack of four frames is ideal for this type of network. Furthermore, the recurrent AE can output consistent depth predictions for a longer sequence. We believe that by further tweaking, the performance of the recurrent autoencoder can be even improved and that it is possible to create the best version of the recurrent AE by finding the best training data parameters. We think that in general, the recurrent autoencoder can deal with very complex and noisy data and that it might also be a reasonable choice for other tasks involving the input of whole videos, e.g. for video classification.

We also notice that it is not possible to reliably conclude the real-scenario generaliza-

tion power from the test metric values determined by evaluating on the synthetic test set. As we can see, the standard DDFF manages to achieve a low test loss while having poor real-world results. Also, the recurrent autoencoders which seem to produce the best real-world depth predictions have a lower test score than the DDFF-LSTM Stack Cat-Conv. One issue might be that the synthetic test set which uses different objects and textures might not be challenging enough to properly distinguish the performance of the networks. Unlike real-world data, there is no camera shake, noise or focus breathing. In the end, we mostly rely on our perception to grade our models. As stated in the dataset section it is quite hard to capture real-world depth maps which themselves would be only estimations of the actual depth which would make evaluation metrics also unreliable and less meaningful.

In the end, we can say that our models can outperform the original DDFF by far which was not able to find suitable depth maps at all. Its pre-trained VGG base makes it more focused on object shapes than on the circle of confusion. Comparing with the complex four-step Video Depth-From-Defocus approach, ours is a simple, one step CNN structure which is end-to-end trainable. Since the dataset and the implementation of this work is not publicly available, we, unfortunately, could not directly compare it with ours. We are confident though that our approach can compete with theirs. A usual problem with deep learning is the requirement of a large, high quality labeled dataset. Such datasets are often hard to obtain in particular for very special needs like in our case where we require video data with focus ramping and ground truth depth data. Our training data is generated completely automatically and it is possible to generate arbitrary large datasets. No effort is required like it would be the case when having to record a large real-world dataset. Any lens or setting can be simulated in Blender and the ground truth depth is perfectly accurate unlike it would be the case when using RGB-D cameras. Yet the models do not have any issues to generalize to real-world data later because of the domain-independence of the CoC.

The limitation of our approach is that the performance of the networks depends on the setting of the recording. A setting with lots of reflective, transparent or single-color objects makes it hard to estimate the occurring CoC which is going to cause wrong depth predictions in the end. Furthermore, there is no notion of confidence for a certain pixel prediction. Another issue is that the CoC in the frames needs to be always clearly visible. This is often hard to achieve especially when dealing with larger distances where the CoC change is hardly distinguishable. Special camera optics are required to maintain a small depth of field even for large focus distances.

Figure 5.3: Colormap.

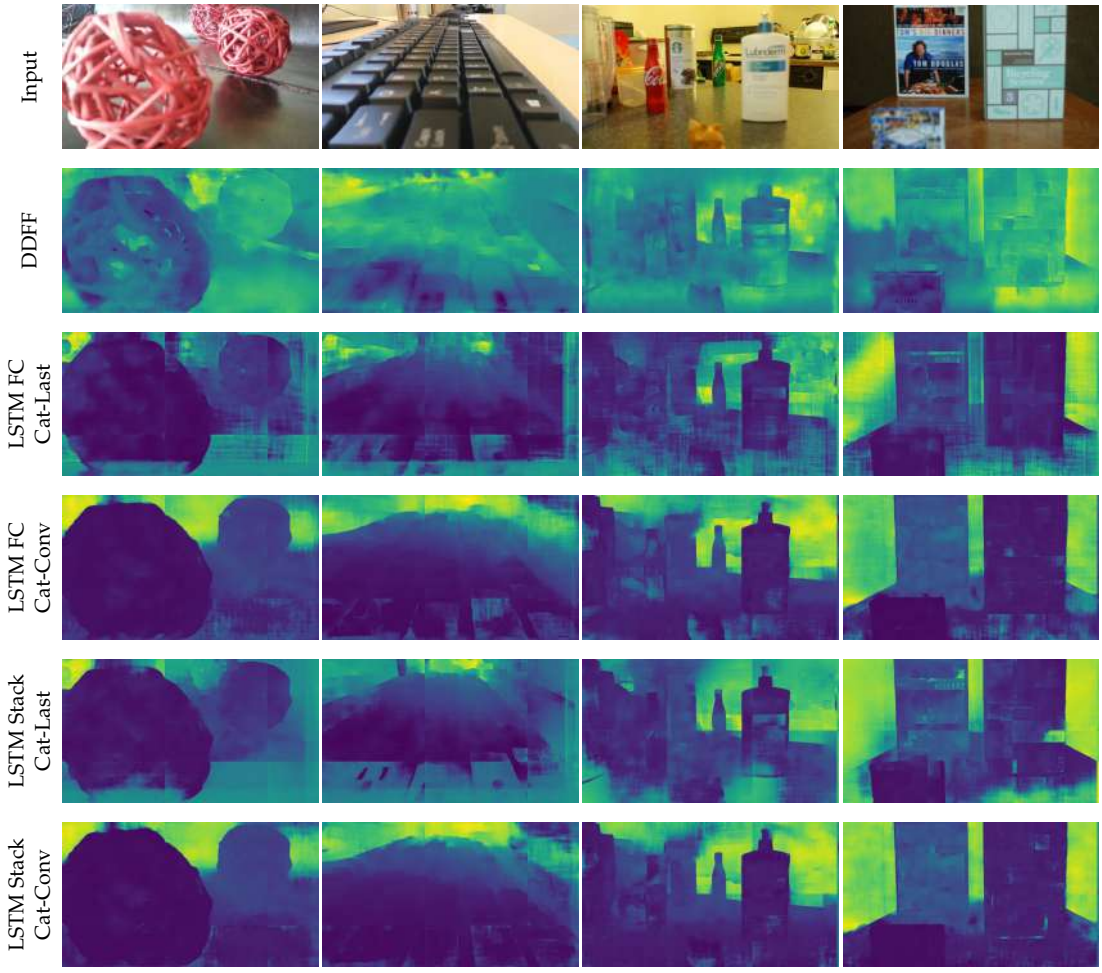


Figure 5.4: Comparison of the DDFF and the LSTM-DDFF variations on the Suwajanakorn test set.

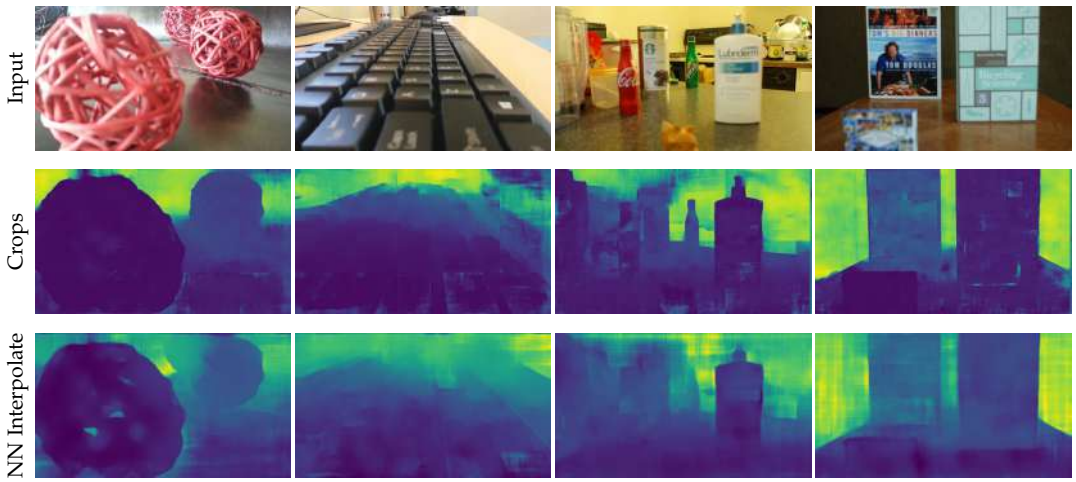


Figure 5.5: Comparison of subdivision by cropping and downscaling for the LSTM-DDFF Stack Cat-Conv.

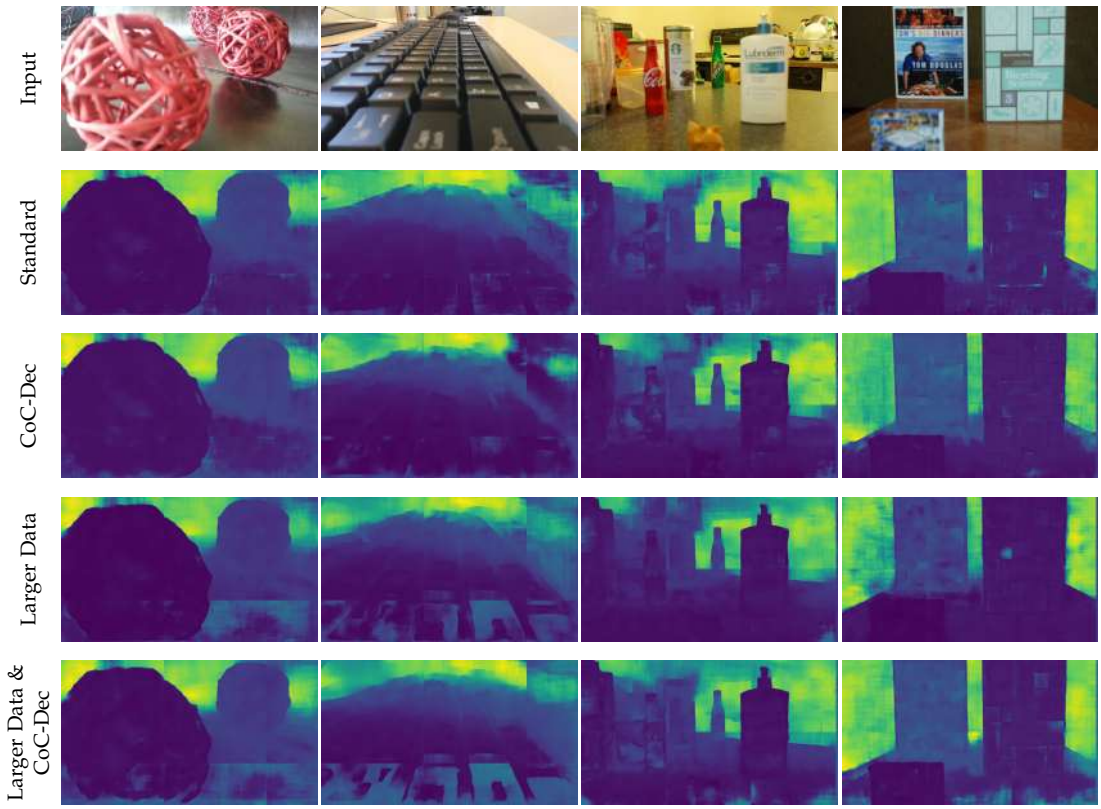


Figure 5.6: Tests for the LSTM-DDFF Stack Cat-Conv trained on the larger dataset and using CoC supervision.

Figure 5.7: Suwajanakorn tests for the recurrent autoencoder variations.

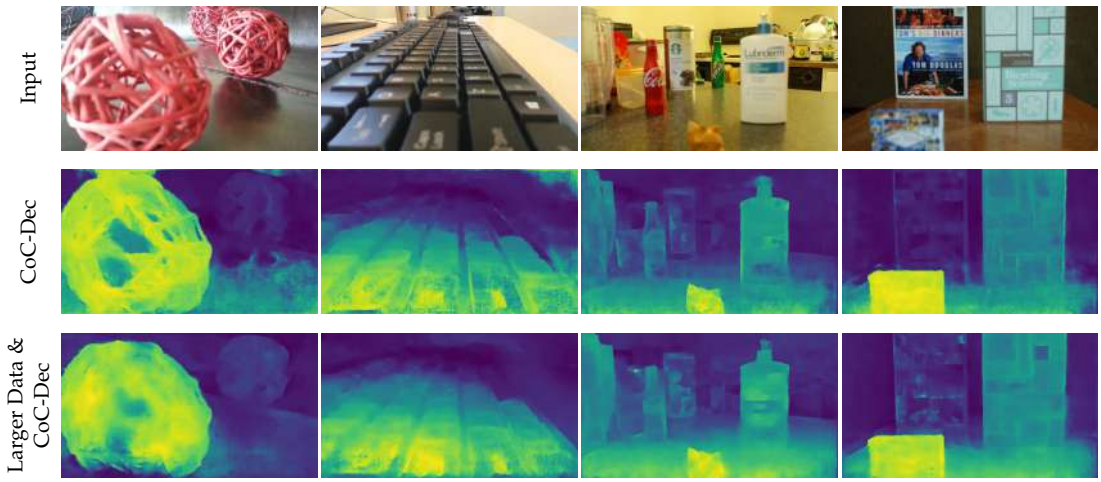


Figure 5.8: CoC output of recurrent autoencoders which have a CoC decoder. The CoC maps correspond to the depth frames in Figure 5.7.

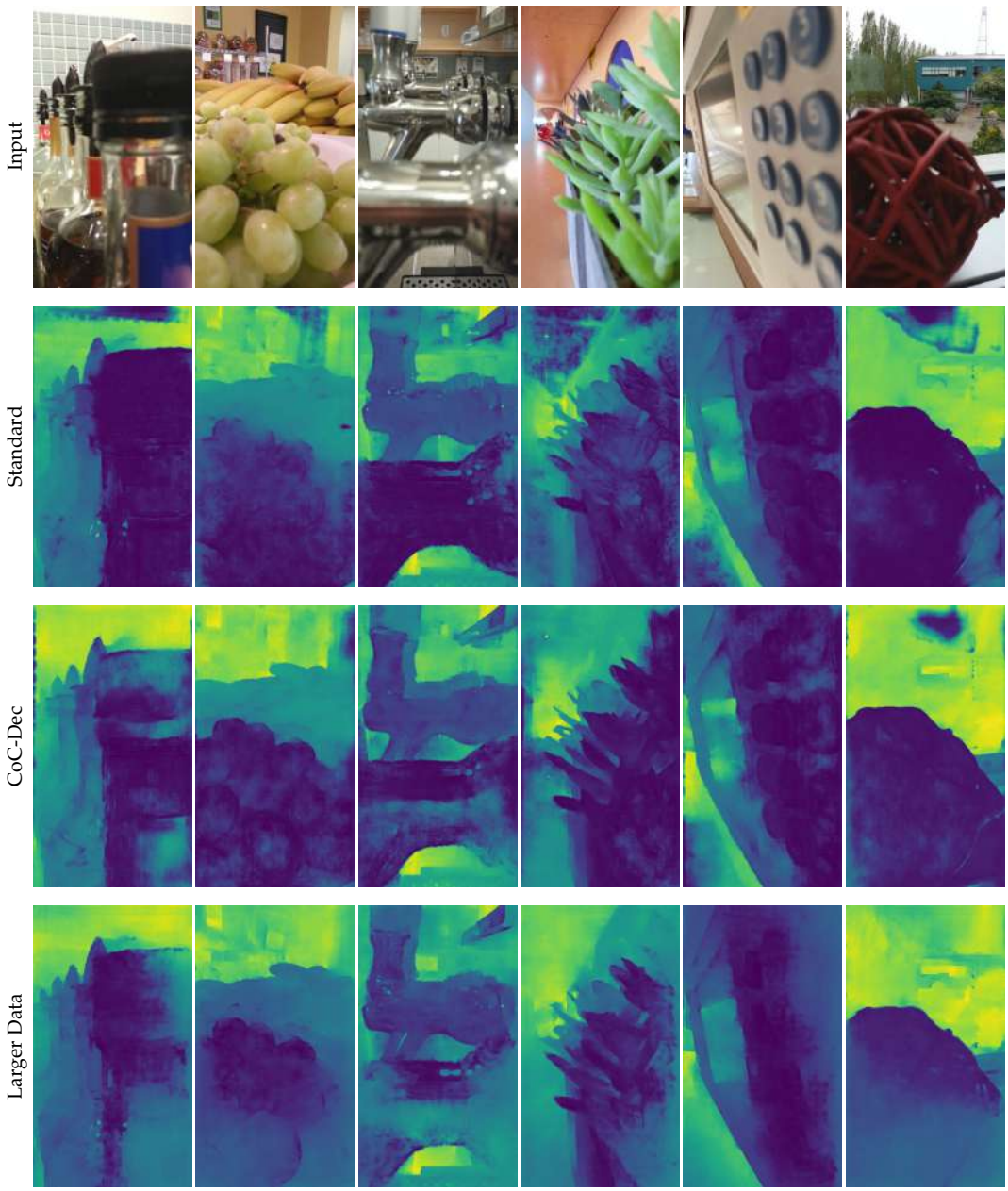


Figure 5.9: Tests on the portrait Suwajanakorn samples recorded by a mobile phone. Tests are performed for the recurrent autoencoder variations including training on the larger dataset and using CoC supervision. The Larger Data & CoC-Dec version is omitted here because of poor performance and space reasons.

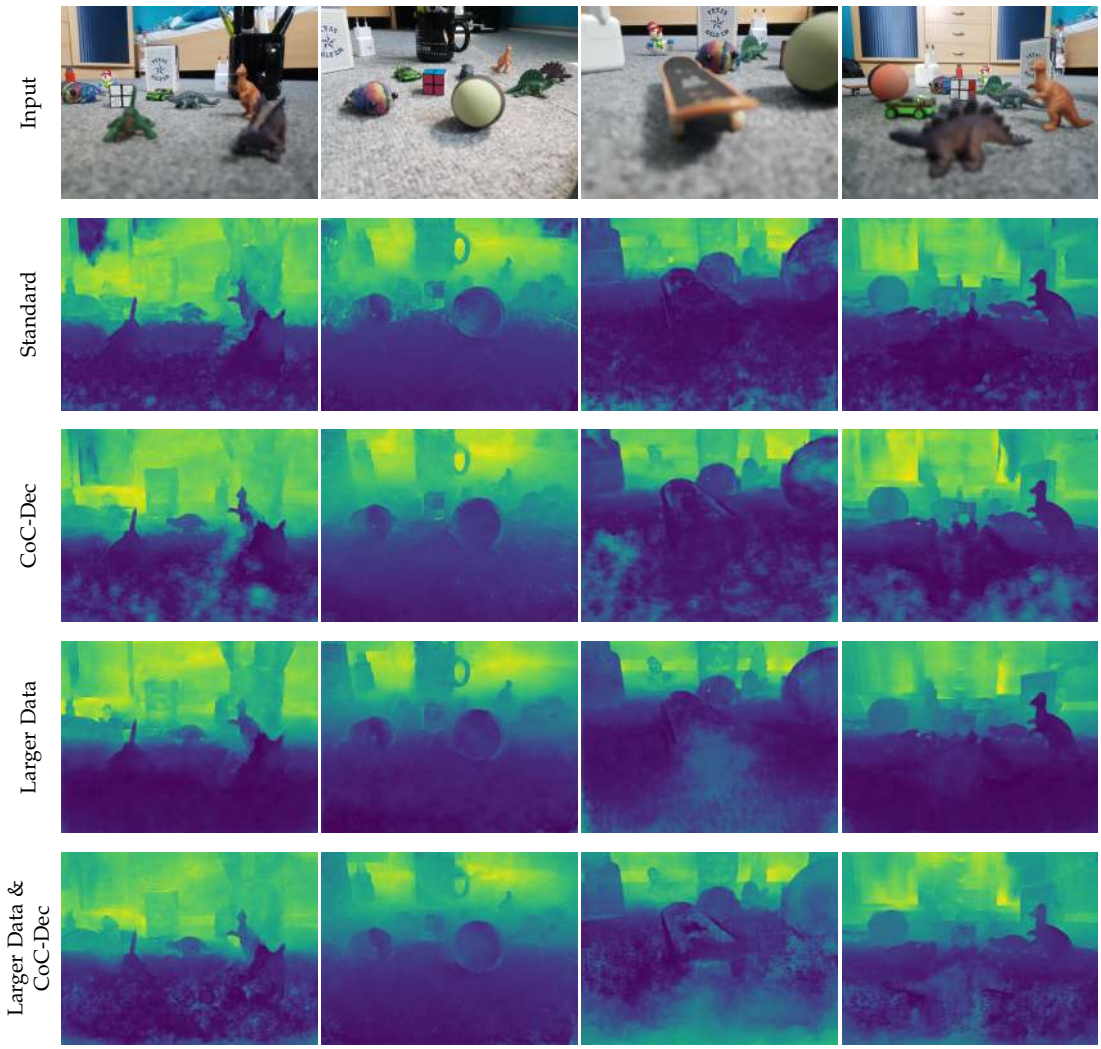


Figure 5.10: Recurrent AE tests for the carpet dataset.

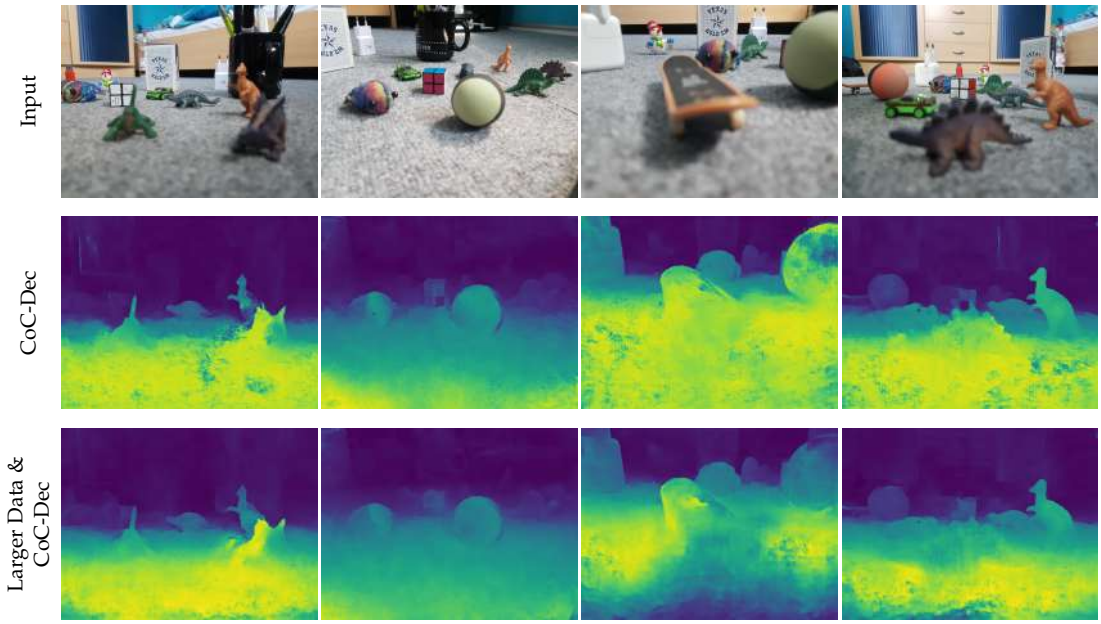


Figure 5.11: Recurrent AE CoC results for the carpet dataset.

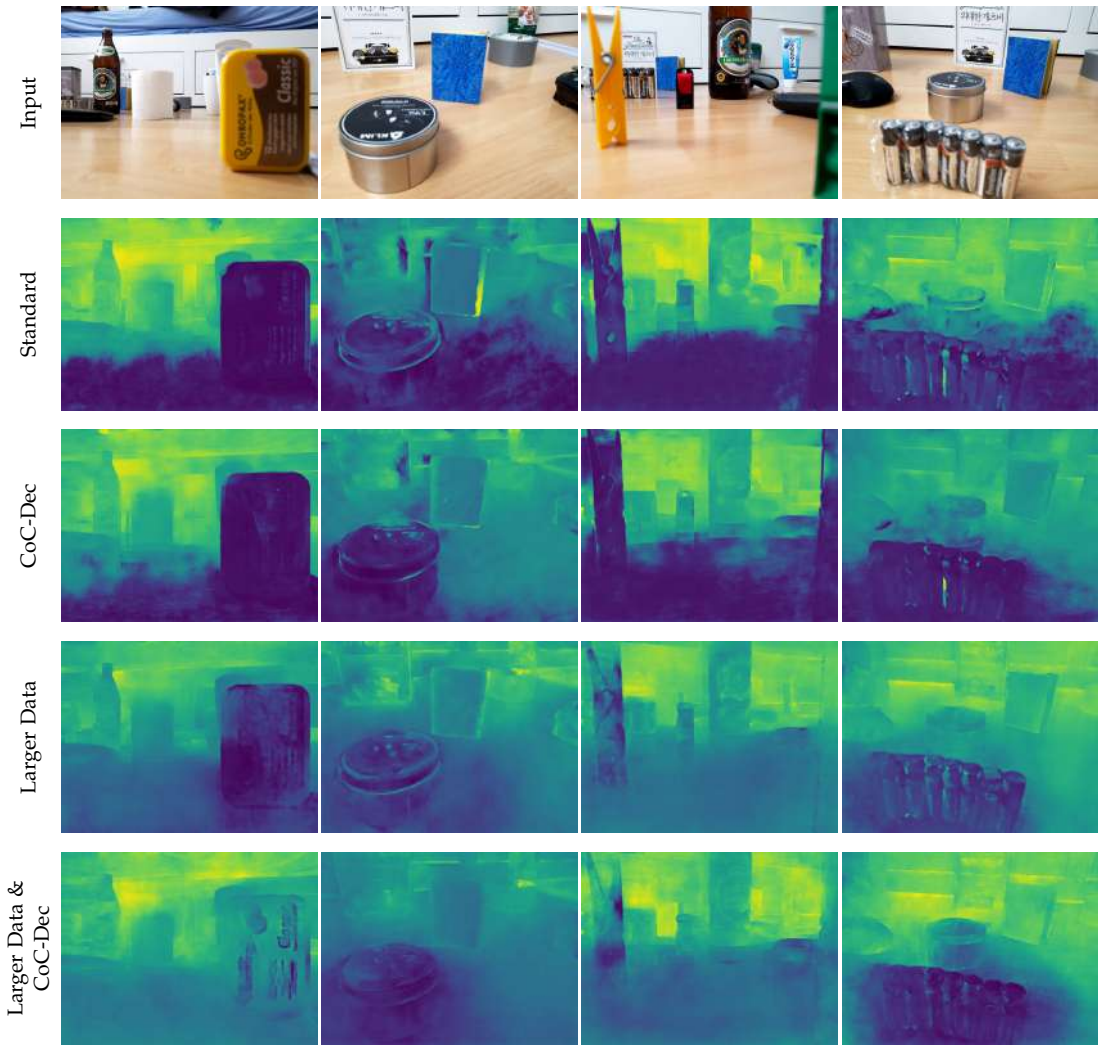


Figure 5.12: Recurrent AE tests for the wooden floor dataset.

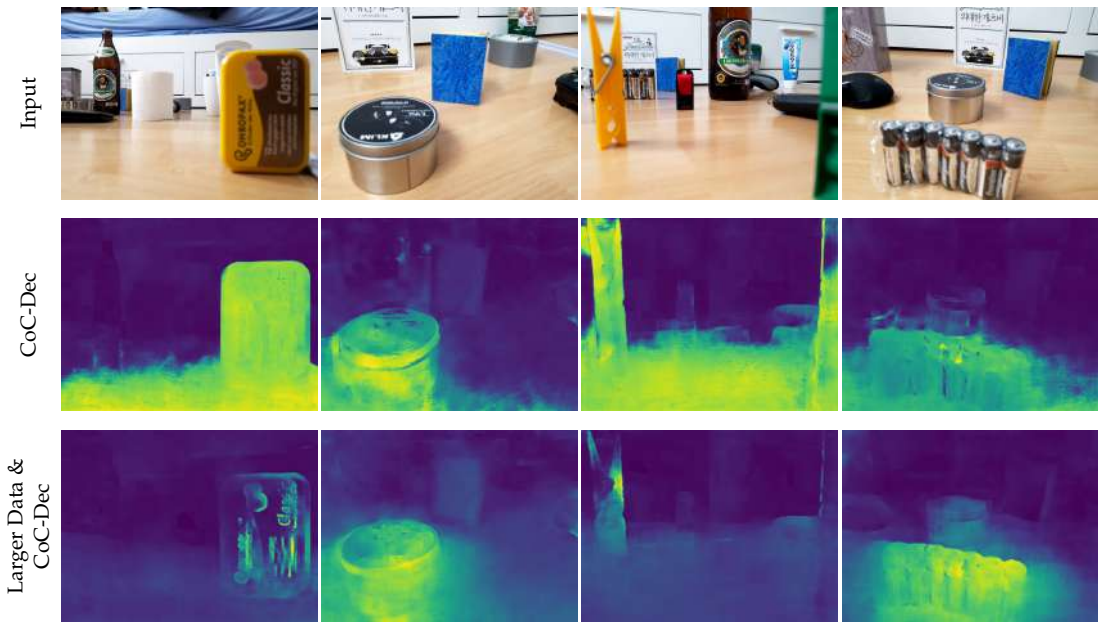


Figure 5.13: Recurrent AE CoC results for the wooden floor dataset.

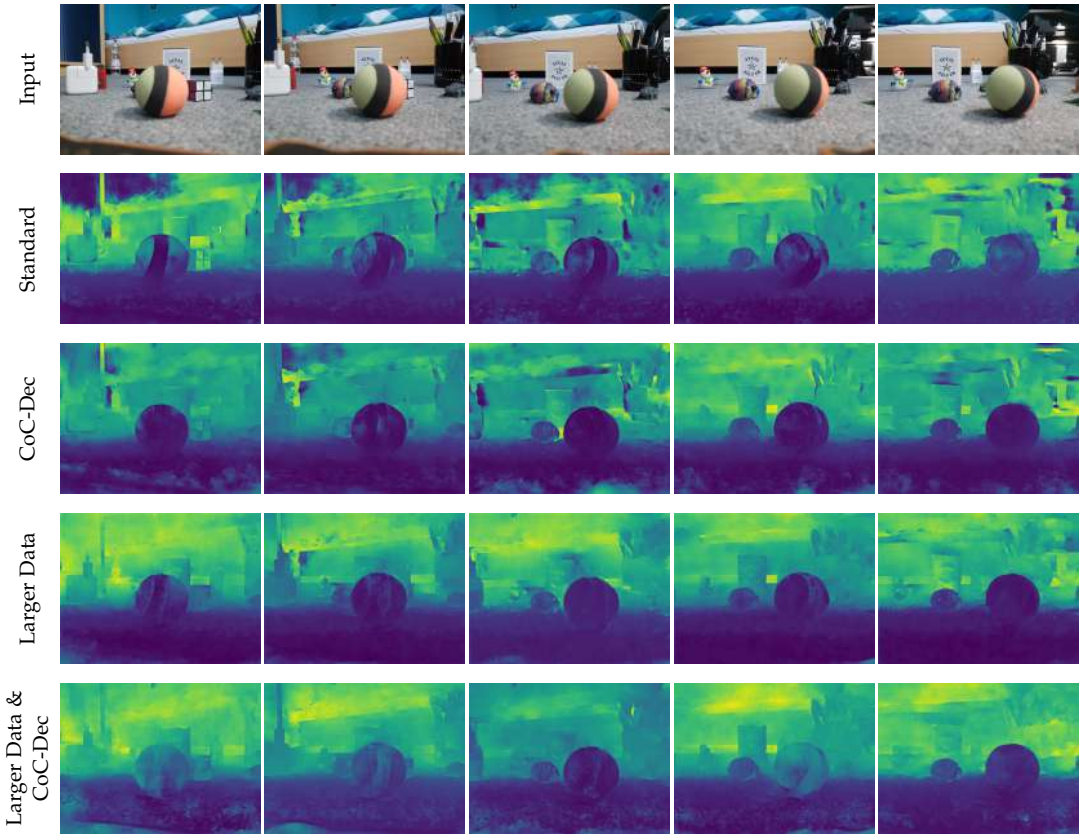


Figure 5.14: Consistency of multiple focus ramps in sequence for the recurrent AE on the carpet dataset.

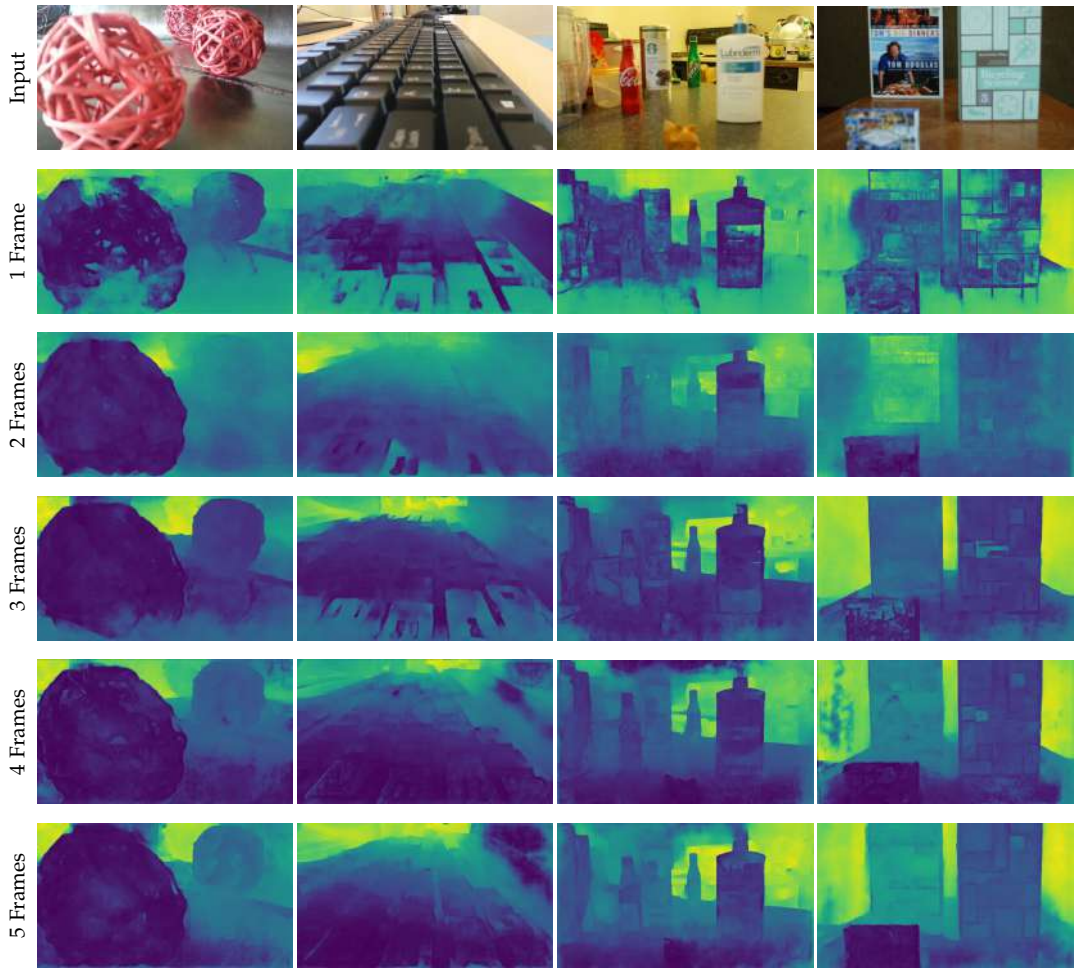


Figure 5.15: Tests for the recurrent AE trained and tested with different focus stack sizes.

6 Conclusion

Concluding our work, we introduced several CNN architectures which can predict plausible depth maps from videos where the focus distance is sweeping from a minimum to a maximum value. We showed how to collect real-world datasets with ground truth depth data using two developed Android applications running on a Samsung Galaxy S7 phone and an RGB-D camera. We performed proper intrinsic and extrinsic calibration for the phone and the RGB-D color sensor using checkerboard captures and the Camera Calibration Toolbox for Matlab. We registered the depth maps to the color recordings using the parameters obtained from the calibration procedure. To synchronize the Android and RGB-D video streams we sent the PC system time to the Android device via a socket connection and calculated timestamps relative to this common clock. We created a small real-world dataset recorded in an office at the university. Due to the lens shift and the depth registration process, there are lots of missing depth values in the ground truth. Reflective, dark or transparent objects also cause unknown depth areas. Besides real-world data, we developed an automatic dataset generation mechanism using several sources for 3D models and textures and the Blender Cycles render engine to produce realistic render results along with perfect depth ground truth. Arbitrary random sequences can be created and rendered without the need of any user interaction. We further created a dataset with small, textured objects and one with large untextured ones. Finally, we evaluated the performance of all proposed network configurations and found that the recurrent autoencoder behaves best. Besides the recurrent AE, the LSTM-DDFF which stacks feature maps to a long sequence and uses a convolution for the skip connection synthesis provided good results. However, the input resolution is fixed and inputting arbitrarily sized frames requires cropping or downscaling which degrades the quality in the form of artifacts. The recurrent AE produces consistent and plausible depth maps for static as well as for very dynamic scenes while requiring only four frames per focal stack. It was however not possible to find the perfect configuration or dataset for the recurrent autoencoder. For both the models trained on the small objects and the ones trained on the small and large objects, there were situations where one outperformed the outer and vice versa. Furthermore, the successful inference of depth maps is dependent on clearly distinguishable defocus blur in each focus stack frame. Too far distances, large, untextured areas and reflections thus negatively affect the predictions.

In future work, we would like to do further experiments with the recurrent autoencoder and the synthetic dataset used for training to determine the conditions to build one best recurrent autoencoder depth prediction system. Besides using the CoC for depth inference we also believe that the recurrent autoencoder is a reasonable choice for structure from motion (SfM). Its ability to deal exceptionally well with scene movement makes it a promising system for SfM. In the end, one could combine the CoC based prediction with the structure-from-motion based prediction to further improve the depth inference quality. We believe that the recurrent autoencoder has the potential to use both methods to achieve superior performance.

List of Figures

List of Tables

Bibliography

[AD18] M. Aittala and F. Durand. "Burst Image Deblurring Using Permutation Invariant Convolutional Neural Networks." In: *The European Conference on Computer Vision (ECCV)*. Sept. 2018.

[Bal19] S. Balaban. "Deep learning and face recognition: the state of the art." In: *CoRR* abs/1902.03524 (2019). arXiv: 1902.03524.

[Bar+09] C. Barnes, E. Shechtman, A. Finkelstein, and D. B. Goldman. "PatchMatch: A Randomized Correspondence Algorithm for Structural Image Editing." In: *ACM Transactions on Graphics (Proc. SIGGRAPH)* 28.3 (Aug. 2009).

[Ber+14] M. Berning, D. Kleinert, T. Riedel, and M. Beigl. "A study of depth perception in hand-held augmented reality using autostereoscopic displays." In: *2014 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*. Sept. 2014, pp. 93–98. DOI: 10.1109/ISMAR.2014.6948413.

[Bho19] A. Bhoi. "Monocular Depth Estimation: A Survey." In: *CoRR* abs/1901.09402 (2019). arXiv: 1901.09402.

[BKC15] V. Badrinarayanan, A. Kendall, and R. Cipolla. "SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation." In: *CoRR* abs/1511.00561 (2015). arXiv: 1511.00561.

[BMP17] F. Basso, E. Menegatti, and A. Pretto. "Robust Intrinsic and Extrinsic Calibration of RGB-D Cameras." In: *CoRR* abs/1701.05748 (2017). arXiv: 1701.05748.

[Bou01] J.-Y. Bouguet. "Camera calibration toolbox for matlab." In: http://www.vision.caltech.edu/bouguetj/calib_doc/htmls/ref.html. 2001.

[Bra00] G. Bradski. "The OpenCV Library." In: *Dr. Dobb's Journal of Software Tools* (2000).

[Cas+18] V. Casser, S. Pirk, R. Mahjourian, and A. Angelova. "Depth Prediction Without the Sensors: Leveraging Structure for Unsupervised Learning from Monocular Videos." In: *CoRR* abs/1811.06152 (2018). arXiv: 1811.06152.

[Cha+17] C. R. A. Chaitanya, A. S. Kaplanyan, C. Schied, M. Salvi, A. Lefohn, D. Nowrouzezahrai, and T. Aila. "Interactive Reconstruction of Monte Carlo Image Sequences Using a Recurrent Denoising Autoencoder." In: *ACM Trans. Graph.* 36.4 (July 2017), 98:1–98:12. ISSN: 0730-0301. DOI: 10.1145/3072959.3073601.

[Clo19] ClockworkMod LLC 2019. *Vysor*. Version 2.1.7. https://www.vysor.io/. May 13, 2019.

[Com17] B. O. Community. *Blender - a 3D modelling and rendering package*. Blender Foundation. Stichting Blender Foundation, Amsterdam, 2017.

[DV16] V. Dumoulin and F. Visin. "A guide to convolution arithmetic for deep learning." In: *ArXiv* abs/1603.07285 (2016).

[GBC16] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[Goo19a] Google LLC 2019. *Android Camera2Basic Sample*. https://github.com/googlesamples/android-Camera2Basic. Jan. 31, 2019.

[Goo19b] Google LLC 2019. *Android Camera2Video Sample*. https://github.com/googlesamples/android-Camera2Video. Jan. 31, 2019.

[GRS19] GRSites. *Free Background Textures Library*. http://www.grsites.com/archive/textures/. Aug. 7, 2019.

[GVS18] S. Giancola, M. Valenti, and R. Sala. "State-of-the-Art Devices Comparison." In: *A Survey on 3D Cameras: Metrological Comparison of Time-of-Flight, Structured-Light and Active Stereoscopy Technologies*. Cham: Springer International Publishing, 2018, pp. 29–39. ISBN: 978-3-319-91761-0. DOI: 10.1007/978-3-319-91761-0_3.

[Haz+18] C. Hazirbas, S. G. Soyer, M. C. Staab, L. Leal-Taixé, and D. Cremers. "Deep Depth From Focus." In: *Asian Conference on Computer Vision (ACCV)*. Dec. 2018. eprint: 1704.01085.

[Hin+12] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. "Improving neural networks by preventing co-adaptation of feature detectors." In: *CoRR* abs/1207.0580 (2012). arXiv: 1207.0580.

[HLW16] G. Huang, Z. Liu, and K. Q. Weinberger. "Densely Connected Convolutional Networks." In: *CoRR* abs/1608.06993 (2016). arXiv: 1608.06993.

[HS97] S. Hochreiter and J. Schmidhuber. "Long Short-Term Memory." In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735.

[IS15] S. Ioffe and C. Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." In: *CoRR* abs/1502.03167 (2015). arXiv: 1502.03167.

[Joo19] Joost. *Texture.ninja*. https://texture.ninja/. Aug. 7, 2019.

[KB14] D. P. Kingma and J. Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: 1412.6980 [cs.LG].

[KRT16] H. Kim, C. Richardt, and C. Theobalt. "Video Depth-From-Defocus." In: *International Conference on 3D Vision (3DV)*. Oct. 2016, pp. 370–379. DOI: 10.1109/3DV.2016.46.

[KSH12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1097–1105.

[LSD14] J. Long, E. Shelhamer, and T. Darrell. "Fully Convolutional Networks for Semantic Segmentation." In: *CoRR* abs/1411.4038 (2014). arXiv: 1411.4038.

[Mil91] D. L. Mills. "Internet time synchronization: the network time protocol." In: *IEEE Transactions on Communications* 39.10 (Oct. 1991), pp. 1482–1493. ISSN: 0090-6778. DOI: 10.1109/26.103043.

[Pas+17] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. "Automatic Differentiation in PyTorch." In: *NIPS Autodiff Workshop*. 2017.

[PC82] M. Potmesil and I. Chakravarty. "Synthetic Image Generation with a Lens and Aperture Camera Model." In: *ACM Trans. Graph.* 1.2 (Apr. 1982), pp. 85–108. ISSN: 0730-0301. DOI: 10.1145/357299.357300.

[Reh+17] E. Rehder, C. Kinzig, P. Bender, and M. Lauer. "Online stereo camera calibration from scratch." In: *2017 IEEE Intelligent Vehicles Symposium (IV)*. June 2017, pp. 1694–1699. DOI: 10.1109/IVS.2017.7995952.

[RFB15] O. Ronneberger, P. Fischer, and T. Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation." In: *CoRR* abs/1505.04597 (2015). arXiv: 1505.04597.

[SHS15] S. Suwajanakorn, C. Hernandez, and S. M. Seitz. "Depth from focus with your mobile phone." In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2015, pp. 3497–3506. DOI: 10.1109/CVPR.2015.7298972.

[SZ14] K. Simonyan and A. Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2014. arXiv: 1409.1556 [cs.CV].

[Tan+16] S. Tang, Q. Zhu, W. Chen, W. Darwish, B. Wu, H. Hu, and M. Chen. "Enhanced RGB-D Mapping Method for Detailed 3D Indoor and Outdoor Modeling." In: *Sensors* 16 (Sept. 2016), p. 1589. DOI: 10.3390/s16101589.

[Tan+18] C. Tan, F. Sun, T. Kong, W. Zhang, C. Yang, and C. Liu. "A Survey on Deep Transfer Learning." In: *CoRR* abs/1808.01974 (2018). arXiv: 1808.01974.

[Tuy19] R. Tuytel. *Texture Haven*. https://texturehaven.com/textures/. Aug. 7, 2019.

[WCH19] Z. Wang, J. Chen, and S. C. H. Hoi. "Deep Learning for Image Super-resolution: A Survey." In: *CoRR* abs/1902.06068 (2019). arXiv: 1902.06068.

[Zaa19] G. Zaal. *HDRI Haven*. https://hdrihaven.com/hdris/. Aug. 7, 2019.

[ZJ16] Q. Zhou and A. Jacobson. "Thingi10K: A Dataset of 10,000 3D-Printing Models." In: *arXiv preprint arXiv:1605.04797* (2016).