Technische Universität München

Interdisciplinary Project

INSTITUTE FOR HUMAN-MACHINE COMMUNICATION
TECHNISCHE UNIVERSITÄT MÜNCHEN
Univ.-Prof. Dr.-Ing. habil. G. Rigoll

Fundamental Matrix Estimation from Multi-View Silhouette Images using Deep Learning

Philip Müller

Advisor:	Dr.-Ing. Mohammadreza Babaee
Started on:	01.10.2018
Handed in on:	31.12.2018

Abstract

Fundamental matrix estimation is an important task in computer vision. Existing approaches mostly rely on texture images for solving this task. When only silhouette images are available most methods give poor results or need additional input data like temporal sequences of images. In this project an approach is investigated which takes a single silhouette image pair as input for estimating the fundamental matrix. It is based on deep learning, but instead of using a single network, a GAN is trained during model training as well as during the prediction phase to learn the relation between the first and the second image of a sample. Then the internal layer activations of the GAN are used as input to another neural network, which was trained on a dataset and now regresses the fundamental matrix. This work describes the creation of training datasets, the model architecture and two different training procedures. The approach is analyzed in experiments and possible extensions to it are discussed.

Contents

1

Introduction

In multi-camera vision an important task is to estimate the parameters of the cameras in a scene. These parameters include camera intrinsic parameters as well as extrinsic parameters, which describe the relative positions and rotations of the cameras. For stereo vision this information can be encoded in the fundamental matrix. This means that the matrix contains rich information about the relation of the two cameras in a stereo vision scene and thus is very valueable for many computer vision applications like camera calibration and localization, image rectification, depth estimation, and 3D reconstruction. [8] To estimate the fundamental matrix, images taken from the two cameras can be used. Depending on the used method one or more images per camera are needed. In this project a silhouette image pair is used, which only needs one image per camera as input. These images are binary images and do not contain any texture. For texture images there alreday exist different techniques to perform the estimation: There are classical approaches, which first find matching points in the two images and then try to solve an optimization problem based on these points. Then there are also deep learning based approaches, which learn the fundamental matrix end-to-end. Estimating the fundamental matrix for silhouette images is a lot harder than when textures are available as there is much less information that can be used as input. Thus, using the approaches for texture images on silhouette images would lead to poor results. This is why there also exist methods specialized for silhouette images. But these often require additional input like temporal sequences of images instead of a single image per camera. [1] To be able to estimate the fundamental matrix only given the silhouette image pair, a different approach is investigated in this project. The approach is based on deep learning but does not use a single neurol network to regress the matrix. Instead it uses a GAN that was pre-trained at training time and then learns the relation between the two images at prediction time. The learned information in the GAN is then used to regress the fundamental matrix using another neural network that was trained on a silhoutte image fundamental matrix dataset.

[1] For more details see chapter 2.

1.1 Goal of this Project

The goal of this project is to develop, implement, train, and analyze a neural network model and training procedure which can regress the fundamental matrix in a stereo-view scene given two silhouette images. Each training sample consists of the two silhouette images and a fundamental matrix. The silhouette images are binary images of some object or environment in the same scene taken with different camera settings. The corresponding fundamental matrix describes the relation of the two camera settings. [2]

1.2 Approach

The model consists of three neural networks: the generator, the discriminator, and the regressor. Generator and discriminator together form a GAN (see [5]) that is trained independently for each silhouette image pair to generate the second image given the first one. The regressor then uses the internal features of the GAN, which have been learned during GAN training for some sample, to estimate the fundamental matrix of that given sample. This means that during model training the GAN and the regressor are trained. To predict the fundamental matrix for a given image pair only the GAN is trained on this pair. The features from the GAN are then given to the regressor, which uses the weights learned during model training to regress the fundamental matrix. [3]

1.3 Structure of this Document

After related work has been mentioned in chapter 2, chapter 3 covers the structure and creation of training datasets. Then, in chapter 4, the neural network model architectures are described in detail. This chpater is followed by chapter 5, which covers the training procedure of the model and explains how the model can be used for prediction. In chapter 6 the performed experiments for studying the model are described. This also includes the results of the experiments and their interpretation as well as an analysis of the problems of the models. The document then closes with chapter 7, which covers possible optimizations and extensions, and includes a final conclusion.

[2] See chapter 3 for details on the dataset.

[3] See chapter 4 for details on the neural network models and chapter 5 for details on training and prediction.

2

Related Work

There are several other works that try to solve problems similar to the goals of this project. First of all, there is [11], which also tries to regress the fundamtal matrix using a deep neural network. In contrast to this project, [11] does not regress the matrix using silhouette images, but it uses input pairs of normal images with texture. It just uses a simple neural network without a GAN and without complex training methods because with textured images the learning problem is easier. There is also [8], which is based on [11] and attacks exactly the same learning problem. But in [8] a siamese neural network architecture is investigated additionally and a slightly more complex fundamental matrix reconstruction layer is used. Then there is [4], which like [11] uses textured images as input. But it regresses the 2D homography from the image pair instead of the fundamental matrix, and thus its use is limited to some constrained scenes. [2], like this project, uses silhouette images to regress the fundamental matrix. But unlike this project, which only uses a single silhouette images pair, it needs a temporal sequence of images pairs to estimate the matrix. Provided with this sequence, it uses motion barcodes for the regression instead of a neural network. Besides these complex methods, there are also the classical approaches to estimate the fundamental matrix, as described in [6, pp. 281-285]. These approaches first need to find corresponding points in both images and then estimate the fundamental matrix using some optimization method on the found points. They do not use a neural natwork. One of such methods is used to compute the baselines in this project (see chapter 6). Beside the listed works there are a lot of different works that also tackle similar problems. The cited works are some of the most relevant ones and the ones most related to this project.

3

Datasets

This chapter describes the creation of the datasets used for training, validation, and testing. These are sets of samples (x, y) where each x consists of two silhouette images and the ground truth y is represented by a 3×3 matrix. The silhouette images are binary images of width and height 128, which means that x has dimension $128 \times 128 \times 2$. Both images within a sample were taken from the same scene but with different camera parameters, which includes the camera extrinsics (the position and rotation of the camera in the scene) and camera intrinsics (the properties of the pinhole camera model). The 3×3 matrix y is the fundamental matrix that describes the relation of the camera parameters of the two images in the sample to each other. Images of one sample may have been taken from the same or a different scene as the images of another sample.

3.1 Image Generation

To create the fundamental matrix dataset, first silhouette images with their corresponding camera parameters are needed. Within the scope of this project three possibilities to procure those silhouette images have been studied, which are described in the following sections.

3.1.1 Silhouette Image Dataset

This is the most straightforward way to get silhouette images: An available dataset that already contains silhouette image samples is used. These samples of single silhouette images are required to be labelled with the corresponding camera parameters or with the camera projection matrices, which are based on the camera intrinsics and extrinsics. The images should all be taken from the same scene but with different camera parameters.

3.1.2 Texture Image Dataset and Preprocessing

Another way to get silhouette images is to use a texture image dataset containing textured images with greyscale or color channels. Like in the silhouette images dataset,

each image needs to be labelled with its camera parameters or projection matrix. They should also all be taken from the same scene with different parameters. To get silhouette images from those texture images, they need to be processed in the following way: If the provided images are colored, they first have to be converted to greyscale. After that or if the images have already been provided in greyscale, a thresholding operation is applied to each image to convert it to a binary image. Because of shadows and other texture related shades in the image, the results might not represent perfect silhouettes and might be very disturbed. To reduce this effect the morphological operations `OPEN` and `CLOSE` are applied in an alternating fashion for several iterations.

3.1.3 Synthetic Image Generation

The third method to produce silhouette images is synthetic image generation. This method does not require any dataset with tagged images. Instead it only needs a 3D model and a rendering engine. As rendering engine the open source tool Open3D [12] is used. 3D models are provided in the `ply` file format.

To create a synthetic image dataset a single 3D model is loaded from its `ply` file into the Open3D scene. The visualizer of Open3D, which is used to render the images, is configured to produce images in the required image dimension 128×128. The background color is set to black, and the loaded model mesh is painted white uniformly. No normals are defined for the mesh, and no special shader or rendering style is selected. With this configuration the renderer does not consider any lighting and the rendered images will not contain different shades. Each pixel is either colored white when the mesh occludes the pixel or it is colored black when it does not. [1] Thus, this setting can be used to create silhouette images of the scene where the silhouettes are colored white and the rest of the image is black.

After the model has been prepared a set of camera parameters, consisting of camera intrinsics and extrinsics, is created based on predefined ranges. The ranges depend on the size of the model, its center position, and on which parameters should be animated. [2] Then for each of these defined camera parameters the camera is configured accordingly, and a single image is rendered. This rendered image is then captured and saved to the image dataset, where it is labelled with the camera parameters that where used for its creation. For the created images a small processing step is needed. Although these are already silhouette images, there may be pixels that neither have the value 0 nor 255 but instead have values in between. This happens due to interpolation occurring during image saving or rendering. To fix this a thresholding operation with threshold 127 is applied to each image.

[1] Note that this is not completely true, as there might be cases when intermediate values are produced by some interpolation that happens when saving the images. But the principle of silhouette generation stays the same and the values are corrected by some post-processing.

[2] E.g. it could be decided to only animate the camera extrinsics but not the intrinsics. Then the intrinsics would be set to constant values and for the extrinsics a set of different values would be provided by animating the parameters within given ranges.

3.2 Combination to Fundamental Matrix Dataset

After the silhouette images for some scene and their corresponding camera parameters have been prepared, these samples have to be combined into the fundamental matrix dataset, which is needed for training. Each fundamental matrix sample is based on two image samples. This means that for creating the fundamental matrix dataset any possible combination of two image samples could be considered. But any combination that would lead to the fundamental matrix being zero is excluded. [3] So for example combinations of images with themselves are not included in the dataset. It is very important that all combined images come from the same scene, and that the values of the camera parameters are all at same scale. If this would not be the case, then no meaningfull fundamental matrix could be computed.

3.2.1 Fundamental Matrix Computation

For each of these combinations of two images the fundamental matrix is computed using the camera parameters of both images in the following way: First the projection matrix $\boldsymbol{P_i}$ is computed for each of the images $i \in \{a, b\}$:

$$\boldsymbol{P_i} = \boldsymbol{K_i}[\boldsymbol{R_i}|\boldsymbol{t_i}] \tag{3.1}$$

where $\boldsymbol{K_i}$ is the camera intrinsic matrix of image i and $[\boldsymbol{R_i}|\boldsymbol{t_i}]$ are the camera extrinsics consisting of rotation $\boldsymbol{R_i}$ and translation $\boldsymbol{t_i}$. [6, p. 156]

After that, the fundamental matrix $\boldsymbol{F}$ is computed using the projection matrices $\boldsymbol{P_a}$ and $\boldsymbol{P_b}$ of the two images. There are different ways to compute $\boldsymbol{F}$. In this project the following variant is used, which is based on the Matlab implementation [3]: [4]

$$\boldsymbol{F} = \begin{pmatrix} \det\begin{bmatrix}(\boldsymbol{P_a})_{2:3,:}\\(\boldsymbol{P_b})_{2:3,:}\end{bmatrix} & \det\begin{bmatrix}(\boldsymbol{P_a})_{3:1,:}\\(\boldsymbol{P_b})_{2:3,:}\end{bmatrix} & \det\begin{bmatrix}(\boldsymbol{P_a})_{1:2,:}\\(\boldsymbol{P_b})_{2:3,:}\end{bmatrix} \\ \det\begin{bmatrix}(\boldsymbol{P_a})_{2:3,:}\\(\boldsymbol{P_b})_{3:1,:}\end{bmatrix} & \det\begin{bmatrix}(\boldsymbol{P_a})_{3:1,:}\\(\boldsymbol{P_b})_{3:1,:}\end{bmatrix} & \det\begin{bmatrix}(\boldsymbol{P_a})_{1:2,:}\\(\boldsymbol{P_b})_{3:1,:}\end{bmatrix} \\ \det\begin{bmatrix}(\boldsymbol{P_a})_{2:3,:}\\(\boldsymbol{P_b})_{1:2,:}\end{bmatrix} & \det\begin{bmatrix}(\boldsymbol{P_a})_{3:1,:}\\(\boldsymbol{P_b})_{1:2,:}\end{bmatrix} & \det\begin{bmatrix}(\boldsymbol{P_a})_{1:2,:}\\(\boldsymbol{P_b})_{1:2,:}\end{bmatrix} \end{pmatrix} \tag{3.2}$$

3.3 Data Normalization

3.3.1 Fundamental Matrix

After the fundamental matrices have been computed they are normalized to simplify the learning task. Here it is important to use the same normalization as the one that

[3] These combinations are excluded because the normalization of the matrix would fail as its norm would be 0, and diving by a zero norm is not defined.

[4] This variant is used because after implementing this computation in Python the correctness of the code could easily be verified by comparing the results against the Matlab implementation.

is used in the neural network model of the regressor. See section 4.3.5 for details on normalization.

3.3.2 Silhouette Image Pair

The images are normalized as well. As they are binary silhouette images, which only have the pixel values 0 and 255, this normalization is trivial: Each pixel value is divided by 255 so that all pixels either have the value 0 or 1 after normalization. The image normalization is done while loading the images. A normalization which includes the mean or standard deviation of one or more images is not needed as these images are binary images and thus the input data is already maximal simplified.

3.4 Mixing Multiple Datasets

By following the previous steps a dataset for a single scene can be created. But, if the neural network model should also be able to work on differents scenes, then the created dataset is too small and too focused on one single scene. Any model trained with this dataset would be overfitting on this scene. So, to be applicable for real world problems, a large dataset with images from many different scenes is required. To create such a dataset the previous steps need to be followed to create many small datasets for different scenes by using different 3D models or different image datasets. In the end all these small datasets are concatenated to a large dataset, which then contains images from different scenes.

4

Model

After the dataset creation has been covered, this chapter explains the model used in the project. This includes the description of the neural network architectures and the explanation of the design decisions that were made for each of the networks. Details about the training and prediction methods which are used with this model are not covered by this chapter. Those are described in chapter 5.

The basic model architecture consists of three neural networks which are connected to each other: the generator, the discriminator and the regressor. Generator and discriminator together form a GAN (see [5]) with a single channeled image as the input and another single channeled image as output of the generator. This GAN is used to learn the relation between the two images of a sample and thus is trained with each sample independently. Like [7] this is a image-to-image GAN, but the architecture details differ as the model of this project is not a conditional GAN like in [7]. Also, the generator used in this project is multi headed with one head being the normal output for the generated image. The other outputs represent internal features of the GAN. These should, after the GAN has been trained, contain information about the image relation. They are connected to the regressor, which then regresses the fundamental matrix. Figure 4.1 shows the top-level architecture of the model, including generator, discriminator and regressor. In the following sections the three neural networks are described in detail.

4.1 Generator

The generator model is the first half of the GAN and is conceptually based on the U-Net architecture (see [9]). It starts with a single $128 \times 128 \times 1$ image as input which is the first image in the image pair of a sample. This input is then passed through the downsampling blocks which reduce the image size while increasing the number of channels. The downsampled features are then passed to the bottleneck block and then upsampled again. During upsampling the image size is increased while the number of channeles is reduced. In the end a single channel convolution is applied before the image

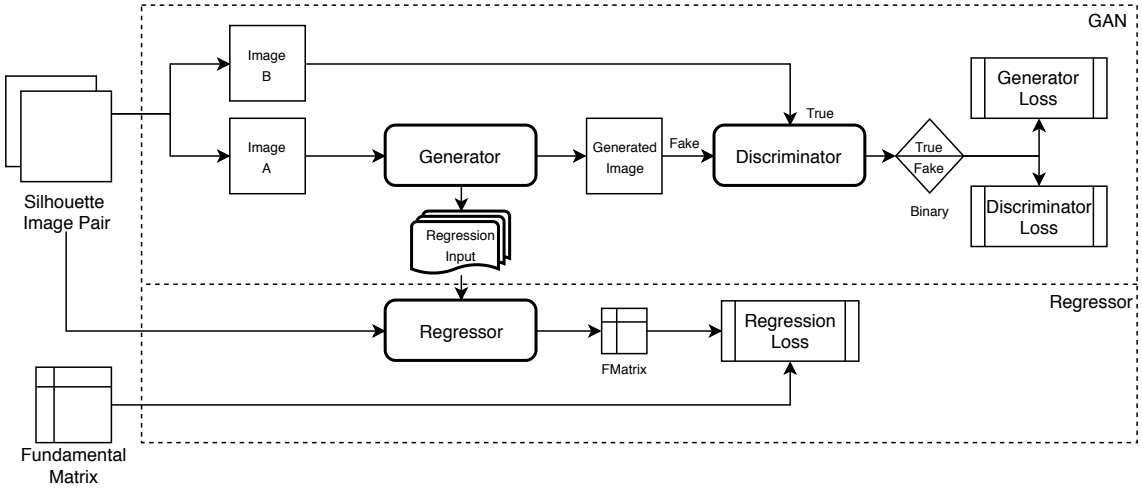

Figure 4.1: Top-level Model Architecture

is outputted. The architecture of the generator is shown in figure 4.2. [1]

4.1.1 Downsampling

Downsampling of the input is done using four consecutive downsampling blocks. Each downsampling block consists of two convolutions followed by a max pooling operation. The convolutions use a kernel size of 3×3, stride 1, same padding and ReLu activations. The number of channels depends on the downsampling block and is doubled after each block starting with 64. For the max pooling layers a kernel size and stride of 2×2 is used, which leads to halfing the image width and height in each downsampling block. The values of the second convolutional layer in each downsampling block are also used as part of a model output and in the skip connections to the upsampling part of the generator. After all blocks have been applied, the image was downsampled to dimension $8 \times 8 \times 512$. Table 4.1 provides a detailed description of the downsampling layers of the generator.

4.1.2 Bottleneck Block

The downsampling block is followed by the bottleneck block. This block is the part of the generator which differs the most from the U-Net architecture. While in the U-Net architecture convolutions are used in the bottleneck, in this generator fully connected layers are used. As the input to the bottleneck block is given in 2D, it first needs to be flattended to be used in the fully connected layers. The flattening operation is followed by a dropout layer. Then a fully connected layer with ReLu activations, called the bottleneck layer, is applied. The output of this layer is used as the bottleneck output of the generator. After the bottleneck layer a second fully connected layer with

[1] Note that the generator does not use batchnorm, as always only one sample is passed through the network.

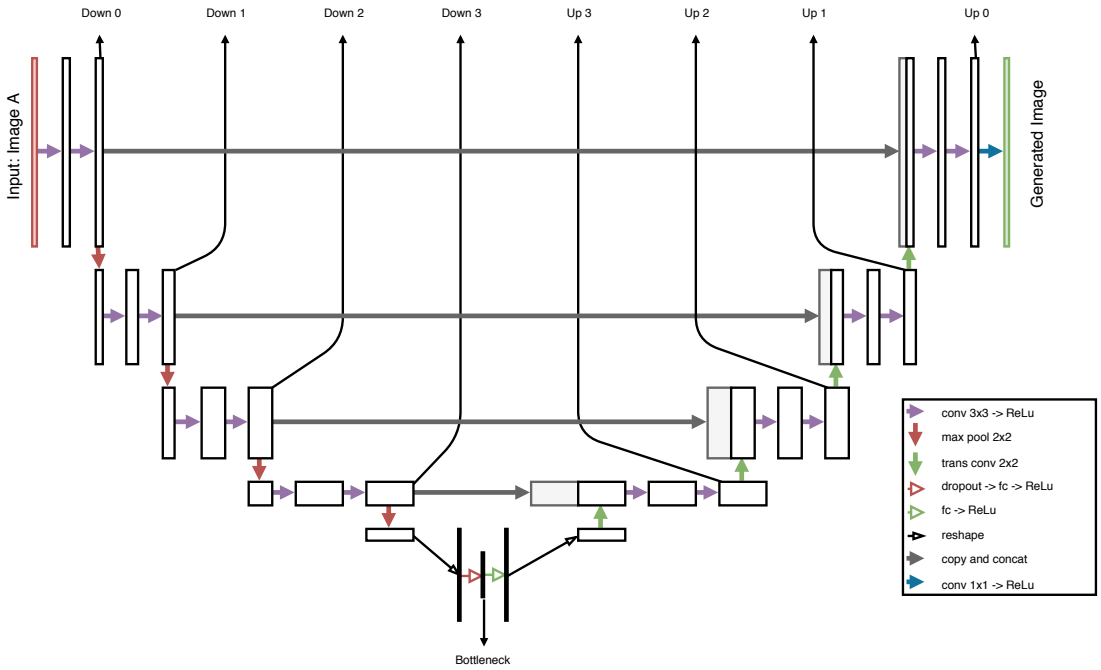

Figure 4.2: Generator Architecture

ReLu activations is applied. This layer upsamples the data again by having as many neurons as the dimension $8 \times 8 \times 512$ would require, which is 32768. The fully connected layer is followed by a reshape operation, so that the output of this layer has dimension $8 \times 8 \times 512$, which is equal to the dimension of the input to the bottleneck block. For a detailed description of the bottleneck block layers see Table 4.2.

4.1.3 Upsampling

After the data has been processed by the bottleneck block, it has to be upsampled to the original image size again. This is done by four successive upsampling blocks. Each block starts with a transposed convolution using kernel size and stride 2×2, which doubles the width and height of the image. Then the result of the transposed convolution is concatenated with the corresponding downsampling output which has the same dimension. Each block ends with two convolutions with kernel size 3×3, stride 1, same padding, and ReLu activations. The number of channels of the convolutions and the transposed convolutions is the same within each block. It starts with the value 512 for the first block and halfs after each block. After the last upsampling block a single convolution with one channel, kernel size 1×1, stride 1, same padding, and ReLu activations is applied. This layer is used as feature selector for the output image. Details of the layers can be taken from Table 4.3.

Stage	Layer	Operation	Output Dim	Model Output
Single Image Input	*input*		*128x128x1*	
Downsample 0	down0_conv1	conv [3x3, 64], ReLu	128x128x64	
	down0_conv2	conv [3x3, 64], ReLu	128x128x64	down0
	down0_pool	max pool [2x2]	64x64x64	
Downsample 1	down1_conv1	conv [3x3, 128], ReLu	64x64x128	
	down1_conv2	conv [3x3, 128], ReLu	64x64x128	down1
	down1_pool	max pool [2x2]	32x32x128	
Downsample 2	down2_conv1	conv [3x3, 256], ReLu	32x32x256	
	down2_conv2	conv [3x3, 256], ReLu	32x32x256	down2
	down2_pool	max pool [2x2]	16x16x256	
Downsample 3	down3_conv1	conv [3x3, 512], ReLu	16x16x512	
	down3_conv2	conv [3x3, 512], ReLu	16x16x512	down3
	down3_pool	max pool [2x2]	8x8x512	

Table 4.1: Generator Downsampling Layers
Note: The first row represents the image input to the generator and is not a real layer

Stage	Layer	Operation	Output Dim	Model Output
Downsaple	*downsample*		*8x8x512*	
Bottleneck	bottleneck_flatten	flatten	32 768	
	bottleneck_dropout	dropout	32 768	
	bottleneck_fc1	fc [bottleneck_size], ReLu	bottleneck_size	bottleneck
	bottleneck_fc2	fc [32 768], ReLu	32 768	
	bottleneck_unflatten	reshape [8x8x512]	8x8x512	

Table 4.2: Generator Bottleneck Block Layers
Note: The first row represents the output of the downsampling part and is not a real layer

4.1.4 Generator Inputs and Outputs

Image Input As only input the generator has a single binary image of dimension $128 \times 128 \times 1$. This is the first of the two images of each sample.

Generated Image Output The generator has multiple outputs. There is the generated image output, which returns a single channel image with dimension $128 \times 128 \times 1$. This output is needed for the GAN training and is given to the discriminator. It represents the values of the last layer of the upsampling block.

Bottleneck Output Another output is the bottleneck output. This output contains the values of the bottleneck layer in the middle of the model. It is used as an important input to the regressor, as it contains the most compact information of the image generation and thus the relation of the input and the output image. The output is 1D-shaped, and its exact dimension is specified by the hyperparameter `bottleneck_size`.

Derived Features Outputs There are also four so called "derived features outputs". These outputs represent the down- and upsampled features in the generator

Stage	Layer	Operation	Output Dim	Model Output
Bottleneck	*bottleneck*		*8x8x512*	
Upsample3	up3_tconv	transposed conv [2x2, 512]	16x16x512	
	up3_concat	concatenate [up3_tconv, down3]	16x16x1024	
	up3_conv1	conv [3x3, 512], ReLu	16x16x512	
	up3_conv2	conv [3x3, 512], ReLu	16x16x512	up3
Upsample2	up2_tconv	transposed conv [2x2, 256]	32x32x256	
	up2_concat	concatenate [up2_tconv, down2]	32x32x512	
	up2_conv1	conv [3x3, 256], ReLu	32x32x256	
	up2_conv2	conv [3x3, 256], ReLu	32x32x256	up2
Upsample1	up1_tconv	transposed conv [2x2, 128]	64x64x128	
	up1_concat	concatenate [up1_tconv, down1]	64x64x256	
	up1_conv1	conv [3x3, 128], ReLu	64x64x128	
	up1_conv2	conv [3x3, 128], ReLu	64x64x128	up1
Upsample0	up0_tconv	transposed conv [2x2, 64]	128x128x64	
	up0_concat	concatenate [up0_tconv, down0]	128x128x128	
	up0_conv1	conv [3x3, 64], ReLu	128x128x64	
	up0_conv2	conv [3x3, 64], ReLu	128x128x64	up0
Out	out_conv	conv [1x1, 1], ReLu	128x128x1	generated_image

Table 4.3: Generator Upsampling Layers
Note: The first row represents the output of the bottlneck block and is not a real layer

at different scales. They are created by concatenating the corresponding downsampled and upsampled outputs of the generator. For example, to create the output `derived_features_0` the layer outputs `down0` and `up0` are concatenated. Details for all derived features outputs are described in Table 4.4.

Derived Features Output	Output Creation	Output Dim
derived_features_0	concatenate [down0, up0]	128x128x128
derived_features_1	concatenate [down1, up1]	64x64x256
derived_features_2	concatenate [down2, up2]	32x32x512
derived_features_3	concatenate [down3, up3]	16x16x1024

Table 4.4: Derived Feature Outputs of the Generator

4.1.5 Design Decisions

As described at the beginning of this section the general structure of the generator with its down- and upsampling parts is based on the U-Net architecture [9]. This inlcudes the layer sequence, the idea of skip connections with concatenations, and layer parameters like the number of channels, kernel sizes, and activation functions. Only the bottleneck block differs a bit from that architecture. U-Net was choosen as a basis, because it is a successull model for image-to-image tasks that uses a dense feature representation in the middle of the network and skip connections for better gradient flow. The idea of dense feature representations is important for learning meaningfull features that can be

used in the regressor. Strong gradient flow helps stabilizing the training of the GAN, which is particulary important for this model. [2]

Concept of Bottleneck Block The bottleneck block implements the idea of the dense feature representations. After training the GAN, these features should contain information about the relation between the input and the generated image. This has the following reason: When the generator learns how to generate the second image, most information, except for the data transported throught the skip connections, needs to flow from the input image through the bottlneck layer. In a trained GAN the generated image should be similar to the original second image. Thus, the learned representation in the bottleneck layer should also represent the relation between the two original images. This representation can then be used in the regressor.

The reason why this model uses fully connected layers instead of convolutions in the bottlneck block, like U-Net does, is the following: The U-Net architecture is used for image segmentation tasks, which are translation invariant and are mostly based on local features. But the task to generate a second view from the first view of a scene is neither translation invariant nor does it only require local features. Here translations matter, and the generator needs to respect the relation of pixels in different regions of the image. To represent this relation, and to include information from different parts of the image, fully connected layers are used. This is because they are far more flexible for this kind of task. For diversifying the learned features in the bottlneck layer and as a regularization mechanism a dropout layer is applied before the bottleneck layer. This can help preventing overfitting, and is especially important, because the GAN is always only trained with one sample at a time. [3]

4.2 Discriminator

The other half of the GAN is the discriminator. This model is only used as an adversarial to the generator and helps it learning the relation between the two input images. It consists of a downsampling part and a classification part, which are both described in the following paragraphs. [4]

4.2.1 Downsampling

Downsamping in the discriminator is very similar to that in the generator. It takes the input image, which is either the output of the generator or the second image of a sample, and applies four downsampling blocks to it. Each block consists of two convolutional and one max pooling layer. These convolutions have a 3×3 kernel, same padding, stride 1, and ReLu activations. Their number of channels depends on the downsampling block

[2] See section 6.6.1.

[3] See section 6.6.2.

[4] Note that the discriminator does not use batchnorm, as always only one sample is passed through the network.

and doubles after each block starting with 64 channels in the first block. The image size is halfed in each block by the pooling layer, which uses a kernel size and stride of 2×2. Table 4.5 shows the details of the layers.

Stage	**Layer**	**Operation**	**Output Dim**
Single Image Input	*input*		*128x128x1*
Downsample 0	down0_conv1	conv [3x3, 64], ReLu	128x128x64
	down0_conv2	conv [3x3, 64], ReLu	128x128x64
	down0_pool	max pool [2x2]	64x64x64
Downsample 1	down1_conv1	conv [3x3, 128], ReLu	64x64x128
	down1_conv2	conv [3x3, 128], ReLu	64x64x128
	down1_pool	max pool [2x2]	32x32x128
Downsample 2	down2_conv1	conv [3x3, 256], ReLu	32x32x256
	down2_conv2	conv [3x3, 256], ReLu	32x32x256
	down2_pool	max pool [2x2]	16x16x256
Downsample 3	down3_conv1	conv [3x3, 512], ReLu	16x16x512
	down3_conv2	conv [3x3, 512], ReLu	16x16x512
	down3_pool	max pool [2x2]	8x8x512

Table 4.5: Discriminator Downsampling Layers
Note: The first row represents the image input to the discriminator and is not a real layer

4.2.2 Classification Block

After the input has been downsampled it is processed by the classification block, which binary classifies the image as true (being the second original image) or fake (meaning generated by the generator). This block starts with a flattening operation followed by a dropout layer. Then two fully connected layers with 1024 and 512 neurons respectively are applied. Both use ReLu activation functions. In the end a fully connected layer with a single neuron and the sigmoid activation function is used to calculate the probability of the input image being true. In Table 4.6 the details of the layers are described.

Stage	**Layer**	**Operation**	**Output Dim**
Downsample	*downsample*		*8x8x512*
Classification	class_flatten	flatten	32 768
	class_dropout	dropout	32 768
	class_fc1	fc [1024], ReLu	1024
	class_fc2	fc [512], ReLu	512
	class_sigmoid	fc [1], sigmoid	1

Table 4.6: Discriminator Classification Block Layers
Note: The first row represents the output of the downsampling part of the discriminator and is not a real layer

4.2.3 Discriminator Inputs and Outputs

Generated Image Input The discriminator has one single channel image as its only input. As described, this image is either the second image from the sample or the generated image output of the generator. It has dimension $128 \times 128 \times 1$.

Binary Classification Output As only output the discriminator has a single probability used for binary classifying the input image. This output should have high probability (which means classifying the image as true) when the input image is the second image of the sample. It should have low probability (which means classifying the image as fake) for the generated image.

4.2.4 Design Decisions

The discriminator model uses an ordinal binary image classification architecture that first downsamples the image using convolutional and max pooling layers, and then classifies it using fully connected layers. In the downsampling block the layer sequence and configuration resembles the downsampling block of the generator. The reason for this is to give both the regressor and the discriminator equal power when they are trained against each other. So it should be less probable that one of them gets to good compared to the other one, which would destabilize the training process. The dropout layer is used as a regularization to prevent overfitting of the discriminator on the true and generated images.

4.3 Regressor

Contrary to generator and discriminator the regressor is not part of the GAN. As already described, it uses different internal features of the generator to regress the fundamental matrix. The regressor model is based on [11]. But, unlike in [11], in the downsampling part additional inputs from the generator are included instead of just using the image pair as input. The model starts with the image pair, which is downsampled while including the derived features at the corresponding image scales. After that, the bottleneck input is concatenated. These steps are followed by a block of fully connected (FC) layers. The model then ends with a regression block that predicts the fundamental matrix. Downsampling part and fully connected layers are also inspired by [4]. Figure 4.3 shows the architecture of the regressor.

4.3.1 Normalization of Regression Inputs

Before the bottleneck input and the derived features are used in the regressor, they are normalized so that the regressor can easily handle different value ranges. This normalization is done independently for each training batch using a batch normalization layer on each of the inputs. The layers are applied directly to the inputs, and the outputs

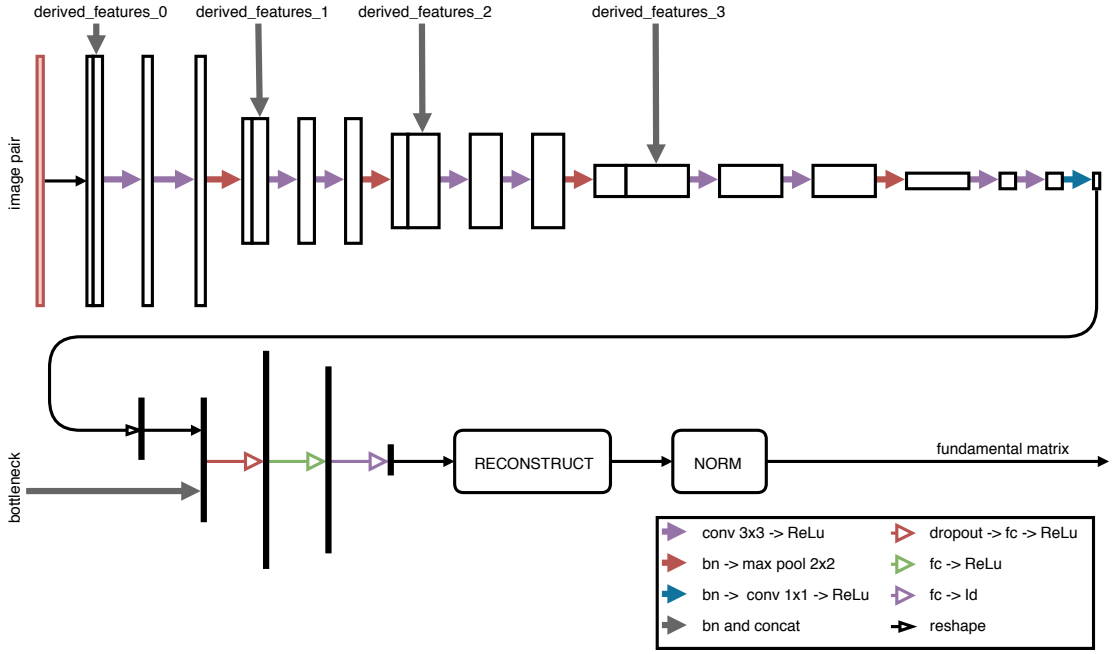

Figure 4.3: Regressor Architecture (with reconstruction layer, image input and all derived features)

of these layers are then used in the downsampling block. [5] The image pair input is not normalized within the model, because it was already normalized before the training.

4.3.2 Downsampling

The downsampling part starts with the image pair input, which contains both images of the sample concatenated to a two-channeled image. This input is then processed by four consecutive downsampling blocks. Each of these blocks first concatenates the corresponding derived features from the generator. This means that in the first block `derived_features_0` is concatenated, then 1 and so on. After the concatenation, two convolutional layers with kernel size 3×3, stride 1, same padding, and ReLu activations are applied in each block. The number of channels starts with 64 in the first block and doubles after each block. Then, at the end of each block, a batchnorm followed by a max pooling layer with kernel size and stride 2×2 is applied, which leads to halfing the image with and height. After the data has been processed by all downsampling blocks, it is passed to two consecutive convolutional layers with 8 channels, kernel size 3×3, stride 1, same padding, and ReLu activations. These layers are followed by a batchnorm and a 1×1 convolution with a single channel, same padding, stride 1, and ReLu activation. After that, a flattening operation is applied. In a final step of the downsampling part the

[5] Note about the used notation: In the downsampling block the normalized inputs are referenced by the name of the original inputs, but the batchnorm layers are applied before the inputs are used in the downsampling layers.

downsampled features are concatenated with the bottleneck input from the generator. Table 4.7 describes all layers of the downsampling part in detail.

4.3.2.1 Downsampling without Derived Features or Image Pair Input

The downsampling part differs from the previous description, if not all derived features and the image pair input should be used in the regressor. [6] Derived features that should not be included in the regressor are just not concatenated in the corresponding downsampling block. If the image pair input should not be used, then downsampling starts with the derived features of highest scale that are used in the model. [7] In the case where only the bottleneck input is included in the regressor the downsampling part is not needed at all. Instead the bottleneck input is directly used in the next blocks.

Stage	**Layer**	**Operation**	**Output Dim**
Img Pair Input	*input*		*128x128x2*
Downsample 0	down0_concat	concatenate [input, derived_features_0]	128x128x130
	down0_conv1	conv [3x3, 64], ReLu	128x128x64
	down0_conv2	conv [3x3, 64], ReLu	128x128x64
	down0_bn	batchnorm	128x128x64
	down0_pool	max pool [2x2]	64x64x64
Downsample 1	down1_concat	concatenate [down0_pool, derived_features_1]	64x64x320
	down1_conv1	conv [3x3, 128], ReLu	64x64x128
	down1_conv2	conv [3x3, 128], ReLu	64x64x128
	down1_bn	batchnorm	64x64x128
	down1_pool	max pool [2x2]	32x32x128
Downsample 2	down2_concat	concatenate [down1_pool, derived_features_2]	32x32x640
	down2_conv1	conv [3x3, 256], ReLu	32x32x256
	down2_conv2	conv [3x3, 256], ReLu	32x32x256
	down2_bn	batchnorm	32x32x256
	down2_pool	max pool [2x2]	16x16x256
Downsample 3	down3_concat	concatenate [down2_pool, derived_features_3]	16x16x1280
	down3_conv1	conv [3x3, 512], ReLu	16x16x512
	down3_conv2	conv [3x3, 512], ReLu	16x16x512
	down3_bn	batchnorm	16x16x512
	down3_pool	max pool [2x2]	8x8x512
Features	features_conv1	conv [3x3, 8], ReLu	8x8x8
	features_conv2	conv [3x3, 8], ReLu	8x8x8
	features_bn	batchnorm	8x8x8
	features_select	conv [1x1, 1], ReLu	8x8x1
	features_flatten	flatten	64
	features_concat	concatenate [features_flatten, bottleneck]	64 + bottleneck_size

Table 4.7: Regressor Downsampling Layers
Note: The first row represents the image pair input to the regressor and is not a real layer

[6] This can be controlled by hyperparameters.

[7] E.g.: If no image pair input is used but all derived features are, then downsampling starts with `derived_features_0`. If these features should also not be used it starts with `derived_features_1`.

4.3.3 FC-Block

After the data has been downsampled and the features prepared, they are processed in the fully connected block. This block is used to learn the complex relations between the different features. It starts with a dropout layer followed by two fully connected layers having 1024 and 512 neurons respectively. Both layers use ReLu as activation function. The details of this block are shown in Table 4.8.

Stage	**Layer**	**Operation**	**Output Dim**
Features	*features*		*64 + bottleneck_size*
FC	fc_dropout	dropout	64 + bottleneck_size
	fc_fc1	fc [1024], ReLu	1024
	fc_fc2	fc [512], ReLu	512

Table 4.8: Regressor FC Block Layers
Note: The first row represents the output of the downsampling part of the regressor and is not a real layer

4.3.4 Regression Block

The FC-block is followed by the regression block, which uses the results of the FC-block to regress the fundamental matrix. For this either a fully connected regression layer or a reconstruction layer may be used.

4.3.4.1 FC Regression Layer

The easiest way to regress the fundamental matrix is to use a single fully connected layer with nine neurons and identity activation functions followed by a reshape operation to dimension 3×3. See Table 4.9 for this. The severe drawback of this method is that the

Stage	**Layer**	**Operation**	**Output Dim**
FC block	*fc_block*		*512*
Regressor	reg_fc	fc [9], Identity	9
	reg_reshape	reshape [3x3]	3x3

Table 4.9: Regressor Regression Block Layers (without reconstruction)

mathematical properties of the fundamental matrix are not enforced.

4.3.4.2 Reconstruction Layer

A better way to regress the fundamental matrix is to use a reconstruction layer as described in [11]. This layer constructs the matrix in a way that it satifies the mathematical properties of a fundamental matrix, which is a rank 2 matrix with seven degrees

of freedom. For this it builds the matrix from camera intrinsics and extrinsics following the mathematical definition of the fundamental matrix:

$$\boldsymbol{F} = \boldsymbol{K_2}^{-T}[\boldsymbol{t}]_{\times}\boldsymbol{R}\boldsymbol{K_1}^{-1} \tag{4.1}$$

Where $\boldsymbol{K_1}$ and $\boldsymbol{K_2}$ are the camera instrinsics of the first respectively second image. These intrinsics are very restricted and only contain one focal length f_i for each image which is used for both the x and y axis. The principal point and the skew are not included. $\boldsymbol{R}$ is the relative camera rotation and $[\boldsymbol{t}]_{\times}$ is the cross product matrix of the relative camera translation $\boldsymbol{t}$. They are defined as follows:

$$\boldsymbol{K_i}^{-1} = \begin{bmatrix} f_i^{-1} & 0 & 0 \\ 0 & f_i^{-1} & 0 \\ 0 & 0 & 1 \end{bmatrix}, [\boldsymbol{t}]_{\times} = \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix}, \boldsymbol{R} = \boldsymbol{R_x}(r_x)\boldsymbol{R_y}(r_y)\boldsymbol{R_z}(r_z) \tag{4.2}$$

Where $\boldsymbol{R_x}$, $\boldsymbol{R_y}$, $\boldsymbol{R_z}$ are the rotation matrices around the x, y and z axis. Altogether the eight parameters $f_1, f_2, t_x, t_y, t_z, r_x, r_y$, and r_z are needed to reconstruct the matrix. To provide these values the previous fully connected layers are followed by a single fully connected layer with eight neurons and identity activations. This layer regresses these eight parameters and passes them to the reconstruction function, which then returns the reconstructed 3×3 fundamental matrix. See Table 4.10 for the layer details. [11] [6, pp. 154,244]

Stage	**Layer**	**Operation**	**Output Dim**
FC block	*fc_block*		*512*
Regressor	reg_fc	fc [8], Identity	8
	reg_reconstruct	reconstruction_layer	3x3

Table 4.10: Regressor Regression Block Layers with Reconstruction

4.3.5 Normalization Layer

After the fundamental matrix has been regressed, it is normalized. Any fundamental matrix could be multiplied by any constant, real scalar and would still describe the same image relation. This would not brake the constraints of the matrix. The normalization thus simplifies the learning task by making it easier to compare different fundamental matrices. Based on the results of [11], the frobenius (FBN) norm and the maximum absolute value (ABS) norm have been studied for normalization of the matrix:

FBN With this norm, the matrix is normalized so that it lies on a 9-dimensional sphere:

$$\mathcal{N}_{\text{FBN}}(\boldsymbol{F}) = \|\boldsymbol{F}\|_F^{-1}\boldsymbol{F} \tag{4.3}$$

where $\|\boldsymbol{F}\|_F$ is the frobenius norm of $\boldsymbol{F}$.

ABS This normalization restricts all matrix entries to be within $[-1, 1]$:

$$\mathcal{N}_{\text{ABS}}(\boldsymbol{F}) = (\max_{i,j} |\boldsymbol{F}_{i,j}|)^{-1} \boldsymbol{F} \tag{4.4}$$

The normalization layer takes the regressed fundamental matrix as input and outputs the normalized matrix. So it has both an input and an output of dimension 3×3. To successfully train the regressor the ground truth needs to be normalized with the same norm that is used in the normalization layer. [11]

4.3.6 Regressor Inputs and Outputs

Bottleneck Input The most important input of the regressor is the bottleneck input. This input gets the data taken form the bottleneck output of the generator and has the same dimension as the hyperparameter `bottleneck_size`. It is added to the features at the end of the downsampling block.

Derived Feature Inputs Additionally, the derived features of the generator may optionally be used as inputs. Either none, one or more of them can be included. They are concatenated in the downsampling block of corresponding scale.

Image Input Also the image pair of each sample can optionally be used in the regressor. This input is a binary image of dimension $128 \times 128 \times 2$ and is used at the beginning of the downsampling block.

Fundamental Matrix Output This is the only output of the regressor and returns the regressed fundamental matrix. It is taken from the output of the normalization layer

4.3.7 Design Decisions

As templates for the architecture of the regressor [11] and [4] have been used. This includes the general structure with the downsampling part followed by fully connected layers and the regressoin block at the end. From [11] also the idea of the reconstruction layer and the normalization layer have been taken. Position features, like they were used in [11], have not been adopted, because these did not improve the performance in [11]. These two models, [11] and [4], were used as templates, because they are both successfull in learning camera relations from two given images, which is very similar to the task of this project. In [11] the task even was to regress the fundamental matrix. The layer configurations of the downsampling part are similar to that of the downsampling part of the generator, so that the derived features can easily be concatenated. The single channeled convolutional layer at the end of the downsampling part is used as a feature selector. It is included to compact the features learned from the derived features and the input images, so that they are represented by a similar number of neurons as

the bottleneck input before they are both given to the FC-block. Batchnorm layers are used to stabilize training. This is particulary important as the input features from the generator may have large value ranges. These ranges come form the fact that the GAN is trained for each image pair independently and thus may learn very different representations of the image relations. The dropout layer is used as a regularization and thus to help the regressor to generalize.

5

Training and Prediction

After the model architecture has been discussed in the previous chapter, this chapter explains how that model, including the GAN and the regressor, can be trained. It also explains how the trained model can then be used for prediction.

5.1 Training

Within this project two different training methods have been investigated: While the first method trains GAN and regressor together as a combined model, the second one first trains the GAN and then uses stored intermediate results of the generator to train the regressor. In both variants the GAN is trained with only a single sample at a time, because it needs to learn the relation of the two images of each sample independently. That is required so that this information can be used by the regressor to estimate the fundamental matrix.

This section first describes both training methods in detail and then compares the benefits and drawbacks of both methods.

5.1.1 Combined Training

The combined training method trains the GAN and the regressor mostly as a combined model. When the model is fit to a training dataset it is trained for several training epochs (also called fit epochs). Each training epoch starts with randomizing the order of the training samples. Then this method iterates over all training samples in the training set in a way that each sample is handled independetly.

5.1.1.1 Combined GAN and Regressor Training

For each of the samples (x, y) the GAN and the regressor are trained together for several GAN epochs (sample epochs). Within each GAN epoch the GAN is trained similar to [5]: The first image x_a of the image pair of the sample is given to the generator $G(\cdot)$, which then outputs a generated image $G_\mathrm{img}(x_\mathrm{a})$. Using this generated image and the true second image x_b from the image pair, the discriminator $D(\cdot)$ is trained for one or more

iterations. It is trained to binary classify the generated image as `fake` and the true image as `true` using *binary-cross-entropy (bce)*:

$$\min_{D} \text{bce}(D(x_\text{b}), \texttt{true}) + \text{bce}(D(G_\text{img}(x_\text{a})), \texttt{fake}) \tag{5.1}$$

After the discriminator has been trained, the generator and the regressor are trained as a combined model. To create this combined model the generated image output of the generator $G_\text{img}(x_\text{a})$ is connected to the inputs of the discriminator, and the regressor output $G_\text{reg}(x_\text{a})$ of the generator is connected to the regressor. Optionally also the image pair is given as input to the regressor. This combined model has two outputs: A binary classification probability and the regressed fundamental matrix. For the classification output *binary-cross-entropy (bce)* loss with the label `true` is used and for the fundamental matrix *mean-squared-error (mse)* loss with the true fundamental matrix y. Both losses are weighted and summed up to a combined loss using w_G as weight for the generator and w_R as weight for the regressor. Minimizing this combined loss finishes the GAN training epoch:

$$\min_{G,R} w_G \cdot \text{bce}(D(G_\text{img}(x_\text{a})), \texttt{true}) + w_R \cdot \text{mse}(R(G_\text{reg}(x_\text{a}), (x_\text{a}, x_\text{b})), y) \tag{5.2}$$

The discriminator weights might be frozen during this training.

5.1.1.2 Regressor only Training

After GAN and regressor have been trained together, the regressor can be trained alone for zero or more iterations. This is still done within one sample iteration within one training epoch. Before the regressor can be trained, first the generator is used to create the regression inputs, consisting of bottleneck input and derived features. This is done by passing the first image into the trained generator only once and then taking the regressor output $G_\text{reg}(x_\text{a})$. Optionally the image pair is added to the generated regression inputs. These are then used to train the regressor several iterations using the *mean-squared-error (mse)* loss with the true fundamental matrix y of the sample:

$$\min_{R} \text{mse}(R(G_\text{reg}(x_\text{a}), (x_\text{a}, x_\text{b})), y) \tag{5.3}$$

The generator is not trained during these iterations.

After this, the method continues with the next sample and after all samples have been processed the next epoch begins.

5.1.1.3 Number of Training Iterations

In combined training the GAN together with the regressor have to be trained very often as they are trained for several GAN epochs for each sample in each training epoch. The number of training iterations of the GAN combined with the regressor is:

$$\text{fit\_epochs} \times |X_\text{TRAIN}| \times \text{sample\_train\_epochs} \tag{5.4}$$

And the total number of discriminator iterations during the training is:

$$\text{fit\_epochs} \times |X_{\text{TRAIN}}| \times \text{sample\_train\_epochs} \times \text{train\_discr\_iterations} \tag{5.5}$$

The regressor alone may also be trained several times in each training epoch for each sample:

$$\text{fit\_epochs} \times |X_{\text{TRAIN}}| \times \text{regressor\_train\_iterations} \tag{5.6}$$

5.1.1.4 Validation

Validation takes place at the end of a training epoch after GAN and regressor have been trained. For validation the image pairs of the validation set are given to the prediction algorithm (see section 5.2). This algorithm returns the predicted fundamental matrices. During the prediction the model weights are not changed to not influence the training. The predicted fundamental matrices and the true matrices from the validation set are then used to compute the validation metrics.

5.1.2 Separated Training

An alternative training method is the separated training, where GAN and regressor are never trained together but each on its own. This method consists of two major steps: First the GAN without the regressor is trained and the regression inputs, consisting of bottleneck input and derived features, are generated for each sample. This is done only once at the beginngin of the training. After that the regressor is trained separately for several epochs.

5.1.2.1 GAN Training

Before the regression inputs can be generated the GAN has to be trained for each sample. To do this, the training algorithm iterates over all samples in the training set. Each sample is processed on its own, no batches are used. For each sample the GAN is trained without the regressor for several GAN epochs. A GAN epoch in this training method is very similar to a GAN epoch in the combined training algorithm: First the generated image $G_{\text{img}}(x_{\text{a}})$ is produced by putting the first image x_{a} of the sample into the generator $G(\cdot)$. Then the discriminator $D(\cdot)$ is trained for one or more iterations. As before, the discriminator is trained using *bce* to classify the true second image x_{b} as `true` and the generated image as `fake`:

$$\min_D \text{bce}(D(x_{\text{b}}), \texttt{true}) + \text{bce}(D(G_{\text{img}}(x_{\text{a}})), \texttt{fake}) \tag{5.7}$$

After that, the generator is trained. In contrast to the combined training method, here the regressor is not trained together with the generator. To train the generator the generated image output $G_{\text{img}}(x_{\text{a}})$ of it is connected to the discriminator. Now this connected model is trained using *bce* with the first image as input and `true` as target:

$$\min_G \text{bce}(D(G_{\text{img}}(x_{\text{a}})), \texttt{true}) \tag{5.8}$$

The discriminator weights might again be frozen for this.

5.1.2.2 Regression Input Generation

After the GAN has been trained the regression input has to be generated. This is done within each sample iteration. It means that for each sample first the GAN is trained, then the regression input is generated, and only then the algorithm continues with the next sample. To generate the regression input the first image of the sample is put into the trained generator and the regressor output $G_{\text{reg}}(x_{\text{a}})$ of the generator is taken. If the input image pair should be used as well for regressor training, the image pair is added to the regressor output of the current sample. Either only the regressor ouput or output and image are then stored together as regression input x_{reg} of the sample. After iterating over all samples, all regression inputs have been stored in the regression input set X_{reg} for later use in regressor training.

5.1.2.3 Regressor only Training

Now the regressor can be trained. The training set for the regressor consists of the regression input set X_{reg} and the set of fundamental matrices, which is Y_{TRAIN}. Each regression input in X_{reg} is labelled with the corresponding fundamental matrix from Y_{TRAIN}. X_{TRAIN}, which consists only of the training image pairs, is not needed. The regressor is trained in several training epochs. At the beginning of each epoch the regressor training set is randomized and after that split into batches $\{(X_{\text{reg}}^{(1)}, Y_{\text{TRAIN}}^{(1)}), \ldots, (X_{\text{reg}}^{(m)}, Y_{\text{TRAIN}}^{(m)})\}$, where $m = \left\lceil \frac{|X_{\text{reg}}|}{\text{regressor\_batch\_size}} \right\rceil$ is the number of batches. For each of these batches i of regression inputs and fundamental matrices the regressor is trained using the *mse* loss:

$$\min_{R} \text{mse}(R(X_{\text{reg}}^{(i)}), Y_{\text{TRAIN}}^{(i)}) \tag{5.9}$$

5.1.2.4 Number of training iterations

In separated training the GAN (without the regressor) is only trained in the beginning and thus the number of GAN training iterations is independent of the number of regressor training epochs. This means that the GAN is trained for the following number of iterations:

$$1 \times |X_{\text{TRAIN}}| \times \text{sample\_train\_epochs} \tag{5.10}$$

The total number of training iterations for the discriminator is given by:

$$1 \times |X_{\text{TRAIN}}| \times \text{sample\_train\_epochs} \times \text{train\_discr\_iterations} \tag{5.11}$$

As the regressor is trained in each epoch, the number of regressor iterations depends on the number of epochs, and as it is trained with batches it also depends on the used batch size. It is given by:

$$\text{regressor\_epochs} \times \left\lceil \frac{|X_{\text{TRAIN}}|}{\text{regressor\_batch\_size}} \right\rceil \tag{5.12}$$

5.1.2.5 Validation

In this training method validation is done a bit differently as in the combined method. To speed up validation, the regression input data for all validation samples is created only once. [1] The regression inputs for validation are created in the same way as the regression inputs for training. The model weights are not influenced by this, because the weights are saved before training the GAN with each validation sample and restored afterwards. The validation regression inputs are created directy after the training regression inputs but still before the regressor training epochs. These validation regression inputs are then used at the end of each regressor epoch to predict fundamental matrices for the validation samples. These predicted matrices together with the true matrices are then given to the validation metrics to compute the validation results.

5.1.3 Comparison of Training Methods

Both training methods could in theory train the model. The main differences are the training speed and the degree of coupling between the generator and the regressor. When the combined training method is used the generator and the regressor learn together and gradients can flow from the regressor back to the generator. This means that the regressor can influence the generator, which might help the generator to learn good feature representations. But in combined training the regressor is only trained with single samples because it is coupled to the generator which always only uses single samples for training. In separated training however, the regressor can be trained with batches. This has the benefit that batchnorm can be used to stabilize training. Also higher learning rates might be used, as the regressor is trained for multiple samples at once. In combined training multiple regressor iterations could be used after each GAN training. But the number of used regressor iterations should not be too high in combined training, because the regressor would then be trained too much on each single sample and might overfit on it. This is especially a problem for the last sample that is used in the last training epoch. Another benefit of separated training is that it needs a lot less training iterations, as the GAN is only trained once for each sample. [2] This fact and the use of batches during regressor training makes the separated training method a lot faster than the combined one.

5.2 Prediction

Due to the rather complex model and training methods also prediction becomes more complicated. Nevertheless a single prediction algorithm can be used for both training methods. As the GAN has to be trained for each sample on its own during prediction, no batches can be used. Instead the algorithm iterates over all samples and predicts the fundamental matrix for each of them independently. For each sample the GAN is

[1] The generation of the regression inputs is what takes the most time during validation.
[2] Compare sections 5.1.1.3 and 5.1.2.4.

trained several GAN epochs (sample epochs). It is trained in the same way as in the training methods by first training the discriminator on true and generated images:

$$\min_D \mathrm{bce}(D(x_\mathrm{b}), \texttt{true}) + \mathrm{bce}(D(G_\mathrm{img}(x_\mathrm{a})), \texttt{fake}) \tag{5.13}$$

Then the generator is trained without the regressor, as no true fundamental matrices are available. So only the normal generator loss is used:

$$\min_G \mathrm{bce}(D(G_\mathrm{img}(x_\mathrm{a})), \texttt{true}) \tag{5.14}$$

The number of GAN epochs during prediction does not need to be the same as for training. Anyhow, for simplicity and to reduce the number of hyperparameters the same number of epochs is used for both. After the GAN has been trained for a sample, the first image of that sample is given to the generator, and the outputs of the generator are given to the regressor, optionally together with both images of the sample. The regressor then predicts the fundamental matrix for the sample:

$$\boldsymbol{F}_\mathrm{pred} = R(G_\mathrm{reg}(x_\mathrm{a}), (x_\mathrm{a}, x_\mathrm{b})) \tag{5.15}$$

In order to not influence the model weights during prediction they are saved before the GAN is trained for each sample and restored afterwords. By doing so, the prediction neither influences the training nor the prediction of following samples, while the weights do not need to be frozen which would make prediction impossible.

6

Experiments

This chapter covers the experiments that were performed within the project to compare different models, to analyze them and to measure their performance. The setting of the experiments is described, and the results are presented. After that, some problems of the models are analyzed.

6.1 Compared Models and Training Methods

Three different model variants were compared in the experiments. One model uses only the bottleneck input of the regressor. Neither the derived features nor the image pair are given to the regressor. The second model uses the bottleneck input and all four derived feature inputs. In the third model also the image pair input is used, which means that the regressor gets the bottleneck data, all four derived features, and the image pair as input.

Although both norms, frobenius and maximum absolute value norm, have been investigated during the project, in the final experiments only the frobenius norm was used. All three models used the reconstruction layer, because in [11] this lead to the best results.

As training method only separated training was used during all of the final experiments. Combined training was not investigated in detail, because it is much slower than separated training. [1] Experiments using combined training would have required very much training time and thus there would have been less time for hyperparameter optimization and comparison of the models.

6.2 Used Datasets and Experiment Setup

During this project many different datasets have been created: datasets from texture images and synthetic datasets from different 3D-models using different camera ranges. Also, multiple synthetic datasets were combined to form large datasets with 100000 and

[1] See section 5.1.3.

more samples. These large datasets are usefull to train the model for real life problems, but using them would lead to very long training times of multiple weeks or even more just for training a single model. As the timespan of this project was limited, only smaller datasets were used during the experiments. This left some time for a bit of hyperparameter tuning. The texture image datasets have not been used for training, because they contained disturbances caused by shadows which could not be correctly thresholded.

This is why the training of the final models and the hyperparameter tuning for them were performed on the *synthetic horse dataset* [2], which was synthetically generated (see section 3.1.3) and contains about 2000 samples. This dataset was created from a single 3D model of a horse using constant intrinsic camera parameters. The images were taken from different perspectives while the camera was always directed towards the 3D-object in the scene. The distance to the object was constant. Figure 6.1 shows the image pair of a sample in the dataset. Within the dataset this sample is labelled with some fundamental matrix, which is not shown in the figure. This *synthetic horse dataset* was

Figure 6.1: Image Pair of a Sample of the *synthetic horse dataset*

split into a training, a validation, and a test set. Then, before the experiments all three compared models were trained on the training dataset and their hyperparameters were tuned on the validation dataset.

After all models have been tuned and trained using the *synthetic horse dataset*, two experiments were performed on them. In the first experiment the trained models were tested on the test dataset of the *synthetic horse dataset* to evaluate their performance. [3] Additionally, in the second experiment, they were tested on a completely different dataset, the *temple silhouette dataset* [4]. This dataset is a silhouette dataset that was created using a texture dataset, the *temple texture dataset*, and thresholding (see section 3.1.2). It contains a different 3D-model, a temple instead of a horse. An example sample of this dataset is shown in Figure 6.2. The purpose of the second experiment was to show

[2]The exact name of this dataset is *synthetic_horse_rotation* but for simplicty the name *synthetic horse dataset* is used in this document.

[3]It should be noted that in this experiment all samples used for testing have not been seen by any of the models during training, because the models were trained and tuned on different parts of the *synthetic horse dataset* as they were tested on.

[4]The exact name of this dataset is *templeSparseRing_silhouette* but for simplicty the name *temple silhouette dataset* is used in this document.

Figure 6.2: Image Pair of a Sample of the *temple silhouette dataset*

how the models behave on data that strongly differs form the training set. Although the models were tested against the *temple silhouette dataset* in this experiment, it is important to know that they were still trained on the *synthetic horse dataset*. [5]

6.3 Hyperparameter Tuning

In order to get the models working in the final experiments, their hyperparameters had to be tuned. This tuning was based on the evaluation metric *epi-abs*, which is explained in section 6.4. As already described, the validation set of the *synthetic horse dataset* was used for this.

Due to the long training times of the models and the limited time scope of the project, not all hyperparameters could be fully optimized. Some hyperparameters [6] were only slightly tuned by trying a few different values around good guesses that were used as starting points. These good guesses were based on some of the models that were used as templates. [7]

6.3.1 Pre-Tuning GAN Hyperparameters

Many hyperparameters of the GAN, like the GAN learning rates, the bottleneck size, the number of GAN training iterations, the skip connections, and the freezing of the discriminator have been pre-optimized by only training the GAN without the regressor on a few different image pairs. For these training rounds the generated images were evaluated using *mean-squared-error* with the original second image as well as by subjectively comparing different generated images [8]. Based on these results optimal parameter ranges were selected. The hyperprameter tuning of the whole model on the full dataset could then be restricted to these ranges. This pre-optimization is not optimal, as it

[5] Actually, in both experiments exactly the same trained model weights were used.

[6] Like the number of channels in the convolutional layers and the number of neurons in the fully connected layers.

[7] For example the models of [9], [11] and [4].

[8] It was manually checked, which generated images better represented the expected, original images.

just uses a part of the model and only a few samples, but it saved a lot of time in hyperparameter tuning.

6.3.2 Tuning Regressor Learning Rate

During the final experiments the main focus of hyperparameter tuning was on finding good regressor learning rates, although also some other hyperparameters have been tuned [9]. To speed this up, the following technique was used: The model is trained once starting with a very low learning rate. After each epoch the learning rate is multiplied with a small constant greater than 1, so that the learning rate is exponencially increasing with each epoch. After training, the results were analyzed to find the epoch ranges where the greatest decrease in the error metrics took place. For these epochs the learning rate ranges were computed and then learning rate tuning could focus on these ranges. Then, after the optimal learning rate was found, the models could be trained using normal learning rate decay, where the learning rate is decreasing with each epoch. This hyperparameter tuning procedure is based on the method described in section 3.3 of [10].

6.3.3 Hyperparameter Tuning of the different Models

The hyperparameters for the three models that were compared in the final experiments have not all been tuned independently. To save time, first the hyperparameters were trained on the bottleneck-only model. These tuned hyperparameters were then adopted in the other two models and only slightly optimized for them.

6.4 Evaluation

To analyze and compare the results, specialized metrics for the fundamental matrix were used. The *mean-squared-error loss*, which is used for training the regressor, directly compares the predicted matrix elements with the correct matrix elements. In contrast to this, the fundamental matrix metrics are more sophisticaed and check how well the epipolar constraint (6.1) holds for the predicted fundamental matrix.

$$\boldsymbol{x'}^T \boldsymbol{F} \boldsymbol{x} = 0 \tag{6.1}$$

Here $\boldsymbol{x}$ and $\boldsymbol{x'}$ are the homogenious coordinates of a 2D point in the first image respectively a point in the second image. $\boldsymbol{F}$ is a fundamental matrix. If $\boldsymbol{F}$ is the fundamental matrix that describes the relation of the two images, and $\boldsymbol{x}$, $\boldsymbol{x'}$ both correspond to the same point in 3D space, then this constraint holds. [6, pp. 245]

[9] E.g. the GAN sample epochs, GAN and regressor channels, different regularization techniques, regressor learning rate decay, using batchnorm layers.

6.4.1 Finding Corresponding Points

As all the fundamental matrix metrics that will be described are based on the constraint (6.1), these metrics need a set of correponding points. Each pair in this set consists of two homogenous 2D-points, one point in the first image and the other point in the second image, for which the constraint (6.1) holds. Two different approaches were used to find such points.

Random Sampling One way to create a set of corresponding points is to use random sampling. As input this variant only needs the true fundamental matrix $\boldsymbol{F}$ and the image size of the two images (which is the same for both). First, a set of points $\boldsymbol{p}_i$ is uniform randomly sampled from all the image coordinates of the first image, with $0 \leq (\boldsymbol{p}_i)_x \leq$ width and $0 \leq (\boldsymbol{p}_i)_y \leq$ height. Then for each point the epiline $\boldsymbol{l}_i$ is calculated by $\boldsymbol{l}_i = \boldsymbol{F}\boldsymbol{p}_i$. Now using uniform randomly drawn x-coordinates within the image width, some points $\boldsymbol{q}_{i,j}$ are sampled on this line in the second image. [10] Then for all combinations of i, j the pairs $(\boldsymbol{p}_i, \boldsymbol{q}_{i,j})$ are added to the final set of point correspondences. [11] After that, analogously, points $\boldsymbol{q}_i$ are drawn from the coordinates of the second image, and points $\boldsymbol{p}_{i,j}$ are sampled on the epilines of those points in the first image. The only difference is that now the epilines are calculated by $\boldsymbol{l}'_i = \boldsymbol{F}^T\boldsymbol{q}_i$. All these points are again added as pairs to the final set of point correspondences. [6, pp. 245]

SIFT An alterantive variant to find the corresponding points uses the SIFT keypoint detection algorithm and a FLANN (Fast Approximate Nearest Neighbor) based matcher from OpenCV as well as ratio testing. This method only requires the two original images as input, whereas the true fundamental matrix is not needed. This is because it tries to directly detect keypoints in the images and then matches them as corresponding points. Here no calculations involving the fundamental matrix are required. This variant is based on [1].

6.4.2 Epipolar Metrics

Using a set of found corresponding points $(\boldsymbol{p}, \boldsymbol{q})$ and the predicted fundamental matrix $\boldsymbol{F}_{\text{pred}}$ the following four metrics were used for evaluation. Each of them was applied once on corresponding points found by random sampling and once on points found using SIFT.

EPI-ABS (Epipolar Constraint with Absolute Value):

$$\mathcal{M}_{\text{EPI-ABS}}(\boldsymbol{F}_{\text{pred}}, \boldsymbol{p}, \boldsymbol{q}) = \sum_i |\boldsymbol{q}_i^T \boldsymbol{F}_{\text{pred}} \boldsymbol{p}_i| \tag{6.2}$$

[10] For each $\boldsymbol{p}_i$ the $\boldsymbol{q}_{i,j}$ have to be computed independetly, as each $\boldsymbol{p}_i$ may have a different epiline. This is why the i is also included in $\boldsymbol{q}_{i,j}$.

[11] This means that each point $\boldsymbol{p}_i$ will occur in multiple of such pairs in the set.

EPI-SQR (Epipolar Constraint with Squared Value):

$$\mathcal{M}_{\text{EPI-SQR}}(\boldsymbol{F}_{\text{pred}}, \boldsymbol{p}, \boldsymbol{q}) = \sum_i (\boldsymbol{q}_i^T \boldsymbol{F}_{\text{pred}} \boldsymbol{p}_i)^2 \tag{6.3}$$

SSD (Sampson Distance):

$$\mathcal{M}_{\text{SSD}}(\boldsymbol{F}_{\text{pred}}, \boldsymbol{p}, \boldsymbol{q}) = \sum_i \frac{(\boldsymbol{q}_i^T \boldsymbol{F}_{\text{pred}} \boldsymbol{p}_i)^2}{(\boldsymbol{F}_{\text{pred}} \boldsymbol{p}_i)_1^2 + (\boldsymbol{F}_{\text{pred}} \boldsymbol{p}_i)_2^2 + (\boldsymbol{F}_{\text{pred}}^T \boldsymbol{q}_i)_1^2 + (\boldsymbol{F}_{\text{pred}}^T \boldsymbol{q}_i)_2^2} \tag{6.4}$$

SED (Symmetrical Epipolar Distance):

$$\mathcal{M}_{\text{SED}}(\boldsymbol{F}_{\text{pred}}, \boldsymbol{p}, \boldsymbol{q}) = \sum_i (\boldsymbol{q}_i^T \boldsymbol{F}_{\text{pred}} \boldsymbol{p}_i)^2 \cdot \left(\frac{1}{(\boldsymbol{F}_{\text{pred}} \boldsymbol{p}_i)_1^2 + (\boldsymbol{F}_{\text{pred}} \boldsymbol{p}_i)_2^2} + \frac{1}{(\boldsymbol{F}_{\text{pred}}^T \boldsymbol{q}_i)_1^2 + (\boldsymbol{F}_{\text{pred}}^T \boldsymbol{q}_i)_2^2} \right) \tag{6.5}$$

To evaluate on a whole validation or test dataset the metrics were averaged over all the samples in the dataset. [12] [11] [6, pp. 287-288]

6.4.3 Baseline Algorithm

As there are currently almost no known algorithms available for estimating fundamental matrices for silhouette image pairs, a classical texture based algorithm was used to compute the baselines. This algorithm can use silhouette images as input, even though it was not designed for it and thus the results might be bad. In this baseline algorithm the SIFT keypoint detection algorithm and the FLANN based matcher from OpenCV as well as ratio testing were used to find corresponding points. These points where then given to the LMedS algorithm to estimate the fundamental matrix for each sample. The estimated fundamental matrices can then be evaluated by the metrics. This procedure to find the fundamental matrix is based on [1].

6.5 Results

In this section the results of the two experiments are described and interpreted. First the results of testing the models against the test set of the *synthetic horse dataset* are explained and after that the results on the *temple silhouette dataset*.

6.5.1 Results on Synthetic Horse Dataset

Table 6.1 shows the results of the first experiment, where the models were tested on the *synthetic horse dataset*. It compares the three final models of this project, the one

[12] For this, corresponding points need to be found for each sample in the dataset independently.

only with the bottleneck input, the one with bottleneck and derived feature inputs, and the one that additionally uses the image pair as input. In addition, the table contains two baseline result columns for which the baseline algorithm as described in section 6.4.3 was used. The first baseline column shows the results of the baseline algorithm tested on the test set of the *synthetic horse dataset* while the second baseline column shows the results on some texture image dataset, the *temple texture dataset* [13]. This dataset contains normal greyscale texture images instead of silhouette images and is completely independent of the test set used for the other baseline and the models. The purpose of this texture baseline solely is to give an impression on what results the baseline algorithm typically gives on a normal texture dataset as that algorithm was not designed to be used with silhouette images. The rows of the table show the results of the models and baselines measured with different epipolar metrics that used either SIFT or random sampling to determine the corresponding points. In each row the best result of the three compared models is emphasized. The last row shows for how many samples of all samples in the test set the model or baseline method could not estimate any fundamental matrix.

Model		**Model 1**	**Model 2**	**Model 3**	**Baseline**	**Texture***
derived features		none	0, 1, 2, 3	0, 1, 2, 3	-	-
image pair input		no	no	yes	-	-
SIFT	**epi-abs**	**1.87e+01**	2.08e+01	3.20e+01	6.87e+00	1.31e+01*
	epi-sqr	1.72e+03	**1.04e+03**	4.84e+03	4.07e+02	1.89e+03*
	ssd	2.17e+03	4.27e+03	**1.60e+03**	4.31e+02	6.23e+02*
	sed	2.79e+06	**1.09e+05**	1.17e+05	2.58e+03	3.75e+03*
RANDOM	**epi-abs**	1.27e+02	**9.00e+01**	2.81e+02	4.72e+01	6.99e+03*
	epi-sqr	**1.07e+07**	3.42e+07	2.73e+07	5.56e+05	2.97e+11*
	ssd	3.45e+03	6.28e+03	**3.39e+03**	1.92e+03	2.97e+03*
	sed	1.13e+09	3.08e+08	**5.40e+07**	1.85e+07	7.54e+11*
Missing Samples**		0/245	0/245	0/245	166/245	45/192*

Table 6.1: Experiment Results on *synthetic horse dataset*

* Baseline on texture dataset: These metrics were not computed on the *synthetic horse dataset* but on the *temple texture dataset*.

** The number of samples that could not be estimated, because not enough matching points could be found. These samples were ignored when computing the metrics.

It can be seen that all three compared models show very similar results. The differences between the models are very small, and there is no model which is strictly worse or better than the others in all of the metrics. If for some application the *ssd* and *sed* metrics are more important than the other metrics, then there seems to be a small benefit when using the third model with images and derived features. However, as this benefit is very small, none of the models can be favorized in general. The benefit might even become completely irrelevant when more hyperparameter tuning was done. So the derived features and the image pair inputs neither seem to be required for opti-

[13] The exact name of this dataset is *templeSparseRing* but for simplicty the name *temple texture dataset* is used in this document.

mal results nor do they seem to hurt performance. From this it can be concluded that the bottleneck input contains the most important information about the image relation: Although the dervied features and the image pair may contain the same information, they do not seem to contain much data that is not already covered by the bottleneck input.

When comparing the models against the baseline results, it first has to be noted that in the silhouette baseline no matrix could be estimated for more than half of the samples, and even the texture baseline could not find solutions for all samples. In contrast to the baseline methods, the compared models could estimate all matrices. The results of the silhouette baseline are roughly by factor 10 better than the results of the models. This does not mean very much, though, as the metrics of the baseline were computed only on the few samples for which matrices could be estimated. Compared to the texture baseline the SIFT results of the models are only slightly worse. For the random sampled metrics the results were mostly even better. This makes sense as the baseline algorithm uses SIFT for estimation and thus the SIFT results might be slightly biased towards the baselines. [14]

One drawback of all three models compared to the baselines is the long prediction time, which is caused by the GAN training required during prediction of each sample. While the prediction of a sample in the baseline algorithm does not even take a second, it takes about a minute or even more for the models.

Concluding, the first experiment showed that all three compared models were able to regress fundamental matrices for silhouette images in a quality only slightly worse than the baseline algorithm does on the texture dataset. Although the estimation quality might not be good enough for some applications, the models still outperformed the baseline algorithm on the silhouette dataset, because that algorithm could not even estimate many of the matrices.

6.5.2 Results on Temple Silhouette Dataset

Table 6.2 shows the results of the second experiment, performed on the *temple silhouette dataset*. It is structured like table 6.1. Beside the compared models two baselines are shown in the table that both were computed using the baseline algorithm. The first baseline was computed on the same dataset as the models, the *temple silhouette dataset*, while the texture baseline was computed on the *temple texture dataset* like in table 6.1. It was just added for better comparison. Also, it needs to be noted that the *temple texture dataset* was used to create the *temple silhouette dataset*, so the second one is just the silhouette version of the first one.

In the results it can be seen that all three models were worse than on the *synthetic horse dataset*. For some metrics the results are worse by factor 10 or sometimes even more. This makes sense as the models have only seen images of the horse 3D-model

[14] It should be noted that the silhouette images and the texture images do all just show single objects instead of whole scenes. This might be a reason why the baseline algorithm does not perform better on the texture dataset.

Model		**Model 1**	**Model 2**	**Model 3**	**Baseline**	**Texture***
derived features		none	0, 1, 2, 3	0, 1, 2, 3	-	-
image pair input		no	no	yes	-	-
SIFT	**epi-abs**	5.62e+02	**1.48e+02**	1.54e+03	1.94e+01	1.31e+01*
	epi-sqr	9.10e+05	**1.00e+05**	3.80e+06	2.54e+04	1.89e+03*
	ssd	1.80e+03	2.51e+03	**1.58e+03**	4.11e+02	6.23e+02*
	sed	1.01e+04	1.80e+04	**8.98e+03**	2.97e+03	3.75e+03*
RANDOM	**epi-abs**	**3.47e+05**	8.55e+05	9.61e+05	6.58e+04	6.99e+03*
	epi-sqr	**1.44e+14**	1.38e+16	1.25e+15	4.54e+13	2.97e+11*
	ssd	5.05e+03	6.99e+03	**4.38e+03**	5.45e+03	2.97e+03*
	sed	**4.17e+11**	4.01e+13	5.83e+11	1.54e+11	7.54e+11*
Missing Samples**		0/192	0/192	0/192	50/192	45/192*

Table 6.2: Experiment Results on *temple silhouette dataset*
* Baseline on texture dataset: These metrics were not computed on the *temple silhouette dataset* but on the *temple texture dataset*. It is the same dataset as used in the texture baseline of table 6.1.
** The number of samples that could not be estimated, because not enough matching points could be found. These samples were ignored when computing the metrics.

during training and never seen images of the temple, which means that they overfitted on the horse 3D-model. To get better results on completely unseen scenes the model would need to be trained on larger datasets (see section 7.1). But, despite the overfitting, the results for the *ssd* metrics are similar to the tests on the *synthetic horse dataset*, and for the *SIFT sed* metric the results are even better. This shows that the models did not only learn dataset specific features but also some general features relevant for the fundamental matrix. This generalization ability may have been enabled by learning the image relation during GAN training and then using the learned results for regression.

The baseline results are better than the results on the *synthetic horse dataset*, because for the *temple silhouette dataset* it could estimate a lot more matrices. This may be, because the dataset is easier to use by the baseline algorithm. But it also shows a drawback of the models compared to the baseline algorithm: While the models strongly depend on the datasets that were used for training and perform worse on completely different datasets, the baseline algorithm by design generalizes well as it was not trained using a specific dataset.

In conclusion the second experiment showed that, although the models did not fully generalize and are worse than the baseline algorithm when confronted with completely different data, they still have some generalization ability.

6.6 Analysis

By analyzing the influence of different hyperparameters [15] on the results, the regressor training histories, the GAN training histories, and the generated images, two main problems of these models could be identified: Instability of GAN training and overfitting

[15]Especially: Learning rates, number of channels and regularization methods of the generator and discriminator; the number of GAN training epochs for each sample.

during GAN training.

6.6.1 GAN Training Instability

The first problem is the instability of the GAN training. GANs are typically hard to train because the generator and the discriminator are trained adversarial. If one of them gets much better than the other one in some epoch, then the other one might not be able to learn anything anymore from this point on. To achieve high training stability the hyperparameters of the GAN, especially the learning rates and model capacities, have to be optimized very precisely. In this project that optimization is very hard, because the same GAN model is not only used for one learning problem but for a set of related learning problems. Each sample pair defines a new learning problem, for which the GAN needs to be trained. But its hyperparameters cannot be optimized for a single image pair, as the dataset contains many different of them. Another problem is that the GAN is trained with only a single sample at a time. If the training becomes unstable in some epoch, the whole GAN training for that sample fails and cannot get stable anymore. Whereas, if mutiple samples were used, the other samples in one batch could hold the training stable when a single sample has problems with training stability. These described problems make training of the GAN model in this project very hard, but a stable training process is required to learn meaningfull features for the regressor. To attack these problems the following was done: First of all, the learning rates used in the GAN (generator and discriminator) were choosen to be very small. In this way neither the generator nor the discriminator can get too good for the other one too fast. Also, the regressor uses many batchnorm layers, especially on the bottleneck and derived feature inputs. These layers help the regressor to handle large input values which might have been produced by some unstable sample. Using these layers, the regressor can stay stable although the GAN training for some samples may have been unstable.

6.6.2 GAN Overfitting

Another problem that occurred was overfitting during GAN training. The regressor uses the features of the GAN as input, thus it is important that these features contain very much information about the image relation. This means that the GAN needs to learn the image relation while not overfitting on the image pair. If it overfits on the image pair, and the generator learns to generate the second image in a constant way, independent from the input image, then the features in the GAN would not contain any information about the image relation. They would be worthless for the regressor. But as the GAN is trained only with a single image pair at a time, overfitting is very likely to happen. To prevent this there are several strategies: First, the small generator learning rate, which is already needed for training stability, lets the generator overfit very slow. In addition, the number of sample epochs was choosen to be as minimal as possible while the generator still learns enough about the image relation. Furthermore, the number of neurons in the bottleneck layer was choosen to be small (128). Additionally regularization is used, especially in the bottleneck block. Experiments showed that dropout helped the most, while weights

and activity regularization did not really improve the results. They also showed that, while a dropout layer before the bottleneck layer improves performance, a dropout layer after the bottleneck drastically hurts the performance, because it destabilizes the GAN training. This might be, because it drastically reduces the capacity of the bottleneck.

The goal of all the techniques used against overfitting is to allow the GAN to learn just enough about the image relation to be valuable for the regressor while not memorizing the generated image.

6.6.3 Training Stability vs. Overfitting Tradeoff

When optimizing the hyperparameters of the model there are situations where either GAN overfitting can be reduced or the training stability of the GAN can be increased, but not both. In these situations it may even hurt the training stability when overfitting is reduced. One example for this is the number of channels and neurons in the generator and discriminator. If these numbers were descreased, then overfitting could be reduced, as it would decrease the model capacity. But experiments showed that this would also make training less stable and thus would lead to worse results. Increasing the number of channels would make training easier, but would also lead to more overfitting. The same is true for regularization methods, especially for weights and activity regularization. More regularization leads to less overfitting but also to less training stability. When using dropout as regularization only before the bottleneck layer but not after it, the described effect was not very large. This is why in the final models only dropout is used for regularization. The reason for the effect being not that large with dropout could be that dropout helps to diversify the learning process while still giving the model a lot flexibility.

This tradeoff between training stability and overfitting only exists for some hyperparameters. There might also be possibilities to optimize one of the aspects without hurting the other. Some of the ideas presented in section 7.2 try to attack this problem.

7

Outlook and Conclusion

After the experiments and their results have been discussed, this chapter covers possible next steps and extensions to this project.

7.1 Further Training and Optimization

Hyperparameter Tuning During this project not all hyperparameters could be tuned, and even those that have been tuned could be optimized even further. For example, the combined training method wasn't even investigated during the final experiments. So there might be a huge potential to improve the models that could be exploited when more time was spend on hyperparameter tuning. This is why further hyperparameter tuning could be the next step after the project.

Larger Datasets Also, during the experiments the model was only trained on a limited dataset which only contained image pairs from a single scene. To get the model ready for real live applications it would need to be trained on much larger datasets which would need to contain millions of images from probable thousands of different scenes.

7.2 Extensions

The models and training procedures could also be extended or adapted in order to produce better results or to solve some of the problems discussed in section 6.6. Some possible extensions are shortly described in the following sections. These could be further investigated in possible followup projects.

7.2.1 Mixed Training methods

One possible extension could be to combine both training methods instead of just using one of them. The model could be first trained with the separated training method (see section 5.1.1), which might be better in adapting the generator and the regressor to each other and might even help to stabilize the training. But combined training is very

slow. This is why after some epochs of combined training the model could be further trained separately (see section 5.1.2) to speed up the training of the regressor. These two training steps could also be repeated several times, which would mean that both training methods were used in an alternating fashion for multiple iterations.

7.2.2 Pre-Training of Regressor

Another extension to the training methods could be to pre-train the regressor before the actual training. In this pre-training the regressor would only be trained with the image pair but without the features from the generator. These features would be set to 0 during pre-training and their weights would be frozen. It is interesting whether the pre-training could improve the overall performance or whether it might even hurt it, because it could make using the generator features during the actual learning process harder.

7.2.3 Additional Loss for Generator

The generator training could also be extended with an additional loss on the generated image output. This loss could for example be the *mean-squared-error* or *mean-absolute-error* loss on the generated image with the second image of the sample as ground truth. It would then be weighted and combined with the generator GAN loss. The additional loss might help the generator to generate the correct image and thus to learn better features for the regressor. It might even stabilize the GAN training. But it could also hurt the performance if the generator gets to good compared to the discriminator or when it overfits and then no meaningfull features could be learned for the regressor.

7.2.4 Conditional GAN

To optimize the GAN training, the discriminator could also be conditioned on the input image, which is given to the generator. The discriminator would then have two inputs. One of them is always the true image. The second is either the generated image (when it tries to classify the fake image) or the true image again (when it tries to classify the true one). This might help the discriminator with classification. But it could also make the discriminator too good so that the generator would not learn anything anymore. Another option would be to condition the discriminator on the regressed fundamental matrix. For that variant some questions would stay open, like how this could work with separated training: there the GAN needs to be completely trained before the matrix can be regressed. Also, this idea might even hurt performance a lot, because in the first GAN training rounds for each sample the regressed fundamental matrix is most likely very wrong and thus the discriminator would condition on a wrong fundamental matrix.

7.2.5 Using Generator Weights as Regression Input

In this project the activities of layers in the generator were used as features for the regressor. This idea could be extended by also using learned parameters like the weights of some generator layers as features. This could be done, because the GAN is trained for each image pair individually and so not only the layer activations contain information about the relation of the images in the sample but also the weights and biases do.

7.2.6 Siamese Regressor Architecture

The regressor architecture could be adapted, so that the downsampling part of it, which processes the image pair and the derived feature layers, is a siamese network. This would mean that the first image with the derived feature layers from the downsampling part of the generator and the second image with the derived features from the upsampling part are processed independently. They could then be combined and processed together before the bottlneck input is concatenated. This idea is based on [8].

7.2.7 Extended Reconstruction Layer

The reconstruction layer of the regressor could also be adapted. In the model used in this project the reconstruction layer just considers the focal length for each of the camera intrinsics of the two cameras. These intrinsic matrices could be extended to more general camera models which might also consider parameters like the principal point, the skew and different focal lengths for x and y to represent non-square pixels. With these extended intrinsic matrices more parameters than just eight, as in the normal reconstruction layer, would be needed. These would be regressed by the previous fully connected layer. An example of an extended reconstruction layer can be found in [8]. [6, p. 157]

7.2.8 Using Multiple Image Pairs with the same Fundamental Matrix

This optimization would require changes in the available dataset. While currently each sample contains a single image pair as input data, this could be extended to multiple image pairs. So each sample would then consist of a list of image pairs and a single fundamental matrix. All of these image pairs of a sample have to be taken with the same camera configuration so that their relations are all described by the same single fundamental matrix. But the images could be taken form different scenes, as long as the two images within each of the pairs are taken from the same scene. These images do not even have to be temporary related. Given this dataset, during training and prediction the GAN would not need to be trained on a single image pair, but could instead be trained on all the image pairs in the image pair list of a sample. This could reduce all the problems caused by training the GAN only with a single image pair, like unstable

GAN training or overfitting during GAN training. The training of the regressor would not be changed.

7.3 Conclusion

The goal of this project was to develop a neural network based model for estimating fundamental matrices given silhouette images. It should learn the relation of the two images in each sample using a GAN, and then use learned features within the GAN as input to a regressor to find the matrix. In order to fullfill this task, first different approaches were implemented to generate training datasets. Then the model was implemented including two different training methods of which one was investigated in detail. In the final experiments of the project, three different variants of the implemented model were compared. All of these three, concrete models could be trained successfully to regress fundamental matrices using just two silhouette images as input. Although these results might not be good enough for some applications, they showed that the models can achieve results that come near to that of classical approaches on texture datasets, and that they can outperform these classical approaches on a silhouette dataset. This is worth even more as the lack of other currently known, promising solutions to learning fundamental matrices only from silhouette image pairs indicates the hardness of the task. The results also implied that the approach used in this project enabled the models to have some generalization ability. By analyzing the model results a few problems of the approach could be identified. For these problems and as general optimizations, possible extensions to the model and the training procedure were described and discussed.

In this project the approach to use learned features of one neural network as input features to another neural network could be successfully implemented. This approach could be applied to fundamental matrix estimation using silhouette images. Despite all the remaining problems, there is a huge potential in such models. Similar approaches might even help to solve other complex learning problems that are not limited to computer vision applications.

A

Model Hyperparameters

Hyperparameter	**Model 1**	**Model 2**	**Model 3**
derived_feature_layers	none	0, 1, 2, 3	0, 1, 2, 3
use_images	no	no	yes
lr_G	9.0e-06		
lr_D	7.0e-06		
lr_R	3.0e-05	3.0e-05	2.2e-05
lr_R_decay	0.96	0.96	0.95
sample_epochs	150		
disrc_iterations	1		
freeze_discriminator	no		
regressor_epochs	25	35	24
regressor_batch_size	32		
train_method	separated		
dataset	synthetic_horse_rotation		
bottleneck_size	128		
generator_channels	64, 128, 256, 512		
generator_skip_connections	yes		
generator_dropout	0.75		
generator_bottleneck_activity_reg	none		
generator_bottleneck_weight_reg	none		
regressor_batchnorm	yes		
regressor_bottleneck_batchnorm	yes		
regressor_derived_features_batchnorm	-	yes	
regressor_channels	-	64, 128, 256, 512, 8	
regressor_dense_neurons	1024, 512		
regressor_dropout	0.75		
reconstruction_type	reconstruct		
norm	fro		
discriminator_channels	64, 128, 256, 512		
discriminator_classifier_neurons	1024, 512		
discriminator_dropout	0.75		

Table A.1: Model Hyperparameters

Bibliography

[1] Opencv python tutorial: Epipolar geometry. https://docs.opencv.org/3.4/da/de9/tutorial_py_epipolar_geometry.html.

[2] Gil Ben-Artzi, Michael Werman, and Shmuel Peleg. Epipolar geometry from temporal signatures and dynamic silhouettes. *CoRR*, abs/1506.07866, 2015.

[3] David Capel, Andrew Fitzgibbon, Peter Kovesi, Tomas Werner, Yoni Wexler, and Andrew Zisserman. Matlab functions for multiple view geometry: vgg_f_from_p. http://www.robots.ox.ac.uk/~vgg/hzbook/code/vgg_multiview/vgg_F_from_P.m.

[4] Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. Deep image homography estimation. *CoRR*, abs/1606.03798, 2016.

[5] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative Adversarial Networks. *ArXiv e-prints*, June 2014.

[6] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, Cambridge, United Kingdom, 2004.

[7] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks. *CoRR*, abs/1611.07004, 2016.

[8] O. Poursaeed, G. Yang, A. Prakash, Q. Fang, H. Jiang, B. Hariharan, and S. Belongie. Deep Fundamental Matrix Estimation without Correspondences. *ArXiv e-prints*, October 2018.

[9] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015.

[10] Leslie N. Smith. No more pesky learning rate guessing games. *CoRR*, abs/1506.01186, 2015.

[11] Guandao Yang, Omid Poursaeed, Hanqing Jiang, Qiuren Fang, Bharath Hariharan, and Serge Belongie. Deep fundamental matrix estimation. `http://www.guandaoyang.com/pdf/FMatrix_CS6670.pdf`.

[12] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3d: A modern library for 3d data processing. *CoRR*, abs/1801.09847, 2018.