Rheinische
Friedrich-Wilhelms-
Universität Bonn

Institute for Computer Science
Department VI
Autonomous Intelligent Systems

Rheinische Friedrich-Wilhelms-Universität Bonn

Master Thesis

Structured and Robust Video Segmentation of Automotive Scenes

Author:
Moritz Austermann

First Examiner:
Prof. Dr. Sven Behnke

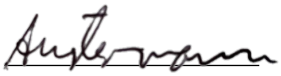
Second Examiner:
Dr. Jens Behley

Date: May 21, 2024

Declaration

I hereby declare that I am the sole author of this thesis and that none other than the specified sources and aids have been used. Passages and figures quoted from other works have been marked with appropriate mention of the source.

Bonn, 20.03.2024



Place, Date

Signature

Abstract

Video semantic segmentation is a common task in image processing and computer vision, which can be used to better understand the content of an image sequence and act accordingly, which is important for example for autonomous robots. The main challenges of video semantic segmentation are to overcome visibility challenges in images like lighting, noise, and occlusion, classification challenges like intra-class variance, and to keep the predicted semantic segmentations temporal coherent to avoid flickering. Many approaches to video semantic segmentation utilize default, strided or dilated convolutional layer to extract features in an encoder-decoder-based architecture. These often include opportunities to reintegrate spatial information and information from previous frames through skip connections, recurrent connections, and CRF- or MRF-based approaches. Shortcomings of these models are, that they don't produce humanly interpretable results and their accuracy can decrease drastically when inputs change, since their intermediate representations are unstructured. Hence, these models aren't robust to changes in the image. Because of this it is hard to understand why a model fails when it does. Furthermore, these models don't have an explicit way to deal with camera transformations between frames, which degrades the usefulness of their past knowledge in scenarios like autonomous driving.

To deal with this problem we create the Recurrent Structured Filter, which is part of a modular model with an encoder-decoder-based architecture. It is composed of one module that predicts the ego-camera transformation, one module that predicts the depth and warps the hidden state to account for the camera transformation given the predicted depth map, and one module that predicts the residual optical flow of moving objects in the image. To train and evaluate our model, we create a dataset containing RGB images, depth maps, camera transformations, semantic segmentation labels, and optical flow of autonomous driving scenes using the CARLA simulation and train and evaluate our model on it. In our experiments, we show that our model outperforms our two baselines on our dataset with a mIoU of 0.2928 and an accuracy of 73.60%, while computing interpretable internal representations.

Contents

1. Introduction

Semantic segmentation is a common task in image processing and computer vision. The semantic segmentation can be used to better understand the content of an image and act accordingly. This is important for example for autonomous robots, who need to make sense of their surroundings and react to them. The task of semantic segmentation can be extended to video semantic segmentation, where we have an image sequence and can use the temporal order of the images.

Semantic segmentation can be formally defined as the task of assigning semantic or class labels c to each pixel of an image x. This makes it harder than predicting a single label for the whole image. The main challenges of semantic segmentation are common visibility problems of images, like lighting, noise, and occlusion. Additional typical problems of semantic segmentation are low inter-class variance and high intra-class variance. If those did not exist a simple pattern matching approach would be sufficient.

Common approaches to solving the task of semantic segmentation are different variants of convolutional models. This includes fully convolutional networks, convolutional models using graphical models, and dilated convolutional models. In addition to that a variety of architectures have been used for semantic segmentation, including Encoder-Decoder-based models, Recurrent neural network-based models, and Multi-scale and pyramid network-based models(Minaee et al. 2022). Many models face the problem that their intermediate representations are not humanly interpretable. This especially becomes a problem when the model performs poorly. In this case the intermediate states of those models offer no way to explain the problem since they are not humanly interpretable, but just learned. Furthermore, many models are vulnerable to small purposeful changes, which alter the visual appearance only slightly but lead models to predict completely wrong results. This can also lead to loss of temporal coherence. Additionally, these models can't handle camera transformations between frames directly, because they often don't have a concept of camera transformations.

In this thesis, we take a look at video semantic segmentation and try to improve the results using a modular model. Our model is made up of the ego-motion filter, which predicts the camera transformation from the image features, the feature filter, which predicts a depth map for the current image and uses it and the

predicted camera transformation to warp the current image state, which is used to grasp the scenes contents. This state and the current image features are then used to predict semantic segmentation labels and residual optical flow for the current image. Our work is based on Wagner et al. 2018.

For our approach, we generate a synthetic dataset for autonomous driving using the CARLA simulator(Dosovitskiy et al. 2017). Our dataset includes RGB images, depth maps, camera transformations, semantic segmentation labels, and optical flow.

In our experiments we outperforms our baselines in both mean Intersection over Union (mIoU) and accuracy. Additionally, we compare semantic segmentation labels for our model, the two baselines, and the ground truth. Furthermore, we analyze predictions of depth maps in comparison with the ground truth and showcase fail cases.

Our main contributions are:

- We propose an encoder-decoder-based modular model, with one module to predict the ego-camera transformation, one module to predict the depth and warp the hidden state to account for the camera transformation given the predicted depth map, and one module to predict the residual optical flow of moving objects in the image.

- We create a synthetic dataset based on the CARLA simulator(Dosovitskiy et al. 2017) for our purposes.

- Our model outperforms both of our baselines in case of mIoU and accuracy on our dataset, while having interpretable representations.

The structure of this thesis is the following.

Chapter 2: Here we explain the fundamentals of convolutional and recurrent networks, which are commonly used for machine learning and are core aspects of our model. Furthermore, we explain the popular encoder-decoder architecture used for many models, including ours. At last, we define optical flow in general.

Chapter 3: In this chapter we talk about the related work of this thesis. This includes approaches to semantic segmentation in general, and DeeplabV3(Chen, Papandreou, Schroff, et al. 2017), DeeplabV3+(Chen, Y. Zhu, et al. 2018) and modular interpretable models(Wagner et al. 2018) in depth as they inspired this thesis.

Chapter 4: We describe our dataset in this chapter. This includes the motivation to make a new dataset and the process of data generation and preprocessing.

Chapter 5: In this chapter we explain our model in detail. This includes how the different modules work and why we assembled them this way. Furthermore, we explain our training strategy and give implementation details

Chapter 6: This chapter contains the evaluation. First, we define the metrics by which we access the performance of our model on the dataset. Then we introduce the baselines in detail. After that we show qualitative and quantitative results and analyze them.

Chapter 7: The last chapter contains our conclusion to this thesis and ideas for future work.

2. Fundamentals

This chapter explains the deep learning fundamental concepts used in this thesis.

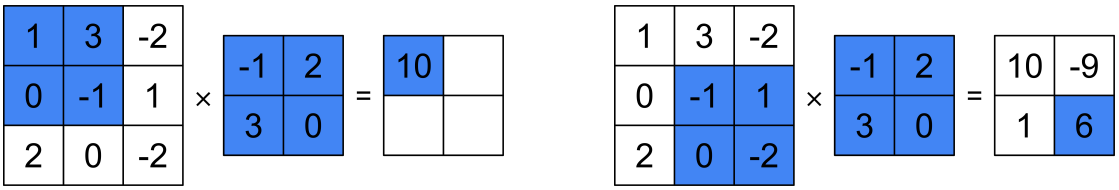
2.1. Convolutional Networks

Convolutional layers are a common approach in neural networks. The reason for this is the reduced number of weights and hence increased training speed compared to fully connected layers. The weight sharing is particularly advantageous for tasks like for example for feature extraction, since these tasks are location independent, which means they can be computed in the same way no matter where they appear in the input. Therefore convolutional layers can focus on improving the extraction of these features while fully connected layers need to train and store a huge number of duplicate weights. Normally several convolutional layers are stacked to extract features of features, which translates to higher level features, and to increase the receptive field of the kernels. Additionally, several parallel kernels are trained to extract different features. The convolutional layers are then followed up by an activation or pooling layer. Activation layers run the features through a non-linear function, like tanh or $ReLu$. This increases the capability of the model since otherwise, the output would boil down to a linear product of the inputs regardless of the depth of the model. Pooling layers take several often adjacent feature values to compute the output. Well-known pooling methods are max-pooling, where only the maximum of the values is propagated, and mean-pooling, where the mean of the values is propagated.

2.1.1. Discrete Convolution

A discrete convolution in this context has a kernel k with a fixed size $m \times n$ and calculates the output y from the input x using k, like Equation (2.1) states. Fig. 2.1 shows an exemplary calculation of a convolution with a 3×3 input and a 2×2 kernel.

$$y[i,j] = \sum_{u=0}^{m} \sum_{v=0}^{n} x[i-u, j-v] \cdot k[u,v] \tag{2.1}$$



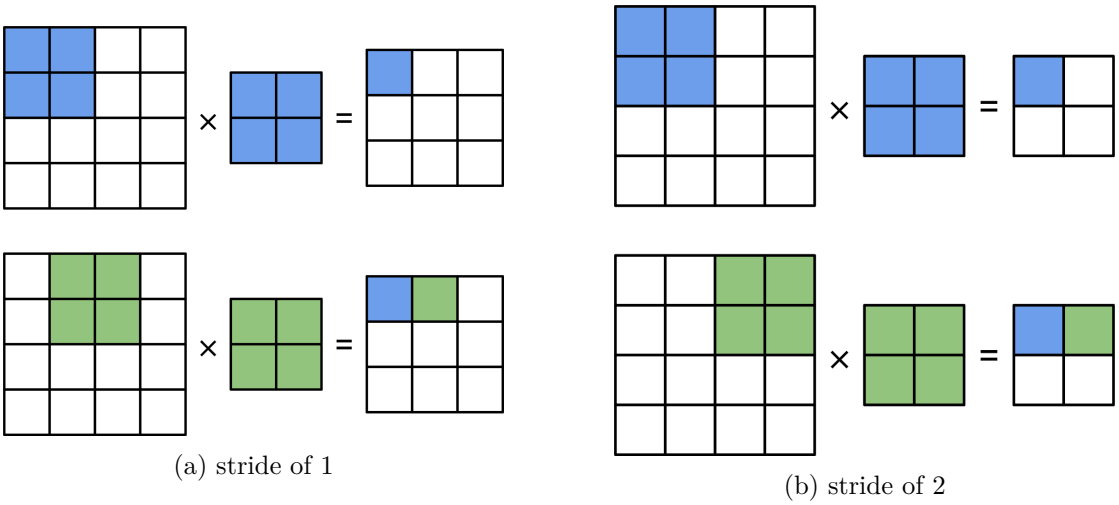
(a) first step (b) last step

Figure 2.1: This shows an exemplary discrete 2D convolution. The colored cells in the input and kernel are used to calculate the colored output cell.

2.1.2. Strided Convolution

Strided convolution is a variant of discrete convolution, which is used to reduce the output size. This type of convolution has a stride s, which determines the number of cells in the input between adjacent cells in the output. Therefore, given an input x, a kernel k of size $m \times n$, and a stride s the output y is calculated as Equation (2.2) states it. Hence, a stride of 1 is equivalent to discrete convolution and strided convolution can be seen as a natural extension to discrete convolution. The effect of different strides can be seen in Fig. 2.2.

$$y[i,j] = \sum_{u=0}^{m} \sum_{v=0}^{n} x[i \cdot s - u, j \cdot s - v] \cdot k[u,v] \tag{2.2}$$



(a) stride of 1 (b) stride of 2

Figure 2.2: This shows an exemplary strided convolution with different strides.

2.1.3. Atrous Convolution

Atrous convolution is a special kind of convolution used to increase the receptive field of convolution without reducing the size of the output and therefore without loss of spatial resolution. Atrous convolutional layers have a kernel k of size $m \times n$ and an atrous rate r and compute their output y at position i, j using the input x like Equation (2.3) states. The atrous rate r determines how far the input positions for any output are apart. An atrous rate r of 1 is equivalent to discrete convolution and atrous convolution can be seen as a natural extension of discrete convolution(Chen, Papandreou, Schroff, et al. 2017). A visual representation of atrous convolutions can be seen in Fig. 2.3, which contains 3x3 atrous convolutions with atrous rates of 2,4,8 and 16, respectively.

$$y[i,j] = \sum_{u=0}^{m} \sum_{v=0}^{n} x[i - r \cdot u, j - r \cdot v] \cdot k[u,v] \tag{2.3}$$

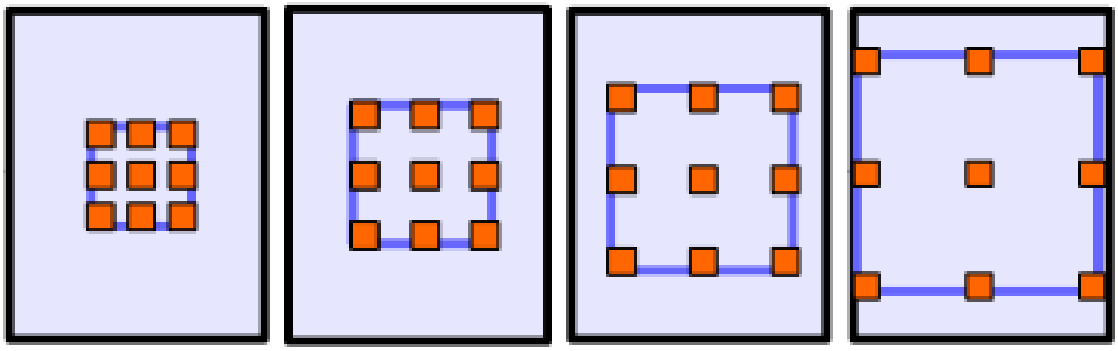


Figure 2.3: 3x3 atrous convolutions with atrous rates of 2,4,8 and 16 (Chen, Papandreou, Schroff, et al. 2017)

2.2. Recurrent Networks

Recurrent networks are feedforward network with added connections to the current or previous layers. The recurrent edges and the feedforward edges form cycles, which lead to the network having a state instead. Furthermore, this network state introduces a notion of time into the computation. The input to recurrent models is an input sequence $\mathbf{x} = (x_0, x_1, ..., x_T)$. The input to nodes with incoming recurrent edges at time step t consists of the current data x_t and the output from the previous network state h_{t-1}. The output y_t at time step t is calculated from the hidden node values $h^{(t)}$ at time step t. Previous input x_{t-1} can influence the output y_t through the recurrent edges(Lipton, Berkowitz, and Elkan 2015). The structure

of a simple recurrent network can be seen in Fig. 2.4. One common problem of recurrent neural networks is the problem of vanishing or exploding gradients. This is because of the repeated integration of weights and because of the possibly long computation paths if connected inputs are far apart in the sequence. There are some variants of neural networks to tackle this problem.

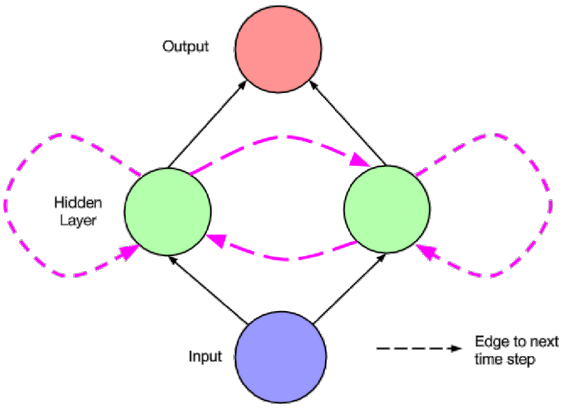


Figure 2.4: A simple recurrent neural network from Lipton, Berkowitz, and Elkan 2015.

2.2.1. Long Short-Term Memory (LSTM)

LSTM models(Hochreiter and Schmidhuber 1997) are neural network, which contain LSTM memory cells. LSTM memory cells are a handcrafted structure and can be seen in Fig. 2.5a. They were designed to combat the problem of exploding or vanishing gradients, and model long-range dependencies.

The internal state s_c is the recurrent part of the memory cell and is the reason for its name. s_c is a node with linear activation and incoming self-recurrent connection, which gives it a state. In this way, the internal state stores its information from one time step to the next.

Since the model needs to react to the input there are several ways to change the internal state s_c. To achieve this gates are used. Gates are designed to filter which information is used and which information is ignored. The functionality of a gate consists of an activation function, like tanh, and a pointwise multiplication. Equation (2.4) shows the computation of a gate G with two inputs a and b. a is fed through tanh and b is multiplied pointwise with the output. This leads to the result, that a is used to decide which parts of b are kept and which parts are ignored and to which extent.

$$G(a, b) = \tanh(a) \circ b \tag{2.4}$$

The functionality of gates is used in different ways to shape the internal state s_c and the output of a LSTM memory cell. First of all, we have the nodes g_c and i_c, where c is used to index the LSTM memory cells. Both nodes receive the input of the current time step $x^{(t)}$ and the output of the hidden layer from the last time step $h^{(t-1)}$. Typically these are summed and fed through an activation function like tanh. The original names for those nodes are input node for g_c and input gate for i_c. Both nodes are used to form a gate, which uses $x^{(t)}$ and $h^{(t-1)}$ for both gate input a and b and whose output is added elementwise to the internal state s_c. In this way, the gate decides what part of the new input can influence the internal state s_c.

The second way to influence the internal state s_c is the forget gate f_c. This gate receives $x^{(t)}$ and $h^{(t-1)}$ as input a for the gate and s_c as input b for the gate. This means, that the forget gate decides which part of the internal state s_c is forgotten from the last time step using the current input to decide.

Combining all of the previous components gives us the full calculation for the internal state $s_c^{(t)}$ at a given time step t, which can be seen in Equation (2.5).

$$s_c^{(t)} = G\left(i_c^{(t)}, g_c^{(t)}\right) + G\left(f_c^{(t)}, s_c^{(t-1)}\right) \tag{2.5}$$

The last part is the output gate o_c, which is a node that also receives $x^{(t)}$ and $h^{(t-1)}$ and is used as input a for a gate. The internal state $s_c^{(t)}$ is then used as input b to compute the memory cell's output $v_c^{(t)}$ at time step t like Equation (2.6) states(Lipton, Berkowitz, and Elkan 2015).

$$v_c^{(t)} = G\left(o_c^{(t)}, s_c^{(t)}\right) \tag{2.6}$$

2.2.2. Gated Recurrent Unit (GRU)

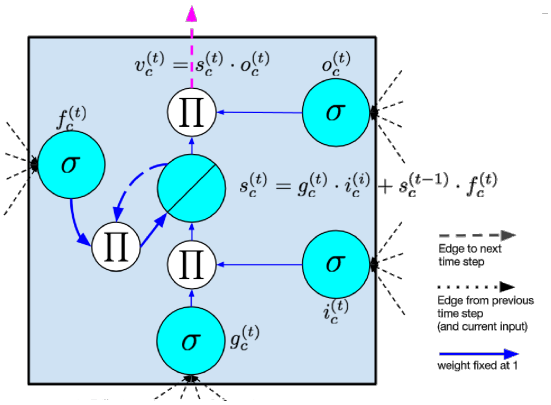
GRUs is another attempt to solve the problem of vanishing or exploding gradients for recurrent neural networks. They are similar in their approach to LSTMs as they have a self-recurrent connection to retain a network state, which is modified using gates. Fig. 2.5b shows the structure of a GRU cell. The first apparent difference is that there is no independent output. For GRU cell, the output h_t is also the cells' state h_t at time step t.

A GRU cell has two inputs, the current input x_t and the previous output of the GRU cell h_{t-1}. These are multiplied with their respective weight W and U, added, and then fed through a sigmoid function like Equations (2.7) and (2.8) state for the computation of z_t and r_t. It should be noted, that those calculations use different weights.

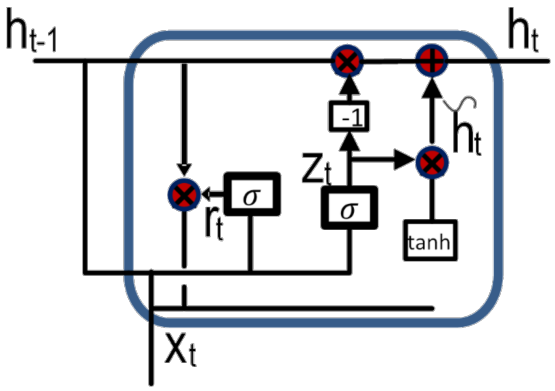
$$z_t = \sigma\left(W^{(z)}x_t + U^{(z)}h_{t-1}\right) \tag{2.7}$$
$$r_t = \sigma\left(W^{(r)}x_t + U^{(r)}h_{t-1}\right) \tag{2.8}$$

r_t is used for the reset gate, which uses a gate function like Equation (2.4) to reset parts of the state h_{t-1} to calculate the intermediate result $\tilde{h}_t$ like Equation (2.9) states. z_t in contrast is used for the update gate, which is used in the final computation of the current output and state h_t. Equation (2.10) shows us the calculation, where z_t is used as a factor to blend the previous state h_{t-1} and the current intermediate state $\tilde{h}_t$ together.

$$\tilde{h}_t = \tanh\left(Wx_t + r_t \circ Uh_{t-1}\right) \tag{2.9}$$
$$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t \tag{2.10}$$



(a) LSTM memory cell with a forget gate from Lipton, Berkowitz, and Elkan 2015. The blue circles represent nodes and the white circles represent element-wise multiplication.



(b) structure of a GRU cell(Zhao et al. 2018)

Figure 2.5: structure of specific recurrent neural networks

2.3. Encoder-Decoder Architecture

The encoder-decoder-architecture is a common neural network architecture consisting of an encoder and a decoder. The basic idea of the encoder-decoder-architecture is shown in Fig. 2.6. The encoder is usually made up of convolu-

tion and pooling layers, which reduce the size, but increase the dimension of the representation. We see, that the input is processed by the encoder to extract a representation of the input, which is contained in the hidden state of the model. This high-dimensional, but small representation is then fed into the decoder. The decoder often consists of upsampling and convolution layers, which increase the shape, but decrease the channel dimension until the output has the same spatial dimension as the input. In this architecture, the state serves as a bottleneck to force the encoder-decoder model to reduce the input into a succinct representation, which is then used to generate the desired output.

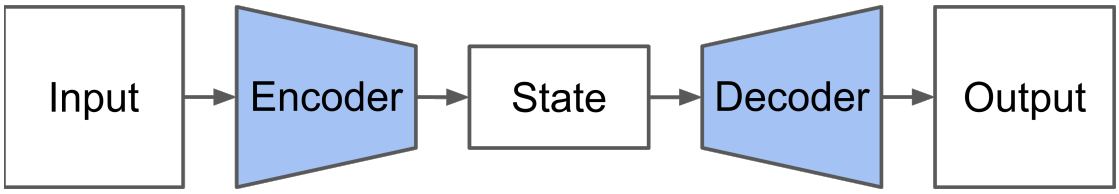


Figure 2.6: basic encoder-decoder structure

The general encoder-decoder-architecture is often improved by using skip connections between layers with output of the same shape. The output of these encoder layers is concatenated to the input of the decoder layers to make spatial information, which gets lost easily in the downsampling encoder operations, available to the decoder. An example of this architecture is U-net(Ronneberger, Fischer, and Brox 2015), whose structure can be seen in Fig. 2.7.

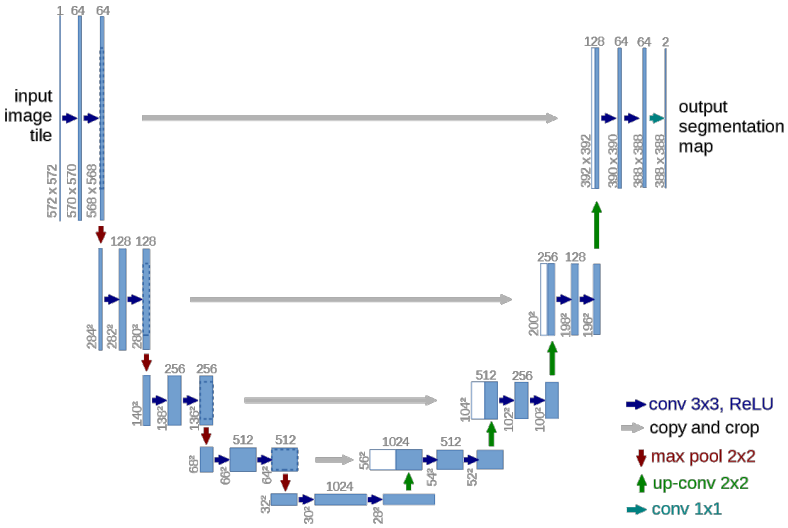


Figure 2.7: Architecture of U-Net(Ronneberger, Fischer, and Brox 2015), where the encoder-decoder-architecture utilizing skip-connections can be clearly seen.

2.4. Optical Flow

The Optical flow between two images H and I is a 2D vector field in the shape of the images describing the movement of pixels. There is the forward optical flow, which describes the movement from one frame to the next, and the backward optical flow, which describes the movement from the current frame to the previous frame. Let's assume, that H and I are consecutive frames in an image sequence and H is the frame before I. If the pixel $H(x, y)$ in the current frame moves to $I(x+u, y+v)$ in the second frame the forward optical flow of position (x, y) would be (u, v) like the following equation states $\overrightarrow{O}(x, y) = (u, v)$. In comparison the backwards optical flow of $(x+u, y+v)$ would be $(-u, -v)$ like this equation shows $\overleftarrow{O}(x+u, y+v) = (-u, -v)$. Fig. 2.8 shows the optical flow from the view of a rotating observer to help the general understanding of what optical flow represents.

From here on we will focus on the forward optical flow, since we use it in this thesis. In the following equations, we derive the calculation of the forward optical flow by generalizing from this specific offset, as a derivative of the image by time.

$$
\begin{aligned}
\overrightarrow{O} &= I(x+u, y+v) - H(x, y) \\
&\approx I(x, y) + \frac{\partial I}{\partial x} \cdot u + \frac{\partial I}{\partial y} \cdot v - H(x, y) \\
&= (I(x, y) - H(x, y)) + \frac{\partial I}{\partial x} \cdot u + \frac{\partial I}{\partial y} \cdot v \\
&\approx \frac{\partial I}{\partial t} + \frac{\partial I}{\partial x} \cdot u + \frac{\partial I}{\partial y} \cdot v \\
&\approx \frac{\partial I}{\partial t} + \nabla I \cdot [u, v]
\end{aligned}
$$

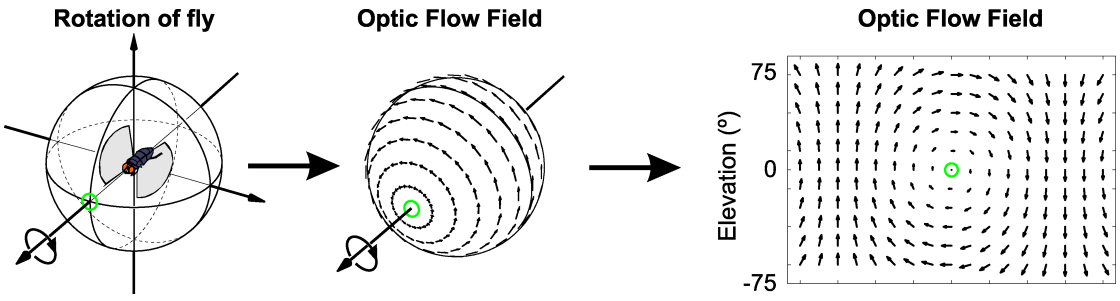


Figure 2.8: Optical flow from the view of a rotating observer(Huston and Krapp 2008)

3. Related Work

In this chapter we describe methods, which are related to this thesis.

3.1. Semantic Segmentation

Semantic segmentation describes the process of assigning a class label to each pixel of an image. It is often used to help make sense of an image and to act accordingly. The common approach for semantic segmentation is to use a deep convolutional model and train it using class labels to output the semantic segmentation directly.

There are several different models, that can be used to predict semantic segmentation. The most common approaches to solving the task of semantic segmentation utilize different variants of convolutional models. The simplest version is fully convolutional networks. Fig. 3.1a shows the structure from Long, Shelhamer, and Darrell 2015, which uses a fully convolutional network for semantic segmentation. We see, that the model is made up of consecutive convolutional layers and pooling or activation layers, which decrease their size and increase their dimension. In the end, the output is upsampled to the desired shape and dimension. In addition to that dilated convolutional models are commonly used, which are convolutional models, that partly contain dilated convolution layers instead of default convolution layers. An example of this is DeeplabV3, whose structure can be seen in Fig. 3.3(Chen, Y. Zhu, et al. 2018). There are also less popular variants of convolutional models, like convolutional models using graphical models, like Conditional Random Fields (CRFs) or Markov Random Fields (MRFs), and convolutional models with active contour models. Fig. 3.1b shows the structure of an exemplary model of this type. Chen, Papandreou, Kokkinos, et al. 2016 presents a convolutional model for semantic segmentation, which works like the fully convolutional model for the first part. In the end, it uses a CRF to improve the precision of the final semantic segmentation. This is necessary since the location invariance of convolutional layers and the low output size of the later convolutional layers lead to a loss of precise location information helpful for exact segmentation. The CRF retains this information and therefore improves the result. The MRF and contour-based models employ a similar approach.

Additionally, a variety of architectures have been used for semantic segmentation, including Encoder-Decoder-based models, which follow the general structure shown in Fig. 2.6. SegNet(Badrinarayanan, Kendall, and Cipolla 2017) is an example of an encoder-decoder-based model for semantic segmentation, whose architecture can be seen in Fig. 3.1c. It consists of 5 blocks of several convolutional layers followed by a pooling layer, which downsamples the shape and increases the dimension of the current features. These blocks have the task of extracting features from the image. After those there are 5 blocks consisting of an upsampling layer, which increases the shape, followed by several convolutional layers. These should bring the output to the correct size while retaining and refining the feature information at their respective positions. The last layer is a softmax layer to generate class labels. Furthermore, recurrent neural network-based models like ReSeg(Visin et al. 2016), whose structure is shown in Fig. 3.1d, are also used for semantic segmentation. ReSeg consists of 4 different recurrent neural networks, which are indicated by the colored arrows. The input is preprocessed by a backbone mode and then fed in turns through sets 2 recurrent neural networks. In the end, the features are upsampled by an upsampling layer and fed through a softmax layer.

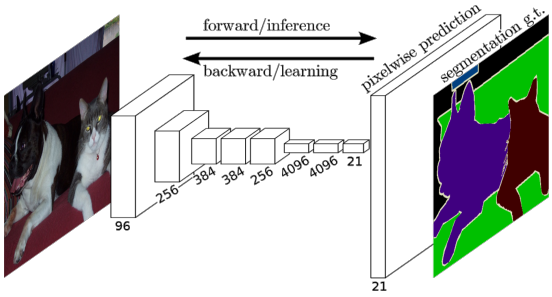
Multi-scale and pyramid network-based models, like DMNet(J. He, Z. Deng, and Qiao 2019), are intended to combat the problem of scale variants of objects in scenes. To achieve that the image is processed at several different scales. Finally, there are also less common approaches like Generative models with adversarial training, where one model tries to predict semantic segmentation labels and an adversarial model tries to distinguish between ground truth and predicted semantic segmentations(Luc et al. 2016). At last there are also Attention-based models, that weights the features that multi-scale approaches produce(Chen, Yang, et al. 2016).

We are mainly going to use the DeeplabV3 and DeeplabV3+ models as our basis for the encoder and decoder. Hence, those are going to be explained in the following subsections.

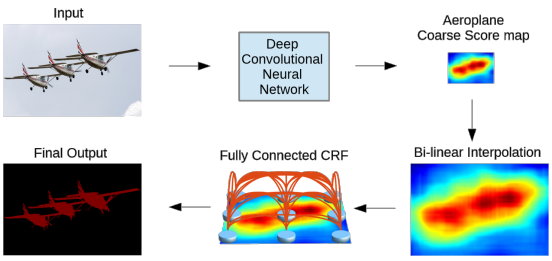
3.1.1. DeeplabV3

One popular model structure is DeeplabV3(Chen, Papandreou, Schroff, et al. 2017), which is based on the idea of using atrous convolutions.

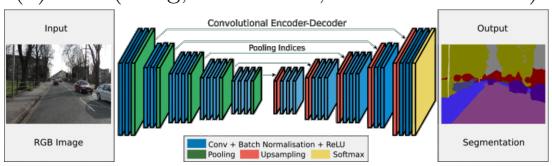
Normally a convolutional model consists of a sequence of convolution and pooling layers. If the stride of the convolutional layers is larger than 1 then the size of the output is reduced. The same is true for most pooling layers. This is fine for a small reduction of the features in comparison to the input but leads to problems if the spatial dimension of the features is reduced too much through deep convo-



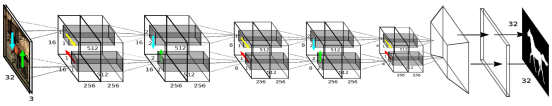
(a) FCN(Long, Shelhamer, and Darrell 2015)



(b) CRF-based convolutional model(Chen, Papandreou, Kokkinos, et al. 2016)



(c) SegNet(Badrinarayanan, Kendall, and Cipolla 2017)



(d) ReSeg(Visin et al. 2016)

Figure 3.1: architectures for semantic segmentation

lutional models. A common approach to fix this is to add upsampling layers and pass information from earlier features of this size through skip connections to help the upsampling process.[Ronneberger, Fischer, and Brox 2015] Instead DeeplabV3 uses a few convolutional and pooling layers to reduce the input to the intended feature size and then follows up with atrous convolutional layers with increasing atrous rate. The reason for this approach is that the key advantage of subsequent convolutional layers is an increase in the receptive field of the later layers. This effect is also achieved with the subsequent atrous layers with increasing atrous rate r, without loss of spatial resolution. This comparison can be seen in Fig. 3.2. The output stride mentioned in the figure is the factor of the input and output sizes and is equivalent to the total stride from the input to the respective step.

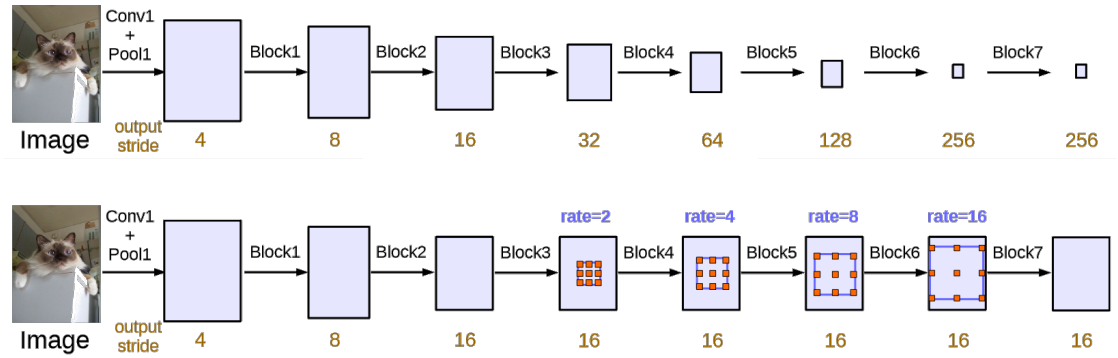
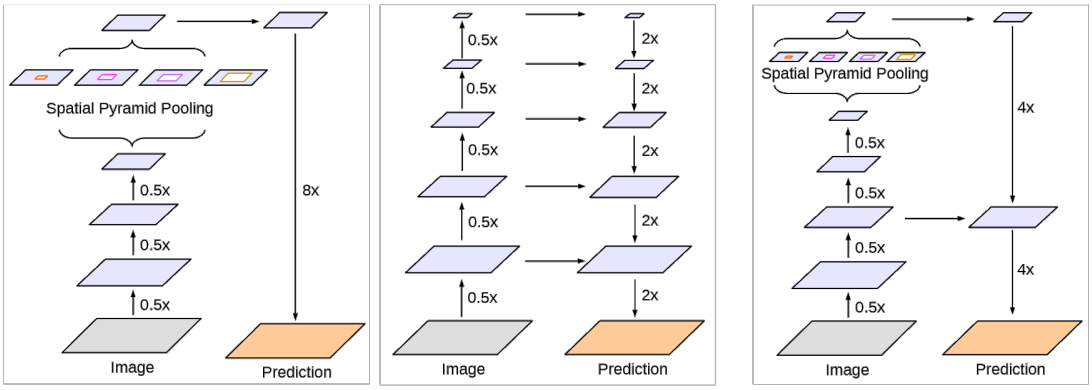


Figure 3.2: Comparison between a standard deep convolutional model and a deep atrous convolutional model like DeeplabV3(Chen, Papandreou, Schroff, et al. 2017)

3.1.2. DeeplabV3+

DeeplabV3+(Chen, Y. Zhu, et al. 2018) tries to improve on DeeplabV3 by incorporating skip connections and a decoder as well as a spatial pyramid pooling module. This can be seen in Fig. 3.3. The DeeplabV3+ model downsamples the image 4 times to half its current size through the use of strided convolution. Additionally, there are skip connections with 1×1 strided convolution from the beginning to the end of each of these blocks. Then a spatial pyramid pooling layer follows. Spatial pyramid pooling is a method where atrous convolutions with different atrous rates are computed at the last step and their output is concatenated. This tries to capture multi-scale information. In DeeplabV3 the output of this is directly upsampled by a factor of 8 to the original size, which makes it hard for the predictions to line up with the object contours, because of the loss of spatial information through the use of convolution and the huge size difference. Therefore, DeeplabV3+ upsamples the dense features first by a factor of 4 and combines them with earlier features of the same size from a skip connection in the semantic decoder. The earlier features have lost less spatial information and can be used to match the object contours, while the upsampled dense features contain information about the correct semantic segmentation. The output of the semantic decoder is then upsampled by 4 again to reach the original input size. This two-step process loses less spatial precision and can benefit from the spatial information in earlier features.



(a) Spatial Pyramid Pooling (b) Encoder-Decoder (c) Encoder-Decoder with Atrous Conv

Figure 3.3: architecture comparison between DeeplabV3 (a), a typical Encoder-Decoder-model using spatial pyramid pooling (b) and DeeplabV3+ (c) (Chen, Y. Zhu, et al. 2018)

3.2. Video Semantic Segmentation

Video semantic segmentation is semantic segmentation on image sequences. A natural first approach for this is to use models trained for single-image semantic segmentation and apply them to every image in the sequence. The drawback of this strategy is that the temporal coherence of subsequent image frames is not utilized. Therefore many models try to incorporate temporal coherence.

This can be done in several ways. One common idea is to pass a feature representation of the images from one frame to the next(Jin et al. 2017). Alternatively some form of postprocessing, like for example CRF-based spatiotemporal reasoning, can be used to refine the image-wise semantic segmentations(Chandra, Couprie, and Kokkinos 2018). Another popular refinement strategy is to use optical flow to project features forward to the next frame(Gadde, Jampani, and Gehler 2017).

Also, additional information can be used to estimate the changes from one image to the next to improve the results of the model. One previous approach used depth and the 6D camera transformation to warp the images accordingly(Wagner et al. 2018).

A different attempt at video semantic segmentation is to improve inference time at the cost of accuracy, which can be important for real-time tasks or scenarios where computational resources are limited. These attempts often reduce the computational cost by forgoing the imagewise semantic segmentation for each image and instead predict the semantic segmentation for a fixed or variable set of keyframes and try to cheaply predict the semantic segmentation of non-keyframes from the result of keyframes(W. Wang et al. 2021). Predicting the optical flow and using it to compute the next frames' semantic segmentation is a classic way to generate results for non-keyframes(X. Zhu et al. 2017). Other efforts are to use a shallow neural network to forecast the semantic segmentation from keyframes to non-keyframes(Jain, X. Wang, and Gonzalez 2019).

This thesis tries to grasp the temporal coherence of successive image frames using a hidden state representation to improve the semantic segmentation compared to the imagewise semantic segmentation. We will use predicted depth and camera transformations to warp the image and optical flow to correct the remaining difference.

3.2.1. Modular and Interpretable Models

All models rely on a good intermediate representation to generate high-quality output, but the previous models have set no limitations to their representations and offer no interpretability of the intermediate representations, which could be

helpful when the model does not perform well or when trying to recover from badly corrupted images. This problem is addressed by the functionally modularized representation filter by using a recurrent design, which enables the model to use previous frames of an image sequence of single frames that go missing or are badly corrupted. To allow recurrent calculations the model has a hidden state $\mathcal{H}$, that can be split into a high-dimensional hidden state $\hat{r}$ to explain the scene features and a low-dimensional hidden state h to explain the scene dynamics. Additionally, the model is split into multiple modules, which handle separate tasks important for information transfer in image sequences and produce humanly interpretable intermediate results, that are known to be important for the task and are used in the prediction.

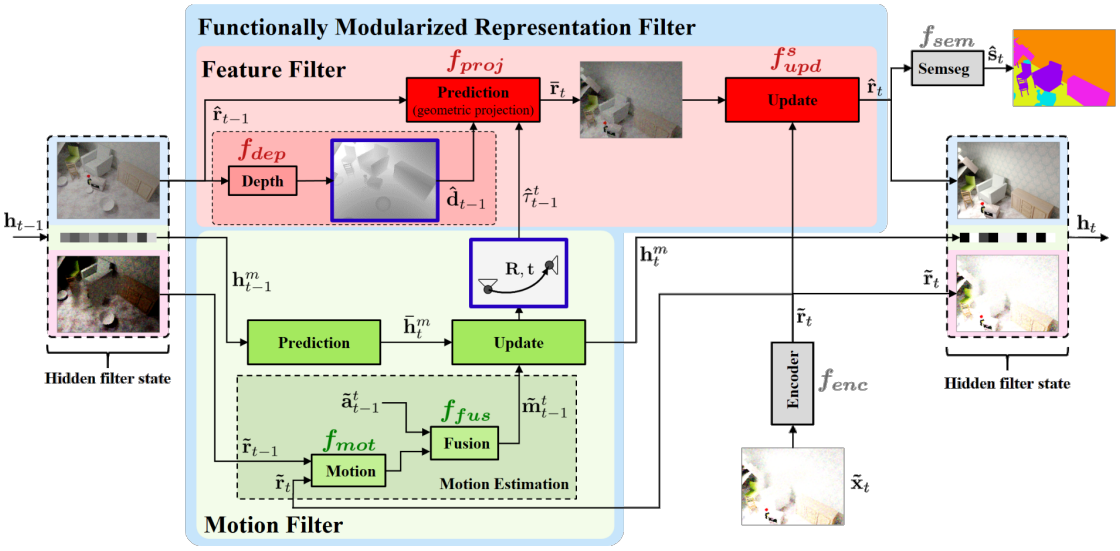


Figure 3.4: This shows one prediction step of the functionally modularized representation filter from Wagner et al. 2018, which is the architectural basis of this thesis.

First, the input frame $\tilde{x}_t$ is processed by the backbone to generate the image features $\tilde{r}_t$ for each time step t.

The first module is the motion filter, which gets the current and last image features $\tilde{r}_t$ and $\tilde{r}_{t-1}$ and the last low dimensional recurrent hidden state h^m_{t-1} as input. The camera motion filter then combines the current and past features using a motion encoder f_{mot} and fusion model f_{fus} with the current acceleration $\tilde{a}^t_{t-1}$ as additional input into camera motion estimate $\tilde{m}^t_{t-1}$. The low-dimensional hidden state h^m_{t-1} is then processed by a prediction model f_{pred} before the output and the motion estimate are then merged by an update gate model f^c_{upd}. The output of the camera motion filter is a 6D translation and rotation of the camera to the next frame $\hat{\tau}^t_{t-1}$ and the new low-dimensional hidden state h^m_t.

The second module is the feature filter module, which receives the previous high dimensional hidden state $\hat{r}_{t-1}$ and the camera motion $\hat{\tau}^t_{t-1}$ of the camera motion filter. The high dimensional hidden state $\hat{r}_{t-1}$ is used by a depth decoder model f_{dep} to predict the current depth map $\hat{d}_{t-1}$. The predicted depth map $\hat{d}_{t-1}$ and the camera motion are $\hat{\tau}^t_{t-1}$ used to warp the high dimensional hidden state using a projection f_{proj}. This warped high dimensional hidden state $\bar{r}_t$ is combined with the current image features $\tilde{r}_t$ using an update gate f^s_{upd} to compute the new high-dimensional hidden state $\hat{r}_t$, which is the output of the feature filter module.

The current high dimensional hidden state $\hat{r}_t$ is fed into a semantic decoder f_{sem} to predict semantic segmentations $\hat{s}_t$.

However, the functionally modularized representation filter was only trained on relatively simple data, where its performance is harder to grasp. Furthermore, the model takes care of the ego-motion, but ignores the motion of the individual objects in the scene. Instead it just hopes, that the semantic decoder can fix that using the current image features. We plan to overcome these limitations, by introducing a new module, the object motion module, which predicts the residual optical flow for each frame. Additionally, we train and evaluate our model on our dataset, which should be much more challenging.

4. Data

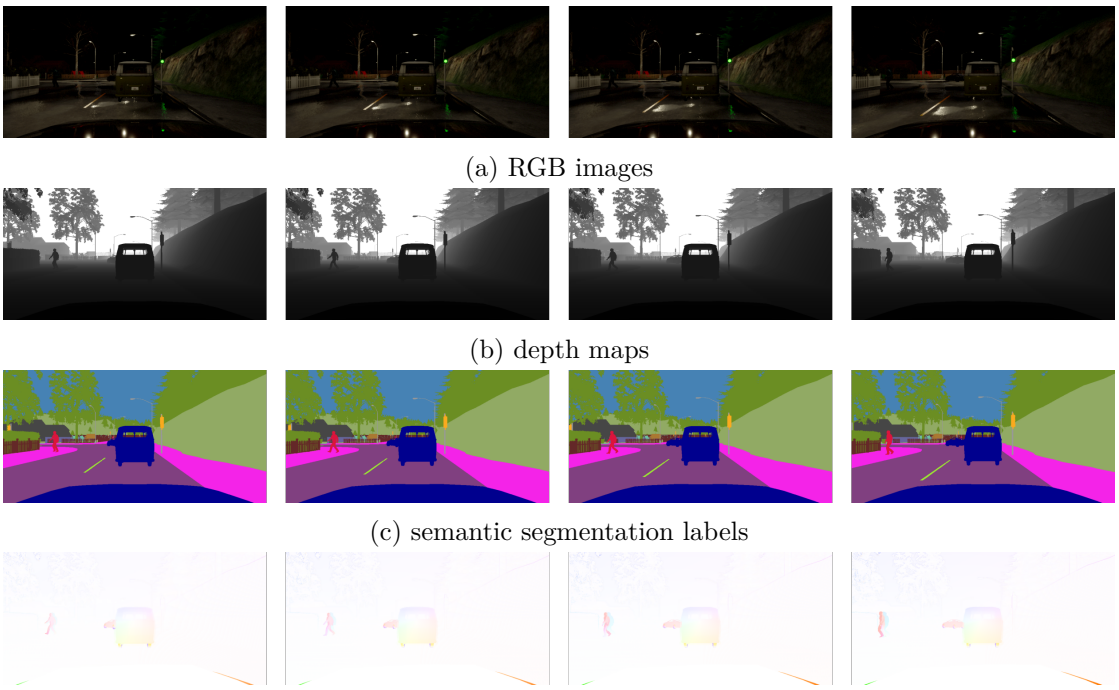
In machine learning high-quality data is needed to train models. Since we use multiple intermediate results to increase robustness and interpretability, we also need to train our model to output these intermediate results. Therefore, we need a dataset, which includes synchronized ground truth data for depth, semantic segmentation, camera transformation, and optical flow, matching the corresponding RGB images. Due to the lack of large-scale real datasets fulfilling these requirements, we decided to use synthetic data, which can be generated to contain all of this information. In Addition to this synthetic data contains no noise from the capturing process of ground truth data. This entails that the transfer of this method to real-world data is more difficult and could face bigger hurdles. Still, it enables much easier testing and development of methods given perfect data.

In future work, this method could be transferred to a real-world dataset, which lacks some of the ground truth data, by either replacing the missing ground truth with pseudo labels generated from a powerful foundation model like "Segment Anything"(Kirillov et al. 2023) or by learning the required representations in a self-supervised manner like SfM-Net(Vijayanarasimhan et al. 2017).

4.1. Data Generation

We generate our dataset using the CARLA simulation. CARLA(Dosovitskiy et al. 2017) is an open-source autonomous driving simulator created to make it easier to train and test autonomous driving simulations, perception algorithms and driving agents. The simulator contains several helpful features to ease data generation. First, it already includes many assets for maps, vehicles, and pedestrians. Additionally, the project can display different weather conditions and times of day. Another helpful feature is the traffic manager, which can control all vehicles and let them move around the map realistically and autonomously. Then it also implements sensors to capture and save RGB images, depth, semantic segmentation, and optical flow. The absolute camera pose can easily be captured as well to enable the calculation of the ground truth camera transformation.

To generate our dataset, we captured 12000 sequences of 20 frames each with a capturing frequency of $6.\overline{6}$ fps. For each time step, we compute an RGB image, a depth map, semantic segmentation labels with 23 classes similar to Cityscapes, forward optical flow for the image, and the 6D camera position. The sequences are generated using the maps "Town01", "Town02", "Town03", "Town04", "Town05", "Town06", "Town07" and "Town10" from Carla. The sequences from "Town02" and "Town10" are used as the validation set and the sequences from the remaining maps are used as the training set. In each map, a handpicked number of random vehicles including cars, bikes, buses, and trucks are spawned and moved around the map using Carla's traffic manager. Additionally, a fixed car is spawned from which the data will be captured. This car will be called the *ego-vehicle*. The ego-vehicle also moves around autonomously by using the traffic manager. Since the focus of this work is on data with a moving camera all sequences where the ego-vehicle didn't move at least 3 meters in the whole sequence were rejected. Fig. 4.1a shows RGB images, depth maps, semantic segmentation labels, and optical flow maps from a sequence. Additional example data can be seen in Appendix A.



(a) RGB images

(b) depth maps

(c) semantic segmentation labels

(d) optical flow from moving objects

Figure 4.1: Generated sequence from our Carla dataset

4.2. Augmentations

Since we use synthetic data our images don't suffer from some problems, that real-world images and real-world scenarios have to account for. To make the model more robust to difficult conditions and increases the amount of available data we apply different augmentations to the images. These augmentations include the following.

The images and all ground truth data of a sequence are mirrored along the vertical axis with a fixed chance. This is done to increase the amount of available data. Since we only do semantic segmentation we don't have to consider problems of left and right lanes changing and can do this augmentation without problems. This does hinder building a bias of vehicle placement based on expected driving behavior. Another augmentation is Gaussian noise that is added to the images, which can be seen in Fig. 4.2a and is supposed to simulate typical image noise. In the next augmentation, a grey clutter is added to the images, which is shown in Fig. 4.2b. We draw red circles around the clutter in the first frame of the sequence to make the clutter augmentation easier to recognize. This should simulate slight occlusions of the image, which could for example be caused by dirt on the camera lens. The last augmentation is illumination changes that are applied to the images. These can be severe and cause the images to be almost completely dark or completely white, which can be seen in Fig. 4.2c. These intense illumination changes can be caused by big changes in brightness in the captured scene or by over- or underexposing the camera to light.

(a) Noise

(b) Clutter

(c) Illumination

Figure 4.2: Generated image sequences with different augmentations applied to them.

4.3. Residual Optical Flow

Section 2.4 explains the general optical flow. Our image sequences are made from a moving observer's viewpoint with other moving objects in the scene. However, the biggest influence for the total optical flow o is the optical flow caused by the movement of the camera itself $\tilde{o}$, which we will call the ego-optical flow because it is caused by the movement of the ego-perspective. Since the camera moves, everything around it, for example, the ground and static objects, seem to have moved from one frame to the next. The second, smaller part of the optical flow is the optical flow caused by the movement of other moving objects $\bar{o}$, like vehicles or pedestrians, what we will call the residual optical flow since it is the remaining optical flow of the scene. We assume that the total optical flow o is the sum of those two types of optical flow, like Equation (4.1) states.

$$o = \tilde{o} + \bar{o} \tag{4.1}$$

$$\bar{o} = o - \tilde{o} \tag{4.2}$$

For our model, it is important to separate the ego-optical flow $\tilde{o}$ and the residual optical flow $\bar{o}$ because we need only the residual optical flow $\bar{o}$ for our training. The CARLA simulator however only captures the total optical flow o_t at time step t. Since we have the depth d_t and camera transformation $\tilde{\tau}_t^{t+1}$ we can calculate the ego-optical flow $\tilde{o}_t$. From there on we can calculate the residual optical flow $\bar{o}_t$ how Equation (4.2) states.

To calculate the ego-optical flow $\tilde{o}_t$ at time step t we take the depth map d_t of shape $N \times M$ and covert it to a 3-dimensional point cloud d_t^{3d} using backprojection. After that we apply the camera transformation $\tilde{\tau}_t^{t+1}$ to the 3 dimensional point cloud d_t^{3d} to get an ego-warped point cloud $\bar{d}_{t+1}^{3d}$. For this, we converted the point cloud d_t^{3d} to homogeneous coordinates and reshaped them to shape $N \cdot M \times 4$. We then project this point cloud $\bar{d}_{t+1}^{3d}$ back into the image using the camera intrinsics c and the point clouds depth $\bar{d}_{t+1}^{3d}[d]$ and end up with an ego-warping map $\bar{w}_t^{t+1}$, which contains for every pixel from time step t the coordinate this pixel will be in time step $t+1$. When we calculate the difference between this and an identity warping map, we receive the ego-optical flow $\tilde{o}_t$. Equations (4.4), (4.5), (4.6) and (4.7) show the calculations explained here, with $A^\frown B$ being the concatenation of A and B and $\oslash$ being the Hadamard division. Furthermore $I(n, m)$ is a 3-dimensional array of shape $n \times m \times 2$, where each position contains the current coordinates, which is explained by the Equation (4.3). Fig. 4.3 shows the results for an example, where Subfig a contains the total optical flow, Subfig b contains

the ego-optical flow, and Subfig c contains the residual optical flow. We see in the total optical flow in Fig. 4.3a, that everything moves towards us and to our sides. This is because the ego-vehicle is moving in this sequence. One remarkable exception is the car in front of us, which does not have any optical flow. This is because the car moves with the same speed as we do and therefore effectively does not move in the image. Additionally, the pedestrians to the left and the left-turning car further in the front are more pronounced in the total optical flow, this is because their movement adds to their total optical flow instead of negating it. To check our observations we have a look at Fig. 4.3b, where we see the ego-optical flow. As expected this does explain much of the total optical flow. Here we can see, that the car in front of us should move because of our movement, and the other car and the pedestrian blur with their background. When looking at Fig. 4.3c we see the moving objects. We can easily make out the two cars and the pedestrian. The only other visible residual optical flow is in the bottom left, which is probably road markings. We should note, that the ego-vehicles hood does not have any optical flow in the Fig. 4.3. This is because we mask the ego-vehicles hood since it always stays the same place and can therefore be masked to avoid confusing the model.

$$I(n,m)[x,y] = (x,y) \tag{4.3}$$
$$d_t^{3d} = I(N,M) \frown d_t \tag{4.4}$$
$$\bar{d}_{t+1}^{3d} = \tilde{\tau}_t^{t+1} \cdot d_t^{3d} \tag{4.5}$$
$$\bar{w}_t^{t+1} = c \cdot \bar{d}_{t+1}^{3d} \oslash \bar{d}_{t+1}^{3d}[d] \tag{4.6}$$
$$\tilde{o}_t = \bar{w}_t^{t+1} - I(N,M) \tag{4.7}$$

(a) total (b) ego (c) residual

Figure 4.3: total optical flow and its' components

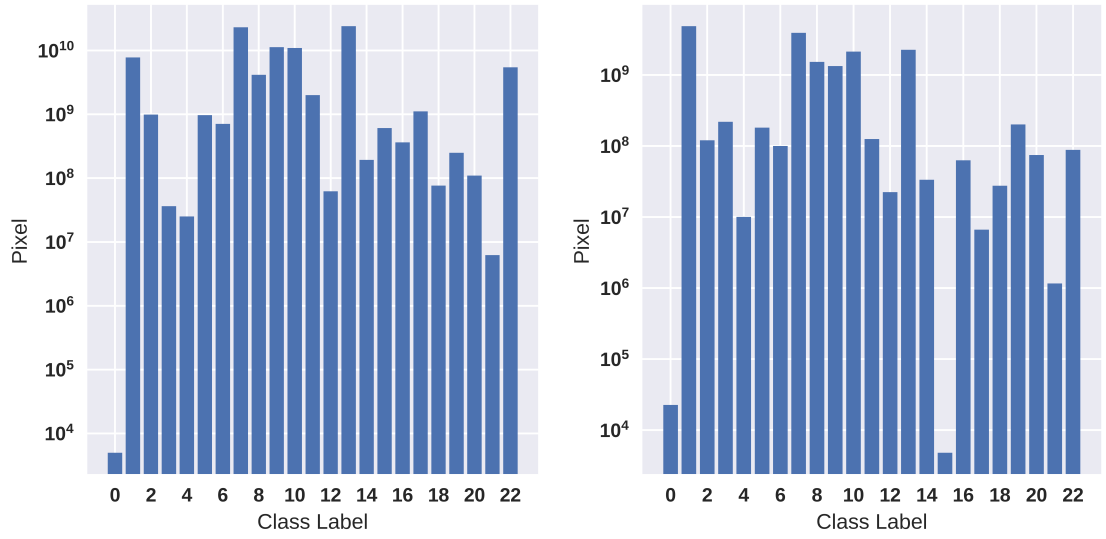
Label	Class	Label	Class	Label	Class
0	Unlabeled	8	Sidewalk	16	Rail track
1	Building	9	Vegetation	17	Guard Rail
2	Fence	10	Vehicles	18	Traffic Light
3	Other	11	Wall	19	Static
4	Pedestrian	12	Traffic Sign	20	Dynamic
5	Pole	13	Sky	21	Water
6	Road Line	14	Ground	22	Terrain
7	Road	15	Bridge		

Table 4.1: all class labels for the semantic segmentation and the corresponding classes

4.4. Class Distribution

We have 23 different classes for the semantic segmentation labels. For each class label the corresponding class is stated in Table 4.1. However, not all classes appear with the same frequency. This imbalance is shown in Fig. 4.4, which is split into training data in Subfig a and validation data in Subfig b.

The most common classes are ”Building”, ”Road” and ”Sky” because they are common in the image and take a large number of pixels in each image. The important class ”vehicles” appear commonly, which is good. Pedestrians are another important class, but they appear much less frequently. This is on the one hand, because each pedestrian takes a smaller number of pixels, when they appear in the image, because of their smaller size. On the other hand is the CARLA simulator designed for car traffic and the CARLA maps allowed only a very limited number of 13 pedestrians to spawn, while some maps contain up to 200 vehicles. We see, that the class ”unlabeled” is rare in the dataset, which is expected for a synthetic dataset. The class distributions of training and validation data seem very similar with one notable exception. The validation data contains almost no bridges, while the training data possesses an adequate amount of bridge pixels. This difference comes down to the fact, that training and validation data is captured on different maps and the maps for validation seemingly contain no or almost no bridges.



(a) Training Set (b) Validation Set

Figure 4.4: histograms for the number of pixels each class occurs in the data

5. Model

Many models face two problems, which we want to address in this thesis. The first one is that small purposeful changes, which alter the visual appearance only slightly, can lead models to predict completely wrong results. This can also lead to loss of temporal coherence. The second problem plays into this. When problems occur the intermediate states of those models offer no way to explain the problem, since they are not humanly interpretable, but just learned.

We use a model made up of modules, which each handle a separate part of the pipeline and produce humanly interpretable intermediate results. This has the additional advantage of making the pipeline more robust since we know that the humanly interpretable intermediate results are important for the task and avoid situations where a small change in the input can mess up the result, although the input seems unchanged. Our model follows an encoder-decoder-based design with the Recurrent Structured Filter as a recurrent neural network, which can be seen in Fig. 5.1. The structure of the Recurrent Structured Filter can seen in Fig. 5.2 and is explained in detail in the following section. The general overview is the following. The backbone $\mathcal{B}$ encodes the images $\tilde{x}_t$ into image features $\tilde{r}_t$. The previous and current image features and the low-dimensional hidden state h_{t-1} are then used by the ego-motion filter $\mathcal{C}$ to predict the camera transformation $\hat{\tau}_{t-1}^{t}$ from the last to the current frame. Meanwhile, the feature filter $\mathcal{F}$ predicts the past images' depth $\hat{d}_{t-1}$ from the previous high-dimensional hidden state $\hat{r}_{t-1}$. The predicted depth and camera transformation are used to ego-warp the previous high-dimensional hidden state $\hat{r}_{t-1}$. This result $\bar{r}_t$ is then used together with the current image features $\tilde{r}_t$ to predict the residual optical flow $\hat{o}_t$ in the object motion filter and to calculate the current high-dimensional hidden state $\hat{r}_t$, which is then used by the semantic decoder to predict the semantic segmentation labels $\hat{s}_t$.

5.1. Modules

Here we break down each module of our pipeline. First, we have a backbone model $\mathcal{B}$, which computes features $\tilde{r}_t$ for the RGB image $\tilde{x}_t$ at time step t, which will be used by multiple modules in our pipeline.

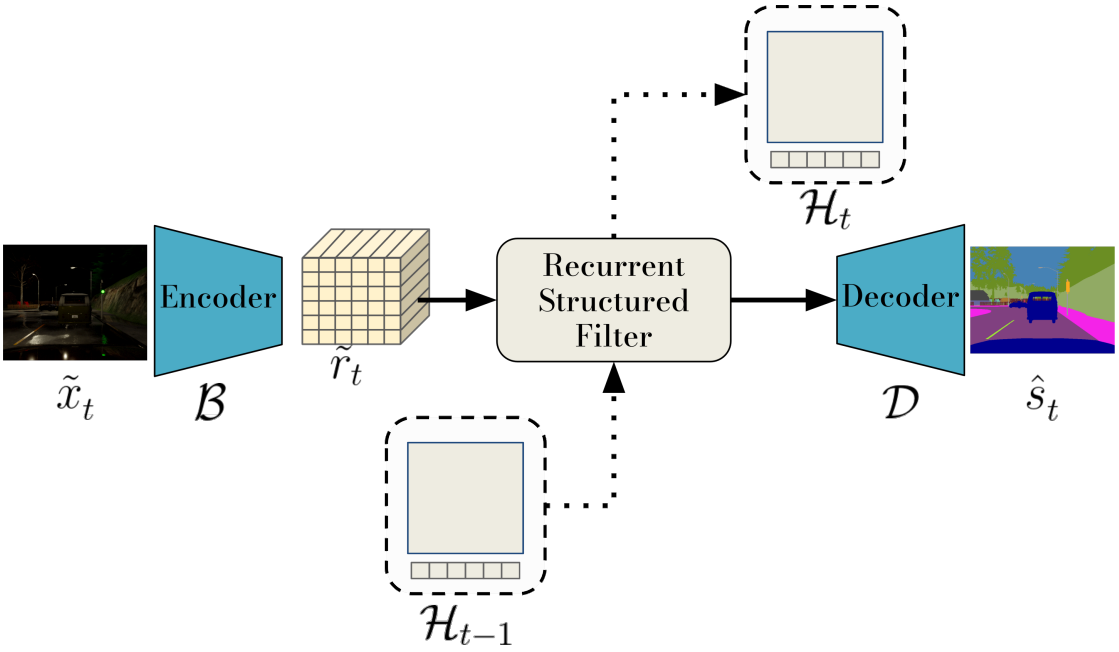


Figure 5.1: Overview of our model: The Encoder $\mathcal{B}$ computes the image features $\tilde{r}_t$ from the images $\tilde{x}_t$ for time step t. These are then fed into the Recurrent Structured Filter, which also receives the previous hidden state $\mathcal{H}_{t-1}$ and outputs the current hidden state $\mathcal{H}_{t-1}$ for the next step and the high-dimensional hidden state $\hat{r}_t$ for the decoder $\mathcal{D}$. Then the semantic decoder predicts the current semantic segmentations $\hat{s}_t$.

The pipeline has a low dimensional hidden state h to grasp the motion of the ego-vehicle and a high dimensional hidden state $\hat{r}$ to explain the scene features, which are used to transfer information from one frame to the next.

5.1.1. Ego-Motion Filter

The first module is the Ego-Motion Filter $\mathcal{C}$, which gets the current and last image features $\tilde{r}_t$ and $\tilde{r}_{t-1}$ and the last low dimensional hidden state h_{t-1} as input. The camera motion filter then combines the current and past features using a motion encoder f_{mot} and camera encoder f_{ce} into a camera motion estimate $\tilde{m}^t_{t-1}$. The reason to have two subsequent models is that the motion encoder might be shared between modules. The low-dimensional hidden state h_{t-1} and the motion estimate are then merged by an shallow fully connected neural network f^c_{upd}. The output of the camera motion filter is a 6D translation and rotation of the camera to the next frame $\hat{\tau}^t_{t-1}$ and the new low-dimensional hidden state h_t. These calculations are clarified in Equations (5.1) and (5.2).

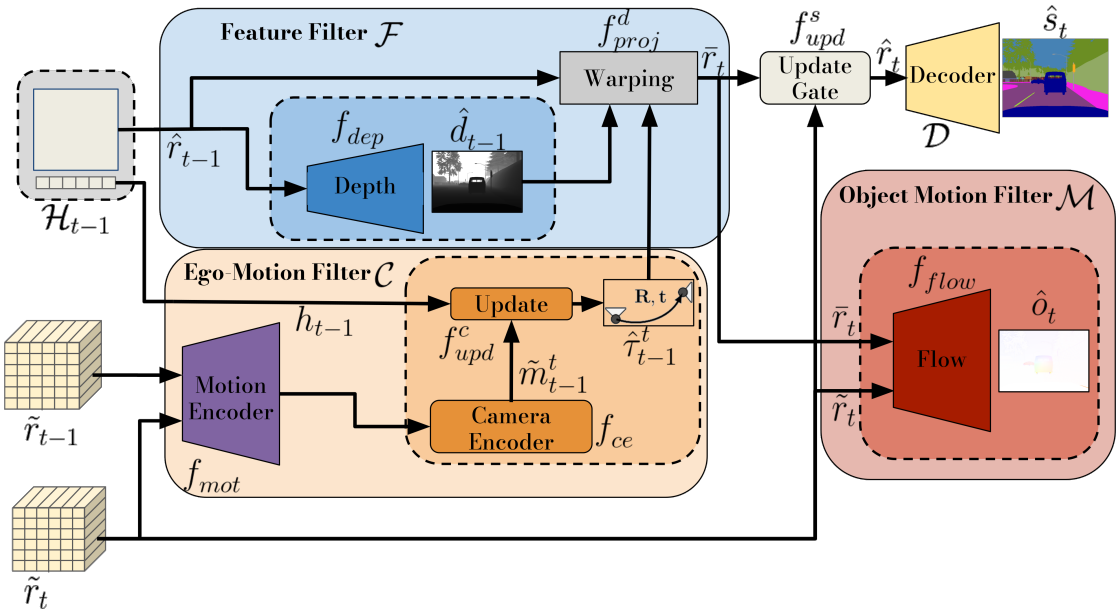


Figure 5.2: Proposed structured recurrent filter: In the top left are the low- and high-dimensional recurrent hidden states, which serve as input for all modules. In the center top is the feature filter module. In the center bottom is the ego-motion filter module and in the bottom right is the object motion filter module. In the top right is the update gate, which produces the next recurrent hidden state. This is also used by the semantic decoder to predict the semantic segmentation.

$$\tilde{m}^t_{t-1} = f_{ce}\left(f_{mot}\left(\tilde{r}_{t-1}, \tilde{r}_t\right)\right) \tag{5.1}$$

$$\hat{\tau}^t_{t-1}, h_t = f^c_{upd}\left(\tilde{m}^t_{t-1}, h_{t-1}\right) \tag{5.2}$$

5.1.2. Feature Filter

The second module is the Feature Filter module $\mathcal{F}$, which receives the previous high dimensional hidden state $\hat{r}_{t-1}$ and the camera motion $\hat{\tau}^t_{t-1}$ of the camera motion filter $\mathcal{C}$. The high dimensional hidden state $\hat{r}_{t-1}$ is used by a depth decoder model f_{dep} to predict the current depth map $\hat{d}_{t-1}$. The predicted depth map $\hat{d}_{t-1}$ and the camera motion are $\hat{\tau}^t_{t-1}$ used to warp the high dimensional hidden state using a projection f^d_{proj}. We do this since we will be using the high-dimensional hidden state to predict the semantic segmentation. And since we predicted a camera transformation and depth map, we can calculate how our surroundings should change if no other object moves. Incorporating this information into the high-dimensional hidden state makes it a much more accurate representation of

the scene content. The calculation of f^d_{proj} is explained below in the subsection about ego-warping. This ego-warped high dimensional hidden state $\bar{r}_t$ is combined with the current image features $\tilde{r}_t$ using an update gate f^s_{upd}, which is explained below, to compute the new high-dimensional hidden state $\hat{r}_t$, which is the output of the feature filter module. These calculations are stated in Equations (5.3), (5.4) and (5.5).

$$\hat{d}_{t-1} = f_{dep}\left(\hat{r}_{t-1}\right) \tag{5.3}$$

$$\bar{r}_t = f^d_{proj}\left(\hat{r}_{t-1}, \hat{d}_{t-1}, \hat{\tau}^t_{t-1}\right) \tag{5.4}$$

$$\hat{r}_t = f^s_{upd}\left(\bar{r}_t, \tilde{r}_t\right) \tag{5.5}$$

Ego-Warping

To calculate f^d_{proj} at time step t we take the current predicted depth map $\hat{d}_{t-1}$ of shape $N \times M$ and covert it to a 3 dimensional point cloud $\hat{d}^{3d}_{t-1}$. After that we apply the predicted camera transformation $\hat{\tau}^t_{t-1}$ to the 3 dimensional point cloud $\hat{d}^{3d}_{t-1}$ to get an ego-warped point cloud $\hat{d}^{3d}_t$. For this, we converted the point cloud $\hat{d}^{3d}_{t-1}$ to homogeneous coordinates and reshaped them to shape $N \cdot M \times 4$. We then project the ego-warped point cloud $\hat{d}^{3d}_t$ back into the image using the camera intrinsics c and the point clouds depth $\hat{d}^{3d}_t[d]$ and ends up with an ego-warping map w^{t+1}_t, which contains for every pixel from time step $t-1$ the coordinate this pixel will be in time step t. The ego-warping map w^t_{t-1} is then applied on the previous high-dimensional hidden state $\hat{r}_{t-1}$ to receive an ego-warped high-dimensional hidden state $\bar{r}_t$. Equations (5.6), (5.7), (5.8) and (5.9) show the calculation of f^d_{proj} with $A^\frown B$ being the concatenation of A and B and $\oslash$ being the Hadamard division. Furthermore $I(n, m)$ is a 3-dimensional array of shape $n \times m \times 2$, where each position contains the current coordinates, which is explained by the Equation (4.3).

$$\hat{d}^{3d}_{t-1} = I(N, M)^\frown \hat{d}_{t-1} \tag{5.6}$$

$$\hat{d}^{3d}_t = \hat{\tau}^t_{t-1} \cdot \hat{d}^{3d}_{t-1} \tag{5.7}$$

$$w^t_{t-1} = c \cdot \hat{d}^{3d}_t \oslash \hat{d}^{3d}_t[d] \tag{5.8}$$

$$\bar{r}_t = w^t_{t-1}\left(\hat{r}_{t-1}\right) = f^d_{proj}\left(\hat{r}_{t-1}, \hat{d}_{t-1}, \hat{\tau}^t_{t-1}\right) \tag{5.9}$$

Update Gate

The update gate f^s_{upd} is similar to the gates in Subsection 2.2.1 based on Equation (2.4). However, our update gate f^s_{upd} is more inspired by Equation (2.10). We add a convolutional layer f^p_c for the ego-warped high dimensional hidden state $\bar{r}_t$, another convolutional layer f^c_c for current image features $\tilde{r}_t$, and a bias b_u. These are then summed, fed through an elementwise sigmoid function sig, and used as weight matrix g_t. The weight matrix g_t contains values in the range $[0, 1]$, that decide what should be kept from the high-dimensional hidden state and what should be replaced by values from the current image features $\tilde{r}_t$. The Equations (5.10) and (5.11) show the computation of our update gate, where $\circ$ is the Hadamard product.

$$g_t = sig\left(f^p_c\left(\bar{r}_t\right) + f^c_c\left(\tilde{r}_t\right) + b_u\right) \tag{5.10}$$

$$f^s_{upd}\left(\bar{r}_t, \tilde{r}_t\right) = g_t \circ \tilde{r}_t + (1 - g_t) \circ \bar{r}_t \tag{5.11}$$

5.1.3. Object Motion Filter

The third module is the Object Motion Filter $\mathcal{M}$, which gets the ego-warped high dimensional hidden state $\bar{r}_t$ from the feature filter module and the current features $\tilde{r}_t$. Since the feature filter already ego-warped the high dimensional hidden state it already includes all movement caused by the ego-movement. Therefore, the flow decoder f_{flow} only needs to predict the residual optical flow $\hat{o}_t$ because the cause for the ego-optical flow is already removed. This step is shown in Equation (5.12).

$$\hat{o}_t = f_{flow}\left(\bar{r}_t, \tilde{r}_t\right) \tag{5.12}$$

5.1.4. Semantic Decoder

The current high dimensional hidden state $\hat{r}_t$ is fed into an sematic decoder $\mathcal{D}$ to predict semantic segmentations $\hat{s}_t$, like Equation (5.13) states.

$$\hat{s}_t = \mathcal{D}\left(\hat{r}_t\right) \tag{5.13}$$

5.1.5. Summary

Therefore, the whole pipeline of our method is shown in Equations (5.14), (5.15), (5.16) and (5.13). In addition to that Equation (5.17) shows the parallel calculation of the object motion filter.

$$\tilde{r}_t = \mathcal{B}(\tilde{x}_t) \tag{5.14}$$
$$\hat{\tau}_{t-1}^t, h_t = \mathcal{C}(\tilde{r}_{t-1}, \tilde{r}_t) \tag{5.15}$$
$$\hat{r}_t = \mathcal{F}\left(\hat{r}_{t-1}, \hat{\tau}_{t-1}^t, \tilde{r}_t\right) \tag{5.16}$$
$$\hat{o}_t = \mathcal{M}(\bar{r}, \tilde{r}_t) \tag{5.17}$$

5.2. Training Strategy

The modular nature of our model and the number of meaningful intermediate results allows us to employ a more sophisticated training strategy to ease the training process and enforce our model to learn interpretable representations.

5.2.1. Loss Functions

Since we have multiple intermediate results as well as semantic segmentation, we also utilize several different losses to force our model to learn all the representations.

We use Cross-Entropy (CE) loss $\mathcal{L}_S$ between predicted $\hat{s}$ and ground truth semantic segmentations $\bar{s}$ to train the semantic segmentation. This is shown in Equation (5.18) for C classes and with ground truth semantic segmentations of shape N and predicted semantic segmentations of shape $N \times C$.

$$CE(\hat{s}, \bar{s}) = -\frac{1}{N}\sum_{n=1}^{N} \log\left(\frac{\exp(\hat{s}_{n,\bar{s}_n})}{\sum_{c=1}^{C}\exp(\hat{s}_{n,c})}\right) \tag{5.18}$$

The loss for the 6D camera position is split into rotation loss $\mathcal{L}_R$ and translation loss $\mathcal{L}_R$. The translation loss $\mathcal{L}_T$ is the L1 norm between the predicted $\hat{t}$ and ground truth translation $\bar{t}$. The rotation loss $\mathcal{L}_R$ is the L1 norm between the Euler angles of the predict $\hat{e}$ and ground truth rotation $\bar{e}$. These losses are also stated in Equations (5.20) and (5.19).

$$\mathcal{L}_R(\hat{e}, \bar{e}) = |\hat{e} - \bar{e}| \tag{5.19}$$
$$\mathcal{L}_T\left(\hat{t}, \bar{t}\right) = \left|\hat{t} - \bar{t}\right| \tag{5.20}$$

We use the L1 norm between predicted $\hat{d}$ and ground truth depth $\bar{d}$ as loss $\mathcal{L}_D$ to train the depth like Equation (5.21) indicates.

$$\mathcal{L}_D\left(\hat{d}, \bar{d}\right) = \left|\hat{d} - \bar{d}\right| \tag{5.21}$$

For the training of the optical flow, we use a loss $\mathcal{L}_O$ inspired by RAFT(Teed and J. Deng 2020). We predict N iteration of the optical flow for each time step with $\hat{o}_t$ being the t-th iteration of the current prediction. Then we calculate the mean L1 norm between each iteration of the predicted and ground truth optical flow $\bar{o}$. These norms are then exponentially decayed by a factor γ. γ is a hyperparameter, which can be set arbitrarily, but we keep it at the recommended value of 0.8 in all of our training. It should be noted here, that the loss decays from the last to the first iteration and not the other way around. The intention behind this is, that the model should focus on predicting the correct optical flow at the end since this is the one that we will use. The earlier iterations are just trained as an intermediate step for the last iteration and therefore their weight is decayed. Equation (5.22) states the calculation of $\mathcal{L}_O$.

$$\mathcal{L}_O\left(\hat{o}, \bar{o}\right) = \sum_{n=1}^{N} \gamma^{N-n} |\hat{o}_n - \bar{o}| \tag{5.22}$$

5.2.2. Training Stages

We train the pipeline in multiple individual steps each focusing on a specific part of the pipeline to make training easier and iterate faster on training approaches.

Multitask Pretraining

The first step is a general multitask pretraining. The architecture is shown in Fig. 5.3. For this training step, we only train on single images or image pairs where necessary. Additionally, we only use mirroring and normalization augmentation for the data in this training step. We start with this simplified training stage to get the core parts of each module to accomplish moderate results. In this first training step, all modules will start from random initialization and therefore will perform poorly in the beginning. The only exception is the backbone, which is pretrained on ImageNet(J. Deng et al. 2009). If we trained the whole pipeline at once from random initialization each module would have to train to predict the correct output, but they would also train to compensate for the poor intermediate results of the other modules. The same can be said inside each module, where each part of the module not only needs to learn to predict the correct result but also to adjust for the poor results of prior parts. This could lead to vanishing or exploding gradients and would lead to a continuous readjustment between the

modules, which slows the training down. Therefore, we start with the minimal possible number of dependent models, while still training as much of our model as possible. For the same reason, we don't start with training whole modules, but just key parts of each module.

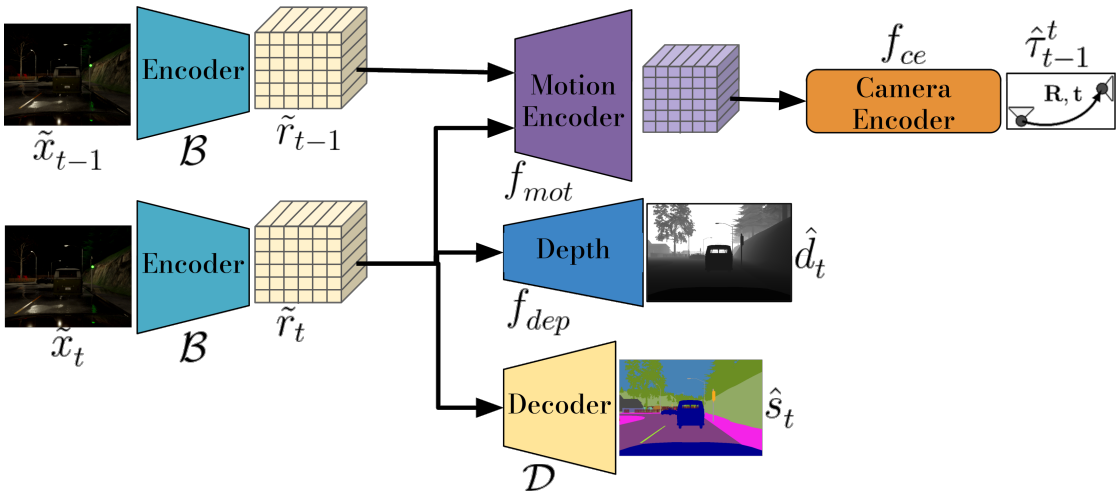


Figure 5.3: Training procedure for the multitask pretraining

The camera predictor is trained relatively close to the full pipeline. We feed two consecutive images into the Motion Encoder f_{mot}. The result is then used by the camera encoder f_{ce} to predict a camera motion estimate $\tilde{m}^t_{t-1}$. Since we don't train with a longer image sequence we don't have a hidden state $\mathcal{H}$. Therefore, we omit f^c_{upd} and predict the 6D translation and rotation of the camera to the next frame $\hat{\tau}^t_{t-1}$ directly. This training is based on Equations (5.1) and (5.2).

For the feature filter only the depth decoder f_{dep} is trained. This submodel would normally receive the previous high-dimensional hidden state $\hat{r}_{t-1}$ as input. As we explained before we don't have the hidden state and for the same reason we are going to substitute it with the image features $\tilde{r}_t$. We base this training on Equation (5.3).

We modified the training of the semantic segmentation decoder just like the training of the depth decoder. The only difference in the pipeline is that the depth decoder receives the previous high-dimensional hidden state and the semantic decoder receives the current high-dimensional hidden state. But since we work on single images only there is no difference and we can use the same image features for both. The training is modified from Equation (5.13).

The optical flow decoder is not trained in this step, because it depends too much on other modules for the previous ego-warped high-dimensional hidden state h_t.

Ego-Motion Training

After the multitask pretraining the ego-motion filter is trained on its own just like the whole pipeline in Fig. 5.2 suggests. The low-dimensional hidden state h_0 is initialized randomly and then learned. From there on the whole ego-motion filter module is trained on image sequences using the rotation $\mathcal{L}_R$ and translation loss $\mathcal{L}_T$. This training is done using only the mirroring and normalization augmentation for the data and can be run in parallel to the feature filter training and the object motion training. We set up the training in this way to train the whole ego-motion filter and hence also the update gate f^c_{upd}, which until now was not trained. Additionally, we now train on image sequences and therefore enable the module to learn to use the low-dimensional hidden state to retain information from one frame to the next.

Feature Filter Training

The feature filter is trained in an additional step after the multitask pretraining. The training procedure is inspired by the training in the functionally modularized representation filter(Wagner et al. 2018). For each image sequence a single image is copied 4 times and a strong noise is applied to 2 quadrants of each image with each quadrant being chosen two times. This can be seen in Fig. 5.4. Before this, the mirroring and normalization augmentation is applied to the image. The model for this training consists of the complete feature filter as it is described in Fig. 5.2 and the semantic decoder. We do this training stage in this way to train the whole feature filter module and the update gate f^s_{upd}, which has been untrained until now. Furthermore, the module and hence the depth decoder are working on the high-dimensional hidden state $\hat{r}$, which enables the module to retain information from one frame to the next. This is necessary since we now train the module on image sequences. Therefore, this training aims to force the model to remember big parts of the output in the form of the high-dimensional hidden state $\hat{r}$, since this is an important part of this module's functionality. For this reason, there is no movement in the sequence since the images are all the same apart from the noise and for the same reason is the noise so strong that the image is lost in the corresponding quadrants. Additionally, we fine-tune the backbone $\mathcal{B}$ to adjust to the now changed input. Here we use the depth L1 loss $\mathcal{L}_D$ and the CE $\mathcal{L}_S$ to train this stage. This training can be done in parallel to the ego-motion filter training and the object motion training.

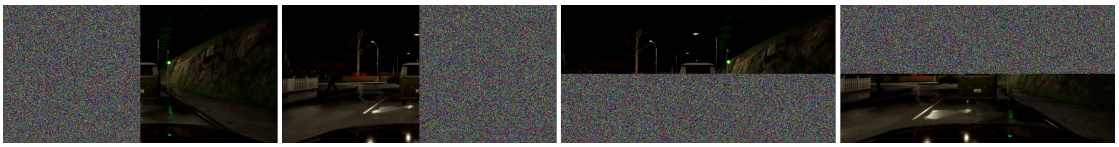


Figure 5.4: Here is a sequence of input images used for feature filter training. As the reader can see each frame has a strong noise applied to it.

Object Motion Training

The object motion filter is trained without the remaining models in a simplified manner with just image pairs and using only the mirroring and normalization augmentation for the data. The ground truth optical flow is calculated as Equations (4.1) and (4.2) state it. To predict the optical flow $\hat{o}$ the complete object motion module from Fig. 5.2 is used. Instead of the current high-dimensional hidden state $\hat{r}_t$ at time step t we use the past image features $\tilde{r}_{t-1}$, which are then ego-warped using the ground truth depth $\bar{d}_{t-1}$ and camera transformation $\bar{\tau}^t_{t-1}$ similar to Equation (5.4), and the current image features $\tilde{r}_t$ as input. The object motion loss $\mathcal{L}_O$ stated in Equation (5.22) is used for the training.

End-to-End Fine-Tuning

The end-to-end fine-tuning is accumulating all trained modules from the ego-motion filter training, the feature filter training, and the object motion training and trains the whole pipeline on image sequences to adjust individual parts to each other. All previously explained losses are used for this training. To make the data more challenging and the model more robust we introduce the Gaussian noise, cluttering, and illumination augmentations for this last training step.

5.3. Implementation Details

We train on images of size 512×1024. In training a random subset of each image sequence from the training data is used, while in validation for a required sequence length of N, the first N frames of the validation data are used. The high-dimensional hidden state is of size $(512 \times 64 \times 128)$ and the low-dimensional hidden state is of size 128.

Our backbone is based on ResNet18(K. He et al. 2015) and we use the pretrained weight from torchvision. But we set the atrous rate for the second to last layer to 2 and the atrous rate for the last layer to 4, which is inspired by DeeplabV3(Chen, Papandreou, Schroff, et al. 2017). The hidden dimension of our backbone's output

is 512. Our motion encoder consists of 3 blocks, which are made up of a convolutional layer with a 3×3 kernel, followed by a batch normalization layer, followed by a ReLu layer. The hidden dimension of all of the blocks is 512. The camera encoder also consists of 3 blocks, which are made up of a convolutional layer with a 3×3 kernel, followed by a batch normalization layer, followed by a ReLu layer. The hidden dimension of the first block is 512, for the second block it is 256 and for the third block, it is 128. Our depth predictor consists of 3 blocks each consisting of a convolutional layer, a batch normalization layer, and a ReLu layer. The hidden dimension of these blocks is 384 for the first and second blocks and 1 for the third block. Furthermore, the kernel size for the first block is 3×3 and 1×1 for the second and third block. Our flow decoder is based on the small RAFT model(Teed and J. Deng 2020). We use both the context and the feature encoder and run the RAFT model for 12 iterations, of which we take the last as output. The semantic decoder receives features from the backbone using skip connections. The hidden dimension of these skip features is 64, but they are still modified by the warping of the feature filter module. These skip features are fed through a block of 1×1 convolutional layer, a batch normalization layer, and a ReLu layer with a hidden dimension of 48. Then the current high-dimensional hidden state is upsampled and concatenated. These combined features are then fed through two blocks of convolution, batch normalization, and ReLu. The kernel sizes are 3×3 and the hidden dimensions are 256. After this follows a dropout layer and a last convolution layer with a hidden dimension of 23, which is the number of classes of our dataset.

The Table 5.1 contains the loss, learning rate, and the number of trained epochs for each of our training stages in our final training.

Training Stage	Loss	LR	Epochs
Multitask Pretraining	$\mathcal{L}_S + \mathcal{L}_T + \mathcal{L}_R + \mathcal{L}_D$	$3 \cdot 10^{-4}$	60
Ego-Motion Training	$\mathcal{L}_T + \mathcal{L}_R$	$3 \cdot 10^{-4}$	90
Feature Filter Training	$\mathcal{L}_D$	$3 \cdot 10^{-4}$	45
Object Motion Training	$\mathcal{L}_O$	10^{-5}	100
End-to-End Fine-Tuning	$\mathcal{L}_S + \mathcal{L}_T + \mathcal{L}_R + \mathcal{L}_D + \mathcal{L}_O$	$5 \cdot 10^{-5}$	60

Table 5.1: Training parameters used by our training stages

6. Evaluation

We evaluate our models and the baseline performance on a set of common metrics. All models will be evaluated on image sequences of 20 frames.

6.1. Metrics

The first metric we use is mIoU, which is a standard metric for semantic segmentation. Equation (6.1) shows the calculation of mIoU given predicted semantic segmentation labels $\hat{s}$ and ground truth semantic segmentation labels $\bar{s}$ consisting of one class per pixel. Additionally, we have C classes, δ_{ij} is the Kronecker-delta of i and j, s_n is the n-th value of the semantic label s, and N is the total number of element our semantic label posses. In mIoU each class is weighted the same in the final averaging. But since the classes "unlabeled", "bridge", and "water" are very rare we are going to omit them in the calculation of mIoU. In addition to that we omit the class "other", because of its unspecific nature.

$$mIoU\left(\hat{s}, \bar{s}\right) = \sum_{c=1}^{C} \frac{\sum_{n=1}^{N} \delta_{\hat{s}_n c} \cdot \delta_{\bar{s}_n c}}{\sum_{n=1}^{N} \delta_{\hat{s}_n c} + \delta_{\bar{s}_n c} - \delta_{\hat{s}_n c} \cdot \delta_{\bar{s}_n c}} \tag{6.1}$$

The last metric that we are going to use is accuracy. Accuracy is the percentage of image labels, that were predicted correctly. Equation (6.2) shows the calculation of the accuracy for predicted $\hat{s}$ and ground truth semantic labels $\bar{s}$, which both have a size of N and both give a single class for each pixel.

$$accuracy\left(\hat{s}, \bar{s}\right) = \frac{\sum_{n=1}^{N} \delta_{\hat{s}_n, \bar{s}_n}}{N} \tag{6.2}$$

6.2. Baselines

We also evaluate different baselines in contrast to our model. The first baseline predicts the semantic segmentation imagewise for the whole sequence. This baseline will be called "imagewise baseline" and its structure can be seen in Fig. 6.1. We use the DeeplabV3+ encoder as the backbone and the decoder as the semantic

decoder. The reasoning behind this baseline is that we want to measure the effect our recurrent structured filter has. When we contrast this baseline with our model in Fig. 5.1, we see that this baseline is like our model, but without the recurrent structured filter. Hence this baseline knows nothing about past frames and works on every frame as an individual image. This baseline does not incorporate temporal coherence in any way. The advantage of this baseline is its simple structure, which makes training especially easy and fast.

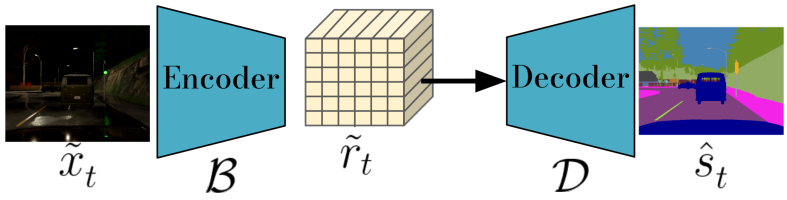


Figure 6.1: This showcases the architecture of the imagewise baseline.

The second baseline is based on convolutional LSTMs. The architecture can be seen in Fig. 6.2. Convolutional LSTMs are LSTM models, which use convolution as an activation function to integrate their inputs before they enter the LSTM memory cell, which is described in Subsection 2.2.1. Our convolutional LSTM baseline has two layers. The intention behind this baseline is to compare how our model performs in contrast to a more general approach to incorporate temporal coherence. The advantage of this baseline is, that it can use past information to improve its prediction in contrast to the imagewise baseline. But this makes more training necessary and this baseline has no inherent structure that guides its predictions.

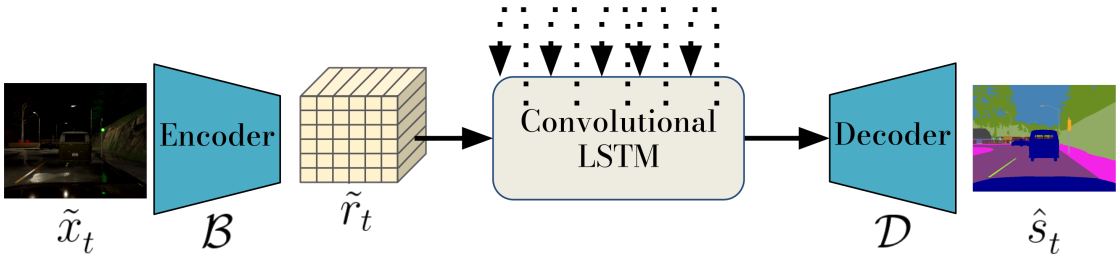


Figure 6.2: This showcases the architecture of the convolutional LSTM baseline.

6.3. Results

In this section we have a look at the qualitative and quantitative results of our model and the baselines.

6.3.1. Qualitative Results

Here we take a look at the qualitative results of this thesis. The two baselines receive RGB images and predict only semantic segmentation labels. Hence, we can only compare the predicted and ground truth semantic segmentation. Our model also returns the predicted depth, which we can also compare to the ground truth.

We have a closer look at semantic segmentation results from Fig. 6.3. Our example is an image sequence of 4 frames. We see that augmentation was applied to the image sequence. This is especially clear from the heavy illumination change in the third frame. We also see the clutter augmentation, although the placement for this is in the middle of the sky should cause no problems for any model. The first and second frames are relatively similar. Hence we first take a look at these. All models can detect the turning car in front of us, but while the convolutional LSTM baseline only detects a very rough outline of the car, is the imagewise baseline able to match the contour more closely and our model produces an almost ideal outline. The large building in the background and the left part of the background are predicted moderately good by our model. In contrast, the imagewise baseline only detects parts of the large building and the left background while making up trees in the background where there are none. The convolutional LSTM baseline predicts both poorly and only guesses rough shapes of the background.

The third frame differs from the second and first frame because of its strong illumination. Our model can make use of its hidden state and predict the turning vehicle and the vehicles in the center with great accuracy particularly given the input. The imagewise baseline also predicts the turning vehicle with great accuracy but loses the center vehicles almost completely. In comparison the convolutional LSTM baseline detects the turning and center vehicles with comparable accuracy to the last frame, which gives it a better result for the center vehicles than the imagewise baseline, but a worse outline of the turning vehicle than both other models. The background prediction quality also stays the same for the convolutional LSTM baseline, which could be explained by the recurrent LSTM memory cells in the baseline, but given its poor prior performance, this still leaves it with a bad result. In contrast, the imagewise baseline cannot benefit from past knowledge and hence achieves bad classifications in the rest of the frame. However, our model can effectively use its hidden state to fill gaps in the image features and predict the remaining image adequately.

The fourth frame is similar to the third frame in that it is overexposed to light. The results in this frame are therefore similar to the results from the frame before.

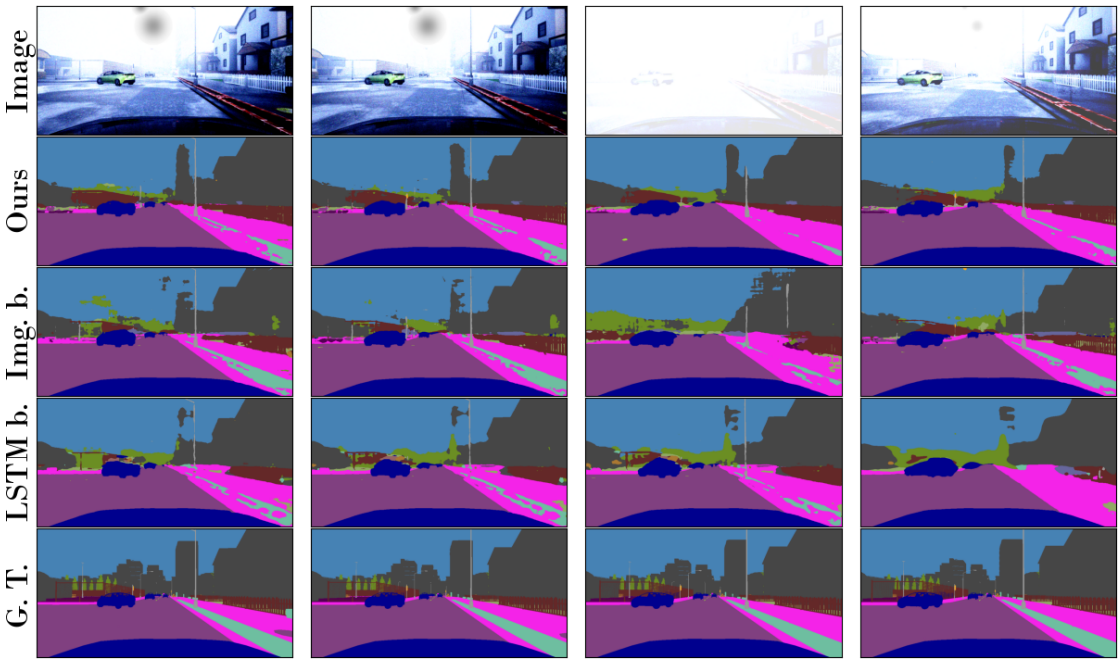


Figure 6.3: RGB images, semantic segmentation predictions for our model and both baselines and ground truth semantic segmentation labels

Furthermore, we take a look at another qualitative result for semantic segmentation. Fig. 6.4 is structured in the same way like Fig. 6.3. We see, that the first, second, and fourth frames are dark but visible, the third frame however is almost completely black.

In the first frame, all models detect the car in front of the ego-vehicle with good accuracy. Furthermore, they also detect the remaining image pretty well except for the right background, which is too dark.

The second frame looks similar to the first and is detected equally well.

The third frame is completely dark, which can be handled easily by our model. The prediction of our model is like its prediction before, which makes for a good prediction. The car in front of the ego-vehicle just loses its sharp contours. The same is true for the convolutional LSTM baseline, which predicts are blurred version of its previous prediction and therefore achieves a good result. The imagewise baseline cannot use past knowledge and gives a pretty poor prediction. Most importantly it detects only a small part of the vehicle in front of the ego-vehicle, which could be interpreted as a vehicle, that is much farther away by a subsequent autonomous driving agent. This false estimate could lead to a dangerous crash. Additionally, the right half of the prediction of the imagewise baseline is bad. It detects vehicles, that are not there, and misses a part of the sidewalk.

The fourth frame is again similar to the second frame and all models return to their previous prediction quality.

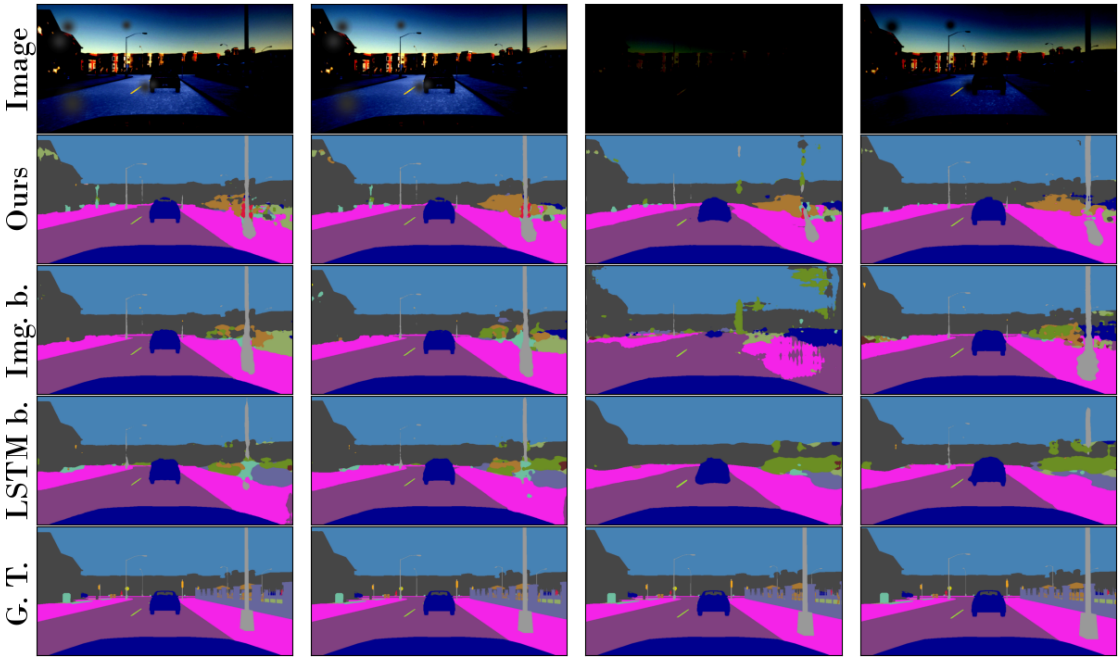


Figure 6.4: RGB images, semantic segmentation predictions for our model and both baselines and ground truth semantic segmentation labels

More qualitative comparisons of the semantic segmentation labels between our model and the two baselines can be seen in Appendix B.

Since our model also predicts depth maps, we have a look at the depth predictions as well. In the Fig. 6.5 we see the RGB images, the predicted and ground truth depth maps, and the weight matrix of the update gate f^s_{upd}, which updates the high-dimensional hidden state, which will be used for the depth prediction in the next frame. For the update gate white means, that the current image features are used for the output, and black means, that the ego-warped high-dimensional hidden state is retained.

We see, that the depth prediction in the first frame is already good. This is the cause because we omit the first depth prediction, which is done using only the learned initialization of the high-dimensional hidden state without any input from the image. Instead we shift the depth predictions by one step, which matches with the way we trained this filter. Since the first hidden state is only learned it is not surprising that the update gate almost exclusively takes the image features as the new high-dimensional hidden state. The only exceptions are some parts, which are sky or road in our image and are in places, that most often have those classes in these positions. Hence, there is no necessity to change the state.

The second predicted depth is very good. The update gate uses image features especially in the position of the turning car since it moves on its own and its movement cannot be explained by the ego-warping of the state.

The third image is barely visible. Nonetheless, the depth map looks just like the depth maps before. This is because the update gate was able to filter the bad visibility and the corresponding poor image features of the frame out. As we can see the update gate takes almost only the ego-warped high-dimensional hidden state since the image features are probably poor. A notable exception is again the turning car.

The fourth frame is almost identical in its result to the second frame, although its visibility is a bit worse. We notice, that the ego-vehicle hood is always using the current image features, which is probably explained by strong biases making the value in these positions useless because the hood is always going to be in this position.

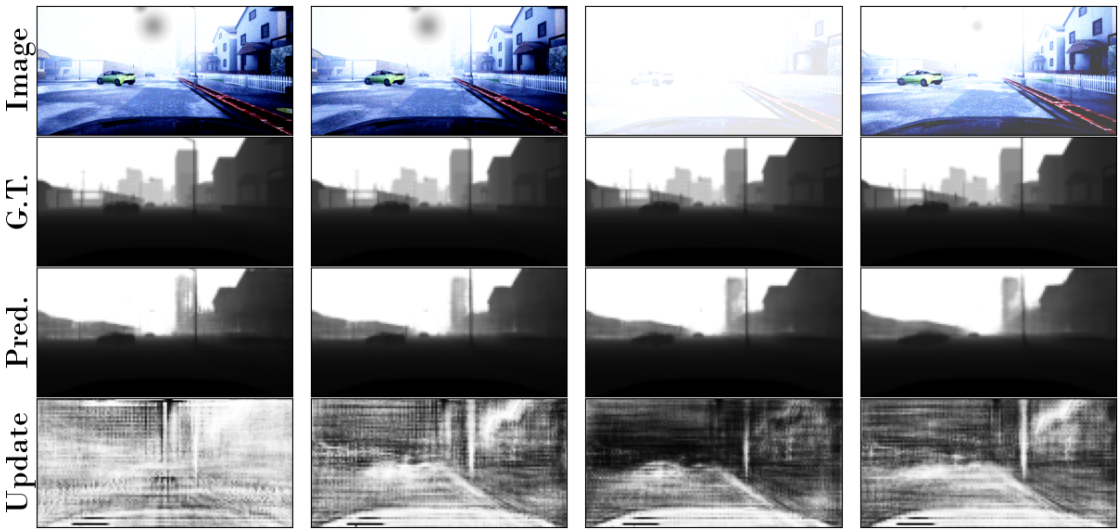


Figure 6.5: RGB images, predicted depth maps and weight matrix of the update gate f^s_{upd} from our model and ground truth depth maps

More qualitative comparisons of the depth maps and update gates predicted by our model and the ground truth depth maps can be seen in Appendix B.

Fig. 6.6 contains results from our model. The figure shows the RGB image in the top row, the weight matrix of the update gate f^s_{upd} in the center row, and the semantic segmentation in the bottom row. The weight matrix of the update gate f^s_{upd} indicates if the previous ego-warped high-dimensional hidden state $\bar{r}_t$ or the current image features $\tilde{r}_t$ is used for the next high-dimensional hidden state $\hat{r}_t$, which es then used by the semantic decoder. This means, that the update gate images indicate, which part of the current image influences the state, which is

used for the predicted semantic segmentation in the same column. The darker the image of the update gate is, the more of the state is kept and the brighter it is the more of the image features are adopted.

We see, that in the first frame, the update gate is mostly very bright, which seems intuitive, since we expect the model to use the image features almost exclusively for the first frame since we haven't integrated anything from the image yet. Only a narrow part in the center of the frame is kept. This part contains mostly the ego-vehicle hood and the road directly in front of the ego-vehicle. The reason for this is, that the starting high-dimensional hidden state is learned and can therefore incorporate biases.

The update gates in the second, third, and fourth frames are similar. We see that the car in front of us, and the trees are mainly using the current image features. For the car, this is because the current ego-warped high-dimensional hidden state already includes our movement, but not the movement of the car. Therefore we need the image features to account for that. In the case of the trees, the reason for using the image feature is probably to gain sharper contour predictions, since trees have a jagged outline and are harder to predict exactly. Another reason the update gate uses the image features for the car and the trees is to fill disocclusion points. The remaining parts of the image are mostly taken from the ego-warped high-dimensional hidden state because it already contains the ego-warping and explains the changes to static objects with sufficient precision.

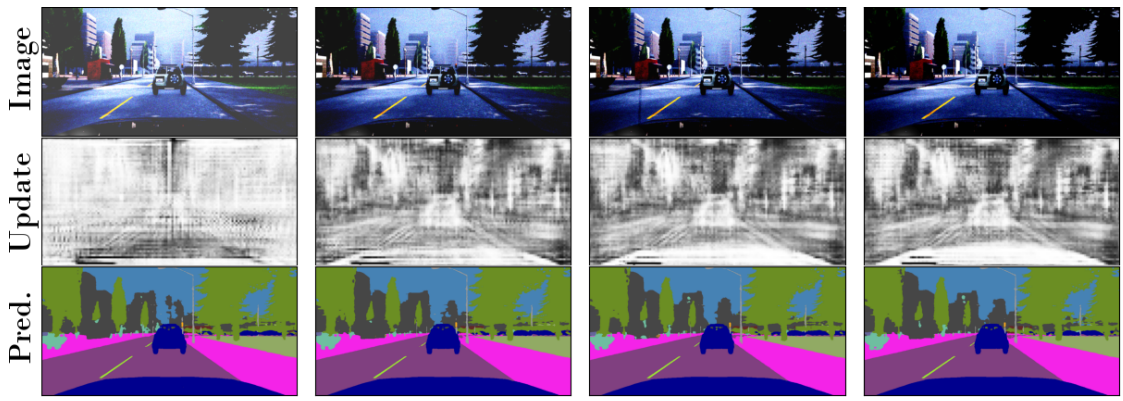


Figure 6.6: This figure contains RGB images, weight matrix of the update gate f^s_{upd}, and semantic segmentations from our model. For the update gate black means, that only the eqo-warped high-dimensional hidden state $\bar{r}_t$ is kept, and white means, that only the current image features $\tilde{r}_t$ are kept. The brightness determines the blending between them.

Our dataset also contains image sequences, where our model fails to achieve good results. Fig. 6.7 contains such a sequence. We see the RGB images in the top row,

our models predicted semantic segmentations in the center row, and the ground truth semantic segmentations in the bottom row. We see strong discrepancies regarding the sidewalk and the trees in the background. Furthermore, the shape of the vehicle in the front and to the left is rough. At last, the two vehicles to the right in the back are missing. When looking at the image it reveals itself, that this result is due to the poor visibility, because of the darkness. Taking this into account makes the detection of the two vehicles and the detection of the traffic light good. Additionally, there is no way to detect the trees or vehicles in the background, since they are not visible. The sidewalk is also not visible and was probably only detected, because of the bias in the trained high-dimensional hidden state.

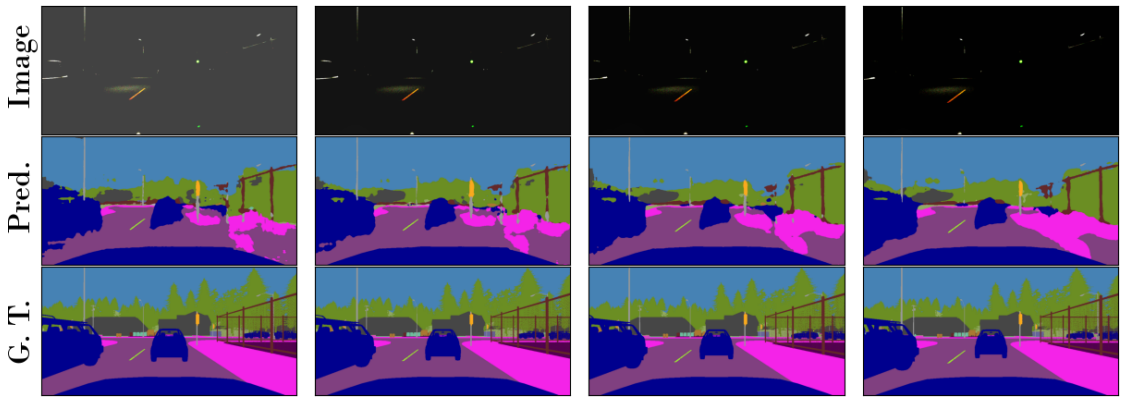


Figure 6.7: This figure contains a case where our model fails. In the top row are the RGB images, in the center row are our model's predictions, and in the bottom row are the ground truth semantic segmentation labels.

6.3.2. Quantitative Results

In Table 6.1 we have a look at the quantitative results for our model as well as the two baselines. We see, that our model performs slightly better than the imagewise baseline and noticeably better than the convolutional LSTM baseline in mIoU. For accuracy, we perform better than both baselines.

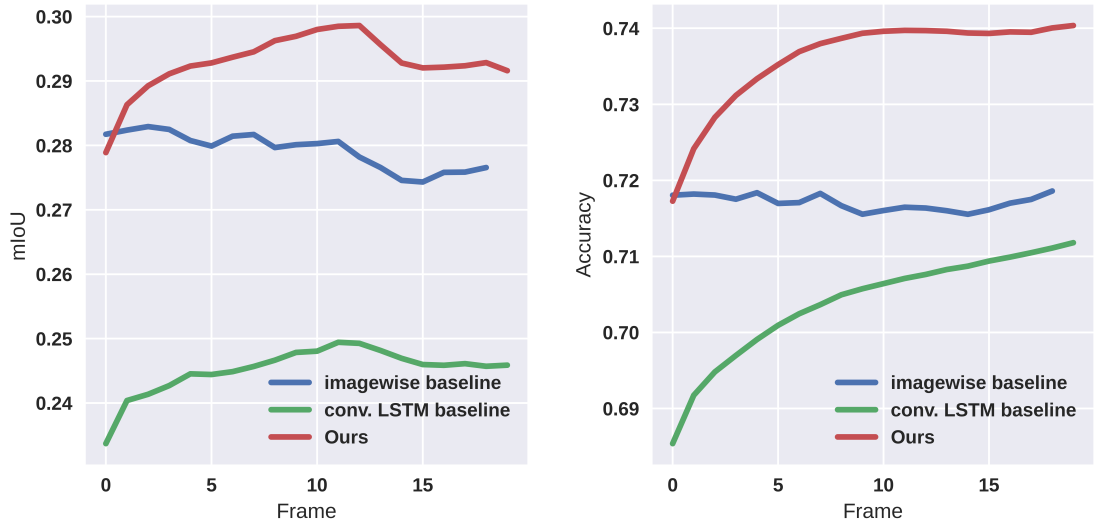
Model	**mIoU**	**accuracy**
imagewise baseline	0.2793	71.71%
conv. LSTM baseline	0.2452	70.38%
ours	**0.2928**	**73.60%**

Table 6.1: total metrics for our model and the two presented baselines

In the next step, we take a closer look at the metrics throughout the sequence. Fig. 6.8 shows the performance of our model and both baselines for each frame independently. In Subfig a we can see, that the convolutional LSTM baseline achieves a much lower mIoU than the other models. In the first frame, it has a mIoU of 0.2337, which jumps to 0.2404 in the second frame and then improves to 0.2494 in the 12th frame. After that, the mIoU slowly decreases to 0.2459. In contrast to that the imagewise baseline reaches the best mIoU of 0.2817 at the first frame and then slowly decreases to 0.2766 at the last frame. Our model starts slightly below the imagewise baseline at 0.2789, improves strongly to 0.2893 in the third frame, and then improves slowly to 0.2986 in the 13th frame. After that, it decreases slowly to 0.2916 in the last frame.

When we look at the accuracy over time in Subfig b we see a similar trend. The convolutional LSTM baseline starts the weakest with an accuracy of 68.54% and then improves first rapidly, later moderately over the whole sequence until it reaches an accuracy of 71.18% for the final frame. The imagewise baseline again starts with the best first frame at 71.80% accuracy and then stagnates around this value until it ends up with 71.86% accuracy for the last frame. Our model also starts very slightly below the imagewise baseline with an accuracy of 71.73% and then improves first quickly, after that slower until it reaches 73.93% at the 10th frame. Then our model stagnates around this value and ends with an accuracy of 74.04% for the last frame.

The results match our expectations to some extent. It seems intuitive, that the imagewise baseline achieves similar results for all frames, like in the accuracy because it treats all frames the same no matter where they are in the sequence. The convolutional LSTM baseline and our model both improve their results throughout the sequences since they can retain information to help them with the semantic segmentation of the next frame. This improvement decreases as the prediction gets better until the performance stagnates. As we see in the accuracy our model reaches this stagnation faster and at a higher value, which in our interpretation means that our model is better fit to use its retained information for the next frame than the convolutional LSTM baseline. But there are observations, which are harder to explain. For some reason, all models receive a noticeable drop in mIoU performance at the 11th or 12th frame. An explanation for the convolutional LSTM model and our model could be that those models were only trained for sequences of 10 frames and this leads to problems. But since the imagewise baseline shows a similar drop it could be suspected that there is some underlying difficulty in the data. Additionally, it could be suspected that our model stagnates to such a strong degree after the 10th frame because of this training routine.



(a) mIoU

(b) Accuracy

Figure 6.8: This image shows how our model and the two presented baselines performed on the metrics for each frame in a sequence.

Tables 6.2 and 6.3 show the mIoU and accuracy for each class for our model and the two baselines. Our model outperforms both baselines for most classes. Furthermore, we have a look at the most important class for an autonomous vehicle scenario, namely pedestrians and vehicles. The pedestrians are best perceived by our model in both the mIoU metric with 0.09471 and the accuracy with 10.39%. In comparison to that the imagewise baseline achieves a mIoU of 0.07163 and an accuracy of 7.67% and the convolutional LSTM baseline lacks behind with a mIoU of 0.00284 and an accuracy of 0.29 %. However, the performance is lower than one would wish for pedestrians since they are very important to recognize to avoid fatal crashes. The lower performance is probably because of two reasons. First pedestrians are smaller and more narrow than cars. This makes them harder to spot especially when the image is downsampled by repeated convolutions. Additionally, the Carla simulation is more designed for cars. Even on the large maps only at most 13 pedestrians were able to spawn and roam around, while some maps are inhabited by 200 vehicles. This together with the smaller appearance of pedestrians leads to a smaller number of pixels and fewer images containing pedestrians as Fig. 4.4b indicates. Additionally, pedestrians, which are in frames with poor visibility as shown in Fig. 6.7 are very hard to see because they are often on the sidewalk, where the visibility is especially bad.

Another important class is the class of vehicles. All models achieve very good results for this class. The imagewise model performs best with a mIoU of 0.8218

followed by the convolutional LSTM baseline with 0.8121 and last our model with 0.8059. For accuracy the imagewise baseline also leads with 92.11%, followed by the convolutional LSTM baseline with 91.64%, followed by our model with 91.58%. All of these results are pretty close together and promising as the vehicles are a key part of image sequences since they are the only class besides the pedestrian that moves.

Then we take a look at the classes "Traffic Sign" and "Traffic Light", which are also important for any driving scenario. Our model detects both the best with a mIoU of 0.1037 for traffic signs and 0.3908 for traffic lights and an accuracy of 11.09% for traffic signs and 45.56% for traffic lights. As with the pedestrians, the imagewise baseline performs worse with a mIoU of 0.1007 for traffic signs 0.3807 for traffic lights, and an accuracy of 10.46% for traffic signs and 43.67% for traffic lights. At last, the convolutional LSTM perceives both classes much worse with a mIoU of 0.0369 for traffic signs and 0.2441 for traffic lights and an accuracy of 3.86% for traffic signs and 27.23% for traffic lights. Traffic signs and lights both are small and therefore don't have many pixels in the data sets, which explains the overall performance of all models on the traffic signs. In comparison to that the traffic lights are perceived much better although they have a similar number of pixels in the data sets, share a similar shape and size, and appear at similar places in the images. This is probably due to the factor, that the ego-vehicle has to stop in some sequences at a traffic light, which then leads to images with big and easily spotable traffic lights, while this is much less common for traffic signs.

Some classes are perceived extremely poorly. This includes the classes "Unlabeled", "Bridge", and "Water", which all have mIoUs between 3.046e-14 and 6.239e-11 and accuracies between 4.71e-12% and 1.38e-09% for all models. We can explain this by looking at Fig. 4.4b, where we see, that all of those classes possess a very low number of pixels, which causes that single detection or misses has a huge impact on the performance. Additionally, when we look at Fig. 4.4a we see, that the classes "Unlabeled", and "Water" also have a very low number of pixels in the training data, hence the model is rarely trained to detect them. This seems intuitive for the class "Unlabeled" since in our synthetic dataset unlabeled pixels should rarely or never happen. Furthermore, water should also rarely occur in a setting specifically designed for driving. When looking at the pixel numbers for the class "Bridge" we see, that this class is common for the training data set, but rare for the validation dataset. This can be explained by the fact, that we use different maps for training and validation datasets.

Another poorly performing class is "Other" where our model receives a mIoU of 8.474e-05 and accuracy of 8.48e-03%, hence marginally outperforming the imagewise baseline with a mIoU of 4.819e-05 and accuracy of 4.82e-03% and the

convolutional LSTM baseline with a mIoU of 3.851e-06 and accuracy of 3.86e-04%. The name of the class suggests, that this is a very unspecific catch-all class for anything uncategorized. This indicates a very high intra-class variance making it hard to accurately predict this class. In addition to that, Fig. 4.4a shows, that this class is not that common in the training data, giving another reason for the poor performance. On the other hand, since everything in this class did not receive a class of its own we can assume, that they are not of any importance to the task of driving, which is why these performances don't matter.

Some classes like "Building", "Road", "Sidewalk", and "Sky" have great results for mIoU and accuracy. Our model achieves the highest mIoU for all of these classes with 0.5756 for buildings, 0.8085 for roads, 0.5395 for sidewalks, and 0.6170 for the sky. In comparison to that the imagewise baseline performs worse with a mIoU of 0.5535 for buildings, 0.7804 for roads, and 0.5997 for the sky, while performing worst of all models for sidewalks with mIoU of 0.4718. The convolutional LSTM baseline has the worst mIoU for buildings with 0.5201 and the sky with 0.5941. For roads with 0.7817, and sidewalks with 0.4862 it achieves the second-best result. When looking at the accuracy our model performs best for buildings with 68.54%, followed by the imagewise baseline with 65.83% and the convolutional LSTM baseline with 60.86%. For sidewalks and the sky, our model has the highest accuracy of 65.82% and 89.59%. The second best results for these classes are achieved by the convolutional LSTM with accuracies of 61.44% and 88.25% and the imagewise baseline is last with 58.05% and 87.43% respectively. For roads, all models have relatively similar results. The imagewise baseline achieves 92.11%, the convolutional LSTM baseline 91.64%, and our model 91.58%. All of these metrics are pretty high. This is on one hand since all of these classes have huge amounts of pixels in the dataset, which leads to better results for the mIoU. On the other hand, all of these classes have distinct places in the image, where they are likely to show up and where there is a high likelihood of them showing up, which can be incorporated into a strong bias making the prediction easier.

Class Label	imagewise baseline	conv. LSTM baseline	ours
Unlabeled	5.928e-11	**6.239e-11**	**6.239e-11**
Building	0.5535	0.5201	**0.5756**
Fence	0.08601	0.05302	**0.1110**
Other	4.819e-05	3.851e-06	**8.474e-05**
Pedestrian	0.07163	0.00284	**0.09471**
Pole	**0.1985**	0.0996	0.1822
Road Line	**0.3965**	0.3191	0.3922
Road	0.7804	0.7817	**0.8085**
Sidewalk	0.4718	0.4862	**0.5395**
Vegetation	0.2487	0.2414	**0.2686**
Vehicles	**0.8218**	0.8121	0.8059
Wall	0.04117	0.03599	**0.05165**
Traffic Sign	0.1007	0.0369	**0.1037**
Sky	0.5997	0.5941	**0.6170**
Ground	**0.008811**	0.001474	0.004389
Bridge	3.558e-14	3.046e-14	**4.157e-14**
Rail Track	**0.015205**	0.006019	0.003788
Guardrail	0.2563	0.1998	**0.3008**
Traffic Light	0.3807	0.2441	**0.3908**
Static	0.1326	0.0848	**0.1420**
Dynamic	0.01378	0.008306	**0.02383**
Water	9.401e-13	**9.416e-13**	9.405e-13
Terrain	0.1282	0.1308	**0.1478**

Table 6.2: Classwise mIoU for our model and both baselines

Class Label	imagewise baseline	conv. LSTM baseline	ours
Unlabeled	**2.64e-10%**	2.58e-10%	2.58e-10%
Building	65.83%	60.86%	**68.54%**
Fence	12.04%	7.09%	**16.99%**
Other	4.82e-03%	3.86e-04%	**8.48e-03%**
Pedestrian	7.67%	0.29%	**10.39%**
Pole	**31.97%**	14.66%	29.86%
Road Line	**43.65%**	34.55%	42.41%
Road	**92.11%**	91.64%	91.58%
Sidewalk	58.05%	61.44%	**65.82%**
Vegetation	43.50%	**47.60%**	44.15%
Vehicles	90.91%	90.01%	**92.77%**
Wall	9.29%	8.12%	**9.72%**
Traffic Sign	10.46%	3.86%	**11.09%**
Sky	87.43%	88.25%	**89.59%**
Ground	**1.05%**	0.17%	0.49%
Bridge	**1.38e-09%**	1.30e-09%	1.30e-09%
Rail Track	**2.92%**	1.23%	0.83%
Guardrail	33.70%	27.84%	**43.70%**
Traffic Light	43.67%	27.23%	**45.56%**
Static	17.11%	10.80%	**17.38%**
Dynamic	1.47%	0.87%	**2.53%**
Water	**4.96e-12%**	4.71e-12%	4.71e-12%
Terrain	**37.35%**	33.33%	35.25%

Table 6.3: Classwise accuracy for our model and both baselines

7. Conclusion

In this thesis, we seek to modernize the Functionally Modularized Representation Filter by Wagner et al. 2018 with a residual optical flow predictor to grasp object motion to address challenging autonomous driving scenarios. Thus, we create the Recurrent Structured Filter, which has an encoder-decoder-based structure and contains an ego-motion filter to predict the camera transformation, a feature filter to predict the depth and warp the current hidden state accordingly to the predicted depth and camera transformation, and an object motion filter to predict the residual optical flow. Our model is suited to deal with image sequences containing a moving camera and moving objects.

The advantage of our model is that it has a recurrent hidden state to retain information from past frames. This can be used to integrate temporal coherence into the model, but it additionally provides the model with means to produce adequate predictions even if single images are missing. Additionally, the model produces humanly interpretable, intermediate results, specifically the depth maps, the camera transformation, and the residual optical flow. This helps us to understand the inner workings of the model, which is especially important if errors occur. But even when errors don't occur these intermediate results offer the advantage of making the model robust for example to faulty sensors or intentional image compromising. That is the case because we know that the intermediate results are important to generate the output and accordingly can our model still produce adequate predictions as long as the intermediate results aren't faulty as well.

Furthermore, we generate a synthetic dataset for autonomous driving using the CARLA simulator(Dosovitskiy et al. 2017). Our dataset contains RGB images, depth maps, semantic segmentation labels, and optical flow maps. We generate 12000 sequences for our dataset using 8 different maps. This allows us to train our model on a much more challenging dataset than what the architecture based on the functionally modularized representation filter was trained on until now(Wagner et al. 2018).

Our model outperforms both of our baselines with a total mIoU of 0.2928 and an accuracy of 73.60%, while having interpretable representations.

Future work could integrate the residual optical flow into another warping step for the high-dimensional hidden state to enable the model to account for object movement. Additionally, our model could be trained and evaluated on more popular datasets to compare it better with other models. Alternatively, it could be trained and evaluated on a real-world dataset to see how well our method transfers to real-world scenarios. For both cases, there would need to be a way to deal with missing ground truth data. One possibility to handle this problem would be to replace the missing ground truth data with pseudo labels generated from a powerful foundation model like "Segment Anything"(Kirillov et al. 2023). Another possibility would be to learn the required representations in a self-supervised manner like SfM-Net(Vijayanarasimhan et al. 2017).

LSTM Long Short-Term Memory

GRU Gated Recurrent Unit

CRF Conditional Random Field

MRF Markov Random Field

mIoU mean Intersection over Union

CE Cross-Entropy

A. Data Samples

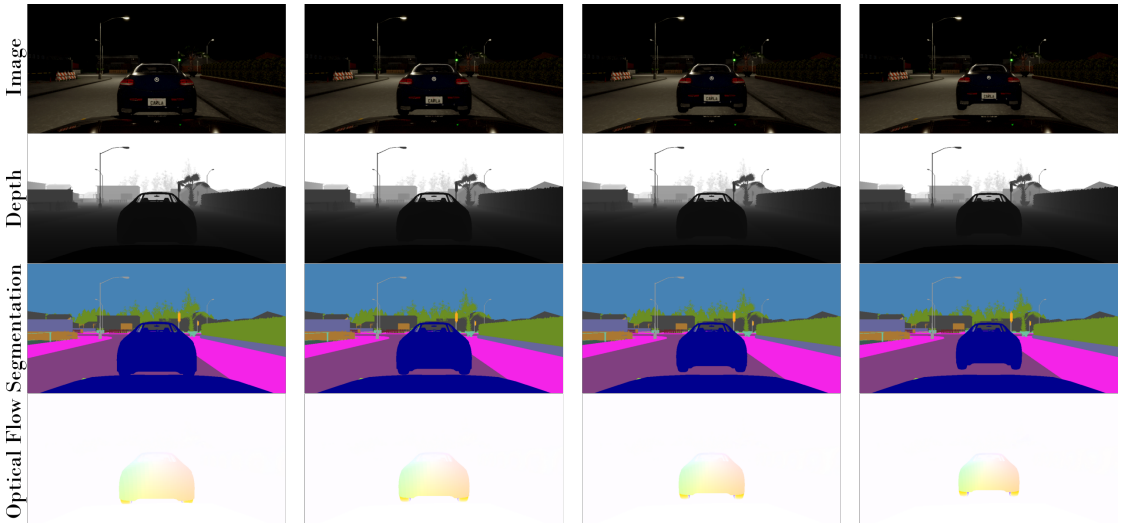


Figure A.1: RGB images, depth maps, semantic segmentation labels and residual optical flow maps

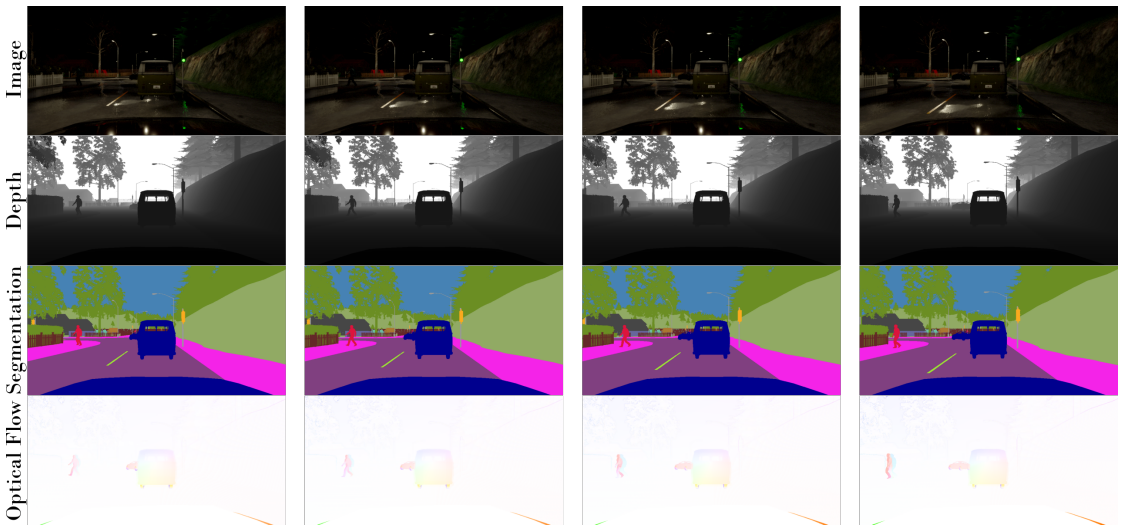


Figure A.2: RGB images, depth maps, semantic segmentation labels and residual optical flow maps

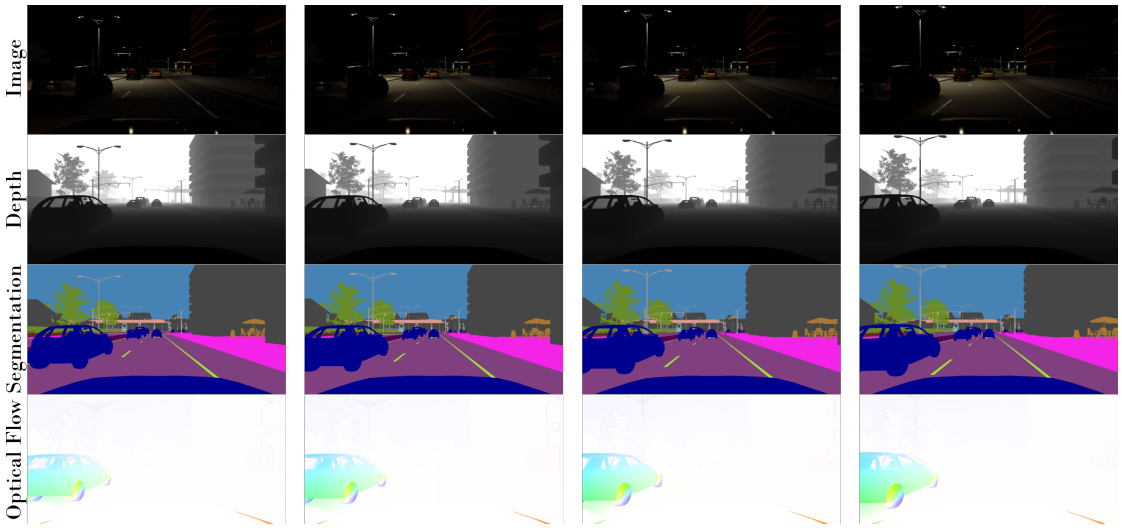


Figure A.3: RGB images, depth maps, semantic segmentation labels and residual optical flow maps

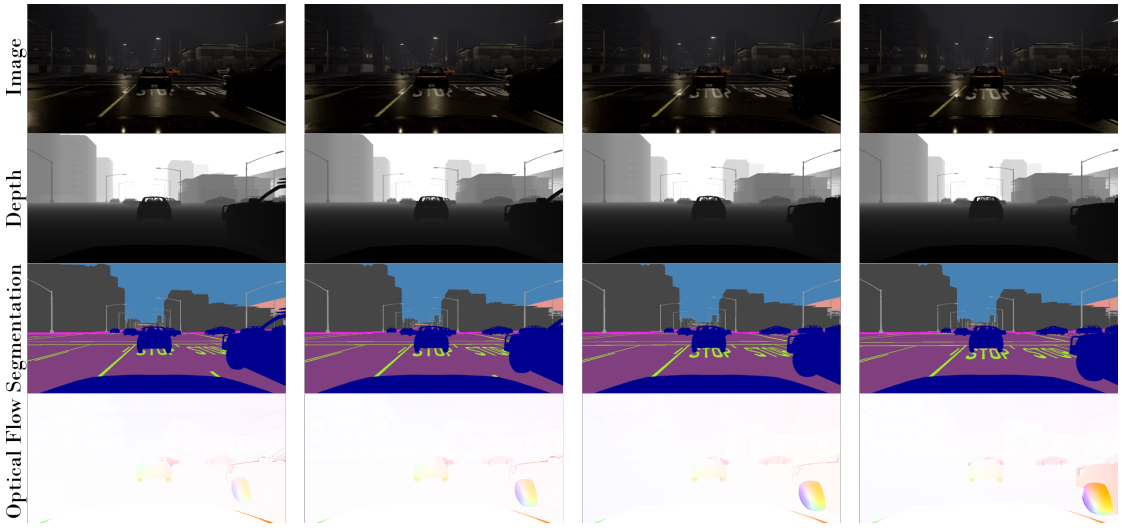


Figure A.4: RGB images, depth maps, semantic segmentation labels and residual optical flow maps

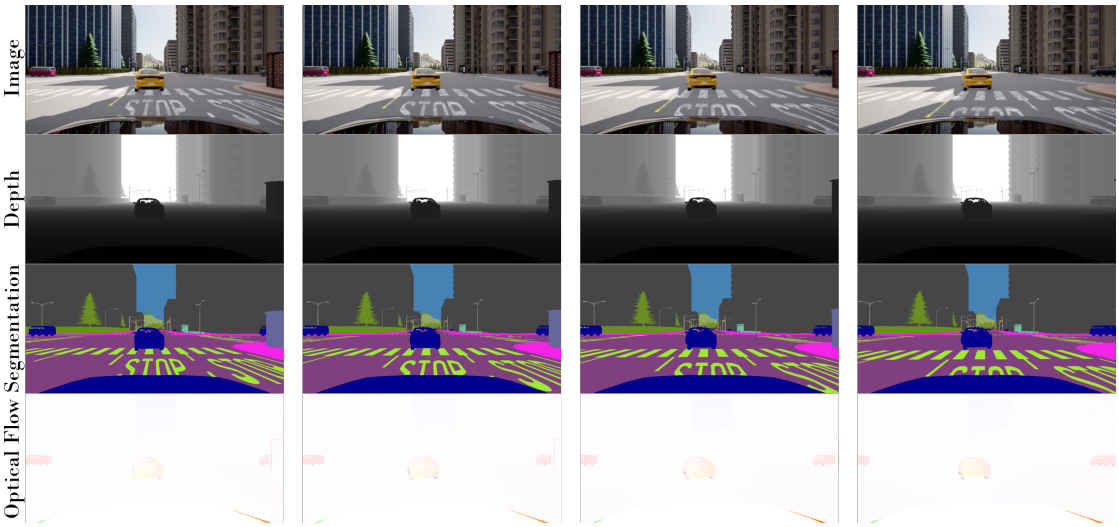


Figure A.5: RGB images, depth maps, semantic segmentation labels and residual optical flow maps

B. Qualitative Results

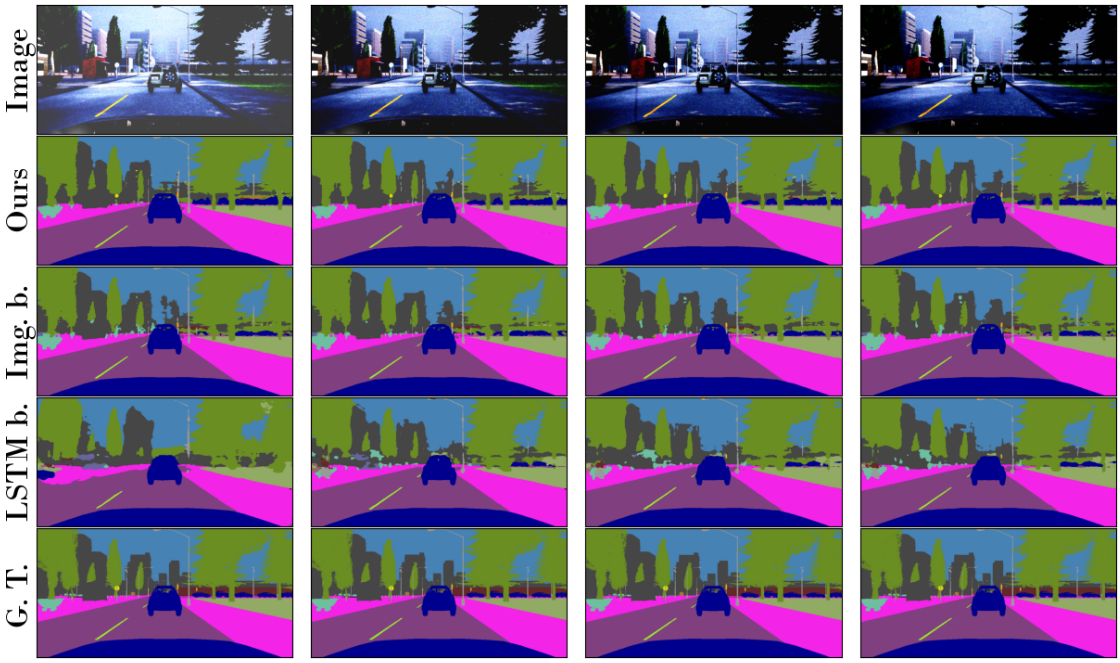


Figure B.1: RGB images, semantic segmentation predictions for our model and both baselines and ground truth semantic segmentation labels

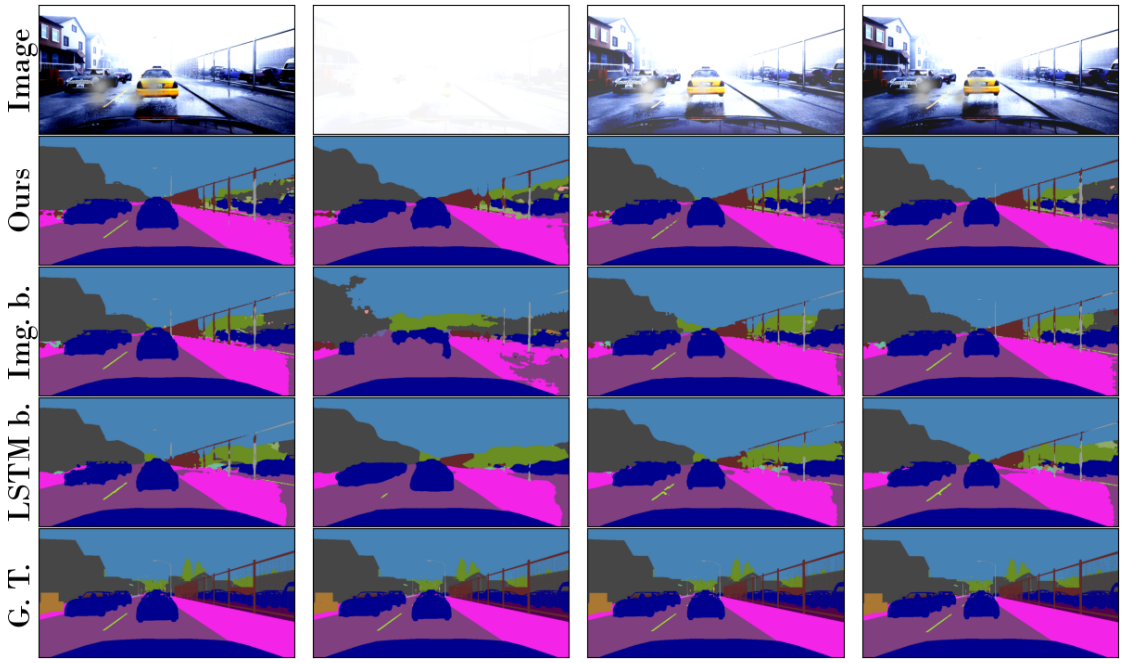


Figure B.2: RGB images, semantic segmentation predictions for our model and both baselines and ground truth semantic segmentation labels

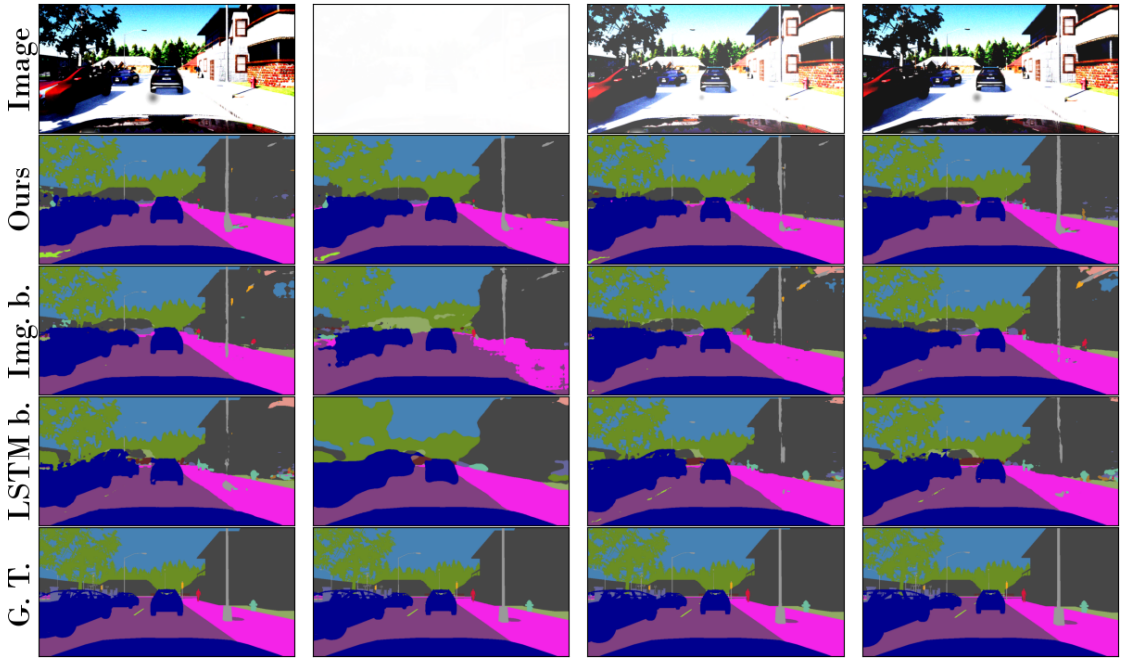


Figure B.3: RGB images, semantic segmentation predictions for our model and both baselines and ground truth semantic segmentation labels

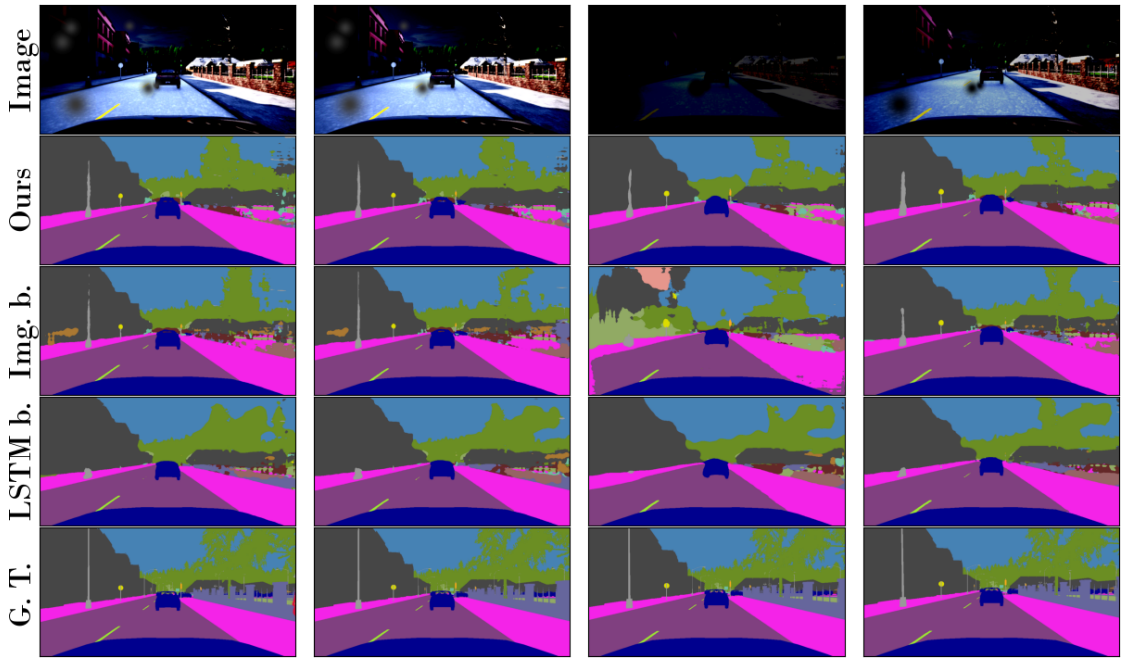


Figure B.4: RGB images, semantic segmentation predictions for our model and both baselines and ground truth semantic segmentation labels

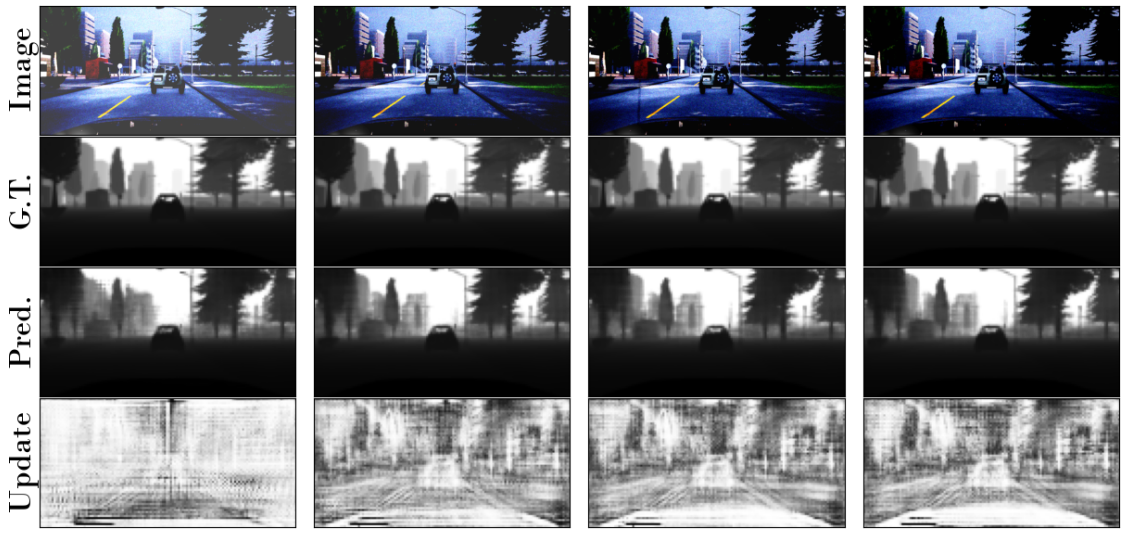


Figure B.5: RGB images, predicted depth maps and weight matrix of the update gate f^s_{upd} from our model and ground truth depth maps

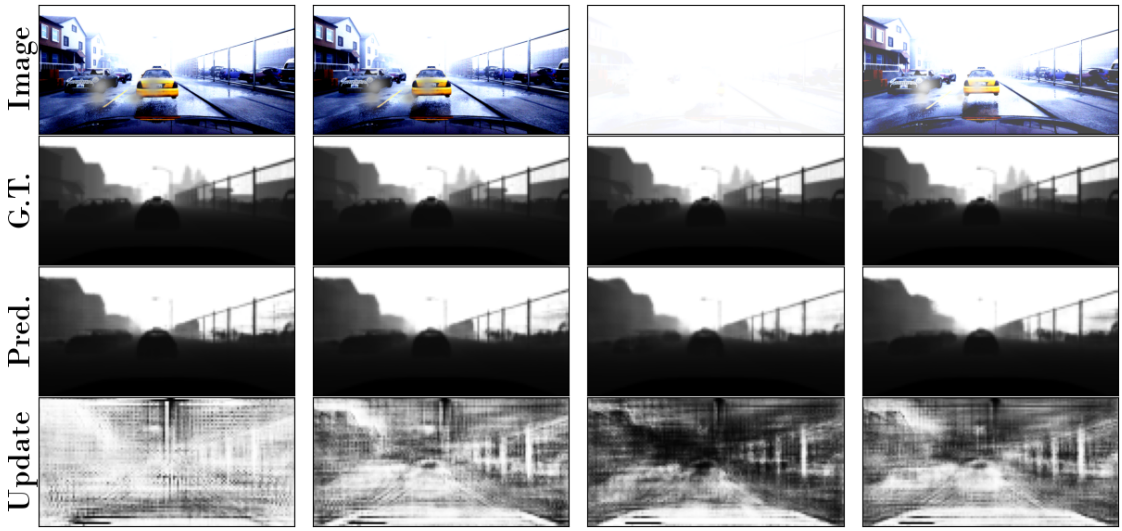


Figure B.6: RGB images, predicted depth maps and weight matrix of the update gate f^s_{upd} from our model and ground truth depth maps

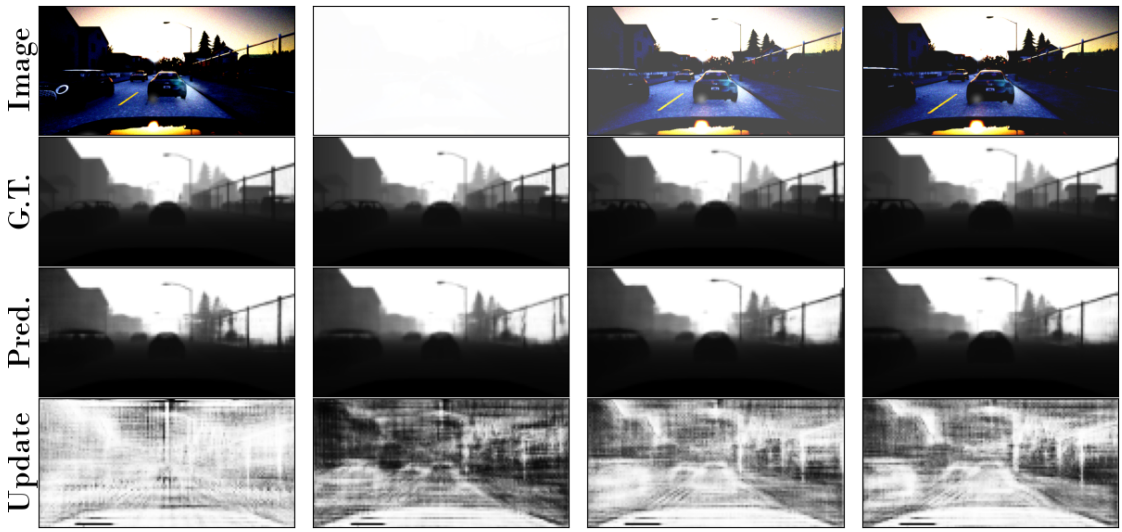


Figure B.7: RGB images, predicted depth maps and weight matrix of the update gate f^s_{upd} from our model and ground truth depth maps

List of Figures

List of Tables

Bibliography

Badrinarayanan, Vijay, Alex Kendall, and Roberto Cipolla (Dec. 2017). "SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation." In: *Ieee transactions on pattern analysis and machine intelligence* 39.12, pp. 2481–2495. ISSN: 1939-3539. URL: https://ieeexplore.ieee.org/abstract/document/7803544 (visited on 03/16/2024).

Chandra, Siddhartha, Camille Couprie, and Iasonas Kokkinos (2018). "Deep Spatio-Temporal Random Fields for Efficient Video Segmentation." In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 8915–8924. URL: https://openaccess.thecvf.com/content_cvpr_2018/html/Chandra_Deep_Spatio-Temporal_Random_CVPR_2018_paper.html (visited on 03/08/2024).

Chen, Liang-Chieh, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille (June 7, 2016). *Semantic Image Segmentation with Deep Convolutional Nets and Fully Connected CRFs*. arXiv: 1412.7062 [cs]. URL: http://arxiv.org/abs/1412.7062 (visited on 03/16/2024). preprint.

Chen, Liang-Chieh, George Papandreou, Florian Schroff, and Hartwig Adam (Dec. 5, 2017). *Rethinking Atrous Convolution for Semantic Image Segmentation*. arXiv: 1706.05587 [cs]. URL: http://arxiv.org/abs/1706.05587 (visited on 10/10/2022). preprint.

Chen, Liang-Chieh, Yi Yang, Jiang Wang, Wei Xu, and Alan L. Yuille (2016). "Attention to Scale: Scale-Aware Semantic Image Segmentation." In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 3640–3649. URL: https://openaccess.thecvf.com/content_cvpr_2016/html/Chen_Attention_to_Scale_CVPR_2016_paper.html (visited on 03/16/2024).

Chen, Liang-Chieh, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam (Aug. 22, 2018). *Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation*. arXiv: 1802.02611 [cs]. URL: http://arxiv.org/abs/1802.02611 (visited on 02/29/2024). preprint.

Deng, Jia, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei (June 2009). "ImageNet: A large-scale hierarchical image database." In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009 IEEE Conference on Computer Vision and Pattern Recognition, pp. 248–255. URL: https://ieeexplore.ieee.org/document/5206848 (visited on 03/19/2024).

Dosovitskiy, Alexey, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun (Nov. 10, 2017). *CARLA: An Open Urban Driving Simulator*. arXiv:

1711.03938 [cs]. URL: http://arxiv.org/abs/1711.03938 (visited on 03/19/2024). preprint.

Gadde, Raghudeep, Varun Jampani, and Peter V. Gehler (2017). "Semantic Video CNNs Through Representation Warping." In: Proceedings of the IEEE International Conference on Computer Vision, pp. 4453–4462. URL: https://openaccess.thecvf.com/content_iccv_2017/html/Gadde_Semantic_Video_CNNs_ICCV_2017_paper.html (visited on 03/08/2024).

He, Junjun, Zhongying Deng, and Yu Qiao (2019). "Dynamic Multi-Scale Filters for Semantic Segmentation." In: Proceedings of the IEEE/CVF International Conference on Computer Vision, pp. 3562–3572. URL: https://openaccess.thecvf.com/content_ICCV_2019/html/He_Dynamic_Multi-Scale_Filters_for_Semantic_Segmentation_ICCV_2019_paper.html (visited on 03/16/2024).

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (Dec. 10, 2015). *Deep Residual Learning for Image Recognition*. arXiv: 1512.03385 [cs]. URL: http://arxiv.org/abs/1512.03385 (visited on 03/17/2024). preprint.

Hochreiter, Sepp and Jürgen Schmidhuber (Nov. 1997). "Long Short-Term Memory." In: *Neural computation* 9.8, pp. 1735–1780. ISSN: 0899-7667. URL: https://ieeexplore.ieee.org/abstract/document/6795963 (visited on 03/07/2024).

Huston, Stephen J. and Holger G. Krapp (July 22, 2008). "Visuomotor Transformation in the Fly Gaze Stabilization System." In: *Plos biology* 6.7, e173. ISSN: 1545-7885. URL: https://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.0060173 (visited on 03/16/2024).

Jain, Samvit, Xin Wang, and Joseph E. Gonzalez (2019). "Accel: A Corrective Fusion Network for Efficient Semantic Segmentation on Video." In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 8866–8875. URL: https://openaccess.thecvf.com/content_CVPR_2019/html/Jain_Accel_A_Corrective_Fusion_Network_for_Efficient_Semantic_Segmentation_on_CVPR_2019_paper.html (visited on 03/08/2024).

Jin, Xiaojie, Xin Li, Huaxin Xiao, Xiaohui Shen, Zhe Lin, Jimei Yang, Yunpeng Chen, Jian Dong, Luoqi Liu, Zequn Jie, Jiashi Feng, and Shuicheng Yan (2017). "Video Scene Parsing With Predictive Feature Learning." In: Proceedings of the IEEE International Conference on Computer Vision, pp. 5580–5588. URL: https://openaccess.thecvf.com/content_iccv_2017/html/Jin_Video_Scene_Parsing_ICCV_2017_paper.html (visited on 03/08/2024).

Kirillov, Alexander, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C. Berg, Wan-Yen Lo, Piotr Dollar, and Ross Girshick (2023). "Segment Anything." In: Proceedings of the IEEE/CVF International Conference on Computer Vision, pp. 4015–4026. URL: https://openaccess.thecvf.com/content/ICCV2023/html/Kirillov_Segment_Anything_ICCV_2023_paper.html (visited on 03/09/2024).

Lipton, Zachary C., John Berkowitz, and Charles Elkan (Oct. 17, 2015). *A Critical Review of Recurrent Neural Networks for Sequence Learning*. arXiv: 1506.00019

[cs]. URL: http://arxiv.org/abs/1506.00019 (visited on 03/05/2024). preprint.

Long, Jonathan, Evan Shelhamer, and Trevor Darrell (2015). "Fully Convolutional Networks for Semantic Segmentation." In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 3431–3440. URL: https://openaccess.thecvf.com/content_cvpr_2015/html/Long_Fully_Convolutional_Networks_2015_CVPR_paper.html (visited on 03/16/2024).

Luc, Pauline, Camille Couprie, Soumith Chintala, and Jakob Verbeek (Nov. 25, 2016). *Semantic Segmentation using Adversarial Networks*. arXiv: 1611.08408 [cs]. URL: http://arxiv.org/abs/1611.08408 (visited on 03/16/2024). preprint.

Minaee, Shervin, Yuri Boykov, Fatih Porikli, Antonio Plaza, Nasser Kehtarnavaz, and Demetri Terzopoulos (July 2022). "Image Segmentation Using Deep Learning: A Survey." In: *Ieee transactions on pattern analysis and machine intelligence* 44.7, pp. 3523–3542. ISSN: 1939-3539.

Ronneberger, Olaf, Philipp Fischer, and Thomas Brox (2015). "U-Net: Convolutional Networks for Biomedical Image Segmentation." In: *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*. Ed. by Nassir Navab, Joachim Hornegger, William M. Wells, and Alejandro F. Frangi. Lecture Notes in Computer Science. Cham: Springer International Publishing, pp. 234–241. ISBN: 978-3-319-24574-4.

Teed, Zachary and Jia Deng (Aug. 25, 2020). *RAFT: Recurrent All-Pairs Field Transforms for Optical Flow*. arXiv: 2003.12039 [cs]. URL: http://arxiv.org/abs/2003.12039 (visited on 03/17/2024). preprint.

Vijayanarasimhan, Sudheendra, Susanna Ricco, Cordelia Schmid, Rahul Sukthankar, and Katerina Fragkiadaki (Apr. 25, 2017). *SfM-Net: Learning of Structure and Motion from Video*. arXiv: 1704.07804 [cs]. URL: http://arxiv.org/abs/1704.07804 (visited on 03/15/2024). preprint.

Visin, Francesco, Marco Ciccone, Adriana Romero, Kyle Kastner, Kyunghyun Cho, Yoshua Bengio, Matteo Matteucci, and Aaron Courville (2016). "ReSeg: A Recurrent Neural Network-Based Model for Semantic Segmentation." In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops, pp. 41–48. URL: https://www.cv-foundation.org/openaccess/content_cvpr_2016_workshops/w12/html/Visin_ReSeg_A_Recurrent_CVPR_2016_paper.html (visited on 03/16/2024).

Wagner, Jörg, Volker Fischer, Michael Herman, and Sven Behnke (Oct. 15, 2018). *Functionally Modular and Interpretable Temporal Filtering for Robust Segmentation*. arXiv: 1810.03867 [cs]. URL: http://arxiv.org/abs/1810.03867 (visited on 03/04/2024). preprint.

Wang, Wenguan, Tianfei Zhou, Fatih Porikli, David Crandall, and Luc Van Gool (Dec. 30, 2021). *A Survey on Deep Learning Technique for Video Segmentation*. arXiv: 2107.01153 [cs]. URL: http://arxiv.org/abs/2107.01153 (visited on 10/10/2022). preprint.

Zhao, Rui, Dongzhe Wang, Ruqiang Yan, Kezhi Mao, Fei Shen, and Jinjiang Wang (Feb. 2018). "Machine Health Monitoring Using Local Feature-Based Gated Recurrent Unit Networks." In: *Ieee transactions on industrial electronics* 65.2, pp. 1539–1548. ISSN: 1557-9948. URL: https://ieeexplore.ieee.org/abstract/document/7997605 (visited on 03/16/2024).

Zhu, Xizhou, Yuwen Xiong, Jifeng Dai, Lu Yuan, and Yichen Wei (2017). "Deep Feature Flow for Video Recognition." In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2349–2358. URL: https://openaccess.thecvf.com/content_cvpr_2017/html/Zhu_Deep_Feature_Flow_CVPR_2017_paper.html (visited on 03/08/2024).