Better Mixed Reality Through Depth Estimation on Machine Learning

Simen Røe Fjøsne

Thesis submitted for the degree of
Master in
Informatics: Programming and System Architecture
60 credits

Department of Informatics
Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

Spring 2022

Better Mixed Reality Through Depth Estimation on Machine Learning

Simen Røe Fjøsne

Abstract

Computer depth estimation is an ever-increasing research field in computer vision because of its use in a variety of different applications: from autonomous cars that rely on depth to avoid objects, to AR devices that rely on depth to display virtual objects accurately in the real world. In this thesis, we will explore stereo depth estimation, a way of estimating depth using a camera pair.

Today, stereo depth is estimated in two ways: either by block-matching or with machine learning. This thesis aims to assess whether a combination of the two approaches is possible, and if it is possible, does it provide any benefits over existing methods. We design a pipeline that combines a well-established block-matching algorithm and a state-of-the-art stereo depth estimation model to address these questions.

The results of our thesis suggest that a combination of block-matching and machine learning may work with a bit more experimenting, however, more work needs doing before we can conclude if our approach provides any benefits versus existing approaches.

Contents

List of Figures

List of Tables

Preface

First of all, I want to thank my supervisor Carsten Griwodz for his continued guidance and support throughout the creation of my thesis. I've really appreciated and learned a lot from our weekly meetings.

I also want to thank the University Centre for Information Technology at the University Of Oslo for letting me use their machine learning cluster and Simula Research Laboratory for letting me use their eX^3 cluster. Without access to these resources, this thesis might not have been possible.

Lastly, I want to thank my family for their continued support and words of encouragement throughout my entire course of study.

Chapter 1

Introduction

Computer vision is one of the fastest-growing fields in computer science due to an increasingly more technological and autonomous world. Everything from the cameras in our phones, to modern cars, uses some sort of computer vision software to either track or detect movement, identify objects, etc. Because of this, the need for efficient computer vision software has drastically increased. In this thesis we will take a closer look at the importance of computer vision, more specifically we will look at the importance of depth estimation in computer vision. We will carry out this thesis with mixed reality and augmented reality in mind and will explore the need for accurate depth perception from this perspective.

Mixed reality (MR) is the merging of virtual elements into the real world in a way that feels natural for the user. Today it is commonly used in movie making where special effects transform sets into big open worlds, also referred to as "movie CGI". Another application is in Augmented Reality (AR) where virtual elements are placed into the user's field of view through an AR display on devices like smartphones or head-mounted displays (HMD) like the one in figure 1.1. HMDs, unlike smartphones, are worn on the user's head, freeing the user's hands for interaction with the virtual objects. This is useful for simulation and training purposes where interaction between the user and virtual objects may be necessary. However, this entails that the head-mounted display is accurate enough to distinguish between physical objects and virtual objects scattered around in the same scene, which unfortunately isn't the case with today's AR technology.

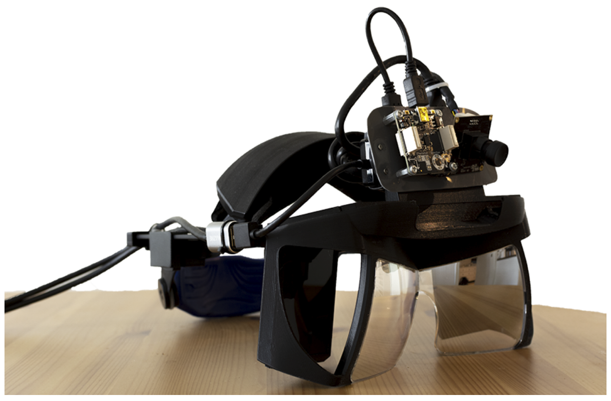


Figure 1.1: Project North Star head-mounted AR display

The AR devices of today (both smartphones and HMDs) are capable of displaying virtual objects in approximately their correct positions in the physical world, even when the devices are moved around (although it is still far from perfect). Their main shortcoming is the ability to represent the dimension of depth, that is, determining if a virtual object should be obscured by a physical object in the real world. This is one of the main problems preventing us from utilizing the full potential of AR as a method for creating realistic models, for instance, used for training purposes (e.g., training simulators) or where 3D visualization is essential (e.g., the correct model of human anatomy). The main objective of this thesis is to explore how this problem can be solved by trying to build upon current state-of-the-art computer depth estimation solutions.

More specifically, we will explore stereo depth estimation solutions using machine learning. Stereo depth estimation is a method where a computer uses two front-facing cameras shifted horizontally apart from each other, to estimate depth. This is useful on head-mounted AR displays as the cameras can be mounted on top of them like shown in figure 1.1, and give the display accurate real-time depth information. The idea behind using a stereo camera setup to estimate depth comes from the human ability to estimate depth using our eyes. With a pair of front-facing eyes, we can pick up on subtle differences, or the *disparity* in the positions of objects between the eyes' point-of-view. This difference is interpreted by our brain as depth. This is why we can lose our sense of depth when covering one of our eyes. The bigger the disparity is between the left and right eye, the closer the object is. This is easy to verify by holding out a finger in front of our face, and quickly opening and closing the left and right eyes in an alternating fashion. When moving the finger further away, we can see less disparity between the left and right eye.

Today there are numerous ways of approaching stereo depth estimation in computer vision. The traditional approach is to use algorithms based on single-pixel matching or block-matching (larger areas of pixels, i.e., blocks of pixels) between left and right image pairs, but in recent years, there have been breakthroughs in the machine learning field allowing for ML models to learn and predict accurate depth with the use of one (monocular depth), two (stereo depth) or even more cameras (multi-view depth). In this thesis, we will combine the traditional approach with the newer machine learning approach to find out if there is any benefit to combining the two. In the next section, we outline our research questions for our thesis, along with the research method we employ to answer them.

1.1 Research Question

We have the following questions we want to answer:

1. Does block-matching help in any way during the training of a stereo depth estimation model?
2. Will block-matching improve the results of a model vs. a model without the use of block-matching?

We believe that using block-matching to extract depth information from stereo images as a preliminary step to training a neural network, gives the network enough information about the scene to first of all reduce training time, while secondly improving the accuracy of the network. In other words, we want to test if block-matching has any positive impact on a stereo depth estimation neural network.

In this thesis, we mainly focus on implementing a neural network that estimates depth from a combination of stereo images and disparity maps computed by a block-matching algorithm. There are plenty of ways of estimating depth, for example by using single images (called monocular depth estimation), or multiple images from different angles (called multi-view depth estimation). Both of these methods fall outside the scope of this thesis because we are only interested in stereo depth estimation. There are also numerous stereo depth estimation algorithms that do not use machine learning, but these also falls outside this thesis' scope.

1.2 Research Method

Due to the scope of computer science, there are disagreements among computer scientists on if computer science should be a part of mathematics, engineering, or natural sciences. This disagreement has led to different paradigms with different practices. Eden (2007), introduces three paradigms: "the rationalist paradigm", "the technocratic paradigm" and "the scientific paradigm". Each of them has its roots in mathematics, engineering, and natural sciences respectively. Alternatively, Comer et al. (1989) proposes three different paradigms, but also these are rooted in the same three categories. In our thesis, we choose between the research paradigms proposed in Comer et al. (1989). They define each paradigm as follows [Comer et al., 1989, p. 10]:

Theory

The theory paradigm is rooted in mathematics and consists of four steps:

1. Characterize objects of study (define)
2. Hypothesize possible relationships among them (theorem)
3. Determine whether the relationships are true (proof)
4. Interpret results

Abstraction

The abstraction paradigm is rooted in the experimental scientific method and consists of four stages:

1. Form a hypothesis
2. Construct a model and make a prediction
3. Design an experiment and collect data
4. Analyze results

Design

The final paradigm, design, is rooted in engineering and does also consist of four steps:

1. State requirements
2. State specifications
3. Design and implement the system
4. Test the system

In our thesis, we plan on using the design paradigm. Our thesis involves machine learning on the topic of computer vision and with this comes the implementation and testing of a machine learning model where we have clear-cut requirements and specifications. With this in mind, the design paradigm seems the most convenient and straightforward paradigm to use out of the three listed above. We could also argue for using the abstraction paradigm, as a neural network is in all essence a model. We use "model" and "neural network" interchangeably throughout this thesis. Machine learning is also experimental by nature as there are many parameters that can be tweaked to completely change the results of a model, making the need to experiment a fundamental and crucial part. With that being said, we stick to the design paradigm as it is the most familiar and easy to adhere to out of the three paradigms listed above.

1.3 Thesis Outline

This thesis consists of six chapters, starting with the introduction. In the current chapter, we've presented the motivation and scope for the thesis, as well as our research question and paradigm. This chapter is followed by a background section where we lay the foundation for the rest of the thesis by explaining the core concepts and technologies we will use, along with related work done on the topic of stereo depth estimation. In the third chapter, we present our design in detail, explaining each part we plan to implement. The design is followed by the implementation where we go into detail about how we implemented the design. In the evaluation chapter, we present our results and discuss them. We will end the thesis with a conclusion where we summarize what we did, and how it went, as well as propose future work related to this thesis.

Chapter 2

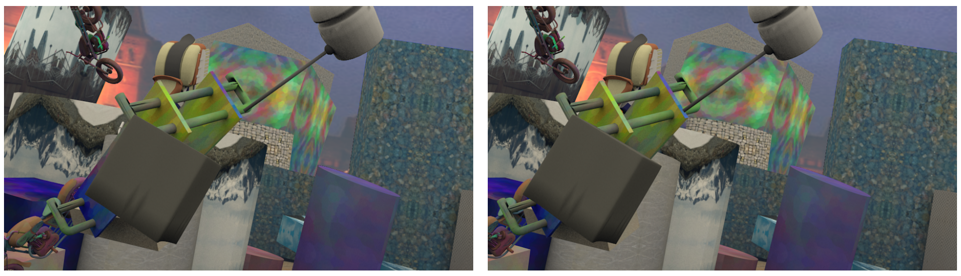
Background

Computer vision (CV) is a field in computing where computers extract visual information from cameras or similar instruments to get an understanding of their surroundings. Naturally, computer vision has many use-cases, from biometric face recognition software in a phone, to CGI in movies, and self-driving (autonomous) cars. Especially autonomous cars rely on accurate and fast algorithms to prevent fatal accidents.

In the following sections, we will go through the prerequisite knowledge needed for later parts of this thesis. In the first part, we explain the basics of stereo depth estimation with the use of stereo matching. The second part introduces core machine learning concepts, such as training and validation, model generalization, and convolutional neural networks. Lastly, we point to related work done on stereo depth estimation.

2.1 Block-Matching

Throughout the introduction of this thesis, we have stated that we will use block-matching in combination with machine learning to estimate depth in stereo images. In this section, we shall explain block-matching, or the more general case of *stereo matching*. A stereo matching algorithm matches similar pixels, or blocks of pixels, between two images. In our case, we match blocks of pixels between a left-right image pair. The left and right images are displaced horizontally, causing the pixels to be displaced between them simultaneously. Figure 2.1 shows the left and right image

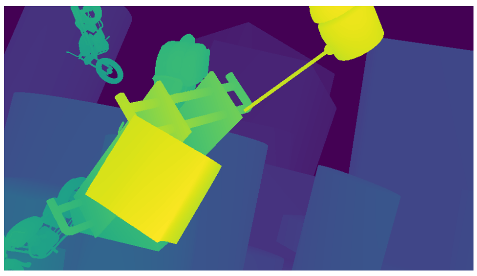


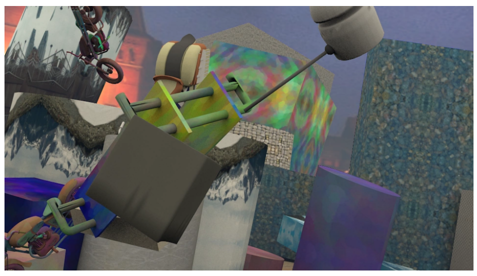
(a) Left image (b) Right image

Figure 2.1: A left and right image sample from the SceneFlow dataset [Mayer et al., 2016].

of a data sample from the SceneFlow dataset [Mayer et al., 2016]. Notice how the objects in the scene have shifted slightly between the left and right image. This displacement in pixels is called the *disparity*. With the disparity, we can calculate a real depth value for a point using *epipolar geometry*, which is further explained in section 2.2. A stereo matching algorithm either computes the disparity for all pixels, or the disparity for a subset of pixels. The former is called *dense* stereo matching, while the latter is called *sparse* stereo matching. In this thesis, we will only focus on dense stereo matching, while sparse stereo matching falls outside this thesis' scope.

Dense stereo matching algorithms choose one image as the *reference image*. For every pixel in the reference image, the algorithm searches for the corresponding pixel in the other image. This also the case for blocks of pixels. This search is done along the *scan lines* which specifies in what direction the algorithm will search, or scan, for matching pixels. The scan lines can go in all four cardinal directions as well as diagonally. In stereo matching algorithms, these scan lines usually go in the horizontal direction, as they assume that any pixel can only be shifted in that direction. Due to this assumption, most stereo matching algorithms require all left-right image pairs to be *coplanar*, i.e., the images are only displaced horizontally. This is either done by calibrating both cameras to be coplanar, or through software by transforming each pixel in both images onto the same image plane. This is called rectification, something we will not go into detail on in this thesis. The disparity of each pixel is stored in a *disparity map*. The disparity map is a map with the same height and width as the reference image and contains the disparity of each pixel. Because the disparity map is represented like this, we can visualize disparity maps as colored


(a) Disparity map


(b) Reference image (left image)

(c) Viridis colormap

Figure 2.2: A ground-truth disparity map (a) from the SceneFlow FlyingThings3D dataset [Mayer et al., 2016] and its reference image (b). The colors of the disparity map correlates to different disparities and are based on the color map (c) from OpenCV.

images. Figure 2.2 illustrates a disparity map, where each pixel contains the disparity value of each pixel in the reference image. The pixel brightness corresponds to how far a pixel has shifted between the left and right image. A brighter pixel corresponds to a bigger disparity value, and thus a bigger shift between the left and right image.

Scharstein and Szeliski (2002) has proposed a taxonomy of stereo matching algorithms based on similarities observed in most algorithms. In the paper, they detail four steps that stereo matching algorithms perform [Scharstein and Szeliski, 2002, p. 10]:

1. Matching cost computation
2. Cost aggregation
3. Disparity computation/optimization
4. Disparity refinement

Depending on the algorithm, these steps are not always done in this specific order, and some of the steps may not be needed at all. Matching cost computation is based on a *matching criterion*. This criterion is a function that computes the *cost*, or the similarity, between pixels. The higher the matching cost is, the less similar the pixels are. One of the most common matching criteria for block-matching algorithms is the *sum of absolute differences* (SAD) between blocks of pixels. It computes the cost between two blocks of pixels by calculating the absolute value between each corresponding pixel in both blocks, and adding them together:

$$\mathrm{SAD}(v^L_{x,y}, v^R_{x,y}) = \Sigma_{x,y} \left| v^L_{x,y} - v^R_{x,y} \right|_1$$

Here, $v^L_{x,y}$ and $v^R_{x,y}$ are the values of the pixels (or pixel intensities) at the same position (x, y) in the left and right blocks. Blocks are usually matched within a certain *disparity range*. The disparity range determines the number of pixels in the search direction it looks for a matching block. The range is usually between 0 and a maximum disparity D. When the costs of all blocks within the disparity range have been computed, the costs form a *disparity space image* (DSI) [Scharstein and Szeliski, 2002, p. 9]. The DSI is the representation of all matching costs for all pixels within the disparity range. In this thesis, we refer to the DSI as the *cost volume*, because it represents a volume (three dimensions) of cost values at all pixels (x, y) for all disparities d in the disparity range $d \in [0, D]$. We can think of the DSI, or cost volume, as a three-dimensional matrix with size (D, H, W), where D is the maximum disparity and H and W are the height and width of the reference image.

Depending on the type of algorithm, the next step is the aggregation of costs over the cost volume. Cost aggregation usually only happens in *local* stereo matching algorithms. A local algorithm is different from a *global* algorithm in the sense that local algorithms bases themselves on local *windows*, small areas of the cost volume. Global algorithms don't usually perform cost aggregation, as they instead rely on minimizing a global cost function to compute disparities [Scharstein and Szeliski, 2002, p. 10]. This makes the global stereo matching algorithms more computationally expensive than local algorithms, although more accurate. Local algorithms are usually easier to design and implement as well. The simplest form of cost aggregation is either by summing or averaging the matching costs over local windows in the cost volume. These windows may be two-dimensional, i.e., small regions of size N×M in the height and width dimension at a fixed disparity d, or three-dimensional by extending the N×M window into the disparity dimension C×N×M [Scharstein and Szeliski, 2002, p. 11]. There are also more complex and robust ways of aggregating the matching costs, but we will not go into further details on these methods.

Disparity computation is the step of computing the disparity values based on the aggregated costs (for local algorithms), or by minimizing a cost function (for global algorithms). As explained in Scharstein and Szeliski (2002), disparity computation for local stereo matching algorithms is trivial. We perform a winner-takes-all optimization over the disparity space and choose the disparities that lead to the lowest costs. This can be done by applying an `argmin` function on the matching cost volume over the disparity dimension. The function returns the indices in the disparity dimension that has the lowest cost. In global stereo matching algorithms, the objective is to minimize a global cost function over the cost volume to estimate a disparity for each pixel. We will not go into detail on this and instead refer to Scharstein and Szeliski (2002) for further reading. The final step of a stereo matching algorithm is to refine the disparities estimated in the previous step. Disparity refinement is performed to "clean up" the estimated disparity map by removing wrongful disparity estimates, for example disparities in occluded regions (this is explained in section 2.3). It is also a way to estimate sub-pixel disparities, as the disparity computation step usually only computes the integer disparities for each pixel [Scharstein and Szeliski, 2002, p. 13].

The stereo matching algorithm we will use in this thesis is the semi-global matching (SGM) algorithm proposed in Hirschmuller (2008). We shall not go into great detail on the implementation of the algorithm but instead give a short overview. Following the four steps outlined in Scharstein and Szeliski (2002), Hirschmuller (2008) proposes a mutual information (MI) based matching criterion to compute the cost volume. MI, as its name suggests, is information present in two random variables. In the SGM algorithm's case, these variables are two pixels. The cost aggregation step is performed as an approximation of a global energy function defined in the paper [Hirschmuller, 2008, p. 330]. The disparity computation is performed as a simple winner-takes-all algorithm over the disparity dimension. Disparity refinement is also performed to refine the estimated disparities. For a more detailed explanation of the SGM algorithm, we refer to the aforementioned paper.

We will use the SGM implementation by the open-source computer vision library OpenCV[1], called `StereoSGBM`. The SGM implementation by Hirschmuller, is originally a single-pixel stereo matching algorithm.

[1]https://opencv.org/

OpenCV's `StereoSGBM` implementation is a modified version of the original implementation that implements block-matching. The differences in OpenCV's implementation, and the original implementation is outlined in the OpenCV documentation [OpenCV, 2021]. The main difference is the implementation of block-matching instead of single pixel matching. It also uses a different matching cost criterion than the one proposed in Hirschmuller (2008).

2.2 Epipolar Geometry

To see how the disparity value of a pixel can be used to infer depth in a scene, we need to know more about the geometry of a stereo camera setup. We refer to figure 2.3 below which is a top-down view of a stereo camera setup where the thick black lines are the camera lenses. Given a point P in a scene, we can estimate its depth Z if we know the baseline (the distance between the cameras' sensors), the focal length (distance from the sensors to the lenses), and the disparity value of the point. In figure 2.3 the camera sensors are denoted as S and S' for the left and right camera respectively. The focal lengths are the same for both cameras and are denoted as f. The baseline B is considered as one of the sides in a triangle closed off by the line from each sensor in the cameras, to the point P. These lines are called *epipolar lines* and the enclosed triangle is called the *epipolar plane*. Because of the different point-of-views between the left and right camera, P is located at different locations in each image. These are denoted as X and X'. Note that we are only considering the horizontal direction, as we assume the cameras to be calibrated to be coplanar.

In figure 2.3 we can create three different triangles, two smaller triangles at each camera with heights f, and a triangle enclosed in the epipolar plane. By dividing the epipolar plane in half with the line Z, we can prove that the small triangles are geometrically similar to the larger triangles simply by matching angles. Because of this similarity, the combination of the two smaller triangles is geometrically similar to the epipolar plane, and thus the ratios between sides can be expressed as:

$$\frac{B}{Z} = \frac{X + X'}{f}$$

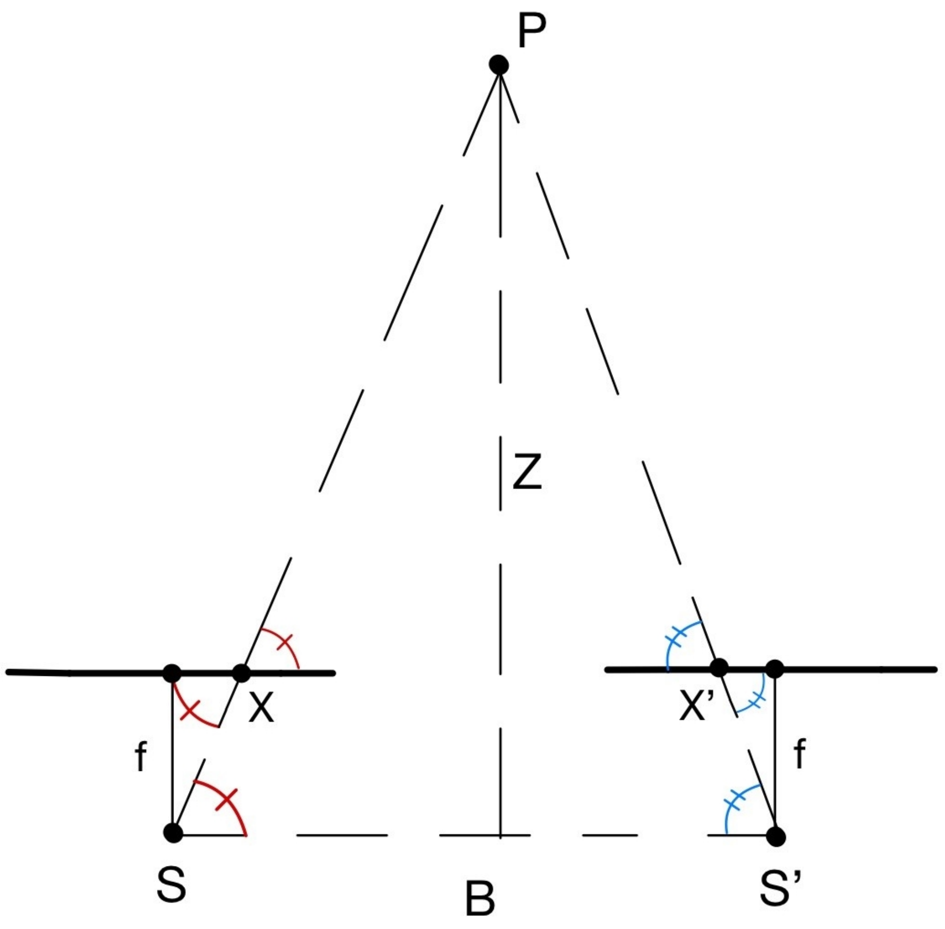


Figure 2.3: Using two cameras, we can calculate the depth of an object by triangulating its position

As we know, $X + X'$ is the displacement of the point P between the left and right image, or the disparity value of the point P. If we replace the term with d and solve for Z, we get:

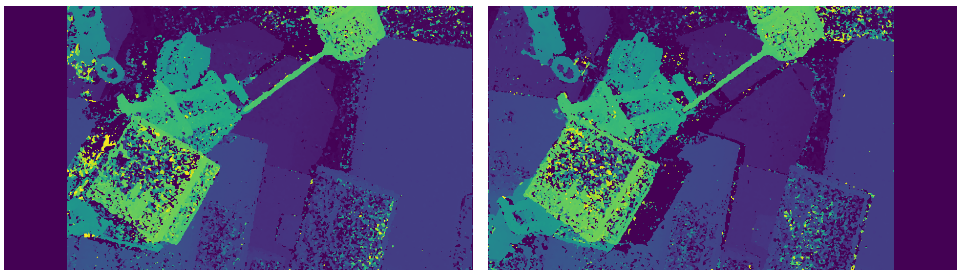
$$\frac{B}{Z} = \frac{d}{f}$$
$$Z = \frac{B \cdot f}{d}$$

With the above equation, we can now get a real depth measure of a pixel by only needing to know the disparity value, baseline and focal length.

2.3 Problems With Stereo Matching

In the recent decade, the use of deep learning has become more and more prominent in stereo depth estimation. This is because of the inherent problems classic stereo matching algorithms suffer from. There are multiple problems related to stereo matching that machine learning solves. The biggest problem is occluded areas. With a stereo camera setup like we have on our head-mounted display (figure 1.1), we capture a scene from slightly different point-of-views. These cameras pick up mostly the same points in a scene, but there are some areas one camera sees, that the other camera doesn't. These are occluded areas. The most obvious examples of this are the leftmost side of the left camera and the rightmost side of the right camera. Using the left camera as a reference, it is impossible to find all pixels that are outside the view of the right camera. In figure 2.1, we can see more of the object in the lower-left corner in the left image, which isn't there in the right image. These pixels are impossible for stereo matching algorithms to calculate the disparity for. This is also the case when comparing the rightmost side of the right image, with the rightmost side of the left image in figure 2.1. To differentiate the between different types of occlusions, we call these *leftmost* and *rightmost* occlusions.

The other type of occluded areas are areas in one image that are covered by objects in the other image. Again, we can observe this by looking at the areas to the left of the large gray box in figure 2.1 (a) with the same area in 2.1 (b). We can clearly see that the right camera has no information about



(a) Left disparity map (b) Right disparity map

Figure 2.4: Disparity maps computed by the SGM algorithm. (a) uses the left image as the reference image, while (b) uses the right as the reference image.

what is behind the gray box, while the left camera has. This means it is impossible to calculate the disparities in this area. A final problem of stereo matching algorithms is large texture-less areas. These areas are hard for algorithms to understand because all pixels have the same pixel intensity. The algorithm is therefore uncertain about the disparities of these pixels. This results in *speckled areas*, areas with spurious disparities. In figure 2.4 we have computed the disparity map of the left and right image from figure 2.1. Here we can clearly see examples of the problems we've stated. Note the leftmost and rightmost occlusions in (a) and (b). We used the left image as a reference in (a) and the right image as a reference in (b). Also note the occluded areas to the right of the foreground objects in figure 2.4 (b) as an example of the other type of occlusion. Lastly, we can observe the overwhelming amount of speckles in the top-right of the disparity maps, as well as on the barrel in the foreground. These are most likely errors due to the problems texture-less areas impose.

2.4 Machine Learning

In this section, we will explain the essential machine learning concepts that are needed as a prerequisite for the rest of the thesis. First of all, we touch on the training of neural networks before we introduce the concepts of loss and activation functions. We will end this section with an introduction to convolutional neural networks, which are the main building block in any neural network that has to do with computer vision.

2.4.1 Training a Neural Network

We begin by explaining the most important step in any neural network, the training. The training phase of neural networks comprises two parts, forward propagation, and backpropagation. We refer to figure 2.5 to explain these concepts. It shows a multi-layer perceptron, what we traditionally think of when thinking of machine learning and artificial intelligence. It is a simple directed, acyclic graph (DAG) where the data flow goes from the input layer, through the hidden layers, and ends in the output layer. In our example, we have three input neurons and two output neurons, but in reality, the number of neurons is usually bigger. Each neuron in the network is represented as a number that is determined by the neurons from the layer before, and the weights connecting them. Each edge in the graph is a weight connecting two neurons and is initialized to random values, usually between 0 and 1. In the forward propagation, data flows through the graph, from the input neurons to the output neurons that make a prediction based on the input. The output layer can be interpreted differently depending on the application of the neural network. In classification networks, we have an output neuron per class that outputs a probability of the input being of that class. Lets say the two neurons in the output layer in figure 2.5 is "cat" and "dog". If the "cat" node has a value of 0.9, while the "dog" node has a value of 0.4, the network is 90% sure the input was a cat, while 40% sure it was a dog. We, therefore, pick "cat" as the prediction since the network was more confident the input depicted a cat, than a dog. When the network has made a prediction, the weights of the network need to be updated based on how correct the prediction was. This is done during the backpropagation. If the network mislabeled the input, it may decrease the weights that lead to that prediction. If it labeled the input correctly, it may increase the weights instead. The combination of forward propagation and backpropagation over multiple iterations, called epochs, will create an increasingly accurate neural network.

2.4.2 Loss Functions

Essentially all neural network training boils down to minimizing a function called the *loss function*. The loss function takes the predicted labels from the neural network and compares them to the true labels, which we call the

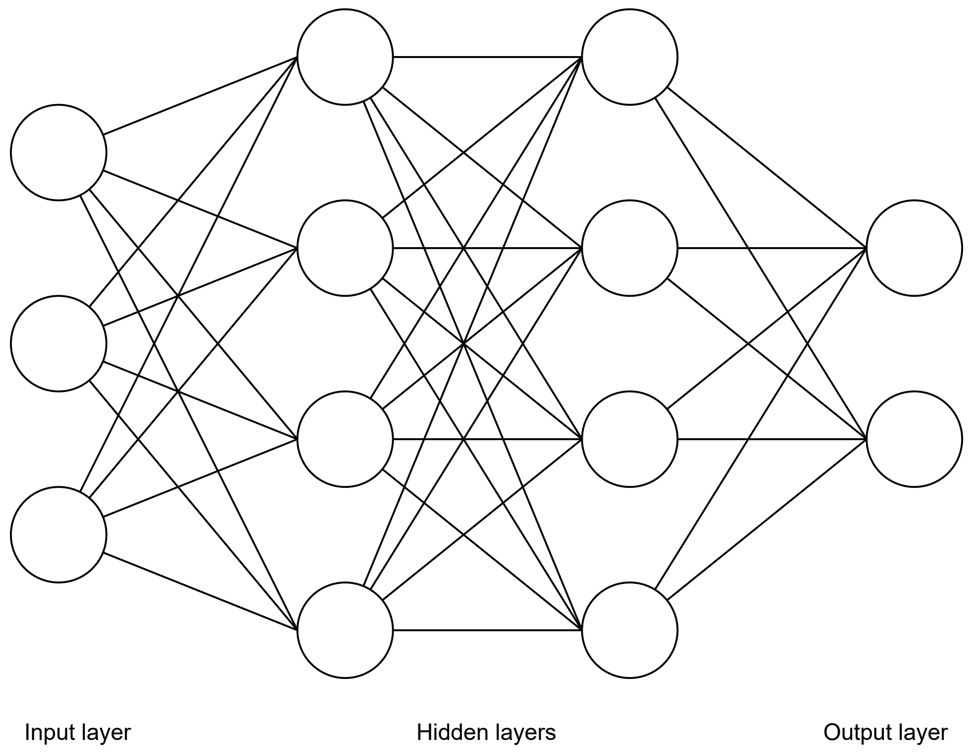


Figure 2.5: Fully-connected multi-layer perceptron

ground-truth in this thesis. The loss, or cost, is calculated and is a measure of how good the predicted values are versus the ground-truth values. A higher loss implies a worse prediction. To minimize the loss we use an *optimizer*. The optimizer determines how the loss function gets minimized and there are multiple optimizers to choose from, depending on the neural network and its use-cases. The choice of loss function and optimizer has a big impact on the neural network. A bad choice of loss function may lead to the optimizer not optimizing the correct weights, while a bad choice of optimizer may lead to poor optimization of the weights. We can think of the relation between the loss function and the optimizer like this: the loss function determines *what* weights should be updated, while the optimizer determines *how* these weights should be updated.

To control the weight updates during the backpropagation, we use a parameter called *learning rate*. This is the rate at which the weights of a network are updated. By increasing or decreasing the learning rate, we have more control over the training of the network. Usually, the learning rate is very small (in the magnitudes ~10^{-2} and ~10^{-3}) to prevent the weights from exponentially increasing or decreasing, or exploding as it

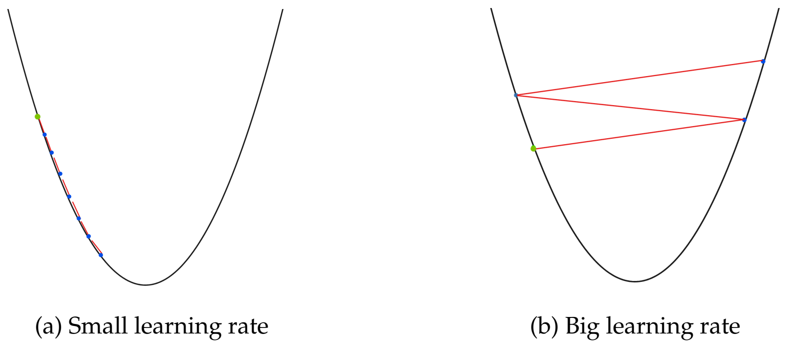


Figure 2.6: Low vs. high learning rate. By using a smaller learning rate, the neural network slowly approaches the local minima. With a high learning rate, the neural network jumps over the local minima and never recovers. It diverges towards infinity.

is often called. This causes the network to diverge away from the *local minima*, the point we would like the network to end up in. In the local minima, the neural network is at its best. We can think of the learning rate as the size of the step towards the local minima. The illustrations in figure 2.6 shows the effect of using a small versus a big learning rate. Starting at the green point with a small learning rate, the neural network slowly approaches the local minima over time. With a bigger learning rate, it causes the neural network to jump over the local minima, from there on it quickly diverges from the local minima jumping back and forth, never reaching it. It is therefore a necessity to use a small enough learning rate to keep the neural network from diverging. However, there is also a reason for not keeping the learning rate too small. With too small of a learning rate, the convergence towards the local minima becomes longer, increasing training time. Figure 2.7 illustrates the loss curves using different learning rates. Ideally, we want a learning rate that finds the local minima as fast as possible without diverging.

2.4.3 Model Generalization

Although a network may train well and increase its accuracy during training, we can not be sure if this accuracy holds up to unseen data, i.e., data that the network hasn't learned before. We want our neural network to *generalize* well to unseen data. After all, a neural network that is only accurate on the training dataset and inaccurate on everything else

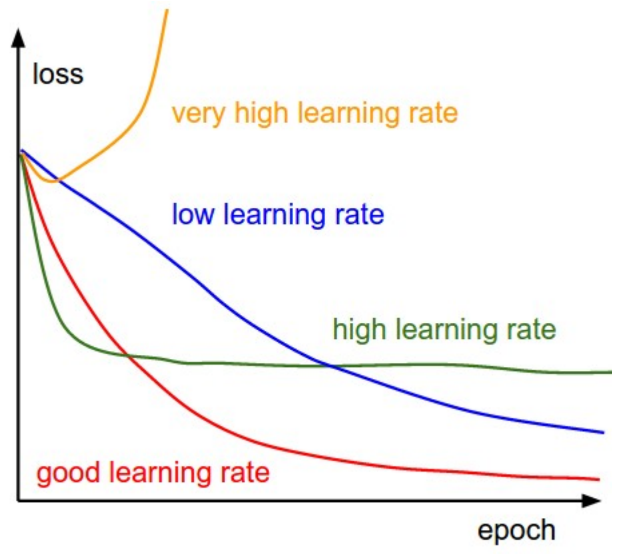
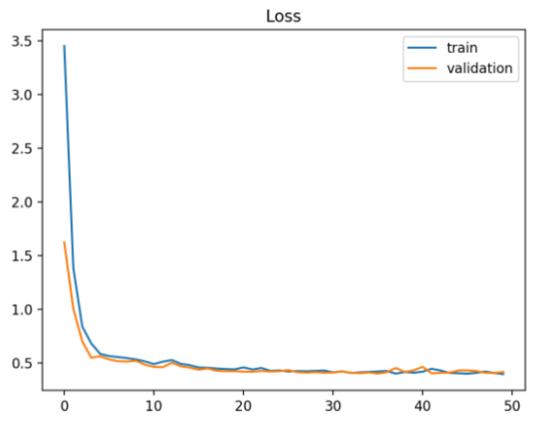


Figure 2.7: Effects of different learning rates on training. Higher learning rates leads to divergance.

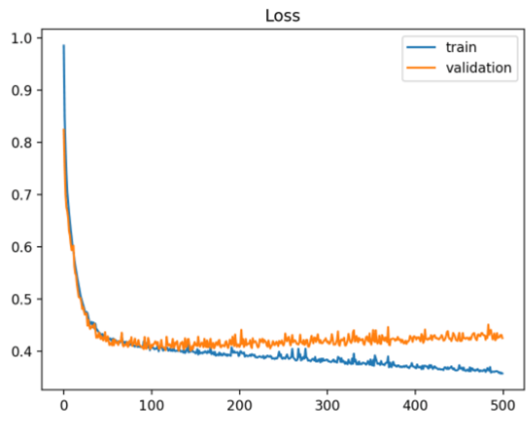
Source: https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10

is useless. Generalization is the goal of all neural network training, and we, therefore, need a way to see if our model generalizes well. To do this, we can split the dataset into two parts, a training split, which is the part of the dataset that we use for training, and a validation split. The validation split is used to validate our neural network against unseen data, to check if the training has any effect on it. It is therefore important to not do backpropagation during the validation phase, as the network shall not have any prior knowledge about the validation set. To check if the model generalizes well, we can plot the mean loss of all dataset samples as a function of epochs for both the training and validation split. This leads to two loss curves, one for the training set and one for the validation set. Ideally, we want both curves to converge and perfectly overlap each other. This would mean the neural network is as good on unseen data, as it is on the training set. Typically we want the loss curves to look something like the ones in figure 2.8 (a).

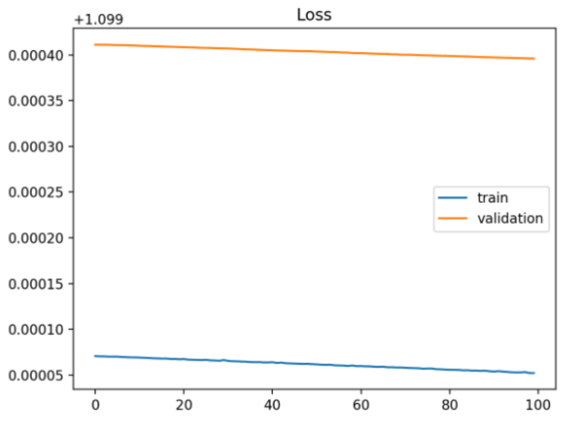
We can diagnose a lot about a model by looking at the mean loss over epochs. Two very important trends to look out for is *overfitting* and *underfitting*. Overfitting is when the validation loss is increasing, while the training loss is decreasing, as illustrated in figure 2.8 (b). This is a sign



(a) Good fitting model



(b) Overfitting model



(c) Underfitting model

Figure 2.8: Trends we can encounter during the training of a neural network.

Source: https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/

that the neural network is overfitting to the training dataset by learning features that are very common in the training set, but not common in the validation set. This causes the network to become biased towards these features since they result in the smallest loss, and disregard more general features that might cause a slightly bigger loss. Let's take figure 2.9 as an example. It shows a group of points in blue and red colors. We want to train a neural network that can calculate a line that divides these points into a blue side and a red side. In this case, the green line fits better than the black line, as it correctly divides all red points to one side and the blue points to the other. However, if we introduce a new set of points the network has never seen before, the green line would most definitely fail as it is way too specialized towards the specific points shown in figure 2.9. On the contrary, the black line would be more likely to fit a different set of points, as it is more generalized. In this case, the green line is overfitted towards that specific group of points and it shows why generalization is so important. To prevent overfitting, we usually need to reduce the complexity of our network. This means reducing the number of weights, forcing the network to learn only the most important features. We can also use other measures such as dropout layers. Dropout layers have a random chance of dropping neurons in a layer (i.e., sets them to 0), features that if not for the dropout layers the model could become biased towards, and thus overfit on them.

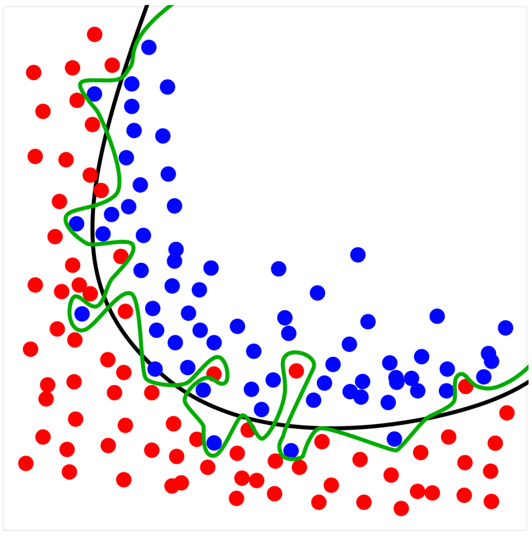


Figure 2.9: Regression model that divides points into a red and blue side using a regression line

Source: https://en.wikipedia.org/wiki/Overfitting

Underfitting is the opposite of overfitting. In this scenario, the network doesn't learn enough information from the training set. This results in models with poor performance. We can identify underfitting with loss curves that converge slowly and fast (or not converging at all) illustrated in figure 2.8 (c). The cause of underfitting is usually due to the network not learning enough information about the dataset. This can be remedied by increasing the size of the dataset, or increasing the complexity of the network, i.e., increasing the number of learnable weights.

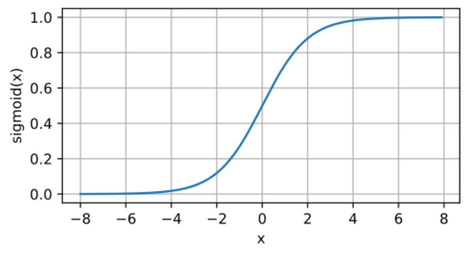
2.4.4 Activation Functions

A single layer in a neural network is in itself a linear function on the form $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$ where $\boldsymbol{x}$ is the input to the layer $\boldsymbol{f}$ with weights $\boldsymbol{W}$ and bias $\boldsymbol{b}$. More specifically, this is an affine function because the function does not intercept the origin due to the constant $\boldsymbol{b}$. A neural network containing multiple such layers is therefore an affine function in itself since the combination of multiple affine functions still results in an affine function. A neural network with only linear layers will always assume that a change in input will linearly change the output. This assumption is in most cases wrong and hinders the learning of more complex features. If the network only learns the linear relations it misses out on more complex non-linear relations in the input. To introduce non-linearity to a neural network, we employ what is called an *activation function*. This function is applied to most if not all layers in a neural network to break the linear relation between input and output. They are called activation functions since they determine what neurons in a layer should activate. With an activation function $\boldsymbol{\psi}(\cdot)$, we can turn the affine, linear layer $\boldsymbol{f}$ into a non-linear layer $\boldsymbol{\psi}(\boldsymbol{f})$.

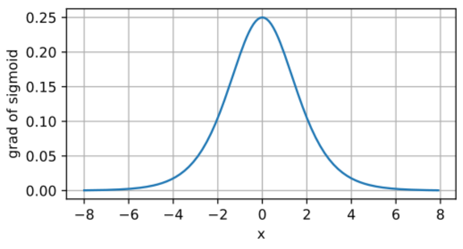
Another advantage of activation functions is the ability to force neurons to be within a certain range. Take the logistic sigmoid activation function as an example:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Figure 2.10 (a) shows the plot of said function. As we can observe, any input x that is passed through the sigmoid function is squeezed between



(a) Sigmoid function



(b) Derivative of sigmoid function

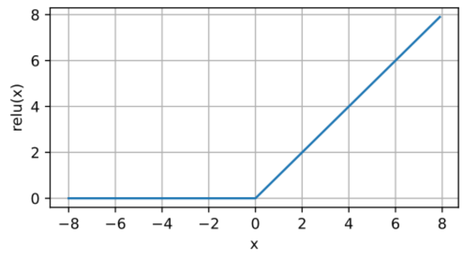
Figure 2.10: The sigmoid function along with its derivative.

Source: https://d2l.ai/chapter_multilayer-perceptrons/mlp.html

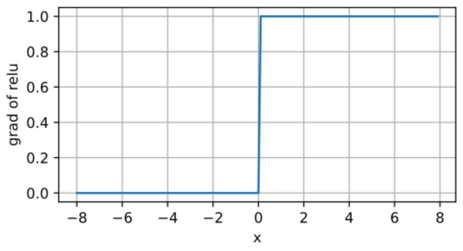
zero and one. This is useful in a classification network, where the output labels should be a probability in this range. Other activation functions, functions similarly, like the hyperbolic tangent, which transforms the input to the range -1 to 1. The most popular activation function of today because of its simplicity and computational efficiency is the rectified linear unit or ReLU. It simply deactivates all neurons with values less than zero. Every positive number stays the same. See figure 2.11 (a). The advantage of the ReLU function is that the derivative of the function is pretty straightforward:

$$\frac{\partial}{\partial x}\text{ReLU'}(x) = \begin{cases} 0, & x <= 0 \\ 1, & x > 0 \end{cases}$$

With its simple derivative, it addresses the *vanishing gradients* problem. The



(a) ReLU function



(b) Derivative of ReLU function

Figure 2.11: The ReLU function along with its derivative.

Source: https://d2l.ai/chapter_multilayer-perceptrons/mlp.html problem of vanishing gradients is prominent in deep neural networks with many layers. With certain activation functions that squeeze the input into small ranges, like the sigmoid and tanh functions, the derivative of large values is very small. As illustrated in figure 2.10 (b), very small or large values of x cause very small *gradients*, or changes to the weights. This can prevent the neural network from learning entirely.

In this section, we've mainly focused on multi-layer perceptrons, but everything mentioned in the previous sections can be applied to other types of neural networks as well, like convolutional and recurrent neural networks.

2.4.5 Convolutional Neural Networks

Multi-layer perceptrons are suboptimal for image processing as it doesn't provide any spatial context about an image. In this thesis, we will use a convolutional neural network (CNN), a network specifically for image processing. The main building block of a CNN is the convolutional layer. These layers use a kernel, a window that slides, or convolves, across an image and output the cross-correlation (sum of products) between the kernel, and the area it covers into a *feature map*. Multiple parameters change how a convolutional layer behaves: the kernel shape, stride, dilation, and padding. The kernel shape is self-explanatory. It is the shape of the kernel, both width and height. The kernel does not have to be square-shaped and can have a different width and height. Kernels are usually small, around 3×3, 5×5, or 7×7, but can be bigger or smaller depending on their use-cases. Note that the kernel shape directly influences the output shape, as a 3×3 kernel compress a 3×3 area into one pixel. The stride determines how many pixels the kernel is shifted by after every cross-correlation operation. With a stride of one, the kernel skips one pixel, meaning it overlaps with the previous area. The dilation is the space in between each element in the kernel. A dilation set to two means each element in the kernel is spaced out with a single space in between. Lastly, the padding parameter is the only parameter that doesn't affect the kernel but affects the input image itself. When padding is applied, a border of pixels is applied around the input image, most commonly pixels with a value of zero. If we pad an image by two, it adds a two-pixel wide border of zeros around the entire image. Note that with padding set to two, it increases the spatial dimensions (width and height) of the input image by four, as the border is applied to all outer edges. All these parameters have an impact on the output shape. This shape can be calculated with the formula proposed by Dumoulin and Visin in their guide on convolution arithmetic, ''A guide to convolution arithmetic for deep learning'' [Dumoulin and Visin, 2016]:

$$o = \left\lfloor \frac{i + 2p - k - (k-1)(d-1)}{s} \right\rfloor + 1$$

The output shape o is determined by the input shape i, the padding p, the kernel size k in the same spatial dimension as i (if i is the width, then k needs to be the width of the kernel), the dilation d and the stride s. It is worth knowing this formula as it is important to know the output shapes of convolutional layers when designing a CNN.

Another property of a convolutional layer is the number of input and output channels. These properties determine what type of image is accepted as input to a layer, and how the output of a layer looks. An image usually comprises one or three channels and each channel contains some information about each pixel in the image. A grayscale image only contains one channel that specifies the luminosity of each pixel using a byte to represent a value between 0 and 255. The higher the pixel value, the brighter the pixel. On the other hand, an RGB image comprises three channels, one channel for each of the red, green, and blue values of each pixel. The mixing of these three channels results in the final color of the pixel. See figure 2.12. There are also other ways of representing an image with three channels, like the YUV image format which splits the image into a luminance channel (Y) that represents the luminosity of each pixel, and two chrominance channels (U and V, often referred to as Cb and Cr) for information about the red and blue nuances of the image. The RGB model is the most widely used. The reason the number of input channels is important to a convolutional layer is that it applies the kernel on each channel separately. The number of output channels specifies the number of channels the output map has. For each output channel, the convolutional layer creates a separate kernel, i.e., if a convolutional layer has 16 output channels, 16 different kernels convolves over each input channel. In this way, we can control the number of learnable parameters, or weights, since each kernel is an N$\times$M window of weights that is updated during backpropagation.

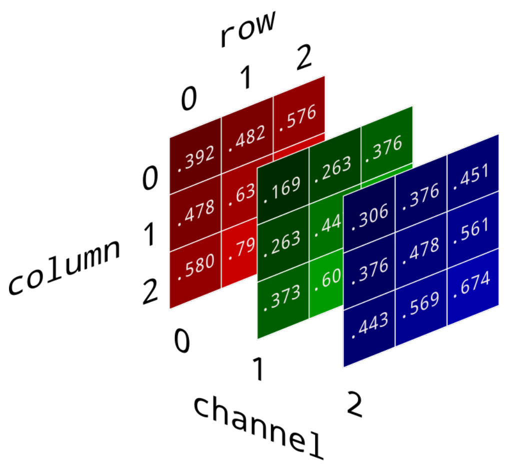


Figure 2.12: RGB color channels of a 3×3 image

Source:
https://www.kdnuggets.com/2019/12/convert-rgb-image-grayscale.html

2.5 Related Work

Stereo depth estimation through stereo matching and machine learning is one of the most researched fields in computer vision, as the use-cases are many. In this section, we discuss different machine learning approaches to calculating stereo depth. Early approaches, like the MC-CNN architectures proposed in Žbontar and Le Cun (2015a) and Žbontar and LeCun (2015b), saw the use of convolutional neural networks to estimate the matching cost of each pixel. With the inception of the large SceneFlow stereo vision dataset [Mayer et al., 2016], deeper machine learning architectures were possible. DispNet [Mayer et al., 2016], introduced in conjunction with the SceneFlow dataset, was the first model to take advantage of the large dataset. It is a simple autoencoder network comprising an encoder that downsamples the left and right images into their most important features and a decoder that upsamples the features to a dense disparity map. It comprises 26 layers in total, making it far larger than the MC-CNN architecture with its mere eight layers.

Chang and Chen (2018) introduces a pyramid stereo matching network (PSMNet), an architecture that downsamples feature maps to different resolutions using pooling layers and creates a cost volume from the upsampled features. Gu et al. (2019) proposes the use of cascaded cost volumes, cost volumes created from a feature pyramid (feature maps at different resolutions). Coarse-resolution disparity maps are predicted from coarse-resolution cost volumes. The lower resolution disparity maps are used to predict higher resolution disparity maps. Tankovich et al. (2020) proposes HITNet, a hierarchical approach of iteratively refining tiles (small sections of a disparity map) of increasing sizes. In this thesis, we will implement the HITNet architecture to use in our experiments. The next chapter goes more in-depth on the architecture of HITNet.

Chapter 3

Design

In the following chapter, we explain our design. We outline the complete pipeline and go into detail about every part of it. Also, we quickly mention datasets we plan on using during the training of our model, and give reasons for why we use them.

3.1 Datasets

In this section, we explain the datasets we will use for our experiments, and how we plan to use the data as input to the model. The data we feed into our model will have the largest impact on performance, as bad input data leads to bad results. That is why we need a good dataset to train our model. Fortunately, there are multiple stereo vision datasets for our purposes. The KITTI Vision Benchmark Suite [Geiger et al., 2012; Menze and Geiger, 2015], the Middlebury stereo datasets [Scharstein et al., 2014], and the SceneFlow datasets [Mayer et al., 2016] are all datasets used in stereo vision. Each dataset has its advantages and disadvantages. Both the KITTI and Middlebury datasets have real-world data, acquired by different means. The KITTI datasets comprise two versions, a 2012 dataset, and a 2015 dataset. Both datasets are collected by a camera and sensor rig mounted to a car. See figure 3.1.

The Middlebury datasets comprise six different versions collected in the timespan 2002 to 2021. In the latest dataset, Scharstein et al. (2014) acquired its data from a mobile device on a robot arm. Both of these methods result

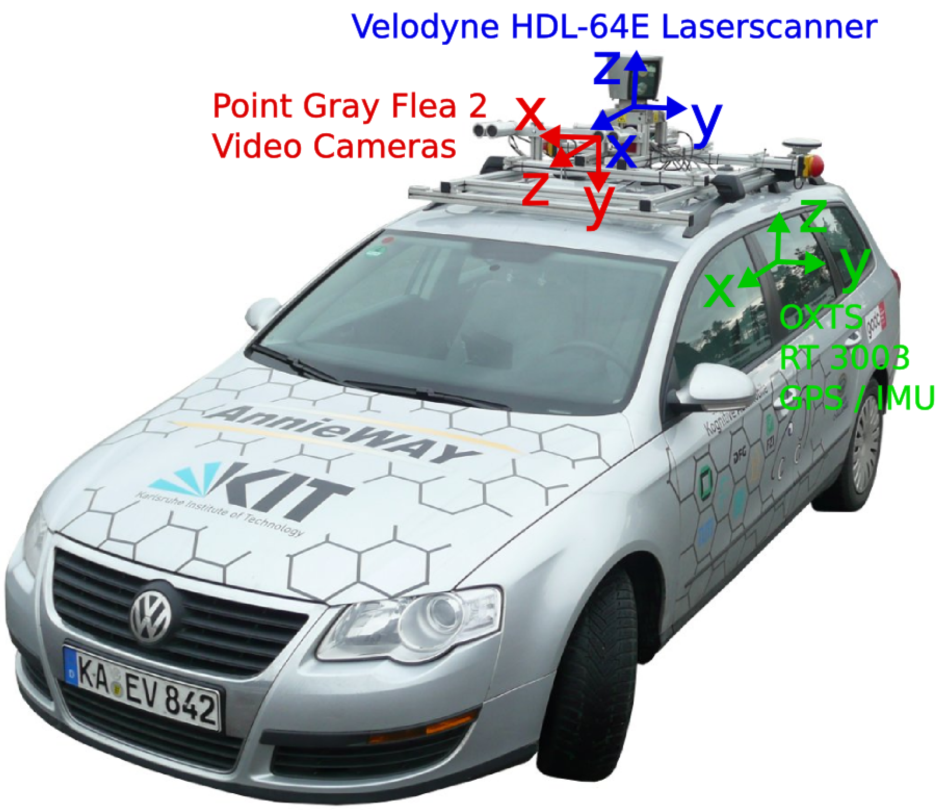


Figure 3.1: Camera and sensor rig utilized to collect data for the KITTI 2012 and 2015 datasets

Source: http://www.cvlibs.net/datasets/kitti/setup.php

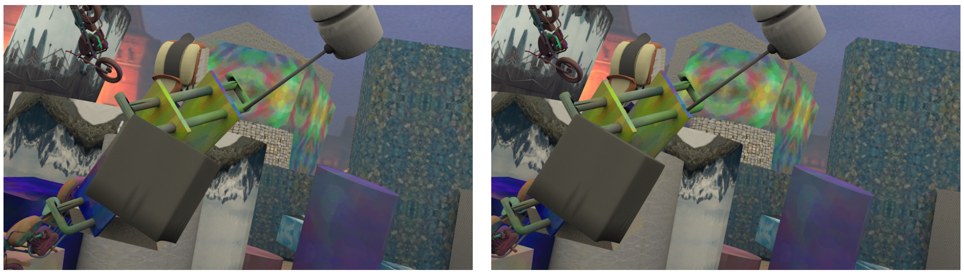


Figure 3.2: A stereo image pair taken from the SceneFlow FlyingThings3D dataset [Mayer et al., 2016]

in accurate datasets, but both suffer from the same problem: the dataset size. When training a model, we want a large dataset with a large variety of samples to ensure the model generalizes well. Both the KITTI and Middlebury datasets only contain a handful of samples, which excludes these datasets as viable training sets. The solution to this problem is to use another larger dataset for the training of the model while using KITTI and Middlebury as benchmarks for pretrained models. Lastly, both of these datasets have leaderboards for different performance metrics. This makes it easy to find state-of-the-art stereo vision models, and directly compare the results of our model with theirs.

The dataset we will use for training is the SceneFlow dataset. This dataset contains over 39000 synthetically generated stereo image pairs with accompanying ground-truth images split into three different datasets: *Driving*, *Monkaa*, and *FlyingThings3D*. We will use the latter as it comes pre-split into a training and validation set. Combined, the FlyingThings3D dataset contains around 25000 data samples (~21000 training samples and ~4000 validation samples) making it quite suitable for training. The disadvantage of using the SceneFlow dataset is that the data is synthetic. This can lead to our model suffering a performance drop when switching from the synthetic domain, the to real-world domain. This is not a concern for us at the moment, but it is worth mentioning if we decide to use our model for any real-world applications in the future.

3.2 The Pipeline

Before explaining our pipeline, we will explain our reasoning behind it. The main reason for choosing our pipeline is our need for a state-of-the-art machine learning architecture that we can build upon using the semi-global matching algorithm. The architecture of choice proposed by Tankovich et al. (2020), is also one of the fastest performing models on the KITTI benchmark leaderboards. This is important for applications in mixed and augmented reality, where real-time performance is crucial for the best possible user experience. Slow depth estimation leads to low framerates and stuttering. The HITNet model has an inference time, or prediction time, of 0.02 seconds which in theory means it can process around 50 frames per second. It achieves this performance due to its small number of model parameters (weights and neurons) versus other approaches. Even with the small number of parameters, HITNet still compares to the state-of-the-art accuracy-wise. The combination of speed and accuracy is what makes Tankovich et al. (2020) approach appealing to us.

The overview of the pipeline is illustrated in figure 3.3 and is the same as proposed in Tankovich et al. (2020), with the slight difference where we use semi-global matching to produce additional disparity maps based on the left and the right image. We believe that passing additional depth information as input to our model will result in a more accurate predicted disparity map. Given the extra depth information, we expect the model to converge faster, as the initial disparity maps will give the model a "head start" on the training versus other stereo depth estimation networks that only use left-right image pairs. The main parts of the HITNet pipeline, which are also present in our pipeline, are an initialization module and a propagation module. The upcoming sections is a detailed description of the entire HITNet model, thus all subsequent sections (excluding section 3.3) is based on Tankovich et al.'s paper "HITNet: Hierarchical Iterative Tile Refinement Network for Real-time Stereo Matching" [Tankovich et al., 2020].

3.3 Image Inpainting

In this thesis, we will implement the pipeline as close as possible to the original HITNet model, but we also want to explore the idea of only using the left image along with its disparity map as input. This is to alleviate some of the extra overhead caused by the SGM algorithm. However, with this, we encounter a problem. In multiple parts of the pipeline, we need to

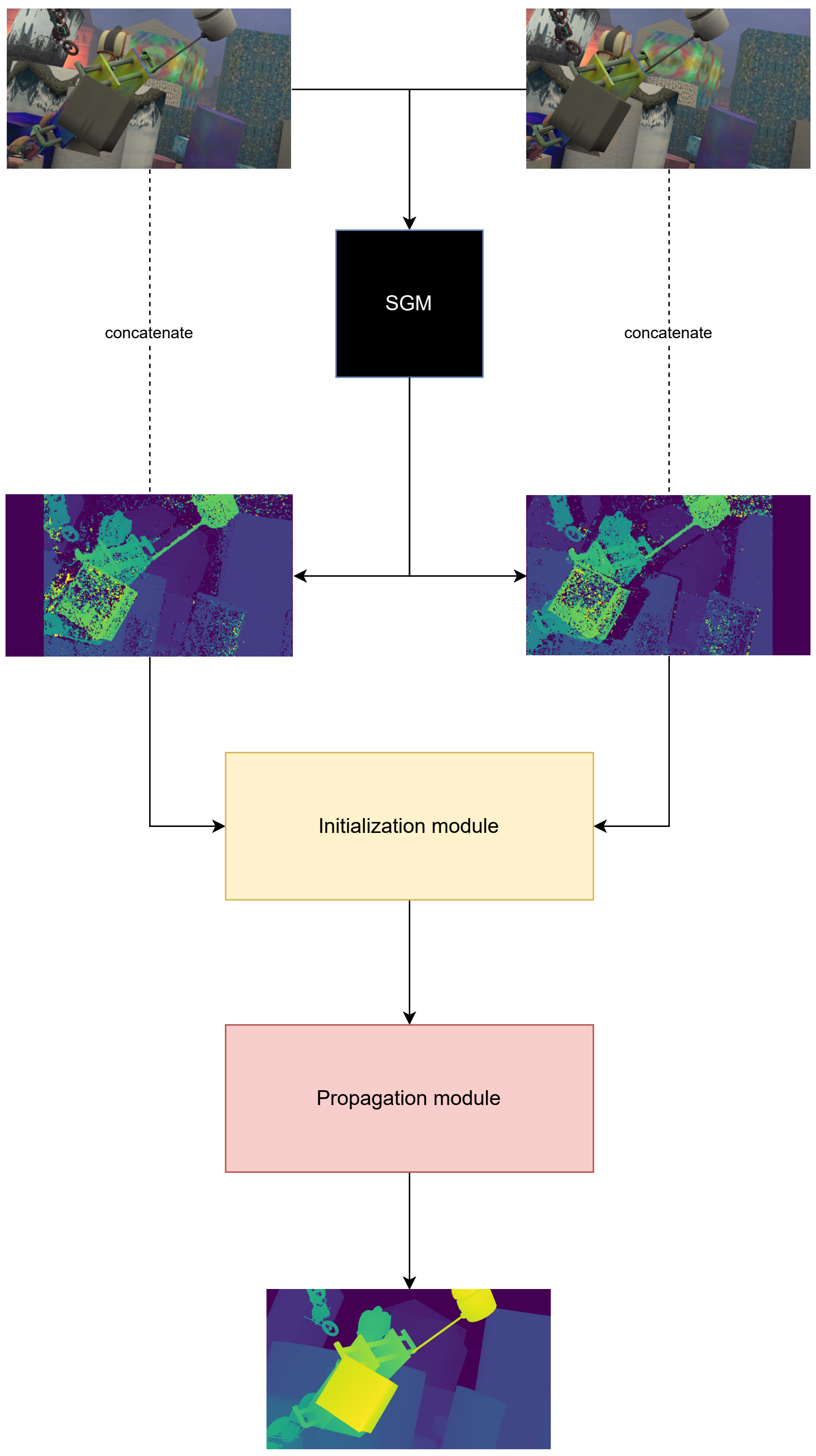


Figure 3.3: Our stereo network pipeline

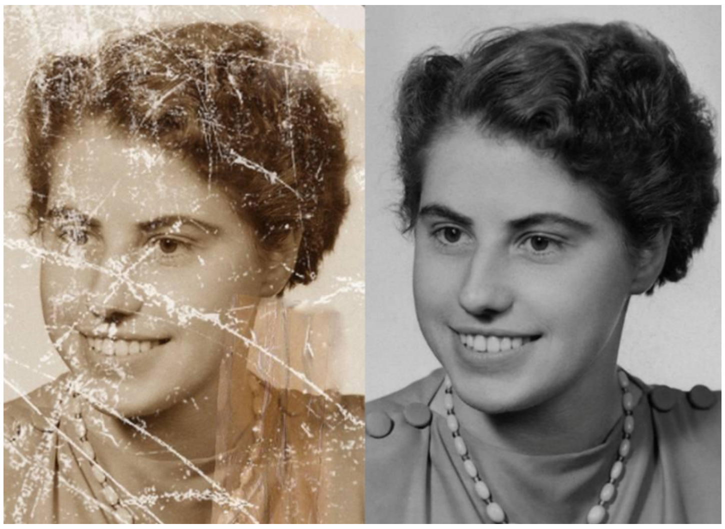


Figure 3.4: Old photo restored by image inpainting

Source: https://towardsdatascience.com/how-to-perform-image-restoration-absolutely-dataset-free-d08da1a1e96d

compute the matching cost between pixels in feature maps extracted from the left and right images. This means we need a way of representing the right image. A way of doing this, without using the original right image, is to warp the left image into the right image. This is relatively simple to do, as we have the left image's disparity map at our disposal. As explained in chapter 2, the disparity of a pixel is the length a pixel has moved between two images in the x-direction. We can therefore recreate the right image from the left image by using the following relation

$$x_r = x_l - d$$

We can find any pixel (x_r, y) in the right image by choosing pixel $(x_l - d, y)$ in the left image. Note that we assume coplanarity between the images. With a perfect disparity map, this can recreate a perfect representation of the right image, but with any traditional block-matching algorithm comes problems relating to occluded areas (discussed in section 2.3). Since occlusion errors happen in obscured areas in the right image, there is no way of warping these areas correctly. Our solution to this is to use image inpainting.

Image inpainting is the process of restoring lost information in images with

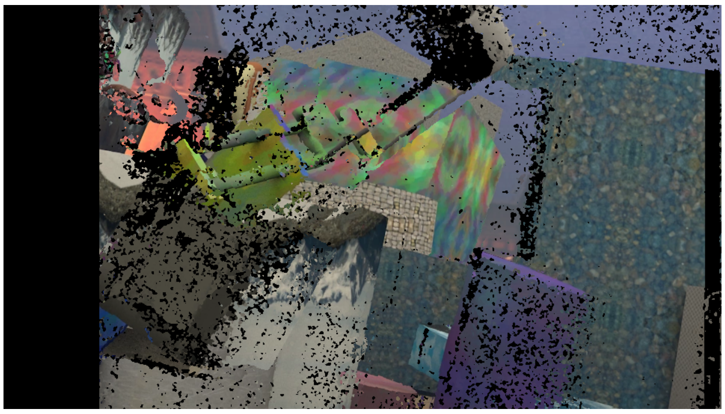


Figure 3.5: Warped right image using the left image and its disparity map

machine learning. It is used to restore old paintings and photographs that have been worn out over time, and to remove objects from an image, filling in the missing area with the background. Image inpainting in itself is a big topic in computer science. Because it falls outside the scope of this thesis, we shall not go into detail on how image inpainting works. We will instead use an already established model that we think will do well in our use case. Our image inpainting model of choice is proposed in Liu et al. (2018). They propose a model that accurately fills in irregularly shaped holes, holes that are commonly produced by the SGM algorithm. See figure 3.5 for the warped left image.

In figure 3.6 we have visualized how the inpainting module will work in our pipeline. We feed the left image along with its disparity map into the inpainting network, and pass the output of the network as input to HITNet (denoted by the three dots at the end of figure 3.6). We will train the inpainting module separately and treat it like a black box, just like the SGM algorithm. We will not go into further detail on the model architecture and instead refer to the original paper Liu et al. (2018) for further reading.

If this approach works, we plan to integrate the image inpainting module into our pipeline. The module will precede the initialization module in figure 3.3. The initialization module will in that case take the left image along with its disparity map, together with the recreated right image as input.

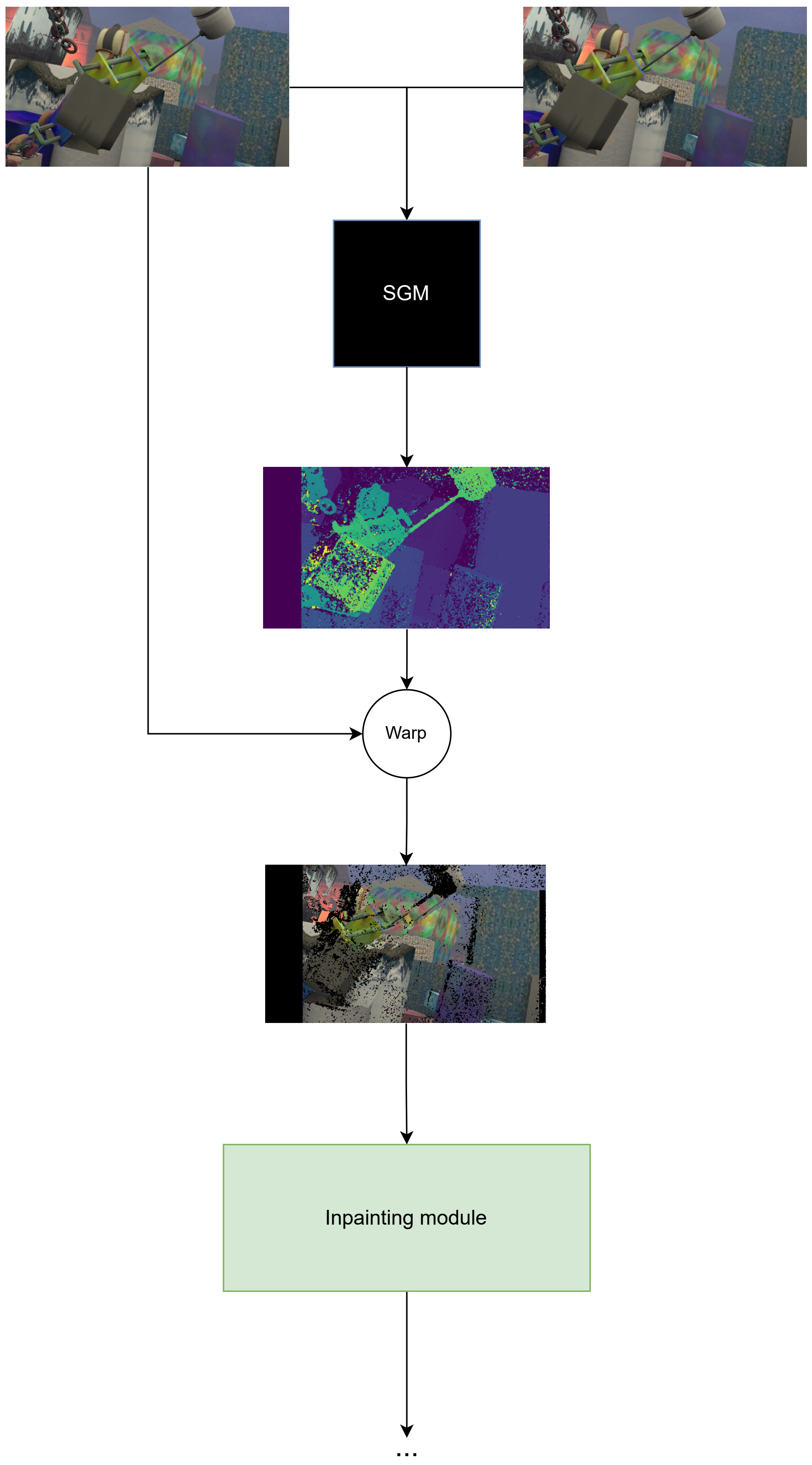


Figure 3.6: Use of inpainting before the rest of the pipeline

3.4 Initialization Module

This section will cover the first step in the pipeline, which is the extraction of features from the right and left images, and the initialization of the tile hypothesis maps at each resolution. Figure 3.7 is a detailed overview of this module. The colored boxes represent parts of the module that are trainable, while the circles represent other, non-trainable operations such as feature matching.

3.4.1 Feature Extraction

We start by explaining what a feature is in the context of computer vision. This will give a better understanding of the sections to come. Let us take an image classification network as an example. In a classification network, we want to classify the thing depicted in an image. Without any prior knowledge, the network is useless as it will not know the difference between a dog and a cat. To find these differences, the network needs to know the features that differentiate them. A cat more likely has pointy ears, as opposed to a dog that has more rounded ones. Cats have whiskers, while dogs have large snouts. These are different features that the network will learn to differentiate cats from dogs. Figure 3.8 shows the weights of the kernels in a CNN layer. The layer has learned to find low-level features such as edges. The deeper in the network a convolutional layer is, the more complex features the kernels will learn. Figure 3.9 shows kernels deeper in a convolutional neural network that extract higher-level face-like features, like eyes and noses. The features extracted by CNN kernels are aggregated into a single image called a *feature map*. The feature maps are usually passed into other CNN layers to extract even more complex features.

The feature extraction network in the initialization module (colored green in figure 3.7) will take the left and right images, along with their disparity maps computed by the SGM algorithm, and extract features that are relevant for depth estimation. We will use a single feature extraction network and pass the left and right samples through it separately. This leads the network to learn features in both images, and the correlation between them. It also reduces the number of model parameters versus having two separate networks for the left and right samples, ultimately

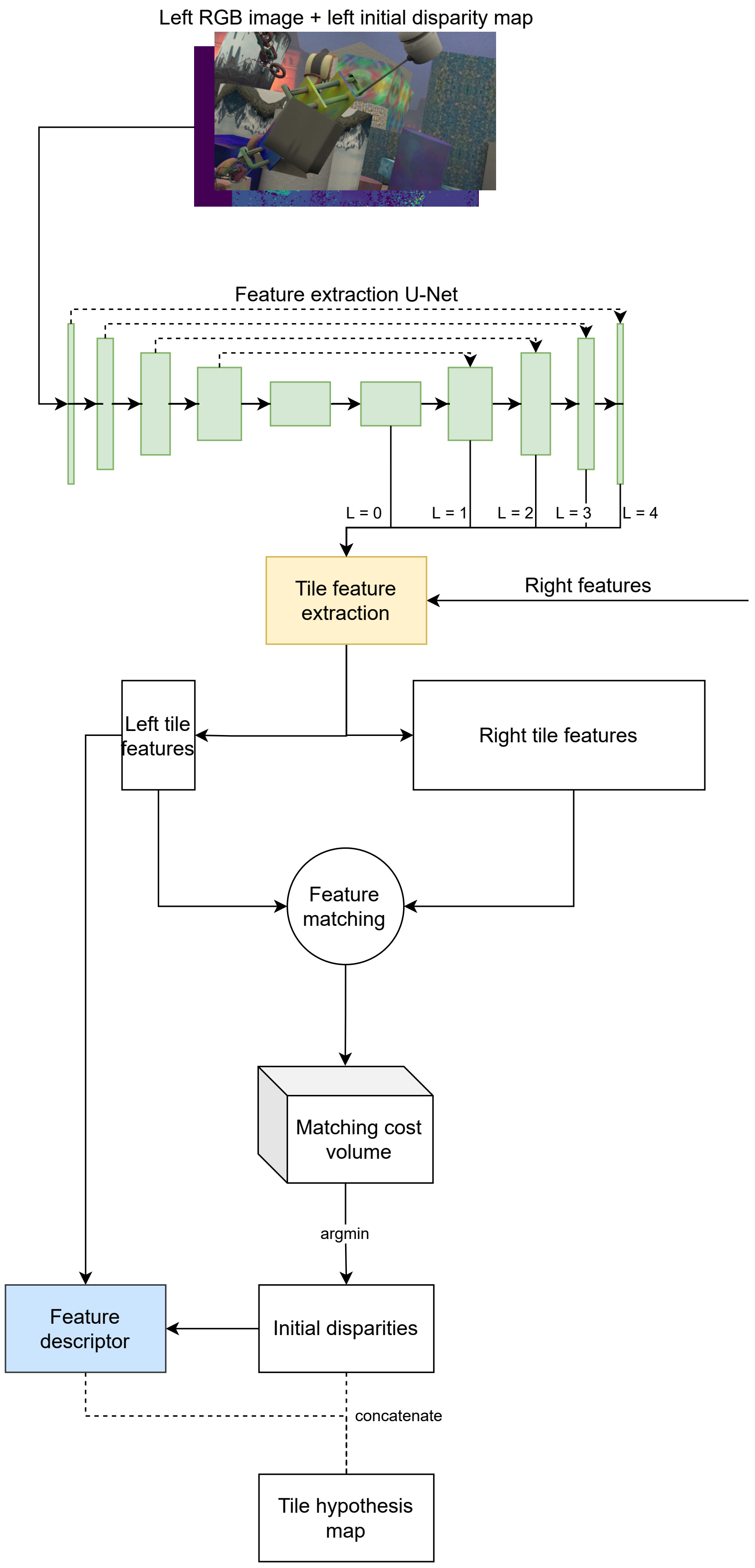


Figure 3.7: Feature extraction and initialization of tile hypotheses

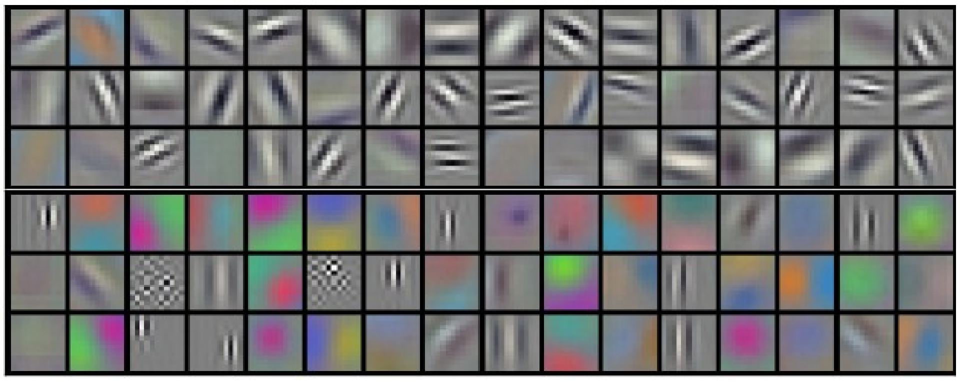


Figure 3.8: Visualization of CNN layer kernels. These are features a CNN kernel picks up during training.

Source: https://discuss.pytorch.org/t/why-do-we-want-many-output-channels-in-a-convolutional-neural-network/8789

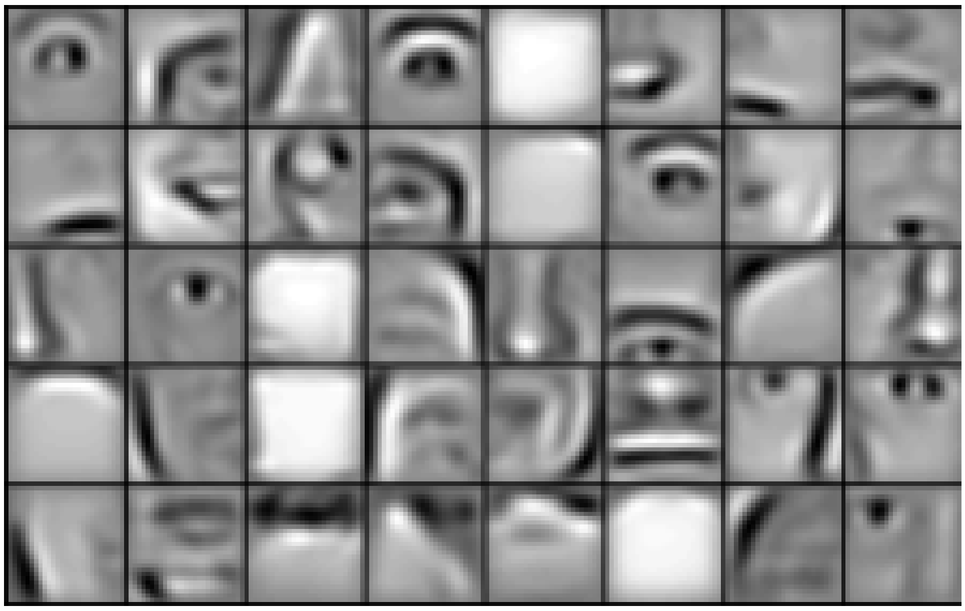


Figure 3.9: Higher-level features extracted by deep CNN kernels. From Lee et al. (2009)

resulting in faster training and inference times.

The feature extraction network will be implemented as a five layer U-net [Ronneberger et al., 2015]. Each layer in the encoder (downsampling layers) has a skip connection to the respective layer in the decoder (upsampling layers). At each upsampling step in the decoder, we extract the feature map ϵ_l^L and ϵ_l^R at each resolution $l \in [0, 4]$ for the left and right samples. These feature maps will be used in later steps of the network. A single layer in the encoder will consist of two convolutions followed by leaky ReLU activation functions as non-linearities. The first convolution is a single-strided convolution with a 3×3 kernel, while the second one uses a 2×2 kernel with stride two to halve the resolution. A single layer in the decoder will consist of the same 3×3 convolution as in the encoding layer, but will instead be followed by a double-strided 2×2 transposed convolution to double the resolution of the input feature map. Each convolution will be followed by a leaky ReLU activation, the same as in the encoder layers.

3.4.2 Tile Hypothesis Maps

Before we can explain the design for the rest of the initialization module, we need to explain the concept of *tiles* and *tile hypotheses* which are an integral part of HITNet. As explained in Tankovich et al. (2020), a *tile* is a local N×M area in a feature map. A tile is extracted using a N×M convolution that convolves over the feature map. The extracted tiles are called tile features and are similar to a feature explained in the previous section, a scaled-down representation of the tile in the original feature map. See figure 3.10.

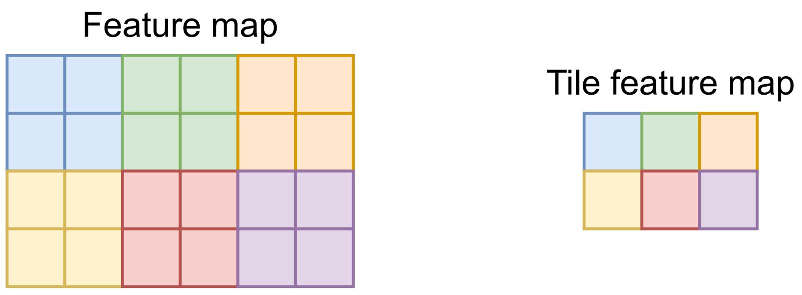


Figure 3.10: A 6×4 feature map and its corresponding tile features after extracting 2×2 tiles

A *tile hypothesis* is the combination of a *tile disparity* d, the *slant gradients* d_x and d_y of the tile and a tile feature descriptor $\mathbf{p}$ on the form

$$\left[d, d_x, d_y, \mathbf{p}\right]$$

The tile disparity is the disparity value for the tile, i.e., the disparity for the entire N×M area. The slant gradients represent the orientation of the tile. It can be thought of as the normal vector of the tile (the vector perpendicular to a plane's surface). Figure 3.11 shows the slant gradients for each pixel in the image. The brighter the pixel, the more slanted the surface is. Note the gradient on the barrel in the bottom right caused by the round shape of the object.

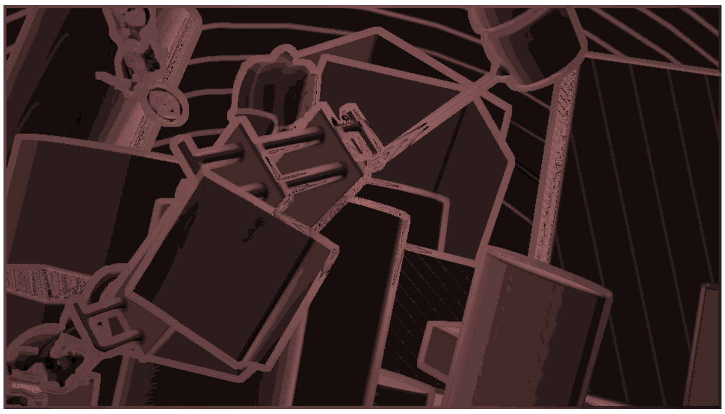


Figure 3.11: Ground-truth slant gradient map of an image in SceneFlow

The tile feature descriptor is a description tied to a tile hypothesis and can be used by the neural network to attach additional information to the tile hypothesis. The tile descriptor is a simple convolutional layer and is therefore trainable. This means the model itself can attach information it sees fit. A tile hypothesis is created for each tile in a feature map, giving us multiple tile hypotheses. All tile hypotheses in a feature map are aggregated into a single tile hypothesis map. In the propagation module, the tile hypothesis maps are updated iteratively to increasingly refine each tile, giving us the final disparity map at the last iteration. This is further explained in section 3.5.

3.4.3 Tile Feature Extraction

The next step in the initialization process is to extract tile features from the feature maps ϵ_l^L and ϵ_l^R for all resolutions l. Tile features are essentially the same as features explained in section 3.4.1. The extracted tile features represent a tile in the feature map. The tile feature extraction is done by a single convolution with kernel 4×4. The convolution convolves over each ϵ_l^L and ϵ_l^R at all resolutions l separately and extracts tile features resulting in tile feature maps $\tilde{\epsilon}_l^L$ and $\tilde{\epsilon}_l^R$. Extracting tile features at all resolutions l will give us different sized tiles based on the resolution the tile feature was extracted from. This is because when extracting features in the coarsest resolution feature map, which is 1/16th of the full resolution, we extract $4 \cdot 16 = 64$ sized tiles in the original feature map. A larger tile gives the network information about a greater area in the full-sized feature map and thus will give the model a much greater spatial context. The model should therefore be able to identify large objects and large texture-less areas (which notoriously has been hard for a stereo depth estimation algorithm to deal with), while also understanding smaller and more detailed areas from tiles extracted from higher resolutions. The tile feature extraction is done iteratively for all resolutions, from the coarsest to the finest, and results in tile feature maps that cover increasingly finer areas in the original feature map. More specifically, all tile feature maps with resolution l, where $l \in [0, 4]$ ($l = 0$ is the coarsest and $l = 4$ is the finest), will cover:

- $l = 0$: 4×4 tile feature extraction in the $W/16 \times H/16$ feature map will cover 64×64 pixels in original resolution ($4 \cdot 16 = 64$)
- $l = 1$: 4×4 tile feature extraction in the $W/8 \times H/8$ feature map will cover 32×32 pixels in original resolution ($4 \cdot 8 = 32$)
- $l = 2$: 4×4 tile feature extraction in the $W/4 \times H/4$ feature map will cover 16×16 pixels in original resolution ($4 \cdot 4 = 16$)
- $l = 3$: 4×4 tile feature extraction in the $W/2 \times H/2$ feature map will cover 8×8 pixels in original resolution ($4 \cdot 2 = 8$)
- $l = 4$: 4×4 tile feature extraction in the $W \times H$ feature map will cover 4×4 pixels in original resolution ($4 \cdot 1 = 4$)

To make the above list easier to understand, take figure 3.12 as an example.

Here we extract 2×2 tile features in a 2x downscaled feature map. The colored areas in the full resolution feature map is the areas each tile feature covers.

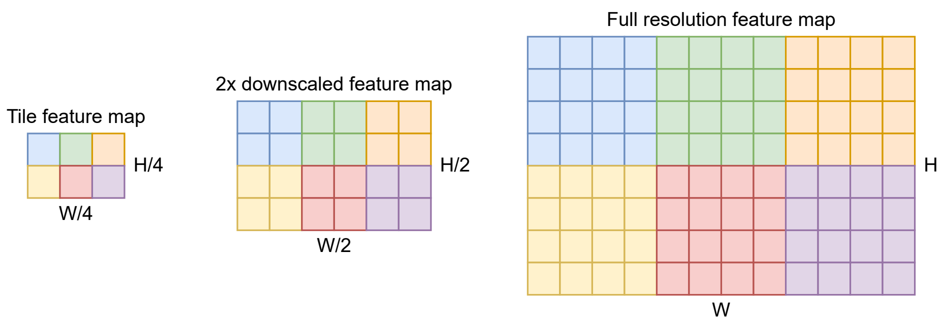


Figure 3.12: 2×2 tile features extracted from the 2x downscaled feature map, and the areas the tiles cover in the original feature map

As explained in Tankovich et al. (2020), we will use different strides when convolving over the left and right feature maps ϵ_l^L and ϵ_l^R (note the lack of the tilde symbol, meaning the feature maps from the feature extraction network). This is done to more effectively compute matching cost, while also keeping the resolution the same. For ϵ_l^L we use $(4, 4)$ strides (jumping four pixels ahead in x- and y-direction). For ϵ_l^R we use $(1, 4)$ strides (jumping four pixels ahead in the y-direction, but only one in the x-direction). This results in non-overlapping tiles from the left feature maps ϵ_l^L, and overlapping tiles (in the x-direction) from the right feature maps ϵ_l^R. Notice in figure 3.7, that the width of the left tile features are a quarter of the width of the right tile features due to the different strides.

3.4.4 Computing Matching Cost

Matching cost is used to evaluate how well two pixels match each other. There are many different matching cost functions, for example: *sum of absolute differences* (SAD), *mean-squared error* (MSE) and *mean absolute difference* (MD). We will use SAD as is also done by Tankovich et al. (2020). The sum of absolute differences between two pixels $a_{x,y}$ and $b_{x,y}$ at positions (x, y) in two different images is expressed mathematically as

$$\text{SAD}(a_{x,y}, b_{x,y}) = ||a_{x,y} - b_{x,y}||_1$$

where $||\cdot||_1$ is the L1 norm, or the Manhattan distance (absolute value). The higher the cost, the worse the match.

We want to calculate the matching cost between the newly created left and right tile feature maps $\tilde{e}_l^L$ and $\tilde{e}_l^R$ at all resolutions l. Because of the difference in the width of the two feature maps, we need to match every pixel in $\tilde{e}_l^L$, with every fourth pixel in the x-direction in $\tilde{e}_l^R$. For each pixel in $\tilde{e}_l^L$, we search for the best matching pixel in $\tilde{e}_l^R$ until a certain limit, the maximum disparity (as explained in section 2.1). The matching cost between every left and right tile feature map at every resolution l can be expressed as

$$\rho(l, x, y, d) = \left|\left|\tilde{e}_{l,x,y}^L - \tilde{e}_{l,4x-d,y}^R\right|\right|_1$$

Note the subscripts. In the left tile feature map we index every point (x, y), while in the right tile fature map we index every point $(4x - d, y)$. The disparity value d is all disparities $d \in [0, D]$ where D is the maximum disparity.

We will store the matching costs per pixel along the channel dimension, giving us a three-dimensional matching cost volume with size $D \times W \times H$. With the cost volume, we want to find the disparity d with the lowest matching cost for each tile feature. This is done using the winner-takes-all approach explained earlier. This disparity will become the tile disparity in the tile hypotheses introduced in section 3.4.2.

3.4.5 Initialization of tile hypotheses

The final step of the initialization module is to initialize the tile hypotheses for all pixels at all resolutions. As a quick reminder: a tile hypothesis comprises four parts: a tile disparity d, the slant gradients of the tile d_x and d_y, and a learnable feature descriptor $\mathbf{p}$. These are concatenated into a vector

$$[d, d_x, d_y, \mathbf{p}]$$

for all pixels in the feature map. From the matching cost function we already have the tile disparity d which we call d^{init}. The slant gradients d_x and d_y are initialized to 0. This is because we don't know anything about each tile's orientation yet. Therefore, all tiles start as front-facing tiles. The feature vector $\mathbf{p}$ is predicted by a simple convolutional layer consisting of a single 1×1 convolution (to not reduce the spatial dimensions) followed by a leaky ReLU activation. The output of this layer is $\mathbf{p}$. At the end of the initialization module, we have an initial tile hypothesis map for all resolutions $l \in [0, 4]$ with tile hypotheses $\mathbf{h}^{init}$ on the form

$$\mathbf{h}_l^{init} = [d^{init}, 0, 0, \mathbf{p}]$$

These tile hypotheses will be further refined and updated in the propagation phase.

3.5 Propagation Phase

The propagation module takes the created tile hypotheses $\mathbf{h}_l^{init}$ from the initialization module and iteratively refines them into increasingly more accurate disparity maps. The output of the final layer of the propagation module is the final prediction of the model. To get to the final prediction, the tile hypotheses needs to go through a series of warping and tile update steps which will be further explained in the following sections.

This module iterates over each tile hypothesis map starting at the coarsest resolution ($l = 0$), increasing the resolution at each iteration, ending at the finest resolution ($l = 4$). At each iteration it uses the refined tile hypothesis map from the previous iteration, along with the unrefined tile hypothesis map for the current one. Because of this, the first iteration will be slightly different, since we only have one single tile hypothesis map to refine. We refer to figure 3.14 for the single tile hypothesis map case, and figure 3.15 for the multiple tile hypothesis maps case.

3.5.1 Warping

There are three distinct steps during the propagation phase, starting with the warping step. This step will warp the right features ϵ_l^R (from the feature extraction network in the intialization module) into an approximated left feature map using the tile hypothesis map. As we explained in 3.4.5, a tile hypothesis comprise the tile disparity d and its slant gradients d_x and d_y. These represent the geometry of a 4×4 tile in the left feature map. If we upsample the tile geometry by 4, we get said 4×4 tile from the feature map ϵ_l^L at resolution l. To upsample the tile geometry (d, d_x, d_y) into a 4×4 tile, we use the equation of a plane. Just like Tankovich et al. (2020), We denote the resulting local disparity map as $\mathbf{d}'$. The formula for upsampling the tile disparity is as follows [Tankovich et al., 2020, p. 5]:

$$\mathbf{d}'_{i,j} = d + (i - c) \cdot d_x + (j - c) \cdot d_y$$

where c is the center of the tile and can be computed as $c = (t - 1)/2$ where t is the tile size, which in our case is 4. Therefore, $c = 1.5$. The local indices i and j are in the range [0, 3] and correspond to each index in the upsampled 4×4 disparity map. This means we calculate the disparity for each pixel (i, j) separately, based on the tile's disparity and slant gradients. The loss function (explained in section 3.6) takes into account a tile's geometry (d, d_x, d_y), meaning the model will learn these parameters over time which will result in more accurate local disparity maps.

For each upsampled 4×4 disparity map, we warp the corresponding area in the right feature map into the left feature map, using the assumption that a pixel in the left feature map is equal to the same pixel in the right feature map, but shifted by the disparity of that pixel. This is the same assumption we made in section 3.3 about the image inpainting module, where we also warp the right image into the left image using the relation

$$x_r = x_l - d$$

Warping all 4×4 tiles in the right feature map into the left feature map, we get a complete warped right feature map $\epsilon_l^{R'}$. The correctness of $\epsilon_l^{R'}$ will be based on the correctness of the upsampled disparity map. Thus, when the tile geometry improves, the warped right feature map improves as well.

3.5.2 Computing Matching Cost

In this step, the goal is to build a local cost volume around each tile hypothesis in the tile hypothesis map using the warped tiles from the previous step. For every 4×4 tile in the left feature map ϵ_l^L, we compute the matching cost of in the corresponding 4×4 tile in the warped right feature map $\epsilon_l^{R'}$. This is done by calculating the absolute differences between each pixel in the tiles, the same as in section 3.4.4. This results in a cost vector $\boldsymbol{\phi}$ [Tankovich et al., 2020, p. 5]

$$\boldsymbol{\phi}(\epsilon_l^L, \mathbf{d}') = [c_{0,0}, c_{0,1}, c_{0,2}, ..., c_{3,3}]$$

where $c_{i,j}$ is the cost computed between corresponding tiles in left feature map ϵ_l^L and the warped right feature map $\epsilon_l^{R'}$. More specifically

$$c_{i,j} = \left|\left|\epsilon^L_{l,4x+i,4y+j} - \epsilon^{R'}_{l,4x+i-\mathbf{d}'_{i,j},4y+j}\right|\right|_1$$

Doing this for all tile hypotheses in the tile hypothesis map, we get a cost vector $\boldsymbol{\phi}$ for all tile hypotheses. As stated in Tankovich et al. (2020), we displace each local disparity map $\mathbf{d}'$ with ± 1. This is done to "(...) build up a local cost volume which allows the network to refine the tile hypotheses effectively" [Tankovich et al., 2020, p. 5]. By displacing the disparity map, we get two extra cost vectors, creating a local cost volume for each tile hypothesis

$$\left[\boldsymbol{\phi}(\epsilon_l^L, \mathbf{d}' \text{ - } 1), \boldsymbol{\phi}(\epsilon_l^L, \mathbf{d}'), \boldsymbol{\phi}(\epsilon_l^L, \mathbf{d}' + 1)\right]$$

3.5.3 Tile Updates

In the tile update step we will create a CNN that takes the local cost volume for each tile hypothesis, along with the tile hypothesis map itself, and predicts a tile update $\boldsymbol{\Delta h}$ for each tile hypothesis, along with a confidence measure w that is used to judge the correctness of the tile hypothesis. This step works differently based on the number of tile hypothesis maps used as input. We refer to figure 3.14 for the single hypothesis map scenario, and figure 3.15 for the multiple hypothesis map scenario.

The tile update network will be implemented as a CNN with ResNet blocks. ResNet is a network proposed in the paper "Deep Residual Learning for Image Recognition" [He et al., 2015]. Its main contribution is the ResNet block (seen in figure 3.13). It utilizes residual connections (also known as skip connections or shortcut connections) between the input and output and element-wise adds them at the end of the block. The reason behind the ResNet block is this: consider two identical blocks, one with a residual connection and one without, like in figure 3.13. We want the weighted layers to learn the function $f(x)$. The left block in figure 3.13 has to learn the entire function $f(x)$, but because of its residual connection, the residual block (right block) only has to learn the residual function $f(x) - x$, hence its name *residual block*. This makes it easier to train deep neural networks as it reduces much of the complexity during training.

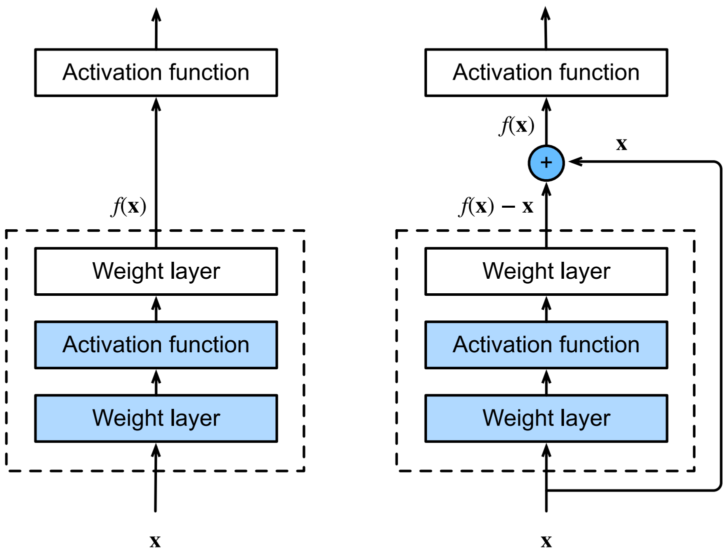


Figure 3.13: A block containing weighted layers to the left, a ResNet block to the right, [Zhang et al., 2020]

The tile update network will contain an input layer, a simple 1×1 convolution followed by a leaky ReLU activation. Then follows two residual blocks implemented the same way as the block in figure 3.13, using 3×3 convolutions and the leaky ReLU function. The last layer is a single 1×1 convolution to reduce the number of input channels to the desired amount of output channels. This will be how most tile update layers will be configured with the exception of three final tile updates at the end of the propagation phase.

As mentioned, tile updates will work differently based on the number of input tile hypothesis maps. It will start of with the coarsest tile hypothesis map with tile hypotheses that cover 64×64 tiles in the full resolution. In this part, we explain how tile updates will work in the first iteration when only one tile hypothesis map is available. After warping the right feature map ϵ_l^R into its warped counterpart $\epsilon_l^{R'}$ and the matching cost is computed, the tile hypothesis map along with the cost volume is passed into the tile update module. It will predict updates $\boldsymbol{\Delta h}$ for each tile hypothesis $\boldsymbol{h}$. The deltas are added together with its tile hypothesis, resulting in a updated tile hypothesis $\boldsymbol{h'}$, like so: $\boldsymbol{h'} = \boldsymbol{h} + \boldsymbol{\Delta h}$. As training goes on, the tile update network will learn to predict better deltas, due to the loss imposed on the confidence measure w which is also predicted by the tile update network. In the single hypothesis map scenario, w isn't used in any way other than in the loss function. The output of the tile update module is an updated tile hypothesis map. This updated tile hypothesis map will be used during the next iteration, propagating the tile update through all resolutions, thus giving the current module the name *propagation module*. Before we can pass the updated tile hypothesis to the next iteration, the map has to be upsampled to match the resolution of the next feature maps. The tile disparities d are upsampled using the plane equation from section 3.5.1. The slant gradients d_x and d_y and the feature description $\mathbf{p}$ will be upsampled using nearest-neighbor interpolation.

In the remaining iterations, we use two tile hypothesis maps and their matching cost volumes as input to the tile update network. From the previous iteration we already have an updated and upsampled tile hypothesis map $\boldsymbol{h_{coarse}}$, and from the current iteration we have a tile hypothesis map from the intialization module, $\boldsymbol{h_{init}}$. For both tile hypothesis maps, we run the warping step and the matching cost step, resulting in two cost volumes. Both the upsampled tile hypothesis map $\boldsymbol{h_{coarse}}$ and its cost volume, along with the current iteration's tile hypothesis map $\boldsymbol{h_{init}}$ and its cost volume will be passed into the tile update network. The tile update network will in the case with two tile hypothesis maps, predict deltas for both maps $\boldsymbol{\Delta h_{coarse}}$ and $\boldsymbol{\Delta h_{init}}$ along with confidence values w_{coarse} and w_{init} for each tile hypotheses in the two tile hypothesis maps. We then compare the confidence measures and choose the tile hypothesis with the highest confidence. For all the tile hypotheses where $w_{init} \leq w_{coarse}$, then the updated tile hypothesis map $\boldsymbol{h'}$ will use the coarse tile hypotheses and their deltas. If $w_{init} > w_{coarse}$ we do the opposite. Put differently:

$$\begin{aligned} \boldsymbol{h'} &= \boldsymbol{h_{init}} + \boldsymbol{\Delta h_{init}}, & w_{init} > w_{coarse} \\ \boldsymbol{h'} &= \boldsymbol{h_{coarse}} + \boldsymbol{\Delta h_{coarse}}, & w_{init} \leq w_{coarse} \end{aligned}$$

After the tile update, the updated tile hypothesis map $\boldsymbol{h'}$ will be upsampled the same way as previously with a factor of two. This process will then be repeated until the last tile hypothesis map has been updated.

3.5.4 Final Tile Updates

After the final iteration of the propagation, we will be left with updated tile hypothesis maps at different resolutions. Before we predict the final disparity map, we refine the updated tile hypothesis map with tile size 4×4. This means running a tile update module three times, upsampling the tile hypothesis map by a factor of two after each time until we reach a tile size of 1×1. Before these tile refinements, we do not warp or calculate matching costs for the tile hypothesis map, and will only calculate tile updates and confidence values based on the tile hypothesis map itself. We use the tile update module for the single tile hypothesis map scenario.

3.6 Loss Function

To train the model we implement the same loss function as is described in Tankovich et al. (2020). At its simplest, it is the sum of three different loss functions imposed on different parts of the network. This is to effectively train the different parts of the pipeline. The total loss L is defined as

$$L = \textstyle\sum_l L_l^{init} + L_l^{prop} + L_l^{slant} + L_l^{w}$$

for all resolutions l. Each part of the equation imposes a loss on four different parts of the model, initialization loss, propagation loss, slant loss, and tile update confidence loss. We will now go into more detail on each of these in the following sections.

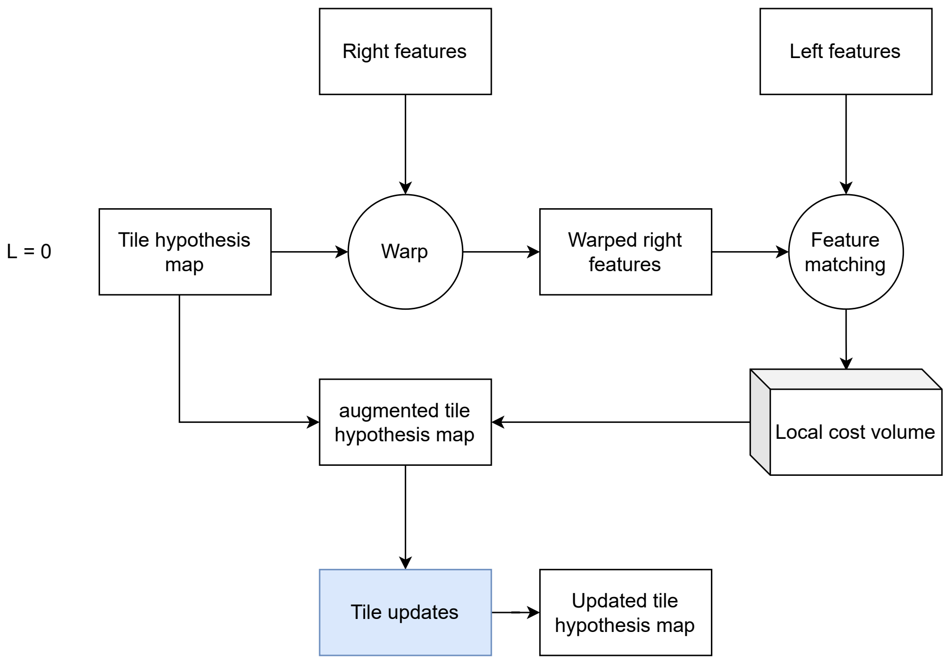


Figure 3.14: Propagation with single tile hypothesis map

3.6.1 Initialization Loss

To train the initialization module, we implement an $L1$ contrastive loss [Hadsell et al., 2006]. A contrastive loss aims to train the network to group neighbors together while pushing non-neighbors apart [Hadsell et al., 2006]. In this case, the function pulls disparities that cause a matching cost lower than a margin β towards a loss of zero, while pushing disparities with matching costs greater than the margin towards an infinite loss. This will force the feature extraction network in the initialization module to extract features with a matching cost smaller than the margin to minimize the loss. In Tankovich et al. (2020) the loss function is defined as:

$$L^{init}(d^{gt}, d^{nm}) = \psi\left(d^{gt}\right) + \max\left(\beta - \psi\left(d^{nm}\right), 0\right)$$

$\beta > 0$ is the margin. The function takes the ground-truth disparities d^{gt} and d^{nm} which is the non-matching disparity with the lowest cost, defined as

$$d^{nm} = \text{argmin}_{d \in [0,D] / \{d | d \in [d^{gt} - 1.5, d^{gt} + 1.5]\}} \rho(d)$$

In simpler terms, d^{nm} is the disparity in the range $[0, D]$, minus the

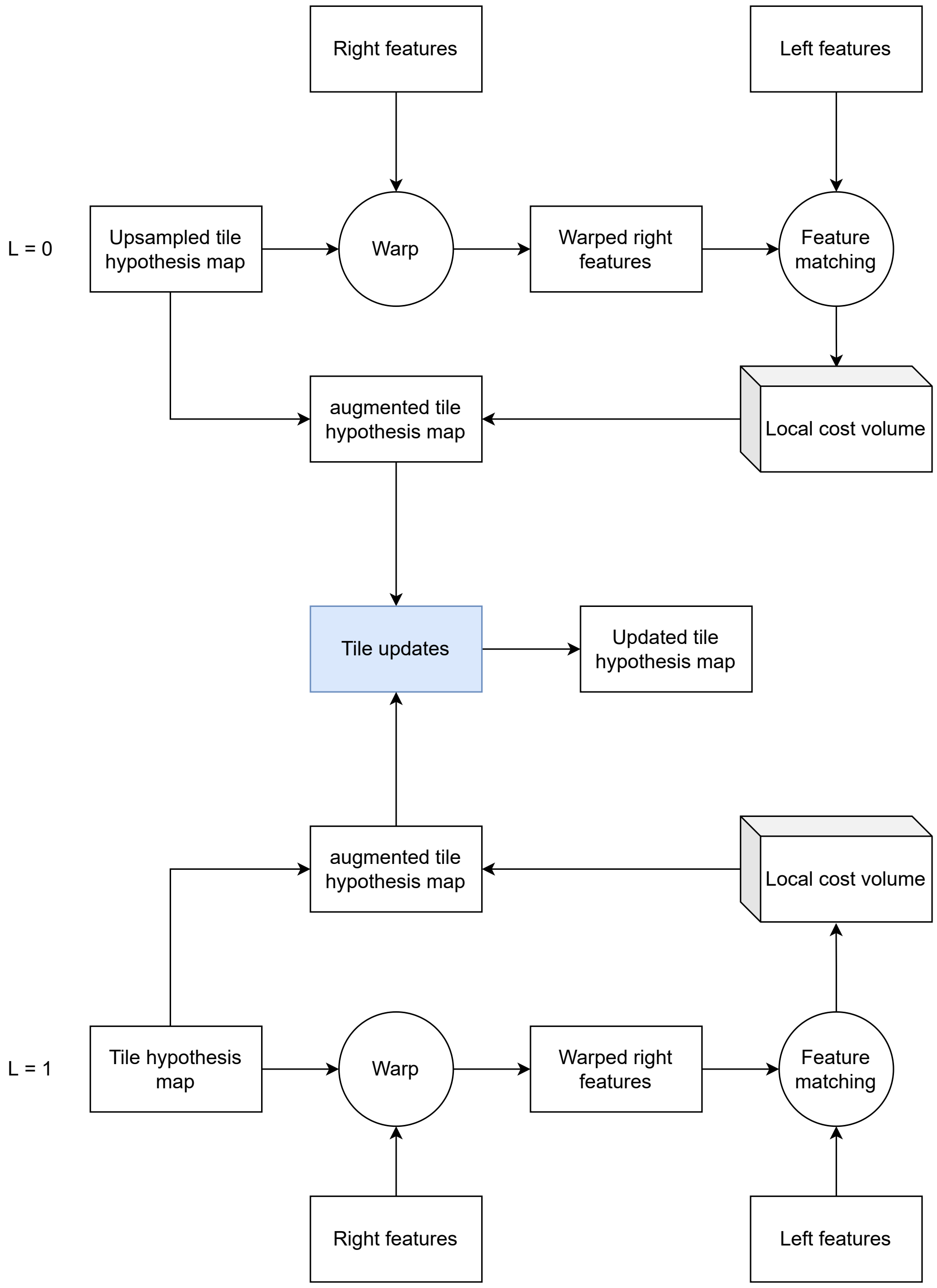


Figure 3.15: Tile update with multiple tile hypothesis maps

disparities within ± 1.5 of the ground-truth disparity d^{gt}, with the lowest cost ρ. Here, ρ is the cost volume calculated in the initialization phase (section 3.4.4). ψ is a function that calculates the matching cost for subpixel disparities. The subpixel cost is calculated as follows

$$\psi(d) = (d - \lfloor d \rfloor)\rho(\lfloor d \rfloor + 1) + (\lfloor d \rfloor + 1 - d)\rho(\lfloor d \rfloor)$$

ρ is the matching cost volume from the initialization phase and $\lfloor \cdot \rfloor$ is the floor operation. The reason behind calculating the matching cost of the subpixels in the image is because "Ground-truth disparities are given with subpixel precision, however matching in initialization happens with integer disparities." [Tankovich et al., 2020, p. 6]. Notice how the subpixels are isolated when subtracting the integer disparity $\lfloor d \rfloor$ from the floating-point disparity d.

This is done for all cost volumes at all resolutions l. Since the ground-truth disparities come in full resolution, we downsample them to the correct resolution using max-pooling with a 2×2 kernel and stride of two. This will extract the biggest disparity value in all non-overlapping 2×2 areas in the ground-truth disparity maps. This halves the resolution while keeping the most dominant disparity values.

3.6.2 Propagation Loss

In the propagation module, loss is imposed on three parts: the disparity predictions d, the slant prediction d_x and d_y, and the confidence measure w. This is done at all resolutions l, just like with the initialization loss. The only difference is that we upsample d, d_x, d_y, and w to full resolution instead of downsampling the ground-truth disparity maps. We upsample the disparities d with the plane function as described in section 3.5.1, while using nearest-neighbor upsampling to upsample d_x, d_y and w. Upsampling w with nearest-neighbor upsampling is an assumption we make, as there isn't a good explanation as to if and how w is upsampled in Tankovich et al. (2020). The loss imposed on the tile disparities d, which we call propagation loss from now on, is expressed as:

$$L^{prop}(d, d_x, d_y) = \rho(\min(\left| d^{diff} \right|, A), \alpha, c)$$

where $d^{diff} = d^{gt} - \hat{d}$ is the difference between the ground-truth disparity and the predicted disparity $\hat{d}$, also called the error. We also have three constants A, α and c. A is a threshold used to truncate $\left|d^{diff}\right|$, meaning if $\left|d^{diff}\right|$ is greater than A, it is cut of at A. In Tankovich et al. (2020), A is set to one. This means the loss function only takes into account the predicted disparities with a loss less than one. α and C are parameters passed to the function ρ below, which is an adaptive and general loss function that changes based on the values of α and c. The loss function was proposed by Barron (2017) in his paper "A General and Adaptive Robust Loss Function". In its simplest form ρ is defined as:

$$\rho(x, \alpha, c) = \frac{|\alpha - 2|}{\alpha}\left(\left(\frac{(x/c)^2}{|\alpha - 2|} + 1\right)^{\alpha/2} - 1\right)$$

We use the same values for α and c as in Tankovich et al. (2020), which is $\alpha = 0.9$ and $c = 0.1$. The function will therefore closely resemble a smooth $\mathcal{L}1$ loss, which we shall not go into the details on. We refer to Barron (2017) for examples on how the loss function acts with different values of α and c.

To train the slant gradient predictions d_x and d_y in the tile update layers, Tankovich et al. (2020) proposes the following loss function:

$$L^{slant}(d_x, d_y) = \left\| \begin{matrix} d_x^{gt} - d_x \\ d_y^{gt} - d_y \end{matrix} \right\|_1 \cdot X_{\left|d^{diff}\right| < B}$$

For both d_x and d_y we calculate the $\mathcal{L}1$ norm (Manhattan distance) between the ground-truth slant gradients d_x^{gt} and d_y^{gt}, and multiply it with the indicator function X

$$X_{\left|d^{diff}\right| < B} = \begin{cases} 0, & d^{diff} \geq B \\ 1, & d^{diff} < B \end{cases}$$

This means we only take into account the slant gradients where the difference $\left|d^{diff}\right|$ is less than the constant B. To generate the ground-truth slant gradients d_x^{gt} and d_y^{gt}, we use least-squares fitting to fit a plane in a 9×9 area around each pixel in the ground-truth disparity map. This is done as a preprocessing step to save time during training because finding the least-squares fit is a time-consuming operation.

The final loss is a simple function imposed on the confidence values w. It is defined as

$$L^w(w) = \max(1 - w, 0) \cdot X_{\left|d^{diff}\right| < C_1} + \max(w, 0) \cdot X_{\left|d^{diff}\right| > C_2}$$

This loss function will train the network to increase the confidence if the distance $\left|d^{diff}\right|$ is less than the threshold C_1, and decrease the confidence if the distance is greater than the threshold C_2.

As a final note, and as explained in 3.5.4, we will implement three final tile update layers that refine and predicts the final disparity map. All loss functions in the propagation module are applied to these with the same parameters α, c, B, C_1, and C_2. The only difference is that we remove the threshold A, setting it as $A = \infty$. This means we compute the loss of all disparities ($\left|d^{diff}\right|$ will never be greater than ∞).

Chapter 4

Implementation

In this section, we will go into the implementation details of this thesis. We start off by go through how we integrated the SceneFlow dataset into the training cycle using PyTorch, and explain the data augmentation used on the dataset. This is followed by the implementation details of the model, from the initialization phase to the propagation phase. We will also briefly go through our implementation of the inpainting network we discussed in the design chapter. Lastly, we will explain the process of plane fitting to calculate the ground-truth slant maps we needed for the loss function.

4.1 Data Preparation

Data preparation is one of the most important steps in getting an accurate model that generalizes well. In our experiments, we use the left-right image pairs of the SceneFlow dataset and two disparity maps computed by the SGM algorithm based on the stereo image pairs. We, therefore, have a variety of data defined in different ranges. The RGB images, when converted to PyTorch tensors are in the range [0, 1]. The disparity maps are in the range [-1, ∞). This means we need to do some data preparation before training since neural networks don't play well with a lot of different data in very different data ranges. We will in the subsequent subsections explain our dataset implementation, and how we plan to augment and preprocess the data before training.

4.1.1 Custom SceneFlow Dataset Implementation

PyTorch comes with many downloadable datasets which are already ready to use for training. This is not the case for SceneFlow which we intend to use during our experiments. This requires us to implement our own custom dataset using PyTorch. Fortunately, this is relatively simple to do. In PyTorch, a dataset is defined by a `Dataset` class. We define our custom dataset as a subclass of `Dataset`. The subclass needs to implement the two methods `__len__` and `__getitem__` as well as a constructor. These are methods called by a `DataLoader`, an object that iterates over the dataset, and batches data samples, making them ready for training. The `Dataset` class is mainly responsible for reading the dataset from disk into memory and performing data augmentation on the data if specified. This is done in the `__getitem__` method. The signature is shown below.

```
def __getitem__(self, i):
    pass
```

The only argument is an index value that is used to access the i-th sample in the dataset. We, therefore, need a way of indexing the dataset. We solve this by adding all file paths to a list in the constructor of the SceneFlow `Dataset` class. Due to the size of the SceneFlow dataset, there is no way of loading all left-right image pairs, along with their ground-truth disparity maps directly into memory. The next best solution is therefore to load their disk location in form of file paths into memory, and load the data samples into memory when they are needed by the `DataLoader`. The list of file paths can then be accessed by the index variable passed as an argument to the `__getitem__` method. The `__len__` method is self-explanatory, it is a simple method call that returns the size of the dataset and can be easily implemented simply by returning the length of the file path list.

The SceneFlow's "FlyingThings3D" dataset already comes split into a training and validation dataset. These are split into two different folders and we can therefore create two different SceneFlow `Dataset` objects, one for the training split and one for the validation split. This means we also need two separate `DataLoader` objects that iterates over each split separately. Each `DataLoader` batch data samples together into *mini-batches*. A mini-batch contains multiple data samples that will be processed by the neural network in parallel. This saves on training time but comes at the cost of using more memory. Memory management is important when training a neural network, as a GPU only has a limited amount of memory it can use to store the neural network and data samples. We, therefore, need to set the batch size to a size that the GPUs can handle. In our experiments, the batch size is set to 8. This means the neural network will process eight data samples in parallel.

4.1.2 Data Augmentation

We augment the data samples differently based on the dataset split. For both the training and validation split, we read the stereo RGB images from disk and convert them to PyTorch tensors using the `torchvision.transforms.ToTensor` class. This will convert the RGB images into tensors in the range [0, 1]. We also normalize the image tensors using the mean and standard deviation of the training set. The mean and standard deviation is computed per-channel over the entire training dataset. We do not include the validation dataset into the calculation, as it would leak information about the validation set into the training set. This is called *data leaking* and is bad because it would give our model information about the validation set. We of course want the training and validation sets completely separated from each other to get the most accurate results possible. The combined mean μ and standard deviation σ for the red, green and blue channel for the FlyingThings3D training dataset is

$$\mu = (0.4210, 0.4001, 0.3655)$$
$$\sigma = (0.1909, 0.1764, 0.1613)$$

Each RGB image is normalized per-channel using μ and σ above. This will scale all images to have a mean close to zero and standard deviation close to one. This is applied using the formula where x is the value to be normalized.

$$\bar{x} = \frac{x - \mu}{\sigma}$$

In PyTorch we can normalize a tensor per-channel using the `torchvision` library's `transforms.Normalize` class. It takes two lists in the constructor, one list containing the per-channel mean and one list containing the per-channel standard deviation. This normalization is done on both the training set and validation set.

For the training set only, we do a random crop of size 320×960 as is also done in Tankovich et al. (2020). The original size of the SceneFlow images is 540×960. Note that we refer to the height dimension first, meaning each image is 540 pixels in height and 960 pixels in width. We do this because this is how PyTorch expects the order of the image dimensions to be in. The random crop does two things: it introduces slightly new data samples every time during training, which is good for the generalization of the network. It also keeps the width and height dimensions even when they're downsampled during the feature extraction. The random crop is performed using the class `transforms.RandomCrop` from `torchvision`. This constructor expects the crop size.

We do not random crop the validation data, as we want our stereo depth estimation model to process complete images. For example, when estimating depth on an AR display, we want a disparity map of the whole scene, not only for a small crop of the scene. We do however run into a problem with the width and height dimensions being rounded down during the feature extraction. The rounding errors propagate through the network, causing errors and shape mismatches later on. To alleviate this, we zero-pad the images with a 36 high border at the top of the image. This is done using the `torch.nn.functional.pad()` function.

To compute the initial disparity maps we feed into our model, we use OpenCV's SGM implementation. `cv2.StereoSGBM_create()` initializes the algorithm, while the `compute()` method calculates the disparity maps. We mostly use the default parameters, with a slightly different `numDisparities` and `blockSize` attributes. We use the following parameters:

```
minDisparity=0,
numDisparities=128,
blockSize=5,
P1=0,
P2=0,
disp12MaxDiff=0,
preFilterCap=0,
uniquenessRatio=0,
speckleWindowSize=0,
speckleRange=0,
mode=cv2.StereoSGBM_MODE_SGBM,
```

We won't explain each parameter, for more information we refer to the OpenCV documentation [OpenCV, 2021].

The computed disparity maps are min-max normalized, i.e., transformed into the range [0, 1] to be in the same range as the RGB images. OpenCV's SGM implementation labels all pixels it couldn't calculate the disparity for, with a negative number. We want to preserve this, as it gives the model a notion on where occluded areas are located. We, therefore, create a bitmask identifying all valid disparities (disparities ≥ 0) and only normalize these.

We also apply random crop (for the training set) and the zero-padding (for the validation set) to the disparity maps to keep the dimensions equal with the RGB images.

4.2 Feature Extraction

As explained in the design chapter, the feature extraction network is a small U-Net with skip connections between layers. We also want to return the output of each decoding layer as they are used to extract tile features at different resolutions. The U-Net architecture is pretty straightforward to implement in PyTorch. We first needed to define the convolutions we are going to use in each layer in the encoder and decoder. For the encoder, we want a convolution that halves the resolution of the input. Tankovich et al. (2020) implements these layers with strided convolutions. As explained previously, the stride of a convolution is the number of pixels the kernel is shifted by when it convolves over the input. The default stride of a convolution is usually one, meaning the kernel shifts one pixel at a time. To halve the resolution of the input, Tankovich et al. (2020) uses a 2×2 kernel with stride 2. This will halve the resolution of the input as it computes the cross-correlation of all non-overlapping 2×2 areas in the input. We can verify this by using convolution arithmetics from Dumoulin and Visin (2016):

$$o = \left\lfloor \frac{i + 2p - k - (k-1)(d-1)}{s} \right\rfloor + 1$$

We do not change the padding p or the dilation d and keep them as 0 and 1 respectively. We can therefore simplify the formula

$$o = \left\lfloor \frac{i - k}{s} \right\rfloor + 1$$

If we set the kernel size $k = 2$ and stride $s = 2$ we get

$$\begin{aligned}
o &= \left\lfloor \frac{i-2}{2} \right\rfloor + 1 \\
o &= \left\lfloor \frac{i}{2} - \frac{2}{2} \right\rfloor + 1 \\
o &= \left\lfloor \frac{i}{2} \right\rfloor - 1 + 1 \\
o &= \left\lfloor \frac{i}{2} \right\rfloor
\end{aligned}$$

We observe that the output shape o will be half of the input shape i floored. Before The 2×2 strided convolution, we apply a 3×3 non-strided convolution as is done in Tankovich et al. (2020) to increase the complexity and learnable parameters of the feature extraction network. We do not want to reduce the spatial dimensions in this convolution, so we add padding of 1 to keep the input at full resolution. This can be verified using the formula above as well. Between the convolutions, and after the final convolution, we apply the leaky ReLU activation function, a variation of the regular ReLU function, but with a small slope coefficient for the negative numbers. This coefficient is set to 0.01 in our models. In PyTorch, the encoder layer is implemented like this:

```
def feat_encoder_conv(in_c, out_c):
    return nn.Sequential(
        nn.Conv2d(in_c, out_c, kernel_size=3, padding=1),
        nn.LeakyReLU(0.01),
        nn.Conv2d(out_c, out_c, kernel_size=2, stride=2),
        nn.LeakyReLU(0.01),
    )
```

`torch.nn.Sequential` is a class that sequentially executes all layers it contains when it is called. In this case, it first executes the first convolution, then the leaky ReLU, then the second convolution, and lastly the final leaky ReLU. In PyTorch, a convolution is defined using the `torch.nn.Conv2d` class. The mandatory parameters of this class are the number of input channels, output channels, and kernel size. Note that the number of input channels to the second convolution (`out_c`) needs to match the number of output channels of the first convolution. Failing to match these will result in an error. Also, notice the use of padding and stride.

For the decoding layers, we want to double the resolution of the input. This is done using transposed convolutions. A transposed convolution, as its name suggests, is the transposed version of a regular convolution. It is transposed in the sense that each element in the input map is multiplied with each element in a kernel that convolves over the output map. The output of the multiplication is added to each pixel of the output map. See figure 4.1 for a transposed convolution with a 3×3 kernel over a 2×2 input map. The kernel has a stride of 3 to more easily show what happens in a transposed convolution. With a non-strided transposed convolution, the kernel will overlap. In that case, the results of the multiplication will be added to the value that is already in the pixel. The transposed convolution is a way of upsampling an input map to a higher spatial resolution. We can also use interpolation to upsample feature maps, but a transposed convolution has the advantage of having a learnable kernel that is updated during training. This means the transposed convolution can weigh the input and choose which feature is the most important to include in the upsampling.

Tankovich et al. (2020) implements the decoder layer similarly to the encoder layer, with the only difference being the use of a transposed convolution instead of the double-strided regular convolution. We can

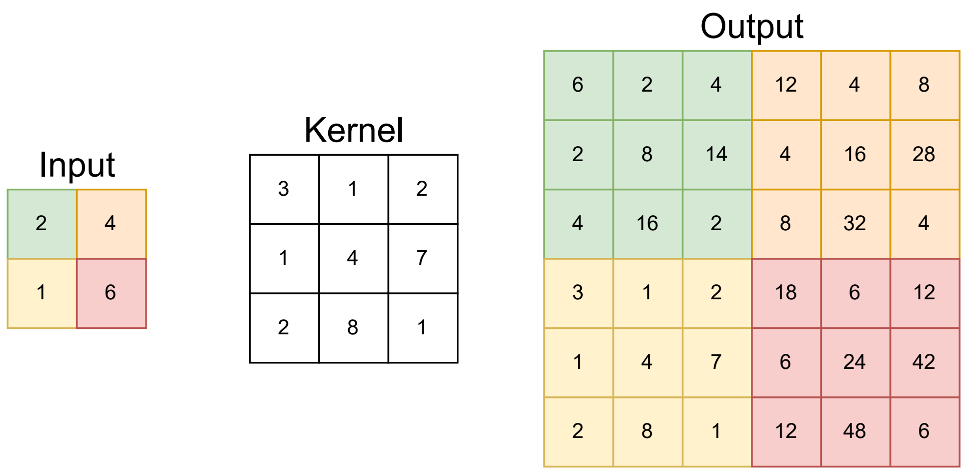


Figure 4.1: Example of transposed convolution

implement this in PyTorch like this:

```
def feat_decoder_conv(in_c, out_c):
    return nn.Sequential(
        nn.Conv2d(in_c, out_c, kernel_size=3, padding=1),
        nn.LeakyReLU(0.01),
        nn.ConvTranspose2d(out_c, out_c, kernel_size=2, stride=2),
        nn.LeakyReLU(0.01),
    )
```

The only difference is the use of `torch.nn.ConvTranspose2d`. Note that the transposed convolution uses the same arguments as the double-strided convolution in the encoder.

To create our feature extraction U-net using these layers, we need to implement a PyTorch `Module` like so:

```
class FeatureExtractor(nn.Module)
    def __init__(self, in_c):
        pass

    def forward(self, x):
        pass
```

In the constructor, we define each layer of the `Module` using the functions we defined previously. We define the encoder and decoder as two separate `ModuleDict` objects, dictionaries that map a `Module` to a key:

```
self.encoder = nn.ModuleDict()
self.decoder = nn.ModuleDict()
```

Consequently, we can define our layers for both the encoder and decoder like this:

```
self.encoder.add_module("conv0", feat_encoder_conv(in_c, 16))
...

self.decoder.add_module("conv0", feat_decoder_conv(32, 32))
...
```

The `forward()` method is where we implement the logic for the forward propagation through the feature extraction network. Since we have five encoding layers that take the output from the previous layer as input, we program the encoder's forward pass like this:

```
def forward(self, x):
    d0 = self.encoder.conv0(x)
    d1 = self.encoder.conv1(d0)
    ...
```

The decoder is a bit more complicated, as we need to implement the skip connections between the encoder layers and decoder layers. We can do this by concatenating the output from the same layer in the encoder, with the input from the previous layer in the decoder. This requires the tensors to have the same height and width. They are concatenated in the channel dimension using `torch.cat((x, y), dim=1)`. It looks like this:

```
...
u3 = self.decoder.conv3(torch.cat((u2, d1), dim=1))
u4 = self.decoder.conv4(torch.cat((u3, d0), dim=1))
```

`self.decoder.conv4` is the last layer in the decoder and takes the concatenation of `u3`, the output of the previous decoder layer, and `d0`, the output of the first encoder layer. The output of the `forward()` function is a list containing all output maps `[u0, u1, u2, u3, u4]` which are all of different resolutions.

Table 4.1 shows the complete architecture of the feature extraction network with all five encoder and decoder layers. It shows the input of each layer, along with its output. It also shows the number of input and output channels ("C in" and "C out"), kernel size (K), padding (P), and stride (S). It also specifies the activation function used. Note that the kernel, padding, and stride are defined by two numbers separated by commas. This is to distinguish between the two convolutions a single layer is comprised of. The || operator is the concatenation operator. Also note that the number of input channels of the first encoder layer varies. This is because we implement two models, an experimental model that accepts an additional channel due to the extra disparity map, and a control model that only accepts three-channel RGB images. The number of input and output channels are the same as is specified in Tankovich et al. (2020). The same feature extraction network is used to extract features from the left and right images separately.

Layer name	**Input**	**Output**	**C in**	**C out**	**K**	**P**	**S**	**Activation**
enc_conv0	x	d0	4/3	16	3,2	1,0	1,2	Leaky ReLU
enc_conv1	d0	d1	16	16	3,2	1,0	1,2	Leaky ReLU
enc_conv2	d1	d2	16	24	3,2	1,0	1,2	Leaky ReLU
enc_conv3	d2	d3	24	24	3,2	1,0	1,2	Leaky ReLU
enc_conv4	d3	d4	24	32	3,2	1,0	1,2	Leaky ReLU
dec_conv0	d4	u0	32	32	3,2	1,0	1,2	Leaky ReLU
dec_conv1	u0 || d3	u1	56	24	3,2	1,0	1,2	Leaky ReLU
dec_conv2	u1 || d2	u2	48	24	3,2	1,0	1,2	Leaky ReLU
dec_conv3	u2 || d1	u3	40	16	3,2	1,0	1,2	Leaky ReLU
dec_conv4	u3 || d0	u4	32	16	3,2	1,0	1,2	Leaky ReLU

Table 4.1: Arcitecture of feature extraction U-net.

4.3 Initialization Module

The initialization module does the tile feature extraction and matching cost computation to initialize our tile hypotheses. In this section we will explain how these were implemented, starting with the tile feature extraction.

4.3.1 Tile Feature Extraction

In the tile feature extraction phase, we extract tile features from the output of the feature extraction network, as described in the design chapter. This is done using a 4×4 convolution with different strides depending on if we extract tile features from the left, or right feature maps. In the left feature map, we use 4×4 strides, moving the kernel every fourth pixel in both height and width directions. This results in a tile feature map that is 1/4 the original feature map resolution in both width and height. For the right feature maps, we change the stride to 4×1, meaning we convolve over every pixel in the width-direction, while skipping every fourth pixel in the y-direction. We use the same convolution to extract tiles from both the left and right feature maps, so we need to change the strides in between. We can do this simply by changing the `stride` attribute of the convolution: `extractor.stride = (4, 4)` and `extractor.stride = (4, 1)`. Referring to table 4.1, we can see that each decoder layer has a different number of output channels. Due to this, we need to define multiple tile feature extraction convolutions with input dimensions that match the output dimension sizes of the decoder layers. We use a `torch.nn.ModuleList` and append all convolutions to that list using the `append()` function. The convolution in itself is a simple convolution with a 4×4 kernel and one output channel. The stride is set as we discussed earlier. The tile feature maps extracted are all appended to lists.

4.3.2 Matching Cost Computation

When the tile features are extracted, we iterate over the lists containing the tile feature maps and compute the matching costs between each tile feature map in increasing resolution order, from the lowest resolution to the highest resolution. Between each left tile feature map and right tile feature map, referred to as `l_tile` and `r_tile` from now on, we create a cost volume as a PyTorch tensor:

```
cost_volume = torch.zeros(
    l_tile.shape[0],
    self.max_disp,
    l_tile.shape[2],
    l_tile.shape[3],
    device=l_tile.device,
)
```

`torch.zeros()` creates a zero-filled tensor with the shape specified by the arguments. In this case, it will mostly have the same dimensions as `l_tile`, except for the channel dimension where we expand it to match the maximum disparity. Also, note that we specify the device we want to create the tensor on. We create it on the same device as `l_tile`, which would be the GPU. Doing tensor operations on tensors on different devices will lead to errors.

The cost volume is filled with the costs calculated between every element in `l_tile` and `r_tile`. To do this we need to remember that `r_tile` is four times wider than `l_tile`, therefore we need to compute the cost of between every pixel in `l_tile`, and every fourth pixel in `r_tile`. We can do this by using Python list slicing, which is a way of accessing lists, or in this case tensors. The cost computation is done like so:

```
for d in range(self.max_disp):
    ...
    costs = torch.abs(l_tile - r_tile[:, :, :,
                                  self.max_disp - 1 - d :-d : 4])
    ...
```

This line of code calculates the absolute difference element-wise between `l_tile` and `r_tile`, using the formula from section 3.4.4. Notice how we index `r_tile` to keep within the disparity range between zero and the maximum disparity, while skipping every fourth pixel. The `r_tile` tensor is multi-dimensional with the shape (B, C, H, W). Since we only care about matching in the width dimension W, we specify that we want all elements over the batch dimension B, the channel dimension C and height dimension H by using colons. Empty colons select the whole range within that dimension. We use `l_tile` as the reference image and search for the matching pixel in `r_tile` using the relation

$$x_l = x_r + d$$

where d is iterated through the range [0, `self.max_disp`). `self.max_disp` is set to 320 for our experiments. This means when x_l is less than the maximum disparity, i.e., for the leftmost `self.max_disp` pixels in `l_tile`, we are out of bounds in `r_tile`. To prevent this, we zero-pad `r_tile` with `self.max_disp - 1` on the left-hand side. The matching cost of each pixel at all disparities d is stored in the cost volume at channel index `d`

```
cost_volume[:, d, :, :] = costs
```

When the matching cost has been completed for all disparities, we want to find the disparities with the lowest matching cost. This is done using `torch.min()` on the cost volume over the channel dimension. This returns both the indices and values with the lowest cost. The indices are used as the tile disparity for each tile hypothesis, while the values are used in the tile feature descriptor.

4.3.3 Tile Hypothesis Initialization

Using the tile disparities and lowest matching costs, we construct all initial tile hypotheses. We reiterate the definition:

$$\boldsymbol{h}^{init} = \left[d^{init}, d_x, d_y, \mathbf{p}^{init}\right]$$

d_x and d_y are initialized to 0. The tile feature descriptor $\mathbf{p}^{init}$ is determined by a simple 1×1 convolution followed by a leaky ReLU activation function. We also need one tile feature descriptor for each resolution, since the number of input channels will vary. The number of output channels for all descriptors is set to 13, as specified in Tankovich et al. (2020). The feature descriptor takes the left tile feature maps `l_tile` from the previous section as well as the lowest matching costs computed by `torch.min()`. They are concatenated along the channel dimension before being fed to the feature descriptor. The tile disparities from the previous section, the slant gradients d_x and d_y, and the output of the feature descriptor are concatenated together along the channel dimension. This results in a tile hypothesis map for each resolution.

4.4 Propagation Module

The propagation module iterates over all the tile hypothesis maps as well as the left and right feature maps from the feature extraction network. In this section, we refer to the tile hypothesis map as `hyp`, and the left and right feature maps as `l_feat` and `r_feat`. As explained in the design chapter, there is a slight difference in the first iteration, versus the rest of the iterations. In the first iteration, we only have access to the lowest resolution tile hypothesis map and feature maps. We therefore go through both the single hypothesis case, and the multiple hypotheses case. First, we shall describe the implementation of the warping and cost calculation function we use to warp `r_feat` into `l_feat` using `hyp` and calculate the matching cost between the warped `r_feat` and `l_feat`.

4.4.1 Warping and Matching Cost Calculation

To warp `r_feat` into `l_feat`, we need to upsample `hyp` such that their width and height dimensions match. `hyp` is one-fourth the resolution of the feature maps `l_feat` and `r_feat`, because each tile hypothesis represents a 4×4 tile in the feature maps. We upsample `hyp` to the same resolution disparity map by using the plane equation

$$\boldsymbol{d'}_{i,j} = d + (i - 1.5)d_x + (j - 1.5)d_y$$

We implement this with a double for-loop that iterates over the local tile indices i and j in the range [0, 4). This is done for all tile hypotheses in `hyp`. When we upsample a tile hypothesis, we concatenate the local disparity map to a tensor that contains all 4×4 local disparity map of `hyp`. We then use `torch.nn.functional.pixel_shuffle()` to shuffle all the 4×4 local disparity maps from the channel dimension and tile them along the width and height dimensions. This way we get a complete disparity map containing all 4×4 local disparity maps. With the upsampled disparity map, we warp r_feat into l_feat. After warping r_feat into a new feature map r_warped, we compute the matching cost between l_feat and r_warped using the sum of absolute differences.

As is stated in the HITNet paper, "In order for the network to iteratively increase the accuracy of the disparity predictions, we provide the network a local cost volume in a narrow band (±1 disparity) around the planar patch using in-network image warping allowing the network to minimize image dissimilarity" [Tankovich et al., 2020, p. 2]. What this means is that we shift the tile disparities by ±1 before we upsample the tiles, and warp r_feat. We, therefore, need to upsample and warp three times. We do this in a for-loop like so:

```python
for delta_d in range(-1, 2):
    d = hyp[:, 0, ...] + delta_d
    dx = hyp[:, 1, ...]
    dy = hyp[:, 2, ...]

    # upsample and warp
    ...
```

At the end of every iteration, we append the cost to a list, that is consequently returned from the function. The list of costs is used by the tile update network to predict tile updates.

4.4.2 Tile Updates

A tile update layer comprises an input layer, several residual blocks, and a final layer. The first layer was implemented as a 1×1 convolution followed by a leaky ReLU with a slope coefficient of 0.01. Then follows two residual blocks. Each residual block comprises three 3×3 convolutions with padding and dilation equal to one. Each convolution is followed by a leaky ReLU function, both with slope coefficients of 0.01. There is no activation function following the final convolution. A more detailed description of the residual block layers follows in table 4.2. Note that the dilation factor (D) is listed.

Layer name	Input	Output	C in	C out	K	P	S	D	Activation
res_conv0	x	out0	32	32	3	1	1	1	Leaky ReLU
res_conv1	out0	out1	32	32	3	1	1	1	Leaky ReLU
res_conv2	out1	out2	32	32	3	1	1	1	-

Table 4.2: Arcitechture of a single residual block.

Table 4.3 shows how the tile update network looks for the first iteration, i.e., when only one tile hypothesis map is available. The shaded rows in the table are two residual blocks of the type in table 4.2.

Layer name	Input	Output	C in	C out	K	P	S	D	Activation
first_layer	init_hyp	out1	64	32	1	0	1	1	Leaky ReLU
res_blk0	out1	out2	32	32	-				
res_blk1	out2	out3	32	32					
last_layer	out3	update	32	17	1	0	1	1	-

Table 4.3: Architecture of tile update network using single tile hypothesis map as input.

Note that the number of output channels of the last layer is 17. The tile update network's job is to predict deltas for the tile hypotheses, along with a confidence value that expresses how certain the network is that the predicted deltas are correct. A single tile hypothesis has a size of 16 in the channel dimension, therefore the first 16 channels of `last_layer` are the deltas, while the last channel contains the confidence values. The deltas are added together with the input tile hypothesis map `init_hyp`. The updated `init_hyp` is returned as it will be used for the next iteration. Before we start on the next iteration, however, we need to upsample `init_hyp` by a factor of 2. We upsample the tile disparities with the plane equation, while using `torch.nn.functional.interpolate()` to upsample the slant gradients d_x and d_y and the feature description $\mathbf{p}$ with nearest-neighbor interpolation.

In the remaining iterations, we use two different tile hypothesis maps, the one from the previous iteration called `refined_hyp`, and the unrefined tile hypothesis map from the current iteration called `init_hyp`, created in the initialization module. We've provided the architecture of these tile update networks in table 4.4. The only difference from the architecture in table 4.3

is the number of input channels of the first layer, and output channels of the last layer. This is to account for the extra tile hypothesis map used as input, as we want to predict deltas and confidence scores for both tile hypothesis maps. We do this to choose the tile hypotheses and their deltas with the highest confidence score between `init_hyp` and `refined_hyp`. This has been explained in the design chapter. Once the refined tile hypothesis map has been selected, and the deltas have been applied, we return the new refined tile hypothesis map, upsample it, and pass it on to the next iteration. This is done until all tile hypothesis maps have been refined.

Layer name	Input	Output	C in	C out	K	P	S	D	Activation
first_layer	init_hyp || refined_hyp	out1	128	32	1	0	1	1	Leaky ReLU
res_blk0	out1	out2	32	32	-				
res_blk1	out2	out3	32	32					
last_layer	out3	update	32	34	1	0	1	1	-

Table 4.4: Architecture of residual block using two tile hypothesis maps as input. Note the difference in the number of input and output channels.

4.4.3 Final Tile Updates

We also need to implement the three final tile updates that are applied to the refined 4×4 downsampled tile hypothesis map (i.e., the refined tile hypothesis map that contains tile hypotheses that cover 4×4 tiles in the original resolution). We call these post tile updates. For these tile updates, we include extra residual blocks with dilated convolutions. Table 4.5 shows how these residual blocks are implemented. Note the use of dilatons. To predict the tile updates, we use the tile update network defined in table 4.3. The difference is that we use six of the residual blocks defined in table 4.5 instead. The third and final post tile update, predicts the final disparity map and slant map.

Before going into the implementation details of the loss function, we refer to figure 4.2. The figure illustrates an enlarged 64×64 tile, to give a better understanding of how the tiles are refined during the propagation stage. In the example, we can see artifacts caused by smaller and smaller tile sizes. With the smallest tile being 4×4. The more these tiles are refined, the more accurate the disparity map becomes.

Layer name	Input	Output	C in	C out	K	P	S	D	Activation
res_conv0	x	out0	32	32	3	1	1	1	Leaky ReLU
res_conv1	out0	out1	32	32	3	1	1	2	Leaky ReLU
res_conv2	out1	out2	32	32	3	1	1	4	Leaky ReLU
res_conv3	out2	out3	32	32	3	1	1	8	Leaky ReLU
res_conv4	out3	out4	32	32	3	1	1	1	Leaky ReLU
res_conv5	out4	out5	32	32	3	1	1	1	-

Table 4.5: Architecture of residual blocks used for last tile update layers.

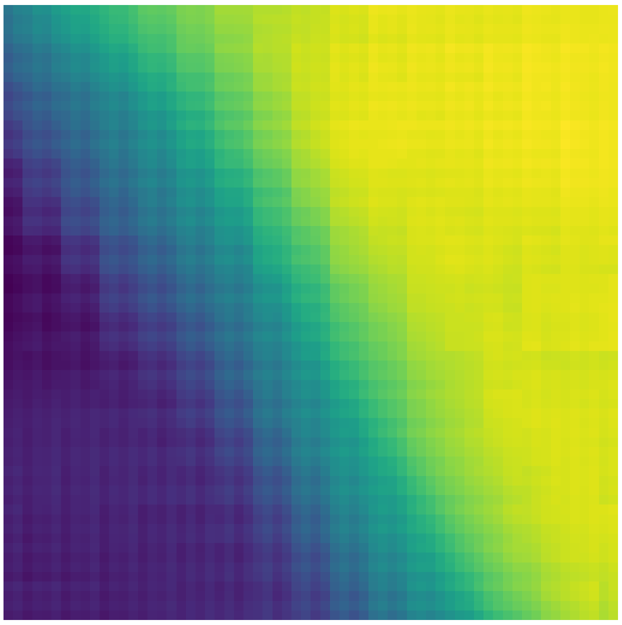


Figure 4.2: A 64 $\times$ 64 tile.

4.5 Loss Function

The loss function takes the cost volumes computed in the initialization phase, the tile geometry (d, d_x, d_y) refined throughout the propagation phase, and the confidence scores predicted by the tile update networks as input. We've implemented the loss function in two parts, the initialization loss that uses the initialization cost volumes, and the propagation loss that use the tile geometry and the confidence scores. We've stuck closely to Tankovich et al. (2020), using max-pooling to downsample the ground-truth disparity maps in the initialization loss, and upsampling the tile geometry using the plane equation for the tile disparity, and nearest-neighbor upsampling for the slant gradients. A small uncertainty however

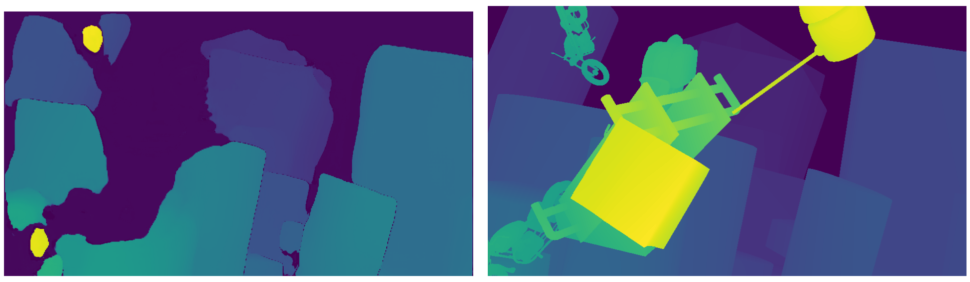

(a) Predicted disparity map (b) Ground-truth

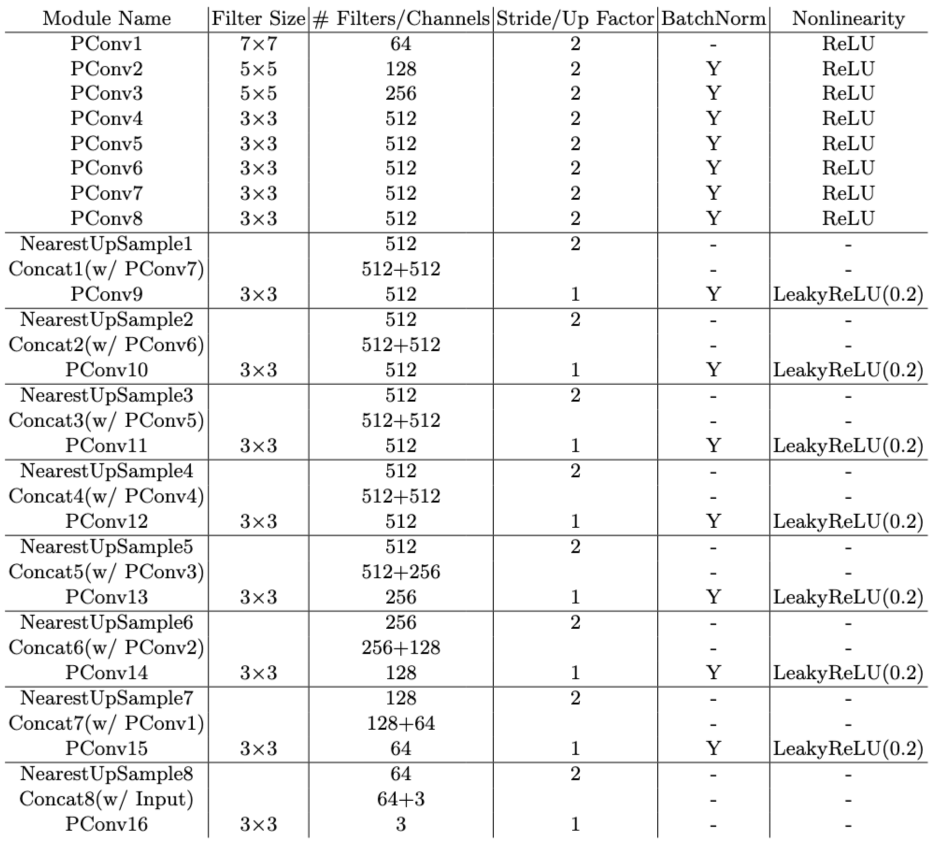
Figure 4.3: Results of model with error in propagation loss implementation.

is if the confidence score maps should be upsampled using nearest-neighbor upsampling or not. We decided to do so, although not implicitly stated in the paper.

The loss function has a lot of parameters. We used the same parameters as in Tankovich et al. (2020) as they seemed to work for their model. The parameters in the initialization loss are set to $\alpha = 0.9$ and $c = 0.1$, and the parameters for the propagation loss are $A = B = C_1 = 1$ and $C_2 = 1.5$. During the implementation of the propagation loss, we encountered a small but crucial mistake. In the HITNet paper, it is stated: "For the last several levels, when only a single hypotheses is available, loss is applied to all pixels ($A = \infty$)" [Tankovich et al., 2020, p. 6]. For the last three tile update layers, they remove the truncation A (i.e., set $A = \infty$). During the initial implementation of the propagation loss, we glossed over this information. As we tested our network, we noticed that it had a hard time predicting disparities for the foreground objects. As illustrated in figure 4.3, we can see that the predicted disparity map lacks disparities for the foreground objects. This was due to the truncation threshold A being applied to the last three refinement layers. This prevented the model from learning to refine the disparities of foreground objects. Fixing this error results in much better disparity maps.

4.6 Image Inpainting

The idea behind image inpainting was to fill in missing pixels of a warped image. After warping the left image into the right image using the left image's disparity map, we would apply the image inpainting module proposed by Liu et al. (2018) to recreate the rest. The model was implemented by following the architecture in figure 4.4. The network uses partial convolutions during the eight first layers. We used their official implementation of the convolution found on Github[1]. We also trained the network using their custom loss function which is found in the same Github repository. We will not go into the details of partial convolutions, as image inpainting is not a part of this thesis' scope. Instead we refer to the paper Liu et al. (2018).



Module Name	Filter Size	# Filters/Channels	Stride/Up Factor	BatchNorm	Nonlinearity
PConv1	7×7	64	2	-	ReLU
PConv2	5×5	128	2	Y	ReLU
PConv3	5×5	256	2	Y	ReLU
PConv4	3×3	512	2	Y	ReLU
PConv5	3×3	512	2	Y	ReLU
PConv6	3×3	512	2	Y	ReLU
PConv7	3×3	512	2	Y	ReLU
PConv8	3×3	512	2	Y	ReLU
NearestUpSample1		512	2	-	-
Concat1(w/ PConv7)		512+512		-	-
PConv9	3×3	512	1	Y	LeakyReLU(0.2)
NearestUpSample2		512	2	-	-
Concat2(w/ PConv6)		512+512		-	-
PConv10	3×3	512	1	Y	LeakyReLU(0.2)
NearestUpSample3		512	2	-	-
Concat3(w/ PConv5)		512+512		-	-
PConv11	3×3	512	1	Y	LeakyReLU(0.2)
NearestUpSample4		512	2	-	-
Concat4(w/ PConv4)		512+512		-	-
PConv12	3×3	512	1	Y	LeakyReLU(0.2)
NearestUpSample5		512	2	-	-
Concat5(w/ PConv3)		512+256		-	-
PConv13	3×3	256	1	Y	LeakyReLU(0.2)
NearestUpSample6		256	2	-	-
Concat6(w/ PConv2)		256+128		-	-
PConv14	3×3	128	1	Y	LeakyReLU(0.2)
NearestUpSample7		128	2	-	-
Concat7(w/ PConv1)		128+64		-	-
PConv15	3×3	64	1	Y	LeakyReLU(0.2)
NearestUpSample8		64	2	-	-
Concat8(w/ Input)		64+3		-	-
PConv16	3×3	3	1	-	-

Figure 4.4: Architecture of image inpainting model [Liu et al., 2018, p. 18]

We also want to go more in-depth on how we implemented the warping function to warp the left image into the right image. The left image can be indexed using a two-dimensional array with an x and y index for the rows (height) and columns (width) respectively. Here $x \in [0, H)$ and $y \in [0, W)$ where H and W is the height and width of the image. We can find the right image's pixels in the left image by offsetting the x indices with the disparity values d of each pixel. This way we can recreate the right image

[1]https://github.com/NVIDIA/partialconv

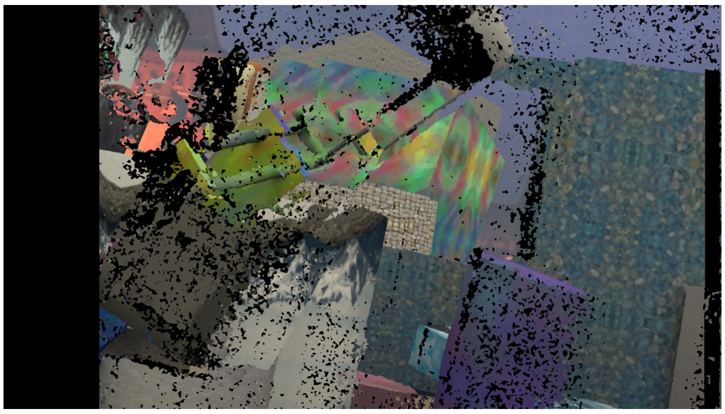


Figure 4.5: Warped image

by filling in every pixel (x, y) in the right image, with the pixel $(x + d, y)$ in the left image. There is one problem when doing this, that being the invalid disparities of the disparity map. The SGM implementation in OpenCV labels all pixels with invalid disparities with a negative number. We, therefore, need to create a bit-mask from the disparity map, that is true for all pixels with valid disparities (with $d \geq 0$), and false for all pixels with invalid disparities (with $d < 0$). This way we can fill in the pixels that have a valid disparity while keeping the invalid pixels empty. We can clearly see the bit-mask, or rather the inverse of the bit-mask, by looking at all the black holes in figure 4.5. These black holes would thereafter be filled in by the inpainting network.

We trained the image inpainting network separately from the HITNet model, using the right image as ground-truth. The results of this module will be presented in chapter 5.

4.7 Plane Fitting

A crucial part of the HITNet model is the ability to predict the slants for each tile. As a reminder, a tile is represented by the tile disparity d, and the slant gradients d_x and d_y. The loss function uses the slant gradients to impose a loss on them to allow the model to predict better slant gradients. Because of their use in the loss function, we need ground-truth slant maps to calculate the cost of the predicted slant maps. As mentioned in chapter 3, we do as is described in Tankovich et al. (2020), fitting a $9{\times}9$ plane around each pixel in every ground-truth disparity map. Naturally, this has to be done before training due to the computational cost of fitting a plane around every pixel in every ground-truth disparity map. The main idea is to iterate over all ground-truth disparity maps and convert them to their real three-dimensional coordinates. As it turns out, OpenCV has a function for exactly this, `reprojectImageTo3D()`, that takes the disparity map as input along with the camera's intrinsic matrix, and projects each pixel in the disparity map to their real-world coordinate representation. This allows us to convert a one-dimensional disparity map into a three-dimensional point cloud with coordinates (x, y, z) for each pixel. The function needs to know the camera's focal length and other specifications, which are provided in the *camera intrinsic matrix*. This matrix contains information about the camera, such as its focal length and the principal point. The intrinsic matrix accepted by `reprojectImageTo3D()` looks like this

$$\begin{pmatrix} f_x & 0 & c_x \\ 0 & f_x & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

Here, f_x and f_y are the focal lengths in pixel units, and (c_x, c_y) is the principal point. The principal point is the center point of the image plane, the plane that is captured by the camera. With the SceneFlow dataset, Mayer et al. has provided the camera intrinsic matrix for their cameras[2]. This matrix is shown below.

$$\begin{pmatrix} 1050.0 & 0.0 & 479.5 \\ 0.0 & 1050.0 & 269.5 \\ 0.0 & 0.0 & 1.0 \end{pmatrix}$$

When the ground-truth disparity map has been converted into its three-dimensional representation, we then needed to divide each 3D map into 9×9 areas that we could fit the planes too. For this part we used PyTorch's `torch.unfold()` function. This function takes a tensor as input and unfolds the tensor into N×M windows, the size of which is specified by a parameter. We want to center a 9×9 window around every point. To achieve this, we first zero-pad the 3D point cloud with a border of 4 pixels.

[2]The camera intrinsics matrix can be found on their website: https://lmb.informatik.uni-freiburg.de/resources/datasets/SceneFlowDatasets.en.html

This is done as we want each 9×9 area to be centered around every pixel. Without the zero padding, the four outermost edge points would not be included.

For each 9×9 window, we want to find the least-squares fit of a plane. PyTorch has a function that does this for us, `torch.linalg.lstsq()`. Least squares are often used in data fitting as it minimizes the squares of the residuals. A residual is the error $\hat{y} - y$ between the predicted point $\hat{y}$ and the actual point y. `torch.linalg.lstsq()` tries to fit a plane such that the squares of the residual for all points are as small as possible. It takes as input two tensors A and $\boldsymbol{b}$, which represents the same matrices in the mathematical equation of the least-squares problem

$$A\boldsymbol{x} = \boldsymbol{b}$$

`torch.linalg.lstsq()` minimizes the residuals $||A\boldsymbol{x} - \boldsymbol{b}||_F^2$ where $||\cdot||_F$ is the Frobenius norm, or the matrix norm. It finds the $\boldsymbol{x}$ values that results in the least residual cost. This solution is the gradients, or the normal vector, of the plane. A is a matrix containing all x and y coordinates in the point cloud as well as a third column of ones. The vector $\boldsymbol{b}$ contains the z coordinates of the point cloud. Passing these as arguments to the least-squares function, we get a solution (x, y, z). We only want the x and y components, as the HITNet model is only dealing with 2-dimensional tiles, making the z coordinate redundant. Doing this for all 9×9 windows of a single disparity map results in a ground-truth slant gradient map. See 3.11, for an example of a ground-truth slant map. The brighter the pixel, the more slanted the surface is.

4.8 Training Details

We trained our models using the Adam optimizer algorithm with an initial learning rate of $4e^{-4}$ and weight decay of $1e^{-6}$. The learning rate is in accordance with Tankovich et al. (2020) who also uses the Adam optimizer with the same learning rate. Tankovich et al. (2020) also uses learning rate scheduling to decrease the learning rate as it gets further into the training cycle. Learning rate scheduling is used to decay the learning rate during training. This can be done in a lot of different ways, but in our case, we use

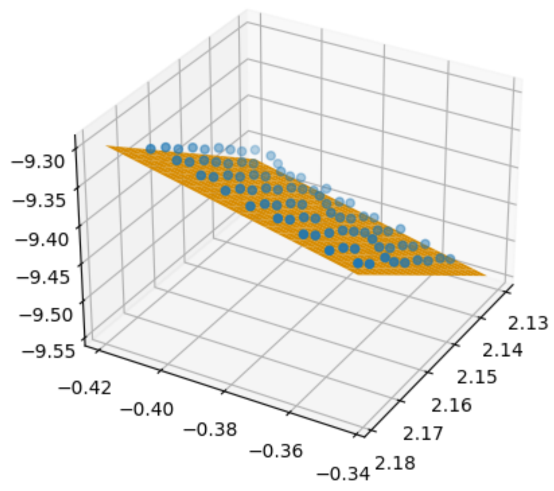


Figure 4.6: A plane fitted to a 9×9 point cloud using PyTorch's least squares algorithm. The plane isn't exactly in the correct position, but we only care about the orientation of the plane, which seems correct.

multi-step learning rate scheduling. This is an algorithm that decays the learning rate by some factor at certain epochs. We multiply the learning rate by a factor of 0.1 at the 50th, 75th, and 90th epochs. This means the learning rate is $4e^{-4}$ between the 0th and 29th epoch, $4e^{-5}$ between the 30th and 74th epoch, $4e^{-6}$ between the 75th and 89th epoch, and $4e^{-7}$ from the 90th epoch and onward. This gradual decrease in learning rate will shorten the steps the optimizer takes towards the local minima, decreasing the chance of stepping over it.

We implemented and tested our models on the University Centre for Information Technology's (USIT) machine learning infrastructure. The cluster provided us with Nvidia RTX 2080ti GPUs to do our testing and debugging on. We switched over to using Simula Research Laboratory's eX^3 computing cluster when it was time to train our model. This is due to their more powerful Nvidia A100 GPUs and more robust resource management between users using the Slurm Workload Manager.

Chapter 5

Evaluation

In this section, we present the results of our thesis and discuss them. We want to answer the research questions we stated in the introduction to this thesis. For clarity we restate them here:

1. Does block-matching help in any way during the training of a stereo depth estimation model?
2. Will block-matching improve the results of a model vs. the same model without the use of block-matching?

The first question can be answered by looking at the loss curves presented in the previous chapter. We can get a lot of information about the training process by inspecting these in more detail. To answer the second question, we take a look at the performance results. Before going into the results of our experiments, we will first discuss the results of the inpainting module we implemented as a possible part of our pipeline.

5.1 Inpainting Results

We shall quickly discuss the results of our image inpainting module, illustrated in figure 5.1. As is quite obvious, the results of the model are underwhelming. There may be many reasons for this. The first reason may be the choice of model. The model might be unsuited for such a tough task of filling in lots of small and large holes left by the SGM algorithm. The

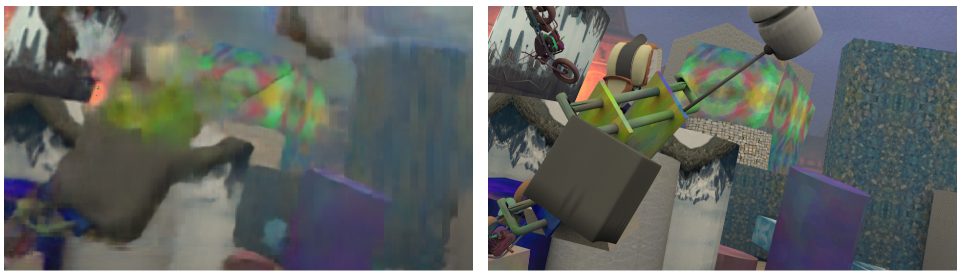

(a) Prediction (b) Ground-truth (right image)

Figure 5.1: Results of inpainting network.

leftmost hole of the warped image, the large black bar, is especially hard to fill in, as large holes are a common problem for inpainting networks in general. This is due to the lack of enough context when approaching the center of the hole. Another problem might have been the warping of the left image. As is apparent, the warping of the left image into the right image is not perfect. We could probably see improvements if we tweaked the parameters of the SGM algorithm to make the computed disparity map more accurate. The inpainting network is not the main focus of this thesis, and will instead propose this topic as future work in the next chapter. In the next sections, we go over the results of our main experiments and discuss them.

5.2 Training Results

In this section we refer to figures 5.2 and 5.3 which illustrates the loss curves of the training and validation sets over 115 epochs. Figure 5.2 plots the loss of the control model, the model that does not accept disparity maps as input. Figure 5.3 plots the loss of our experimental model, the model that do accept disparity maps. From the graphs, we can see that both models improved on the training set, because the training loss, shown in blue, is decreasing in both models. As shown by the red colored line, the validation loss is quite a bit higher than the training loss. This is the case for both models, although the validation loss for the control model (figure 5.2) has a net decrease even though it is quite volatile for the first 75 epochs or so. For our experimental model (figure 5.3) we see the same volatility in validation loss, but instead of a net decrease, we see a net increase. This is a sign of overfitting. In both cases, we do not see a good model fit, where the

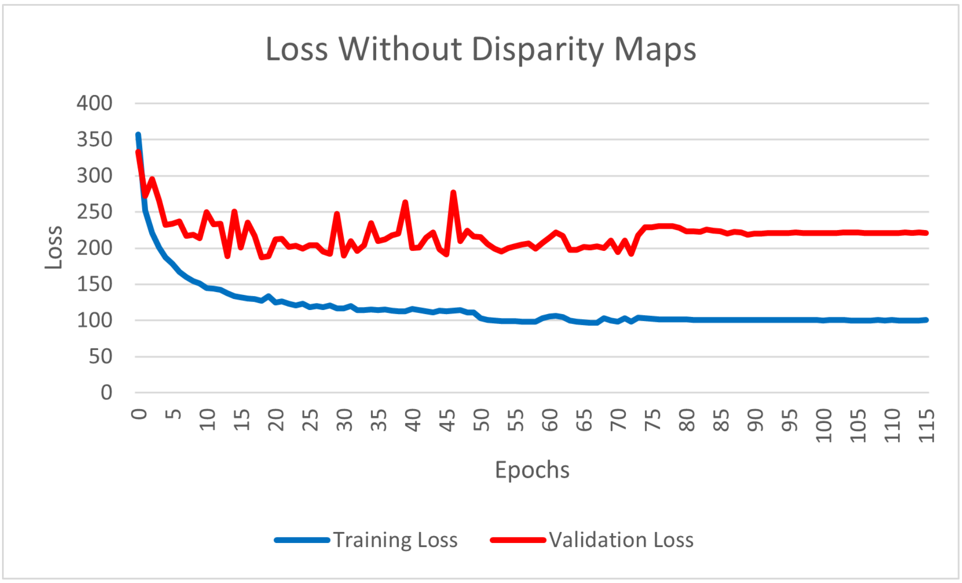


Figure 5.2: Training and validation loss curves without disparity maps as input.

validation and training loss is about the same.

5.3 Performance Results

In this section we refer to figures 5.4 to 5.8. These illustrate the progress of the different performance metrics we recorded during the 115 epochs of training. We also explain what each metric represents.

5.3.1 Peak Signal-To-Noise Ratio

We start by explaining what peak signal-to-noise ratio (PSNR) is before we explain our results. PSNR is a metric that tells us how noisy an image is compared to another. It can be used to evaluate compression algorithms on how good the compressed image is versus the uncompressed image. Since it measures the noise between images, it is measured in decibels (dB). It is computed between two images A and B with the following formula

$$\mathrm{PSNR}(A, B) = 20 \cdot \log_{10}(\mathrm{MAX}) - 10 \cdot \log_{10}(\mathrm{MSE}(A, B))$$

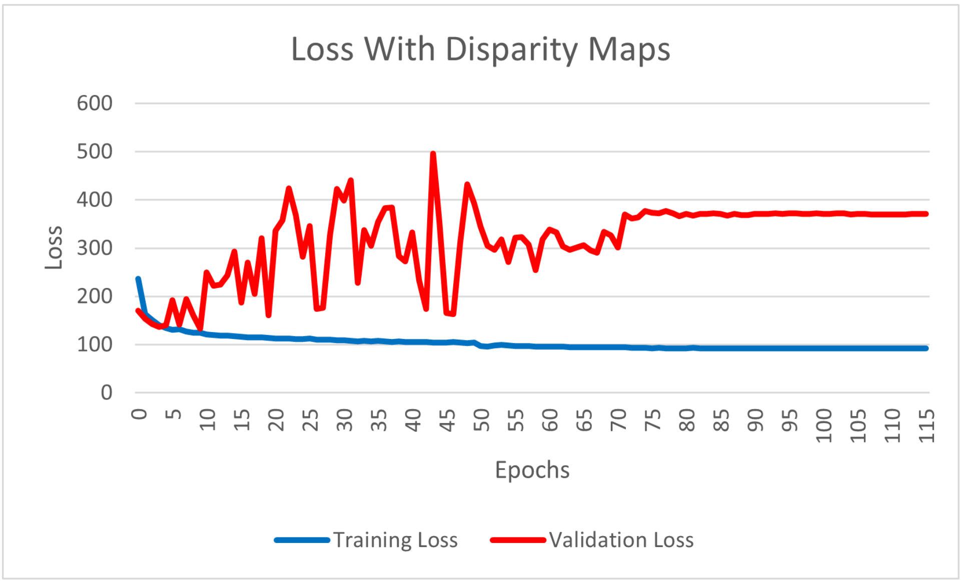


Figure 5.3: Training and validation loss curves with disparity maps as input.

where the MSE is the mean squared error of all pixels between images A and B with resolutions N×M

$$\text{MSE}(A, B) = \frac{(A - B)^2}{N \cdot M}$$

Note that we take the $\log_{10}$ of MAX where MAX is the maximum value a pixel can hold. Since we calculate the PSNR of disparity maps, which are single-channel images, the max pixel value is 255.

Figure 5.4 illustrates the mean PSNR per epoch for the control model (in green) and the experimental model (in orange). For PSNR, a higher value is better. We can see that after the final epoch, the best-performing model by far is the control model. We can also see that the PSNR is oscillating and unstable for the first 75 epochs, which was also the case for the loss curves. However, the control model in green is more stable than the experimental model in orange. Both stabilize after the 75th epoch mark, which we also could see in the loss curves.

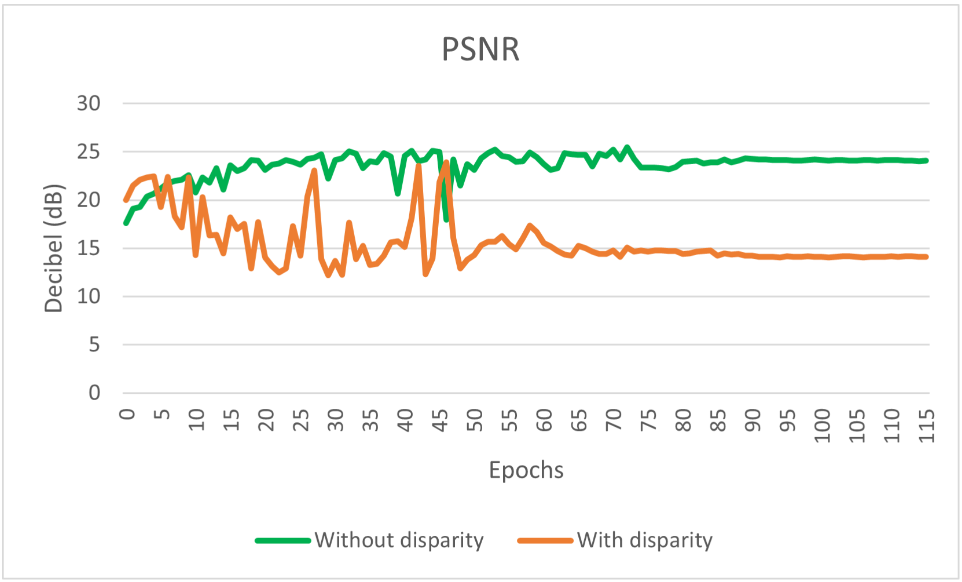


Figure 5.4: peak signal-to-noise ratio

5.3.2 Percentage of Erroneous Pixels

The percentage of erroneous pixels is a measure of how many disparities of the predicted disparity map have a greater error than a certain threshold. In other words, if the error between the predicted disparity $\hat{d}$ and ground-truth disparity d, $\left|\hat{d} - d\right|$ is greater than the threshold, that pixel is considered erroneous, or bad. In figure 5.5, we plot the percentage of disparities with an error greater than one, i.e., $\left|\hat{d} - d\right| > 1$. Figure 5.6 plots the percentage of bad pixels with a greater error than three, i.e., $\left|\hat{d} - d\right| > 3$. We refer to these figures as bad1 and bad3 respectively. We would expect the bad3 metric to be smaller than the bad1 metric, as it has a higher error threshold, and thus more disparities will fall in the category of a "good" pixel. This metric was proposed by Scharstein and Szeliski (2002) and is used to give us an idea of how much of the predicted disparity map is correct within a certain threshold.

We want this metric as small as possible, as it means a bigger percentage of the predicted disparity map is correct. If we inspect the two figures, we can see that the disparity maps improve over epochs. What is different from the PSNR, is that our experimental model (in orange) is better than the control model (in green), both for the bad1 and bad3 metrics. Although the difference is small for the bad3 metric, it is quite noticeable for the bad1 metric. We can also see that the control model is more volatile than the

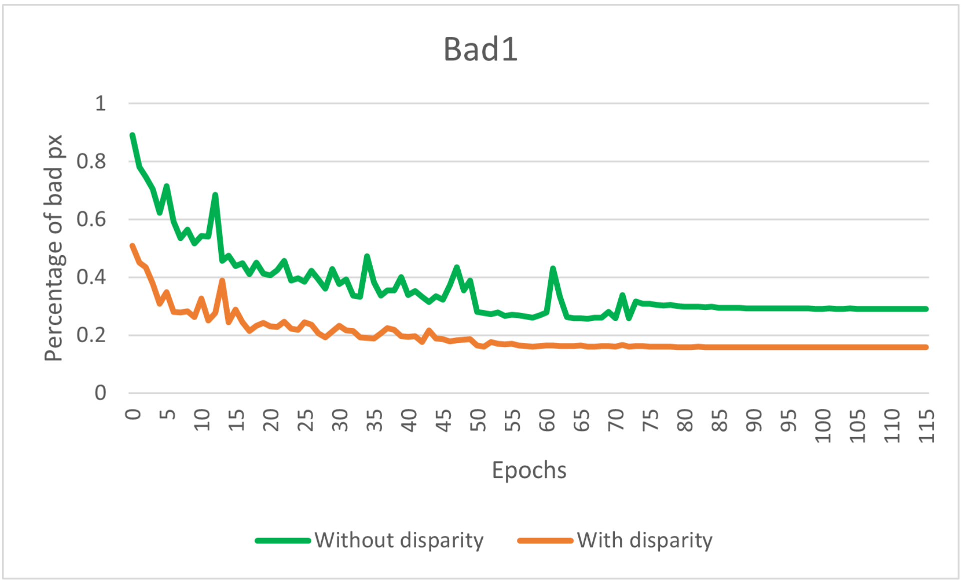


Figure 5.5: Percentage of pixels with an end-point error greater than 1 pixel

experimental model for the first 75 epochs, which has been the opposite case for the other metrics.

5.3.3 Root-Mean-Square Error

Root-mean-square (RMS) error is the third metric we evaluate our models by. It gives us the root of the mean error between the predicted disparities $\hat{d}$, and the ground-truth disparities d for all disparities in a disparity map. The formula is given as:

$$\mathrm{RMS}(\hat{d}, d) = \sqrt{\frac{\left|\hat{d} - d\right|^2}{N}}$$

Figure 5.7 shows the plots of RMS between the two models. The most noticeable thing about the graph is the massive fluctuations in our experimental model (in orange) for the first 50 epochs before it settles afterward. We can also see that the control model (in green) has an overall better score than the experimental model.

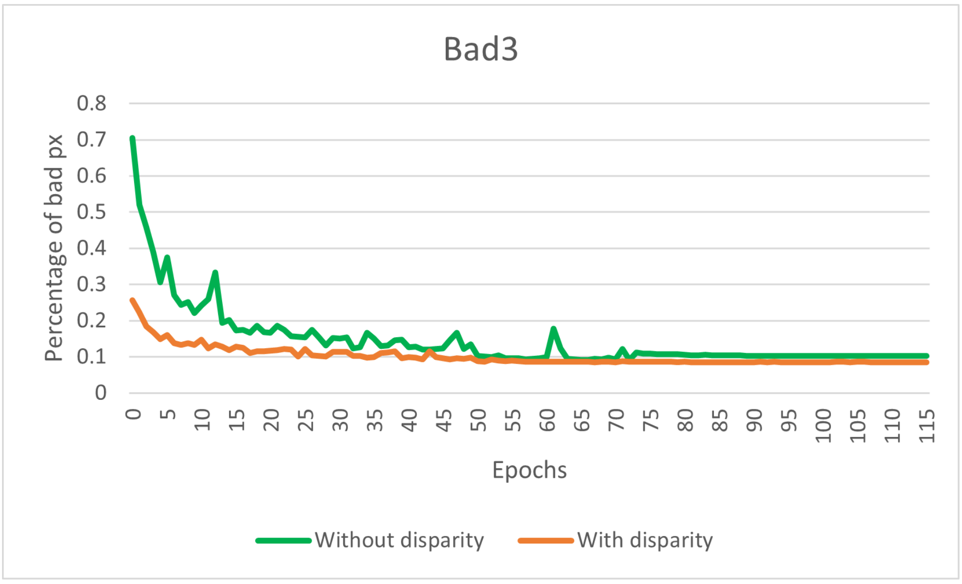


Figure 5.6: Percentage of pixels with an end-point error greater than 3 pixel

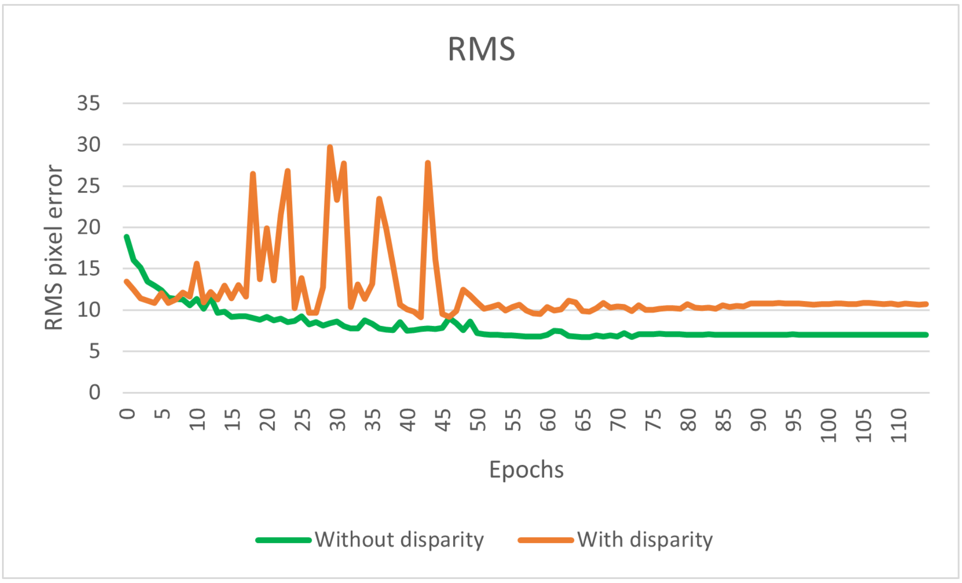


Figure 5.7: Root-mean-square error

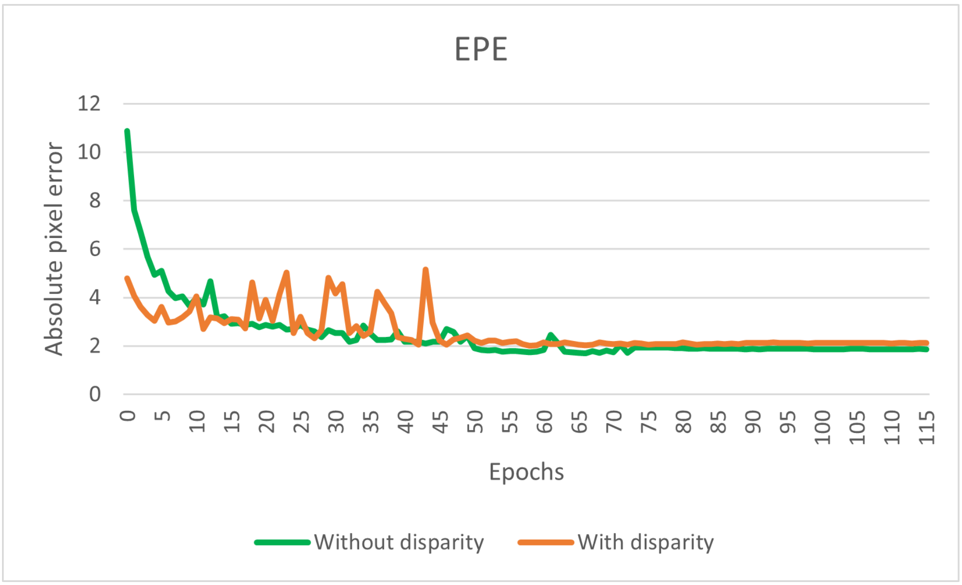


Figure 5.8: End-point error

5.3.4 End-Point Error

As the final performance metric, we use the end-point error (EPE). It is similar to the root-mean-square error and is a measure of the error between the predicted disparity map, and the ground truth. The reason for tracking the end-point error of our models is to more easily compare them with other state-of-the-art models. The end-point error is simply the absolute error between the predicted disparity $\hat{d}$ and the ground-truth disparity d:

$$\text{EPE}(\hat{d}, d) = \left|\hat{d} - d\right|$$

As is mentioned in Tankovich et al. (2020), we exclude all pixels with ground-truth values bigger than 192 from the end-point error calculations, to get more accurate results that are comparable to the original HITNet model.

If we look at figure 5.8, we can see the change in EPE over epochs. We can see the same developments that are present in the RMS plot, mainly that the experimental model is much more inconsistent during the first 50 epochs while stabilizing afterward. Unlike in the RMS plot, we can see that both models converge towards the same end-point error after a while.

5.4 Disparity Map and Slant Map Results

The figures (a) and (b) in figure 5.10 shows the final prediction of the two models, and figures (a) and (b) in figure 5.11 shows the final slant maps. The figures from 5.12 and onward shows how the disparity maps and slant maps have been refined over the 115 epochs.

By inspecting the final disparity and slant maps and comparing them to their ground-truth maps, we can see that the control model produces better disparity maps. This is also evident in the results presented previously. As for the slant maps, both need further improvements to be viable for the models. Improvements to the slant maps will further improve the disparity maps. Due to the subpar slant maps, the accuracy of the disparity maps suffers the consequences.

We can look at the evolution of these maps from figures 5.12 and onward. After the first epoch, we can see that our experimental model (disparity map marked with (a)) is quite a lot better than the control model (disparity map marked with (b)). However, this changes after a few epochs when the control model surpasses the experimental model. This is evident from the disparity maps in figure 5.13 and onward. We can also see that the quality of the disparity maps in figure 5.14 has worsened versus the disparity maps in figure 5.13. As for the slant maps, we can notice small improvements in the slant maps from epoch to epoch, although the results aren't really resembling the ground-truth at any point.

5.5 Discussion

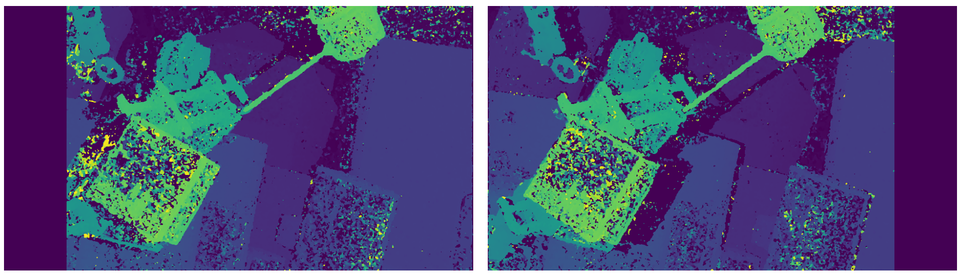
As has been observed in all previous plots, both models are very volatile. The loss curves show a trend of both models overfitting. This may be a case of a too high learning rate. Recalling back to section 4.8 about the training details of our networks, we mentioned the use of learning rate scheduling to decay the learning rate over time. We divided the learning rate by ten at the 50th, 75th and 90th epoch. These decreases are visible in figures 5.2 and 5.3. If we observe the graph at these points, we can see that the graph changes. This is especially noticable in the validation loss curves. The loss becomes more stable when the learning rate decreases.

This leads us to believe that our initial learning rate of $4e^{-3}$ is too high and should be decreased. The use of a small learning rate is also discussed in the HITNet paper: "Indeed, empirically we found that using a small initial learning rate $1e^{-4}$ and training for longer achieves the best results on multiple datasets without showing sign of overfitting" [Tankovich et al., 2020, p. 12], although in our case, we would benefit from an even lower learning rate. We also observe that the experimental model (figure 5.3) is more sensitive to the learning rate, as the validation loss is much more volatile. The experimental model also overfits almost immediately.

Another observation we've made is that the experimental model is better than the control model in all metrics for the first five epochs. This is an important observation, as it upholds our idea that disparity maps do have a positive impact on both the training and accuracy of a neural network. We explain the decrease in accuracy after the first five epochs as so:

After the five first epochs, the experimental model starts to overfit. It still improves on the training dataset, while it worsens on the validation dataset. This is not the case for the control model, as the validation loss still decreases after the fifth epoch. Since the only difference between the two models is the SGM algorithm, it is easy to recognize that the problem lies with the algorithm. We believe the main problem is that the experimental model becomes too biased toward the disparity maps. We also believe that the SGM algorithm performs worse overall on the validation set. This results in a model that after the five first epochs, has understood that the information in the disparity maps is useful. However, if the disparity maps are very inaccurate, it doesn't change the model's intuition to use the disparity maps. This is what we believe is happening to the experimental model on the validation set. The SGM algorithm produces inaccurate disparity maps of the data samples in the validation set that the model will heavily rely on. This severely reduces the performance of the model on unseen data. It is a classic case of overfitting. The reason the experimental model is doing good for the first five epochs is most likely due to the weights of the model not being updated to weigh the disparity maps more than the RGB images. This leads to the model taking into account more features of the RGB images in its prediction. This is also why the validation loss is at its lowest for the first five epochs.

A solution to this problem is to control the model's weights more closely,

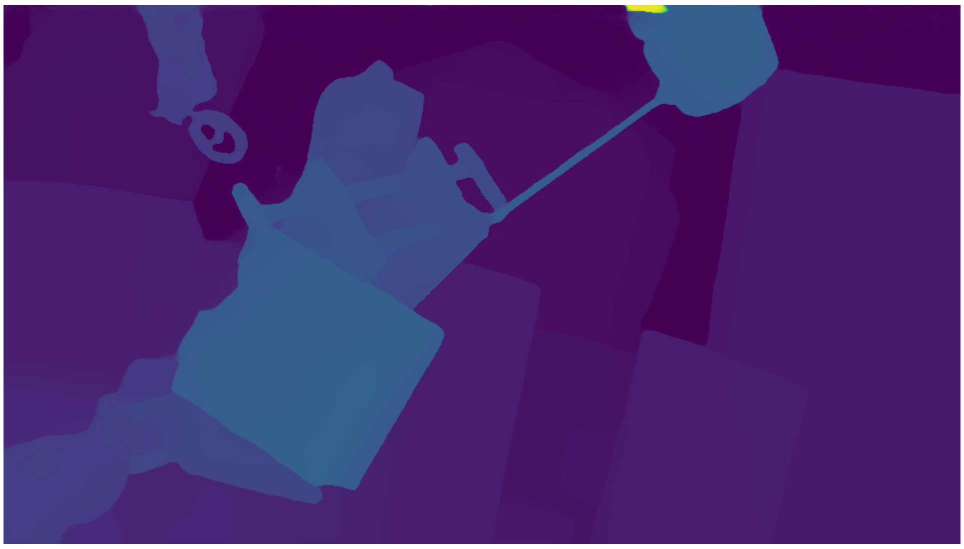


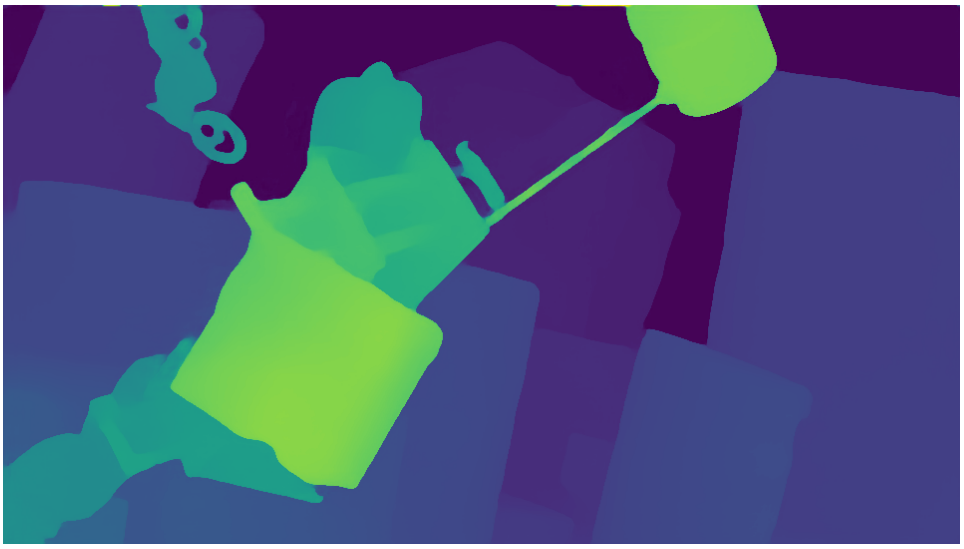
(a) Left disparity map (b) Right disparity map

Figure 5.9: Disparity maps computed by the SGM algorithm. (a) uses the left image as the reference image, while (b) uses the right as the reference image.

for example by using dropout layers. In our model, the feature extraction network processes both the RGB image and the disparity map at the same time (the RGB image is concatenated with the disparity map). By creating two separate feature extraction networks, one for RGB images and disparity maps separately, it is easier to fine-tune the feature extraction of both inputs.

There is also the discussion of if the SGM algorithm is the correct algorithm for such a solution. The initial disparity maps fed into our experimental model have been of fairly poor quality, as shown in figure 5.9. A solution could be to tweak the SGM parameters to improve performance. A problem with this, however, is that the same parameters are used for all samples in the SceneFlow dataset. Although the parameters may improve the disparity maps of some samples, they may also decrease the disparity map quality of other samples. We, therefore, need to find the parameters that fit well for any arbitrary left-right image pair, which takes a lot of trial and error. Another solution is to change the block-matching algorithm to another algorithm, maybe even a pretrained stereo depth estimation network. A pretrained network already has found the best general parameters to use for any arbitrary sample. By swapping the SGM algorithm with a state-of-the-art stereo depth model, we could also more easily test if disparity maps as input have any positive effects on disparity map predictions. The disadvantage of this is that we move away from classic block-matching algorithms which is what this thesis is about.


(a) With disparity maps


(b) Without disparity maps

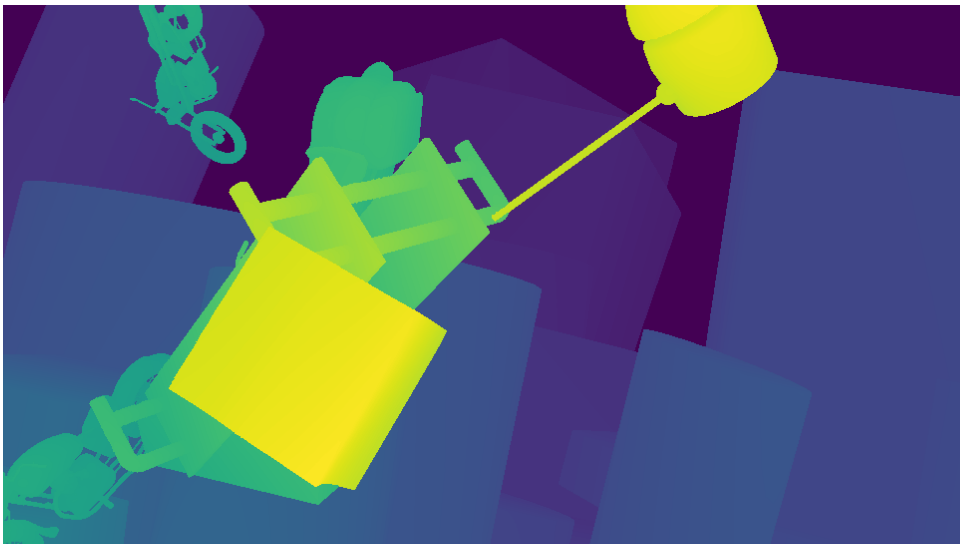
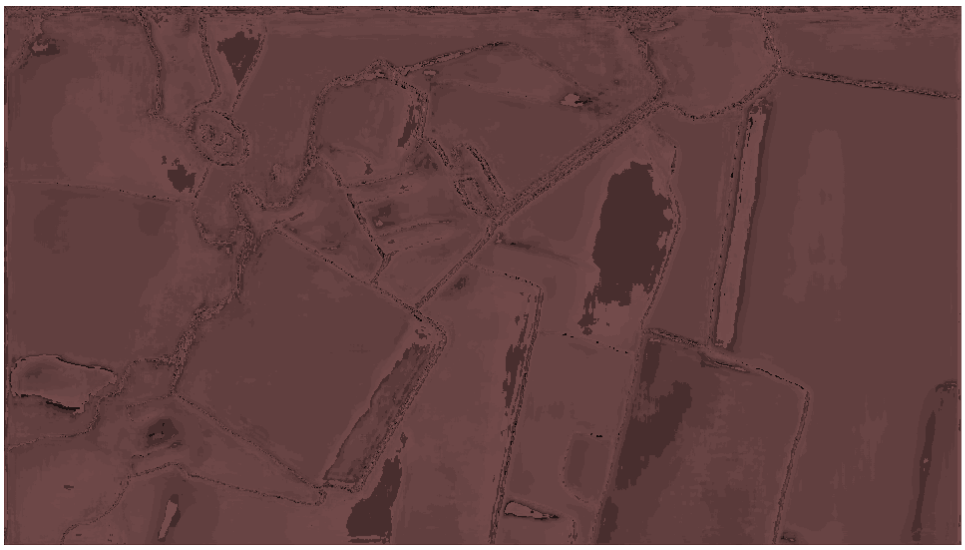
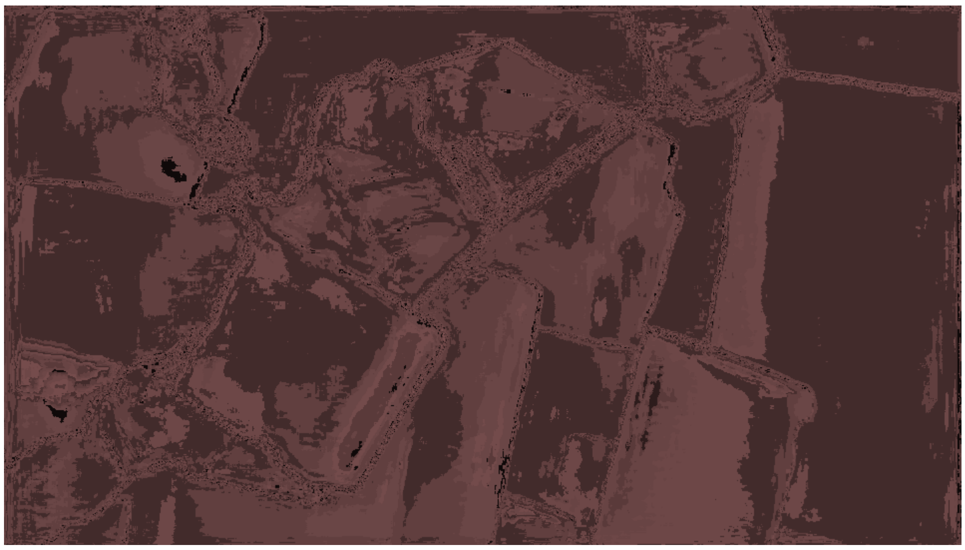

(c) Ground-truth

Figure 5.10: Final disparity map predictions


(a) With disparity maps


(b) Without disparity maps

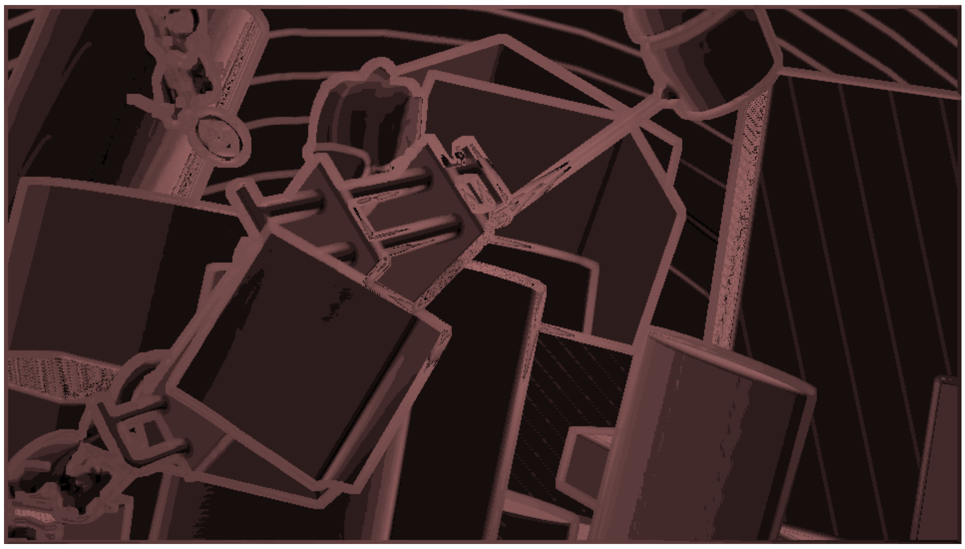
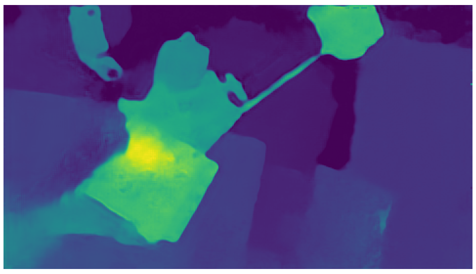
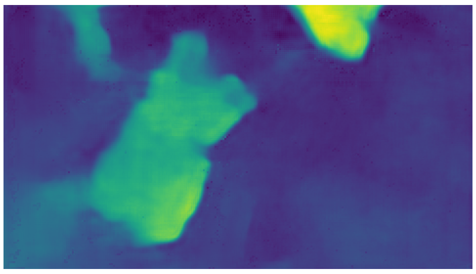

(c) Ground-truth

Figure 5.11: Final slant map predictions


(a) With disparity maps


(b) Without disparity maps

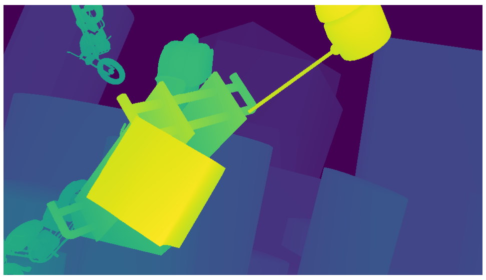
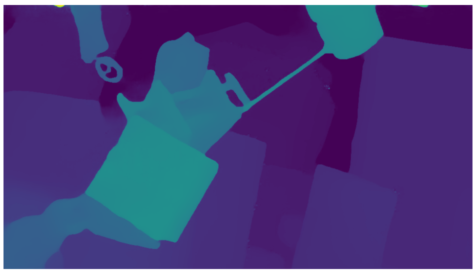
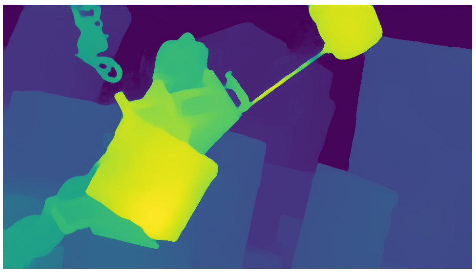

(c) Ground-truth

Figure 5.12: Results after a single epoch, comparing with and without disparity maps as input.


(a) With disparity maps


(b) Without disparity maps

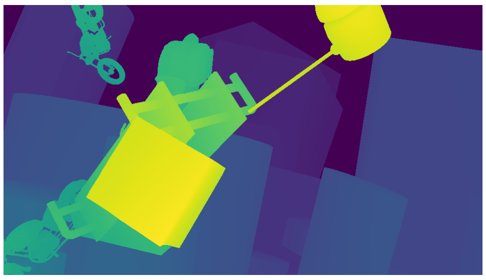
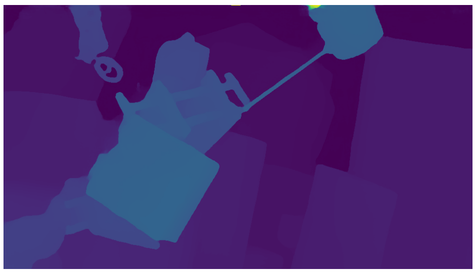
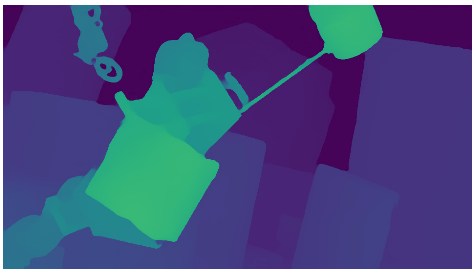

(c) Ground-truth

Figure 5.13: Results after 39 epochs


(a) With disparity maps


(b) Without disparity maps

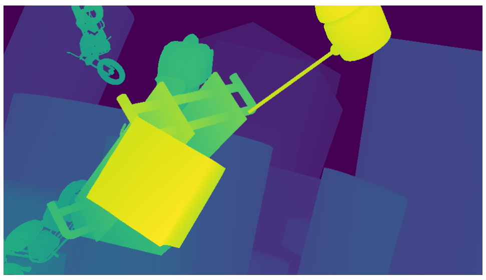
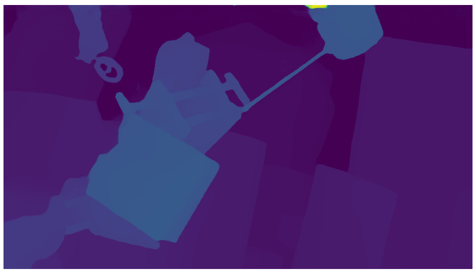
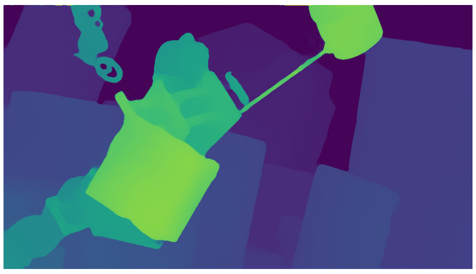

(c) Ground-truth

Figure 5.14: Results after 77 epochs


(a) With disparity maps


(b) Without disparity maps

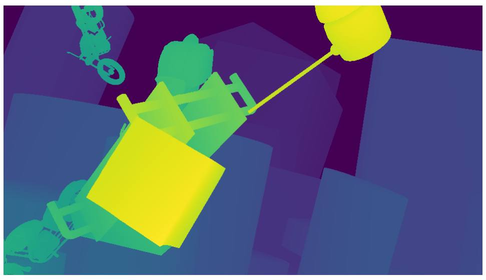
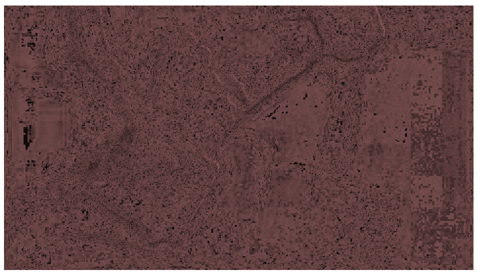
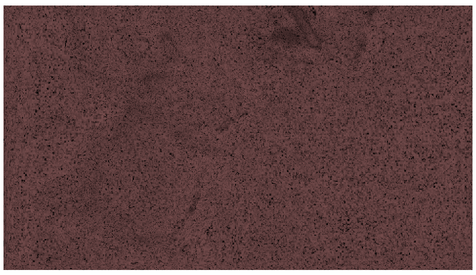

(c) Ground-truth

Figure 5.15: Results after 115 epochs


(a) With disparity maps


(b) Without disparity maps

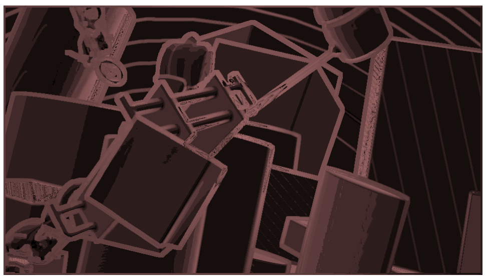
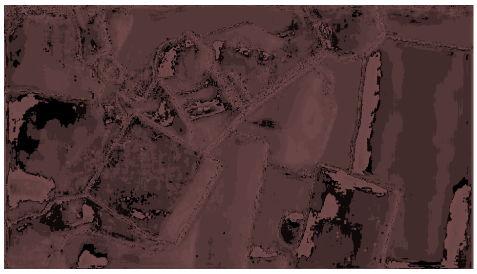
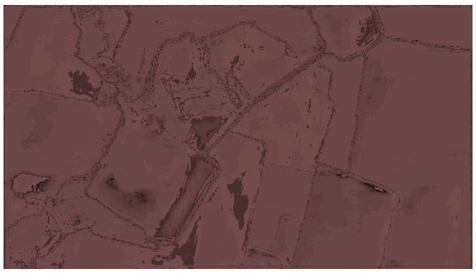

(c) Ground-truth

Figure 5.16: Slants after first epoch.


(a) With disparity maps


(b) Without disparity maps

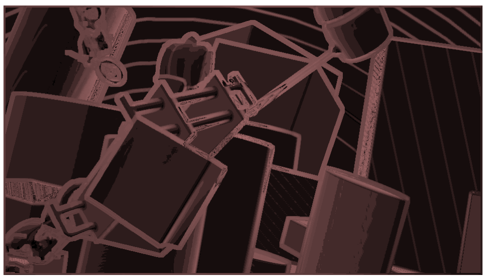
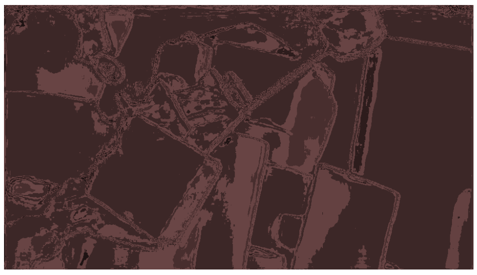
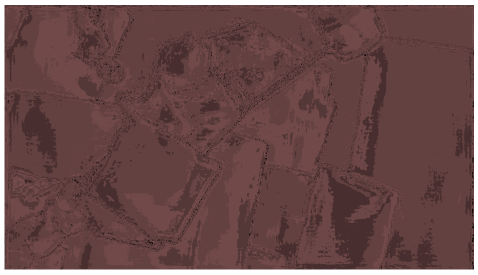

(c) Ground-truth

Figure 5.17: Slants after 39 epochs.


(a) With disparity maps


(b) Without disparity maps

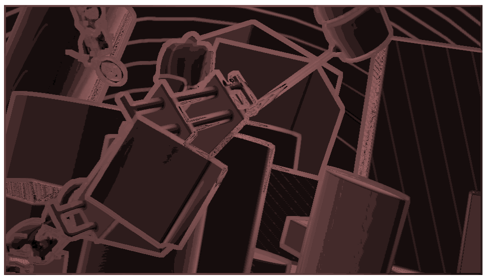
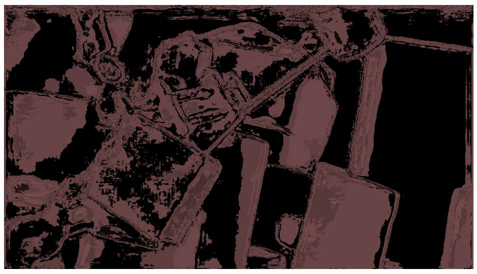
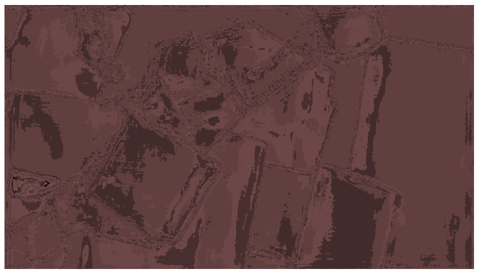

(c) Ground-truth

Figure 5.18: Slants after 77 epochs.


(a) With disparity maps


(b) Without disparity maps

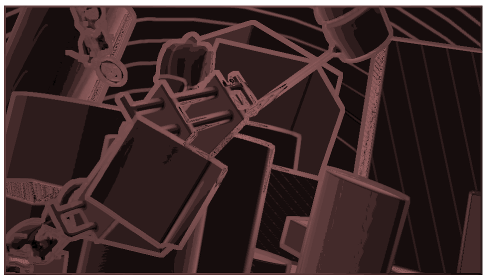

(c) Ground-truth

Figure 5.19: Slants after 115 epochs.

Chapter 6

Conclusion and Future Work

In this thesis, we have explored the idea of using the combination of block-matching and machine learning to predict accurate disparity maps. With accurate depth estimation, we could address the problems present in mixed and augmented reality today, where depth estimation is especially important. We used a combination of the semi-global matching algorithm, and the HITNet stereo depth estimation model to test if disparity maps computed by the SGM algorithm have any positive impact on machine learning disparity map predictions. We tested this by implementing two identical models based on Tankovich et al. (2020), one only accepting the left-right image pair (the control model), and one accepting both the left-right image pair along with their disparity maps (the experimental model). We also discussed the possibility of using image inpainting to recreate the right image. We've tested our models and gotten results that lead us to believe that our approach does not provide any notable benefits to today's machine learning approaches. However, solutions to the problems have been discussed.

6.1 Future Work

We believe there is still research to be done on the topic of this thesis, proposed in the bullets below:

- There is still room to tweak both the block-matching algorithm and the ML model. This includes tweaking the SGM parameters presented in section 4.1.2 to improve the performance of the block-matching algorithm. There is also a need to experiment with model hyperparameters such as the learning rate. Testing different optimizing and learning rate scheduling techniques are also needed. Another idea is to incorporate new layers such as batch-normalization layers and dropout layers. We refer to the discussion in section 5.5.

- We also propose the idea of completely replacing the SGM algorithm with either another classic block-matching algorithm or even another stereo depth machine learning method to provide the model with more accurate disparity maps.
- As for the small detour we had with the image inpainting module, we believe there is still lots of research to be done. Image inpainting could be especially interesting in conjunction with the topic of stereo depth estimation, where occlusion errors are a big problem. Filling in occluded areas using image inpainting seems like an interesting research topic. We refer to the paper "Softmax Splatting for Video Frame Interpolation" [Niklaus and Liu, 2020] for further reading.

Bibliography

Barron, J. T. (2017). A General and Adaptive Robust Loss Function. http://arxiv.org/abs/1701.03077

Chang, J. R. & Chen, Y. S. (2018). Pyramid Stereo Matching Network. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 5410–5418. https://doi.org/10.1109/CVPR.2018.00567

Comer, D. E., Gries, D., Mulder, M. C., Tucker, A., Turner, A. J. & Young, P. R. (1989). Computing as a discipline (P. J. Denning, Ed.). *Communications of the ACM*, *32*(1), 9–23. https://doi.org/10.1145/63238.63239

Dumoulin, V. & Visin, F. (2016). A guide to convolution arithmetic for deep learning. http://arxiv.org/abs/1603.07285

Eden, A. H. (2007). Three Paradigms of Computer Science. *Minds and Machines*, *17*(2), 135–167. https://doi.org/10.1007/s11023-007-9060-8

Geiger, A., Lenz, P. & Urtasun, R. (2012). Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite. *Conference on Computer Vision and Pattern Recognition (CVPR)*.

Gu, X., Fan, Z., Dai, Z., Zhu, S., Tan, F. & Tan, P. (2019). Cascade Cost Volume for High-Resolution Multi-View Stereo and Stereo Matching. http://arxiv.org/abs/1912.06378

Hadsell, R., Chopra, S. & LeCun, Y. (2006). Dimensionality Reduction by Learning an Invariant Mapping. *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, *2*, 1735–1742. https://doi.org/10.1109/CVPR.2006.100

He, K., Zhang, X., Ren, S. & Sun, J. (2015). Deep Residual Learning for Image Recognition. http://arxiv.org/abs/1512.03385

Hirschmuller, H. (2008). Stereo Processing by Semiglobal Matching and Mutual Information. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *30*(2), 328–341. https://doi.org/10.1109/TPAMI.2007.1166

Lee, H., Grosse, R., Ranganath, R. & Ng, A. Y. (2009). Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. *Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09*, 1–8. https://doi.org/10.1145/1553374.1553453

Liu, G., Reda, F. A., Shih, K. J., Wang, T.-C., Tao, A. & Catanzaro, B. (2018). Image Inpainting for Irregular Holes Using Partial Convolutions. http://arxiv.org/abs/1804.07723

Mayer, N., Ilg, E., Häusser, P., Fischer, P., Cremers, D., Dosovitskiy, A. & Brox, T. (2016). A Large Dataset to Train Convolutional Networks for Disparity, Optical Flow, and Scene Flow Estimation. *IEEE International Conference on Computer Vision and Pattern Recognition (CVPR)*. http://lmb.informatik.uni-freiburg.de/Publications/2016/MIFDB16
arXiv:1512.02134

Menze, M. & Geiger, A. (2015). Object Scene Flow for Autonomous Vehicles. *Conference on Computer Vision and Pattern Recognition (CVPR)*.

Niklaus, S. & Liu, F. (2020). Softmax Splatting for Video Frame Interpolation. http://arxiv.org/abs/2003.05534

OpenCV. (2021). OpenCV: cv::StereoSGBM Class Reference. Retrieved May 10, 2022, from https://docs.opencv.org/4.5.4/d2/d85/classcv_1_1StereoSGBM.html

Ronneberger, O., Fischer, P. & Brox, T. (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation. http://arxiv.org/abs/1505.04597

Scharstein, D., Hirschmüller, H., Kitajima, Y., Krathwohl, G., Nešić, N., Wang, X. & Westling, P. (2014). High-Resolution Stereo Datasets with Subpixel-Accurate Ground Truth. https://doi.org/10.1007/978-3-319-11752-2_3

Scharstein, D. & Szeliski, R. (2002). A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms. *International Journal of Computer Vision*, *47*(1), 7–42. https://doi.org/10.1023/A:1014573219977

Tankovich, V., Häne, C., Zhang, Y., Kowdle, A., Fanello, S. & Bouaziz, S. (2020). HITNet: Hierarchical Iterative Tile Refinement Network for Real-time Stereo Matching. http://arxiv.org/abs/2007.12140

Žbontar, J. & Le Cun, Y. (2015a). Computing the stereo matching cost with a convolutional neural network. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, *07-12-June*(1), 1592–1599. https://doi.org/10.1109/CVPR.2015.7298767

Žbontar, J. & LeCun, Y. (2015b). Stereo Matching by Training a Convolutional Neural Network to Compare Image Patches. http://arxiv.org/abs/1510.05970

Zhang, A., Lipton, Z. C., Li, M. & Smola, A. J. (2020). *Dive into Deep Learning*. https://d2l.ai/