Eindhoven University of Technology

MASTER

Global Motion Estimation using Machine Learning

Kole, Casper F.

Award date:
2021

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Department of Mathematics and Computer Science
Architecture of Information Systems Research Group

Global Motion Estimation using Machine Learning

Thesis

Casper F. Kole

Supervisors:
prof.dr.ir. N. Meratnia
ir. J. Kettenis

Assessment Committee:
prof.dr.ir. N. Meratnia
ir. J. Kettenis
prof.dr.ir. G. de Haan
prof.dr.ir. B. Speckmann

Eindhoven, August 2021

Abstract

Videos are made up of frames, and each video has a frame rate, i.e., the number of frames that are shown per time unit. A video with a higher frame rate will result in smoother motion and also sharper-looking moving objects. The task of frame rate up-conversion is to increase the frame rate of a given video, for instance from 30 to 120 frames per second.

Motion estimation is an important tool for the task of frame rate up-conversion. When the motion of each element in a video frame is correctly estimated, this information can be used to create new frames. These new frames can be inserted into the original video, so the frame rate of the video is artificially increased. A subset of motion estimation is global motion estimation, where not the motion of every element in the scene is found, but only the motion of the largest part of the scene. This global motion can be captured using a model, known as a parametric motion model. With this model, generic motion transformations can be described such as translation, zooming and rotation.

In this thesis, we propose a simple convolutional neural network approach to solve the problem of global motion estimation using motion vectors as input. We present a systematic approach to generate artificial datasets of frames using randomly generated global parametric motion models. We train the proposed convolutional neural network on (i) the generated pixels in the datasets as well as (ii) motion vectors to analyse to what extent a convolutional neural network approach is able to estimate the global motion in a video scene. We also compared a simple regression approach to an approach where a classification neural network is used as support for a regression neural network. Our experiments show that the parametric models estimated by our simple convolutional neural network can accurately describe the global motion in a video scene. Noteworthy is also that we used motion vectors as input, which gave much better results than using raw image data, which has been the common approach. Moreover, by using these estimated models as support for a motion estimation algorithm, the accuracy of the motion estimation algorithm significantly increased.

Preface

During the last year of my Master's degree, I attended the course Video Processing, taught by Gerard de Haan. This course ignited my interest in the topic and made me decide on video processing as my research topic. It is also thanks to Gerard de Haan that I was brought into contact with V-Silicon, the company for which this thesis was developed. So hereby I would like to thank him for indirectly helping me with writing this thesis.

Nirvana Meratnia was also a great help as my thesis supervisor at the university, and I want to thank her for that as well as for having meetings with me every week. Similarly, I want to thank Jeroen Kettenis who was my supervisor at V-Silicon and who also helped a considerable amount when I was stuck or needed to discuss a problem. For help with proofreading, I want to thank my friends Daniël and Rowin. I would also like to thank my girlfriend Quinty for the moral support she gave me.

List of Figures

List of Tables

List of Algorithms

Contents

Chapter 1

Introduction

Video has become one of the most popular media in the last century. Each video has a frame rate, i.e., the number of frames that are shown per second. When the frame rate of a video is too low, movement may seem blurred or jerky. For this reason, there is the desire to up-convert the frame rate so that motion is smoother. To perform this frame rate up-conversion, we are required to solve the problem of motion estimation.

The problem of motion estimation is to find motion vectors for each pixel in a single frame of a video over time. When we thus have frame x and subsequent frame y, we could obtain the motion vectors for the pixels in frame x to move in such a way that we obtain frame y. When we know these motion vectors, we could theoretically create new artificial frames between x and x, hereby increasing the frame rate of the video.

Global motion estimation (GME) is a subset of this problem that tries to describe the global motion, which is the motion for a major part of the scene. In most cases, the motion is caused by moving the camera, for example when the camera zooms in. However, global motion can also be caused by something in the scene itself, e.g., in the case of a video where the camera tracks a moving object, the background is actually moving, and the foreground sits still. Knowing the global motion can be of great help when performing local motion estimation, where motion is estimated for each pixel or group of pixels in a frame individually. GME can be used to support local motion estimation as well, for instance when no satisfying motion vector for a certain pixel can be found, the global motion is used. Another application of GME is moving artificially generated textures, for example, grass on a soccer field. When such a texture is moved with local motion, one pixel with a wrong motion vector will distort the texture, while with a global motion description, even if the description is flawed, the entire texture still moves together.

In this thesis, we will try to solve the problem of GME from the angle of machine learning. Though local motion estimation has been researched thoroughly in the context of machine learning, we could find no other papers researching GME in this context. Moreover, neural networks that perform motion estimation have always worked with the raw video as input, meaning the direct pixel values. In this thesis, we will also look at using motion vectors that have already been estimated as input for the neural network. The expectation is that local motion vectors contain more useful information than raw image data for global motion estimation. In his paper on parametric models [6], Gerard de Haan also uses motion vectors to estimate global motion, so the idea is that that would work for the machine learning approach as well.

The topic of this thesis is an assignment from V-Silicon [27], which is a company that specializes in video processing. They already have developed techniques that are fast and accurate for computing local motion. However, they found that the method they have developed for computing global motion was not accurate enough, and they believe that machine learning might be a solution for

this problem. This also means that the developed software in this thesis, if satisfactory, would be part of the motion estimation software pipeline of V-Silicon. We will use their local motion estimation software to perform a rough motion estimation that gives the motion vectors that will be used as input for the neural network that will perform global motion estimation.

1.1 Problem Description

The problem we address is how to estimate the global motion for a frame sequence consisting of subsequent frames, making use of machine learning, where the input for the machine learning approach is local motion vectors. The research question we have is:

> *To what extent can convolutional neural networks provide a global parametric motion model, true to the real global motion in a video scene, using motion vectors as input?*

What and how such a global parametric model works, will be explained in Chapter 2.

1.2 Challenges

There are a few challenges with this problem. Firstly, there is a problem with defining what part of a video scene follows the global motion. In some scenes this might be obvious, but in other scenes it is not. For example, when we have a scene where the foreground is so large that it takes up more than half of the scene, it could be said that the foreground motion is the global motion. So part of the problem is also finding what part of the scene can be considered to have the largest consistent motion vector field.

Almost any neural network needs a dataset to train on to learn how to solve a problem, without such a dataset, the problem cannot be solved. Of course there exist large datasets with raw video footage, however, for a conventional neural network this dataset also has to be labelled. With a labelled dataset we refer to a dataset where each sample also contains a tag, this can be anything, like a name or a number for instance. In general, the label is what the neural network should output for that sample. In that case, the label is also sometimes called the ground truth. Though there exist neural networks that do not require labelled data, the largest challenge for motion estimation in a neural network context is lack of labelled data. For a simple problem like classifying animals in pictures, a labelled dataset would contain pictures of animals and each picture would have a label in the form of the name of the species of the animal that is depicted. For our global motion estimation problem specifically, a labelled dataset would contain frame sequences with as label a description of the global motion. Such a dataset does not exist. Even for the more popular problem of local motion estimation, there are only a couple of datasets describing the true local motion. Those datasets are also not large enough to train a complex neural network on. Solutions to the problem of a lack of labelled data, such as neural networks that do not require labelled data, are discussed in Chapter 2.

Lastly, there is the challenge of how to describe global motion. When describing local motion in a frame sequence, this is mostly done by creating a motion vector field of equal size to the frames in the frame sequence and showing a motion vector for each pixel in the frame. To save data, pixels can be grouped into blocks, as pixels that are part of the same object usually have the same motion anyway. For global motion, since the idea is that all areas in the frame sequence that follow the global motion can be described similarly, we need a more succinct way of describing the motion than giving a motion vector for each pixel or each block of pixels. Such a way of describing can be called a global parametric model, of which there are many different versions, as we will see in Chapter 2.

1.3 Contributions

The main contributions of this thesis are as follows.

- A systematic approach to generate artificial datasets of frames using randomly generated global parametric motion models.
- A simple CNN architecture for global motion estimation that uses local motion vector fields as input. The same CNN architecture also contributes to improving the accuracy of a local motion estimation algorithm.
- Providing a comparison of a neural network with a pure regression approach versus one that also uses a classification neural network as a support.

Chapter 2

Background Information

In this section, we will give a short theoretical background of some motion estimation concepts and of the type of neural network that we will use to solve the problem, namely convolutional neural networks.

2.1 Global Parametric models

To describe the global motion in a scene, a parametric motion model can be used [6]. There are different types of parametric models to describe global motion. The more parameters and terms used in the model, the more complex behaviour it can describe. The largest set of parametric motion models is described in [4] written by Bouthemy et al. Their model set, plus one model from [30], are shown in Table 2.1. This table is split into 3 categories based on the degree of the formula. The first group only contains the Translation model, this formula only contains constant values, the second group from Translation + Rotation to Planar Perspective are only equations of the first degree and the third group from Pan-Tilt to Full Quadratic contain equations of the second degree.

Before we discuss the different parametric models, we will explain the formula for a parametric model and how it is used to calculate local motion vectors. The formula is used to calculate a motion vector for a single pixel at a time, it takes as input 3 values, the x-coordinate of the pixel, the y-coordinate of the pixel and the index of the frame in the video of the pixel. The index of the frame is only relevant for selecting the parameters for that frame. Only the coordinates, in combination with the parameters, are used to calculate the motion vector.

The first model described in Table 2.1, **Translation**, is the simplest and contains only 2 parameters. Since the formula does not include the terms x or y, that means each pixel in the frame will get the same motion vector. In the case where p_1 has the value 2 and p_2 3, every pixel will be assigned the motion vector (2, 3). A pixel with such a motion vector will be moved 2 positions to the right, and 3 positions down between frames n and m. This would result in the entire image being shifted to the bottom right. Negative values for the motion vectors mean the pixels go in the opposite direction, so a motion vector (-1,-8) means the pixel is moved 1 position to the left and 8 positions to the top. Motion vector values do also not have to be whole numbers, this means that pixels move on what is called a sub-pixel level. Naturally, images only allow pixels to be placed at indices with whole numbers, so to get a correct image, we have to either move or interpolate pixels that are not placed on whole pixel positions.

Parametric models can describe more types of movement than translation. The second model in Table 2.1 can also describe **rotation**. Rotation is not the exact type of movement that is described, for that a cosine and sine function would be necessary. The more apt term for the specific type of movement that is described is skewing. The difference between skewing and rotating is

Motion Model	**No. Dims.**	**Formula**
Translation	2	$d(x,y) = (p_1, p_2)$
Transl. + Rotation	3	$d(x,y) = (p_1 + p_3 y, p_2 - p_3 x)$
Transl. + Zooming	3	$d(x,y) = (p_1 + p_3 x, p_2 + p_3 y)$
Transl. + Rot. + Zoom.	4	$d(x,y) = (p_1 + p_3 x + p_4 y, p_2 - p_4 x + p_3 y)$
Full Affine	6	$d(x,y) = (p_1 + p_3 x + p_5 y, p_2 + p_4 x + p_6 y)$
Planar Perspective	8	$d(x,y) = ((p_1 + p_3 x + p_5 y)/(p_7 x + p_8 y + 1), (p_2 - p_4 x + p_6 y)/(p_7 x + p_8 y + 1))$
Pan-Tilt	2	$d(x,y) = (p_1 + p_1 x^2 + p_2 xy, p_2 + p_2 y^2 + p_1 xy)$
Pan-Tilt-Zoom	3	$d(x,y) = (p_1 + p_3 x + p_1 x^2 + p_2 xy, p_2 + p_3 y + p_2 y^2 + p_1 xy)$
Planar Surface Rigid Motion	8	$d(x,y) = (p_1 + p_3 x + p_5 y + p_7 x^2 + p_8 xy, p_2 + p_4 x + p_6 y + p_7 y^2 + p_8 xy)$
Full Quadratic	12	$d(x,y) = (p_1 + p_3 x + p_5 y + p_7 x^2 + p_9 xy + p_{11} y^2, p_2 + p_4 x + p_6 y + p_8 y^2 + p_{10} xy + p_{12} y^2)$

Table 2.1: The set of 2-dimensional parametric models that are used to model the displacement of pixels in a video [4][30].

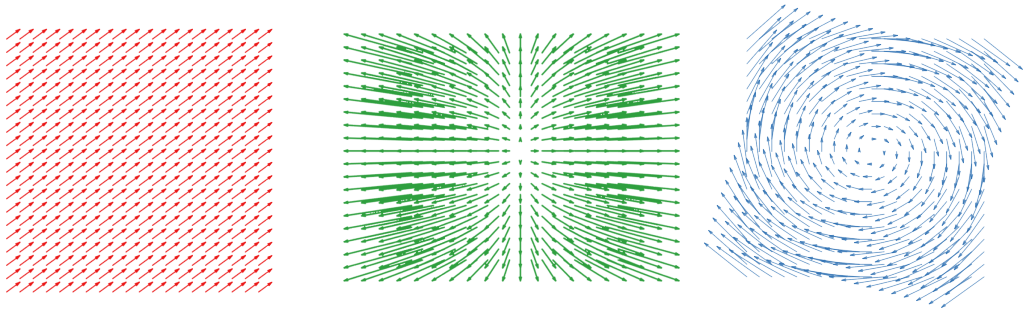


Figure 2.1: *Vector fields generated with an affine parametric model. The red vector grid shows a translation where only p_1 and p_2 are modified. The green vector field shows a zoom where only p_3 and p_4 are modified. The blue vector field shows a rotation where only p_5 and p_6 are modified.*

that with skewing, not only does the image rotate, it also increases in size. Skewing is used as an approximation of rotation in this case, and that is why we refer to it as rotation from here on. There is only a clear difference between skewing and rotating for rotations of more than approximately 20 degrees. In realistic videos, from one frame to another, rotations are only a few degrees, so using skewing as an approximation gives no problems.

The third model adds parameters to describe **Zooming**. Zooming here means that the image can be zoomed in or out. Both rotation and zooming occur around the centre of the image: the image should be rotated around the centre and the image is zoomed into or out of the centre. Conventionally, pixels in an image of size $n \times m$ are described, so the top-left pixel has position $(0, 0)$ and the bottom-right pixel has position (n, m). This convention is problematic if we want these transformations to happen around the centre of the image. To make rotation and zooming occur around the centre, the x and y coordinates of the pixels in the image are described, so the centre pixel of the image is at position $(0, 0)$.The result is that for an image of size $n \times m$, the top-left pixel has coordinates $(-n/2, -m/2)$ and the bottom-right pixel coordinates $(n/2, m/2)$.

Both the fifth model and the sixth model in Table 2.1 contain the 3 types of movement we have already discussed: translation, rotation and zooming. However, there is an important difference, the Full Affine (FA) model, has a higher number of dimensions than the Translation + Rotation + Zooming (TRS) model, meaning it uses more parameters. TRS can be seen as a subset of FA,

where $p_6 = p_2$ and $p_5 = -p_3$. This restriction causes for zooming and rotation transformations that the horizontal and vertical movement is always equal. This restriction causes the model to capture more realistic video motion, as a natural rotation or zooming motion always has equal horizontal and vertical motion.

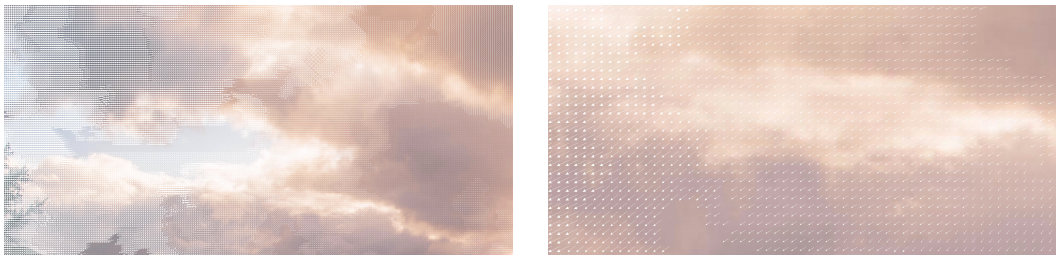
The last model with a linear formula in Table 2.1 is the Planar Perspective model. It is defined as a fraction, where the nominator is identical to the Full Affine model. The denominator contains 2 extra parameters, which according to [30] "... make it possible to take into account global deformations associated with perspective projections".

We then get to the set of models with quadratic formulas. The first one is called Pan-Tilt, which can describe panning and tilting motions. **Panning** is when the camera that is used to create a video is rotated horizontally, **Tilting** is when the camera is rotated vertically. These are both very common movements in realstic videos. The second model, Pan-Tilt-Zoom, is the Pan-Tilt model including the Zooming parameters. The Planar Surface Rigid Motion (PSRM) model is a combination of the Pan-Tilt and the Full Affine model. This model can describe all the common types of motion in videos: translation, rotation, zooming and pan-tilting. Lastly, there is the Full Quadratic model, which contains a large number of parameters, namely 12, and can describe very complex motion. However, such complex motion is extremely rare, so this model is too complex in most cases, as the PSRM model covers most regular types of motion.

2.2 Visualizing Motion Vectors

Once motion vectors have been computed for each pixel, it is good to be able to visualize this vector field. This way, one can find mistakes or compare different motion estimation methods, as will be done in this thesis. In the fields of mathematics and physics, a vector is often displayed as a line with an arrowhead. This method, pictured in Figure 2.2, visualizes the motion very directly and can give a precise insight into where the pixels are moving. The problem with this method is that at a glance it is hard to tell the direction of the arrows, since they are quite small. Zooming is necessary to actually see where the vectors are pointing and to which pixel they belong.

The more common way to visualize motion vectors in video processing is to use colour. For this approach, the YUV colour encoding system is used. The YUV system, like RGB, uses 3 components: the Y component describes brightness and the U and V component hue. A motion vector consists of 2 components, those are mapped to U and V and then a constant value for Y is chosen. The advantage of using YUV over RGB is that only 2 components are varied, but we still have access to the full colour spectrum. An example of the mapping of the colour plane to U and V can be seen in Figure 2.3. A pixel with a motion vector (-3,4), would for example result in a red



(a) Full image (b) Zoomed in

Figure 2.2: *A frame with its motion vectors visualized as arrows.*

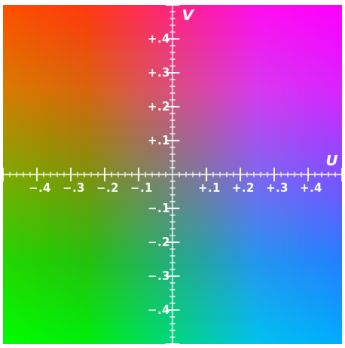


Figure 2.3: *The UV colour plane, showing how colour is mapped to the U-axis and the V-axis.*

pixel when we map the x component to U and the y component to V.

To convert the motion vector field to an image using YUV, we pick an appropriate constant value for Y, If we assume that like RGB, values range from 0 to 255, picking the middle value, 128, means the picture should not be too bright or too dim enough. Then for each pixel, we map the x and y components to U and V, respectively. To get a correct picture, the vectors have to be normalized beforehand by dividing them by the largest absolute vector component and multiplying by 60.

As almost all programs and screens that display images nowadays work with RGB, we have to convert the YUV values for the pixels to RGB to view them. To convert to RGB, the YUV vector is multiplied by a matrix. The specific constants in this matrix can vary depending on the RGB colour range that should be converted to. We chose the matrix used for HD Television by the Advanced Television Systems Committee (ATSC)[28]. This matrix can be seen in Equation 2.1.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.28033 \\ 1 & -0.21482 & -0.38059 \\ 1 & 2.12798 & 0 \end{bmatrix} \begin{bmatrix} Y \\ U \\ V \end{bmatrix} \tag{2.1}$$

The result of this visualization can be seen in Figure 2.4. The video for which motion estimation was performed happened to have the motion occur in clusters. These clusters were hardly visible when visualizing the motion vectors as arrows, but when we use colour, we can immediately identify these clusters of motion vectors. We only need to take a look at Figure 2.3 to see the direction of the motion vectors in each cluster. For example, on the left side of the image, we see a large yellow cluster. In Figure 2.3, yellow values appear around coordinate (-4, 1), this means the direction of these motion vectors is to the left and slightly upwards.

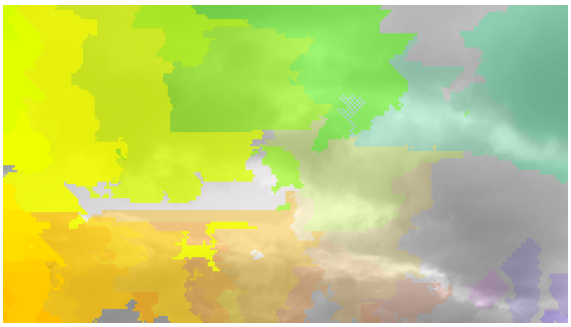


Figure 2.4: *A frame with its motion vectors visualized with YUV, where the x component is mapped to U and the y component to V. In this image, the length of the motion vector is also mapped to an alpha component so the shorter a vector is, the more it will show the original colour of the pixel.*

2.3 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a specific type of neural network that excels at learning about spatially related data[2]. Since video data is also spatially related, be it the pixels from an image or the motion vectors estimated for those pixels, it is natural that we will make use of CNNs. Using another type of neural network, for instance, would be extremely inefficient, and the reason for this will be explained below.

Neural networks contain large groups of parameters called weights and biases, whose appropriate value is "learned" for a specific task. These values are learned during a training phase where the network gets large sets of samples as input and has to learn to give a certain appropriate output. Each time the network makes a mistake, the values of these weights and biases are tuned such that this mistake does not happen in the future. The idea of a CNN is that it contains neurons that can efficiently detect spatial patterns, such that the CNN does not require many parameters. Such spatial patterns can be an edge or a circle, but these patterns can also be much more abstract.

If we take a fully connected neuron, the default type of neuron in a neural network, and give it an image as input, it would need an extremely large dataset to train its weights and biases. Colour images have for each pixel 3 values: red, blue and green, each value ranging from 0 to 255. That means that an image, that is only 64 pixels wide and 32 pixels tall, consists of $64 \times 32 \times 3 = 6144$ values. A fully connected neuron has a weight for each input variable, so it would have 6144 weights. The average fully connected layer in a neural network consists of multiple fully connected neurons, so a fully connected layer of 200 neurons, would have $6144 \cdot 200$ weights. Moreover, each neuron also has a bias variable, so in total there are $(6144 \cdot 200) + 200 = 1,229,000$ parameters. This is an extremely high number of parameters, which is not efficient, as it would require a large amount of training to tune all parameters to the optimal value.

A convolutional neuron looks only at a subset of input data at a time. Unlike a fully connected neuron that looks at all input data at once, a convolutional neuron iterates over the input data in blocks and gives an output for each block. So it might look at a square of 3×3 pixels, its kernel size, at once and give one output for that block, after that it shifts to another place on the image and do the same, this way it goes over the whole image. A visualization of this approach can be seen in Figure 2.5. Specifically, what the convolutional neuron does at each iteration to produce its output, is to take the dot product of its weights and the block of pixels it is looking at.

Since its input is only 9 pixels at a time, that means it only has $9 \times 3 = 27$ weights that need to be tuned, much less than the fully connected neuron, whose number of parameters is dependent on the image size. What is also useful is that it looks at local patterns instead of at the full image.

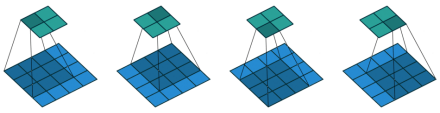


Figure 2.5: *A convolutional neuron with a kernel of size* 3×3 *going over an image of size* 4×4.

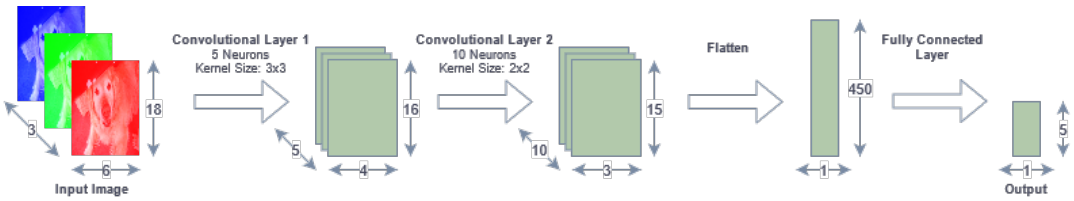


Figure 2.6: *An example of a convolutional neural network that takes images as input with 2 convolutional layers and 1 fully connected layer [9].*

Imagine that our task is to find whether there is a ball in our image, since this ball can be at any location in the image, a CNN could learn the shape of a ball pattern and check if this local pattern is somewhere in the image. The fully connected layer on the other hand is only able to look at the full image, so it would have a difficult time identifying local patterns.

For each pixel in the image, the convolutional neuron will try to compute an output value. However, this is not possible for pixels around the edge of the image. The green squares in Figure 2.5 are the output matrix of a neuron, and the blue squares are the input matrix. We see that the output matrix is smaller than the input. The size of the output matrix is dependent on the kernel size of the neuron and naturally the size of the input matrix. For an image of size $n \times m$ a neuron, with kernel size $k \times k$ will have a size $(n - k + 1) \times (m - k + 1)$. A CNN is made up of groups of neurons, called layers, all neurons inside a layer go over an image and then their output matrices are concatenated. If we let 5 CNNs go over an image of size 18×6, each neuron having a kernel size of 3×3, the output will be a matrix of size $16 \times 4 \times 5$. As the output of this convolutional layer is a spatially related matrix, we can use this output matrix as the input for another convolutional layer. The deeper in the network we go, usually the more neurons a layer consists of. So the second layer could for example have 10 convolutional neurons looking at blocks of 2×2. This layer outputs a matrix of size $15 \times 3 \times 10$.

Figure 2.6 shows the example layers described earlier. For each layer, the output matrix is reduced in width and height, but since each layer uses more neurons, the matrix gets "deeper". In a basic CNN architecture, where the desired output is only a list of values instead of a matrix, the last layer is usually a fully connected layer, which will give the final output. For a fully connected layer, the spatial relations between the data is not used, so before a fully connected layer, the input matrix is flattened, meaning all values are moved to the same dimensions. If the desired output is an image, instead of having this final fully connected layer, one or more deconvolutional layers can be attached, these do the opposite of a convolutional layer, instead of shrinking the width and height of the input and increasing the depth, they enlarge the width and height and shrink the depth. These deconvolutional layers are repeated until the output is of the desired size and shape. In such an architecture, the convolutional layers can be seen as the encoder component and the deconvolutional layers as the decoder.

2.4 Related Works

In this section, we present the state-of-the-art on performing regression with convolutional neural networks and performing motion estimation using machine learning in general.

2.4.1 Estimating a Parametric Model

Bouthemy et al. [4] showed that overfitting can be a big problem for estimating parametric models. They did not make use of a neural network for estimation, instead, they opted for a modified Least Squares algorithm. They tried to estimate parameters for parametric models of different complexities. The set of models they used was similar to those shown in Table 2.1. For a frame sequence that only contained a translation for global motion, the parametric model that estimated the global motion with the highest accuracy was a model that only estimated the translation parameter. Estimations with parametric models that also described other transformations such as rotation scored much lower, likely because they overfit. Their results showed that in general, when a transformation can be described by a set of parameters, estimations with a parametric model that includes only that exact set of parameters have the highest accuracy. Estimations with parametric models that include more than that set score worse, but still comparably. However, when a parametric model lacks some of the parameters needed to describe a transformation, it scored extremely low. What we learn from this is that choosing the correct type of parametric model is very important, and that using a too complex parametric model is better than one that is too simple.

2.4.2 Regression with CNN

The output that our neural network should give is a set of numerical values. Such a problem is called a regression problem, as opposed to a classification problem, where only a label or a set of labels is given as output. Mahendran et al. [18] and Su et al. [24] both used a CNN to solve a regression problem that is somewhat related to motion estimation. The problem they tried to solve is 3-D pose estimation, where the 3-D pose of an object is reconstructed from an image. To describe the 3-D pose, they used 3 parameters, somewhat similar to our problem where we use a set of parameters to describe global motion.

What Su et al. [24] found was that a CNN trained to estimate the 3D pose of one class of objects, e.g. cars, does not perform well on other classes, e.g. bikes. They argued that this is because of the large geometric differences between the classes. To solve this, they split their network into 2 stages, the first stage consisting of only convolutional layers and the second of only fully connected layers. Then for each class of objects, the first stage is the exact same, but the second stage is unique. So the reasoning would be that each second stage should learn to estimate the 3D pose of its respective object class.

Su et al. turned the regression problem into a classification problem first by using discretized bins. Using the fact that a rotation is at most 360 degrees, they created 4 classes: (0°, 90°), (90°, 180°), (180°, 270°) (270°, 360°). Then the neural network groups each sample it sees into one of these classes, as if it were a classification problem, meaning the output would be an array the size of the number of classes and each value in the array would be the likelihood that the angle of the 3D pose falls in that bin. After this classification, the input image was rotated by the average classified amount. Then with this rotated image, a second network did a finer classification with 1 bin for every degree in the range (-45°, 45°). So if for example the first network says that the image falls in the second bin, (90°, 180°), the image is rotated by 135°. Then the second network could say the input image is rotated -23°, then that means the original input image has a rotation of $135 - 23 = 112°$.

Mahendran et al. [18] tried to solve the same problem as Su. Mahendran et al. did not turn the regression problem into a classification problem and instead perform regular regression. They did however make a few modifications to the representation of the 3-D pose and the loss function. We will not get into detail about these modifications. What is important to note is that, though their results are worse in general than those in [24], they are comparable and outperform them on certain datasets. This shows that regression using convolutional neural networks is in fact possible. Moreover, Mahendran et al. have shown that regression is possible for a problem that is comparable to the one discussed in this thesis.

The approach used by Su et al. where the regression problem is turned into a classification problem does not seem feasible for the problem of global motion estimation. While training, for each sample, they made a rough estimation of the rotation and rotate the sample accordingly to perform a finer estimation. As they only perform a rotation, doing this adjustment is quite simple, but for our problem that is not the case. If we were to adjust after making a rough estimation for a parametric model, that means for each pixel in the image we would have to compute the motion vector and then adjust each pixel with their respective motion vector. This is much more computationally heavy. For this reason, it is fortunate that Mahendran et al. have results that use regression and are comparable to those of Su et al.

2.4.3 Motion Estimation with Machine Learning

As far as we could find, no research has been done yet on global motion estimation using machine learning. However, there has been a plethora of research on general motion estimation using machine learning. In [13], Junhwa Hur and Stefan Roth give a detailed overview of the history and state of the art of motion estimation using machine learning. Specifically, they talk about the estimation of motion using optical flow, which is one of the most popular methods of estimating motion in videos. In summary, methods using optical flow assume that for 2 consecutive frames in a video, every pixel in the first frame will also appear in the second frame. This can be put in a formula as seen in Equation 2.2. Where $I(x, y, t)$ is the intensity, so the value of a pixel at location (x, y) at time t.

$$I(x, y, t) = I(x + \Delta x, y + \Delta y, t + \Delta t) \tag{2.2}$$

Before machine learning, motion estimation with optical flow was done using energy minimization, where, using some constraints, Equation 2.2 is solved. CNNs were first used both to extract features, which were then used for the energy minimization algorithms, but also on their own to estimate optical flow. The first architecture that only made use of a CNN was FlowNet [8] by Fischer et al., whose results were still worse than those of the energy-based methods. Later research showed that CNNs could be more accurate at motion estimation and faster than energy-based methods [13]. The first architecture that outperformed the energy-based methods was FlowNet2 by Ilg et al. [14]. They simply stacked multiple FlowNet networks and thereby increased the accuracy of their architecture by 50% over the original FlowNet. After this, many improved architectures were designed, such as PWC-Net [25].

A big problem for motion estimation using machine learning is a lack of labelled data. There is only a small set of labelled datasets of videos and their motion vector fields: KITTI [19], MPI Sintel [5] and Middlebury [22]. These datasets are too small to train a convolutional neural network on, as they only contain around 10-1000 images. As a solution, Fischer et al. [8] made an artificial dataset of more than $22,000$ frame pairs called Flying Chairs. As background for their frames, they took random images from an online open source image database with the tags "city", "landscape" and "mountain". As for the foreground, they used renders of 3D chair models from a public online dataset [3], to which they applied affine transformations. These are the same

Figure 2.7: *An example frame sequence from the Flying Chairs dataset [8] and the corresponding motion vector field.*

transformations that are described with the Affine parametric model, so it includes translation, rotation and zooming. They then used this dataset as input to train a convolutional neural network, called FlowNet which had as input 2 frames and had to output a motion vector field of the same size as the frames. They built 2 different network architectures: FlowNet-S and FlowNet-C. In FlowNet-S, the input frame pairs were concatenated pixel-wise as 1 large matrix of size $H \times W \times 6$. There are 6 values per element as each element is 2 concatenated pixels and each pixel contains 3 colour values. Then this matrix is put through the network. In FlowNet-C the frames are first processed separately through several convolutional layers and then these outputs are merged in a so-called "correlation" layer. This correlation layer is similar to a convolutional layer, but instead of iterating over an input matrix with a kernel with set weights, it iterates over an input matrix with another input matrix. They did not find that one architecture was significantly better than the other, only that FlowNet-C overfitted slightly more and that it had problems with large displacements.

Another solution to the dataset problem described by Hur and Roth [13] is to use a neural network that does not need labelled data, a so-called unsupervised network. They describe multiple papers that propose such networks. The first unsupervised learning technique that showed promise was developed by Ahmadi et al. [1]. As a loss function for training their convolutional neural network, they used a slightly modified Equation 2.2. By minimizing the loss function, the network learns to predict motion vector fields. However, their results were not better than that of FlowNet [8]. However, their research still suggests that unsupervised learning is a viable alternative to supervised learning, if labelled data is not available for the target domain, according to Hur and Roth.

These improvements were made by using a couple of techniques to improve the architecture. One of these is the concept of pyramids, where the frame pair is transformed to multiple lower-resolution versions, then first the CNN goes over the pair with the lowest resolution, then the one with the second-lowest resolution and so on. The CNN in this architecture can be quite simple since its task each iteration is quite simple as it only needs to refine the outcome from the last iteration. The first network that made use of this architecture, SPyNet, had 96% fewer parameters than FlowNet and comparable accuracy. The first architecture that outperformed the energy-based methods was FlowNet2 by Ilg et al. [14]. They simply stacked multiple FlowNet networks and thereby increased the accuracy of their architecture by 50% over the original FlowNet. After this, many improved architectures were designed, such as PWC-Net [25].

Chapter 3

Methodology

In this chapter, we discuss the general plan of action in this thesis. A visual overview of the methodology can be seen in Figure 3.1. In this figure, red blocks denote the separate tasks for this project. Red arrows point to the output of these tasks, and grey arrows denote the input of the tasks. Green circles denote data and Green cylinders denote sets of data.

In what follows, we briefly explain the purpose of each task. More details about the tasks and our solutions are to be found in the following chapters.For the local motion estimation task, we use the software from V-Silicon.

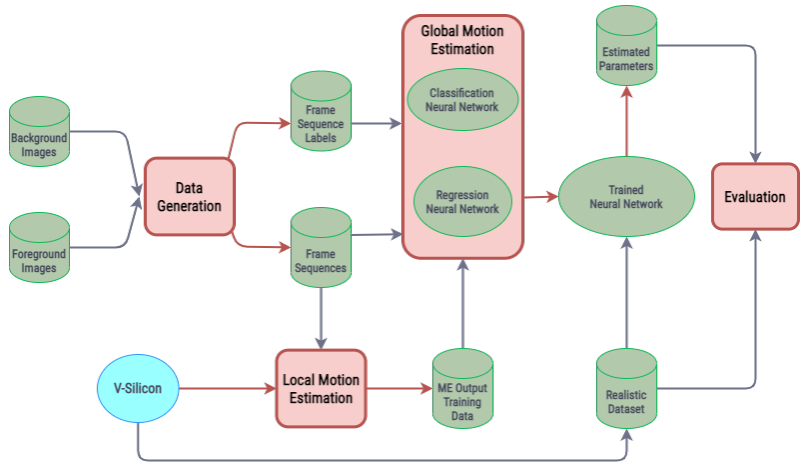


Figure 3.1: *A visual overview of the methodology.*

3.1 Data Generation

There are 2 general approaches to train a neural network, as we have seen in Subsection 2.4.3, supervised and unsupervised. Unsupervised is attractive since we do not need to find a labelled dataset, which is a big problem for supervised motion estimation neural networks. However, supervised approaches have had better results and are more common. A solution to the dataset problem has been to use an artificial dataset like Fischer et al. [8] did for FlowNet. Unfortunately, their dataset has almost the opposite type of movement that we want, as the frame sequences in their dataset contain static backgrounds, while for our problem backgrounds with motion are necessary. However, it is much simpler to create a dataset for the problem of global motion estim-

ation than for the problem of local motion estimation, for which the Flying Chairs dataset was created. That dataset needed to generate motion vectors for each pixel in a frame sequence. For our problem, we are only interested in finding a parametric model that fits the global motion in a frame sequence. So for a labelled artificial dataset for global motion, we only need to keep track of the parametric motion model that was used to create a sample. We thus decide to generate our own artificial labelled dataset.

To build this artificial dataset, we can use open source images, generate random values for parametric models, and move images according to the parametric model to obtain new frames. Then, in addition, we can add some images to act as foreground in those frames as well to make the data more realistic.

3.2 Local Motion Estimation and Global Motion Estimation

We will use the motion estimator software provided by V-Silicon to perform motion estimation on the frame sequences in the artificial dataset. The resulting motion vector fields, in combination with other data supplied by the motion estimator, would be used as input to train a neural network. Since neural networks in the literature have made use of image data as input, we test this approach as well. This is shown in Figure 3.1 as a grey arrow from Frame Sequences to the Global Motion Estimation task.

The output of the global motion estimation for a frame sequence would be the values for the parametric model describing the global motion in the sequence. Such a task where numeric values are estimated is called regression. We will perform this regression with a neural network. The output of the neural network for each sample is compared to the actual values in the parametric model that was used to generate that sample. These true values are available as the labels generated by the dataset. The difference can then be used to backpropagate the neural network, making it "learn" how to estimate global motion from its input. We will naturally have to find an apt architecture for the neural network and tune the hyperparameters for training such as the batch size and learning rate.

Moreover, we saw in the literature that Bouthemy et al. described an overfitting problem when estimating global motion with a parametric model [4]. We therefore also develop a different approach where we only estimate parameters that are believed to be necessary to describe the global motion in a sequence. So if there is no rotation occurring in the frame, the parameters describing rotation are set to 0 and no estimation is performed for those. For this approach, we build another neural network that, instead of estimating the values for the parameters in a parametric model, classifies which parameters are needed to describe the global motion. This is shown in Figure 3.1 as the Classification Neural Network. Its actual output will be a set of numeric values between 0 and 1 that represent the probability that a certain transformation is present. We then use the output of this classification neural network together with the regression neural network, to ensure the regression network does not overfit.

3.3 Evaluation

After we have trained the neural network on the task of global motion estimation, we need to evaluate the results of the neural network on a realistic dataset. Since there is no realistic dataset with ground truth for the global motion, we will use another way to measure how good the global motion has been estimated. We will use the neural network to estimate a parametric model on

samples from an unlabelled dataset with realistic frame sequences. Then this parametric model will be used to compute local motion vectors. We can then use a local motion estimation algorithm with and without these computed local motion vectors as support and measure the improvement. If the local motion vectors as support improve the accuracy of the local motion estimation algorithm greatly, then we can say that the neural network performs well enough to be useful.

Chapter 4

Building an Artificial Dataset of Videos with Global Motion

4.1 Methodology

To build a supervised neural network, we need a dataset to train on. As mentioned before, a labelled dataset with the necessary input does not exist yet for our problem, so we created this ourselves. In Chapter 3, it was explained that we want to run a motion estimator on the samples in this dataset. The input data for a motion estimator is 2 images that are supposedly consecutive frames from a video sequence, so this is what our artificial dataset should contain.

First, we find a large dataset of suitable images. Then for each image sample, we generate random values for the parameters in a global parametric model and use this parametric model to transform these images and so create an artificial frame sequence that moves via a global parametric model. This way we create our own artificial labelled dataset for global parametric model estimation. To make our dataset more realistic, we add some images to work as our foreground that does not follow the global motion. The final dataset will consist of images generated and stored in pairs. Each pair will also have a label that contains the parameters for the global parametric model that was used to make the frame pair.

4.2 Configuring the Parametric Model

4.2.1 Pixel Interpolation

To create our displaced image, for each pixel $p(x, y, n)$ with coordinates (x, y) in the frame, n we calculate a displacement vector (d_x, d_y) and then subtract that vector from (x, y) to get the location of the pixel in the previous frame $p(x - d_x, y - d_y, n - 1)$. Naturally, d_x and d_y are not always whole numbers, in that case $p(x - d_x, y - d_y, n - 1)$ is not a valid pixel position. The simplest solution [29] to this problem is to round the values $x - d_x$ and $y - d_y$ to a whole number to get the closest pixel position. This approach is known as Nearest Neighbour interpolation and results in a fairly ordinary image. However, it does cause artefacts, most commonly aliasing, and more importantly, a motion vector component in the range $[n - 0.5, n + 0.5]$, will be rounded to n and thus a considerable amount of precision is lost.

An improved way of interpolating, that is not too computationally heavy, is bilinear interpolation [15] and this is the approach we have chosen to use. A downside of bilinear interpolation is that the resulting image is blurred, but it does not have the aliasing artefacts that can be seen with Nearest Neighbour interpolation. The difference between the 2 interpolation techniques can be

(a) Interpolated bilinearly (b) Interpolated using Nearest Neighbour

Figure 4.1: *2 images created by sampling from an image by zooming in, the first image makes use of bilinear interpolation and the second of Nearest Neighbour.*

seen in Figure 4.1. In summary, it uses the nearest four existing pixels to compute a colour value. When we have an image location $p(x, y)$, the four neighbours are $p(\lfloor x \rfloor \lfloor y \rfloor)$, $p(\lfloor x \rfloor \lceil y \rceil)$, $p(\lceil x \rceil \lfloor y \rfloor)$ and $p(\lceil x \rceil \lceil y \rceil)$. How exactly the colour values of those four neighbours are mixed is dependent on the distance from those neighbours to the pixel position we are computing a colour value for. If the neighbours $p(\lfloor x \rfloor \lfloor y \rfloor)$, $p(\lfloor x \rfloor \lceil y \rceil)$, $p(\lceil x \rceil \lfloor y \rfloor)$ and $p(\lceil x \rceil \lceil y \rceil)$ respectively have the colour values a, b, c, d, then the colour value for $p(x, y)$ is computed as in Equation 4.1.

$$\begin{aligned}((1 - (x - \lfloor x \rfloor)) \cdot a) + ((1 - \lceil x \rceil - x) \cdot b) \cdot (1 - (y - \lfloor y \rfloor)) + \\ ((1 - (x - \lfloor x \rfloor)) \cdot c) + ((1 - \lceil x \rceil - x) \cdot d) \cdot (1 - (\lceil y \rceil - y))\end{aligned} \tag{4.1}$$

4.3 Parametric Model

In Table 2.1 we see that a couple of basic camera movements can be modelled: translation, zooming, rotation, panning and tilting. As seen in the paper by Bouthemy et al. [4], not being able to describe all types of possible is highly detrimental to the motion estimation accuracy. So we want a parametric motion model that can model at least these 5 types of global motion. 2 models can model extra types of motion, the planar perspective model and the full quadratic model. The full quadratic model has the highest number of dimensions and can create very complex motion fields. Though these complex motion fields might be useful in some cases, they do not occur in realistic videos. Moreover, having so many degrees of freedom might be problematic, as it would likely cause a neural network to overfit on a video that contains only a simple type of motion. The Planar Perspective model should in theory be able to model panning and tilting, as it is supposed to take perspective on a plane into account. Since this model does not make use of quadratic equations, it should also be cheaper to calculate than the regular panning and tilting models. The problem with this model however is that it makes use of a fraction. This fraction causes there to be an extra dependence between the parameters in the numerator and the denominator of the fraction, which makes it especially difficult to generate realistic values for the parameters. Moreover, what is problematic with a global motion model that models perspective is that perspective is usually quite local. In a video scene, perspective usually only applies to the surface all objects stand on, such as a floor. Objects on that floor move differently from the perspective model of the floor, based on the depth in the scene they are located. This leaves us with only 1 model that can model all the basic camera movements: the Planar Surface Rigid Motion model (PSRM), so this is the model we will use.

When discussing the parameters in the parametric model, referring to the parameters as p_1 and

p_7 is quite unintuitive, so we will use names that refer to what they do, as indicated in Equation 4.2.

$$\begin{cases} d_x(x,y,n) = t_x(n) + z_x(n) \cdot x + r_x(n) \cdot y + p_x(n)x^2 + p_y(n)xy \\ d_y(x,y,n) = t_y(n) + z_y(n) \cdot y + r_y(n) \cdot x + p_y(n)y^2 + p_x(n)xy \end{cases} \tag{4.2}$$

Where t stands for translation, z for zooming, r for rotation, and p for pan/tilting. As p_x refers to panning and p_y to tilting.

We saw in Section 2.1 that sometimes a restriction in the number of independent parameters can make the possible motions that a parametric model can generate, more natural. For example, a zooming motion by a camera always zooms in equally on the horizontal and the vertical axis. In the PSRM model, it is possible to zoom in more on one axis than the other. We could thus conclude that using a PSRM model is not as useful as one where zooming always happens equally. For this reason, when generating the dataset, where we want only realistic and natural movement, we want to restrict this. Using this notion, we update our model to have only 6 dimensions, giving us the final equation for the parametric model that we will use with Equation 4.3. Where H refers to the height of the image and W to the width. In this equation, the vertical zoom z_y is equal to the horizontal zoom z_x multiplied by the aspect ratio of the image. The vertical rotation r_y is equal to the negative horizontal rotation $-r_x$ multiplied by the inverse of the aspect ratio. The reason for why we multiply by the height and with are explained in Section 4.3.

$$\begin{cases} d_x(x,y,n) = t_x(n) + z_x(n) \cdot x + r_x(n) \cdot y + p_x(n) \cdot x^2 + p_y(n) \cdot xy \\ d_y(x,y,n) = t_y(n) + (H/W) \cdot z_x(n) \cdot y - (W/H) \cdot r_x(n) \cdot x + p_y(n) \cdot y^2 + p_x(n) \cdot xy \end{cases} \tag{4.3}$$

Implementing the Parametric Model

None of the literature, as far as we could find, describes a way to implement a parametric motion model or how to configure the parameters. So we have to find our own method for this. Firstly, we choose to normalize the coordinates of the pixels in the image so that both the x and y coordinate range from -1 to 1, with the centre pixel having value (0,0) and the top left (-1,-1). We also denormalise each parameter when calculating the displacement, meaning we multiply the horizontal parameters with the width and the vertical parameters with the height. We perform these steps so that images of different sizes, given the same parametric model, will result in the same motion.

Due to this normalization, we have to also take some extra measures when the aspect ratio of the frame is not 1:1. When we take an image where the aspect ratio is not 1:1 and we normalize the coordinates of the pixels to be in the range (-1, 1), the aspect ratio of the pixels have technically been changed. In other words, the pixels are no longer squares. In that case, to have equal horizontal and vertical motion, we also have to multiply the parameters for zooming and rotation by the respective aspect ratio. Take for instance a video where the width of the frame is twice as large as the height, in other words, the aspect ratio is 2:1. In that case, to have equal horizontal and vertical zooming z_y should be set equal to $1/2 \cdot z_x$, since z_x causes horizontal zooming, z_y vertical zooming and the width of the image is twice the height. For rotation, we have to do the same, only we multiply with the negative inverse of the aspect ratio. So for the case where the aspect ratio is 2:1 r_y should be set to $-2 \cdot r_x$. The reason we multiply with the negative inverse is that the *horizontal rotation parameter r_y is multiplied by the* vertical coordinate y in the parametric model formula.

If we took an image and performed some displacement on it, like moving each pixel 10 places to the right, then the 10 left-most columns of the new image would be empty, as the origin of these pixels would lie outside the original image. This is why we need to take as our "original" image

Figure 4.2: *An example transformation where the image is shifted half of its width to the right. In the left-most image, the red border indicates the location of the cropped image that will function as frame 1 and the blue section indicates the location of the displaced image that will function as frame 2 in the dataset.*

or first frame, a cropped out part of a larger base image, perform a transformation based on the cropped image, and if we need pixels that are not part of the cropped section, we take pixels from the original image. This approach has been visualized in Figure 4.2.

4.4 Background Images

4.4.1 Image Resolution

To know the resolution of the source images for our dataset, we need to decide on an image resolution for the generated frame sequences. Larger images require more computing power to process, but since they contain more information, they might give better results. We want to try giving directly the images as input to the neural network and first putting the images through a motion estimator.

The motion estimator that we will use, provided by V-Silicon, takes as input images of size 1920×1080. We can upscale our images to this resolution, but with upscaling, always some degree of detail is lost. Upscaling is also considerably easier if the source resolution is a multiple of the target resolution. The images in the Flying Chairs dataset [8] have a size of 512×384. This is quite close to, 480×270 which when multiplied by 4 gives us 1920×1080. So this seems an appropriate size for the samples in our dataset when used as input for the motion estimator.

We have detailed how we derived the appropriate image resolution, 480×270. This means that the background images need to be considerably larger, since we have shown that if we want to transform an image using a parametric model, we need to crop out a part of a larger image. For this reason, we want images at least twice as wide and twice as high, which means the original background image would be of size 960×540. Images that are larger or do not have a 16:9 aspect ratio are also fine, since we will cut out sections that will become our frame sequence. If an image is much larger, we could even use it multiple times, so instead of having 10,000 images of size 960×540, we would only use 5,000 images of size 1920×1080. This does not decrease the amount of memory or computation power, but it does mean that we need to find significantly fewer source images. Lastly, we want our images to be somewhat realistic. A human portrait or a photo of a butterfly are not very realistic backgrounds in a video, so just like [8] we would like images with landscapes.

4.4.2 Image Source

As the source for our images, we used Unsplash [26], a website with high quality, open-source images. We downloaded around 1000 images each, under the tags “background”, “city”, “landscape”,

"mountains", "sky", "buildings" and "clouds". These are in total 8000 images of a resolution of at least 1920×1080. As explained before, we can make 2 frame sequences per image, so we could make 16,000 frame sequences.

4.5 Configuring the Parameters of the Parametric Model

For our dataset, we want to generate random values for the parameters. However, the values we generate should be realistic so that the motion we generate could be from a natural video. To do this, we need to understand how the value of these parameters cause displacement and what the minimum and maximum value can be for each parameter. This section will thus describe the range of each parameter in our model.

For the translation parameters, it is the case that setting t_x to 1 means the original image will be moved its width to the left for the displaced image. Similarly, the displaced image is moved upwards by its entire height when t_y is set to 1. If our crop is 1/3 of the original image, that means that 1 is the maximum value for t_x and t_y and -1 is the minimum value. As values higher than this maximum or lower than this minimum, means pixels outside the original image are required. We can thus say that if we set the size of our cropped image to be $(n-2)/n$ of the original image, $-1/(n-2) \leq t_x, t_y \leq 1/(n-2)$, where n is a positive integer.

Zooming has a maximum value: 0.5. In that case, the image is zoomed in so that it only samples from 1 pixel. If we increase the value any more, the resulting image will start containing more pixels again, only now it is mirrored on the horizontal and vertical axis. The smallest value for the zooming parameter is again based on how large we crop it, it scales similar to the translation parameters, if our crop is 1/3 of the original image, z_x is set to -1, the displaced image will be equal to the uncropped image. So we can say that $-1/(n-2) \leq z_x \leq 0.5$, if our image is cropped to $(n-2)/n$ of the original image and n is a positive integer.

As pointed out before, the rotation parameter does not really rotate an image. The image is actually skewed. If the image were in fact rotated, then some value for r_x, that is not 0, should give us the original image. What happens for a high value for the rotation parameter is that not only is the image rotated, it also becomes larger. This effect can be seen in Figure 4.3. It also seems that the image can be rotated at most 90 degrees, which is only approximated with extremely high values for the parameters. When we set our crop to 1/3 of our original image, and the original image has an aspect ratio of 16:9, we find that $-0.56406250 < r_x < 0.56406250$. Currently, we have not found what caused these values. However, we can use this as a general idea of how the rotation parameters behave.

Lastly, we have the panning and tilting parameters. These behave similarly to the translation parameters. With a crop ratio of 1/3, p and t can be between -1 and 1. So we can say if we set the size of our cropped image to be $(n-2)/n$ of the original image, $-1/(n-2) \leq p, t \leq 1/(n-2)$, where n is a positive integer.

We now know, for most parameters, what their theoretical range is and how they behave. The actual range of these parameters is much more restricted than the absolute values when we take into account that we want multiple parameters to have non-zero values. For instance, if we were to set z_x to 1, we cannot change any of the other parameters, if we set z_x,t_x and t_y, to 0.5, then we cannot do any rotation. Luckily, all of these movements are much too extreme for our dataset. The standard frame rate for realistic video is 24 frames per second [21], if we take this into account, the difference between 2 consecutive frames should be quite minimal if we want to be realistic. Say we want to make a video sequence of 5 frames, that means that when t_x is set to 0.2, 20% of the second frame will consist of pixels that were not in the first frame and the fifth frame will

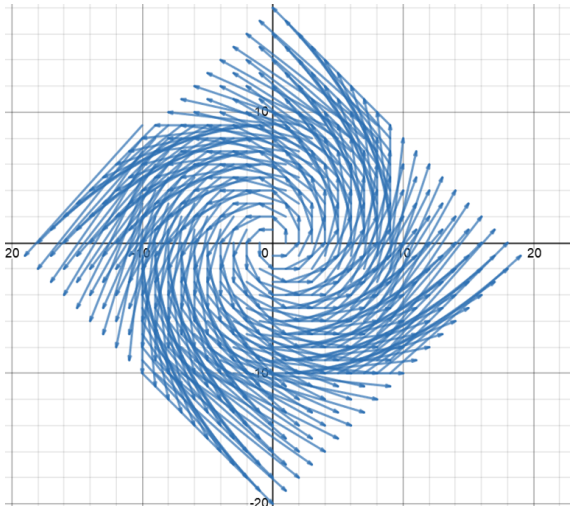


Figure 4.3: *An example of a motion vector grid showing a skewing transformation. The original points are in a grid from -10 to 10, creating a rectangle of size* 20×20. *The parametric model used for the displacement has some high values for rotation. The resulting displaced rectangle has sides that have a length of 26.*

contain none of the pixels in the first frame. Thus, for a video playing at 24 frames per second, with t_x set to $1/24 \approx 0.042$, after 1 second, the first frame of the video will be entirely gone, which is still incredibly fast. Therefore, we choose to restrict the range of the parameters a considerable amount to keep their values realistic, with this restriction comes the benefit that we should not have the problem that parts of the displaced image are empty due to motion vectors going outside the original uncropped image. Unfortunately, we do not have any way to calculate a reasonable range for each parameter. Based on video footage, we have decided on the following ranges:

- $-0.1 \leq t_x, t_y \leq 0.1$
- $-0.1 \leq z_x \leq 0.1$
- $-0.05 \leq r_x \leq 0.05$
- $-0.1 \leq p_x, p_y \leq 0.1$

For our randomized parameters, we take random values in the respective range for each parameter. More specifically, we take samples from a Gaussian distribution where we set the mean of the distribution to 0 and the σ to the maximum value described above, divided by 3. This is because 99.6% of the values in a Gaussian distribution falls in the range of $(-3\sigma, 3\sigma)$. So we can say that practically all values sampled from this distribution will fall between the ranges for the parameters we described. And since these ranges are not a hard border, it would not be too problematic for a value to fall outside these defined ranges.

Realistic videos do not always contain all these types of global motion. Usually there is only one or two at the same time, though we do not have sufficient data to know this for sure. Therefore, when generating random values for each parameter, we make it, so there is a 50% chance that the parameter is set to 0. So each possible set of global transformations should appear in our dataset, also the empty set where there is no global motion.

(a) Example of a processed image from the Chairs dataset

(b) Example of a processed image from the COIL-100 dataset

(c) Example of a processed image from the Lego dataset

Figure 4.4: *Samples from the foreground datasets*

4.6 Foreground

If our dataset would only consist of a "background" that we transform using a global parametric model, it would not be very realistic. Most realistic videos have some objects in the foreground that move differently than the background. So to make our dataset more realistic, we want to add something similar. Since motion estimation programs do not look at the shape or colour of what is moving, we can use any type of object to function as "foreground", as long as we do not transform it with the parametric model. Though, as we also want to try using the images directly as input data, we will try to have a diverse set of foreground objects.

4.6.1 Foreground datasets

In realistic videos, foreground objects can come in numerous shapes and sizes. We cannot possibly find a source of images that would cover all the possible types of foreground, still, we want a diverse set of foreground objects. Otherwise, there is the risk that the neural network learns to recognize the foreground purely by the shape of the object. We want the neural network to learn what is foreground purely by the fact that it does not follow the global motion.

We used 3 datasets for the foreground:

- the COIL-100 dataset [20], which is a dataset containing photos of 100 objects from different angles. The dataset is quite old, but the objects are quite varied, making it a good dataset for foreground objects.
- the 3D-chairs dataset, used for the Flying Chairs dataset [3], from here on called the Chairs dataset. This dataset is remarkable due to its size, as shown in Table 4.1, it contains more than 80,000 images of almost 1,400 models of different types of chairs.
- A dataset of 3D renders of Lego bricks[11], from here on called the Lego dataset. The images in this dataset are all greyscale, but this does not matter as the motion estimator used, converts any input to greyscale anyway. The images in this dataset have a high resolution and a transparent background, making it an excellent dataset for foreground objects.

Each dataset consists of images of different objects from different angles. For example, the Chairs dataset contains 1393 different chairs and of each chair there are images from 62 different angles. Samples from these datasets are shown in Figure 4.4. Preprocessing is performed on all foreground images, where the background in each image is removed.

Table 4.1 shows the size of the 3 foreground datasets. As shown in this table, the Chairs dataset is much larger than the other two. If we randomly chose images from all of these datasets, the

Dataset	N. of Images	N. of Images per Object	N. of Objects
Chairs	86,366	62	1,393
COIL-100	7,200	72	100
Lego	6,379	~ 400	16
Total	99,945	-	1,509

Table 4.1: An overview of the size of the datasets used for the foreground images.

chance that the image would come from the Chairs dataset is $\sim 86\%$. From Table 4.1 we can also see that Chairs contains only around 1400 different chairs, therefore we randomly take 5 images per chair to get a new dataset of only 7000 images, which is around the same size as the other 2 datasets. This way, when we take a random foreground image, the chance of the sample coming from each dataset is almost equal.

4.6.2 Placing the foreground

We pick random coordinates to place our foreground on the background image. We also make sure to place the same foreground images on all the frames in a generated frame sequence, as it would not be realistic if each frame in a sequence contained different foreground objects. We also generate a random small motion vector for each object, so not all foreground objects are stationary. We then use this motion vector to calculate the new position of the object at the displaced image, so the foreground object appears to be moving.

4.7 Algorithm

This section gives a short overview, we shortly summarize the process of how a frame sequence is created.

1. Select a background image I.
2. Generate random parameters $p_{1,\dots,6}$ for the parametric model.
3. Generate random coordinates $c_x^{D_1}, c_y^{D_1}$ for the crop from the background image.
4. Make the first frame D_1 by cropping out an image of size 480×270 from the I, where the coordinates of the top-left pixel are $(c_x^{D_1}, c_y^{D_1})$.
5. Generate a motion vector field $\vec{M}$ of size 480×270 with a parametric model using $p_{1,\dots,6}$.
6. Using $\vec{M}$, retrieve pixels from I, to build the second frame D_2.
7. Select a number of foreground images $F_{1,\dots,n}$ and generate random coordinates $c_x^{F_{1,\dots,n}}, c_y^{F_{1,\dots,n}}$ and motion vectors $\vec{v}_{1,\dots,n}$ for these images.
8. In D_1, place $F_{1,\dots,n}$ on the locations $(c_x^{F_{1,\dots,n}}, c_y^{F_{1,\dots,n}})$. respectively For D_2 place $F_{1,\dots,n}$ on the locations $(c_x^{F_{1,\dots,n}}, c_y^{F_{1,\dots,n}}) + \vec{v}_{1,\dots,n}$, So the original coordinates in addition to the motion vectors.
9. The 2 frames D_1 and D_2 are now complete for our frame sequences. For our labelled dataset, we now save this frame sequence together with all the relevant metadata about the frame sequence. Relevant metadata includes the file paths of the frame sequence, the parametric model parameters $p_{1,\dots,6}$, foreground images $F_{1,\dots,n}$, foreground coordinates $c_x^{F_{1,\dots,n}}, c_y^{F_{1,\dots,n}}$ and foreground motion vector $\vec{v}_{1,\dots,n}$.

Chapter 5

Global Motion Estimation

In this section, we discuss the neural network that is used to solve our problem, to perform global motion estimation. Firstly, we explain the input and output

5.1 Input of the Neural Network

In the past, the input for a neural network performing motion estimation has always been raw image data, so the colour values for each pixel in an image. This was the case for all neural networks discussed in 2, such as FlowNet [8] and PWC-Net [25]. Naturally, we want to try this approach, as it has been such a popular choice. However, we also test a novel approach where we feed the neural network as input the output of a motion estimator. The specific motion estimator we will use is the motion estimator software developed by V-Silicon, the output of this software includes different types of motion vector fields and also other data types. A complete overview of the output of the motion estimator for 2 frames is given in Figure 5.1. Further on in this section, we will explain all the terms in this figure. To obtain this output, we run the motion estimator on the frame sequences that we have generated with the approach detailed in Chapter 4. More specifically, we would want to run their lowest level Local Motion Estimation (LME). This motion estimator groups each input frame's pixels in a grid of 60×34 blocks, for each block it will output a 2-dimensional motion vector. So the resulting output will be a matrix of size $60 \times 34 \times 2$. We also want to run their Global Motion Estimation (GME) component on the 2 input frames. This module groups the pixels in a grid of 8×4 blocks, and also output a 2-dimensional motion vector for each block, so the output will be a matrix of size $8 \times 4 \times 2$.

The GME component also outputs an extra feature for each global motion block, called a "period counter". This value says how much repetition occurs in that block. Generally, it is the case that the more repetition there is in a block, the more difficulty the motion estimator has with estimating the motion for that block. Since a block with a repetitive pattern makes it harder to know how much the pixels exactly moved. For example, a chessboard has a highly repetitive pattern, in a video of a moving chessboard, it is hard to say for a motion estimator how much it was moved, since all the black squares look like each other. So we can see this period counter as a value that says how reliable the estimated motion for that block is. We can pass this period counter as input, so the neural network could have to learn that the higher the period counter for a certain block, the less accurate that block's motion vector probably is. This would make the total input equal to a matrix of $60 \times 34 + 8 \times 4 \times 3$.

The motion estimator also outputs 3 types of metrics per block, these metrics are represented as numeric values. Firstly, there is the SDD (Sum of Detail Differences), which says how much detail is in a block of pixels. It does this by looking at differences between the pixel values inside the block. Just like the local motion vectors, the SDD has a size of 60×34 and is always a positive

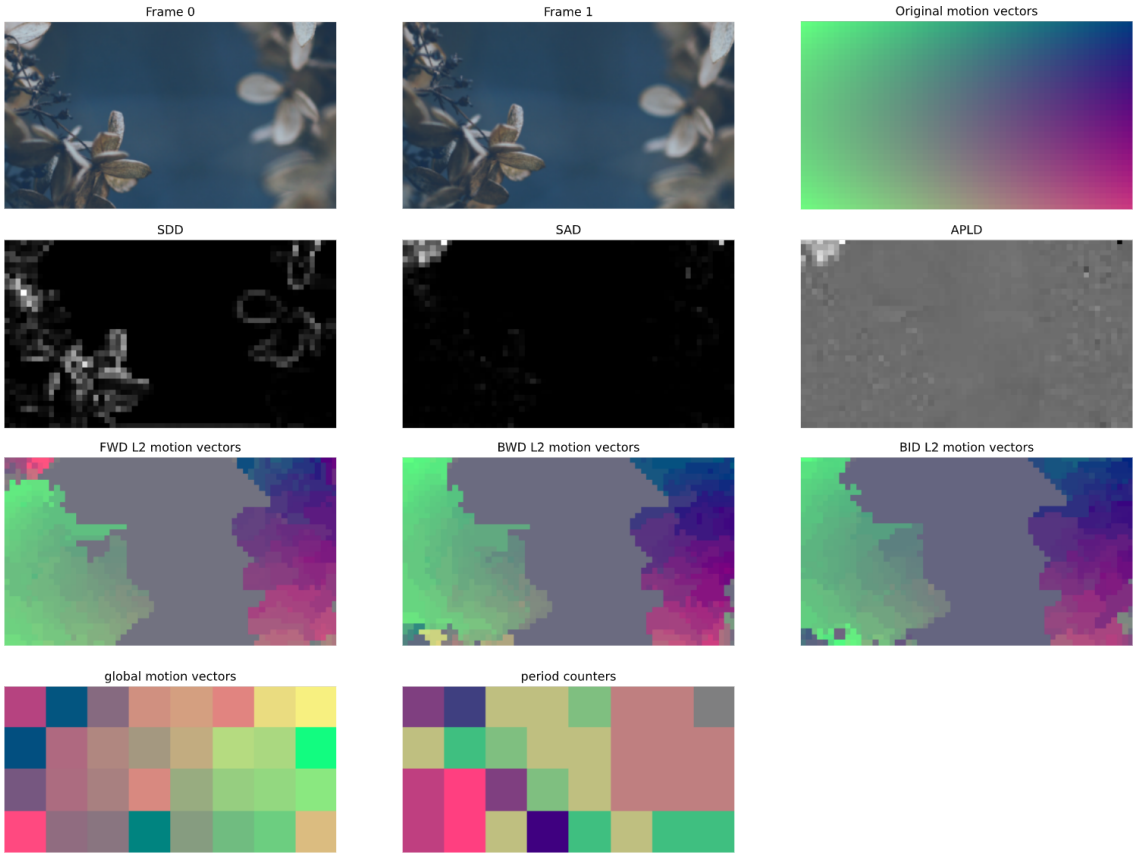


Figure 5.1: *Images showing the different types of output from the motion estimator for an example sequence. The parameters for this sample were: [0.0040, 0.0037, 0, -0.0370, 0, -0.5816*

number. Just like the period counters, the SDD can also be seen as a proxy error metric. When there is less detail in a block, that makes it harder to estimate the motion. For instance, think of an image where a large part of the image is the same colour. Since there is no detail in this area, it is hard for the motion estimator to say if there was any motion at all. The SDD values outputted by the motion estimator always come from the first frame in a sequence, so when we measure the motion from frame i to frame $i+1$, we measure the SDD for frame i. The formula to compute the SDD for a block with block size b is shown in Equation 5.1. In this equation, $p(i,j)$ denotes the colour value for a pixel with coordinates (i,j).

$$SDD = \sum_{i=1}^{b}\sum_{j=1}^{b} |p(i,j) - p(i-1,j)| + |p(i,j) - p(i,j-1)| \tag{5.1}$$

The second LME metric is the SAD (Sum of Absolute Differences), which is an error metric that tells us how well the motion vectors assigned by the motion estimator fit the frame sequence. How a motion estimator works is that it looks at 2 frames and tries to match blocks in frame i to blocks in frame $i+1$. If a block p in i at position (x,y) matches to a block q in $i+1$ at position $(x+5, y-1)$, we can say that p has a motion vector of $(5,-1)$. If block p and block q are very similar, then there is a higher chance that the motion vector is correct. To see how similar block p is to block q, we use the SAD metric, which takes the absolute difference between each pixel in p with the pixel at the same location in q and computes the sum of these differences. The formula to compute the SAD for 2 blocks p and q, where both blocks contain n pixels, can be seen in Equation 5.2.

$$SAD = \sum_{i=1}^{n} |p_i - q_i| \tag{5.2}$$

The last LME metric is also an error metric called the APLD (Average Pixel Level Difference). which works almost the same as the SAD. Instead of taking the sum of the absolute differences, it takes the absolute sum of the differences. The formula to compute the APLD for 2 blocks p and q, where both blocks contain n pixels, can be seen in Equation 5.3.

$$APLD = |\sum_{i=1}^{n} (p_i - q_i)| \tag{5.3}$$

Feeding these motion vectors and metrics to a neural network should in theory be much more insightful for motion estimation than giving as input colour values of pixels. This is because the patterns that the neural network should find in the colour values are ones that the motion estimator has already found for its own task.

5.1.1 Motion Estimator Directions

Lastly, there are 3 ways the motion estimator can compute the motion, forwards, backwards and bidirectional. If we have consecutive frames i and $i+1$, then forwards would compute the motion from i to $i+1$, backwards from $i+1$ to i. Bidirectional is slightly more complex, it can be set at a point in time between the two frames and will then compute the motion vectors from both frames to that position.

The reasoning behind these multiple directions has to do with occlusion. Say we have a video of a scene where a person is walking in front of a building from left to right, as in Figure 5.2. This means that in frame i and frame, $i+1$ different parts of the building in the background is covered. On the left side of the person, part of the building that was covered in frame i will be uncovered in frame $i+1$, shown in red in Figure 5.2(c). On the right side of the person,

(a) First frame

(b) Second frame

(c) The red area is being uncovered from the first to the second frame, the blue area is being covered

Figure 5.2: *Frames from a video showcasing occlusion.*

parts of the building visible in frame i will be covered in frame $i+1$, shown in blue in Figure 5.2(c).

Both of these areas are problematic, the area that is being covered exists in frame i but not in frame $i+1$, this means that it is impossible to assign a motion vector to this area since the pixels have no destination. The area that is being uncovered has the reverse problem, it only exists in frame $i+1$, so the motion vectors in this area have no source. However, the covered area is only problematic for motion estimation from frame i to $i+1$, called forward motion estimation and the uncovered area is only problematic for motion estimation from $i+1$ to i, called backwards motion estimation. Bidirectional motion estimation tries to perform motion estimation from both frame i and frame $i+1$ to an imaginary frame $i+0.5$. In this case, the problematic areas will be a mix of the covered and uncovered areas. Solutions to the occlusion problem try to mix and match between the results of these 3 types of motion estimation.

5.2 Neural Network

The output of the neural network for global motion estimation is different from a traditional motion estimation network. Instead of a motion vector field, we only want the parameters for the global parametric model that describe the motion from the first to the second frame. The parametric model we have chosen to generate the artificial dataset is also the one used to estimate global motion. This model uses 6 parameters, so the output of the neural network will be 6 numeric values, whose ranges are explained in 4.5. Such a problem where a (set of) value(s) is the output is known as regression. As we output multiple numerical values, our problem is more specifically known as multiple output regression. Since our output is only 6 parameters, the architecture of our network can be a lot simpler than that of FlowNet [8]. FlowNet has an encoding part consisting of a series of convolutional layers that leads to a very small set of data, and then a decoding part consisting of a series of deconvolutional layers to get an output as large as the input images. Our network will only have an encoding part, as it reduces the matrix of width 60 and height 34 to 6 values. We tried multiple famous CNN architectures, AlexNet [16], VGGNet [23], ResNet [12] and LeNet [17], but LeNet and VGGNet, specifically VGGNet-16, were the only network architecture that produced any worthwhile results. Both of the architectures for these neural networks are shown in Figures 8.1 and 8.2 in the Appendix.

5.2.1 Regression

The regression neural network has the task to output the parameter values that were used in the parametric model to create each frame sequence. As we are estimating actual values, we do not make use of an activation function at the end of the network. the Mean Absolute Error (MAE) is the metric used for the loss function to train the neural network. For a true label y and an

estimated label $\hat{y}$ of length n, the MAE is computed as in Equation 5.4.

$$MAE = \frac{\sum_{i=1}^{n} |y_i - \hat{y}_i|}{n} \tag{5.4}$$

5.2.2 Classification

As said in Chapter 3, performing classification before regression, might improve the outcome of the regression. This is why we are interested in performing classification with a neural network to find which types of global motion transformations are present in a frame sequence.

The global motion classification problem can be solved in 2 ways: as a single-label problem or as a multi-label problem. When we solve the problem as a single-label problem, we see each possible set of parameters as a class, so in total there are 64 classes. Then the output of the network is thus an array with 64 values, each representing the probability that the respective parameter set was present in the frame sequence. Solving it as a multi-label problem means that the output of the network is only an array of 6 values, where each value represents the probability that the respective parameter is present. So if for example horizontal transformation and rotation are present in the frame sequence, the label of that sequence is [1,0,0,1,0,0] for the multi-label problem. For the single-label problem, we convert the sequence to a class by interpreting the multi-label label as a binary number. So in this case, the label would be an array of length 64 with almost all zeros, only the 18th item is a one. This is because "100100" in binary is 18.

These different types of formulating the problem require different types of activation and loss functions. For the single-label problem, the activation function we use is softmax and the loss function mean cross-entropy. Softmax maps the input to a probability distribution, meaning the sum of the output array becomes exactly 1.

We make use of the cross-entropy loss function because it is used to calculate the difference between probability distributions. For a true label y and an estimated label $\hat{y}$ of length n, the formula for cross-entropy is shown in Equation 5.5.

$$CE = \frac{-\sum_{i=1}^{n} y_i \cdot \log(\hat{y}_i)}{n} \tag{5.5}$$

For the multi-label problem, we use sigmoid as the activation function and mean multiple binary cross-entropy as the loss function. The sigmoid function maps input values to a range of $[0, 1]$, which can be interpreted as a probability distribution. For the multi-label problem, the estimated and true labels are not probability distributions, instead, each individual element in the label is a probability. This is also why we use binary cross-entropy, each element is either present or not in a sample. For a true label y and an estimated label $\hat{y}$ of length n, the formula for multiple binary cross-entropy is shown in Equation 5.6.

$$BCE = \frac{-\sum_{i=1}^{n} y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i)}{n} \tag{5.6}$$

5.2.3 Solving Overfitting

We also developed an approach similar to Su et al. [24] as described in Section 2.4.2 to combat overfitting as described in Section 2.4.1. Su et al. made different endings of their neural network for each class. What we did was for each type of transformation we made a different fully connected layer, so in total there are 6 different fully connected layers. For this approach, we first perform classification on each sample, which tells us which parameters have a non-zero value. Then for the regression part, we first run the sample through the convolutional layers, then for

each parameter that has a non-zero value we run the output of the convolutional layers through the fully connected layer for that parameter. So say we have a sample with label [0.3, 0, 0, -0.1, 0, 0]. Then we first run the classifier, which should output [1,0,0,1,0,0]. Now we run the sample through the convolutional layers of our regression network and obtain the convolutional output, only 2 parameters need to be estimated, the first and the fourth, so we send the convolutional output through their respective fully connected layer. This approach was implemented on LeNet as this was the more simple neural network, therefore we refer to this approach from hereon as LeNet-Split. An advantage of LeNet-Split is that parameters that are 0 stay 0, so they do not get any value. The biggest theoretical advantage would be that each fully connected layer should "learn" its respective parameter, and thus hopefully be more precise than one fully connected layer that has to learn all parameters at the same time.

Chapter 6

Experiments & Evaluation

6.1 Neural Network Evaluation

6.1.1 Datasets

To train the neural networks, we generate multiple artificial datasets. Each dataset contains 13,000 samples, 10,000 of which are used for training and the remainder for testing. The local motion estimator from V-Silicon has been run on these datasets, so each sample in the dataset contains, next to the already existing 2 frames, also the output of the motion estimator for that sample.

The first artificial dataset has no foreground objects. We expect foreground to make the global motion estimation task more difficult, and we also want to check how well a neural network trained on a dataset without foreground performs. This is the dataset that is used in the following sections to test different input types.

We also created artificial datasets with foregrounds. Section 4.6 outlined how these foreground objects were added to the dataset. Tuning the foreground is quite complex, we can describe how many foreground objects are placed, where they can be placed and how large they are. We made 2 different dataset configurations for this reason, one with very conservative values for each of these aspects and one with more extreme values. In the conservative dataset, named *fore1*, we include in each sample 2 foreground images where the largest side of the images is at least the size of one block in our motion estimator and at most 3/4th of the height of the image it will be placed in. The second dataset, named *fore2*, has a random number of foreground objects, at most 4, and their largest side can be 3 times as large as the height of the image it will be placed in. The results with these datasets with foreground are described in Section 6.1.7.

6.1.2 Evaluation Metrics

To measure the accuracy of the classification network, we use 2 metrics, Per Sample Accuracy (PSA) and Per Parameter Accuracy (PPA). For the PSA, only if the output of the network is correct for every parameter in a sample, we see a classification as correct. The PPA looks at one parameter at a time. If the input set for the neural network is 2 sequences with respective labels [1,1,1,0,0,0], [0,0,0,0,1,1] and the network gives as output: [1,1,1,0,0,0], [0,1,0,0,1,1], then the PSA is 50%, as it has only 1 of the 2 samples fully correct, however the PPA is 92% as it has 11 of the 12 parameters correct.

To measure the accuracy of the regression network, we check how close the estimated values are to the true values, to measure this, we use the MAE, which was used also as a loss function to train the network. The equation for the MAE was given in Equation 5.4. For the classification

task, we use accuracy as a metric, which is a percentage, meaning it has a minimum value of 0% and a maximum value of 100%. MAE on the other hand is not a percentage, as the name suggests, it gives an actual value that can range from 0 to theoretically infinity. Naturally, the closer the MAE is to 0, the better. However, if the values of an MAE are computed over variables with an average value of 0.0001 then an MAE of 0.1 is equally bad as an MAE of 100 over variables with an average value of 0.1. To know thus if the regression network has a "good" MAE, we have to keep in mind the range of the global motion parameters we set, which was [-0.1, 0.1] for all parameters except the rotation parameter, which was set at half that range, [-0.05, 0.05].

6.1.3 Classification Results

Table 6.1 gives an overview of the PSA and PPA for some different sets of input data for the LeNet model with the multi-label classification task on the dataset without foreground. Only the results for LeNet are shown as for both tasks, testing LeNet and VGGNet on the same input did not give significantly different results. More specifically, the difference in PPA between the two network architectures was less than 1 percentage point for any input type. If we were to randomly classify samples, the PSA would be $0.5^6 \approx 0.0156$ and the PPA 0.5. With this in mind, we see that the neural network scores quite poorly with images as input, only having an accuracy 15 percentage points higher than the random approach. There was not enough time to test each different set of input, but we can see that there are multiple sets that perform equally well, all scoring a PPA of 88%. These results show that using the motion estimator's output as input significantly increases the accuracy of the neural network, as opposed to using image data as input.

The neural network with the single-label version of the task was tested on a few of these sets of input data, always performing worse. With only the forward motion vectors as input, it scored a PSA of 21%. With all motion estimator data as input, it scored a PSA of 35%. Both accuracies are much lower than the multi-label approach scored, so we can conclude that the multi-label approach had significantly better accuracy.

Input Data	**PSA**	**PPA**
Image	0.08	0.65
FWD, BWD	0.28	0.80
FWD	0.30	0.81
FWD, GM	0.34	0.82
FWD, GM, PC	0.35	0.85
FWD, SAD	0.44	0.86
FWD, SAD, SDD, APLD	**0.51**	**0.88**
FWD, BWD, BID	**0.51**	**0.88**
FWD, BWD, BID, SAD, SDD, APLD	**0.51**	**0.88**
FWD, BWD, BID, SAD, SDD, APLD, GM	**0.51**	**0.88**
FWD, BWD, BID, SAD, SDD, APLD, GM, PC	**0.51**	**0.88**

Table 6.1: *Accuracy scores on a test dataset from training LeNet with different input data on the multi-label classification task. The best score is shown in bold. FWD, BID and BWD stand for forward, bidirectional and backwards motion vectors respectively, GM stands for global motion vectors and PC period counters.*

6.1.4 Regression Results

An overview of different input data and the resulting MAE is given in Table 6.2. For training and testing, the dataset was the same one that was used for the classification task. The table

Input Data	LeNet	VGGNet
Image	0.01156	
FWD, GME, PC	0.00284	
FWD	0.00257	0.00164
FWD, SAD	0.00256	
FWD, SDD, SAD, APLD	0.00250	0.00162
FWD, BWD	0.00250	
FWD, BID, BWD	0.00222	
FWD, BID, BWD, SAD, SDD APLD	0.00216	0.00153
FWD, BID, BWD, SDD, SAD, APLD, GM	0.00197	**0.00131**
FWD, BID, BWD, SDD, SAD, APLD, GM, PC	0.00197	0.00133

Table 6.2: *The MAE scores from training LeNet and VGGNet with different input data. The best score is shown in bold. FWD, BID and BWD stand for forward, bidirectional and backwards motion vectors respectively, GM stands for global motion vectors and PC period counters.*

for VGGNet could not be filled fully as there was not enough time to run all configurations, and after running the experiments with LeNet it was clear which set of input data was optimal. The results show that the more input data is provided, the better the results, only the period counters did not seem to have any impact. Perhaps some of the other input data is also superfluous, but as they are all relatively small, there is no significant difference in computation time. The lowest scored MAE was 0.00131, this was achieved using VGGNet. We see that for this task, VGGNet gave significantly better results than LeNet. However, LeNet, due to its smaller size, was much faster to train on, so it was still used for testing purposes.

To gain more insight into how well the neural network performs, we defined another error metric. For this metric, we round down the true and predicted values, and check if they are equal. This way, we can compute a PSA for the regression as well. When rounding to 3 digits, the PSA was 50% for the network, with an MAE of 0.00131. Whether the resulting neural network is accurate enough to be valuable needs to be evaluated, which is done in Section 6.2.

6.1.5 LeNet-Split Evaluation

The accuracy of the network trained with the approach described in Section 5.2.3 was comparable to that of the regular regression networks described before. This means that splitting the ending into different fully connected layers does not increase accuracy. The reason that this method worked for Su et al. was that their network was not able to estimate the 3D pose of different types of input well, due to large geometric differences. In our case, especially with motion vectors as input, this is not the case, as we have not sorted our input in classes with what is shown in the frame, but instead with the type of transformation that occurs. It appears that there is not a significant problem of overfitting, as this approach with multiple ends yielded no better scores than a regular CNN.

Moreover, the speed of Lenet-Split is incredibly low. Normally a neural network does not get individual samples as input, it gets a batch of samples that have been concatenated as a matrix. Then, by using matrix operations, it can very quickly send this batch through the convolutional and fully connected layers. We still use this approach up until the fully connected layers, for those layers we have to break the batch apart and loop over each relevant parameter per sample. So not only is this slower because we cannot make use of matrix operations, but also because we have to send the convolutional output through fully connected layers multiple times. This last fact can happen as many as 6 times per sample. Another problem is that the classification can be incorrect, which causes significant errors.

6.1.6 Unused Parameter Elimination

Though LeNet-Split did not give the desired results, it had one interesting advantage over the conventional approach. Namely, the fact that for transformations that do not occur in a sample, the parameter is estimated to be exactly 0. Practically, this means that unused parameters have been eliminated in the estimation process. In the regular network, these parameters, from here on out called unused parameters, would never be set to exactly 0, they would be given a somewhat low value. We can eliminate these unused parameters by running a classification network after the regression network has estimated values. The classification network classified the unused parameters per sample, and with that information we can set those parameters to 0. This way, the error will be 0 for unused parameters. Since there are in our training set on average only three types of transformations present, this could decrease the error significantly. A downside of this approach is naturally that it slows down the estimation process, as we have to run another neural network over each sample. An example of the theoretical outcome of this approach can be seen in Equation 6.1.

$$\begin{aligned}
&\textbf{Expected:} \\
&\begin{bmatrix} 0.763 & -0.118 & 0.0 & -0.267 & 0.0 & -0.502 \end{bmatrix} \\
&\textbf{Prediction:}(MAE = 0.0572) \\
&\begin{bmatrix} 0.871 & -0.108 & 0.056 & -0.252 & 0.067 & -0.415 \end{bmatrix} \\
&\textbf{After classification:}(MAE = 0.0367) \\
&\begin{bmatrix} 0.871 & -0.108 & 0.0 & -0.252 & 0.0 & -0.415 \end{bmatrix}
\end{aligned} \tag{6.1}$$

This approach was tested during training of the LeNet model. The classifier used had a PPA of 88%. The final MAE of the regression network was 0.0210 before eliminating unused parameters and 0.0181 afterwards, so a reduction of around 14%. Figure 6.1 shows how the difference in MAE changed over time during training.

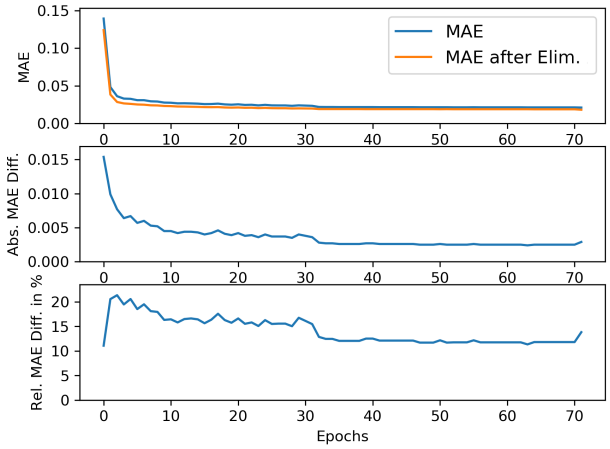


Figure 6.1: *3 Plots showing how classification after regression can reduce the MAE. The first plot show the MAE over time before and after classification. The second plot shows the absolute difference at each moment and the third plot shows the relative percentual difference.*

However, when this approach was tested on networks making use of the VGGNet architecture, which has a lower MAE than the LeNet architecture, it was found that the elimination step can

MAE	After Elim. MAE
0.00246	0.00228
0.00193	0.00189
0.00160	0.00166
0.00167	0.00183

Table 6.3: *The results from running differently trained neural networks on an evaluation dataset. The first column shows the MAE from just only perform regression, the second column shows the MAE after eliminating unused parameters with a classification network.*

increase the MAE. This is likely due to the error of the classification network, as this network only has a PPA of 88%. So when it wrongly classifies a parameter, that can increase the error considerably. For example, when a parameter value should be 0.8, and the classifier believes that parameter to have a value of 0, then for that parameter the error will be 0.8, which is almost 50 times as high as the MAE in that case. So it makes sense that when the MAE of a network gets to a certain level, errors made by the classifier become so costly that it increases the MAE. An overview of this effect is presented in Table 6.3, to train neural networks with different MAE scores, we trained different architectures, some LeNet some VGGNet, with different sets of input. The results indicate that for an MAE below ~ 0.0019 eliminating unused parameters is not advantageous.

6.1.7 Foreground Results

The idea behind adding foreground is that it makes the dataset more similar to realistic video, since realistic videos contain both background and foreground objects, where the background generally moves according to a global parametric model and the foreground does not. Since our neural network should find the global parametric model, that means it should only pay attention to the background. With the artificial dataset with foreground, the neural network should thus learn that it cannot "trust" all the input data to contain information on the global parametric model. This means naturally that adding foreground should result in a higher error relative to a neural network trained on a dataset without foreground, as the foreground makes estimating the parametric model more difficult. Though hypothetically, the neural network should thus learn to handle frame sequences with foreground and thus this should result in a lower error on a realistic dataset, as natural video scenes also contain foreground.

As shown in Section 6.1.1, we have 3 datasets, one without foreground and two with foreground. We trained 3 identical neural networks on these datasets. The neural network that was used, was the VGGNet model with as input type the set that gave the best score in Table 6.2. Afterwards, we ran each neural network on a verification dataset for each different type of dataset. The results of the neural networks on these verification datasets are shown in Table 6.4. The dataset where samples have no foreground is given the name *no fore*. We see that fore1 is on average a more difficult dataset to estimate global motion parameters for than no fore, and fore2 is, in turn, more difficult than fore1. This supports our hypothesis, where we said that foreground makes the estimation task more difficult, and we see that indeed datasets that contain more foreground have higher error scores. Which neural network will perform better on a realistic dataset will be shown in Section 6.2.

Dataset\Neural Network	no fore	fore1	fore2	Average
fore2	0.0054	**0.0045**	0.0043	0.0047
fore1	0.0045	0.0022	**0.0034**	0.0034
no fore	0.0013	**0.0016**	0.0024	0.0018
Average	0.0037	0.0028	**0.0034**	

Table 6.4: *The MAE results from running some differently trained neural networks on different evaluation datasets. Numbers in red are the best scores for that row, numbers in bold are the second best.*

6.2 Evaluation on Realistic Video

The best way to evaluate the usefulness of the neural network approach is to incorporate it into the motion estimator developed by V-Silicon. However, this piece of software is complex, and cannot easily be modified. Instead, we opted to build a simple motion estimator that makes use of the same algorithm as the motion estimator from V-Silicon. The algorithm in question is 3DRS, short for 3-Dimensional Recursive Search and first developed by De Haan et al. [7]. How this algorithm works will be explained in Section 6.2.1. In Section 6.1.1, we defined 3 different datasets and trained 3 neural networks on these datasets. We will test all of these neural networks in this chapter. We run the 3DRS algorithm with 3 different approaches: 1) Without using the global parametric model 2) With the true parametric model, 3) With the parametric model estimated by a neural network. The first approach gives us a lower bound on the performance of the neural network, and the second approach gives us an upper bound. We can then evaluate the third approach by analysing whether it is closer to the first or second approach. We can also run 3DRS on a dataset that is not artificial to see how well the neural network operates on realistic video scenes. Naturally, for such a video, there is no known true parametric model.

6.2.1 3DRS

3-Dimensional Recursive Search is a local motion estimation algorithm, meaning that it assigns a motion vector to each pixel in an image. In this section, we will describe how this algorithm works.
The first step in the motion estimation process is to change the image from a 3-channel RGB format to a 1-channel format. This reduces the computation time considerably, and the amount of lost information is insignificant. Then we divide the frame into blocks of a certain size b. We will only perform motion estimation for each block, so all pixels in a block will be assigned the same motion vector. Thus, the larger the block size, the fewer computations we perform, but also the less precise our motion vectors will be. For any object in a video frame it will be the case that all parts of that object have approximately the same motion vector, so as long as one block does not contain multiple objects, theoretically, it is not problematic that all the pixels in a block get the same motion vector.

Then, for each block at location (n, x, y), where n is the frame number, x the horizontal location and y the vertical location, we perform the following process. We collect the motion vectors from a few of its spatial and temporal neighbours, a spatial neighbour is a block in the same frame and a temporal neighbour is a block in the previous frame. The specific neighbours used are up to preference, so the neighbours we have chosen in this section are not the only ones that can be used. The spatial neighbours we use are, for instance $(n, x-b, y-b)$ and $(n, x+b, y-b)$, coloured dark red in Figure 6.2. An example of a temporal neighbour would be $(n-1, x+2b, y+2b)$, coloured dark green in Figure 6.2. For this temporal neighbour, it will thus take the motion vector from the block at this location in the previous frame.

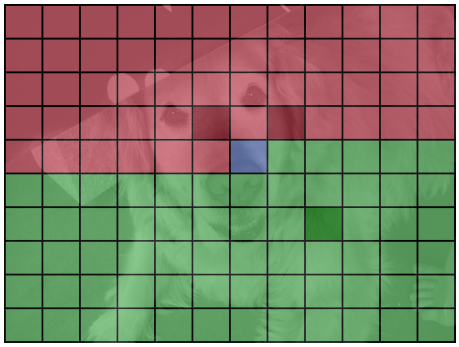


Figure 6.2: *The locations of spatial and temporal neighbours when performing 3DRS. Motion estimation has already been performed on red blocks and still has to be performed on green blocks. The blue block is where currently motion estimation is being performed.*

So we collect from each neighbour the motion vector that has already been assigned to them. Then we create a set of new motion vectors, known as the candidate set, by adding a randomly generated small vector called an update vector to the vectors of the neighbours. So, for example, the motion vectors of our respective neighbours are $(1, 2)$, $(1, 3)$ and $(2, 5)$. We want to make 3 candidates per neighbour, so for the first neighbour's motion vector we could have the update vectors $(0, 0)$, $(1, 0)$ and $(2, 0)$. That means our candidate set after going over the first neighbour is $(1, 2), (2, 2), (3, 2)$. This process is repeated for each of the neighbours, so the candidate set will contain in total 9 motion vectors. We expand this candidate set by one vector: the zero vector $(0, 0)$, this vector means there is no motion. In case we have performed global motion estimation, we can also expand the candidate set by a global candidate as well.

Now that we have our candidate set, we evaluate these vectors and assign the best one to our block. We do this by finding the block at the end of our motion vector in the next frame when we set the current block to be the origin of the motion vector. This block in the next frame is compared to our current block, with a metric called SAD that measures how closely the blocks resemble each other. The lower the SAD, the more similar the blocks are. So for block (n, x, y) and motion vector $(1, 2)$, the destination block is $(n, x + 1, y + 2)$. Then we compute the SAD for the blocks (n, x, y) and $(n, x+1, y+2)$. The SAD is calculated by taking the absolute differences of the pixel values at each location and summing them. So if the blocks are identical, the SAD will be 0. The maximum value for the SAD is 255 times the number of pixels in a block. This occurs when the pixels in one block all have a value of 0 and the pixels in the other block all have a value of 255.

Once we have calculated the SADs for each vector in the candidate set, the motion vector with the lowest SAD is assigned to the current block. This process is performed for every block in the image.

6.2.2 Implementing 3DRS

We have to make a few modifications to the 3DRS implementation, so it runs well for our specific problem. Since our dataset sequences only have 2 frames, we only have spatial neighbours, no temporal neighbours. For the spatial neighbours, we use the ones described in the section above: $(n, x - b, y - b)$ and $(n, x + b, y - b)$.

The 3DRS algorithm generates random update vectors when creating the candidate set. This means that if we run the algorithm twice on the same frame sequence, the resulting motion vector fields will differ. Due to this, running the algorithm multiple times give different results. This difference can be as large as 0.03. For this reason, we have seeded the algorithm during testing, so that for each time we run the motion estimation algorithm the same update vectors are chosen.

Block size	Downscale ratio	N. updates	Mean SAD
2	4	3	37.02
2	8	3	37.40
2	8	6	32.98
8	4	3	969.66
8	8	3	1018.57
8	8	6	988.31

Table 6.5: *The mean SAD results from running 3DRS on one image sample of size* 480×270.

Nonetheless, this still means that if different update vectors were chosen, the results might have been different. For this reason, when we see a difference smaller than 0.03 in the results, we can say that the difference is almost insignificant.

Furthermore, there are a few variables that we have to set a constant value for. There are the block size, the downscale ratio and the number of update vectors. The downscale ratio should not have an impact on the accuracy, it only means the resulting motion vector field will contain fewer blocks. However, block size matters a considerable amount, the larger a block size the more pixels are assigned the same vector, in general this means the average SAD increases with the block size. The number of update vectors has a similar effect, since the more update vectors, the more possible vectors we try, which in turn results in a higher chance of finding a more fitting vector for a block. Naturally, decreasing the block size and number of updates and decreasing the downscale ratio results in a lower computation cost, so the trade-off for each variable is accuracy for speed. We ran the 3DRS algorithm with different settings for each variable, the results are shown in Table 6.5. The results in this table have been grouped by block size, as this had the largest impact. In this table, we see the same relations we already described between the mean SAD and the variables, though for a block size of 8, downscaling seems to have a significant impact as well.

From these results, We decided to set the block size to 2×2 and the number of updates to 6. These are relatively high values, but since these 2 variables can impact the accuracy considerably this is still desirable, so accuracy reduction is minimized. In turn, we set the downscale ratio such that the size of the frames is 135×240, so for an image that is 1920×1080 the downscale ratio will be 32.

6.2.3 3DRS Results

Table 6.6 and Figure 6.3 show the results from running the 3DRS algorithm with the neural networks as support. To generate these results, we run the 3DRS algorithm on all samples in the datasets. For each sample, we take the mean of the SADs from all the motion vectors. Then to get the final value for each dataset, we take the mean of these mean SADs.

The realistic dataset, whose results are shown in the fourth row, consists of 350 frames taken from 8 video samples provided by V-Silicon that contained clear global motion. The other datasets are the same ones used in Table 6.4. When we compare the results per dataset, it seems surprising that the realistic dataset has a much lower SAD than the artificial datasets. However, this makes sense as the artificial datasets contain samples where the frames have been warped considerably more than what is usual in realistic videos. The most common type of motion is translation, where something on the screen is just moving left, right, up or down. When we find a perfect motion vector for translation from a certain frame to the next, the SAD will be low, as with this motion vector the block in the first frame and the second frame will be identical. However, for a sample with a rotation, a block in a certain frame will not exactly exist in the second frame. Thus, we will in general have a higher SAD for a sample with a zooming transformation than a sample with

Dataset	No GM	Orig. GM	Approach [6]	V-Sil.	no fore	fore1	fore2
fore2	30.70	20.85	32.43	27.55	**22.30**	21.67	22.32
fore1	35.53	23.86	37.15	31.98	**25.80**	24.66	25.83
no fore	34.30	20.86	37.28	30.21	21.33	**21.59**	22.38
realistic	17.94	-	16.62	17.82	15.84	**15.65**	15.58

Table 6.6: *The mean SAD results from running 3DRS with global candidates provided by differently trained neural networks on different evaluation datasets. Each column had a different source of global candidates. Numbers in red are the best scores for that row, numbers in bold are the second best. The No GM column shows the results for performing local motion without global motion support. For the Orig. GM column, global motion support is provided by the actual global motion vectors used to generate each sample, naturally this is not available for the realistic dataset. The V-Sil. column shows the results where V-Silicon's global motion approach was applied. For the Approach [6] column, global motion support was generated using the approach proposed in paper by Gerard de Haan on parametric motion models [6]. The other columns all had the global motion support generated using a convolutional neural network approach, where the no fore network was trained on a dataset without foreground. The fore1 network and fore2 network were trained on the the fore1 dataset and fore2 dataset respectively.*

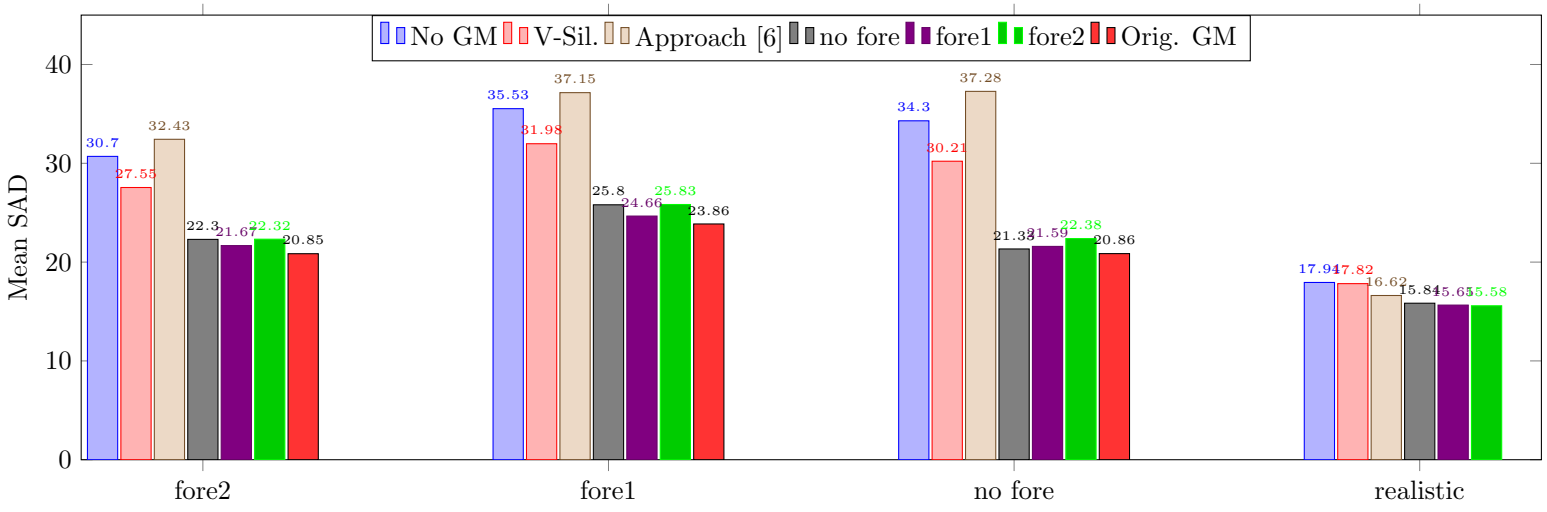


Figure 6.3: *Bar chart diagram with the results from Table 6.6. The blue bars are the results for performing local motion without global motion support. The pink bars are the results where V-Silicon's global motion approach was applied. For the beige bars, global motion support was generated using the approach proposed in paper by Gerard de Haan on parametric motion models [6]. The grey, purple and green bars all had the global motion support generated using a convolutional neural network approach, where the no fore network was trained on a dataset without foreground. The fore1 network and fore2 network were trained on the the fore1 dataset and fore2 dataset respectively. For the red bar, the global motion support is provided by the actual global motion vectors used to generate each sample, naturally this is not available for the realistic dataset.*

a translation, even if we have perfectly matching motion vectors for a block. To be sure this table did not contain any unusual outliers, we ran the motion estimation multiple times with a different seed to make sure the results were consistent.

The first column in Table 6.6 shows the mean SAD when we do not include any global candidates, these give for each dataset the highest SAD score, showing that providing global candidates always improves the results. The second column shows the results when we provide global candidates, where the global candidates are generated from the true global parametric model. Naturally, this is only possible on the artificial datasets. Unsurprisingly, the scores in this column are always the lowest, as the supplied global candidates match exactly. The reason that the SAD is not 0 is that we cannot move pixels exactly according to their motion vectors when generating the frames. These 2 columns give a respective upper and lower bound for the motion estimation results.

The third column denoted with the title "Approach [6]" contains the results where the global candidates were computed using the approach developed by De Haan and Biezen [6]. This is a heuristic approach that uses a parametric motion model that has 4 variables, for horizontal translation, vertical translation, horizontal zoom and vertical zoom. We observe that this approach does not score particularly high on the artificial datasets. This is likely because the parametric model used in this approach is not able to properly describe rotation, panning or tilting. As such, frames including these motions will not be able to be described well. One may notice that this approach performs well, though, on the realistic dataset, in which its produced results are better than the results for V-Silicon, but worse than those for CNN.

The fourth column contains the results, where the global candidates are provided by the global motion estimation currently in use by V-Silicon. The fifth to seventh columns show the results where the global candidates were provided by different neural networks. Each column header refers to the dataset the neural network was trained on. The results from the neural networks are significantly better than those from V-Silicon and De Haan and Biezen, even coming quite close to the results with the original global motion. Unlike in Table 6.4 where each neural network performed best on its respective dataset, this is not the case here. We see that the network trained on fore2 is not the best on its respective dataset, it performs about equal to the network trained without foreground and the network trained on fore1 performs best. The reason for this is not known. From the artificial datasets, we would say that the network trained on fore1 is the best, which was also shown in Table 6.4. However, we see that it is the network trained on fore2 that performs best on the realistic dataset. This likely means that the fore2 dataset is most similar to natural videos. This seems reasonable, as the fore2 dataset contains a considerable amount of foreground, whereas fore1 only contains quite small foreground objects.

6.3 Masking Foreground

Results when purely looking at the SAD metric seem very good, as using global candidates generated by the neural networks decreased the average SAD considerably. However, when looking at the exact output for some individual samples, we see something problematic. Figure 6.4 shows the motion vector field generated from the parametric model estimated by the neural network on the fore2 dataset and the motion vector field generated by the local motion estimator from V-Silicon. We see that there is a foreground object moving with the yellow vector field, and the background is moving with the purple vector field. Since the neural network is supposed to only capture the motion of the background, it should have just given only a purple vector field, but this is not the case, instead, we see a vector field where the centre is coloured yellow and the outside blue and purple. It seems like the neural network generated a vector field that blends the motion of the foreground and the background. We already expected this problem to occur, and the hypothesis was that adding foreground in the dataset should teach the network to only pay attention to the

(a) Motion vectors, estimated by the motion estimator from V-Silicon.

(b) Motion vector field generated from the parametric model estimated by a neural network.

Figure 6.4: *Motion vector fields generated from a sample from a realistic video.*

background, but the network used for this result was already trained on the fore2 dataset, so it seems that did not work for this sample.

To try to solve this issue, we had a plan with 2 steps. Firstly, we find a way to find the location of foreground and background in a frame sequence. Secondly, we give this information in some way to the neural network, so it would only pay attention to the background. Note that we want the neural network to pay attention to global motion, but to make matters easier to understand, we refer to areas with global motion as background and other areas as foreground.

To build a mask, we first thought of using a neural network. Since we already had a large artificial dataset with foreground which we could use for this. For each frame we know the exact coordinates of the foreground, so we can easily build a mask where for each pixel we give it a value of 0 if it contains foreground and a value of 1 if it does not. The input used is the same as for our neural network, so it includes all the metrics from the motion estimation. Since this input has a shape of, 34×60 we also resize the output mask to be of this shape. This approach yielded fairly good results on the artificial datasets. However, on the realistic dataset, this method gave quite mediocre results as shown in Figure 6.5.

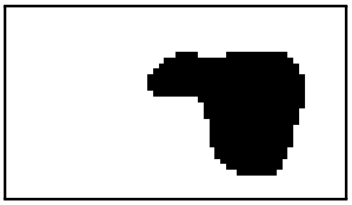
Finding that a neural network approach, at least this one, did not give satisfactory results, we tried looking for a heuristic approach. We developed the algorithm shown in 1 that is both very fast and also quite simple. The parameters δ and ε determine the thresholds for what is and is not included in the mask. We came to set these parameters to respectively 8 and 144, which gave the most satisfactory results. The resulting masks from this algorithm are not perfect, as shown in Figure 6.5e, mostly because the input motion vector field is not perfect, but it has much better accuracy than the neural network approach. We can only compare the heuristic method to the neural network visually, as we have no access to a dataset with true background masks. Though it is clear from analysing samples such as in Figure 6.5 that the heuristic approach is superior to the neural network approach.

Second came incorporating this mask into the neural network. Eppel tried a very similar task with image recognition [10]. As input for his neural network, he has images and the neural network outputs a class, for example “person” or “elephant”. He then introduces a mask that, similar to our mask, has a 1 for the region of interest and a 0 for other pixels. He then tried 7 different approaches of incorporating the mask into the neural network. For example, one approach is multiplying the mask with the image before inputting the image in the neural network. Another approach first passes the image through some layers of the network before multiplying. In their results, it was shown that although some approaches gave better results than others, the simplest approach of multiplying the mask with the image beforehand still gave an accuracy of 70%. Other methods had accuracies ranging from 73-83%. Since trying the more difficult but more accurate approaches would mean adapting the neural network, we opted for trying this simplest approach since it was much easier to adapt and if it were as effective as in Eppel’s study we could try the

Algorithm 1 Building a background mask from a motion vector field

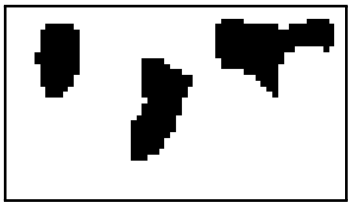
```
1: function checkVectorDifference(u, v, count)
2:     if |u_x − v_x| + |u_y − v_y| < δ then
3:         count ← count + 1
4:     else
5:         count ← 0
6:     end if
7:     return count
8: end function

9: function BackgroundMask(motion vector field V of size (n, m))
10:    HoriCount ← [n, m]
11:    VertCount ← [n, m]
12:    Mask ← [n, m]
13:    for y ← 0 to n do
14:        count ← 0
15:        for x ← 1 to m − 1 do
16:            count ←checkVectorDifference(V[y, x], V[y, x − 1], count)
17:            HoriCount[y, x] ← count
18:        end for
19:        count ← 0
20:        for x ← m − 1 to 0 do
21:            count ←checkVectorDifference(V[y, x], V[y, x + 1], count)
22:            HoriCount[y, x] ← count + HoriCount[y, x]
23:        end for
24:    end for
25:    for x ← 0 to m do
26:        count ← 0
27:        for y ← 1 to n − 1 do
28:            count ←checkVectorDifference(V[y, x], V[y − 1, x], count)
29:            VertCount[y,x] ← count
30:        end for
31:        count ← 0
32:        for y ← n − 1 to 0 do
33:            count ←checkVectorDifference(V[y, x], V[y, x + 1], count)
34:            VertCount[y, x] ← count + VertCount[y, x]
35:            if HoriCount[y, x]+VertCount[y, x] > ε then
36:                Mask[y, x] ← 1
37:            else
38:                Mask[y, x] ← 0
39:            end if
40:        end for
41:    end for
42:    return Mask
43: end function
```

(a) A sample from the artificial fore2 dataset

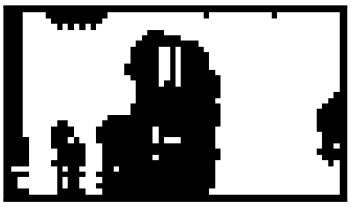


(b) Mask by neural network on sample from (a)

(c) A sample from the realistic dataset



(d) Mask by neural network on sample from (c)



(e) Mask by algorithm 1 on sample from (c)

Figure 6.5: *Background masks generated by a neural network trained on the fore2 artificial dataset.*

more difficult approaches to see if those were more effective for us as well. However, trying this simplest approach gave very bad results. It achieved an MAE of 0.0078 on the fore2 dataset, which is 66% worse than the average shown in Table 6.4. For samples with a high level of foreground, the newly trained network estimated the parameters much too low. What seemed to happen is that, since multiplying with the mask gives a large number of zero vectors, the network thinks that there is barely any motion. Since this approach already gave such bad results, we did not try the approaches that were more difficult to implement.

Instead of trying to train a network to only pay attention to the background, we then tried a more simple approach, where we try to "fill" the foreground from the vector fields with background motion vectors. For this, we use a simple algorithm where we loop over the vectors in a motion vector field 4 times, from different directions. For each block that is not part of the background according to the background mask, we assign it a new value equal to the median of the vector value from 3 neighbours in the opposite direction of which we go through the vector field. For instance, when we move through the vector field from top to bottom, left to right, we look at the top neighbour, the left neighbour and the top-left neighbour. As said, we go through the motion vector field 4 times, resulting in 4 different motion vector fields, which we combine by taking the median.

The result of this "fill" algorithm can be seen in Figure 6.6b. From here on, we will refer to this new motion vector field as the filled motion vector field. Figure 6.6c shows the output of the neural network trained on the no fore dataset with the filled motion vector field from Figure 6.6b as input. We see that now the neural network indeed gives a fully purple vector field, matching much closer to the actual background motion vectors than the vector field shown in Figure 6.4. We can also numerically show that the neural network indeed looks more at the background motion vectors with this approach. We generate local motion vectors from the estimated parametric models, the same way global candidates are generated for the 3DRS evaluation process. Then we compare these local motion vectors to the local motion vectors outputted by the motion estimator from V-Silicon. However, we only compare the motion vectors at blocks that are part of the background, according to the background mask. We compare the motion vectors by computing the vector difference, which is done with Equation 6.2, where $\hat{u}_x$ and $\hat{u}_y$ denote the x and y components of

(a) Original motion vectors

(b) Motion vectors with foreground removed

(c) Motion vector field generated from a parametric model, estimated by a neural network run on the filled vectors.

Figure 6.6: *A motion vector field, estimated by the motion estimator from V-Silicon. In the second image, we have run the algorithm to remove the foreground vectors.*

no fore	fore1	fore2	no fore-F	fore1-F	fore2-F
1.71	1.86	1.95	1.59	**1.68**	1.70

Table 6.7: *The median vector differences when running some differently trained neural networks on the realistic dataset. The median vector length for this dataset was 6.83. Numbers in red are the best scores, numbers in bold are the second best. The no fore network was trained on a dataset without foreground. The fore1 network and fore2 network were trained on the the fore1 dataset and fore 2 dataset respectively. The networks suffixed with "-F" are the same networks as the ones without the suffix with the same name, except that they got as input the filled motion vectors for each sample.*

the vector $\hat{u}$ respectively.

$$\Delta(\hat{u}, \hat{v}) = |(\hat{u}_x - \hat{v}_x)| + |(\hat{u}_y - \hat{v}_y)| \tag{6.2}$$

The resulting median vector differences with this approach are shown in Table 6.7. The reason we take the median and not the mean is that the average vector length differs considerably between samples. A sample with a high amount of motion will have larger motion vectors than one with less motion. The networks suffixed with "-F" had as input the filled motion vector fields. We see that these neural networks perform better than their counterparts. Taking into account the median vector length, the differences in median vector difference are quite significant between the neural networks that used the filled motion vector fields and those that did not. As the median vector difference is consistently lower for the networks that used filled motion vector fields, we can say that they match the background motion vectors better and thus that our approach indeed worked. Especially if we take into account the median vector length for the dataset, the improvement is quite significant.

6.4 3DRS Results for Filled Motion Vectors

Now that we have managed to make the neural network "focus" on the background motion instead of the foreground motion, we would expect that giving the parametric models estimated with these filled motion vectors to the 3DRS algorithm would result in a lower error, as the background should take up more space in each sequence than the foreground. However, when looking at the results from running 3DRS as shown in Table 6.8. Some of the results already shown in Table 6.6 have also been added again for comparison. From the results in this table, we cannot draw a conclusion that using the filled motion vectors improves the results. For the neural networks trained on no fore and fore2, the results become worse, but for the neural network trained on fore1, they become better. It could be that the neural networks without the filled motion vectors

perform better for some neural networks, because, as we saw in Figure 6.4, that the parametric model models both the background and foreground. In that case, it might give correct motion vectors in more locations than a parametric model that only models the background.

No GM	V-Sil.	no fore	fore1	fore2	no fore-F	fore1-F	fore2-F
17.94	17.82	15.84	15.65	**15.58**	16.00	15.52	15.85

Table 6.8: *The mean SAD results from running some differently trained neural networks on the realistic datasets. Numbers in red are the best scores, numbers in bold are the second best. The No GM column shows the result where no global motion support was added. The V-Sil. column shows the result where V-Silicon's global motion approach was applied for support. The others all had the global motion support generated using the convolutional neural network approach, where the no fore network was trained on a dataset without foreground. The fore1 network and fore2 network were trained on the the fore1 dataset and fore 2 dataset respectively. The networks suffixed with "-F" are the same networks as the ones without the suffix with the same name, except that they got as input the filled motion vectors for each sample.*

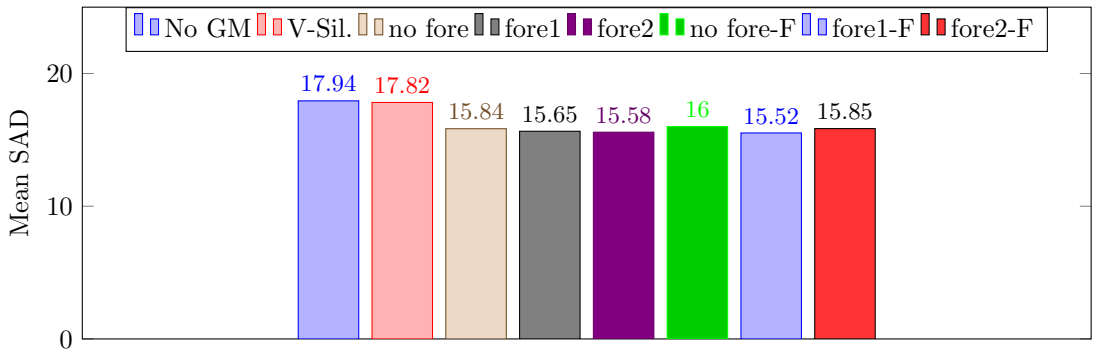


Figure 6.7: *A Bar chart diagram with the results from Table 6.8. The blue bar shows the result without global motion support. The pink bar is the result where V-Silicon's global motion approach was applied. The beige, grey, and purple bars all had the global motion support generated using the convolutional neural network approach, where the no fore network was trained on a dataset without foreground. The fore1 network and fore2 network were trained on the the fore1 dataset and fore2 dataset respectively. The green, light blue and red bars have the same networks as the beige, grey and purple ones, respectively, except that they got as input the filled motion vectors for each sample.*

Chapter 7

Conclusion

In this section, we will first summarize the findings in this thesis, and second we will discuss the used approach and give direction to possible future research.

7.1 Summary

In this thesis, we have studied whether we could estimate the global motion in a video scene with a neural network. For this purpose, we have developed a method to create an artificial dataset and used this dataset to train a neural network. Using the local motion estimation results from the software provided by V-Silicon as input, our neural network estimated the parameter values with a mean absolute error of 0.0013, where the range of the parameter values was between -0.1 and 0.1. Verifying our results on a dataset of real video scenes, we have shown that using the estimated global motion as a support for 3DRS, a local motion estimation algorithm, significantly decreased the error of the local motion estimation.

We have also shown that with simple techniques we could create a background mask and use that to erase foreground motion vectors in a motion vector field. This was necessary as the neural network's global motion is sometimes a mix between the foreground and background motion, while we want purely the background motion. Using the new vector field as input made the neural network improve at estimating the background motion, but when used to support 3DRS, the error was on average slightly higher than using the original motion vector fields.

7.2 Discussion and Future Research

In this thesis, we have shown that we can in fact build a neural network that can closely estimate the global motion in a frame sequence. However, we have seen that it is not flawless. As was shown in Figure 6.4, the network does not always model the global motion, which in the case of Figure 6.4, is the background motion. These flaws are likely caused by differences between the artificial dataset the neural network was trained on and real video scenes. If the artificial dataset could be improved, to contain more complex types of foreground that for example move in 3D or to contain layered backgrounds, the neural network could be improved even more.

In Section 6.3, we used some relatively simple processing steps in an attempt to erase foreground motion vectors from the estimated motion vector fields. With this step, it could be said that the task for the neural network has been made almost effortless when it only has the task of finding a parametric motion model for a frame sequence only containing background motion. A least-squares approach, as in [4], might work equally well if not better in this case. Such an approach

would have the advantage of having much less computational cost than our machine learning approach.

The realistic dataset supplied by V-Silicon could also be problematic, as it was a relatively small, manual selection of sequences that contained global motion. There might be a bias in the current realistic dataset which, would be less likely to exist if we had evaluated with a larger dataset of real video scenes.

Consistency over time is also a factor that could not be addressed in this thesis. When we have a frame sequence of more than 2 frames, and we first give frame 1 and 2 to the neural network and then frame 2 and 3, the output of the neural network for both frames should be similar. Evaluating whether the neural network was consistent over time was not possible, as there was no realistic data with known global motion. The output for the first sample could also be useful for estimating the parameters for the second sequence. Since the neural network does not have a concept of memory or allow for more than 2 input frames, this is not possible currently. Though the artificial dataset can be modified to output more than 2 frames, there was not enough time to try using multiple frames as input for the neural network as input. This might also improve its accuracy as it would have more information about the motion in the scene.

Different architectures for the neural network could also be explored in the future. Currently, we have only tried a supervised approach with common CNN architectures such as VGGNet. Neural networks used for motion estimation shown in Chapter 2 have quite complicated architectures, like the iterative design of FlowNet2, so perhaps adapting the FlowNet2 architecture could achieve better results. Theoretically, an unsupervised approach would also solve the global motion estimation problem, as the output of the neural network could be used to reconstruct a new frame, which could be compared to actual frames to compute a loss. Such an unsupervised method would probably not work as well as a supervised method, as unsupervised methods have been shown to perform worse than supervised methods in [13]. However, if it would work, it would mean that there would not be a dataset problem as an unsupervised neural network can be trained on unlabelled and thus real videos.

Bibliography

[1] Aria Ahmadi and Ioannis Patras. Unsupervised convolutional neural networks for motion estimation. In *2016 IEEE International Conference on Image Processing (ICIP)*, pages 1629–1633, 2016. doi:10.1109/ICIP.2016.7532634.

[2] S. Albawi, T. A. Mohammed, and S. Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6, 2017. doi:10.1109/ICEngTechnol.2017.8308186.

[3] Mathieu Aubry, Daniel Maturana, Alexei A. Efros, Bryan C. Russell, and Josef Sivic. Seeing 3d chairs: Exemplar part-based 2d-3d alignment using a large dataset of cad models. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3762–3769, 2014. doi:10.1109/CVPR.2014.487.

[4] P. Bouthemy, B. M. T. Acosta, and B. Delyon. Robust selection of parametric motion models in image sequences. In *2016 IEEE International Conference on Image Processing (ICIP)*, pages 3743–3747, 2016. doi:10.1109/ICIP.2016.7533059.

[5] D. J. Butler, J. Wulff, G. B. Stanley, and M. J. Black. A naturalistic open source movie for optical flow evaluation. In A. Fitzgibbon et al. (Eds.), editor, *European Conf. on Computer Vision (ECCV)*, Part IV, LNCS 7577, pages 611–625. Springer-Verlag, October 2012. doi:10.1007/978-3-642-33783-3_44.

[6] G. De Haan and P. W. A. C. Biezen. An efficient true-motion estimator using candidate vectors from a parametric motion model. *IEEE Transactions on Circuits and Systems for Video Technology*, 8(1):85–91, 1998. doi:10.1109/76.660831.

[7] Gerard de Haan, Paul W.A.C. Biezen, Henk Huijgen, and Olukayode A. Ojo. True-motion estimation with 3-d recursive search block matching. *IEEE Transactions on Circuits and Systems for Video Technology*, 3(5):368–379, October 1993. doi:10.1109/76.246088.

[8] Alexey Dosovitskiy, Philipp Fischer, Eddy Ilg, Philip Häusser, Caner Hazirbas, Vladimir Golkov, Patrick van der Smagt, Daniel Cremers, and Thomas Brox. Flownet: Learning optical flow with convolutional networks. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 2758–2766, 2015. doi:10.1109/ICCV.2015.316.

[9] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *Computing Research Repository (CoRR)*, abs/1603.07285, 2016. URL: https://arxiv.org/abs/1603.07285v2.

[10] Sagi Eppel. Classifying a specific image region using convolutional nets with an ROI mask as input. *Computing Research Repository (CoRR)*, abs/1812.00291, 2018. URL: http://arxiv.org/abs/1812.00291, arXiv:1812.00291.

[11] Joost Hazelzet. Images of lego bricks. https://www.kaggle.com/joosthazelzet/lego-brick-images. [Online; accessed 19-March-2021].

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. doi:10.1109/CVPR.2016.90.

[13] Junhwa Hur and Stefan Roth. Optical flow estimation in the deep learning age. In Nicoletta Noceti, Alessandra Sciutti, and Francesco Rea, editors, *Modelling Human Motion: From Human Perception to Robot Design*, pages 119–140. Springer International Publishing, Cham, 2020. doi:10.1007/978-3-030-46732-6_7.

[14] Eddy Ilg, Nikolaus Mayer, Tonmoy Saikia, Margret Keuper, Alexey Dosovitskiy, and Thomas Brox. Flownet 2.0: Evolution of optical flow estimation with deep networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1647–1655, 2017. doi:10.1109/CVPR.2017.179.

[15] "kang and atul". Image processing – bilinear interpolation, Apr 2019. [Online; accessed 23-March-2021]. URL: https://theailearner.com/2018/12/29/image-processing-bilinear-interpolation/.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[17] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi:10.1109/5.726791.

[18] Siddharth Mahendran, Haider Ali, and René Vidal. 3d pose regression using convolutional neural networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 494–495, 2017. doi:10.1109/CVPRW.2017.73.

[19] Moritz Menze, Christian Heipke, and Andreas Geiger. Object scene flow. *ISPRS Journal of Photogrammetry and Remote Sensing*, 140:60–76, 2018. Geospatial Computer Vision. doi:10.1016/j.isprsjprs.2017.09.013.

[20] Sameer A. Nene, Shree K. Nayar, and Hiroshi Murase. object image library (coil-100). Technical report, Department of Computer Science, Columbia University, Feb 1996.

[21] Cherish Ortiz. Intro to video frame rates and frames per second shooting speeds, Jul 2021. [Online; accessed 13-July-2021]. URL: https://www.borrowlenses.com/blog/intro-to-video-frame-rates-and-frames-per-second-shooting-speeds/.

[22] Daniel Scharstein and Richard Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision*, 47(1-3):7–42, 2002. doi:10.1023/A:1014573219977.

[23] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL: http://arxiv.org/abs/1409.1556.

[24] Hao Su, Charles R. Qi, Yangyan Li, and Leonidas J. Guibas. Render for cnn: Viewpoint estimation in images using cnns trained with rendered 3d model views. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 2686–2694, 2015. doi:10.1109/ICCV.2015.308.

[25] Deqing Sun, Xiaodong Yang, Ming-Yu Liu, and Jan Kautz. Pwc-net: Cnns for optical flow using pyramid, warping, and cost volume. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8934–8943, 2018. doi:10.1109/CVPR.2018.00931.

[26] Unsplash. Beautiful free images & pictures. https://unsplash.com/. [Online; accessed 19-March-2021].

[27] V-Silicon. V-silicon. https://v-silicon.com/. [Online; accessed 15-July-2021].

[28] Wikipedia. YUV — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/YUV, 2021. [Online; accessed 12-March-2021].

[29] Krystian Wojcicki. Nearest neighbour interpolation, Aug 2020. [Online; accessed 23-March-2021]. URL: https://kwojcicki.github.io/blog/NEAREST-NEIGHBOUR.

[30] Titus Zaharia and Francoise J. Preteux. Parametric motion models for video content description within the MPEG-7 framework. In Edward R. Dougherty and Jaakko T. Astola, editors, *Nonlinear Image Processing and Pattern Analysis XII*, volume 4304, pages 118 – 132. International Society for Optics and Photonics, SPIE, 2001. doi:10.1117/12.424968.

Chapter 8

Appendix

Hyperparameter	Value
No. epochs	70
batch size	8
optimizer	Adam
learning rate	5e-5
loss function	Mean Absolute Error

Table 8.1: *Hyperparameters used for training the VGGNet model for the regression problem, described in Section 5.2.1.*

Hyperparameter	Value
No. epochs	50
batch size	8
optimizer	Adam
learning rate	5e-5
loss function	Binary Cross Entropy

Table 8.2: *Hyperparameters used for training the LeNet model for the classification problem, described in Section 5.2.2.*

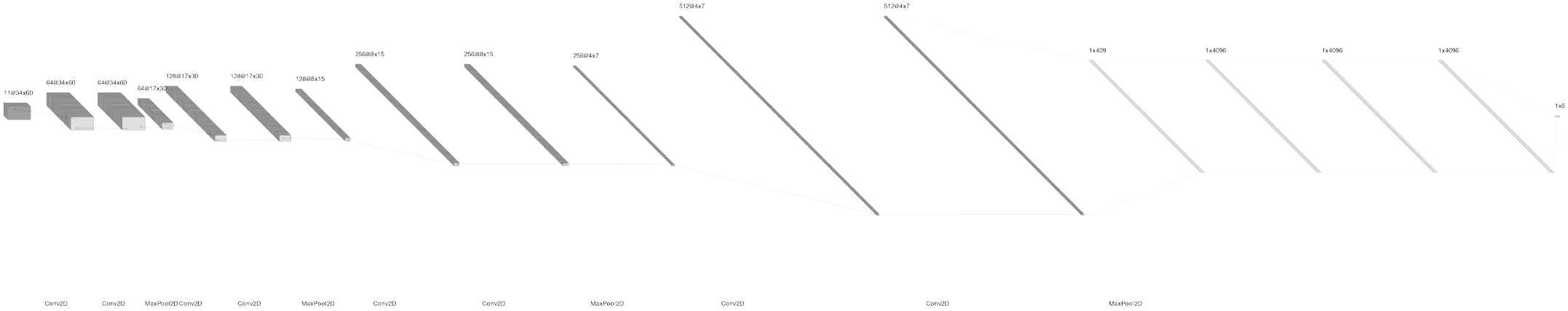


Figure 8.1: *The VGGNet network architecture used in this thesis.*

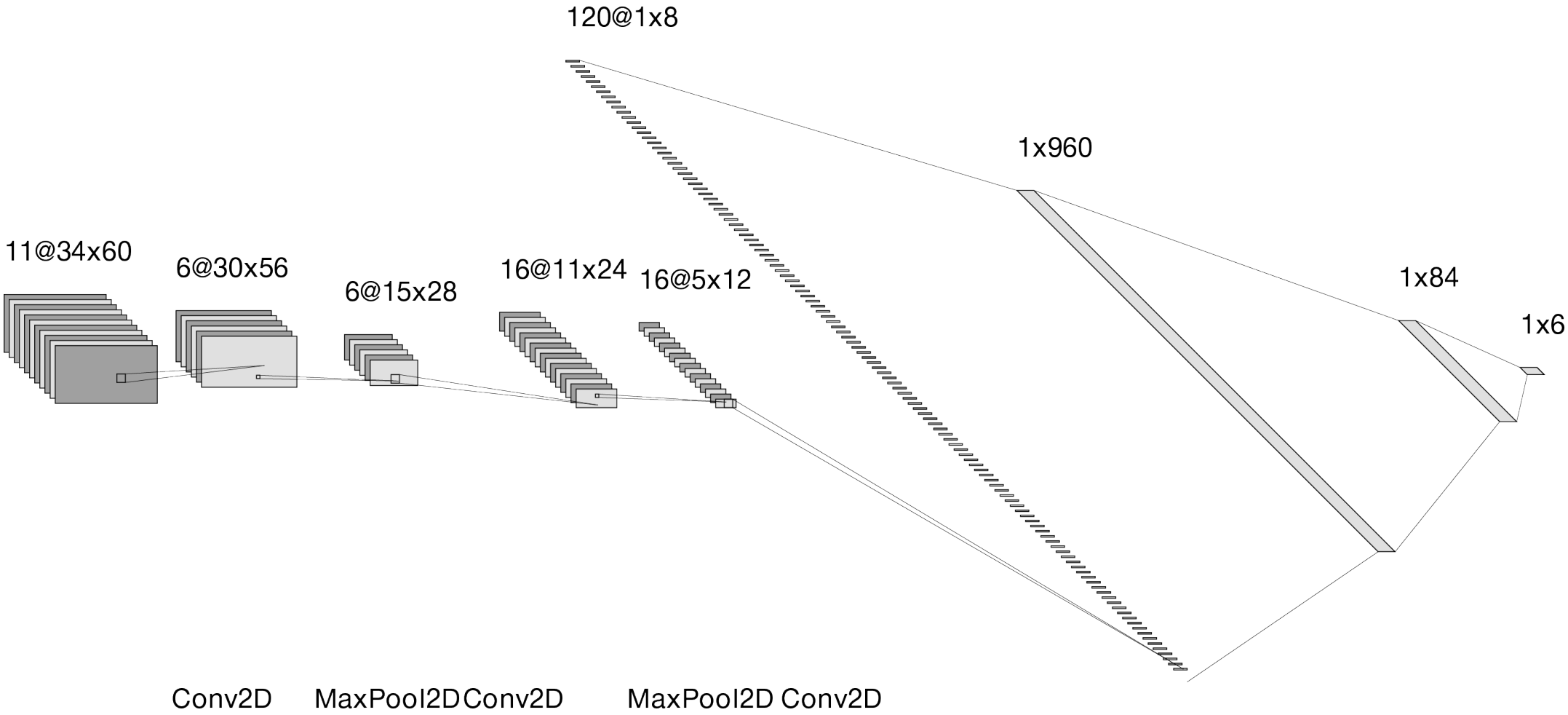


Figure 8.2: *The LeNet network architecture used in this thesis.*