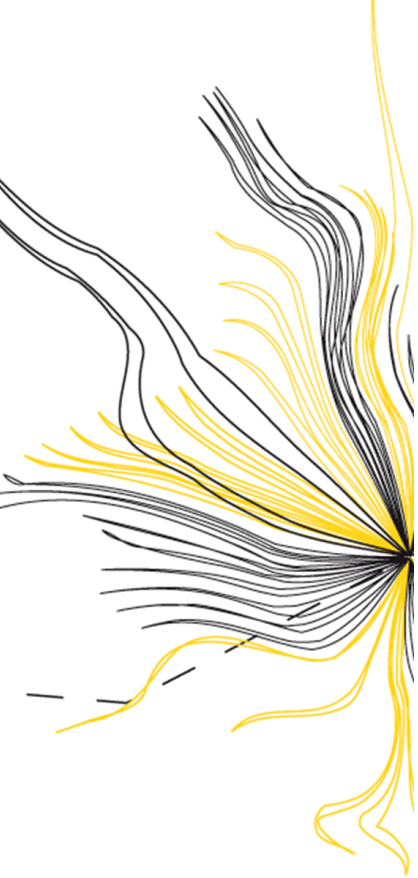


UNIVERSITY OF TWENTE.

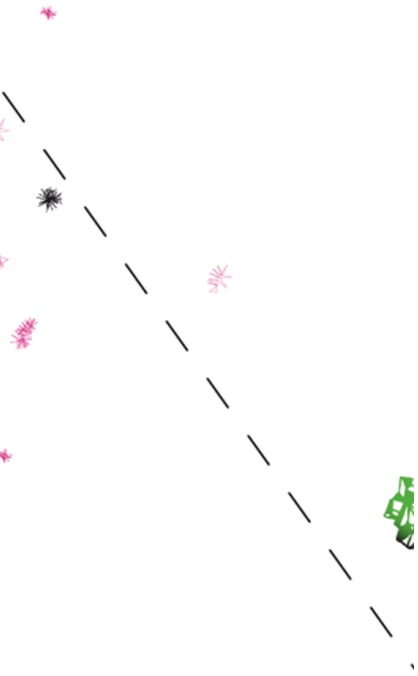
Faculty of Electrical Engineering, Mathematics & Computer Science

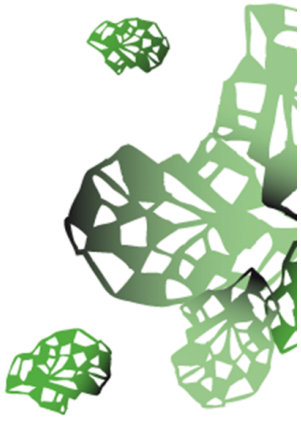
Tracking of moving objects using mathematical imaging

Jesse Zwienenberg

B.Sc. Thesis

July 2017





Supervisor:
Leonie Zeune

Applied Analysis group
Department of Applied Mathematics
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

Estimating the motion of objects in image sequences is a problem which arises in several research areas like image processing, bio-medical imaging and machine vision. The motion induced on the image plane by the objects is called the optical flow and in this work we compute this using two vastly different methods. Firstly we look at variational models which use the image derivatives in the setting of convex energy functional minimization to compute the optical flow between images. The second method that we discuss is known as deep learning and revolves around the usage of convolutional neural networks. To compare the performance of both methods we implemented the variational models from scratch and for the deep learning approach we use a publicly available pre-trained model.

Contents

Chapter 1

Introduction

1.1 Overview

The analysis of image motion plays a big role in computer vision. This is a field which aims to give computers the ability to understand images and videos on a high level. Application wise there is a very wide range of areas in which the analysis and estimation of image motion is useful. These applications range from recognizing human activity in the recordings of video surveillance systems to tracking biological cells in microscopic videos. Other notable examples of applications are motion compensation for video compression and estimating the 3D scene layout from 2D video footage.

In this paper we are specifically interested in determining the underlying 2D motion of objects in image sequences. This is what is called the optical flow. So given any two consecutive frames in a sequence of images we are looking to create a field of vectors such that every vector represents how the object at that specific position is moving in between the two frames. This is called a flow field and an example of what this looks like, is shown in Fig. 1.3. This field represents the optical flow between two images from the 'Hamburg Taxi' sequence [1].

Figure 1.1: Image 1 [1]

Figure 1.2: Image 2 [1]

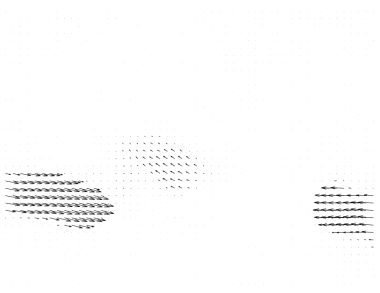


Figure 1.3: Flow field

The first method for computing the optical flow that we look at is the variational method. This method follows from the mathematical formulation of basic assumptions about image motion. We solve the minimization problem that follows from these assumptions using a primal-dual algorithm.

The second method is the deep learning method, which uses very large convolutional neural networks. These networks extract abstract image-features and compute the optical flow based on these features. The models are trained on data of which the underlying motion is known. Through this training the models learn how they should extract and combine the image-features to accurately compute the optical flow.

Variational methods are driven by theory and have been around in image analysis for a long time. On the other hand deep learning methods are data-driven and only recently gained popularity. Since there is such a big difference between the approaches of both methods it is interesting to see how their performances compare with each other.

1.2 Outline

In this work we review two different approaches to optical flow computation. We start by taking a closer look at the optical flow and describing it in a mathematical way in Chapter 2. We describe problems that come up which make computing the optical flow a difficult task.
Then in Chapter 3 we discuss variational methods for optical flow computation, the first of the two approaches that we discuss. After introducing the general concept of variational methods we explore and compare different possibilities inside this framework. After this we look at a numerical implementation of these methods.
In Chapter 4 we look at a totally different approach known as deep learning. We show what convolutional neural networks look like and how they can be used to compute the optical flow.
In Chapter 5 we put the discussed methods in practice. We show how they perform on several datasets and compare aspects of their performance.
Finally we present the drawn conclusions and suggestions for further research in Chapter 6.

Chapter 2

Optical Flow

Finding the optical flow comes down to recognizing corresponding objects between two images. So for any object X from the first image we are given the task of finding an object Y in the second image which corresponds to object X. If we are able to find such an object Y, we can estimate the optical flow with the vector between the position of X and the position of Y (scaled according to the time between the images). For this task of finding corresponding objects we can use the assumption that if we make sure the time between two images is small enough, the object will look the same in both images, the only difference being that it might be slightly translated across the axes of the image. There are several obvious cases in which this assumption fails, examples are objects which become occluded or inconstant lighting casting shadows on the scene which change the appearance of moving objects over time. These things are points of attention, but in most cases the majority of objects in a scene will look approximately the same in consecutive images, so this assumption is a reasonable starting point.

2.1 Optical flow constraint

Let us introduce some notation to turn this assumption into an equation. Let $x = (x_1, x_2)^\intercal \in \mathbb{R}^2$ denote a spatial position and $t \in [0, T]$ a certain point in time. Now let $u(x, t)$ be a representation of the appearance of the pixel at position x on the image taken at time t. This representation can have multiple dimensions, for example the RGB-values of the pixel. In this work however we use $u(x, t)$ to denote the brightness, the simplest representation of the pixel.

Consider an object X which is located at position x on the image taken at time t. Let's denote the displacement of X between this image and the next one by the vector Δx and the time between the images by the scalar Δt. We expect the brightness of object X to be the same in both images so now we can formulate the following *brightness constancy constraint*:

$$u(x,t) = u(x + \Delta x, t + \Delta t) \; . \tag{2.1}$$

We can rewrite the right-hand side as a Taylor Series at the point (x,t) and under the condition that both Δx and Δt are small we can can ignore the higher order terms.

$$u(x + \Delta x, t + \Delta t) \approx u(x,t) + \frac{\partial u}{\partial x_1}\Delta x_1 + \frac{\partial u}{\partial x_2}\Delta x_2 + \frac{\partial u}{\partial t}\Delta t \; .$$

When it comes to determining optical flow the displacement Δx is unknown and the displacement per time unit $\frac{\Delta x}{\Delta t}$ is what we actually want to compute. So the next step is canceling out the common term $u(x,t)$ and dividing everything by Δt, which yields:

$$0 = \frac{\partial u}{\partial x_1}\frac{\Delta x_1}{\Delta t} + \frac{\partial u}{\partial x_2}\frac{\Delta x_2}{\Delta t} + \frac{\partial u}{\partial t} \; .$$

In the literature this is often written in a slightly cleaner way by using u_t to denote $\frac{\partial u}{\partial t}$ and ∇u to denote the gradient of the image data, a vector containing the two spatial derivatives of the image, $\left(\frac{\partial u}{\partial x_1}, \frac{\partial u}{\partial x_2}\right)^\intercal$. Also $v = (v_x, v_y)^\intercal$ is used to denote the estimation of the optical flow $\frac{\Delta x}{\Delta t} = \left(\frac{\Delta x_1}{\Delta t}, \frac{\Delta x_2}{\Delta t}\right)^\intercal$, yielding:

$$\nabla u \cdot v + u_t = 0 \; . \tag{2.2}$$

This equation is generally known as the *optical flow constraint*. It is used in the variational framework introduced in Chapter 3.

2.2 Difficulties

The optical flow constraint exposes some inherent problems of motion perception. The constraint forms a linear system with n linear equations, where n is the number of dimensions of $u(x,t)$. This system has a unique solution if there are two independent equations since there are two unknowns, v_x and v_y. The brightness is one-dimensional so in our case the system consists of only a single equation, hence this system does not yield a unique solution. Only the component in the direction of the image derivatives $(u_x, u_y)^\intercal$ can be determined. We can not say anything about the component in the direction perpendicular to these derivatives, based on this system. Furthermore when all image derivatives are zero the optical flow constraint gives us no information about the motion at all. This occurs in the interior of uniform regions, where any v would satisfy the optical flow constraint.

This problem of an underdetermined system as a result of the optical flow constraint is known as the aperture problem. This problem tells us that certain combinations of motions of objects can cause identical looking images. As a consequence there will be cases in which there is no way to uniquely determine the underlying motion based on the image data alone. This means that if we want to use the optical flow constraint to compute the optical flow at locations like this we need extra constraints to obtain unique solutions.

In the previous section we mentioned a couple of occasions where the constancy assumption (equation 2.1) is not fulfilled at all. Apart from these situations we need to take into account that image data is not perfect and consequently the brightness constancy constraint will not hold perfectly most of the time. First of all the data is a discretization of the reality, this can cause objects to be displayed slightly different between images especially when the resolutions of the images are low. Secondly, noise in the data can cause disturbances in the fulfillment of the optical flow constraint.

Furthermore we need to keep in mind that to arrive at the optical flow constraint we assumed Δx and Δt to be small. When Δx is too big we will not be able to find the correct corresponding object using the optical flow constraint. This comes from the fact that the optical flow constraint at location x only uses the image derivatives at location x. These derivatives only describe the local environment so image information far away from x is completely ignored. On the other hand when Δt is too large the assumption that any found displacement between corresponding objects represents the actual optical flow is not reasonable anymore. Objects might as well have traveled across the whole image and back again in the meantime.

For an in-depth analysis of other basic problems and concepts related to optical flow estimation we refer to [2].

2.3 Representing flow fields

Representing the flow field with arrows is difficult to interpret in some cases, so often the motion is indicated by a color coding instead. In Fig. 2.4 we see how the flow field from Fig. 2.3 can be represented using colors. The color indicates the direction of the vector and the intensity goes up as the absolute value of the vector gets larger, which indicates a higher velocity. This is visualized by the colorwheel in Fig. 2.5.

Figure 2.1: Image 1 [1]

Figure 2.2: Image 2 [1]

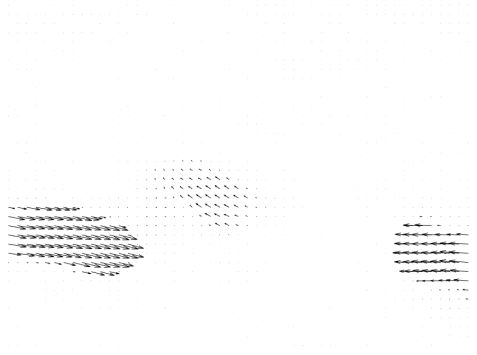


Figure 2.3: Arrows

Figure 2.4: Color-coding

Figure 2.5: Colorwheel

2.4 Benchmarks

The Middlebury database [5] is often used as a benchmark to assess the relative performances of optical flow algorithms. It is a small dataset of image sequences of which the ground-truth flow is determined through different measurements. This dataset addresses different challenging aspects of flow estimation and it introduced an online evaluation and ranking for optical flow algorithms.

Figure 2.6: Middlebury

Some more image sequences and their ground-truth flows are presented by the KITTI database [6]. This data base includes the frames of videos recorded by the camera on top of a driving car. The ground truth flow is determined using a laserscanner which is also located on top of the car. This dataset contains realistic data of outdoor scenes, however the ground-truth flow is sparse since the movement of the sky can not be captured using the laser scanner.

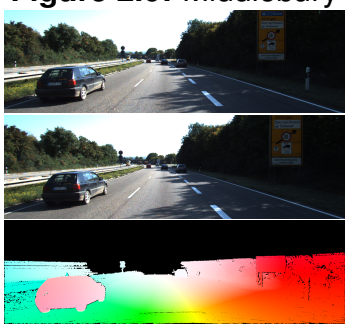


Figure 2.7: KITTI

Another commonly used benchmark is the MPI Sintel dataset [7]. This dataset consist of rendered scenes of an animated movie. Since the scenes are artificial the ground-truth can be easily determined. This dataset is the largest of the three and contains over a thousand image pairs.

Examples of image pairs and their ground-truths from each of these datasets are displayed in Fig. 2.6, 2.7 and 2.8.

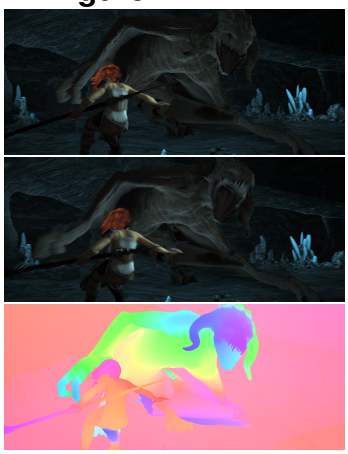


Figure 2.8: MPI Sintel

Chapter 3

Variational Methods

The first implementation of a variational method for optical flow computation was the Horn-Schunk method [8] constructed by Horn and Schunk in 1981. Since then better and more complex methods have been developed (e.g. [3]), but the essence of variational methods has remained the same. As mentioned earlier most variational methods for optical flow estimation use the optical flow constraint as a foundation. The main problem of this constraint is its possible ambiguity due to the aperture problem. To overcome this problem and to get to a unique result, an additional constraint is added. This constraint should impose some kind of structure on the solution that we would expect actual flow fields to have.

The additional constraint that the Horn-Schunk method used was based on the assumption that flow fields vary smoothly almost everywhere. For the interior areas of objects this assumption makes a lot of sense, in these areas we can expect neighboring points to have similar velocities. However, for points on the edges of objects it is different. At the edges of objects neighboring points can belong to entirely different objects which can have entirely different velocities so discontinuities can be expected. This means that methods using a smoothness constraint are likely to have difficulties determining the correct flow around the edges of objects.

3.1 Variational models

The basic idea of variational models for optical flow computation is to use a secondary constraint alongside with the optical flow constraint to find a solution which agrees with both of them as well as possible. This is done through the usage of energy functionals. For a given image sequence u we want to assign a certain measure of 'energy' to any possible flow field v, this energy serves to indicate how well both constraints are met. When the constraints are fulfilled this energy gets very small and violations of the constraints lead to higher energies. Through minimizing

this energy we wish to find a v with a very low energy, indicating it fits our constraints well. In general these variational models are of the form:

$$\min_{v} \quad \mathcal{D}(u, v) + \alpha \mathcal{R}(v) \ . \tag{3.1}$$

Here, $\mathcal{D}(u, v)$ is called the *data term*, it takes as input the image data u and a possible solution v. To make sure the optimal solution $\hat{v}$ of the expression (3.1) does not violate the optical flow constraint too much, this function is defined in such a way that the output gets bigger as v obeys the optical flow constraint less strictly.

The other term $\mathcal{R}(v)$, called the *regularization term*, does not use the image data. The purpose of this term is basically to measure how likely it is for any particular v to be an actual flow field, regardless of the image sequence. As mentioned before a way of doing this is by looking at the smoothness of the field v. In general this is what most regularization terms do, they get very small when v is smooth and larger when the smoothness constraint is violated.

The α is a scalar deciding the relative importance between the two terms. Choosing the appropriate α comes down to deciding how well we expect the actual flow field to agree with the optical flow constraint. For noisy images we need to set α to a higher value. Due to the noise, the actual flow field violates the optical flow constraint by some amount, so we really need the regularization term to enforce smoothness even if that means the optical flow constraint gets violated more. For 'cleaner' images we do not really need the regularization term to force the smoothness as much. In this case its task is better described as picking the most smooth field out of the possible solutions v that agree with the optical flow constraint really well. For this task it is better to set α to a lower value.

3.1.1 Data terms

We mentioned that we want the data term to give a certain 'punishment' to possible solutions v for violating the optical flow constraint. We define such a function by calculating how much it violates the optical flow constraint for every position x and then take a norm. Conventional choices for this norm are either the L^1 norm or the squared L^2 norm, yielding:

$$\mathcal{D}_{L1}(u, v) \coloneqq \|\nabla u \cdot v + u_t\|_1 \quad \text{and}$$

$$\mathcal{D}_{L2}(u, v) \coloneqq \frac{1}{2} \|\nabla u \cdot v + u_t\|_2^2 \ .$$

The squared L^2 norm takes the least squares approach to get to a solution, which is a method commonly used for approximating solutions of overdetermined systems. The Horn-Schunk method [8] is using this as its data term. One property

of this squared L^2 norm is that outliers do have a huge impact on its value, so this method does not allow the solution to violate the optical flow constraint by a very large amount. This is not always preferable. In this respect the L^1 norm handles outliers in a more robust way. Here, outliers in the data are less destructive to the solution.

3.1.2 Regularization terms

The regularization terms that we mention here serve to give some measure of smoothness to the flow fields. This can be done by looking at the gradient of the flow field ∇v, which you want to be close to 0 most of the time if you expect the flow field to be smooth. Two standard choices for the regularization term are:

$$\mathcal{R}_{TV}(v) \coloneqq \|\nabla v\|_1 \quad \text{and}$$

$$\mathcal{R}_{L2}(v) \coloneqq \frac{1}{2} \|\nabla v\|_2^2 \ .$$

The Horn-Schunk method [8] uses the last option of the two, here the L^2 norm is used to punish possible solutions v for having a gradient which is not close to 0 at several positions. Just like with the L^2 data term, the L^2 regularization term is punishing outliers really heavily. This will generally lead to solutions which do not have any of these outliers, meaning it is likely to be a completely smooth field. Again, this might not always be what we want, flow fields need not to be smooth everywhere. In fact most of the time we want the field to have sharp edges instead of smooth transitions at the very edge of the objects. The total variation (TV) of v is a regularization term which generally allows these sharp edges to exist, since the punishment for outliers is not as extreme in this case. Also where the L^2 term has very low punishment for small deviations from 0, the TV term enforces this constraint linearly. So when minimizing the TV term, there is still a relatively big incentive to push small deviations from 0 even closer to 0. This will generally result in solutions which are approximately constant everywhere except at the edges of objects where sharp edges occur.

Extended regularization terms

We can choose between $\mathcal{R}_{L2}$ and $\mathcal{R}_{TV}$ to either create a smooth solution or a solution with sharp edges. However in practice we want the solution to have both properties. An attempt to combine these properties is described in [4], where an

extension of the following form is proposed:

$$\mathcal{R}(v) = \inf_{w} \quad \alpha_0 \sum_{i=1}^{2} \|\nabla v_i - w\|_1 + \alpha_1 \mathcal{S}(w) \ . \tag{3.2}$$

Here, a new variable w is introduced to shift the derivatives of v, and a function $\mathcal{S}(w)$ which forces this w to be small. The usage of the L^1 norm in $\|\nabla v_i - w\|_1$ is supposed to create the sharp edges and we need w to facilitate the general smoothness of v. Also instead of a single value for α we will have two parameters. Here α_0 has a role similar to that of α in the standard regularization terms, it determines the weight of the smoothness constraint relative to the optical flow constraint. Now α_1 can be chosen in proportion to α_0. When $\frac{\alpha_0}{\alpha_1}$ is chosen to be large this indicates that the piece-wise constant parts outweigh the smooth parts and the opposite when $\frac{\alpha_0}{\alpha_1}$ is small.

We can achieve the smoothness in the solution by using the squared L^2 norm of w as $\mathcal{S}(w)$. Doing this makes sure w will be small and does not contain outliers, which enforces a certain smoothness on w. We expect $\nabla v_i - w$ to be piece-wise constant, since we use the L^1 norm. If we then add the small smooth field w we can expect ∇v to be piece-wise smooth, which is what we wanted.

Another way of achieving smoothness is by minimizing higher order derivatives of v. If we define $\mathcal{S}(w)$ to be the L^1 norm of ∇w we achieve something similar to this since the first part of the extended regularization term leads to w being close to ∇v. Effectively we are minimizing something which is close to $\nabla^2 v$. By letting this part of the regularization term work alongside the first part we want to generate solutions which are smooth and contain sharp edges

This gives us the following definitions for the extended regularization terms:

$$\mathcal{R}_{TV/L2}(v) \coloneqq \inf_{w} \quad \alpha_0 \sum_{i=1}^{2} \|\nabla v_i - w\|_1 + \frac{\alpha_1}{2} \|w\|_2^2 \quad \text{and}$$

$$\mathcal{R}_{TV/TV}(v) \coloneqq \inf_{w} \quad \alpha_0 \sum_{i=1}^{2} \|\nabla v_i - w\|_1 + \alpha_1 \|\nabla w\|_1 \ .$$

3.2 Numerical realization

The optical flow constraint at a certain pixel is only dependent on the solution v at that pixel. However, this is not the case for the regularization terms. Changes at a certain position in v will cause changes in ∇v in an area around this position. This means we can not divide the problem into independent parts and to solve the minimization problem we need an efficient method which solves problems of the form:

$$\min_{v} \quad \mathcal{D}(u, v) + \alpha \mathcal{R}(v) \ .$$

A commonly used method for solving minimization problems is gradient descent. This method uses the partial derivatives of the objective function to determine where the minimum is located. This minimum is iteratively approached by taking steps in the direction of the negative of the gradient. One of the requirements of this methods is that the objective function needs to be differentiable. The regularization terms that we mentioned are non-smooth and hence not differentiable. For this reason basic minimization schemes like gradient descent do not work.

A minimization scheme that does work for this problem is the first-order primal dual algorithm described in [9]. This algorithm converges to a solution at the rate $O(1/N)$ and applies to problems of the form:

$$\min_{x} \quad F(Kx) + G(x) \ . \tag{3.3}$$

Where F and G are proper, convex[1], lower semi-continuous[2] functions and K is a continuous, linear operator.

The algorithm acts on a so-called primal-dual problem which follows from the primal problem (3.3). This primal-dual problem is of the form:

$$\min_{x} \max_{y} \quad \langle Kx, y \rangle + G(v) - F^*(y) \ . \tag{3.4}$$

The algorithm iteratively approaches the solution $\hat{x}, \hat{y}$ of the primal-dual problem (3.4). In every iteration the value x and y are approximated as a function of the x and y of the previous iteration. How this iterative scheme is defined can be seen in Fig. 3.1.

[1] $f\left(\theta x + (1-\theta)y\right) \leq \theta f(x) + (1-\theta) f(y) \quad \forall \theta \in [0, 1]$

[2] $\liminf_{x \to x_0} \ f(x) \geq f(x_0)$

Primal-dual algorithm [9]

Choose: $\tau, \sigma > 0, \ \theta \in [0,1], \ (x^0, y^0) \in X \times Y, \ \bar{x}^0 = x^0$

$$\begin{aligned} y^{n+1} &= (I + \sigma \partial F^*)^{-1} (y^n + \sigma K \bar{x}^n) \\ x^{n+1} &= (I + \tau \partial G)^{-1} \left(x^n - \tau K^* y^{n+1}\right) \\ \bar{x}^{n+1} &= x^{n+1} + \theta \left(x^{n+1} - x^n\right) \end{aligned}$$

Where:

$$(I + \tau \partial F)^{-1}(y) = \arg\min_x \left\{ \frac{\|x - y\|^2}{2\tau} + F(x) \right\}$$

Figure 3.1: Scheme showing how variables are updated in the algorithm [9]

Here, F^* is used to denote the convex conjugate of F. The parts which are essential for the efficiency of this algorithm are the operators $(I + \tau \partial F^*)^{-1}$ and $(I + \sigma \partial G)^{-1}$ which are called the proximal operators. The algorithm will be efficient when these can be solved efficiently.

3.2.1 Implementation

To implement the models discussed in section 3.1 we need to convert the data- and regularization terms to functions F and G, determine suitable operators K and K^* and find a way to compute the proximal operators. To make sure that the algorithm is efficient we need to formulate our problem (3.1) in such a way that the proximal operators can be solved efficiently. The proximal operators for the data terms can be solved efficiently as a function of v, so we can use $\mathcal{D}$ as our function G. However for the regularization terms this is not the case. The gradients used in these terms makes them complicated to invert. To overcome this problem we can put the gradient inside the operator K. In the scheme of the algorithm we can see that we do not need to invert this operator, we only need to determine the adjoint of this operator. The adjoint operator of the gradient is just the negative divergence, so we can use the operator $K = \nabla$ without a problem. Now we can treat $\mathcal{R}$ as a function of ∇v instead of just v, which leaves us with a proximal operator which can be solved efficiently. For the extended regularization terms slightly different operators for K

can be used to make the algorithm work.

Implementation of the data terms

For the data terms we do not need the linear operator K so they can be transformed to G straight up. The data term $\mathcal{D}_{L1}$ leads to:

$$G(v) = \|v \cdot \nabla u + u_t\|_1$$

and the term $\mathcal{D}_{L2}$ yields:

$$G(v) = \frac{1}{2} \|v \cdot \nabla u + u_t\|_2^2 \ .$$

In case we use the regularization term $\mathcal{R}_{TV/L2}$ we will change this function G slightly.

Implementation of the regularization terms

For the regularization terms we will need the operator K to act on v and possibly on w. For both the standard terms $\mathcal{R}_{TV}$ and $\mathcal{R}_{L2}$ we can define $K = \nabla$ as a linear operator and $K^* = -\nabla\cdot$ as its adjoint operator. For $\mathcal{R}_{TV}$ we get:

$$F(Kv) = \alpha \|Kv\|_1 = \alpha \|\nabla v\|_1$$

and for $\mathcal{R}_{L2}$ we get:

$$F(Kv) = \frac{\alpha}{2} \|Kv\|_2^2 = \frac{\alpha}{2} \|\nabla v\|_2^2 \ .$$

For the extended regularization terms the variable w comes into play. Our primal variable will change from v to (v, w). In the case of $\mathcal{R}_{TV/L2}$ we want our F to be a function of $\nabla v - w$, so we get: $K = \begin{bmatrix} \nabla & -I \end{bmatrix}$ and $K^* = \begin{bmatrix} -\nabla\cdot & -I \end{bmatrix}^\mathsf{T}$.

Now F can be defined as:

$$F\left(K(v,w)\right) = \alpha_0 \|K(v,w)\|_1 = \alpha_0 \|\nabla v - w\|_1 \ .$$

We want to add the last part of this regularization term $\mathcal{R}_{TV/L2}$ to the function G so depending on the data term this becomes:

$$G(v,w) = \|v \cdot \nabla u + u_t\|_1 + \frac{\alpha_1}{2} \|w\|_2^2 \quad \text{or}$$
$$G(v,w) = \frac{1}{2} \|v \cdot \nabla u + u_t\|_2^2 + \frac{\alpha_1}{2} \|w\|_2^2 \ .$$

For the extended regularization term $\mathcal{R}_{TV/TV}$ we do not need to make changes to the original function G and we want F to be a function of $(\nabla v - w, \nabla w)$ which we get using the following operator and its adjoint:

$$K = \begin{bmatrix} \nabla & -I \\ 0 & \nabla \end{bmatrix} , \qquad K^* = \begin{bmatrix} -\nabla\cdot & 0 \\ -I & -\nabla\cdot \end{bmatrix} .$$

Now F can be defined as:

$$F\left(K(v,w)\right) = \alpha_0 \|\nabla v - w\|_1 + \alpha_1 \|\nabla w\|_1$$

Calculating the proximal operators

We are left with the task of computing the proximal operators. For any combination of the terms that we discussed, these will be efficiently solvable, there is even a closed form representation for all of them.

For the function G we have the following possibilities:

$$\begin{aligned} G(v) &= \|v \cdot \nabla u + u_t\|_1 \\ G(v) &= \frac{1}{2} \|v \cdot \nabla u + u_t\|_2^2 \\ G(v,w) &= \|v \cdot \nabla u + u_t\|_1 + \frac{\alpha_1}{2} \|w\|_2^2 \\ G(v,w) &= \frac{1}{2} \|v \cdot \nabla u + u_t\|_2^2 + \frac{\alpha_1}{2} \|w\|_2^2 \ . \end{aligned}$$

The first option which comes from the usage of $\mathcal{D}_{L1}$ leads to:

$$v = (I + \sigma \partial G)^{-1}(\tilde{v}) = \tilde{v} + \begin{cases} -\tau \nabla u & \text{if } \rho(\tilde{v}) < -\tau (\nabla u)^2 \\ \tau \nabla u & \text{if } \rho(\tilde{v}) > \tau (\nabla u)^2 \\ \frac{\rho(\tilde{v})}{\nabla u} & \text{if } |\rho(\tilde{v})| \leq \tau (\nabla u)^2 \end{cases} \tag{3.5}$$

where $\rho(v) = \nabla u \cdot v + u_t$.

The second option for G comes from the usage of $\mathcal{D}_{L2}$ and leads to a proximal operator which is representable as:

$$v = (I + \sigma \partial G)^{-1}(\tilde{v}) = \left(\frac{a_3 b_1 - a_2 b_2}{a_1 a_3 - a_2^2}, \frac{a_1 b_2 - a_2 b_1}{a_1 a_3 - a_2^2} \right) \tag{3.6}$$

for:

$$\begin{aligned} a_1 &= 1 + \tau u_x u_x & b_1 &= \tilde{v}_1 - \tau u_x u_t \\ a_2 &= \tau u_x u_y & b_2 &= \tilde{v}_2 - \tau u_y u_t \\ a_3 &= 1 + \tau u_y u_y \ . \end{aligned}$$

For the last two instances of G where the variable w plays a role due to the usage of the regularization term $\mathcal{R}_{TV/L2}$ we will get a solution of the form (v, w) where w can be computed seperately. We can calculate v in the same way as we just did in (3.5) and (3.6), where there was no w.

$$(I + \sigma \partial G)^{-1}(\tilde{v}, \tilde{w}) = (v, w)$$

where

$$w = \frac{\tilde{w}}{1 + \tau \alpha_1} \tag{3.7}$$

For the function F we have the following possibilities:

$$\begin{aligned} F(Kv) &= \alpha \|Kv\|_1 \\ F(Kv) &= \frac{\alpha}{2} \|Kv\|_2^2 \\ F(K(v,w)) &= \alpha_0 \|K(v,w)\|_1 \\ F(K(v,w)) &= \alpha_0 \|\nabla v - w\|_1 + \alpha_1 \|\nabla w\|_1 \ . \end{aligned}$$

For the last instance we can calculate its solution $(I + \tau \partial F^*)^{-1}(\tilde{p}, \tilde{q}) = (p, q)$ separately for p and q as functions of respectively $\tilde{p}$ and $\tilde{q}$ where $(\tilde{p}, \tilde{q}) = K(v, w) = (\nabla v - w, \nabla w)$. So all we need to do is to find the proximal operators of the convex conjugate of the following functions:

$$\begin{aligned} F(p) &= \alpha \|p\|_1 \quad \text{and} \\ F(p) &= \alpha \|p\|_2^2 \ . \end{aligned}$$

In case we choose the regularization term $\mathcal{R}_{TV}$ or $\mathcal{R}_{TV/L2}$ we want to compute the proximal operator using the convex conjugate of $F(p) = \alpha \|p\|_1$ and for $\mathcal{R}_{TV/TV}$ we want to do this twice, for p and for q. First we determine the convex conjugate F^* which is:

$$F^*(p) = \alpha \delta_{B(L^\infty)}\left(\frac{p}{\alpha}\right) \quad \text{where } \delta_{B(L^\infty)}\left(\frac{p}{\alpha}\right) := \begin{cases} 0 & \text{for } \|p\|_\infty \leq 1 \\ \infty & \text{otherwise} \end{cases} .$$

Using this, the proximal operator can be determined which results in:

$$p = (I + \tau \partial F^*)^{-1}(\tilde{p}) = \min(\alpha, \max(-\alpha, \tilde{p})) \ . \tag{3.8}$$

For the regularization term $\mathcal{R}_{L2}$ we want to compute the convex conjugate of $F(p) = \alpha \|p\|_2^2$ which is:

$$F^*(p) = \frac{1}{2\alpha} \|p\|_2^2 \ .$$

Using this leads to the following proximal operator:

$$p = (I + \tau \partial F^*)^{-1}(\tilde{p}) = \frac{\alpha}{\alpha + \sigma} \tilde{p} \, . \tag{3.9}$$

Rigorous proofs of all these claims can be found in [10] .

Chapter 4

Deep Learning

Recently deep learning methods have become popular in several fields, including computer vision. An example of an interesting application is described in [11]. Originally these methods were only used for tasks like classification which is vastly different from computing the optical flow in the sense that the latter requires a per-pixel solution. With recent progress in areas which do require per-pixel solutions like semantic segmentation and depth estimation it has become increasingly interesting to approach the optical flow problem using deep learning methods.

4.1 How deep learning works

A deep learning model is in essence a function with a very large amount of parameters, which predicts the 'label' of the input. What this label is depends on the application, in our case it would be the optical flow of the input images. A general deep learning model could be described as:

$$\hat{y} = f(x\,;\theta)\ .$$

Here $\hat{y}$ is the prediction of the label of input x using parameters θ. This function f is described by a large artificial neural network which consists of many neurons in a layered structure. You typically have an input layer which receives the input $x = x_0$ and gives x_1 as output using the parameters θ_1. This output is then used as input for the next layer. The layers after the input layer, the so-called hidden layers, sequentially determine $x_2, x_3, \cdots$ according to their parameters:

$$x_i = f_i(x_{i-1}\,;\theta_i)\ .$$

Finally the last layer, the output layer, takes the output of the last hidden layer and determines the prediction $\hat{y}$..

$$\hat{y} = f_n(x_{n-1}\,;\theta_n) = f_n(f_{n-1}(x_{n-2}\,;\theta_{n-1})\,;\theta_n) = \cdots\ .$$

The layers consist of a certain amount of neurons and some 'activation functions'. Every neuron in layer i has connections to neurons from the previous layer $i-1$ through which it receives its inputs x_{i-1}. Each connection has a certain weight, the weights of all the neurons in a single layer form θ_i. The neuron uses these weights to take a weighted sum of its inputs and pass it through the activation function. The outputs of all these neurons form x_i, which is given as input to the next layer. The purpose of the neural network is to grasp certain abstract features of the input and make a prediction for the solution based on these features. The depth created through the use of multiple layers enables the network to model more complex behavior.

4.1.1 Activation functions

If we were to choose linear activation functions we would just get a linear transformation of the input at every layer. This would defeat the whole purpose of having many layers since it would result in an output which is just a linear combination of the inputs. We want the network to be able to catch more interesting non-linear behavior so that is why it is important to choose nonlinear activation functions. Common choices are either the sigmoid function:

$$\phi(x) = \frac{1}{1+e^{-x}}\ , \tag{4.1}$$

or a rectified linear unit(ReLU):

$$\phi(x) = \begin{cases} x & x > 0 \\ 0 & \text{otherwise} \end{cases}\ . \tag{4.2}$$

4.1.2 Learning

The architecture of the neural network facilitates the extraction of features but deciding which features are important to the solution and how these features should be extracted is something which is done by the weights of the connections. We could try to set these parameters manually, but except for the fact that the amount of weights gets enormous we also do not know how abstract features contribute to our solution most of the time. Instead of manually doing this, the parameters will be decided through supervised learning. This means that we use data of which we know the correct solution to train the network so it learns how it can make accurate predictions. So given data $x_1, x_2, \cdots$ of which we know the correct labels $y_1, y_2, \cdots$ we want to change the parameters θ in such a way that for all i:

$$f(x_i\ ;\theta) = \hat{y}_i \approx y_i$$

To achieve this, the parameters will be adjusted through a process which is known as backpropagation. Backpropagation uses a stochastic variant of the gradient descent method to determine how it should adjust the parameters that it wants to train. The objective function which we want to minimize will be some measure of how well the network is predicting the solution:

$$L(\theta) = \sum_i \ell\left(y_i, \hat{y}_i\right) = \sum_i \ell\left(y_i, f(x_i\,;\theta)\right)\ .$$

Here the loss function ℓ determines how good the prediction $\hat{y}_i$ is. A typical choice for this function ℓ is the Euclidean loss:

$$\ell\left(y_i, \hat{y}_i\right) = \|y_i - \hat{y}_i\|_2^2\ .$$

To minimize $L(\theta)$ we iteratively adjust θ and move it in the direction of steepest descent:

$$\theta^{i+1} = \theta^i - \lambda \nabla_\theta L(x\,;\theta)$$

Here λ denotes the stepsize, which is called the learning rate in this context. To compute the direction of steepest descent $-\nabla_\theta L(x\ ;\theta)$ we need to calculate the partial derivatives of the objective function L with respect to the parameters θ. The last layer directly influences the output of the network so the partial derivatives with respect to the parameters of this layer can easily be calculated as:

$$\frac{dL}{d\theta_n} = \frac{dL}{d\hat{y}} \frac{d\hat{y}}{d\theta_n}\ .$$

Given that the activation functions are differentiable we can now calculate the partial derivatives in the previous layer using the chain rule for differentiation:

$$\begin{aligned}
\frac{dL}{d\theta_{n-1}} &= \frac{dL}{d\hat{y}} \frac{d\hat{y}}{dx_{n-1}} \frac{dx_{n-1}}{d\theta_{n-1}}\ , \\
\frac{dL}{d\theta_{n-2}} &= \frac{dL}{d\hat{y}} \frac{d\hat{y}}{dx_{n-1}} \frac{dx_{n-1}}{dx_{n-2}} \frac{dx_{n-2}}{d\theta_{n-2}}\ , \\
&\cdots\ .
\end{aligned}$$

This can be extended all the way to the beginning of the network. For the partial derivatives with respect to weights in the start of the network we get a really long chain of partial derivatives. The derivative of the sigmoid function has at most an absolute value of 0.25, so if we would use the sigmoid function (4.1) as activation function for every layer, the partial derivatives at the beginning get exponentially smaller with the number of layers. This vanishing derivative makes it hard to train the parameters at the start of the network. ReLU activation functions (4.2) have

a derivative of 1 for positive inputs preventing the gradient from vanishing, for this reason variants of ReLU are used more often recently.

The stochastic variant of the gradient descent method differs from regular gradient descent in that it approximates the gradient using only a small and randomly chosen batch B of the training data at each iteration:

$$\nabla_\theta L(x\,;\theta) \approx \nabla_\theta L_B(x\,;\theta)$$

where

$$L_B(\theta) = \sum_{i \in B} \ell\left(y_i, f(x_i\,;\theta)\right)\ .$$

This stochastic variant has two major upsides. Firstly this makes computing the gradient much less time consuming, especially since networks are generally trained on extremely large data sets. Secondly it adds a certain amount of randomness to the updates of the weights, which serves to avoid situations where the network gets stuck at a local minimum.

One problem which naturally comes up when systems with a large amount of parameters are optimized, is overfitting. This occurs when the system represents meaningless properties which are very specific to the training data instead of representing general features which still make sense for data outside the training set. This is also a serious problem in deep learning, but this can be easily solved by the dropout technique described in [15]. During training several neurons get ignored randomly to prevent the system from getting too reliant on a very small set of neurons.

4.1.3 Convolutional neural networks

For tasks related to computer vision, convolutional neural networks (CNNs) are the method of choice. These networks use several convolutional layers instead of regular ones. A convolutional layer applies a convolution with a couple of different kernels on its input to create several new images. By stacking the new images created by the different kernels on top of each other we can look at it as a 3D volume. The sizes of the new images remain approximately the same when convolutions are applied, but the depth of the 3D volume gets exponentially bigger depending on the amount of kernels used. To stop this from getting out of hand and at the same time prevent overfitting, pooling layers are used. The purpose of these pooling layers is to downsample the images. A common way of doing this is by completely dividing the images in 2x2 regions and taking the maximum of the values in each region creating an image which is 4 times smaller.

In Fig. 4.1 an example of a convolutional neural network is shown. Instead of using pooling layers this network sometimes performs the convolutions with a stride of 2, meaning it does not place a kernel centered at every pixel of the image only at every other pixel, in both the x and y direction that is. Also a ReLU (4.2) nonlineartity is performed after each layer.

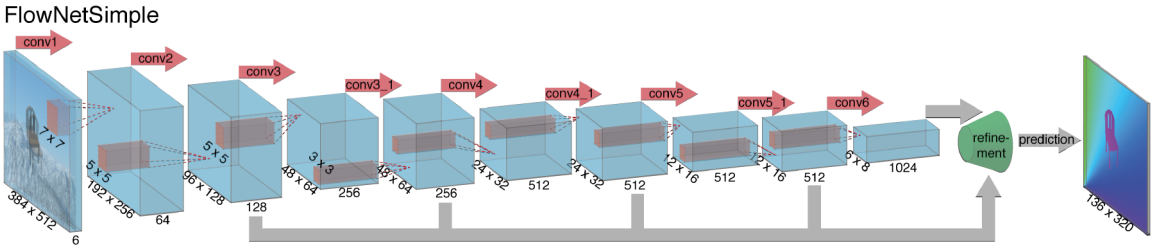


Figure 4.1: The architecture of FlowNetSimple [13]

The basic idea behind convolutional neural networks in general is to extract low level features from the image by passing it through a layer of neurons. The output of these neurons which contains low level feature information gets fed to the following layer of neurons. These neurons effectively look for the existence of certain combinations of lower level features, yielding an output which represents more abstract higher level features. This will repeat itself a number of times depending on the depth of the network, hierarchically extracting features of different levels of abstraction. At the end of the network the features are combined into a prediction for the solution.

The advantage of using convolutional layers is that we can detect the interesting features without the size of the network getting out of control. For detecting low level features it is not interesting to compare pixels which are far apart, so by using convolutions we take advantage of the 2D-structure of the data.

4.2 FlowNet 2.0

There are several different frameworks availible for deep learning implementations. Caffe [12] is one of these and the model we will be using is built on this framework. This model is called FlowNet2.0 [14] and it is an extension of FlowNetSimple [13], the model displayed in Fig. 4.1 and even uses this as one of its components. Fig. 4.2 shows the architecture of FlowNet2.0, here FlowNetS is short for FlowNetSimple.

As seen in the architecture, FlowNet2.0 uses a network of networks to compute the optical flow. The top half is responsible for the large displacements, here FlowNetC gives an approximation of the optical flow which is then used to warp the second image and passed into an instance of FlowNetS. Again the resulting flow is

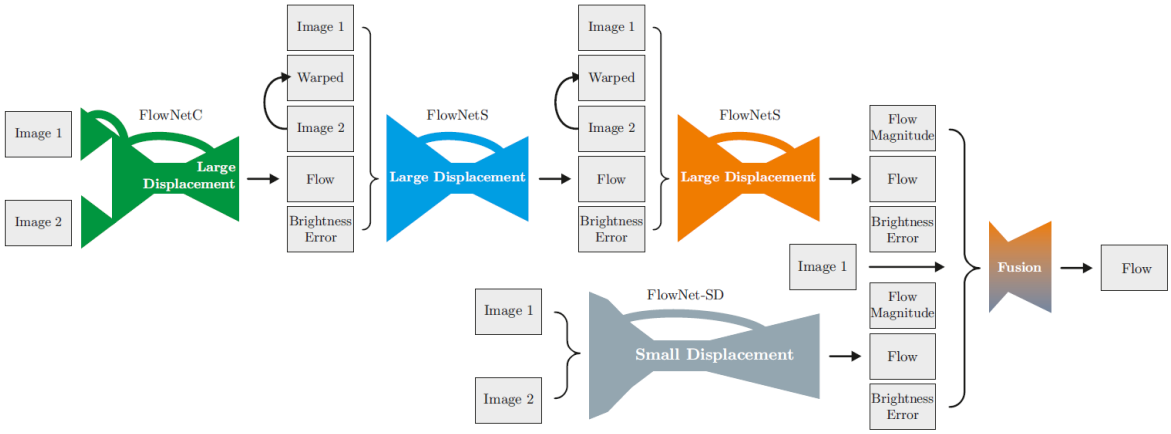


Figure 4.2: The architecture of FlowNet2.0 [14]

used to warp the second image and a new flow estimation is produced. Since these implementations have shown to be unreliable when it comes to estimating fine motions, another network, FlowNet-SD, is used to estimate the small displacements. Together they will be combined into the final flow estimation. A more extensive explanation of how each part works can be found in [14].

The training schedule is something which is just as important as the design of the architecture. For this network it has shown to be advantageous to not train it end-to-end but to train each component separately, so this is what has been done. Realistic training data for optical flow is hard to come by since it is difficult to determine the ground truth flow. The datasets we mentioned in section 2.4 are too small to train networks as large as these. So for the training of this network the FlyingChairs dataset [13] was created. This dataset took publicly available images from Flickr and placed some 3D models of chairs in front of it. Both the background and the chairs are moved slightly in between each image pair. An example of an image pair from the FlyingChairs dataset is displayed in Fig. 4.3.

Figure 4.3: Image pair from FlyingChairs with ground-truth flow [13]

Chapter 5

Testing

Now we will put the discussed models to use in order to evaluate different aspects of their performance. We will show several examples of solutions that the methods give us under certain circumstances. We performed the test on different image pairs but the examples shown here will all be of the *RubberWhale* image pair from the Middlebury dataset [5], displayed earlier in Fig. 2.6. The results should come close to the ground-truth displayed in Fig. 5.1.

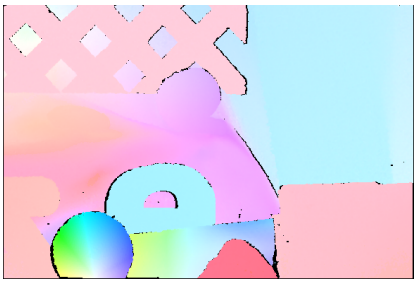


Figure 5.1: Ground-truth

5.1 Error measures

To rate the performance of a method we will use two different error measures, the absolute endpoint error (AEE) and the angular error (AE).

The absolute endpoint error measures the Euclidean distance between the predicted flow-vector v and the ground-truth flow-vector v_{gt} for all position in the image plane and takes the average value of this. So for an image with P pixels we have:

$$AEE = \frac{1}{P}\sum_{i=1}^{P} \|v(i) - v_{gt}(i)\|_2 \ .$$

The angular error measures the angle between the predicted flow-vector and the ground-truth flow-vector for all positions in the image plane and takes the average value of this. The error is calculated using the normalized vectors $\hat{v}$ and $\hat{v}_{gt}$:

$$AE = \frac{1}{P}\sum_{i=1}^{P} \arccos\left(\hat{v}(i) \cdot \hat{v}_{gt}(i)\right) \ .$$

5.2 Variational Methods

For the variational part of the testing we implemented the models of section 3.1 in MATLAB, the code can be found in Appendix A.

5.2.1 The effect of alpha

We explained that α determines how much smoothness is enforced on the solution. To see the effect of α we calculated solutions using different values of α. In Fig. 5.2 an example of this is shown where the L1-TV method is used to compute solutions.

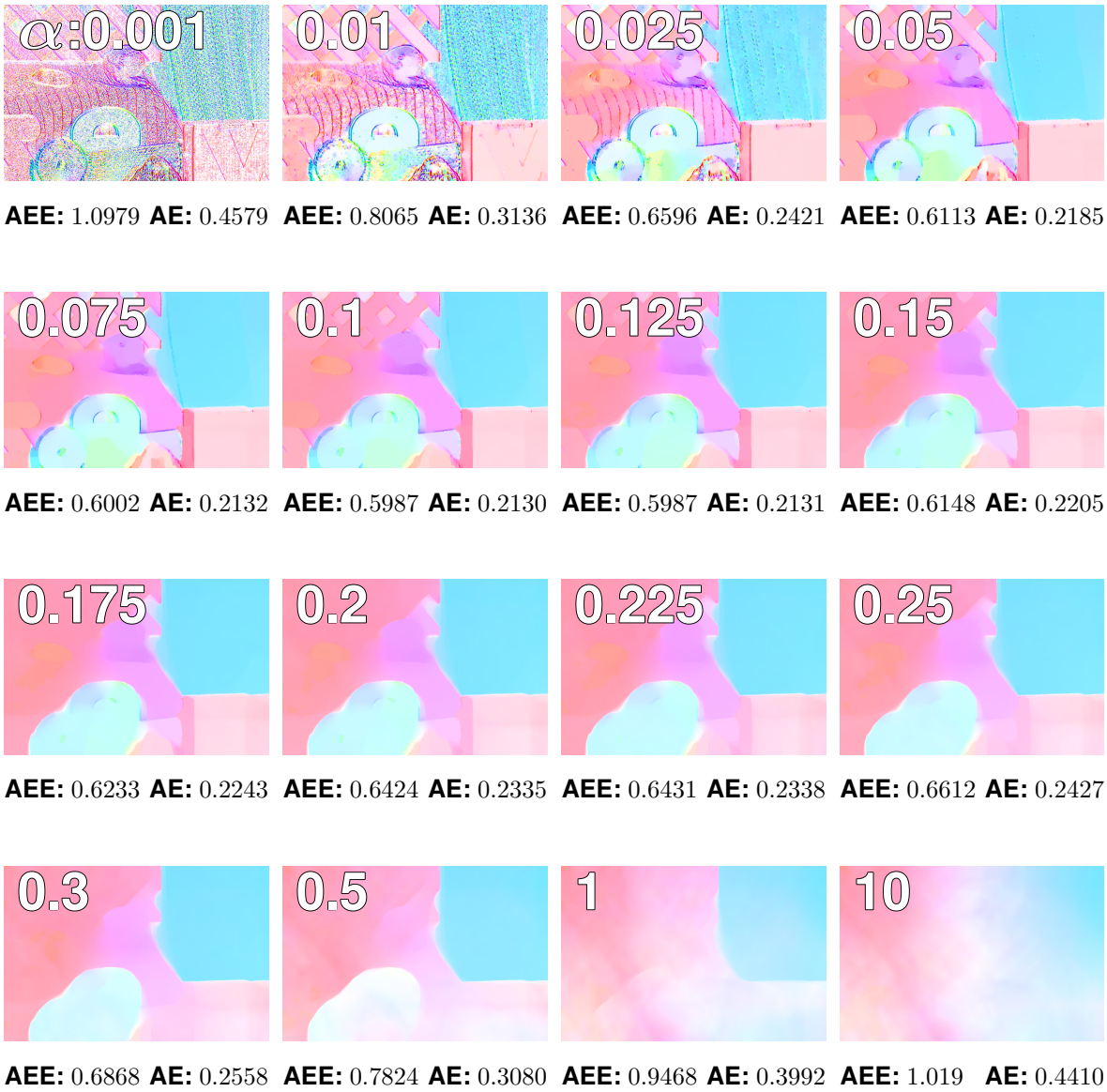


Figure 5.2: L1-TV method for different values of α.

We can see that for very small values of α the solutions show very fine details which are not realistic at all. As the parameter α is set higher, the solution gets better

and better. When α is set to some value around 0.1 we can see that it produces a result which is very similar to the ground-truth flow in Fig. 5.1 and the error measures also indicate that this α performs best. When we set α to an even higher value we see that the enforced smoothness is starting to ruin the solution. The optimal value for α differs depending on the image sequence as well as the used data- and regularization term.

5.3 Deep Learning

5.3.1 Components

An example of a solution given by the deep learning method can be seen in Fig. 5.3. We can see that it comes pretty close to the ground truth by just looking at it, but the error measures indicate this as well.

AEE: 0.1873 **AE:** 0.0717

Figure 5.3: FlowNet2.0

Since the architecture of FlowNet2.0 (Fig. 4.2) contains multiple components which all individually compute the optical flow as well we can look at the intermediate results after each component. The output of FlowNet2.0 is a direct combination of two different components, one responsible for the large displacements and one for the short displacements. The component responsible for large displacements is a chain of smaller components is indicated by FlowNet-CSS (Fig. 5.4) and the other component which is responsible for the smaller displacements is called FlowNet-SD (Fig. 5.5). We can see that these components also produce a pretty decent result if we let them work on their own.

AEE: 0.2899 **AE:** 0.1120

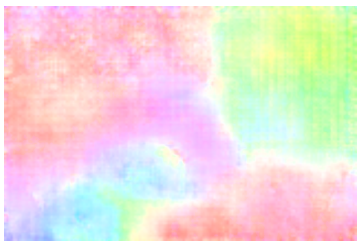
Figure 5.4: FlowNet-CSS

AEE: 0.2408 **AE:** 0.0906

Figure 5.5: FlowNet-SD

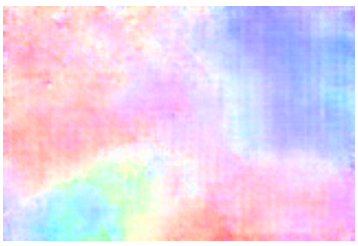
What is interesting is that when we further extract FlowNet-CSS in its smaller components FlowNet-C (Fig. 5.6) and FlowNet-S (Fig. 5.7) we see that the results

they produce come nowhere near the ground-truth.



AEE: 0.8617 **AE:** 0.3343

Figure 5.6: FlowNet-C



AEE: 0.8759 **AE:** 0.3521

Figure 5.7: FlowNet-S

However using two of these components, which perfrom really badly on their own, in succession gives us already pretty decent results as can be seen in the following figures (Fig. 5.8 and Fig. 5.9). The FlowNet-ss network from Fig. 5.9 even uses two components FlowNetS which were reduced in size by a factor $\frac{3}{8}$.

AEE: 0.3430 **AE:** 0.1401

Figure 5.8: FlowNet-CS

AEE: 0.5301 **AE:** 0.2089

Figure 5.9: FlowNet-ss

5.3.2 Performance

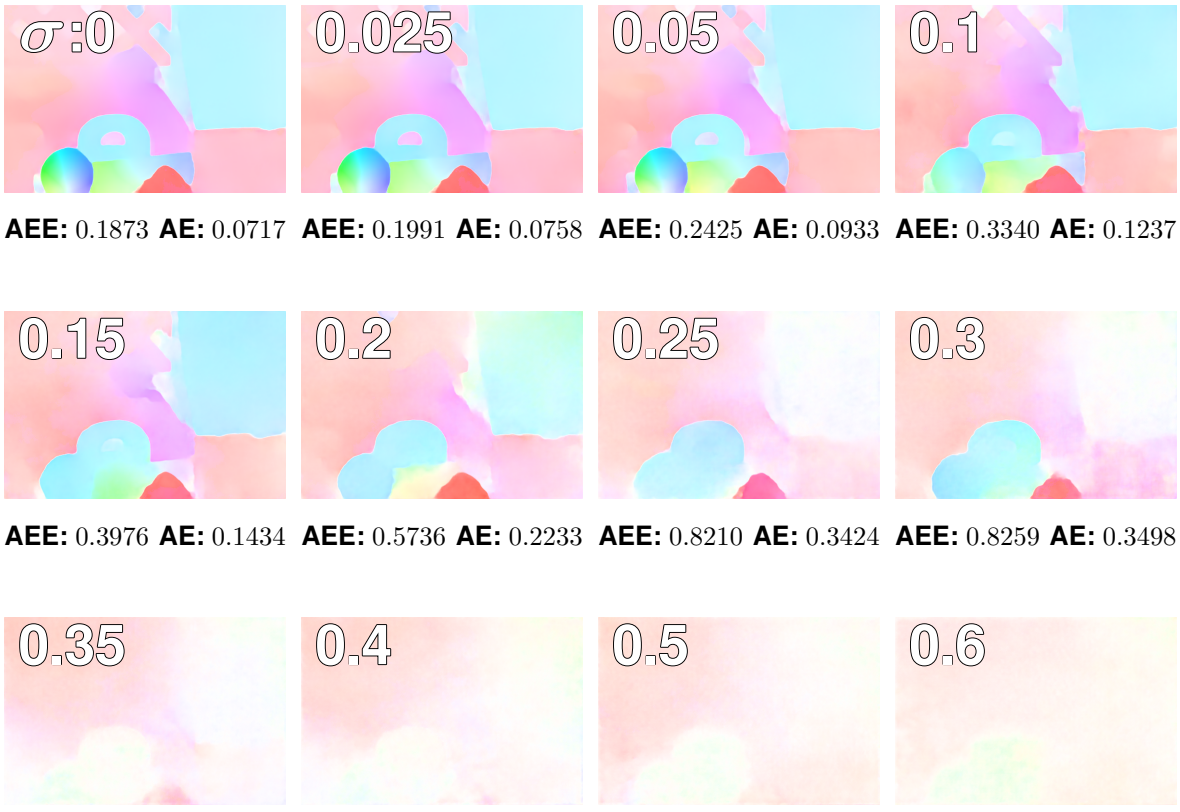
We tested the FlowNet2.0 model on the Middlebury dataset [5] containing 8 image pairs and a subset of the Sintel dataset [7]. This dataset contains 23 scenes, we tested the FlowNet2.0 model on the first image pair of every scene. The results of these tests can be seen in Table 5.1. The difference between the clean and final version of Sintel is that in the final version effects like fog and motion blur are added.

	AEE	AE
Middlebury	0.3556	0.0526
Sintel clean	1.5312	0.0602
Sintel final	2.5728	0.1058

Table 5.1: Performance of FlowNet2.0

5.3.3 Robustness against noise

In order to test how robust the FlowNet2.0 model is we tested it on some images to which some gaussian noise was added. In Fig. 5.10 can be seen how the deep learning model of FlowNet2.0 performed on the *RubberWhale* image pair.



AEE: 0.1873 **AE:** 0.0717 **AEE:** 0.1991 **AE:** 0.0758 **AEE:** 0.2425 **AE:** 0.0933 **AEE:** 0.3340 **AE:** 0.1237

AEE: 0.3976 **AE:** 0.1434 **AEE:** 0.5736 **AE:** 0.2233 **AEE:** 0.8210 **AE:** 0.3424 **AEE:** 0.8259 **AE:** 0.3498

AEE: 1.0564 **AE:** 0.4614 **AEE:** 1.0906 **AE:** 0.4816 **AEE:** 1.0694 **AE:** 0.4680 **AEE:** 1.1551 **AE:** 0.5187

Figure 5.10: The output of FlowNet2.0 for image pairs to which gaussian noise was added with different values for the variance σ.

We see that the FlowNet2.0 model handles small levels of noise really well. Even for a noise level of 0.2 the solution is not that bad considering how noisy the image in Fig. 5.11 is. One of the reasons why this model can handle small amounts of noise really well is that the training data of the model is augmented with random amounts of gaussian noise. Because of this noisy training data, the model has learned how to deal with noise.

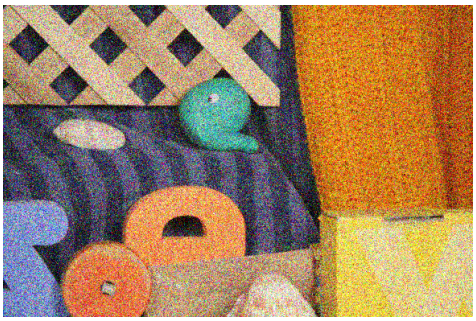


Figure 5.11: Image for $\sigma = 0.2$.

In Table 5.2 the FlowNet2.0 model is tested on the same datasets as in Table 5.1 but with a certain amount of added gaussian noise.

	Middlebury		Sintel clean		Sintel final	
σ	AEE	AE	AEE	AE	AEE	AE
0	0.356	0.053	1.531	0.060	2.573	0.106
0.025	0.385	0.056	1.510	0.069	2.413	0.112
0.05	0.435	0.064	1.434	0.075	2.3876	0.107

Table 5.2: Performance of FlowNet2.0 on data with noise.

One thing which stands out in Table 5.2 is that for both the Sintel datasets the AEE gets smaller as noise is added. This is mostly due to one particular image pair in the dataset, this image pair is displayed in Fig. 5.12.

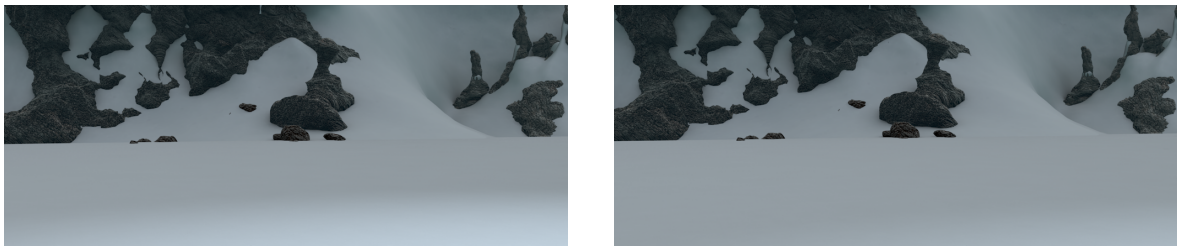


Figure 5.12: Image pair [7].

The computation of the optical flow of the uniform region in the bottom of this image pair is difficult. This results in solutions which are very different from the ground-truth as can be seen in Fig. 5.13. As we add noise, the errors at the bottom of the image get less extreme and hence the error measures get smaller.

Ground-Truth flow.

AEE: 10.926 **AE:** 0.2085 **AEE:** 9.3077 **AE:** 0.2188 **AEE:** 5.6519 **AE:** 0.2986

Figure 5.13: The output of FlowNet2.0 for the image pair from Fig. 5.12 to which gaussian noise was added with different values for the variance σ.

The error measures displayed in Fig. 5.11 are also really large compared to the average of the dataset, so it was skewing the average performance as well. When we leave the image pair from Fig. 5.12, the only extreme outlier, out of the evaluation, we get the following results:

	Sintel clean		Sintel final	
σ	AEE	AE	AEE	AE
0	1.104	0.053	2.069	0.085
0.025	1.156	0.063	2.177	0.096
0.05	1.243	0.065	2.272	0.101

Table 5.3: Performance of FlowNet2.0 on data with noise.

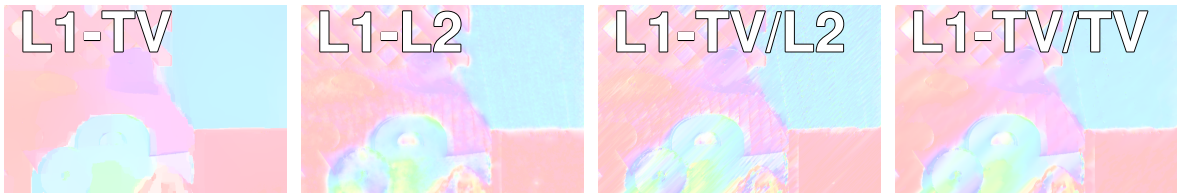
In Table 5.3 we can see that the FlowNet2.0 handles small levels of noise pretty well. Also if we compare the errors on the Sintel clean dataset to the errors on the Sintel final dataset, we see that the model apparently has more difficulties handling the effects added in Sintel final than small levels of noise.

5.4 Comparison

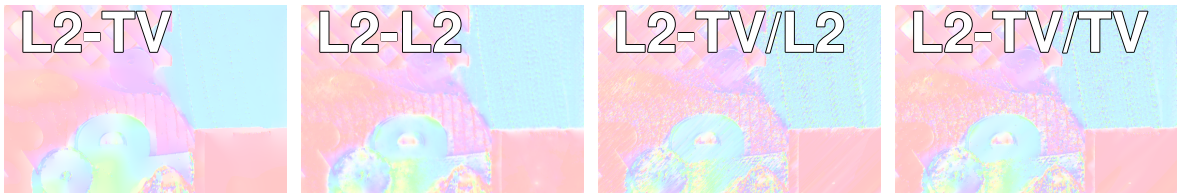
To compare the different approaches somewhat fairly we made some changes to the data sets to compensate for the limitations of our variational models. They were all turned into grayscale images, since our implementation of the variational models can not handle color. Also to compensate for the fact that our variational models do not account for large displacements, we scaled the ground-truth flow down to a maximum magnitude of 1. Using this downscaled ground-truth flow $\hat{v}_{gt}$ we created a new image pair through bilinear interpolation. The data from the first image $u(x, 1)$ remained the same and for the second image we used the interpolation of $u(x+\hat{v}_{gt}, 1)$ instead of $u(x, 2)$.

5.4.1 Performance

In Fig. 5.14 we can see how well the different models perform on the interpolated dataset we just described. We can see that of all our variational implementations the L1-TV/TV model performs the best on the *RubberWhale* image pair, just barely beating the error measures of L1-TV. If we compare the variational models to the deep learning models, we see that the deep learning models outperform every single one of our variational implementations.



AEE: 0.2098 **AE:** 0.1033 **AEE:** 0.2339 **AE:** 0.1148 **AEE:** 0.2467 **AE:** 0.1206 **AEE:** 0.2095 **AE:** 0.1032



AEE: 0.2216 **AE:** 0.1086 **AEE:** 0.2510 **AE:** 0.1230 **AEE:** 0.3396 **AE:** 0.1674 **AEE:** 0.2989 **AE:** 0.1469

AEE: 0.1353 **AE:** 0.0701 **AEE:** 0.1685 **AE:** 0.0869 **AEE:** 0.2061 **AE:** 0.1043

Figure 5.14: Comparison of the performance between the different models.

5.4.2 Robustness against noise

In Table 5.4 we compared the robustness of the deep learning implementation with that of the variational models.

	Middlebury					
	0		0.025		0.05	
	AEE	AE	AEE	AE	AEE	AE
FlowNet2.0	0.178	0.085	0.216	0.102	0.267	0.125
L1-TV	0.473	0.221	0.630	0.305	0.760	0.380
L1-L2	0.489	0.232	0.643	0.313	0.762	0.383
L2-TV	0.330	0.146	0.742	0.363	1.056	0.511
L2-L2	0.343	0.152	0.747	0.375	0.932	0.479
L1-TV/L2	0.450	0.212	0.625	0.302	0.758	0.380
L1-TV/TV	0.541	0.263	0.678	0.338	0.789	0.403
L2-TV/L2	0.697	0.269	1.769	0.602	2.124	0.664
L2-TV/TV	0.949	0.364	1.791	0.609	2.140	0.667

Table 5.4: Middlebury

In Table 5.4 we again see that the variational models perform very poorly compared to the deep learning method. Although the deep learning methods are definitely outperforming the variational models, there are two factors which skew these results. Firstly the stopping condition of the algorithm was set at a relatively low amount, namely maximum 1000 iterations. Most of the time the algorithm is not fully converged by this time, but to keep the total runtime of the test at a reasonable limit, we could not set it much higher. We see that the L2-TV model is the best performing variational implementation, but its low error measures could also just indicate that it converges the fastest. Secondly, the parameter α was not as optimized as it could have been. We ran some tests on one image pair, and used the best performing α's on the whole dataset. For the images with noise we set α to a slightly higher value in the hope of getting better results.

Another thing which stands out, is that the models with a L^2 data term handle noise significantly worse than the ones with a L^1 data term. This is what we expected. As we mentioned in subsection 3.1.1, the L^1 data term is more robust to noise.

Chapter 6

Conclusions and recommendations

6.1 Conclusions

An upside of variational models is its theoretical foundation. This theoretical foundation makes it easier to understand the performance of these models. One example of this is that we know that the L^2 dataterm is not that robust against noise, so we know what needs to be changed to improve this. Comparing this to deep learning methods we see that this does not happen as often for these methods. These models can generally only be improved through just trying several things and see which works best. The lack of theoretical foundation for deep learning methods makes it hard to predict which choices will work well and which will not.

On the other hand deep learning methods perform impressively well, the deep learning implementation of FlowNet2.0 [14] outperforms our implementations of variational methods by a large margin. It is especially surprising that the training data apparently does not need to be highly realistic, artificial image data which is created in a fairly simple way, is already enough to get really accurate solutions. Another upside of deep learning is that it computes the solution really fast. Where the variational models take minutes to converge, the FlowNet2.0 model can give the solutions in under a second. The computation time is traded in for a lot of training time, which does require a lot of GPU-power.

6.2 Recommendations

For the variational approach we used an implementation which is definitely not representative of the state-of-the-art variational methods. Our implementation has several weaknesses.

Firstly our implementation can not handle large displacements. We mentioned in section 2.2 how the displacement needs to be small if we wanted the optical flow

constraint to work. This problem could be solved by a strategy mentioned in [4], which involves computing the flow on several scaled down versions of the images and formation of image pyramids, which is mentioned in [4]. The top of the pyramid contains the images which are scaled down the most and the original images are at the bottom. Then starting at the top we can compute the optical flow and use it to warp the second image in the next layer. The warping compensates for the larger displacements detected in the previous layer, so after the warping only small displacements should remain. This can be repeated until finally the optical flow of the original image can be computed.

Moreover we could have used the color instead of the brightness of the pixels. Through using the brightness instead of the color we lose a lot of information as we only use one channel per pixel instead of three.

Bibliography

[1] [Online]. Available: http://i21www.ira.uka.de/image_sequences/

[2] F. Becker, S. Petra, and C. Schnörr, "Optical flow," in *Handbook of Mathematical Methods in Imaging*, 2015.

[3] A. Bruhn, J. Weickert, and C. Schnörr, "Lucas/kanade meets horn/schunck: Combining local and global optic flow methods," *International Journal of Computer Vision*, vol. 61, pp. 211–231, 2005.

[4] M. Burger, H. Dirks, and L. Frerking, "On optical flow models for variational motion estimation," *CoRR*, vol. abs/1512.00298, 2015.

[5] S. Baker, D. Scharstein, J. P. Lewis, S. Roth, M. J. Black, and R. Szeliski, "A database and evaluation methodology for optical flow," *Int J Comput Vis*, vol. 92, pp. 1–31, 2011.

[6] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: The kitti dataset," *International Journal of Robotics Research (IJRR)*, 2013.

[7] D. J. Butler, J. Wulff, G. B. Stanley, and M. J. Black, "A naturalistic open source movie for optical flow evaluation," in *European Conf. on Computer Vision (ECCV)*, ser. Part IV, LNCS 7577, A. Fitzgibbon et al. (Eds.), Ed. Springer-Verlag, Oct. 2012, pp. 611–625.

[8] B. K. Horn and B. G. Schunck, "Determining optical flow," *Artificial Intelligence*, vol. 17, no. 1, pp. 185 – 203, 1981. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0004370281900242

[9] A. Chambolle and T. Pock, "A first-order primal-dual algorithm for convex problems with applications to imaging," *Journal of Mathematical Imaging and Vision*, vol. 40, pp. 120–145, 2011.

[10] L. Frerking, "Variational methods for direct and indirect tracking in dynamic imaging," 2016.

[11] N. Huttinga, “Insights into deep learning methods with application to cancer imaging,” 2017.

[12] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional architecture for fast feature embedding,” *arXiv preprint arXiv:1408.5093*, 2014.

[13] A. Dosovitskiy, P. Fischer, E. Ilg, P. Häusser, C. Hazirbas, V. Golkov, P. van der Smagt, D. Cremers, and T. Brox, “Flownet: Learning optical flow with convolutional networks,” *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 2758–2766, 2015.

[14] E. Ilg, N. Mayer, T. Saikia, M. Keuper, A. Dosovitskiy, and T. Brox, “Flownet 2.0: Evolution of optical flow estimation with deep networks,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul 2017. [Online]. Available: http://lmb.informatik.uni-freiburg.de//Publications/2017/IMKDB17

[15] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: a simple way to prevent neural networks from overfitting,” *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

Appendix A

A.1 MATLAB implementation of variational models

```
function [v] = OpticalFlow(image1, image2, alpha, data, reg,
    max_iterations, tolerance, theta, x_start, y_start)
    %Image size
    [y_,x_] = size(image1);
    xy_ = x_ * y_;

    %Nabla
    %(i+1) - (i-1)
    [d1x,d1y] = GradientMatrix1(x_,y_);
    %(i+1) - (i)
    [d2x,d2y] = GradientMatrix2(x_,y_);

    %Image Derivatives
    uy = d1x*image1(:);
    ux = d1y*image1(:);
    ut = image2(:)-image1(:);

    %a square
    a_s = ux.^2+uy.^2+1e-8;

    %Matrices for building operator K
    O = sparse(xy_,xy_);
    I = speye(xy_,xy_);

    %Data-term
```

```
25      data_options = {'L2','L1'};
26      switch data
27          case data_options(1)
28              %Data-term:    L2
29              proxG = @(x,tau) prox_L2_data(x,tau);
30
31          case data_options(2)
32              %Data-term:    L1
33              proxG = @(x,tau) prox_L1_data(x,tau);
34
35          otherwise
36              disp('Error: invalid Data-term')
37      end
38
39      %Regularization-term
40      reg_options = {'L2','TV','TV/L2','TV/TV'};
41      switch reg
42          case reg_options(1)
43              %Regularization-term:    L2
44              x_size = 2;
45              y_size = 4;
46              proxFS = @(y,sigma) prox_L2_S(y,sigma);
47              K = [    d2x      O;
48                       d2y      O;
49                       O        d2x;
50                       O        d2y];
51              Ks = K';
52
53          case reg_options(2)
54              %Regularization-term:    TV
55              x_size = 2;
56              y_size = 4;
57              proxFS = @(y,sigma) prox_TV_S(y,sigma);
58              K = [d2x    O;
59                   d2y    O;
60                   O      d2x;
61                   O      d2y];
62
63              Ks = K';
```

```
64
65          case reg_options(3)
66              %Regularization-term:    TV/L2
67              x_size = 4;
68              y_size = 4;
69              proxFS = @(y,sigma) prox_TV_S(y,sigma);
70              K = [d2x    O    -I    O;
71                   d2y    O    -I    O;
72                   O      d2x   O   -I;
73                   O      d2y   O   -I];
74              Ks = K';
75              w_extension = true;
76
77          case reg_options(4)
78              %Regularization-term:    TV/TV
79              x_size = 4;
80              y_size = 8;
81              proxFS = @(y,sigma) prox_TV_S(y,sigma);
82              K = [d2x   O     -I      O;
83                   d2y   O     -I      O;
84                   O     d2x    O     -I;
85                   O     d2y    O     -I;
86                   O     O     d2x     O;
87                   O     O     d2y     O;
88                   O     O      O     d2x;
89                   O     O      O     d2y];
90              Ks = K';
91              w_extension = false;
92
93          otherwise
94              disp('Error: Invalid Regularization Term')
95      end
96
97      %Other Parameters
98      if x_size>2
99          sigma = 0.5;
100         tau = 0.25;
101     else
102         sigma = 1/3;
```

```
103         tau = 0.2;
104     end
105
106     %Initial Values
107     if nargin>8
108         x = x_start;
109     else
110         x = zeros(x_size*xy_,1);
111     end
112     if nargin>9
113         y = y_start;
114     else
115         y = zeros(y_size*xy_,1);
116     end
117     x_bar = x;
118     dx = Inf;
119     iterations = 0;
120
121     %Main Loop
122     while iterations<max_iterations && dx>tolerance
123         x_old = x;
124
125         y = proxFS(y + sigma*K*x_bar,sigma);
126         x = proxG(x - tau * Ks * y,tau);
127         x_bar = x + theta * (x - x_old);
128
129         dx = sum(abs(x(1:2*xy_) - x_old(1:2*xy_)))/xy_;
130         iterations = iterations + 1;
131     end
132
133     %Resize x
134     v = reshape(x(1:2*xy_),[size(image1),2]);
135
136     %Functions
137     function [x] = prox_L1_data(xhat,tau)
138         rho = ut + ux.*xhat(1:xy_) + uy.*xhat(xy_+1:2*xy_);
139
140         cond1 = rho < -tau * a_s;
141         val1 = tau * [ux;uy];
```

```
142         cond2 = rho > tau * a_s;
143         val2 = -val1;
144         cond3 = 1-cond1-cond2;
145         val3 = -[rho;rho] .* [ux./a_s;uy./a_s];
146 
147         x = xhat(1:2*xy_) + val1 .* [cond1;cond1] + val2 .* [
               cond2;cond2] + val3 .* [cond3;cond3];
148         if(x_size>2)
149             if w_extension
150                 w = (xhat(2*xy_+1:end))/(1+tau*alpha(2));
151                 x = [x;w];
152             else
153                 x = [x;xhat(2*xy_+1:end)];
154             end
155         end
156     end
157     function [x] = prox_L2_data(xhat,tau)
158         a1 = 1 + tau*ux.*ux;
159         a2 = tau * ux.*uy;
160         a3 = 1 + tau*uy.*uy;
161         b1 = xhat(1:xy_) - tau*ux.*ut;
162         b2 = xhat(xy_+1:2*xy_) - tau*uy.*ut;
163 
164         v1 = (a3.*b1-a2.*b2)./(a1.*a3-a2.^2+1e-8);
165         v2 = (a1.*b2-a2.*b1)./(a1.*a3-a2.^2+1e-8);
166         x = [v1;v2];
167         if(x_size>2)
168             if w_extension
169                 w = (xhat(2*xy_+1:end))/(1+tau*alpha(2));
170                 x = [x;w];
171             else
172                 x = [x;xhat(2*xy_+1:end)];
173             end
174         end
175     end
176     function [x] = prox_TV_S(xhat,~)
177         isotropic = true;
178         if x_size>2
179             alpha_ = alpha(1);
```

```
180         else
181             alpha_ = alpha;
182         end
183         if ~isotropic
184             %Anisotropic
185             x = max(-alpha_,(min(alpha_,xhat(1:4*xy_))));
186         else
187             %Isotropic
188             xnorm = reshape(xhat(1:4*xy_),[xy_,4]);
189             xnorm1 = (sum(xnorm(:,1:2).^2,2)).^0.5;
190             xnorm2 = (sum(xnorm(:,3:4).^2,2)).^0.5;
191             xnorm = [xnorm1;xnorm1;xnorm2;xnorm2];
192             x = xhat(1:4*xy_)./(max(1,xnorm/alpha_));
193         end
194         if y_size>4
195             alpha_ = alpha(2);
196             if ~isotropic
197                 %Anisotropic
198                 w = max(-alpha_,(min(alpha_,xhat(4*xy_+1:end)
                        )));
199             else
200                 %Isotropic
201                 xnorm = reshape(xhat(4*xy_+1:end),[xy_,4]);
202                 xnorm1 = (sum(xnorm(:,1:2).^2,2)).^0.5;
203                 xnorm2 = (sum(xnorm(:,3:4).^2,2)).^0.5;
204                 xnorm = [xnorm1;xnorm1;xnorm2;xnorm2];
205                 w = xhat(4*xy_+1:end)./(max(1,xnorm/alpha_));
206             end
207             x = [x;w];
208         end
209     end
210     function [x] = prox_L2_S(xhat,tau)
211         x = xhat/(1+tau/alpha);
212     end
213 end
```