ETH zürich

ETH Library

Self-Supervised Learning of Non-Rigid Residual Flow and Ego-Motion in Dynamic 3D Scenes

Master Thesis

Author(s):
Tishchenko, Ivan

Publication date:
2020

Permanent link:
https://doi.org/10.3929/ethz-b-000431668

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Self-Supervised Learning of Non-Rigid Residual Flow and Ego-Motion in Dynamic 3D Scenes

Master's Thesis

Ivan Tishchenko

May 29, 2020

Advisors: Sandro Lombardi, Dr. Martin Oswald

Supervisor: Prof. Dr. Marc Pollefeys

Department of Computer Science, ETH Zürich

Computer Vision and Geometry Lab

Abstract

Recent supervised 3D deep learning methods have demonstrated a remarkable performance in the area of scene flow estimation. Nevertheless, one common pitfall of the majority of methods is that they rely on large quantities of synthetic data to train deep neural networks with millions of parameters. This is caused by the fact that acquiring the ground truth for scene flow is very hard or sometimes impossible for real world datasets. Training on synthetic datasets introduces the domain gap when the model trained on a synthetic dataset from one domain will not necessary generalize to real world data from another domain. Furthermore, the majority of current methods cannot be applied to datasets with incomplete labels or to real worlds datasets with no ground truth at all. We address this issue by extending the current state of the art supervised approaches with self-supervisory signals based on the temporal consistency of a sequence of point clouds.

Secondly, most of the scene flow methods model scene flow as a per point translation vector without any assumption of ego-motion. In our work we propose a different way for learning total scene flow, where we jointly learn the non-rigid part in form of the residual non-rigid flow and the rigid part of the motion by learning the ego-motion flow. We investigate the effect of self-supervision on a model that jointly learns the decomposed flow and contrast it to models which learn the total flow directly.

Our solution allows both hybrid training with a supervised loss and self-supervisory signals as well as training in a fully self-supervised mode. The former can be beneficial in a weakly supervised setting, where the labels are inaccurate or incomplete. The later can be used for training on real world datasets with no ground-truth at all or for fine-tuning in self-supervised mode after supervised pre-training.

Experiments showed that complementing the supervised part of the total loss with the self-supervised signals allowed us to outperform the current state of the art supervised baseline. Internal comparison has demonstrated that pre-training on one dataset and fully supervised fine-tuning on another dataset can help to alleviate the domain gap. On simpler scenes fully supervised models have delivered comparable performance to supervised methods in terms of qualitative evaluation. Scene flow decomposition has improved the performance if trained in a fully self-supervised mode.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my advisors Sandro Lombardi and Dr. Martin Oswald for their outstanding supervision, insightful ideas and continuous support. Their guidance and suggestions during our weekly meetings and beyond them have significantly contributed to the quality of this work.

I would like to thank Prof. Marc Pollefeys for providing an opportunity to work on this project in his group and for the excellent and motivating environment.

My special thanks go to Dr. Petri Tanskanen and Dr. Olivier Saurer for a valuable discussion on selecting a topic for this thesis and sharing their experience on doing research.

Finally, I want to thank my family and friends for always being there for me with their constant support and encouragement during this project.

Nomenclature

$\mathcal{X}_t$	Point cloud at time step t
T_{rel}	Relative camera pose between time step t and $t+\delta$
$\vec{x}$	Point in a point cloud $\mathcal{X}$
$\vec{d}$	Total scene flow
$\vec{d}_{nr}$	Non-rigid residual scene flow
$\vec{d}_{cm}$	Ego-motion scene flow
$\vec{v}$	Optical flow
$\vec{v}_{nr}$	Projected non-rigid residual flow
$\vec{v}_{cm}$	Projected ego-motion flow
A_d^*	Permutohedral lattice of dimension d
EPE3D	End Point Error 3D
ROE	Relative Orientation Error
RLE	Relative Location Error
NN	Nearest Neighbour
FB	Forward-Backward Consistency

Contents

Abstract i

Acknowledgments ii

Nomenclature iii

Contents v

List of Figures vii

List of Tables viii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Problem Statement . . . 3
- 1.3 Focus of This Work . . . 5
- 1.4 Thesis Structure . . . 5

2 Related Work 7
- 2.1 Deep Scene Flow . . . 7
- 2.2 Self-Supervised Learning . . . 8
- 2.3 Ego-Motion and Relative Camera Pose Estimation . . . 9
- 2.4 Feature Comparison . . . 10

3 Theory 11
- 3.1 3D Motion Model . . . 11
- 3.2 Learning in 3D Space . . . 14
- 3.3 Neural Networks for Point Clouds . . . 15
  - 3.3.1 Permutohedral Lattice Networks . . . 16
- 3.4 Self-Supervised Learning . . . 18
  - 3.4.1 Forward-Backward Consistency . . . 19

3.4.2 Cycle Consistency 21

4 Method **23**
4.1 Overview 23
4.2 Self-Supervisory Signals 24
4.2.1 Nearest Neighbour Search 25
4.3 Model's Architecture 26
4.3.1 Encoder 26
4.3.2 Relative Pose Regressor 27
4.3.3 Decoder 28
4.4 Loss Functions 28

5 Experiments **33**
5.1 Datasets 33
5.1.1 Input Data and Ground Truth 33
5.1.2 FlyingThings3D 34
5.1.3 RefRESH 35
5.1.4 KITTI 35
5.2 Evaluation Metrics 36
5.3 Training Setup 37
5.4 Quantitative Study 37
5.4.1 Self-Supervision in Total Scene Flow Learning 37
5.4.2 Non-Rigid Flow and Ego-Motion 41
5.4.3 Comparison to Related Methods 45
5.5 Qualitative Study 46

6 Conclusion **51**
6.1 Future Work 52

A Additional Qualitative Results **55**
A.1 FlyingThings3D 55
A.2 RefRESH 55
A.3 KITTI 55

Bibliography **59**

List of Figures

1.1 Example input point clouds 2
1.2 Example of total scene flow on a pair of point clouds 4

3.1 Full scene flow model and its relation to optical flow 12
3.2 Visualization of the full scene flow model 13
3.3 Delaunay tessellation for two dimensional permutohedral lattice 16
3.4 Constriction of a permutohedral lattice 17
3.5 Illustration of position vector embedding and a Splat-Blur-Slice operation 19
3.6 Demonstration of forward-backward consistency 20
3.7 Demonstration of cycle consistency 21

4.1 High-level overview of the learning pipeline 24

5.1 Convergence of loss functions in self-supervision experiments . . 40
5.2 Convergence of loss functions in non-rigid flow and ego-motion experiments 43
5.3 Qualitative evaluation on FlyingThings3D 48
5.4 Qualitative evaluation on RefRESH 49
5.5 Qualitative evaluation on KITTI 50

6.1 Lower bound for total flow estimation 53

A.1 Additional qualitative evaluation on FlyingThings3D 56
A.2 Additional qualitative evaluation on RefRESH 57
A.3 Additional qualitative evaluation on KITTI 58

List of Tables

2.1 Comparison to related work. 10

5.1 Overview of scene flow datasets. 33
5.2 Evaluation of total scene flow in self-supervision experiments . . 39
5.3 Evaluation of total flow from non-rigid flow and ego-motion . . 42
5.4 Evaluation of non-rigid flow 44
5.5 Evaluation of relative camera pose estimation 45
5.6 Quantitative comparison to other methods 45

Chapter 1

Introduction

This chapter servers as an overview and a starting point for the whole project. Section 1.1 introduces the reader to the area and presents the motivation of our work while discussing examples of potential application areas. In section 1.2, we define the problem this project is addressing along with our main contributions. We finish the chapter with an overview of the structure of this thesis in Section 1.4.

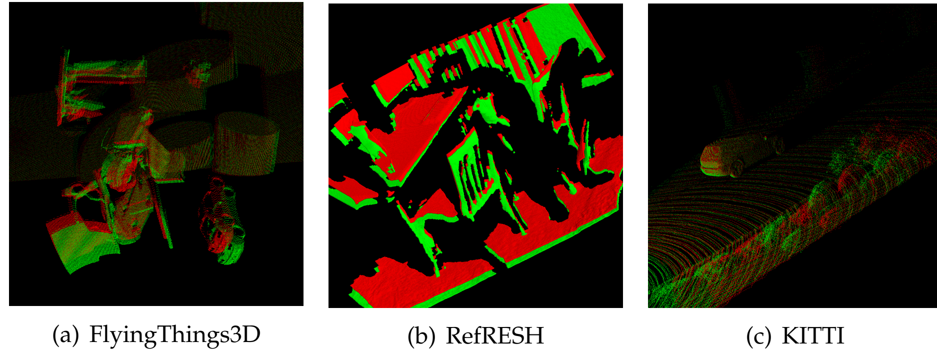
1.1 Motivation

Recent advances in Convolutional Neural Networks (CNNs) allowed us to achieve human-level performance in 2D computer vision tasks. Researchers have been able to design very accurate systems for understating 2D motion, which is most commonly represented in a form of optical flow.

However, the world and all dynamic objects that we perceive around us are three dimensional (3D). In order to build fully autonomous agents we need to enable these agents to perceive the world in 3D and also to do reasoning about it directly in 3D. Estimation and understanding of motion in 3D scenes is an essential building block to advance emerging technologies in the areas such as dynamic reconstruction, robotic perception, drone navigation, autonomous driving, VR/AR, human-computer interaction etc.

Rapid advances in data acquisition such as LiDAR, RGB-D cameras and 3D scanners facilitated a direct acquisition of 3D representations of the scene in form of point clouds. Driven by the success in 2D motion estimation with deep neural networks, the focus of current research on 3D motion estimation has shifted to scene flow estimation with deep learning based approaches. Modern methods [16, 30, 46] are able to process 3D point clouds directly without relying on stereo or RGB-D inputs. Advances in network architectures and hardware capabilities allow us to directly process a large number

of point clouds fast and efficiently. Figure 1.1 shows three examples of input point clouds from various types of common scene flow datasets.



(a) FlyingThings3D (b) RefRESH (c) KITTI

Figure 1.1: Example input point clouds. (a) are point clouds produced from a synthetic dataset. (b) shows points from a semi-synthetic dataset (c) is a point cloud from a real world dataset.

Deep learning based methods [16, 31] have been successfully used for supervised scene flow estimation. However, one pitfall common for most of the learning based methods is that in order to achieve good results one needs to employ very deep networks with a large number of trainable weights. This in turn requires large amounts of labelled observations. Acquiring the ground truth of scene flow for real world data is intractable or sometimes even impossible. As a consequence, researches have relied on using rendered synthetic data-sets [33, 31, 36, 35] to train the networks on. This spawns a discrepancy between real world and synthetic data. Concretely, networks trained on synthetic data are not guaranteed to generalize to raw real world data. The reason for a gap is that synthetic datasets are not always shape and depth realistic and that they generally represent one specific narrow domain. The way to tackle these issues is to utilize large amounts of unlabeled data.

Self-supervised learning has shown its effectiveness in bridging the gap between synthetic and real-world data for the case of optical flow [29]. Although most of the methods based on fully self-supervised learning are not yet performing as good as the supervised learning versions, the difference in performance becomes smaller and smaller. The main advantage of fully self-supervised methods is that we do not need any labelled data to train the network on. Additionally, the combination of supervised learning and self-supervision is possible with pre-training and fine-tuning. In this way we can utilize the best parts of two approaches. Furthermore, researching self-supervision is also beneficial from the supervised learning perspective. Specifically, we can design self-supervised losses (signals) which serve as

implicit regularizers for the supervised part of the loss. This kind of configuration can also be useful in a weakly supervised setting, where the method can still learn from datasets with incomplete labels and put more weight on the self-supervised part of the loss if the labels in the dataset are inaccurate.

Another common property of learning based approaches for scene flow [16, 30] is that most of them model scene flow holistically as a 3D translation vector without any distinct separation of camera and object motion. While this type of scene flow modelling is acceptable in case of static scenes, it is beneficial for the case of dynamic scene to disentangle the observer (camera) motion from the object motion. This view on scene flow will allow us to deduce which part of the scene flow is induced by the ego-motion of an observer and which is induced by the object itself. For practical applications it is often useful to obtain pure object motion, relative camera pose and the total motion in just one single forward pass.

1.2 Problem Statement

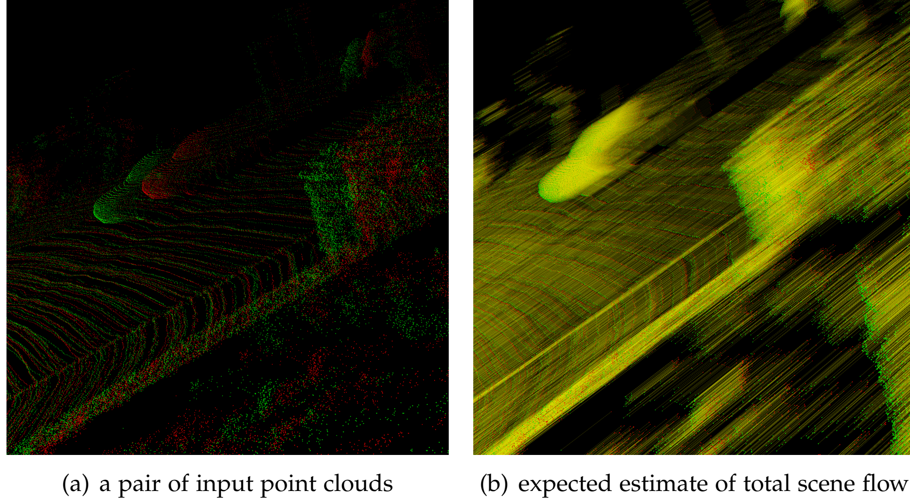
Let $\mathcal{X}_t \in \mathbb{R}^{n \times d}$ be a point cloud representation of the scene at time step t and $\mathcal{X}_{t+\delta} \in \mathbb{R}^{m \times d}$ be a point cloud at time step $t + \delta$. This pair of point clouds has the following properties:

- Points clouds dimensions n and m may or may not be the same.
- Points in these two point clouds are not ordered.
- The correspondences between a pair of point clouds are not enforced i.e. some point $\vec{x}$ in $\mathcal{X}_t$ may not be represented in $\mathcal{X}_{t+\delta}$

Each point $\vec{x} \in R^d$ in the point cloud $\mathcal{X}$ is represented as a vector of spatial location in 3D space. Every point is also associated with a signal vector $\vec{v} \in \mathbb{R}^k$, where a signal in general case can include information such as a color vector, a normal vector, an intensity vector in addition to the original spatial position vector, which can make the signal dimension $k > d$. In the scope of this work our signal is represented by the spatial location only. Therefore, $\vec{v} = \vec{x}$ with dimensions $d = k = 3$.

Having $\mathcal{X}_t$ and $\mathcal{X}_{t+\delta}$ our main goal is to estimate total scene flow $\vec{d} \in \mathbb{R}^3$ sometimes also known as 3D motion field. Total scene flow is a per-point displacement vector which describes where a point in $\mathcal{X}_t$ will be in $\mathcal{X}_{t+\delta}$. Total scene flow is closely related to optical flow. Specifically, if we would project total scene flow vector onto a 2D image plain, we will get the optical flow. Figure 1.2 contains a visualization of input point clouds and the total flow we are trying to estimate.

Furthermore, we propose to decompose the total scene flow into a pair of components:



(a) a pair of input point clouds (b) expected estimate of total scene flow

Figure 1.2: Example of total scene flow on a pair of point clouds. (a) red and green represent the membership to corresponding point clouds (b) is a visualization of total scene flow in a form of yellow displacement vectors.

$$\vec{d} = \vec{d}_{nr} + \vec{d}_{cm} \tag{1.1}$$

where $\vec{d}_{nr} \in \mathbb{R}^3$ represents the per point non-rigid flow which is induced by the moving object itself. Additionally, there is the ego-motion flow $\vec{d}_{cm} \in \mathbb{R}^3$ which represents per point motion induced by the movement of an observer (camera). Ego-motion flow is defined in the following equation:

$$\vec{d}_{cm} = (R_{rel} - I_3)\,\vec{x}_t + t_{rel} \tag{1.2}$$

where $R_{rel} \in \mathbb{R}^{3\times3}$ is the relative rotation of the camera and $t_{rel} \in \mathbb{R}^3$ represents relative translation of the camera from time step t to time step $t+\delta$. Furthermore, variables $\vec{d}_{cm}$, $\vec{x}_t$ and T_{rel} can also be expressed in homogeneous coordinates using a more compact form:

$$\vec{d}_{cm} = \left(\begin{bmatrix} R_{rel} & t_{rel} \end{bmatrix} - I_4\right)\vec{x}_t \tag{1.3}$$

where $T_{rel} = \begin{bmatrix} R_{rel} & t_{rel} \end{bmatrix} \in R^{4\times4}$ is the relative camera pose between a point cloud at time step t and $t+\delta$ expressed in homogeneous coordinates.

Hence, our second goal is to estimate the relative camera pose T_{rel} as well as $\vec{d}_{nr}$. We can combine the two using equations 1.2, 1.1 and obtain total scene flow $\vec{d}$.

1.3 Focus of This Work

Our work complements previous works on scene flow learning with 3D deep learning [16, 46, 30] by adding self-supervision and a different approach for learning scene flow which disentangles ego-motion from the object motion. Furthermore, it is important to note that there have been projects [37, 54, 50] which explored self-supervision for scene flow in parallel with us. However, all of these methods rely on total scene flow only. While there have been works on separating the camera motion and object motion for scene flow, most of them learn on 2.5D data [31, 28], which makes them inapplicable if the original data is in point cloud format. Moreover, they also do not offer an end-to-end learning pipeline but rather learn different components with separate networks. Besides that, these works do not utilize self-supervision [6, 31].

In summary, the contribution of this work is a combination of the following parts into an end-to-end 3D deep learning pipeline. Our work aggregates:

- Learning of total scene flow with 3D deep learning from point clouds. This is achieved by modification of total flow architecture HPLFlowNet [16] to fit the purpose of joint learning of non-rigid flow and relative ego-motion.
- Investigation of self-supervision for scene flow learning in form of supervised learning with self-supervisory signals, fine-tuning in fully self-supervised mode and fully self-supervised learning from scratch. This includes design and implementation of architecture agnostic self-supervisory signals.
- Novel view on 3D scene flow which decomposes the total flow into non-rigid and ego-motion parts. Comparison with a traditional motion model of holistic total scene flow. Studying the effect of self-supervision on the decomposed model.

1.4 Thesis Structure

This thesis is structured in the following way. Chapter 2 reviews related work and compares its features to the ones of our project. Concretely, it covers the areas such as classical scene flow estimation, deep scene flow estimation, ego-motion estimation and self-supervision. Chapter 3 provides a theoretical basis for our approach and formally discusses and arguments our architectural choices. We present the details of our method in chapter 4, which includes giving details on the overview of the pipeline, our model and loss functions. Chapter 5 presents our experiments, their setup and their outcome. Our experimentation consists of an internal quantitative comparison and an internal qualitative evaluation along with a quantitative

comparison to the related methods. Finally, we conclude on the results and on the outcome of the entire project and outline the directions for future work in chapter 6. We provide additional visualizations of scene flow predictions in the appendix A.

Chapter 2

Related Work

In this chapter we introduce the literature related to our approach. Section 2.1 overviews the evolution of scene flow estimation specifically focusing on approaches based on 3D deep learning. Section 2.3 summarizes the work conducted on ego-motion estimation and relative camera pose estimation with deep neural networks. In section 2.2 we introduce self-supervision and how it can be applied to scene flow learning. Finally, in section 2.4 we conduct a comparative analysis between the features of our work and other most related approaches.

2.1 Deep Scene Flow

Scene flow was formally defined in the seminal work of Vedula et. al [48], in which they proposed a three-dimensional dense non-rigid motion model for describing motion directly in 3D analogously to optical flow in 2D. Our work is based on current approaches in 3D scene flow estimation when we learn the scene flow directly on 3D point clouds [30, 46, 16] with an end-to-end pipeline rather than utilizing stereo or RGB-D inputs [19, 21] and learning the motion in form of optical flow and depth separately [31, 28, 32]. To obtain large amounts of training data researchers have been employing mostly synthetic datasets [33, 31, 36, 35] and went from the stereo or RGB-D format into the point cloud format.

Point cloud based methods [30, 39, 40, 16] which learn scene flow from point clouds share a common pattern. Specifically, this patter consists of taking a pair of point clouds, optionally fusing [58] the information from both frames, encoding them into a feature vector and then up-sampling them to get a per-point scene flow output.

A group of approaches has proposed to learn scene flow with a hierarchical permutohedral lattice [16, 46, 23], in contrast to learning on a regular 3D

grid or voxels, which are known to be computationally restrictive if one is to go into more than three dimensions or to increase the density of input point clouds. Approaches based on permutohedral lattice additionally put a focus on efficiency and aim to process as many points as possible in one forward pass to avoid chunking. The majority of them take their inspiration from earlier works which had success in high-dimensional image filtering using permutohedral lattices [2, 5, 1], which latter adopted a learning based path with a rise of deep learning and evolved into permutohedral networks [25, 22].

2.2 Self-Supervised Learning

Work by Wang et. al [49] has demonstrated how cycle-consistency of time and its special case forward-backward consistency can serve as a powerful source of self-supervision for a wide variety of visual correspondence tasks. These techniques found their applications in the realm of optical flow [29, 52]. Liu et. al [30] proposed to use cycle-consistency for scene flow estimation in their self-supervised loss which was used in a combination with a supervised part.

Self-supervision for scene flow learning is an active field of research and there are concurrent works [37, 54, 50] to ours. Our method is largely influenced by findings of Mittal et. al [37], where they proposed to extend the idea of cycle-consistency [30, 49] with the nearest neighbour anchoring and the nearest neighbour constraint inspired by ICP [7], which employs distance metrics between point sets as a source of self-supervision. Similar idea was previously utilized by Fan et. al [13] in their Earth Mover's Distance (EMD) constraint, which essentially forced the points of one set to be close to the points in the other set. Some works [54, 13] extended the idea of nearest neighbour loss used in [37] to both forward and backward directions in form of Chamfer Distance (CD) constraint.

An alternative source of self-supervision was investigated by Wang et. al [50], where they exploited geometric structural properties of 3D data such as surface normals in order to to pose a point-to-plane error constraint and additionally used a cosine similarity constraint for the flow vector along with their supervised part of the loss. Constructing of self-supervision constraints based on surface properties were also investigated by Wu et. al [54]. In their approach they proposed to use a smoothness constraint to enforce local spatial smoothness along with the Laplacian regularization [45] to enforce that the Laplacian coordinate vector is similar across a pair of frames. In this case Laplacian coordinate vector acts as a local descriptor of the surface.

Overall, the best performance is achieved in case when a supervised loss

is used along with self-supervisory constraints [50, 54]. However, it is also possible to train with pure self-supervision, in this case it is important to select the constraints which would accurately approximate the supervised loss and naturally reflect the properties of 3D data such as the shape differences [13, 37].

2.3 Ego-Motion and Relative Camera Pose Estimation

Most of scene flow methods model scene flow as a 3D translation vector [16, 30, 46]. An alternative view on scene flow is to disentangle camera motion from the object motion in real world dynamic scenes. In this kind of approach, one generally models camera motion as a rigid transformation and the rest of the motion as non-rigid per point translation [31, 28, 12, 41]. Behl et. al [6] proposed to jointly learn per-point rigid transformation along with ego-motion and bounding boxes in an end-to-end learning pipeline on point clouds.

Camera pose estimation with deep learning originally started from learning with CNNs on RGB images [24, 43, 34, 10]. These works proposed to learn an absolute camera pose transform or a relative camera pose transform with drop-in replacement modifications of popular CNNs, where the networks were adapted from classification to pose regression by modifying the final layers. Most of these approaches encode the information from two frames with a pair of Siamese branches, which share the weights and regress the pose either as a 7 dimensional vector in case when the rotation is modelled with quaternions [34, 24], or a 6 dimensional vector if Euler angles are used for the rotation portion of the pose [31].

Several RGB and RGB-D based methods applied similar techniques for the relative camera pose part [8, 31]. Similar ideas have been also used for optical flow and ego-motion learning [27, 56, 26]. Sattler et. al [42] have showed that estimating the pose only from 2D images is prone to poor generalization. In contrast to image based pose estimation with CNNs, our approach is more closely related to [6, 14, 20] where we learn the relative pose from point clouds with a 3D neural network.

Another area of research closely related to our work is iterative point cloud registration with 3D deep neural networks [57, 3, 15]. The learning based part of these approaches is similar to [6, 14] if one considers an estimation of a rigid transformation. When it comes to the algorithmic idea, they are inspired by iterative registration of point clouds with ICP [4, 7]. Specifically, they estimate an initial rigid transformation with a point based network and then apply that transformation to the inputs. Afterwards, the same process is repeated K times. At the end they obtain a final pose, which is a matrix product of K intermediate steps.

The downside of applying a similar idea to our approach is the computational bottleneck. Most of these methods use less powerful and smaller networks such as [39, 40]. Another point to consider is the fact that it might be intractable in terms of GPU memory and time to unroll the computation into K steps with the network [16] we base our model on. Furthermore, if one is to include self-supervision the computational burden will additionally increase. Finally, in our work we aim to learn the pose and non-rigid flow with an end-to-end pipeline, unrolling the computation of camera pose into K steps would mean that we have to separate a network into two sequential networks, one for the camera pose and one for the flow.

2.4 Feature Comparison

We present a feature comparison of our work and approaches of other authors in order to summarize an overview of the related literature along with the contribution from chapter 1. Feature comparison of our work and most related approaches is demonstrated in table 2.1.

Approach	Scene representation	Self-supervision	Ego-motion	End-to-end
HPLFlowNet [16]	Point clouds	-	-	+
FlowNet3D [30]	Point clouds	+	-	+
Learning Rigidity [31]	RGB-D	-	+	-
PointFlowNet [6]	Point clouds	-	+	+
Just Go with the Flow [37]	Point clouds	+	-	+
PointPWC-Net [54]	Point clouds	+	-	+
FlowNet3D++ [50]	Point clouds	+	-	+
Ours	**Point clouds**	**+**	**+**	**+**

Table 2.1: Comparison to related work.

We can deduce from the table that our approach meets project's requirements the best by combining self-supervision and the ability to learn egomotion into an end-to-end point cloud based network.

Chapter 3

Theory

In this chapter we introduce the reader to the theoretical background needed to comprehend our work. Section 3.1 explains the scene flow model crucial for understanding of our method and all of the experiments. In section 3.2 we motivate the choice of our selected architecture and compare learning in 3D against learning in 2.5D. Section 3.3 provides the theory on various ways for processing point clouds. In section 3.4 we describe the theory of self-supervision we used in our method.

3.1 3D Motion Model

Continuing the discussion from chapter 1 we present the scene flow motion model, its relation to optical flow and how we can differentiate between the ego-motion and the actual motion of the scene. Full scene flow model with its relation to optical flow is illustrated in figure 3.1.

In this setup we have two camera poses $T_1 \in \mathbb{R}^{4\times4}$ and $T_2 \in \mathbb{R}^{4\times4}$ from two time steps which describe a rigid transformation from the corresponding camera coordinate system into the world coordinate system:

$$T_1 = \begin{bmatrix} R_1 & t_1 \\ 0^T & 1 \end{bmatrix} \qquad T_2 = \begin{bmatrix} R_2 & t_2 \\ 0^T & 1 \end{bmatrix} \tag{3.1}$$

In our case [1] the centers in world coordinates of coordinate systems of a pair of cameras corresponding to T_1 and T_2 are located at $C_1 = t_1$ and $C_2 = t_2$. The orientations of cameras in world coordinates are simply given by R_1 and R_2.

[1] Please note that some of the data-set and rendering software may follow the world to camera transformation convention when it comes to camera poses. In that case the equations for camera centers, relative pose etc. will a have a different form. For more detail we refer the reader to [18], which one of the examples where the opposite convention is used.

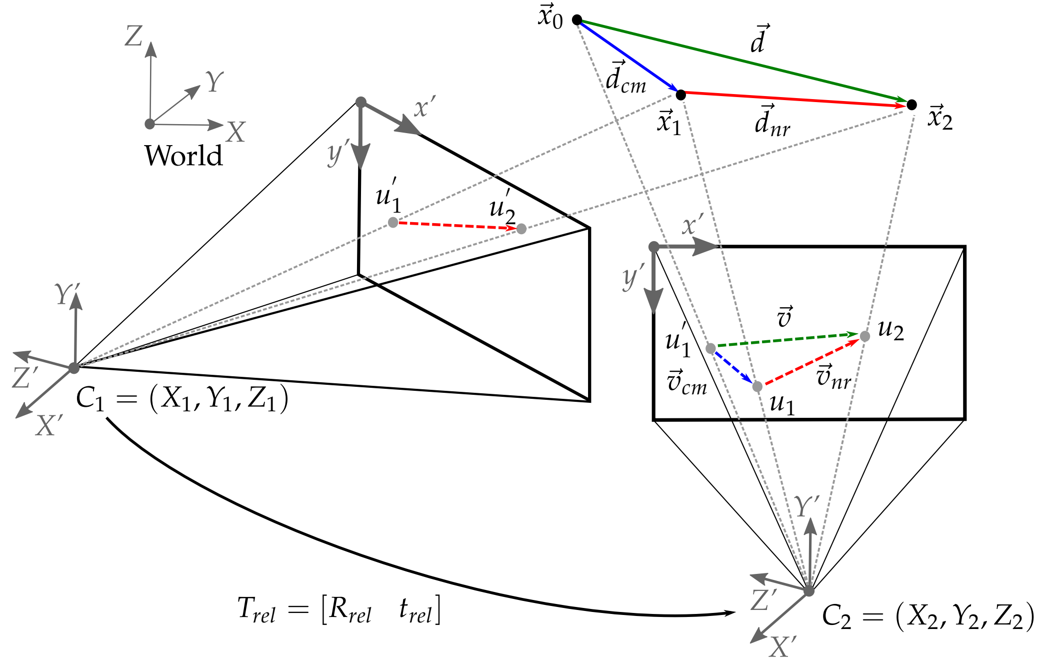


Figure 3.1: Full scene flow model and its relation to optical flow. The legend is the following. Dashed lines represent 2D optical flow vectors, solid lines represent 3D scene flow vectors. Ego-motion of both optical flow and scene flow are depicted in blue. Non-rigid residual optical flow and non-rigid residual scene flow are depicted in red. The total scene flow and optical flow are indicated in green. Projections of 3D points onto a 2D plane are notated as $u \in R^2$.

Figure 3.1 contains three 3D points in homogeneous coordinates $\vec{x}_0 \in \mathbb{R}^4$, $\vec{x}_1 \in \mathbb{R}^4$ and $\vec{x}_2 \in \mathbb{R}^4$, where $\vec{x}_0$ and $\vec{x}_2$ represent the same point cloud correspondence in a pair of frames. Point $\vec{x}_1$ represents where the original point $\vec{x}_0$ will be located after adding the ego-motion part to it.

To transform a 3D point from a coordinate system with a center C_1 into a coordinate system with a center C_2 we use a rigid transformation $T_{rel} = \begin{bmatrix} R_{rel} & t_{rel} \end{bmatrix} \in \mathbb{R}^{4\times4}$. Since we work with camera to world convention, the basic principle to derive the relative pose is to take an arbitrary point in the coordinate system of camera with a center C_1, then transom it into the world coordinates and finally transform from world into the coordinate system of the second camera. The formula for the relative rotation becomes:

$$R_{rel} = R_2^T R_1 \tag{3.2}$$

The realtive translation is computed as:

$$t_{rel} = R_2^T (t_1 - t_2) \tag{3.3}$$

By combining R_{rel} and t_{rel} we obtain a relative rigid transformation in homogeneous coordinates:

$$T_{rel} = \begin{bmatrix} R_{rel} & t_{rel} \\ 0^T & 1 \end{bmatrix} \tag{3.4}$$

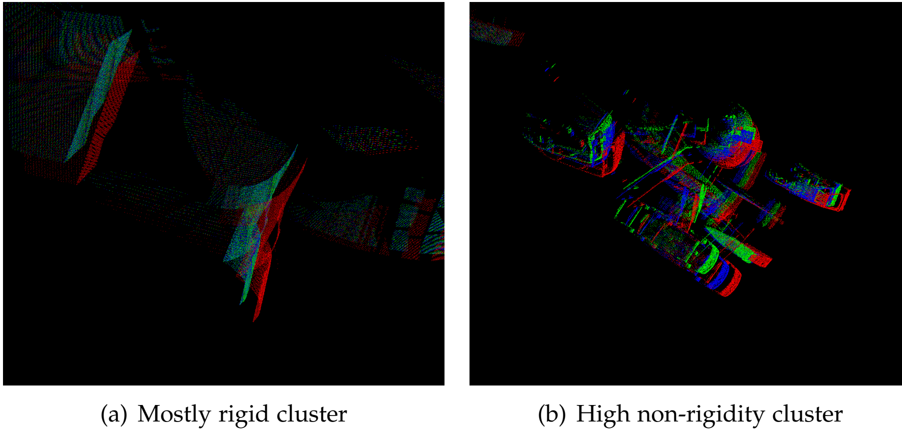
Now we can use the above relative transformation to model camera motion. For this we take the input from the first frame $\vec{x}_0$ transform it with T_{rel} and then subtract the original $\vec{x}_0$. The result of it will be a 3D ego-motion flow vector, which presented in homogeneous coordinates is $\vec{d}_{cm} \in \mathbb{R}^4$:

$$\vec{d}_{cm} = (T_{rel} - I_4)\, \vec{x}_t \tag{3.5}$$

If we are to apply $\vec{d}_{cm}$ to $\vec{x}_0$ we would get $\vec{x}_1$, which is essentially the same point with ego-motion included. The actual motion of an object is modelled by a non-rigid residual flow described as $\vec{d}_{nr} \in R^4$. Thus, we can construct the total scene flow vector using vector summation rule:

$$\vec{d} = \vec{d}_{nr} + \vec{d}_{cm} \tag{3.6}$$

3D point's total flow would have a different ratio of non-rigid and ego-motion flow depending on the level of rigidity. For points of static objects in a scene, non-rigid residual flow $\vec{d}_{nr}$ will be equal to $\vec{0}$ and the total flow would only be induced by ego-motion flow $\vec{d}_{cm}$. In the case of scenes with a static observer the total flow will be purely induced by $\vec{d}_{nr}$. Figure 3.2 summarizes our ideas in 3D.



(a) Mostly rigid cluster (b) High non-rigidity cluster

Figure 3.2: Visualization of the full scene flow model. Points in red are representing $\mathcal{X}_1$. Points in green represent $\mathcal{X}_2$. Blue points are $\mathcal{X}_1$ transformed with T_{rel}. (a) and (b) represent different clusters of the same scene. In (a) we can see that T_{rel} is able to align static points. In (b) T_{rel} brings $\mathcal{X}_1$ closer to $\mathcal{X}_2$ and rest of the motion is non-rigid and thus is modelled by per-point non-rigid flow $\vec{d}_{nr}$.

If we now consider 2D image planes of figure 3.1, there are 2D points $u \in \mathbb{R}^2$, which represent a projection of 3D points into an image plane. A pair of points $u_1^{'}$, $u_2^{'}$ and u_1, u_2 represent the projection of the same 3D point onto the corresponding image plane. For illustration purposes we also show a point from the left plane $u_1^{'}$ on the right plane.

On the right image plane we can find a vector $\vec{v} \in \mathbb{R}^2$, which represents optical flow. Optical flow can be viewed as a 2D analog of total scene flow. Concretely, to obtain optical flow $\vec{v}$ we can just project 3D total scene flow $\vec{d}$ onto the plane. Hence, networks which learn total scene flow can also be used for optical flow estimation if the projection matrix is known. Analogously to total scene flow we decompose optical flow using vector summation rule into:

$$\vec{v} = \vec{v}_{nr} + \vec{v}_{cm} \tag{3.7}$$

where $\vec{v}_{nr} \in \mathbb{R}^2$, $\vec{v}_{cm} \in \mathbb{R}^2$ represent the non-rigid and rigid components of the optical flow respectively. Similarly to the whole optical flow $\vec{v}$, we can obtain $\vec{v}_{nr}$ and $\vec{v}_{cm}$ by projecting 3D vectors $\vec{d}_{nr}$ and $\vec{d}_{cm}$ into an image plane.

3.2 Learning in 3D Space

It is possible to estimate total scene flow from 2D/2.5D data such as RGB, stereo or RGB-D. These kind of approaches generally follow the following steps. For plain RGB we need to learn the depth first. In case of stereo inputs, one has to learn disparity maps. Once we have disparities D, we can get the depth Z of each pixel by using baseline B and focal length f with a classical depth formula:

$$Z = f\frac{B}{D} \tag{3.8}$$

Once we have the depth the process for all 2D/2.5D formats is similar. Concretely, one takes a pixel u_1 projects it into 3D to obtain a point p_1 by using the standard projection formula:

$$p = (Z\frac{(x - c_x)}{f}, Z\frac{(y - c_y)}{f}, Z) \tag{3.9}$$

Then we have to learn optical flow in a separate network. Further, we warp a pixel u_1 with an estimate of optical flow which would give us a warped pixel u_2. To obtain a 3D point p_2 from u_2 we again use the formula 3.9. Once we have both points, the total scene flow $\vec{d}$ is just a difference between p_2 and p_1.

As it was noted in [16] these multi-step 2D/2.5D approaches generally do not outperform 3D approaches. The reason is that we have multiple sources

of error accumulation such as depth estimation, optical flow estimation and then projection into 3D. Another reason is that optimization in such case is indirect because the networks learn on 2D metrics, while scene flow evaluation metrics are all in 3D.

Therefore, most of the modern approaches learn scene flow directly on point clouds since they preserve more information about 3D geometry. Unlike 2D/2.5D methods, most of 3D methods take point clouds as inputs, encode them into a compact latent representation and finally decode it to get a 3D per point scene flow.

The computational advantage is that one can learn scene flow directly within one single network. This way we also avoid multi-step error accumulation, which we had in the case of 2D/2.5D. Additionally, training metrics are in 3D, which avoids indirect optimization. Furthermore, in some scenarios the data is only available in form of 3D point clouds, which cannot be processed by 2D/2.5D based methods.

3.3 Neural Networks for Point Clouds

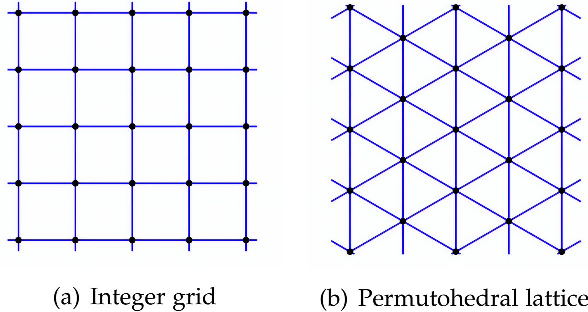
The most naive way to learn from point clouds is to learn with standard 2D/3D CNNs by adding a heuristic pre-processing step. In a pre-processing step we can voxelize the point clouds i.e. convert point clouds into voxels (e.g. [55]). An alternative way is to go from 3D into 2D. To convert point clouds into 2D images multi-view projections of a point clouds can be used (e.g. [47]). By doing either one of these two steps one essentially discretizes the input signal into a 2D/3D grid and thus standard 2D/3D convolution operations can be applied. However, in this case we lose natural geometric invariances from our learning signal when we convert them to grids. Furthermore, prepossessing step requires extra computation and using a dense grid is restrictive in terms of GPU memory.

Therefore, direct learning on point clouds requires a special type of neural architecture other than traditional 2D/3D CNNs. Point based networks offer a viable alternative to their voxel based counterparts. First attempts of point based networks were essentially point-based encoders (e.g. [39, 40, 30]). These type of networks encode each point or alternatively a neighbourhood of points individually using a shared multi-layer perceptron (MLP) and then form a global aggregated feature by using global pooling. The downside of these methods is that they do not utilize structural information of multiple points in a patch or they lose spatial relationship between points when performing the aggregation step during global pooling.

3.3.1 Permutohedral Lattice Networks

Another branch of architectures, which are able to process point clouds directly are inspired by high dimensional image filtering [1, 2] and are based on mathematical structures called permutohedral lattices as opposed to regular integer grids. These type of architectures called permutohedral lattice networks preserve structural information while also focusing on scalability and computational efficiency for sparse and high-dimensional signals.

While it's possible to use sparse convolutions to make learning on regular 3D integer grids more efficient by avoiding unnecessary computations, they are still inefficient in terms of scalability. Specifically, if input feature dimension d is to increase by for example adding RGB, normal and intensity information to spatial location, the number of neighbourhood vertices enclosing a input point will respectively increase exponentially $O(2^d)$ in d. On the other hand, if we are to use permutohderal lattice the number of enclosing vertices will grow linearly $O(d)$ in d. The scalability aspect of regular integer grid and permutohhedral lattice is illustrated in 3.3 with the help of Delaunay tessellations.



(a) Integer grid (b) Permutohedral lattice

Figure 3.3: Delaunay tessellation for two dimensional permutohedral lattice. (a) the number of vertices of a simplex is 2^d (b) the number of vertices of a simplex is $d+1$. Image from [5].

Permutohedral Lattice

Permutohedral lattice is the mathematical structure which all permutohedral lattice networks have in common. As proposed by Baek and Adams [5], permutohedral lattice A_d^* of dimension d is defined as:

$$A_d^* := \{T(\vec{x}) \mid \vec{x} \in (d+1)\mathbb{Z}^{d+1}\} \tag{3.10}$$

where $T(\vec{x})$ is the projection of a regular scaled grid $\mathbb{Z}^{d+1}$ onto the hyperplane H_d along vector $\vec{1}$ the diagonal of a unit cube, in which the sum of coordinates is 0.

Projection $T(\vec{x})$ is a map defined as:

$$T : \vec{x} \mapsto \vec{x} - \left(\frac{\vec{x} \cdot \vec{1}}{\|\vec{1}\|^2} \right) \vec{1} \tag{3.11}$$

And hyperplane H_d is:

$$H_d = \{ \vec{x} \mid \vec{x} \cdot \vec{1} = 0 \} \subset \mathbb{R}^{d+1} \tag{3.12}$$

Figure 3.4 illustrates the construction of A_d^* with $d = 1$. We can deduce from the figure that permutohedral lattice is a more sparce representation than regular grid because some points of the original grid are projected into the same point on the lattice.

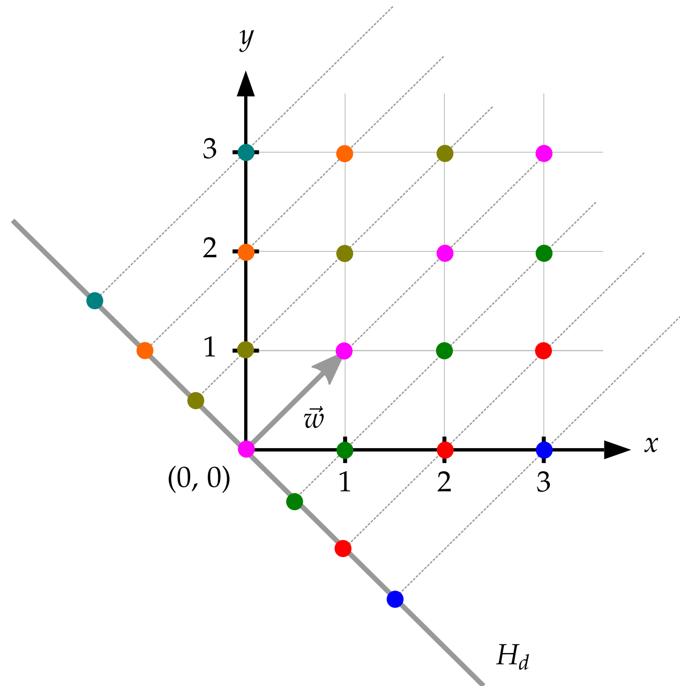


Figure 3.4: Constriction of a permutohedral lattice. We construct A_d^* with $d = 1$ by projecting the regular 2D integer grid of xy plane onto the hyperplane H_d along the vector $\vec{w} = (1, 1)$. The points projected onto the hyperplane form the lattice. The direction of projection is showed with light-gray dashed lines. Grid points of the same color are projected into the same point on a hyperplane.

Bilateral Convolutions

The basic building blocks of any lattice network are Bilateral Convolution Layers (BCL), which are essentially learnable extensions of high-dimensional

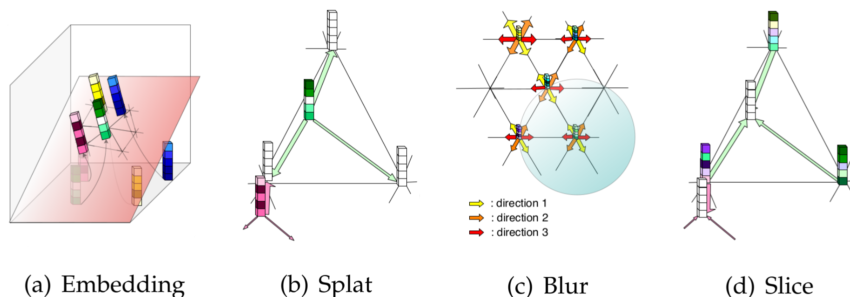
bilateral filters [22, 25]. In case of BCL each point of a point cloud is described by a position $p \in \mathbb{R}^d$ and has a signal $v \in \mathbb{R}^k$. As in [16] our signal is equal to the position vector $v = p$, however it possible for a signal vector to also contain colors, normals and intensities. The properties of a permutohedral lattice allow us to embed a position vector into a high-dimensional subspace H_d and then efficiently perform a Splat-Blur-Slice operation.

We start with embedding the position vector p into H_d which can be performed by basis change in $O(d)$ [1]. Then we splat a signal of a point onto lattice vertices. For this we need to find an enclosing simplex of a point. Since permutohedral lattices tessellate high-dimensional subspace H_d with uniform simplices we can efficiently find an enclosing simplex. It was formally proven [5] that each simplex of a lattice is a permutation and translation of $d + 1$ vertices in a canonical simplex which are identified by k-remainder. Due to this property, each lattice simplex is uniquely identified by the distance to closest zero remainder $\vec{l_0}$ and the ordering of $d + 1$ coordinates of a relative point position $\vec{x} - \vec{l_0}$.

Adams [1] proposed an efficient algorithm to find an enclosing simplex of a point in $O(d^2)$ time. Once the enclosing simplex is found we can splat the signal onto vertices using barycentric interpolation since all simpleces are uniform. The blur step is essentially a sparse version of traditional CNN convolution operation but preformed on a fixed neighbourhood of lattice points which gathered a signal. Finding the neighbours for blurring is deterministic and can be performed in $O(d^2)$ [1]. The number of neighbours for a blur step for a point with position p will be $(r + 1)^{d+1} - r^{d+1}$, where r is the radius of a convolution. We can change the receptive field of a convolution operation by scaling the entire lattice. Furthermore, scaling allows us to build a hierarchical representation of the input similar to CNNs, where we first capture coarse features in the beginning of the network and finer features in the later layers. The final step is slicing, which the inverse of splatting. The same barycentric coordinates from the splatting step are used to return the filtered signal to it's original spatial position. Embedding into H_d and a Splat-Blur-Slice operation are summarized in figure 3.5.

3.4 Self-Supervised Learning

Currently scene-flow estimation is mostly formulated as a supervised learning problem [16, 6]. To enable supervised learning we first need to obtain a dataset with scene flow ground-truth. In general to achieve that researchers take a stereo or RBG-D dataset and use optical flow to project that information into point clouds. Since optical flow captures correspondences between pixels the same correspondence also holds for points after projection into 3D. Taking a difference between two order set of point clouds allows us to



(a) Embedding (b) Splat (c) Blur (d) Slice

Figure 3.5: Illustration of position vector embedding and a Splat-Blur-Slice operation. In this illustration dimension $d = 2$ and the lattice is a subspace of $\mathbb{R}^3$. Stacked cubes represent the signal vector. (a) position vectors are embedded onto H_d, (b) scatter signals onto lattice vertices according to barycentric weights, (c) blurring with a learnable filter with radius $r = 1$, (d) gathering of filter signals into the original positions using barycentric weights computed in (a). Image from [23].

obtain the ground truth for the scene flow and learn the scene flow with supervised learning. On the other hand, self-supervised learning offers a viable alternative for training on unlabelled real world datasets.

The main idea of self-supervision is to extract the learning signal from using the correspondences in the data itself without relying on ground-truth. Utilizing the consistency of temporal, geometric, structural and photometric properties between frames of point clouds provides rich sources of self-supervision. We can construct loss constraints as a proxy for the supervised loss based on the before-mentioned properties of the data. It is important to balance such constraints with weights or other constraints to avoid degenerate or trivial solutions. Once the self-supervised proxy of the loss is constructed we can either train it in a full-supervision mode or add it to the supervised loss term. The former can be used it form of fine-tuning on unlabelled data after supervised training [37] or for training where there is no labelled data at all in full self-supervision. The later is used to improve supervised methods in form of implicit regularization [30]. In our work we mostly rely on temporal and geometric consistency of a point cloud sequence.

3.4.1 Forward-Backward Consistency

Forward-backward consistency is inspired by 2D computer vision where it was successfully applied to optical flow [29], tracking [49] etc. Forward-backward consistency utilizes the temporal continuity of a pair of neighbouring image frames or if extended into 3D of a point cloud pair.

The general idea of a forward-backward consistency constraint is to take a point cloud $\mathcal{X}_t$ in time step t and $\mathcal{X}_{t+\delta}$ in time step $t + \delta$ and predict the total

forward scene flow $\vec{d}_f$ for each point with a 3D deep neural network. Once we have the per-point forward scene flow we can apply them to points of $\mathcal{X}_t$ to get an estimate $\widehat{\mathcal{X}_{t+\delta}}$ to where the points from frame t will be translated into frame $t+\delta$. From there we can once again estimate the per-point total scene flows $\vec{d}_b$ with the same network but this time in a reverse direction going from points $\widehat{\mathcal{X}_{t+\delta}}$ to $\mathcal{X}_t$. Once we have the backward scene flows $\vec{d}_b$ for each point we can apply them to $\widehat{\mathcal{X}_{t+\delta}}$ and obtain $\widehat{\mathcal{X}_t}$, which is supposed to be close to the original starting point cloud $\mathcal{X}_t$. To measure the consistency standard per-point euclidean distance can be used to construct a loss constraint. We can now either use it as a proxy for the supervised loss or use it in a combination with the supervised loss. Figure 3.6 contains an illustration of the forward-backward consistency principle.

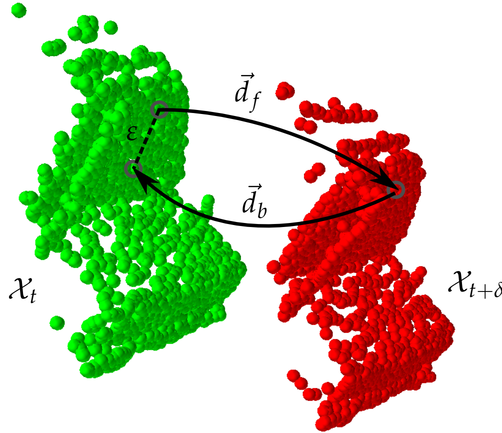


Figure 3.6: Demonstration of forward-backward consistency. $\mathcal{X}_t$ and $\mathcal{X}_{t+\delta}$ are point clouds in subsequent frames. Forward flow $\vec{d}_f$, which was predicted by the network is applied to a point in $\mathcal{X}_t$ get an expected position in $\mathcal{X}_{t+\delta}$. We then predict and apply the flow $\vec{d}_b$ backwards. The residual ε is the learning signal.

It is easy show that forward-backward property holds in the following experiment. If we apply the ground truth scene flow to some non-occluded point p in forward direction from time step t into $t+\delta$ and then apply the ground truth scene flow in reverse direction from $t+\delta$ back to t, the result will align with the original point p with zero error. Despite the fact that points in real point clouds are often occluded or not even represented in the neighbouring frame, the property still holds for our scene flow estimation formulation because we do not necessarily assume point to point correspondence between time steps.

3.4.2 Cycle Consistency

Forward-backward consistency is a special case of cycle consistency, where the size of a cycle $K = 1$. The motivation of cycle consistency is to generalize to arbitrary cycle size K in order to harvest learning signal from long sequences.

In order to obtain the learning signal we first predict total scene flow in forward direction and apply them to current point cloud at time step t, at time step $t+1$ we have a transformed point cloud, from there we repeat the same procedure. We keep propagating the prediction and the warp procedure for the next steps $t_0, ..., t_K$. Overall, we would need to perform K network predictions. To propagate the signal in the reverse direction, we start from the corresponding transformed point cloud at steps $t_1, ...t_k$. Then from each of these K time steps we perform backward flow prediction and then apply it to the transformed point clouds iteratively until we reach the beginning of a cycle at step t_0. The amount of network predictions for the reverse direction would be $\frac{K(K+1)}{2}$. Therefore, the total amount of network predictions for cycle-consistency is $\frac{K(K+3)}{2}$, which grows in $O(k^2)$. The principle of cycle consistency is illustrated in figure 3.7. The learning signal ε is the sum of residuals from each of K cycles $\varepsilon = \sum_{i=1}^{k} \varepsilon_k$.

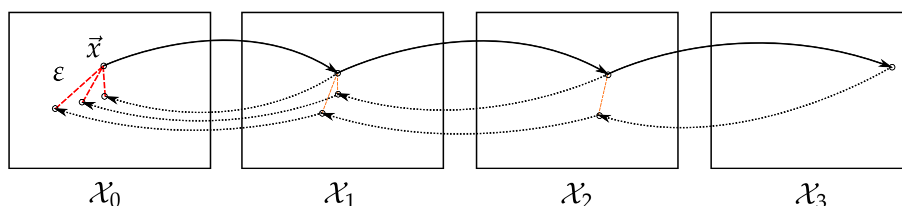


Figure 3.7: Demonstration of cycle consistency. In this case cycle size $K = 3$. We start from point $\vec{d}$. Solid lines indicate forward flow predicted by the network. On the other hand, dashed arrows represent the revers flow predicted by the network. Orange displacements show the intermediate residuals. The learning signal is the sum of K residuals shown in red.

Furthermore, skipping frames in a sequence is advantageous for capturing different magnitudes of the scene flow. One way to implement this strategy is to to do augmentation where the frames are dropped in the middle of the sequence. Alternatively, skip connections can be applied to circumvent time steps in both forward and backward direction. The downside of cycle consistency is that it can be restrictive in terms of speed of computation and GPU memory, especially if used with large networks.

Chapter 4

Method

This chapter introduces our method for solving the problem presented in chapter 1. Section 4.1 starts the chapter with a high-level overview of the entire pipeline starting from the inputs, intermediate steps going up to the output. In section 4.2 we go into details of our implementation of forward backward consistency. Section 4.3 goes into details of our neural network architecture. Finally, in section 4.4 we introduce the loss functions used to train our models.

4.1 Overview

As it was defined in chapter 1 our main goal is given a pair of point clouds $\mathcal{X}_t \in \mathbb{R}^{n \times d}$ and $\mathcal{X}_{t+\delta} \in \mathbb{R}^{m \times d}$ to estimate total scene flow $\vec{d} \in \mathbb{R}^3$ for each n points of $\mathcal{X}_t$. Additionally our goals are to estimate n per-point non-rigid flows $\vec{d}_{nr} \in \mathbb{R}^3$ and ego-motion in form of a rigid relative camera pose transformation $T_{rel} \in \mathbb{R}^{4 \times 4}$.

Figure 4.1 provides a high level overview of the proposed pipeline. A pair of point clouds $\mathcal{X}_t$ and $\mathcal{X}_{t+\delta}$ are fed to the encoder. The encoder consists of two identical Siamese branches, which share the weights. Periodically, the information from each Siamese branch is fused into an unified representation. The main task of an encoder is to learn a low dimensional latent representation vector to describe 3D motion.

Afterwards the network branches in two separate components, namely into a pose regressor network and into a flow decoder. Pose regressor's task it to learn a relative camera pose between a pair of point clouds. Then the relative pose is combined with the first input of the pipeline $\mathcal{X}_t$ to obtain an ego-motion motion flow for each point of the point cloud $\mathcal{X}_t$. Flow decoder's task is to up-sample the latent vector into a per-point non-rigid flow. Once

the non-rigid flows and ego-motion flows are ready we can combine the two to obtain total scene flow vectors for each point.

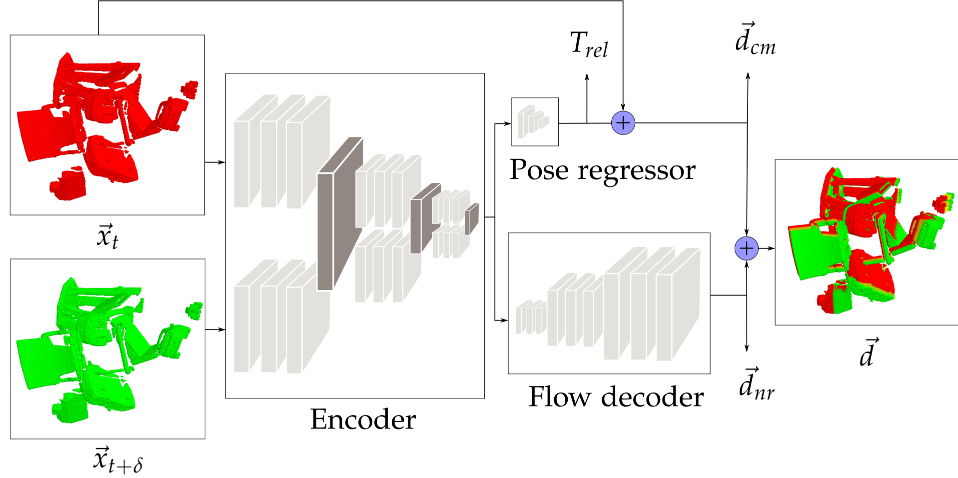


Figure 4.1: High-level overview of the learning pipeline. Points of a pair of point clouds $\vec{x}_t$ $\vec{x}_{t+\delta}$ depicted in red and green respectively goes through an encoder and splits up into a pose regressor and a decoder. The final output is total scene flow $\vec{d}$, which is depicted as yellow vectors between a pair of point clouds. Intermediate outputs are relative camera pose T_{rel}, non-rigid scene flow $\vec{d}_{nr}$ and ego-motion scene flow $\vec{d}_{cm}$. The functions which combine different components to output the corresponding scene flow components are depicted in purple.

4.2 Self-Supervisory Signals

Our self-supervisory signals are implemented based on forward-backward consistency, the theory of which was described in section 3.4. A naive algorithm to implement forward-backward consistency is to take two tensors $\mathcal{X}_1$ and $\mathcal{X}_2$ and to predict a tensor of total scene flows D with the same size as $\mathcal{X}_1$. Then D and $\mathcal{X}_1$ are summed to get a tensor of warped points $\widetilde{\mathcal{X}_2} = \mathcal{X}_1 + D$. Afterwards, we again predict a tensor of flows, but this time in reverse direction $\widetilde{D}$ from $\widetilde{\mathcal{X}_2}$ to $\mathcal{X}_1$. Our learning signal in this case are Euclidean distances of points in the starting point cloud $\mathcal{X}_1$ and the alignment at the end of a cycle $\widetilde{\mathcal{X}_2} + \widetilde{D}$.

However, in practice we found that it's hard to optimize a loss function based on the naive algorithm. The overall loss function containing the naive-self supervision term was failing to converge and qualitative evaluation revealed that the magnitude of total scene flow vectors was close to 0. Therefore, we came to the conclusion that the main cause of failing optimization are degenerate solutions, since the forward-backward consistency loss is at

its minimum when the network predicts the flow of $\vec{0}$. One measure to alleviate the issue was to introduce linear weight scaling, where loss weight of self-supervision w was set to 0 at the beginning of training and then linearly increased later. Nevertheless, the loss function was still failing to converge once we added the naive constraint.

The same issue was also encountered in a concurrent work by Mittal et. al [37]. We adopt their proposed strategy with the nearest neighbour anchoring to tackle degenerate solutions by balancing the loss function with an additional constraint allowing it to converge. Instead of just predicting a tensor of reverse flows from warped points $\widetilde{\mathcal{X}_2}$, on the other hand the idea of nearest neighbour anchoring is to predict reverse flows $\widetilde{D}$ from $\mathcal{X}_2^*$:

$$\mathcal{X}_2^* = \frac{(\widetilde{\mathcal{X}_2} + \mathcal{X}_{nn})}{2} \tag{4.1}$$

where $\mathcal{X}_{nn}$ is a tensor of nearest neighbours for each point of $\widetilde{\mathcal{X}_2}$ to points inside $\mathcal{X}_2$.

Furthermore, we balance an anchored version of a forward-backward consistency constraint to prevent degenerate solutions by posing an additional nearest neighbour constraint proposed in [37, 13]. The nearest neighbour constraints aims at minimizing the Euclidean distance between the points of a warped point cloud $\widetilde{\mathcal{X}_2}$ and their corresponding nearest neighbours $\mathcal{X}_{nn}$ inside a point cloud $\mathcal{X}_2$.

4.2.1 Nearest Neighbour Search

We need to implement an algorithm for finding the nearest neighbour for each point of a warped point cloud $\widetilde{\mathcal{X}_2}$ to points inside a target point cloud $\mathcal{X}_2$, in order to implement an anchored version of a forward backward-consistency constraint along with a nearest neighbour constraint. Since most of our computation is performed on a GPU our algorithm should be easily parallelizable.

When it comes to nearest neighbour search on the CPU many machine learning libraries rely on k-d trees, the run time complexity of which is $O(n \log n)$. However, making efficient parallel computation with k-d trees on GPUs is non trivial due to that fact that nearest neighbour search algorithm based on k-d trees is naturally sequential.

In contrast to k-d trees, our nearest neighbour search approach based on building a similarity matrix for two sets of points due to the following reasons. While the asymptotic run-time of similarity matrix based algorithm is $O(n^2)$, it is trivial to vectorize to enable efficient parallelization on both a GPU and a CPU. Furthermore, it additionally saves us data transfer

of outputs from the GPU to the CPU when computing the nearest neighbours. A similarity matrix between two sets of point clouds $\mathcal{X}_1 \in \mathbb{R}^{m \times d}$ and $\mathcal{X}_2 \in \mathbb{R}^{n \times d}$ is a matrix $\Sigma \in \mathbb{R}^{m \times n}$ which contains $m \times n$ Euclidean distances $\|x_1 - x_2\|_2^2$ between all points from the first set $x_1 \in \mathbb{R}^d$ to all the points in the second set $x_2 \in \mathbb{R}^d$. Once the similarly matrix Σ is computed, we just need to find a minimum in n column for each of m rows to get the indices which indicate the nearest neighbours.

Finally, if further speed up of execution is desirable, algorithms based on parallelization of k-d trees [38] can be adapted to our problem. Alternatively Approximate Nearest Neighbor (ANN) search algorithms can also be utilized to speed up the computation [51, 59]. We leave the exploration of algorithmic aspects of nearest neighbour search for future work.

4.3 Model's Architecture

Our model contains three main components, which are an encoder, pose regressor and a decoder. An encoder and decoder part of our model are based on HPLFlowNet proposed by Gu et. al [16]. HPLFlowNet is a state of the art architecture based on permutohedral lattice for estimating scene flow. Our relative pose regression network is inspired by [8, 31, 34, 24, 10]. The overall number of trainable parameters for the modified architecture is 23,645,609. Inference on a pair of point cloud with 8192 points takes approximately a third of a second. We describe the individual components of the pipeline and their relation to each other in the following subsections.

4.3.1 Encoder

An encoder of HPLFlowNet consist of a pair of Siamese branches, which share the weights. An encoder starts with 1D convolutions, which gradually increase the number of channels of input point clouds keeping the number of points the same. Each branch builds a representation by embedding and splatting of point clouds onto a permutohedral lattice. The representations are gradually down sampled by using a Splat-Blur layer (DownBCL) based on the steps of Splat-Blur-Slice operation, which was described in section 3.3. The splatting step is using barycentric interpolation to place the signal vectors onto simplex vertices. Once the signals are in place a standard convolution operation is applied to a lattice neighborhood of size $f = (r+1)^{d+1} - r^{d+1}$, where r is the number of hops from the lattice vertex, d is a dimension of a permutohedral lattice, which is implemented by using a standard 2D convolution known from CNN. Convolution's filter size is $f \times 1$ and it is applied on the tensor of point neighborhoods of size $f \times n$, where n is the number of points, which gathered a signal after splatting. Once the blurring is performed, the same procedure is repeated again but

on a scaled down lattice, which gradually reduces the number of splatted points after each Splat-Blur layer.

The information from each branch is fused together periodically by using a correlation layer (CorrBCL). A correlation layer takes a point neighbourhood with size $g = (r+1)^{d+1} - r^{d+1}$ from one branch and aligns it with a neighborhood of the same size g in the other branch, where r is the number of hops from the lattice point and d is the dimension of a permutohedral lattice. Afterwards, each vertex inside a pair of aligned neighbourhoods becomes a center of an offset neighbourhood with a size $h = (r+1)^{d+1} - r^{d+1}$. A pair of patches from each branch of a total size $g \times h \times n$, where n is the chuck size common for both branch, are concatenated together across channels into one tensor, on which a blur operation is performed. The blur operation is implemented in two stages with a 3D convolution followed by a 2D convolution. The filter size of a 3D convolution is $1 \times h \times 1$ and for 2D convolution it is $g \times 1$. After the correlation layer the information of two branches is fused into one representation and passed down to the subsequent correlation layers.

4.3.2 Relative Pose Regressor

HPLFlowNet's authors did not assume any prior on rigidity for the output of an encoder, we on the other hand use the same encoder to learn a joint latent representation for both non-rigid and rigid parts of the motion. As it was described in section 3.1, our rigid part of flow is modelled by a relative camera pose transformation T_{rel}. To regress the pose we take the output of an encoder with the shape $64 \times n$ and feed it into a pose regression network. At the beginning of the network the outputs are aggregated over n points. We selected global average pooling for our aggregation operation, since it demonstrated better experimentation results compared to global maximum pooling and spatial average/maximum pyramid pooling. Afterwards we connect a vector of 64 values to a fully connected layer (FC1) with 2048 neurons and then to a second fully connected layer (FC2) with the same number of neurons. Both FC1 and FC2 used ReLU as an activation function and have a dropout rate $p = 0.5$. We learn the relative pose jointly as a 6 dimensional vector, where the rotation part is modelled with 3 Euler angles and translation contains 3 XYZ coordinates. Hence, the final regression layer contains 6 neurons with linear activation functions, since we require real valued outputs. We did not encounter any issues due to Gimbal lock caused by using Euler angles, since the relative camera rotation is very small for most of the datasets. Furthermore, we also experimented with learning the rotation part in form of quaternions but Euler angles gave better stability during optimization and slightly better results overall. A possible explanation for this behaviour could be that quaternion q and $-q$ represent the same rota-

tion [53, 43]. We have also tried splitting up a pose regresssor network into two separate branches for rotation and translation, however we experimentally found that it gives worse performance, since often either rotation or translation was ignored buy the network and did not converge. Moreover, a slightly better performance was achieved when the values for the translation and rotation parts of the loss were kept in similar ranges by balancing with a scalar.

4.3.3 Decoder

An encoder part of the network is aimed at up-sampling of a latent representation into per-point non-rigid flows. The representation is increased in dimensions by utilizing Blur-Slice layer (UpBCL), which is essentially a reverse operation for DownBCL. After blurring a neighbourhood of size $f = (r+1)^{d+1} - r^{d+1}$ by using a 2D convolution with a filter size $f \times 1$, the signals are being distributed to the original position by using the same barycentric weights, which were computed for the corresponding DownBCL layer. The scales of the lattice are also the same as during the down-sampling stage but come in reverse order, which allows us to go from lattices with coarse simplices to finer lattices while increasing the number of points. Inputs to UpBCLs consist of a concatenated representations of symmetrical DownBCL and the results of a previous UpBCL or from a correlation layer.

Similarly to an encoder, the authors of the HPLFlowNet architecture do not reason about rigidity of the flow in the decoder. On the other hand, we propose to decode the non-rigid part of the scene flow instead of decoding the total scene flow. After obtaining an estimate of a camera pose from the pose regression network we can compute the ego motion flow by combining the relative camera pose and $\mathcal{X}_1$ using the equation 3.6. Finally, the total flow is obtained by summing up the the non-rigid flow predicted by a decoder and an ego-motion flow build with the help of a pose regression network.

4.4 Loss Functions

Overall, all of our loss functions are based on the Euclidean distance. Most of our models used a total loss function, which consisted of terms to penalize different part of the motion or to constrain our optimization problem. Please note that for the convenience of loss presentation, we will omit the vector symbol notation for different components of the flow in contrast to chapter 3.

Our main supervised loss measures an average of the Euclidean distance over n points between the ground truth total scene flow vector $d \in \mathbb{R}^3$ and

the estimate of total scene flow vector $d \in \mathbb{R}^3$ and is defined as:

$$\mathcal{L}_{epe3d} = \|d - \hat{d}\|_2 \tag{4.2}$$

Our self-supervised signals are formed by the forward-backward consistency loss $\mathcal{L}_{fb}$ and the nearest neighbour loss $\mathcal{L}_{nn}$:

$$\mathcal{L}_{ss} = w_{fb}\mathcal{L}_{fb} + w_{nn}\mathcal{L}_{nn} \tag{4.3}$$

where scalars w_{fb}, w_{nn} are used to weight the self-supervised terms. Additionally, we have experimented with modelling w_{fb} as a piecewise linear function $w_{fb}(x)$:

$$w_{fb}(x) = \begin{cases} 0 & x < 21 \\ 0.1x & 21 \leq x < 41 \\ 2 & x \geq 41 \end{cases}$$

where argument x represents epoch number. The motivation behind it is to alleviate degenerate solutions, when the network predicts $\vec{0}$ for scene flow in the earlier epochs. However, we found in practice that this heuristic has low influence, if we are to complement $\mathcal{L}_{fb}$ with $\mathcal{L}_{nn}$. Nevertheless, this rule could be useful in a weakly supervised setting, where we still have the labels but they might be inaccurate and thus more weight is put on to the self-supervised parts of the loss.

Let $x_1 \in \mathbb{R}^3$ be the point inside a point cloud $\mathcal{X}_1 \in \mathbb{R}^{n\times 3}$ and $x_2 \in \mathbb{R}^3$ be the point belonging to the second point cloud $\mathcal{X}_2 \in \mathbb{R}^{n\times 3}$. As described in section 4.2, we define the forward-backward consistency loss as the Euclidean distance between the input point x_1 and the alignment to the original point $\widetilde{x_1}$. The final formula for forward-backward consistency, which we average over all the n points of $\mathcal{X}_1$ is:

$$\mathcal{L}_{fb} = \|x_1 - \widetilde{x_1}\|_2 \tag{4.4}$$

The alignment $\widetilde{x_1} \in \mathbb{R}^3$ to the original point at the end of the cycle is defined as warping of an anchor point x_2^* with the predicted reverse flow $\widetilde{d} \in \mathbb{R}^3$ from the anchor to the original position:

$$\widetilde{x_1} = x_2^* + \widetilde{d} \tag{4.5}$$

An anchor point $x_2^* \in \mathbb{R}^3$ is defined as the average between $\widetilde{x_2} \in \mathbb{R}^3$, which is the original point x_1 warped with forward flow d and its nearest neighbour $x_{nn} \in \mathbb{R}^3$:

$$\widetilde{x_2} = x_1 + d \qquad x_2^* = \frac{\widetilde{x_2} + x_{nn}}{2} \tag{4.6}$$

The second self-supervised loss term is the Euclidean distance between the warped point $\widetilde{x_2}$ and its nearest neighbour x_{nn}, which we also average over n points:

$$\mathcal{L}_{nn} = \|\widetilde{x_2} - x_{nn}\|_2 \tag{4.7}$$

Up to this points the losses we defined can be used to train both categories of models, the ones where we learn the total scene flow holistically and the one where we learn learn the non rigid and rigid parts jointly. The total loss to train the models without scene flow decomposition is the combination of 4.2 and 4.3:

$$\mathcal{L} = w_{epe3d}\mathcal{L}_{epe3d} + \mathcal{L}_{ss} \tag{4.8}$$

Where w_{epe3d} is the weight to control the degree of supervision in the total loss. We set $w_{epe3d} = 0$ for the experiments with full self-supervision. We need to extend the total loss from the above if we are to train the model, which jointly learns the non-rigid flow and ego-motion. Since non-rigid flow is modelled as a displacement vector, we can also adapt a similar loss as in equation 4.2 to estimate the deviation of the estimate of non-rigid flow vector $\widehat{d_{nr}} \in \mathbb{R}^3$ and the ground truth non-rigid flow vector $d_{nr} \in \mathbb{R}^3$. Hence, the loss for the non-rigid part of the flow, which we average over n points, is defined as:

$$\mathcal{L}_{nr} = \|d_{nr} - \widehat{d_{nr}}\|_2 \tag{4.9}$$

When it comes to the relative camera pose loss we have two separate losses for translation and rotation, which define the relative camera pose loss:

$$\mathcal{L}_r = w_{rot}\mathcal{L}_{rot} + \mathcal{L}_t \tag{4.10}$$

Similarly to [31, 34, 24] we balance the magnitudes of a rotation and a translation with the weight $w_{rot} = 10$. The translation loss $\mathcal{L}_t$ is the Euclidean distance between the ground truth relative translation $t \in \mathbb{R}^3$ and the prediction $\hat{t} \in \mathbb{R}^3$:

$$\mathcal{L}_t = \|t - \hat{t}\|_2 \tag{4.11}$$

We used Euler's angles to model the rotation. Therefore, the loss for rotation is:

$$\mathcal{L}_{rot} = \|r - \hat{r}\|_2 \tag{4.12}$$

Similarly to [31, 24] we decided not to enforce any constraint to normalization of rotation in order not to complicate the optimization problem with one additional constraint. Furthermore, we confirmed in experimentation that the prediction rotations become close to the ground truth even without normalization.

Finally, we can combine the loss in 4.8 with the rigid loss in 4.10 and non-rigid loss in 4.9 to formulate the total loss for the case of decomposition of total flow into rigid and non-rigid parts:

$$\mathcal{L} = w_{epe3d}\mathcal{L}_{epe3d} + w_{nr}\mathcal{L}_{nr} + w_r\mathcal{L}_r + \mathcal{L}_{ss} \tag{4.13}$$

where weights w_{epe3d}, w_{nr}, w_r are used to balance different components of the flow and which we all set to zero in case of full self-supervision.

Please not that in case of flow decomposition we used the formulas 3.5, 3.6 we defined in the motion model to assemble the total scene flow from the non-rigid and rigid parts:

$$\hat{d} = \widehat{d_{nr}} + (\hat{R} - I_3)x_1 + \hat{t} \tag{4.14}$$

where $\hat{R}$ is the rotation matrix obtrained from the estimate of Euler angles $\hat{r}$. Once we have an estimate of total flow $\hat{d}$ we cam compute the lose for total scene flow $\mathcal{L}_{epe3d}$ as in 4.2.

Chapter 5

Experiments

5.1 Datasets

This section gives an overview of datasets used in further experiments. Table 5.1 contains the statistics of all datasets used in this work.

Similarly to [16, 33], we use a subset of *FlyingThings3D* where extremely hard samples were omitted. The ground-truth for camera pose for each frame can be obtained from the full dataset. Additionally *RefRESH* Rendering Toolkit [31] was used to render a dataset with all the necessary ground-truth. The rendered dataset has a large number of observations and the ground truth for camera poses, which makes it useful in both self-supervision and scene flow decomposition experiments. We can exclude *KITTI Scene Flow 2015* [35, 36] as a candidate for training based on the low number of images and only used it for evaluation to test model's ability to generalize on real world data from a different domain.

Dataset	Observations	Camera pose	Optical flow	Realism
FlyingThings3D	23402	+	+	Synthetic
RefRESH	44027	+	+	Semi-synthetic
KITTI	200	-	+	Realistic

Table 5.1: Overview of scene flow datasets.

5.1.1 Input Data and Ground Truth

Point Cloud Generation

As we mentioned in section 3.2, point cloud representations $\mathcal{X} \in \mathbb{R}^{N\times3}$ consisting of N points $p \in \mathbb{R}^3$ of the scene are obtained from the intrinsic cam-

era matrix, optical flow and the corresponding depth values. Obtaining the depth value varies on the dataset. It is either taken directly from depth maps (*RefRESH*) or calculated using the projection formula in case of stereo data (*FlyingThings3D, KITT*):

$$Z = f\frac{B}{D} \tag{5.1}$$

where D stands for disparity (taken from a disparity map), f is camera's focal length, B is the baseline, c_x and c_y is the principal point (all taken from the camera's extrinsic matrix).

Hence, the final point cloud p for all datasets we used in our experiments is computed as:

$$p = (Z\frac{(x - c_x)}{f}, Z\frac{(y - c_y)}{f}, Z) \tag{5.2}$$

Relative Camera Pose

Since the ground truth for the camera poses of both FlyingThings3D and RefRESH uses the convention camera to world, we can use the following equations from section 3.1 to construct the ground truth for relative camera pose:

$$R_{rel} = R_2^T R_1 \qquad t_{rel} = R_2^T(t_1 - t_2) \tag{5.3}$$

By combining R_{rel} and t_{rel} we obtain the relative pose:

$$T_{rel} = \begin{bmatrix} R_{rel} & t_{rel} \\ 0^T & 1 \end{bmatrix} \tag{5.4}$$

5.1.2 FlyingThings3D

Since the subset dataset of FlyingThings3D does not have any camera pose ground truth, we match the frames of the subset dataset with the frames in the full dataset. However, in the process we discovered that some frames even in the original data-sets are missing the ground-truth for camera pose. Hence, we remove 64 observations, which are missing the corresponding camera poses from the experiments which involve camera pose estimation. In this case the size of the dataset becomes 19586 observations for training and 3816 observations for validation. In all experiments, where we do not learn the camera pose, the original subset is used with 19640 observations for training and 3824 observations for validation. In practice we found that removing missing camera pose observations does not affect model's generalization ability because the distribution of both dataset is the still the same. Similarly to [16] we use one quarter of the validation set for validation and the entire validation set for testing.

5.1.3 RefRESH

We have decided to render our own dataset using RefRESH rendering toolkit [31], since the authors provided dataset did not contain the bidirectional optical flow needed for occlusion map estimation. We limited ourselves to keyframe rate $k = 1$. Similarly to [31] we use scenes *apt0, apt1, apt2, copyroom, office0, office1, office2* for training and scene *office3* for validation (one quarter) and testing (entire scene).

Optical Flow Occlusion Maps

Occlusion maps are a crucial component for creating an appropriate scene flow ground truth. Points at time step t which will become occluded at time step $t + \delta$ need to be removed from the point clouds. Otherwise, incorrespondence between points in two frame will provide invalid scene flow, which makes learning impossible for the network.

Unlike *FlyingThings3D*, *RefRESH* toolkit does not provide occlusions masks explicitily. Therefore, following [31, 44, 9] the algorithm based on forward-backward consistency property of the optical flow was used.

Specifically, point x at step t is considered occluded in $t + \delta$ if the difference between the forward flow f_f and backward flow f_b mapped into the coordinates of $x + f_f$ by interpolation is larger than some threshold $T = 0.1$ and not occluded otherwise.

Post Processing

Because the aforementioned algorithm is an approximation of the true occlusion maps further post processing needed to be applied. Following [31] we use mathematical morphology with window size 10 to erode the outliers on the borders. However, this procedure still misses a lot of outliers on the edges. To cope with this, we additionally apply the forward-backward consistency algorithm in the opposite direction, which removes more points on the edges. Finally, to eliminate the remaining outliers we remove the pair of points whose flow magnitude is larger than $\mu_{mag} + 2\sigma_{mag}$, where μ_{mag} is the mean of euclidean distances between a pair of corresponding points and σ_{mag} are their standard deviation. The dataset was scaled by 10 to match *FlyingThings3D* in the number of splatted lattice points.

5.1.4 KITTI

Due to low number of observations in the scene flow dataset, we only use the dataset for evaluation purposes. *KITTI* is a good candidate, which allows us to test model's ability to generalize on real-world data from an unseen

domain acquired directly from a LiDAR scanner. Following [30, 16] we use 142 observations of KITTI omitting some incomplete observations.

5.2 Evaluation Metrics

When it comes to estimating the total scene flow and non-rigid flow in 3D we rely on metrics as in [16, 37, 54]. We also evaluate 2D optical flow similar to [16]. Overall, 2D and 3D metrics employ the Euclidean Distance to measure the deviation from the ground truth vector.

To evaluate relative camera pose we use metrics similar to [34, 24, 43]. For estimating rotations we use Relative Orientation Error (ROE) sometimes referred as Geodesic Distance [14]. We derive ROE by exploiting the property of the trace for the rotation matrices and the distance between two rotations [17]. The minimum angle α to align an estimated $\hat{R}$ and ground truth rotation R is:

$$\alpha = \arccos\left(\frac{trace(\hat{R}R^T) - 1}{2}\right)\frac{180}{\pi} \tag{5.5}$$

For evaluating relative camera pose translation we use the Euclidean Distance, which we measures the deviation in meters of the predicted camera pose location and ground truth vector.

Full list of evaluation metrics we are using in our qualitative studies are:

- **EPE3D** (m). Mean of Euclidean Distances between the predicted and ground truth pair of 3D vectors over all points.
- **Acc3D(0.05)**. Strict notion of accuracy. Percentage of points with an EPE3D $<$ 0.05m or relative error $<$ 5%.
- **Acc3D(0.1)**. Relaxed notion of accuracy. Percentage of points with EPE3D $<$ 0.1m or relative error $<$ 10%.
- **Outliers3D**. Percentage of points with EPE3D $>$ 0.3m or relative error $>$ 10%.
- **EPE2D** (px). Mean of Euclidean Distances between the scene flow and the ground truth projected into 2D over all points.
- **Acc2D**. Percentage of points with EPE2D $<$ 3px or relative error $<$ 5%.
- **RLE** (m). Mean of Euclidean Distances between the relative translation of the camera pose.
- **ROE** (degrees). Mean of minimum rotation angles needed for aligning an estimated and ground truth rotations.

5.3 Training Setup

This section describes experimental setup common to all of the experiments. We used batch size $B = 1$, dimensions of input point clouds $\mathcal{X}_1$ and $\mathcal{X}_2$ were set to $\mathbb{R}^{8192 \times 3}$. The initial learning rate for *FlyingThings3D* datasets were set to 10^{-4} and scaled by $s = 0.7$ after 85 epochs and then after every 35 epochs. Identical decay procedure of the learning rate was also used while training on *RefRESH* but with the initial learning rate of 10^{-5}, because we found through the experimentation that the loss was not converging in the beginning of training with a larger learning rate. All models trained in the experiments section are optimized using the Adam optimizer. All models were trained on GPUs of bare metal servers provided by ISG, Leonhard cluster and Microsoft Azure.

5.4 Quantitative Study

Our quantitative study consists of two experiments groups, which follow the same structure. In each group, we first describe the goal of an experiment and its setup. Afterwards we present the experiment results in form of an internal comparison. Finally, we select the best models from two groups and compare them with other methods in subsection 5.4.3.

5.4.1 Self-Supervision in Total Scene Flow Learning

This experiment group investigates the effects of self-supervision on a model in different configurations, which learn total scene flow directly and do not assume any decomposition of scene flow into a rigid and a non-rigid part. We make a comparison of all experiments and select the best one for evaluation with alternative methods. The loss function for optimization for experiments in this group is defined in equation 4.8.

Supervised Baseline

The goal of this experiment is to obtain the supervised baseline we can compare our self-supervised models to. The supervised baseline is HPLFlowNet [16] trained for 500 epochs. We observed that training on *FlyingThings3D* allowed our model generalize better to other domains and real-world data compared to training on other datasets such as *RefRESH*. Better generalization of models trained on *FlyingThings3D* could be accredited to low degree of noise in the point clouds, which a result of accurate raw input data such as optical-flow, disparities and occlusion masks, which we used to build input point clouds. Large number of observations is also a factor which contributes to better generalization. Therefore we chose training on *FlyingThings3D* for the baseline. We evaluate on all 3 datasets.

Self-Supervisory Signals

The purpose of this experiment is to show that adding self-supervisory signals in form of additional loss terms to main supervised loss lets us outperform the baseline. Specifically, we achieved an approximately 13% decrease in EPE3D. An explanation for better for better performance compared to the supervised baseline are self-supervision terms, which act regularizers when training the model. Due to time constraints, we train the model on *FlyingThings3D* for a total of 400 epochs. However, even after 400 epochs our model outperforms the baseline trained for 500 epochs and training until full convergence will demonstrate the same effect. Evaluation is performed on all three datasets.

Full Self-Supervision

In this experiment we demonstrate that our model can learn in fully self-supervised mode from scratch. This could be useful for training the network on datasets where the ground-truth of scene flow is not available at all. We set the weight for supervised loss term to zero to simulated the situation where there is no ground-truth i.e. ground-truth is not used for optimization in training and for model selection in validation. In fully-supervised case we have to rely on $\mathcal{L}_{FB} + \mathcal{L}_{NN}$ term which acts as a proxy of $\mathcal{L}_{EPE3D}$ in order to select the best model during validation after each epoch. We trained the model on *FlyingThings3D* for 100 epochs and evaluated on all 3 datasets. Our method is able to outperform some other self-supervised classical methods such as ICP [7] and perform close to some of the supervised learning methods. Although our method did not outperform state of the art supervised learning methods it can be used in situations where there is no ground-truth at all where fair qualitative results are sufficient.

Fine Tuning with Self-Supervision

The goal of this experiment is to pre-train a model on a dataset with ground-truth in a supervised way with self-supervisory signals and then fine tune that model on unseen data which doesn't have any ground-truth. We initialize the network trained in a supervised way with self-supervisory signals for 300 epochs on FlyingThings3D. Further, we fine-tune the weights on RefRESH for another 40 epochs in a fully self-supervised way. We test the model on all three datasets to investigate the effects of fine tuning.

Summary

To investigate the convergence we present figure 5.1, which demonstrates the loss curves for both training and validation on FlyingThings3D. Fine tuning on RefRESH was omitted from the plots because of different x-axis

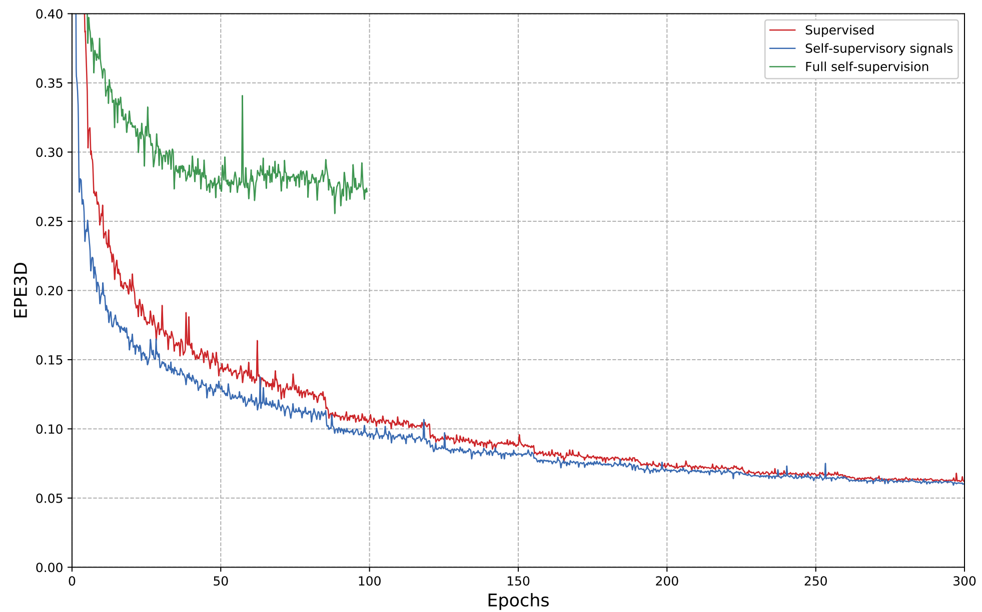
and different magnitudes of loss. We only present quantitative evaluation in this case. Due to very lengthy training times, not all models could be trained to their full convergence. Nevertheless, we can distinctly see that the self-supervisory signals model consistently outperforms the baseline by a margin and that the same trend is to hold if we are to train further. Similarly, we see that the fully-supervised model demonstrates higher loss values than the supervised model and the supervised model with self-supervisory signals. From the convergence plots we can conclude that the best configuration is the supervised model with self-supervisory signals.

Table 5.2 shows evaluation results from the conducted experiment group for self-supervision investigation. We evaluate on all three datasets.

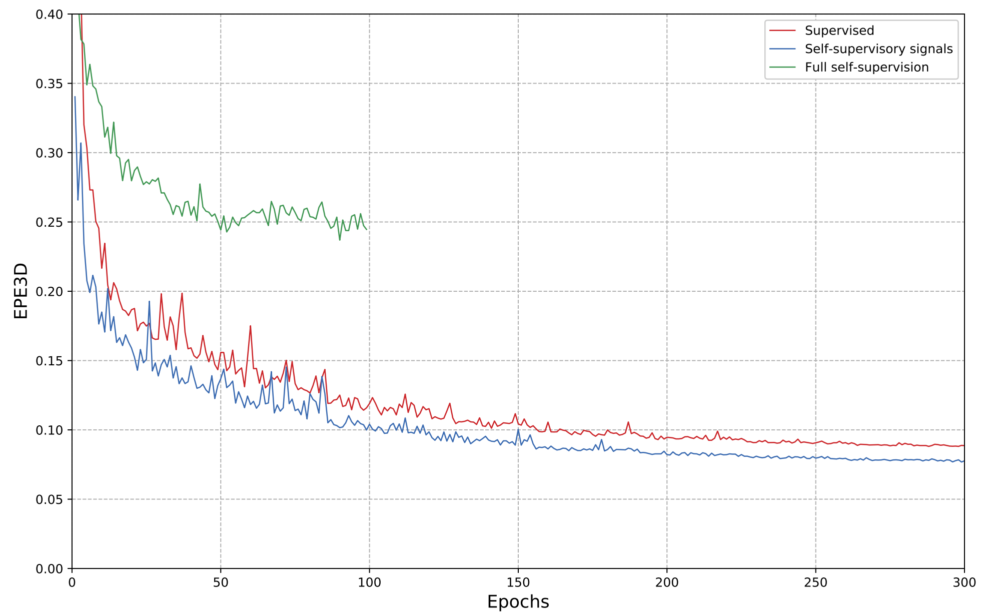
Dataset	Configuration	EPE3D	Acc3D(0.05)	Acc3D(0.1)	Outliers3D	EPE2D	Acc2D
FlyingThings3D	Supervised	0.0853	0.5863	0.8399	0.4562	4.9491	0.6586
	Self-supervisory signals	**0.0753**	**0.6293**	**0.8749**	**0.4199**	**4.3328**	**0.6914**
	Full Self-supervision	0.2534	0.1323	0.3691	0.9099	12.5009	0.2229
	Fine tuning	0.1184	0.3979	0.6998	0.6458	6.5231	0.5249
RefRESH	Supervised	0.1826	0.2045	0.4829	0.9271	8.3610	0.2519
	Self-supervisory signals	0.1795	0.2143	0.4932	0.9209	8.3073	0.2500
	Full self-supervision	0.2111	0.1219	0.3587	0.9694	11.2949	0.1477
	Fine tuning	**0.1556**	**0.3038**	**0.5812**	**0.8715**	**7.0802**	**0.3496**
KITTI	Supervised	0.1409	0.4022	0.7054	0.4880	5.6305	0.5374
	Self-supervisory signals	**0.1172**	**0.4699**	**0.7720**	**0.4164**	**4.9085**	**0.5950**
	Full self-supervision	0.4641	0.1863	0.3353	0.8417	16.4514	0.2665
	Fine tuning	0.2309	0.2631	0.4837	0.7136	8.6720	0.4173

Table 5.2: Evaluation of total scene flow in self-supervision experiments on FlyingThings3D, RefRESH and KITTI. The best results of a combination of a metric and a dataset are shown in bold.

From the table we can conclude that introducing self-supervisory signals into the supervised model has consistently improved the performance compared to the supevised configuration on all datasets for the vast majority of the metrics. We can also conclude the effectiveness of fully self-supervised fine-tuning on RefRESH because it has improved the performance on all metrics for the same dataset. Notably, the cost of increased performance for pretraining on one dataset and fine-tuning on a different dataset carried a decrease in a generalization ability for other datasets. A possible explanation for this behaviour could be different data distributions of RefRESH and the other two datasets, which are more similar in data distribution. Furthermore, we confirmed that fully self-supervised learning demonstrated worse results than the configurations, which contained the supervised part. Longer training time is expected to improve the performance of a self-supervised model, however it could it be restrictive in terms of training times.



(a) Training



(b) Validation

Figure 5.1: Convergence of loss functions in self-supervision experiments.. The loss function for both training (a) and validation (b) is EPE3D. The experiments are trained and validated on FlyingThings3D. For the purpose of visualization we present the region of 300 epochs on the x-axis and the size of one epoch is 4910. The supervised baseline was trained for 500 epochs. Combined training with a supervised loss and self-supervisory signals was trained for 400 epochs. A fully self-supervised model was trained for 100 epochs.

5.4.2 Non-Rigid Flow and Ego-Motion

This experiment group studies the effect of self-supervision on a model, which is learning a non-rigid residual scene flow and relative camera pose jointly. We perform an internal comparison and select the best configuration for the comparison with other methods. The loss function for optimization for experiments in this group is defined in equation 4.13.

Joint Supervised Learning

In this experiment we obtain the supervised baseline for joint learning of non-rigid and rigid parts of the flow. We train the model on the subset of *FlyingThings3D* with available camera pose data for a total of 500 epochs. The rotation part of the camera pose loss is balances with the corresponding weight $w_{rot} = 10$ to keep the translation and rotation errors in the same range. We evaluate the model on all three datasets for 3D scene flow and 2D optical flow metrics. To evaluate camera pose metrics we only use *FlyingThings3D* and *RefRESH*.

Self-Supervisory Signals

So far we only investigated how self-supervision and decomposition of total flow into rigid and non-rigid parts could be beneficial individually. The goal of this experiment is to combine self-supervisory signals with joint learning of non-rigid and rigid parts of the flow. We trained the model on FlyingThings3D for 230 epochs. The evaluation is performed on all three datasets.

Full Self-Supervision

The purpose of the experiment is to study the effect of full self-supervision, similarly as in the experiment group without scene flow decomposition. In this experiment we learn non-rigid flow and ego-motion jointly without any ground-truth from scratch. We trained the model on FlyingThings3D for a total of 70 epochs and evaluate it on all three datasets.

Summary

Figure 5.2 presents the convergence of EPE3D for scene flow decomposition experiments. From the convergence we can deduce that the best configuration is supervised training with self-supervisory signals as it was in the experiments with no decomposition of total flow into non-rigid and rigid parts. Note that not all models could be trained until full convergence due to time constraints of lengthy training. Nevertheless, we can confirm once again that introduction of self-supervisory signals to the supervised part improves the validation performance as it was previously the case in the experiments with no scene flow decomposition. In constant to models with

no scene flow decomposition, a fully supervised model demonstrated better results and even performed on par with the supervised version in case of scene flow decomposition.

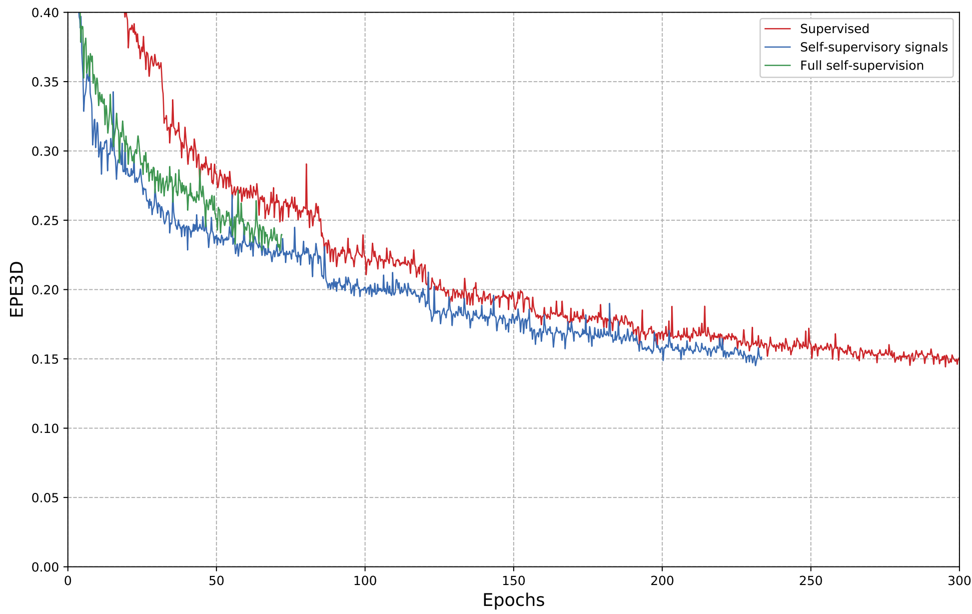
Quantitative evaluation for joint learning of non-rigid flow and ego-motion are evaluated separately as total scene flow, non-rigid flow and relative camera pose. We start by presenting the qualitative evaluation of total scene flow in table 5.3. These metrics are similar and comparable to the ones from the total flow learning experiments in table 5.2.

Dataset	Configuration	EPE3D	Acc3D(0.05)	Acc3D(0.1)	Outliers3D	EPE2D	Acc2D
FlyingThings3D	Supervised	0.1333	**0.3062**	0.6500	**0.7274**	7.6906	**0.4703**
	Self-supervisory signals	**0.1232**	0.2769	**0.6722**	0.7388	**6.8230**	0.4588
	Full self-supervision	0.2150	0.1759	0.4538	0.8735	10.7544	0.2991
RefRESH	Supervised	0.2421	0.0330	0.1921	0.9871	6.1629	0.3078
	Self-supervisory signals	**0.2007**	0.0764	0.3375	0.9724	**4.9052**	**0.4326**
	Full self-supervision	0.2009	**0.1297**	**0.3901**	**0.9640**	5.5907	0.3868
KITTI	Supervised	0.3252	0.0522	0.2461	0.8764	12.2543	0.1980
	Self-supervisory signals	**0.2493**	0.0896	**0.3553**	**0.8001**	**9.1295**	0.2553
	Full self-supervision	0.4488	**0.1888**	0.3510	0.8283	15.8564	**0.2825**

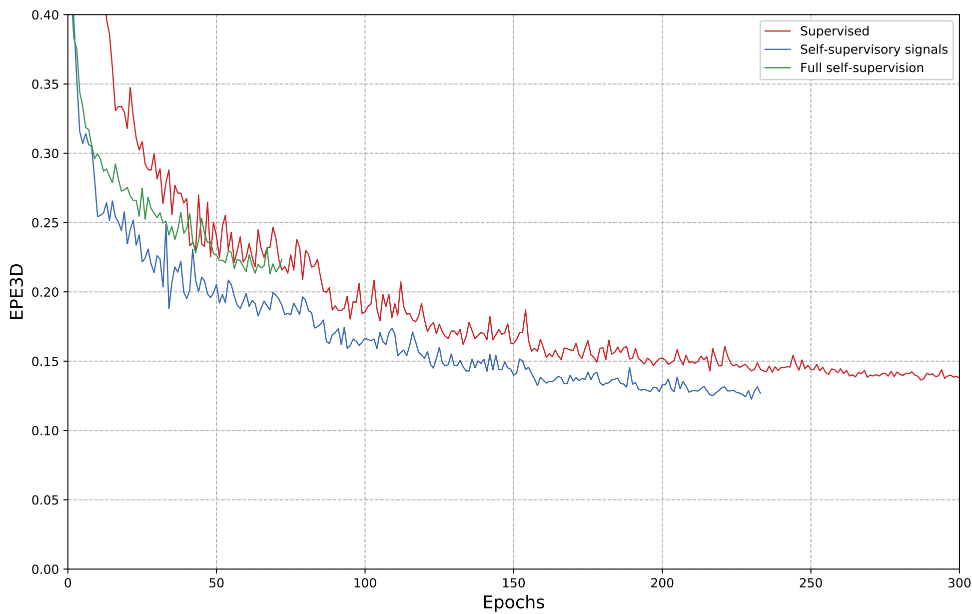
Table 5.3: Evaluation of total flow on FlyingThings3D and RefRESH and KITTI. Total flow evaluated in this table is obtained from jointly learned non-rigid flow and ego-motion. We evaluated our models on the entire subset of FlyingThings3D regardless of availability of camera poses. The best results of a combination of a metric and a dataset are shown in bold.

From the table we can conclude that self-supervisory signals improved the supervised version for most of the metrics. In some cases such as Acc3D(0.05), Outliers3D on FlyingThings3D, supervised training with self-supervisory signals performed slightly worse than purely supervised version. This is explained by the fact that supervised training with self-supervisory signals configuration was trained for a significantly shorter period of time. We expect this configuration to outperform the supervised one with longer training, similarly as it was the case with no scene flow decomposition.

Notably, a fully self-supervised model performed significantly better after introducing scene flow decomposition even at 70 epochs compared to 100 epoch in the version with no scene flow decomposition. Furthermore, a fully self-supervised prototype showed better generalization ability on RefRESH performing on par with the models, which had some degree of supervision in the loss function. This could be accredited to the rigid part of flow in the total loss function, which models the motion between the point cloud in form of a rigid transformation. Rigid transformation of the entire point cloud along with nearest neighbour search implicitly resemble the ICP algorithm.



(a) Training



(b) Validation

Figure 5.2: Convergence of loss functions in non-rigid flow and ego-motion experiments.. The loss function for both training (a) and validation (b) is EPE3D. The experiments are trained and validated on a subset of FlyingThings3D with available camera pose data. For the purpose of visualization we present the region of 300 epochs on the x-axis and the size of one epoch is 4897. The supervised baseline was trained for 500 epochs. Combined training with a supervised loss and self-supervisory signals was trained for 230 epochs. Fully self-supervised model was trained for 70 epochs.

Overall, the performance for models, which contained the supervised loss term in the total loss, performed slightly below the level of the same models without scene flow decomposition. Nevertheless, scene flow decomposition could still be beneficial because we obtain more information about the motion in one single forward pass. In case of self-supervision scene flow decomposition demonstrated better results contrasting the supervised configurations. To evaluate scene flow decomposition beyond the total flow we propose to consider the same metrics but for the non-rigid flow along with the relative camera pose evaluation.

Tables 5.4 presents the evaluation for non-rigid part of the flow on the datasets, which had the ground truth of camera poses.

Dataset	Configuration	$EPE3D_{nr}$	$Acc3D(0.05)_{nr}$	$Acc3D(0.1)_{nr}$	$Outliers3D_{nr}$	$EPE2D_{nr}$	$Acc2D_{nr}$
FlyingThings3D	Supervised	0.1210	**0.5904**	0.6815	0.9483	8.6211	**0.6042**
	Self-supervisory signals	**0.1172**	0.5782	**0.6871**	**0.9469**	**8.2311**	0.5979
	Full self-supervision	0.5000	0.0210	0.0982	0.9843	25.2101	0.0666
RefRESH	Supervised	0.2423	**0.1823**	0.3407	0.9995	**3.7163**	**0.7575**
	Self-supervisory signals	**0.2369**	0.1671	**0.3417**	**0.9995**	4.4170	0.6759
	Full self-supervision	0.3704	0.0174	0.0929	0.9998	12.5493	0.1169

Table 5.4: Evaluation of non-rigid flow on FlyingThings3D, RefRESH Similarly to total flow, the same 3D and 2D evaluation metrics are also applicable to non-rigid component of the flow and are indicated with an underscript (nr). The best results of a combination of a metric and a dataset are shown in bold.

Overall, we can see that self-supervisory signals have once again improved the supervised model. Full self-supervision on the hand, demonstrated worse results compared to the models with some degree of supervision. A possible explanation could be that our self-supervised losses are more suited for rigid motion prediction.

Finally, we conclude the quantitative evaluation with the results of relative camera pose estimation on FlyingThings3D and RefRESH, which are presented in table 5.5.

From the table we can conclude that a supervised model and a supervised model with self-supervisory signals performed similarly in terms of ROE and RLE. This could be explained by the fact that later was trained for a shorter period of time. We expect to get better performance of self-supervisory signals model, if we are to increase the training time. Fully self-supervised model performed similarly to the supervised counterparts on RefRESH for ROE and even outperformed them for RLE. A possible explanation is a relatively high ratio of rigid to non-rigid motion and low magnitudes of rotations in RefRESH as opposed to FlyingThings3D.

Dataset	Configuration	ROE	RLE
FlyingThings3D	Supervised	**0.5793**	**0.1837**
	Self-supervisory signals	0.6209	0.1862
	Full self-supervision	2.4788	0.3661
RefRESH	Supervised	**1.1107**	0.4047
	Self-supervisory signals	1.1111	0.3349
	Full self-supervision	1.1896	**0.1781**

Table 5.5: Evaluation of relative camera pose estimation on FlyingThings3D and RefRESH. Camera poses for KITTI are not available. The best results of a combination of a metric and a dataset are shown in bold.

5.4.3 Comparison to Related Methods

In previous experiments we investigated approaches and configurations of our method relative to each other in an internal evaluation. In contrast, this section aims to compare our method to alternative scene flow evaluation methods from the related work mentioned in chapter 2. Our main parameter for comparison is total scene flow and its 2D counter part optical flow since most of the scene flow methods do not decompose the total flow. Table 5.6 shows the comparison with a variety of supervised, hybrid and self-supervised scene flow estimation methods.

Dataset	Method	Sup.	EPE3D	Acc3D(0.05)	Acc3D(0.1)	Outliers3D	EPE2D	Acc2D
FlyingThings3D	ICP [7]	Self.	0.4062	0.1614	0.3038	0.8796	23.2280	0.2913
	Ours	**Self.**	**0.2149**	**0.1750**	**0.4531**	**0.8736**	**10.7544**	**0.2981**
	FlowNet3D [30]	Hyb.	0.1136	0.4125	0.7706	0.6016	5.9740	0.5692
	SPLATFlowNet [46]	Full.	0.1205	0.4197	0.7180	0.6187	6.9759	0.5512
	Original BCL [16]	Full.	0.1111	0.4279	0.7551	0.6054	6.3027	0.5669
	HPLFlowNet [16]	Full.	0.0804	0.6144	0.8555	0.4287	4.6723	0.6764
	Ours	**Hyb.**	**0.0753**	**0.6293**	**0.8749**	**0.4199**	**4.3328**	**0.6914**
KITTI	ICP [7]	Self.	0.5181	0.0669	0.1667	0.8712	27.6752	0.1056
	Ours	**Self.**	**0.4488**	**0.1888**	**0.3510**	**0.8283**	**15.8564**	**0.2825**
	FlowNet3D [30]	Hyb.	0.1767	0.3738	0.6677	0.5271	7.2141	0.5093
	SPLATFlowNet [46]	Full.	0.1988	0.2174	0.5391	0.6575	8.2306	0.4189
	Original BCL [16]	Full.	0.1729	0.2516	0.6011	0.6215	7.3476	0.4411
	HPLFlowNet [16]	Full.	**0.1169**	**0.4783**	**0.7776**	**0.4103**	**4.8055**	0.5938
	Ours	**Hyb.**	0.1172	0.4699	0.7720	0.4164	4.9085	**0.5950**

Table 5.6: Quantitative comparison to other methods evaluated on FlyingThings3D and KITTI. Column *Sup.* describes model's degree of supervision, where *Self.* stands for self-supervised, *Full.* means fully-supervised and *Hyb.* means hybrid training i.e. supervised with self-supervisory signals. The best results of a combination of a metric and adataset are shown in bold. Evaluation data is adapted from [16].

From the table we can conclude that the best model in the supervised cat-

egory for FlyingThings3D is a our supervised model with self-supervisory signals without scene flow decomposition. Our supervised model with self-supervisory signals trained for 400 epochs was able to outperform the supervised HPLFlowNet trained by the authors for 600 epochs. Concretely, our main metric EPE3D decreased by more than 6%. The secondary metrics Acc3D(0.05), Acc3D(0.1), Outliers3D improved by more than 2% each. When it comes to 2D metrics, EPE2D decreased by almost 8% and Acc2D improved by more than 2%. On the other hand, it can be seen that the performance of our model on KITTI is similar to the one of supervised HPLFlowNet. To improve the performance on that particular dataset fine-tinning can be used.

We expect the gap to increase further, if we are to continue to train our model further and a more accurate comparison would be to compare both models at 400 epochs. For example, at the mark of 400 epochs the supervised HPLFlowNet demonstrated EPE3D equal to 0.0866 on FlyingThings3D and 0.1434 on KITTI. If we are to compare it to our model, the improvement in EPE3D for FlyingThings3D is 13% and for KITTI it is 18%.

When it comes to self-supervised category a clear winner is a fully supervised model, which relies on scene flow decomposition. Notably, scene flow decomposition was less useful in the supervised case than in the self-supervised one. We anticipate better performance with longer training times for this prototype.

5.5 Qualitative Study

This section presents qualitative results for some selected models, which we previously investigated quantitatively in the previous sections. The visualizations of scene flow predictions on FlyingThings3D are presented in figure 5.3, evaluations on RefRESH are contained in figure 5.4 and finally evaluation on KITTI is in figure 5.5. In qualitative evaluation we aim to compare two group of experiments in terms of visual quality of scene flow prediction. The group is the one with no assumption on rigidity of the motion and the second group learns total scene flow by learning non-rigid and rigid parts of the flow jointly. For each of the two groups we decided to present the visualizations for the supervised training with self-supervisory signals because it consistently outperformed the purely supervised approach on most of the metrics. Furthermore, we visualize our fully self-supervised models since qualitative evaluation could offer us insights on closing the gap between the supervised models.

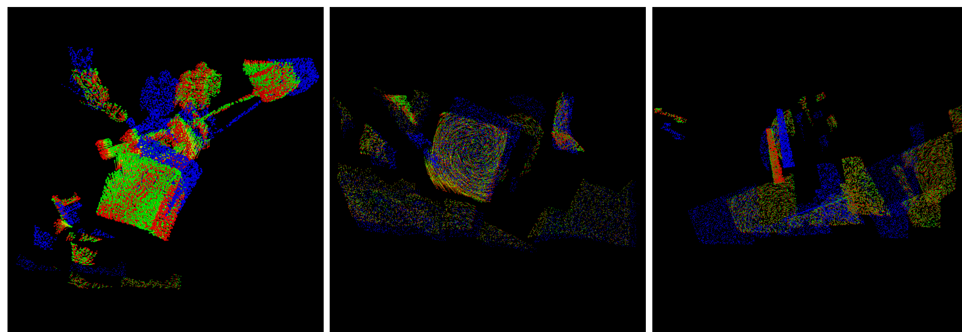
Visualizations have the following convention: an input point cloud $\mathcal{X}_1$ is shown in blue, the points of an input point cloud $\mathcal{X}_1$ warped with the corresponding total scene flow estimates are shown in green, an input point cloud $\mathcal{X}_2$ is shown in red and yellow residuals indicate the error of total

scene flow estimation. We set the number of points $N = 8192$ for both point clouds.

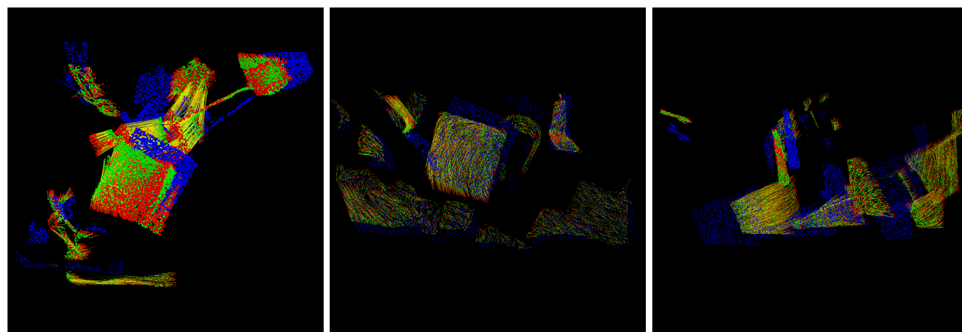
From figure 5.3 we can conclude that the models, which had a supervised component along with self-supervisory signals in their total losses performed the best in terms of qualitative results on FlyingThings3D. The best among them was the one without total scene flow decomposition because the one with scene flow decomposition demonstrated noise on small clusters. Longer training could potentially help to tackle this issues. When it comes to fully self-supervised models, the model with scene flow decomposition has demonstrated better qualitative results than the model without a decomposition. One can see that the overall magnitude of scene flow errors was lower in case of joint learning of non-rigid flow and ego-motion, although it has a few outliers possibly induced by the non-rigid part of the decomposition. Overall, we can conclude that fully self-supervised methods perform on par with their supervised counterparts on simpler scene with distinct objects and edges but fail to give more accurate results on incomplete or cluttered parts of the scene.

Figure 5.4 presents qualitative evaluation on a dataset from a different domain the models were trained on. We can see tha the dataset has a clear distinction of the rigid and non-rigid parts and also some noise in the ground truth despite post-processing. We can see that all models performed similarly in terms of qualitative results. Similarly to FlyingThings3D, the supervised learning model with self-supervisory signals demonstrated the best performance and had the least magnitude of error in the predictions. Overall, self-supervised models performed similarly to supervised models with self-supervisory signals but have demonstrated larger magnitude of errors. We can concluded again that scene flow decomposition introduced a slight improvement in qualitative results in case of self-supervision.

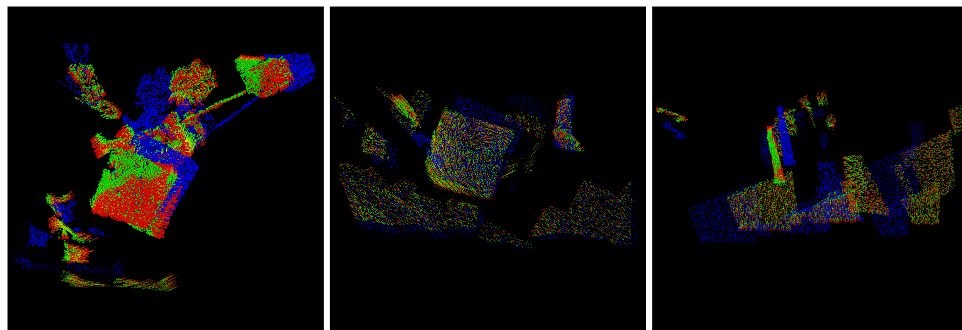
Finally, we conclude our qualitative evaluation with predictions on a real world dataset captured with a LiDAR device in figure 5.5. From the visualizations we can see that the overall magnitude of the flow is small as it was the case with RefRESH. Hence, the self-supervised models performed similarly to supervised models with self-supervisory signals. Nevertheless, self-supervised models have struggled with some incomplete shapes and have an overall larger magnitude of error. Similarly to RefRESH, scene flow decomposition contributed to slightly better qualitative results in case of self-supervision.



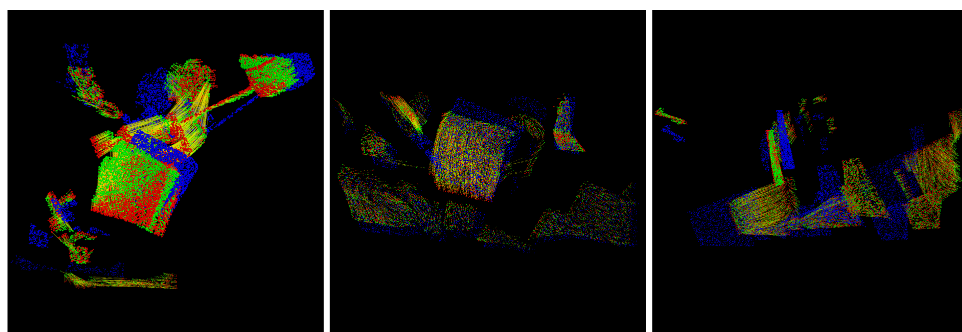
(a) Supervised learning with self-supervisory signals



(b) Full self-supervision

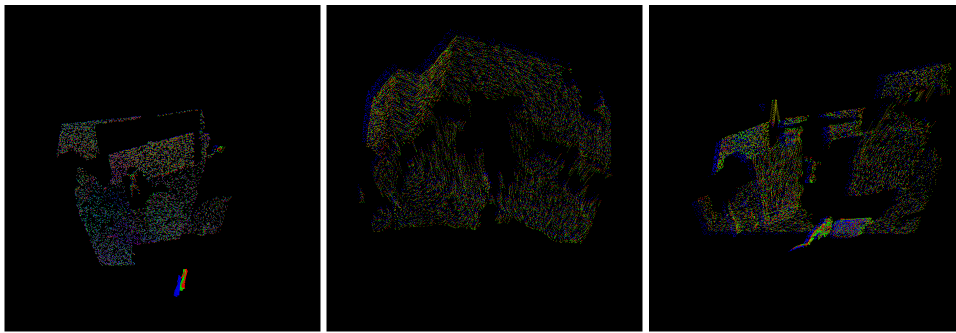


(c) Supervised joint learning of non-rigid flow and ego-motion with self-supervisory signals

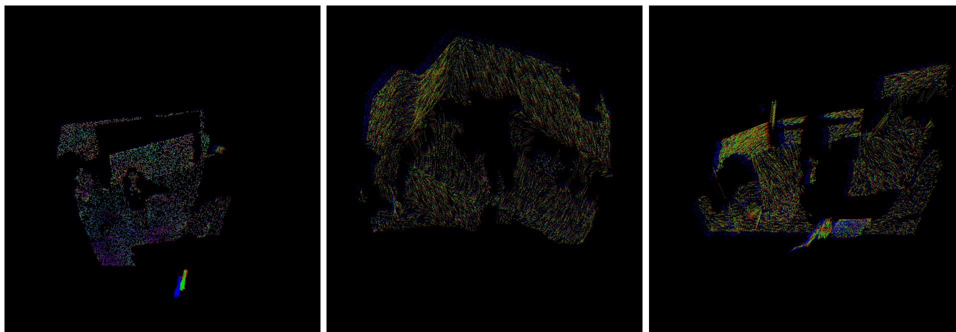


(d) Fully self-supervised joint learning of non-rigid flow and ego-motion

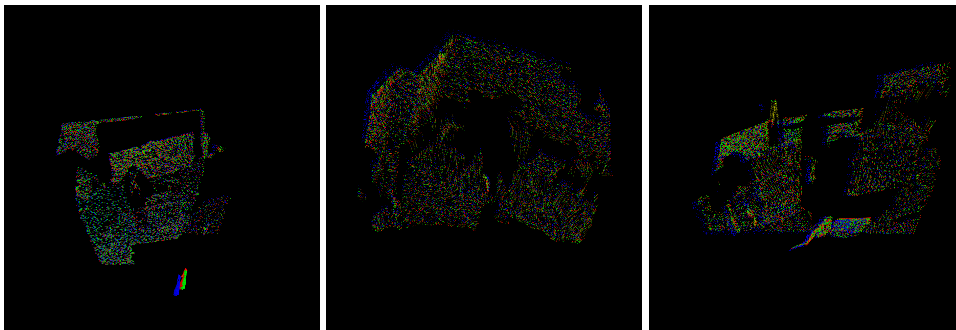
Figure 5.3: Qualitative evaluation on FlyingThings3D. The first column contains a challenging scene with large degree of cluttering and incomplete shapes, on which (b) and (d) have performed worse than (a) and (c). On the other hand, (b) and (d) were able to perform close to (a) and (c) on scenes with distinct edges and shapes.



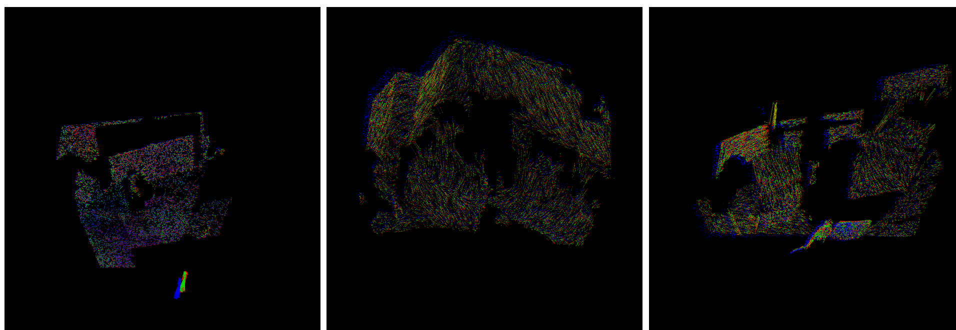
(a) Supervised learning with self-supervisory signals



(b) Full self-supervision

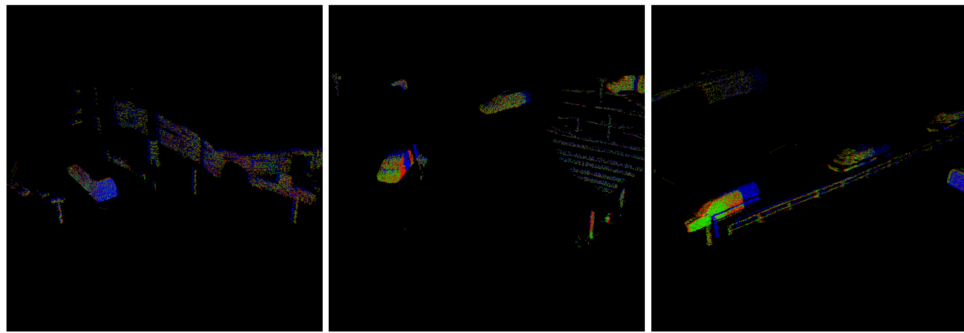


(c) Supervised joint learning of non-rigid flow and ego-motion with self-supervisory signals

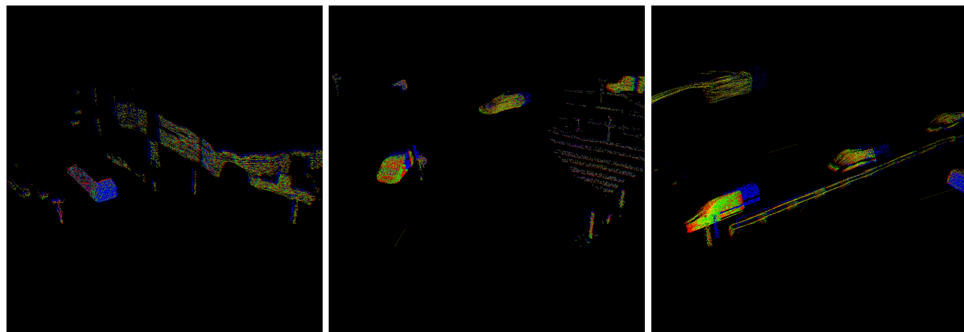


(d) Fully self-supervised joint learning of non-rigid flow and ego-motion

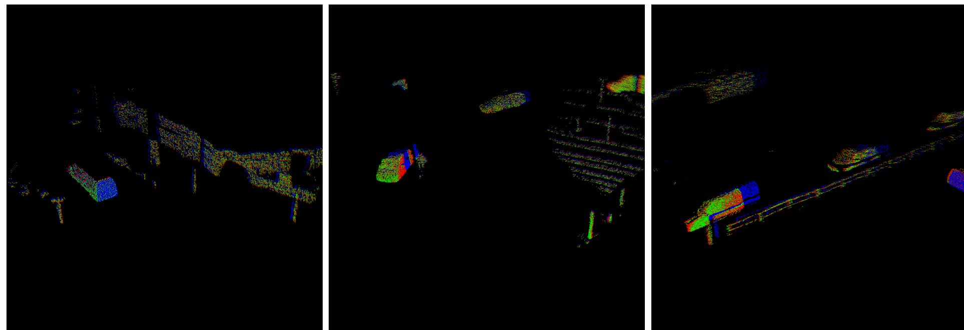
Figure 5.4: Qualitative evaluation on RefRESH. The first two scene have a clear separation of rigid and non-rigid parts and thus the performance for all models is similar. The last column contains an observation with noise in it and all models have struggled on a noise part.



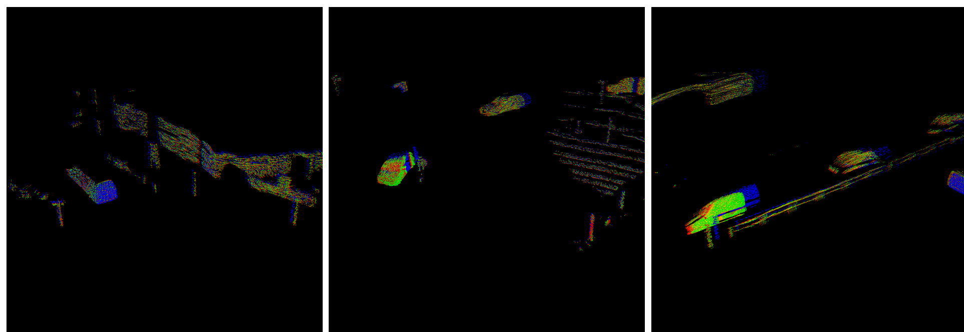
(a) Supervised learning with self-supervisory signals



(b) Full self-supervision



(c) Supervised joint learning of non-rigid flow and ego-motion with self-supervisory signals



(d) Fully self-supervised joint learning of non-rigid flow and ego-motion

Figure 5.5: Qualitative evaluation on KITTI. Overall, all of our models including the self-supervised ones were able to generalize to real world data. (b) and (d) show larger magnitude of error on incomplete shapes.

Chapter 6

Conclusion

This chapter concludes the thesis and outlines the directions of future work. In this thesis, we presented a novel way to scene flow estimation, which investigated and unified three core ideas in our approach. These main areas are 3D deep learning from point clouds, self-supervised learning and decomposition of total flow into non-rigid residual flow and ego-motion.

Our approach employed temporal consistency property of a sequence of point clouds to design self-supervisory learning signals. Self-supervision was investigated in form of hybrid training, fully self-supervised training and fine tuning, which we contrasted and compared to purely supervised mode. Implemented method allows training on datasets with complete availability of ground-truth, on datasets with incomplete labels and on real world datasets with no ground-truth at all. We performed an internal comparison of different configurations of the method along with an external comparison with current state of the art supervised approaches to determine the best approach. We tested our models across different domains on the synthetic dataset FlyingThings3D, on the semi-synthetic dataset RefRESH and on the real world dataset KITTI.

Through experimentation we confirmed our initial hypothesis that hybrid training can outperform purely supervised training on most of the metrics we defined. In hybrid training we complemented the supervised loss with our self-supervisory loss terms, which allowed us to outperform the current state of the art supervised methods. For example we reduced EPE3D on FlyingThings3D by more than 6%, despite being trained for 2/3 of the epochs the supervised baseline [16] was trained on (table 5.6). Furthermore, our internal comparison showed that supervised training with self-supervisory signals consistently outperforms purely supervised configurations.

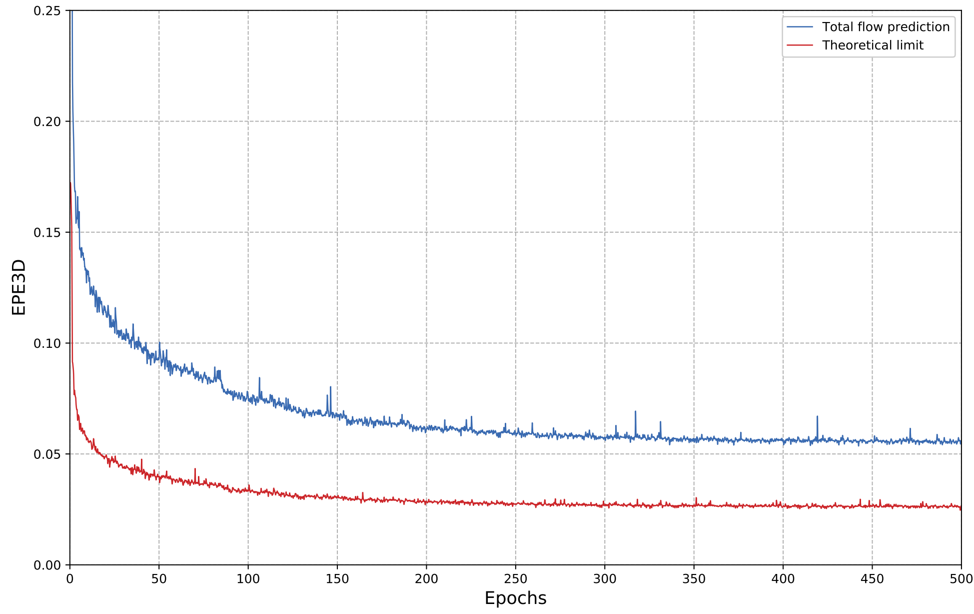
We investigated the potential of self-supervised applied to scene flow estimation. Although self-supervised configurations were not able to outper-

form the supervised baselines in terms of quantitative results, they nevertheless showed promising results especially in terms of qualitative assessment. Our observation was that self-supervision was able to perform close to supervised methods in terms of qualitative results on simpler scene with for example complete shapes and distinct edges but struggled with incomplete shapes or cluttered regions of the scene. Self-supervised training from scratch can especially be useful in scenarios with real world data where there was no ground truth at all and where qualitative results is the main evaluation criteria. We demonstrated that supervised pre-training and self-supervised fine tuning could be a viable alternative to self-supervised training from scratch. Concretely, we achieved an improvement of 13% in EPE3D when fine tuned on RefRESH using pre-training in supervised mode with self-supervisory signals on FlyingThings3D (table 5.2).

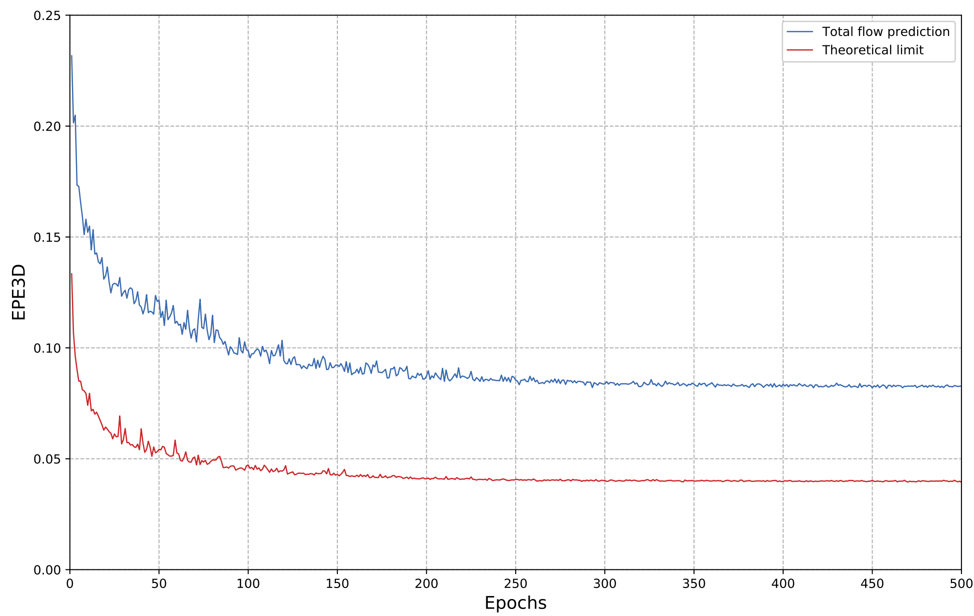
When it comes to decomposition of total scene flow into a rigid and non-rigid parts, fully self-supervised configuration was the one, which benefited the most. Concretely, we have observed better performance on the majority of the metrics when using decomposition of the flow during fully-supervised training. For example an improvement in EPE3D trained on FlyingThings3D was 15% (tables 5.2, 5.3). Additionally, we observed a slight qualitative improvement when using scene flow decomposition for self-supervision. On the hand, introducing scene flow decomposition decreased the performance for configurations, which contained the supervised part in them. Nevertheless, one can still benefit from using scene flow decomposition perspective of the flow because we get more information about static and moving parts of the scene along with the total flow after performing an inference pass.

6.1 Future Work

There are several directions to extend the work of this thesis. The main direction of scene flow decomposition is to investigate sequential learning of the rigid and non-rigid parts of the flow. One of the goals for our work was to learn the relative pose and non-rigid flow together with one single network. An alternative path to our approach could be to first estimate the relative pose separately and then transform the input and learn the non-rigid flow only. For learning the relative pose one could potentially employ either a classical point cloud registration algorithm such as [60] or a deep learning based point cloud registration [11]. To obtain a theoretical improvement we applied the ground truth of relative camera pose, applied it to the inputs and obtained the predictions for total flow, which we compared to the algorithm, which predicts the total flow directly. The limit for training and validation loss functions if we are to have the perfect relative camera pose estimation is shown in figure 6.1.



(a) Training



(b) Validation

Figure 6.1: Lower bound for total flow estimation. The red curve demonstrates the total scene flow losses if we can achieve zero error relative camera pose predictions. Both runs were trained on RefRESH.

Including self-supervisory signals to our self-supervised loss proposed in concurrent works [50, 54] could be an effective way to complement our temporal self-supervisory signals with the geometric self-supervisory signals and potentially help in both hybrid and fully-supervised training. Including the colors to the signal vector of a point cloud can provide a foundation for posing a photometric consistency constraint between point cloud frames.

Another direction for improvement is computational speed and efficiency. Original implementation of the network we used contains some parts, which are computed on the CPU, which makes the computation of forward and backward passes slow, which significantly increases the training time. We implemented the GPU based version of permutohedral layers, which allowed for our preliminary design to reduce the training time per epoch epoch by roughly 12%. Furthermore, unlike the original implementation, our implementation allowed us to train with batch size more than 1, which can significantly speed up the computation. Nevertheless, future investigation in avoiding anomalies during cycle consistency is needed to show the benefit of self-supervision. A further computational improvement is implementing a GPU based k-d trees [38] to allow an efficient nearest neighbour search then the one with similarity matrices. Furthermore, speeding up the computation will also allow a deeper investigation of larger cycle sizes k beyond 1 for the cycle consistency loss.

Appendix A

Additional Qualitative Results

A.1 FlyingThings3D

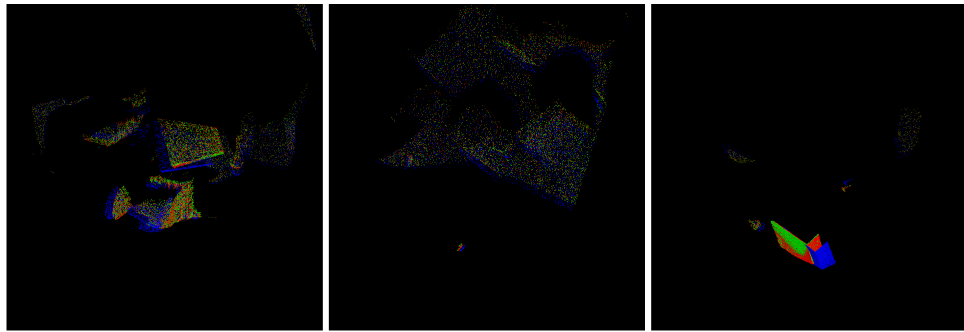
Figure A.1 provides more examples of prediction results on FlyingThings3D.

A.2 RefRESH

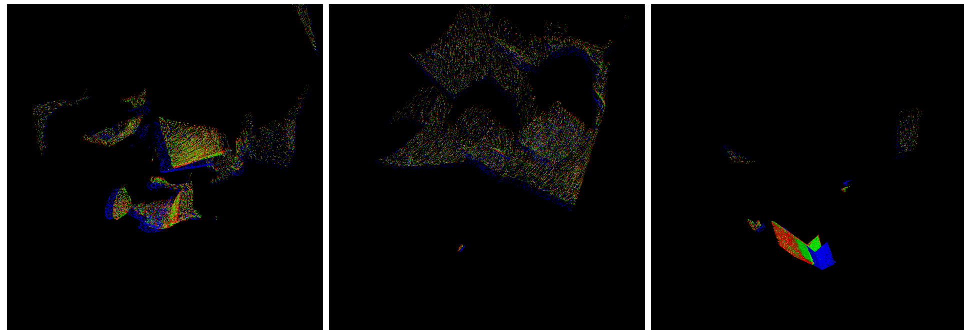
Figure A.2 provides more examples of prediction results on RefRESH.

A.3 KITTI

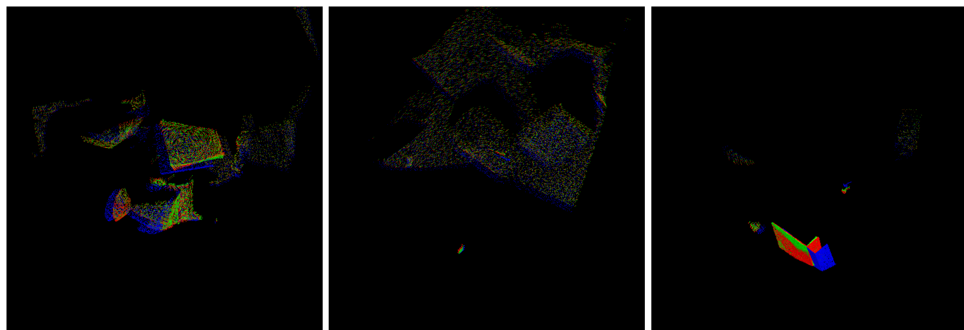
Figure A.3 provides more examples of prediction results on KITTI.



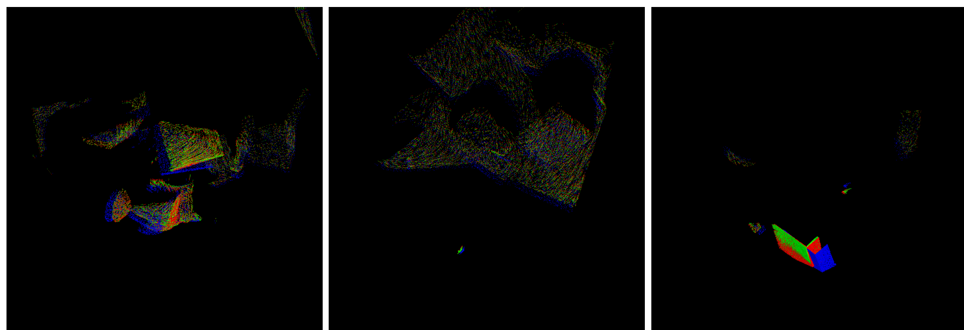
(a) Supervised learning with self-supervisory signals



(b) Full self-supervision

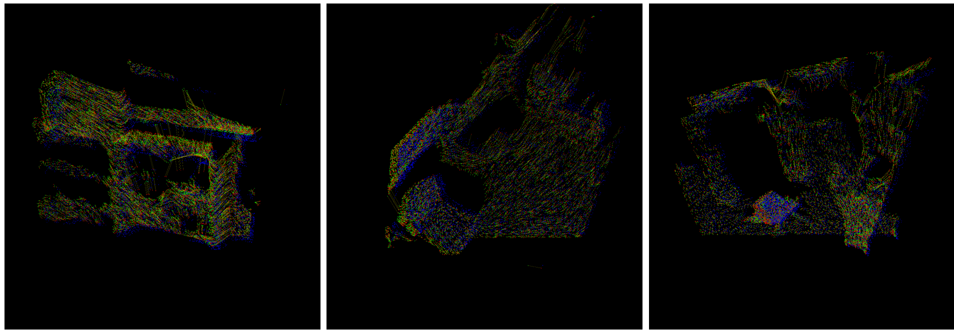


(c) Supervised joint learning of non-rigid flow and ego-motion with self-supervisory signals

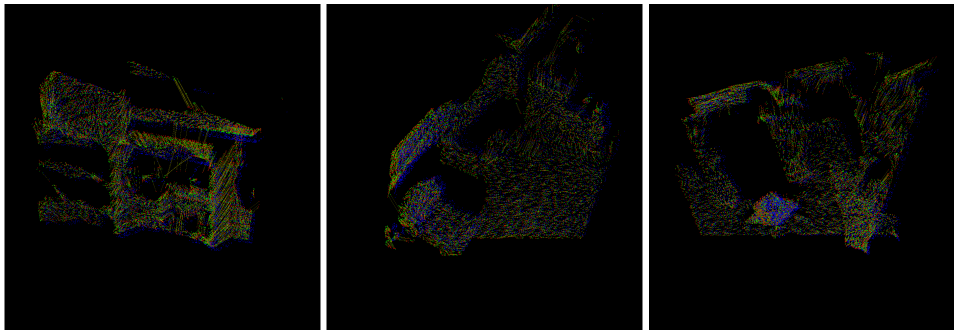


(d) Fully self-supervised joint learning of non-rigid flow and ego-motion

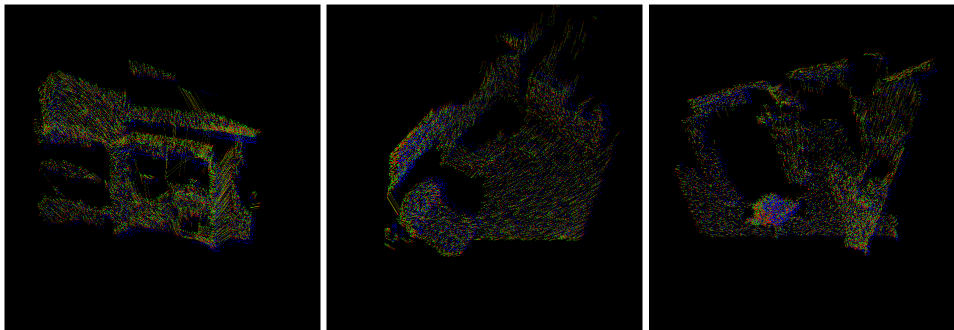
Figure A.1: Additional qualitative evaluation on FlyingThings3D. The number of points is 8192.



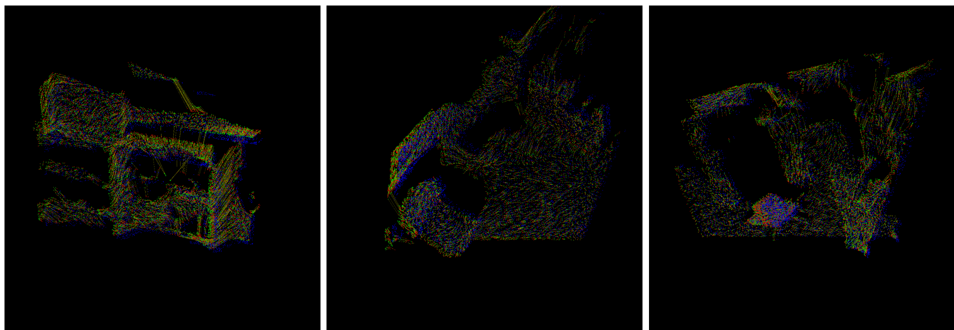
(a) Supervised learning with self-supervisory signals



(b) Full self-supervision

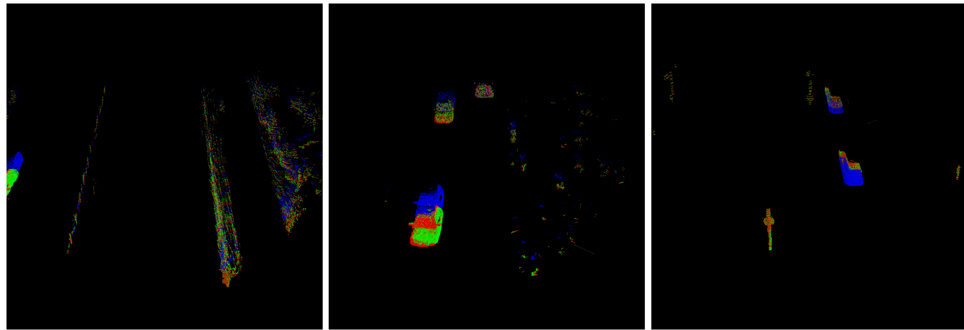


(c) Supervised joint learning of non-rigid flow and ego-motion with self-supervisory signals

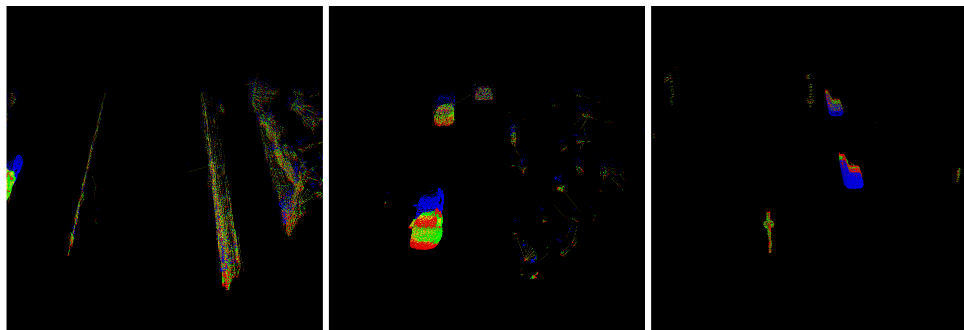


(d) Fully self-supervised joint learning of non-rigid flow and ego-motion

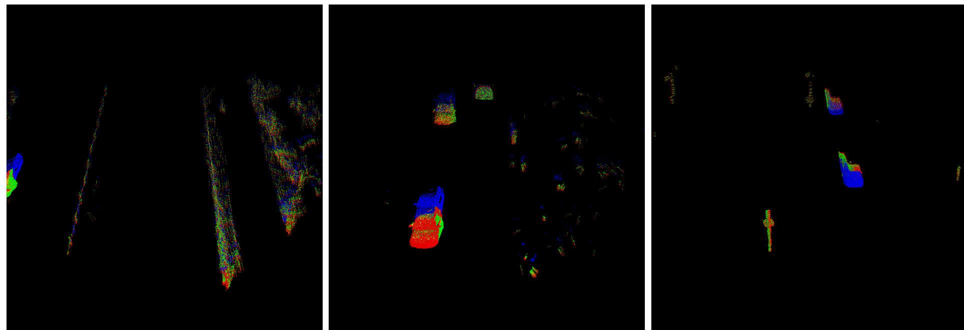
Figure A.2: Additional qualitative evaluation on RefRESH. The number of points is 8192.



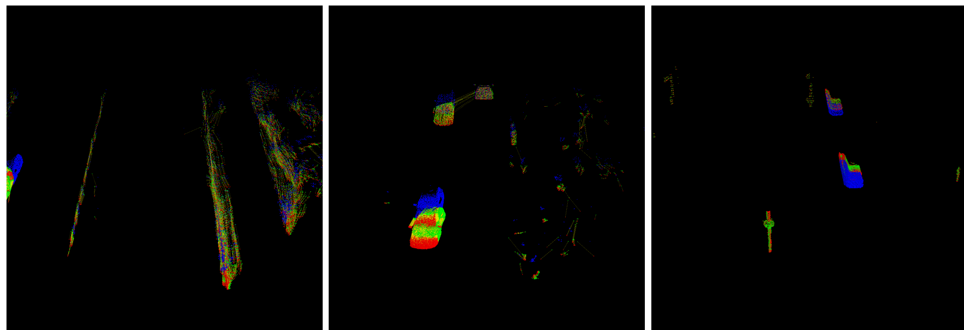
(a) Supervised learning with self-supervisory signals



(b) Full self-supervision



(c) Supervised joint learning of non-rigid flow and ego-motion with self-supervisory signals



(d) Fully self-supervised joint learning of non-rigid flow and ego-motion

Figure A.3: Additional qualitative evaluation on KITTI. The number of points is 8192.

Bibliography

[1] Andrew Adams, Jongmin Baek, and Myers Abraham Davis. Fast high-dimensional filtering using the permutohedral lattice. In *Computer Graphics Forum*, volume 29, pages 753–762. Wiley Online Library, 2010.

[2] Andrew B Adams. High-dimensional gaussian filtering for computational photography. 2011.

[3] Yasuhiro Aoki, Hunter Goforth, Rangaprasad Arun Srivatsan, and Simon Lucey. Pointnetlk: Robust & efficient point cloud registration using pointnet. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7163–7172, 2019.

[4] K Somani Arun, Thomas S Huang, and Steven D Blostein. Least-squares fitting of two 3-d point sets. *IEEE Transactions on pattern analysis and machine intelligence*, (5):698–700, 1987.

[5] Jongmin Baek and Andrew Adams. Some useful properties of the permutohedral lattice for gaussian filtering. In *Technical report*. Stanford University, 2009.

[6] Aseem Behl, Despoina Paschalidou, Simon Donné, and Andreas Geiger. Pointflownet: Learning representations for rigid motion estimation from point clouds. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7962–7971, 2019.

[7] Paul J Besl and Neil D McKay. Method for registration of 3-d shapes. In *Sensor fusion IV: control paradigms and data structures*, volume 1611, pages 586–606. International Society for Optics and Photonics, 1992.

[8] Jiawang Bian, Zhichao Li, Naiyan Wang, Huangying Zhan, Chunhua Shen, Ming-Ming Cheng, and Ian Reid. Unsupervised scale-consistent

depth and ego-motion learning from monocular video. In *Advances in Neural Information Processing Systems*, pages 35–45, 2019.

[9] Daniel J Butler, Jonas Wulff, Garrett B Stanley, and Michael J Black. A naturalistic open source movie for optical flow evaluation. In *European conference on computer vision*, pages 611–625. Springer, 2012.

[10] Jorge L Charco, Boris X Vintimilla, and Angel D Sappa. Deep learning based camera pose estimation in multi-view environment. In *2018 14th International Conference on Signal-Image Technology & Internet-Based Systems (SITIS)*, pages 224–228. IEEE, 2018.

[11] Christopher Choy, Wei Dong, and Vladlen Koltun. Deep global registration. *arXiv preprint arXiv:2004.11540*, 2020.

[12] Ayush Dewan, Tim Caselitz, Gian Diego Tipaldi, and Wolfram Burgard. Rigid scene flow for 3d lidar scans. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1765–1770. IEEE, 2016.

[13] Haoqiang Fan, Hao Su, and Leonidas J Guibas. A point set generation network for 3d object reconstruction from a single image. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 605–613, 2017.

[14] Ge Gao, Mikko Lauri, Yulong Wang, Xiaolin Hu, Jianwei Zhang, and Simone Frintrop. 6d object pose regression via supervised learning on point clouds. *arXiv preprint arXiv:2001.08942*, 2020.

[15] Zan Gojcic, Caifa Zhou, Jan D Wegner, Leonidas J Guibas, and Tolga Birdal. Learning multiview 3d point cloud registration. *arXiv preprint arXiv:2001.05119*, 2020.

[16] Xiuye Gu, Yijie Wang, Chongruo Wu, Yong Jae Lee, and Panqu Wang. Hplflownet: Hierarchical permutohedral lattice flownet for scene flow estimation on large-scale point clouds. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3254–3263, 2019.

[17] Richard Hartley, Jochen Trumpf, Yuchao Dai, and Hongdong Li. Rotation averaging. *International journal of computer vision*, 103(3):267–305, 2013.

[18] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2 edition, 2004.

[19] Eddy Ilg, Tonmoy Saikia, Margret Keuper, and Thomas Brox. Occlusions, motion and depth boundaries with a generic network for disparity, optical flow or scene flow estimation. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 614–630, 2018.

[20] Eldar Insafutdinov and Alexey Dosovitskiy. Unsupervised learning of shape and pose with differentiable point clouds. In *Advances in neural information processing systems*, pages 2802–2812, 2018.

[21] Mariano Jaimez, Christian Kerl, Javier Gonzalez-Jimenez, and Daniel Cremers. Fast odometry and scene flow from rgb-d cameras based on geometric clustering. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3992–3999. IEEE, 2017.

[22] Varun Jampani, Martin Kiefel, and Peter V Gehler. Learning sparse high dimensional filters: Image filtering, dense crfs and bilateral neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4452–4461, 2016.

[23] Samuel Joutard, Reuben Dorent, Amanda Isaac, Sebastien Ourselin, Tom Vercauteren, and Marc Modat. Permutohedral attention module for efficient non-local neural networks. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 393–401. Springer, 2019.

[24] Alex Kendall, Matthew Grimes, and Roberto Cipolla. Posenet: A convolutional network for real-time 6-dof camera relocalization. In *Proceedings of the IEEE international conference on computer vision*, pages 2938–2946, 2015.

[25] Martin Kiefel, Varun Jampani, and Peter V Gehler. Permutohedral lattice cnns. *arXiv preprint arXiv:1412.6618*, 2014.

[26] Seokju Lee, Sunghoon Im, Stephen Lin, and In So Kweon. Learning residual flow as dynamic motion from stereo videos. *arXiv preprint arXiv:1909.06999*, 2019.

[27] Yi Li, Gu Wang, Xiangyang Ji, Yu Xiang, and Dieter Fox. Deepim: Deep iterative matching for 6d pose estimation. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 683–698, 2018.

[28] Liang Liu, Guangyao Zhai, Wenlong Ye, and Yong Liu. Unsupervised learning of scene flow estimation fusing with local rigidity. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, pages 876–882. AAAI Press, 2019.

[29] Pengpeng Liu, Michael Lyu, Irwin King, and Jia Xu. Selflow: Self-supervised learning of optical flow. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4571–4580, 2019.

[30] Xingyu Liu, Charles R Qi, and Leonidas J Guibas. Flownet3d: Learning scene flow in 3d point clouds. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 529–537, 2019.

[31] Zhaoyang Lv, Kihwan Kim, Alejandro Troccoli, Deqing Sun, James M Rehg, and Jan Kautz. Learning rigidity in dynamic scenes with a moving camera for 3d motion field estimation. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 468–484, 2018.

[32] Wei-Chiu Ma, Shenlong Wang, Rui Hu, Yuwen Xiong, and Raquel Urtasun. Deep rigid instance scene flow. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3614–3622, 2019.

[33] Nikolaus Mayer, Eddy Ilg, Philip Hausser, Philipp Fischer, Daniel Cremers, Alexey Dosovitskiy, and Thomas Brox. A large dataset to train convolutional networks for disparity, optical flow, and scene flow estimation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4040–4048, 2016.

[34] Iaroslav Melekhov, Juha Ylioinas, Juho Kannala, and Esa Rahtu. Relative camera pose estimation using convolutional neural networks. In *International Conference on Advanced Concepts for Intelligent Vision Systems*, pages 675–687. Springer, 2017.

[35] Moritz Menze and Andreas Geiger. Object scene flow for autonomous vehicles. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[36] Moritz Menze, Christian Heipke, and Andreas Geiger. Joint 3d estimation of vehicles and scene flow. In *ISPRS Workshop on Image Sequence Analysis (ISA)*, 2015.

[37] Himangi Mittal, Brian Okorn, and David Held. Just go with the flow: Self-supervised scene flow estimation. *arXiv preprint arXiv:1912.00497*, 2019.

[38] Naohito Nakasato. Implementation of a parallel tree method on a gpu. *Journal of Computational Science*, 3(3):132–141, 2012.

[39] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 652–660, 2017.

[40] Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. In *Advances in neural information processing systems*, pages 5099–5108, 2017.

[41] Julian Quiroga, Thomas Brox, Frédéric Devernay, and James Crowley. Dense semi-rigid scene flow estimation from rgbd images. In *European Conference on Computer Vision*, pages 567–582. Springer, 2014.

[42] Torsten Sattler, Qunjie Zhou, Marc Pollefeys, and Laura Leal-Taixe. Understanding the limitations of cnn-based absolute camera pose regression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3302–3312, 2019.

[43] Yoli Shavit and Ron Ferens. Introduction to camera pose estimation with deep learning. *arXiv preprint arXiv:1907.05272*, 2019.

[44] Maria Shugrina, Ziheng Liang, Amlan Kar, Jiaman Li, Angad Singh, Karan Singh, and Sanja Fidler. Creative flow+ dataset. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5384–5393, 2019.

[45] Olga Sorkine. Laplacian mesh processing. *Eurographics (STARs)*, 29, 2005.

[46] Hang Su, Varun Jampani, Deqing Sun, Subhransu Maji, Evangelos Kalogerakis, Ming-Hsuan Yang, and Jan Kautz. Splatnet: Sparse lattice networks for point cloud processing. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2530–2539, 2018.

[47] Hang Su, Subhransu Maji, Evangelos Kalogerakis, and Erik Learned-Miller. Multi-view convolutional neural networks for 3d shape recognition. In *Proceedings of the IEEE international conference on computer vision*, pages 945–953, 2015.

[48] Sundar Vedula, Simon Baker, Peter Rander, Robert Collins, and Takeo Kanade. Three-dimensional scene flow. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, volume 2, pages 722–729. IEEE, 1999.

[49] Xiaolong Wang, Allan Jabri, and Alexei A Efros. Learning correspondence from the cycle-consistency of time. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2566–2576, 2019.

[50] Zirui Wang, Shuda Li, Henry Howard-Jenkins, Victor Prisacariu, and Min Chen. Flownet3d++: Geometric losses for deep scene flow estimation. In *The IEEE Winter Conference on Applications of Computer Vision*, pages 91–98, 2020.

[51] Patrick Wieschollek, Oliver Wang, Alexander Sorkine-Hornung, and Hendrik Lensch. Efficient large-scale approximate nearest neighbor search on the gpu. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2027–2035, 2016.

[52] Sascha Wirges, Johannes Gräter, Qiuhao Zhang, and Christoph Stiller. Self-supervised flow estimation using geometric regularization with applications to camera image and grid map sequences. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, pages 1782–1787. IEEE, 2019.

[53] Jian Wu, Liwei Ma, and Xiaolin Hu. Delving deeper into convolutional neural networks for camera relocalization. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5644–5651. IEEE, 2017.

[54] Wenxuan Wu, Zhiyuan Wang, Zhuwen Li, Wei Liu, and Li Fuxin. Pointpwc-net: A coarse-to-fine network for supervised and self-supervised scene flow estimation on 3d point clouds. *arXiv preprint arXiv:1911.12408*, 2019.

[55] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 3d shapenets: A deep representation for volumetric shapes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1912–1920, 2015.

[56] Zhichao Yin and Jianping Shi. Geonet: Unsupervised learning of dense depth, optical flow and camera pose. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1983–1992, 2018.

[57] Wentao Yuan, David Held, Christoph Mertz, and Martial Hebert. Iterative transformer network for 3d point cloud. *arXiv preprint arXiv:1811.11209*, 2018.

[58] Jure Žbontar and Yann LeCun. Stereo matching by training a convolutional neural network to compare image patches. *The journal of machine learning research*, 17(1):2287–2318, 2016.

[59] Jialiang Zhang, Soroosh Khoram, and Jing Li. Efficient large-scale approximate nearest neighbor search on opencl fpga. In *Proceedings of the*

IEEE Conference on Computer Vision and Pattern Recognition, pages 4924–4932, 2018.

[60] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Fast global registration. In *European Conference on Computer Vision*, pages 766–782. Springer, 2016.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Self-Supervised Learning of Non-Rigid Residual Flow and Ego-Motion in Dynamic 3D Scenes

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	First name(s):
Tishchenko	Ivan

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

Zurich, 29.05.2020

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.