Rheinische
Friedrich-Wilhelms-
Universität Bonn

Institute for Computer Science
Department VI
Autonomous Intelligent Systems

Rheinische Friedrich-Wilhelms-Universität Bonn

Master Thesis

Temporal Semantic Segmentation using Sparse Permutohedral Lattices

Author:
Peer Schütt

First Examiner:
Prof. Dr. Sven Behnke

Second Examiner:
Dr. Jens Behley

Supervisor:
Radu Alexandru Rosu

Date: September 18, 2021

Abstract

Semantic segmentation answers the question of which parts of a scene belong to the same object. It is a core ability required by autonomous agents as it is crucial for navigation and interaction with the environment. Most recent approaches only take information from a single scan as input and therefore disregard temporal information. This renders the agent incapable of using past information, segmenting moving and stationary objects or correcting faulty semantic segmentations. Incorporating temporal information can be used to address these issues. For this task, a backbone network for semantic segmentation is extended using methods from recurrent neural networks. A sequence of consecutive LiDAR point clouds are used as its input and the network outputs a prediction for every point of the last point cloud in the sequence. Different recurrent modules are evaluated and compared to each other. We present competitive results for the network's performance on the SemanticKITTI dataset that contains LiDAR scans from real urban environments.

Contents

1 Introduction

Understanding and segmenting the surrounding area comes to us humans quite natural. However, it still proves to be a challenging problem for autonomous agents that have to cope with environments, which are designed by humans for humans. To measure their capabilities regarding scene understanding, a variety of tasks have been proposed. One of them being semantic segmentation:

In this thesis, we deal with the semantic segmentation of 3D point clouds, which is defined as the process of predicting a class for every point in the point cloud. It is one of the most crucial capabilities of autonomous agents, because an agent has to distinguish between different objects and classes in order to navigate and interact with its environment.

For 3D point clouds, semantic segmentation is especially challenging, due to undersampling of the scene, a lack of explicit structure in point clouds that are recorded in real-world scenarios and the scale of point clouds, whose size can be orders of magnitude larger when compared to images that depict the same scene. In this thesis, we assume that all clouds have been recorded continuously by a sensor like a laser scanner or a depth camera.

Segmenting single point clouds is already a heavily researched area with a large number of different architectures and methods. However, research into the generalization of semantic segmentation to segmenting sequences of point clouds is currently sparse. This topic is also called semantic segmentation of 4D point clouds or temporal semantic segmentation. This task has an important benefit over the single scan segmentation: It allows an agent to distinguish between moving and stationary objects and provides temporal information that can not be inferred from single scan segmentation.

Additionally, planning and decision making of an autonomous agent needs to account for dynamic objects. Information about which vehicles are parked, moving or just waiting for a green light is crucial for navigating in an urban environment. Temporal information might even prove beneficial for the detection of smaller objects, that naturally have a smaller number of points, by leveraging information from past clouds — if the same object appears in many consecutive scans it is not as likely to be overlooked. An example for this in urban scenarios are pedestrians or bicyclists.

In consonance with the previously presented motivation, these goals for this thesis were formulated:

- Extend the network LatticeNet (Rosu et al. 2020) in order to integrate temporal information.
- Segment moving from non-moving objects in real urban environments.
- Provide state-of-the-art results on the SemanticKITTI (Behley et al. 2019) dataset.

LatticeNet (Rosu et al. 2020) is a recently proposed architecture that is able to produce state-of-the-art segmentation results for the SemanticKITTI dataset. This dataset is one of the few that provides point cloud sequences with ground truth annotations, which enables to train models to perform temporal semantic segmentation. Furthermore, an official competition[1] exists for this dataset in which both single scan segmentation and temporal segmentation methods can be compared.

To achieve the goals we will extend LatticeNet by adding recurrent connections to it. The proposed architecture processes a sequence of point clouds and returns a per-point segmentation for the last cloud in its input sequence.

An intuitive adaption of existing state-of-the-art single scan segmentation network to the temporal segmentation problem is to accumulate the point cloud sequence into a single cloud and train the model on these point clouds. However, these approaches are limited by their memory consumption and resolutions constraints. Furthermore, such an approach requires the network to establish the temporal relationships between points from its unstructured input and is able to access a temporal context, which is intrinsic to recurrent architectures. In contrast, our network takes multiple point clouds sequentially as input and can therefore decide to reuse parts of the network from the previous timesteps.

[1] `https://competitions.codalab.org/competitions/20331`

2 Fundamentals

In this chapter, we define the notation (Sec. 2.1), present the structure permutohedral lattice (Sec. 2.2), explain recurrent neural networks (Sec. 2.3), introduce the backbone network LatticeNet (Sec. 2.4) and discuss scene flow (Sec. 2.5).

2.1 Notation

Bold upper-case characters are used to denote matrices and bold lower-case characters to denote vectors. When the term point is used it refers to one point p of a point cloud P. The term cloud is used synonymously to point cloud. The term coordinate is only used to describe the spatial position of a lattice vertex. In this thesis, $\mathbb{N}$ is defined as the set of all non-negative integers and $\mathbb{N}^+$ is defined as $\mathbb{N} \setminus \{0\}$.

Each point of a cloud is defined as a tuple $p = (\mathbf{g}_p, \mathbf{f}_p)$. $\mathbf{g}_p \in \mathbb{R}^d$ represents the coordinates of the point and $\mathbf{f}_p \in \mathbb{R}^{f_d}$ denotes the features stored at this point p, like normals and reflectance. A point cloud contains m points and is denoted by $P = (\mathbf{G}, \mathbf{F})$. $\mathbf{G} \in \mathbb{R}^{m \times d}$ denotes the positions matrix and $\mathbf{F} \in \mathbb{R}^{m \times f_d}$ the feature matrix of the cloud. The feature matrix $\mathbf{F}$ can also be filled with zeros, if per-point features are not provided or needed.

2.2 Permutohedral Lattice

Due to the sparseness, the uneven sampling density and the overall lack of structure in 3D data, we have to define a structured way to store this data. For this we utilize permutohedral lattices (Adams, Baek, and Davis 2010; Rosu et al. 2020).

The permutohedral lattice was first introduced by Adams, Baek, and Davis 2010. A d-dimensional lattice is created by projecting the scaled regular grid $(d+1) \in \mathbb{Z}^{d+1}$ along the vector $\mathbf{1} = [1, \ldots, 1]$ onto the hyperplane H_d: $\mathbf{p} \cdot \mathbf{1} = 0$.

The vertices of a lattice (Sec. 2.2) are defined as a tuple $v = (\mathbf{c}_v, \mathbf{x}_v)$, where $\mathbf{c}_v \in \mathbb{Z}^{(d+1)}$ denotes the coordinates of the vertex and $\mathbf{x}_v \in \mathbb{R}^{v_d}$ represents the values stored at vertex v. A full lattice contains k vertices and is defined as

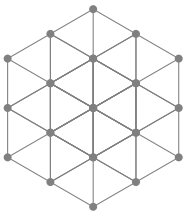


Figure 2.1: An example permutohedral lattice for $d = 2$ that tesselates the space into triangles: It represents the datastructure that all operations of the network are performed on.

$V = (\mathbf{C}, \mathbf{X})$, where $\mathbf{C} \in \mathbb{Z}^{k \times (d+1)}$ represents the coordinate matrix and $\mathbf{X} \in \mathbb{R}^{k \times v_d}$ the value matrix.

The input space is split into uniform d-dimensional simplices by the lattice. For the case $d = 2$ the space is tessellated by triangles as it is illustrated in Fig. 2.1. For every d-dimensional point of the cloud that should be represented by the lattice the enclosing simplex can be computed in $O(d^2)$ by a rounding algorithm. We denote the set of neighbors of vertex v with $N(v)$ and $|N(v)| = 2(d + 1)$.

The implementation for permutohedral lattices (Rosu et al. 2020) saves the lattice sparsely into memory using a hash map. This allows us to only allocate the simplices that contain the 3D area of interest. The hash map is discussed in more detail in Sec. 2.4.4. Due to this sparse allocation the actual number of neighbors can be smaller than $2(d + 1)$, because neighboring vertices might not be allocated yet. An example neighborhood of a lattice is visualized in Fig. 2.7.

Compared to a standard cubic voxel representation, the permutohedral lattice scales better to higher dimensions, since the number of vertices for each simplex is given by $d + 1$, in contrast to 2^d for standard voxels.

2.3 Recurrent Neural Networks

The explanations in this section are based on the book "Deep Learning" written by Goodfellow, Bengio, and Courville 2016. They describe recurrent neural networks (RNNs) in the following manner:

> "Recurrent neural networks, or RNNs (Rumelhart, Hinton, and Williams 1986), are a family of neural networks for processing sequential data. Much as a convolutional network is a neural network that is specialized for processing a grid of values $\mathbf{X}$ such as an image, a recurrent neural network is a neural network that is specialized for processing a sequence of values $\mathbf{X}^{(1)}, ..., \mathbf{X}^{(\tau)}$. Just as convolutional networks can

> readily scale to images with large width and height, and some convolutional networks can process images of variable size, recurrent networks can scale to much longer sequences than would be practical for networks without sequence-based specialization. Most recurrent networks can also process sequences of variable length."(Goodfellow, Bengio, and Courville 2016)

We will focus on RNNs that take a finite sequence $X = (\mathbf{X}^{(1)}, ..., \mathbf{X}^{(\tau)})$ as input and output a finite sequence $O = (\mathbf{O}^{(1)}, ..., \mathbf{O}^{(\delta)})$ with $\tau, \delta \in \mathbb{N}^+$. One can distinguish between three basic recurrent neural network types which perform a sequence-to-sequence mapping:

- **One-to-many**: The input is a sequence consisting only of one value $\mathbf{X}^{(1)}$ and uses this information to output a sequence of values $\mathbf{O}^{(1)}, ..., \mathbf{O}^{(\delta)}$.
- **Many-to-one**: The input is a sequence consisting of values $\mathbf{X}^{(1)}, ..., \mathbf{X}^{(\tau)}$ and the network outputs a single value $\mathbf{O}^{(\delta)}$ after processing all input values.
- **Many-to-many**: The input is a sequence consisting of values $\mathbf{X}^{(1)}, ..., \mathbf{X}^{(\tau)}$ and the network outputs a sequence of values $\mathbf{O}^{(1)}, ..., \mathbf{O}^{(\delta)}$. It is important to note that between reading the first sequence value $\mathbf{X}^{(1)}$ and generating $\mathbf{O}^{(1)}$ an arbitrary amount of input values can be processed. An example can be seen in Fig. 2.2 on the left.

RNNs work with a hidden state $\mathbf{H}$, that is updated after processing each input $\mathbf{X}^{(t)}$ and used for computation of the output $\mathbf{O}$. It should aggregate all relevant information about the current input $\mathbf{X}^{(t)}$ and all previous inputs $\mathbf{X}^{(1)}, ..., \mathbf{X}^{(t-1)}$. Therefore $\mathbf{H}^{(t)}$ is calculated using $\mathbf{X}^{(t)}$ and $\mathbf{H}^{(t-1)}$. This can be written as:

$$\mathbf{H}^{(t)} = f(\mathbf{H}^{(t-1)}, \mathbf{X}^{(t)}), \tag{2.1}$$

where f is the recurrent neural network. Processing a timestep t refers to processing one input value $\mathbf{X}^{(t)}$ and does not always relate to a certain time passed.

Gradient computation for RNNs is done by calculating the gradient on the unrolled computational graph according to the backpropagation algorithm common for neural networks. This method is called backpropagation through time (BPTT). One example of this unrolled computational graph can be seen on the right side of Fig. 2.2.

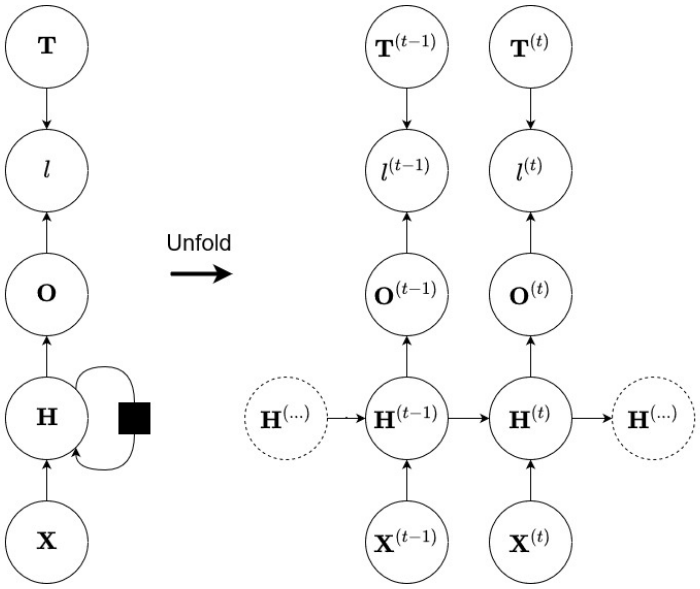


Figure 2.2: Computational graph of a many-to-many RNN that maps an input sequence $\mathbf{X}$ to an output sequence $\mathbf{O}$ and compares this to the target output $\mathbf{T}$ by computing the loss l. On the left the network is shown with the recurrent connection, while it is visualized unfolded on the right side (Goodfellow, Bengio, and Courville 2016).

2.3.1 Deep RNNs

A further development of RNNs are the Deep RNNs, as shown in Fig. 2.3. Three types are presented by Goodfellow, Bengio, and Courville 2016:

- RNNs with multiple hidden states at different depths of the network (Fig. 2.3(a)): These can be advantageous for networks that employ downsampling with convolutions. Using multiple of these convolution steps could help the network incorporate more information from the previous inputs.

- RNNs with an additional multilayer perceptron (MLP) that is computed from the hidden state (Fig. 2.3(b)): Adding this MLP could prove useful, but increases the length of the shortest path between the input and the output, which can be problematic during the backpropagation. Additionally the number of parameters increases.

- RNNs with a direct connection between $\mathbf{H}^{(t-1)}$ and $\mathbf{H}^{(t)}$ and an additional state computed by an MLP (Fig. 2.3(c)): This is a combination of both previous types.

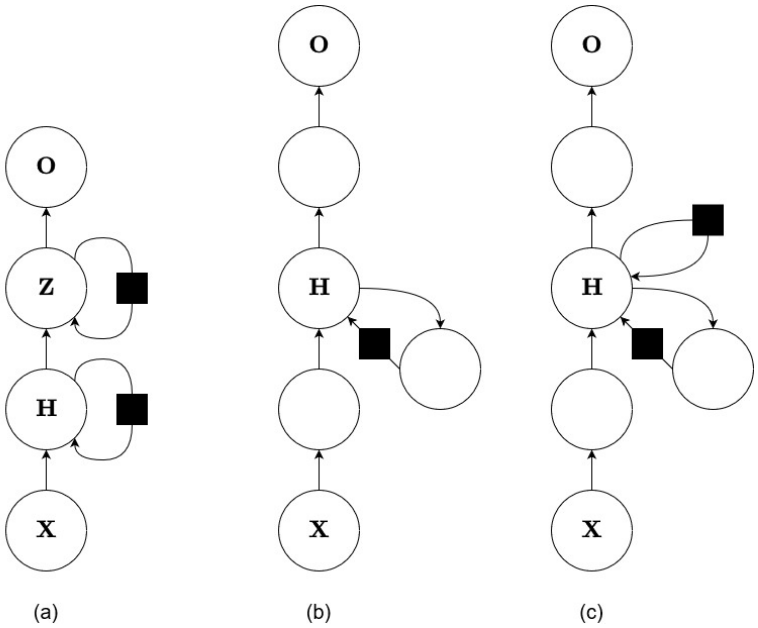


Figure 2.3: Different architectures for RNNs known as Deep RNNs (Goodfellow, Bengio, and Courville 2016).

2.3.2 The challenge of long-term dependencies

Gradients propagated over many timesteps tend to vanish or explode, because exponentially smaller weights are given to long-term interactions compared to short-term interactions, since the gradients are multiplied with many Jacobians. The probability of success with sequence lengths larger than 10 are nearly zero, if the basic concept is not altered (Goodfellow, Bengio, and Courville 2016). Selected methods that were developed to cope with this challenge are presented here:

Long short-term memory (LSTM) Long short-term memory (LSTM) cells are explicitly designed to avoid the long-term dependency problem (Hochreiter and Schmidhuber 1997). The cell state, which changes only slowly over time and has minor linear interactions, is the core of a LSMT cell. This cell state is influenced by the hidden state and the input at the current time step. The structure of a LSTM is visualized in Fig. 2.4a. The standard formulation of a LSTM cell can be

given by the following equations:

$$\mathbf{D}^{(t)} = \sigma(\mathbf{W_D} \cdot [\mathbf{H}^{(t-1)}, \mathbf{X}^{(t)}] + b_D), \tag{2.2}$$
$$\mathbf{I}^{(t)} = \sigma(\mathbf{W_I} \cdot [\mathbf{H}^{(t-1)}, \mathbf{X}^{(t)}] + b_I), \tag{2.3}$$
$$\tilde{\mathbf{K}}^{(t)} = tanh(\mathbf{W_K} \cdot [\mathbf{H}^{(t-1)}, \mathbf{X}^{(t)}] + b_K), \tag{2.4}$$
$$\mathbf{K}^{(t)} = \mathbf{D}^{(t)} \cdot \mathbf{K}^{(t-1)} + \mathbf{I}^{(t)} \cdot \tilde{\mathbf{K}}^{(t)}, \tag{2.5}$$
$$\mathbf{O}^{(t)} = \sigma(\mathbf{W_O} \cdot [\mathbf{H}^{(t-1)}, \mathbf{X}^{(t)}] + b_o), \tag{2.6}$$
$$\mathbf{H}^{(t)} = \mathbf{O}^{(t)} \cdot tanh(\mathbf{K}^{(t)}), \tag{2.7}$$

where $\mathbf{D}, \mathbf{I}, \mathbf{O}, \tilde{\mathbf{K}}, \mathbf{K}$ are the forget gate, input gate, output gate, memory cell content and new memory cell content, σ is the sigmoid function, $tanh$ is the hyperbolic tangent function and $\mathbf{W}, b$ are the respective weight matrices and biases.

Gated recurrent units (GRU) GRUs were proposed by Cho et al. 2014 as a variant of LSTM cells. They combine the forget and input gates into a single "update gate", getting rid of the cell state. Therefore this model is simpler than a standard LSTM cell. The performance of LSTM and GRU are comparable (Chung et al. 2014). The structure of GRU is visualized in Fig. 2.4b. The standard formulation of a GRU can be given by the following equations:

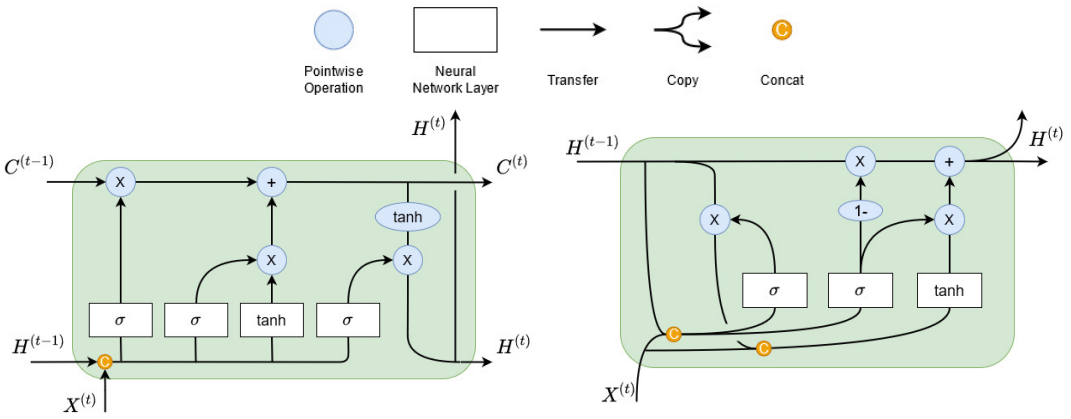
$$\mathbf{R}^{(t)} = \sigma(\mathbf{W}_R \cdot [\mathbf{H}^{(t-1)}, \mathbf{X}^{(t)}]), \tag{2.8}$$
$$\mathbf{Z}^{(t)} = \sigma(\mathbf{W}_Z \cdot [\mathbf{H}^{(t-1)}, \mathbf{X}^{(t)}]), \tag{2.9}$$
$$\tilde{\mathbf{H}}^{(t)} = tanh(\mathbf{W} \cdot [\mathbf{R}^{(t)} \cdot \mathbf{H}^{(t-1)}, \mathbf{X}^{(t)}]), \tag{2.10}$$
$$\mathbf{H}^{(t)} = (1 - \mathbf{Z}^{(t)}) \cdot \mathbf{H}^{(t-1)} + \mathbf{Z}^{(t)} \cdot \tilde{\mathbf{H}}^{(t)}, \tag{2.11}$$

where $\mathbf{R}, \mathbf{Z}$ are the reset and update gate, respectively.

Tackling the problem of vanishing/exploding gradients To address the problem of vanishing/exploding gradients the sequence length in this thesis is limited to a time horizon of no more than five inputs. Nonetheless, the GRU and LSTM are utilized as recurrent modules, because they are not only usable for long-term dependencies, but also for the basic propagation of information across timesteps.

2.4 LatticeNet

LatticeNet (Rosu et al. 2020) is used as the backbone network for this thesis. To combine states of different timesteps one has to compute correspondences between



(a) Long short-term memory (LSTM)

(b) Gated Recurrent Unit (GRU)

Figure 2.4: The structure of a LSTM cell and a GRU (Olah 2015).

the value matrices at different depths of the network. Fortunately, LatticeNet already provides this correspondence by design. To explain this we will elaborate the input to LatticeNet (Sec. 2.4.1), describe how selected operations work on permutohedral lattices (Sec. 2.4.3) and how the hashmap that stores the lattice values is created and maintained (Sec. 2.4.4). The definitions in the next three subsections were adapted from Rosu et al. 2020 and for detailed explanations of LatticeNet we refer to Rosu et al. 2020. The architecture of LatticeNet is presented in Fig. 2.5.

2.4.1 Input

The input to LatticeNet is a point cloud $P = (\mathbf{G}, \mathbf{F})$. To change the area of influence of the permutohedral lattice the scaling factor $\boldsymbol{\sigma} \in \mathbb{R}^d$ was defined. With it we can change the positions $\mathbf{G}$ to the scaled positions matrix $\mathbf{G}_s = \mathbf{G}/\boldsymbol{\sigma}$. The higher the scale $\boldsymbol{\sigma}$, the fewer vertices will be needed to cover the point cloud and the coarser the lattice will be. This is visualized in Fig. 2.6 and Fig. 2.7. Unless otherwise specified, we refer to $\mathbf{G}_s$ as $\mathbf{G}$.

2.4.2 Extended Notation

We denote with J_v the set of points p for which vertex v is one of the vertices of the containing simplices. Furthermore, we denote with $\mathcal{P}$ the PointNet module, with $\mathcal{D}_G$ and $\mathcal{D}_F$ the distribution of the point positions and the points features, respectively.

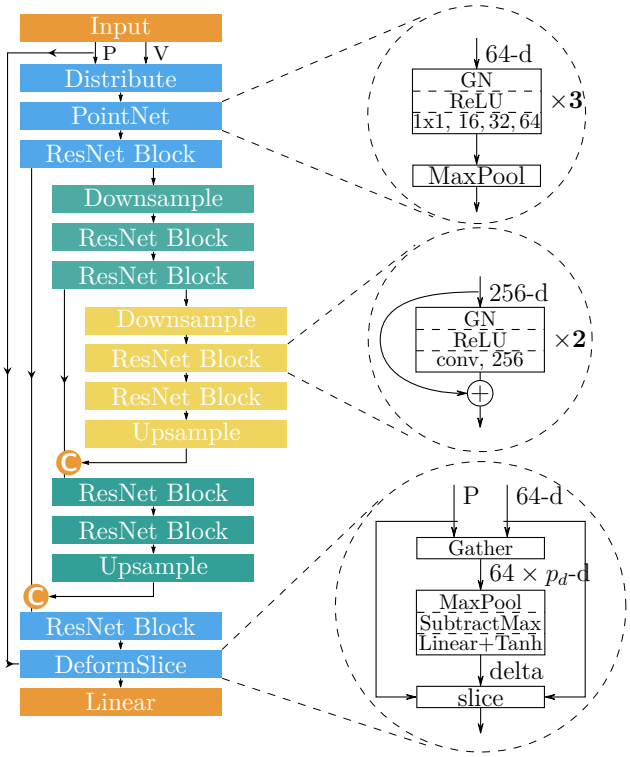


Figure 2.5: Architecture of LatticeNet (Rosu et al. 2020).

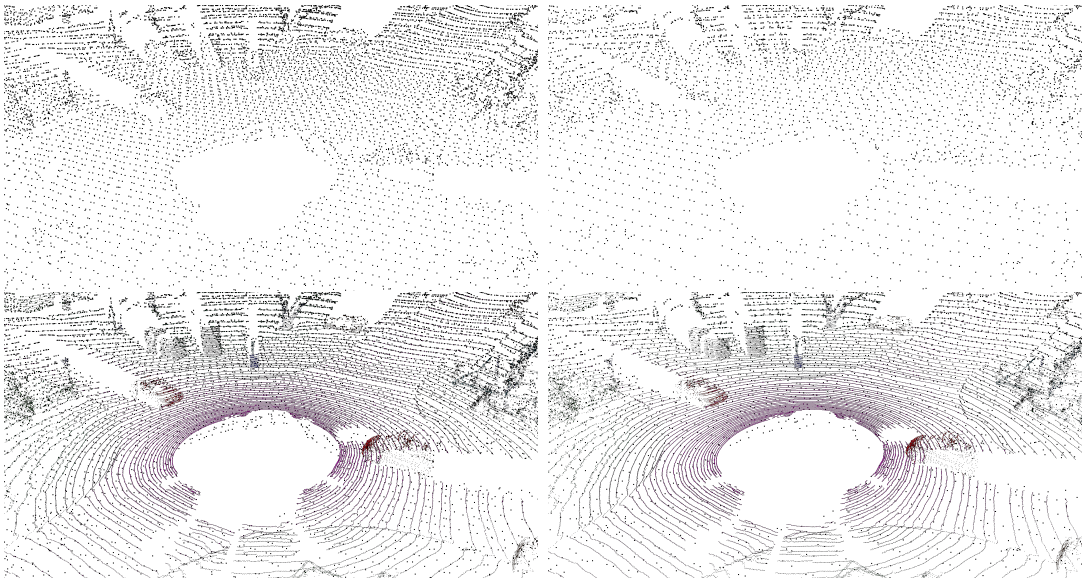


Figure 2.6: Comparison between lattice tesselation for different scaling factors $\boldsymbol{\sigma}$: The two scaling factors presented are 0.6 (left) and 0.9 (right). $\mathbf{G}_{0.6}$ consists of 42582 positions, while $\mathbf{G}_{0.9}$ consists of 22793 positions. The fine lattice covers the area more densely. Additionally the sparse allocation of lattice vertices is easy to see; only the areas where points are or were in the last timestep are covered with lattice vertices.

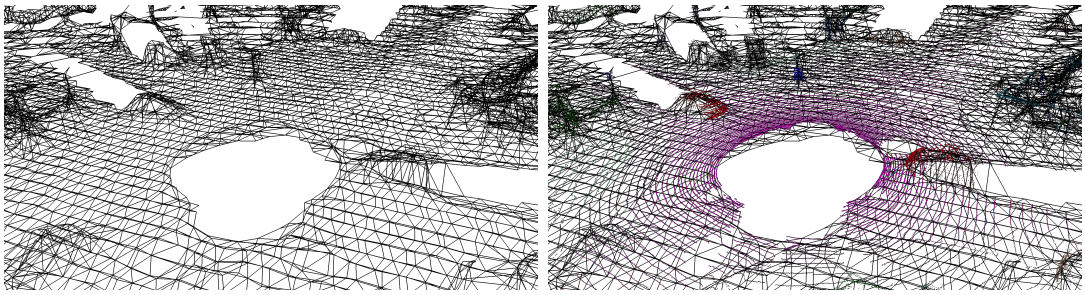


Figure 2.7: The neighborhood in a lattice visualized: On the left only the neighborhood is displayed, while on the right the input cloud is added to the scene. A scaling factor $\boldsymbol{\sigma}$ of 0.9 was used. It is important to note is that every vertex has at most $2(d+1)$ neighbors.

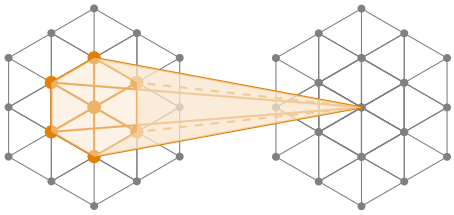


Figure 2.8: Convolution on the lattice structure: The one-hop neighborhood and the center vertex are convolved similarly to standard 2D convolutions. A not allocated neighbor is assumed to have a value of zero (Rosu et al. 2020).

2.4.3 Selected Operations on Permutohedral Lattice

In this section, we will explain selected operations on permutohedral lattices that are necessary to understand the temporal correspondence of two lattices across timesteps. For all other operations we refer to Rosu et al. 2020.

Convolution: The convolution operation on lattices operates the same as standard spatial convolutions. The weighted sum of all neighbors and the center vertex is computed. Our convolutions are performed on the one-hop neighborhood of a vertex and therefore we convolve the values of $2(d+1)+1$ vertices (Fig. 2.8).

Distribute The points of the input cloud have to be distributed onto the lattice structure. For this we define the distribute operation as the feature vectors that each lattice vertex receives: The distribute operators $\mathcal{D}_G$ and $\mathcal{D}_F$ concatenate

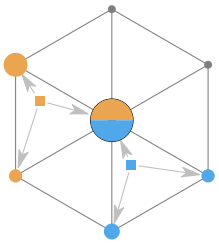


Figure 2.9: Distribute operation: Distributing stores all the features of the contributing points, causing no loss of information and allows further processing by the network. (Rosu et al. 2020)

coordinates and features of the contributing points:

$$\mathbf{x}_v = \mathcal{P}(\mathbf{D}_{v_g}; \mathbf{D}_{v_f}), \tag{2.12}$$

$$\mathbf{D}_{v_g} = \mathcal{D}_G(P, V) = \{\, \mathbf{g}_p - \mu_v \mid p \in J_v \,\}, \tag{2.13}$$

$$\mathbf{D}_{v_f} = \mathcal{D}_F(P, V) = \{\, \mathbf{f}_p \mid p \in J_v \,\}, \tag{2.14}$$

$$\mu_v = \frac{1}{|J_v|} \sum_{p \in J_v} \mathbf{g}_p, \tag{2.15}$$

where $\mathbf{D}_{v_g} \in \mathbb{R}^{|J_v| \times d}$ and $\mathbf{D}_{v_f} \in \mathbb{R}^{|J_v| \times f_d}$ are matrices containing the distributed coordinates and features, respectively, for the contributing points into a vertex v. The matrices are concatenated and processed by a PointNet $\mathcal{P}$ to obtain the final vertex value $\mathbf{x}_v$.

We distribute coordinates and point features in two different ways. The coordinates are centered, before they are distributed, while the point features are distributed as they are.

2.4.4 Temporal Correspondence

All points p of the input cloud P get distributed to their corresponding lattice vertices — each d-dimensional point gets distributed to $d + 1$ lattice vertices and each of them has coordinates $\mathbf{c}_v$ and values $\mathbf{x}_v$. For this data, Rosu et al. 2020 designed an implementation to save this information sparsely in memory: The coordinates $\mathbf{c}_v$ act as the key to be inserted into a hash map, while the values $\mathbf{x}_v$ are stored there. The hash map assigns each vertex a unique number that is used to index the entries vector. This procedure is visualized for a single point in Fig. 2.10.

If the corresponding value in the entries vector is -1 ("empty") we atomically increase the size of the keys $\mathbf{C}$ and values array $\mathbf{X}$ and insert the key $\mathbf{c}_v$ and the value $\mathbf{x}_v$ vectors at the new row that is located at the bottom of the respec-

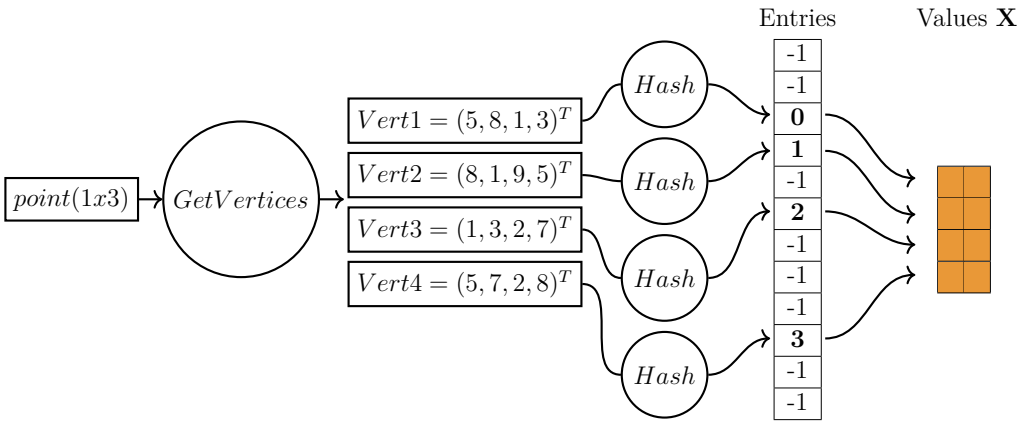


Figure 2.10: Lattice storage: The encircling four-dimensional simplex is calculated for the three-dimensional input point. The value and position of each encircling vertex is calculated and then hashed to generate the value in the entries array. The value in the entries vector is the corresponding row of the value matrix.

tive matrix. Otherwise the stored value vector is updated with the new values $\mathbf{x}_v$. Therefore, the values are stored sequentially in memory while the entries are sparsely populated.

Usually the hash map is reset after processing each cloud, because otherwise the entries vector would contain an ever increasing number of entries and $\mathbf{X}$ would therefore keep on growing with many rows that do not contain values. In contrast, for the sequence learning it is not reset after each cloud of the input, but after a whole sequence is processed. The reason for this is that the hash map can be utilized to map the coordinates of all vertices of each lattice of the sequence to the same row in the coordinate and value matrices $\mathbf{C}$ and $\mathbf{X}$. Therefore, the correspondence between vertices that are allocated at the same positions during different timesteps is given and we are able to compare the values of vertices across time. A reset hash map would most likely assign the same coordinates $\mathbf{c}_v$ to different rows in the values matrix.

2.5 Scene Flow

For the task of segmenting moving from non-moving objects scene flow could be utilized: ”Scene flow is the three-dimensional motion field of points in the world, just as optical flow is the two-dimensional motion field of points in an image.“ (Vedula et al. 1999).

For the prediction of scene flow mostly supervised learning is applied. A problem

of this type of learning is the need for large-scale labeled flow data. There are many synthetic datasets (Mayer et al. 2016) that provide this, but only a small amount of large-scale real-world datasets, because these are costly to create. One real-world dataset that provides such information is the KITTI dataset (Menze and Geiger 2015). Therefore, most state-of-the-art networks are trained on synthetic data and sometimes refined on small real-world datasets (Gu et al. 2019; Liu, Qi, and Guibas 2019; Mittal, Okorn, and Held 2020).

Scene flow could be incorporated in our network by either using the prediction as additional input data for the backbone network or by combining the flow prediction with the semantic segmentation from the backbone network to refine the segmentation, similar to recently proposed approaches which perform panoptic segmentation (Kirillov et al. 2019; Mohan and Valada 2020; Xiong et al. 2019). These networks often merge the prediction of semantic and instance segmentation for the final prediction.

Combining scene flow and semantic segmentation by using scene flow as an additional input was not pursued, since it would have constituted a combination of multiple networks into one pipeline rather than an extension to LatticeNet. Instead, the Abstract Flow (AFlow) module has been designed that aims at implementing an abstract flow in the lattice structure and is discussed later in this thesis (Sec. 4.3).

3 Related Work

In this chapter, temporal semantic segmentation in recent work will be discussed. An overview over the state-of-the-art for semantic segmentation as a whole will not be given, because it is a topic too broad for this thesis that only focuses on a certain sub-sector of it. For an overview regarding semantic segmentation on point clouds, we refer to Guo et al. 2020 and Rosu et al. 2020.

Many previously proposed approaches which address temporal semantic segmentation can be grouped by their type of input: methods which process sequences of clouds in a recurrent manner to predict the class labels (Duerr et al. 2020; Shi et al. 2020), and methods that accumulate multiple clouds into one single cloud to solve the task as a single-cloud segmentation (Behley et al. 2019; Thomas et al. 2019).

The advantage of processing a sequence of clouds is that for each input only the current cloud needs to be processed as the past information has already been summarized and stored in memory. However, they require more complex architectures than the methods that segment a large aggregated point cloud.

Shi et al. 2020 present their U-Net based architecture SpSequenceNet for temporal semantic segmentation. Two consecutive point clouds are voxelized and given as input to an encoder network. Connections are added between the two timesteps in order to gather temporal information before decoding the representation into class probabilities of the last point cloud. They designed two modules to combine the information from the two consecutive point clouds; the Cross-frame Global Attention (CGA) and the Cross-frame Local Interpolation (CLI) module. The CGA is based on attention networks and uses the data from P_{t-1} to focus the network on the important features of P_t by weighting all features from P_t based on the corresponding feature vectors in P_{t-1}. The CLI module fuses information between the point clouds of the sequence by combining the spatial and temporal information: For each point p in P_t the three-neighborhood of it in P_{t-1} is analyzed and combined to a new feature vector that is concatenated with the feature vector of the point in P_t and then passed through a residual layer. In contrast to our approach Shi et al. can only process sequences of length two and they voxelize the point cloud, which leads to a loss of information and discretization artifacts.

Duerr et al. 2020 present their recurrent architecture TemporalLidarSeg that

uses temporal memory alignment to predict the semantic labels of sequences of point clouds. Their sequences have the potential to be of unlimited length. They project the 3D point clouds onto the 2D plane and use a U-net backbone network to output per-frame feature matrices $\mathbf{F}^{(t)}$. These feature matrices are then combined with the feature matrices of the hidden state $\mathbf{H}^{(t-1)}$ using the temporal memory unit, which uses the real-world poses of each point cloud to compute the transformation from the coordinate system of $\mathbf{H}^{(t-1)}$ to $\mathbf{F}^{(t)}$. The 2D semantic segmentation is at last projected back into the 3D representation. Similarly to our approach, they require the poses of the point clouds, but they additionally need the mapping from 3D to 2D and therefore do not work directly on the point cloud. Additionally, our method uses multiple fusion points in contrast to a single fusion point in their approach.

Kernel Point Convolution (KPConv) (Thomas et al. 2019) operates directly on the point clouds by assigning convolution weights to a set of kernel points located in Euclidean space. Points in the vicinity of these kernel points are weighted and summed together to form feature vectors. The kernel function is defined as the correlation between the location of the kernel point and the distance to the points in the radius neighborhood. To be robust to varying densities, the input clouds are subsampled at every layer of the network using a grid subsampling and the convolutions use an adaptive neighborhood radius. The kernel points are usually static, but can also be learned by the network itself to adapt to more challenging tasks. Due to memory limitations, their approach cannot process a complete point cloud for outdoor scenes. To address the memory constraints, Thomas et al. fit multiple overlapping spheres into the point cloud and evaluate these. The final results are generated by a voting scheme. In contrast to our method, KPConv performs temporal semantic segmentation by accumulating all clouds of the sequence into one large point cloud and uses no recurrent architecture.

DarkNet53Seg (Behley et al. 2019) and TangentConv (Tatarchenko et al. 2018) were used as the two baseline networks for the segmentation of 4D point clouds in the SemanticKITTI (Behley et al. 2019) dataset. The input for these were accumulated clouds of the sequences. DarkNet53Seg (Behley et al. 2019) is an extension of SqueezeSeg (Wu et al. 2018) — a U-net with skip connections that uses the spherical projection of LiDAR point clouds to predict point-wise labels. These are subsequently refined by a conditional random field and clustering. TangentConv (Tatarchenko et al. 2018) is based on the notion of tangent convolution - a different approach to construct convolutional networks on surfaces that assumes the data is sampled from locally Euclidean surfaces. The points of the input are projected onto a tangent plane around them. These tangent images can then be used as 2D grids for convolutions. Based on this input, Tatarchenko et al. designed

a U-net with skip connections. In contrast to our approach, both DarkNet53Seg and TangentConv were developed to output dense per-point predictions for single point clouds and therefore contain no recurrent connection.

4 Architecture

In this chapter, we explain our extensions of LatticeNet. The differences in the input are highlighted (Sec. 4.1), the position of the recurrent connections is elaborated (Sec. 4.2), the fusion modules are presented (Sec. 4.3) and it is described how both are combined with the backbone network to form our RNNs (Sec. 4.4).

4.1 Method

The input to our network is a sequence of point clouds $P = (P_0, P_1, ..., P_{n-1})$, where $P_i = (\mathbf{G}, \mathbf{F})$ with $n \in \mathbb{N}^+$ and $0 \leq i < n$. The parameter n is also referred to as sequence length. The network outputs the likelihood for each possible class for every point $p \in P_{n-1}$. An overview is given in Fig. 4.1. All clouds P_i within the input sequence are transformed into a coordinate system which positions the point clouds relative to P_0.

Just like it was done for LatticeNet, the positions $\mathbf{G}$ are scaled by a scaling factor $\boldsymbol{\sigma} \in \mathbb{R}^d$ as $\mathbf{G}_s = \mathbf{G}/\boldsymbol{\sigma}$. If not otherwise stated, we refer to $\mathbf{G}_s$ as $\mathbf{G}$. The matrix $\mathbf{F}$ denoting the per-point features contains the reflectance value from the LiDAR scanner or is filled with zeros. In addition, the network is also able to support the input of accumulated point clouds. For this P is transformed to only contain a single point cloud P_{concat}. $\mathbf{G}$ and $\mathbf{F}$ of P_{concat} are the result of concatenating the position and feature matrices of the sequence $(P_0, P_1, ..., P_{n-1})$. The network then outputs a prediction for $P = (P_{concat})$.

The basic operations of LatticeNet can still be applied in our temporal setting, because the above definitions for a sequence of clouds P follow the same specifications as for LatticeNet.

We insert recurrent connections at various points of the LatticeNet architecture as illustrated in Fig. 4.2. At these newly introduced recurrent units, the states of two lattices $V^{(t-1)}$ and $V^{(t)}$ have to be fused. We refer to the value matrix of each lattice, $\mathbf{X}^{(t-1)}$ and $\mathbf{X}^{(t)}$ respectively, as state of the network. To compute the hidden state $\mathbf{H}^{(t)}$, we fuse the previous hidden state $\mathbf{H}^{(t-1)}$ together with the current state of the network $\mathbf{X}^{(t)}$. For this, a correspondence between the coordinate matrices $\mathbf{C}$ of both lattices has to be known. This correspondence is achieved by

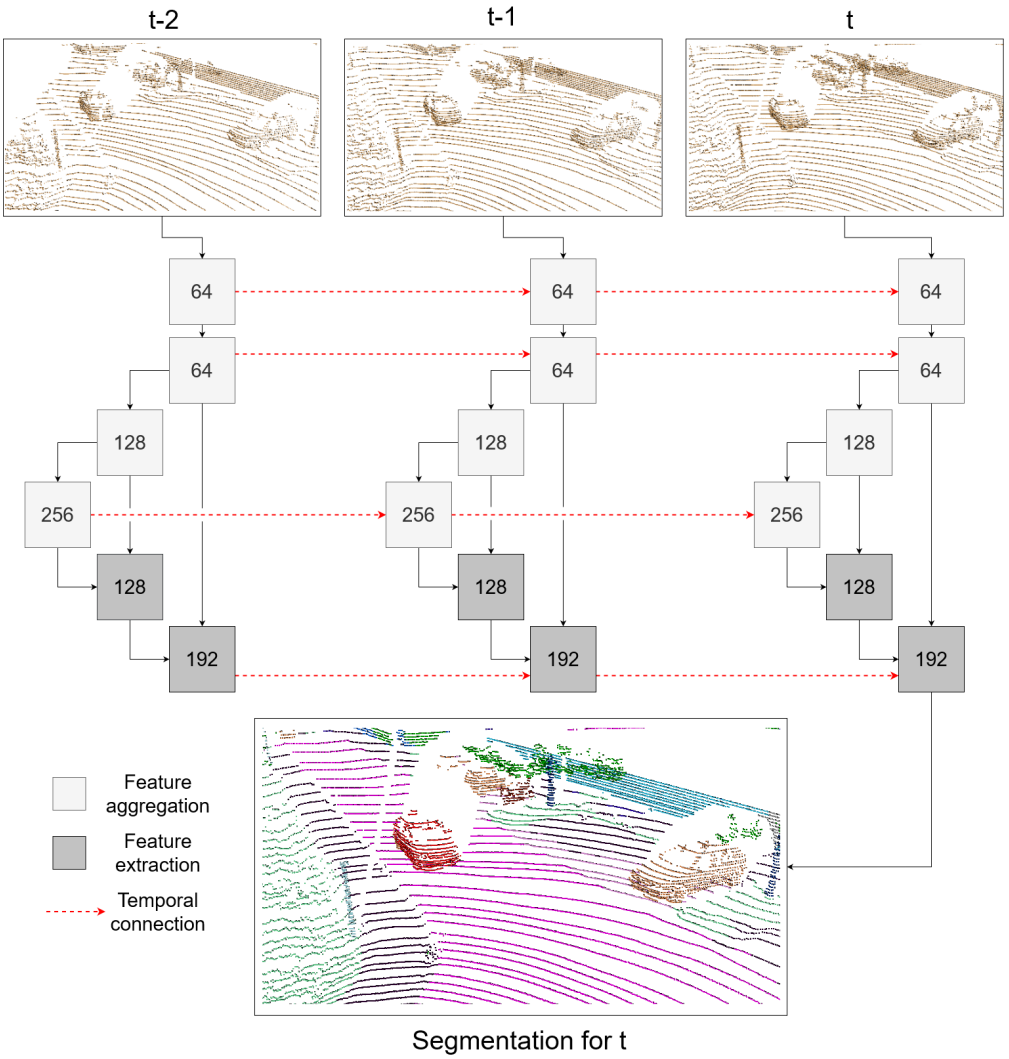


Figure 4.1: We use multiple consecutive point clouds with a common reference frame as the input to our backbone network. The value matrices of different timesteps are fused together to allow information propagation through time. The semantic class for each point in the last point cloud is predicted. The number in the squares correspond to the dimension of the value matrix in the recurrent network.

transforming the point clouds P_i into a common frame and the distribute operation of LatticeNet (Sec. 2.4.4). Vertices corresponding to previously unknown areas in the input are inserted at the end of the coordinate and value matrices $\mathbf{C}$ and $\mathbf{X}$. A toy example for this fusion procedure is given in Fig. 4.3.

4.2 Fusion Positions

Our RNN is a many-to-one deep RNN, with recurrent connections positioned at different depth of the network. We refer to the recurrent connections as fusion positions and propose a temporal extension to the LatticeNet with four fusion positions: An early fusion immediately after the PointNet, a middle fusion before the downsampling, a bottleneck fusion at the U-Net bottleneck layer and a late fusion after the upsampling. The extended/modified architecture is illustrated in Fig. 4.2. We chose these four promising positions based on the following observations:

Early Fusion The PointNet acts as an early feature descriptor and was therefore chosen as the earliest fusion position. It can influence the decisions of the network at early stages, but it suffers from a small receptive area.

Middle Fusion The middle fusion can be seen as a compromise between the concepts of early and late fusion. Compared to the early fusion more information has been aggregated, because of the ResNet-Layers in-between.

Bottleneck Fusion This fusion position is placed before the first upsampling and after the last downsampling. At this point the highest value dimension has been reached by the network and the lattice structure is the coarsest, therefore having the highest receptive area per lattice vertex.

Late Fusion Positioning the late fusion after the upsampling layer allows our network to retain fine grained lattice features, which are in a global context. The global scene understanding provided through upsampling enables classification of large objects like cars and buses. Since those objects have a high likelihood to be dynamic objects, a correct classification of them is important for any possible application. Note that late fusion alone is not sufficient, since the network has few layers left to combine these features with features extracted for the currently processed cloud.

4.3 Fusion Modules

In this section, multiple fusion modules are presented and discussed that can be used to fuse $\mathbf{H}^{(t-1)}$ and $\mathbf{X}^{(t)}$. For the last point cloud of the sequence, $\mathbf{H}^{(n-1)}$ is used to generate the networks output. For $t = 0$, no computation is performed with $\mathbf{H}^{(0)} = \mathbf{X}^{(0)}$.

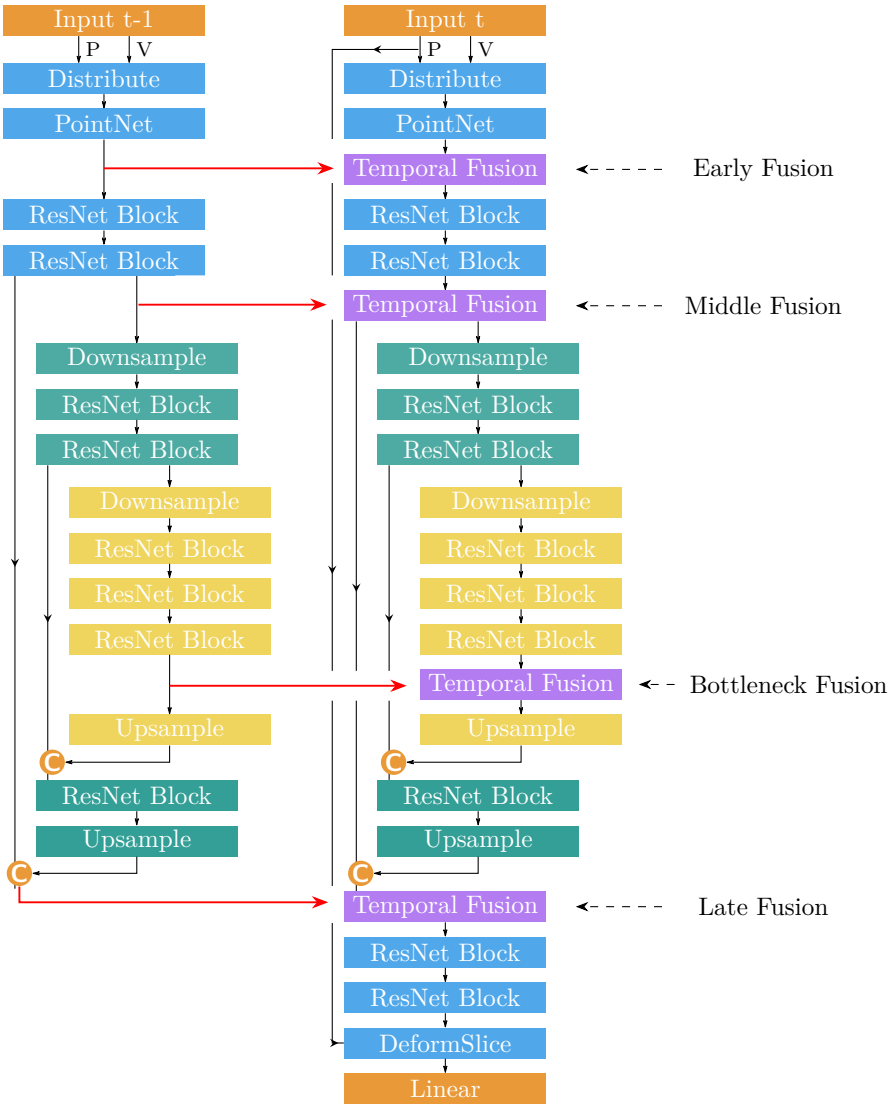


Figure 4.2: Recurrent architecture: The values from previous timesteps are fused in the current timestep at multiple levels of the network. This allows the network to distinguish moving objects from static ones. The architecture visualized in this figure is designed for a sequence length of two.

In order to fuse $\mathbf{H}^{(t-1)}$ and $\mathbf{X}^{(t)}$ they need the same shape and therefore $\mathbf{H}^{(t-1)}$ is padded. The padding value differs based on certain properties of the layers and we elaborate each of them individually. If not otherwise stated the padding value is 0. A simplified overview of this fusion process is presented in Fig. 4.3.

Linear Layer The matrices $\mathbf{H}^{(t-1)}$ and $\mathbf{X}^{(t)}$ are concatenated and input into a linear layer followed by a non-linearity, in our case a ReLU. This shallow temporal

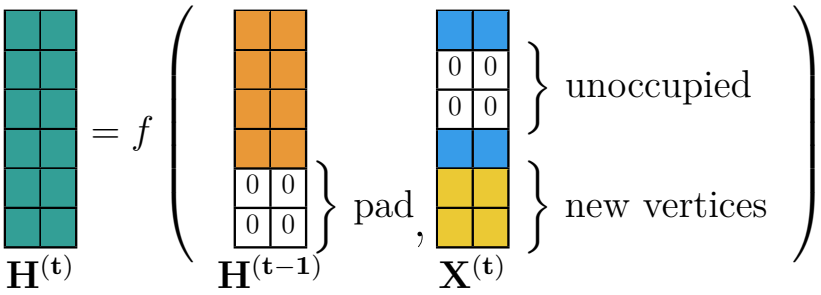


Figure 4.3: Temporal fusion: The values from the previous time-step $\mathbf{H}^{(t-1)}$ are zero-padded in order to account for the new vertices that were allocated at the current time-step $\mathbf{X}^{(t)}$. Lattice vertices that are not hit/occupied are assumed to have a value of zero. The value matrices are afterwards fused by the chosen fusion module.

connection is a basic multilayer perceptron (MLP).

MaxPool Layer The matrices $\mathbf{H}^{(t-1)}$ and $\mathbf{X}^{(t)}$ are stacked along a new axis and the element-wise maximum of both matrices is taken as the new value. $\mathbf{H}^{(t-1)}$ is padded with -9999, because the values of $\mathbf{X}^{(t)}$ can be negative. A maximum operator with a zero padding would otherwise discard some information. Additionally, the zero entries in $\mathbf{X}^{(t)}$, which are caused by no point hitting this specific lattice vertex, are changed to -9999 entries, because if not, similar to the first case, information would be deleted from $\mathbf{H}^{(t-1)}$.

Long short-term memory (LSTM) The functionality of LSTM cells is explained in Sec. 2.3.2. $\mathbf{H}^{(t-1)}$ and $\mathbf{X}^{(t)}$ are used as input to the respective gates of the network. For the first LSTM cell in the sequence both the cell state and the hidden state are initialized with a zero vector, because we have no information regarding $\mathbf{H}^{(-1)}$. We use the output $\mathbf{H}^{(t)}$ as our new value vector. The cell state is never explicitly used for further computations.

Gated recurrent units (GRU) GRUs are discussed in Sec. 2.3.2. Since they only use an input and a hidden gate and no cell state like LSTMs, they are better suited for our task. We will nonetheless use both LSTM and GRU, because we are interested in the comparison of their performance. Again, the output is used as the new value vector $\mathbf{H}^{(t)}$.

Cross-Frame Global Attention (CGA) Shi et al. 2020 designed the CGA module to guide the network's computation (Fig. 4.4). Inspired by self-attention mecha-

nisms, their approach uses $\mathbf{H}^{(t-1)}$ to compute an attention map for $\mathbf{X}^{(t)}$. $\mathbf{H}^{(t)}$ is computed by element-wise multiplication of the modified $\mathbf{H}^{(t-1)}$ with $\mathbf{X}^{(t)}$.

In the implementation in this thesis, $\mathbf{H}^{(t-1)}$ gets padded by zeros and then modified by the CGA module. After this modifications, all entries that were added by the zero padding get changed to one before $\mathbf{H}^{(t-1)}$ is multiplied with $\mathbf{X}^{(t)}$. The entries in $\mathbf{X}^{(t)}$ that now get multiplied by these padded entries are not changed in the CGA layer and are just passed through it.

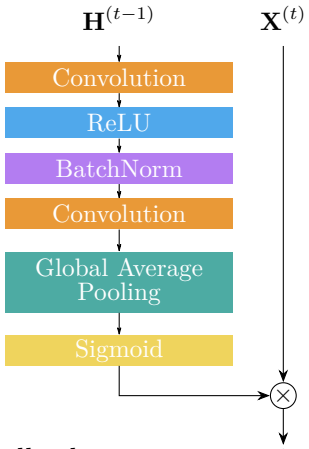


Figure 4.4: CGA Module with all relevant computational layers. Shi et al. state that it is based on self-attention and aims at directing the computation towards interesting areas of $\mathbf{X}^{(t)}$. In this figure $\times$ represents the element-wise multiplication.

Abstract Flow (AFlow) This module is inspired by the CLI module from SpSequenceNet (Shi et al. 2020) that aims to fuse local information and capture temporal information between two point clouds. Our AFlow module can be seen as a convolution with an adaptive convolution kernel. This resembles the ideas presented in Pixel-Adaptive Convolution (Su et al. 2019). Therefore, AFlow is designed to extract partial differences between $\mathbf{X}^{(t)}$ and $\mathbf{H}^{(t-1)}$. First, the nearest neighbors $N(v)$ of each lattice vertex v with $v \in V^{(t)}$ in the lattice from the previous timestep $V^{(t-1)}$ are identified (Fig. 4.5). They are used to generate a new local value vector $\mathbf{x}$ to fuse temporal information and at the same time to summarize the surrounding area in the previous timestep. The neighbors $N(v)$ of each lattice vertex are given by the one-hop neighborhood. The neighboring vectors from $V^{(t-1)}$ are denoted with $N_{\mathbf{H}}(v)$. For $d = 3$ the number of neighbors is given by $|N_{\mathbf{H}}(v)| = 8$. The value vectors of the vertices in $N_{\mathbf{H}}(v)$ are weighted according to their distance to the value vector $\mathbf{x}_v$ in $\mathbf{X}^{(t)}$. The weight is calculated

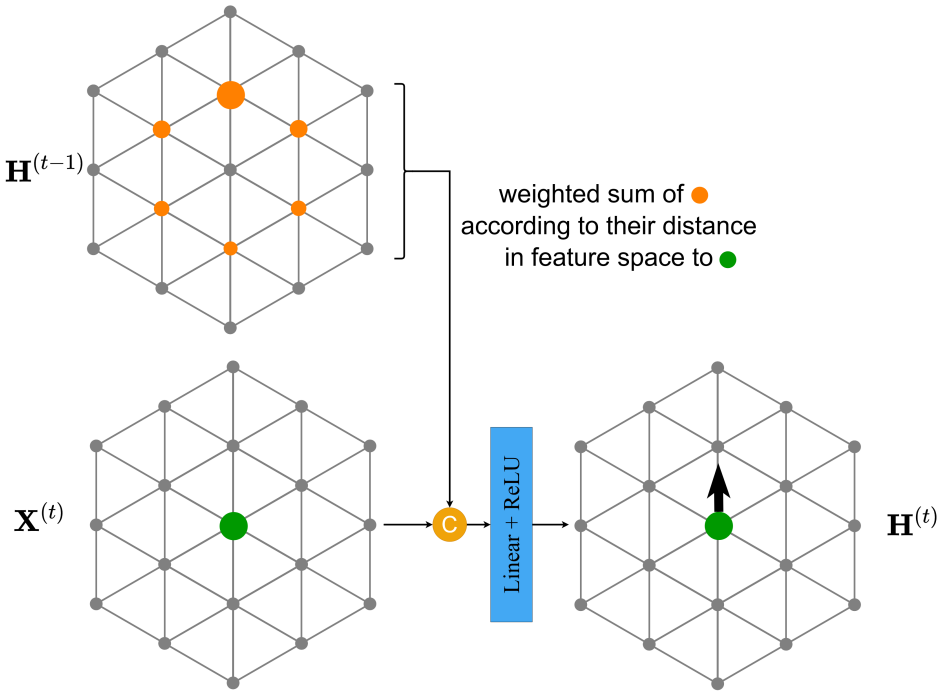


Figure 4.5: Abstract Flow module: Features from the one-hop neighborhood of the previous timestep $\mathbf{H}^{(t-1)}$ are compared with the center value $\mathbf{x}_v$ at the current time. A weighted sum is computed based on the distance and the result is concatenated with $\mathbf{x}_v$ and fused using a linear layer and a ReLU. A direction can also be established in lattice space between the center value and the most similar value from the previous timestep, yielding a coarse notion of the movement in the 3D scene. This direction is visualized with an arrow.

as

$$\forall i \in N_{\mathbf{H}}(v) : w_i = (\alpha - min(dist(\mathbf{x}_v, \mathbf{h}_i), \alpha)) \cdot \beta, \tag{4.1}$$

where $\mathbf{h}_i$ is the value vector of the i-th neighbor of $\mathbf{x}_v$ and α and β are learnable parameters that are initialized with $\alpha, \beta = 0.1$. The parameter α impacts the maximum distance a neighbor can have from the value vector we are evaluating at the moment, while β controls the maximum value of the resulting value vector. We denote with *dist* the Euclidean distance between the value vectors $\mathbf{x}_v$ and $\mathbf{h}_i$. The AFlow value vectors $\mathbf{l}_v$ of the matrix $\mathbf{L}$ are calculated as

$$\mathbf{l}_v = \sum_{i=1}^{8} \mathbf{h}_i \cdot w_i. \tag{4.2}$$

$\mathbf{L}$ is then concatenated with $\mathbf{X}^{(t)}$ and passed through a linear layer followed by a non-linearity to get the new value matrix $\mathbf{H}^{(t)}$.

The weights calculated in AFlow measure the similarity between values at different timesteps. Similar value vertices that move through time correspond to moving object in 3D space. In Fig. 4.5 we visualize the design of the module and the direction between the center value vertex and the most similar value vertex at the previous timestep. Further experiments with the directionality in 3D space are discussed in Sec. 5.4.

It is important to note that not all of the neighbors might be initialized due to the sparse initialization of the lattice. Therefore, all not allocated vertices are ignored in the calculation of AFlow.

4.4 Network Architectures

The fusion positions (Sec. 4.2) and modules (Sec. 4.3) can be combined arbitrarily.

To distinguish the different architectures the following notation will be used: The four fusion positions will be separated by a hyphen, e.g. GRU-GRU-AFlow-GRU refers to a network that has a GRU for the early, middle and late fusion and a AFlow module for the bottleneck fusion. If one of the fusion positions is not used a slash is inserted, e.g. GRU-/-/-/ only uses a GRU for the early fusion, while not utilizing the other fusion positions. The linear and the maxpool module are represented by LIN and MAX respectively in this definition. As previously described the network can use an accumulated cloud P_{concat} as its input. A network that uses P_{concat} is named ACCUM, but is just a LatticeNet that differs in its input type.

With six possible fusion modules at four different positions the number of possible architectures is quite big and has to be reduced by filtering out networks that have overt difficulties in representing temporal relationships or can be expected to perform similarly. Therefore, AFlow will only be used for the bottleneck and late fusion, because the early and middle fusion lack the value dimension to benefit from the similarity measure in the value space. Additionally, GRU and LSTM will not be mixed in a network, because they serve the same purpose. Furthermore, only networks that at least utilize three fusion positions, and always utilize the early and middle fusion, will be examined, because we predict that only using fewer than two fusion position results in deteriorating classification performance.

5 Evaluation

In this chapter, we discuss details regarding the implementation (Sec. 5.1), explain which dataset is used (Sec. 5.2) and analyze the quantitative (Sec. 5.3) and qualitative (Sec. 5.4) results.

5.1 Implementation

LatticeNet was implemented by Rosu et al. 2020 and all lattice operators are exposed to PyTorch (Paszke et al. 2017). All convolutions are pre-activated using a ReLU unit (He et al. 2016; Huang et al. 2017).

For the lattice scale a $\boldsymbol{\sigma}$ of 0.6 was used, because it is the lowest $\boldsymbol{\sigma}$ that can still fit into the GPU memory. Additionally, a coarse lattice with $\boldsymbol{\sigma} = 0.9$ was used for intermediate tests, that are further explained in Sec. 5.3.

The models were trained using the Adam optimizer (Kingma and Ba 2014; Loshchilov and Hutter 2017) with a learning rate of 0.001 and a weight decay of 10^{-4}. The learning rate was reduced by a cosine annealing scheduler (Loshchilov and Hutter 2016). The number of epochs between two restarts was chosen as three, because it proved effective.

5.2 Dataset

We use the SemanticKITTI (Behley et al. 2019) dataset. It provides 3D LiDAR-scans from real urban environments and semantic per-point annotations for moving and non-moving classes. It is based on the KITTI dataset (Geiger et al. 2013). The annotations are done for a total of 19 different classes in the single scan task and 25 different classes for the multiple scans task. The classes and corresponding colors are displayed in Fig. 5.5. The scans vary in size from 82K to 129K points with a total of 4.549 billion annotated points. Additionally to the x, y and z coordinates the reflectance values are given for each point. We process each scan entirely without any cropping. In total the dataset contains 43,552 scans, where the train split contains 19,130, the validation set 4071 and the test set 20,351 scans.

Generating Predictions The hyperparameters sequence length n and cloud scope s, with $s \in \mathbb{N}$, have to be chosen for the dataset. For SemanticKITTI $s = 3$ was chosen, which means that between clouds in the sequence P two clouds in the dataset are skipped. The sequence length n defines the cardinality of the input for the network and was chosen as $3 \leq n \leq 5$. It is our belief that $n < 3$ does not allow the network to aggregate enough information, while $n > 5$ leads to memory and time constraints in addition to the problem of vanishing/exploding gradients. We found that a sequence length of $n = 4$ worked best for our models.

Data augmentation The training data was augmented for the training to emphasize generalization. The methods used for this were random translations in the direction of the x and z axes, a random rotation around the y-axis, random mirroring of the x and/or z axis and artificial noise. The equivalent data augmentations were applied to all point clouds within the same sequence.

5.3 Quantitative Results

As described previously (Sec. 4.4) there exists a multitude of possible architectures. Therefore, their number has to be reduced to filter out the potentially best performing architectures. Additionally, the number of submissions to the SemanticKITTI competition website[1] is limited to ten tries to counteract learning by heart on the results.

The architectures will be evaluated initially based on the mean Intersection-over-Union (mIoU) of the validation set, while using a coarse lattice with a scaling factor $\boldsymbol{\sigma} = 0.9$ and a shallow network. Afterwards the best ones are chosen and their results on the test set with the fine lattice are analyzed. If not otherwise stated we will use the reflectance values of the input clouds P for the feature matrix $\mathbf{F}$. The validation set was chosen for the tests, because it is not known to the network during training and the ground truth segmentation is provided, and the choice of $\boldsymbol{\sigma} = 0.9$ with a shallow network based on time constraints. The inference time is a lot smaller in comparison to the fine lattice and allows faster testing. In previous tests, the performance of the coarse lattice with a shallow network proved as a good indicator for the performance of the fine lattice.

The resulting mIoU for some of the tested networks are reported in Tab. 5.1. Overall, the fusion modules LSTM, GRU and AFlow are present in the top performing network configuration. A combination of GRU and AFlow was able to reach the highest mIoU score. As expected using the AFlow module for the bottle-

[1] `https://competitions.codalab.org/competitions/20331`

Table 5.1: Intermediate tests: Results on the validation set of SemanticKITTI for different recurrent architectures with a scaling factor $\boldsymbol{\sigma} = 0.9$, a shallow network and reflectance values as per-point features.

Approach	LIN-LIN-/-LIN	GRU-GRU-/-GRU	LSTM-LSTM-/-LSTM	LIN-LIN-AFlow-LIN	MAX-MAX-AFlow-MAX	GRU-GRU-AFlow-GRU	GRU-GRU-AFlow-AFlow	LSTM-LSTM-AFlow-LSTM	CGA-CGA-AFlow-CGA	ACCUM
mIoU	42.5	43.2	42.7	41.9	42.2	**43.6**	42.6	43.2	41.8	40.4

Table 5.2: Results on the test set of SemanticKITTI for selected architectures. Overall, networks that utilize reflectance as input perform better.

Approach	mIoU	with reflectance
LSTM-LSTM-AFlow-LSTM	46.7	✓
GRU-GRU-AFlow-AFlow	46.9	✓
GRU-GRU-AFlow-GRU	**47.1**	✓
GRU-GRU-/-GRU	44.1	✓
GRU-GRU-AFlow-GRU	42.8	x
LatticeNet-MLP (Rosu et al. 2021)	45.2	x

neck fusion proved advantageous in comparison to omitting it. Still the difference between LSTM/GRU with and without AFlow is only 0.4/0.5. An explanation for this would be the distance in value space that is calculated by AFlow: The highest dimension of the value matrix in the shallow network is 128 in comparison to 256 in the full RNN network as depicted in Fig. 4.2. Additionally, we compare the fusion modules to the CGA module designed by Shi et al. 2020. It proved overall worse than the other fusion modules and was therefore not further investigated. The worst performing network is the ACCUM network, which was expected, since the network does not utilize the recurrent architecture.

The four best networks from the intermediate tests were retrained with the fine lattice and the full network. Furthermore, we wanted to analyze the impact of the reflectance as input to the network and due to this, we calculated the mIoU on the test set of SemanticKITTI for networks that were trained with and without reflectance values as their input. We report them in Tab. 5.2. LatticeNet-MLP (Rosu et al. 2021) corresponds to a LIN-LIN-/-LIN network that was trained

without reflectance values, but was named differently, since these results are already published. In comparison to the results on the validation set the AFlow module now makes a clear difference with an improved mIoU of 3.0 points compared to the base-network that only utilizes GRUs. Adding another AFlow layer at the late fusion instead of an GRU resulted in slightly worse results, which can be explained by the lower value matrix dimension (192 in comparison to 256 in the bottleneck) and therefore a worse comparability in the space of the value matrices. The LSTM-networks performed a bit worse, than their GRU counterparts. This was already predicted in their definitions in Sec. 4.3.

Without using the reflectance as input, the result for the GRU-GRU-AFlow-GRU model deteriorated significantly. The reason for this could be that the reflectance is a very useful feature for distinguishing similar value vectors and omitting it leads to inferior results. This applies to the other recurrent blocks as well, albeit not so much, because they rely on different update mechanisms, which is reflected by the good performance of LatticeNet-MLP in comparison to GRU-GRU-AFlow-GRU when omitting the reflectance values.

Finally, we need to compare our network's results to the state-of-the-art on SemanticKITTI: We chose the best performing network GRU-GRU-AFlow-GRU and LatticeNet-MLP. The IoU for 23 of the 25 classes are presented in Tab. 5.3[2]. We improved performance in relation to the already published architecture LatticeNet-MLP (Rosu et al. 2021). Our network's performance is comparable to TemporalLidarSeg (Duerr et al. 2020), but is outperformed by KPConv (Thomas et al. 2019) with a mIoU that is smaller by 4.1 points in comparison to KPConv.

Our networks beat the competition for the classes sidewalk, building, terrain, pole and moving-motorcyclist. Especially moving-motorcyclist is an important class for urban scenarios, since they are usually represented by only a fraction of the points of a car and are quite vulnerable traffic participants. The segmentation results for vegetation and traffic sign are equal to the previously best reported results for these classes.

It is important to note that KPConv (Thomas et al. 2019) cannot process the whole cloud due to memory constraints, but has to rely on fitting multiple spheres into the cloud to ensure that each point is tested multiple times. The final result is then determined by a voting scheme, in contrast to our approach that processes the whole cloud at once with a single prediction per point. TemporalLidarSeg (Duerr et al. 2020), on the other hand, relies on the spherical projection of the 3D cloud to perform 2D operations, while our approach is able to utilize the 3D cloud without any projection.

[2]We do not report the classes bicyclist and motorcyclist, because no points with these classes are part of the test set and therefore their IoU is always zero.

Table 5.3: State-of-the-art results on SemanticKITTI in comparison to our best performing network.[2]

Approach	mIoU	car	bicycle	motorcycle	truck	other-vehicle	person	road	parking	sidewalk	other-ground	building	fence	vegetation	trunk	terrain	pole	traffic sign	moving-car	moving-bicyclist	moving-person	moving-motorcyclist	moving-other-vehicle	moving-truck
TangentConv (Tatarchenko et al. 2018)	34.1	84.9	2.0	18.2	21.1	18.5	1.6	83.9	38.3	64.0	15.3	85.8	49.1	79.5	43.2	56.7	36.4	31.2	40.3	1.1	6.4	1.9	**30.1**	**42.2**
DarkNet53Seg (Behley et al. 2019)	41.6	84.1	30.4	32.9	20.2	20.7	7.5	91.6	**64.9**	75.3	**27.5**	85.2	56.5	78.4	50.7	64.8	38.1	53.3	61.5	14.1	15.2	0.2	28.9	37.8
SpSequenceNet (Shi et al. 2020)	43.1	88.5	24.0	26.2	29.2	22.7	6.3	90.1	57.6	73.9	27.1	91.2	**66.8**	84.0	66.0	65.7	50.8	48.7	53.2	41.2	26.2	36.2	2.3	0.1
KPConv (Thomas et al. 2019)	**51.2**	**93.7**	44.9	**47.2**	**42.5**	**38.6**	**21.6**	86.5	58.4	70.5	26.7	90.8	64.5	**84.6**	**70.3**	66.0	57.0	53.9	**69.4**	**67.4**	**67.5**	47.2	4.7	5.8
TemporalLidarSeg (Duerr et al. 2020)	47.0	92.1	**47.7**	40.9	39.2	35.0	14.4	**91.8**	59.6	**75.8**	23.2	89.8	63.8	82.3	62.5	64.7	52.6	**60.4**	68.2	42.8	40.4	12.9	12.4	2.1
LatticeNet-MLP (Rosu et al. 2021)	45.2	91.1	16.8	25.0	29.7	23.1	6.8	89.7	60.5	72.5	26.9	**91.9**	64.7	82.9	65.0	63.7	54.7	47.1	54.8	44.6	49.9	**64.3**	0.6	3.5
GRU-GRU-AFlow-GRU	47.1	91.6	35.4	36.1	26.9	23.0	9.4	91.5	59.3	75.3	**27.5**	89.6	65.3	**84.6**	66.7	**70.4**	**57.2**	**60.4**	59.7	41.7	9.4	48.8	5.9	0.0

Table 5.4: Average time used by the forward pass and the maximum memory used during training.

	SemanticKITTI	
	[ms]	[GB]
LSTM-LSTM-AFlow-LSTM	151	20
GRU-GRU-AFlow-GRU	154	20
GRU-GRU-AFlow-AFlow	159	22
GRU-GRU-/-GRU	140	18
KPConv (Thomas et al. 2019)	225	15
SpSequenceNet (Shi et al. 2020)	477	3

As an ablation study, we wanted to compare the performance of our architectures in comparison to the state-of-the-art (Tab. 5.4). The measurements were performed on a NVIDIA GeForce RTX 3090 and the inference time was measured on the validation set. Each AFlow module increases the inference time and memory consumption, caused by the high number of weights in the AFlow module and the distance calculation per vertex. We are able to segment the cloud faster than KPConv (Thomas et al. 2019), because we are able to reuse value matrices from previous segmentations due to our recurrent architecture. In addition, we are significantly faster than SpSequenceNet (Shi et al. 2020), which takes more than thrice as long as our best performing architecture.

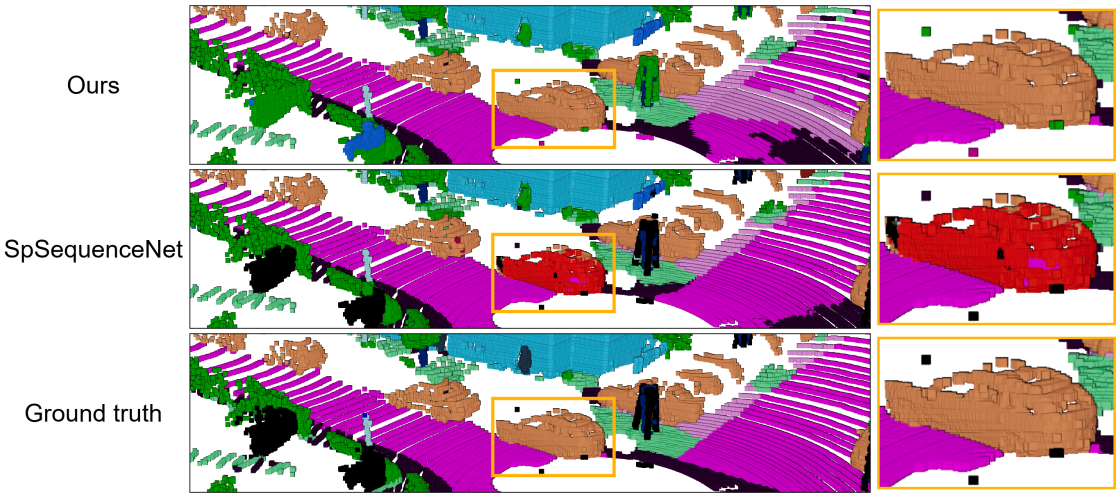


Figure 5.1: In comparison to SpSequenceNet we are able to better segment stationary ( ) and moving cars ( ) in small streets with a high number of cars in the vicinity. SpSequenceNet on the other hand is able to better distinguish between parking space ( ) and road ( ).

5.4 Qualitative Results

We recorded a video of the output of our best network for the validation set of SemanticKITTI. It can be found *here*[3]. As a comparison we provide the ground truth segmentation *here*[4]. The colormap for the qualitative results is presented in Tab. 5.5.

We provide frame-by-frame comparisons between our semantic segmentation results, the results of SpSequenceNet (Shi et al. 2020) and the ground truth in Fig. 5.1 and Fig. 5.2. This comparison uses point clouds from the validation set. We compare our proposed architecture to SpSequenceNet, because it is the best performing network that provides a working implementation with pre-trained models[5] for SemanticKITTI.

In order to analyze the effects of the AFlow model we mapped the directionality from lattice space to 3D space to obtain a coarse direction for the movement of the 3D objects within the scene. The coordinates of lattice vertices are approximated in 3D by the average of the points that contribute to them. In Fig. 5.3 we show one car at two different timesteps. For each relevant lattice vertex in 3D we draw an arrow that shows the most similar feature from the current timestep towards the previous one. We see that for the car driving towards the left, the directionality

[3] `https://uni-bonn.sciebo.de/s/sjYJ3HTXeUvdmZS`
[4] `https://uni-bonn.sciebo.de/s/rNvJrxEvGLojcMM`
[5] `https://github.com/dante0shy/SpSequenceNet`

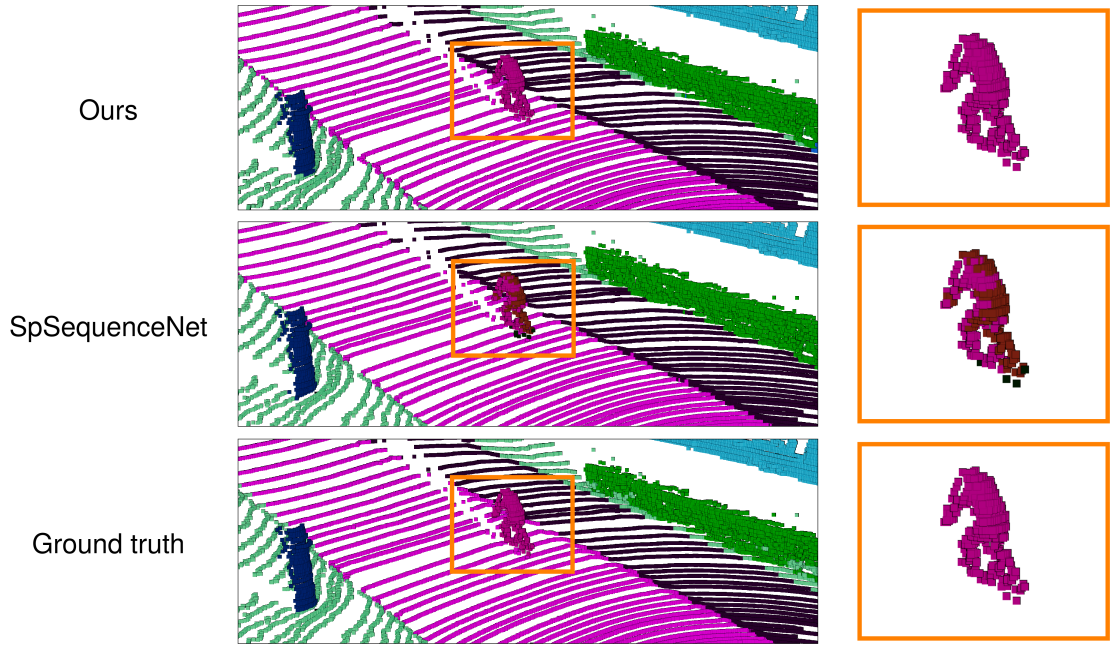


Figure 5.2: In comparison to SpSequenceNet we are able to better segment moving-bicyclists (■), a quite challenging class due to the small number of points per object.

from AFlow corresponds to the inverse of the driving direction and therefore, the module was able to extract the direction.

A failure case of our architecture are potentially moving objects, which are waiting/standing still for an amount of time that exceeds our temporal scope. In the SemanticKITTI dataset this applies for cars that are waiting at crossroads — a situation quite common in urban scenarios. Examples of this are presented in Fig. 5.4 and Fig. 5.5. This should not result in problems for an autonomous agent that takes actions based on this segmentation, because the object actually is standing still and is correctly classified as moving once it starts driving again. This leads to worse results on the IoU of the class moving-car (Tab. 5.3). A larger temporal scope might give the network the ability to track the object as moving. However, an important consideration is that after remaining still for a period of time a car might actually be parked and will not start moving again in the near future. To distinguish between parking and waiting the network has to learn the correspondence between road and car. A waiting car is on a road, while a parked car is on parking areas or close to sidewalks.

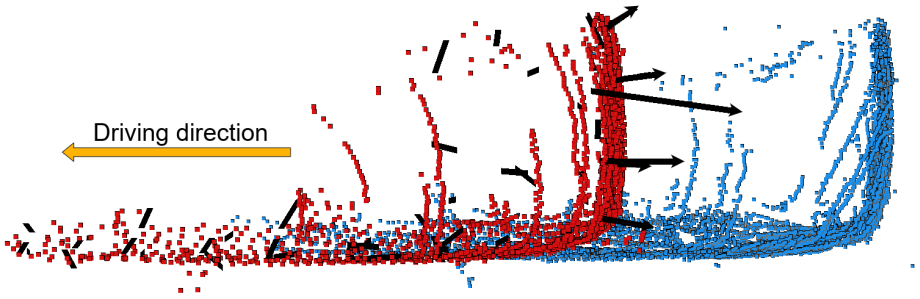
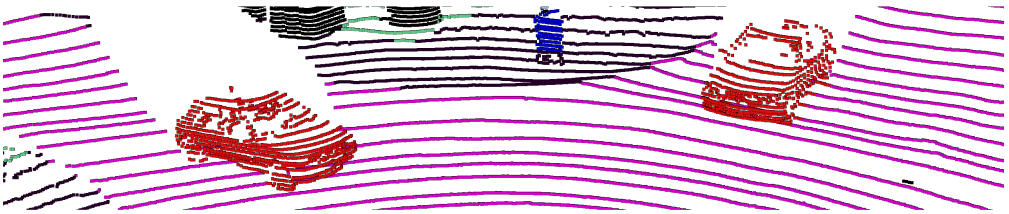
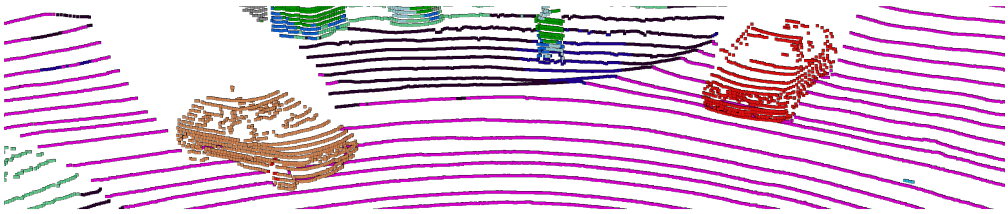


Figure 5.3: Visualization of the AFlow module on the segmentation: Birds-eye view of the same car at two different timesteps. The correspondence between the car in the previous timestep (■) and the current timestep (■) is made by the module and therefore the car is correctly segmented as moving-car (■).

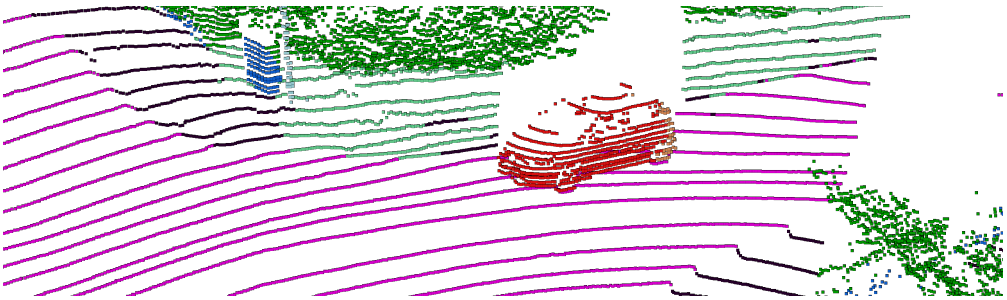


(a) Ground truth segmentation.

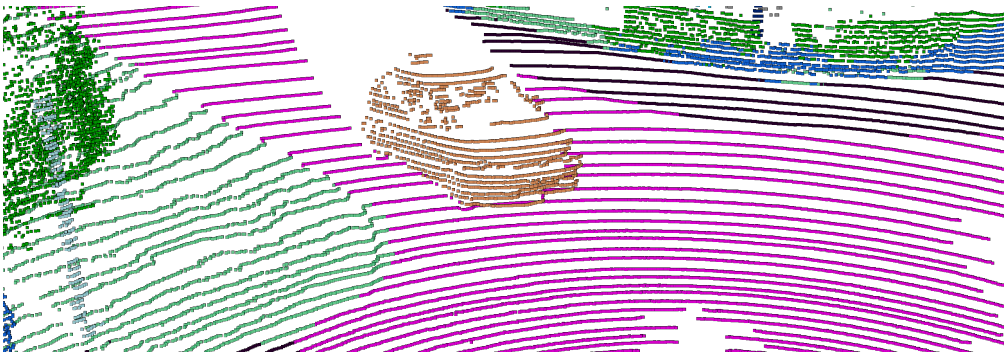


(b) Predicted segmentation.

Figure 5.4: Failure case: The prediction fails for the car on the left side, because it is predicted as car (■) instead of moving-car (■). The reason for this is that the car is waiting at the crossroads for many timesteps.



(a) Car is driving towards the junction.



(b) Car is waiting at the junction.

Figure 5.5: Failure case: While the car is driving towards the junction most of its points are segmented correctly as moving-car ( ). While it is waiting at the junction our network classifies it as stationary car ( ), because it did not move for an amount of time.

Table 5.5: The colormap used for the visualization of SemanticKITTI. They follow the colors proposed by the authors.

Class label	Color	Class label	Color
unlabeled		building	
car		fence	
bicycle		vegetation	
motorcycle		trunk	
truck		terrain	
other-vehicle		pole	
person		traffic-sign	
bicyclist		car (moving)	
motorcyclist		bicyclist (moving)	
road		person (moving)	
parking		motorcyclist (moving)	
sidewalk		truck (moving)	
other-ground		other-vehicle (moving)	

6 Conclusion

In this thesis we presented a novel extension to LatticeNet that is able to process sequences of point clouds as its input and can successfully segment moving from non-moving objects. We achieved competitive results on the SemanticKITTI dataset for the multiple scan task. While we are only the second best performing network w.r.t. mIoU, we are able to predict the classes for every cloud faster than the best approach KPConv.

To integrate temporal information, we successfully extended LatticeNet to a RNN. This is achieved by adding four fusion positions at different depths in the LatticeNet architecture, which can be arbitrarily combined with six fusion modules. These modules vary in complexity from a basic MLP to the novel AFlow module that uses neighboring vertices from the hidden state to extract the movement direction in the lattice space. Our best performing network uses a combination of GRU modules with an AFlow fusion module.

Our network is able to segment moving and non-moving classes. LatticeNet-MLP provides significantly better results on the class moving-motorcyclist than the competitors. Distinguishing moving from stationary objects is still a challenge for our network. We observed that our best-performing network is unable to model the correspondence between road and car, which inhibits it from distinguishing between waiting and parked cars.

In the future, we would like to test our temporal architectures on additional datasets that provide sequentially captured point clouds and ground truth segmentation for stationary and moving objects. With these we would like to further investigate which type of classes are problems for our network and how it can be changed to improve our performance on them. Motivated by CLI, our proposed AFlow module demonstrated its capability to capture temporal feature correspondences, which align well with the actual scene movement. Based on these observations, we propose that further research into recurrent networks, which explicitly utilize temporal fusion based on scene flow, could be appropriate.

List of Figures

List of Tables

Bibliography

Adams, Andrew, Jongmin Baek, and Myers Abraham Davis (2010). "Fast High-Dimensional Filtering Using the Permutohedral Lattice". In: *Computer Graphics Forum*. Vol. 29. 2. Wiley Online Library, pp. 753–762.

Behley, Jens, Martin Garbade, Andres Milioto, Jan Quenzel, Sven Behnke, Cyrill Stachniss, and Jürgen Gall (2019). "SemanticKITTI: A Dataset for Semantic Scene Understanding of LiDAR Sequences". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 9297–9307.

Cho, Kyunghyun, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio (2014). "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation". In: *Arxiv preprint arxiv:1406.1078*.

Chung, Junyoung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio (2014). "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling". In: *arXiv preprint arXiv:1412.3555*.

Duerr, Fabian, Mario Pfaller, Hendrik Weigel, and Jürgen Beyerer (2020). "LiDAR-based Recurrent 3D Semantic Segmentation with Temporal Memory Alignment". In: *Intl. Conf. on 3D Vision (3DV)*. IEEE, pp. 781–790.

Geiger, Andreas, Philip Lenz, Christoph Stiller, and Raquel Urtasun (2013). "Vision meets robotics: The KITTI dataset". In: *The International Journal of Robotics Research (IJRR)* 32.11, pp. 1231–1237.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep learning*. MIT press.

Gu, Xiuye, Yijie Wang, Chongruo Wu, Yong Jae Lee, and Panqu Wang (2019). "HPLFlowNet: Hierarchical Permutohedral Lattice FlowNet for Scene Flow Estimation on Large-scale Point Clouds". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3254–3263.

Guo, Yulan, Hanyun Wang, Qingyong Hu, Hao Liu, Li Liu, and Mohammed Bennamoun (2020). "Deep Learning for 3D Point Clouds: A Survey". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (2016). "Identity mappings in deep residual networks". In: *European Conference on Computer Vision (ECCV)*. Springer, pp. 630–645.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long Short-Term Memory". In: *Neural Computation* 9.8, pp. 1735–1780.

Huang, Gao, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger (2017). "Densely connected convolutional networks". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4700–4708.

Kingma, Diederik P and Jimmy Ba (2014). "Adam: A method for stochastic optimization". In: *Arxiv preprint arxiv:1412.6980*.

Kirillov, Alexander, Kaiming He, Ross Girshick, Carsten Rother, and Piotr Dollár (2019). "Panoptic segmentation". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 9404–9413.

Liu, Xingyu, Charles R Qi, and Leonidas J Guibas (2019). "FlowNet3D: Learning Scene Flow in 3D Point Clouds". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 529–537.

Loshchilov, Ilya and Frank Hutter (2016). "SGDR: Stochastic gradient descent with restarts". In: *Computing Research Repository (CoRR)* abs/1608.03983.

– (2017). "Fixing weight decay regularization in adam". In: *Computing Research Repository (CoRR)* abs/1711.05101.

Mayer, N., E. Ilg, P. Häusser, P. Fischer, D. Cremers, A. Dosovitskiy, and T. Brox (2016). "A Large Dataset to Train Convolutional Networks for Disparity, Optical Flow, and Scene Flow Estimation". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Menze, Moritz and Andreas Geiger (2015). "Object Scene Flow for Autonomous Vehicles". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Mittal, Himangi, Brian Okorn, and David Held (2020). "Just Go with the Flow: Self-Supervised Scene Flow Estimation". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 11177–11185.

Mohan, Rohit and Abhinav Valada (2020). "EfficientPS: Efficient Panoptic Segmentation". In: *International Journal of Computer Vision (IJCV)* 129, pp. 1551–1579.

Olah, Christopher (2015). *Understanding LSTM Networks*. `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`. Accessed: 2021-09-17.

Paszke, Adam, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer (2017). "Automatic Differentiation in PyTorch". In: *NIPS Autodiff Workshop*.

Rosu, Radu Alexandru, Peer Schütt, Jan Quenzel, and Sven Behnke (2020). "LatticeNet: Fast Point Cloud Segmentation Using Permutohedral Lattices". In: *Proceedings of Robotics: Science and Systems (RSS)*.

– (2021). "LatticeNet: Fast Spatio-Temporal Point Cloud Segmentation Using Permutohedral Lattices". In: *Autonomous Robots (AURO)*.

Rumelhart, David E, Geoffrey E Hinton, and Ronald J Williams (1986). "Learning representations by back-propagating errors". In: *Nature* 323.6088, pp. 533–536.

Shi, Hanyu, Guosheng Lin, Hao Wang, Tzu-Yi Hung, and Zhenhua Wang (2020). "SpSequenceNet: Semantic Segmentation Network on 4D Point Clouds". In:

IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 4574–4583.

Su, Hang, Varun Jampani, Deqing Sun, Orazio Gallo, Erik Learned-Miller, and Jan Kautz (2019). "Pixel-adaptive convolutional neural networks". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 11166–11175.

Tatarchenko, Maxim, Jaesik Park, Vladlen Koltun, and Qian-Yi Zhou (2018). "Tangent convolutions for dense prediction in 3D". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3887–3896.

Thomas, Hugues, Charles R. Qi, Jean-Emmanuel Deschaud, Beatriz Marcotegui, François Goulette, and Leonidas J. Guibas (2019). "KPConv: Flexible and Deformable Convolution for Point Clouds". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Vedula, Sundar, Simon Baker, Peter Rander, Robert Collins, and Takeo Kanade (1999). "Three-Dimensional Scene Flow". In: *Proceedings of the Seventh IEEE International Conference on Computer Vision*. Vol. 2. IEEE, pp. 722–729.

Wu, Bichen, Alvin Wan, Xiangyu Yue, and Kurt Keutzer (2018). "SqueezeSeg: Convolutional Neural Nets with Recurrent CRF for Real-Time Road-Object Segmentation from 3D LiDAR Point Cloud". In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 1887–1893.

Xiong, Yuwen, Renjie Liao, Hengshuang Zhao, Rui Hu, Min Bai, Ersin Yumer, and Raquel Urtasun (2019). "UPSNet: A Unified Panoptic Segmentation Network". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 8818–8826.