Segmentation, Classification and Tracking of Objects in LiDAR Point Cloud Data Using Deep Learning

Segmentering, klassificering och följning av objekt från LiDAR-data med djupinlärning

Robin Bernståle, Hjalmar Lind

Master's thesis
2022:E1

Lund University

Faculty of Engineering
Centre for Mathematical Sciences
Mathematics

Centrum Scientiarum Mathematicarum

Abstract

The purpose of this thesis was to explore deep learning methods of segmentation, classification and tracking of objects in LiDAR data. To do this a complete pipeline was developed, consisting of background filtering, clustering, tracking, labeling and visualization. The objects that were focused on were pedestrians, cyclists, cars and animals, in different environments. Background segmentation and object detection was done using classical methods, using distance filtering and DBSCAN for clustering. Four deep neural networks were trained for object classification and two for semantic segmentation, with different parameters to compare performance. To process point clouds generated by a LiDAR, a specialized architecture was needed, which is why PointNet layers were used to build the models. Tracking was managed by a recurrent neural network structure, capable of predicting object trajectory, updating measurements and data association. An additional classical tracking algorithm, was also developed as a baseline for comparison. The networks were trained purely on simulated data from an autonomous driving simulator, with the aim of also functioning on real world data. The models were compared and evaluated on both simulated data and real world LiDAR data. The results showed that classification using PointNet as foundation works well, even on real world data, being able to accurately classify both humans and vehicles. Semantic segmentation proved not to be suitable for the task, lacking in performance. The deep learning tracker showed great potential, but was difficult to properly train to outperform the classical tracker.

Acknowledgements

We would like to thank our supervisor Anders Heyden for his continuous feedback and help along the way. We would also like to thank Emma Engdahl and Erik Lundell for their help with the Carla simulator, with an extra thank you to Erik for his weekly extra support during the project.

Contents

1 Introduction

LiDAR, which stands for light detection and ranging, is a method of measuring distances by bouncing light pulses on the surroundings, to generate a 3D view in the form of a point cloud. They are used in cars and robots that need to interact with their environment. Considering for example that a LiDAR works just as well when it is dark outside as during the day, or that is capable of creating a point cloud with higher resolution than a common radar, there are certainly benefits in for example the surveillance sector. Analogue to this we specifically want to examine data from a stationary LiDAR, much alike how a security camera would work. This is in great contrast to a big field of LiDAR research, where the application is towards self-driving cars – where the LiDAR as such is always moving. In our specific research area we want to specifically be able to detect, track and classify moving objects, while avoiding any false alarms.

Detection refers to finding interesting objects in the scene, such as people, animals and vehicles. For LiDAR data in particular, we need to identify which points belong to each object, and separate them from the background. We then classify each object, which means giving them a label such as "human", "car" or "animal". A classifier can work in many different ways, but typically some features are extracted from the objects, that can be used to determine its type. Finally the objects are tracked over time in the LiDAR data, which means keeping track of which object is which over the course of a recording.

This task has been researched by previous master thesis students, most recently by Berntsson and Winberg [1]. They focused a lot on classical implementations, with some elements of deep learning, achieving some very good results. We hope that using deep learning to a greater extent will create an even better, more robust and more sustainable pipeline.

1.1 Purpose

The purpose of this project is to create an online deep learning-based pipeline for detecting, tracking and classifying objects within a LiDAR-recording. This project will mainly focus on tracking and classifying objects, while a basic detection algorithm will also be implemented due to its necessity for the latter steps. The deep learning-based tracking and classification will in turn be evaluated against common metrics and the tracking in particular will be compared to a baseline of Kalman filtering and the Hungarian Algorithm.

1.2 Limitations

As stated above, the detection steps will be left at bare minimum. In this case, a basic background filtering and clustering algorithm will be implemented but not evaluated or compared to other methods. Similarly, there are probably ways in which detection, tracking and classification can benefit each other to create a more dynamic pipeline instead of going step by step. This will also not be examined any further in this study.

1.3 Statement of contribution

During this master's thesis project, the work has been divided such that Hjalmar Lind has mainly worked with developing and evaluating the methods for classification and Robin Bernståhle has mainly worked with developing and evaluating the methods for tracking. The final results has then been discussed and agreed upon by both authors, which together developed the final pipeline.

2 Background

2.1 LiDAR Sensor

A light detection and ranging (LiDAR) device is an active sensor that emits light that bounces on surrounding objects, which then gets detected by the sensor. The travel time of the light is used to measure the distance to the target object. Being an active sensor, meaning it uses its own light source, enables the LiDAR to work regardless of the environmental light. This makes it less prone to factors such as time of day or weather, compared to something like a traditional camera. There are many types of LiDAR sensors for different applications and from various manufacturers. Some use a rotating sensor, mechanical mirrors or small vibrations to move the laser. The LiDAR sensor our algorithms are supposed to run with is Hydra, by Luminar, see [2].

2.1.1 The Luminar Hydra

Hydra is a LiDAR sensor designed to be mounted on cars, but can just as easily be used in other applications. It uses has two sensors, or "eyes", which operate separately and move the laser using microelectromechanical systems. Each sensor emits a laser pulse, waits for all the bounces to return, and then shifts its firing angle. This is done hundreds of thousands of times per second in different directions to generate a 3D view of the environment, in the form of a so called point cloud. Each point in a point cloud has an x-,y- and z-coordinate as well as the intensity of the returned light pulse. The sensor specifications provided by Luminar, see [3], are in Table 2.1.

Range at <10 % reflectivity	250 m
Max range	500 m
Frames per second	1 - 30
Lines scanned per second	640
Range precision	0.01 m
Max returns per point	3
Horizontal field of view	120°
Min horizontal resolution	0.07°
Vertical field of view	0 - 30°
Min vertical resolution	0.03°
Wavelength	1550 nm

Table 2.1: Luminar Hydra specifications.

2.2 Detection, tracking and classification

In Figure 2.1 is a diagram of the general pipeline when processing the LiDAR point cloud data in order to receive individual objects which have both tracks and labels (”human”, ”car”, ”animal” etc.) assigned to them in each frame of time. Note in particular how the tracking module operates in both the current timestamp and in a temporal manner, transferring its knowledge (”state”) from one timestamp to the next. While there exist different approaches in each respective module, with this thesis we aim to solve the tracking and classification steps using deep learning.

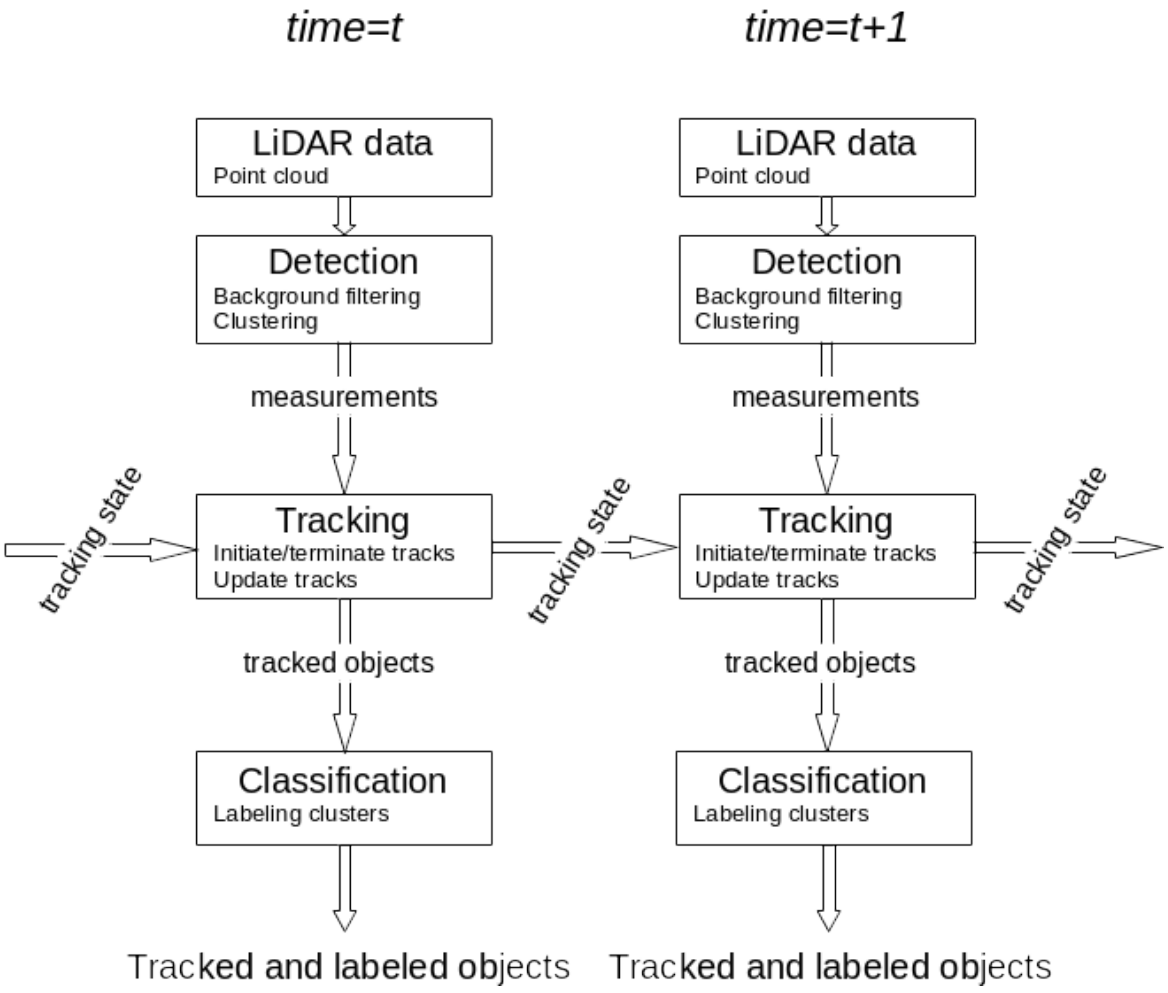
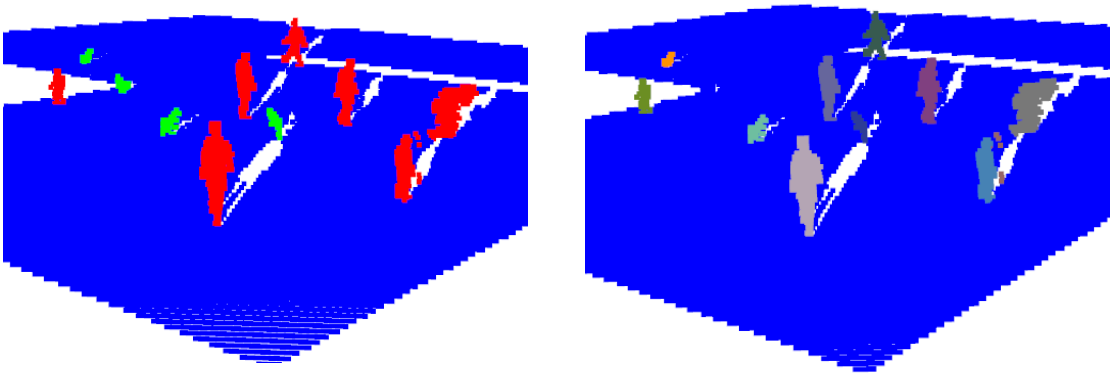


Figure 2.1: The general pipeline of detection, tracking and classification in LiDAR point cloud data over time.

3 Theory

3.1 Classical Instance Segmentation

Segmentation of image points is the process of partitioning the points into multiple segments, or objects. It is usually further divided into *semantic segmentation*, see Figure 3.1a and *instance segmentation*, see Figure 3.1b, which are separate problems with their own challenges. Semantic segmentation is the act of grouping together points which belong to the same type of object, or class, while the goal of instance segmentation is to group together points belonging to each individual object. Since our objective is to identify people, animals and vehicles, instance segmentation or some form of object detection is required somewhere along the way. After an object has been detected either ordinary classification or semantic segmentation can be used to determine which class it belongs to.



(a) Semantic segmentation in simulated LiDAR data, each type of object is assigned one color.

(b) Instance segmentation in simulated LiDAR data, each individual object is assigned one color.

Figure 3.1: The two types of segmentation.

There are many ways to do instance segmentation, but since it is not the primary focus of this thesis, a rather simple classical approach was taken. Below follows a two-step approach to classical instance segmentation, which is used to find individual objects of interest in the LiDAR data.

3.1.1 Filtering the Background

The points in a point cloud can be split into two different categories, *foreground* and *background*. The foreground consists of points belonging to objects we want to classify and track, such as humans, vehicles and animals. The background on the contrary consists of static objects such as buildings, ground or trees. There are many ways that the filtering of background can be done, below is a description of the way it is done in this project.

Max-Distance Filtering

A very simple and rather effective method, used by for example Xiao et al. [4], is a maximum distance filter. An empty scene is scanned for a few seconds, and taken as the static environment, i.e. the background. After that, anything moving in front of the background can be seen as dynamic points, and are kept after the filtering. The background is represented by a fixed maximum distance for each direction, and anything beyond it is filtered away. This method has a few obvious flaws. The background is not always entirely static; wind can shake trees and flags for example. The scene needs to be empty for a few seconds, this is not always easy to accommodate. Finally, if the LiDAR sensor moves or vibrates the whole scene shakes and the filtering deteriorates.

3.1.2 Clustering Objects

When the foreground has been extracted from the original point cloud we are left with the objects we want to identify, as well as some noise from the background. The goal of instance segmentation is to find all points belonging to the same object and group them together, to prepare them for tracking and classification. Grouping data points together this way is known as *clustering*, and is a well-studied problem with many applications. That said, there is no clustering algorithm that works in every scenario, different algorithms work well for different problems. Knowing how the structure of the data and the applications is the key to picking the right clustering algorithm, and the using it appropriately with the right parameters. We can list a set of requirements that our clustering algorithm needs to adhere to:

- **Work for an unknown number of clusters**, there is no way of telling how many objects there are in the data ahead of time.
- **Robust to outliers**, there will be noise from the background which should not belong to any cluster.
- **Work for any geometry and cluster size**, objects will have different shapes and sizes.
- **Be fast**, not much time can be put into the pre-processing of the point cloud, since it is only a small part of the pipeline.
- **Work for large cluster sizes**, usually thousands of points need to be clustered at once.
- **Euclidean distance metric**, what separates objects from one another is the physical distance.

One particular algorithm stands out from the rest when it comes to these criteria, one that has passed the test of time, density-based spatial clustering of applications with noise (DBSCAN), see [5]. It is a density-based clustering algorithm, separating high-density clusters from low-density areas. This makes it so that DBSCAN works on

clusters of any shape, with the biggest drawback being that the clusters need similar density.

DBSCAN works by creating clusters out of *core samples*. A core sample is defined as a point having at least *min_samples* within distance ϵ of itself. If a core sample is within distance ϵ to another core sample, they belong to the same cluster. If a point is not a core sample, but within distance ϵ of a core sample, it belongs to the same cluster as the core sample. Any other points, that are not core samples, or are not within distance ϵ to a core sample, are outliers.

This means that DBSCAN has two parameters that need be chosen, *min_samples* and ϵ. The most critical parameter is ϵ, as it is entirely dependant on the density of the data. If ϵ is high we allow more clusters to form in low-density areas, and several smaller clusters can merge into larger ones. On the other hand if ϵ is low we require higher density for cluster formation. Choosing ϵ correctly is not trivial, and there is likely now value which will cluster all points correctly, but we will later see how it can be calculated. The other parameter, *min_samples*, controls how many samples are needed to form a cluster, which can be useful in noisy data. By choosing *min_samples* right we can ensure that noise points become outliers, which don't interfere with the classification and tracking.

3.2 Deep Feedforward Networks

An *artificial neural network* (ANN) is a system of functions, loosely based on neurons in the biological brain, see [6]. The neurons, or nodes, in an ANN have connections to each other, and are typically organized in layers. The deep feedforward network is a type of ANN where connections between nodes do not form a cycle, contrary to connections in a recurrent neural network, which will be described later. These deep feedforward networks can be used to classify images, or do semantic segmentation in images, with the help of convolutional layers. One of the simplest feedforward networks is the multilayer perceptron (MLP), which will be described below to illustrate how a neural network operates.

3.2.1 The Multilayer Perceptron

The MLP has three or more layers, consisting of the input layer, the output layer and one or more hidden layers, see Figure 3.2. Each layer consists of a number of nodes, and each node is connected by a weight to all nodes in the next layer. The input layer simply takes an input vector of some predetermined length and sends the content of each index to every node in the first hidden layer, which is where it gets interesting.

At each node in a hidden layer the values from the previous layer are multiplied by their respective connection weights and summed up. Before being sent off to the next layer, a bias is added, and the sum is sent through a non-linear activation function. For node i in a hidden layer this would correspond to Equation 3.1, where m is the number of nodes in the previous layer, ϕ is the activation function, w_i are the weights,

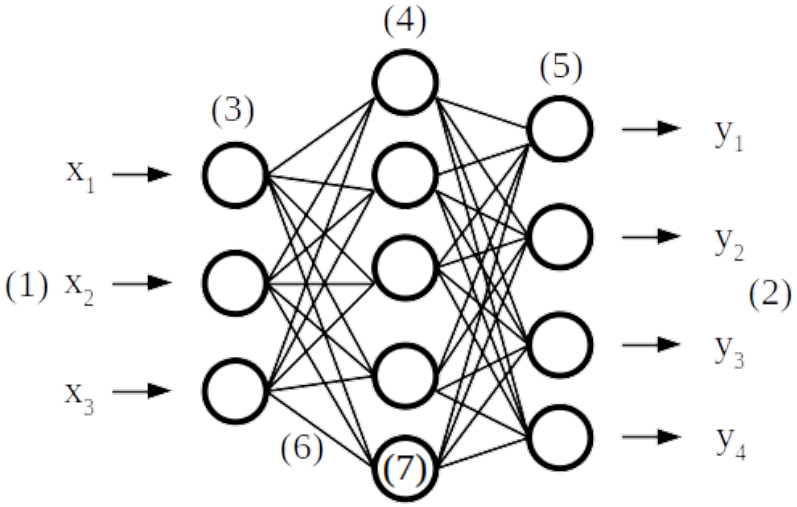


Figure 3.2: A multilayer perceptron. (1) Input. (2) Output. (3) Input layer. (4) Hidden layer. (5) Output layer. (6) Node connections. (7) Nodes.

b_i is the bias and x are the values from the previous layer.

$$z_i = \phi\Big(\sum_{j=1}^{m} w_{ij}x_j + b_i\Big) \tag{3.1}$$

There are several non-linear activation functions that can be used, but a common one is the rectified linear unit, ReLU, see Equation 3.2.

$$\phi(x_i) = \max(x_i, 0) \tag{3.2}$$

If a classifier is what you want, which is a common use for an MLP, the final layer, the output layer, should have as many nodes as types of objects you want to be able to detect. For example, if the network is supposed to identify pedestrians, cars, cyclists and animals, you would want four output nodes. To determine the output as a probability for each class, you would apply the softmax function, see Equation 3.3, on the output of each node i, where m is the number of output nodes.

$$\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^{m} \exp(z_j)} \tag{3.3}$$

This sums up how a type of feedforward network can take an input vector, pass it through its layers, and output a probability. However for the output to make any sense, the network needs to be *trained* to work like we want it to.

3.2.2 Deep Network Training

An important property for deep networks is that they are trainable. If a network is supposed to learn the difference between humans and cars, we need to pass hundreds of images containing these objects through the network, and the network should by itself figure out the differences. The network we just looked at can have tens of thousands of trainable weights and biases, so there needs to be a way for the network to know how these should be tuned for a good result. This is where cost functions and gradient-based learning is introduced.

A cost function is simply a function that outputs a number based on what the network predicts compared to what the correct answer is. If the network predicts correctly it gives a low cost, but if it is wrong it gives a high cost. Typically the cross entropy between the predicted probabilities and the true probabilities is used for classification problems for example.

The gradient based learning is then responsible for tuning the parameters responsible for producing the cost, which results in higher tuning for incorrect predictions. This is done through an algorithm called *backpropagation*. Backpropagation computes the gradient of each trainable parameter with respect to the cost. This way if we want the cost to go down, which we do, we know how the weights and biases should be tuned.

3.3 Deep Learning on Point Clouds

We will now look into how a network architecture can be designed to work on LiDAR data, instead of ordinary images. The output from a LiDAR sensor is a set of points in $\mathbb{R}^N$ known as a *point cloud*, which has certain geometric properties. Unlike pixels in a 2D image, point clouds are highly irregular, having varying amounts of points, no order among points and with different shapes and sizes. This leads to issues when using typical convolutional neural networks, which require highly regular input shapes.

To operate on point clouds some researchers transform them to other, easier to work with, data structures. Common approaches are turning the points into 3D voxels or sets of 2D images taken from different views. However, this leads to a trade-off between accurately representing the data and the size of the data. For instance, using a very fine grained 3D voxel grid to represent a point cloud leads to a huge matrix representation, while still requiring some compression of the data. On the other hand using a more coarse grid leads to a smaller representation, but with a significant loss of information.

3.3.1 PointNet Architecture

Since it is not desired to convert the points to a different data form, a deep neural network intended to work on point clouds needs to adhere to their properties. This is precisely what the creators of PointNet, Qi et al. [7], do when creating the architecture of their network, see Figure 3.3. They list the main properties of point clouds as follows:

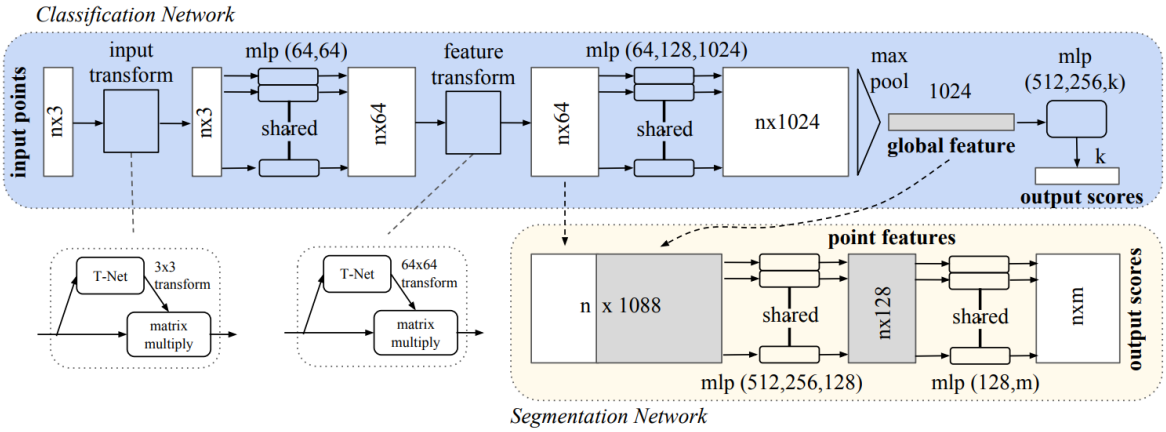


Figure 3.3: The architecture of PointNet. The top half is the classification network, and the bottom half is added if segmentation is desired instead. MLP stands for multilayer perceptron and the numbers within the parenthesis are layer sizes. Image from the paper on PointNet by Qi et al. [7].

- Point clouds are unordered, and any network that takes N points as input needs to be invariant to the $N!$ number of permutations of said input

To handle different input permutations PointNet uses a symmetrical function to aggregate the information from each point. The output from a symmetrical function is invariant to the input order of the arguments, i.e. $f(x_1, x_2) = f(x_2, x_1)$. Here, the symmetrical function of choice is the max pooling operator.

It works by stacking the feature vectors to form a matrix, where each feature vector is a row, then taking the maximum value of each column. This outputs a vector, which supposedly holds all necessary information about the point cloud. This means the input is an $n \times m$ matrix, and after the max pooling, a vector of length m is output, see Equation 3.4.

$$\begin{aligned} &\text{Column-wise max}\left(\mathbf{X} = (x_{ij}) \in \mathbb{R}^{n \times m}\right) = \\ &(\max(x_{11}, ..., x_{n1}), ..., \max(x_{m1}, ..., x_{mn})) \in \mathbb{R}^m \end{aligned} \tag{3.4}$$

Since the order of the points may not matter before the pooling, each point has to go through the same transformations up until the pooling. This is why it is crucial that each MLP has shared weights between each channel, and that the input and feature transformation work the same way for each point.

- Points are never isolated, close neighbours form an important subset. As such, the model needs to capture local structures, and interactions between structures

The global feature vector contains information about all points, and can be used with a classifier to generate the output classification scores. However the segmentation network needs to be aware of the surroundings of each individual point to label them,

as points are not isolated. This is solved in PointNet by appending the global feature vector to each individual feature vector, creating a matrix where each row contains both local and global information about the point. With this information a classifier can be trained to do semantic segmentation of point clouds as well.

- A point cloud representation should be invariant to rigid transformations, which includes rotations, translations and reflections

This task is solved in PointNet by applying an affine transformation to the input coordinates. The transformation matrix is dependent on the input, and is predicted by a small network called T-net. The transformation network works like a mini-PointNet, with point independent feature transformation, max pooling for feature aggregation and finally fully connected layers that predict the 3×3 matrix. The same principle is applied during the feature transformation, except the output is a larger matrix.

Since point clouds generally have a varying number of points, and a neural networks generally cannot be set to have an arbitrary input size, there seems to arise a problem. However, if we want to input less than n points to the network, the input can simply be padded with copies of itself. Since every point goes through the same transformations, two identical input points will still be identical at the max pooling operation, at which point any copies will have no effect on the outcome. For this reason n should be seen as the maximum number of expected input points, as with any more points than n, down-sampling is required.

3.3.2 PointNet++ Architecture

Less than a year after the making of PointNet, the creators had improvements to make on the point cloud processing pipeline, creating PointNet++, see [8]. Although PointNet is a pioneer in deep learning on point sets, being able to encode the features of a point cloud in a single fixed sized vector, it had some issues. Particularly with incorporating features from many different scales in the same point cloud, and working with point clouds with varying density.

The problem of looking at features at different scales has already been solved in deep learning on regular images, by the CNN. In a CNN there are one or more convolutional layers, which aim to capture local features by having a kernel slide over the image, working with a few grouped up pixels at a time. Since the input to a CNN is always the same shape and pixels are always equally spaced, this is not a hard task, it is simply matrix multiplication in different positions in the image.

The PointNet++ Layer

PointNet++ aims to work in a similar way, by processing the point cloud in a hierarchical fashion to aggregate information at different scales, but faces two obstacles: how to split the point set into smaller, overlapping parts at each level, and how to extract

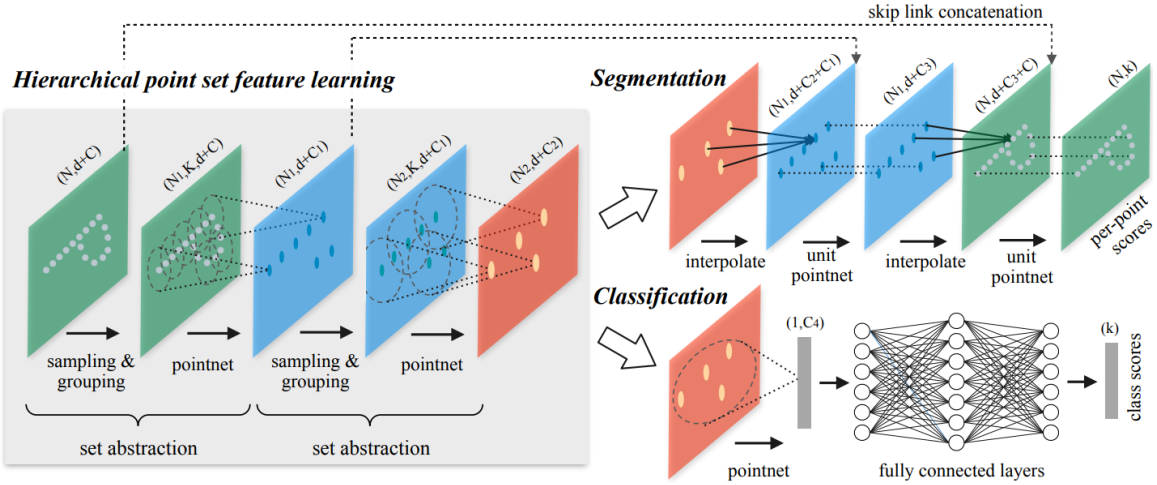


Figure 3.4: The architecture of PointNet++. One PointNet++ step consists of sampling, grouping and a small PointNet. Either segmentation and classification can be achieved, with different architectures. Image from the paper on PointNet++ by Qi et al. [8].

local features. A summary of the PointNet++ architecture is found in Figure 3.4. The solution the creators propose is finding key points in the point cloud, selecting neighborhood points to create a local subset and using a small version of the original PointNet to encode the data. This task is completed in three steps:

1. Sampling. The centroid for each group is chosen using iterative farthest point sampling. This is done by first choosing a random initial point to be the first centroid. The next centroid is chosen to be the point most distant to all previously found centroids. This is repeated until desired number of centroids have been found. This is guaranteed to have better coverage over the entire point cloud, compared to random sampling.

2. Grouping. Here a radius is specified, and each centroid adds all points within this radius to their group. Since the balls can overlap, the same point can belong to several different groups. The number of points in each group varies, but the fixed region scale is preferred for local pattern recognition. The radius, or the ball size, in this layer can be compared to the kernel size in the convolutional layer of a CNN.

3. Feature extraction. For the last step, local features are extracted from each subset of points within a ball, by a small PointNet module. The points are first translated to a local coordinate system, with the centroid point as the origin. Similar to how a convolutional filter is passed over an image, the same small PointNet is used to encode the features of each ball in one layer.

These three steps are what makes up a PointNet++ layer. The output from one layer is a smaller number of points in a larger feature space, like how a CNN operates. In practice, several PointNet++ layers and some fully connected layers are used to form the entire classification network, see Figure 3.4. However if semantic segmentation is desired each point in the original point set needs its own prediction score. Instead

of skipping sub-sampling and letting every point pass each layer, which would take considerable computation time, the creators have another solution.

Feature Propagation for Semantic Segmentation

When the features have been extracted in the last abstraction layer, the red layer in Figure 3.4, they are propagated up to the original point set, in a form of up-sampling. This is done by interpolating feature values for the points that were lost in the sub-sampling done in each layer. The interpolation is done according to Equation 3.5, where feature $f(x)$ in a layer is the inverse distance weighted average of the k nearest neighbours' features in the previous layer. The interpolated features are then skip link concatenated with the corresponding set abstraction features and fed through a PointNet layer. These steps are repeated until the per-point scores for semantic segmentation have been formed.

$$f(x) = \frac{\sum_{i=1}^{k} w_i(x) f_i}{\sum_{i=1}^{k} w_i(x)} \quad \text{where} \quad w_i(x) = \frac{1}{d\left(x, x_i\right)^2} \tag{3.5}$$

Multi-Scale Feature Learning

As mentioned earlier, the original PointNet architecture was not robust to varying density within the point cloud. Non-uniform sampling density is often the case with point cloud data from a LiDAR sensor, since the distance between points is greater for objects hit far away. The creators of PointNet++ introduce a simple solution to this, calling it a density adaptive PointNet layer. It works by applying the grouping stage multiple times with different scales, followed by PointNets to extract features. The multiple features are then combined according to local pattern densities, which is something the network learns how to do optimally. It should be noted that this is an expensive operation, since running PointNet on large scale neighbourhoods for every centroid is computationally heavy.

3.4 Recurrent Neural Networks

A recurrent neural network (RNN), in contrast to the commonly considered feed-forward neural network, makes use of cycles within its network structure. As such, an RNN can be effectively used to learn temporal dependencies, such as predicting time series and memorizing earlier calculations. Different structures have been proposed to this extent, of which two will be discussed; the *Simple RNN:s (SRNN)* by Jeffrey L Elman [9] and Michael I Jordan [10] and the *Long Short-Term Memory Neural Network (LSTM)* by Sepp Hochreiter and Jürgen Schmidhuber [11].

3.4.1 Simple Recurrent Neural Networks

Two very similar variations of the Simple Recurrent Network (SRNN) were proposed by Elman and Jordan, respectively. Referring to these as the Elman and Jordan networks, a simple cycle reusing either the output or the hidden state can be incorporated in the networks using the state- and output equations. First, let y_t be the network output and h_t be the hidden state at time t, where $t = 0, 1, 2, \ldots$ is a discrete-time variable. The output can then be calculated using the output equation

$$y_t = \sigma_y \left(W h_t + b_y \right), \tag{3.6}$$

where σ_y is an activation function and W and b_y are learnable weights and biases. Now let x_t be an input vector at time t. The Elmer and Jordan networks can be stated with the state equations as

$$h_t = \sigma_h \left(W_h h_{t-1} + W_x x_t + b_h \right) \quad \text{(Elman)} \quad \text{and} \tag{3.7}$$
$$h_t = \sigma_h \left(W_y y_{t-1} + W_x x_t + b_h \right) \quad \text{(Jordan)}, \tag{3.8}$$

where once again W_h, W_x and b_h are learnable weights and biases and σ_h is an activation function. Although the state equations look very similar, the hidden state is updated directly from the previous hidden state in the Elman network, whereas one needs the output (Equation 3.6) of the network to update the hidden state in the Jordan network (it should however be noted that the output is itself a function of the previous hidden state). The Elman network as such incorporates an extra layer for the output, whereas the Jordan network updates its state directly from its output.

3.4.2 Long Short-Term Memory Neural Networks

To properly optimize the SRNN, the Backpropagation Through Time (BPTT) (described in detail in [12]) algorithm can be employed, which essentially calculates partial derivatives of the loss function over both time and weight space in order to obtain a gradient. Hochreiter and Schmidhuber argues in [11] that the SRNN suffers from the vanishing/exploding gradient problem: in effect, SRNN can not learn long term dependencies due to either too small or too large partial derivatives propagating through time with BPTT. To remedy this, the authors introduce the Long Short-Term Memory (LSTM) neural network, which utilizes learnable "remember" and "forget" channels in order to gather or forget information from previous timesteps.

An LSTM cell consists of two channels propagating through time, carrying the hidden states h and c. The hidden state h is similar to the one in SRNN, where an output y_t can be calculated in the same manner as in Equation 3.6. Furthermore, the hidden state c_t similarly propagates through time but is entirely hidden and can be passed through gates controlling when its information should be added to memory or be forgotten. This allows the LSTM to learn longer time-dependencies than SRNN. The different channels and gates of the LSTM unit can be seen in Figure 3.5.

The LSTM unit is more complex than the SRNN, and as such has more complicated

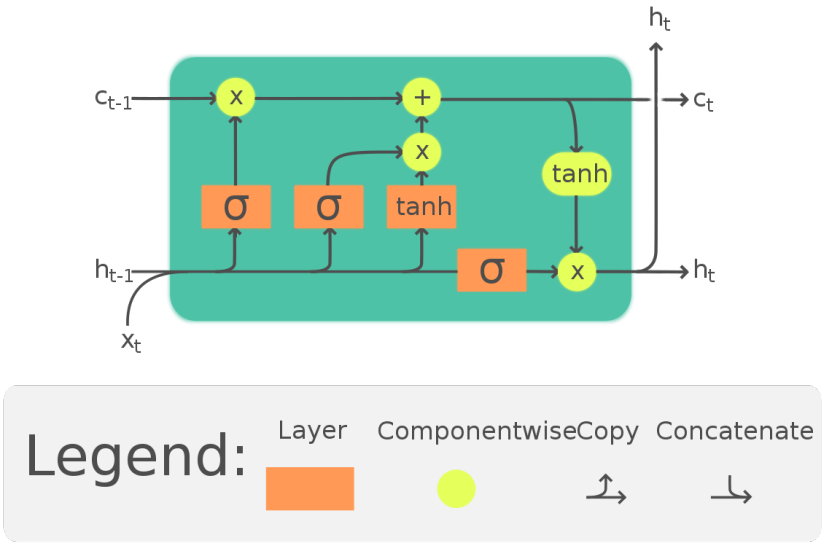


Figure 3.5: An LSTM cell with the input x and the hidden states h and c for time t [13].

state equations. The state equations for the LSTM are

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \quad \text{(forget)} \tag{3.9}$$
$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \quad \text{(input)} \tag{3.10}$$
$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \quad \text{(output)} \tag{3.11}$$
$$\hat{c}_t = \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \quad \text{(new long term memory)} \tag{3.12}$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \hat{c}_t \quad \text{(long term memory)} \tag{3.13}$$
$$h_t = o_t \circ \sigma_h(c_t) \quad \text{(hidden state),} \tag{3.14}$$

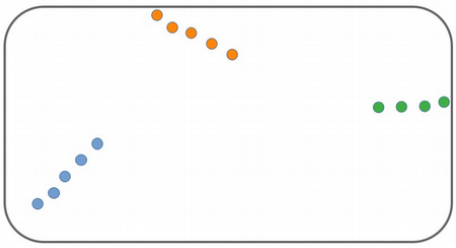
where $W_{(\cdot)}$ and $U_{(\cdot)}$ are learnable weights and $b_{(\cdot)}$ learnable biases at respective gates and $\sigma_\cdot$ are activation functions.

3.5 Tracking objects from sensor data

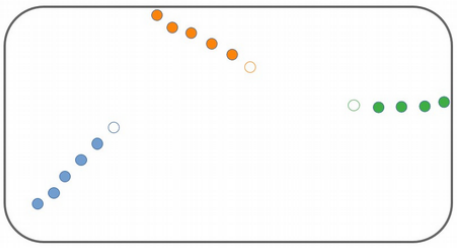
Tracking objects from sensor data means to distinguish present objects from each other over time within the setting measured by the sensor. A *tracker* in turn, is a method employed to accomplish this task. A robust tracker should be able to identify where different currently tracked objects are, disregard noise in the form of "false objects", handle missed objects as well as identify when objects enter or leave the setting. Properly tracking objects is as such a complex and sometimes difficult task.

Noise and missed objects are to be expected from time to time due to imperfections of the sensor as well as the complexity of the setting. In the case of a LiDAR sensor, after filtering out the background, "semi-static" objects such as trees swinging in the wind may be identified as moving objects. This should of course be considered noise. Similarly, a tracked person could become occluded at times and as such not be seen by the sensor and registered as a moving object.

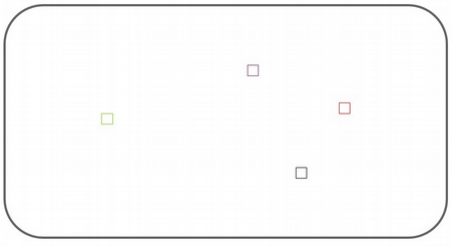
Here, we introduce a common approach to the tracking task. The tracking is essentially divided into three steps in each point of time. First of all, *predictions* of each of the previously tracked objects position are made. Secondly, using the newly acquired sensor data of different objects, we make an *association* between the measured objects and predictions, where for example proximity and other similarity measures are taken into account. And finally, using the available information of predictions and associations, we make decisions in order to *update* tracks. These decisions include conclusions about where the previously tracked objects are (alternatively if they have left the setting), if there are any new objects which should initialize new tracks and if there is any noise to be disregarded. An example of this pipeline at a point of time can be seen in Figure 3.6 with steps a-f.



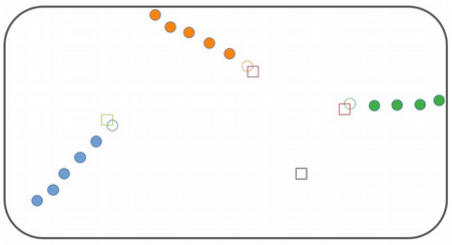
(a) Three current tracks.



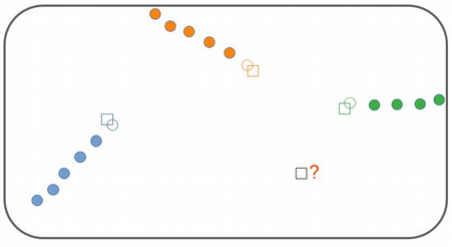
(b) Predictions (circle) of each track.



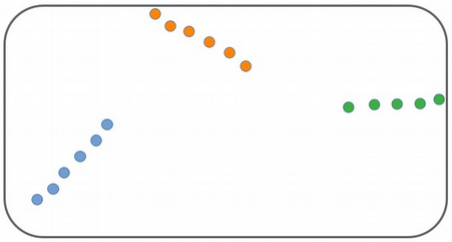
(c) New measurements (square).



(d) New measurements compared to tracks and predictions.



(e) Associate each track to a measurement (same coloring). What to do with fourth measurement?



(f) Ignore last measurement. Other solution could be to introduce new track.

Figure 3.6: Example of tracking pipeline with three current tracks and four measurements.

3.5.1 Kalman filter

A common approach to predict and update present tracks at a point in time is the *Kalman filter*. Given a track, we define a state in time as $s_t = (x, y, z, \dot{x}, \dot{y}, \dot{z}, \ddot{x}, \ddot{y}, \ddot{z})$, which contains the position, velocity and acceleration of the tracked object at time t.

Here, we assume that the state is generated by a Gaussian process such that

$$s_t \in N(Fs_{t-\Delta t}, Q), \tag{3.15}$$

where F is a transition matrix defined as

$$F = \begin{pmatrix} 1 & 0 & 0 & \Delta t & 0 & 0 & \frac{1}{2}(\Delta t)^2 & 0 & 0 \\ 0 & 1 & 0 & 0 & \Delta t & 0 & 0 & \frac{1}{2}(\Delta t)^2 & 0 \\ 0 & 0 & 1 & 0 & 0 & \Delta t & 0 & 0 & \frac{1}{2}(\Delta t)^2 \\ 0 & 0 & 0 & 1 & 0 & 0 & \Delta t & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & \Delta t & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \Delta t \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

and Δt is the size of the timestep. As can be understood from the transition matrix, the motion model for a walking object is as such defined as (for y and z as well)

$$x_t = x_{t-\Delta t} + \Delta t \dot{x}_{t-\Delta t} + \frac{1}{2}(\Delta t)^2 \ddot{x}_{t-\Delta t} \tag{3.16}$$

$$\dot{x}_t = \dot{x}_{t-\Delta t} + \Delta t \ddot{x}_{t-\Delta t} \tag{3.17}$$

$$\ddot{x}_t = \ddot{x}_{t-\Delta t}. \tag{3.18}$$

Furthermore, the covariance matrix Q represents noise in the process, such as measurement errors and imperfections. Given the state vector $x_{t-\Delta t}$ and a measurement z_t, a prediction and updating schedule (which takes the associated measurement into account for each track – see Data Association below) can be implemented as outlined in [14] (p. 588).

3.5.2 Data Association

Given a set of M measurements $z = (z_1, z_2, \ldots, z_M)$ and N predictions $x^* = (x_1^*, x_2^*, \ldots, x_N^*)$ of tracks, the task is to, for each prediction, either associate a measurement or conclude that a measurement for that track is missing. Furthermore, each measurement must of course not be assigned to more than one track. The problem can be stated as a combinatorial optimization problem, where the sum over all assignments under a similarity function d (which could include, for example, position and some sort of appearance) is to be minimized (a more precise formulation can be found in [15]). The problem can be stated with three requirements:

1. The assignments of measurements and predictions should be minimized over some similarity function.

2. Each measurement may at most be assigned to one prediction.

3. Each prediction may at most be assigned to one measurement.

Requirements 2. and 3. mean that each measurement originates from one track only[1], but we should state that it also includes missed detections (0-1 assignment) and clutter/noise (1-0 assignment).

One common approach to solving this problem is the bipartite matching Hungarian Algorithm, where the problem is solved by creating the cost matrix C with $C_{ij} = d(x_i^*, z_j)$. A complete description of the algorithm can be found in [16].

[1]We should note that it is not uncommon in practice for two measurements to originate from the same track. In the case of a LiDAR-setting, this could be the case when clustering of an object (such as a pedestrian) mistakenly results in two clusters. If we were to follow the 1-1 assignment rule, one of these measurements should at this stage be disregarded as clutter – however there of course exist other approaches as well, at both pre- and postprocessing time.

4 Method

4.1 Simulation of LiDAR Data

Since we were going to use deep learning for classification, segmentation and tracking we needed training data for our models. We did not have access to the Luminar Hydra LiDAR sensor until late into the project, so we could not take real life data and annotate it ourselves. However, even if we would have been able to, it would have been an extremely time consuming process, which would have slowed us down too much. Instead we resorted to simulating data, using the open-source autonomous driving simulator CARLA by Dosovitskiy et al. [17], see Figure 4.1.

Figure 4.1: Example of a frame from the CARLA simulator.

The idea of simulating data was that we could generate a lot of data quickly, from different scenarios and environments. The advantage of using CARLA was that it had a built-in Python API, which let us spawn pedestrians, cyclists and cars easily. The people and vehicles in the world were controlled by the simulation, and acted like you would expect them to in traffic. There were several off-the-shelf urban environments to use, with roads, buildings and vegetation. Most importantly it had the capability to simulate a LiDAR, which we could control with the Python API.

That said CARLA did have some limitations. There was little variation how people looked and moved in CARLA, which gave us a very homogeneous dataset to work with. There were no animals in CARLA by default, but we managed to add one dog. The LiDAR scan pattern was configurable to a certain degree, but not as much as the Luminar Hydra LiDAR. Finally, CARLA does not calculate intensity of the returned beams, meaning we lost out on one dimension of data.

In the simulation we randomized the location of the LiDAR every frame, and recorded the world at 10 frames per second, with about 200 pedestrians and 50 vehicles moving

in the simulation simultaneously. This was done for 5 different worlds, with the actors being replaced every 1000 frames. The LiDAR data was saved as CSV files, which were later processed and formatted into the datasets that were needed.

4.2 Dataset Analysis

In total we constructed three datasets for each of classification, segmentation and tracking using CARLA. Below follows a brief description of each dataset.

4.2.1 Classification Dataset

For the classification dataset we focused on things we could find in the real world, that were also readily available in the simulator. The classes became *pedestrian*, *cyclist*, *car* and *other*. We collected 10 000 objects of each class from the simulator, with the criteria that they consisted of at least 50 data points and were smaller than 5m in every dimension. The only exception was the "other" category which had 20 000 objects, since it was much more diverse, see Table 4.1 for its contents.

Object type	Quantity
Vegetation	8 000
Static object	4 500
Dynamic object	2 000
Pole	2 000
Traffic sign	1 000
Guard rail	1 000
Animal (dog)	1 000
Traffic light	500

Table 4.1: Contents of the "other" class in the classification dataset.

In Figure 4.2 there is a histogram of the number of points in all 50 000 objects in the dataset. It is not a problem that the number of points vary, on the contrary it is good that the models get to train on both sparse and dense point clouds. However there has to be a fixed input shape to the models, so there must be a maximum number of points. Looking at the histogram we can tell that the vast majority of objects fit within 500 points, so the input shape for the classification models was set to $2^9 = 512$ points. Since most objects had less than 512 points, those objects were padded with copies of themselves. This has no impact on the training or classification result, as discussed earlier. For the few objects with more than 512 points, random points were dropped until they had the right amount of points.

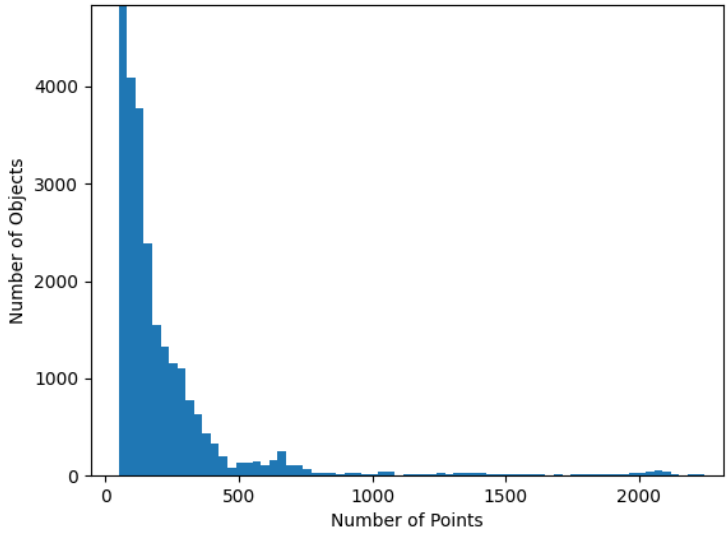


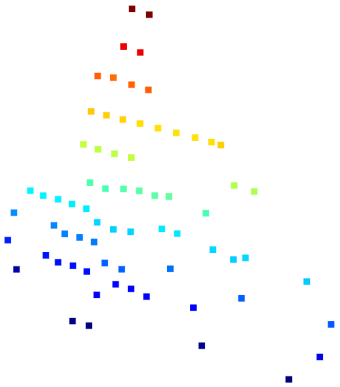
Figure 4.2: Histogram depicting the number of data points per object in the generated dataset.

Since CARLA has no intrinsic noise, noise was added to the data to make it more realistic. Adding noise to the input data helps networks generalize [18], since they need to learn patterns instead of exactly how the training data looks like, thus reducing overfitting. The Hydra LiDAR sensor has minimal angular noise, but a small radial noise. To mimic this a uniform radial noise $\epsilon \in (-2.5, 2.5)$ was added to each data point. In addition to this the data was normalized, by making sure it was zero-mean and fit within the unit sphere, see Equation 4.1.

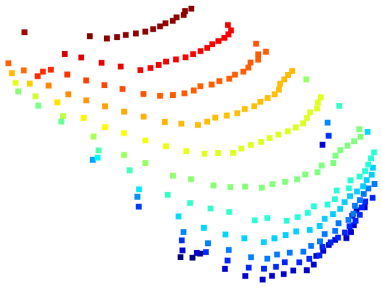
$$\begin{aligned} \mathbf{X}_{input} &= (\mathbf{x}_1, ..., \mathbf{x}_{512}) \text{ where } \mathbf{x}_i = (x_i, y_i, z_i) \\ \frac{1}{512} \sum_{k=1}^{512} \mathbf{x}_k &= \vec{0} \; \forall \; \mathbf{X}_{input} \\ |\mathbf{x}_i| &< 1 \; \forall \; \mathbf{X}_{input} \end{aligned} \tag{4.1}$$

In Figure 4.3 there is one example point cloud from each class, generated by the simulated LiDAR.

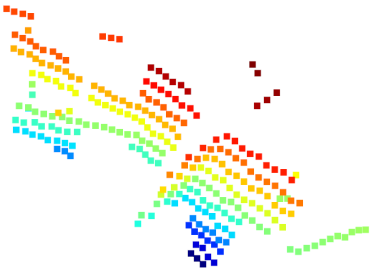
(a) Point cloud of simulated human.



(b) Point cloud of simulated cyclist.



(c) Point cloud of simulated car.



(d) Point cloud of simulated miscellaneous object.

Figure 4.3: The four simulated classes in CARLA as seen by the LiDAR.

4.2.2 Semantic Segmentation Dataset

For the semantic segmentation dataset the classes were almost the same, except the other class was replaced with the background class, and since animal did not fit in the background class it was made into its own class. This means the classes were *pedestrian*, *cyclist*, *car*, *animal* and *background*. Since the background points in any given frame were vastly overrepresented and supposed to be filtered away anyway in the preprocessing step, only 1 % was kept in each frame. For this dataset 10 000 frames were collected, with the only criterion that there were at least 3 objects captured within each frame. The distribution of points over all frames was according to Table 4.2.

Class	Pedestrian	Cyclist	Car	Animal	Background
Percentage	8.2	8.4	14.5	1.0	67.9

Table 4.2: Class representation in the semantic segmentation dataset.

In Figure 4.4 there is a histogram of the number of points in the semantic data frames. Just like for the classification dataset, the maximum input size needed to be set. For the semantic dataset an input size of $2^{11} = 2048$ points was enough for the majority of the data. In the same way as before, all data was made the same shape, normalized and noise was added.

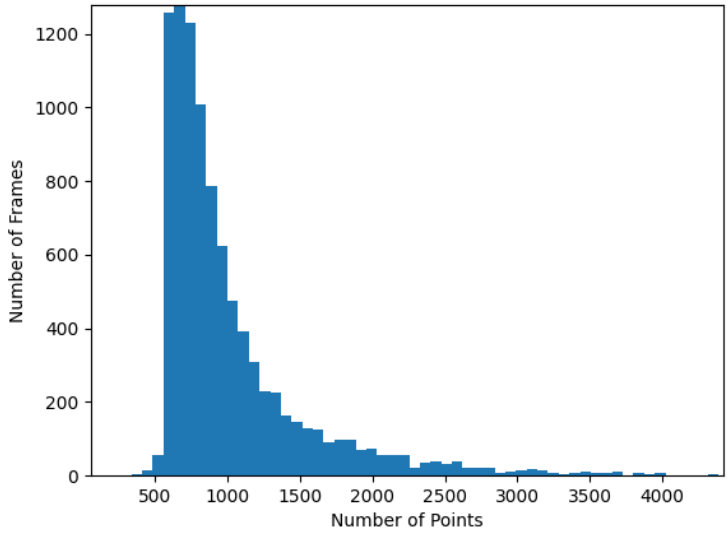


Figure 4.4: Histogram depicting the number of data points per frame in the generated semantic dataset.

4.2.3 Tracking Dataset

Using the recorded positions of pedestrians and cars in Carla every tenth of a second repeatedly for 100 seconds (resulting in 1000 frames per track), we created approximately 1500 simulated tracks of each and stored these as as distinct tracks.

Using the stored tracks, we created a new simulation environment ("track simulator") to generate track data over time. The simulation environment spawned and despawned *pedestrians* and *cars* ("track") randomly on a predetermined xy-plane-segment over a given timeframe, generating frames with recorded positions and other relevant values needed to train the Deep Tracker (for details, see section 4.4.3). When each new track was to be spawned, the track simulator randomly chose a track segment from the 3000 prestored tracks, which in turn was randomly rotated and spawned on a random location together with a small Gaussian noise added to each position. Using these methods containing several random steps, the training data could almost be augmented ad infinitum and in addition easily generated on the spot.

4.3 Filtering and Clustering

4.3.1 Distance Filtering

The max distance filtering has two steps, calibrating the filter and actually performing the filtering. To calibrate the filter some initial frames need to be chosen as background points, preferably entirely static frames as everything within these frames will be considered background. Usually we pick the first 50 to 100 frames, depending on how noisy the environment is. We also pick a threshold distance ϵ the points need to have from the background, in order to be considered foreground. The threshold is usually

between 0.1 to 0.2 meters, also depending on the environment.

Points from the LiDAR sensor are represented by floating point values $\mathbf{p}_i = (r_i, \theta_i, \phi_i)$, but to create a representation of the background we need to map them to a matrix with integer indexes. The Hydra scans 64 horizontal lines when recording at 10 frames per second, but there is no discrete pattern along these lines. This means there are only 64 different θ_i, which we can map to matrix indices directly, but we need to discretize the horizontal angles ourselves. To mimic the hydras resolution we use steps of $\Delta\phi = 0.1$ in the discretization of the 120°field of view.

This gives us a matrix $\mathbf{B} = (b_{ij}) \in \mathbb{R}^{64\times1200}$ to represent the background, where i is the line number and j is a horizontal angle interval of length $\Delta\phi = 0.1$. Using the discretization above, each point within the LiDAR's field of view can be mapped to an entry b_{ij} in matrix $\mathbf{B}$. To construct $\mathbf{B}$ we go through all initial frames and map each data point to an in index ij. We then chose b_{ij} to be the radius of the closest point among points mapped to index ij. The closest point is chosen since everything beyond it will be filtered away.

During the filtering phase a frame $\mathbf{F}$ consisting of n points $\mathbf{p}_i = (r_i, \theta_i, \phi_i)$ is input to the filtering algorithm. Each angle pair (θ_i, ϕ_i) is mapped to an entry b_{ij} in $\mathbf{B}$, holding the distance from the sensor to the background for that specific direction. If the radius r_i plus the threshold ϵ is closer to the sensor than the background b_{ij}, the point passes the filter. If $r_i + \epsilon > b_{ij}$ the point is too close to the background, and filtered away.

Sometimes the initial frames are not enough to fill matrix $\mathbf{B}$ with enough information about the background, due to noise and jittering. This is why after the initial frames have been stored in $\mathbf{B}$, we perform max pooling with kernel size 3×3 and stride 1, filling holes in the background matrix, making the filtering more robust.

4.3.2 Clustering

Clustering was done with the DBSCAN algorithm, in particular the scikit-learn implementation, by Pedregosa et al. [19]. All the points that pass the filtering phase are simply input directly into the clustering algorithm, which outputs a cluster label for each point. The parameters we need to chose are the maximum allowed distance between core points ϵ and the minimum number of points in a cluster. When clustering objects in the simulated data, $\epsilon = 0.3$ yielded the most satisfactory results, but in the real world data this often yielded more clusters than desired. The real world data was a bit more sparse, so $\epsilon = 0.5$ was empirically chosen as a better fit. The minimum number of points in a cluster was chosen to be 5, which was robust to most noise.

Each cluster could then be sent to the tracker and thereafter the classification model. If semantic segmentation was used, clustering could be performed in parallel instead.

4.4 Model Implementations

4.4.1 PointNet++ Classification

For the classification problem four models were created from PointNet++ layers. In two of the models we used the multi scale learning feature, and in the other two it was turned off. In addition to this two of the models had smaller layer sizes, and two had larger layer sizes. This was so that we could see how these factors affected the performance as well as the inference time of the models.

All four models consist of three PointNet++ layers, which is what was used in the original paper by Qi et al. [8]. In addition to this there are a few dense layers, and dropout layers to reduce overfitting. *Sample size* refers to the number of sampled local ball shaped regions, with radius *sampling radius*. After the sampling the regions are processed by a small PointNet with fully connected layers of size *mlp*.

The input size to each model is as described in the dataset section, $\mathbf{X}_{input} = (\mathbf{x}_1, ..., \mathbf{x}_{512})$ where $\mathbf{x}_i = (x_i, y_i, z_i)$, which is a matrix of shape 512×3. The input can easily be augmented to fit more features, such as intensity, if the opportunity arises. The output from each model is a vector of length 4, containing the probability of the object being each of the 4 classes.

After each layer there is batch normalization and an activation function, which is chosen to be the ReLU function. The only exception is the output layer, which simply uses the softmax function produce the probabilities. Below follows a short description of each model, and each model can be read about in detail in Appendix A.

The small single scale model is the simplest of the four classification models. In each PointNet++ layer it only scans at a single scale. It has 23 292 model parameters.

The big single scale model has more parameters in each layer compared to its smaller counterpart, with 89 972 parameters.

The small multiple scales model collects features at multiple scales in the first two PointNet++ layers, but still does not have very many parameters, with 28 168.

The big multiple scales model, it is the most advanced of the four models. It both utilizes the multiple scale feature, and has more parameters in each layer to support it, with 108 076 parameters.

4.4.2 PointNet++ Semantic Segmentation

For the semantic segmentation two models were created from PointNet++ layers, one with many parameters and one with less parameters. They were constructed much like the classification models, except they have one more feature abstraction layer, and then four feature propagation and interpolation layers, which were not needed in the classification case.

The input size to these models are $\mathbf{X}_{input} = (\mathbf{x}_1, ..., \mathbf{x}_{2048})$, four times bigger than to the classification models. The biggest difference when it comes to semantic segmentation is the output size. Instead of a single vector of probabilities, it outputs a matrix with each row being a probability vector associated with each of the input points. This means the output is $\mathbf{Y}_{output} \in \mathbb{R}^{2048\times5}$, since we have 5 semantic classes. The models can be read about in detail in Appendix A, below is a brief description.

The Small Semantic Segmentation Model collects features at a single scale at each layer, then uses feature propagation to give each point its own semantic label. It uses 63 855 model parameters.

The Big Semantic Segmentation Model has the same functionality, but has vastly more parameters to support it, at 246 871 parameters.

4.4.3 Deep Tracker

The Deep Tracker is a direct implementation of the recurrent network structure presented in [20] by Milan et al. A summary of its implementation can be found in the appendix. In short, it contains two modules; the predict-update-birth/death module ("predict/update module") and the data association module ("DA module"). In total, this network aims to solve the tracking problem with the steps mentioned in section 3.5. It should here be noted that the predict/update module is an extended Jordan network with the "sub-module" named Predict in the figure being a standard Jordan Network. In addition, the DA module is nothing less than a standard LSTM.

Here follows a brief explanation of some of the variables involved in the, defined in the same way as in [20]. At a given time $t+1$, an array $x_t = (x_t^1, x_t^2, \ldots, x_t^N)$ consists of N positions where each x_t^i denotes the current position[1] of track i at time t. Since there are not necessarily always N tracks apparent in the setting, we denote "false tracks" by the position (0,0). As such, the network is able to track up to N objects at a time. Since it is not necessarily possible to read which positions correspond to a "true track" solely from the positions array x, we employ another array $\varepsilon_t = (\varepsilon_t^1, \varepsilon_t^2 \ldots, \varepsilon_t^N)$, containing N probabilities of each position in x_t being a "true track". To consider a track to be true is then a problem of defining a proper probability threshold. At the given time, new measurements z_{t+1} are provided by the LiDAR in the form of a number of positions. The array $z_{t+1} = (z_{t+1}^1, z_{t+1}^2, \ldots, z_{t+1}^M)$, similarly to x, contains the given measurements. In turn, since the number of measurements are variable as well, we denote "non-measurements" by the position (0,0). Hence, the network is able to provide up to M measurements, where M should be at least somewhat larger than N to properly be able to account for extra measurements derived from noise. Now, given that the network works in each timestep $t+1$, out task is to yield current positions x_{t+1} based on the previous positions x_t, the previous track probabilities ε_t and new measurements z_{t+1}. This has in detail been described in section 3.5, and we will here be describe it in the context of Milan et als network:

[1] In reality, x^i could actually be any kind of feature vector containing more information than just the current position – for example size, appearance, bounding box measurements etc.

1. Given x_t and previous hidden state h_t, we make a prediction with the Predict submodule of x_{t+1} – denoted x^*_{t+1}.

2. Using the prediction x^*_{t+1} and the measurements z_{t+1}, we create the distance matrix C_{t+1}, with the matrix elements $C^{ij}_{t+1} = \|z^i_{t+1} - x^j_{t+1}\|$. Using the distance matrix, the DA module traverses through each track as its "time dimension", at each step i outputting a distribution $\mathcal{A}^i_{t+1}$ of $M+1$ probabilities that a given measurement corresponds to the position x^i_{t+1}. The first M probabilities directly translates to the M possible measurements, and the last probability denotes "missing measurement". Using argmax on each row in $\mathcal{A}_{t+1}$ then yields a data association between positions in x_{t+1} and measurements in z_{t+1}.

3. Aggregating the predictions (including the hidden state from the Jordan network for extra information), measurements, data associations and track probabilities, the Update submodule produces new updated positions x_{t+1} and tracks probabilities ε_{t+1}. The concatenate-dot product-element-wise-multiplication flow in the Update module first repeats the z_{t+1}-array N times and appends the corresponding predictions x^*_{t+1} (concatenating). In the next step, a tensor dot product is made which ensures that on each position $i = 1, \ldots, N$ in the concatenated array a weighted sum between measurements and corresponding prediction is made with the data association probability distribution $\mathcal{A}^i_{t+1}$ – giving the highest meaning to the most probable measurement (or to the prediction if a measurement is deemed missing). Yielding new candidate positions for updating the tracks, these are then multiplied by the corresponding track probabilities in ε_t (element-wise multiplication). A final step containing learnable weights now aggregates these new tracks with information from the Prediction module (h_{t+1}) to update tracks and tracks probabilities. Considering that candidates were calculated earlier, we may assume that these weights have extra purposes; they could for example act as "self-regulation" when the tracks probabilities start to decrease down to a certain threshold until the tracks positions are essentially missing (that is, close to (0,0)). The final output ε^*_{t+1} from the Birth/Death submodule simply denotes the absolute difference of probabilities between timeframes. We do not use this output when employing the model, but rather during its training (see section 4.5.3).

4.5 Model Training

4.5.1 Classification Models

All four classification models went through the same training and testing scheme. The dataset of 50 000 objects was split into 80 % training data, and 20% validation data, and made into batches of 64 objects each. The training and validation data was processed by the model until the validation accuracy changed by less than $\Delta = 0.01$ over 10 epochs, after which the best model weights were saved.

The gradient descent optimization algorithm used was Adaptive Moment Estimation (Adam) with a learning rate of $\alpha = 0.001$. The loss function was sparse categorical

cross-entropy, which measures how different two discrete probability distributions are.

Since early stopping was used the models trained for a varying number of epochs, but all models converged in 20 to 40 epochs, which took only a few minutes for the smallest model and about 20 minutes for the biggest model. The training was sped up by using a GPU.

4.5.2 Semantic Segmentation Models

The training scheme for the semantic segmentation models was almost the same, with the same data split, same early stopping criterion and same optimizer, but with smaller batches of 4, since each object was much more complex. They also took about the same amount of time to train as the biggest classification model. However the loss function had to be modified, due to the dataset.

There is a large class imbalance in the semantic dataset, with the background class being much larger even after removal of 99 % of the points. On the other hand animals are very underrepresented, with only 1 % of the data. If nothing was done, this would have lead to very biased models, since the loss function will be minimized by predicting the most common class, while avoiding the least common class. This was countered by creating a weighted loss function for the semantic segmentation model. After the dataset split was done, class weights were calculated such that the weight for each class was inversely proportional to its occurrence in the training dataset. The loss is then calculated by taking the sparse categorical cross-entropy and multiplying the loss with the weight of the ground-truth label. This means the loss becomes greater if the model misclassifies a point with an uncommon ground-truth label.

4.5.3 Deep Tracker

Following the findings in [20], we trained the different modules separately in order to ensure convergence. We trained each module on two seconds long sequences in batches of 10 sequences using RMSprop and BPTT.

The two seconds long sequences were created using the tracks simulator we introduced in section 4.2.3. Below follows a summary of the parameters used when generating a simulated two seconds sequence with each parameter is valid in a frame-to-frame basis. Note that when probabilities are given in a list-form $[\cdot]$, it should be read as randomly sampled from that list. Furthermore, if we provide two lists, the second list denotes the sampling probability p_s from the first list of probabilities.

- A new object has a probability of $p = [0.1, 0.2, 0.5, 0.8, 0.9]$ of spawning. An object will however not spawn if $N = 10$ objects are already present.
- In each frame, a spawned object has a probability of $p = [0.01, 0.1, 0.9]$, $p_s = [0.75, 0.2, 0.95]$ of despawning.
- When an object should spawn, first a choice of whether to sample from the prestored cars tracks or pedestrians tracks is made with $p = 0.5$.

- Let current frame be denoted by i. From the prestored tracks, a $20 - i$-long sequence is randomly sampled from a random track. See Figure 4.5 for an example of this process.

- The sample is augmented by randomly rotating it along its first position with an random angle $\theta \in [0, 2\pi)$ and randomly placed on a xy-plane segment with the dimensions $(x_{\min}, x_{\max}, y_{\min}, y_{\max}) = (5, 100, -100, 100)$.

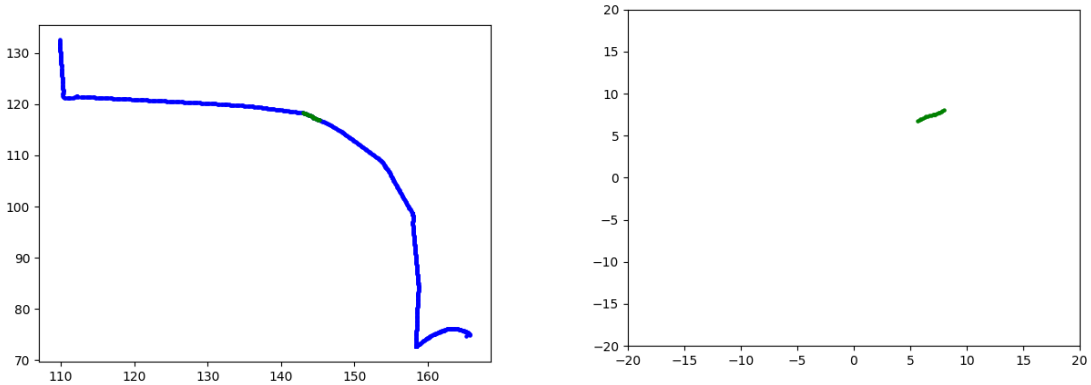
- When measuring the position of a spawned object, a small Gaussian noise is added to the coordinates.

- A measurement of a spawned object can go missing with a probability $p = 0.01$.

- A number of false measurements may be added to the measurements with a probability $p_{\text{noise}} = 0.5$ for each "available" slot in the z-array, adjusted for the number of current tracks. This in turn yields a lot of noise among the measurements, ensuring that the model is robust against noise.

- The coordinates are finally normalized by dividing with the maximum span of the plan (in this case 100m).

The tracks simulator steps through each frame (20 in total) using these parameters and creates the proper training data along the way, finally batching them up into 10 batches. The different probabilities were empirically derived and provided a good balance between crowded and less crowded time sequences as well as exposed the model to a proper amount of noise and measurement failures.



(a) Track data from Carla with random segment.

(b) Random segment augmented and placed at random coordinates.

Figure 4.5: Process of acquiring random augmented track segment for pre-determining track of spawned object.

As we stated before, we trained the two modules separately in order to have any chance at convergence. This approach is further discussed by Milan et al. [20]. We trained The

predict/update module with the loss function described in the same article, containing terms for each output[2]:

1. x^*_{t+1}: Mean squared error
2. x_{t+1}: Mean squared error
3. ε_{t+1}: Mean binary cross-entropy
4. ε^*_{t+1}: Absolute value.

The final term ensured that the probability array did not change too much from one frame to the next – without it, Milan et al. [20] argues that the model learns to make too harsh decisions on whether a track is still in place or not when a measurement goes missing. Finally, following the authors, we used RMSprop as the optimizer together with BPTT for calculating gradients with a learning rate of 0.003, which in turn was updated every 20 000 iterations with a decay rate of 0.95. A batch consisted of ten two-second long sequences, resulting in a total of 500 000 unique sequences generated throughout the training.

We trained the DA module using the mean categorical cross-entropy as its loss function. Since the data association task is done per frame and does not have a direct temporal dependence, there was no need to do a BPTT through a whole sequence – meaning that all frames were concatenated into making a batch of 200 frames. Instead, as discussed earlier, the "timestepping" in the LSTM is done track-by-track, in which the BPTT algorithm comes into play. The training is in every other aspect done in the same manner as for the Predict/update module.

4.6 Evaluation Metrics

4.6.1 Classification and Semantic Segmentation

To evaluate the models there are many metrics that can be used, see [21]. Below are brief descriptions of some common metrics we used to help us understand how the different models performed and how they compare.

The **confusion matrix**, see Table 4.3 below, reveals a lot of information about a classification model, and is the foundation on which many of the other metrics build upon. It shows how many instances of each class that were correctly classified, the green boxes, and how many were incorrectly classified, the red boxes. Importantly it reveals what a wrongly classified object was predicted to be, what it was *confused* for, which can be very helpful.

Accuracy is one of the most popular metrics in multi-class classification, it is intuitive and easy to understand. It is simply a matter of dividing the sum of correct predictions

[2]Note that, since both the prediction step and the update step works towards finding the correct next x, we used the same ground-truth data for both of these.

		Predicted				
	Classes	Pedestrian	Cyclist	Car	Other	Total
True	Pedestrian	1	2	3	4	10
	Cyclist	5	6	7	8	26
	Car	9	10	11	12	42
	Other	13	14	15	16	58
	Total	28	32	36	40	136

Table 4.3: Example of confusion matrix. The green boxes are where the true and predicted class coincide, which is a correct prediction. The red boxes are incorrect predictions.

by the total amount of predictions. It tells us how often the model was correct over the entire set of data. One has to be careful when using accuracy with imbalanced datasets, since it can hide large errors on underrepresented classes.

Before we move on we need to understand *True Positives* (TP), *False Positives* (FP), *False Negatives* (FN) and *True Negatives* (TN). These are defined individually for each given class, see Table 4.4 for an example with the pedestrian class. TP refers to correctly predicting that an object is of the given class, while TN refers to correctly predicting that it is not. Meanwhile, FP refers to incorrectly predicting the given class, while FN refers to incorrectly predicting that it is not of the given class.

		Predicted			
	Classes	Pedestrian	Cyclist	Car	Other
True	Pedestrian	TP	FN	FN	FN
	Cyclist	FP	TN	TN	TN
	Car	FP	TN	TN	TN
	Other	FP	TN	TN	TN

Table 4.4: TP, FP, FN and TN for the pedestrian class. Note that a cyclist predicted as a car is still a TN since the model did not predict it to be a pedestrian.

We can now define **precision**, which tells us for a given class how much can we trust the model when it predicts positive, see Equation 4.2. Here TP and FP are defined according to the table above. When taking the average precision for each class, we get the **macro-precision**, which is what we are actually interested in.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{4.2}$$

We can also define **recall**, which tells us how many of the positives the model catches, see Equation 4.3. If the model never predicts positive, even when it should, it can have great precision, but terrible recall, which is why both are needed for a good model. In the same way as earlier we define **macro-recall** as the average recall for each class.

$$\text{Recall} = \frac{TP}{TP + FN} \tag{4.3}$$

Recall and precision are combined in the so called **F1-score**, see Equation 4.4, which is simply a harmonic average between the two. Since a good model should have both good precision and recall it can be enough to look at the F1-score, but sometimes one is more important than the other, in which case we might not be as interested in the F1-score. For the **macro-F1-score** we input the macro-precision and macro-recall in Equation 4.4 below.

$$\text{F1-Score } = 2 \cdot \left(\frac{\text{precision } \cdot \text{ recall}}{\text{precision } + \text{ recall}} \right) \tag{4.4}$$

Inference time is the time it takes for one forward propagation through the network, from input to output. Since the entire pipeline is on an execution time budget, comparing inference time is important when evaluating the models. It is simply calculated by measuring the time it takes for the model to process the entire test set, in batches of one, and dividing by the number of test items.

A metric we were interested in studying in this specific task was how many data points that were needed to accurately classify a pedestrian. This is because the amount of laser pulses that hit an object decreases when the object is farther away from the sensor, and we wanted to investigate at what range the classifier could be trusted. In practice we remove data points from the input data to simulate the person being far away, and check the accuracy versus the number of input points.

4.6.2 Tracking

We used similar and equivalent metrics to those described above were to evaluate the tracking system. The modular nature of the steps in tracking allows the different steps to be evaluated separately, such as the different methods for prediction or data association – these different metrics all measured performance in a frame-to-frame basis. Consider however a sequence of frames in which a number of distinct tracks is present. Applying a tracker for the whole sequence, we can compare the tracks measured by the tracker to the ground-truth tracks. One such comparison would be the **mostly tracked** (ML), which measures the rate of ground-truth tracks which have been at least 80 % measured into a track by the tracker during a sequence. Similarly, the metric **mostly lost** returns the proportion of tracks where less of 20 % of their tracks went missing.

In order to evaluate prediction, with the tracker making predictions x^* on a single track, the sequential **mean absolute error** (MAE)

$$\text{MAE} = \frac{1}{n} \sum_{t=1}^{n} |x_t - \tilde{x}_t| \tag{4.5}$$

is a natural measurement.

The data association task can essentially be considered a classification task. In each frame, each track need to either be classified into (associated with) the correct measurement or the extra class "missing measurement". We should however note that while

this is the natural way of the Deep Tracker to solve the data association problem, the Hungarian algorithm lacks the ability to disassociate a target – the task at hand from its very definition is to associate each and every target to a measurement in a 1-1 fashion. The HA will more or less choose the "second best" measurement if a measurement for a current track would be missing. This can lead to bad data associations, which in turn can propagate the problem if this association disturbs the following methods of updating and predicting. We could for example add a post-processing *rejection condition* in order to remedy this, where for example a proposed data association between prediction x_i^* and measurement z_j may be rejected if the similarity $C_{ij} > D$ for some pre-determined value D.

From the classification point of view, there are a number of tracks in each frame that need to be "classified" into the correct measurement. From here, we can define the track-wise True Positives, False Positives, False Negatives and True Negatives in the same manner as the metrics are defined above. As the number of tracks and measurements tend to vary over time, we are more interested in the average rates of these metrics. As such, these metrics are over each track to create the macro-metrics (just as described above). Consequently, we use accuracy, precision, recall, F1-score and inference time the methods.

While comparing the predictions and data associations with each other, it would be good if there also was a way to look at the macro-perspective: "how good are the models at tracking?". To evaluate this, we use two metrics. Consider a sequence of frames. In each sequence a number of tracks will at some points in time spawn and despawn (tracks not despawned at the end of the sequence are considered to despawn at the end of the sequence). Comparing these ground-truth tracks to the tracks found by a tracker, we consider a ground-truth track *mostly tracked* if at least 80 % of its positions are found in a track from the tracker. We consider a track vice versa as *mostly lost* if less than 20 % of its positions are found in a track by the tracker.

We create two models to compare with the Deep Tracker modules: two "baseline" models, where the first model consists of a Kalman filter for prediction and updating, and the Hungarian algorithm for performing the data association. The Kalman filter uses parameters found in a previous master's thesis project by Berntsson and Winberg [1]. The first model in particular implements the simplest tracking logic; any measurement not associated with a track initiates a new track and a track is terminated as soon as a measurement goes missing. The second model similarly implements a Kalman filter and the Hungarian algorithm, but adds two rules. First of all, track initiation and termination is determined by a "track probability" p_t with a decay parameter $\alpha = 0.7$ for each time a measurement goes missing. Here, a track is terminated when $p_t < 0.2$. A track is initiated with $p_t = 0.5$ as soon as a measurement is not associated with a current track. Additionally, a *rejection condition* with $C_{ij} > 1$ is added in order to reject "false" associations.

5 Results

In this section all results are documented, which includes tests on the classification models, semantic segmentation models, tracking models and the whole pipeline.

5.1 Classification Models

In Table 5.1 we can see the four classification models evaluated with the metrics described in the Method chapter. The bold numbers are the best results in each category.

Model	Accuracy	Precision	Recall	F1-Score	Inference Time (ms)
Small, Single Scale	0.9847	0.9832	0.9858	0.9845	**6.29**
Big, Single Scale	0.9886	**0.9885**	0.9890	0.9887	6.41
Small, Multiple Scales	0.9778	0.9764	0.9796	0.9779	10.61
Big, Multiple Scales	**0.9901**	**0.9885**	**0.9919**	**0.9902**	10.70

Table 5.1: Evaluation metrics for the classification models. Best result in each category is in bold.

Next there is a confusion matrix for each of the four models, Tables 5.2, 5.3, 5.4 and 5.5. In the confusion matrices the classification result for each object in the test set is found. The columns show what the models predicted, while the rows show the true label of the object. Ideally all objects appear on the diagonal of the confusion matrix, since that is where prediction and ground truth coincide. Confusion matrices are good for visualizing which classes are easily mixed up, or if any model has bias towards certain classes. In the bottom row of each table the total number of predictions for each class is found, and in the right column the total number of ground truth labels.

		Predicted				
	Classes	Pedestrian	Cyclist	Car	Other	**Total**
True	Pedestrian	1922	23	0	20	1965
	Cyclist	15	2008	0	10	2033
	Car	0	0	2012	5	2017
	Other	41	32	7	3 905	3985
	Total	2048	2063	2019	3940	10000

Table 5.2: Confusion matrix for the small single scale model.

		Predicted				
	Classes	Pedestrian	Cyclist	Car	Other	**Total**
True	Pedestrian	2000	10	0	19	2029
	Cyclist	8	1987	0	10	2005
	Car	0	2	1988	14	2004
	Other	17	33	1	3911	3962
	Total	2025	2032	1989	3954	10000

Table 5.3: Confusion matrix for the big single scale model.

		Predicted				
	Classes	Pedestrian	Cyclist	Car	Other	**Total**
True	Pedestrian	1944	26	0	16	1986
	Cyclist	15	1940	0	44	1999
	Car	0	0	2008	3	2011
	Other	86	19	13	3886	4004
	Total	2045	1985	2021	3949	10000

Table 5.4: Confusion matrix for the small multiple scales model.

		Predicted				
	Classes	Pedestrian	Cyclist	Car	Other	**Total**
True	Pedestrian	1994	17	0	2	2013
	Cyclist	2	2027	1	7	2037
	Car	0	0	1993	1	1994
	Other	39	21	9	3887	3956
	Total	2035	2065	2003	3897	10000

Table 5.5: Confusion matrix for the big multiple scales model.

Finally the accuracy versus number of points in the object for pedestrian classification was measured, see Figure 5.1. For each number of points 1000 classifications were made, and the accuracy depicts how many times the models predicted pedestrian correctly. Since the number of points in an object depends on its distance to the LiDAR sensor, it is an estimation of how good the models work on people far away.

In Figure 5.2 are some cases were the classification models failed to predict the correct label. In most cases we can see why the classification of these objects was tough.

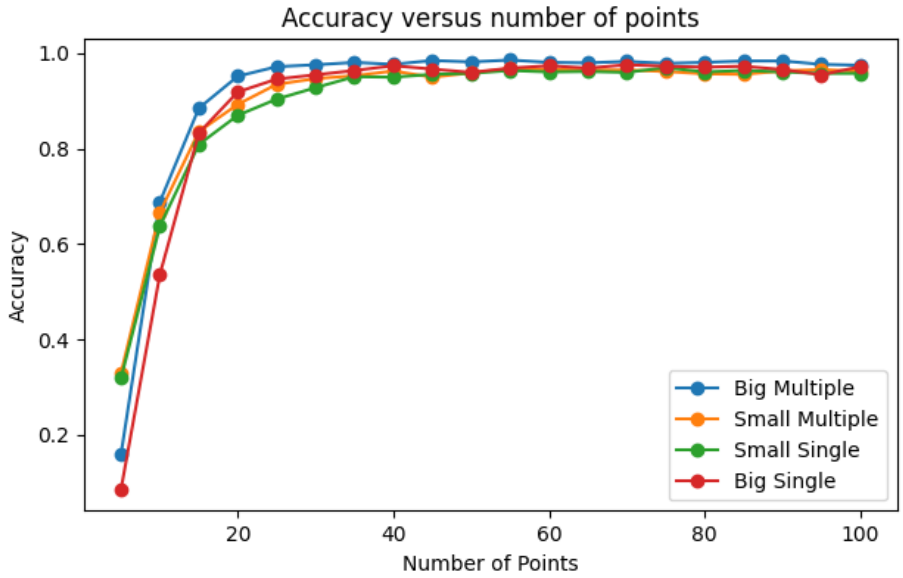
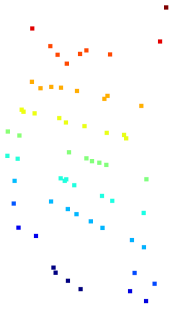
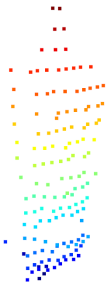


Figure 5.1: Classification accuracy versus the number of points in the pedestrian.

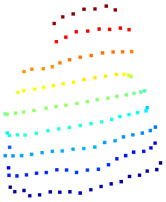
(a) Predicted: pedestrian. True: cyclist.



(b) Predicted: other object. True: cyclist.



(c) Predicted: pedestrian. True: other object.



(d) Predicted: cyclist. True: pedestrian.

Figure 5.2: Four objects the classification algorithms failed to classify correctly.

5.2 Semantic Segmentation Models

In Table 5.6 we can see the two semantic segmentation models evaluated with the metrics described in the Method chapter. The bold numbers are the best results in each category.

Model	Accuracy	Precision	Recall	F1-Score	Inference Time (ms)
Small, Semantic Segmentation	0.8759	0.6702	0.7913	0.6777	**14.66**
Big, Semantic Segmentation	**0.9593**	**0.8528**	**0.9551**	**0.8853**	16.87

Table 5.6: Evaluation metrics for the semantic segmentation models. Bold is best result in each category.

Tables 5.7 and 5.8 contain the confusion matrices for the two semantic segmentation models. These work in the same way as for the classification case, but instead of each object generating 1 output, each frame generates 2048 outputs, which is why the numbers are much bigger.

		Predicted					
	Classes	Pedestrian	Cyclist	Car	Animal	Background	**Total**
True	Pedestrian	187 100	73810	17 392	38 690	15 482	332 474
	Cyclist	43 488	233 884	41 234	25 406	4 369	348 381
	Car	5 397	17 291	547 741	13 674	3 936	588 039
	Animal	1 060	1526	1336	32 953	3 503	40 378
	Background	50 891	140 65	18 519	134 623	2 568 630	2 786 728
	Total	287 936	340 576	626 222	245 346	2 595 920	4 096 000

Table 5.7: Confusion matrix for the small semantic segmentation model.

		Predicted					
	Classes	Pedestrian	Cyclist	Car	Animal	Background	**Total**
True	Pedestrian	308 953	13 360	1 680	10 773	4 581	339 347
	Cyclist	15 249	320 624	1 937	2 518	1 075	341 403
	Car	3 795	6 655	586 297	2 471	1 969	601 187
	Animal	378	284	55	40 405	452	41 574
	Background	13 196	6 709	6 225	32 317	2 714 042	2 772 489
	Total	341 571	347 632	596 194	88 484	2 722 119	4 096 000

Table 5.8: Confusion matrix for the big semantic segmentation model.

In Figure 5.3 we find number of points versus accuracy for semantic segmentation of pedestrians. It cannot be calculated the same way as for the classification models, due to the output shape being a vector instead of a scalar. Instead if the majority of the points are given the pedestrian label the segmentation is considered correct.

In Figure 5.4 we see two cases where the semantic segmentation was rather poor, with less than 75 % of points correctly labeled. In this segmentation gray is background, red is pedestrian, green is cyclist, black is animal and blue is car.

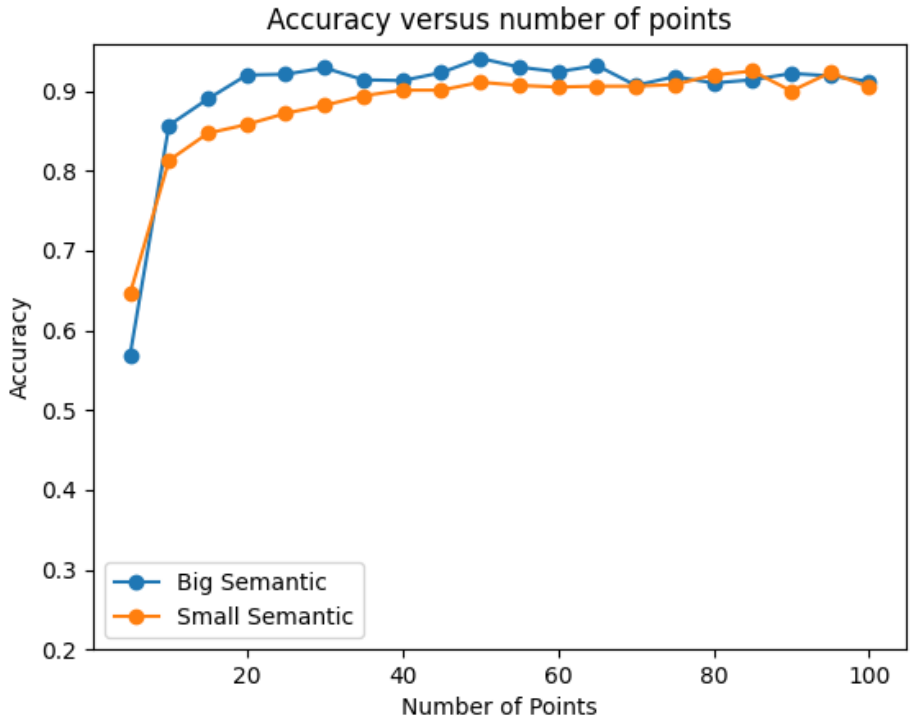
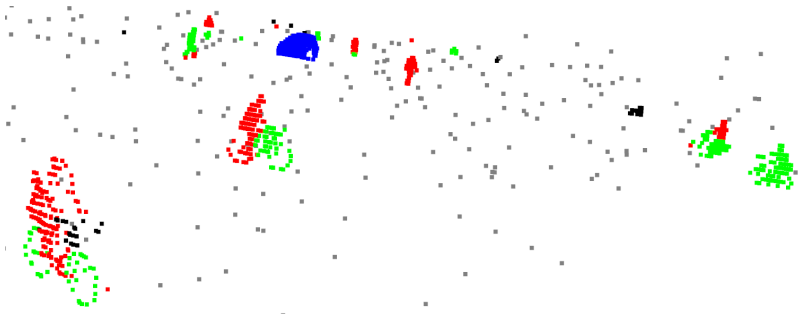


Figure 5.3: Semantic segmentation accuracy versus the number of points in the pedestrian.



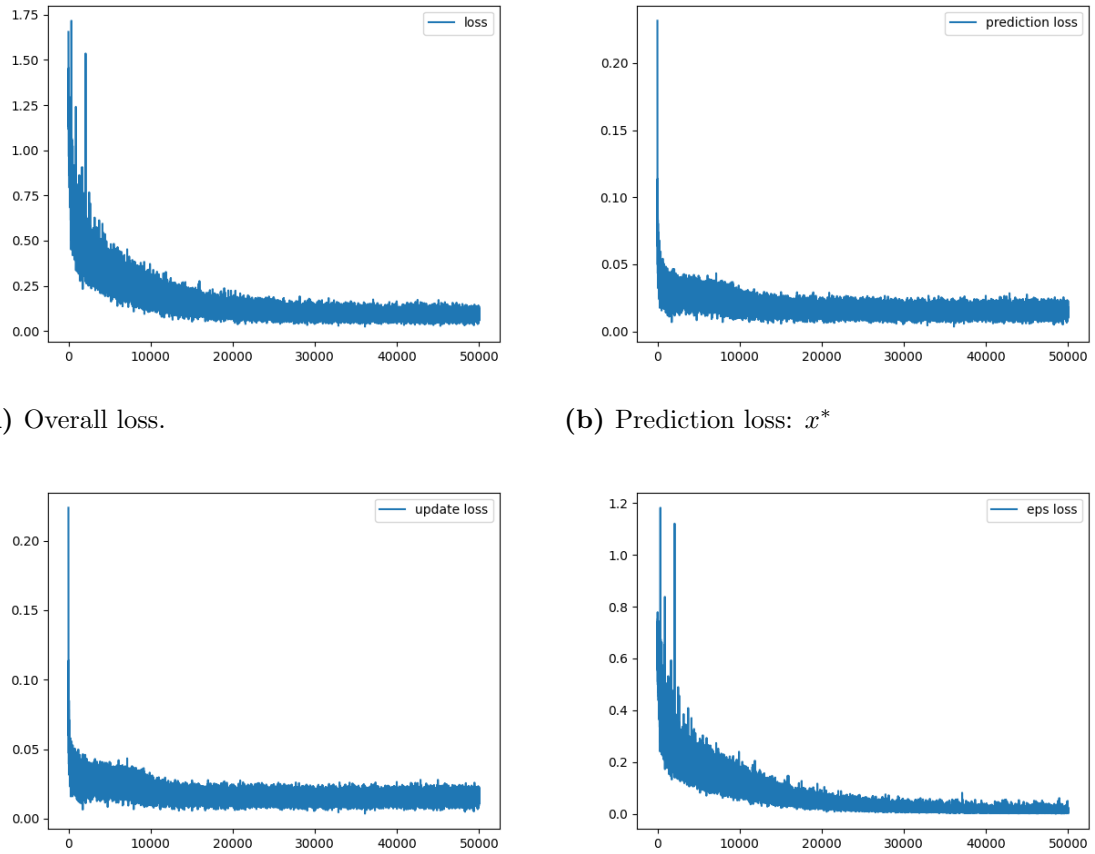
(a) Semantic segmentation with less than 75 % accuracy.

(b) Semantic segmentation with less than 75 % accuracy.

Figure 5.4: Two cases where semantic segmentation had well below average accuracy.

5.3 Tracking Results

Below in figure 5.5 are the convergence curves for the Predict/update module. The initial impression of these plots should probably be that the model seems to have converged properly towards solving its task. While this may be true in some regard, it will below become apparent that the model unfortunately had not converged properly. We analyzed this further in section 5.3.1 below.



(a) Overall loss.

(b) Prediction loss: x^*

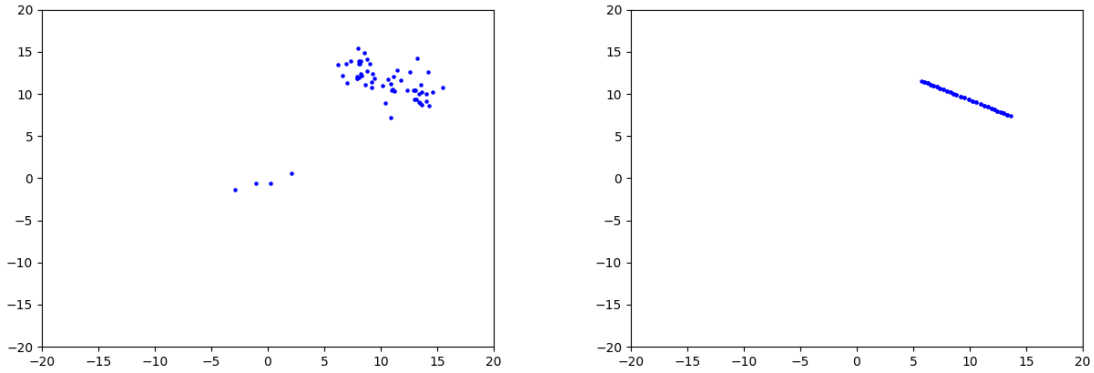
(c) Update loss: x

(d) Tracks probabilities loss: ε

Figure 5.5: Loss curves from training the Predict/update module.

The trained Predict/update module, despite appearing to have converged, did not behave properly. We could narrow the down towards the prediction submodule, and how it did not predict and update the tracks properly. That a poor prediction affects the update output was itself not a surprise, seeing how they shared the same loss function and had very similar loss curves in both shape and magnitude (Figure 5.5). The problem is illustrated in Figures 5.6 and 5.7, despite the loss having converged significantly, the predictions were still off by several meters. As explained before, a "false track" should be close to (0,0), which we can see in practice in the two figures – "track 0" reported somewhat close to (0,0) during a few seconds before the track entered the scene. In these examples, a single track was randomly spawned within 1 minute. The model was in Figure 5.6 continuously fed with ground truth data x

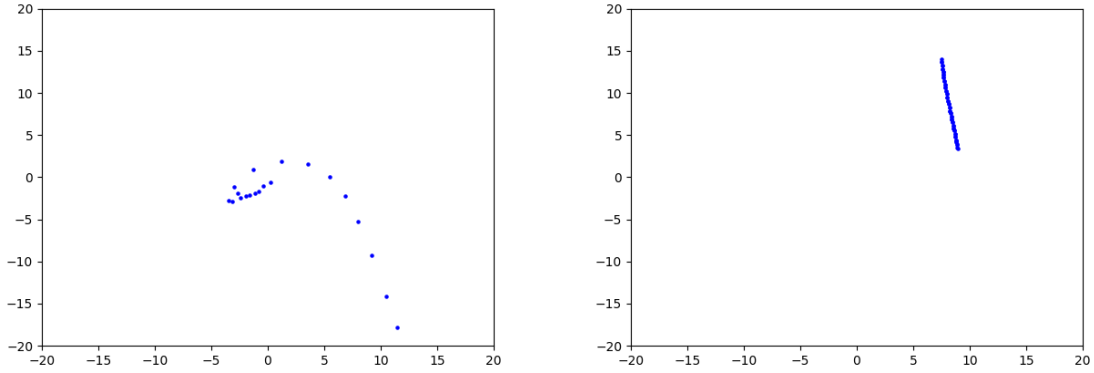
and ε in each timestep to give it the best of possible chances. It indeed gave off a MAE ≈ 0.009 similar to the value the model had converged into. Taking normalization into account, this corresponded to approximately 0.9 m, which however was not close to what could be observed in practice in the figure. In the second example in Figure 5.7, the module was continuously fed its previous output x_t and ε_t in the next timeframe, which is how we thought the model should be implemented in practice – however the prediction diverged out of control and yielded a MAE $\approx$ 20m after 1 minute. At this stage we deemed that a comparison with a Kalman filter unnecessary as the module was unable to solve its task.



(a) Predicted path of a single track.

(b) Ground truth of track.

Figure 5.6: Predicted track of a 1-minute sequence where the Predict/update module was fed with ground truth data for updated x and track probability ε.

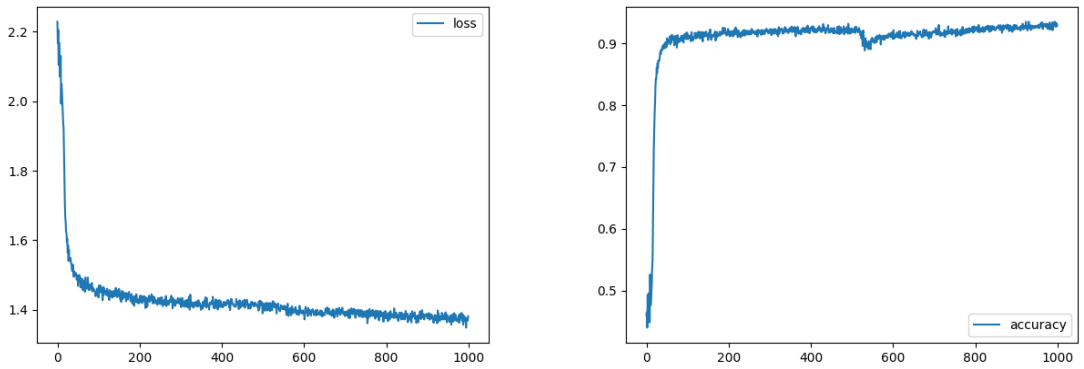


(a) Predicted path of a single track.

(b) Ground truth of track.

Figure 5.7: Predicted track of a 1-minute sequence where the Predict/update module was fed with its own output x_t and ε_t in the next timestep $t + 1$.

We can see that the DA module had converged properly, as evidenced in figure 5.8. This in turn indicated that while the Predict/update module was infeasible for the task, the DA module could still be evaluated – however with a Kalman filter used for prediction and updating.



(a) Loss **(b)** Accuracy

Figure 5.8: Loss and accuracy curves from training the DA module.

We implemented the two baseline models introduced in section 4.6.2 together with a third model, in which we used the trained DA module to solve the data associations in each step. the baseline models, as we described earlier, consisted of a Kalman Filter and the Hungarian Algorithm, in which the latter was adjusted by incorporating a rejection condition and track probabilities with decay. We tested the models on 100 different 60-seconds sequences, created in the same manner as when creating training data (see section 4.5.3).

As we can see in table 5.9, all three models were not too different from each other in terms of the first four metrics. We can note that the first baseline model seemed to balance precision and recall well, while also having the highest accuracy, however with a somewhat cost into precision. The data association in the two other models seemed to associate measurements to current tracks almost perfectly, although at a cost of not finding all tracks. This does make sense considering the second baseline model, where the rejection condition ultimately "filters out" false associations. We should here however note that the DA module did not seem to properly learn 1-1 associations. Finally, while the data association solving operated within the time budget of 0.1 seconds in the final model, the two other models considerably outclassed it.

Model	**Accuracy**	**Precision**	**Recall**	**F1-score**	**1-1 associations**	**Inference Time (ms)**
kalman_hu	0.915	0.930	0.974	0.944	100 %	0.008
kalman_hu_adj	0.868	1.000	0.842	0.906	100 %	0.008
kalman_lstm	0.887	0.986	0.873	0.919	76.7 %	15.62

Table 5.9: Data association metrics, averaged over frames in 100 60-seconds sequences.

Results from the overall evaluations can be seen in table 5.10. While these metrics should not be seen as the absolute determination of the "best" model, the simple baseline model seemed to perform the best. One very interesting result in all these cases is that a track does not necessarily have to be considered "mostly tracked" or

"mostly lost", since these only encompass the extremes of the scale (more than 80 % or less than 20 %). Nevertheless, all three models seemed to have a rather strict relation to the evaluation set – either a track was found by the tracker, or it was not.

Model	Mostly tracked	Mostly lost	Inference time (ms)
kalman_hu	82.5 %	17.5 %	0.494
kalman_hu_adj	19.7 %	80.3 %	0.535
kalman_lstm	40.5 %	59.4	15.94

Table 5.10: Tracking metrics, averaged over 100 60-seconds sequences.

5.4 Pipeline Results

From a qualitative perspective by applying the models on real recordings, the kalman_hu_adj seemed to work the best. The simple baseline kalman_hu had quite a few ID swaps from time to time, which probably was due to a measurement gone missing for a single frame. The kalman_lstm tracker had similar problems, but particularly showed issues with objects walking too close together. As such, a few examples from the kalman_hu_adj tracker is shown below together with classifying tracked objects, which includes background filtration, clustering, classification or semantic segmentation, and tracking, as well as visualization. The dark gray points are filtered away background points. Each cluster and tracked object is represented by a bounding box, and the color of the object is the classification. Red is pedestrian, green is cyclist, blue is car, yellow is other and grey is uncertain. For consistency and robustness each tracked object needed to receive the same predicted label 8 times in the previous 10 frames to obtain its classification. In Figures 5.9 and 5.10 we see the big multiple scales classification algorithm at work while in Figure 5.13 we see the small single scale model. In Figures 5.12 and 5.11 we see the big semantic segmentation network, with the classical tracker.

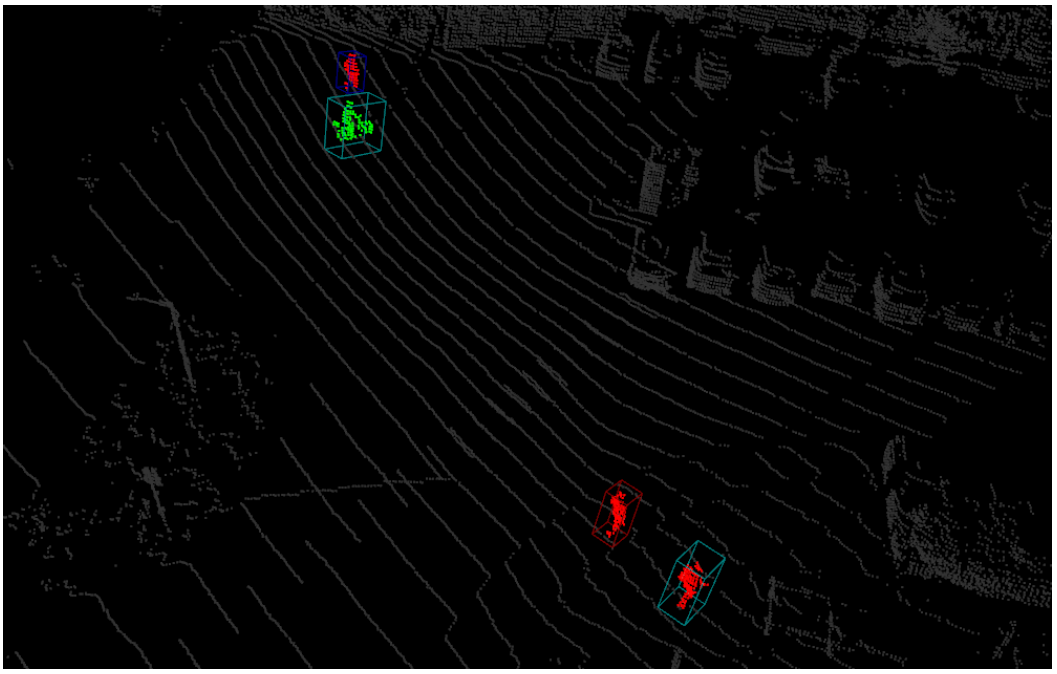


Figure 5.9: Tracking and classification using the big multiple scales model in real life LiDAR data. Three people and one cyclist are tracked and correctly classified.

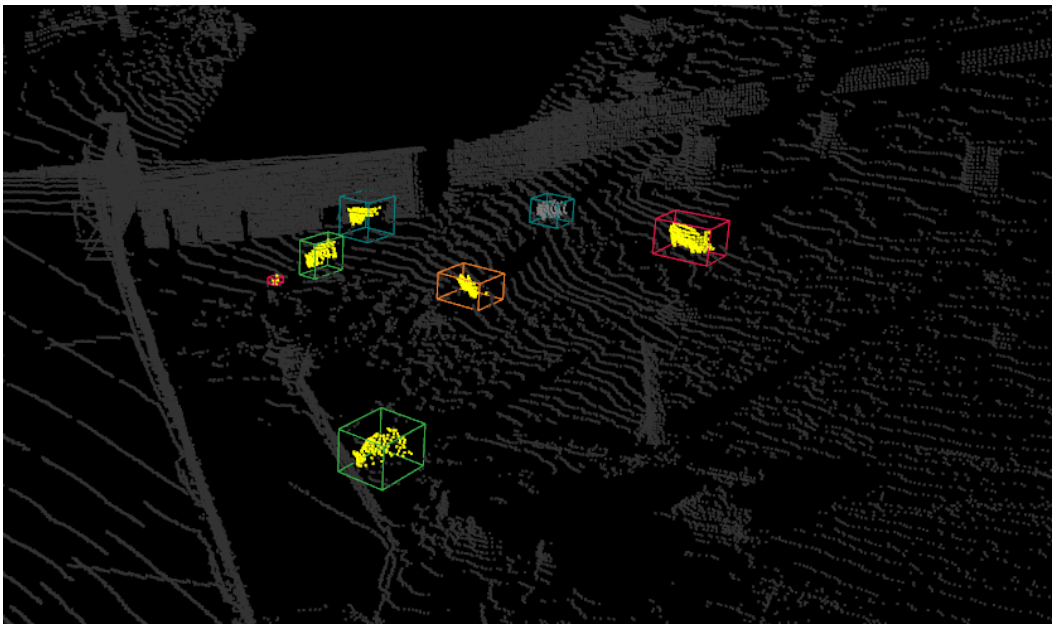


Figure 5.10: Tracking and classification using the big multiple scales model in real life LiDAR data. Here we see a few pigs being classified as "other" objects.

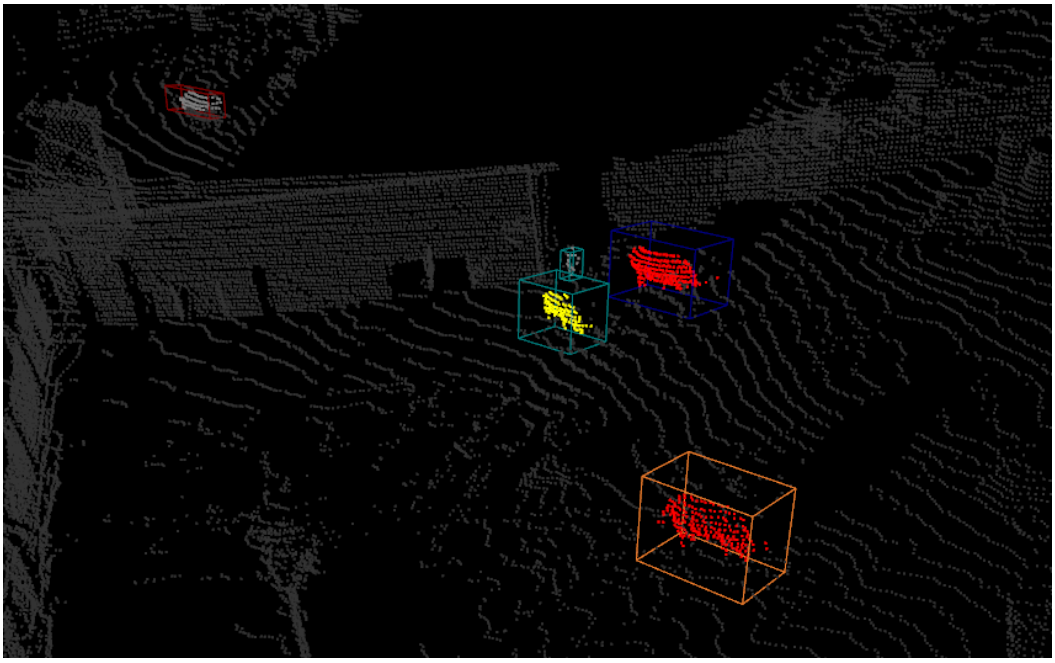


Figure 5.11: Tracking and classification using the small single scale model in real life LiDAR data. Here we see a few pigs being classified as people, "other" and uncertain. Later in the video the same pigs often changed classification, also showing as cyclist.

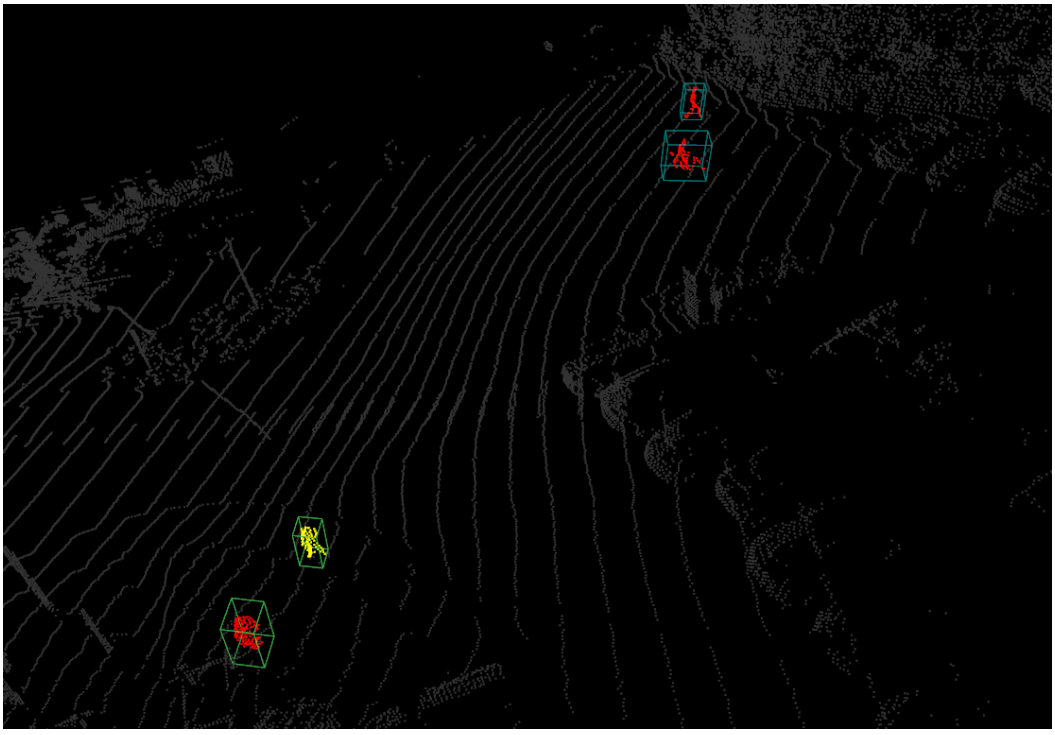


Figure 5.12: Tracking and semantic segmentation using the big model in real life LiDAR data. Here we see two people correctly classified, one person classified as "other", and one cyclist classified as pedestrian.

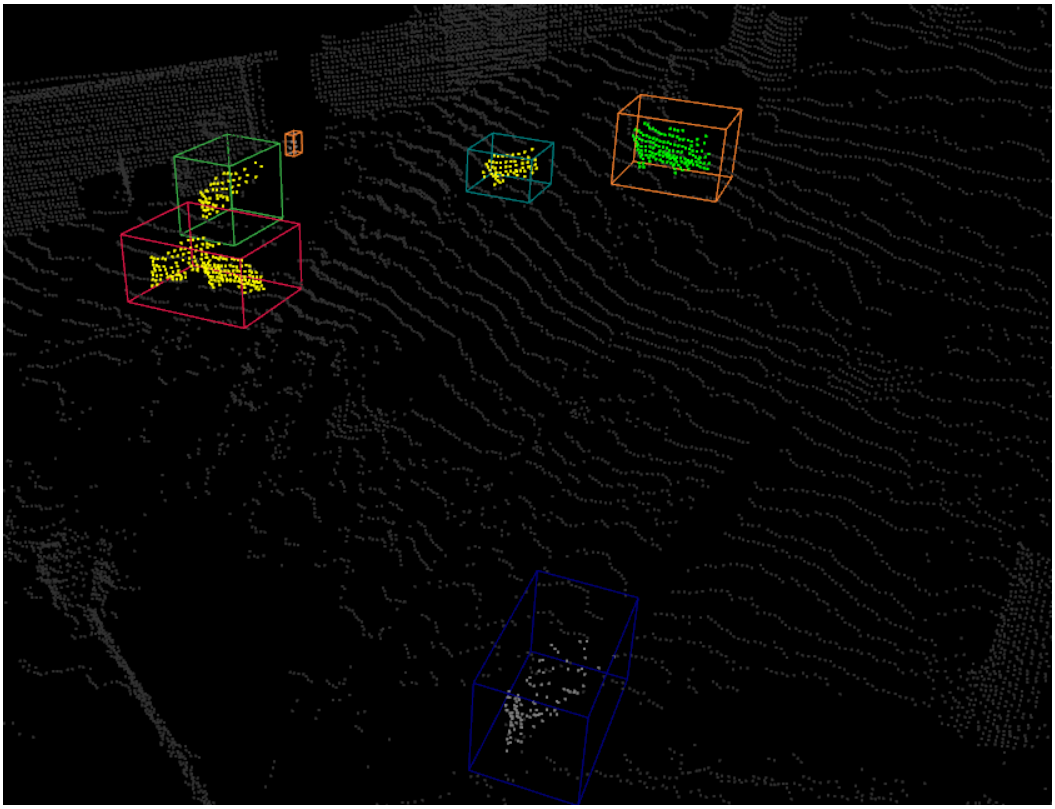


Figure 5.13: Tracking and semantic segmentation using the big model in real life LiDAR data. Here we see a few pigs being classified as "other" objects, but one is also classified as a cyclist.

6 Discussion

6.1 Filtering and Clustering

Below follows a bit of discussion on the background filtration and clustering. Even though it was not the primary focus, it was an essential part of the pipeline, which the classification and tracking built upon. This discussion is based on qualitative study of the algorithms on real world LiDAR data, see Section 5.4.

Under the right conditions the background filtration worked great, for example when filming the pigs in Figure 5.10. There was no wind, and the pigs were not in the picture for the first few seconds, so a good background model could be calculated. There was almost no noise, which is thanks to the smoothing of the filter, the threshold and the amazing quality of the LiDAR. On the film seen in Figure 5.9 the background filtration was passable. There was a lot of wind, which meant a lot of objects were detected in a nearby tree. There were also time when the whole LiDAR moved, which created a lot of artifacts. Thankfully the classification and tracking algorithm discarded the artifacts fast.

The clustering also worked very well most of the time. It was robust to noise, it was fast and in most cases the clustering was correct. One thing it could not handle was if two objects were in close proximity to each other, since any distance less than $\epsilon = 0.5$ meant they were the same object. This was often the case with people walking together, and we also see it with the pigs in Figure 5.13. Another problem was that ϵ was the same for all distances, since farther from the LiDAR the point clouds were more sparse.

Both the background filtration and clustering work as well as expected, but have potential for improvement, which is discussed in later in chapter Future Work.

6.2 Classification Models

In this part the results from the classification models, Section 5.1 are covered. Four models were compared, all trained and evaluated on the same simulated dataset. They were also tested on real world data, captured in Lund using the Luminar Hydra LiDAR.

6.2.1 Evaluation on Test Data

We can immediately tell that all four models work well on classifying objects from the same type of simulated data that they were trained on. In Table 5.1 we see for example accuracy and f1-score, which are above 97 % for all models. The big multiple scales

model (BMSM) unsurprisingly has the highest score in each category, surpassing 99 % in both accuracy and f1-score. Interestingly the lowest score in each category is held by the small multiple scales model (SMSM), indicating that it had too few parameters to fully utilize the multiple scales feature.

By studying the confusion matrices in Tables 5.2, 5.3, 5.4 and 5.5 we get some more insight about the models. All models sometimes confuse cyclists and pedestrians, which is expected since sometimes only the top half of the object is visible, meaning they look identical, see Figure 5.2d for an example of this. Cars are the easiest object to classify for all models, and are almost never confused for cyclists or pedestrians. They are sometimes confused for objects in the "other" category, since they can have similar shapes and sizes to walls or other static objects. The hardest category to identify, creating both false positives and false negatives is the "other" category. This is to no surprise since it is the most diverse, where objects similar to other classes can occur.

Accuracy versus the number of points was another metric we were interested in, see Table 5.1. There is a clear trend of accuracy increasing with each increment of 5 points, up to above 95 % at around 40 points. It is also clear that the BMSM needs fewer points than the others to do an accurate classification. Interestingly the SMSM beats the small single scale model (SSSM) at fewer points, while having a bit lower accuracy overall. This shows what the multiple scales feature still serves some purpose in the small model.

In Figure 5.2 we see four cases where the models failed. In Figure 5.2a we see an incorrectly labeled cyclist, but most of the bike is not visible so it looks like a pedestrian. In Figure 5.2b we instead see only a piece of the bike, which is incorrectly classified as "other", it seems to be a problem that the person riding is missing. Figure 5.2c shows an "other" object with a human-like structure, being classified as a pedestrian. Finally in Figure 5.2d we see an upper-body being classified as a cyclist, which belonged to a pedestrian. Most of these are not surprising, but shows that having humans represented in two different classes is sometimes a problem.

In Table 5.1 we also see inference time, which is the time it takes for one forward pass through the network. The biggest network took 11 ms while the smallest only needed 6 ms. Clearly the multiple scales feature takes a lot more processing time, requiring almost double. It also shows that the number of parameters in each layer barely affects the inference time, but this is a bit misleading. The GPU calculates a lot of things in parallel, so the bigger model uses more of the GPU when it is doing inference. This means we could use bigger batch sizes for the smaller networks, bringing down the calculation time drastically for many classifications. In the pipeline it can vary how many objects need to be classified each frame, so classifications can't always be done in big batches.

6.2.2 Evaluation on Real World Data

In Section 5.4 there are some examples of classification in real world LiDAR data, which ultimately is what the models are supposed to be used for. In these tests

the classification rule was that an object had to have the same predicted label for 8 out of 10 frames to receive its classification. This means that a model needs to consistently predict the same label as the tracked object moves, creating a more robust classification. In Figure 5.9 there are three people and one cyclist correctly classified by the BMSM, which shows it consistently classified them correctly more than 80 % of the time. In Figure 5.10 we instead see it working on pigs, classifying them as other, which is great considering it was not trained on pigs. In Figure 5.11 we instead see the SSSM doing predictions on the same data, but with poor results. The Pigs are classified as humans, cyclists, other and sometimes not at all.

The models sometimes struggled classifying people who didn't move like in the simulation. For example when people put their arms up, crouched or did any other odd movement. They also always failed when two or more people were clustered to the same object, which was expected but unfortunate. In the real data objects were more often obscured, either by other objects or the environment, which also made them tougher to classify.

6.2.3 Evaluation of Dataset

The foundation of all models was the dataset generated with CARLA. As such, how well the simulated data replicates real world LiDAR data obviously has a huge impact on how well the models work. In many of the real world cases the models work well, even for types of data not encountered during the training, which is good. However, large improvements to the classification models could be made with a better dataset. Below are some observations and thoughts on the dataset.

One of the biggest drawbacks of the simulated data is that it does not contain any returned beam intensity. When studying the real world data it is clear that the intensity in each point holds a lot of information, for example skin, clothes, animals and vegetation have different reflectivity. Having access to this data could definitely improve classification, and it could most likely be used in background filtration and clustering as well. Another drawback is the lack of different animations in people, which was mentioned previously. More diverse training data would make the classifiers more robust to odd poses and movements.

For many use cases it is important that the models recognize animals, so that they do not incorrectly classify them as humans. In CARLA there were no animals per default, and they were difficult to add in a realistic way. For many cases the "other" category proved to be enough. Since it was very diverse, a previously unseen object had a higher chance to look like the "other" class than a human or a cyclist, at least using the BMSM. The "other" category was also good at catching swaying trees or other noise, which was positive.

The best option would of course be if there was annotated real world data for all classes, but it is a very time consuming process to manually annotate. One potential solution is to use this pipeline for object detection and classification as a way to annotate the data, then manually check if the data looks correct. The other solution is to simply make the simulator more advanced, with more animations, more objects and material

reflectivity.

6.2.4 Conclusions

All models work very well on simulated data, which suggests that the data is too simple. The biggest model could have handled more advanced data than what was available. Either real world data or a better simulator could solve this issue. Even with good results on the training data, the SSSM is much worse at generalizing its training to real world data than the BMSM. It seems it is not worth scaling down the number of parameters in the models, since the results get worse and we do not save much time. The multiple scales feature is also worth using, showing much better results on real world data.

6.3 Semantic Segmentation Models

In this part the results from the semantic segmentation models, Section 5.2 are covered. Two models were compared, both trained and evaluated on the same simulated dataset. They were also tested on real world data, captured in Lund using the Luminar Hydra LiDAR.

6.3.1 Evaluation on Test Data

Semantic segmentation is a much tougher problem to solve than classification, for several reasons. Each point in the whole object should receive its own label, instead of one joint classification. There is no telling where one object starts and where the next begins, and there is no telling how many objects are in the picture. We therefore expect lower accuracy and f1-score for these models overall.

The evaluation scores can be found in Table 5.6. Accuracy looks good for the big semantic segmentation model (BSSM), and a bit worse for the small semantic segmentation model (SSSM). However the f1-score looks worse for both models, due to the precision, especially for the SSSM. The reason for this is revealed in the confusion matrices.

In Table 5.7 we see the confusion matrix for the SSSM. The first thing to note here is the bias towards predicting animal, with 6 times as many predicted animal points as actual animal points. Most of these points come from the background class, but also from pedestrian and cyclist. Cyclist and pedestrian also have very bad scores, the accuracy is only saved by car and background, being the biggest classes. In Table 5.5 we instead see the confusion matrix for the BSSM. The results here are better, but the bias towards predicting animal is still prevalent, predicting it twice as often as it should. In addition it mixes up pedestrian and cyclist points sometimes.

What has happened here is when we introduced the weighted loss function to steer the bias away from the common classes, we instead introduced a bias towards the least

common classes. Since the models were penalized for picking animal, they steered towards picking it more often to minimize the loss function.

If we look at Figure 5.3 we see accuracy versus the number of points in the object. The models performed fairly well at this task, but the accuracy increase is not as stable as for the classification models. Sometimes adding more points result in a worse segmentation. It could be that more points result in the object being split up by the segmentation models, which is not a problem in with the classification models.

In Figure 5.4 we see two of the worst test cases by the BSSM, with total accuracy score less than 75 %. These images highlight many of the problems with the semantic segmentation models. Firstly it is even harder for the BSSM to distinguish cyclists from pedestrians than it is for the classification models. This is because semantic segmentation can split the person from the bike, giving the person a pedestrian label and the bike a cyclist label. This is a case of the model being too clever in a way, since it recognizes the human riding the bicycle. It is also possibly a flaw in the dataset, which we will cover below.

Another issue, which is clear from Figure 5.4b, is that animal points are being input in many places where they should not be. In this figure two of the cyclists, the car and the background have animal points. From these images we see that the BSSM has a hard time giving one object only a single label. It seems some object detection or instance segmentation is necessary for better results.

In Table 5.6 we also find inference time, which is about 15 ms for the SSS model and 17 ms for the BSS model. These are acceptable computation times for one frame. As mentioned earlier it is not surprising the big model has similar inference time to the small model, as it simply uses more of the GPU when inferring. These computations cannot be done in batches, since only one inference per frame is needed.

6.3.2 Evaluation on Real World Data

To see performance on real world data for the BSSM see Figures 5.12 and 5.13. Unfortunately the flaws displayed in the last section are only amplified on the real world data. In the first image we see two people correctly classified, but another person classified as background and a cyclist as a pedestrian. In second image we find that the pigs are mostly correctly classified, but one also predicted to be a cyclist. The real world data is more difficult, but the results here are not passable for any real application. We believe some problems lie in the dataset and how the models were trained, which is discussed below.

6.3.3 Evaluation of Dataset

Much of the discussion on the classification dataset holds for this dataset as well. CARLA is a limiting factor in many ways, having limited number of objects and animations and no return beam intensity. A real world dataset would have solved a few issues. That said, there were some problems with how we created the CARLA

dataset.

The class imbalance was a major issue when training the models, there was an overrepresentation of background points and an underrepresentation of animal points. We attempted to handle this problem by introducing training weights, but they introduced other problems instead. This created models that were biased towards animals due to the training weights, and biased towards background due to the overrepresentation. In hindsight the animal class should have been removed, and the background class should have been made smaller to match the other classes.

Another issue was the similarity between cyclist and pedestrian, the model often split up the cyclist or misclassified the pedestrians. A solution could be to split the cyclist class into pedestrian and cycle, as to not confuse the model so much. Then if a cluster contains both cycle and pedestrian points it could be considered a cyclist. Another solution is to put pedestrian and cyclist in the same class, since they have many common features.

6.3.4 Conclusions

The semantic models were overall significantly worse at labeling than the classification models. This was no surprise, due to the harder task, with no help from the clustering algorithm. However the models suffered a lot from how the dataset was constructed. We suspect that a big increase in performance on real world data could be achieved a better dataset. The small model was significantly worse than the bigger model, for this task the total number of parameters played a big role in how well the models performed.

6.4 Tracking models

A few issues occurred in the results in the different tracking modules. the largest issue was of course the Predict/update module which did not seem to converge properly. The DA module similarly had some issues itself, both standalone and in comparison to the baseline models.

6.4.1 Predict/update module

While the model *seemed* to have converged properly from the looks of the convergence curves, it still could not properly solve its task. We have two main theories as to why this happened, with the first being the simplest. Primarily, it could simply have been an issue of not having found the "correct" set of hyperparameters or training schedule. This would have meant that while the model had the potential of converging, it still had some way to go. This is in itself a difficult task to confirm since there are so many different alternatives to try out. Considering however how small the losses were, there did not seem to be much more space for the model to learn in the current setting.

A second theory essentially expands upon the aforementioned observation. One perhaps key aspect to the training was that while we implemented BPTT, in each timestep over a training schedule the model was fed with ground truth data in the input channels. Perhaps this led to the model finding a solution which fit well with receiving the "correct" input (as close to the small Gaussian noise added) in the next timestep, regardless of its output? Implementing the module would highly suggest this, but does not necessarily explain how the loss still converged. Would it perhaps have been better to build a model which solely took measurements z as a sequence, and in each timestep reused its former output as input? In each timestep the model could have output its updates (commonly referred to as "time distributed" outputs), which would then be compared to the ground truth data. This would expose the model much more to how it would be used in practice. Maybe even more so, as discussed in section 4.4.3, this may have been necessary in order for the update weights to converge properly and ensure that they fulfilled their (at least theorized) purpose of self-correcting tracks while the tracks probabilities went up or down until a certain threshold. None of this is unfortunately discussed by Milan et al. [20] and neither explored by us, the latter being due to lack of time.

6.4.2 DA module

The DA module converged properly and furthermore seemed to be good at solving the data association task. There were of course some flaws, one of them mainly being that it had not seemed to learn 1-1 associations, whereas the network structure of an LSTM would suggest that it surely had the possibility. One possibility could have been that the training data was not general enough for the module to learn this. Another way of forcing it to learn 1-1 associations could have entailed some sort of loss term penalizing this behavior. Finding a continuous differentiable function to represent 1-1 associations is however not trivial. We must also be note that this module takes a substantial amount of time to calculate its solution when compared to the Hungarian algorithm, suggesting that it should probably be much better than HA at solving the task in order to be "worth it". One key aspect to look into could be how well both of the models scales when there are a lot more tracks and measurements to associate.

The qualitative study of real LiDAR data pointed at the module performing subpar with objects close together. This behavior can not stem from the training data, since its variety yielded both sparse and crowded simulated tracks. Let's instead consider the input matric C, which consists of the distances between track probabilities and measurements. Since neural networks requires fixed feature vector sizes, the measurement array z was forced to be fixed. The solution to this was to simply make it a fixed size of $M = 15$ measurements, but of which not all element in it were necessarily filled with actual measurements; it would instead at "empty" elements contain (0,0)-positions. This could in effect yield, for each row (which corresponded to a track), a set of distances in C that were equal – all of those distances stemming from "none" measurements. When a few objects walk too close together, the (almost) same structure appears in the C-matrix – several distances equal to one another. Perhaps this could have led the module to believe that these measurement were "false", when in reality they were not? This explanation is enforced by how the model often in these cases

returned "missing measurement" in these cases. Perhaps further training on crowded tracks could remedy this. Considering how the module works comparatively just as well as the baseline models, fixing this issue may just increase the metrics somewhat more.

6.4.3 Conclusions

While not having reached all the way, there still seems to be some final leads in the quest of finding a good deep learning-based tracker. For the time being however, the classical approach of implementing a Kalman filter together with the Hungarian Algorithm stands strong in the evaluations. There is certainly no doubt why these methods entails the classical approach of tracking.

A full on study of the track probability and decay parameters was not made here, neither was it the purpose of this project. It should probably be mentioned that exploring this area further would probably make the adjusted baseline model even better.

7 Conclusion

The conclusions we can draw about the entire pipeline are as follows. Filtering and clustering worked well enough for us to adequately test the different models on real world data. The methods were quite simple, with room for improvement, but it was not the target of this thesis. Cluster labeling is best left to the classification algorithms, as semantic segmentation unnecessarily complicated for the task. The best classification network for this task is the big multiple scales model, which outperforms the other models without spending too much time. Training on the simulated data for classification creates decent models, but more complex and realistic data is needed for further improvement.

Using neural networks to to tracks objects remains a challenge, but a promising one. Considering that earlier studies have achieved better results than us on this area, together with comparable performance in the data association area, some hope remains. This study has furthermore shown how prerecorded tracks (simulated or non-simulated) can be augmented with some simple transformations in order to "expand" the dataset. This can probably help training tracking models in the future.

8 Future Work

Following the work on this thesis, one obvious project moving forward would be to train the Deep Tracker properly. The easiest approach would probably entail programming the Predict/update-module as an RNN with the hidden state containing the SRNN vector h as well as the outputs x, z and ε – meaning that in each timestep we would only input new measurements. Formalizing the module in this way would probably allow the BPTT to act properly and as such train the module in the same way as it would be employed.

As stated before, the classifiers evaluated in this thesis would most certainly become even better by having access to real world training data. Given that we now have a working pipeline, we could create an unsupervised annotation process by extracting the point clouds belonging to the same tracks as found by the tracker. Say for example that we would have a track where our pipeline has classified the tracked object as a pedestrian 90 % of the time. Having saved the point cloud for the object in each timestep, there exist at least[1] 10 % of point clouds that the classifier does not recognize as pedestrians but probably should. A full study could be made on the subject, trying to answer questions regarding how well such an iterative training could make the classifier and within which conditions the training performs as best.

The preprocessing steps are currently pretty basic, and there are several areas to improve upon. One direction to take is to incorporate deep learning as early as at the object detection step, i.e. doing instance segmentation using a neural network. The authors of *Associatively Segmenting Instances and Semantics in Point Clouds*, see [22], do precisely this by using a Siamese neural network that does both instance and semantic segmentation. This way each object is simultaneously detected and classified. Training such a network to work better than our current network would be challenging however.

A more straightforward approach is to find new ways to filter and cluster points. A big problem with clustering objects in LiDAR data is that the point density naturally varies across the scene, making it hard to choose ϵ. The authors of *An improved DBSCAN method for LiDAR data segmentation with automatic Eps estimation*, see [23], describe a method of automatically estimating ϵ. Not needing to adjust ϵ for each new scene is a great improvement for the clustering. There are also many ways to segment the background, and filters can be stacked for improved effect. A more dynamic background model that changes over time would be desirable.

[1] While we are at it, why not take all the point clouds in the track, including those that have been classified as pedestrians?

Bibliography

[1] Jacob Berntsson and William Winberg. *Pedestrian detection and tracking in 3D point cloud data on limited systems*. eng. Student Paper. 2021.

[2] Matt Weed. *Sensor(y) overload: Making sense of lidar*. Jan. 2021. URL: https://www.luminartech.com/sensory-overload-making-sense-of-lidar/.

[3] Luminar. *Hydra Specifications*. Jan. 2021. URL: https://www.luminartech.com/thank-you-hydra/.

[4] W. Xiao, B. Vallet, K. Schindler and N. Paparoditis. 'Simultaneous Detection and Tracking of Pedestrian from Panoramic Laser Scanning Data'. In: *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences* III-3 (2016), pp. 295–302. DOI: 10.5194/isprs-annals-III-3-295-2016. URL: https://www.isprs-ann-photogramm-remote-sens-spatial-inf-sci.net/III-3/295/2016/.

[5] Martin Ester, Hans-peter Kriegel, Jörg Sander and Xiaowei Xu. 'A density-based algorithm for discovering clusters in large spatial databases with noise'. In: AAAI Press, 1996, pp. 226–231.

[6] Ian Goodfellow, Yoshua Bengio and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[7] Charles R. Qi, Hao Su, Kaichun Mo and Leonidas J. Guibas. 'PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation'. In: *arXiv e-prints*, arXiv:1612.00593 (Dec. 2016), arXiv:1612.00593. arXiv: 1612.00593 [cs.CV].

[8] Charles Ruizhongtai Qi, Li Yi, Hao Su and Leonidas J. Guibas. 'PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space'. In: *CoRR* abs/1706.02413 (2017). arXiv: 1706.02413. URL: http://arxiv.org/abs/1706.02413.

[9] Jeffrey L Elman. 'Finding structure in time'. In: *Cognitive science* 14.2 (1990), pp. 179–211.

[10] Michael I Jordan. 'Attractor dynamics and parallelism in a connectionist sequential machine'. In: *Artificial neural networks: concept learning*. 1990, pp. 112–127.

[11] Sepp Hochreiter and Jürgen Schmidhuber. 'Long Short-Term Memory'. In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[12] Ronald J Williams and David Zipser. 'Gradient-based learning algorithms for recurrent'. In: *Backpropagation: Theory, architectures, and applications* 433 (1995), p. 17.

[13] Wikimedia Commons. *The Long Short-Term Memory (LSTM) cell can process data sequentially and keep its hidden state through time.* 2021. URL: https://commons.wikimedia.org/wiki/File:LSTM_Cell.svg.

[14] Stuart Jonathan Russell and Peter Norvig. *Artificial intelligence: a modern approach.* Prentice Hall series in artificial intelligence. Pearson Education, 2010. ISBN: 9780132071482.

[15] Huajun Liu, Hui Zhang and Christoph Mertz. *DeepDA: LSTM-based Deep Data Association Network for Multi-Targets Tracking in Clutter.* 2019. arXiv: 1907.09915 [cs.LG].

[16] Harold W Kuhn. 'The Hungarian method for the assignment problem'. In: *Naval research logistics quarterly* 2.1-2 (1955), pp. 83–97.

[17] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez and Vladlen Koltun. *CARLA: An Open Urban Driving Simulator.* 2017. arXiv: 1711.03938 [cs.LG].

[18] Russell D. Reed and Robert J. Marks. *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks.* MIT Press, 1999.

[19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot and E. Duchesnay. 'Scikit-learn: Machine Learning in Python'. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[20] Anton Milan, Seyed Hamid Rezatofighi, Anthony R. Dick, Konrad Schindler and Ian D. Reid. 'Online Multi-target Tracking using Recurrent Neural Networks'. In: *CoRR* abs/1604.03635 (2016). arXiv: 1604.03635. URL: http://arxiv.org/abs/1604.03635.

[21] Margherita Grandini, Enrico Bagli and Giorgio Visani. *Metrics for Multi-Class Classification: an Overview.* 2020. arXiv: 2008.05756 [stat.ML].

[22] Xinlong Wang, Shu Liu, Xiaoyong Shen, Chunhua Shen and Jiaya Jia. *Associatively Segmenting Instances and Semantics in Point Clouds.* 2019. arXiv: 1902.09852 [cs.CV].

[23] Chunxiao Wang, Min Ji, Jian Wang, Wei Wen, Ting Li and Yong Sun. 'An improved DBSCAN method for LiDAR data segmentation with automatic Eps estimation'. In: *Sensors* 19 (Jan. 2019), p. 172. DOI: 10.3390/s19010172.

Appendix A

Model Details

A.1 Classification Models

All four models consist of three PointNet++ layers, which is what was used in the original paper by Qi et al. [8]. In addition to this there are a few dense layers, and dropout layers to reduce overfitting. *Sample size* refers to the number of sampled local ball shaped regions, with radius *sampling radius*. After the sampling the regions are processed by a small PointNet with fully connected layers of size *mlp*.

A.1.1 The Small Single Scale Model

Nr.	Layer	Parameters	Nr. of Parameters
1	PointNet++, single scale	sample size: 64 sampling radius: 0.4 mlp: (8, 8, 16)	344
2	PointNet++, single scale	sample size: 16 sampling radius: 0.6 mlp: (16, 16, 32)	1 328
3	PointNet++, global set to vector	sample size: - sample radius: - mlp: (32, 64, 128)	12 256
4	dense	output units: 64	8 256
5	dropout	probability: 0.5	-
6	dense	output units: 16	1 040
7	dropout	probability: 0.5	-
8	dense	output units: 4	68
			Total: 23 292

A.1.2 The Big Single Scale Model

Nr.	Layer	Parameters	Nr. of Parameters
1	PointNet++, single scale	sample size: 128 sampling radius: 0.3 mlp: (16, 16, 32)	1 072
2	PointNet++, single scale	sample size: 32 sampling radius: 0.5 mlp: (32, 32, 64)	4 704
3	PointNet++, global set to vector	sample size: - sample radius: - mlp: (64, 128, 256)	47 040
4	dense	output units: 128	32 896
5	dropout	probability: 0.5	-
6	dense	output units: 32	4 128
7	dropout	probability: 0.5	-
8	dense	output units: 4	132
			Total: 89 972

A.1.3 The Small Multiple Scales Model

Nr.	Layer	Parameters	Nr. of Parameters
1	PointNet++, multiple scales	sample size: 128 sampling radius: 0.3, 0.4, 0.6 mlp: (4, 4, 8), (8, 8, 16), (8, 12, 16)	924
2	PointNet++, multiple scales	sample size: 64 sampling radius: 0.4, 0.6, 1.0 mlp: (8, 8, 16), (16, 16, 32), (16, 16, 32)	4 088
3	PointNet++, global set to vector	sample size: - sample radius: - mlp: (32, 64, 128)	13 792
4	dense	output units: 64	8 256
5	dropout	probability: 0.5	-
6	dense	output units: 16	1 040
7	dropout	probability: 0.5	-
8	dense	output units: 4	68
			Total: 28 168

A.1.4 The Big Multiple Scales Model

Layer Nr.	Layer	Parameters	Nr. of Parameters
1	PointNet++, multiple scales	sample size: 256 sampling radius: 0.2, 0.3, 0.5 mlp: (8, 8, 16), (16, 16, 32), (16, 24, 32)	2 904
2	PointNet++, multiple scales	sample size: 128 sampling radius: 0.3, 0.5, 0.9 mlp: (16, 16, 32), (32, 32, 64), (32, 32, 64)	14 832
3	PointNet++, global set to vector	sample size: - sample radius: - mlp: (64, 128, 256)	53 184
4	dense	output units: 128	32 896
5	dropout	probability: 0.5	-
6	dense	output units: 32	4 128
7	dropout	probability: 0.5	-
8	dense	output units: 4	132
			Total: 108 076

A.2 PointNet++ Semantic Segmentation

Like earlier *Sample size* refers to the number of sampled local ball shaped regions, with radius *sampling radius*. After the sampling the regions are processed by a small PointNet with fully connected layers of size *mlp*. After that follows feature propagation and interpolation layers, such that all points are given a score.

A.2.1 The Small Semantic Segmentation Model

Layer Nr.	Layer	Parameters	Nr. of Parameters
1	PointNet++, single scale	sample size: 256 sampling radius: 0.2 mlp: (8, 8, 16)	344
2	PointNet++, single scale	sample size: 64 sampling radius: 0.3 mlp: (16, 16, 32)	1 328
3	PointNet++, single scale	sample size: 16 sampling radius: 0.5 mlp: (32, 32, 64)	4 704
4	PointNet++, single scale	sample size: 4 sampling radius: 0.9 mlp: (64, 64, 128)	17 600
5	PointNet++, feature propagation and interpolation	mlp: (64, 64)	16 896
6	PointNet++, feature propagation and interpolation	mlp: (64, 64)	10 752
7	PointNet++, feature propagation and interpolation	mlp: (64, 32)	7 552
8	PointNet++, feature propagation and interpolation	mlp: (32, 32, 32)	3 456
9	dense	output units: 32	1 056
10	dropout	probability: 0.5	-
11	dense	output units: 5	165
			Total: 63 855

A.2.2 The Big Semantic Segmentation Model

Layer Nr.	Layer	Parameters	Nr. of Parameters
1	PointNet++, single scale	sample size: 512 sampling radius: 0.15 mlp: (16, 16, 32)	1 072
2	PointNet++, single scale	sample size: 128 sampling radius: 0.25 mlp: (32, 32, 64)	4 704
3	PointNet++, single scale	sample size: 32 sampling radius: 0.45 mlp: (64, 64, 128)	17 600
4	PointNet++, single scale	sample size: 8 sampling radius: 0.85 mlp: (128, 128, 256)	67 968
5	PointNet++, feature propagation and interpolation	mlp: (128, 128)	66 560
6	PointNet++, feature propagation and interpolation	mlp: (128, 128)	41 984
7	PointNet++, feature propagation and interpolation	mlp: (128, 64)	29 440
8	PointNet++, feature propagation and interpolation	mlp: (64, 64, 64)	13 056
9	dense	output units: 64	4 160
10	dropout	probability: 0.5	-
11	dense	output units: 5	325
			Total: 246 871

A.3 Deep Tracker

	Layer	Output shape	Nr. of Parameters	Connected to
Inputs	input_X	[(1, 1, 20)]	0	
	input_z	[(1, 15, 2)]	0	
	input_eps	[(1, 10)]	0	
Prediction	rnn_prediction	(1, 300)	96300	input_X
	pred_xs	(1, 20)	6020	rnn_prediction
	pred_xs_reshape	(1, 10, 2)	0	pred_xs
Data Association	dist_matrix	(1, 10, 15)	0	input_z pred_xs_reshape
	lstm_inner	(1, 10, 500)	1032000	dist_matrix
	lstm_out	(1, 10, 16)	33088	lstm_inner
	A	(1, 10, 16)	0	lstm_out
Update	update_concat	(1, 10, 16, 2)	0	input_z pred_xs_reshape
	update_dot	(1, 2, 10)	0	A update_concat
	update_mult	(1, 2, 10)	0	update_dot input_eps
	update_flatten	(1, 20)	0	update_mult
	update_w	(1, 300)	6300	flatten
	update_add	(1, 300)	0	update_w rnn_prediction
	update_tanh	(1, 300)	0	update_add
	update_x	(1, 20)	6020	update_tanh
	update_eps	(1, 10)	3010	update_tanh

Total: 1,182,738

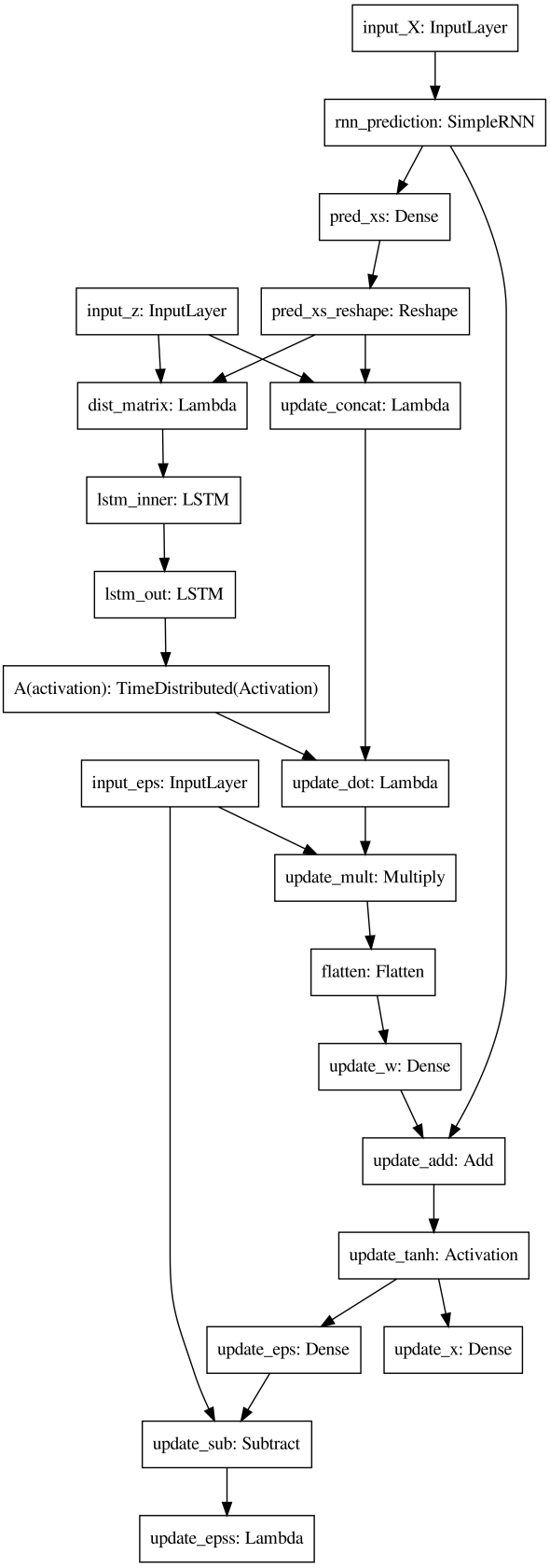


Figure A.1: Graphical visualization of the Deep Tracker.

Master's Theses in Mathematical Sciences 2022:E1
ISSN 1404-6342

LUTFMA-3461-2022

Mathematics
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lth.se/