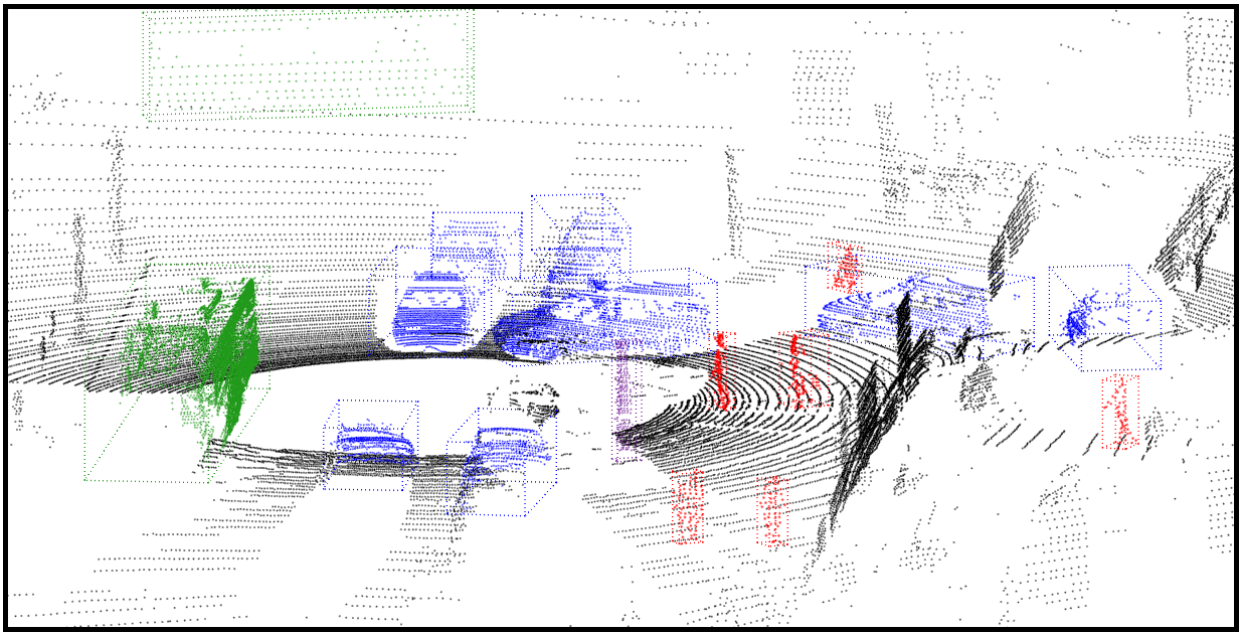
CHALMERS
UNIVERSITY OF TECHNOLOGY



Object Classification using 3D Convolutional Neural Networks

Master's thesis in Systems, Control and Mechatronics

AXEL BENDER
ELÍAS MAREL ÞORSTEINSSON

Department of Energy and Environment
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2016

Master's thesis 2016:11

Object Classification using 3D Convolutional Neural Networks

Classification of Voxelized LiDAR Point Cloud Data using 3D Convolutional Neural Networks

Axel Bender
Elías Marel Þorsteinsson

CHALMERS
UNIVERSITY OF TECHNOLOGY

Energy and Environment
Division of Physical Resource Theory
Complex Systems
Chalmers University of Technology
Gothenburg, Sweden 2016

Object Classification using 3D Convolutional Neural Networks
Classification of Voxelized LiDAR Point Cloud Data using
3D Convolutional Neural Networks
Axel Bender & Elías Marel Þorsteinsson

Supervisor: Robert Björkman, Semcon AB
Examiner: Peter Nordin, Energy and Environment

Master's Thesis 2016:NN
Department of Energy and Environment
Division of Physical Resource Theory
Complex Systems
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A 3D point cloud scan created by a Velodyne LiDAR sensor showing a typical traffic scenario which is the subject for object classification.

Typeset in LaTeX
Printed by TeknologTryck
Gothenburg, Sweden 2016

Object Classification using 3D Convolutional Neural Networks
Classification of Voxelized LiDAR Point Cloud Data using
3D Convolutional Neural Networks
AXEL BENDER & ELÍAS MAREL ÞORSTEINSSON
Department of Energy and Environment
Chalmers University of Technology

Abstract

In the development of autonomous driving and active safety systems, knowledge about the vehicle's surroundings is critical. When it comes to making decisions in real driving scenarios, the location and relative movement of surrounding vehicles, pedestrians and even static objects gives invaluable information to the system responsible for decision making. To know whether an object is a car or pedestrian, the system has to distinguish between the different features to predict object type. LiDAR sensors are among the most commonly used sensors in the development of modern AD systems as they produce dense images of their surroundings that are relatively resistant to changing light and weather conditions. Many classification methods use feature extraction or transformations to evaluate the 3D information using methods commonly used in 2D image analysis. In this thesis we evaluate the performance of training convolutional neural networks directly on 3D data, bypassing any information loss through data extraction or transformation and allowing the intensity hit of points to be used. The effectiveness of the method is evaluated on a dataset created from the KITTI Vision Benchmarking Suite. Our results show a total accuracy score of 96.35% and with a mean accuracy of 95.67% on a dataset trained on 7 classes.

Keywords: Autonomous Driving, LiDAR Sensor, Machine Learning, Artificial Neural Networks, Convolutional Neural Networks, Object Classification

Acknowledgements

We would like to thank our examiner, Peter Nordin for his invaluable input and guidance throughout the thesis work as well as our supervisor at Semcon, Robert Björkman who was always ready to aid us along the way and help us integrate our work with the AD-Tool team. We would also like to thank Jens Henriksson for his advice and numerous suggestions along the way and the Semcon Brazil team who generously supplied code to allow us to incorporate more data in our work. Finally, we would like to thank the authors of VoxNet, Daniel Matuarna and Sebastian Scherer as well as the community behind the Theano framework, whose work made this project possible to begin with.

Axel Bender and Elías Marel Þorsteinsson, Gothenburg, September 2016

Contents

List of Figures

List of Tables

Glossary

chunk Chunks of 16 batches that each contain 32 objects are used to utilize memory better during training and cutting down training times.. 35

cuDNN Deep Neural Network library (cuDNN) is a GPU-accelerated library of primitives for deep neural networks.. 35

epoch The process going once through the entire training dataset during training.. 23, 35

iteration A batch of items is processed at once in the training algorithm to better utilize memory.. 35

MNIST Handwritten digit database, commonly used for training and benchmarking various image processing systems and classifiers.. 12

overfitting When training a classification model, there is a risk of the model adapting overly to noise in the training set rather then learning general correlations. This has a high risk of happening if training is maintained for too long.. 9, 13, 35

scatter ratio Inverse ratio between the number of voxels an object has and the distance to the object's center of mass.. 21

supervised learning A method of ANN training, where a known set of input and their corresponding outputs are fed through the network and the error gradient, calculated through a loss function, is used to adjust the weight matrix through backpropagation.. 8, 9

test set The unseen data that is used to evaluate the performance of the model after training.. 25, 26, 27, 35, 36

training set The part of the dataset used for learning while training the model.. 25, 26, 29, 35

validation set When an ANN has finished training on the training data set, the accuracy of the ANN is tested using a data set it has not seen before, giving a more realistic perception of the predictability of the model.. 9, 35

visual field A term used for the data that is observable by the input layer of an Artificial Neural Network. 12

weight A computational variable adjusted during training of ANNs. Determines the neural network's response to a certain input. Also known as synaptic weights or connection weights.. 7, 8, 9, 14

List of Acronyms

ANN Artificial Neural Network. xv, 1, 2, 3, 4, 7, 8, 9, 10, 11, 13, 19, 23, 25, 26

CNN Convolutional Neural Network. xi, 2, 4, 11, 13, 19, 20, 21, 22, 25, 30, 34, 36, 45, 46, III
cuDNN CUDA Deep Neural Network library. 20

FFNN Feed Forward Neural Network. 27, 43

KNN K-Nearest Neighbours. 27, 43

LiDAR Light Detection and Ranging. 1, 3, 4, 15, 16, 17, 24, 26, 27, 30, 45
LSTM Long Short-Term Memory. 4, 15

MLP Multilayer Perceptron. 3

PCD Point Cloud Data. 11, 15, 16, 22, 23, 24, 25, 26, 27, 28, 29
PCL Point Cloud Library. 26, 28

ReLU Rectified Linear Unit. 9, 14
RF Receptive field. 11, 12, 22
RFC Random Forest Classification. 27, 43
RNN Recurrent Neural Network. xi, 4, 13, 14, 15

STC Stanford Track Collection. 25, 26, 31
SUOD Sydney Urban Objects Dataset. xi, xiii, 23, 30, 31, 32, 31, 32, 39, 40
SVM Support Vector Machine. 27, 43

Tanh Hyberbolic tangent. 8, 9, 14

VCC Volvo Car Corporation. xiii, 27, 29, 37, 39, 43

1
Introduction

Autonomous driving has been a major focus point of engineering- and software development research in the recent years. This has been facilitated mostly by improvements and lowered cost of computer hardware capable of running the complex algorithms needed. Large amounts of sensor data is being utilized to localize and ultimately guide the vehicle through its environment and this requires fast processing and considerable computational power.

Semcon is currently developing a framework for autonomous driving as part of a large cooperation project with other companies in the Gothenburg area. The project is referred to as AD-Tool and will be a modular framework where parts of the project can be easily added in as they are developed.

This thesis aims to supplement and improve a part of the project where objects found in a traffic situation are detected and classified from 3D LiDAR data. The objects can then be tracked and eventually used in localizing and steering the vehicle through it environment. This will be done by implementing and training a convolutional neural network to classify common types of urban objects such as cars, trucks, cyclists, pedestrians, etc.

1.1 Background

Artificial Neural Networks (ANNs) are a type of computer model that simulate the behaviour of biological neural networks such as the human brain. The models consist of artificial neurons that can be used to estimate nonlinear functions and behaviours. The method dates back to the 1940's but it gained a lot of attention from computer science researchers in the 1980's with the birth of the backpropagation algorithm in 1975 [7]. However, the development of neural networks was limited at first by the computational power and cost efficiency of computer hardware at the time.

A single neuron on its own has little useful computational power, however the human brain as a whole has only recently been outmatched in pure computational speed by the world's most powerful supercomputers [8]. With that in mind, writing algorithms and building software frameworks based on the structure and behaviour of the human brain, might be the next step towards making autonomous robots capable of making human-like decisions.

A very simplified example of how the human brain identifies an object, for an instance a person's face, is to say that the information gets broken down into simpler

components that then get pieced back together to form the bigger picture. The way that the information from the optical nerve gets processed can be envisioned as it moving along a net of neurons, connected together by neural pathways. Each time that the information gets to a crossroad a decision is made. Based on a fraction of the information, say the curve of a nose or the shape of an eye, the path taken through the neural net is chosen. In the end, depending on which neurons were stimulated, the human brain identifies the face to be one that it has seen before, having triggered close to the same neural response upon learning these facial features.

This example illustrates the need for using ANNs with many hidden layers to even come close to representing similar architecture in a computer, as found in the human brain. There is also the fact that human brain is remarkably good at associative learning and using context to draw conclusions quickly.

To utilize the internal structure and context of the point cloud dataset, the idea is to use Convolutional Neural Networks (CNNs) with deep learning that allow for simulation of the most basic cognitive capability of a human brain. Using deep learning can however be very computationally expensive and can make the algorithms and methods used in training ANNs for object classification very complicated.

In 1989 Yann LeCun attempted to use backpropagation with a deep neural network, a network with multiple hidden layers of artificial neurons that better represents the structure of the human brain. The time to train the dataset took approximately 3 days, which is perhaps too long for most practical applications. [9]. With more computational power and efficient algorithms, capabilities of ANNs have risen. By today's standards the concept of deep learning can be considered to be a viable option for training ANNs.

ANNs have already been proven to excel at identifying objects from images, doing voice recognition and even creating art [4]. The purpose of this project is to look at how effective ANNs are at identifying objects in a grid of 3D points from a Velodyne LiDAR sensor. This approach of training ANNs using deep learning to classify objects, compared to the more traditional method of matching objects to a library of templates, will be the main focus of this Master's thesis.

1.2 Motivation

The proficiency of neural network applications when it comes to classification of 2D image data is well documented, even when using close to 1000 possible image categories, as was the case in the implementation of *AlexNet* [10]. Inspired by the positive results of 2D ANN implementations, research groups and companies have looked towards applying the more successful variants of neural networks to 3D data such as RGB-D images, CAD models and 3D point cloud scans.

The main obstacle in handling 3D data with ANNs that researchers are still working to overcome is that almost all neural network development frameworks and training software is specifically designed to take inputs in the form of 2D images. This has led to different approaches such as projecting 3D data to 2D, a method that

is sometimes called 2.5D projection [11]. Other older methods use so-called point histograms as a way of quantifying the dimensions of the 3D point clusters in a way that 2D neural networks can handle [12]. Voxelizing the 3D data into occupancy grids stored as binary matrices has been shown to have relatively high success in comparison to the aforementioned input methods [6].

What drives this development is to see if the success that has been shown in classification of 2D images can be applied in the same degree to 3D data. With regards to autonomous driving, the motivation behind using LiDAR sensor data in this way is that the main alternative sensor type that still provides depth perception to the vehicle's AI algorithms are stereoscopic cameras which can be very unreliable as a single data source in different lighting and especially in certain weather conditions. LiDAR data is also completely unaffected by shadows, which can be hard to adjust for with regular camera images and video feed. Another of the LiDAR's strength over cameras is its 360° field of view and the fact that it generates images that are not directional or skewed by the camera's narrow field of view. Having a high definition camera as an additional source of sensor data for redundancy is generally a good idea.

1.3 Related Work

Even though the usage of Artificial Neural Networks on 3D point cloud data is still a fresh research topic, a number of articles and thesis work have been published over the last few years on the topic. A paper published in 2013, at The Brazilian Conference on Intelligent Systems, approaches the subject of a Multilayer Perceptron (MLP) neural network implemented on 3D point cloud data where 2.5D projection is applied to the data [11]. This article gave valuable insight in using deep learning and ANNs to recognise objects in a point cloud data. The article also mentioned a few interesting ways to segment the point cloud data into candidate objects for classification and removing unnecessary data points, such as the ground, from the data set.

Another interesting article published in 2015 explored an efficient way of representing 3D data, using a method of storing 3D points called point feature histograms [13]. This method is usually used to stitch together two overlapping point clouds by finding common point features and positioning and orientating the point clouds so that a estimated error between them is minimized. To do this the data is stored in a very efficient way using 2D like matrices. Another article by the same team implements this method in a simple but effective object detection algorithm trained on RGB-D data [14].

By following through on a few of the cited sources in the previously mentioned article [11] lead to some very interesting papers concerning data segmentation and its importance. Various different methods such as the socalled *fast segmentation method* [12] and the *clustering method* for efficient segmentation of 3D point data [15] are described. In the current version of the AD-tool a method for segmentation

is being developed alongside this project. For the purposes of this thesis it is assumed that this method will be used to deliver segmented clusters of data points to the neural network for classification and a training dataset will be chosen with this in mind.

A promising solution to the problem of input manipulation of the 3D point cloud data was found in a paper released in 2014. In the paper the volumetric space of a 3D point cloud was voxelized using a relatively simple algorithm to make it input compatible with the neural network framework used [6]. Other design features were introduced in this paper that served as groundwork and inspiration for the data flow and framework setup in this project. This includes an novel approach to data augmentation during training and testing. The training data is rotated a set number of times around the LiDAR's rotational axis to simulate multiple viewpoints during training in a process called rotation augmentation.

Closely related to the problem of the object classification is the complex task of tracking each detected object in a flow of 3D point cloud frames. This is done by assigning unique tags for each object in the frame and maintaining the same tag throughout the "lifetime" of the tracked object. This kind of tracking will eventually be implemented alongside online object classification in the AD-Tool software.
The problem of multiple object tracking is a open research problem in the field of robotics and is the topic of many new research papers. Recently, a paper on the topic was published where Recurrent Neural Network (RNN) were used to accomplish for the first time, an end-to-end artificial learning approach to multi-object tracking [16]. The recurrent aspect of RNNs, to deal with sequential information, along with Long Short-Term Memory (LSTM) seems well suited for this task and given that object classification is so closely related to the goal of object tracking, it can be beneficial to utilize some of the methods used in this paper to possibly aid in upcoming parts of the AD-Tool tracking implementation.

1.4 Objective

This thesis is focused on designing and training CNNs to accept inputs in the form of 3D point cloud data. This involves designing the architecture of the neural network to best suite the task and restructuring the point cloud input data to be compatible with the input structure of the neural network. After formulating a neural network structure suited for the purpose, the task is to select a method of training the ANN, using a known set of inputs and corresponding outputs, to classify objects. This involves selecting an appropriate dataset that fits the purpose of the network.

An account of a few relevant research articles can be seen in Section 1.3. When the stand-alone neural network has been implemented and trained to satisfaction, the code must be made available in a form that can be combined with the AD-Tool software package.

1.5 Purpose and Aim

The goal of the thesis is to evaluate the performance of using neural networks to classify objects from 3D data, both in an effort to utilize the intensity hits of the 3D points as well as maintaining the 3D structure of the object without abstracting any data via 2D conversion. The resulting classifier could also be a very useful tool in detecting segmentation errors and faulty tracklets in point cloud data that is used for tracking purposes.

2
Theory

Before diving into the more hands-on parts of the implementation and framework architecture, it is essential to start from the beginning and have a look at the base elements of neural networks and the algorithms and tools that make up the foundation of their development. Estimating the limits of the hardware is also valuable, so this section will also cover the engineering behind the sensors used to collect the necessary data for this project.

2.1 Artificial Neural Networks

Artificial Neural Networks are connected systems of learning algorithms that through fast, weighted computations estimate an output, given a set of inputs. The framework of ANNs is inspired by biological neural networks. It resembles its biological counterpart in the way it breaks down the inputs and systematically distributes them amongst its computational units. It is then the collaboration between them or parallel computation that dictates the estimated output.

General ANNs consist of a weight matrix, which shape corresponds to the number of connections between the artificial neurons of the network and the dimensionality of the input data. The ANNs are then divided into different layers. The number of layers and their dimensions depend greatly on the network's intended purpose. The layers are often referred to as input layers, output layers and hidden layers. Figure 2.1 shows a visual representation of a simple ANN example. This section covers the building blocks of neural networks and the common functions used to train and optimize their behaviours as well as looking at more specific forms of neural networks that have had particular success in their intended applications.

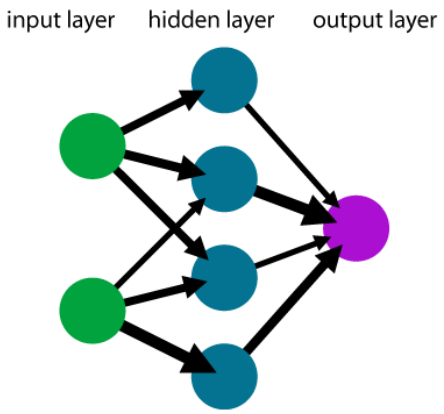


Figure 2.1: A visual representation of a simple artificial neural network [1].

2.1.1 Layers

Modern ANNs used for object identification, can be broken down into three layers. The input layer is responsible for encoding the incoming data into smaller units,

based on the resolution that the neurons are operating under.

In thhe hidden layers is where the actual processing of information happens. Typically the input layer is fully connected to the hidden layer, however the neurons occupying the hidden layer are not interconnected. The data segments get sent from the input neuron to its corresponding neurons on the hidden layer, where each neuron performs a feedforward calculation. This process is described in Equation 2.1.

$$y_j = \sigma(b + \sum_{i=1}^{N} w_{ij} y_i) \tag{2.1}$$

Where y_j is output from neuron j on to the corresponding neurons of the next layer, σ is the activation function chosen for the ANN, b is a bias term for the current layer, w_{ij} is the weight matrix between current neuron j and neurons i from the previous layer and y_i is the outputs from the previous layer.

The final layer of any ANN is a fully-connected output layer. After the hidden layer the data goes through logical regression, a supervised learning algorithm, in order to estimate which of the output categories the input belongs to. It is at this point a method called backpropagation is used to calculate the accuracy of this estimate and sends back information to the hidden layer, to adjust the current weights between the different layers. The number of neurons on the output layer usually corresponds to the number of categories up for classification. [17].

2.1.2 Activation functions

Activation functions are nonlinear functions that compute the response of a hidden-layer neuron to a set of inputs. The nonlinearity allows the network to compute nontrivial problems, using a small number of nodes.

The sigmoid function is a nonlinear function that is defined mathematically as seen in Equation 2.2.

$$\sigma(x) = 1/(1 + e^x) \tag{2.2}$$

It was originally favoured as an activation function for its resemblance to the firing rate of a biological neuron. The sigmoid function normalizes the input to a neuron between 0 and 1, further simulating the firing of a biological neuron from not firing to fully-saturated firing at maximum frequency. The drawbacks of using a sigmoid function and the reason it has fallen out of favour in modern ANNs is that they saturate and effectively kill gradients. The problem surfaces when it comes to training the ANN using backpropagation. The local gradient of the saturated neurons is multiplied to the gradient of the loss function in order to adjust their weights and minimize the loss function gradient. The gradient of the saturated neuron being so close to zero causes them to effectively stop learning and the neuron becomes stagnant [18].

Hyberbolic tangent (Tanh) is another commonly used activation function which is essentially a scaled sigmoid function, as can be seen in Equation 2.3.

$$tanh(x) = 2\sigma(2x) - 1 \tag{2.3}$$

This means that it saturates like the sigmoid function, however it is zero-centered and normalizes its input between -1 and 1. Which means that neurons in later hidden layers receives data that is zero-centered. This reduces the risk of the gradient of the weight matrix w becoming either all positive or all negative during backpropagation, avoiding the following zig-zag behaviour of the loss function. This makes for a better training loss curve.

Rectified Linear Unit (ReLU) is another common activation function. It can be described mathematically as seen in Equation 2.4.

$$f(x) = max(0, x) \tag{2.4}$$

where the activation is a threshold at zero. Due to its linear, non-saturating form it has been shown to greatly accelerate the convergence of stochastic gradient descent, compared to sigmoid or Tanh functions [19]. The downside of ReLUs is that a large gradient flowing through them can cause the weights to be updated in in such a way, that the neuron will never fire from any data point from then on. This can be avoided by carefully selecting an appropriate learning rate. If the learning rate is too high, it will increase the frequency of this problem during training. Learning rates are often adjusted in between training cycles in a process called *learning rate scheduling*.

2.1.3 Dropout

Training ANNs is a time consuming process that can't be accurately validated while running. The reason for this inability to correctly estimate the accuracy of the ANN is a concept known as *overfitting*. Typically when fitting a model to a set of training data, using too many input parameters, the simulation becomes highly reliant on the data it was modelled to. This leaves the model unreliable in predicting an outcome, when a validation set is introduced. To minimize overfitting some form of noise is usually introduced to the training, alongside the training data. In the case of ANNs this deviation, from the symmetry of the training data, is achieved using a method called dropout.

Dropout is usually applied between the layers that perform feedforward calculations on the input data. It is a stochastic method of removing a fixed percentage of the data before passing it along to the next layer. This lessens the likelyhood that the ANN does not start to anticipate a certain symmetry in its input.

2.1.4 Training algorithms

Training any ANN is by far the most time consuming part out of the whole process. After completing feedforward calculations the ANN makes an estimate of what category the input belongs to. Using backpropagation, which is a supervised learning method, the network can compare the estimate to the known output to adjust the weights and bias for every layer in the ANN. The focus of backpropagation is usually a quadratic cost function defined as in Equation 2.5.

$$C = \frac{1}{2n} \sum_{x} ||y(x) - \alpha^L(x)||^2 \tag{2.5}$$

Here n is the total number of training examples, $y(x)$ is the expected output of the ANN and $\alpha(x)^L$ a vector of the activation functions output, with L denoting the number of layers in the network, summed over training examples x.

The objective of *backpropagation* is first to compute partial derivatives of the cost function, $\partial C/\partial w_{ij}^l$ and $\partial C/\partial b_i^l$, with respect to the weight matrix and bias for every layer. Backpropagation will provide the procedure to compute the error in each neuron i on layer l in the ANN, namely δ_i^l, and correlate it to the partial derivatives of the cost function. Where δ_i^l is defined as the partial derivative of the cost function w.r.t. the weighted input z_i^l to activation function σ for neuron i on layer l, or by Equation 2.6:

$$\delta_i^l = \frac{\partial C}{\partial z_i^l} \tag{2.6}$$

In essence, the method of backpropagation uses four equations to calculate its outcome. First is the equation for calculating the error in the output layer.

$$\delta_i^L = \frac{\partial C}{\partial \alpha_i^L}\sigma'(z_i^L) \;\; matrix\ form: \;\; \delta^L = \Delta_\alpha C \odot \sigma'(z^L) \tag{2.7}$$

The first term of Equation 2.7, $\Delta_\alpha C$, is the rate at which the cost function changes w.r.t. the activation output vector and $\sigma'(z^L)$ measures how fast the activation function *sigma* is changing at z_i^L.

The second Equation 2.8 calculates the error δ^l in terms of the error calculated for the next layer, δ^{l+1}.

$$\delta^l = ((w^{l+1})^T\delta^{l+1}) \odot \sigma'(z^l) \tag{2.8}$$

This effectively propagates the error calculated at the output layer backwards to the input layer of the ANN. Equation 2.9 relates the change in cost w.r.t. the bias for each layer in the network to the error in neurons on the same layer.

$$\frac{\partial C}{\partial b_i^l} = \delta_i^l \tag{2.9}$$

Having already calculated the error for neurons in each layer, using Equations 2.7 and 2.8, and it being exactly equal to the rate of change in cost w.r.t. the bias, means that the backpropagation is halfway towards linking the error δ to $\partial C/\partial w_{ij}^l$ and $\partial C/\partial b_i^l$.

Lastly, Equation 2.10 describes what is called *momentum* links the rate of change of the cost w.r.t the weight in the network to the difference of the activation output from previous layer to the neuron error in current layer.

$$\frac{\partial C}{\partial w_{ij}^l} = \alpha_k^{l-1}\delta_i^l \tag{2.10}$$

With these four Equations 2.7, 2.8, 2.9 and 2.10 the backpropagation method can update the weight matrix and bias using gradient descent (Equation 2.11), based on the error calculated for neurons in that layer [17].

$$w^l \rightarrow w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (\alpha^{x,l-1})^T$$
$$b^l \rightarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l} \tag{2.11}$$

2.2 Convolutional Neural Networks

When modifying ANNs to work on data sets where the structure or locality of the data is an important aspect, as is the case with Point Cloud Data (PCD), the concept of Convolutional Neural Networks (CNNs) becomes appealing. The reason why CNNs are preferred in this case is that its architecture takes advantage of the spacial locality of its input data. This attribute of CNNs is mainly due to the generation of feature maps using convolution filters. Unlike the regular ANN, where each unit on the input layer is connected to every unit on the first hidden layer, each hidden unit in a CNN is connected only to a sub-region of the input image. Segmenting the input data into these so called Receptive fields (RFs) and not having the hidden layers fully connected, reduces the amount of feedforward calculations performed and prevents the number of hidden units from scaling up, when resolution of the input is increased. In addition to the layers described for the traditional ANN in Section 2.1, CNNs generally include both convolution and pooling layers in their hidden layers.

2.2.1 Convolution filters

The input to a convolutional layer of a CNN is, in the case of images, a matrix of $m \times m \times r$ dimensions. Where m is the height and width of the image in pixels and r is the depth of the input data, in coloured images $r = 3$ for RGB. With higher dimensionality r can be expanded to accommodate for the data. On the convolutional layer k linear filters with dimensions $n \times n \times q$, where $n < m$ and $q \leq r$, are convolved with the input data. The size of the convolutional filters governs the locally connected structure of the hidden layer and each filter is convolved with the input image, creating k feature maps of size $m-n+1$ [2].

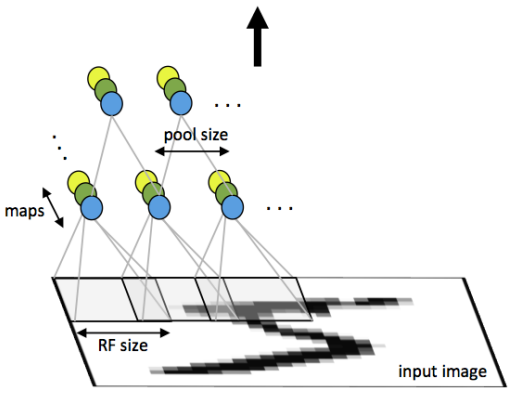


Figure 2.2: Convolution layer with downsampling. Units belonging to the same colour share the same parametrization (weight vector and bias), the different colours represent different filter maps [2].

2.2.2 Feature maps

From the input layer the RFs are convolved with linear filters and after adding a bias term and applying a non-linear function, the feature maps are generated. This subject is covered in detail in Section 2.1.2. Using the notation that the k-th feature map at a given layer is h^k, whose filter is determined by the weight matrix W^k and bias b_k, results in Equation 2.12 which describes the feature map pre-activation as a function of the input channel x [20].

$$h_{ij}^k = tanh((W^k * x)_{ij} + b_k) \tag{2.12}$$

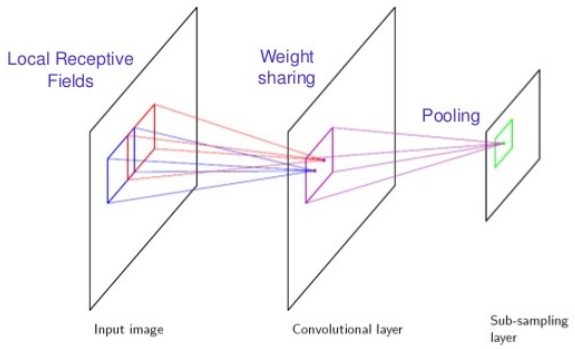


Figure 2.3: Convolutions between a linear filter and the receptive fields, resulting in a feature map, followed by pooling and sub-sampling [3].

Each filter is convolved with every RF on the visual field, producing the same number of feature maps as there are convolutional filters. Units belonging to the same feature map, share between them both weight and bias connecting the RFs to that feature map.

For every feature map there is therefore one weight matrix mapping each point of data on a RF to its corresponding unit on the feature map in question. This results in each feature map focusing on different aspects of the layer below and every unit within a feature map covering different locations on the input data, while keeping the number of stored parameters low.

The size of a RF is governed by the resolution of the input data, while remaining large enough for the hidden units to distinguish a pattern and establishing direction of the data segment, typically for small images (e.g. MNIST) a 3x3 matrix.

2.2.3 Nonlinear down-sampling

Due to the overlapping nature of RFs, the process of convolution leaves some units within a local neighborhood of the feature map to focus on the same parts of the input image. This translates to unnecessary activation calculations for upper layers. One solution to this problem is known as max-pooling.

Max-pooling is a form of nonlinear down-sampling that partitions the feature maps into non-overlapping sets and selects the largest activation within each one. At the pooling layer these local maxima represent each set in a smaller sub-sampled matrix, which then gets passed on as input to the next hidden layer. The pooling layer is defined by size and stride, where the size defines the dimensions of the pooling-filter and stride is the step size in which the filter passes over the feature map. Example of max-pooling is shown in Figure 2.4.

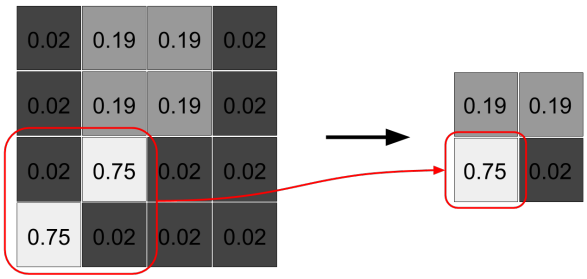


Figure 2.4: Max-pooling of the feature map's activation outputs. Filter size 3x3 and stride is 2.

This method effectively reduces the number of units on the hidden layers and introduces local translations invariance [20]. It can also help to reduce overfitting and frees up memory. Benefits towards reduced overfitting only works on small data sets and since max-pooling reduces the resolution of the convolutional layer, should be used with care.

2.2.4 Loss layer

The last layer of a CNN is where the loss function (i.e. cost function, Equation 2.5) computes the deviation between the predicted- and true labels. The loss layer has the fully-connected output layer and the ground truth distribution as inputs. The loss functions can vary depending on the intended purpose for the ANN. An example loss function, used here and in similar work, is cross entropy.

Cross-entropy (here categorical cross-entropy) computes the categorical cross-entropy between predictions and targets.

$$L_i = -\sum_j t_{i,j} \; log(p_{i,j}) \tag{2.13}$$

The function computes the error δ_i, w.r.t the parameters in the fully connected layer. The error is then propagated through the pooling- and convolutional layers, where the it is up-sampled, in order to adjust the weights between the input- and hidden layers.

2.3 Recurrent Neural Networks

Although the main interest of this project is the object classification of segmented 3D point cloud clusters, a closely related part of the AD-Tool project at Semcon is the tracking and tagging of these clusters. Multiple object tracking in real-world scenarios is a complex and challenging research topic and recently the usage of recurrent neural networks in real time object tracking has been shown to have competitive results to previous solutions that often use very complex state modelling and are rarely computed online [16].

The main way RNNs differ from the conventional feedforward neural networks is that they include at least one feedback loop between the input and output. By doing this, RNNs utilize sequential information in the input and by doing so the network gains a certain amount of memory that can help in learning tasks that have a routine element. The most common application for RNNs is language modelling and speech recognition, where the order of the inputs (words) and the connection between them gives valuable information about the meaning.

In addition to the weights calculated when training feedforward neural networks using backpropogation, the feedback loops in the hidden layer of a RNN have weights that represent a hidden state that makes up the memory of the network. The way the weights are calculated differs from conventional neural networks, the process is referred to as *unfolding* and involves calculating the weights of the network for a sequence of a known length where the number of layers in the network represents the length of the sequence being calculated [21]. Figure 2.5 shows an example of a simple RNN with one input and one output unfolded into a full network.

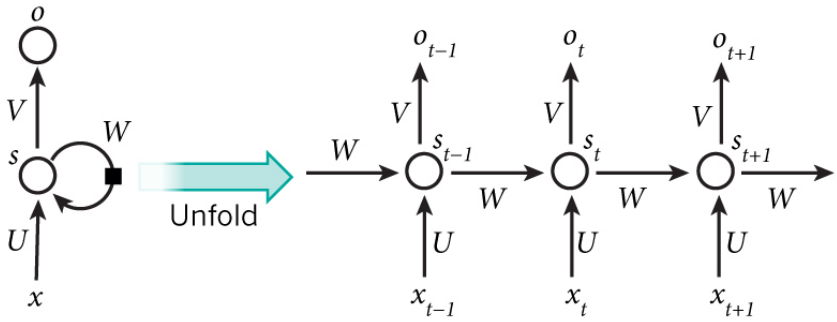


Figure 2.5: A typical RNN structure being unfolded into a three layer network [4].

When writing out a network in its unfolded form each node of the diagram represents a timestep marked by t. x_t is the input at time t and o_t is the output at time t. U, V and W are the weights of the network calculated during training and s_t is the hidden state of the network at timestep t. s_t is calculated by summing over the current input and the last hidden state after passing through an activation function as seen in Equation 2.14.

$$s_t = f(Ux_t + Ws_{t-1}) \tag{2.14}$$

where f is an activation function, commonly ReLU or Tanh. As can be seen, every time a new state is calculated the previous state, s_{t-1}, is required. Since there exists no previous state at the first timestep, s_{t-1} is usually initialized as all zeros. Finally, the output at each timestep o_t is calculated as seen in Equation 2.15.

$$o_t = softmax(Vs_t) \tag{2.15}$$

The weights U, V and W calculated during training are shared over each of the timesteps and RNNs therefore have relatively few weights compared to other types of neural networks.

2.3.1 LSTM

A Long Short-Term Memory (LSTM) network is a commonly used variant of RNNs that is currently used in many state-of-the-art applications and research projects due to their proficiency in recognizing long-term dependencies in the data they are trained on. Their structure is more complex then the base type of RNNs and the complexity is mostly in the way the hidden state is calculated. Instead of a single activation function that is chained through all time steps, LSTM networks have four calculation steps and the memory of the network, the hidden state is referred to as the network's *cell*. The four calculation steps are designed to allow the network to "select" which information to keep from the previous states [21].

The details of these calculations is beyond the scope of this project but the functionality aspect of LSTM is particularly interesting for the object tracking side of AD-Tool which is under development at Semcon alongside the classification which is the subject of this thesis. It is therefore of great interest to analyse the input- and structural possibilities of the networks that are likely to be used in tracking to best facilitate the sharing of any common elements and information between the classification algorithms and tracking methods.

2.4 Point Cloud Data

3D Point Cloud Data (PCD) is a digital representation of volumetric information, most commonly portraying outdoor scenes or landscapes. PCD is also used in conjunction with CAD modelling and 3D printing. Point clouds are usually created by scanning real world objects and landscapes for further analysis and replication in computer simulations. The usage of point clouds in robotics and landscape mapping has steadily increased as the scanners have become both more affordable and increasingly powerful.

The most common sensor type used to create PCD is called a Light Detection and Ranging (LiDAR) which works on the same principles as a radar. In the same way that radars transmit radio waves and receive them back as they reflect from objects in their path, LiDARs send out beams of light that reflect off any object in their path. The time it took for the laser beam to travel to and from the object, or *flight time* is then used to calculate its distance from the LiDAR. The collision location between the laser beam and the object is then represented as a single point in a 3D point cloud. Most LiDARs contain multiple light sources that make up an array of laser beams to cover an wide vertical angle. The sensor is then rotated to span the area around it.

The amount of data stored in the resulting PCD depends on the exact model and type of sensor used. The Cartesian coordinates of each point are always included but additional data is often available. This can be e.g. the intensity of the reflected laser beam, the ring number of the laser source corresponding to the hit, a timestamp

for the laser hit and the azimuth angle from the center of the LiDARs field of view. There are even methods of portraying the intensity of each reflected point as color and store the information as RGB values.

The method for storing this data varies from manufacturer and the intended purpose of the PCD. The most bare-bones file format is the .XYZ format which only contains the Cartesian coordinates of the measured points in a ASCII file format where each line represents a measured point and the coordinates themselves are separated by a comma or white space. Since more information is often required the most common file formats .LAS or .CSV, which are also ASCII files, but can include many fields that are then described by contained headers or external help documentation. Common practice is to convert these ASCII files to binary data (.BIN) to speed up their processing. This however makes the information hard to interpret for the user so it is usually done only when high processing speed is needed.

2.4.1 Velodyne HDL-64E

Velodyne LiDAR is a leading developer of light-weight high functionality LiDARs that can be mounted on mobile robots and vehicles. The flagship product of the company is the Velodyne HDL-64E, which uses 64 laser beams spread over a 26.8° vertical angle and achieves a 360° horizontal field of view by spinning at 300-900 RPM around its base [22]. The sensor can map up to 2.2 million points per second and the Cartesian accuracy is listed as <2 cm, at its operating range of upwards to 120 m [23]. The company also produces lighter and lower functionality LiDARs that are aimed at smaller robots and applications that do not require the range and point density of the Velodyne HDL-64E.

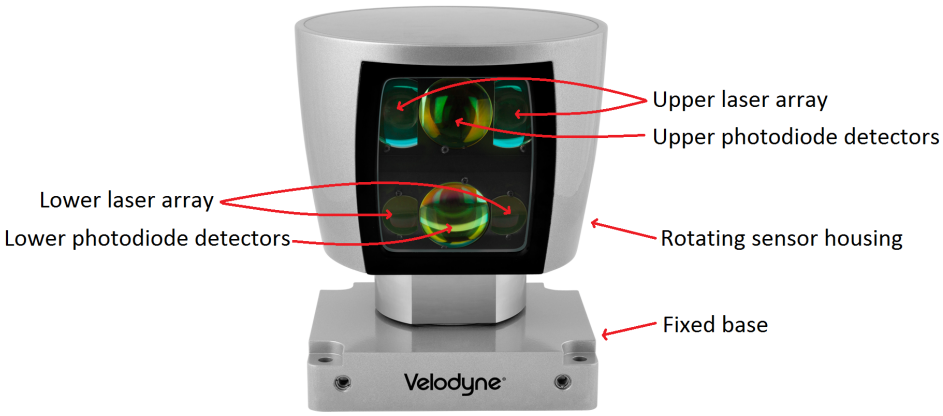


Figure 2.6: A Velodyne HDL-64E LiDAR sensor.

The sensor itself consists of a rotating head with a sensor array on one side mounted behind a glass panel. The laser arrays and photo sensors that are used to scan the surroundings of the sensor are split into an upper level and a lower level, each comprising of three lenses. For both levels, the lasers are fired out of the two

outer lenses, 16 lasers in the right lens and 16 in the left while all laser returns are picked up through the middle lens for each level. An annotated photograph of the HDL-64E can be seen in Figure 2.6.

Velodyne, an acoustics manufacturer moved into the development of laser range finding when two of the company's founders took part in the DARPA Grand Challenge, a driverless vehicle challenge where the vehicles were set to follow a route marked by GPS coordinates and had to navigate the terrain with onboard sensors. They saw the need for sophisticated laser range finding sensors and developed their first edition of a LiDAR sensor for the 2005 challenge. The sensor that the team used in the 2005 challenge was about 100 pounds and instead of competing in the next DARPA challenge, Velodyne decided to commercialize the sensor. In the 2007 DARPA Grand Challenge, five of the six top teams all used Velodyne's HDL-64E LiDAR for obstacle avoidance and guiding [24]. The HDL-64E has been used to create many of the 3D point cloud data sets available to the public, including all the datasets considered for this project (see Section 3.2) and the sensor is considered state-of-the-art in vehicle LiDAR technology.

3
Methods

With ANN implementations, the architecture of the network is paramount to the success of the application. This includes the choice of inputs and outputs, layer structures and filtering as well as the training methods and learning parameters. However, before addressing all of the low level parameter choices and tweaks, the software framework that is the foundation of the neural network itself must be carefully selected for the task at hand.

Programming a network structure as complex as CNN training is not elementary and even more complicated when it comes to making it work with GPU accelerated libraries that allow much faster training times than training on a regular CPU. Deep learning applications are therefore very seldom programmed from scratch unless for a very specific purpose.

A range of deep learning frameworks are available as open source licence with different levels of support and hardware compatibility, selecting one that is right for each application project is an important decision and might single-handedly make or break a project's success. After establishing the base structure of the network, the next important part is knowing how to use the tools of the framework to implement the characteristics and learning features that are right for the application.

Training is another key point to a network's application success and the training dataset used is vital to its ability to recognize the range of data it will be used on eventually. Choosing the right dataset for this specific task with the necessary range and sample size is therefore instrumental.

3.1 Architecture

In this section titled architecture the focus is on describing both the support framework chosen for training and computing the CNNs and the type of CNN architecture used in this project. The motivation for both the framework and architecture will be discussed and modifications and implementations introduced.

3.1.1 Framework

When working with CNNs there are a few frameworks worth mentioning in contrast to the framework chosen for this project. First framework considered for this project was **CUDA-convnet2**, prized for being a fast C++/CUDA implementation

of CNNs, by Alex Krizhevsky. Second honorable mention is TensorFlow, developed by Google researchers working on the Google Brain Team, which flexible architecture allows for deploying computations to one or more GPUs or CPUs. The main attribute of TensorFlow is the modular GUI that allows for easy assembly of the layers and components of a CNN.

Theano ended up being framework chosen for this project due to its flexibility and efficiency working with CNNs, allowing for up to 3x faster convolutions. Theano allows for specifying models symbolically and compiles the representation to well optimized code for both CPUs and GPUs. Furthermore it does symbolic differentiation, for gradient descent, which simplifies the process of training CNNs and updating the weight matrix, regardless of the dimensionality or input to the CNN.

There are a few Theano dependencies which are listed as follows:

- python-numpy
- python-scipy
- python-dev
- python-pip
- python-nose
- libopenblas-dev [25]

Theano is built to work with latest version of CUDA toolkit and samples [26], which for this project was CUDA 7.5 for GPU accelerated computing. When compiling CNNs for GPUs, Theano replaces its default convolution with a cuDNN-based implementation, if the cuDNN package is installed. When training large CNNs, with many convoluted hidden layers, its vital to utilize GPU accelerated computing. It can take up to 3x to 15x longer to train a CNN on a high end CPU than it would take using a high end GPU, depending heavily on the task [27]. Theano has good integration with GPU accelerated computing libraries and well documented examples of working with CUDA.

This project needs a modified Theano framework to work on 3D Convolutional Neural Network. The solution comes in the form of VoxNet, a modified implementation of Theano by Daniel Maturana [6]. In addition to Theano VoxNet relies on Lasagna [28], a lightweight library to construct and train CNNs in Theano. Lasagna provides memory and layer management, along with useful functions and algorithms to implement with CNNs.

3.1.2 CNN Layers

Deep neural networks are often classified by the number of layers, or their "depth". When it comes to CNNs this metric is quite hard to define due to the nature of the convolutional filters and the numerous calculations that are sometimes applied to one set of feature maps. The question becomes, what is defined as a single layer? Is it the number of convolutions or is the result of each convolution, drop-out or other operations considered a separate layer?

The standard for this was set by Karpathy et. al in in their ImageNet paper [19]. There, each convolution is considered it's own layer and fully connected layers add to the count. In our case there are therefore two convolutional layers and two fully connected layers before the output, making the depth in total 4 for the whole network. Figure 3.1 shows the breakdown of the CNN along with the product of each significant layer.

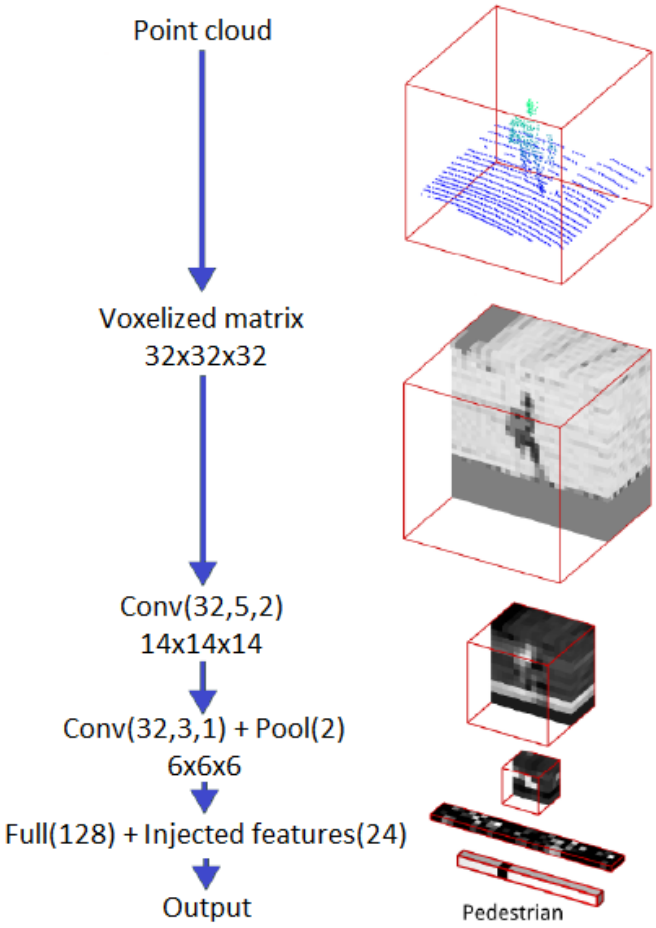


Figure 3.1: The structure of the convolutional neural network.

3.1.3 Concatenation Layer

Aside from the traditional layers of a CNN, input-, output-, convolutional-, dropout-, fully connected and pooling layers, described in section 2.2, a concatenation layer to merge the product of the second convolutional layer with an injected input was used. This addition to the CNN was implemented due to the dynamic voxelization discussed in section 3.3. By injecting the dimensions of an object subjected to the CNN, along with other derived features, after the second convolutional layer and before the output, it is possible to diminish the negative impact of dynamically scaling the input, while keeping the benefits of maintaining shape and reducing loss of information.

The injected parameters used were x, y, z dimensions $[1x3]$ for the current object and Z-scores for these dimensions corresponding to a comparison between the object

and the average size of each known training class [$3x7$]. This resulted in a total of 24 injected parameters merged with the second fully connected layer. Another set of parameters considered was to use scatter ratio Z-score [$1x7$] from a comparison between the current object and every training class and average number of high intensity points to the number of intensity points of the object [$1x7$].

3.1.4 Input data manipulation

One of the more difficult challenges faced in this thesis is to manipulate the input of PCD onto a form more suitable for the architecture of CNNs. For 2D CNNs a typical input would be a 2D image, sometimes a grayscale- or RGB image, will be an $n \times n$ matrix with input channel r corresponding to the number of layers in the image. Transferring 3D information into a format that a CNN can process can be seen as having r represent the 3rd dimension, changing the input matrix from $n \times n \times r$ into $m \times m \times m$. This requires no manipulation of the input rather just a small adjustment to the architecture of the CNN. It does however not solve the problem of making the PCD a suitable input to a CNN. The distribution of the points in PCD is neither uniform or discrete which creates a problem when segmenting the input matrix into RFs to create feature maps in the hidden layers. A solution to this problem is to create a volumetric occupancy grid out of the PCD which is represented by a 3D matrix.

Volumetric occupancy grid have been used in 3D environmental mapping for unmanned aerial vehicles, using range detection methods such as laser scanners or sonar [29]. Occupancy grids represent the state of an environment as a matrix of randomly initialized variables, each number corresponding to a voxel, while maintaining a probabilistic estimate of the occupancy of each voxel based on incoming sensor data and previous input. The benefits of using occupancy grids are they allow for efficient estimation of free, occupied and unknown space of the working environment, even if using multiple sensory inputs that do not have the same origins or occur at he same time instance. This method discretizes the PCD from floating points in 3D space into even spaced voxels in a 3D grid [6].

3.2 Dataset selection

A range of object classification datasets have been created by research institutes and universities around the world. The focus of each dataset varies by its intended use case and only a handful are aimed at autonomous driving and traffic situations. Until recently, most labelled training data sets provided for object classification have been in the form of video footage, either monoscopic or stereoscopic.

When it comes to 3D point cloud data sets, the selection becomes more limited. The biggest and most popular dataset, the KITTI Vision Benchmark Suite was created by Karlsruhe Institute of Technology in cooperation with Toyota Technological Institute at Chicago in 2012. Along with 3D PCD, the dataset contains timestamped GPS data, stereoscopic video footage in both color and grey-scale. Another popular choice is the Stanford Track Collection, which is a compilation of segmented objects that are portrayed as a series of tracks within a customized data format known as

TrackManager. The dataset is intended for use in real time tracking applications but could be used to develop object classification applications as well.

3.2.1 Sydney Urban Object Dataset

The Sydney Urban Objects Dataset (SUOD), created by the Australian Centre for Field Robotics is a smaller dataset than KITTI but more since it only consists of PCD it is more focused on concise labelling of the 3D point cloud objects rather than focusing on the video images. As in the other dataset mentioned before the PCD was created by a Velodyne HDL-64E and each scan frame is broken down into smaller 3D point clouds consisting only of one object. The entire scans are also available before segmentation. The dataset contains 588 labelled objects of 26 classes ranging from trucks, cars and pedestrians to cyclists, traffic signs and poles. The data is supplied both in ASCII .CSV format as well as binary .BIN format and the data fields can be seen in Table 3.1.

Table 3.1: The data fields provided in the Sydney Urban Object dataset

Field name	**Description**
t	The timestamp in microseconds since epoch
intensity	The intensity of the laser return hit (0-255)
id	Laser array ID (1-64)
x,y,z	Cartesian coordinates in meters
azimuth	Azimuth angle from sensor's front face
range	Distance of laser return in meters
pid	Point ID in the original scan

Even though the dataset consists of 26 different classes, the applications that we are comparing our Sydney dataset training results with (Voxnet [6] and ORION [5]) use 14 classes to train and test on. This is done because using all 26 classes for ANN training on an ANN is quite unsuitable since 10 of the classes would contain 5 or less objects. The 14 classes used for early training efforts and validation of the neural network framework can be seen in Table 3.2. The ratio of objects used for training and testing was aimed at being 3:1.

Table 3.2: The 14 classes of Sydney Urban Object dataset used in VoxNet and ORION benchmarks [5][6].

Class ID	Training objects	Test objects	Total objects
4wd	16	5	21
building	15	5	20
bus	12	4	16
car	66	22	88
pedestrian	114	38	152
pillar	15	5	20
pole	16	5	21
traffic_lights	35	12	47
traffic_sign	38	13	51
tree	25	9	34
truck	9	3	12
trunk	42	13	55
ute	12	4	16
van	27	8	35
sum	442	146	588

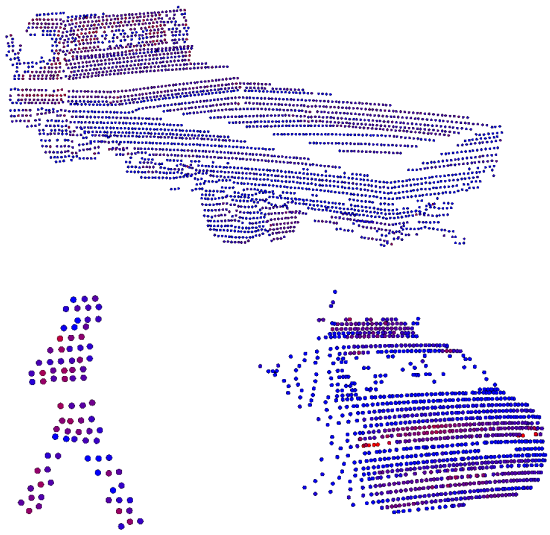


Figure 3.2: Segmented objects from the Sydney Urban Object Dataset. Top: A flatbed truck seen from the side. Bottom left: A pedestrian seen from the side. Bottom right: A sedan car seen from the back. Red points indicate high intensity return and blue indicate low intensity.

When selecting the dataset to use for training of the neural network in our project, the first criteria was that the 3D PCD was available in a form where the initial scans were either available, or could be extracted without extensive work, be segmented into clusters representing only the object of interest as this was the assumption made at the start of the project. Segmentation of point clouds without any embedded data to aid the process is a challenging problem and has its own field of research. For this particular project the focus is on the classification as the segmentation is being performed by another group of researchers at Semcon.

Having access to the intensity return of the laser points is a valuable piece of information, since using the intensity as an input to the neural network might increase the accuracy of the classification due to the fact that the location of points that return a high intensity value gives

Table 3.3: The number of total tracked objects in the Stanford Track Collection.

Class ID	Training objects	Test objects	Total objects
car	904	847	1751
cyclist	187	140	327
pedestrian	205	112	317
sum	1296	1099	2395

a whole new layer of information about the object. E.g. if the backside of a car is facing the LiDAR, highly reflective surfaces such as tail lights or a licence plate have a relatively standardized location on a normal car and utilizing this information when training a neural network to learn to recognize a car could increase the accuracy of the network.

The Australian Centre for Field Robotics has created two Linux packages intended to help the user visualize and navigate the dataset. The first of these is called *Comma* and mostly consists of functions and applications to convert between known PCD formats. The second package is called *Snark* and it is mostly used for PCD visualization and is provides useful tools to customize the visualized output.

The Sydney Urban Object Dataset was selected as the first focus of the voxelization algorithm development and initial training since it was the dataset that best fits the requirements of the ANN framework as well as the compatibility it comes to data format. A sample of the objects supplied in the dataset can be seen in Figure 3.2. Later on the voxelization process and training was extended to include two other datasets who will be discussed in more detail. Eventually the main training data will consist of KITTI data with the other two datasets being used for augmenting the classes that were the most sparse and being used for validation and benchmarking the recognition of completely unknown objects when inputted to the CNN.

The main strength of the Sydney dataset is its versatility and number of labelled classes but a weakness that makes it quite unsuitable for high accuracy Artificial Neural Network (ANN) training is the limited number of objects compared to other datasets.

3.2.2 Stanford Track Collection

The Stanford Track Collection (STC) was created with the purpose of object tracking more than classification. However it contains relatively many separate objects that can be used for object classification. The labelled objects are only classified into three classes, cars, cyclists and pedestrians. The number of individual objects in the dataset can be seen in Table 3.3.

Since all of the individual objects that occur in the dataset are represented multiple times through the different poses special care must be taken to avoid including poses of the same object in both the training set and test set. Doing so would create a bias in such a way that the network is conditioned to recognize very similar poses

Table 3.4: The number of total poses for all the tracked objects in the Stanford Track Collection.

Class ID	Training objects	Test objects	Total objects
car	92255	59173	151428
cyclist	31165	25410	56575
pedestrian	32281	22203	54484
sum	155701	106786	262487

of the same objects and when the trained weights are used on the test set it loses some of it's "unseen" quality.

Even though STC has plenty of segmented objects, one main drawback causes it to be unsuitable for training ANNs for object classification. Since there are only three classes and because they are quite distinctive in shape and size, the network can quickly learn to differentiate between them without much training. Even though this would give good results it fails to evaluate the ability of the trained network to distinguish between similar classes, which tends to be what causes issues in object classification. For this reason the Stanford dataset was more used as a large reservoir of unseen data for later testing on networks trained using a combination of diverse datasets.

To be able to use the STC dataset for voxelization it must be converted from the TrackManager format that it is provided in. This can be done using PCL, which the team in charge of the segmentation process had already experimented with. They supplied us with a simple reader written in C++ that could be modified to convert the data into individual .csv files with xyz coordinates and intensity. This was done for all available poses of the objects in dataset, which is a substantial amount of poses. The statistics can be seen in Table 3.4.

3.2.3 KITTI

The KITTI Vision Benchmark Suite is a dataset created by Karlsruhe Institute of Technology and Toyota Technological Institute at Chicago for multiple computer vision challenges with regards to autonomous driving. These include benchmarking challenges in object detection, object tracking, lane detection, visual odometry and scene flow evaluation. The dataset is more aimed towards the usage of stereo vision cameras but the set also includes Velodyne LiDAR data as ground truth. The PCD from the LiDAR is also supplied even though it is not intended as the main method of tracking and identifying objects. The point clouds are presented in three forms, firstly as raw data in tracklet form organized by the driving run, or log, that they were collected in, secondly as PCD scans organized into training and testing sets and thirdly as tracklets split into training and testing.

The latter two are intended for benchmarking different methods of object identification and tracking and the actual testing is intended to be done by a test server hosted by the institute. Object identification in this case refers to both localizing and classifying the objects. Therefore the sets are only supplied with labels and

bounding boxes for the training set, but the test set is unlabelled. This makes it impossible to test locally. Additionally the training labels for the object set are supplied in camera coordinates but not the LiDAR coordinates, but can be converted. Another reason the object identification set is poorly suited for this project is that the test set does not come with information about the location or indices of the segments in the test set, so segmentation is a required part of the object identification method.

The raw data tracklets however include labels for all the PCD frames supplied directly in the LiDAR's coordinates, so the data can be organized into labelled training and test sets locally, which results in a greater number of objects. This is exactly what was done to create the main dataset for benchmarking for this project. A reason supporting this decision was that a recent master's thesis project done at Chalmers Technical University for Volvo Car Corporation (VCC), comparing four different object classification methods used data extracted from the same KITTI raw data tracklets [30]. The dataset created for this thesis was therefore made in effort to match the size and classes of the dataset used there. This was done partly as a request from the AD-Tool team, since estimating the performance of the method against other methods that were developed for the same application purposes.

The VCC thesis involved evaluating four different classification methods as well as looking into segmentation of point cloud data. Each of the classification methods take as input a set of features extracted from the 3D PCD. The report considers two different sets of features, considering different aspects of the object classes. The four methods evaluated in the thesis work are K-Nearest Neighbours (KNN), Support Vector Machine (SVM), Random Forest Classification (RFC) and a version of a Feed Forward Neural Network (FFNN).

The same classes are used in all three of the supplied versions. The raw set includes 1235 individual objects in total, but multiple instances, or poses of the same object can be extracted from the tracklets. The classes supplied, number of individual objects in each class and the number of poses for each class can be seen in Table 3.5. It is important to mention that the total number of available logs for download in the raw dataset do not all have matching tracklet files with labels and bounding boxes for extraction. The scans that did not have matching tracklets were instead used for manual extraction of poles, since that class is need for comparison with the VCC thesis.

Table 3.5: The number of total poses for all the tracked objects in the KITTI raw tracklets.

Class ID	Individual objects	Total poses
car	932	39923
cyclist	43	1550
misc	33	1115
pedestrian	84	2340
person_sitting	16	234
tram	9	500
truck	22	1738
van	96	5408
sum	1235	52808

The tracklets for the KITTI raw dataset are supplied as xml files with information about each individual object and it's occurrences, or poses, in the included PCD files. The fields supplied in the tracklets can be seen in Table 3.6.

Table 3.6: The information fields relevant to point cloud data included in the KITTI raw data tracklets. Fields relevant to stero vision are omitted.

Field name	Description
objectType	The label of the object
h	The height of the 3D bounding box
w	The length of the 3D bounding box
l	The width of the 3D bounding box
first_frame	The index of the first occurrence of the object
poses	The number of occurrences for the object
x	The x location of the midpoint of the 3D bounding box
y	The y location of the midpoint of the 3D bounding box
z	The z location of the midpoint of the 3D bounding box
rx	The orientation of the 3D bounding box around the x-axis
ry	The orientation of the 3D bounding box around the y-axis
rz	The orientation of the 3D bounding box around the z-axis

The PCD files are supplied in .BIN format and must be converted to .CSV to allow it to be run through the voxelization algorithm. This was done by creating a C++ conversion script using the PCL library in Visual Studio 2013. After converting all the included PCD files to ASCII .CSV files that are readable by Matlab, a segmentation script was created in Matlab to extract every pose of every individual object using the bounding box information in the tracklet files and store them as individual files. This is done to allow the comparison of different voxelization algorithms without the need of extracting the object being voxelized each time.

Table 3.7: The main dataset created for benchmarking from KITTI raw data tracklets.

Class ID	Training	Testing	Validation	sum
car	6302	1765	865	8932
cyclist	429	123	55	607
misc	115	28	19	162
pedestrian	482	152	63	697
pole	165	44	22	231
truck	821	241	96	1158
van	1579	469	220	2268
sum	**9893**	**2822**	**1340**	**14055**
Ratio of total	70.39%	20.08%	9.53%	

3.2.4 Benchmarking dataset

The main dataset used for benchmarking in this report was created by walking through the raw data tracklets and voxelizing the poses of each object. Since the training of neural networks suffers greatly from unbalanced datasets and due to the fact that cars make up over 75% of the total poses in the raw data tracklets, the number of cars in the dataset created from the tracklets must be reduced. This was done by randomly picking a limited number of poses for each car object when walking through the tracklets. Since the number of poses differs for each car, care must be taken to pick a similar number of poses for each car so that one car does not appear much more often than another.

To do this, the tracklet for each car was split up into 25 segments, starting a tenth of the way into the tracklet. A pose is then picked from each segment so the same number of poses are pick if there exist 200 poses for the car or 25. However, if less than 25 poses exist, each one is used. Other classes were limited in numbers only to give a fair comparison to the VCC thesis that the method is being benchmarked against.

Since the raw data tracklets do not include poles, 231 poses of 26 individual poles were manually extracted from the raw PCD scans. This was also done to match the dataset used by the VCC thesis [30]. After segmenting a sufficient number of poses for each class, the resulting objects were split into three sets; A training set, a validation set for use during training and a testing set. Care was taken to not include poses of the same object in the same set as this would create a bias during testing on the unseen data. The resulting dataset used for benchmarking can be seen in Table 3.7.

3.3 Voxelization

The input format of the first layer of the 3D CNNs is a 32x32x32 array and therefore a pipeline of data transformations must be constructed to convert a segmented object from the point cloud scan created by the Velodyne LiDAR into a format that the network accepts. The method used was to generate an occupancy grid in the form of a matrix where the elements of the matrix represent a hit from the laser in the area. VoxNet uses so called binary occupancy grid where 1 is used to represent a hit and 0 the lack of a hit. This idea was expanded to include the intensity if the LiDAR hit as well.

The process was created using Matlab as it allows for accessible prototyping, file handling and visualization during the construction of the voxelization process. Another motivation for using Matlab to create the occupancy grid is that the original framework includes a conversion script to transform a .MAT array to numpy format stored as pickled .NPY.Z files that are compatible with the input layer of the network.

As part of the conversion to numpy arrays, the voxelized array result from Matlab is subject to zero-padding. This is essentially adding rows and columns of zeros around the edges of the 3D matrix. This is done to allow randomly chosen displacement augmentation, or "jittering", during training. During the process, the entire content of the array is randomly shifted up or down one seat in the occupancy grid with respect to one of it's three dimensions. This is done as a way of injecting noise to the networks input during training to reduce the risk of overfitting.

3.3.1 Intensity occupancy grids

Two methods were considered to store the intensity of the point cloud data in the voxelized matrix, the first of which is to represent each point by a normalized intensity value, e.g. as floating point numbers in the range [0, 1] or as 8 bit unsigned integers in the range [1, 255], which is how the original data is stored by the Velodyne sensor.

The second method was to use predetermined thresholds to create intensity ranges and represent a hit within each of these ranges with a different integer. The intensity ranges that showed the most promising results and the integers used to represent each intensity range can be seen in Table 3.8.

Table 3.8: The intensity ranges used in the threshold intensity voxelization method and the integers used to represent them in the voxelized matrix.

Intensity ranges	Integer representation
No hit	0
1-129	1
130-239	2
240-255	3

The threshold depicted in Table 3.8 were suggested by the AD-Tool team since they had experimented with intensity mapping and locating highly reflective surfaces as points of interests in scans, such as vehicle licence plates, headlights and tire rims. Figure 3.3 shows the results of the voxelization of a car seen from the back from the Sydney dataset using Matlab. This image shows the intensity before the threshold method is applied. The taillights of the car can be clearly seen as bright pink boxes behind the slightly less reflective purple points of the trim and glass surrounding the lights. Also noticeable is the licence plate of the car as another set of purple points. The rest of the car mostly comprises of light blue points representing low intensity hits. The same car can be seen in Figure 3.2 as a segmented points cloud for comparison.

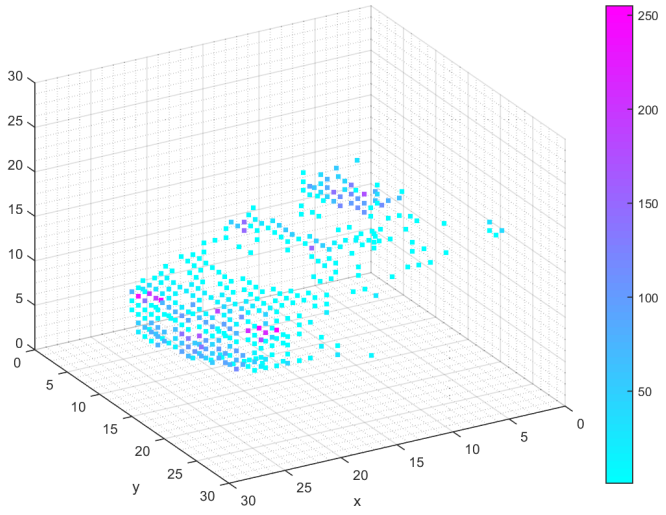


Figure 3.3: A car seen from the back from SUOD after being voxelized in Matlab. The colors of each voxel represent the intensity of the hit before applying the threshold method.

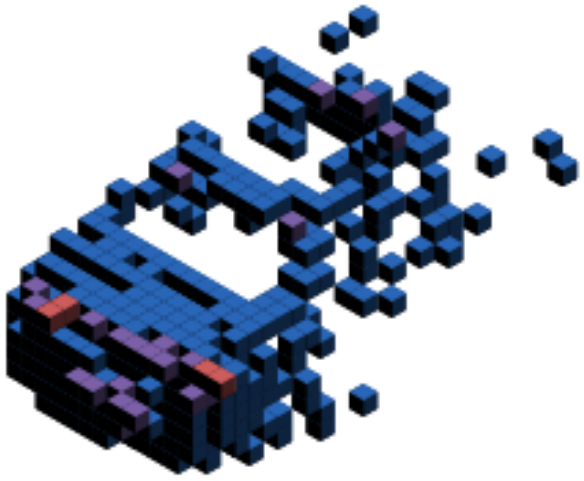


Figure 3.4: A car seen from the back from SUOD after being voxelized using the threshold method. The colors of each voxel represent the intensity threshold of the hit.

After applying the threshold method to the voxelized data and converting the voxelized data to numpy format, the data can be visualized in the exact same form that the network's input sees. Figure 3.4 shows the same car in its final voxelized form. The red voxels indicate high intensity hits, purple indicates medium intensity and blue indicate the lowest intensity hits.

3.3.2 Voxelization area scaling

A recurrent issue when voxelizing point clouds in this manner is the choice of resolution of the voxelized area, which is essentially the chosen size of each voxel edge. In [6] the issue is encountered and voxel sizes of several different values are examined. The

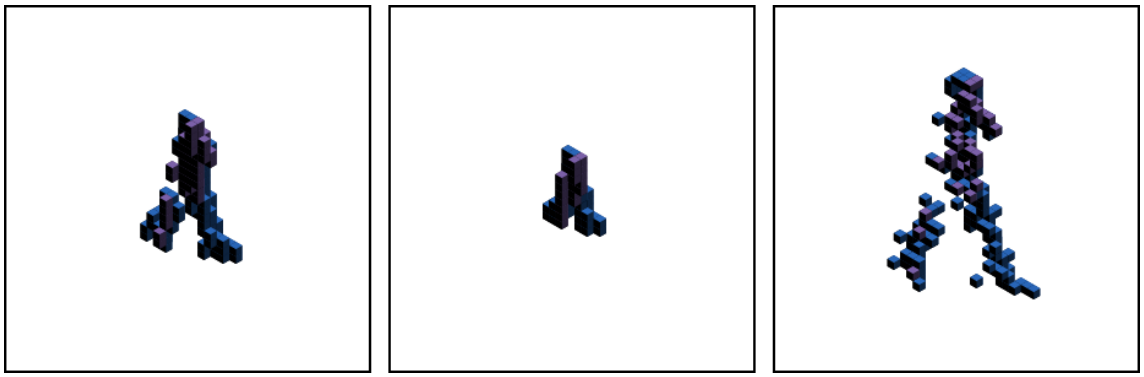


Figure 3.5: Comparison of the results of a voxelizing a pedestrian from SUOD using fixed voxel sizes and the adaptive voxel size method. The left-most pane shows used a fixed voxel size of 0.1 m, the middle pane used 0.2 m and the right-most pane used dynamic voxel-size where the method calculated a voxel size of 0.061 m.

results concluded that for urban objects such as are encountered in datasets like SUOD, STC and KITTI, the most appropriate resolution was determined to be obtained using between 0.1 m and 0.2 m as the voxel edge length. However, neither of these values showed reliable results for both large and small objects. If the voxelization area is too small it will cut off parts of the bigger items used as inputs, which in this case would be objects like trucks and buildings and if the voxelization area is chosen as too large it will shrink smaller objects such as pedestrians and cyclists into a voxel cluster of a relatively small size in the middle of the voxelized space.

With 32x32x32 voxels this translates to a box with an edge length of 3.2 m to 6.4 m respectively. A method was proposed to work as a compromise where two identical networks where trained on two inputs voxelized with separate resolution and the outputs of both were combined via a softmax layer. This turned out to results in the best mean accuracy recorded in the paper for the Sydney dataset.

After experimenting with voxelizing SUOD using different resolutions, an alternative method was examined. This includes calculating the xyz dimensions of the object being segmented and choosing the dimension that was the largest and used that as the edge length for the total voxelized area. The individual voxel length is then calculated by dividing this longest axis dimension by the number of voxels, which in our case is 30.

Figures 3.5 and 3.6 show the advantages of using dynamic scaling for the voxel size. In the left most pane of both figures a voxel size of 0.1 m is used. This causes the pedestrian to be relatively well defined but most of the bus is cut from the voxelization space. This voxel size was used in previous work to suit medium sized vehicles such as cars well. In the middle pane a fixed voxel size of 0.2 m is used. This suits the bus much better, which is almost fully included in the voxelization space, but the pedestrian is reduced to occupy only very few voxels in the occupancy grid. This voxel size was used in previous work to suit large objects such as trucks,

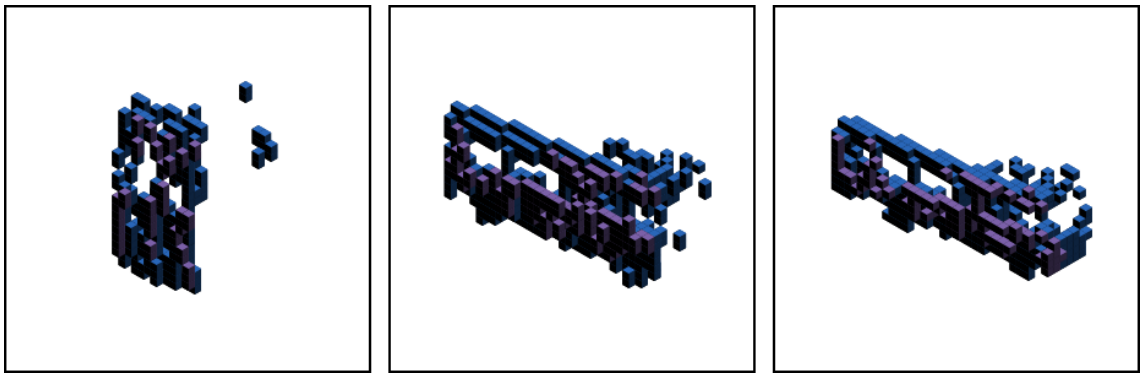


Figure 3.6: Comparison of the results of a voxelizing a bus from SUOD using fixed voxel sizes and the adaptive voxel size method. The left-most pane shows used a fixed voxel size of 0.1 m, the middle pane used 0.2 m and the right-most pane used dynamic voxel-size where the method calculated a voxel size of 0.236 m.

buses, buildings and trees.

In the final right-most pane, the dynamic voxelization was used to calculate the minimum voxel size that includes the full object. For the pedestrian this turned out to be 0.061 m per voxel and for the bus it was 0.236 m per voxel. This makes sure that all objects are fully included and no information about occupancy of points is lost when shown to the network. in previous work multiple networks where trained on voxelized data that used different voxel sizes and a soft max layer was used to merge the results. This is no necessary using our method.

A downside of using the dynamic scaling method is that information about the size of the object is lost. In this way, a car can have very similar shape as a truck and unless the network gets information about the scale of the object it has less information to go by to predict the class correctly. As a solution to this, the distance that each object occupies on the x, y, z axis is inputted straight into the fully connected layer of the network, therefore bypassing the convolutions and giving the network pure size information about the object. This is explained in more detail in Section 3.1.3.

3.3.3 Voxelization algorithm

The Matlab voxelization function created for the project uses a simple and fast algorithm that only requires stepping through the points of the segmented object a single time. Firstly the midpoint of the object is calculated and the dimension of the axis that spans the longest distance, d_{max}, is found. This longest axis dimension is then used to create the voxelization area that spans from the midpoint to the distance $d_{half} = \frac{d_{max}}{2}$ in all directions from it. This voxelization area is then split up into 30 segments on all three axes and the distance d_{vox}, representing the side length of each individual voxel is created. Next vector division with a ceiling function is used to calculate the appropriate matrix seat for each point in the segmented object. A key line in the voxelization algorithm can be seen in Equation 3.1.

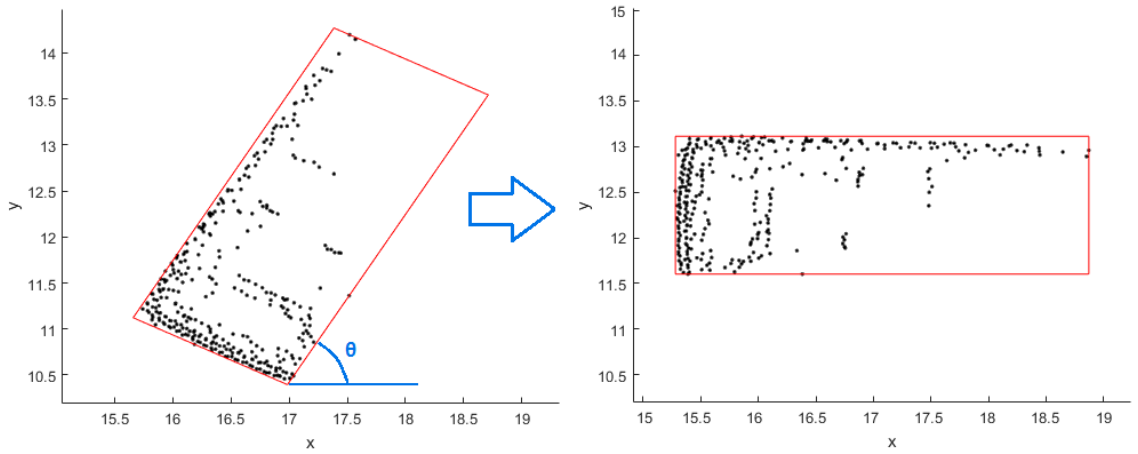


Figure 3.7: The bounding box calculated to determine the angle of an object from the xz-plane and rotate it to a nominal orientation.

$$M_{vox} = \mathbf{ceil}\left(\frac{\overline{x} - x_{start}}{d_{vox}}\right) \tag{3.1}$$

Where M_{vox} is the voxelized 3D matrix, $\overline{x}$ is a 3xN vector containing the xyz coordinates for each point in the segmented object and x_{start} is a 3x1 vector containing the lower boundary of the voxelization area on all three axes. The **ceil** function used here is a mathematical operator that returns the smallest integer that is larger than the input.

Since the intensity of each point is preserved in the voxelization, special care is taken to not overwrite a lower intensity hit with a higher one. This could overwrite important information in the voxelized object, e.g. the high intensity points representing a car's headlights being overwritten by the low intensity points next to them representing the bumper.

3.3.4 Orientation correction

When dealing with objects from a real traffic scenario, the objects can be orientated in any manner, determined by the orientation of the sensor and the object being recorded. Even though the 3D CNN is observant to spatial localities and can recognize shapes occurring in different parts of the matrix space, it is a good idea to normalize the different possible orientations of input in some way. In VoxNet [6] and ORION [5] this is done through rotation augmentation, where the input is augmented by rotating it a N number of times and using all N versions of the object essentially as one input. To allow an output to be formed from this, a majority vote is calculated between the augmented versions of the input.

An alternative is to create a method that can determine the orientation of the object in some way and rotate it to a nominal orientation before voxelizing and processing the object. Our solution to this is to calculate the smallest convex hull that envelops every point in the segmented object and drawing the smallest possible rectangular bounding box around the convex hull. The angle θ, which lies between

the longer of the rectangles sides and the xz-plane can then be calculated and all the points in the objects can then be rotated by θ. Figure 3.7 shows this process implemented on a segmented car from the KITTI dataset. The car is shown in a top-down view and the angle θ can be seen between the one of the longer of the bounding boxes sides and the xz-plane.

3.4 Training

Training the model is done using a Nvidia GTX 970 graphics card for which CUDA is required in order to utilize GPU acceleration. The voxelization algorithm generates integer arrays from the training set that are converted to a numpy.ndarray and stored in compressed (pickled) file structures.

The different sub-sets of the dataset used in machine learning have different naming conventions depending on the field of research. Here the sub-set used to iterate through and train on is called the *training set* and the completely unseen data used to test the trained weights after training is done is called the *test set*. The sub-set used to check for overfitting during training is called the *validation set*. Before training starts, the training file structure containing all the objects to be trained on is shuffled so that the object classes are randomized before training. The training is run for a maximum of 80 epochs and after 20 epochs validation tests are done on unseen data to minimize model overfitting to the training set. Every time that the validation returns a better result than previous epochs, the corresponding weights are stored. If overfitting is detected, the resulting weights are not saved.

An epoch is counted as one iteration through the entire training set. The training framework uses batching to better utilize memory. This is done by grouping up a number of objects and train on them in the same instance. In our case, a iteration size of 16 objects was used and each chunk that is processed in the same instance contains 32 batches. An iteration is considered the training of a single batch. Because of this, the number of iterations and the number of epochs are not the same when training on datasets of different sizes.

The model is loaded into the training script from a config file, where general parameters like input shape, number of classes, learning rate etc. and the model architecture i.e. layer structure and weights initializations are stored. Since training is done using a GPU with the cuDNN installed, both the weight matrices and the layer outputs are stored as *CudaNdarrays*. This greatly speeds up the process of training and testing using the model, however since the model utilizes the custom VoxNet *Conv3dMMLayer* objects as convolutional layers, testing the model on a CPU is not possible without rewriting or swapping out these objects.

During the training process, parameters can be altered between epochs to optimize the result. The main parameter scheduling that the training process benefits from is learning rate. Best practice is to slowly decrease the learning rate to allow the network to learn the general aspects of the training set quickly in the beginning of

the training process and slowly fade out into learning the more subtle details that separate the object classes. If the learning rate is set too high in the beginning of the training, the results will not converge at all, so care must be taken to find an appropriate initial value before lowering the learning rate. In the original VoxNet framework this was done in fixed steps, so that if the number of iterations through the training set reaches some predetermined value, the learning rate is lowered by a set amount. Since the number of iterations needed changes if the number of training objects are changed, this approach was very sensitive to dataset changes.

An alternative way of scheduling the learning rate was therefore chosen. The learning rate was tied to an exponential decay function which would update after each chunk. A number of different mathematical functions were tested for learning rate scheduling, including many variations of exponential decay as well as Gaussian decay and linear decay. The best results were found when using exponential decay with a value of α somewhere between 0.9950 and 0.9999, depending greatly on the size of the training set. For the main benchmarking dataset gathered from the KITTI raw tracklets described in Table 3.7, the optimal value was found to be $\alpha = 0.9993$. The new learning rate was updated using Equation 3.2.

$$lr_{new} = \alpha \cdot lr_{old} \tag{3.2}$$

The testing process is similar to the training in the fact that the testing script is run on a compressed pickle file that contains the entire test set. The file is iterated through in order and run through the trained weights. The predicted class and the true class of each object are stored and used to calculate the accuracy of each class as well as the mean accuracy and macro recall. Both training and testing scripts generate symbolic tensor- functions and variables which get substituted with data when iterating through the input matrices. For each batch a loss function gets computed and accuracy is calculated. During training the outputs of these tensor functions are used to backpropagate through the CNN and the weight matrices get adjusted.

3.5 Tracking

Multi-object tracking in real world scenarios is a challenging problem that is the subject of constant research and development. The level of complexity is mainly caused by the number of targets constantly changing, making it difficult to consistently keep track of a single target's identity. Knowledge about the class of an object that is being tracked can greatly reduce the number of possible matches between frames and semantic knowledge such as the possible velocity of the object can help even further. E.g. if a target that is to be tagged as a pedestrian is moving at 80 km/h down the highway, it is most likely incorrect. Conversely, knowledge about the tracked identity of an object can give the classification method more information to go on. E.g. if an object has been tracked for multiple frames in a row with the same tracking index and has been classified through all those frames as a car and the identity does not change between frames, it is likely that the next frame is also

going to be a car.

A common occurrence is that a target switches identities for only a few frames, sometimes only a single frame, before being consistently tracked once again. If a classification task were to be run by the side of the tracking algorithm, this behaviour could be avoided by looking at the predicted class of the previous instances of the object. If an object that has been tracked for 20 frames as a car suddenly changes into a cyclist, it hints that an identity mismatch might be taking place. Even though a single frame of classification change might only be the result of a bad segment or a classification mistake, a longer lasting mismatch that eventually changes back into the original identity can be used to retrospectively fix the tracklet.

This idea was discussed during the process of the thesis work and since VCC has a storage of logs collected from real world driving scenarios, a handful of these were supplied to us to experiment with the practical usage of the classification method, in addition to benchmarking against the other classification methods. Since the logs are unlabelled, visual inspection was the only available form of performance evaluation.

It soon became clear that the dataset trained from KITTI for benchmarking was poorly suited for this task and a great improvement was found by intentionally not training on the 'misc' class, since it is essentially intended as a dump class for objects that are not of interest. Instead, the network is only trained on objects that are valuable in terms of tracking. Cars and vans were merged since the distinction between the classes is not important when it comes to the traffic behaviour and semantic knowledge for training. The network is therefore trained to recognize cars, pedestrians, cyclists, trucks and poles. Large structures and trees would be excellent candidates for training classes but the lack of training data prevented this.

3.5.1 Confidence Threshold

Instead of attempting to recognize miscellaneous objects, which could be of any shape and size, a confidence interval was introduced to quantify the certainty of the network's prediction for the classes it should recognize. Every time the network gives a prediction about the class of an object, each output of the neural network gets a score. In the original framework the highest scoring class is chosen, without considering the score of the other classes and whether their prediction score was close to that of an optimal score for that class.

Two methods were explored to achieve this, the first was done by normalizing the prediction values of all the classes being trained on and giving them each a percentage score of the total. That way a threshold could be chosen such that the prediction must have a minimum score before being classified. The other method involved gathering prediction data during training and using it to estimate the certainty of the predictions during testing.

Every time the validation set gets a new best accuracy score the correct predictions for each class are saved, making sure that we are comparing only to relatively good prediction scores when determining a confidence threshold. After training is done,

the gathered data is then accessed in the test script and the mean and standard deviation of the set is calculated. This information can be then be used to determine a threshold for each class that a prediction score must be above to be accepted. This method was used only as a practical visualization of the classification and since there is no ground truth available for the data, the performance could not be readily quantified.

4
Results

Test runs were done both on networks trained on the Sydney- and KITTI data described in Section 3.2. The Sydney data was used to compare the additions and improvements made on the original framework and compared to test runs from VoxNet [6] and ORION [5]. The KITTI data was used to evaluate the method on a more realistic urban scenario as well as to allow comparison with the methods evaluated in the VCC classification thesis [30].

Since different evaluation systems are used to compare the results of the VoxNet and ORION on one hand and the VCC thesis on the other hand, the different measurement systems must be explained briefly. Total accuracy is the ratio between the number of correct predictions and the total number of test objects in the set. Class accuracy is the same ratio for each individual class and mean accuracy is the average of the individual class accuracy scores. Recall is the ratio between the number of correct predictions for each class and the total number of predictions for that same class. Macro recall is the average of the recall scores for all classes.

The main results of the VCC thesis are presented as total accuracy and mean accuracy, while VoxNet and ORION only give the result as F1 score. F1 score is an accuracy measure that takes into account both accuracy and recall and it can be calculated using 4.1.

$$F1\ Score = 2 \cdot \left(\frac{Accuracy \cdot Recall}{Accuracy + Recall} \right) \tag{4.1}$$

4.1 Comparison with VoxNet and ORION

Table 4.1 shows the results of testing on the trained network using the dataset described in Table 3.2 in Section 3.2.1. Table 4.2 shows the accuracy of each class as well as the mean accuracy calculated over all classes. To match the size of the datasets used in VoxNet and ORION for fair comparison of our additions and improvements, rotation augmentation with 12 rotations was used.

The total accuracy for the test set is 83.51%, the mean accuracy is 64.46% and macro recall is 79.45%. The calculated F1 score is 81.43%.

Table 4.1: The confusion matrix for a test run of the 14 class dataset from SUOD used to compare to previous work.

Pred/True	4wd	building	bus	car	pedestrian	pillar	pole	traffic_lights	traffic_sign	tree	truck	trunk	ute	van
4wd	1	0	0	2	0	0	0	0	0	0	0	0	0	2
building	0	5	0	0	0	0	0	0	0	0	0	0	0	0
bus	0	2	1	0	0	0	0	0	0	0	0	0	0	1
car	0	0	0	21	0	0	0	0	0	0	0	0	0	1
pedestrian	0	0	0	0	38	0	0	0	0	0	0	0	0	0
pillar	0	0	0	0	0	2	1	1	0	0	0	1	0	0
pole	0	0	0	0	0	0	4	0	0	0	0	1	0	0
traffic_lights	0	0	0	0	1	0	0	9	1	0	0	1	0	0
traffic_sign	0	0	0	0	1	0	0	2	8	0	0	2	0	0
tree	0	0	0	0	0	0	0	0	0	9	0	0	0	0
truck	0	1	0	0	0	0	0	0	0	0	1	0	0	1
trunk	0	0	0	0	0	0	1	1	0	0	0	11	0	0
ute	0	0	0	3	0	0	0	0	0	0	0	0	1	0
van	0	1	0	2	0	0	0	0	0	0	0	0	0	5

Table 4.2: The class accuracy for the 14 class dataset from SUOD used to compare to previous work.

Class ID	Accuracy
4wd	20.00%
building	100.00%
bus	25.00%
car	95.45%
pedestrian	100.00%
pillar	40.00%
pole	80.00%
traffic_lights	75.00%
traffic_sign	61.54%
tree	100.00%
truck	33.33%
trunk	84.62%
ute	25.00%
van	62.50%
Mean accuracy	64.46%

A comparison with the best results of the previous work can be seen in Table 4.3. We compare the results on the Sydney dataset from VoxNet [6] and ORION [5], the second of which builds upon the work of the first with improvements in orientation

estimation. ORION essentially turns the task of classification into a two step task, firstly to estimate the orientation of the object and secondly the class label. Both tasks are performed using deep neural networks.

Both articles use F1 score as their main result, however only the result is given by class in ORION. Four different version of ORION are considered in the article, we will only compare our results to the best resulting method. Since no confusion matrices or specific class accuracies are supplied in the articles, comparison to other evaluation scores than the F1 score is impossible for VoxNet and ORION. Regardless, the total and mean accuracy scores in addition to macro recall are listed in the comparison table.

Table 4.3: The results of the Sydney dataset tests compared to the best results of previous work.

Method	F1 score	Total accuracy	Mean accuracy	Macro recall
Our results	81.43%	83.51%	64.46%	79.45%
VoxNet	72%	N/A	N/A	N/A
ORION	77.8%	N/A	N/A	N/A

4.2 Comparison with VCC Thesis Classification Work

Table 4.4 shows the results of testing on the trained network using the dataset described in Table 3.7 in Section 3.2.4 using binary voxelization, therefore not utilizing intensity information. Table 4.5 shows the calculated accuracy scores of each class for both methods as well as the mean accuracy calculated over all classes.

The total accuracy for the test set is 94.90%, the mean accuracy is 92.34% and macro recall is 98.08%.

Table 4.4: The confusion matrix for the main dataset used for benchmarking using binary voxelization.

Pred/True	car	cyclist	misc	pedestrian	pole	truck	van
car	1752	0	0	0	0	0	13
cyclist	0	123	0	0	0	0	0
misc	1	0	21	0	0	5	1
pedestrian	1	0	0	151	0	0	0
pole	0	0	0	0	44	0	0
truck	1	0	0	0	0	236	4
van	118	0	0	0	0	0	351

Table 4.5: The class accuracy for the main dataset used for benchmarking using binary voxelization.

Class ID	Accuracy
car	99.26%
cyclist	100.00%
misc	75.00%
pedestrian	99.34%
pole	100.00%
truck	97.93%
van	74.84%
Mean accuracy	92.34%

Table 4.6 shows the results of testing on the trained network using the dataset described in Table 3.7 in Section 3.2.4 using the intensity threshold method described in Section 3.3.1. Table 4.7 shows the calculated accuracy scores of each class for both methods as well as the mean accuracy calculated over all classes.

The total accuracy for the test set is 96.35%, the mean accuracy is 95.06% and macro recall is 98.60%.

Table 4.6: The confusion matrix for the main dataset used for benchmarking using intensity threshold voxelization.

Pred/True	car	cyclist	misc	pedestrian	pole	truck	van
car	1753	0	0	0	0	0	12
cyclist	0	123	0	0	0	0	0
misc	0	0	24	0	0	2	2
pedestrian	1	1	0	150	0	0	0
pole	0	0	0	0	44	0	0
truck	1	0	0	0	0	240	0
van	84	0	0	0	0	0	385

Table 4.7: The class accuracy for the main dataset used for benchmarking using intensity threshold voxelization.

Class ID	Accuracy
car	99.32%
cyclist	100.00%
misc	85.71%
pedestrian	98.68%
pole	100.00%
truck	99.59%
van	82.09%
Mean accuracy	95.06%

A comparison with the results of the VCC thesis work can be seen in Table 4.8. Four classification methods are considered in the report using two different feature sets extracted from the 3D point cloud data. One of the feature sets had better results for all four considered methods so we will compare our results to them. Since a confusion matrix is supplied for each of classification method's result, the macro recall and F1 score could be calculated to compare to even though the scores are not specifically listed in the thesis paper [30].

Table 4.8: The results of our KITTI tests compared to the results of the VCC thesis work.

Method	Mean accuracy	Total accuracy	Macro recall	F1 score
Our results (intensity)	95.06%	96.35%	98.60%	97.46%
Our results (binary)	92.34%	94.90%	98.08%	96.46%
VCC thesis SVM	73.36%	96.08%	82.31%	88.66%
VCC thesis KNN	66.82%	93.99%	80.23%	93.99%
VCC thesis FFNN	77.83%	95.87%	83.38%	89.19%
VCC thesis RFC	74.85%	95.14%	80.79%	87.38%

5
Discussion

In this thesis an object classification method was implemented using a deep convolutional neural network. The focus was on the utilization of the 3D information inherent in LiDAR point clouds, such as the original shape and size of an object and the intensity hits of each LiDAR point.

Looking back over course of this thesis work, having finished implementing a working CNN framework and produced compelling results, it can be said that our motivation for choosing this approach for solving the problem of object classification of a 3D point cloud is rightly justified. Same can be said of the improvements added onto the VoxNet framwork, in order to focus the method of 3D CNNs to work with urban object classification.

Implementing this method on a training set of 9893 objects and test set of 2822, both composed of 7 classes resulted in; an F1 score of **97.46%**, total accuracy of **96.35%**, mean accuracy of **95.06%**, and macro recall of **98.60%**.

5.1 Improvements

The results of the comparison with previous work indicate that the additions and changes made to the underlying framework did indeed show an improvement in results. These include the utilization of intensity in training, dynamic voxelization scaling and the feature injection to the fully connected layer both convolutional layers described in Section 3.1.3.

The comparison of using all the above mentioned improvements except for intensity in the comparison using the dataset gathered from KITTI tracklets further indicate that the utilization of intensity does improve classification results. We believe this to be mainly due to the fact that highly reflective surfaces tend to be located in similar relative locations for many of the object classes considered. An example of this is the headlights and licence plate of a car.

5.2 Impediments

As evident by the results in Tables 4.6 and 4.7, a large portion of the false positives were between classes that had many similarities. In our experience that was the case with cars and vans. In both the training and test sets there are blurred lines between what constitutes a car and gets classified as a van. In order to limit these classification errors we attempted implementing a trump voting specialist to supersede the

main CNN when it would classify an object as either a car or a van. This specialist was trained on 4 segments of training data, each containing a balanced number of cars and vans, that the training algorithm would loop through during training. This produced no improvements to the total accuracy and analysis of the results lead to the conclusion that even with focusing on only those two classes during training and exposing the CNN to equal number of cars and vans, only ended up reaffirming the errors made by the main CNN since the training data for both networks have the same origins in the KITTI data set.

5.3 Future Work

The focus of the thesis was to evaluate the viability of using neural network classification methods that are commonly used on 2D images without sacrificing 3D information such as the spatial locality of point cloud data and the intensity hits of the points. To do this, the datasets and classes used had to be easy to benchmark against other methods and the choices made during the creation of the main dataset reflect that. As the project developed however, the focus shifted to the gain of using classification to aid in object tracking by running the classification method on tracklet data.

It is our conclusion that this aspect of the project holds the most promise for practical application until multi-class training can be done on datasets much larger and more versatile than what researchers have access to today. Implementing a confidence threshold such as is described in Section 3.5 has shown great promise in practical examples and further training a network on only the objects that are desired does aid the network in recognizing them. By limiting the amount of false positives as well by introducing a confidence threshold, the overall performance of the classification is improved.

Another avenue of research that we believe could be beneficial, especially when it comes to very similar classes like cars and vans, is to train specialized networks for each class. It can be done in one of two ways; each class has a CNN that is trained to recognize only that class with a binary output, or specialized CNNs that are trained using all classes but with a bias in the number of objects that each CNN trains on. Theoretically this would result in a majority of the specialized CNNs agreeing on what object is being run through the network.

Bibliography

[1] C. C., "Simple Neural Network Example by Murx~enwiki." [Online]. Available: https://creativecommons.org/licenses/by/2.0/

[2] A. Ng, J. Ngiam, C. Y. Foo *et al.*, "Convolutional Neural Network UFLDL Tutorial," *Stanford*, September 2015. [Online]. Available: http://ufldl.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork/

[3] H. Lee, "Tutorial on Deep Learning and Applications," 2010. [Online]. Available: http://www.slideshare.net/CloudyNguyen2/2010-deep-learning-and-unsupervised-feature-learning

[4] Y. LeCun, Y. Benigo, and G. Hinton, "Deep Learning," *Nature*, May 2015. [Online]. Available: http://dx.doi.org/10.1038/nature14539

[5] N. Sedaghat, M. Zolfaghari, and T. Brox, "Orientation-boosted Voxel Nets for 3D Object Recognition," *arXiv:1604.03351v1*, April 2016.

[6] D. Maturana and S. Scherer, "VoxNet: A 3D Convolutional Neural Network for Real-Time Object Recognition," *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems*, September 2015.

[7] P. J. Werbos, *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting.* John Wiley & Sons, 1994.

[8] M. Fischetti, "Computers versus Brains," *Scientific American*, November 2011. [Online]. Available: http://www.scientificamerican.com/article/computers-vs-brains/

[9] Y. LeCun, B. E. Boser, J. S. Denker *et al.*, "Handwritten Digit Recognition with a Back-Propagation Network," 1989. [Online]. Available: http://yann.lecun.com/exdb/publis/pdf/lecun-90c.pdf

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[11] D. Habermann, A. Hata, D. Wolf, and F. Osório, "Artificial Neural Nets object recognition for 3D point clouds," *Proceedings of 2013 Brazilian Conference on Intelligent Systems*, p. 101 to 106, 2013.

[12] M. Himmelsbach, F. v. Hundelshausen, and H.-J. Wuensche, "Fast Segmentation of 3D Point Clouds for Ground Vehicles," *2010 IEEE Intelligent Vehicles Symposium*, June 2010.

[13] R. B. Rusu, N. Blodow, and M. Beetz, "Fast Point Feature Histograms (FPFH) for 3D Registration," *2009 IEEE International Conference on Robotics and Automation (ICRA)*, May 2009.

[14] M. B. Soares, P. Barros, G. I. Parisi, and S. Wermter, "Learning objects from RGB-D Sensors using Point Cloud-based Neural Network," *Proceedings of 23th European Symposium on Artifical Neural Networks, Computational Intelligence and Machine Learning*, University of Hamburg, 2015.

[15] K. Klasing, D. Wollherr, and M. Buss, "A Clustering Method for Efficient Segmentation of 3D Laser Data," *2008 IEEE International Conference on Robotics and Automation*, May 2008.

[16] A. Milan, S. H. Rezatofighi, A. Dick, K. Schindler, and I. Reid, "Online Multi-target Tracking using Recurrent Neural Networks," *arXiv:1604.03635v1*, April 2016.

[17] M. A. Nielsen, *Neural Network and Deep Learning*. Determination Press, 2016.

[18] A. Karpathy, "CS231n Convolutional Neural Networks for Visual Recognition," 2015. [Online]. Available: http://cs231n.github.io/neural-networks-1/

[19] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *University of Toranto*, September 2012.

[20] L. lab, "Convolutional Neural Networks (LeNet)," *Deeplearning.net*, June 2016. [Online]. Available: http://deeplearning.net/tutorial/lenet.html

[21] D. Britz, "Introduction to RNNs," September 2015. [Online]. Available: http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/

[22] T. Deyle, "Velodyne HDL-64E Laser Rangefinder (LIDAR) Pseudo-Disassembled," January 2009. [Online]. Available: http://www.hizook.com/blog/2009/01/04/velodyne-hdl-64e-laser-rangefinder-lidar-pseudo-disassembled

[23] "Velodyne LiDAR HDL-64E Specifications," 2016. [Online]. Available: http://velodynelidar.com/hdl-64e.html

[24] J. Rendleman, "Engines of change," August 2007. [Online]. Available: https://gcn.com/articles/2007/08/03/engines-of-change.aspx

[25] L. lab, "Deeplearning Theano Installation," *Deeplearning.net*, June 2016. [Online]. Available: http://deeplearning.net/software/theano/install.html

[26] N. Corporation, "Cuda Toolkit & Samples (Nvidia)," *NVIDIA Developer*, June 2016. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[27] M. Nandi, "Faster deep learning with GPUs and Theano," *Domino Datalab*, August 2015. [Online]. Available: https://blog.dominodatalab.com/gpu-computing-and-deep-learning/

[28] Lasagne, "Lasagna Installation," *https://lasagne.readthedocs.io*, June 2016. [Online]. Available: https://lasagne.readthedocs.io/en/latest/

[29] K. Schauwecker and A. Zell, "Robust and Efficient Volumetric Occupancy Mapping with an Application to Stereo Vision," *Universität Tübingen*, June 2014. [Online]. Available: http://www.ra.cs.uni-tuebingen.de/publikationen/2014/schauwecker_icra2014.pdf

[30] P. Nygren and M. Jasinski, "A comparative study of segmentation and classification methods for 3d point clouds," 2016.

A

Appendix

This section contains visuals that were considered too bulky to be included in the main text but do however add context and help the reader visualize the training process and the output of the neural network.

Figure A.1 shows a sample of a training report, a vital tool used to manage the learning rate scheduling of the training process as well as indicating if overtraining is occurring.

Figure A.2 shows eight randomly selected objects and their predicted and trued labels as well as an isometrically rendered image of the object. The colors in the rendered image indicate the intensity level of the points in each individual voxel, red voxels indicate a high intensity hit, purple indicate a medium intensity hit and the blue voxels represent a low intensity hit.

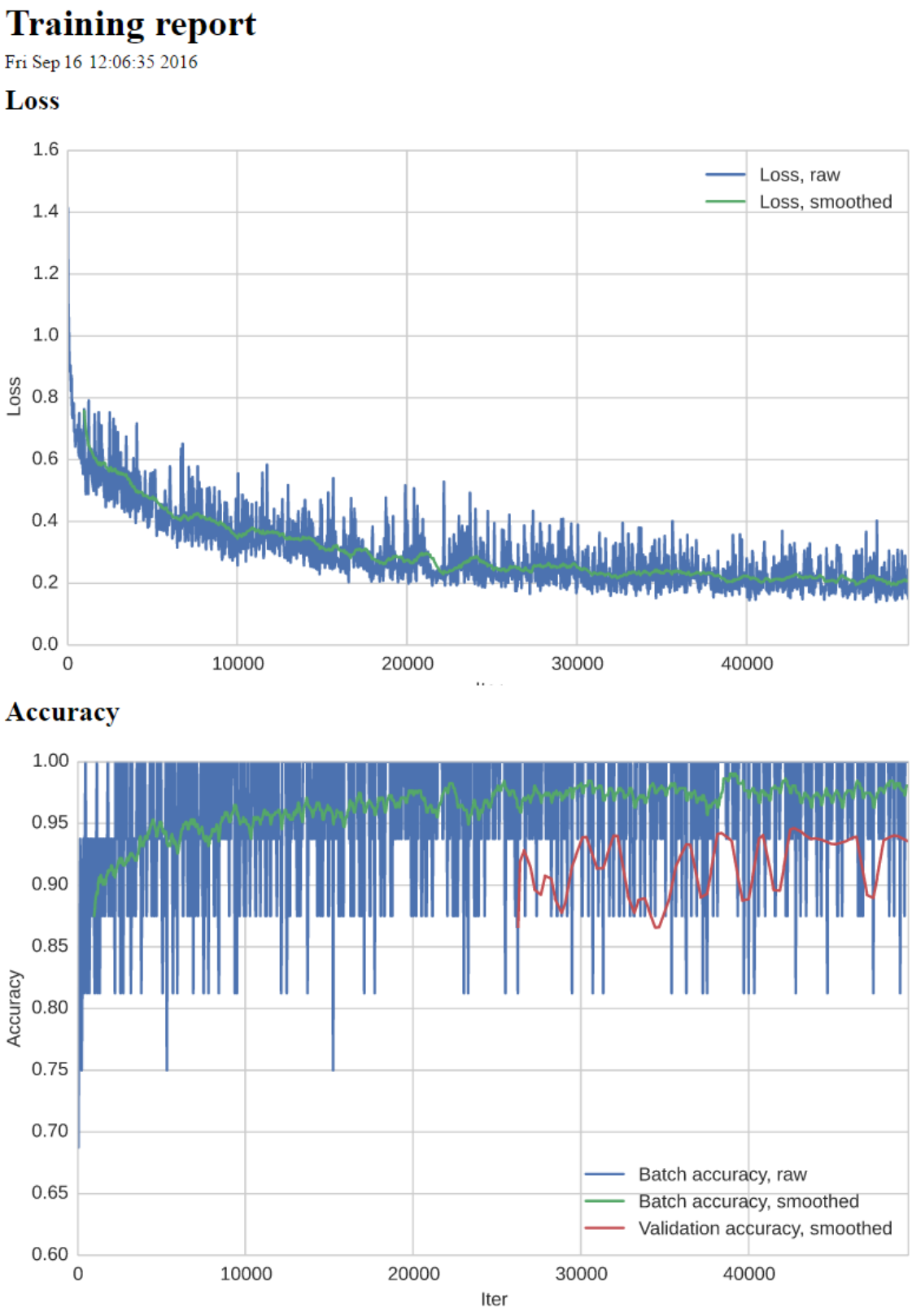


Figure A.1: Training report from a training session of the CNN architecture on the main benchmarking dataset.

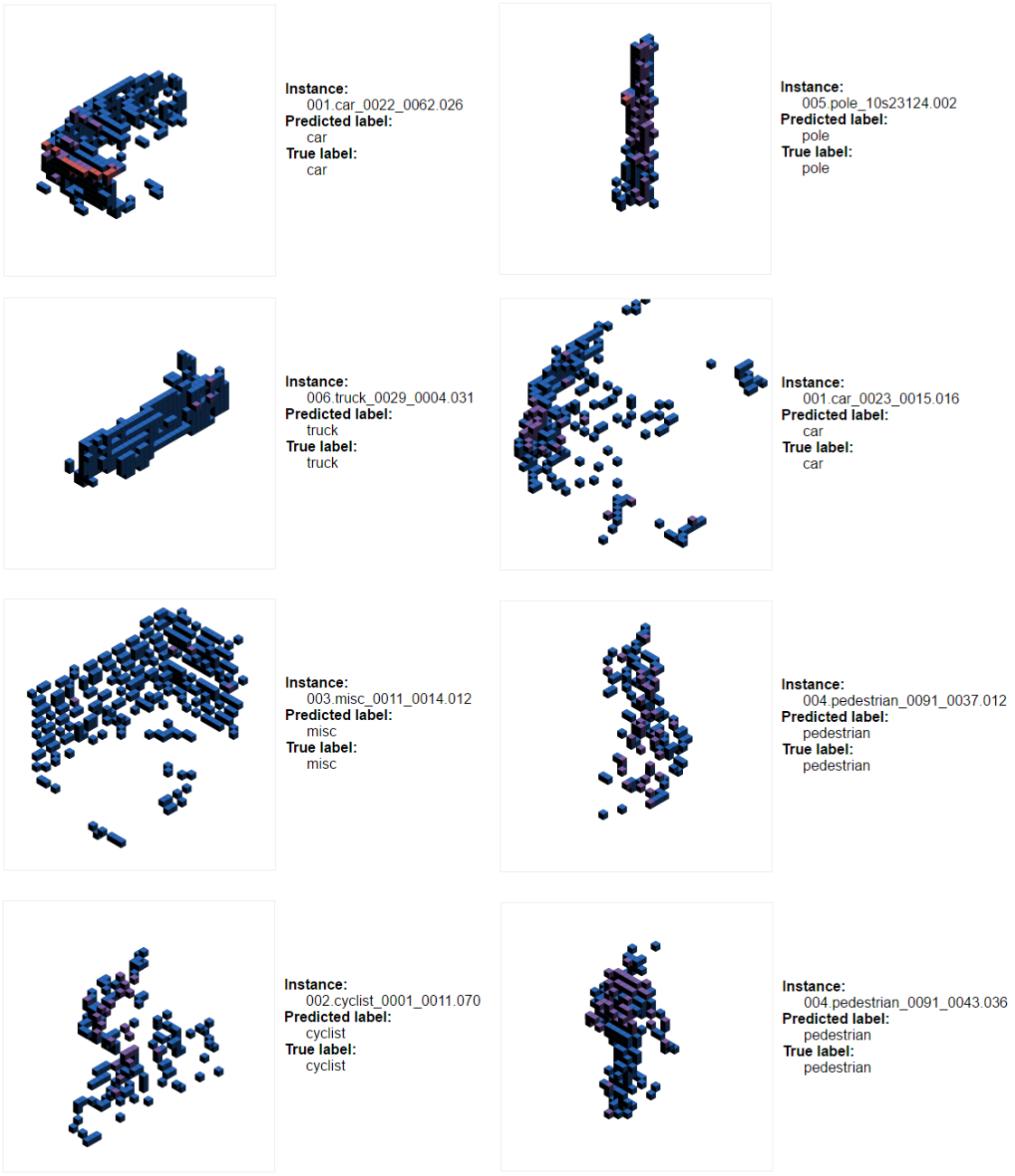


Figure A.2: Isometrically rendered examples from a test run on the main benchmarking dataset constructed from KITTI raw tracklets.