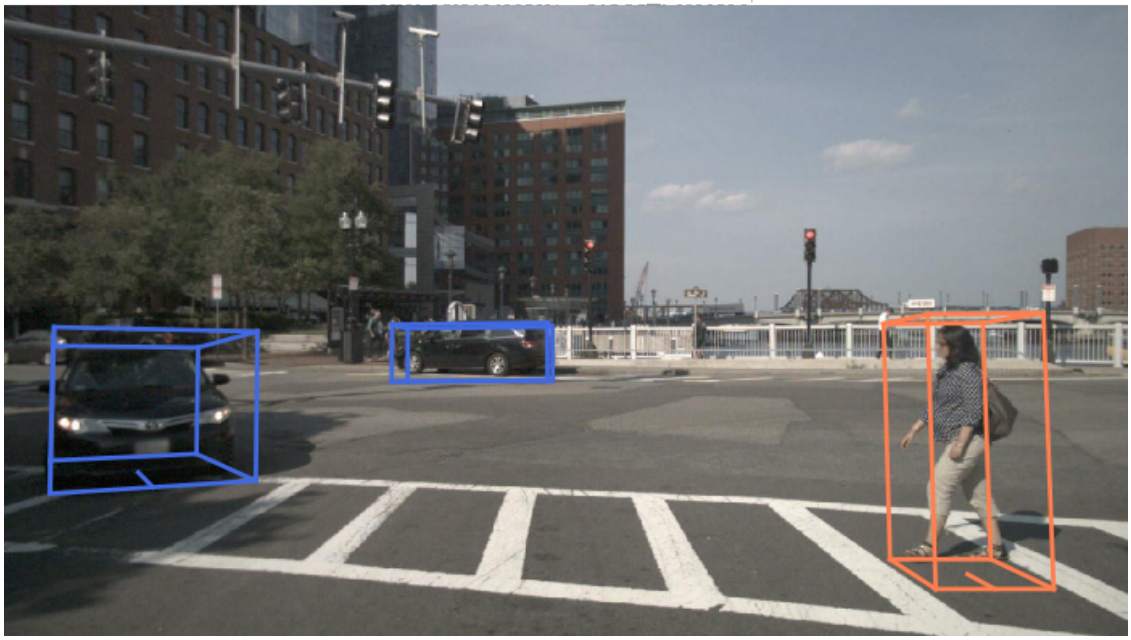
CHALMERS
UNIVERSITY OF TECHNOLOGY



3D Object Detection for Autonomous Driving using Deep Learning

Master's thesis in Computer Science - Algorithms, Languages and Logic

Olof Berg Marklund Oskar Hulthén

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2019

Master's thesis 2019

3D Object Detection for Autonomous Driving using Deep Learning

Olof Berg Marklund Oskar Hulthén

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Division of Signals Processing and Biomedical Engineering
Signal Processing Group
Chalmers University of Technology
Gothenburg, Sweden 2019

3D Object Detection for Autonomous Driving using Deep Learning
OLOF BERG MARKLUND, OSKAR HULTHÈN

Supervisors: Karl Granström, Department of Electrical Engineering
Karl-Magnus Dahlén, Denso
Examiner: Karl Granström, Department of Electrical Engineering

Master's Thesis 2019
Department of Electrical Engineering
Division of Signals Processing and Biomedical Engineering
Signal Processing Group
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: 3D bounding box predictions on a LiDAR point cloud sample from the nuScenes dataset made by a network implemented in this thesis, visualized through projection into a camera image from the same sample.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2019

3D Object Detection for Autonomous Driving Using Deep Learning
OLOF BERG MARKLUND, OSKAR HULTHÈN
Department of Electrial Engineering
Chalmers University of Technology

Abstract

For an autonomous vehicle to truly succeed, there must be a way for the vehicle to interpret its surrounding environment, both quickly and accurately. This master thesis focuses on investigating improvements for a sub-problem of environmental perception, namely 3D object detection, referring to localizing and classifying objects of interest in a specified environment. The objects of interest in a traffic situation are pedestrians and other vehicles.

Recently deep learning approaches have made significant progress in the field of object detection, especially for two-dimensional data such as camera images. Thus, several novel detection models have been developed for real-time tasks such as autonomous driving. However, as images generally lack information about depth, which is essential for environmental perception for autonomous vehicles, these camera-based approaches struggle to perceive distances reliably. An alternative sensor to the camera is LiDAR; it captures the surrounding area with reflected light, in the form of 3D data points known as point clouds. The latest research regarding 3D object detection in point clouds suggests both fast and accurate predictions by encoding the 3D point cloud representation in such a way that a 2D object detection model can be applied.

This thesis investigates the performance of both the point cloud encoders and the 2D detection models used by the state-of-the-art 3D object detection models. The thesis attempts to recreate encoders and detection models to make comparisons of alternative implementations used to produce 3D detections in traffic situations. The end-to-end deep learning models are ultimately trained to perform 3D detections. The models were trained and evaluated on the recently released nuScenes dataset, and the most promising model was able to accurately detect cars in real-time.

Keywords: Deep Learning, Machine Learning, Neural Networks, Autonomous Driving, Convolutional Neural Networks, Object Detection

Acknowledgements

First of all we would like to thank our supervisor, Karl-Magnus Dahlén for giving us the opportunity of doing the master thesis at DENSO, along with Alexander and Christopher for showing interest and support during the project.

We would also like to thank our supervisor Karl Granström for keeping the thesis in an academically interesting perspective and for all the knowledge given in the field.

Lastly we would like to thank our friends and family for the support they have provided during the last five years.

Olof Berg Marklund and Oskar Hulthén, Gothenburg, June 2019

Contents

List of Figures

List of Tables

Abbreviations

AD Autonomous Driving
ANN Artifical Neural Network
AOE Average Orientation Error
AP Average Precision
ASE Average Scale Error
ATE Average Translation Error
BEV Bird's-Eye View
BN Batch Normalization
CEL Cross Entropy Loss
CNN Convolutional Neural Network
Faster R-CNN Faster Region-based Convolutional Neural Network
FoV Field of View
HOG Histogram of Oriented Gradient
IAP Interpolated Average Precision
IoU Intersection over Union
LiDAR Light Detection and Ranging
mAP Mean Average Precision
MSE Mean Squared Error
NDS nuScenes Detection Score
NMS Non-Maximum Suppression
RADAR Radio Detection and Ranging
RoI Regions of Interest
RPN Region Proposal Network
SIFT Scale Invariant Feature Transform
SOTA State-Of-The-Art
SSD Singe Shot Multibox Detector
TPM True Positive Metric
YOLO You Only Look Once

1
Introduction

Autonomous Driving (AD) represents one of the most notable challenges present in modern computer science. The possibilities that AD may provide in peoples' everyday life are vast. Partly in terms of saving lives as most driving accidents have been linked to the human error [4], but also in terms of lifestyle as time spent commuting could, at least in theory, be spent more productively.

One key technology needed for succeeding in the field of AD is environmental perception [30], specifying a way for vehicles to interpret their surroundings. It is incredibly crucial for safe and reliable AD that this technology is both accurate and fast. Firstly, accuracy is vital as the interpreted environment should reflect the surroundings as correctly as possible. A system with accurate environmental perception can better perform tasks such as route planning, avoiding objects, and following the road. Secondly, the speed at which the perception is applied is also vital as AD should be utilized in real-time. If a system is not fast enough the driving situation may have changed from the last environment interpretation, making the processed information irrelevant.

A subproblem to environmental perception is object detection. The proposed master thesis focuses on investigating improvements for object detection in urban driving situations. Object detection in simple terms refers to localizing and classifying specific objects in a given environment. In a driving situation, the objects that should be localized are typically pedestrians and other vehicles. In essence, the objective is to find unpredictable objects that are typically subject to movement from one time step to another.

Due to driving situations being both complex and varying, there exists a problem of generally and reliably recognize pedestrian and other objects, with their different appearance. One of the most promising methods for solving this problem is deep learning; the primary method that is explored further within this thesis.

1.1 Background

There has been significant progress in deep learning systems using neural networks to produce accurate 2D object detection [5] [27] [38]. Typically the inputs to these systems are single images, but when it comes to AD interest lies in 3D space as the estimation of the distance to objects is of utmost importance. The field of 3D

object detection has progressed as well [23], [45], [48], although there is a notable performance gap when comparing to that of data with fewer dimensions, implying that it is an open research field.

One of the most significant issues with training deep learning models is the need for large amounts of annotated data. Annotated data is a term for data that is labeled, i.e. data containing the supposed ground-truth output for each specific input. Annotation in the case of 3D object detection means that the data is labeled with the location, class, and size of the specific objects. Collecting and annotating data is a time-consuming task. However, a public large-scale dataset called nuScenes [7] containing annotated urban traffic scenes has recently been released and is used as a source for training and evaluating deep learning models in this project. nuScenes is a multi-modal dataset containing point clouds and images gathered by the different sensors, such as; Light Detection and Ranging (LiDAR), Radio Detection and Ranging (RADAR), and cameras respectively.

The project is performed at DENSO, a leading supplier of automotive technology solutions. In previous work, the active safety department located in Gothenburg has used filter techniques to track objects appearing in traffic whereas they are now looking at a way to possibly improve their object detection system by introducing deep learning to the task.

1.2 Problem Formulation

The problem of object detection for AD in urban environments has been a popular research area in the last decade. Driving in an urban environment typically leads to unpredictable scenarios, where one has to keep track of both other vehicles and pedestrians moving around in different directions. The unpredictability and complexity of these types of scenarios make it a particularly interesting and challenging environment to research.

The specific techniques implemented in earlier research [27], [45], [8] are highly dependent on the type of input data, or more explicitly, what type of sensors were used to gather the data. In this thesis, the focus lies upon data gathered by a 3D LiDAR, one of the most common sensory inputs for AD, that provides data in the form of three-dimensional point clouds.

1.3 Aim

The thesis aims to compare the accuracy and time efficiency of State-Of-The-Art (SOTA) network architectures for 3D object detection models, to ultimately produce and train 3D object detection models with maximal performance when detecting objects in driving situations provided by the nuScenes dataset.

The thesis further investigates and aim to answer the following questions:

- What performance can be achieved regarding object detection on nuScenes using a LiDAR-only model whilst maintaining a real-time constraint?
- How do different point cloud encoders compare to one another?
- How do different real-time deep learning object detection models compare to one another?

A real-time system is henceforth defined as a system with 100 milliseconds in inference time, referring to the time it takes from receiving an input to producing an output. In other words, a real-time system runs at a rate above 10Hz.

1.4 Limitations

As the tasks described in this thesis is centered around object detection, other subtasks in environment perception, such as road detection or route planning, are not considered.

This thesis focuses on performance in the trade-off between computational speed and performance rather than achieving a commercially viable solution for AD. A problem with the definition of absolute efficiency of a network is that it heavily relies on the hardware it is running on. The hardware used by researchers is typically high-end, meaning that the resulting absolute efficiency that researchers claim their models achieve is tied to the high-end hardware. Furthermore, as the high-end hardware is not expected to end up in commercially viable autonomous cars, the question regarding what kind of performance one can expect in the actual vehicles remains. To specify, this thesis is performed in the purpose of research, meaning that commerciality is not taken into account.

1.5 Ethical Aspects

There is no denying that there are areas present in today's means of transportation that can be improved. One of the main negatives with today's transportation system is the frequency of traffic accidents occurring around the world. According to the world health organization, approximately 1.2 million people die in traffic accidents annually [2], and the U.S Department of Transportation stated that around 93% of traffic accidents are due to human error [4]. The human error could be eliminated through the implementation of fully AD.

Furthermore, AD is a potential time-saver. The U.S population spends around 120 million hours commuting to work each day [1], to put that into perspective, an

average lifespan is approximately 650 thousand hours. In other words, the U.S population spends around 185 lifetimes each day commuting to work. People in the U.S also spend, on average approximately 41 hours each year in traffic congestions [3], where once again the most significant cause is due to human error. By utilizing AD, all this time could be spent towards doing something productive instead.

With the environmental aspect in mind, AD could lead to increased travel due to better accessibility. Increased travel leads to more emissions, and one could argue that humankind would reduce their overall footprint by not promoting any transportation at all. However, comparing a human-driven vehicle with an autonomous one, the latter would drive more efficiently with the help of route planning and calculated throttling, leading to less overall emission.

1.6 Thesis Outline

The outline of the rest of the thesis is structured as follows - Chapter 2 details relevant theory explaining the underlying techniques and concepts for the project. The chapter discusses sensory input, Artifical Neural Networks (ANNs), Convolutional Neural Networks (CNNs), and ends with an explanation of the evaluation of object detection models. The theory chapter is followed by descriptions of novel solutions of 2D and 3D object detection tasks in the related work Chapter 3. Chapter 4 presents the method of how the project was executed; it starts by presenting the dataset and data augmentation in the pre-processing stage, followed by the implementation of the encoders and decoders. The chapter ends with presenting the evaluation method used. The next two chapters in the thesis are Results 5 and Discussion 6, they present and discuss the training and evaluation results for the object detection models. The thesis ends with a Conclusion 7 of what has been accomplished, future work, and takeaways.

2
Theory

This chapter presents the theoretical aspects that are the foundation of this thesis. The chapter begins by describing the advantages of sensors for AD in Section 2.1 and a brief description of object detection in Section 2.2. This is followed by an explanation of the basic concepts of ANN in Section 2.3 and CNN in Section 2.4. The chapter ends with explaining the fundamentals of the evaluation methods for object detection in Section 2.6.

2.1 Sensor Input

For a machine to be able to interpret the real world, there must be some information to process. Gathering information from the real world is generally done with sensors, and the type of sensor specifies the attributes of the data produced. This section contains a brief explanation of the two sensors relevant to the project.

2.1.1 Camera

The camera is an intuitive and cheap option to use when gathering information from the real world. Images are created by projecting the real-world scene onto a 2D plane by capturing the light intensity and frequency detected at each projected location. The projected locations values are saved as RGB pixel values. This essentially means that information about how far away objects are is lost, as there is no depth in regular 2D images. Due to the distance to objects being one of the fundamental questions in environmental perception for AD, typically 3D object detection models do not solely use cameras as sensory input.

Moreover, cameras as input sensors are sensitive to light variations and weather conditions. As information is being stored in the form of the intensity of different colors at each pixel in the images, two images of the same scene in different weather can vary immensely.

2.1.2 LiDAR

LiDAR is a sensor commonly used in autonomous vehicle applications. It gathers data by emitting laser light pulses and detecting the reflected pulses, creating a 3D image called a point cloud, where each data point corresponds to the location in which the emitted light was reflected. Each located data point is saved as the

position in space (typically as cartesian coordinates x, y, z) along with the reflective intensity (r). The position is calculated based on the direction of the emitted light and the time it took until its reflection was received by the sensor, specifically half the time between emission and detection divided by the speed of light. The reflective intensity is calculated based on the intensity of the light pulse being received, in comparison to the one sent.

These types of sensors are also sensitive to some weather conditions for example, rain or snow that might interfere with the emitted light pulses. However, they are not sensitive towards light variations, as the information is stored as coordinates rather than colors emitted from the wavelength of the light.

2.2 Object Detection

Humans are analyzing the world by continuously labeling, predicting, and recognizing patterns of what they can see. Essentially making sense of the surrounding environment, in the case of the human vision system, this is done subconsciously and without much effort. Computer vision refers to the field of trying to automate the tasks that the human visual system can perform, in a machine.

Object detection is a task closely related to computer vision. It refers to both classifying and localizing objects of interest in an environment. The localization of objects in the context refers to producing bounding boxes that try to enclose each of the object of interest. Ideally, a bounding box should be as small as possible whilst containing the entire object. Additionally, each produced bounding box contains a label referring to the classification of the objects. The object detection problem is partly a regression problem in terms of finding the placement and size of the bounding box, and partly a classification problem in terms of labeling each of the produced bounding boxes.

Deep learning approaches are well suited for the task of object detection [38], [27], [39], where a deep neural network learns to extract features from the input data and based on the features, predict bounding boxes labeled with the predicted class. Deep learning object detection methods can generally be split into five parts; feature extraction, regional proposals, classification, regression, and pruning. All these are described in this section, while the basics of how a neural network works are presented in Section 2.3.

2.2.1 Feature Extraction

One of the most common data types for object detection applications are images. Getting a computer to understand the contents of an image, which to a computer is just a matrix of values, is a challenging task [12]. Especially as the image data vary depending on changing variables like lighting conditions. The solution is to learn general features from the actual data, to try to find some representation that

separates the object of interest from the background and from other objects.

Features within an image are typically represented through patches of pixels. Classical methods such as Histogram of Oriented Gradient (HOG) [9] and Scale Invariant Feature Transform (SIFT) [29], [28] create, by looking at the colors of pixels within patches, features from the direction of most change, also known as the oriented gradient.

In the case of deep learning, the features are typically learned by using CNNs. These types of networks can assign importance in the form of weights to specific properties in patches, creating more abstract feature representations. The networks are trained by being shown examples of inputs and their supposed outputs. Finding which properties in the input are essential to make the desired decisions. Given enough examples, ideally, the network would be able to generally differentiate one class from another through the help of the learned features. The specifics regarding CNNs will be described in more detail in Section 2.4.

2.2.2 Region Proposals

One issue with object detection is that the number of objects in an environment can vary, meaning that the target output can vary in size. There are several ways of solving this problem. One of the more classical methods is called sliding window, which consists of, as the name suggests, sliding a window across the input image and extracting smaller sections, where each section is handled as a region proposal. The technique handles objects of different sizes and scales, by scaling the input image and then rerunning the sliding window. The generated region proposals can then be interpreted as unclassified bounding boxes, essentially bounding boxes for the model to further explore whether they contain objects or not. More modern methods for handling region proposals are explained in Section 3.1.

2.2.3 Classification

Each of these aforementioned region proposals (bounding boxes) are subject to classification, to find whether the bounding box contains an object or not. The classification is done either through a separate classification system or as an integrated part of the model generating the region proposals. The classification generally yields a probability for each class to be present in the proposed bounding boxes. Typically, an additional class referring to background is added to discard bounding boxes without objects in them.

2.2.4 Regression

As the generated region proposals are not expected to enclose the objects as closely as possible, additional regression is typically applied. The regression is similarly

as the classification either done separately or as a more integrated part of the region proposals. The goal for the regression is to ultimately tighten (or loosen) the proposed bounding boxes, by offset values, to better contain the objects.

2.2.5 Pruning

Each of the produced bounding boxes from the model is typically subject to pruning. The most widely used method to do this is called Non-Maximum Suppression (NMS). NMS consists of first removing all bounding boxes where the classification probability output is below a set threshold, removing the predictions that the model do not believe are objects. Generated bounding boxes that refer to the same object are also removed. This is done by calculating the overlap between boxes, and if the overlap is above a set threshold, then only the bounding box with the highest classification certainty is kept.

Intersection over Union

A common metric used in object detection is Intersection over Union (IoU). It is used to calculate the overlap between two shapes, and therefore typically applied when matching region proposals boxes to ground truths, as overlap is something to strive for in predictions. It does so by dividing the overlapping area between the boxes with the total area of the boxes combined. The function can be described as in Equation 2.1 and graphically in Figure 2.1.

$$\text{IoU}(b_1, b_2) = \frac{b_1 \cap b_2}{b_1 \cup b_2}. \tag{2.1}$$

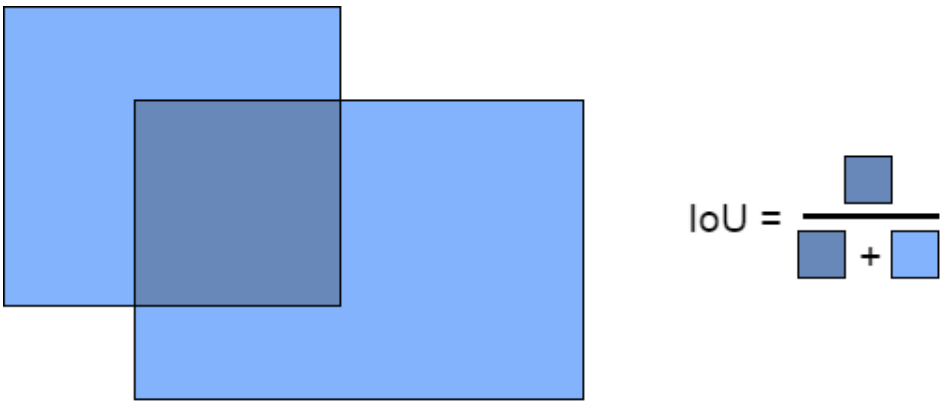


Figure 2.1: Graphical explanation of intersection over union between two rectangles in 2D. The dark area shows the intersection and the sum of the bright and the dark area yields the union of the two rectangles.

One additional note is that the boxes do not need to be axis aligned. However, the actual calculation of the IoU becomes much more complicated if the alignment differs between the boxes. The reasoning why it becomes tricky is that the corresponding shapes of both the intersection and union typically become irregular shapes with complex areas.

Hard Negative Mining

Object detection is not only the task of finding objects of the specified classes, but also the task of finding background. Generally, the amount of background in an input heavily out-weighs the amount of objects of interest, meaning that there are many background classifications made. Each bounding box that is classified as background is called a negative prediction. Training a neural network is done by feeding information about supposed outputs. So, to train an object detection model properly one does not only need the positive predictions (actual objects) but also examples of background.

Classifying whether a bounding box contains background or not can be surprisingly hard, especially as every object class not included in the task is specified as background. If a model for instance is not looking for trams, all trams are considered to be background. Hard negative mining is a common technique to put emphasis on hard negative predictions. This is done during training when the model yields some positive prediction on supposed background. Then that specific hard negative example is fed back into the training process, thus yielding a network that is better at correctly classifying background, and therefore better at correctly distinguishing between objects and supposed background.

2.3 Artificial Neural Networks

Using neural networks for object detection has yielded exceptional results [12]. This section describes the fundamentals of the theory behind how techniques for neural networks have improved its viability in a variety of tasks. The section includes descriptions of different activation functions, optimizers and how to avoid common issues in the training phase of a network [32].

ANNs are frameworks of machine learning techniques that loosely mimic the brain; the fundamental idea is inspired by the Hebbian theory, which attempts to explain the adaptation of a brains neurons on learning [16]. The main building block of an ANN is the artificial neuron, the first computational model of the artificial neuron was introduced by Walter Pitts and Warren McCulloch in 1943 [31].

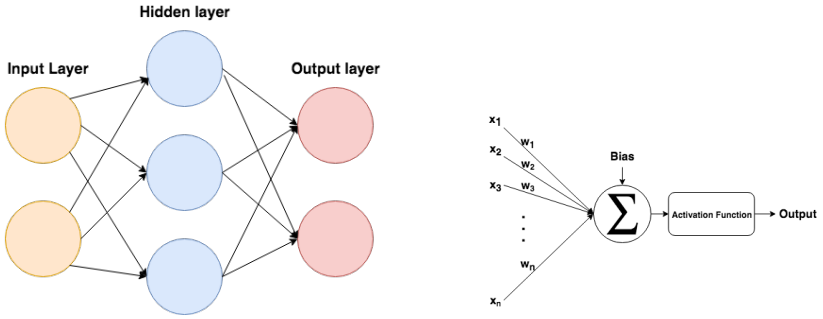
The goal of an ANN is to perform analytic tasks learned by receiving feedback on its performance during the so-called training phase. The feedback given is highly dependent on the learning technique. This project falls under the category of supervised learning, meaning the network gets feedback from the training data containing the desired output, also known as the ground truth. An ANN consists of a connection of computational nodes called neurons, which are usually divided into multiple layers as seen in Figure 2.2 (a). These layers can be one of three types:

- *Input layer:* The samples fed to the network are first processed by the input layer, which should have an appropriate form to represent the sample data.

This layer does not perform any computations. Its only task is to send the data to the next layer.

- *Hidden layer:* The output from the input layer is then sequentially passed to the hidden layers (layers of intermediate neurons in-between the input and output layers). These layers are necessary for more complex non-linear tasks. Here each neuron represents some abstract feature of the input helping to get the correct output. (A neural network with at least one hidden layer can be called a deep neural network (DNN). Hence the name deep learning.)

- *Output layer:* The last layer is called the output layer, the output from this layer should have the form that represents the necessary data for the specific task.



(a) Layers in a NN **(b)** Neuron in a NN

Figure 2.2: (a) A representation of a neural network divided into three layers. (b) The mathematical representation of one neuron in an ANN

An artificial neuron calculates the weighted sum of its input, adds a bias and then with the help of an *activation function* decides which information should be forwarded. If the neuron decides to pass information forward, the neuron is considered *activated*. The output of a neuron is calculated by the following mathematical formulae presented as

$$O_i = g(\sum_{j=1}^{N} w_{ij}x_j - b_i), \tag{2.2}$$

where, w is the learnable weights, x is the input, b is the bias. The function $g()$ is the activation function and N is the number of input values. All N inputs have a weight w_{ij} attached to them, basically saying how much the current node should listen to that specific input. The bias is another learnable parameter, used to better represent the wanted function by enabling shifting. The activation function controls the neurons output, an explanation of different activation functions and their role can be found in Section 2.3.3. The process of calculating the output from an input is called *forward propagation*. The actual learning is the update of the learnable parameters, weights and bias, and is called *backpropagation*, an explanation of this technique is presented in the following section.

2.3.1 Learning

As mentioned previously, the weights and the bias are the learnable parameters of an artificial neuron. These parameters are commonly initialized with random values and are iteratively updated with backpropagation in the training phase [17]. The aim is to find the value of trainable parameters that achieve the lowest loss calculated by the so called *loss function*. The loss function is a metric of how far off the neural networks output was from the desired output. An example of a simple loss function is the Mean Squared Error (MSE) presented as

$$\text{MSE}(O, \hat{O}) = \frac{1}{N}\sum_{i=1}^{N}(O - \hat{O})^2, \tag{2.3}$$

where O represent the output from the network, $\hat{O}$ the desired output, and N the number of outputs.

Finding the weights and biases for all the neurons that achieves the lowest loss becomes an optimization problem, where the objective is to find the global minimum of the loss function. The update of the learnable parameter is done with an optimizer based on *gradient descent* where the parameters are updated according to

$$\theta = \theta - \eta \nabla J(\theta), \tag{2.4}$$

where η is the learning rate, θ is the parameter to be updated, and $\nabla J(\theta)$ is the gradient of the loss function $J(\theta)$. In backpropagation, the gradient is calculated by taking the partial derivative of the loss function with respect to the parameter to be updated. The partial derivative for a specific weight depends on all the weights in the following layers, therefore the backpropagation is performed from the last layer and going backwards (hence the name). The specific dependencies are calculated for each layers weights in the network by using the chain rule. The concept derives from the fact that the neurons that affect the output the most should be updated more.

The final product of the network produces an output based on a generalization of its experience, i.e., the data that the network trained on. A common problem when training a network is that the model learns the details of the training data too well. This leads to the network performing really well on the training set but gets significantly worse results on new data, this problem is known as *overfitting*. In the training stage, the data available is commonly split into two sets: A training set consisting of the majority of the data where the network updates its learning parameters based on the loss, and a validation set used as metric for testing if the network is generalizing.

2.3.2 Optimizers

There are different optimizers used to update the trainable parameters, each with their advantages and disadvantages. The standard gradient descent explained in Section 2.3.1, performs one update after the network has processed the entire dataset.

This means that the update is based on the average of the entire dataset, and it therefore common to converge to a sub-optimal local minimum. There are variants of optimizers based on the gradient descent that solves the aforementioned problem:

- *Stochastic Gradient Descsent*: Performs one update for each training sample. With different input samples, this frequent update leads to a high variance for the trainable parameters, hence the loss function. The high variance could help discover a better local minimum than what the standard gradient descent would. However, the oscillation could also complicate the convergence and make the network overestimate its update value [6].

- *Mini Batch Gradient Descent*: Updates the parameters after a set number of samples (referred to as a batch) has been processed, the batch should be smaller than the full training set. The idea is to process a batch of training samples and then update the network by taking the average loss of the batch. It reduces the variance which can lead to a stable convergence while still finding the global minimum [25].

Still there are some challenges present using gradient descent and its variations, like choosing the correct learning rate and avoiding sub-optimal local minimas. Here follows some techniques that further optimize gradient descent, diminishing the challenges mentioned above.

- *Momentum*, a technique made to soften the oscillation of the updates by adding a fraction of the update vector from the previous step update $V(t-1)$ to the current update $V(t)$. The mathematical formula is presented as

$$\begin{aligned} V(t) &= \gamma V(t-1) + \eta \nabla J(\theta), \\ \theta &= \theta - V(t), \end{aligned} \tag{2.5}$$

  where θ is the learnable parameter, η is the learning rate and γ is the momentum rate, defining the fraction of the momentum from the previous step that should be added in the update [43].

- *Adaptive learning rate*, is a technique which updates the learning rate η differently for each learnable parameter. Here follows three of the most common optimizers using adaptive learning rate.

  - *Adagrad*, updates the learning rate for a specific parameter based on the frequency in which that specific parameter is updated. In essence this means that parameters that are rarely updated keeps a higher learning rate in comparison to those that are frequently updated. By using this type of updating implies that the need for manual tuning of the learning rates diminish. Furthermore an disadvantage of using Adagrad is that it calculates the learning rate by accumulating all calculated gradients in the past, forcing the learning rate to always decrease, which can lead to that the model cease to learn [10].

– *Adadelta*, is an extension of Adagrad which tends to solve the problem of decaying learning rate. It does so by limiting the number of previous gradients that defines the adaptive learning rate to a fixed number. The learning rate is updated similarly to how the parameters are updated in momentum: a fraction of the previous mean of gradients is added [46].

– *Adaptive Moment Estimation* (Adam), is a method that also computes adaptive learning rates for each parameter similarly as Adadelta. However, it also stores an individual momentum for every parameter. This means that Adam calculates both the individual momentum and the learning rate for each learnable parameter, and thus avoids most challenges presented in this section [21].

2.3.3 Activation Functions

The role of the activation function denoted as $g()$ in the Equation for the neuron (2.2) is to limit the output from a neuron and to decide if the neuron should be activated or not, i.e., if the other neurons should listen to this neuron's output. Furthermore the activation functions also make it possible to introduce non-linearities into the neural networks, making it possible to solve complex non-linear problems. There are several different types of activation functions with different properties. All are appropriate depending on what task the network is designed to solve. This subsection describes some pros and cons of the most common activation functions.

Sigmoid

The Sigmoid function is a widely used activation function for classification problems [32]. It limits the neurons output to a range between zero and one. It is defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{2.6}$$

During backpropagation, the weights are updated depending on the gradient in the previous layer starting from the last layer going backward. As shown in Figure 2.3, the gradients are not that steep, towards each edge of the sigmoid function. Thus, nodes in the last layers having small gradients could lead to minimal updates for the nodes in the first layers. It makes the learning process for the network prolonged or to stop completely. This problem is called *vanishing gradient* [19] and was historically one of the biggest problems to get deeper networks to converge in a reasonable time.

Hyperbolic Tangent (tanh)

The Hyperbolic Tangent activation function is commonly referred to as tanh, and it is a scaled version of the sigmoid function, thus leading to a steeper gradient. The

Hyperbolic Tangent is also shown in Figure 2.3. It ranges from -1 to 1, allowing the values to be zero-centered and is defined as

$$\tanh(x) = \frac{2}{1 + e^{-2x}} - 1, \tag{2.7}$$

however the activation function still has the problem of vanishing gradient.

Rectified Linear Unit (ReLU)

ReLU is a widely used activation function for deeper networks [24]. It outputs zero for negative number and outputs the input for positive numbers, as shown in Figure 2.3. ReLU is defined as

$$f(x) = \max(0, x). \tag{2.8}$$

Thus, ReLU is a non-linear activation function. Thanks to the gradient either being zero or one diminishes the vanishing gradient problem [24]. Another aspect is that the neurons with zero output can be ignored, thus lowering the number of computations needed, making the network more efficient. However, if the neuron has negative weight and bias, meaning that the activation function would be at a zero gradient, the update from the backpropagation would also be zero. This phenomenon means that the neuron would need to be adjusted by an external factor to start outputting anything other than zero again, a phenomenon known as the dying ReLU problem.

Another issue is that ReLU's output is not limited, which could lead to activations blowing up (becoming disproportionally big)[32]. On a positive note, the activation function itself has a simpler mathematical formula than the previously mentioned activation functions, further increasing the computation speed.

Leaky Rectified Linear Unit (LReLU)

A solution to the dying ReLU problem is the LReLU where the gradient for negative numbers is changed to have a small gradient. The function is given by

$$f(x) = \max(0, x) - \alpha \max(0, -x), \tag{2.9}$$

with α set to a small value close to zero. The LReLU is illustrated in Figure 2.3.

Softmax

A common activation in the output layer for classification problems with multiple classes is softmax. It turns logits (the raw output scores from the last layer) into probabilities that sum to one. It has the form

$$S(y_i) = \frac{e^{y_i}}{\sum_j^c e^{y_j}} \tag{2.10}$$

where y_i is the output probability for class i in the output layer and c is the number of classes.

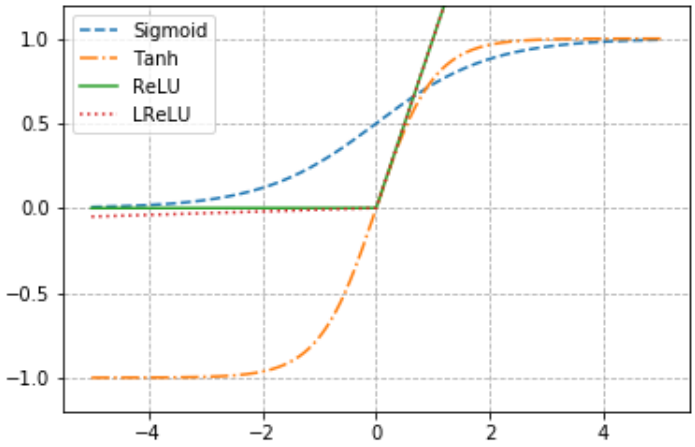


Figure 2.3: Comparison of some of the most common activation functions used in an ANN.

2.3.4 Regularization

One way to achieve a generalized output and avoid overfitting is with a concept called regularization. Regularization makes slight modifications to the network while learning, to force the network to generalize better. One regularization technique is called *dropout*; it prevents each neuron from updating its parameter and producing an output with a probability P. Another common regularization technique is L_2 regularization, which adds a term to the loss function to force the weights to decay towards zero. The technique is presented as

$$L_{new} = L_{prev} + \frac{\lambda}{2m} \sum ||w||^2, \tag{2.11}$$

where the L_{new} is the total loss, L_{prev} is an arbitrary loss function, and the term to the right of the plus sign is the L_2 regulator. λ is the regularization parameter, w is the learnable weights and m is the number of outputs. Thus, by differentiating the loss function with regards to each weight, the gradient will be affected by the added regularization.

An additional problem that can occur when training a neural network is that input values can be disproportionate to each other, this leads to an unnecessary amount of time before the network stabilizes. To counteract this, one can use normalization to force features to values between zero and one. This is typically done in the input layer when, for instance, one has different features of different scales. However, normalizing inputs between layers, thus stabilizing the network has become more common. This technique is called Batch Normalization (BN) and acts as a regulator by adding two new trainable parameters to the layers of choice: A batch mean and a batch standard deviation. These parameters subtract and divide every input to the layer respectively, normalizing the input, allowing the layer to avoid disproportionate input [20].

2.4 Convolutional Neural Networks

A CNN is a specific type of feed-forward neural network, where the architecture is designed to take advantage of the structure of an input (typically images). The networks are generally built up of mainly two building blocks, convolutional layers and pooling layers, which are explained in more detail below. These building blocks are used to represent the data in terms of features, by weighting and modifying identifyiable characteristics of smaller patches within the inputs of each layer. The overall idea behind using these building blocks is that the representation of the input is gradually increased in abstraction as it progresses through the layers. Earlier layers contain simpler structural information such as lines, whilst later layers contain more complex information about how specific objects actually look.

2.4.1 Convolutional Layer

The convolutional layers are as any ANN composed of neurons with learnable weights and biases. Each neuron in a convolutional layer receives some inputs and calculates their output based on the learned weights and biases. However, the big difference lies in the sharing of weights between neurons, where the weights can be visualized as matrices called filters. These filters are applied on the input by the so-called convolutional operation. The convolutional operation is executed by sliding the filter over the input in both directions, rows and columns. At every location, an element-wise multiplication is performed and summed together yielding the result to be placed in the output feature map.

There are four parameters of significance in each convolutional layer:

- Firstly, the filter size, specifying how many weights are located within each filter. Typically filters in the range of 3×3 to 5×5 is the norm as they can handle smaller and local features of objects. Additionally, the depth of each filter is matched with the number of channels present in the input to the layer.

- Secondly, the number of filters, specifying how many filters should be applied to the data within each layer. The number of filters decides the number of channels that are present in the final output feature map from the layer. A simple example of applying five filters on an image can be seen in Figure 2.4.

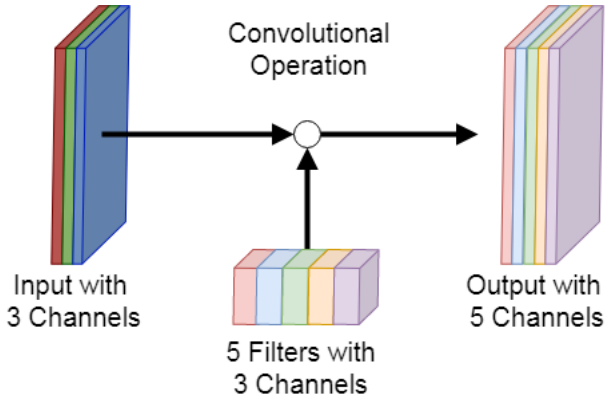


Figure 2.4: Example of the output depth when applying multiple filters. In this case five filters are applied to a three channeled image, meaning that the depth of the resulting feature map is five.

- Thirdly, the stride specifies how much the filter shifts in each step when sliding through the input data. The example present in Figure 2.5 and (2.12) uses a stride of one. If the stride is increased, the output feature map is significantly reduced in size.

- Lastly padding, specifying how much the input should be padded in its borders. Zero-padding is the most commonly used implying that additional columns and rows are added to enclose the input map with zeros effectively increasing the size of the input map. It is generally done to increase the performance, as it enables better extraction of information from the original borders of the input. Additionally, it is applied to preserve the input dimensions through the output feature map, since the convolutional operation generally decreases the output size.

Visualized in Figure 2.5 is the actual convolutional operation with a filter size of 3×3 applied on an input of size 6×6. For simplicity, the operation is applied with a single filter (applied with a stride of one) without depth. However, convolutions are generally performed with multiple filters where the input and filters both have depth, such as in the input layer when using RGB colored images.

The same convolutional operation can be mathematically expressed as

$$O_{i,j} = \sum_{x}^{3} \sum_{y}^{3} F_{x,y} \; I_{i-1+x,j-1+y}, \tag{2.12}$$

where O_{ij} denotes the output for position ij, F stands for the filter of size 3×3 and I marks the input image.

As the weights are in the form of filters, this means that the actual filters are updated in convolutional layers through the backpropagation. Furthermore, as the filters are slid across the entire input data, features are handled the same no matter where in

the input data it is located, meaning that the method is translation invariant.

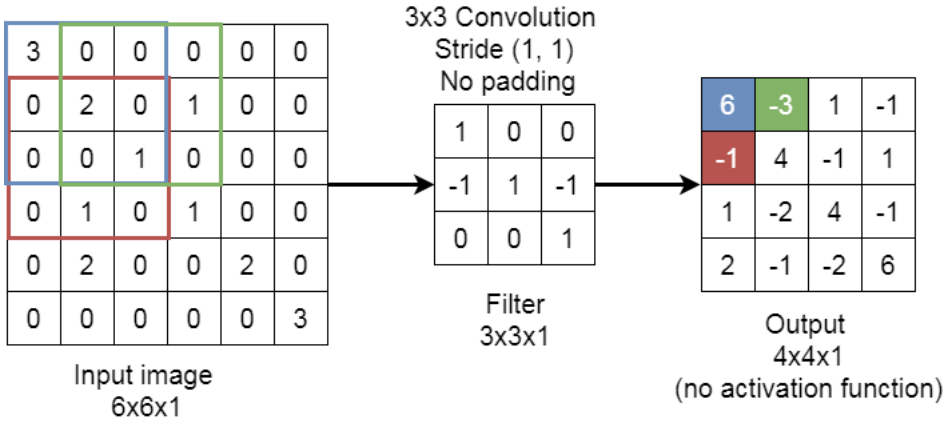


Figure 2.5: An example of a convolutional operation with a single filter of size 3×3 and an input image of size 6×6, applied with a stride of 1 without padding. Yielding an output of size 4×4 by doing an element-wise multiplication between the input image and the filter to be summed for every location in the output feature map.

The dimensions of the output feature map for each layer can be expressed as

$$O = \frac{N - F + 2P}{S} + 1, \tag{2.13}$$

where O is the output dimension for either rows or columns, N is the size of the input map, F is the filter size, P is the number of padding rows/columns, and S is the stride. If the resulting value is a fraction, the value is rounded towards zero to the closest integer.

2.4.2 Pooling Layer

Pooling is used to downsample the size of the output feature map. This is done to reduce the amount of parameters and computations in the network, and ultimately reduce the amount of overfitting done in the entire network (as only the most promising features are kept). Max pooling is the most frequent type of pooling, it splits the input feature map into equally sized regions and only the maximum value present in each region is kept as output as shown in Figure 2.6. The size of the regions and the stride in which the pooling is applied affects the output size as expressed in (2.13) without padding.

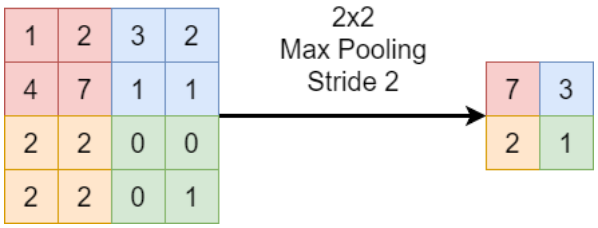


Figure 2.6: Example of the application of maxpooling with a size of 2×2 and a stride of two on an input feature map of size 4×4. The highest value in every 2×2 section in the input feature map is yielded at the corresponding position in the output.

2.4.3 Transposed Convolutional Layer

Similarly to how convolutional layers reduce the size of the output feature map, transposed convolutional layers can be used to do the opposite [11]. By applying the convolutional operation in reverse, one can conduct upsampling of the feature maps. Moreover, the weights are learnable and can similarly to convolutional layers be visualized as filters. At every input location, the filter is multiplied by the input value and placed in the corresponding location in the output map where overlaying values are summed together. The operation for a 3×3 transposed convolution on an example input of 4×4 can be found in Figure 2.7.

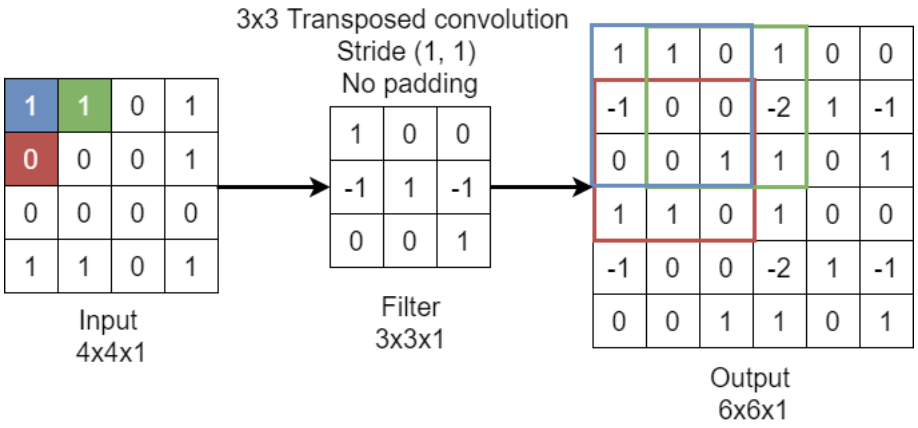


Figure 2.7: An example of a transposed convolutional operation with a single filter of size 3×3 and an input image of size 4×4, applied with a stride of one and without padding. At each input location, the value is multiplied by the entire filter and placed in the same location in the output map. Overlaying values in the output map are summed together, yielding a 6×6 sized output feature map.

2.4.4 Unpooling

As the transposed convolutional layers aim to do a reversion of convolutional layers, unpooling layers aim to do the same reversion for pooling layers. Specifically, for max pooling there is max unpooling to reverse the operation. It creates the reversion by recording the locations of maximum activations during the max pooling [34], in

which positions the maximal values were found during downsampling. Thereafter the unpooling reverses the maximum operation and fills the remaining entries with zeroes. An example of max unpooling the same resulting feature map from Figure 2.6 can be found in Figure 2.8.

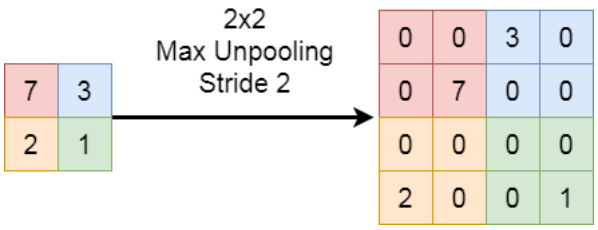


Figure 2.8: Example of the application of max pooling with a size of 2×2 and a stride of two on an input feature map of size 4×4. The highest value in every 2×2 section in the input feature map is yielded as the corresponding position in the output.

2.4.5 Fully Connected Layers

After features have been learned from convolutional and pooling layers, the high-level reasoning from the features can be done through fully connected layers. The use of fully connected layers heavily relies on the type of output aimed for, as it allows the movement from a grid representation to single values. Moving the data representation from a convolutional network into a feedforward network structure. This is typically useful when performing classification or regression based on the input as a whole.

The data translation happens through a flatten layer, that moves each individual feature map matrix value into a vector where each position in the vector is interpreted as an input value to the following fully connected layers. An example visualization of going from a feature map of size $2 \times 2 \times 2$ to a connected layer containing four neurons can be seen in Figure 2.9.

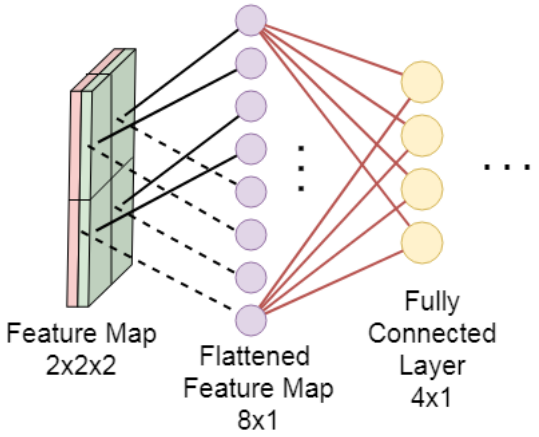


Figure 2.9: An illustration of a feature map of size $2 \times 2 \times 2$ being connected with a fully connected layer of four neurons. Note that the yellow circles are the only neurons in the shown image, and thus the red lines correspond to the present learnable weights. Moreover, the dotted lines refer to the values from the second channel (red) in the input feature map.

2.5 Transfer Learning

A common understanding of deep learning methods is that they require a large amount of data to be able to perform tasks well. While this understanding is true when the size of neural networks is large, there is a workaround called transfer learning. By using some pre-defined network architecture, it is possible to transfer weights that are already trained on a model of the same architecture. Instead of trying to create feature representations based on some data from scratch, one can use transfer learning to use the feature extraction available from training on larger sets of data. This, if applied correctly yields both a possible performance increase but also training time can be cut down significantly.

There are requirements for transfer learning to be profitable; however, first and foremost, it relies on a big assumption that patterns extracted in the original dataset are useful in the new data. Essentially that the data depicts the same type of things. For instance, trying to classify whether tumors are benign or malign through scanned images can typically not use information from another network trained on images of cars. Additionally, the input should be of the same type; e.q, a BEV image typically does not share much information with a front-facing one.

2.6 Evaluation

As mentioned in Section 2.2.2, object detection models generally yield a differing amount of predictions depending on the input data. This varying amount of output bounding box predictions means that evaluating such models is non-trivial. Gener-

ally, there are four types of predictions that can be made from an object detection model.

- True positive (TP): A predicted bounding box that matches with a ground truth object.
- True negative (TN): A correct prediction of background (no bounding box).
- False positive (FP): A predicted bounding box that does not match with any ground truth object (or an additional overlapping prediction).
- False negative (FN): A missed ground truth.

What is considered as a match in the case of object detection varies from task to task. Generally, each dataset has its definition of what defines a true positive depending on what the dataset contains and what task it aims to solve. The two most common metrics to define a match are distance and IoU, i.e, whether a prediction is within a set threshold in terms of either distance or overlap to a ground truth object.

In Figure 2.10, the four possible types of predictions are demonstrated through the relation between ground truths and actual predictions on an IoU metric with an arbitrary threshold.

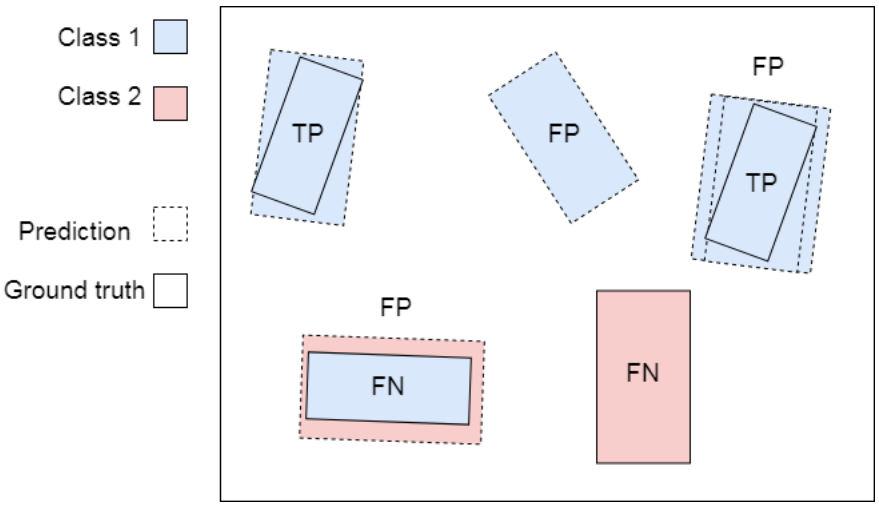


Figure 2.10: Example of the types of predictions that can be made by an object detection model. The illustration consists of examples of True Positive (TP), False Positive (FP) and False Negative (FN) predictions.

Furthermore, each prediction made has an associated score, referring to how confident the model is of the prediction, as mentioned in Section 2.2.3. This score of each prediction can then be used along with the actual predictions to express a relation between how many predictions are correct and how many predictions are made, above a set score threshold. This is known as the relation between *precision*

and *recall*. Precision refers to how many predictions out of the predictions made that are correct, defined by

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}. \tag{2.14}$$

Recall on the other hand, measures how many out of the total amount of ground truth objects that are currently found, defined as

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}. \tag{2.15}$$

Mean Average Precision

Mean Average Precision (mAP) is a common metric to measure the accuracy of object detection models as a whole. The Average Precision (AP) is calculated by taking the value of the precision achieved at different recall values. This is done by successively decreasing the set confidence threshold, allowing more predictions to be made until all ground truths are found (or until the model can't find anymore).

The AP is generally calculated by taking the integral of the precision-recall curve as

$$\text{AP} = \int_0^1 P(R)dR, \tag{2.16}$$

where R is the recall and $P(R)$ is the precision achieved at that specific recall value. An example of a set of predictions and their corresponding precision-recall curve can be seen in Figure 2.11, with a table of the successive decrease in threshold.

In the PASCAL Visual Object Classes Challenge 2012 [12], Everingham et al. proposed an Interpolated Average Precision (IAP) to achieve a steadier curve. IAP is not as heavily punished for making wrong predictions with a high confidence. The interpolated precision is calculated by taking the maximum precision of all the subsequent recall values above the current recall. IAP is typically calculated through numerical integration as

$$IAP = \frac{1}{N} \sum_{n=1}^{N} \max_{n \leq i \leq N} P(R_i), \tag{2.17}$$

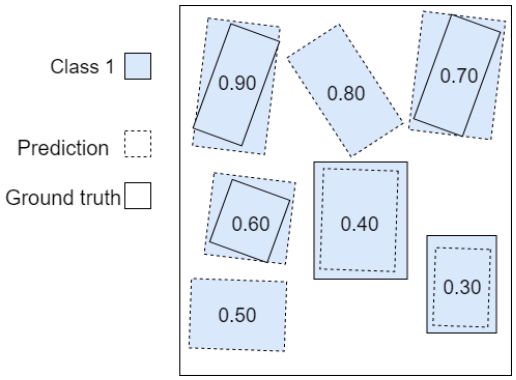
where N is the number of recall levels and R_i is the recall value for level i. In the PASCAL challenge Everingham et al. used eleven recall levels $R \in (0, 0.1, 0.2, ..., 1)$.

The IAP could be graphically explained as the area under the interpolated precision-recall curve in Figure 2.11(c), representing the interpolated precision on the same data used to calculate the precision-recall curve.

Additionally, the mAP is calculated by taking the mean of the used AP for the different classes. According to

$$\text{mAP} = \frac{1}{N} \sum_{c}^{N} \text{AP}(c), \tag{2.18}$$

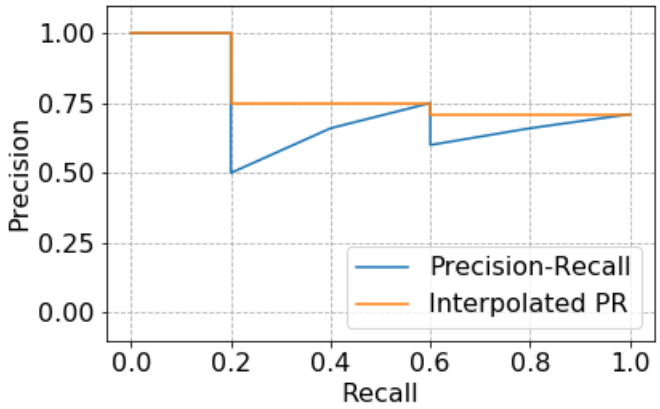
where $\text{AP}(c)$ is the used AP for class c and N is the number of classes.



(a) An example set of predictions with corresponding confidence scores.

Threshold	TP	FP	FN	Recall	Precision
0.90	1	0	4	0.2	1.0
0.80	1	1	4	0.2	0.50
0.70	2	1	3	0.4	0.66
0.60	3	1	2	0.6	0.75
0.50	3	2	2	0.6	0.6
0.40	4	2	1	0.8	0.66
0.30	5	2	0	1.0	0.71

(b) The precision and recall values for different confidence thresholds on the predictions presented in figure (a).



(c) Both the normal and interpolated precision-recall curve based on the values in table (b).

Figure 2.11: An example of a set of predictions with both the explained precision-recall curves. The area under the graphs are the corresponding average precision value for the two curves. The example uses six recall levels $R \in (0, 0.2, 0.4, 0.6, 0.8, 1.0)$, .

2.7 K-Means Clustering

K-means clustering is a heuristic algorithm used to try to find a good statistical partitioning of data. Input to the algorithm is the entire dataset to cluster along

with a k value; the actual k value attributes to how many clusters the algorithm should partition the data into.

k-means clustering was used within the thesis to explore the used dataset. Specifically to try to find default bounding box sizes that reflect the sizes of the objects within the dataset as well as possible. The specifics regarding the usage of k-means clustering for this purpose is detailed in Section 4.4.2.

The basic algorithm works by initially randomly selecting k cluster center points to yield the initial clustering. Thereafter, the entire dataset is iterated, and each point is assigned to the cluster with the center closest in terms of Euclidean distance given by

$$d(a, b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2 + (a_z - b_z)^2}. \tag{2.19}$$

When each point in the dataset has been assigned to a cluster, then each cluster center is recalculated to the average of the entire cluster. The entire algorithm, is detailed in Algorithm 1 explaining the steps in pseudocode.

Algorithm 1 K-means clustering

input K, $points: (x_1, ...x_n)$
Place c_1, ..., c_k centroids randomly
while True **do**
    for Every point x_i **do**
        for Every centroid c_j **do**
            Calculate $d(x_i, c_j)$
        Find the nearest centroid $\min_j d(x_i, c_j)$
        Assign the point x_i to cluster j
        for Every cluster j = 1 ... K **do**
            Calculate new centroid c_j = mean of all points x_i assigned to cluster j
    End loop if there was no update of centroids from previous iteration

3
Related Work

As mentioned in Section 1.1, recently deep learning techniques have made progress in the field of object detection. This chapter aims to give an overview of the current State-Of-The-Art (SOTA) methods for performing both 2D and 3D object detection.

3.1 2D Object Detection

The deep learning methods for 2D object detections can be divided into two different types: single-stage, and two-stage detectors [42]. In two-stage methods, the first stage refers to extracting features and proposing Regions of Interest (RoI), while the second stage refers to computing the final bounding boxes and class probabilities for each RoI. Conversely, single-stage methods do not create RoIs. Instead, the regions used for further classification and regression are chosen deterministically in beforehand.

One of the best performing SOTA two-stage detectors is Faster Region-based Convolutional Neural Network (Faster R-CNN). In general, two-stage methods achieve higher accuracy than single-stage methods. However, they tend to be slower both in terms of training and actual inference time.

Regarding one-stage methods, the current SOTA is commonly known as You Only Look Once (YOLO) and Singe Shot Multibox Detector (SSD). These methods reach lower accuracy but are faster in comparison to the two-stage methods. Due to the speed being of such importance for AD, these types of methods are especially promising for the thesis objective. This section begins with explaining the fundamentals for Faster R-CNN and ends with describing the two-stage detectors YOLO and SSD.

3.1.1 Faster Region-based Convolutional Neural Network (Faster R-CNN)

As mentioned in Section 2.2.2 the original way of producing region proposals was through the sliding-window method, where essentially all possible bounding box placements are considered individual input images to a separate classifier. Girchich et al. proposed a more practical approach in their original R-CNN paper [15]. The approach was based on the selective search algorithm [44], which was used to remove region proposals unlikely to contain objects effectively. The selective search

algorithm consists in simple terms of grouping similar regions of the input together through basic color segmentation.

Further improvements were made by the same authors in two iterations, firstly through the release of Fast R-CNN [14]. The fast R-CNN provided a significant speedup in comparison to the original release. This speedup was achieved by extracting features once for the entire input instead of extracting features for each region proposal individually. To be able to extract feature maps of equal size from each region proposal, an operation called RoI pooling is applied. RoI pooling projects each region proposals into the entire feature map and then performs max-pooling (with stride and size depending on the size of the regions) within each region, yielding feature maps of a set size for every region proposal.

With the release of the latest model iteration named Faster R-CNN [39], Girchich et al. proposed the Region Proposal Network (RPN) to create the region proposals as a part of the network being trained. The RPN replaced the selective search method, which was the computational bottleneck in the previous iterations, yielding the entire (simplified) network shown in Figure 3.1.

The RPN works by applying the sliding-window method on the produced feature map from the initial CNN feature extractor. At each grid location within the feature map, the network proposes multiple region proposals based on predetermined bounding boxes sizes as can be seen in Figure 3.2. These predetermined bounding box sizes are commonly known as anchors and serve as a default size for the region proposals to assume. The RPN thus produces in total $W \times H \times k$ anchor placements where W and H is the width and height of the feature map, and k is the number of anchor boxes used. For RPN to handle objects of differing scales, additional larger anchor boxes must be placed. Furthermore, the RPN sends each of the initial anchor placements through a simple classifier, to predict whether it contains an object or not (binary classification). This simple classification determines whether the anchor placement should be kept or not. Additionally, each positive anchor (those predicted to contain objects) are sent through a simple regressor to refine the bounding box placement generated from the anchor. Finally, the remaining positive anchors are pruned through NMS as explained in Section 2.2.5, producing the final RoIs that are sent further through the network.

The RoIs are then merged with the feature map through RoI pooling, producing the regional feature maps. Finally, each regional feature map is sent through an additional classifier as well as an additional regressor. The final classifier outputs the classification probabilities for each possible class, with an additional *background* class used to discard region proposals. The final regressor tries to, similarly as in the RPN, further refine the location and size of the produced bounding boxes. The refinement is done by trying the estimate the difference (in terms of both position and size) between the proposed bounding box and the actual object.

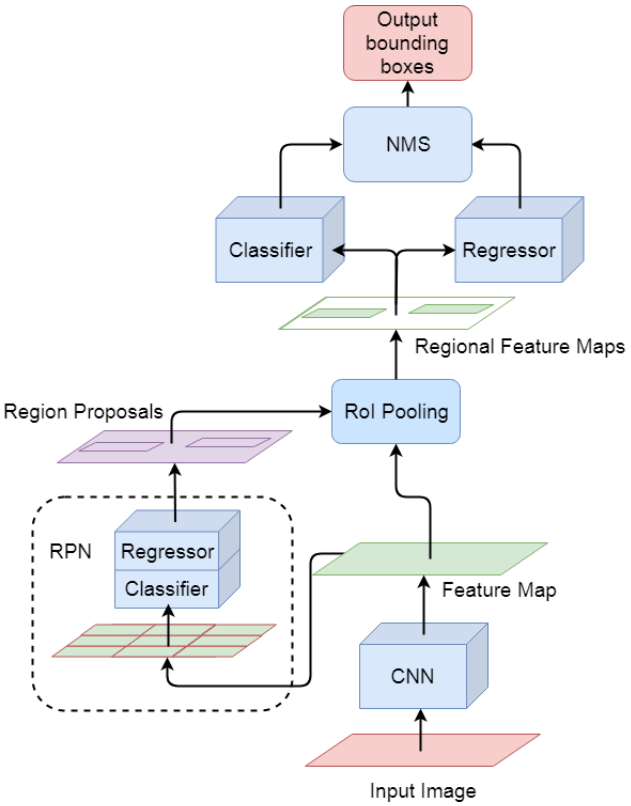


Figure 3.1: An illustration of the simplified flow from an input image to the output bounding boxes in the Faster R-CNN method. A CNN produces a feature map from the input image, that feature map is sent to the RPN generating region proposals. The feature map is then merged with the region proposals through RoI pooling, whereas the resulting feature maps (one per region proposal) are subject to classification and bounding box offset regression to produce the bounding boxes. Finally, the produced bounding boxes are subject to NMS pruning, generating the final outputs.

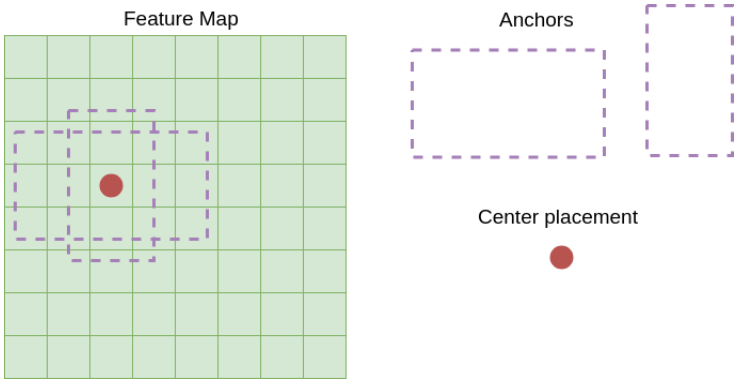


Figure 3.2: An example of creating initial region proposals through the use of an RPN. The RPN applies the sliding-window technique on a feature map, and for each sliding-window placement, predefined boxes called anchors, are placed corresponding to the produced region proposals.

The training is done by first creating the target outputs, these targets are generated through matching the anchors with the ground truths. The closest anchor both in terms of placement and shape is matched to each ground truth. The corresponding matched anchors are filled with the offset and class values, and the unmatched anchors are set to the background class, together creating the supposed targets. These targets are then used along with the actual outputs from the network in the loss function, called multi-task loss[14], which is used to train the model.

3.1.2 Single Shot Multibox Detector (SSD)

In the SSD model, instead of first trying to propose regions which are then used for predictions as in the two-stage methods, Anguelov et al. present a way of predicting bounding boxes directly through a single deep neural network in their SSD model [27]. The SSD encapsulates all computation into a single network, meaning that it is significantly easier to train in comparison to two-stage methods were the different stages typically have to be trained in multiple phases.

The SSD model does, similarly to the RPN as explained in Section 3.1.1, place default sized anchor boxes on the produced feature map as shown in Figure 3.2. However, the feature map produced in SSD is not taken as a direct output from the initial CNN feature extractor. The feature map is rather created by extracting features from different layers in the CNN architecture, yielding representations of different scales. Thus, it enables the model to detect objects of different sizes by placing anchors at different feature map scales as shown in Figure 3.3. The actual anchor placements are done on the grid in each feature map. Yielding $W \times H \times k$ anchor placements per feature map, where k is the number of anchor placements, W and H is the width and height of the specified feature map.

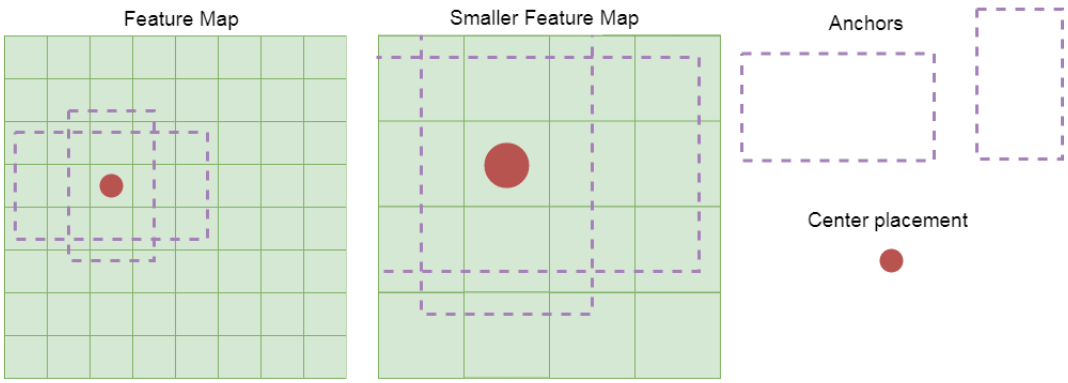


Figure 3.3: An example of the SSD model placing anchors in two differently sized feature maps. These anchor placements refer to bounding box predictions on two different scales when overlayed with the original input image.

The number of feature maps, and at which scale, that are extracted differ between implementations. An example, showing three feature extraction levels is shown in Figure 3.4 along with a simplified structure of the entire SSD flow. The output

yielded from the SSD model for every grid location in the feature maps is in the form of $k \times (B + C)$, where k is the number of anchors, B is the number of values defining a bounding box, and C is the number of classes.

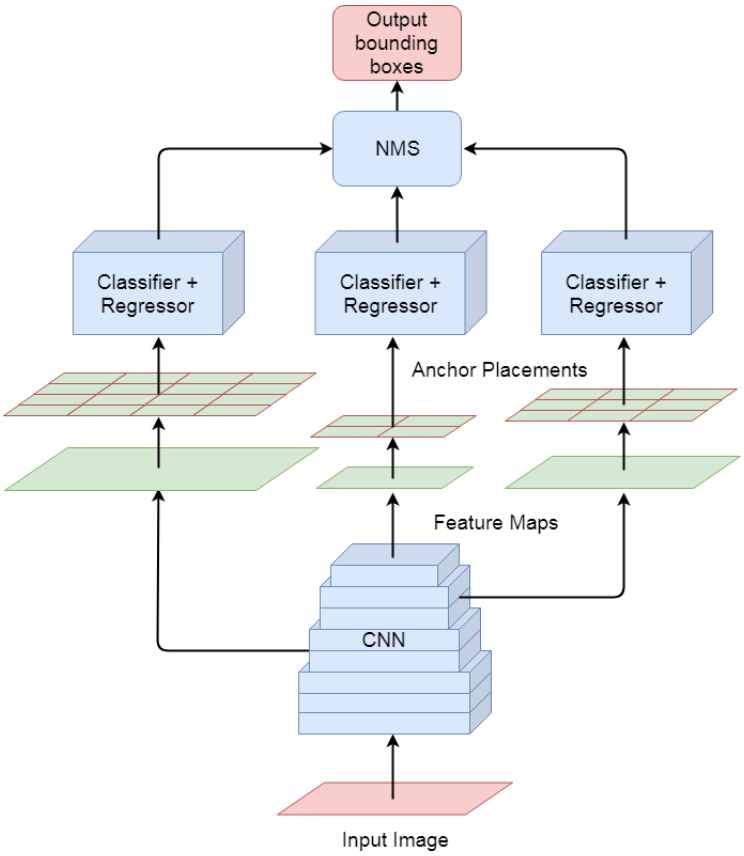


Figure 3.4: A simplified view of the SSD model, showing an example of how SSD extract feature maps from three different levels in the original CNN architecture. Each feature map is divided into a grid where anchor boxes are placed, which are then sent through a convolutional layer yielding the classification and regressional values for the bounding boxes. Finally, the produced bounding boxes are subject to NMS pruning, generating the final outputs.

Similarly to Faster R-CNN, the training consists of generating targets by matching anchor boxes to ground truths. This matching is done in SSD by calculating the IoU between the placed anchor boxes and the ground truths, all overlaps produced above a set threshold is then considered a match. This means that the matching is not done in pairs and that several boxes are allowed to predict the same object, which is something that is commonly pruned in the postprocessing anyway, this simplified the learning problem. The generated targets are then used along with the actual outputs from the network in the loss function to optimize.

3.1.3 You Only Look Once (YOLO)

The first version of YOLO was released in 2016 by Redmon et al. [36], just like SSD it is based on a single pass through one unified network. The produced feature map

is split into a grid, and a set of bounding boxes is predicted for each grid location. In the YOLOv2 [37] by the same authors, they introduce the predictions as offsets from the anchor boxes to be placed within the grid locations similarly to both SSD and Faster R-CNN.

The input image is processed by a CNN to extract the feature map, to later perform the actual predictions on. The simplified network flow is presented in Figure 3.5 showing high similarity to the network flow of the SSD model. The output from the YOLOv2 network for every grid location in the feature map has the form $k(B+1+C)$, where k is the number of anchors, B is the number of values defining a bounding box, C is the number of classes, and 1 denotes the additional confidence score. The additional confidence score refers to the probability that the specific anchor contains an object regardless of class.

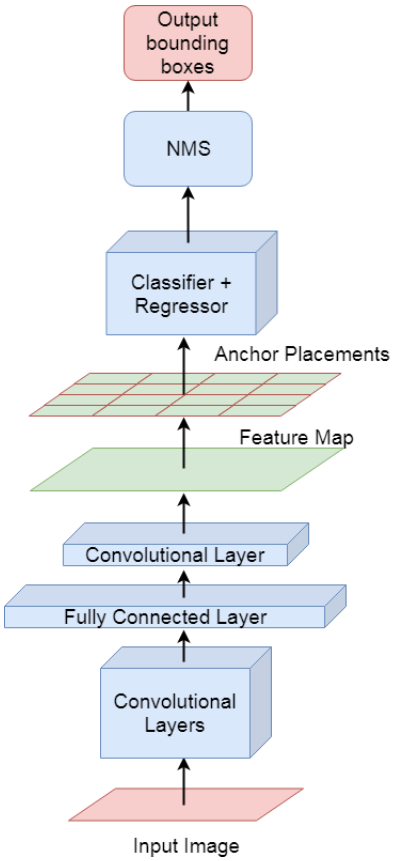


Figure 3.5: A simplified version of the network architecture of the YOLO model. The input image is processed by the convolutional layer to extract a feature map. The anchors are placed onto each grid of the feature map, processed by classifier and regressor yielding the predictions.

The YOLO methods are mainly aimed towards applications that are in need of fast runtime. Thus the CNN structure used in the models are designed thereafter. YOLO has not achieved the highest accuracy, however, they pushed the limits of low inference time when it comes to 2D object detection models. An additional improvement was made in the YOLO-series with the YOLOv3 model [38], it extracts

features from several different layers in the CNN structure. By using feature maps of different scales allows better accuracy regarding prediction on smaller objects, which is one of the main issues with the previous versions in the YOLO-series.

3.2 3D Object Detection

As mentioned in Section 1.2 the problem with 3D data and most specifically point cloud data is that the data is sparse, meaning that there does not exist data points in every possible position. This leads to that convolutional filters (commonly used for feature extraction) cannot be applied directly to the data. To counteract this, the point clouds needs to be encoded into a representation such that values exist for every location in the data.

Most SOTA 3D object detection methods exploit 2D object detection techniques [23], [40], where the predicted bounding boxes used for matching are in 2D with the values for the additional dimension added as extra regression targets. This section describes three examples of promising networks using the 2D object detecting techniques described in Section 3.1, on encoded 3D point cloud representations.

3.2.1 VoxelNet

VoxelNet, presented by Zhou & Tuzel [48], uses an RPN technique similar to that of the SSD explained in Section 3.1.2, to generate 3D predictions on processed point clouds. It solves the problem with sparse 3D point cloud data by discretizing the point cloud into equally spaced voxels (3D pixels) as visualized in Figure 3.6.

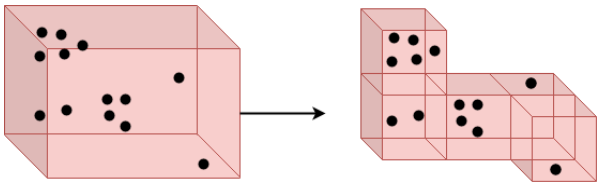


Figure 3.6: A simple illustration of how point clouds are discretized into voxels.

The values of each voxel are based on the data points from the point cloud that are located within the bounds of the voxel. A threshold is set on the number of points to represent each voxel, and if the number of points exceeds the threshold, the points to represent the voxel are randomly sampled. Furthermore, if a voxel contains less point than the specified value, it are padded with zero-valued points. All non-empty voxel is sent through a simple neural network, called the voxel feature extractor, to go from each voxel being expressed as a set of point into a set of features as shown in Figure 3.7.

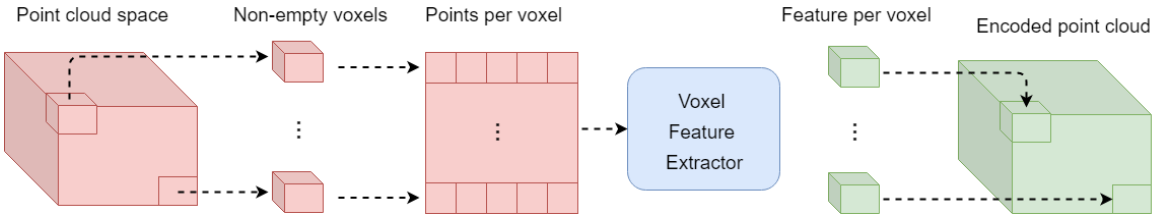


Figure 3.7: A simplified flow of the encoding of point clouds into a dense feature representation for the VoxelNet. The voxel feature extractor refers to a simple neural network taking a fixed set of points for each non-empty voxel as input and produces a feature representation for each voxel. The non-empty voxels from the point cloud input are sent through the voxel feature extractor and then scattered, based on their original position, into the encoded point cloud representation.

For the empty voxel locations, the encoded point cloud is padded with zero-valued features. Thus making the usually sparse point cloud data into a dense representation that a feature extractor can be applied to. The feature extractor used in the VoxelNet implementation uses 3D convolutional layers that, conversely to the convolutional layers explained in Section 2.4.1, strides in three directions on the feature map rather than two. The final produced feature map is then used by the RPN to place three-dimensional anchor boxes, and make predictions. The predictions are made by matching the anchor boxes similarly as in the single shot 2D object detection methods presented in Section 3.1 with the direction as an additional regression target.

VoxelNet achieves a high accuracy but struggles in terms of inference time in comparison to other methods. The reason for the slow inference time is the time consuming 3D convolutional operations. An additional issue with the voxel representation is that, due to the sparsity of point clouds, a significant amount of the produced voxels are empty causing a large number of unnecessary calculations.

SECOND, proposed by Yan et al. [45], is an additional model that is based on VoxelNet as an improvement. As stated, one of the big issues with VoxelNet is that the three dimensional convolutional operations are slow due to the sparsity of the data remaining in the voxel representation. Therefore to solve this issue, SECOND introduced sparse convolutions to the VoxelNet model. The sparse convolutions, in simple terms, refer to skipping convolutional computations when there are not enough points present in the corresponding region in the input point cloud. This enables significant speedup in comparison to the original implementation .

3.2.2 The Complex-YOLO

Complex-YOLO, presented by Simon et al. [40], is a 3D object detection network that expands the YOLOv2 network to handle 3D point cloud data. It handles the issue with the sparsity in the point cloud representation by projecting the point cloud into a two dimensional BEV grid, as shown in Figure 3.8.

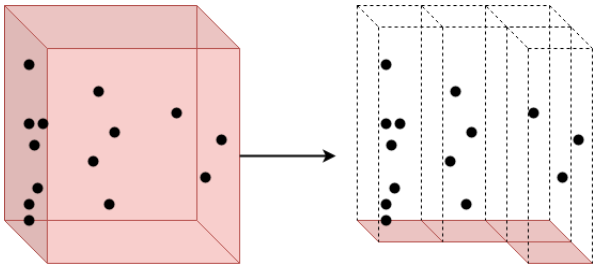


Figure 3.8: Illustration of how a point clouds is projected into a 2D BEV representation.

The values for each projected grid location is based on the points contained within each pillar being projected. In Complex-YOLO there are three values that are directly calculated, the maximum intensity, height along with a normalized density of the points in the pillars. These three values are encoded as three different channels in an image, creating an input that can be directly processed by 2D convolutions as those in the YOLOv2 implementation presented in Section 3.1.3.

The model extends the YOLOv2 network by adding additional regressional targets to refer to the height, z position, and direction of the 3D bounding boxes. The directional regression is done through a complex angle, hence the name of the model, containing two values, t_{im} and t_{re} where the angle α can be calculated through

$$\alpha = \arctan(\frac{t_{im}}{t_{re}}). \tag{3.1}$$

Simon et al. argue that predicting the angle in this way closes the mathematical space of direction, making it easier for the network to converge towards good directional prediction. Conversely, mathematical space of direction is open in a network that predicts the angle directly as it sees a great difference between i.e., 180° +180°, and +540°, which effectively refer to the same orientation.

The Complex-YOLO pushes the limits of low inference while still achieving comparable results when it comes to 3D object detection. In comparison to other 3D object detection methods presented in this section, it also uses a relatively simple point cloud encoding.

3.2.3 PointPillar (PP)

PP presented by Lang et al. [23], is an additional 3D object detection method that is based on the idea of VoxelNet presented in Section 3.2.1. Lang et al. argued that to achieve both better inference time and accuracy, one could instead of voxels base the representation on pillars. Effectively the voxel representation is still used, where the height of each voxel equal the height of the entire point cloud. The point clouds are discretized into equally spaced pillars as presented in Figure 3.9.

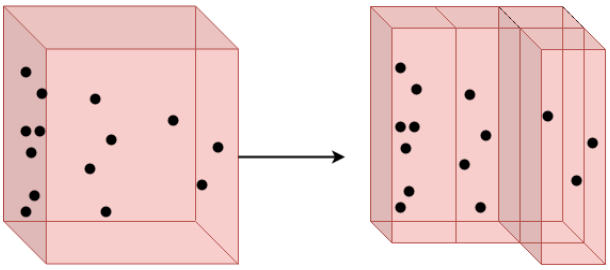


Figure 3.9: Simplified illustration of how point clouds are discretized into pillars for the PP model.

The values for each pillar is based on the point cloud data points that are enclosed in each pillar. A value is introduced on the number of points to represent each pillar, if there are more points present within a pillar they are randomly sampled, and if there are fewer points present, then they are padded with zero-valued points.

Moreover, after the actual split into pillars has occurred, each remaining point is augmented. The augmentation consists of adding the distance to the mean point in the pillar it is located in (x_m, y_m, z_m), along with the offset from the middle of the pillar (x_o, y_o). This augmentation changes the dimensionality of each point from four (x, y, z, r) to nine (x, y ,z ,r , x_m, y_m, z_m, x_o, y_o).

All pillars that are not empty are sent through a simple neural network, called the point feature extractor. The point feature extractor enables each pillar to go from being represented as a set of points into a set of features. The full flow from the pillar representation into the feature representation is presented in Figure 3.10. Additionally, a limit is commonly introduced on the number of non-empty pillars being sent to the point feature extractor, to limit the memory consumption for implementations.

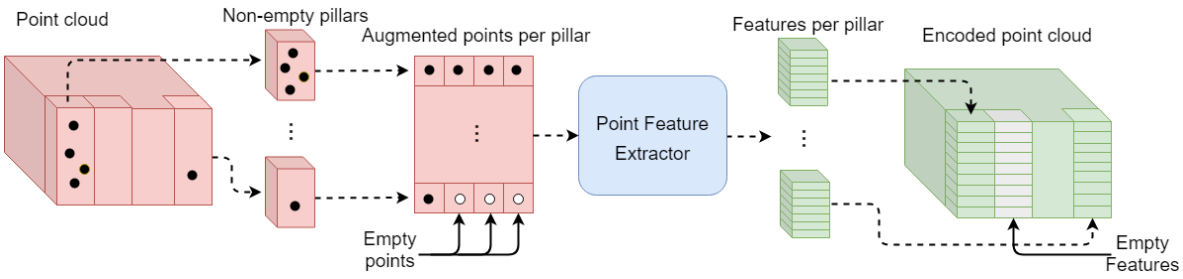


Figure 3.10: A simplified flow of the encoding of point clouds into a dense feature representation for the PP model. The point feature extractor refers to a simple neural network taking a fixed set of points as input and produces a feature representation of the set. The non-empty pillars from the point cloud input are sent through the point feature extractor and then scattered, based on their original position, into the encoded point cloud representation. For pillars containing less points than a set value, empty points are added. For pillars that are empty, corresponding empty features are added into the encoded output.

Due to how the encoded point cloud is structured, it can be interpreted as an information-rich dense image with a depth equal to the number of features output from the point feature extractor for each pillar. This makes it possible to effectively process the entire feature map by 2D object detection methods. The decoder used in the original PP implementation was a modified SSD. The modification added regressional targets in the shape of height, z position and direction, along with an additional binary classification for the direction. The purpose of the directional classification is to be able to predict the heading of an object in relation to the matched anchor, thus being able to more effectively use the anchor boxes original orientation as a point of offsets.

4
Methods

In this chapter, the fundamentals of the methods used to create, train, and evaluate the different model architectures produced in the thesis are described. The chapter starts with explaining the tools and dataset used, followed by a description of the data augmentation and anchor generation. Thereafter, the implementation section contains specifics regarding the creation and configuration of the model architectures. The chapter ends with an explanation of the training settings and evaluation methods for the models.

4.1 Tools

Creating an object detection system entirely from scratch is a time-consuming task, considering the sheer amount of concepts applicable to the deep learning task. Luckily, there are existing libraries and frameworks available to speed up the implementation process. In this section, some of the more comprehensive tools used within this thesis are explained.

4.1.1 Pytorch

The open-source machine learning library - Pytorch [35] is used to simplify the implementation of the neural network models, The library is developed by Facebook and contains functions to create, train and modify neural network architectures with relative ease. By using the framework, most machine learning concepts explained in Section 2.3 such as convolutional layers and backpropagation are implemented with just a few lines of code. Moreover, the framework is optimized towards GPU computations, yielding a significant speed-up to the training process when used in unison with CUDA.

4.1.2 CUDA

CUDA is a parallel computing platform created by Nvidia [33]. CUDA enables users to run parts of their code on the GPU rather than the CPU, to speed up execution. The speed-up is achieved through exploiting the GPU for specific operations such as matrix multiplication, that GPUs can perform more effectively than CPUs. Matrix multiplication is used extensively when performing both forward and backpropagation through a neural network, meaning that CUDA enables significant speed-up when training deep neural network.

4.1.3 Numba

Numba is a Python compiler that makes it possible to translate simple Python functions into optimized machine code during runtime, thus making it possible to write code that is comparable in speed to traditionally compiled languages (such as C) [22]. The most prominent speed-up is achieved in loops that make heavy use of arrays and scalars. In this thesis, Numba provided significant speed-up in comparison to pure Python code by optimizing functions frequently used during training.

4.1.4 Visdom

Visdom is a tool for creating plots and sharing live visualizations, created by the research department at Facebook. Visdom makes it possible to have live plot updates during training.

4.1.5 nuScenes-devkit

Along with the nuScenes dataset described in Section 4.2.2 came a Python development kit for handling data samples and their annotations [7]. It provides essential functions for loading data samples whilst keeping track of annotations belonging to the specific sample. The development kit is also used to visualize 3D predictions within the actual scenes, as seen in Figure 4.1 and 4.2.

4.2 Datasets

As mentioned in section 1.1, training a neural network typically requires a large amount of data. Collecting and annotating this type of data takes a significant amount of time. Fortunately, there are several public datasets available for research purposes. This section introduces two of the largest publicly available datasets for 3D object detection, KITTI and nuScenes. KITTI was not explicitly used within the project, however, it is presented here as virtually all related work presented in Section 3.2 is based on it. The nuScenes dataset is used in this thesis mainly due to the presence of significantly more data in comparison to KITTI.

4.2.1 KITTI Vision Benchmark Suite

The KITTI dataset was released in 2012, by the Karlsruhe Institute of Technology and Toyota Technological Institute, and has been widely used as a standard for comparing deep learning tasks within autonomous driving among researchers since its creation [13]. The entire dataset consists of approximately 15000 data samples roughly split equally into a training and test set. The data was gathered in Karlsruhe, Germany, with two front-facing cameras (stereo) and a 360° LiDAR.

Data Annotation

The KITTI benchmark suite contains annotations for the training set in the shape of 3D bounding boxes as well as class labels for six different classes such as cars, cyclists, and pedestrians. The annotations have been made manually from the camera data, and due to the cameras only being front-facing, less than half of each point cloud contains annotations for present objects.

Evaluation

As the annotations for the test set is not publicly available, the evaluation of a model on the official KITTI test set can only be done through the competition available on the datasets website. However, by the time this thesis was carried out, only submissions with significant novelty leading to a paper in a conference or journal were allowed. KITTI explicitly stated that student research projects are not permitted to use their test evaluation server.

4.2.2 NuScenes Dataset

The nuScenes dataset [7] was released throughout this thesis. The capturing of the data was done with six cameras placed all around the vehicle, effectively yielding a 360° Field of View (FoV), along with a 360° LiDAR scanner (and Radars). The full dataset is provided by Aptiv Autonomous Mobility and includes 1000 scenes each spanning for approximately 20 seconds. The 1000 scenes are collected from dense traffic situations in both Singapore and Boston, the scenes are split into a training set of 700, a validation set of 150 and a test set of 150 scenes. Each scene is annotated with a frequency of 2Hz (2 annotations per second), meaning that there are approximately $1000 \cdot 20 \cdot 2 = 40000$ annotated samples in total.

Data annotation

Similarly to KITTI, the annotations contained for the training and validation set is in the shape of 3D bounding boxes with classes. However, the annotation also contains attributes such as whether a pedestrian is standing, sitting, or moving. In total there are 23 classes present in the annotations (see the full list in Table 4.3). The annotations are applied in 360° around the vehicle, meaning that in contrast to KITTI the entire point cloud can be used. An annotated sample visualized through the six different cameras around the vehicle can be seen in Figure 4.1.

Extending on the samples, the dataset also contains additional sweeps from in between the annotations yielding a total of approximately 400 000 LiDAR sweeps (LiDAR captures data in 20Hz, whilst annotations are applied in 2Hz). A possibility to load up to 10 sweeps per annotated sample is implemented to get a denser point cloud input that is subject to movement. In Figure 4.2 two LiDAR point clouds are presented in a BEV format of the same sample presented in Figure 4.1, the two point clouds are presented with a differing amount of sweeps.

Figure 4.1: Sample from the nuScenes dataset through the six different camera positions around the vehicle. The positions of the pictures reflect the corresponding camera positions, presented from the top: front, front left, front right, back left, back right and back. The annotated ground truth objects present in the pictures are three cars (orange bounding boxes) and one motorcycle (red bounding box).

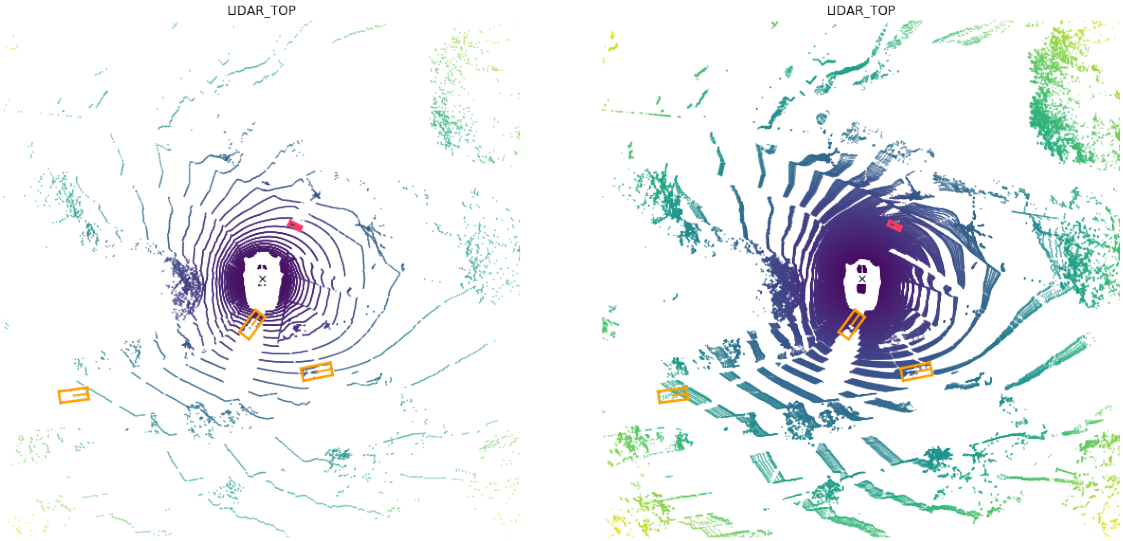


Figure 4.2: Two BEV representation of the LiDAR point clouds, from a nuScenes sample. The left point cloud contains data from one sweep, meaning that only the annotated sample data is represented. The right point cloud contains data from 10 sweeps, meaning that the sample data is overlayed by the previous 9 LiDAR data collections.

nuScenes annotation contains a visibility level for every object. The level is set to an integer between one and four depending on the percentage of how much of the object is visible through the camera sensors as presented in Table 4.1. This parameter is used when exploring the evaluation to compare the performance of the models on different visibility levels.

Level	Visibility
1	0-40%
2	40-60%
3	60-80%
4	80-100%

Table 4.1: The visibility level for an annotated object in relation to how much of the object is visible through the ego vehicles cameras.

Evaluation

With the release of the dataset, nuScenes introduced a new metric, the nuScenes Detection Score (NDS), to balance all aspects of the detection performance [7]. Similarly to KITTI, the evaluation on the test set is done through an evaluation server. Access to this official evaluation server is limited. Thus a separate evaluation metric was implemented. The separate evaluation is presented in Section 4.9, whereas only the best model according to this metric is sent to the official evaluation server.

The NDS contains the mAP metric presented in Section 2.6, with a True Positive Metric (TPM) set to a threshold of 2D center distance. An advantage of using distance as a TPM metric is that it is not as sensitive to small translation errors as the TPM based on IoU for smaller objects such as pedestrians. The distance TPM does not take orientation and scale into account; therefore these are evaluated in separate metrics for the NDS expressed below.

- Average Translation Error (ATE), in terms of 2D Euclidean distance between the centers of predictions and ground truths.
- Average Scale Error (ASE), calculated as $1 - \text{IoU}$ after aligning centers and orientations of predictions and ground truths.
- Average Orientation Error (AOE), referring to the yaw angle difference between predictions and ground truths.

The concluding NDS is a weighted sum of the above mentioned separate metrics, in this sum, there are additional metrics that are outside of the thesis' scope, including velocity and attribute accuracy. One thing to note is that the mAP within the benchmark is calculated only above 10% in both precision and recall, reducing the noise for the measurements.

Considering that the amount of points in a point cloud diminishes with increasing radius from the gathering sensor, an upper bound in terms of range is set on the evaluated predictions. The upper bound varies depending on the class and is presented in Table 4.2.

Detection class	**Detection range**
Bicycle, Motorcycle, Pedestrain	40m
Car, Bus, Construction, Trailer, Truck	50m

Table 4.2: Range limit used in evaluation for each class of interest in the nuScenes dataset, measured from the ego vehicle in meters.

Version 0.1 - Teaser set

Initially as the project started the full nuScenes dataset was not released, however, a teaser dataset consisting of approximately 10% of the full dataset was available. Out of the total of 1000 scenes, 100 scenes were released as version 0.1. As the project began with using this set, the initial experiments are trained on the teaser data rather than the full dataset.

4.3 Merging Classes

Since the dataset was gathered from real-world situations, an expected class imbalance issue arose. The amount of instances present in the nuScenes dataset for each

class is varying from approximately 50 to 500 000 instances for the ambulance and car class respectively, implying a difference of four orders of magnitude between the classes. This imbalance is expected, as one would not expect to see as many ambulances as cars while driving. Based on an imbalance of classes in terms of training an object detection model, one can not expect to achieve good results on a class that is extremely rare in the data.

Therefore decisions were made to merge and remove redundant classes in the dataset, to simplify the object detection problem as a whole. The problem is generally simplified by giving the model less classes to choose from in terms of classification. Presented in Table 4.3 are the result of the merging of the 23 classes along with the counts of annotation for the two different versions of the dataset. Note that the final merging differs from the merging proposed by nuScenes, the reason is that movable objects was deemed out of scope for this thesis.

General Class:	Teaser Count	nuScenes Merging	Full Count	Final Merging
animal	6	Ignore	787	Ignore
movable_object.debris	500	Ignore	3016	Ignore
movable_object.pushable_pullable	583	Ignore	24605	Ignore
movable_object.barrier	18449	Barrier	152087	Ignore
movable_object.trafficcone	7197	Traffic_Cone	97959	Ignore
static_object.bicycle_rack	192	Ignore	2713	Ignore
human.pedestrian.stroller	40	Ignore	1072	Ignore
human.pedestrian.wheelchair	5	Ignore	503	Ignore
human.pedestrian.personal_mobility	24	Ignore	395	Ignore
human.pedestrian.adult	20510	Pedestrian	208240	Pedestrian
human.pedestrian.child	15	Pedestrian	2066	Pedestrian
human.pedestrian.police_officer	39	Pedestrian	638	Pedestrian
human.pedestrian.construction_worker	2400	Pedestrian	9161	Pedestrian
vehicle.bicycle	1685	Bicycle	11859	Bicycle
vehicle.motorcycle	1975	Motorcycle	12617	Motorcycle
vehicle.construction	1889	Construction	14671	Construction
vehicle.car	32497	Car	493322	Car
vehicle.bus.rigid	1115	Bus	14501	Bus
vehicle.bus.bendy	98	Bus	1820	Bus
vehicle.trailer	2383	Trailer	24860	Trailer
vehicle.truck	8243	Truck	88519	Truck
vehicle.emergency.ambulance	19	Ignore	49	Ignore
vehicle.emergency.police	88	Ignore	638	Ignore

Table 4.3: Kept and merged classes, along with the classes respective annotation count for the two dataset versions. The nuScenes merging column presents the merging nuScenes released alongside the full dataset. The final merging column shows the final merging used in this thesis, resulting in 8 classes.

Following is a brief summary of the annotation instructions for some classes in Table 4.3 that are deemed to have names that are non-self explanatory:

- Movable object: Refers to different types of movable inanimate objects that are typically present in traffic situations, such as trash bins and traffic cones.

- Human pedestrian personal mobility: Explained within the annotation as self-propelled vehicles such as skateboards or scooters, on which a person typically travels in an upright position.

- Vehicle emergency police: Refers to all different types of police vehicles, meaning a class containing both motorcycles, cars, and vans.

- Static object bicycle rack: A class added to handle many bicycles at the same location, thus removes the need to annotate each bicycle in the rack individually.

An issue present with classes such as police vehicles is that there are too few objects present in the dataset for them to be adequately trained as individual classes. Additionally, the police vehicles are hard to merge into other classes as they overlap several different classes (trucks and cars) as shown in Figure 4.3. Moreover, according to the annotation instructions above, police motorcycles are supposed to belong to this class, which would cause a greater class overlap issue.

Figure 4.3: Examples of three different police vehicles present under the class vehicle.emergency.police in the nuScenes dataset. The examples show different types of police vehicles that overlap into other classes such as cars and trucks.

Worth noting is that there are additional attributes present in the dataset for certain objects, such as vehicles (moving, stopped, parked), pedestrians (sitting/lying, moving, standing) and bicycles (with rider, without rider). However, predicting at-

tributes was not taken into consideration, and each used annotation was limited to a bounding box with the corresponding class.

4.4 Anchor Generation

Anchor boxes are as explained in Section 3.1 used as a starting point for the shapes of objects. Thus the models do not predict the actual size of the objects, but rather a prediction of how much an anchor should be offset to match the size. The usage of anchor boxes is especially effective when objects have a similar shape as the anchors, implying that the needed offset becomes smaller[47]. Therefore, it is important to choose anchor box sizes that represent the objects in the data well.
As the nuScenes dataset is using real-world coordinates instead of any projection (as in 2D image detectors) all objects should have a fairly similar shape. The difference in size for pedestrians are for instance at the maximum in the range of $\pm 1m$ as can be seen in Table 4.4 along with the smallest and largest object for each of the used classes. However, considering the amount of objects in the dataset the ranges presented in the table are based on possible outliers.

Class	Smallest Object	Largest Object
Pedestrian	[0.30, 0.41, 0.62]	[1.19, 1.59, 2.98]
Bicycle	[0.36, 0.92, 0.75]	[0.93, 3.04, 2.00]
Motorcycle	[0.76, 0.72, 1.22]	[1.82, 4.41, 2.00]
Construction	[0.68, 1.37, 1.13]	[4.79, 15.89, 9.00]
Car	[0.65, 2.02, 1.38]	[2.71, 11.52, 4.49]
Bus	[2.48, 4.41, 2.00]	[5.11, 21.30, 3.77]
Trailer	[1.23, 1.96, 1.39]	[3.57, 34.42, 4.60]
Truck	[1.70, 2.64, 2.00]	[3.44, 22.47, 4.68]

Table 4.4: Smallest and largest objects in the dataset for each of the proposed classes, presented as width, length and height in meters.

One additional aspect to have in mind when choosing anchors is the quantity, as the number of anchor boxes to place linearly increases the number of predictions the model outputs. This means that the number of anchors directly influences the number of calculations in the model, and thus the training and inference time. In essence, the chosen anchors should be aimed to cover the shape of the data as well as possible whilst minimizing the quantity.

4.4.1 Averaging

The most straight forward approach to calculating anchors, that should have good coverage of the dataset, is to take the average of each annotation size for each class individually. The resulting anchors for each class can be found in table 4.5. However, due to some classes having similar shapes, such as buses and trailers, there

should be a possibility to decrease the number of anchor boxes further while still covering the shape of the data.

Class	Avg Box Size	Count	Merged Class:	Avg Box Size	Count
Adult	[0.67, 0.73, 1.77]	208240	Pedestrian	[0.67, 0.73, 1.77]	220105
Child	[0.51, 0.53, 1.38]	2066	- | | -		
Police officer	[0.73, 0.69, 1.83]	638	- | | -		
Construction worker	[0.72, 0.72, 1.74]	9161	- | | -		
Bicycle	[0.60, 1.70, 1.28]	11859	Bicycle	[0.60, 1.70, 1.28]	11859
Motorcycle	[0.77, 2.11, 1.47]	12617	Motorcycle	[0.77, 2.11, 1.47]	12617
Construction	[2.85, 6.37, 3.19]	14671	Construction	[2.85, 6.37, 3.19]	14671
Car	[1.95, 4.62, 1.73]	493322	Car	[1.95, 4.62, 1.73]	493322
Bus rigid	[2.93, 11.23, 3.47]	14501	Bus	[2.93, 11.08, 3.47]	16321
Bus bendy	[2.96, 9.83, 3.45]	1820	- | | -		
Trailer	[2.91, 12.29, 3.87]	24860	Trailer	[2.91, 12.29, 3.87]	24860
Truck	[2.51, 6.93, 2.84]	88519	Truck	[2.51, 6.93, 2.84]	88519

Table 4.5: Average object size for each of the classes both before and after merging, along with the number of annotations of the corresponding class in the dataset. The object sizes are presented as width, length and height in meters.

4.4.2 K-Mean Clustering for Anchor Sizes

The k-means clustering algorithm can also be used to find appropriate anchors to cover the annotations of the dataset. As the goal of the anchor generation is to find the shapes that best represent the data, the metric between the points and the centroids is changed in the algorithm, from the Euclidean distance (2.19) to the IoU (2.1).

Due to the algorithm starting with picking k random boxes means that the output of the algorithm is stochastic. Hence to counteract the randomness, the algorithm is run multiple times to yield a better result. The best result for a specific k value is decided by the average IoU between the resulting anchors and the datasets annotations. The algorithm's average IoU with five repetitions from running for k values between one and ten is presented in Figure 4.4, while the resulting boxes for the five first k values are presented in Table 4.6.

K value	Boxes:	Avg IOU
1	[1.73 4.34 1.45]	54.95%
2	[1.99 4.68 1.77], [0.66 0.73 1.77]	75.73%
3	[1.95 4.60 1.72], [0.66 0.73 1.77], [2.86 9.21 3.61]	79.65%
4	[1.86 4.44 1.64], [0.65 0.73 1.77], [2.89 9.94 3.69], [2.10 4.92 1.88]	81.58%
5	[1.86 4.45 1.64], [0.60 0.61 1.75], [2.89 9.95 3.69], [2.11 4.94 1.88], [0.74, 0.91, 1.78]	83.59%

Table 4.6: The generated anchor boxes from the k-means algorithm, and their respective average IoU coverage of the datasets ground truth boxes.

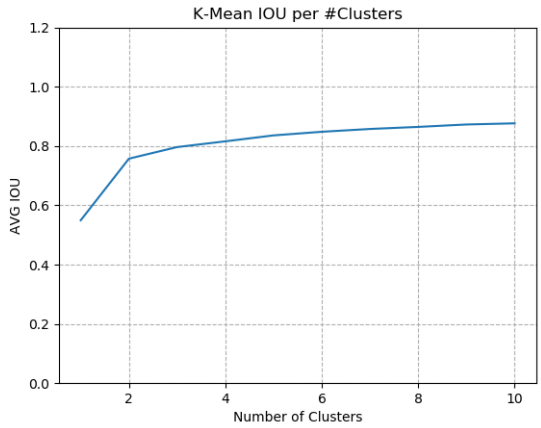


Figure 4.4: The average IoU coverage of the dataset with respect to the k clusters (anchors) generated from the k-means algorithm.

What is especially noteworthy from the graph in Figure 4.4 is that the average IoU reaches almost 80% after representing all shapes by only two anchor boxes. However, considering the imbalance of classes in the dataset this is to be expected, as the generated anchors are most likely covering the over-represented classes; pedestrians and cars. This is further verified by the anchor boxes presented in Table 4.6 as the first two k-mean generated anchors are closely related to the average size for the car and pedestrian class calculated in Table 4.5.

The specific anchors used in the implemented models are dependent on the classes to be predicted. The anchors used for each model are presented in Section 5. Generally, the models trained with multiple classes use multiple anchors, these models use k-mean generated anchors. The models trained on a single class use the average object size for that class as anchor. The reason is that by applying the k-mean algorithm with a k value of one, would yield the average of the data and not the average of the object looked for.

4.5 Implementation

For a convolutional neural network to be able to process a 3D LiDAR point cloud, the data needs to be represented more densely. This is done by encoding the 3D point cloud into two-dimensions. This section starts by describing the two implemented encoders; BEV and the PP encoder.

The data representation, created by the encoders, is interpreted with feature extractors in the shape of convolutional layers in different configurations. The neural network configuration depends on what type of decoder is used to produce predictions.

There are mainly three different SOTA 2D object detection methods presented in

Section 3.1 in the shape of one two-stage detector (Faster R-CNN) and two one-stage detectors (YOLO and SSD). A decision was made to focus on the one stage detectors as they promised faster running time in comparison [18]. Specifically, the presented running times are fast enough to be applied in real time. This section further presents the implementation of the specific model architectures in terms of decoders used in the thesis.

4.5.1 Preprocessing

Due to how a LiDAR gathers point cloud data, there are generally fewer points per object the longer the distance from the sensor to the object is. The more empty space implies more unnecessary calculations when trying to extract features. Limits are introduced to the point cloud to counteract the issue with empty space. The point clouds are redefined as only the points within a fixed grid (for instance, 40×40 meters in Figure 4.2). Furthermore, the actual limitation of the point cloud is based on the chosen discretization, as the networks expect the input to be of a specific size according to

$$\text{size} = \frac{\text{point cloud range}}{\text{discretization}}. \tag{4.1}$$

This means that the relationship between the total point cloud range and the discretization must be kept static for a specific input shape. Additionally, points closer than 1m to the sensor are ignored, as these points are generally reflections from the ego vehicle.

As mentioned in Section 4.2.2, the LiDAR captures data in 20Hz while the annotations are applied in 2Hz, meaning that nine additional LiDAR sweeps can be overlayed for every annotation sample. This overlaying is done by translating and rotating the entire point cloud for a specific sweep based on the offset in ego vehicle position and rotation between the sweep and the annotated sample and then adding all the points into the original sample point cloud.

4.5.2 Encoder - Birds Eye View (BEV)

BEV is one of the simplest ways of representing a 3D point cloud densely. It does so by discretizing the ground plane and then represent each grid position by features calculated from the points within that grid. The baseline features for the implemented BEV representation were the same as presented by Simon et al. in their Complex-Yolo paper [40].

The BEV encoder receives a point cloud that is limited in directions x, y, z and has a discretization size of $x_d,\ y_d$. The point cloud is discretized along the ground plane creating a grid of size $\frac{x \text{ range}}{x_d} \times \frac{y \text{ range}}{y_d}$, where there is a set of points $P_{ij} = [x, y, z, r]^T$ for every position ij in the grid. From each set of points within a grid (visualized

as a pillar) three features are calculated:

- *Height*, the height of the highest point inside a grid.
- *Intensity*, the maximum intensity of a point found within a grid.
- *Density*, a normalized expression for the amount of points within a grid.

The three features constitutes the baseline used in the BEV point cloud encoder. The features are calculated for each grid position ij according to

$$\begin{aligned} F_{\text{height}}(i,j) &= \max(P_{ij} \cdot [0,0,1,0]^T) \\ F_{\text{intensity}}(i,j) &= \max(P_{ij} \cdot [0,0,0,1]^T) \\ F_{\text{density}}(i,j) &= \min(1.0,\ \log(\tfrac{N+1}{32})) \qquad N = |P_{ij}|\,, \end{aligned} \tag{4.2}$$

where P_ij is the set of points and N is the number of points within the set. These values are then entered into the grid, creating an image of size $\frac{x\ range}{x_d} \times \frac{y\ range}{y_d} \times 3$ encoding the information from the point cloud.

4.5.3 Encoder - PP

The point pillar encoder is significantly more complex compared to the straight forward BEV approach. The specifics are explained in Section 3.2.3, but in essence, the encoder receives a point cloud that has been limited in the directions x, y, z and has a discretization size of $x_d,\ y_d$.

The ground plane is split based on the discretization into pillars, where two limits are introduced. Firstly, a limit on the number of points within a pillar set to 35. If the number of points exceeds this limit, the points are randomly sampled. Secondly, a limit on pillars set to 10000, and similarly if the number of pillars generated exceeds its limit, the pillars are randomly picked.

Thereafter each point is extended with additional information from the pillars as explained in Section 3.2.3, extending the four original features to nine. This yields a total point cloud representation of size $10000 \times 35 \times 9$, whereas each pillar is sent through a simple neural network called point feature net as presented in Table 4.7.

Layer	Activation function	Regularization	Input size	Output size
Linear layer	ReLU	BatchNorm	35×9	35×64

Table 4.7: The point feature net used to effectively learn the feature representation from the data. Each pillar sized 35×9 is sent through this small neural network yielding an output sized 35×64.

The point feature net applies a linear transformation of the incoming data x as

$$y = xA^T + b, \tag{4.3}$$

where A and b are learnable parameters. These learnable parameters are updated through the loss function specified by the chosen decoders, explained in the following sections. The point feature net transforms each pillar from a point to a feature representation of size 64, yielding a point cloud representation of size $10000 \times 35 \times 64$. Thereafter a max operation is applied to each pillar only keeping the highest value found for each of the 64 features in the pillar, producing a point cloud representation of 10000×64. Finally, each pillar represented by the features is scattered into a grid, based on their original position in the discretized grid. Yielding an encoded point cloud representation as an image of size $\frac{x\ range}{x_d} \times \frac{y\ range}{y_d} \times 64$.

4.5.4 Decoder - YOLOv2

In the YOLOv2 paper [37], they recommend using an input size such that the resulting feature map is of an odd size. As the feature map is divided into a grid, an odd size would yield a prediction in the middle of the input. A center prediction is of interest, as objects are commonly placed in the center of an image. However, when using the point cloud representations as input, the middle of the feature map typically contains the car collecting the data which is neither annotated or interesting, therefore an input of size 448x448 from the original YOLO [36] is used. Since YOLOv2 convolutional layers downstream the input size of a factor of 32, the model yields a feature map with a size of 14×14 to make predictions on.

The entire network architecture of the implementation can be found in Table 4.8 and is based on the architecture from the original YOLOv2 paper [37]. It consists of a downsampling network that produces features from the input image and a detection head. The detection head layers extract features from different levels of the downsampling network to create the feature map.

For each grid position in the produced feature map anchors are placed. As the thesis implementation aims towards predicting the orientation of the objects, every generated anchor is placed twice, once horizontally and once vertically (effectively multiplying the number of anchors by 2). All objects can be rotated 360°, thus by placing two anchors; the maximal direction target offset can be reduced by half.

The targets for the YOLO model to learn, are defined by the offset from the anchors. The targets are yielded by first finding which grid position is closest to each annotated ground truth center. Secondly, the anchors that best resembles the shape of the ground truth bounding boxes is considered the target boxes. Finally, those target boxes are filled with values to offset for the difference between the anchors and the target boxes size, orientation and class.

The final predictions from the YOLOv2 decoder are made in the shape of

$$\#\text{Anchors} \times (\#\text{Classes} + \#\text{Localization targets} + 1). \tag{4.4}$$

This means that predictions are made for every anchor used in each grid position in the resulting feature map sized at 14×14. Each prediction yields a probability for every class, a set of localization offset values and a confidence score visualized in Figure 4.5.

Downsampling:

#: Layer	Filters	Size	Stride	Pad	Activation	Regularization	Input	Output
1: Convolutional	32	3 x 3	1	1	Leaky ReLU	BatchNorm	448x448x(A)	448x448x32
2: Maxpooling		2 x 2	2				448x448x32	224x224x32
3: Convolutional	64	3 x 3	1	1	Leaky ReLU	BatchNorm	224x224x32	224x224x64
4: Maxpooling		2 x 2	2				224x224x64	112x112x64
5: Convolutional	128	3 x 3	1	1	Leaky ReLU	BatchNorm	112x112x64	112x112x128
6: Convolutional	64	1 x 1	1	0	Leaky ReLU	BatchNorm	112x112x128	112x112x64
7: Convolutional	128	3 x 3	1	1	Leaky ReLU	BatchNorm	112x112x64	112x112x128
8: Maxpooling		2 x 2	2				112x112x128	56x56x128
9: Convolutional	256	3 x 3	1	1	Leaky ReLU	BatchNorm	56x56x128	56x56x256
10: Convolutional	128	1 x 1	1	1	Leaky ReLU	BatchNorm	56x56x256	56x56x128
11: Convolutional	256	3 x 3	1	1	Leaky ReLU	BatchNorm	56x56x128	56x56x256
12: Maxpooling		2 x 2	2				56x56x256	28x28x256
13: Convolutional	512	3 x 3	1	1	Leaky ReLU	BatchNorm	28x28x256	28x28x512
14: Convolutional	256	1 x 1	1	1	Leaky ReLU	BatchNorm	28x28x512	28x28x256
15: Convolutional	512	3 x 3	1	1	Leaky ReLU	BatchNorm	28x28x256	28x28x512
16: Convolutional	256	1 x 1	1	1	Leaky ReLU	BatchNorm	28x28x512	28x28x256
17: Convolutional *a*	512	3 x 3	1	1	Leaky ReLU	BatchNorm	28x28x256	28x28x512
18: Maxpooling		2 x 2	2				28x28x512	14x14x512
19: Convolutional	1024	3 x 3	1	1	Leaky ReLU	BatchNorm	14x14x512	14x14x1024
20: Convolutional	512	1 x 1	1	1	Leaky ReLU	BatchNorm	14x14x1024	14x14x512
21: Convolutional	1024	3 x 3	1	1	Leaky ReLU	BatchNorm	14x14x512	14x14x1024
22: Convolutional	512	1 x 1	1	1	Leaky ReLU	BatchNorm	14x14x1024	14x14x512
23: Convolutional	1024	3 x 3	1	1	Leaky ReLU	BatchNorm	14x14x1024	14x14x1024
24: Convolutional	1024	3 x 3	1	1	Leaky ReLU	BatchNorm	14x14x1024	14x14x1024
25: Convolutional *b*	1024	3 x 3	1	1	Leaky ReLU	BatchNorm	14x14x1024	14x14x1024

A = the number of input features

Additional layers:

#: Layer	Filters	Size	Stride	Pad	Activation	Regularization	From	Input	Output
26: Convolutional	64	1 x 1	1	0	Leaky ReLU	BatchNorm	*a*	28x28x512	28x28x64
27: Reshaping *c*								28x28x64	14x14x256

Detection Head:

#: Layer	Filters	Size	Stride	Pad	Activation	Regularization	From	Input	Output
28: Concatenation	1280						*b* & *c*	14x14(256+1024)	14x14x1280
29: Convolutional	1024	3 x 3	1	1	Leaky ReLU	BatchNorm		14x14x1280	14x14x1024
30: Convolutional	O	1 x 1	1	0	None	None		14x14x1024	14x14xO

O = # Anchors $\times(\#Classes + \#Localization\ targets + 1)$

Table 4.8: The full YOLOv2 decoder architecture, presented in three sub-tables. The downsampling and additional layers table present the feature extraction from an input image with A input features. The detection head extracts features from two different levels of the feature extractor to be concatenated into the final feature map to make predictions on, as well as make the actual predictions.

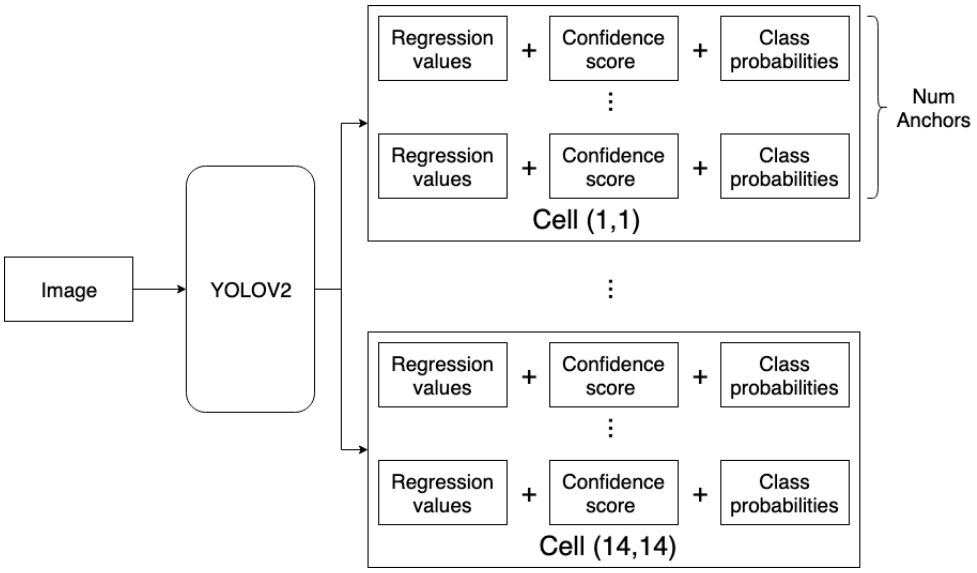


Figure 4.5: In every cell of the feature map, YOLOv2 outputs regression values, confidence score, and class probabilities for every anchor. One class probability is given for every class that the network should classify, and the number of regression values depends on the number of positions the bounding boxes are expressed with. The number of regression values used in 3D object detection for YOLO are seven: center position (x, y, z), width, length, height, and rotation.

Compared to most problems solved with deep learning, object detection answers more than one question, specifically regression and classification within the same network. In the YOLO model, there are three significantly different outputs produced, two classifying and one regressional output.

- The regressional output refers to several values corresponding to the position and direction of the outputted bounding box.
- The confidence score refers to the probability that the specific bounding box center is located within the specific grid position. Essentially a classification score on how confident the network is that the produced bounding box is correct.
- The class probabilities refer to how confident the network is for each class that a specific bounding box should be classified as.

Loss function

A comparison of the produced output with the supposed target is made through the loss function, to find how much the weights in the network should be updated. The loss function used in the YOLOv2 implementation is divided into three smaller loss functions, one for each part of the output.

- *Regressional loss*, that uses the MSE expressed as

$$L_{Reg}(x, y) = \frac{1}{n}\sum_{j}^{n}(x_j - y_j)^2, \tag{4.5}$$

  where x is the produced output for the regressional values for each prediction, y is the target output and n is the total number of predictions.

- *Confidence loss*, also uses the MSE loss according to

$$L_{Cnf}(x, y) = \frac{1}{n}\sum_{j}^{n}(x_j - y_j)^2, \tag{4.6}$$

  where x is the confidence score for the each prediction, y is the target confidence, and n is the number of predictions.

- *Classification loss*, uses the Cross Entropy Loss (CEL) expressed as

$$L_{Cls}(X, y) = \frac{1}{n}\sum_{j}^{n} -\log\frac{e^{X_j[y_j]}}{\sum_{i}^{c} e^{X_j[i]}}, \tag{4.7}$$

  where X is the produced class probabilities for each prediction, y is the target class, c is the total number of classes, and n is the total number of predictions.

The resulting three losses are weighted and summed based on their importance in the network according to

$$L_{Tot} = L_{Reg} + L_{Cnf} + 2.0 \cdot L_{Cls}. \tag{4.8}$$

The doubled weight towards the classification loss is added to try to remedy the imbalance of classes in the dataset.

4.5.5 Decoder - SSD

The original SSD implementation uses a modified VGG16 architecture [27], however most related works within 3D object detection [23], [45], [48] use the same feature extraction network that was originally implemented as a part of the Faster R-CNN implementation[39] by Ren et. al and improved in VoxelNet[48] by Zhou & Tuzel. As the usage of this architecture has been widespread, it became a clear choice as architecture for the SSD implementation in the project.

The resulting model architecture can be expressed as three blocks of fully convolutional layers that downsample the input image. The output from each block is sent through a transposed convolutional layer and are concatenated together to yield the feature map on which the predictions are made. The full architecture for the SSD implementation is presented in Table 4.9, the input and output sizes presented in each layer are based on an input size of 320x400, which was the most common input size used in the project.

Downsampling:

#: Layer	Filters	Size	Stride	Pad	Activation	Regularization	Input	Output
1: Convolutional	64	3 x 3	2	1	ReLU	BatchNorm	320x400x(A)	160x200x64
2: Convolutional	64	3 x 3	1	1	ReLU	BatchNorm	160x200x64	160x200x64
3: Convolutional	64	3 x 3	1	1	ReLU	BatchNorm	160x200x64	160x200x64
4: Convolutional *a*	64	3 x 3	1	1	ReLU	BatchNorm	160x200x64	160x200x64
5: Convolutional	128	3 x 3	2	1	ReLU	BatchNorm	160x200x64	80x100x128
6: Convolutional	128	3 x 3	1	1	ReLU	BatchNorm	80x100x128	80x100x128
7: Convolutional	128	3 x 3	1	1	ReLU	BatchNorm	80x100x128	80x100x128
8: Convolutional	128	3 x 3	1	1	ReLU	BatchNorm	80x100x128	80x100x128
9: Convolutional	128	3 x 3	1	1	ReLU	BatchNorm	80x100x128	80x100x128
10: Convolutional *b*	128	3 x 3	1	1	ReLU	BatchNorm	80x100x128	80x100x128
11: Convolutional	256	3 x 3	2	1	ReLU	BatchNorm	80x100x128	40x50x256
12: Convolutional	256	3 x 3	1	1	ReLU	BatchNorm	40x50x256	40x50x256
13: Convolutional	256	3 x 3	1	1	ReLU	BatchNorm	40x50x256	40x50x256
14: Convolutional	256	3 x 3	1	1	ReLU	BatchNorm	40x50x256	40x50x256
15: Convolutional	256	3 x 3	1	1	ReLU	BatchNorm	40x50x256	40x50x256
16: Convolutional *c*	256	3 x 3	1	1	ReLU	BatchNorm	40x50x256	40x50x256

A = the number of input features

Upsampling:

#: Layer	Filters	Size	Stride	Pad	Activation	Regularization	From	Input	Output
17: Trans Conv *a*	128	1 x 1	1	0	ReLU	BatchNorm	*a*	160x200x64	160x200x128
18: Trans Conv *b*	128	2 x 2	2	0	ReLU	BatchNorm	*b*	80x100x128	160x200x128
19: Trans Conv *c*	128	4 x 4	4	0	ReLU	BatchNorm	*c*	40x50x256	160x200x128

Detection Head:

#: Layer	Filters	Size	Stride	Pad	Act / Reg	From	Input	Output
20: Concatenate *d*	384					*a*, *b*, *c*	160x200x(128×3)	160x200x384
21: Convolutional	(B)	1x1	1	0	None	*d*	160x200x384	160x200x (B)
22: Convolutional	(C)	1x1	1	0	None	*d*	160x200x384	160x200x (C)
23: Convolutional	(D)	1x1	1	0	None	*d*	160x200x384	160x200x (D)

Layer 21 predicts the classification; B = # Classes
Layer 22 predicts the regression values; C = # Anchors ×*#Regression targets*
Layer 23 predicts the direction classification; D = # Anchors ×*#Rotation classes*

Table 4.9: The full SSD decoder architecture, presented in three sub-tables. The downsampling layers applies feature extraction on the input image with A features. The upsampling layers extract features from different levels in the downsampling network and concatenate them, producing a feature map that the detection head uses to predict the classification and localization of the objects.

The final predictions produced from the SSD detector is in the form

$$\#\text{Anchors} \times (\#\text{Classes} + \#\text{Regression targets} + \#\text{Rotation classes}), \tag{4.9}$$

where every anchor placement is subject to three different types of predictions. Firstly in terms of class, what the object actually is. Secondly, the regression, to predict the (3D) bounding box in terms of position, size, and angle. Lastly, a rotation classification, to determine the heading of the object within the anchor.

These predictions are made for every position in the resulting feature map sized 160x200 (assuming an input of 320x400) yielding the output as described in Figure 4.6. This means that the SSD implementation yields 32 000 placements per anchor. Considering the relatively small space that each of the feature map grids occupies, with a point cloud as input, one could expect that many of the anchor locations do not contain any points. Therefore additional masking of anchors is implemented, to ignore all anchor placements that are made on locations that do not contain any points in the original point cloud, significantly reducing the effective number of total predictions.

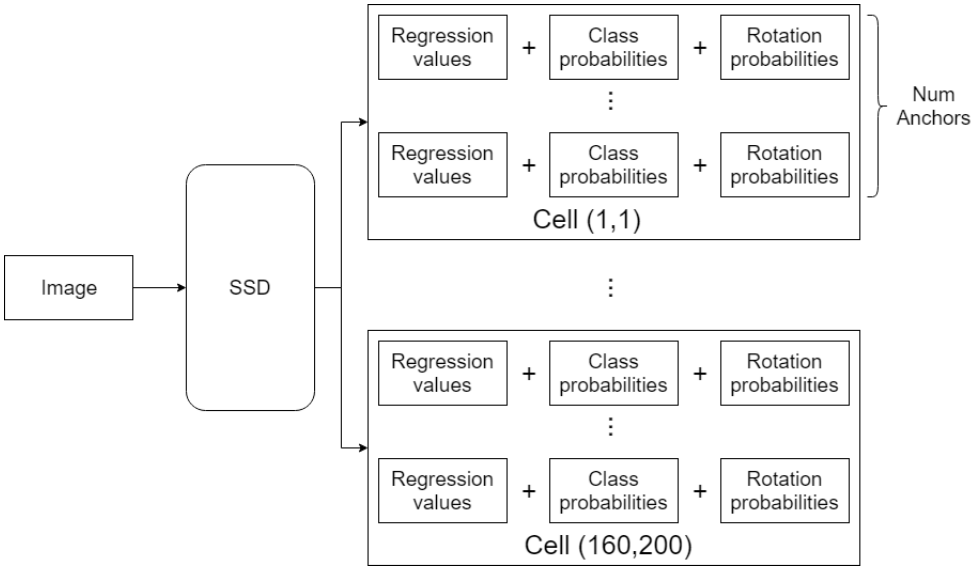


Figure 4.6: In every cell of the produced feature map, SSD outputs regression values, class probabilities, and rotational probabilities for every anchor. One class probability is given for every defined object class, one rotational probability is given for every defined rotational class, and the number of regression values depends on the number of positions the bounding boxes are expressed with. The number of regression values used in 3D object detection are seven: center position (x, y, z), width, length, height, and rotation.

Even though the anchor masking helps to reduce the number of predictions made, there is still an issue present regarding background to class imbalance. To solve this issue the original implementation of SSD[27] uses hard negative mining as explained in Section 2.2.5.

Another approach to solving the background to class imbalance was introduced in the paper presented by Lin et al. [26]. The approach evolved around a newly introduced loss function called *focal loss*. It is a variation of the standard CEL function, where the loss for the predictions with a high probability is reduced. Thus focal loss makes the network update the weights more for the harder predictions, and less for the obvious predictions, such as common background.

Loss function

The SSD detector produces three types of output, divided into localization, classification, and directional classification.

- *Classification loss*, uses the aforementioned focal loss to try to solve the class imbalance issue. The loss is expressed as

$$L_{cls}(X, y) = \sum_{j}^{n} \begin{cases} -\alpha \cdot ((1 - \sigma(X_j))^{\gamma} - log(\sigma(X_j)) & if\ y = X_j \\ -(1 - \alpha) \cdot (\sigma(X_j))^{\gamma} - log(1 - \sigma(X_j)) & otherwise \end{cases}, \tag{4.10}$$

  where $\sigma()$ denotes the sigmoid function, X is the predicted class probabilities, y is the supposed class, n is the number of classes, and γ & α are both free variables. Considering the sheer dominance of background examples in comparison to actual objects in the nuScenes dataset, the variables $\alpha = 0.25$ and $\gamma = 2$ is used, as recommended by Lin et al.

- *Localization loss*, uses the following smooth l_1 loss, introduced in the Fast R-CNN implementation [14],

$$L_{reg}(X, Y) = \sum_{j}^{n} \begin{cases} 0.5 \cdot (X_j - Y_j)^2 & if\ |X_j - Y_j| < 1 \\ |X_j - Y_j| - 0.5 & if\ |X_j - Y_j|\ \geq 1 \end{cases}, \tag{4.11}$$

  where X is the predicted regressional values, Y is the target output and n is the number of regressional targets.

- *Directional loss*, is handled twice, once within the localization loss and once as a classification. The directional classification is done to be able to differentiate objects of opposite heading. Each anchor is placed twice, once horizontally and once vertically. The direction as a regression target handles the offset from the initial rotation (maximum of 45°), whilst the direction as a classification handles which way an object is facing (± 180°).

  The issue with generating targets based on IoU between the anchor and the ground truth, is that it does not take the heading of the ground truth into

account. Because there is no difference between IoU for both headings, the directional classification is added as a target. The directional classification loss is calculated by using CEL expressed as

$$L_{dir}(X, y) = \frac{1}{n}\sum_{j}^{n} -\log \frac{e^{X_j[y_j]}}{\sum_i^c e^{X_j[i]}}, \tag{4.12}$$

where X is the predicted class probabilities, y is the supposed direction, $c(=2)$ is the number of directions within each anchor, and n is the number of predictions.

Furthermore, the total loss of the network is the weighted sum of each of the three individual losses. The factors purpose is to focus the networks updating its weights, based on the importance of the different categories. The total loss is expressed as

$$L_{tot} = 2.0 \cdot L_{reg} + 1.0 \cdot L_{cls} + 0.2 \cdot L_{dir}, \tag{4.13}$$

where the regression loss is multiplied by 2, the classification loss is multiplied by 1, and the direction loss is multiplied by 0.2. The specifically chosen weights are based on the previous implementations using a similar loss implementation [45] [23].

4.6 Training

This section describes the hyperparameters and hardware used when training the implemented models.

4.6.1 Initial Training

Since the full dataset was not released as the thesis began, the initial training was done on the teaser dataset, to show that the implemented models were, in fact, capable of learning the proposed task of object detection. Due to the objective of this experiment was to solely verify the learning capability, no validation or test set was used. Thus, the initial models were trained and evaluated on the same dataset, meaning that the results from the evaluations presented in Section 5.1 is based on previously seen data. The evaluations are therefore skewed, as the models are possibly overfitted towards the same data used for evaluation.

To note, is that if the teaser dataset had been the only source of data in the thesis, a split into training, validation and test sets would have been performed on the teaser dataset. However, this was not the case considering that the full dataset (with a predefined split) was released in time.

4.6.2 Hyperparameters

Hyperparameters are the values which are set on a neural network model before training. Hyperparameter optimization can be a tricky task, it is an iterative process as it requires training the networks to analyze the used hyperparameters. This

is especially tricky due to the implemented networks being complex, as it takes a significant amount of time to train different configurations, limiting the number of hyperparameters that can be explored. In an effort to save time while exploring the hyperparameters, models were simplified such as changed to detect a single class, reduced point cloud range, and increased discretization. Visdom was used to catch to misleading experiments in earlier stages through the plotted losses.

Each implemented model used a batch size of two, as a bigger batch size would require more memory than available in the used GPUs. The networks used the Adam optimizer, as it has shown great promise for training deep neural networks, and as it simplifies the learning rate tuning. Some additional setting that varied between the implementations was:

- *Classes*, was set to either the eight classes defined in Section 4.3 or only the car class.
- *Anchors*, depending on the class configurations different anchor boxes were used.
- *Discretization* defines the quantity of information sent to the network in the form of the encoded point cloud.
- *Point cloud range*, was changed depending on the discretization, such that the networks receive a functioning input size.
- *Inital learning rate*, was configured in attempts to find a global minimum in the loss functions.
- *Milestones* was introduced to force the learning rate to decay every set number of epochs, in attempts to find the global minimum in the loss function.
- The *number of sweeps*, was configured to explore the effects of having a richer point cloud as input.
- *Early stopping*, is used to stop training after a set amount of epochs if the validation loss has not decreased.

4.6.3 Hardware

As various networks with different configurations were trained, using multiple machines enabled time efficiency by parallelization. The machines at our disposal were:

- Machine one had an Intel Core i7 CPU, and a GTX 970 GPU with 8GB video memory.

- Machine two had an Intel Xeon CPU, and a Nvidia k5200 GPU with 8 GB video memory.

- Machine three and four were rented from Google Cloud Platform containing a Nvidia Tesla k80 with 12GB video memory, and an Intel Xeon CPUs.

All the models' inference time presented in Chapter 5 was measured on machine two to get a comparable measurement between the models. Furthermore, the training times were measured with Machine one, by averaging the time spent in different parts of the training phase in an entire epoch.

4.7 Postprocessing

Object detection models commonly output plenty of predictions. The objective of the postprocessing stage is to remove redundant predictions and is done by using NMS, as explained in Section 2.2.5. The NMS contains two thresholds, a confidence threshold, and an IoU threshold. The confidence threshold is set differently depending on how confident a model is in its predictions. Generally, the threshold refers to how many predictions that should be kept. The IoU threshold is applied to remove boxes that supposedly refer to the same object, and should be tuned depending on the nature of the outputted predictions of a model.

4.8 Visualization

The nuScenes data contains several different positioning systems with different origins and axle orientations. As the thesis is based on the LiDAR point cloud data, this is considered the positional system in which the models make predictions. When visualization (or the official evaluation) is made, the predictions must be translated into the positional system of interest. This translation was done with simple affine transformations based on the calibration information for each positional system, containing the required rotation and translation to move from one positional system to the other.

4.9 Evaluation

As mentioned in Section 4.2 nuScenes provide their own evaluation service. However, the number of evaluations to be performed on this service was limited. Therefore, a separate evaluation method was implemented to find the best performing model to then send to the official evaluation service. This section describes how inference time and accuracy were calculated for the implemented evaluation method.

The postprocessing stage used during the evaluation had no probability threshold, and the IoU NMS threshold was set to 0.25 for the models that used the PP encoder and 0.1 for the models that used the BEV encoder.

4.9.1 Inference Time

The inference time is the time it takes for the model to produce its output from an input. More specifically, it is the time from when the model receives the point cloud to when the final predictions are complete. The steps performed in the inference time evaluation is divided into:

- *Encoding*, the time it takes to create the encoded representation from the point cloud input.
- *Forward propagation*, the time it takes to forward propagate the encoded point cloud representation through the model.
- *Postprocessing*, the time spent in the postprocessing stage.

The inference time was calculated by taking the average time it took to process the samples in the validation set, while evaluating the system. The first ten samples was disregarded in the inference time calculation, to avoid the start-up-time of the model.

4.9.2 Accuracy Measurement

To find the best performing model two different evaluation methods were implemented. One method evaluating the average precision of only cars at three different ranges from the ego vehicle, and four different difficulties defined by the visibility parameter mentioned in Section 4.2.2. The other method evaluates the average precision for the eight classes mentioned in Table 4.3. Each evaluation method is performed on the prediction after the postprocessing stage has been applied. The implementation limits the range of the evaluated prediction to 35 meters from the ego vehicle.

As discussed in Section 2.6 there exist different thresholds that define a true positive prediction; this implementation has the option of choosing either distance or IoU as the TPM. With distance as the TPM, a true positive is when the distance between the centers of the prediction and ground truth is less than two meters apart. With IoU as the TPM, a true positive is when the IoU between the prediction and the ground truth is higher than 0.3.

The evaluation is divided into two steps. In the first step, the predicted bounding boxes from the validation set are stored separately depending on their predicted class. The predictions are sorted in descending order of their predicted probability.

The second step iterates over all the predictions for each class, starting with the prediction with the highest confidence. The prediction is compared to all the ground truth bounding boxes for the specific sample. If the prediction is considered a true positive match by the TPM, then the ground truth is removed to ensure that it can not be matched again. All predictions not fulfilling the TPM are set to false positives. When the entire validation set has been processed, the average precision is calculated by taking the area under the precision-recall curve for each class as mentioned in Section 2.6. Finally, the mAP is calculated by taking the mean of the achieved average precisions for each class.

5

Results

In this thesis, two different detectors were trained - SSD and YOLOv2. SSD was trained with two different point cloud encoders PointPillar (PP) and the BEV representation, while YOLOv2 was solely trained on the BEV representation (PP was skipped due to the achieved results). This chapter starts by presenting the results for the aforementioned models trained and evaluated on the teaser dataset. Thereafter sections are presented for each SSD model trained on the full dataset, with differing hyperparameters. Each model section presents the specific parameters used, and the resulting evaluation scores. Finally, the chapter ends with a comparison between all the different models, where the best-resulting model is evaluated by nuScenes evaluation service

5.1 Teaser Dataset

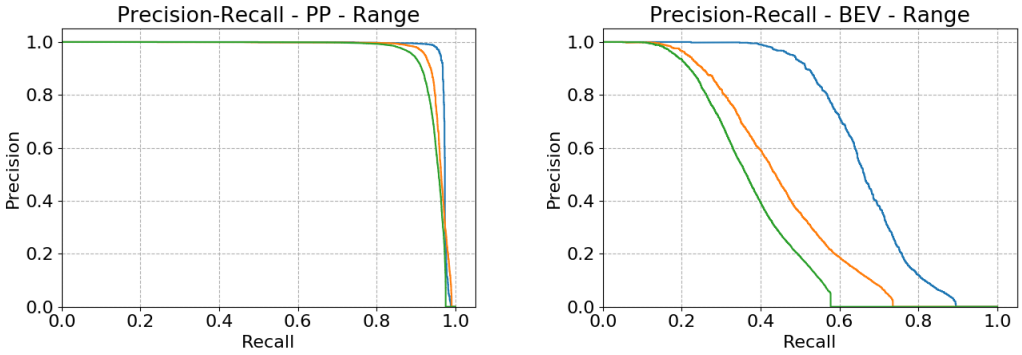
The models presented under this section were trained on the entire nuScenes teaser dataset. Thus no validation or test set was used as more intricately explained in Section 4.6.1. All models presented here were trained for roughly 48 hours, resulting in approximately 90 epochs on the teaser dataset, corresponding to around the same amount of iterations as training 12 epochs on the training set from the full release. All evaluations presented were performed with a TPM of two meters in distance.

5.1.1 SSD

The SSD model were trained with both the PP and BEV point cloud encoders, this was done with the hyperparameters presented in Table 5.1. Furthermore, the model trained on the PP encoder achieved an AP of 94.8% for the car class, and conversly the model trained on BEV achieved an AP of 37.2%. The precision-recall curves for both models based on range and occlusion, along with tables of the corresponding AP is presented in Figures 5.1 and 5.2. Additionally, the inference times of the implemented models are shown in Table 5.2.

Hyperparameters	
Parameter	**Value**
Classes	Only Cars
Number of sweeps	1
Discretization	0.16m × 0.16m
Point cloud range	±34.56m × ±39.68m
Final input size	432 × 496
Initial learning rate	0.0002
Milestones	Every 15th epoch
Anchor (in meters)	[1.6, 3.9, 1.56]*

Table 5.1: Hyperparameters used for training the SSD model on both PP and BEV. * The anchor used was taken from the original official PP implementation[23].

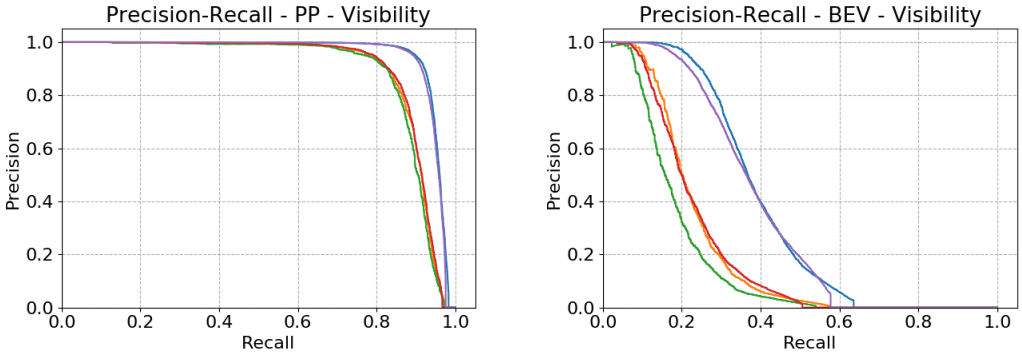


Range	PP AP	BEV AP
≤ 10m	97.1%	66.4%
> 10m and ≤ 20m	96.0%	44.8%
≤ 35m	**94.8%**	**37.2%**

Figure 5.1: The precision-recall curve based on range for the SSD decoder using both the PP and BEV encoders. The training is performed on the teaser data with the hyperparameters presented in Table 5.1, and the evaluation is split based upon the distance from the ego vehicle as shown in the above table.

Inference time		
Section	**PP runtime**	**BEV runtime**
Encoding	16.1ms	19.1ms
Decoding	4.7ms	5.7ms
Postprocessing	75.5ms	67.1ms
Total	96.3ms	91.9ms
In Hertz	**10.4Hz**	**10.9Hz**

Table 5.2: The inference time for the two SSD models using PP and BEV, trained with the hyperparameters presented in Table 5.1. The postprocessing stage used no confidence threshold, and the IoU threshold was set to 0.25 and 0.1 for PP and BEV respectively.



Visibility Level	PP AP	BEV AP
4	95.2%	38.5%
3	89.6%	22.5%
2	88.9%	18.2%
1	90.0%	22.4%
Total AP	**94.8%**	**37.2%**

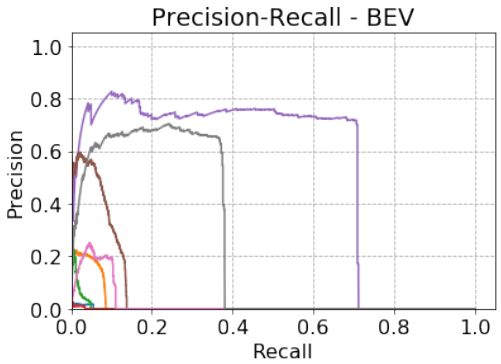
Figure 5.2: Precision-recall curve based on the visibility for the two SSD models using PP and BEV, trained with the hyperparameters in Table 5.1. The models were trained and evaluated on the entire nuScenes teaser dataset, where the evaluation was split based on the annotated objects visibility level. The corresponding AP for each visibility level is presented in the table.

5.1.2 YOLOv2 on BEV

The model using the YOLOv2 decoder was only trained using the BEV point cloud encoder, with the hyperparameters presented in Table 5.3. The model managed to achieve a mAP of 16.8% on all the merged classes with an AP of 1.6% on the car class. The Precision-recall curve and a table with the corresponding mAP for the model is presented in Figure 5.3. Furthermore, the inference time from running the evaluation is presented in Table 5.4.

Hyperparameters	
Parameter	**Value**
Classes	All merged classes
Number of sweeps	1
Discretization	0.178m × 0.178m
Point cloud range	±40m × ±40m
Final input size	448 × 448
Initial learning rate	0.0002
Milestones	Every 15th epoch
Anchors	5 avg teaser set*

Table 5.3: Hyperparameters used for the implemented YOLOv2 model using the BEV point cloud representation. *The anchors are from taking average sizes of the five most occurring objects in the teaser dataset. Note that the discretization and point cloud range is tuned to achieve the input size required for the implemented YOLOv2 decoder.



Class	AP
Pedestrain	0.1%
Car	1.6%
Motorcycle	0.5%
Bicycle	0.0%
Bus	69.6%
Truck	6.1%
Construction	2.4%
Trailer	54.4%
Total mAP	**16.8%**

Figure 5.3: Precision Recall curve and mAP scores for the YOLOv2 model using the BEV point cloud encoder. Both trained and evaluated on the entire nuScenes teaser dataset, with the true positive metric set to a distance of 2 meters.

Inference time	
Section	**BEV runtime**
Encoding	11.6ms
Decoding	5.0ms
Postprocessing	77.0ms
Total	93.6ms
In Hertz	**10.7Hz**

Table 5.4: The inference time for the the YOLOv2 model using the BEV encoder, trained with the hyperparameters presented in Table 5.3. The postprocessing stage used no confidence threshold, and the IoU threshold was set to 0.1.

5.2 Full dataset

Considering the poor performance of the implemented YOLO models, all networks presented in this section were trained using the SSD detector on both the PP and the BEV point cloud encoders. The networks were trained on the training set containing approximately 28000 samples and validated with the validation set of 6000 samples from the full dataset. The full dataset also includes a test set, however as the annotations for this set are not publicly available the evaluations in this section are performed on the validation set. All evaluation presented in this section were performed with a TPM of two meters in distance, and the postprocessing settings presented in Section 4.9.

5.2.1 Baseline

The initial training for the models on the full dataset was done with the hyperparameters presented in Table 5.5 with the training and validation losses presented in Figure 5.4. The only differing parameter between the two was the number of epochs required for early stopping; it was set to 10 for PP and 25 for BEV.

The average time it took to train each iteration for SSD on both PP and BEV is presented in Table 5.6. SSD on PP trained for 30 epochs with the lowest validation loss achieved being 2.067 at epoch 20. SSD on BEV trained for 41 epochs with the lowest validation loss of 3.408 at epoch 26.

Hyperparameters	
Parameter	**Value**
Classes	All merged classes
Number of sweeps	1
Discretization	0.25m × 0.25m
Point cloud range	±40m × ±50m
Final input size	320 × 400
Initial learning rate	0.0002
Milestones	Every 15th epoch
Early stopping	10 / 25
Anchors	3rd entry in Table 4.6 (k-means)

Table 5.5: Baseline hyperparameters used when training SSD on both PP and BEV on the full training set.

Average Training Time		
Section	**PP runtime**	**BEV runtime**
Dataloader:	244ms	232ms
- Load point clouds from files	141ms	115ms
- Split point clouds	52ms	73ms
- Generate targets	51ms	44ms
Load into GPU	8ms	9ms
Forward propagation	54ms	20ms
- Generate features from pillars	8ms	—
- Generate image from features	24ms	—
- Image to prediction	22ms	20ms
Calculating loss	41ms	37ms
Backpropagation	161ms	40ms
Update step	106ms	92ms
Total	**614ms**	**430ms**

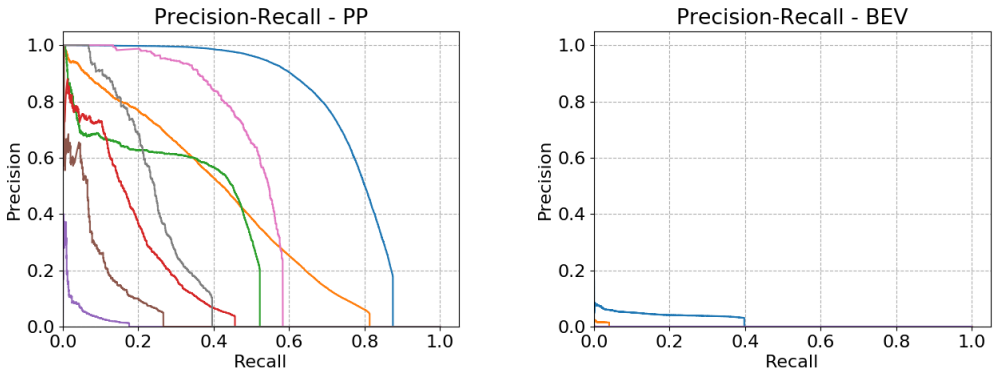
Table 5.6: Average training time per iteration for SSD on PP and BEV split into sections. The time is calculated by training for one full epoch and average the time spent in each section in the program per iteration. The actual training was performed with the parameters found in Table 5.5, with a batch size of two.

Figure 5.4: Training and validation loss per epoch for SSD on PP and BEV during training. The models were trained with the baseline hyperparameters presented in Table 5.5, where the early stopping is set to 10 and 25 respectively for the PP and BEV models.

The SSD model was trained with the PP and BEV encoders with the baseline parameters presented in Table 5.5. The model trained on PP achieved a mAP of 31.2% for all classes, and an AP of 76.7% for cars. The model trained on BEV achieved a mAP of 0.2% for all classes, and an AP for cars at 1.6%. The Precision-recall curve with the corresponding mAP scores for both models is presented in Figure 5.5, and the inference time for the models is shown in Table 5.7.

Inference time		
Section	**PP runtime**	**BEV runtime**
Encoding	17.9ms	15.6ms
Decoding	4.6ms	4.2ms
Postprocessing	75.5ms	56.8ms
Total	98.0ms	76.6ms
In Hertz	**10.2Hz**	**13.1Hz**

Table 5.7: The inference time for the two SSD models using PP and BEV, trained with the hyperparameters presented in Table 5.5. The postprocessing stage used no confidence threshold, and the IoU threshold was set to 0.25 and 0.1 for PP and BEV respectively.



Class	PP AP	BEV AP
Car	76.7%	1.6%
Pedestrain	41.0%	0.3%
Truck	31.6%	0.0%
Motorcycle	17.1%	0.0%
Bicycle	1.1%	0.0%
Construction	7.4%	0.0%
Bus	50.3%	0.0%
Trailer	24.1%	0.0%
Total mAP	**31.2%**	**0.2%**

Figure 5.5: Precision-recall curve and mAP scores for SSD trained on PP and BEV. The models were trained and evaluated on the full dataset with the baseline hyperparameter presented in Table 5.5. The TPM was set to a distance of 2 meters, and only predictions within a distance of 35m from the ego vehicle is handled.

5.2.2 Learning Rate Experiments

Two different types of networks with varying initial learning rate and milestone frequency were trained. The first type was trained with more milestones (MM), and the second type was trained with more milestones and lower initial learning rate (MMLR). No improvement in terms of mAP was achieved in comparison to the baseline models.

The networks under this category were trained using the same hyperparameters as the baseline models, presented in Table 5.5, with exception for the initial learning rate and the milestone frequency. The learning rate was decreased by a factor of 20, and the milestones were changed to occur every other epoch instead of every 15th. The best performing BEV based SSD model trained for 30 epochs and achieved a lowest validation loss of 3.581 at epoch 20. The best performing PP based SSD model trained for 31 epochs and achieved a lowest loss of 2.378 at epoch 21. The training and validation losses per epoch are presented in Figure 5.6.

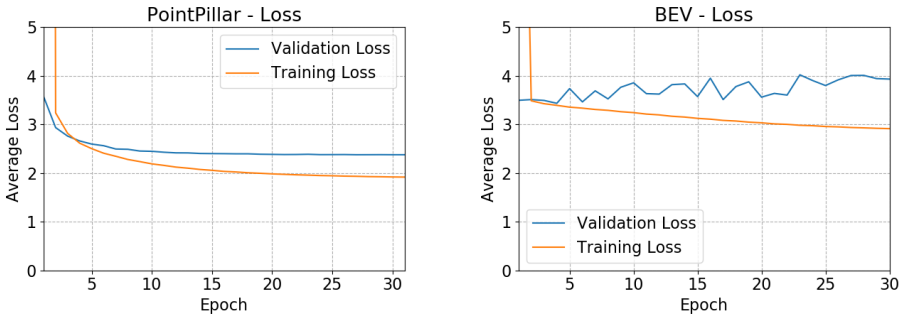
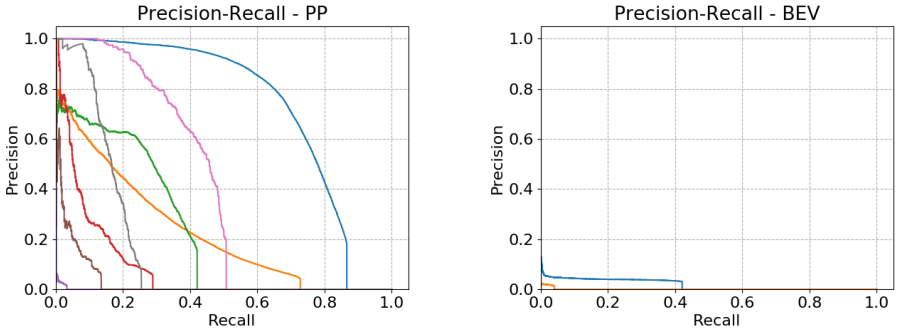


Figure 5.6: Training and validation loss per epoch for SSD on both PP and BEV trained with lower learning rate and more milestones in comparison to the baseline parameters presented in Table 5.5.

The best performing model based on PP achieved a mAP of 23.6% for all classes, and an average precision for cars at 73.6%. The best performing model based on BEV achieved a mAP of 0.2% for all classes, and an average precision for cars at 1.3%. The Precision-recall curve for both models and a table of the mAP score for all models under this category is presented in Figure 5.7. The two additional models presented in the table below used the same learning rate as the baseline hyperparameters, but with solely the above mentioned milestone frequency change.



Class	PP AP MMLR	PP AP MM	BEV AP MMLR	BEV AP MM
Car	73.5%	69.8%	1.3%	1.2%
Pedestrain	22.6%	36.6%	0.0%	0.0%
Truck	23.2%	27.2%	0.0%	0.0%
Motorcycle	8.6%	10.4%	0.0%	0.0%
Bicycle	0.1%	0.6%	0.0%	0.0%
Construction	3.1%	1.0%	0.0%	0.0%
Bus	40.9%	40.1%	0.0%	0.0%
Trailer	16.7%	23.3%	0.0%	0.0%
Total mAP	**23.6%**	**26.1%**	**0.2%**	**0.2%**

Figure 5.7: Precision recall curve and mAP for SSD trained on the PP and BEV encoder, with more milestones (MM), and both more milestones and lower learning rate (MMLR) compared to the baseline parameters presented in Section 5.2.1

5.2.3 Varying Sweeps for One Class

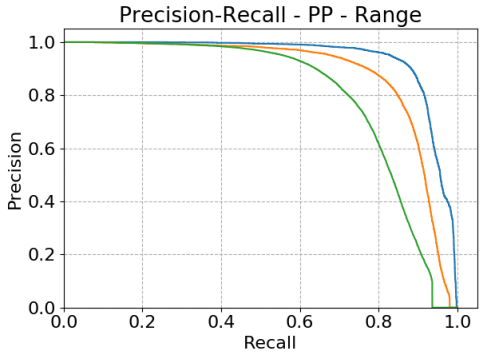
In this experiment the models were limited to only detect cars (to reduce the training time), they used the same hyperparameters as presented in Table 5.5, except that they used a varying amount of LiDAR sweeps per sample. Three different SSD models based on the PP encoder were trained with 1, 5, and 10 sweeps respectively. Two SSD models based on the BEV encoder were trained with 1 and 5 sweeps respectively.

The model using the PP encoder that achieved the lowest validation loss used ten sweeps per sample. It trained for 34 epochs and achieved a total average loss of 1.050 on the validation set. Conversely, the model using the BEV encoder achieved the lowest validation loss when using one sweep per sample with a final average loss of 1.981 on the validation set in 19 epochs before early stopping. In Table 5.8 the training time for the different networks are presented.

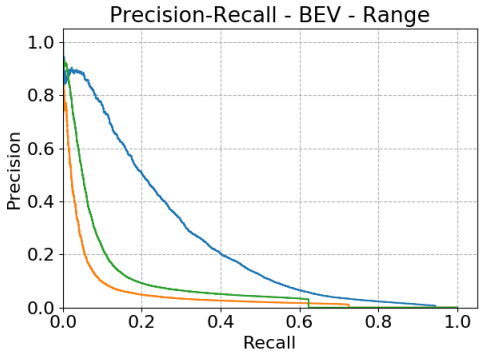
Average Training Time					
Section	**PP1S**	**PP5S**	**PP10S**	**BEV1S**	**BEV5S**
Dataloader	220ms	914ms	1711ms	174ms	940ms
- Load point clouds from files	132ms	650ms	1218ms	105ms	615ms
- Split point clouds	76ms	253ms	482ms	60ms	315ms
- Generate targets	12ms	11ms	11ms	9ms	10ms
Load into GPU	4ms	14ms	15ms	7ms	7ms
Forward propagation	45ms	50ms	51ms	21ms	20ms
- Generate features from pillars	9ms	6ms	6ms	—	—
- Generate image from features	16ms	25ms	26ms	—	—
- Image to prediction	20ms	19ms	19ms	21ms	20ms
Calculating loss	38ms	38ms	38ms	34ms	34ms
Backpropagation	112ms	112ms	112ms	40ms	40ms
Update step	95ms	95ms	95ms	90ms	90ms
Total	**514ms**	**1178ms**	**1927ms**	**366ms**	**1131ms**

Table 5.8: Average training time per iteration for SSD on PP and BEV, using a varying amount of point cloud sweeps. The time is calculated by averaging the time spent in each section during training for one full epoch.

The best performing PP based model according to the evaluation used five sweeps per sample and achieved an AP of 80.3% on the car class. The best performing BEV based model used one sweep and achieved an AP of 9.0%. The precision-recall curve from the best performing model for each encoder presented by range in Figure 5.8 and visibility in Figure 5.9. The inference time for all models trained in this section is presented in Table 5.9.



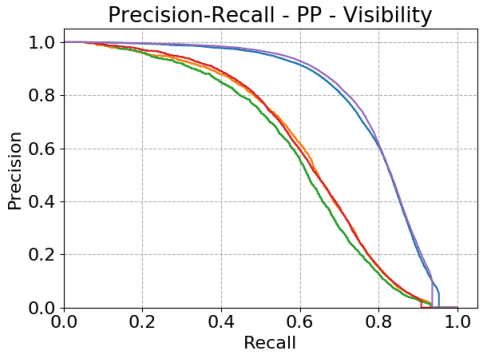
(a) PP 5 sweeps Range



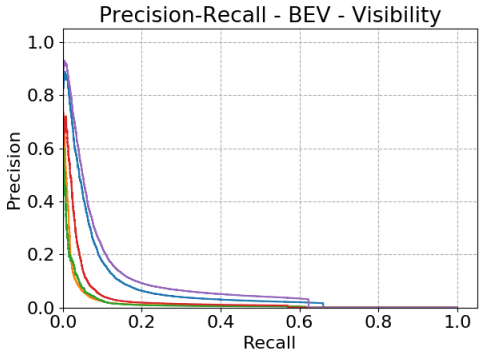
(b) BEV 1 sweep Range

Range	PP1S AP	PP5S AP	PP10S AP	BEV1S AP	BEV5S AP
$\leq$ 10m	95.4%	94.2%	94.0%	25.1%	6.0%
$>$ 10m and $\leq$ 20m	88.1%	88.4%	86.4%	5.2%	1.0%
$\leq$ 35m	**79.4%**	**80.3%**	**79.8%**	**9.0%**	**2.6%**

Figure 5.8: Precision-recall curve based on range for the SSD models using PP and BEV. Trained and evaluated with a varying amount of sweeps as input. The evaluation was split based on the objects annotated range from the ego vehicle. The corresponding AP for each of the different ranges is presented in the table.



(a) PP 5 sweeps Visibility



(b) BEV 1 sweep Visibility

Visibility Level	PP1S AP	PP5S AP	PP10S AP	BEV1S AP	BEV5S AP
4	79.7%	79.5%	78.2%	7.3%	1.8%
3	61.8%	62.1%	60.1%	1.9%	0.4%
2	58.0%	59.6%	56.0%	1.7%	0.3%
1	60.4%	62.3%	60.2%	3.3%	0.7%
Total AP	**79.4%**	**80.3%**	**79.8%**	**9.0%**	**2.6%**

Figure 5.9: Precision-recall curve based on visibility for the SSD models using PP and BEV. Trained and evaluated with a varying amount of point cloud sweeps as input. The evaluation was split based on the objects annotated visibility level. The corresponding AP for each the different visibility is presented in the table.

Inference time					
Section	**PP1S**	**PP5S**	**PP10S**	**BEV1S**	**BEV5S**
Encoding	17.6ms	60.3ms	109.3ms	12.6ms	67.1ms
Decoding	4.2ms	4.1ms	4.1ms	4.2ms	4.2ms
Postprocessing	75.5ms	48.9ms	50.7ms	36.7ms	48.6ms
Total	97.3ms	113.3ms	164.1ms	53.5ms	129.9ms
In Hertz	**10.2Hz**	**8.8Hz**	**6.0Hz**	**18.7Hz**	**7.7Hz**

Table 5.9: The inference time for the SSD models using PP and BEV, trained with a varying amount of LiDAR sweeps per input point cloud. The postprocessing stage used no confidence threshold, and the IoU threshold was set to 0.25 and 0.1 for PP and BEV respectively.

5.2.4 Smaller Discretization for One Class

In this section the models were also limited to only detect cars, the objective was to explore how the discretization effect the accuracy. One PP based and one BEV based model were trained with the hyperparameters presented in Table 5.10.

The PP based model trained for 26 epochs and achieved a lowest validation loss of 0.8165 at epoch 16. The BEV based model trained for 21 epochs and achieved a lowest loss of 1.708 at the 11th epoch. The average training time for both models is presented in Table 5.11. Note that the losses is not comparable to other implementations considering the unique point cloud range, as the loss is only based on objects that are located within the range.

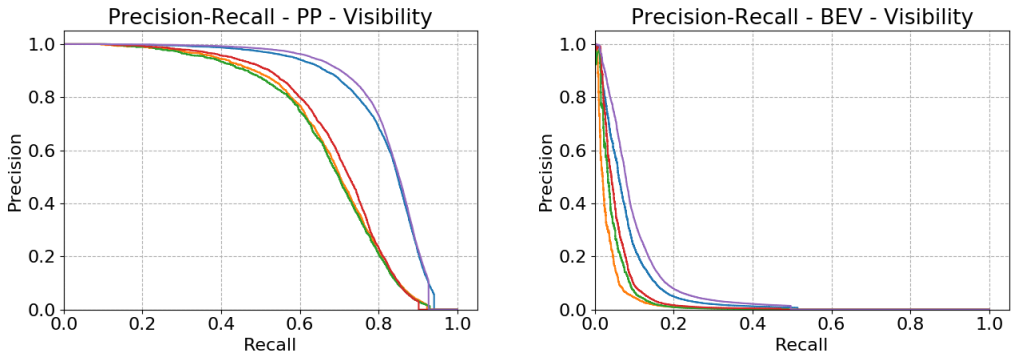
Hyperparameters	
Parameter	**Value**
Classes	Only Cars
Number of sweeps	1
Discretization	0.16m $\times$ 0.16m
Point cloud range	$\pm$25.6m $\times$ $\pm$32m
Final input size	320 $\times$ 400
Initial learning rate	0.0002
Milestones	Every 15th epoch
Early stopping	10
Anchor	Average Car size in Table 4.5

Table 5.10: Table of hyperparameters for SSD on PP and BEV with reduced grid size.

The PP based network achieved an AP of 82.7% on cars, while the BEV based network achieved an AP of 9.8%. In Figure 5.10 and Figure 5.11, the precision-recall curve and AP based on visibility and range is shown respectively. The models inference times is presented in Table 5.12.

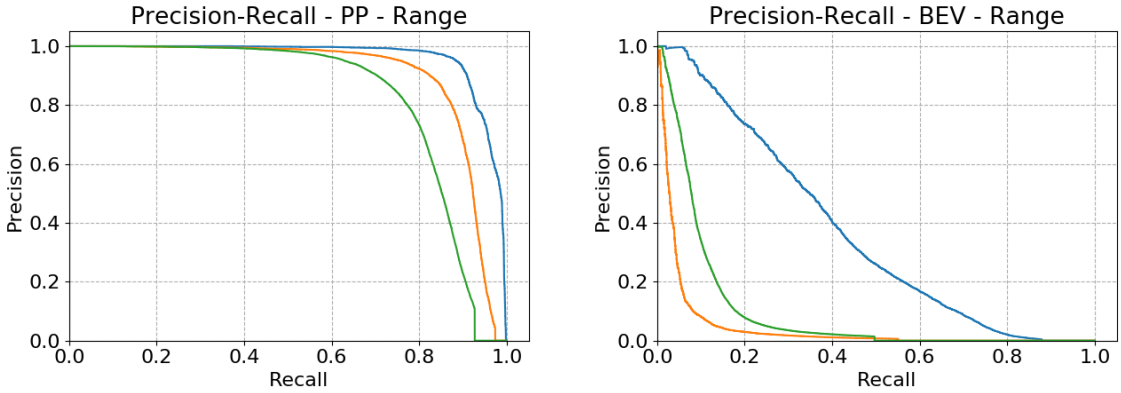
Average Training Time		
Section	**PP**	**BEV**
Dataloader	176ms	172ms
- Load point clouds from files	110ms	103ms
- Split point clouds	51ms	50ms
- Generate targets	15ms	19ms
Load into GPU	5ms	7ms
Forward propagation	51ms	22ms
- Generate features from pillars	8ms	—
- Generate image from features	21ms	—
- Image to prediction	22ms	22ms
Calculating loss	38ms	34ms
Backpropagation	116ms	42ms
Update step	103ms	94ms
Total	**489ms**	**371ms**

Table 5.11: Average training time per iteration for SSD on PP and BEV, trained with a smaller grid size. The training was performed with a batch size of two with the hyperparameters presented in Table 5.10.



Visibility level	PP AP	BEV AP
4	81.6%	7.8%
3	68.2%	3.2%
2	67.4%	4.5%
1	69.8%	5.5%
Total	**82.7%**	**9.8%**

Figure 5.10: The precision recall curve based on visibility for the two SSD models using the PP and BEV encoders. The models were trained with the parameters presented in Table 5.10. The evaluation was split based on the objects annotated visibility level, their respective AP can be seen in the table.



Range	PP AP	BEV AP
$\leq$ 10m	96.3%	36.8%
$>$ 10m and $\leq$ 20m	90.0%	4.6%
$\leq$ 35m	**82.7%**	**9.8%**

Figure 5.11: The precision recall curve based on range for the two SSD models using the PP and BEV encoders. The models were trained with the parameters presented in Table 5.10. The evaluation was split based on the objects annotated range from the ego vehicle, and the respective AP can be seen in the table.

Inference time		
Section	**PP time**	**BEV time**
Encoding	17.0ms	10.6ms
Decoding	4.7ms	4.4ms
Postprocessing	46.9ms	44.2ms
Total	68.7ms	59.2ms
In Hertz	**14.6Hz**	**16.9Hz**

Table 5.12: The inference time for the SSD models using PP and BEV, trained with a smaller discretization size. The postprocessing stage used no confidence threshold, and the IoU threshold was set to 0.25 and 0.1 for PP and BEV respectively.

To further strengthen the results, an example of bounding boxes from the SSD-PP model for a point cloud from the test set is visualized through projection into the different camera images in Figure 5.12. The shown boxes have a postprocessing confidence above 30% and an IoU overlapping to other boxes below 0.25.

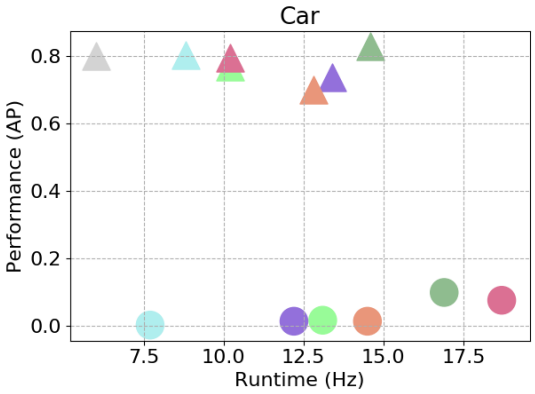
Figure 5.12: Produced bounding boxes from a LiDAR point cloud, with a confidence score above 30% and an IoU overlapping below 0.25. Note that the model does not use these images for training, the bounding boxes are projected into the images for visualization purposes. The model producing the bounding boxes is the SSD model using the PP point cloud encoder, trained on only the car class with a reduced discretization size.

5.3 Comparison

The network based on PP that achieved the highest average precision for cars and had the fastest inference time was the network with smaller discretization, it achieved an AP of 82.7% and had an inference time at 14.6 Hz.

The network based on BEV that achieved the highest average precision for cars and was the network with the smaller discretization, it achieved an AP of 9.8%. The fastest BEV based network used one sweep and achieved a 18.7 Hz inference time.

Presented in Figure 5.13 are the AP for cars and the inference time in hertz for all the networks trained on the full dataset. The SSD models that were trained on the PP encoder is represented by triangles and the SSD models that were trained on the BEV encoder is represented by circles. Note that the first six presented models are trained with eight classes in comparison to the latter that is only trained with one class. However, the baseline models and one sweep models are trained with the same configuration except for the number of classes.



Model	AP Cars	Average Runtime
PP Baseline	76.7%	10.2Hz
BEV Baseline	1.6%	13.1Hz
PP MMLR	73.5%	13.4Hz
BEV MMLR	1.2%	12.2Hz
PP MM	69.8%	12.8Hz
BEV MM	1.3%	14.5Hz
PP One sweep	79.4%	10.2Hz
BEV One sweep	7.5%	**18.7Hz**
PP Five sweeps	80.3%	8.8Hz
BEV Five sweeps	0.2%	7.7Hz
PP Ten sweeps	79.8%	6.0Hz
PP Smaller discretization	**82.7%**	**14.6Hz**
BEV Smaller discretization	**9.8%**	16.9Hz

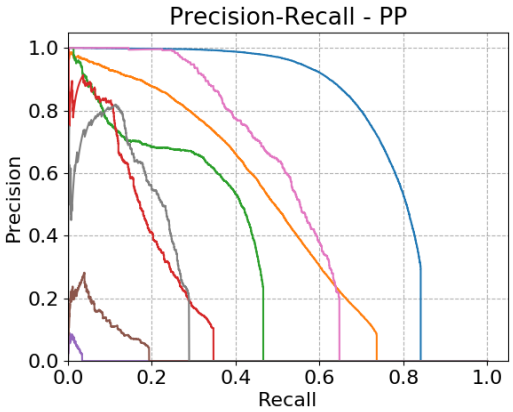
Figure 5.13: Comparison of the accuracy on only cars (AP) and inference time (Hz) for the SSD models based on PP (Triangles) and the SSD models based on BEV (Circles).

5.4 Best Performing Model

The most promising model overall, according to the thesis implemented evaluation was the SSD model using the PP encoder with a decreased discretization size. A model using the same concept except for the point cloud range and number of classes was trained, to be evaluated on the official nuScenes evaluation service. The hyperparameters presented in Table 5.13 was used for the final setup. The network trained for 42 epochs with the lowest validation loss of 2.04 at epoch 32. Note that the loss is not directly comparable to any other model considering that it handles a unique range of annotations. The resulting trained model achieve an accuracy presented in Figure 5.14, and inference time presented in Table 5.14. The model was also evaluated by the official evaluation, presented in Section 5.4.1.

Hyperparameters	
Parameter	**Value**
Classes	All merged classes
Number of sweeps	1
Discretization	0.16m × 0.16m
Point cloud range	±51.2m × ±51.2m
Final input size	640 × 640
Initial learning rate	0.0002
Milestones	Every 15th epoch
Early stopping	10
Anchor	3rd entry in Table 4.6 (k-means)

Table 5.13: Table of hyperparameters for the final model using the PP encoder with the SSD decoder.



Class	**PP AP**
Car	76.4%
Pedestrain	46.7%
Truck	31.6%
Motorcycle	18.2%
Bicycle	0.2%
Construction	2.9%
Bus	52.0%
Trailer	18.7%
Total mAP	**30.8%**

Figure 5.14: Precision-recall curve and mAP score, for the best performing network, trained and evaluated with the hyperparameter presented in Table 5.13. The TPM was set to a distance of 2 meters, and only predictions within 35m in range from the ego vehicle is handled.

Inference time	
Section	**PP time**
Encoding	18.1ms
Decoding	4.9ms
Postprocessing	125.5ms
Total	148.5ms
In Hertz	**6.7 Hz**

Table 5.14: Average inference time for the final model trained with the hyperparameters presented in Table 5.13. The IoU threshold in the postprocessing stage was set to 0.25 and no confidence threshold was used.

To further illustrate the results, produced bounding boxes from the final model on different samples from the test set are visualized in Figures 5.15, 5.16 and 5.17. The first and second example shown in Figures 5.15 and 5.16 depicts bounding boxes that have a confidence above 30%, no IoU overlap to other boxes over 0.4, and is within 35m range from the ego vehicle. The third example, shown in Figure 5.17, illustrate bounding boxes above a significantly lower confidence threshold of 5%.

5.4.1 nuScenes Official Evaluation

The trained model was used to predict all bounding boxes for the entire test set to be sent to the official evaluation server. The response from the evaluation is shown in Table 5.15. The evaluation uses four different TPMs of 0.5, 1, 2 and 4 meters, which are averaged together in the 'mean' column. Furthermore, the ATE, ASE, AOE scores refer to the average translation, scale and orientation error for each of the classes as explained in Section 4.2.2.

nuScenes Official Evaluation Results								
Object Class	**0.5m AP**	**1.0m AP**	**2.0m AP**	**4.0m AP**	**mean**	**ATE**	**ASE**	**AOE**
Car	41.4%	50.7%	56.7%	60.0%	52.2%	0.24m	0.16	1.15
Pedestrian	32.0%	33.5%	35.7%	39.0%	35.0%	0.20m	0.31	1.52
Truck	3.4%	9.3%	13.1%	14.7%	10.1%	0.45m	0.22	1.25
Motorcycle	8.0%	11.6%	12.9%	13.1%	11.4%	0.35m	0.27	1.77
Bicycle	0.0%	0.0%	0.0%	0.0%	0.0%	—	—	—
Construction	0.0%	0.0%	0.1%	0.5%	0.1%	0.83m	0.41	1.81
Bus	3.1%	19.5%	23.9%	27.2%	18.4%	0.53m	0.19	1.25
Trailer	0.0%	0.0%	3.5%	6.9%	2.6%	0.97m	0.21	1.74
Total mAP	**11.0%**	**15.6%**	**18.2%**	**20.2%**	**16.2%**	—	—	—

Table 5.15: Evaluation scores from the official nuScenes evaluation, on the overall best performing network, SSD-PP as presented in Section 5.4. The different AP columns use the differing TPMs presented, and the mean column refers to the mean of all the different TPMs scores. The additional ATE, ASE and AOE metrics refer to the average translation, scale and orientation error for each of the classes.

Figure 5.15: The final SSD-PP models produced bounding boxes from a test set sample. A confidence score above 30% and an NMS IoU threshold of 0.4 were used in postprocessing. The blue bounding boxes denote car predictions, while the orange bounding boxes represent the pedestrian classifications. Only boxes within a range of 35m from the ego vehicle are depicted. The produced bounding boxes enclose the majority of the objects present in the sample with a notable exception of a motorcycle in the front view and a pedestrian in the front-left view.

Figure 5.16: The final SSD-PP models produced bounding boxes from a rainy test set sample. A confidence score above 30% and an NMS IoU threshold of 0.4 were used in postprocessing. The depicted bounding boxes are within a range of 35m from the ego vehicle. The colors of the bounding boxes refer to the classes of the predictions where blue denotes cars, orange pedestrians and green trucks. Most objects present in the sample are predicted with a few bicycles being missed in the front-left view. Furthermore a false positive is present in the back view in the shape of a mailbox being predicted as a pedestrian.

Figure 5.17: The final SSD-PP models produced bounding boxes from a test set sample, with a lower confidence score threshold set to 5% and an NMS IoU threshold of 0.4 as the postprocessing settings. The depicted bounding boxes are within a range of 35m from the ego vehicle. The colors of boxes depict predictions of differing classes where blue denotes car, orange denotes pedestrian, green denotes truck, purple denotes bicycle, and cyan denotes bus. There seem to be proper predictions present for all objects. However, there are false positives present, such as in the back-right view where benches are predicted as bicycles.

6
Discussion

This chapter discuss the result achieved from the models presented in Chapter 5. The chapter starts by discussing the evaluation results for the models trained with the teaser dataset. The teaser data section is followed by a discussion about the evaluation results for the models trained on the full dataset. The chapter ends by discussing the final network submitted to nuScenes official evaluation service and potential future work.

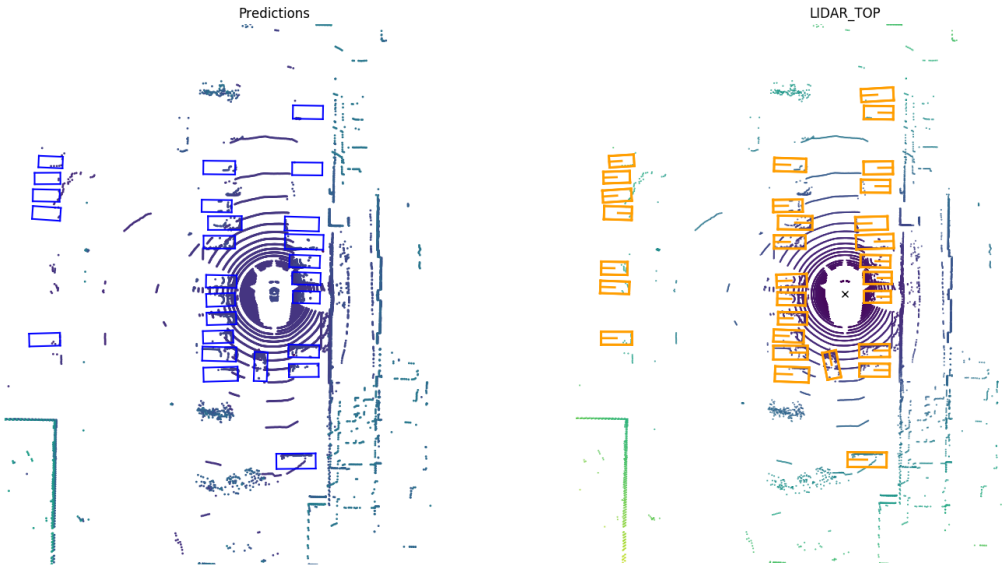
6.1 Teaser Dataset

As previously mentioned, the full dataset was not released until the second half of the master thesis, thus the initial model design was iteratively implemented with just the teaser dataset in mind. The goal was to implement models that could be verified to have the capability of learning the task at hand, and to later use the same model configuration with the release of the full dataset.

The objective of the experiment was to show that the loss could be decreased and that the reduction of loss correlated to better performance in actually finding the defined objects. This was done by training and evaluating on the entire teaser dataset, where the actual results are not of value in other aspects than the verification of which models were capable of learning and should be further investigated.

6.1.1 SSD on PP

As can be seen in Section 5.1.1, the final SSD model achieved promising results when using the PP point cloud encoder, with an AP for cars of approximately 95%. Ultimately showing expected results from a well-performing network, whereas the closer objects are easier to find than those further away, and the objects with the easiest visibility attribute also being the easiest to find. However, considering the data used in the evaluation, this result can only be interpreted as an indication that the implementation works, as it is most likely overfitted towards the same data used for the evaluation. A top-view visualization of produced bounding boxes for a point cloud, that depicts a parking lot, from the model can be seen in Figure 6.1.



(a) Predictions **(b)** Ground truth

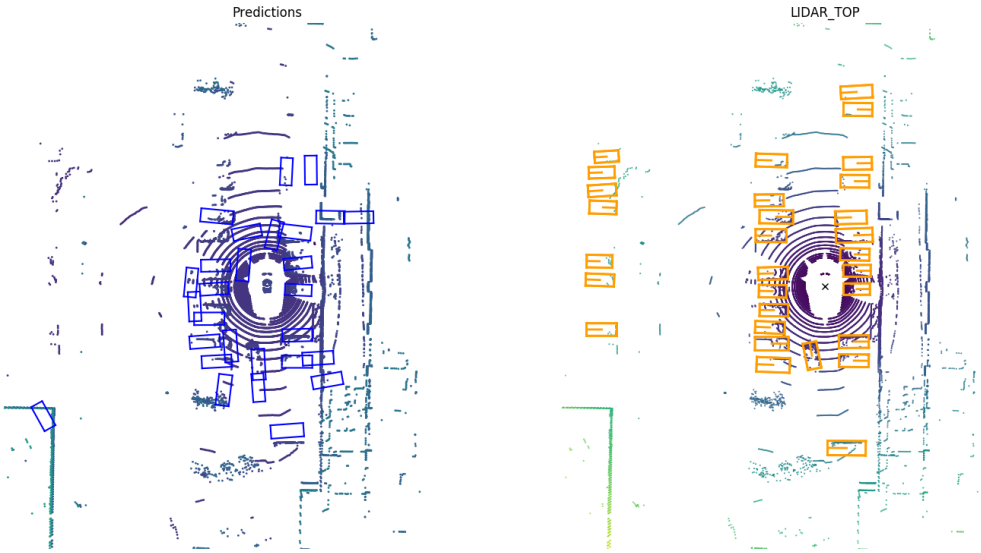
Figure 6.1: An example visualization of the predicted bounding boxes from the SSD on PP model in a top-view format. The model is only trained on one class, so all the shown bounding boxes are cars. Moreover, the bounding boxes shown are the ones with confidence over 30% and are not overlapping with another box by more than 0.4 in IoU. As shown, all of the closer ground truths are predicted correctly while a few of the objects further away are missed (using the mentioned thresholds).

6.1.2 SSD on BEV

The results in Section 5.1.1 for the SSD using the BEV point cloud encoder show that it achieved an AP of 37% on the car class. Furthermore, there is a significant difference in terms of accuracy for the different visibility levels, suggesting that the BEV encoder struggles to encode proper information when objects are somehow occluded. A hypothesis is that the issue with the low performance lies with the model not being confident enough in its produces predictions, as most (actually correct) predictions do not exceed 20% in confidence. Implying that the model has not learned to differentiate between background and objects as clearly as the implementation using the PP encoder has.

An example of a visualization of produced bounding boxes from this model in a top-view is presented in Figure 6.2. By looking at the figure, it is noticeable that the BEV model lacks confidence in comparison to the predictions made by the PP model on the same point cloud input, as shown in Figure 6.1. The extra predictions are mainly due to the confidence threshold is set to a significantly lower value of 10%. However, if the same threshold as PP is used, then no predictions would have

been shown. The predictions are generally close to the ground truths, but only a few would be considered perfect matches.



(a) Predictions **(b)** Ground truth

Figure 6.2: BEV visualization of predicted bounding boxes for the car class from the SSD on BEV model, the same scene as presented in Figure 6.1 is visualized. The shown predicted bounding boxes have a confidence exceeding 10% and no overlap with any other box over 0.1 IoU. Most of the produced bounding boxes are close to the ego vehicle, and fairly close to the ground truths, however, it is clear that the predictions has few perfect matches with the ground truths.

6.1.3 YOLOv2 on BEV

Looking at the results for the YOLOv2 implementation using the BEV representation in Figure 5.3, a conclusion was made that this specific network implementation was not meant for the task. The conclusion was drawn, mainly due to the struggle present when trying to localize smaller objects. As shown in the results, it is clear that there is a correlation between the size of the classes and the performance on that class. To draw a stronger conclusion the YOLOv2 should ideally have been trained with the PP encoder as well. However, considering the extent of the subpar performance on the BEV encoder, this was decided against.

The problem could lie with the use of real world coordinates, whereas most objects are relatively small in comparison to the input point cloud size. Further, the main issue with the implementation is the small produced feature map in which predictions are made, sized at 14×14. The feature map size defines how many anchors

are placed, and by having such a small feature map in relation to the point cloud range of 80×80m anchors are placed every $\frac{80}{14} \approx 5.7$meters in each x-y direction. As each placement in the feature map covers such a larger area, the target generation struggles to match objects. The ground truth objects are not typically placed with such sparsity, thus the sparse anchor placements require significant offset to match the ground truths, especially if the object class is small.

The newer version YOLOv3 [38] make predictions on three different scales of the input and is stated to perform better than YOLOv2 at detecting small objects. However considering that most objects looked for are small, a decision was made to focus on models based on the SSD model, to be further explored on the full dataset. Mainly due to the produced feature map in the SSD implementation being significantly larger, circumventing the issue present with finding the small objects.

6.2 Full Dataset

With the full release of the dataset, training was initiated to try to find the best hyperparameters for the implemented SSD on BEV and PP models. The validation set was used for evaluations, thus the results are skewed towards better accuracy. This is because the weights are saved during training when the lowest loss on the validation set is achieved. Thus by evaluating the model on the same dataset that stopped the training, implies that the evaluation data is not under the definition of what a test set should be; never before seen data.

6.2.1 Baseline Hyperparameters

The first baseline experiment used parameters chosen mainly to minimize training time while still being able to get results for every class. The baseline using the PP encoder produced a promising result for the Car class with an AP of 74%, however, the remaining classes fell short in terms of precision and the mAP for all eight classes became 24%, as shown in Figure 5.5. One reason for the low score is due to how mAP is defined (shown in Section 2.6), as every class' average precision is weighted equally even though there are for instance significantly more cars than bicycles present in the data.

6.2.2 Learning Rate Experiments

Considering the plot of the loss function during the baseline training in Figure 5.4, the initial reaction was that the loss stabilized unexpectedly early and plateaued. The theory became that the initial learning rate in the baseline was too high, causing the training to get stuck in a local minimum [41]. Thus experiments were conducted

on the learning rate by both reducing the initial learning rate and increasing the frequency in which the learning rate was forcefully decayed.

As shown in Figure 5.7, these experiments performed worse in terms of average precision across the board compared to the baseline in Figure 5.5. The dataset used when training the model is huge, and since the loss is plotted for every epoch instead of for every set number of iterations, it was concluded that the loss curve looked reasonable. This further strengthened the case that the loss had reached a fairly good minimum. As an afterthought, by plotting training and validation loss after a set amount of iterations instead of every epoch, it would have been easier to validate the learning rate settings.

6.2.3 Varying Sweeps for One Class

To explore the effects of having more input information, experiments were conducted by having a varying amount of sweeps as input per point cloud sample. By training on only one class, the problem is simplified, making the training converge faster, allowing more parameters to be tested. Furthermore, it also seem to increase the AP for the specific class slightly, as seen in Figure 5.3. Specifically, the one sweep models have the exact same configuration as the baseline models with an exception for the number of classes, where the one sweep only uses the car class.
The lowest validation loss from the PP models showed the expected pattern of achieving a lower loss if more sweeps per sample were used. Interestingly enough, the PP model that achieved the lowest validation loss did not achieve the best evaluation score. The PP model that achieved the highest AP was the model that used five sweeps per sample, while the PP model that achieved the lowest validation loss was the model that used ten sweeps per sample.

The lowest loss not yielding the highest performance in the evaluation is most likely due to the implemented evaluation (using the TPM of distance) not taking object size or orientation into account, while the loss function did. In other words, the lowest loss (which takes rotation and size into account) achieved when training the models, does not necessarily perform the best on the implemented evaluation.

Looking at the BEV based models, using multiple sweeps performed worse than using one sweep. This was not expected, however can be explained by looking at the channels defining the implemented BEV representation. The highest point and the maximum intensity channels would typically not be changed when introducing additional sweeps, the only changing channel would be the one referring to the density of the points within the grid. This implies that by using more sweeps, nothing is added to the feature map in terms of additional information except for noise from the different collection sweeps, making it harder for the network to produce meaningful features. This claim is strengthened by looking at the number of epochs it took to achieve the lowest validation loss, 19 epochs with one sweep and 34 epochs with five sweeps.

This experiment showed that using more than one sweep leads to a higher AP for the PP models, with the cost of a significantly slower inference time. Conversely for the BEV models, increasing the number of sweeps became worse in regards to both the AP and the inference time. The experiment also showed that using more sweeps per sample increased the encoding time, but decreased time spend in postprocessing stage for the PP models, the phenomenon is further discussed in Section 6.3.

6.3 Comparison

Looking at the inference time presented for each model, it is clear that the time spent in the postprocessing stage dominates. The postprocessing settings that were used, were aimed towards achieving the highest mAP. Thus, the confidence threshold was set to zero. Without the initial pruning threshold, no predictions are initially removed, leading to a lot of time-consuming IoU computations for the overlap pruning. These postprocessing settings used in the evaluations are typically not the postprocessing threshold one would use when actually applying the system, as for instance predictions with a confidence below 40% would not be reliable. This means that the presented inference times would be significantly faster when using a proper confidence threshold, tuned for the application task.

Due to the postprocessing stage dominating the inference time, the models that performed fewer predictions, achieved a somewhat fast inference time, even though their encoding time was significantly slower. This is something that is displayed in the varying sweeps inference time Table 5.9. One would expect that using more sweeps would take more time, but thanks to a higher confidence of background predictions, the postprocessing stage becomes notably faster.

The comparison plot and table presented in Figure 5.13 clearly shows that the best performing models, in regards to AP of detection cars were based on the PP encoder (as apparent from all previous experiments). The best performing overall model was the SSD-PP model that used the baseline parameters with a smaller discretization.

Due to the best performing model also producing the least amount of predictions, leading to a faster postprocessing stage, makes it also the fastest of the implemented models using the PP encoder.

In conclusion, the models using the BEV encoder seems to lack information to be able to produce distinguishing features such that appropriate predictions could be made. The best performing BEV model in regards to AP of detecting cars was once again the model that used the smaller discretization size. However, since the BEV model's confidences were so low, no real relation between high AP and fewer predictions can be drawn, thus the fastest BEV model was also the simplest (classifying only cars with one sweep).

6.4 Final Model

The best performing model was retrained with an input of ±51.2 meters in range, and was first evaluated on the thesis implemented evaluation method before sending the models prediction to nuScenes official evaluation service.

The implemented evaluation yielded a slightly lower mAP than that of the baseline. Due to the final model being trained on a significantly larger input makes the evaluation not directly comparable. As the final model is trained on a larger point cloud area, means that it must generalize the features better as there are fewer points available for objects further away from the ego vehicle. Thus, by increasing the point cloud range, the additional objects to find are commonly harder to predict. The reason for the final model being trained on a larger input is as explained in Section 4.2.2, that the official evaluation is ran within a range of 50m for the larger objects, and 40m for the smaller ones. Implying that the model must be trained for an input corresponding to at least the size of this larger range.

The reason for the final model showing worse performance in terms of AP on the nuScenes official evaluation service compared to evaluation method implemented in the thesis, specifically for the 2m TPM, is mainly due to two factors, the differing range and the restriction nuScenes have applied to their AP calculation. Firstly the differing range, which is a significant factor due to the official evaluation considering every prediction within a range of 50m while the other evaluation only considers the predictions within 35m. An expected behaviour of a object detection model is that it should perform worse the further away objects are located, thus a loss in terms of accuracy is expected when increasing the range. The second factor for the lower accuracy is the AP calculation restriction, referring to that the nuScenes evaluation service calculates its AP by taking the area on the precision-recall curve of the predictions over 10% recall and precision, while the thesis implemented evaluation calculates the AP by taking the whole area under the precision-recall curve.

Furthermore the bounding boxes presented in Figures 5.15, 5.16 and 5.17 are not completely as intended. This is due to the annotations that were considered in the project being based on the point cloud gathering LiDAR sensors coordinate system, while the projections into images were performed through the global coordinate system. Thus when translating the predictions from the LiDAR system into the global system, an unwanted pitch and roll are introduced on the predictions equal to that of the relative pitch and roll of the LiDAR sensors mounting position. This is especially clear in the back-left camera view in Figure 5.16.

6.5 Future Work

During the evaluation of the implemented models in the thesis, some clear improvements and variations that would be interesting in future work came to mind. Some possible improvements and issues of the implemented models, that could be investigated further are:

- One clear issue, that was realized during the evaluation stage of this thesis, is that the ignored classes according to Table 4.3 were directly handled as background. This was not ideal, as for instance the bikes that were stationed in a bike rack were handled as background, while the individually annotated bikes were handled as objects. Thus, for a model to achieve a high AP on bikes it must be able to not only find bikes but also differentiate bikes from bikes in racks. In essence this means that it is not surprising that all the implemented models are lousy at predicting bikes. The same problem goes for the police and ambulance vehicles, that are similar to the car and truck class.

  To solve this issue, one could instead of directly using the ignored classes as background, preprocess the point clouds by removing all the points that are located within the annotated bounding boxes for the ignored objects.

- As mentioned in Section 6.4 the predictions of the object were made in the LiDAR coordinate system. By changing the predictions to be made directly into the global coordinate system or alternatively the ego vehicles coordinate system would make the predictions better represent the real world.

- Using additional sweeps per sample showed higher AP for the PP model, and using a smaller discretization proved to also give better results. A further investigation of using more sweeps and smaller discretization in combination would be interesting. This specific configuration was tested, but seemed to require a larger amount of GPU video memory than was available in the machines at our disposal.

- The BEV based models performed far below expectations, a possible reason for this are the chosen feature channels. Thus it would be interesting to see the affects of different and additional feature channels. Perhaps some feature similar to that used in the PP implementation with the mean position within the pillar.

- As is apparent from the results, none of the hyperparameters were exhaustively explored. For instance only 0.25m and 0.16m in grid size was used, where the only conclusion that can be drawn is that the smaller one performs better. It would certainly be interesting to find the optimal grid size, and the same goes for all settings, to ultimately find the optimal model configurations. However, considering the sheer amount of time required for each training this is sadly a nearly impossible task.

- As mentioned in 6.3, the presented inference times are not really representative of how fast the models would produce their predictions in an actual application with a optimized confidence threshold. Thus, a potential improvement of the thesis would be to find confidence thresholds for each model providing a good trade-off between the number of outputted predictions and accuracy. To conclusively produce inference time that better

Following are some of the potentially interesting future work that was deemed out of the scope for the carried out project.

- Implementing versions of additional decoders, such as YOLOv3 [38] and Faster R-CNN [39], would make the comparisons of models more comprehensive.

- An additional interesting future work would be the use of sensor fusion. Both in terms of early fusion, merging the actual input together, and late fusion, merging feature representations. To ultimately investigate how merging additional sensory inputs would affect both accuracy and inference time.

7
Conclusion

The future and security of autonomous vehicles are heavily dependent on environmental perception, and its sub-task 3D object detection which is the focus of this thesis. The field has blossomed the last few years with papers like [23] and [40] pushing the limit of the SOTA. This thesis attempts to recreate the SOTA encoders and detection models to make comparisons of alternative implementations that produce 3D detections of frequently occuring objects in traffic situations, such as pedestrians and other vehicles.

In this thesis, the decoders SSD and YOLOv2 were implemented along with the encoders PP and BEV. Different hyperparameters were investigated to try to find the best performance for the SSD model. An implemented evaluation system showed that PP highly outperformed the BEV encoder in regards to mAP. The experiments performed in the thesis further showed that both PP and BEV achieved a higher accuracy when a relatively small discretization size was used, the experiments also showed that PP performed favorably by using more than one sweep per sample as input. The best performing overall model was evaluated by nuScenes evaluation service, and showed great promise, specifically for the detection of cars, while having a reasonable inference.

Bibliography

[1] Christopher Ingraham, The astonishing human potential wasted on commutes 2016. `https://www.washingtonpost.com/`. Accessed: 2019-01-25.

[2] Global status report on road safety. World Health Organization, 2015.

[3] Katie Pyzyk, Gridlock Woes: Traffic congestion by the numbers, 2018. `https://www.smartcitiesdive.com/`. Accessed: 2019-01-25.

[4] National Highway Traffic Safety Administration. National Motor Vehicle Crash Causation Survey. 2008.

[5] Shivang Agarwal, Jean Ogier Du Terrail, and Frédéric Jurie. Recent advances in object detection in the age of deep convolutional neural networks. *CoRR*, abs/1809.03193, 2018.

[6] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[7] Holger Caesar, Varun Bankiti, Alex H Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, and Oscar Beijbom. nuscenes: A multimodal dataset for autonomous driving. *arXiv preprint arXiv:1903.11027*, 2019.

[8] Xiaozhi Chen, Huimin Ma, Ji Wan, Bo Li, and Tian Xia. Multi-view 3d object detection network for autonomous driving. *CoRR*, abs/1611.07759, 2016.

[9] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *international Conference on computer vision & Pattern Recognition (CVPR'05)*, volume 1, pages 886–893. IEEE Computer Society, 2005.

[10] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[11] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[12] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338, 2010.

[13] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[14] Ross Girshick. Fast r-cnn. In *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[15] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Pro-*

ceedings of the IEEE conference on computer vision and pattern recognition, pages 580–587, 2014.

[16] D.O. Hebb. *The Organization of Behavior: A Neuropsychological Theory*. Taylor & Francis, 2005.

[17] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992.

[18] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, and Kevin Murphy. Speed/accuracy trade-offs for modern convolutional object detectors. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[19] Hidenori Ide and Takio Kurita. Improvement of learning for cnn with relu activation by sparse regularization. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 2684–2691. IEEE, 2017.

[20] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[21] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[22] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 7:1–7:6, New York, NY, USA, 2015. ACM.

[23] Alex H Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. Pointpillars: Fast encoders for object detection from point clouds. *arXiv preprint arXiv:1812.05784*, 2018.

[24] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[25] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 661–670. ACM, 2014.

[26] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988, 2017.

[27] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: single shot multibox detector. *CoRR*, abs/1512.02325, 2015.

[28] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[29] David G Lowe et al. Object recognition from local scale-invariant features. In *iccv*, volume 99, pages 1150–1157, 1999.

[30] Thorsten Luettel, Michael Himmelsbach, and Hans-Joachim Wuensche. Autonomous ground vehicles—concepts and a path to the future. *Proceedings of the IEEE*, 100(Special Centennial Issue):1831–1839, 2012.

[31] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, Dec 1943.

[32] B. Mehlig. Artificial neural networks. *CoRR*, abs/1901.05639, 2019.

[33] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[34] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. Learning deconvolution network for semantic segmentation. In *Proceedings of the IEEE international conference on computer vision*, pages 1520–1528, 2015.

[35] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[36] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.

[37] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. *CoRR*, abs/1612.08242, 2016.

[38] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[39] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015.

[40] Martin Simon, Stefan Milz, Karl Amende, and Horst-Michael Gross. Complex-yolo: Real-time 3d object detection on point clouds. *CoRR*, abs/1803.06199, 2018.

[41] Leslie N Smith. A disciplined approach to neural network hyper-parameters: Part 1–learning rate, batch size, momentum, and weight decay. *arXiv preprint arXiv:1803.09820*, 2018.

[42] Petru Soviany and Radu Tudor Ionescu. Frustratingly easy trade-off optimization between single-stage and two-stage deep object detectors. In *The European Conference on Computer Vision (ECCV) Workshops*, September 2018.

[43] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

[44] Jasper RR Uijlings, Koen EA Van De Sande, Theo Gevers, and Arnold WM Smeulders. Selective search for object recognition. *International journal of computer vision*, 104(2):154–171, 2013.

[45] Yan Yan, Yuxing Mao, and Bo Li. Second: Sparsely embedded convolutional detection. *Sensors*, 18(10):3337, 2018.

[46] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[47] Yuanyi Zhong, Jianfeng Wang, Jian Peng, and Lei Zhang. Anchor box optimization for object detection. *CoRR*, abs/1812.00469, 2018.

[48] Yin Zhou and Oncel Tuzel. Voxelnet: End-to-end learning for point cloud based 3d object detection. *CoRR*, abs/1711.06396, 2017.