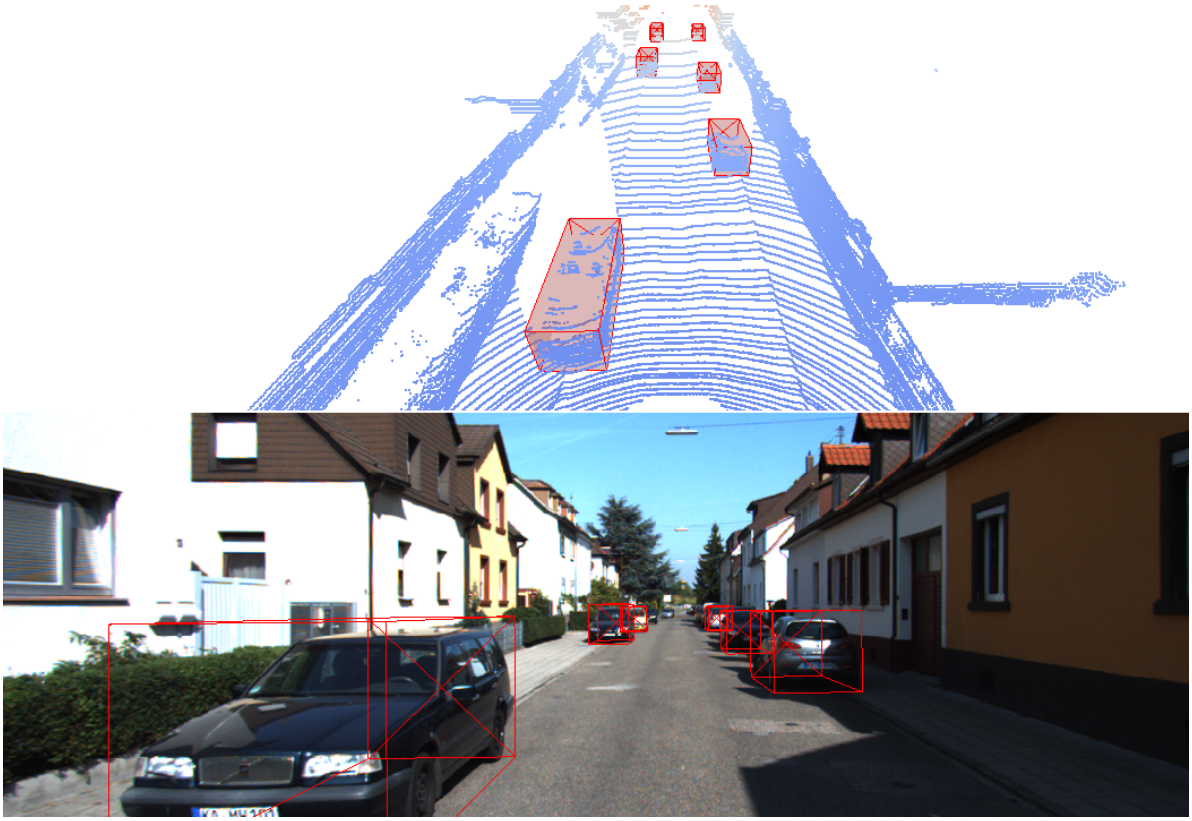
CHALMERS
UNIVERSITY OF TECHNOLOGY



Sensor Fusion with Failure Robustness

A Deep Learning 3D Object Detection Architecture

Master's thesis in Complex Adaptive Systems

JOAKIM BERNTSSON & ADAM TONDERSKI

Department of Physics
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2019

Master's thesis 2019

Sensor Fusion with Failure Robustness

A Deep Learning 3D Object Detection Architecture

JOAKIM BERNTSSON & ADAM TONDERSKI

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Physics
Chalmers University of Technology
Gothenburg, Sweden 2019

Sensor Fusion with Failure Robustness
A Deep Learning 3D Object Detection Architecture
JOAKIM BERNTSSON & ADAM TONDERSKI

Supervisor: Bernhard Mehlig, Department of Physics
Examiner: Bernhard Mehlig, Department of Physics

Master's Thesis 2019
Department of Physics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: 3D object detections in point cloud and image.

Typeset in LaTeX
Gothenburg, Sweden 2019

Sensor Fusion with Failure Robustness
A Deep Learning 3D Object Detection Architecture
JOAKIM BERNTSSON & ADAM TONDERSKI
Department of Physics
Chalmers University of Technology

Abstract

Autonomous Driving is the task of navigating a vehicle without driver interaction. This requires accurate perception of the surroundings, which includes three dimensional object detection. In this area, deep learning methods show great results, and they usually use either images, point clouds, or a fusion of both. These methods are often evaluated with full sensor availability. However, in a safety critical system, unexpected scenarios such as sensor failure must be accounted for. The objective of this thesis is to develop a deep learning architecture that is robust against the case where some sensors stop sending data.
The proposed architecture is inspired by leading LIDAR object detection models, which have reached a high performance. To be able to make detections during LIDAR failure, the network learns to convert the 2D image to the same representation as the LIDAR, in the form of an estimated 3D point cloud. The two point clouds are merged into a common representation, which allows the model to perform the detections jointly and thus work if either sensor fails. The final contribution is a novel training procedure with simulated sensor failure.
The results show that the model is robust against sensor failure, by reaching close to state-of-the-art performance for camera, LIDAR, and fusion with a single model. Additionally, on the KITTI dataset, the model outperforms three specialized versions that trained on camera, LIDAR, and fusion respectively.
A video of the model's detections can be viewed at youtu.be/_rKN_USMUoo.

Keywords: Deep Learning, Sensor Fusion, Object Detection, Sensor Failure, KITTI

Acknowledgements

We would like to thank our supervisors at Zenuity: Bernhard Birkner, Roman Glebov, and Armin Stangl. They provided guidance and insights which made this project possible. We also want to thank our supervisor at Chalmers University of Technology, Bernhard Mehlig, who helped with insight and resources throughout the project. Finally we want to thank Zenuity GmbH for hosting us in Unterschleißheim, Germany, and providing access to the computational power needed to run our experiments.

Joakim Berntsson & Adam Tonderski, Gothenburg, July 2019

Contents

List of Figures

List of Tables

1

Introduction

1.1 Background

The Society of Automotive Engineers [6] has divided the automation of vehicles into multiple levels, from no automation to full automation. During the first levels the safety responsibility lies on the driver, while the two highest moves the responsibility to the vehicle. This complete automation is what is referred to as Autonomous Driving, and has only become a feasible task in recent years due to advancements in algorithms and data collection methods. Automation puts high requirements on the system to be both robust and safe. If a particular sensor breaks, or the environment suddenly changes, the system should be able to safely adapt without any driver assistance.

Driving automation systems are composed of various technologies and methods. The initial step is to record data of the vehicle's surroundings using sensors. Next, this data is used to build a representation of the environment, which is called perception. This representation is then used by control and routing protocols for planning and decision-making, which eventually send instructions to the vehicle. [7]

In recent years perception has seen a surge of deep learning models. Neural networks can learn highly complicated relationships which can help robustness in difficult scenarios on the road. A downside of using deep learning is that not all decisions made by the network can be easily explained. This is a problem in safety critical autonomous systems where liability is a key factor. Therefore, to be able to ensure safety, it is common to have additional systems which are not neural networks, and also include rule-based systems in the control and routing algorithms [7].

Another safety aspect which is relevant for both classical and deep learning methods is what to do in the event of failure. For example in a vehicle with multiple sensors, one of them may break unexpectedly, which has to be accounted for.

1.2 Problem

Revisiting the robustness criteria for autonomous driving, it is evident that using a single sensor for perception could cause issues in the case of hardware failure. Even with proper sensor redundancy, using separate models for each sensor may not

be optimal, since certain objects can be very difficult to distinguish using just one sensor [8]. Finally, using multiple models for different sensor combinations can lead to a large overhead in development cost, and higher resources constraints on the vehicle.

Following this, the main research question to investigate is:

How does one build an object detection network that is robust against sensor failure?

The primary requirement on the network is that the performance has to be satisfactory regardless of which sensor, if any, fails. This will be checked by training a single network with a set of sensors and then evaluating it on all different combinations of those sensors. These results will then be compared against state-of-the-art methods with the goal of being as close as possible. Additionally, the detections will be separated into different distance intervals to see at which point the perception starts to degrade.

Sometimes, networks can generalize better when provided by additional or harder tasks. This leads to the follow-up question:

Can such a network generalize better?

This question will be evaluated by comparing transfer capabilities between different datasets, comparing against specialized models without sensor failure robustness, and analyzing metrics during training and validation.

1.3 Delimitations

Considering the large scope of this project, there are certain delimitations. The model will not be required to run in real-time on any specific hardware. Even though resource constraints are a very important part of autonomous driving perception systems [7], this is a big topic in itself and thus excluded.

All data that is used during training and analysis comes from existing public datasets, thus removing the time consuming task of data analysis, curation, collection, calibration, or annotation. This also makes it easier to compare results against other published methods. Additionally the only considered sensors are cameras and LIDARs, since this is a common sensor setup for object detection models [9]. These datasets also often contain objects of multiple classes, but the class space of the architecture is limited to cars.

Other than complete failure in the form of hardware malfunction, there is also the case of degradation. This can be caused by: weather - such as rain, snow, fog, etc; lighting - day or night; or other. All of which affects the sensors differently. This will not be considered in this thesis, and left as future work.

2

Perception in Autonomous Driving

This chapter will cover an introduction of deep learning concepts and theory, data used for autonomous driving perception, object detection background theory, and related works.

2.1 Deep Learning

Machine learning is a set of algorithms that can adapt their internal parameters to data they are presented with. This adaptation, or *learning*, is in contrast to other traditional algorithms where the behaviour is explicitly set by humans. A building block in machine learning systems are *Neural Networks*, which typically consist of multiple connected layers. With advances in computational hardware it has become possible to combine more and more such layers, which is commonly called *Deep Learning*. These models can have many millions of parameters and therefore require huge amounts of data to find an optimal fitting. However, despite the high requirements on the datasets and computational hardware, deep learning is a very powerful technique that is used in many state-of-the-art systems in computer vision [10], natural language processing [11], speech synthetization [12], and more. This section aims to give a brief introduction to the topic.

2.1.1 Neural Networks

Artificial Neural Networks are computational representations inspired by the natural neural networks that exist in the brain. They consist of simple units, called *neurons*, which are connected to each other to form large and complex networks. An artificial neuron performs simple linear regression followed by a non-linearity. This can be mathematically described by:

$$\hat{\mathbf{y}} = g(\mathbf{w}^T\mathbf{x} - b) \tag{2.1}$$

where $\mathbf{x}$ and $\hat{\mathbf{y}}$ are the inputs and outputs of the neuron [13]. The *learnable* parameters $\mathbf{w}$ and b are called *weights* and *bias* respectively. Finally, there is a non-linear function g, which is called the *activation function*.

A neural network consisting of a single neuron is called a *Perceptron* [13], see Figure 2.1. Due to its simplicity it can only learn a linear separation boundary, but it

serves as good example to explain some of the fundamental concepts relating to neural networks and how they are trained.

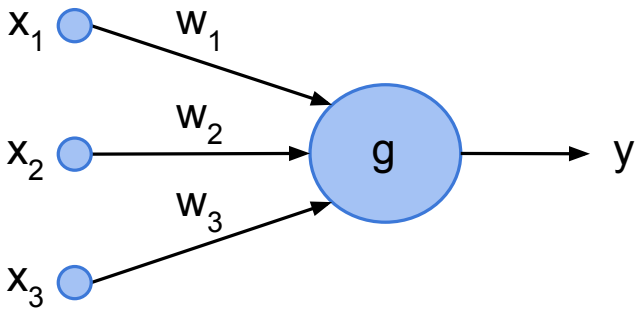


Figure 2.1: Neural network consisting of a single neuron, also known as a Perceptron. Each input is multiplied by a weight w. The results are summed, together with a constant bias, which is not displayed in the image. Finally the activation function g is applied to produce the output y.

A neural network can be seen as function that maps an input to an output, given some learnable parameters:

$$\hat{\mathbf{y}} = f(\mathbf{x}; \theta) \tag{2.2}$$

where θ are the learnable parameters, $\mathbf{w}$ and b in the case of a Perceptron. The learning, or training, of the network consists of finding the optimal values for θ, and can be split into two large categories:

- **unsupervised learning** - where the network is presented with a set of data without any predetermined correct answer. The task here is usually to find some structure in the data, for example dividing images of birds into a different set for each species.
- **supervised learning** - where the target outputs, $\mathbf{y}$, are known and can be used as a guiding signal for the network to learn the mapping from $\mathbf{x}$ to $\mathbf{y}$. An example would be the task of classifying whether an image contains a bird, given a set of images where a human has provided the correct answer for each image. The target output $\mathbf{y}$ is also commonly referred to as the *label* or *ground truth*.

The boundary between these categories is not clearly defined since some supervised problems can be reformulated as unsupervised, and vice versa. [14]

The most popular method for training networks in a supervised fashion consist of first defining a *loss* function $L(\mathbf{y}, \hat{\mathbf{y}})$, computing the gradient of this loss with regards to the network parameters θ, and using this gradient to update the parameters using *gradient descent*. The loss function computes a measure of how wrong the network is with regards to the desired output, for example the squared error:

$$L(\mathbf{y}, \hat{\mathbf{y}}) = \sum (y_i - \hat{y}_i)^2. \tag{2.3}$$

Since the loss describes how wrong the network is in its prediction, the task is to minimize it. Gradient descent is a classical optimization technique that accomplishes

this by iteratively taking small steps in the opposite direction of the loss gradient. Since $\hat{\mathbf{y}}$ is a function of the input $\mathbf{x}$ and parameters θ, shown in (2.2), the gradient with respect to θ can be defined as:

$$Q(\mathbf{w}) = \frac{\partial L(\mathbf{y}, \hat{\mathbf{y}})}{\partial \theta} = \frac{\partial L(\mathbf{y}, f(\mathbf{x}; \theta))}{\partial \theta}. \tag{2.4}$$

The last step is to update the weights. The simplest version of gradient descent follows the gradient directly:

$$\mathbf{w} := \mathbf{w} - \eta Q(\mathbf{w}) \tag{2.5}$$

where η is the *learning rate*. The choice of learning rate greatly affects the training, too large and the parameters jump around randomly, too small and the training never finishes. In practice there are several more sophisticated optimization algorithms that aim to solve various problems with this simplest approach, such as how to deal with very small or large gradients. One such adaptive algorithm is Adam [15].

In practice it is often impossible or impractical to perform this update step based on the entire dataset. Instead, a subset of the data, called a *batch*, is used to compute the loss and approximate the total gradient. This process is repeated many times, until a stopping criteria is reached. This could either be a predefined number of iterations, or based on the loss trend. One such loss based approach is to withhold a part of the dataset in a *validation split* and end the training when the validation loss starts to increase. [14]

2.1.2 Multi Layer Perceptrons

As mentioned previously, a single neuron can only learn simple mappings. Multiple neurons can be combined to capture more complex and non-linear patterns. However, the structure of these neural networks can vary significantly. One alternative is a Multi Layer Perceptron (MLP), where the neurons are arranged in layers, each neuron connected to all neurons of the previous and next layers. This can be seen in Figure 2.2. During training, the gradient is propagated backwards through the layers using the chain rule, which is called *backpropagation*. [14]

The usage of activation functions is especially important here, since it allows the MLP to learn non-linear separation boundaries. Without this activation, the effect of all layers could be summarized as a single linear operation. With enough neurons a single layer can approximate any continuous function, however the number of required neurons can be reduced by stacking multiple, simpler layers. The choice of activation functions is important and some common alternatives are the *sigmoid*, σ, the *hyperbolic tangent*, tanh, and the *rectified linear unit*, ReLU.

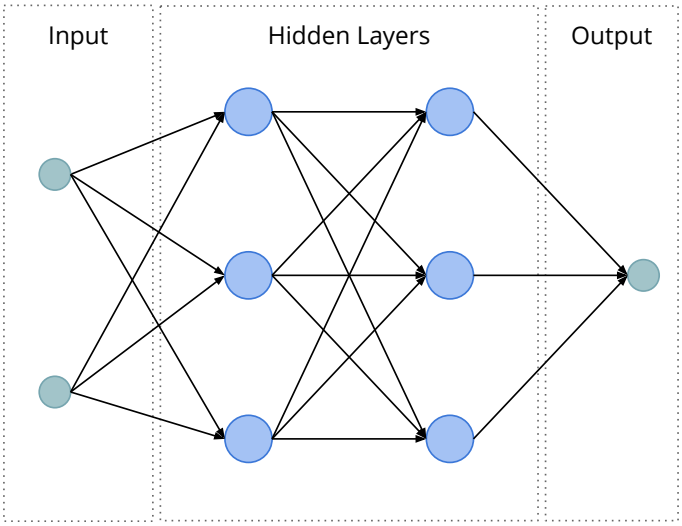


Figure 2.2: Multi Layer Perceptron, MLP. The layers between input and output are commonly called *hidden layers*.

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

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

$$\mathrm{ReLU}(x) = \max(0, x) \tag{2.8}$$

Sigmoid and tanh functions have the useful property of smoothly bounding all values to (0,1) and (-1,1) respectively. However, this squashes large values which potentially leads to very small gradients and slow learning. Therefore, they are usually used on the final network outputs whereas ReLU is the preferred choice for middle layers. [14]

2.1.3 Convolutional Neural Networks

When dealing with a large amount of inputs, the number of connections to the succeeding neurons will increase correspondingly. The most common example of this problem are images, where the total number of pixels can be in the millions. Convolutional Neural Networks, CNNs, are one way of dealing with this, where each neuron is represented as a *kernel*, which is a small patch that slides over the input to compute a transformed representation. See Figure 2.3. This can be seen as applying a simple Perceptron over and over to small parts of the input, and the technique extends to any number of dimensions. An important note is that this type of convolution assumes a meaningful neighbourhood in the data. For example, nearby image pixels are usually highly correlated. Another example where this assumption could hold is time series processing, where the convolution is 1D along the time axis. [14]

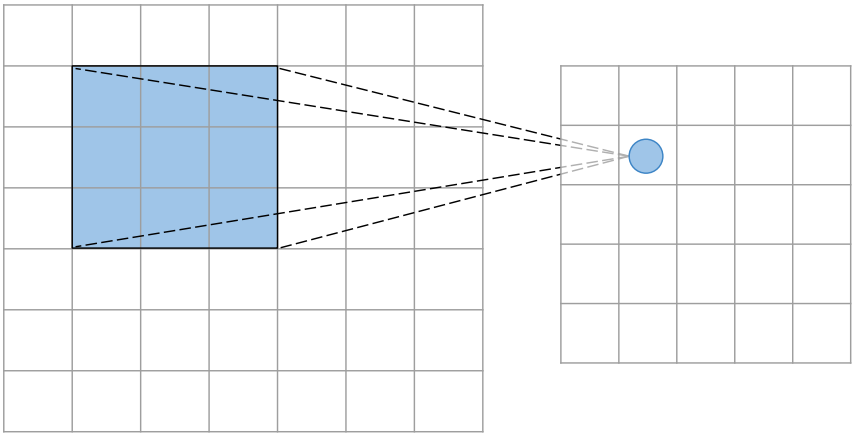


Figure 2.3: 2D convolution, where a 3×3 patch of the input is processed by the kernel to output a single value in the output feature map. This process is repeated over and over until the output is completely populated.

Similarly to MLPs, CNNs are organized in layers, typically consisting of N kernels of some given size ($K_1 \times K_2 \times ...$). If we assume a 2D convolution, this becomes $K_1 \times K_2$. When applied to an input of dimension $W \times H \times C_{in}$, it results in a *feature map* of size $W' \times H' \times N$. The output dimensions (W', H') depend on how the convolution is applied, and N is the number of kernels. The feature map is 3D, but 2D convolutions are applied since only two of the dimensions are spatial, and the third is a feature dimension. Since the convolution cannot be applied to the positions at the edge, either the output dimensions are slightly smaller, or the input is padded to compensate. Another significant parameter which affects the output dimensions is the *striding*, which is the interval at which the kernel is applied. For example, a stride of 1 means that the kernel is applied at every possible position, whereas a stride of 2 means that every other position is skipped and therefore the output dimensions are halved. This technique is often used to *downsample* the feature map while maintaining as much information as possible. The reason for shrinking the feature map will be explained more in the next section. [14]

2.1.4 Deep Neural Networks

In the context of neural networks, the depth is commonly referred to as the number of layers. The idea behind stacking layers is that each layer learns a transformation of the input, and many such layers together can learn a more complex transformation. However, there are some considerations when it comes to scaling neural networks. Increasing the number of parameters in the network usually require more data to avoid *overfitting*, i.e. overspecializing on the training data. Adding additional layers can also increase the required computational resources. Finally, deeper networks are typically harder to train due to the vanishing and exploding gradient problems. [14]

When it comes to deep convolutional networks, the intuition that more layers increases the number of parameters is slightly misleading. An important aspect when

discussing such networks is the *receptive field*, which is the size of the region in the original input that has influence on a particular position in the feature map. Having a large receptive field is important, to make sure that the network can learn large scale relationships in the input. Early on, most architectures tried increasing the receptive field by using large kernel sizes, such as 7×7. However, the same receptive field can be achieved by stacking three 3×3 convolutions. This has both the advantage of more non-linearities and fewer parameters. Another way to increase the receptive field is to downscale the feature maps before applying a convolution. Figure 2.4, aims to provide an intuition for the receptive field effect. [16]

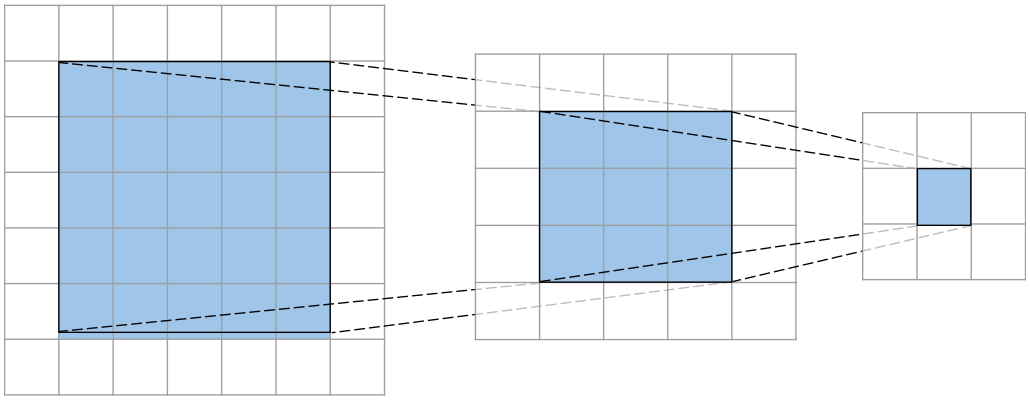


Figure 2.4: Increasing receptive field when multiple convolutional layers are stacked. Two 3×3 kernels produce a combined receptive field of 5×5.

One of the first networks to really push the concept of small kernels and downsampling was VGG [16], and this pattern has seen a continuing development in other image processing networks [3]. Figure 2.5 shows how the input is processed by a series of convolutional layers with intermediate downsampling steps. An MLP is connected to the final feature map to output classifications.

A way to reduce overfitting is to apply *regularization*. One such technique is called *data augmentation*, and is the process of simulate a larger dataset by altering the data. For example, images can be randomly rotated, mirrored, and rescaled. [14]

While regularization can help with overfitting, another major problem are vanishing and exploding gradients. The problem arises from backpropagating through many layers where a lot of gradients are multiplied. If some of these gradients are extreme or close to zero, the training can be hindered. *Batch normalization* [17] is a technique that was introduced to improve the performance of deep networks. The idea is to perform zero-mean and unit-variance normalization of the features, based on the statistics in the current batch. Even though the exact reason why batch normalization works is still debated, experiments show that it can bring significant benefits [18].

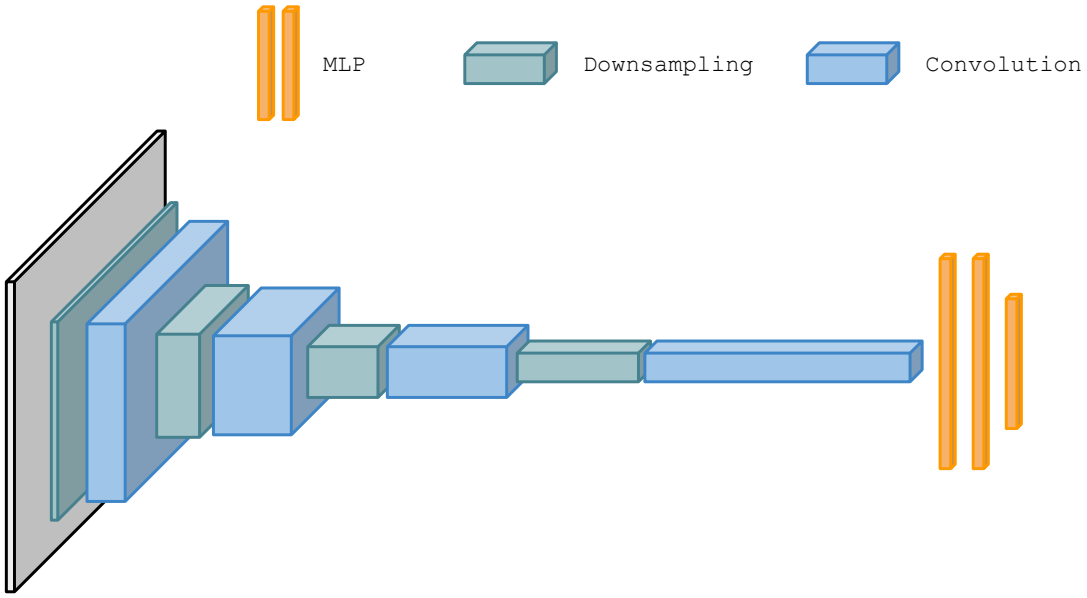


Figure 2.5: An architecture concept inspired by VGG, where convolutions and downsamplings are alternated to produce a small map with a high number of features. Finally an MLP is applied to produce classifications.

2.1.5 Applications within Perception

Deep Learning has lead to breakthroughs in many perception tasks. The examples below are based on images even though the categories mostly apply to other input types as well. Following is a non-exhaustive list of common perception tasks: [9]

- **Image Classification** – assign a class to an image.
- **Object Detection** – detect objects in an image and assign a classification and bounding box to each. This bounding box can either by 2D or 3D, depending on the task.
- **Semantic Segmentation** – assign a class to each pixel in the image.
- **Instance Segmentation** – a hybrid between object detection and semantic segmentation where each pixel is assigned to a specific object instance.
- **Depth Estimation** – estimate the 3D depth for each pixel.

2.2 Data

Much like human drivers use their senses to perceive the environment to be able to drive, vehicles are mounted with a variety of sensors which pass data to the perception models. Below is an overview of common sensors used in autonomous driving systems.

2.2.1 Sensors

A sensor is a hardware device capable of collecting data of the surrounding environment. An individual sensor has various properties, such as: proximity; color; contrast; speed; range; resolution; sensor size; and sensor cost. There are also environmental properties, whether the sensor works in dark and bright conditions, and how weather affects it. Each sensor has its strengths and weaknesses regarding these properties. [19]

Following is a non-exhaustive list of common sensors in autonomous driving: [9]

- **Camera** - Cameras produce images with dense information and detailed textures at high range. Their drawback is the sensitivity to lighting and weather conditions, and lack of depth information. Additionally, objects that are close to each other in the image may be very far apart in the real world.
- **LIDAR** - LIght Detection And Ranging sensors send out laser beams and measure the reflections. This produces a set of points which provide depth information of the surroundings, called *Point Clouds*. LIDARs are less affected by lighting and weather conditions than cameras, however, they have inferior resolution, especially at long ranges. Another important thing to note is that LIDAR systems are very expensive [19].
- **RADAR** - RAdio Detection And Ranging works similarly to LIDAR, where instead of using lasers it uses radio signals. RADARs are even less sensitive to lighting and weather conditions than LIDARs. They can also measure velocity through the Doppler effect but have the drawback of very low resolution.
- **Ultrasonic** - Ultrasonic sensors are often used for near object detection in low speed scenarios where they have accurate distance estimation. They are very cheap, but sensitive to humidity, temperature, and dirt.
- **GNSS & HD Maps** - Global Navigation Satellite Systems and High Definition maps can provide accurate vehicle 3D positioning. Maps can also provide additional information about static parts of the surroundings, for example the height of the terrain [20]. However, they can lack availability since GNSS relies on satellites, and maps can be outdated or non-existent for certain areas.

To combat the weaknesses of individual sensors, one can utilize multiple sensors together in a way that provides more information of the surroundings than they do by themselves. This is called sensor fusion. An example is to combine the rich color and texture information of cameras with the exact positioning information of LIDARs. Additional complementary behaviour of cameras and LIDARs can also be seen in the list above, where cameras have great resolution but are sensitive to disturbances, while LIDARs have inferior resolution but are more robust against weather conditions.

2.2.2 Datasets

A dataset is a collection of data samples, such as images and point cloud files. The requirements of the dataset depend on what sensors the model will use. Some datasets

only have images, and others may have images and point clouds with respective calibration matrices to go between different coordinate systems. The datasets can also have annotations, or ground truth labels, for example three dimensional bounding boxes of objects. [21, 22]

The quality of a dataset is vital to the performance of an architecture trained on it. For supervised learning, annotations are needed, specifically 3D bounding boxes for the task of 3D object detection. The datasets for object detection are very small compared to other computer vision tasks, such as image classification, which also impacts performance of a model [9].

The most commonly used public dataset for 3D object detection in Autonomous Driving is KITTI [21].

2.3 Object Detection

An object detection network can be condensed into two key components: feature extractor and detection head. The purpose of the feature extractor is to find a good high-level representation of the data in feature space. This is then used by the detection head to form bounding boxes around objects. While a feature extractor typically works with input from a single sensor, it can also combine inputs from multiple sensors, called sensor fusion. Another common way of improving the quality of detections is to use multiple stages, where the detection can be done several times to refine them further.

2.3.1 Feature Extraction

This section describes various approaches for extracting features from images and point clouds.

2.3.1.1 Images

In recent years, image recognition benchmarks such as ImageNet [23] have been dominated by deep convolutional architectures. Since AlexNet [24] won the competition in 2012, the depth has kept increasing. However, as mentioned in Section 2.1.4, deeper networks are harder to train. A major breakthrough that helped combat these problems is ResNet [3], which introduced the concept of residual blocks, consisting of a few convolutional layers. The purpose of a residual block is to learn the difference between the input and output, rather than a direct transformation. This output of the residual block can be defined as $\mathcal{F}(x) := \mathcal{H}(x) - x$, where x is the input and $\mathcal{H}$ is the total transformation that we want to avoid computing. In practice this means that we add the input at the end of the residual block as can be seen in Figure 2.6, which is equivalent to computing $\mathcal{H}(x) = \mathcal{F}(x) + x$. The method of adding the input to a residual is also called a skip connection. This creates additional, and shorter, paths for the gradient to flow backwards through the network.

Shorter paths contain fewer multiplications, and therefore a smaller chance of the gradient vanishing.

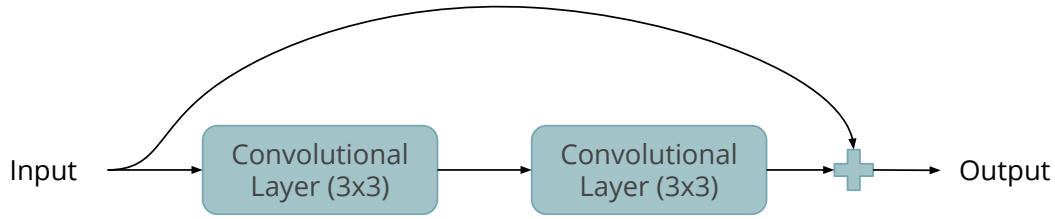


Figure 2.6: A residual *building block* that uses two convolutional layers to produce the residual, which is added to the input of the block to create the output.

ResNet allowed the construction of much deeper networks and other ideas have taken the concept of residuals even further. An example is DenseNet [25], where the residual is over the input of all preceding blocks. Dense refers to the higher connectivity within the network.

The networks above are usually used as encoders to create a small map of very high level features with large receptive fields. However, ideally the final map would be of high resolution so that object detection can be more spatially accurate. A prominent method to accomplish this is a Feature Pyramid Network [4]. The idea is to progressively enlarge the feature map by upscaling it and adding the features from earlier layers in the encoder, yielding a high resolution and high quality output representation. This is illustrated in Figure 2.7.

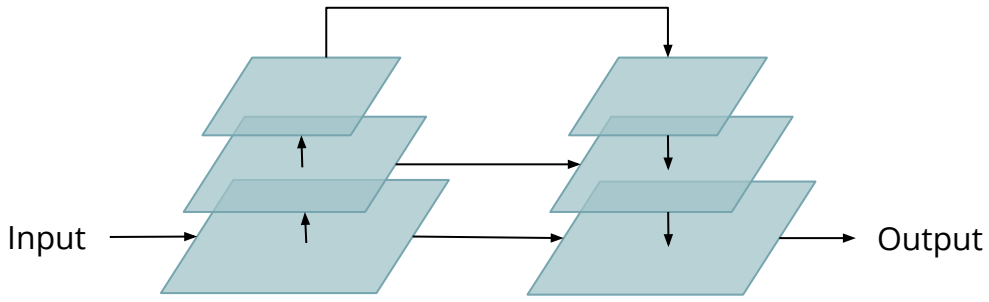


Figure 2.7: A feature pyramid where the input is scaled down twice, and then scaled up to the original resolution.

Deep Layer Aggregation [26] combines the ideas of DenseNet and Feature Pyramid Networks to extract more meaningful and high resolution features. The fundamental

idea is to perform the upscaling in smaller steps, rather than just at the end. A comparison between feature pyramid and deep layer aggregation can be seen in Figure 2.8.

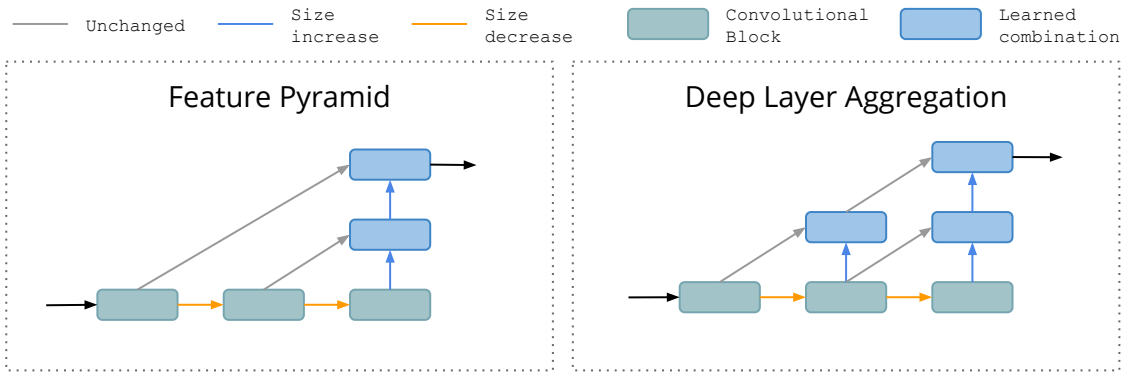


Figure 2.8: Comparison between Deep Layer Aggregation and the traditional Feature Pyramid Network. This is a simplified example with much fewer blocks than commonly used.

2.3.1.2 Point Clouds

A point cloud is an unordered set of points with continuous values which describe a three dimensional environment:

$$[(x_1, y_1, z_1), \ldots, (x_N, y_N, z_N)]$$

With some spatially dependent input data, it is possible to use convolutional approaches to learn relationships. However, since the data is continuous and has no canonical order[1] these methods can not be directly applied.

In order to apply traditional convolutional methods the data has to be discretized. The grid can include all three spatial dimensions, where the voxels[2] can contain aggregated information of the points inside them. A common special case for this in autonomous driving is a 2D top-down bird's eye view grid where cars are usually well separated.

Discretizing point clouds can make the data more voluminous, sparse, and introduce artefacts that can hide natural invariances in the data [27]. Some more recent publications work with the point clouds directly and overcome the fact that there is no canonical order by using a symmetrical function. For example, PointNet [27] uses an MLP with max pooling as the symmetrical function. These methods produce point-wise feature vectors, as opposed to discretization methods which produces a matrix of features in the discretized space.

[1] A canonical order of a collection of items is their natural order
[2] A voxel is a cell in a three dimensional grid.

2.3.2 Header Network

Using features from an extractor, the purpose of the header network is to create the outputs of the network. For object detection the output typically includes per class bounding boxes and confidence scores. The specific encoding of the bounding box depends on the requirements and constraints of the model's use case. In some cases a two dimensional bird's eye view bounding box is sufficient, in which case the encoding could consist of box center location (x, y), size (w, l), and yaw angle θ [28]. In other cases all eight corner points are desired to get more freedom in the shape of the box [29]. The confidence score is usually between 0 and 1, and in the case of multiple classes, there is a score per class.

To form bounding box values and confidence scores, the feature maps are processed into a size depending on the desired resolution, and a depth corresponding to the number of bounding box values plus confidence scores. To keep the regressions invariant of the position in the feature map, the box regression values can relative to the positions in the map [28]. It is also possible to include prior knowledge of the box sizes, for example the size of a typical car. Using such a prior, the network only has to regress the difference against those sizes [8].

The targets containing an object from a specific class are referred to as positive/-foreground, while the rest are negative/background. Objects usually occupy a small part of the input, which creates a foreground-background class imbalance [30]. Two ways to alleviate this problem are Focal Loss [30], and Hard Negative Mining [8]. Focal Loss uses a modulating factor to down-weight the loss for well-classified cases. As an example, five negative cases with $p = 0.1$ have less impact on the loss than a single positive case with $p = 0.5$. Hard Negative Mining samples a subset of the negative cases to reduce the imbalance.

To form the final list of outputs from the dense map of detections, the first step is to apply a threshold to the predicted confidence scores. At this point it is still very likely that there are multiple detections of the same object. Non-Maximum Suppression, NMS, is a technique for removing such duplicates [30, 28, 8, 29]. NMS only keeps the boxes with highest confidences and removes all other detections with a overlap over a certain threshold. The overlap is computed using Intersection over Union, IOU, which is the area of intersection divided by the area of their union. What is left is a list of detections which the network is most certain about, without any duplicates.

Evaluating the list of boxes from the network can be done qualitatively by plotting the boxes in the actual point cloud or image, and including the ground truths as comparison. For a quantitative analysis, a common single-value metric is average precision, AP [8, 29, 28]. The AP is computed by averaging the precision value over a fixed set of recall values [31]. Recall describes how many of the ground truths were found and precision describes the likelihood that a detection is correct. These are defined as $\frac{TP}{TP+FP}$ and $\frac{TP}{TP+FN}$ respectively, where TP is true positives, FP false positives, and FN false negatives. Note that the AP metric puts equal weight on all objects, which is why certain papers present AP divided into distances or difficulty [32]. There are also a wide variety of implementations of this metric, which can

hinder its comparability [8, 32]. The precision and recall values can be plotted against each other, where a flat line means that no matter the predicted probability, the detection is equally likely to be true, whereas a typical downward slope means that the the model gets more and more uncertain as the probability decreases. This kind of plot is called a Precision-Recall curve.

2.3.3 Fusion

Fusing data from different sensors can be done in many different ways. A simple way would be to process the inputs independently in separate networks, and then combine the outputs. The drawback of such a strategy is that the networks can not learn from each other. An alternative to this is to perform the fusion within the actual network, which can be divided into three strategies: early, late, and middle fusion [9].

To explain the different strategies of fusing, consider an example where a front view projection of a point cloud and an image are fused in some way to finally produce detections. This will be used throughout the following paragraphs as a reference example.

Early fusion [9] is when the sensor data is combined as a first step, and the joint representation is processed by the network. This adds very little overhead to the network for each additional sensor since they will share weights. In the reference example, this could mean that the image and front view projections are cropped and resized to the same dimensions, and then the projection and the image would be concatenated into a single tensor and fed into the feature extractor.

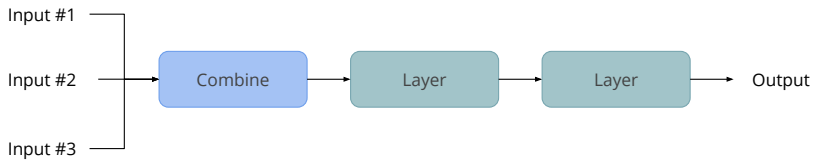
Late fusion [9] uses separate feature extractors for the different sensors, and then combines the feature representations. This is more robust against raw data invariances than early fusion and creates a separation of concerns. Late fusion is however more computational expensive than early. Continuing the reference example, the feature maps of the image and front view projection could be concatenated as a final stage before the header network forms the outputs.

Middle fusion [9] is a compromise between early and late fusion. The intuition is that the individual sensors' feature representations may benefit from each other at different stages of the network. In the reference example this could constitute to concatenating the feature representations between layers in the individual extractors.

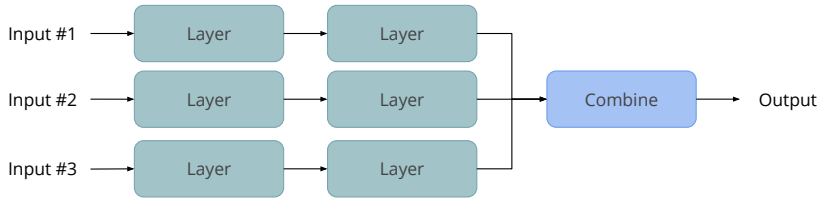
Visualizations of each fusion strategy can be seen in Figure 2.9.

2.3.4 Multi Stage Architectures

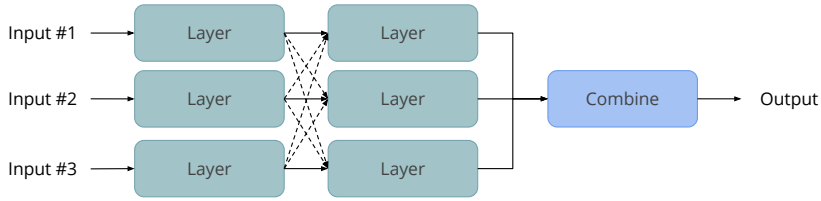
A single stage architecture processes an input, produces features, and outputs predictions just once. This might be the most obvious approach, but can be problematic and not achieve high performance. [1]



(a) Early Fusion



(b) Late Fusion



(c) Middle Fusion

Figure 2.9: Comparison between the different fusion stategies.

A method to increase the performance is to formulate two stages: one proposal stage, with the purpose of finding rough estimates of the actual bounding boxes; and a refinement stage that uses the proposals to extract smaller feature maps to refine the boxes of the final detections. This method can be very effective to reach high performance metrics, seen in for example PointRCNN [33]. An inherent problem is the slow run time, both during training and inference, which comes from the fact that the detections are essentially performed twice.

Due to the speed and simplicity of one-shot detectors, they can be preferable in some applications. One of the first single-stage architectures that maintained low inference times and high performance is the Single Shot MultiBox Detector, SSD [1]. There are also examples in 3D object detection [28].

2.4 Related Work

This is an overview of notable object detection architectures in autonomous driving. Only architectures working on point clouds, images, or both will be considered.

2.4.1 Point Cloud Architectures

Pixor: As mentioned in Section 2.3.1.2, a BEV projection is an efficient simplification with reasonable constraints in the context of autonomous driving. Pixor [28] is an architecture that focuses on exploiting this representation to achieve high speed detections without sacrificing performance. The network's input is a BEV grid with 32 channels representing slices in the height dimension. This is treated as an image and fed through a custom ResNet-based architecture with a Feature Pyramid Network. Finally, a header consisting of a few convolutional layers is used to output dense, per pixel, detections. The architecture is depicted in Figure 2.10.

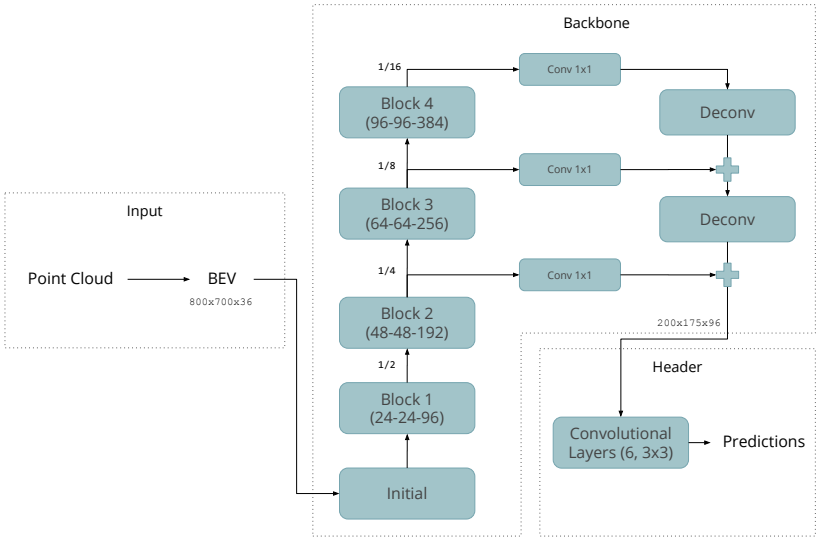


Figure 2.10: The Pixor architecture. The point cloud is discretized into BEV and passed through a RestNET-based feature extractor to create dense predictions.

Pixor++: An architecture that further improves upon Pixor, notably by encoding the angle as $(\cos 2\theta, \sin 2\theta)$ instead of $(\cos \theta, \sin \theta)$. This reduces the periodicity to π instead of 2π, which in other words ignores the facing direction of the object. This simplification makes sense in the context of average precision, since the facing direction does not affect the bounding box overlap. Another performance boost was seen by HDNET [20], which uses the same architecture with incorporated high definition map information to:

- subtract the ground height from the point cloud in order to compensate for varying terrain
- add a semantic road prior to the input channels

Since map is not included in the used public dataset, another network estimates the road segmentation and ground heights. This network is pre-trained on a much larger, private dataset.

VoxelNet: The idea of VoxelNet [34] is to discretize the point cloud into a three dimensional grid, where each voxel is described by a feature vector. The features

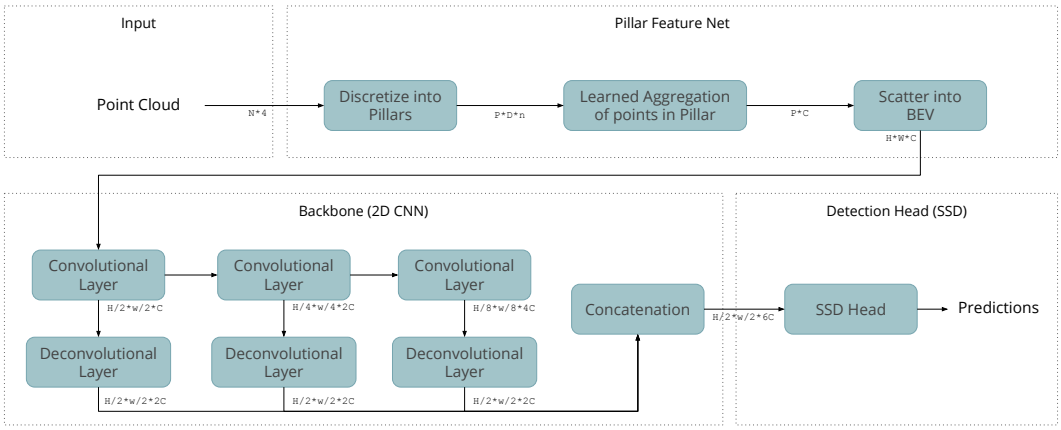


Figure 2.11: Overview of PointPillars, where the details around the SSD head [1] are left out for brevity. The depicted dimensions are: number of points N; number of pillars P; point dimension D; points per pillar n; second point dimension C; and BEV dimensions $H \times W$.

of each voxel are computed by feeding all points inside that voxel through multiple novel "Voxel Feature Encoding" layers. The feature map is processed by 3D convolutions, followed by collapsing the height dimension, which allows for 2D convolutions to be used in the head to form outputs.

PointPillars: A continuation of VoxelNet where real-time detections are the main objective. The majority of the time spent by VoxelNet is during the 3D convolutions. To avoid this, PointPillars uses a BEV grid with feature vectors for each pixel. To encode the point cloud into this representation, all points inside each BEV "pillar" are fed through an MLP and aggregated using max pooling to form a feature vector. Then, 2D convolutions are used for feature extraction followed by an SSD [1] detection head. These changes enable PointPillars to be extremely fast at around 105Hz. An overview of the architecture can be found in Figure 2.11. [35]

LaserNET: An architecture that uses a special front view projection of the point cloud with five channels, which is meant to be more "LIDAR native" since each height pixel corresponds to a LIDAR beam. The five channels are: height, azimuth, range, intensity, and a flag indicating whether the pixel contains a point. Deep Layer Aggregation is used to extract features and a single layer head outputs bounding box parameters to be used in a Gaussian Mixture Model. The large amount of outputs require a lot of data, which is why this model performs well on large datasets, but not on smaller. Another addition in this architecture is the use of "soft NMS" as opposed to the more commonly used "hard NMS". The idea is to not use the NMS hard limit and throw away outputs, but instead re-score them each time using their IOU. [36]

PointRCNN: This architecture operates directly on the point cloud as opposed to the previous ones. It uses the PointNet++feature extractor to generate point-wise features, which are used to generate region proposals for each point. The highest

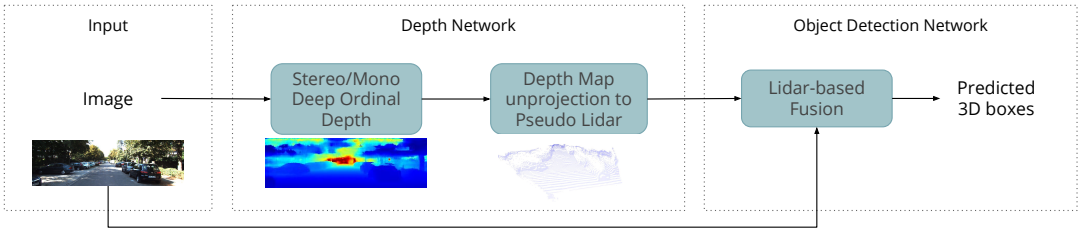


Figure 2.12: Overview of Pseudo-Lidar [2], without any details about the sub-networks.

scoring proposals are used to crop the point cloud, re-center the coordinate system, and output the final box regressions and confidence scores. It is the only state-of-the-art architecture that works exclusively on the direct point clouds. [33]

2.4.2 Image Architectures

This section covers architectures using only images from a mono camera setup as input data for 3D object detection. There are other approaches that use stereo camera setups for easier perspective estimation, but these are not covered here.

MonoGRNet: MonoGRNet [37] consists of four parts: 2D detection, Instance-level depth estimation, 3D location estimation, and 3D box corner regression. The 2D detector is used to find Regions of Interest, RoIs, which are then used in the depth estimation to only find the depth of interesting areas. Then, the 3D location module estimates the object center point in 3D and refines it using the depth information. The final 3D box corner regression uses the RoI-aligned feature maps and the refined 3D location to output eight corners in a local coordinate system.

Pseudo-Lidar: This architecture uses pre-existing work on depth prediction and sensor fusion object detection networks to improve upon state-of-the-art results. First, images are fed through a depth prediction network [38], to create a depth map. The depth map is then unprojected into a pseudo point cloud using camera calibration information. Next, the pseudo point cloud is used instead of a real point cloud in a fusion object detection network, such as AVOD [29] or F-PointNet [39]. An overview of the architecture can be seen in Figure 2.12. An important thing to note is that the depth prediction network is trained on a larger set of images, which includes samples from the object detection validation and test splits. Considering this, the results can not be compared against methods which only train on the object detection samples. [2]

MonoFusion: MonoFusion [40] uses a trained and freezed depth network to generate an estimated point cloud from the image. A front view projection of this cloud is concatenated with the image in a form of early fusion, and then passed through a convolutional feature extractor. The feature maps are then used in a 2D detector head to get region proposals, which are used to crop and combine the feature maps

and front view projections. Finally, a 3D regression sub-network outputs 2D and 3D detections. Similarly to Pseudo-Lidar, the depth network is trained on a larger set of samples that can overlap with the test and validation data for object detection. Even so, MonoFusion is commonly used in comparisons with other architectures [37, 41].

2.4.3 Fusion Architectures

FPointNet: The Frustum-Point-Net [39] approach tries to exploit the fact that 2D image space detection architectures have reached much higher performance than their 3D counterparts. Therefore, the image is first passed through a state of the art CNN architecture to generate 2D proposals. These image regions are then used to crop the subset of the point cloud whose points fall into that image region. This cone-like slice is called a *frustum*. Then, a PointNet is used to perform an point wise classification to additionally filter out points that do not belong to the object. Finally, the remaining points are fed into another PointNet, which outputs the final bounding box parameters.

PointFusion: Similarly to FPointNet, this architecture uses a 2D detector to create region proposals in order to crop relevant regions of the point cloud and feed them into a PointNet. However, as opposed to FPointNet, the cropped image region is also passed through a ResNet to extract image features, which are concatenated with the point-wise features. Finally, these combined features are fed through an MLP to generate a bounding box for each point in the Frustum. The box with the highest score is used as the final detection. [42]

MV3D: Multi View 3D Detection [43] introduces a more balanced fusion process than the approaches above. There are separate convolutional feature extractors for: RGB images; LIDAR Front View projections; and LIDAR BEVs. Then, a region proposal network extracts regions of interest based on the BEV features. These proposals are used to crop and resize features from all three extractors, which are processed together in a series of novel "Deep Fusion Layers", which use a form of a middle fusion. Finally, a detection header uses combined features to output the classification and bounding box regression. An unusual aspect of this architecture is the use of auxiliary losses in the deep fusion layers, that are supposed to stabilize training and force different parts of the architecture to specialize on the different inputs.

AVOD: Similarly to MV3D, the Aggregate View Object Detection [29] architecture uses convolutional layers to separately extract features from RGB images and LIDAR BEVs. However, AVOD tries to address one of MV3D's shortcomings, namely that the region proposal is only based on a single sensor. Instead, AVOD uses a grid of 3D anchors to pair regions from the image and BEV feature maps. The paired regions are then cropped, resized and fused using addition. The combined features are used to create region proposals, which are refined in a second stage, where the procedure is repeated using higher resolution feature maps.

Deep Continuous Fusion: This architecture uses ResNets with Feature Pyramids to extract features from RGB images and LIDAR BEV projections. The main innovation comes from the fusion mechanism, which uses novel "Continuous Fusion Layers" to incorporate image features after each BEV ResNet block, seen in Figure 2.13. The mapping from image space to BEV is performed as:

1. Assign K nearest LIDAR points to each BEV grid cell
2. Project the points onto the image feature map to find image features
3. Pass the image features through an MLP together with the point's offset relative to the BEV position
4. Take the sum of the resulting features

Unlike some of the previous architectures, this feature combination is learnable, which could allow for a more accurate mapping between the sensors. [8]

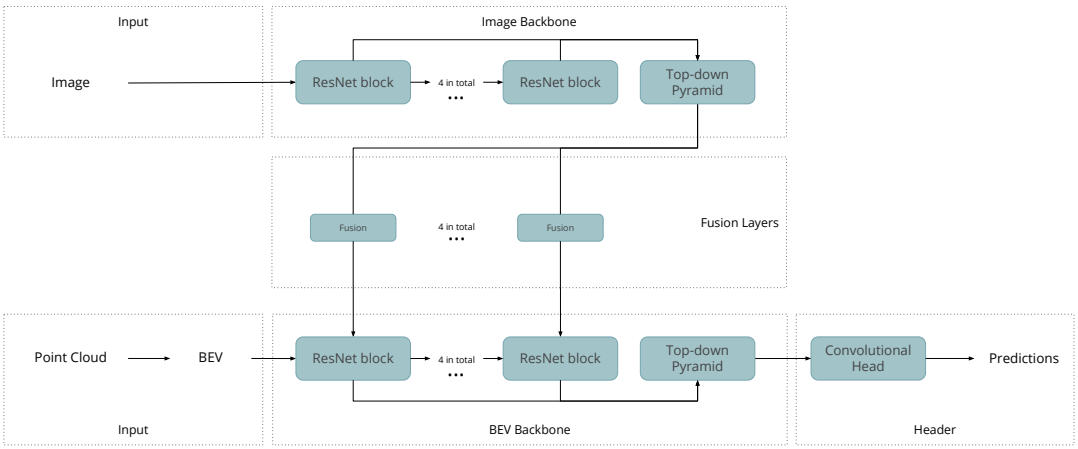


Figure 2.13: Overview of the Deep Continuous Fusion architecture.

2.4.4 Performance Comparison

Table 2.1 compares the performance of the architectures presented in the sections above. All results are from the 3D object detection task on KITTI's test split. Note that the architectures missing values have not evaluated their method on 3D object detection on the test split. In most cases this is due to focusing on detections in BEV instead of 3D. As can be seen in the table, there is a large gap between LIDAR and camera 3D object detection models. The AP numbers are evaluated for IOU 0.7, and divided into three sections: easy, moderate, and hard.

Table 2.1: Comparison of state-of-the-art architectures from the previous sections on KITTI's test split. Note that these have been extracted from the individual papers, with some exceptions where they were taken from KITTI's webpage [5], which are marked with an asterisk. Some publications do not have reported metrics for 3D object detection on the test split.

Input	**Name**	**AP IOU 0.7**		
		Easy	**Mod.**	**Hard**
lidar	Pixor [28]	-	-	-
	Pixor++[36]	-	-	-
	VoxelNet [34]	77.47	65.11	57.73
	PointPillars [35]	79.05	74.99	**68.30**
	LaserNet [36]	-	-	-
	PointRCNN [33]	**84.32**	**75.42**	67.86
mono	MonoFusion [40]	7.08	5.18	4.68
	MonoGRNet [37]	**11.29**	**12.90**	**11.34**
	Pseudo-Lidar [2]	-	-	-
fusion	FPointNet [39]	81.20	70.39	62.19
	PointFusion [42]	77.92	63.00	53.27
	MV3D* [43]	71.09	62.35	55.12
	AVOD [29]	81.94	**71.88**	**66.38**
	DeepCont* [8]	**82.54**	66.22	64.04

3

Architecture

The requirement on the final architecture is that it should work with three different *input combinations*: camera, LIDAR, and fusion. This chapter will present the final architecture that accomplishes this and explain its various components in detail.

3.1 Overview

The previous chapter introduced a number of approaches to object detection, both with a single sensor and through fusion. However, none of the fusion architectures explicitly address the problem of dealing with sensor failure and are therefore prone to be overly reliant on one of the sensors. The goal with this architecture is to take some of the best parts of previous works and combine them in a way that is robust against sensor failure.

An important related work for this architecture is Deep Continuous Fusion [8] with its comparatively simple convolutional feature extractors combined with a fine grained fusion mechanism. However, the mapping from image space to 3D is entirely dependent on the LIDAR data, which is an obvious issue if that sensor fails. Instead, the goal is to have an approach which conceptually is more similar to AVOD[29], where the fusion is performed not from one sensor to the other but rather as a transformation of both into a common representation. In the case of AVOD, this is done by projecting the 3D point cloud into a 2D BEV representation which can be fused with the inherently 2D image. This dimensionality reduction has the potential downside of information loss. Additionally, the feature maps can be misaligned even though they have the same dimensionality, for example the BEV is a top view of the scene whereas the image is a front view.

To solve these issues with Deep Continuous Fusion and AVOD, this architecture raises the image representation from 2D to 3D, without any help from the LIDAR data. To do this, an approach similar to Pseudo-Lidar[2] is employed. The main stages of the architecture are shown in Figure 3.1 can be summarized as:

- **Image to Cloud Estimation** - Convolutional *Feature Extraction* is performed on the image, followed by a *Depth Head* to output a depth map. This map is then unprojected into a 3D *Pseudo Point Cloud* with attached image features.

- **Cloud Combination** - The real and pseudo point clouds are used as input to a *Learned BEV Discretization* module. This results in a *Combined BEV Representation* in the form of a 2D grid with feautures as the third dimension. This is done for multiple scales and used in the next step as input and a form of middle fusion.
- **Object Detection** - The BEV maps are processed in another convolutional *Feature Extraction* network, followed by a simple *Head* which outputs confidence scores and bounding box regression parameters, in a dense one-shot fashion.

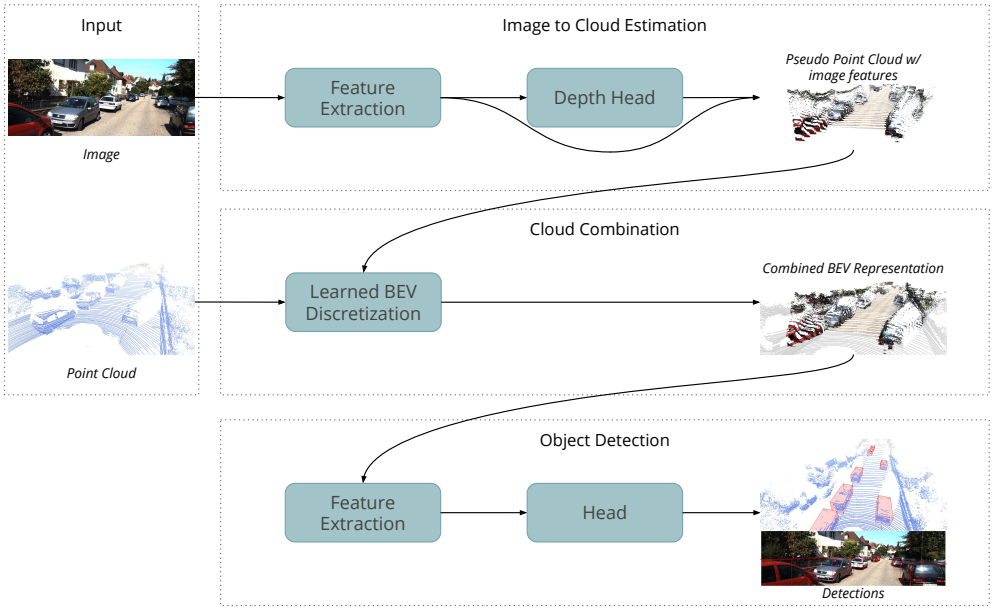


Figure 3.1: Visual representation of the implemented sensor failure robust architecture.

3.2 Image to Cloud Estimation

One of the key components in treating both sensors equally, and therefore being robust to any failures, is learning how to project the 2D image data to 3D. State-of-the-art monocular depth estimation has reached very high performance[38], and can therefore serve as a good approximation of this mapping. In practice this is done by first performing convolutional *Feature Extraction* on the image. The resulting feature maps are then used in the *Depth Head* to output a per pixel depth regression parameters. Finally, each pixel is unprojected to a 3D *Pseudo Point Cloud* using the depth values and camera calibration.

3.2.1 Feature Extraction

Following Deep Continuous Fusion [8], a lightweight ResNet-18 is used to process high resolution images into smaller high-dimensional feature maps. The final feature map is downscaled by a factor of 32.

A smaller feature map is very useful for computational reasons as well as for increasing the receptive field of the network. However, it can lose some of the finer details of the input. In order to recapture the details, a modified Feature Pyramid is used to reduce the output downscaling to a factor of four compared to the input. Instead of using upsampling and convolutional layers, as the original Feature Pyramid Network [4], deconvolutional layers are used which is inspired by Pixor [28]. A visualization of the image feature extraction can be seen in Figure 3.2.

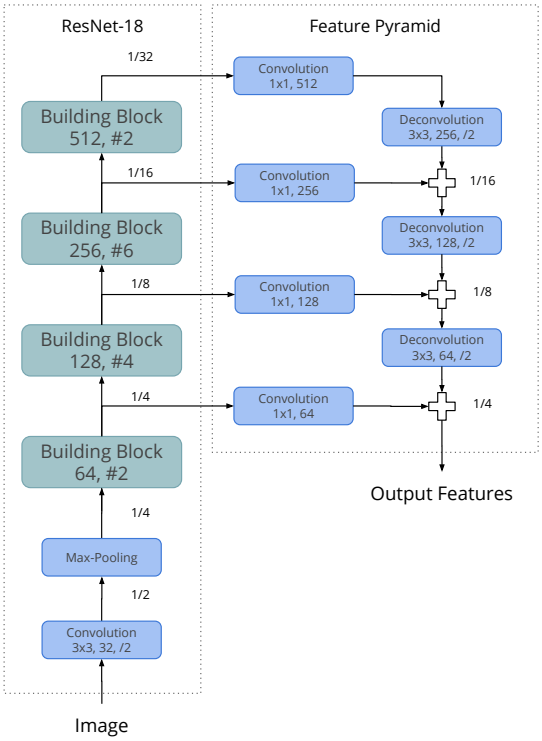


Figure 3.2: The image feature extractor, which is a combination of ResNet-18 [3] and a modified Feature Pyramid [4].

3.2.2 Depth Head

There are several possible ways to train the depth estimation network. In principle, the loss signal from the object detection task could be enough to learn the depth. However, this requires that the entire process is differentiable, and could have issues converging. An easier alternative is to use an auxiliary depth loss to provide a clear training signal. The ground truth is generated by projecting the LIDAR point cloud into the image to create a sparse depth map. An example of this can be seen in Figure 3.3.

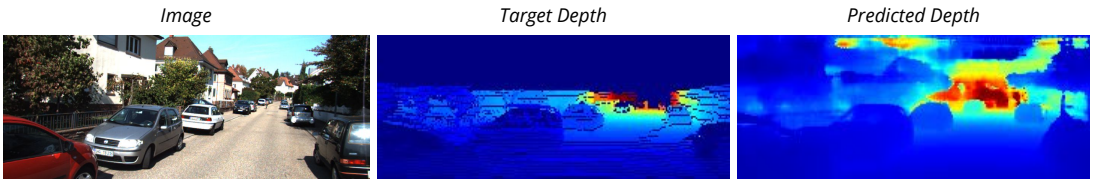


Figure 3.3: The depth network uses images as input and LIDAR as sparse ground truth to produce dense depth maps.

The target encoding and loss formulations are based on Deep Ordinal Regression [38], which achieves the highest score on the KITTI depth estimation benchmark out of all the methods that use monocular camera images[5]. As opposed to most methods, which regress a continuous depth value, the depth is discretized into several *intervals*. This changes the problem from regression to a form of multi-label classification, with the motivation that such a problem is easier to solve. The equation for discretizing the distance from α to β into K uniform thresholds is:

$$t_i = \alpha + (\beta - \alpha) * \frac{i}{K} \tag{3.1}$$

When encoding a depth value, each interval is defined as the probability that the point is further away than the distance of this interval. So the ground truth encoding of the distance α is all zeros, whereas anything further away than β, is all ones. Decoding this type of vector is done by using t_i as the depth value, where i is the number of interval probabilities above a certain threshold.

To produce this probability vector for each pixel, the head consists of a few 1x1 convolutional layers to output $2 * K$ values. These are then formed to an output of size K using pair-wise softmax:

$$\mathcal{P}_i = \frac{e^{y^{2i+1}}}{e^{y^{2i}} + e^{y^{2i+1}}} \tag{3.2}$$

where y is the network output, and i is the threshold index. The pixel location (u, v) is excluded for brevity.

Extending this to a pixel-wise loss would mean

$$\Psi(u, v) = \sum_{k=0}^{l-1} \log(\mathcal{P}_i) + \sum_{k=l}^{K-1} \log(1 - \mathcal{P}_i) \tag{3.3}$$

which is the cross-entropy loss for all correct and incorrect predicted intervals. Finally the total loss can be defined as the average of all pixel-wise ordinal losses as

$$\mathcal{L} = -\frac{1}{N_{gt}} \sum_u \sum_v \Psi(u, v). \tag{3.4}$$

where N_{gt} is the number of pixels that have a ground truth. All pixels without a ground truth are ignored during loss computation.

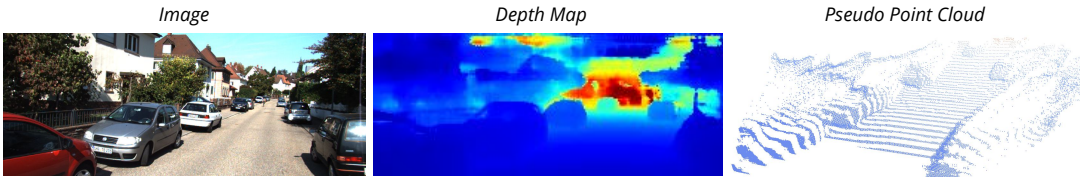


Figure 3.4: The depth estimation module uses a mono camera image, produces a depth map, which is unprojected it into a pseudo point cloud.

3.2.3 Pseudo Point Cloud

The next step is to convert each pixel in the depth map to a 3D point to generate a Pseudo Point Cloud. This is only possible with additional information about the camera's intrinsic calibration. More precisely, a pixel (u, v) can be transformed to a 3D point (x, y, z) in the camera coordinate system as follows:

$$z = D(u, v) \tag{3.5}$$

$$x = \frac{(u - u_c) * z}{f_u} \tag{3.6}$$

$$y = \frac{(v - v_c) * z}{f_v} \tag{3.7}$$

$$\tag{3.8}$$

where D is the depth estimation network, (u_c, v_c) is the camera center and (f_u, f_v) are the horizontal and vertical focal lengths respectively. Lastly, the points are transformed from the camera coordinate frame by multiplying with the camera-to-LIDAR calibration matrix. This is necessary to align the unprojected cloud with the real point cloud.

The resulting pseudo point cloud can be seen in Figure 3.4. In addition to the $(x, y,)z$ spatial coordinates, each point contains the image features from its corresponding pixel position in the final feature map from the Image Feature Extractor. This is a straightforward one-to-one mapping since the depth map has the same dimensions as the image feature map.

3.3 Cloud Combination

This section explains how the information from both sensors is merged into a common representation. This is done by separately applying a *Learned BEV Discretization* encoder to the real and pseudo point clouds and then merging them to a single, high resolution, BEV feature map. Additionally, following Deep Continuous Fusion [8], the Pseudo Cloud is used to generate multiple smaller feature maps to allow for more powerful multi-scale image feature fusion. See Figure 3.5 for a visualization of this combination approach.

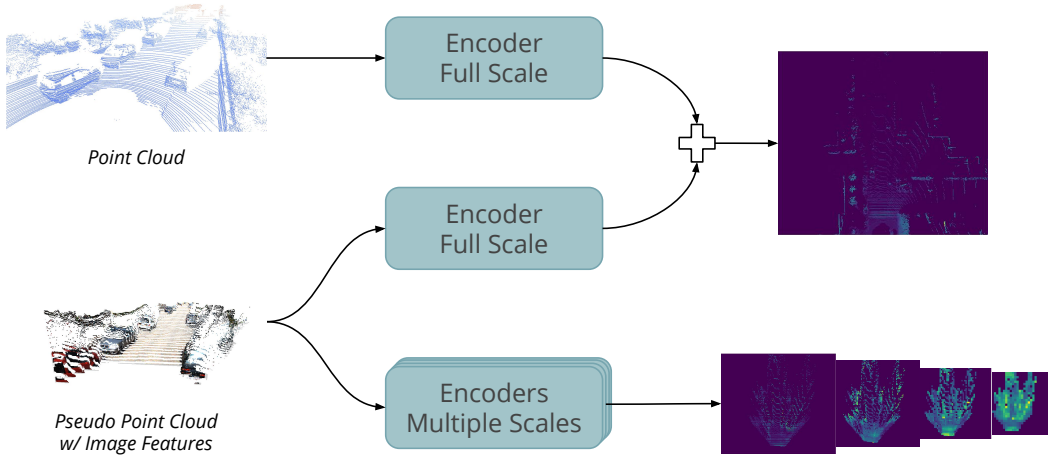


Figure 3.5: The process of merging the pseudo and real point clouds into a common BEV representation. The pseudo cloud is used to generate multiple additional smaller feature maps for more powerful image feature fusion.

3.3.1 Learned BEV Discretization

Chapter 2 introduces a number of ways to extract features from a point cloud. One of the easiest is to discretize it into a BEV grid based on cell occupancy. However, that loses a lot of information, such as the image features. A more flexible way is to process the point cloud directly using some point-wise feature extractor such as PointNet++[44]. However, those methods are not as mature, and more complicated to implement. Therefore, the chosen approach lies in between these approaches in that a point-wise encoder is used to create a discrete BEV representation. This is very similar to the approaches of VoxelNet [34] and especially PointPillars [35].

First, the 3D space is divided into sections that correspond to the discrete cells of a BEV map, called "pillars". Then, all the points are segmented into small groups based on which pillar they belong to. All points receive additional features that are specific to their pillar: the offset to the pillar center, and the difference to the mean feature of all points in the pillar. The offset to the pillar center allows the network to learn local spatial relationships within the pillar, whereas the mean difference allows for interaction between points. To summarize, each point is described as $(\mathbf{f}, \mathbf{f} - \hat{\mathbf{f}}, \mathbf{x} - \mathbf{x_c})$, where $\mathbf{f}$ are the original point features, including the location $\mathbf{x}$ and reflectance or image features; $\hat{\mathbf{f}}$ are the average features of all points in the pillar; and $\mathbf{x_c}$ is the pillar center. A simplified schematic of this can be seen in Figure 3.6.

Once all points in a pillar have received their additional features, they are processed jointly and combined into a single feature vector using a simplified version of PointNet [27]. Each point is passed through an MLP to extract high level point-wise features. Then, the features of all points in the pillar are aggregated using averaging.

As a last step, the encoded pillar features are scattered back into the BEV feature

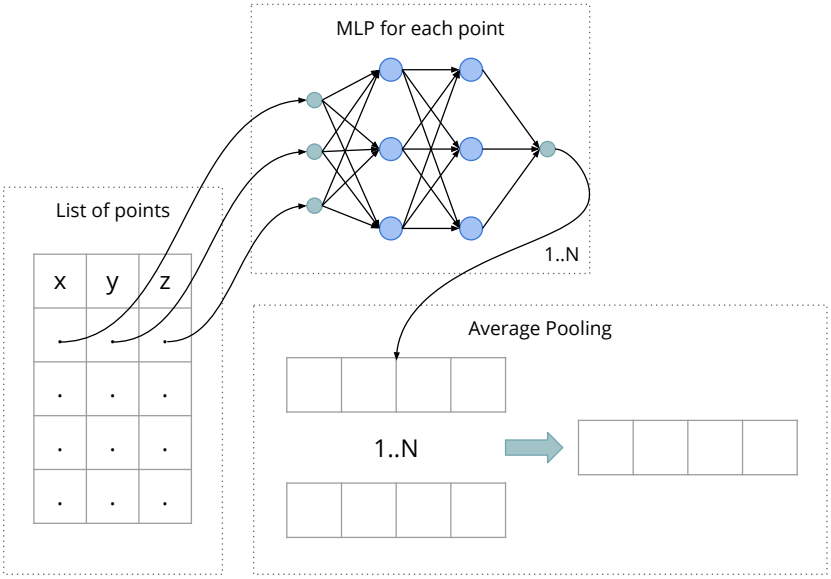


Figure 3.6: Schematic of the PointPillar approach where each point is fed through an MLP before averaging to create the final feature vector. Note that the list of points can be of any dimensionality and include other features than spatial coordinates.

map. Note that many pillars do not contain any points, in which case the features are all set to zero. The final result is a 3D tensor with dimensions $W \times H \times C$, where W and H correspond to the spatial dimensions, and C the feature dimension.

3.4 Object Detection

The last step in the architecture is to perform the object detection. To that end the multi-scale BEV feature maps are processed by a convolutional feature extractor. This is followed by a simple head that outputs dense, one-shot predictions, which are used to train the network end-to-end using backpropagation.

3.4.1 Feature Extractor

The outputs of the *Cloud Combination* module are a set of discrete feature maps which can be treated as images and processed by a standard convolutional feature extractor. The chosen network is heavily inspired by PIXOR [28] and Deep Continuous Fusion [8] and is a modified ResNet architecture which is claimed to be more robust to overfitting [20]. To reduce information loss, the initial max-pooling layer is removed so that the input is only downsampled by a factor of 2 instead of 4. This is followed by four residual groups consisting of $(2, 4, 6, 6)$ ResNet Building Blocks each. The channel dimensions in these groups are $(64, 128, 192, 256)$ respectively.

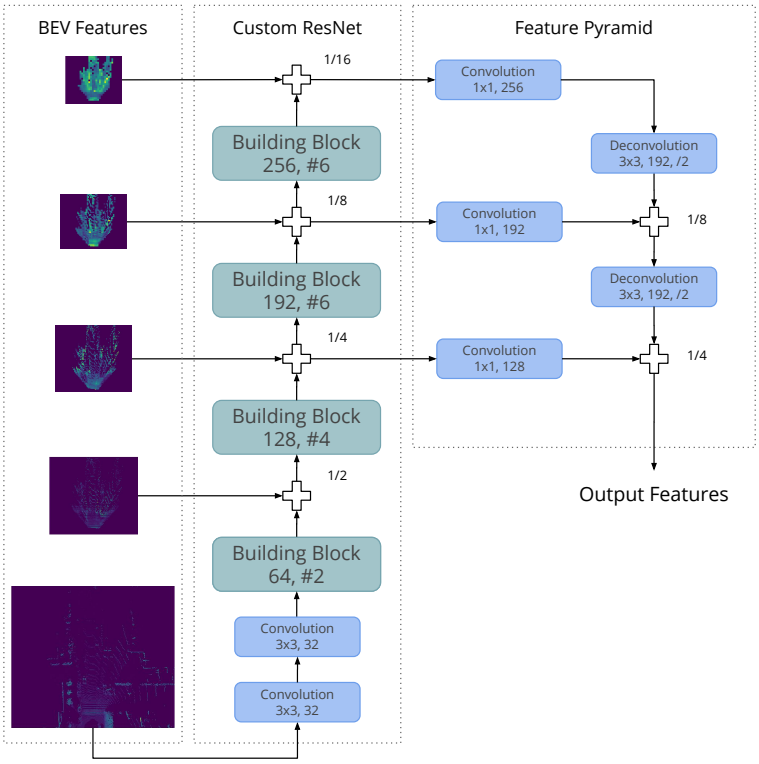


Figure 3.7: The implemented BEV feature extractor is a custom ResNet architecture with a top-down multi-scale feature combination strategy inspired by the Feature Pyramid Network [4]. Multiple BEV feature maps of different scales are added after each group of Building Blocks in a form of middle fusion.

The largest feature map is used as the primary input to the extractor. To further exploit the image information, smaller feature maps are fused after each residual group. As a last step, multi-scale features are aggregated with the same type of Feature Pyramid as in the Image Feature Extractor, see Section 3.2.1. A visualization of the entire BEV Feature Extractor can be seen in Figure 3.7.

3.4.2 Detection Head

In order to convert feature maps to detections, a single 1×1 convolutional layer is used to output per pixel confidence scores and regressions. The score is the probability that the cell contains a car, whereas the regression values describe the location, rotation, and shape of the bounding boxes. The exact form of this encoding, as well as the loss function, has a high impact on the training of the network.

In the case of dense 3D object detection, each output position corresponds to a number of predefined boxes, called anchors. These can vary in both size and rotation, and each class usually has its own anchors. In this architecture there is only a single anchor corresponding to a typical car that is facing forward.

Given the location, rotation and size of an anchor, the network regression targets

are encoded against that anchor as:

$$p_{(x,y,z)} = l_{(x,y,z)} - a_{(x,y,z)} \tag{3.9}$$

$$p_{(l,w,h)} = \log \frac{l_{(l,w,h)}}{a_{(l,w,h)}} \tag{3.10}$$

$$p_\theta = [\cos 2\Delta_\theta, \sin 2\Delta_\theta], \quad \Delta_\theta = l_\theta - a_\theta \tag{3.11}$$

where p is the prediction target, l is the ground truth label and a is the anchor. (x, y, z) are the bounding box center coordinates, (l, w, h) are the length, width and height, and θ is the orientation. One reason for encoding the dimensions using the natural logarithm is that the distribution of sizes is inherently skewed, since they are always positive. The logarithm shifts this so that any network output corresponds to a physically possible size. As long as the size of the vehicles does not vary drastically, the encoded values will also have small magnitudes, which is usually a desirable property. There are of course other transformations that accomplish this, but the logarithm is simple and used in many state-of-the-art 3D object detection architectures [29, 8, 35].

Encoding the rotation using cosine and sine helps stabilize the training due to the fact that angles are periodic, for example π and $-\pi$ represent the same angle while the values differ significantly. Additionally, multiplying the angle by two, changes the periodicity from 2π to π, and therefore does not encode the facing direction. This reduces the search space and has no downside in the concept of Average Precision.

The total loss is a weighted combination of the regression and classification losses:

$$L_{tot} = L_{cls} + \alpha L_{reg} \tag{3.12}$$

In order to handle the large imbalance between the number of positive and negative targets, binary cross entropy focal loss [30] is used for classification. The formulation is:

$$L_{cls} = \frac{1}{N} \sum \alpha_t (1 - p_t)^\gamma \log p_t \tag{3.13}$$

$$p_t = \begin{cases} p & \text{if } y = 1 \\ 1 - p & \text{otherwise} \end{cases} \tag{3.14}$$

$$\alpha_t = \begin{cases} \alpha & \text{if } y = 1 \\ 1 - \alpha & \text{otherwise} \end{cases} \tag{3.15}$$

where N is the total number of targets, positive and negative, y is the label, p is the prediction and α, γ are hyper-parameters.

The regression term is the sum of the smooth L1-loss over all positive targets. The negative targets are ignored since there is no object to regress against:

$$L_{reg} = -\frac{1}{N_{pos}} \sum_{k \in (x,y,z,l,w,h,\theta_1,\theta_2)} \begin{cases} 0.5(p_k - l_k)^2 & \text{if } |p_k - l_k| < 1 \\ |p_k - l_k| - 0.5 & \text{otherwise} \end{cases} \tag{3.16}$$

where N_{pos} is the number of positive targets, p_k are the predicted regression parameters, l_k are the labels and the sum only goes over the positive targets. θ_1 and θ_2 are the two rotational regression parameters from the cosine-sine encoding described above.

4

Method

4.1 Data

A widely used dataset for autonomous driving perception models is KITTI [21]. Most publications include performance metrics on this dataset, which makes it easy to make comparisons when evaluating a particular architecture. However, a downside with KITTI is that it is quite small, which can be a problem when training deep and complicated models. Another, very new, dataset is NuScenes [22], which is more extensive, but much less tested on different architectures. Table 4.1 contains a comparison of the two datasets. It can be very beneficial to use multiple datasets to be able to test transferring capabilities between them, which may indicate how well the architecture can generalize.

The datasets use *.png* format for images, and *.bin* for LIDAR scans. In addition to the actual data from the sensors, there are calibration matrices to go between difference coordinate systems. The files along with calibration matrices are loaded using a data loading class, which keeps track on all sample ID:s to be able to generate batches in random order. The data loader uses lazy loading for the files since keeping everything in memory is not feasible. When a batch has been loaded, it is fed through a couple of preprocessing steps, such as construction and normalization of input and target tensors.

4.2 Training

The training procedure consists of three steps: preprocess the input data; apply data augmentation techniques; and simulate sensor failures. This is followed by a forward- and backpropagation, and finally updating weights of the network.

4.2.1 Preprocessing

Since image size and number of points can vary between samples, the images are cropped and resized, and the point clouds are padded to equal lengths. The size of the images depend on which dataset is used, and is usually the smallest common size.

Table 4.1: Comparison of Kitti and NuScenes datasets.

Property	**KITTI**	**NuScenes**
Locations	Karlsruhe, Germany	Boston, US & Singapore, Singapore
Weather	Sunny and clear.	Various conditions
Time	Daytime	Both day and night
Environment	Rural and highway areas	"Aim for a diverse set of locations" [22]
Annotations	3D box with BEV rotation, 8 classes, only objects in camera view	3D box with rotation quaternion, 25 classes, all objects in scene
Sensors	Stereo front cameras (1392x512), 3D LIDAR (10Hz, 26.8 deg vertical), GPS & IMU	Six surround visual cameras (1600x900), 3D LIDAR (20Hz, 40 deg vertical), 5 radars, GPS & IMU
Synchronization	Cameras trigger when LIDAR sweep ends	Individual cameras trigger when LIDAR sweep passes by them
Dataset size	7481 annotated frames	40000 annotated frames

The reflectance value for the point cloud is bounded by $[0, 1]$ and the pixel values of the image are normalized by the maximum value, i.e. 255. The point cloud spatial coordinates are left unnormalized. The targets for the bounding box regression residuals are normalized with the anchor sizes.

When building the target maps for the detection network, there is a positive signal for ground truths, negative where there are no targets, and finally ignore areas where there can not be any annotations. In the case of KITTI, there are no annotations outside the camera view, thus the ignore region follows the outside of the frustum.

4.2.2 Data Augmentation

Data augmentation is critical when training with relatively small datasets, such as KITTI [36]. In this case the augmentation was performed on both the point cloud and image data. Below is an explanation of the various augmentation methods.

The point cloud is augmented by first performing a uniformly random translation of all three coordinates within $x \pm 5$ m, $y \pm 5$ m, $z \pm 0.25$ m. Then the entire point cloud is rotated around the z-axis by ± 5 degrees. After this, the entire cloud is re-scaled by multiplying the three coordinates with the same random scaling factor of 1 ± 0.15.

The image augmentation is similarly described by a random translation and scaling, but without rotation. In practice this means cropping an area of the original image

Figure 4.1: An image from the KITTI dataset, zoomed in on the upper right corner with symmetrical padding of 20 pixels in both directions. The lines indicate the original image without padding.

of varying size and center location, and then reshaping it to the desired dimensions. If the selected region ends up outside of the original image, it is padded. The padding uses the symmetric strategy, which repeats the data closest to the edge in a mirror fashion, as can be seen in Figure 4.1. This approach attempts to avoid creating hard unrealistic edges, but as seen in the image it can create other unrealistic artifacts. Other alternatives such as zero-padding, i.e. adding black pixels, have not been explored. Finally, there is also the possibility of enabling color augmentation, which uses a scaling factor of $0.8 - 1.2$ for each color channel.

4.2.3 Simulated Sensor Failure

Following the comparison presented in Section 2.4.4, LIDAR architectures vastly outperform image architectures. Thus, when always provided with LIDAR data during training, the architecture can learn to fully rely on that sensor, and ”ignore” the camera. Another possibility, is a complete sensor codependence where the network looks for different features in the image and point cloud and becomes severely crippled even if only the camera fails. To counteract these scenarios, a training procedure with probabilistic sensor failure was added as a special form of data augmentation.

During pre-processing one of the following occurs:

- **camera failure**: the point cloud is unaltered and the image is blacked out, i.e all pixels are set to zero
- **lidar failure**: the image is unaltered and the point cloud is stripped of all points
- **no failure**: both the image and point cloud are unaltered

The idea is to expose the network to sensor failure during training so that it can

learn to become more robust against that scenario. This Simulated Sensor Failure is abbreviated with *SSF* in following sections. The probabilities for each case can be varied, and do not have to be equally distributed.

4.3 Evaluation

Since KITTI was released in 2012, there are a lot of publications and architectures with evaluation metrics using this dataset. Publications either submit their inference results on the hidden test split and submit to the KITTI benchmark, or evaluate on a validation split.

The source code for KITTI's evaluation is available on their homepage [5]. Given directories for the outputs and ground truths, the code will go through all objects and calculate the overall average precision for three difficulty levels: easy, moderate, and hard. An object's difficulty is decided by three criteria: 2D height in the image, occlusion level, and truncation ratio. This means that if an object is small, or not entirely visible, it can be labeled as hard.

In the evaluation cycle, the network runs inference three times and produce text files that can be read by the KITTI evaluation code. The three runs use different input combinations: camera, LIDAR and fusion. KITTI does not contain annotations outside the camera view where there is LIDAR data and thus detections can occur. Therefore, all detections whose center points fall outside the image are removed. This could however in some cases hurt the results, since some ground truths are included even though their center points are outside.

During training there are a couple of metrics that are of interest. For easy overview, a training summary with the follow statistics is produced:

- separate losses, one for each objective, i.e. depth estimation, bounding box regression and classification
- total loss
- average precision on the validation split
- qualitative depth maps
- qualitative visualization of detections in image and BEV

Note that the losses are displayed in graphs over epochs and also separated based on the input combination.

4.4 Implementation

The architecture is written in Python using the TensorFlow [45] machine learning library. The actual components, such as feature extractors and headers, are implemented using TensorFlow's objective oriented API, as described in Chapter 3.

The training and inference of the architecture was performed on Nvidia V100 GPUs, which come in both 16GB and 32GB versions.

5

Experiments

In order to investigate the final architecture, a range of experiments are performed. KITTI is used to evaluate the model and compare it to the current state of the art in monocular camera, LIDAR, and fusion object detection. Additionally, an ablation study is performed to investigate training procedures and potential generalization benefits of sensor failure robustness. Finally the model is evaluated on NuScenes, to see how well it generalizes to another sensor setup and environment.

5.1 KITTI

KITTI contains 7481 samples with 28739 annotated cars which are divided into two roughly equal parts for validation and training. For the 3769 validation samples, there are 14385 annotated cars. To make sure that the sets are as uncorrelated as possible, a common split [8] is used which ensures that all samples from the same driving sequence are in the same set. This also makes it possible to compare validation metrics with other publications. Additionally, KITTI contains equally many samples without public labels, which is called the test split.

The training details for pre-processing and augmentation are described in Section 4.2. The depth estimation network is pre-trained for 100 epochs using Adam with an initial learning rate of 0.0003, which is decayed in steps to a final value of 0.000001. Then, the full model is trained for 80 epochs using Adam with an initial learning rate of 0.0003, which is also decayed to a final value of 0.00001. The model is trained on a single GPU with a batch size of 5.

5.1.1 Results

The evaluation results for the model trained with simulated sensor failure, SSF, are shown in Table 5.1. Each input combination is compared against a few state of the art models, according to the KITTI 3D object detection benchmark [5]. As can be seen in the table, the results for the camera input is in between the state of the art for IOU 0.5, except for the easy difficulty which is two AP points lower. The camera results for IOU 0.7 are inconclusive. The LIDAR input results are close to those of VoxelNet [34], however, significantly behind PointRCNN [33]. Finally, the

Table 5.1: Comparison of our method against the current state of the art on the validation split. The term "ours" refers to the sensor failure robust model, averaged over five trainings. Some values are missing since the authors did not include them in their papers. Pseudo-Lidar [2] was excluded from this comparison due to training the depth estimation network on images in the validation and test split.

	IOU 0.5			**IOU 0.7**		
	Easy	**Mod.**	**Hard**	**Easy**	**Mod.**	**Hard**
Ours, Camera	46.01	32.58	29.72	9.80	7.72	**11.06**
MonoGRNet [37]	**50.51**	**36.97**	**30.82**	**13.88**	**10.19**	7.62
MonoFusion [40]	47.88	29.48	26.44	10.53	5.69	5.39
Ours, LIDAR	96.84	89.57	88.78	82.33	67.98	65.77
VoxelNet [34]	-	-	-	81.97	65.46	62.85
PointRCNN [33]	-	-	-	**88.88**	**78.63**	**77.38**
Ours, Fusion	96.75	89.77	89.08	84.50	72.92	67.34
Deep Cont. [8]	-	-	-	**86.32**	73.25	67.81
AVOD [29]	-	-	-	84.41	**74.44**	**68.28**

fusion results are close to, but slightly behind, both Deep Continuous Fusion [8] and AVOD [29].

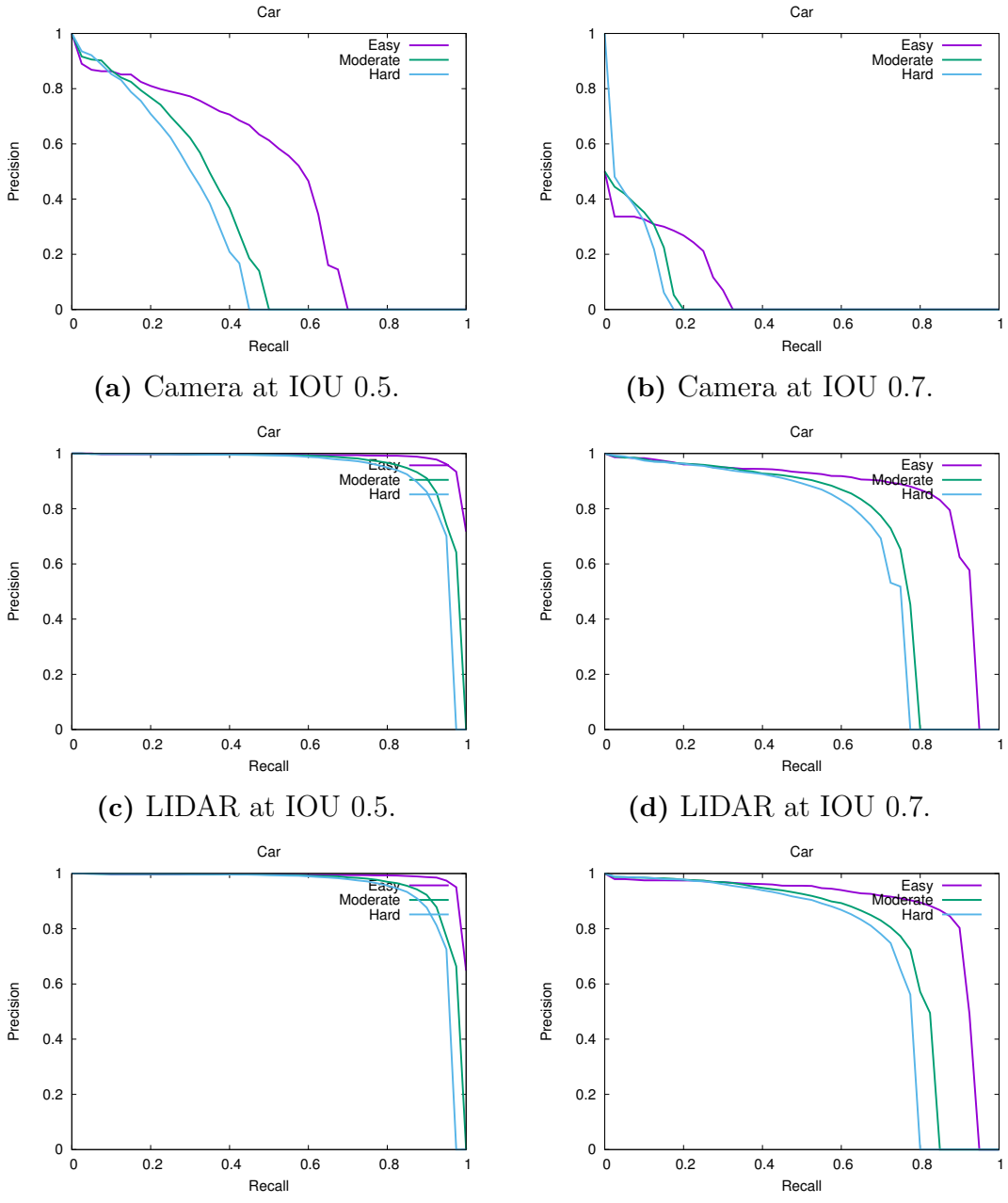
Next, Figure 5.1 shows the precision recall curves that are used to compute the Average Precision. These curves can provide insight into how the precision of the model degrades as the detection threshold is lowered. As can be seen, the curves for LIDAR and Fusion are very similar for both IOU thresholds, while Camera is considerably lower.

The previous evaluation treats all detections equally, which is not necessarily reasonable in the context of autonomous driving. It is much more important to detect nearby vehicles, since those are primarily the ones being reacted to. Therefore, the detections are split up based on the distance from the vehicle along the front facing axis. The Average Precision evaluation is performed separately on these splits and shown in Table 5.2. The easy and moderate difficulty levels were discarded to keep the amount of presented data reasonable. There is a clear trend that the performance decreases significantly for objects that are far away, especially for the camera case.

5.1.2 Ablation study

Here, different versions of the model are compared against each other to investigate the impact of various design decisions.

The first investigation concerns the necessity of training with SSF. To do this, two identical models are trained: one with SSF, and one with full sensor availability. Table 5.3 shows the results when both models are evaluated on all three input



(a) Camera at IOU 0.5.

(b) Camera at IOU 0.7.

(c) LIDAR at IOU 0.5.

(d) LIDAR at IOU 0.7.

(e) Fusion at IOU 0.5.

(f) Fusion at IOU 0.7.

Figure 5.1: Precision-Recall curves for the failure robust model, evaluated with all three input combinations, for both 0.5 and 0.7 as IOU threshold.

Table 5.2: Model AP performance for different object ranges on hard difficulty. The results are averaged over five trainings. The number of annotated cars in each range is: 3964, 5051, 4049, 1321, and 14385, from left to right starting with range 0-15 meters.

Input	IOU 0.5					IOU 0.7				
	0-15	**15-30**	**30-50**	**50-**	**0-**	**0-15**	**15-30**	**30-50**	**50-**	**0-**
camera	60.35	21.92	8.43	0.04	29.72	19.91	4.90	3.04	0.00	11.06
lidar	98.84	89.10	76.95	19.52	88.78	86.54	72.29	39.24	8.12	65.77
fusion	98.87	89.16	77.13	25.40	89.08	87.38	75.05	44.99	4.57	67.34

Table 5.3: Comparison of the model's AP when training with and without simulated sensor failure, abbreviated with *SSF*. The results are averaged over five trainings.

Input Data		IOU 0.5			IOU 0.7		
Eval.	**Training**	**Easy**	**Mod.**	**Hard**	**Easy**	**Mod.**	**Hard**
camera	fusion	0.00	0.01	0.01	0.00	0.00	0.00
	SSF	**46.01**	**32.58**	**29.72**	**9.80**	**7.72**	**11.06**
lidar	fusion	93.78	83.56	78.86	61.10	46.79	42.78
	SSF	**96.84**	**89.57**	**88.78**	**82.33**	**67.98**	**65.77**
fusion	fusion	**97.50**	89.67	88.88	81.52	68.63	65.61
	SSF	96.75	**89.77**	**89.08**	**84.50**	**72.92**	**67.34**

combinations. These results clearly show that without SSF, the model becomes overly reliant on having both sensors available and performs poorly when the camera fails, and not at all when the LIDAR fails.

To further explore the effects of SSF training, a series of models are trained with varying ratios of sensor failure. The most interesting cases are presented in Figure 5.2. As long as both sensors fail to some degree, the exact ratios have a minor impact. However, if the LIDAR never fails, the network is completely unable to perform camera detections, which is in line with the previous results in Table 5.3. Further, if the camera never fails, the network is able to perform some LIDAR detections, but much below the fusion case.

Next, the failure robust model is compared against three separate specialized networks. The four instances of the model are trained with the same hyper-parameters: one with SSF and three with the respective input combination. The results in Table 5.4 show that the network trained with SSF slightly outperforms the specialized models in most categories.

Additionally, in order to investigate the potentially regularizing effects of sensor failure training, Table 5.5 compares the results of the models on the training and validation splits. Here there is a clear trend with lower training scores and higher validation scores for the failure robust model.

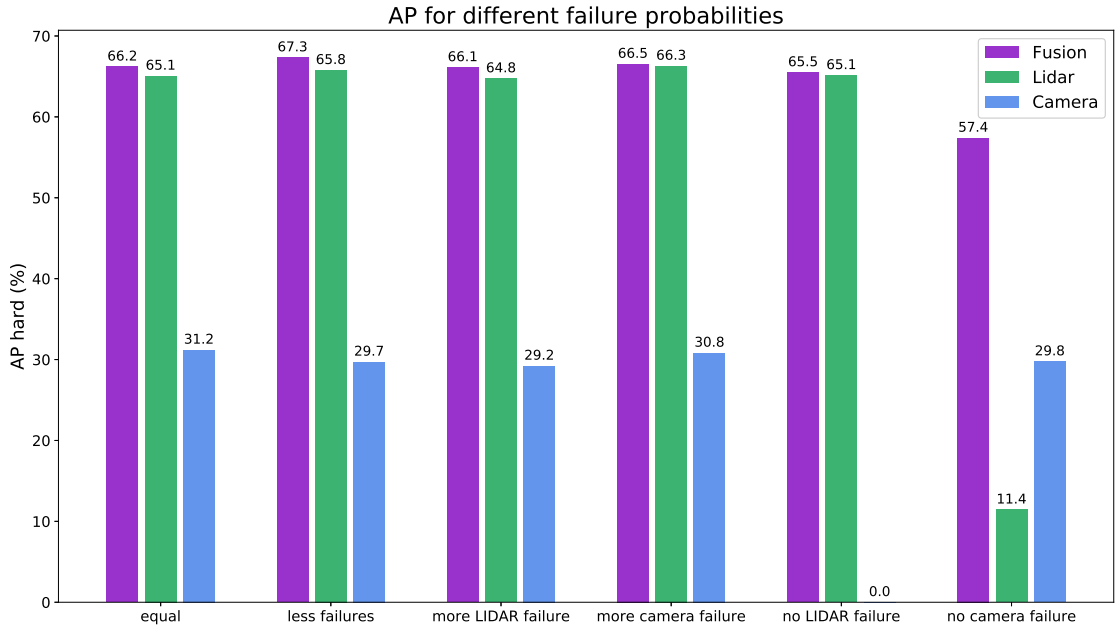


Figure 5.2: AP on KITTI hard validation split, LIDAR and fusion numbers use IOU 0.7 and camera uses IOU 0.5. The terminology for the ratios is as follows: "equal" is 33%/33%/33%; "more" is 50%/25%/25%; "no" is 0%/50%/50%.

Table 5.4: Comparison of the architecture's AP when trained with: only camera, only LIDAR, fusion, or SSF. The results are averaged over five trainings.

Input Data		**IOU 0.5**			**IOU 0.7**		
Eval.	**Training**	**Easy**	**Mod.**	**Hard**	**Easy**	**Mod.**	**Hard**
camera	camera	42.38	30.37	27.45	9.28	**9.89**	9.44
	SSF	**46.01**	**32.58**	**29.72**	**9.80**	7.72	**11.06**
lidar	lidar	**97.26**	89.04	88.31	**82.43**	66.67	65.20
	SSF	96.84	**89.57**	**88.78**	82.33	**67.98**	**65.77**
fusion	fusion	**97.50**	89.67	88.88	81.52	68.63	65.61
	SSF	96.75	**89.77**	**89.08**	**84.50**	**72.92**	**67.34**

Table 5.5: Five run average AP for training and validation. The compared models are trained with: camera, LIDAR, fusion, and simulated sensor failure.

Input Data		**IOU 0.5**		**IOU 0.7**	
Eval.	**Training**	**Train**	**Val.**	**Train**	**Val.**
camera	camera	**73.99**	27.45	**45.68**	10.36
	SSF	65.57	**29.72**	38.71	**11.06**
lidar	lidar	**99.76**	88.31	**99.66**	65.20
	SSF	97.92	**88.78**	89.99	**65.77**
fusion	fusion	**99.62**	88.88	**99.43**	65.61
	SSF	98.35	**89.08**	96.14	**67.34**

Table 5.6: Comparison of the model's AP when training on KITTI and evaluating on NuScenes, with and without simulated sensor failure. The results are averaged over five trainings.

Input Data		IOU 0.5			IOU 0.7		
Eval.	**Training**	**Easy**	**Mod.**	**Hard**	**Easy**	**Mod.**	**Hard**
camera	fusion	0.00	**1.82**	**1.82**	0.00	0.00	0.00
	SSF	**1.38**	1.38	1.38	**0.04**	**0.04**	**0.04**
lidar	fusion	**37.36**	30.68	**28.06**	**11.96**	**10.90**	**9.93**
	SSF	36.88	**31.06**	27.17	9.28	7.84	6.89
fusion	fusion	**35.51**	**29.68**	**26.06**	**12.72**	**10.99**	**10.24**
	SSF	33.14	27.59	24.43	9.63	8.39	7.43

5.1.3 Evaluation on NuScenes

To test how sensor failure robustness translates to a different dataset, a model trained on KITTI is evaluated on NuScenes without additional fine tuning. In order to increase the similarity of the datasets, both coordinate systems are transformed so that the LIDAR is in the center and the axes are aligned.

The 40000 annotated NuScenes samples are split into training and validation sets according to the official python development kit [22]. Here, only the data from the validation split is used, even though there is no training. This is to open up the possibility for comparisons with future experiments.

This evaluation also uses the official KITTI evaluation code in order to allow comparisons across the datasets. Since NuScenes contains objects all around the vehicle, and KITTI does not, the ones with center points outside the image are removed. Since the easy/moderate/hard categories are based on KITTI annotations, they are mimicked based on object visibility and truncation.

In practice, two models are trained on KITTI, as described in Section 5.1: one uses SSF, and the other always has both sensors available. Next, they are evaluated on the NuScenes validation split on the three different input combinations. The results are shown in Table 5.6 and as can be seen, the scores are low compared to the KITTI results. Additionally, SSF reduced the model performance, which is the opposite behaviour observed in the KITTI results in Table 5.3.

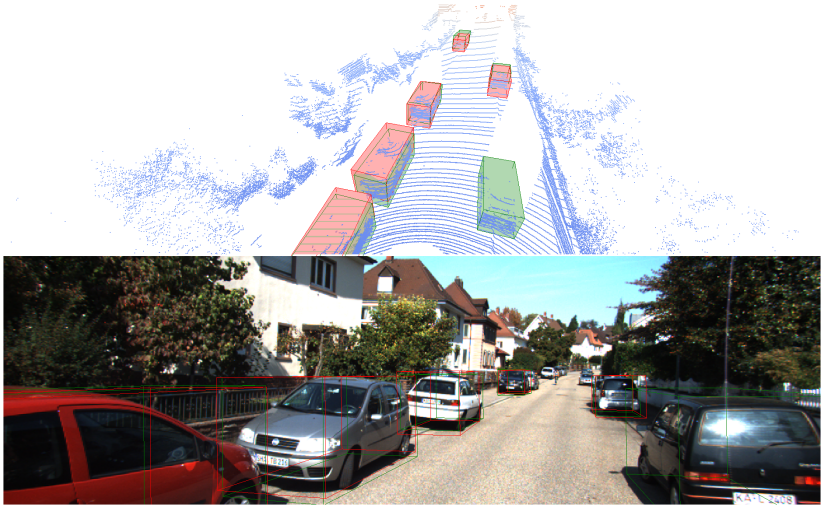
5.2 Qualitative Results

It can be difficult to get an intuitive understanding for how a model performs from numerical metrics such as Average Precision. Therefore, a qualitative evaluation is performed, where the detections are shown in both the point cloud and the image.

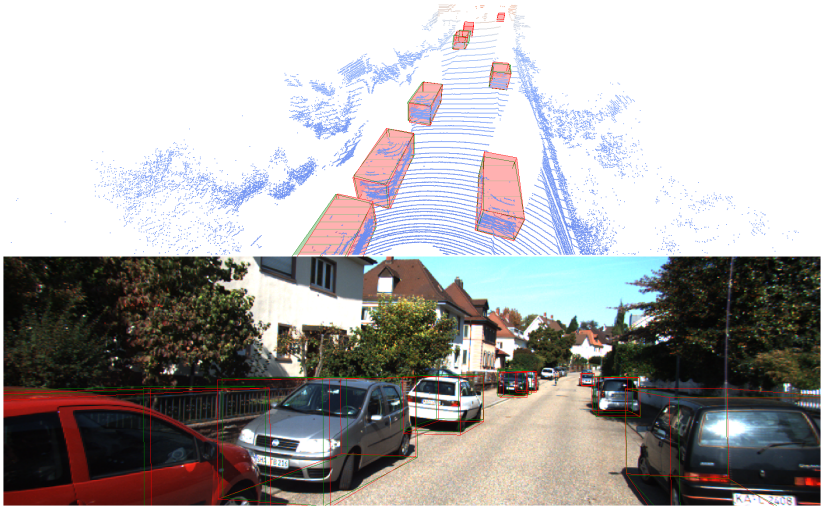
Figure 5.3 shows how the model performs in the different input combinations on a frame from the KITTI validation set. Next, detections on the KITTI test set

are shown in Figure 5.4 and 5.5, where all input combinations are included in the same plot. Figure 5.4 clearly shows the shorter range of the camera only case, while Figure 5.5 shows that on certain images, the modes can perform similarly to each other.

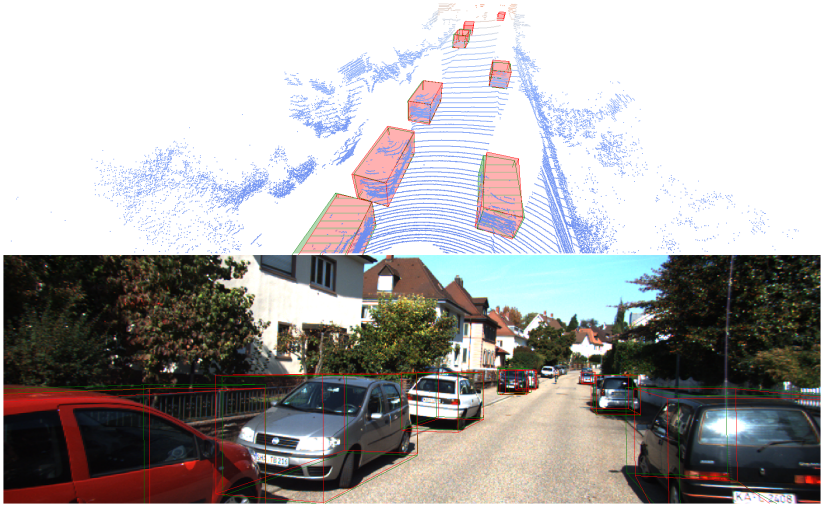
The performance of a model trained on KITTI and evaluated on NuScenes is shown in Figures 5.6 and 5.7. All three input combinations struggle to get correct boxes in Figure 5.6, which is in line with the low metrics in Table 5.6. However, Figure 5.7 all three input combinations manage to find a car at very high range.



(a) Camera



(b) LIDAR



(c) Fusion

Figure 5.3: Qualitative results on the KITTI validation set. Detections are in red, and ground truths in green.

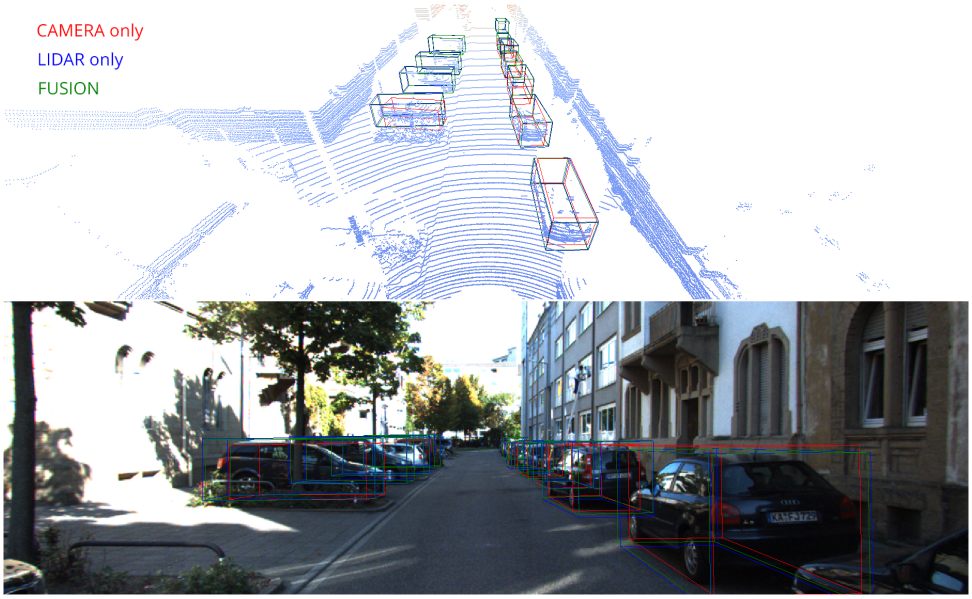


Figure 5.4: Inference results on sample 000059 from the KITTI test set. Each input combination is in a separate color.

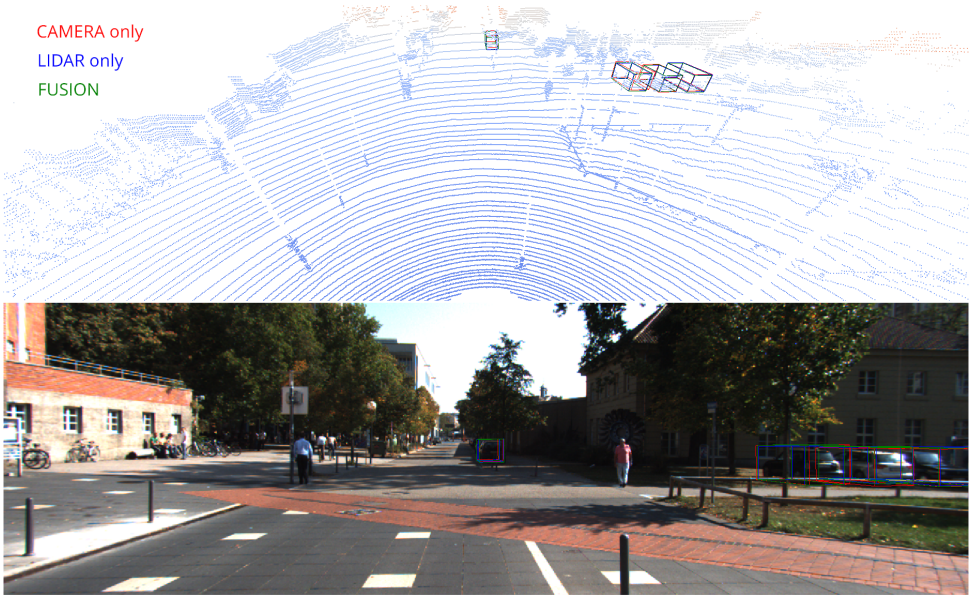


Figure 5.5: Inference results on sample 000063 from the KITTI test set. Each input combination is in a separate color.

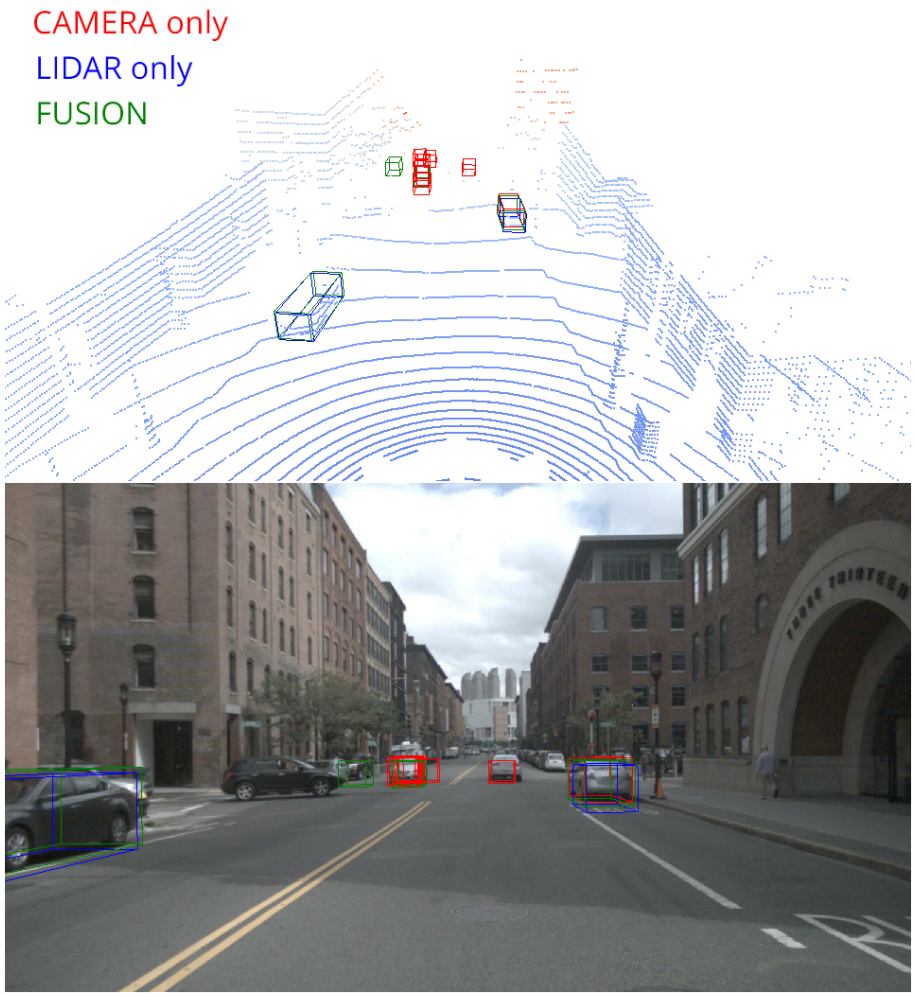


Figure 5.6: Inference on NuScenes validation split. Most cars are found, however, the precision is low.

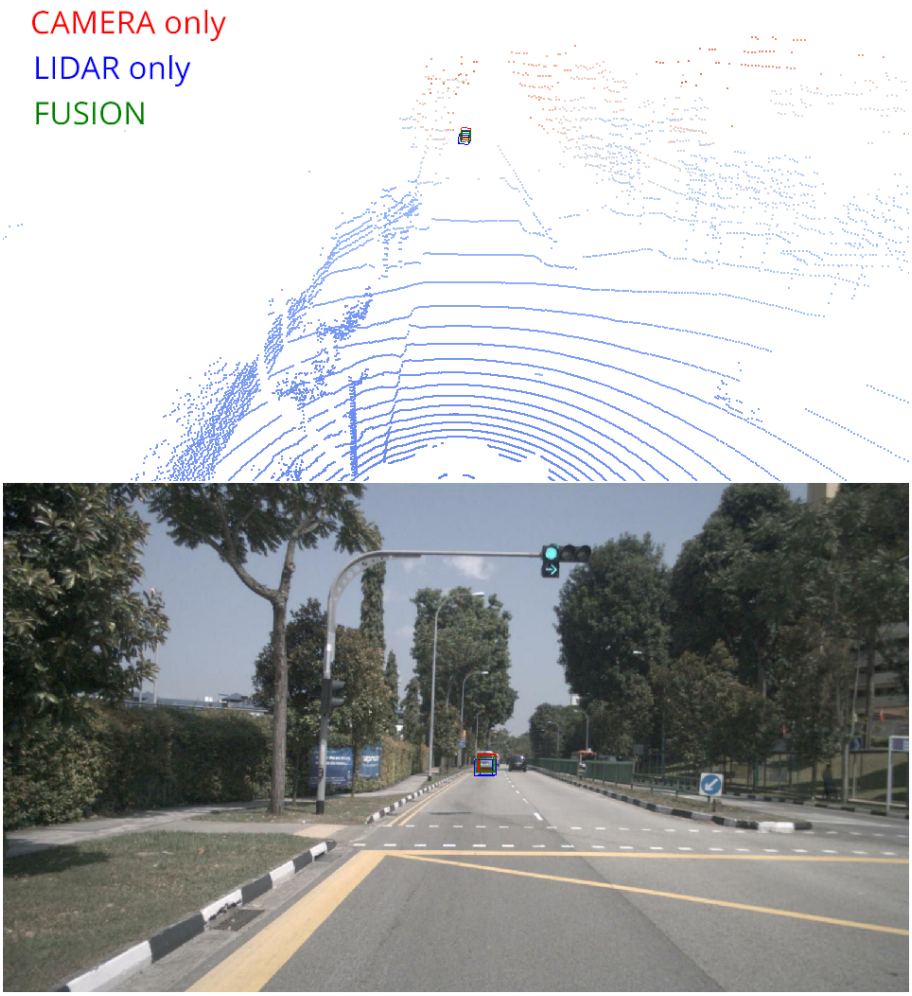


Figure 5.7: Inference on NuScenes validation split. The car at high range is found with all three input combinations with reasonable precision.

6

Discussion

This chapter covers discussions around both the chosen methods as well as the results from Chapter 5. This is followed up by discussions on future work.

6.1 Analysis of Method

This thesis investigated how to build a 3D object detection architecture that is robust against sensor failure. A major part is using multiple sensors as input, which tends to lead to complex models. To simplify the development of the model, it was compartmentalized into individual components, such as depth estimation or point cloud based object detection. This allowed for clear milestones and division of tasks, and worked well overall. One exception was the task of improving the performance during full sensor availability, which was done before incorporating support for LIDAR failure. Some of the developed fusion components proved to be incompatible with the pseudo point cloud approach, and had to be discarded.

The final architecture became conceptually simple on a high level. A reason for this is treating both sensor data in the same representation, i.e. point clouds, leading to a single head and set of detections. Another approach that was considered was to have separate heads for the different data representations, and combining their detections.

A big problem with the current architecture is that when inputting only point clouds, roughly half of the network is performing useless computations, specifically the image backbone, depth header, and pseudo point cloud BEV encoders. A way of dealing with this would be to add point cloud data onto the image before processing it. This could be depth, reflection, and height information as additional channels [36]. Such addition would change the depth estimation task to depth completion when both sensors are available, leading to more precise pseudo point cloud and thus better image feature fusion. Some tests were conducted with this type of early fusion, however, this led to a more complex training and did not show any immediate improvements.

Another way of improving the image feature fusion, would be to also include projections of the image features onto the real point cloud, similar to the original Deep Continuous Fusion [8] mechanism. This could make the fusion more precise, especially when the depth estimation is not aligned with the real point cloud information.

There might be a choice between this and early fusion, since they both concern the alignment of the image features and depth information. This approach was attempted, but the increased complexity of the training procedure deemed it out of scope for the thesis.

The primary dataset in this work, KITTI, is commonly used in other publications. However, it still has some flaws, one of which is that objects are only annotated if they are visible in the camera. It is also missing some annotations for occluded or far away objects. This means that a perfect model would find cars which are not annotated and therefore be punished for it. In other words, an optimally trained network has to learn which cars not to detect, which is obviously not desirable. Another major issue with KITTI is the small number of samples and their relative similarity. Some publications have shown methods that are unable to learn on KITTI, but work very well on larger datasets [36]. Thus it can be quite dangerous to make all decisions based on KITTI results. NuScenes [22] is somewhat better when it comes to both number of samples and number of missed annotations. However, it was released in the middle of the thesis which made it infeasible to use as a development dataset. Additionally there are very few architectures with published results on NuScenes.

Finally, the development of the architecture has only focused on ResNet as the feature extractor, however, there are many other high performing convolutional feature extractors which could be better suited for this architecture. An example is Deep Layer Aggregation [46] which improves upon the top-down feature combination approach. Such a replacement should not require changes in the other components of the architecture, and simply be plug-and-play.

6.2 Analysis of Results

The primary research question was how to build an architecture that is robust against sensor failure. As can be seen in Table 5.1 the different input combinations achieve close to state-of-the-art performance, which indicates that the model is robust. The same table shows that there is still a clear gap for the LIDAR case, which could be explained by the comparison against a point-wise two-stage approach, which may not be a completely fair comparison. In the camera case, the results for IOU 0.5 are in between the state-of-the-art architectures and very close to the best on hard. The results for 0.7 are less conclusive, and the hard difficulty even outperforms easy and moderate significantly, which seems unreasonable. This can be explained by looking at the PR curves in Figure 5.1, where the hard difficulty starts much higher on the precision axis than the other. The KITTI evaluation script only averages over 11 points of the PR-curve, where the first point is decided by the most certain detection, which can cause instability when the AP is low. This can be seen as a flaw in this type of evaluation, and therefore the focus has been on IOU 0.5 for the camera, where the scores are much more stable.

From the distance based evaluation in Table 5.2 it is clear that all input combinations see a performance reduction for objects that are far away. While LIDAR and fusion

see a significant drop after 50 m, the same drop is seen in the camera after only 15 m. These metrics are not enough to confirm that the camera is safe to use for slow speed, short range navigation. However, it is clear that it is not safe to use as a fallback in high speed situations. Another interesting finding is that the fusion benefits only start to show at higher ranges, when the point cloud is quite sparse and the dense image information can make a difference. This is in line with the findings in other publications [8].

The results for training with and without SSF in Table 5.3 clearly show that the training strategy is of great importance. However, Figure 5.2 shows that the actual failure ratio between sensors is not as important, as long as both sensors fail to some degree. For example, if LIDAR data is always available the model seems to completely rely on it, which is indicated by the high LIDAR results and practically non-existing camera detections. A bias towards the LIDAR data is not an entirely unexpected result, since LIDAR only architectures vastly outperform those using images from a monocular camera. When the camera fails, the LIDAR performance degrades, which looking at Table 5.3 may be expected, but the amount of degradation it quite significant. A more surprising result is the fusion metrics when the camera never fails, which has a significant drop from the other cases. A possible explanation for this is that the training gets stuck in a local minima where it relies too much on the camera information since it is always available. Further exploration of the optimal training strategy is necessary, since the experiments in this report are limited to varying the sensor failure probabilities.

The other research question was whether sensor failure robustness leads to generalization benefits. The results of training with and without SSF can be seen in Tables 5.5 and 5.4. Table 5.5 shows a regularization effect where SSF decreased the AP for training while increasing it for validation. The results in Table 5.4 also show generalization benefits of SSF, with higher scores on almost all cases. An explanation for this could be that the regularization effects reduces the overfitting to the missing annotations in KITTI that were mentioned in the method analysis. An example of this can be seen in Figure 6.1, where the detection is not correctly annotated.

Continuing on model generalization, the evaluation results on NuScenes in Table 5.6 contradicts the generalization benefits seen earlier. Here, SSF actually decreased the performance. The reason for this could be that SSF has forced the model to rely on both sensors, and the large differences in the camera sensor for KITTI and NuScenes degrades the performance. This can also be noted by the LIDAR results being better than fusion even for the model without SSF training. A possible solution to this problem would be to analyze the differences in the inputs for NuScenes and KITTI, and try to compensate for them, for example by better normalization of pixel values. Looking at the qualitative evaluation in Figures 5.6 and 5.7 it is evident that the point cloud is much more sparse than in KITTI, and the image color/contrast settings are very different. However, Figure 5.7 still shows good perception at long range, which can indicate that with some changes the model could transfer better.

A final note on the results is that even though average precision is widely used to compare models, it is not necessarily a perfect way to evaluate a 3D object detection model. An example would be that at an increasing distance, the bounding box

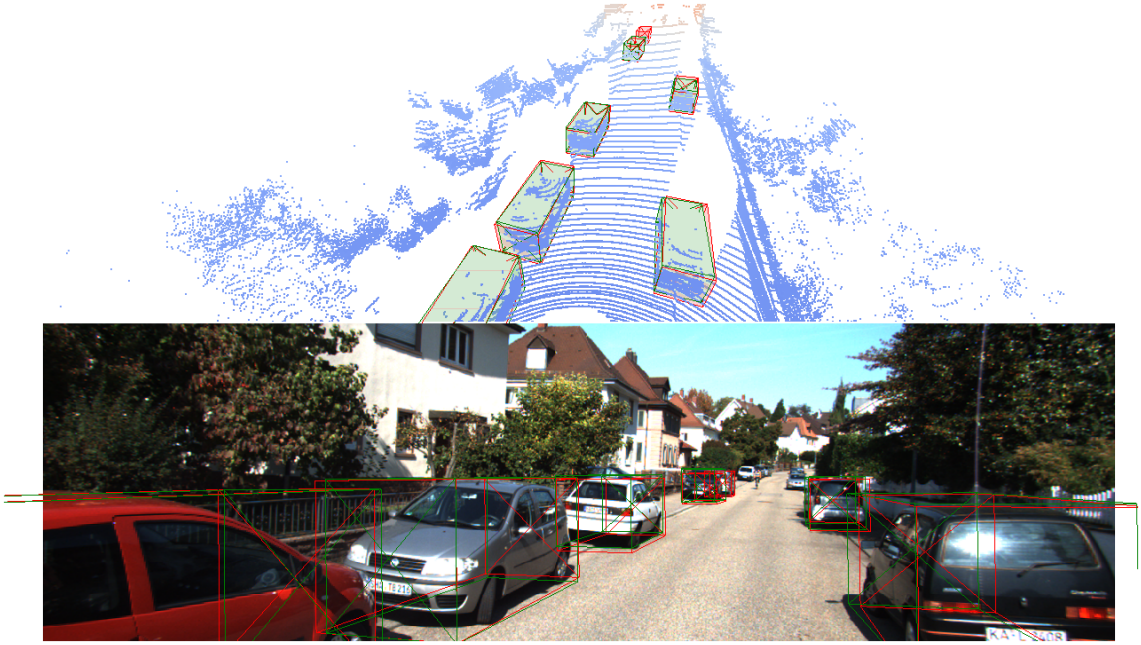


Figure 6.1: An example where the model detects a car at the very back that is not annotated.

precision is less important, and just finding the objects would be enough. However, at close range the preciseness is more important, since it affects the immediate decision making.

6.3 Future Work

Even though the presented architecture works well and fulfills the goals, there are a lot of areas which can be improved. Some topics mentioned in the method and result analysis in Sections 6.1 and 6.2 are left as future work, such as:

- Early fusion by including additional LIDAR-based channels to the image.
- Projecting image features onto the real point cloud.
- Other feature extractors than ResNET.

Another important thing to note is that although the architecture uses a point-wise learnable discretization method, inspired by PointPillars [35], the entire feature extraction could be point-wise, such as PointNet++[44]. This could be a good approach considering the performance of PointRCNN [33], and also bring benefits during image feature fusion.

Finally, complete sensor failure is not the only way for a sensor to provide less information. As mentioned in Chapter 1 there is also sensor degradation, during for example heavy rain or night time. This is a very important part before actually integrating this type of architecture in an autonomous driving system.

7

Conclusion

This thesis investigates the possibility of increasing the robustness of 3D object detection for autonomous driving, with the main research question being:

How does one build an object detection network
that is robust against sensor failure?

This work shows not only how to build such a network, but also covers the potential benefits of doing so, for example increased generalization. The presented architecture uses images and point clouds to produce 3D bounding boxes for all visible cars. To fuse the data into a common representation, the images are elevated an estimated 3D point cloud, which is merged with the real point cloud. This minimizes information loss, and enables the model to only have a single set of detections.

The ablation studies on the architecture prove the importance of a training procedure with simulated sensor failure, which relates closely to the follow up research question:

Can such a network generalize better?

Experiments on KITTI consistently show that failure robustness bring generalization benefits within a single dataset. However, when evaluated on NuScenes, which has a different camera and LIDAR, the failure robust model performs worse. Despite some hypotheses, the reason for this contradiction is unfortunately not clear, and more experiments are required to find a definitive answer.

The final performance of the architecture reaches close to state-of-the-art on camera, LIDAR, and fusion, while being robust against sensor failure. With more future work regarding model improvements and sensor degradation, this type of architecture can provide more robust perception capabilities to an autonomous driving system.

Bibliography

[1] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[2] Yan Wang, Wei-Lun Chao, Divyansh Garg, Bharath Hariharan, Mark Campbell, and Kilian Weinberger. Pseudo-lidar from visual depth estimation: Bridging the gap in 3d object detection for autonomous driving. *arXiv preprint arXiv:1812.07179*, 2018.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[4] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2117–2125, 2017.

[5] Kitti. `http://www.cvlibs.net/datasets/kitti/`. Accessed: 2019-05-03.

[6] SAE International. Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles. Standard, SAE International, June 2018.

[7] Shaoshan Liu, Jie Tang, Zhe Zhang, and Jean-Luc Gaudiot. Computer architectures for autonomous driving. *Computer*, 50(8):18–25, 2017.

[8] Ming Liang, Bin Yang, Shenlong Wang, and Raquel Urtasun. Deep continuous fusion for multi-sensor 3d object detection. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 641–656, 2018.

[9] Di Feng, Christian Haase-Schuetz, Lars Rosenbaum, Heinz Hertlein, Fabian Duffhauss, Claudius Glaeser, Werner Wiesbeck, and Klaus Dietmayer. Deep multi-modal object detection and semantic segmentation for autonomous driving: Datasets, methods, and challenges. *arXiv preprint arXiv:1902.07830*, 2019.

[10] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *CoRR*, abs/1905.11946, 2019.

[11] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you

need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[12] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W. Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *CoRR*, abs/1609.03499, 2016.

[13] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

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

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

[16] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

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

[18] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? In *Advances in Neural Information Processing Systems*, pages 2483–2493, 2018.

[19] Michael Barnard. Tesla & google disagree about lidar — which is right? `https://cleantechnica.com/2016/07/29/tesla-google-disagree-lidar-right/`. Accessed: 2019-05-17.

[20] Bin Yang, Ming Liang, and Raquel Urtasun. Hdnet: Exploiting hd maps for 3d object detection. In *Conference on Robot Learning*, pages 146–155, 2018.

[21] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *The International Journal of Robotics Research*, 32(11):1231–1237, 2013.

[22] Nuscenes. `https://www.nuscenes.org/`. Accessed: 2019-04-09.

[23] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[25] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[26] Fisher Yu, Dequan Wang, Evan Shelhamer, and Trevor Darrell. Deep layer aggregation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2403–2412, 2018.

[27] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 652–660, 2017.

[28] Bin Yang, Wenjie Luo, and Raquel Urtasun. Pixor: Real-time 3d object detection from point clouds. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7652–7660, 2018.

[29] Jason Ku, Melissa Mozifian, Jungwook Lee, Ali Harakeh, and Steven L Waslander. Joint 3d proposal generation and object detection from view aggregation. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1–8. IEEE, 2018.

[30] 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.

[31] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[32] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3354–3361. IEEE, 2012.

[33] Shaoshuai Shi, Xiaogang Wang, and Hongsheng Li. Pointrcnn: 3d object proposal generation and detection from point cloud. *arXiv preprint arXiv:1812.04244*, 2018.

[34] Yin Zhou and Oncel Tuzel. Voxelnet: End-to-end learning for point cloud based 3d object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4490–4499, 2018.

[35] 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.

[36] Gregory P Meyer, Ankit Laddha, Eric Kee, Carlos Vallespi-Gonzalez, and Carl K Wellington. Lasernet: An efficient probabilistic 3d object detector for autonomous driving. *arXiv preprint arXiv:1903.08701*, 2019.

[37] Zengyi Qin, Jinglu Wang, and Yan Lu. Monogrnet: A geometric reasoning network for monocular 3d object localization. *arXiv preprint arXiv:1811.10247*, 2018.

[38] Huan Fu, Mingming Gong, Chaohui Wang, Kayhan Batmanghelich, and Dacheng Tao. Deep ordinal regression network for monocular depth estimation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2002–2011, 2018.

[39] Charles R Qi, Wei Liu, Chenxia Wu, Hao Su, and Leonidas J Guibas. Frustum pointnets for 3d object detection from rgb-d data. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 918–927, 2018.

[40] Bin Xu and Zhenzhong Chen. Multi-level fusion based 3d object detection from monocular images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2345–2353, 2018.

[41] Eskil Jörgensen, Christopher Zach, and Fredrik Kahl. Monocular 3d object detection and box fitting trained end-to-end using intersection-over-union loss. *arXiv preprint arXiv:1906.08070*, 2019.

[42] Danfei Xu, Dragomir Anguelov, and Ashesh Jain. Pointfusion: Deep sensor fusion for 3d bounding box estimation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 244–253, 2018.

[43] Xiaozhi Chen, Huimin Ma, Ji Wan, Bo Li, and Tian Xia. Multi-view 3d object detection network for autonomous driving. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1907–1915, 2017.

[44] Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. In *Advances in Neural Information Processing Systems*, pages 5099–5108, 2017.

[45] Tensorflow. `https://www.tensorflow.org/`. Accessed: 2019-07-04.

[46] Fisher Yu, Dequan Wang, Evan Shelhamer, and Trevor Darrell. Deep layer aggregation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2403–2412, 2018.