3D YOLO: End-to-End 3D Object Detection Using Point Clouds

EZEDDIN AL HAKIM
3D YOLO: End-to-End 3D Object Detection Using Point Clouds

EZEDDIN AL HAKIM
Abstract

For safe and reliable driving, it is essential that an autonomous vehicle can accurately perceive the surrounding environment. Modern sensor technologies used for perception, such as LiDAR and RADAR, deliver a large set of 3D measurement points known as a point cloud. There is a huge need to interpret the point cloud data to detect other road users, such as vehicles and pedestrians.

Many research studies have proposed image-based models for 2D object detection. This thesis takes it a step further and aims to develop a LiDAR-based 3D object detection model that operates in real-time, with emphasis on autonomous driving scenarios. We propose 3D YOLO, an extension of YOLO (You Only Look Once), which is one of the fastest state-of-the-art 2D object detectors for images. The proposed model takes point cloud data as input and outputs 3D bounding boxes with class scores in real-time. Most of the existing 3D object detectors use hand-crafted features, while our model follows the end-to-end learning fashion, which removes manual feature engineering.

3D YOLO pipeline consists of two networks: (a) Feature Learning Network, an artificial neural network that transforms the input point cloud to a new feature space; (b) 3DNet, a novel convolutional neural network architecture based on YOLO that learns the shape description of the objects.

Our experiments on the KITTI dataset show that the 3D YOLO has high accuracy and outperforms the state-of-the-art LiDAR-based models in efficiency. This makes it a suitable candidate for deployment in autonomous vehicles.
För att autonoma fordon ska ha en god uppfattning av sin omgivning används moderna sensorer som LiDAR och RADAR. Dessa genererar en stor mängd 3-dimensionella datapunkter som kallas *point clouds*. Inom utvecklingen av autonoma fordon finns det ett stort behov av att tolka LiDAR-data samt klassificera medtrafikanter.

Ett stort antal studier har gjorts om 2D-objektdetektering som analyserar bilder för att upptäcka fordon, men vi är intresserade av 3D-objektdetektering med hjälp av endast LiDAR data. Därför introducerar vi modellen 3D YOLO, som bygger på YOLO (You Only Look Once), som är en av de snabbaste state-of-the-art modellerna inom 2D-objektdetektering för bilder. 3D YOLO tar in ett point cloud och producerar 3D lådor som markerar de olika objekten samt anger objektets kategori. Vi har tränat och evaluerat modellen med den publika träningsdatan KITTI.

Våra resultat visar att 3D YOLO är snabbare än dagens state-of-the-art LiDAR-baserade modeller med en hög träffsäkerhet. Detta gör den till en god kandidat för kunna användas av autonoma fordon.
Acknowledgements

I would first like to thank Scania CV AB for giving me the opportunity to do my master thesis with them. I would also like to thank my industrial supervisor Christian Larsson and my supervisor at KTH Iman Sayyaddelshad, for their support and advice during the project. Many thanks to my examiner Elena Troubitsyna for her comments and feedback on the report and to my friends Vesna Barros, Mohammad Reza Karimi and Tuncay Dağdelen for reading previous drafts of this report and providing many valuable comments. Finally, a big thank to my family for ongoing support. This accomplishment would not have been possible without them. Thank you.
Contents

Abstract iii
Sammanfattning iv
Acknowledgements v

1 Introduction 1
   1.1 Motivation and Background .................................. 1
   1.2 Aim .......................................................... 3
   1.3 Problem statement .......................................... 3
   1.4 Delimitation ................................................ 3
   1.5 Social and Ethics Aspects .................................. 3
   1.6 Structure .................................................... 4

2 Related Works 5
   2.1 Classical Methods .......................................... 5
   2.2 Modern Methods ........................................... 6
       2.2.1 Hand-Craft Point Cloud Features ..................... 6
       2.2.2 End-to-end Learning ................................ 7

3 Preliminaries 8
   3.1 Artificial Neural Networks ................................ 8
       3.1.1 Basic idea ............................................. 8
       3.1.2 Back-propagation ................................... 10
       3.1.3 Activation functions ................................ 11
       3.1.4 Batch normalization ................................ 12
   3.2 Convolutional Neural Networks ............................ 13
       3.2.1 2D Convolutional layers ............................. 13
       3.2.2 3D Convolutional layers ............................. 14
       3.2.3 Maxpooling .......................................... 15
   3.3 2D Object Detection ....................................... 15
       3.3.1 R-CNN for RGB Images ............................... 15
CONTENTS

3.3.2 YOLO for RGB Images ........................................ 18
3.4 3D Object Detection ............................................. 21
  3.4.1 PointNet .................................................. 21
  3.4.2 VoxelNet .................................................. 21

4 3D YOLO .......................................................... 24
  4.1 Architecture .................................................. 25
    4.1.1 Feature Learning Network ................................ 25
    4.1.2 YOLO Network .......................................... 27
  4.2 Loss function ............................................... 28
  4.3 Inference .................................................... 29
  4.4 Training Details ............................................ 30
    4.4.1 Dataset ................................................ 30
    4.4.2 Network Details ........................................ 31
    4.4.3 Anchors ............................................... 32
    4.4.4 Framework ............................................. 33
  4.5 Evaluation Metrics ......................................... 33
    4.5.1 Precision and Recall ................................... 33
    4.5.2 Average Precision ..................................... 34

5 Results .......................................................... 35
  5.1 KITTI Evaluation Protocol ................................. 35
  5.2 Runtime ..................................................... 35
  5.3 KITTI Evaluation On Validation Set ....................... 36

6 Discussion ....................................................... 40
  6.1 Result Analysis ............................................. 40
  6.2 Dataset ...................................................... 41
  6.3 Future Work ................................................ 41
  6.4 Conclusion ................................................ 42
Chapter 1 | Introduction

1.1 Motivation and Background

Autonomous vehicle development is advancing at a very high pace and self-driving trucks on public roads will soon see the light of day. Autonomous vehicles will have a global impact that will change society, the safety of our roadways and transportation systems.

An autonomous vehicle consists of four fundamental technologies: environment perception and modelling, localization and map building, path planning and decision-making, and motion control [1], as shown in Figure 1.1.

This thesis will only focus on environment perception, which answers a core question, "What is around me?". For safe and reliable driving, it is essential that an autonomous vehicle can accurately perceive its environment and implement a responsive action. The perception can be exploited for tasks such as tracking, visual localization and obstacle detection.

Multiple sensors are used for surrounding environment perception, such as camera, RADAR (Radio Detection and Ranging) and LiDAR (Light Detection and Ranging). LiDAR is a detection system that works on the same principle as RADAR, but instead of using radio waves to determine the range to a target, it uses light in the form of a pulsed laser. By firing off millions of beams of light per second, LiDAR delivers a large set of 3D measurements (points in 3D coordinates), which builds a 3D image of the environment. This set of 3D points is known as a point cloud.
Each of these sensors has its own advantages and disadvantages, and each of them can be used in different situations. Camera preserves much more detailed semantic information of the objects compared to RADAR and LiDAR, but it does not provide accurate depth information. RADAR can detect objects in cloudy weather and at night, however it is unable to detect small objects. LiDAR provides depth information and a 3D view of the vehicle’s surrounding, but it is sensitive to weather phenomena such as fog, rain or snow [1].

Recently, machine learning has shown great progress on camera perception tasks, such as 2D object detection, object recognition and object tracking [2, 3]. However, 3D object detection using LiDAR sensors is needed eagerly for the autonomous vehicles. The advantage of using 3D object detection in autonomous vehicles provides distance measurements of other road users, such as vehicles and pedestrians. However, the main challenge for 3D object detection in autonomous driving is real-time efficiency.

Current 3D object detection methods generally fall into two categories. The first is detected 3D object by using hand-crafted features with an arbitrary classifier. Hand-craft features refer to shape properties of an object that has been extracted by different algorithms. For example, features like edges of a car can be extracted by using an edge detector algorithm on a bird’s eye view of LiDAR points. Bird’s eye view refers to a projection of point cloud onto a 2D ground plane.

The second category is, end-to-end learning (a.k.a feature learning) methods, which detect the object without any hand-crafted features i.e. directly taking a point cloud and producing bounding boxes and labelling the objects. Usually, this refers to neural network architectures that can directly learn the features and classifier as a full pipeline. End-to-end learning reduces the manual engineering and has been shown in different fields to perform better than traditional methods in the field of computer vision, speech recognition and natural language processing [4]. The methods are explained in detail in the following chapter.
1.2 Aim

This research study is part of a collaborative project with Scania CV AB at the department of autonomous vehicle perception. It aims at investigating possibilities to develop a fast and accurate 3D object detection model using only LiDAR data in an end-to-end fashion. The detection is represented by estimating oriented 3D bounding boxes of physical objects and classifying the object category.

Inspired by significant improvement in both efficiency and accuracy of 2D object detection using YOLO and Faster R-CNN [2, 3]. This study attempts to decrease the difficulty in adding a third dimension to the 2D object detection task, by modifying the existing models for 2D object detection, such as YOLO, to be used for 3D objects detection.

1.3 Problem statement

Most 3D object detection models use camera data and LiDAR data as input. In this thesis, we are interested in a LiDAR-based model to detect objects. The main challenge in using only LiDAR is that point cloud data is a highly sparse unordered set, which is a fundamental problem for neural networks [5].

This study will answer the following question:

- What is the possible end-to-end 3D object detection model for LiDAR data, which can detect vehicles in real-time?

In this thesis a real-time model is defined as a minimum of 10 frames per second.

1.4 Delimitation

This study will only focus on the software implementation and modelling. Due to limitation of time and computational resources, the presented model is evaluated on vehicle objects only.

1.5 Social and Ethics Aspects

Object detection plays a very important role in the autonomous vehicles, as false negative predications of the objects may lead to fatal accidents. There are many debates about autonomous vehicles regarding how safe they are and how to hand over control of a vehicle to a robot. There are 1.2 million people
who died in traffic in 2015. Google claims that 90% of these deaths were due to human error, and claims that their autonomous vehicles had driven over 1.12 million kilometres without accidents [6].

Most traffic accidents are caused by the human error, such as aggressive driving style or too long reaction times. We believe the autonomous vehicles will decrease the number of accidents and reduce the human inference. They will also be much more efficient and environmentally friendly due to computerised control of movement and efficient route planning. An autonomous vehicle can simultaneously coordinate, control speed and make room for the road users, which minimize traffic noise.

1.6 Structure

This report is divided into 6 chapters. Chapter 2 starts with addressing some of the classical as well as state-of-the-art methods in 3D object detection. Chapter 3 presents the underlying theory that 3D YOLO builds upon. The basic idea of artificial neural network (ANN) and convolutional neural network (CNN) are presented, followed by a description about the two most common approaches in 2D object detection, Faster R-CNN and YOLO. Finally, the 3D object detection model VoxelNet is described. Chapter 4 presents our method for developing a 3D object detection model, which we denoted as 3D YOLO. The evaluation measurements and the training dataset are explained. Chapter 5 presents 3D YOLO results on the KITTI validation set compared with several state-of-the-art 3D object detection models. In the last chapter, the obtained results and future work are discussed.
This chapter addresses some of the classical as well as state-of-the-art methods in 3D object detection.

### 2.1 Classical Methods

The traditional object detection methods generally segment the point cloud to a set of clusters and classify them as objects.

Segmentation is a significant step in the perception tasks. One segmentation approach is removing the ground plane points from the point cloud, mapping the remaining ones (non-ground points) to grid cells and connecting them as occupied grid cells [7, 8]. Another approach is using graphs to segment the objects [9, 10]. In [9], an Euclidean Minimum Spanning Tree is used for an end-to-end segmentation pipeline and a RANSAC-based edge is used as selection criterion. Golovinskiy and Funkhouser [10] used k-nearest neighbours to construct a 3D graph to encourage neighbouring points to be assigned the same label (see Figure 2.1).

![Figure 2.1: Min-Cut based segmentation of point clouds. (a) The model takes a point cloud as input. (b) Construct a k-nearest neighbours graph (assumes a background prior). (c) The resulting segmentation is created via min-cut. [10]](image)

Since point clouds are sparse and have highly variable point density, a preprocessing step is needed to avoid incorrect segmentation. A common approach is Random Sampling, which is done by uniform sampling from all the points. It can also be used to reduce the dimension of the point clouds [11].
For the classification step, researchers have assumed that the shape model of the object is given and match the object model to the clusters [11]. In practice, it is not possible to model all visible objects. In [12], Teichman et al. took another approach, where hand-crafted features are used to classify the objects.

### 2.2 Modern Methods

#### 2.2.1 Hand-Craft Point Cloud Features

Many works have used voxel grid representation with hand-crafted features [13, 14, 15, 16]. The idea of voxel grid representation in [13] is subdividing the 3D space into equally spaced voxels (3D grid cells) and each voxel is converted into a hand-crafted feature vector (Figure 2.2). The empty voxels (containing no points) are converted to zero feature vectors.

![Figure 2.2](image)

Figure 2.2: The point cloud subdivided into equally spaced cells and each cell transformed to a hand-crafted feature vector. [13]

Besides the voxel grid representation, [13] used a 3D sliding window approach for 3D object detection. At each window location, the hand-crafted feature vector is passed to an arbitrary classifier (for example support vector machine) and the classifier returns a detection score. The same authors in [14] extended their previous work [13] by proposing a 3D objected detector using convolutional neural networks (CNN).

Typically, models that use CNN to detect 3D objects tend to project the input point cloud into a 2D space before passing it to the CNN (e.g. the VeloFCN [17]). However, [14, 16] detect objects natively in the 3D point cloud, which uses CNN to detect the 3D object without projecting the input into a lower dimensional space. [16] Discretizes the point cloud on square grids, where each
grid cell is assigned 1 if it contains a point, otherwise it is assigned to 0. The discretized data represented as a 4D tensor with dimensions of length, width, height and channels, is passed through a Fully Convolutional Network (FCN) (see Figure 2.3).

Figure 2.3: Illustration of the 3D Fully Convolutional Network (FCN) architecture used in [16]. FCN takes a discretized point cloud data as input and produces objects class and bounding boxes as output.

Many research papers have used image-based networks to detect 3D objects, such as Mono3D [18] and 3DOP [19]. Mono3D used monocular images, while 3DOP reconstructs depth from stereo images. Since camera does not provide accurate depth information, the 3D bounding box precision is depended on the accuracy of depth estimation.

An alternative approach that takes advantage of images and combines them with the point cloud data is MV3D [15], where a framework is proposed using information from multiple view points (LiDAR front view, LiDAR bird eye view and camera) to build a 3D object detection network. They used a region-based proposal network for fusing different sources of information and estimating 3D object proposals.

2.2.2 End-to-end Learning

Recently, Zhou et al. [20] proposed VoxelNet, an end-to-end trainable deep network that learns point-wise features directly from point cloud without any manual feature engineering. This method will be elaborated in detail in section 3.4.2.
This chapter presents the underlying theory of the techniques that our model is built upon.

3.1 Artificial Neural Networks

3.1.1 Basic idea

An artificial neural network (ANN) [21] is a collection of connected parametrized computational nodes called (artificial) neurons, which receives inputs and produces outputs (see Figure 3.1). Mathematically, a neuron \( j \) is simply a function \( f : \mathbb{R}^{N} \rightarrow \mathbb{R} \), defined as

\[
f(x; w_j) := \sigma \left( w_{0j} + \sum_{i=1}^{N} w_{ij} x_i \right)
\]

where \( \sigma : \mathbb{R} \rightarrow \mathbb{R} \) is an activation function and \( w \in \mathbb{R}^{N+1} \) is a vector of weights. For clarity we namely ignore the bias term \( w_{0j} \) and write \( f(x; w_j) = \sigma (w_j^\top x) \).

![Schematic view of a multilayer perceptron (MLP).](image)

Figure 3.1: Schematic view of a multilayer perceptron (MLP).
Neurons are arranged in layers, as shown in Figure 3.1. Layers may have different number of neurons and different kinds of activation functions. There are three kind of layers: the input layer $x \in \mathbb{R}^N$, the output layer $y \in \mathbb{R}^K$ and the layers in between the input and output layer, the hidden layers. A hidden layer $l$ can be defined as a function $F^{(l)} : \mathbb{R}^N \rightarrow \mathbb{R}^M$, with

$$F^{(l)}(x; w_1, \ldots, w_M) = \begin{bmatrix} f^{(l)}(x; w_1) \\ f^{(l)}(x; w_2) \\ \vdots \\ f^{(l)}(x; w_M) \end{bmatrix}$$

where $M$ is the number of the neurons in the layer $l$ and $N$ is the number of the inputs. The function $F^{(l)}$ can also be written as a matrix multiplication

$$F^{(l)}(x; W) = \sigma(Wx), \quad W = \begin{bmatrix} w_1^T \\ w_2^T \\ \vdots \\ w_M^T \end{bmatrix}$$

where the activation function $\sigma$ is applied element-wise. A neural network with multiple hidden layers of artificial neurons can learn more complex functions. It is proven that a multi-layer neural network can serve as a universal approximators [22]. Indeed, a neural network with one hidden layer (with finite number of neurons) can approximate any continuous function.

ANNs with $L$ layers can be written as a nested function. Given an input $x^{(1)}$ and hidden layers $x^{(l)}$, where

$$x^{(l)} = \sigma^{(l)}(W^{(l)}x^{(l-1)}), \quad (1 < l < L)$$

then the output layer has the following formula:

$$y = \sigma^{(L)}(W^{(L)}\sigma^{(L-1)}(\ldots (\sigma^{(2)}(W^{(2)}x^{(1)}) \ldots \ldots)))$$

The term ANN is used for all types of neural networks. The simplest and minimal ANN can be designed with just one neuron. There are several types of ANNs for various applications, such as computer vision and speech recognition. The Multilayer Perceptron (MLP) is ANN consisting of at least one hidden layer where all neurons in one layer are fully connected to all the neurons in the next layer (as network in Figure 3.1). The MLP can also be considered as a type of Deep Network, which is defined as neural networks composed of more than one hidden layer.
3.1.2 Back-propagation

The learning problem for a neural network is formulated as searching for a set of weights \( \{ W^{(l)} \}_{l=2}^L \) that minimizes a given loss function \( E : \mathbb{R}^{K \times K} \rightarrow \mathbb{R} \). The learning problem or training a neural network can also be formulated as searching for a function that best maps a set of inputs to their correct output. A loss function (referred to also as cost function or error function), is a function which calculates the difference between the output predicted by neural network, \( y \), and the true value \( \hat{y} \). An example of a loss function is the L2-norm:

\[
E(y, \hat{y}) = \sum_{i=1}^{K} (y_i - \hat{y}_i)^2.
\]

In general, the loss functions in neural networks are non-convex and are not possible in order to find a closed form expression for the minima. Instead, the gradient of the loss function \( \nabla E \) with respect to all weights \( \{ W^{(l)} \}_{l=2}^L \) is calculated and a local minimum will be achieved by using gradient descent.

To solve the gradient of the loss \( \nabla E \) analytically, the chain rule will be used. In practice, the back-propagation algorithm \([23]\) is used for computing the gradient efficiently using the chain rule in a dynamic programming fashion. That is, given a neural network and a loss function, back-propagation propagates the loss at the output layer backward so that the gradients at the hidden layers can be computed using the chain rule and therefore adjusting the weights at each neuron (see Figure 3.2).

![Figure 3.2: Adjusting (learning) weights of the neuron network.](image-url)
3.1.3 Activation functions

The main role of the activation function is to decide whether a neuron should be activated or not, depending on the input. This is inspired by biological neural networks. There are several activation functions for different kinds of learning problems. The choice of different activation function is dependent on the architecture of the network and also on the results one obtains by using them.

An activation function can be linear or non-linear, but a network with linear activation functions can only learn linear problems since summing all layers in the network will give another linear function. The non-linearity allows the network to learn more complex problems. Some of the desirable properties in the activation function are non-linearity, continuous differentiability, finite range and smoothness[24, 25, 26].

The most common activation functions are the sigmoid function [21], the Hyperbolic Tangent (tanh)[27], the Rectified linear unit (ReLU) [28] and the Leaky Rectified linear unit (LReLU) [29], which are presented in Figure 3.3. All four functions are non-linear and share the same basic behaviour, but they have different properties.

The sigmoid function maps the input to output ranging in (0,1). It is defined as

$$\sigma_{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

The sigmoid is a widely used activation function. It is easy to understand and apply, as well as having the properties of smoothness and is continuous differentiability. One of the problems of the sigmoid is vanishing gradients [30]. It occurs when the lower layers have gradients of near 0, leading to slow convergence and instability when training a neural network.

The tanh function looks very similar to the sigmoid function, being just a scaled version of it ranging from -1 to 1. It also has the vanishing gradients problem. Tanh function is given by:
The ReLU function is one of the most widely used activation functions. It has the very simple form

$$\sigma_{ReLU}(x) = \max(0, x)$$

The benefits of using ReLU is that it does not have vanishing gradients problem, since the gradient the ReLU is either 0 or 1. Another advantage of using ReLU is that it does not activate all neurons, if the input is negative it converts it to zero. This makes the network sparse, efficient and easy for computation. ReLU is non-smooth and can only be used in the hidden layers.

Another version of ReLU is Leaky ReLU, which solves the dying problem of ReLU by allowing a small gradient when the neuron is not active. Leaky ReLU is defined as

$$\sigma_{LReLU}(x) = \max(0, x) - \alpha \max(0, -x),$$

with $\alpha$ usually being set to 0.25. For discrete tasks, a common activation function used in the output layer, is the softmax function. Softmax is a generalization of the sigmoid function which is used for multiclass classification. The function is given by

$$\sigma(x)_j = \frac{e^{x_j}}{\sum_{i=1}^{M} e^{x_i}} \quad j = 1, ..., M.$$

### 3.1.4 Batch normalization

While training a neural network, the weights of the layers change, which also changes the distribution of the inputs. This makes the training slower, especially for very deep networks. This problem, known as internal covariate shift, can be reduced by a common technique called batch normalization (BN). [31]

The basic idea is that you normalize the inputs to a layer with a zero mean and unit standard deviation. This makes each layer in the network learn faster and independently of the other layers, illustrated in Figure 3.4.
3.2 Convolutional Neural Networks

A convolutional neural network (CNN) is another type of ANN [32, 21]. A CNN contains three building blocks: convolutional layers, maxpooling layers and fully connected layers.

3.2.1 2D Convolutional layers

In MPL networks, fully connected layers are used, where each neuron receives inputs from all neurons in the previous layer. In some applications such as image classification, MLP networks are inefficient and have many parameters. For example, a single neuron in MLP which receives an image as input with size 64x64x3 (width, height, channel), will have 12,288 weights.

Instead of neurons receiving all the inputs, convolutional layers consider the spacial locality of the input. Each neuron in a convolutional layer receives a small patch of the input, and then applies the convolution operation of the inputs with a weight matrix called kernel and passes the result to an activation function, as shown in Figure 3.5.

The kernel moves along the input with strides in both directions (columns and rows) and at each step it convolved with a patch of the input. The number of the kernels for each convolutional layer is equal to the number of input channels. This operation allows the network to be more efficient by sharing the weights between the neurons in the same layer. The convolution operation has three parameters: kernel matrix dimension, stride and padding. Stride controls the shifts for each step while padding zero-pads the border of the input. Since the convolution operation decreases the dimension of the input (see Figure 3.5), padding is used to preserve the input dimension. Zero padding also improves the performance by extracting information at the borders.
CHAPTER 3. PRELIMINARIES

Formally, a convolution layer with stride = 1, padding = 2 and a $5 \times 5$ kernel is defined as:

$$F_{n,m}(x; w) = \sigma \left( (x \ast w)_{n,m} \right) = \sigma \left( \sum_{k=1}^{C} \sum_{i,j=-2}^{2} w_{i,l}^{k} \cdot x_{n+i,m+j}^{k} \right),$$

where $x \in \mathbb{R}^{N \times M \times C}$ is an input matrix with $C$ channels from the previous layer and $w \in \mathbb{R}^{5 \times 5 \times C}$ is a weight matrix. The output of the convolution layer is a matrix with dimension $N \times M$.

### 3.2.2 3D Convolutional layers

3D convolutional layer consider the spatial and temporal locality [33, 34]. An application example is capturing the motion information in video by applying convolution operation in both images and time. Compared to 2D convolutional layers, where each channel has its own kernel, in the 3D convolutional layers a kernel is defined as a 3D tensor, so all channels share the same 3D matrix weights (see Figure 3.6). In 3D convolution operation, stride and padding are defined in both directions (space and time).

---

**Figure 3.6:** a) Applying 2D convolution on an image with one channel, the output is an image. b) Applying 2D convolution on an image with several channels, the output is an image. c) Applying 3D convolution on an image with several channels, the output is an image with multiple channels. [33]
Formally, a convolution layer with stride = (1,1), padding = (2,2) and a $5 \times 5 \times 5$ kernel is defined as:

$$F_{n,m,c}(x; w) = \sigma \left( \sum_{i,j,k=-2}^{2} w_{i,j}^k \cdot x_{n+i,m+j}^c \right),$$

where $x \in \mathbb{R}^{N \times M \times C}$ is input and $w \in \mathbb{R}^{5 \times 5 \times 5}$ is a weight matrix. The output of the 3D convolution layer is a 3D tensor with dimensions $N \times M \times C$.

### 3.2.3 Maxpooling

Maxpooling is the second building block in CNNs [35, 21]. The idea of pooling is to reduce the dimensionality from one layer to another. It divides the input to a set of equally non-overlap regions, and for each region, it takes the maximum value of the region (see Figure 3.7). Same as convolution operation, it moves along the input with strides at both directions.

![Figure 3.7: Max pooling with a 2x2 filter and stride = 2.](image)

### 3.3 2D Object Detection

This section describes the two most common approaches in 2D object detection for RGB images, R-CNN and YOLO.

#### 3.3.1 R-CNN for RGB Images

CNNs have shown impressive results in image recognition and became the standard model for image classification tasks. Generally, in a classification task an image contains a single focused object, but the sights in real life are composed of multiple overlapping objects. Creating an efficient model that detects many objects in an image is a challenging task (see Figure 3.8).
Detecting objects in an image can be summarized in three steps: (a) creating a set of bounding boxes (i.e. box width, box height and box center) with multiple scales; (b) for each bounding box moving along at x and y directions; (c) for each moving step the image patch in the bounding box is passed as input to a classifier, e.g. a CNN to predict whether the bounding box contains an object and if so, what object. This approach is inefficient and cannot be used in real-time due to expensive computation time.

A better approach is region-based CNN (R-CNN)[37]. Instead of classifying all possible regions in an image, it uses selective search algorithm that generates a small set of regions represented as bounding boxes (∼2000 proposals) that may contain an object. R-CNN works well, but is still inefficient, since it has to pass every single proposed region for every image forward in a pre-trained CNN and classify them using a SVM classifier. A pre-trained CNN is used to extract image features and to determine whether the proposed region contains an object or not.

An improved model of R-CNN that makes it faster is Fast R-CNN [36]. Instead of using a pre-trained CNN to extract features for each region, Fast R-CNN allows to run CNN only once for all region proposals in one input image. This is done by using an operation known as Region of Interest Pooling (RoIPooling), that projects each proposed region (bounding box) from the input image to an area on the last convolution layer (a.k.a feature maps) and then applies maxpooling for each region (Figure 3.9). The architecture of Fast R-CNN comes with a single trainable CNN for feature extraction and classification, where R-CNN has separate models for feature extraction and object classification.
Fast R-CNN has shown remarkable progress results in efficiency, but there remains one bottleneck. Fast R-CNN uses selective search to generate a set of proposal regions and that slows down the process.

A further improved version of Fast R-CNN that solves the bottleneck is Faster R-CNN [3]. The architecture of Faster R-CNN is a single trainable pipeline that does not require an algorithm for region proposals. Selection search algorithm used in R-CNN and Fast R-CNN is replaced with a small conventional network named Region Proposal Network (RPN) that generates proposal regions.

To generate region proposals, a small window slides over the feature map. At each sliding step, the RPN takes the window as input and outputs $k$ proposal regions with confidence scores (probability of being background or foreground) and locations (bounding boxes), shown in Figure 3.10. These $k$ proposal regions are parametrized relative to $k$ reference boxes, called anchors. Anchor boxes are references with multiple scales and aspect ratios to adjust different types of objects. More specifically, RPN outputs $k$ refined anchors (corrections on anchor boxes) with scores containing an object or a background. The anchors idea
simplifies the region proposal problem by learning the proposals relative to the anchors.

The true and false predictions of object detection models is defined with a given threshold for the overlap of the predicted bounding box and the ground truth bounding box. The overlap is defined with an evaluation metric called *Intersection over Union* (IoU) which is computed as

\[
\text{IoU} = \frac{\text{Box}_{\text{pred}} \cap \text{Box}_{\text{GT}}}{\text{Box}_{\text{pred}} \cup \text{Box}_{\text{GT}}}. \tag{3.1}
\]

### 3.3.2 YOLO for RGB Images

*You Only Look Once* (YOLO) is an alternative approach for 2D object detection [2]. The architecture of YOLO is a simple CNN, which is fed by an image just once through the network and outputs the class scores and bounding box coordinates (see Figure 3.12). The Faster R-CNN comes with two networks (a RPN and a Fast R-CNN), which need to be trained iteratively one after the other.
YOLO splits the input image to a grid $S \times S$, and each grid cell predicts $B$ bounding boxes with confidence scores and class scores. During the training, each grid cell assigns a candidate bounding box for an object. During the inference, YOLO predicts $S \times S \times B$ bounding boxes, but most of them are eliminated by keeping the ones with high confidence score and using Non-Max Suppression (NMS) to eliminate duplicate detections (see Figure 3.13).

Comparing YOLO with region-based models, YOLO performs significantly faster, but it also makes a significant number of localization errors. YOLOv2 is the second version of YOLO which is one of the fastest state-of-the-art 2D object detectors [39]. It comes with some key modifications such as using batch normalization layers after each convolutional layers and using anchors instead of the fully connected layers to predict bounding boxes, as shown in Table 3.1. However, YOLO and YOLOv2 have a spatial limitation that it is only possible to one type of class of each grid cell, and therefore it struggles to detect small irregularly shaped objects.
During the training, the multi-part loss function

\[
\mathcal{L} = \lambda_{\text{coord}} \sum_{i=1}^{S^2} \sum_{j=1}^{B} 1_{\text{obj}}^{ij} \left[ (x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
+ \lambda_{\text{coord}} \sum_{i=1}^{S^2} \sum_{j=1}^{B} 1_{\text{obj}}^{ij} \left[ (\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2 \right] \\
+ \sum_{i=1}^{S^2} \sum_{j=1}^{B} 1_{\text{obj}}^{ij} (C_{ij} - \hat{C}_{ij})^2 + \lambda_{\text{noobj}} \sum_{i=1}^{S^2} \sum_{j=1}^{B} 1_{\text{noobj}}^{ij} (C_{ij} - \hat{C}_{ij})^2 \\
+ \sum_{i=1}^{S^2} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2,
\]  

(3.2)

is optimized, where \(1_{\text{obj}}^{ij}\) denotes whether the grid cell \(i\) contains an object and \(1_{\text{obj}}^{ij}\) denotes whether the \(j\)-th bounding box predictor in the grid cell \(i\) is a candidate for that prediction. The two parameters \(\lambda_{\text{coord}}\) and \(\lambda_{\text{noobj}}\) are used to increase the loss from bounding box coordinate and to decrease the loss from confidence predictions for boxes that don’t contain objects, respectively. In [2], the model uses \(\lambda_{\text{coord}} = 5\) and \(\lambda_{\text{noobj}} = 0.5\). Bounding boxes are encoded with four variables: box center \((x, y)\) and box weight and height \((w, h)\). \(C_{ij}\) is the confidence score of the \(j\)-th box in grid cell \(i\). \(p_i(c)\) is the conditional probability of whether cell \(i\) contains an object of class \(c\). The variables without hat denote
the ground truth values and variables with hat symbol denote the output of YOLO.

3.4 3D Object Detection

This section describes 3D object detection models that use LiDAR data only. A bounding box in 3D object detection is represented by a box center \((x, y, z)\), and box weight, height and length \((w, h, l)\), and box rotation \(\theta\).

3.4.1 PointNet

As mentioned in the introduction, a point cloud is a highly sparse and unordered set. The main challenge with modeling orderless points is that the model needs to be invariant to \(N!\) permutations, where \(N\) is the number of points in the point cloud. In [5] PointNet is proposed as an end-to-end classification neural network that directly takes a point cloud as input without any preprocessing and outputs class scores, shown in Figure 3.14. PointNet is efficient and respects the permutation invariance of points in the input.

An improved version of PointNet is introduced in [40], named PointNet++, which respects the spatial localities of points, just like CNNs. PointNet and PointNet++ showed impressive results on 3D object recognition and semantic segmentation tasks with \(~1k\) points input size. A typical size of a point cloud is \(~100k\) points, which makes the training and inference of PointNet difficult and computationally expensive [20].

3.4.2 VoxelNet

Recently, VoxelNet[20] was proposed, which is an end-to-end 3D object detection using point cloud data. The high level idea of VoxelNet is that it subdivides the input point cloud into equally 3D voxels and then transforms points within
each voxel to a trainable feature vector that characterizes the shape information of the contained points. The representation vectors for each voxel stacks together and passes to a region proposal network to detect the objects.

The VoxelNet network consists of three functional blocks to form an end-to-end trainable pipeline: a Feature Learning Network (FLN), a Convolutional Middle Layers (MDL), and a Region Proposal Network (RPN), as shown in Figure 3.15.

The FLN learns descriptive shape information of the objects. FLN uses a modified version of PointNet to learn local voxel features, which is invariant to permutation of the points order. Each voxel vector representation has a fixed dimension $C$, so the output of the FLN is a 4D tensor, since each representation vector is in a 3D coordinate (see Figure 3.15).

The second block adds more context to the shape description. The architecture of the CML is three sequential 3D convolutional layers, each one followed by a BN and ReLU layer. The CML network takes a 4D tensor as input and outputs a 3D tensor.

The last block is like the RPN in Faster R-CNN but with several modifications (see Figure 3.16). RPN takes the feature map from CML as input and outputs 3D bounding boxes with confidence scores. During the inference, bounding boxes with the highest scores are selected and NMS is applied to remove the duplicate boxes.
Figure 3.16: Region Proposal Network architecture [20]
Chapter 4 | 3D YOLO

This chapter presents extension of YOLO for end-to-end trainable 3D object detection network that takes point cloud as input and yields 3D bounding boxes with class scores without using any hand-crafted features. The proposed model will be denoted 3D YOLO. In the following a description of the training and testing procedures is given.

![Diagram of 3D YOLO pipeline]

Figure 4.1: 3D YOLO pipeline: a) the input point cloud are divided into 3D voxel grid cells; b) Feature Learning Network transforms the non-empty voxels to a new feature representation of the point cloud represented as a 3D tensor; c) the 3D tensor passes through the YOLO network and it outputs 3D bounding boxes with class scores.
4.1 Architecture

Briefly, 3D YOLO consists of two networks, the Feature Learning Network (FLN) and a CNN based on You Only Look Once v2 (YOLOv2), as shown in Figure 4.1. The FLN is the same network as used in VoxelNet, which transforms the input point cloud to a new feature space. YOLO takes this new representation of the point cloud as input and outputs class scores and bounding box coordinates.

We used different FLN configurations which will be described in detail in section 4.4.

4.1.1 Feature Learning Network

Point Cloud Preprocessing

All 3D points in the input point cloud are divided into 3D voxel grid cells, where each voxel be of size \((v_D \times v_H \times v_W)\), the voxel representation is illustrated in Figure 3.15. Assume that the 3D space of input point cloud have range \(D, H, W\) along the \(Z, Y, X\) axes respectively. Then the 3D grid will be of size \(D' = D/v_D, H' = H/v_H\) and \(W' = W/v_W\). [20]

Typically voxels have highly variable point densities, due to the sparsity of point cloud and variable distance of objects from the LiDAR sensor. To overcome this issue, we use the Random Sampling [20, 11], which basically samples \(T\) points uniformly at random from the voxel having more than \(T\) points.

Denote \(p_j^{(i)} = [x_j^{(i)}, y_j^{(i)}, z_j^{(i)}, r_j^{(i)}] \in \mathbb{R}^4\) as \(j\)-th LiDAR point in the \(i\)-th non-empty voxel \(V^{(i)}\). \(x_j^{(i)}, y_j^{(i)}\) and \(z_j^{(i)}\) refer to the point coordinates and \(r_j^{(i)}\) the received reflectance, where \(i = 1, ..., R\) and \(j = 1, ..., T\). Then, extend each point \(p_j^{(i)}\) to include its offset from the \(i\)-th voxel centroid, and denote the extended point by \(\hat{p}_j^{(i)} = [x_j^{(i)}, y_j^{(i)}, z_j^{(i)}, r_j^{(i)}, x_j^{(i)} - v_x^{(i)}, y_j^{(i)} - v_y^{(i)}, z_j^{(i)} - v_z^{(i)}] \in \mathbb{R}^7\), in which \((v_x^{(i)}, v_y^{(i)}, v_z^{(i)})\) is centroid of the voxel \(V^{(i)}\) [20].

Network

The FLN takes a non-empty voxel \(V_{in}^{(i)} = \{\hat{p}_1^{(i)}, ..., \hat{p}_T^{(i)}\} \in \mathbb{R}^{T \times 7}\) as input and outputs a feature vector with fixed dimension \(C\), denoted as \(V_{out}^{(i)} \in \mathbb{R}^C\). We also convert an empty voxel (containing no points) to the zero feature vector. The architecture of FLN is a chain of connected Voxel Feature Encoding (VFE) layers, as shown in Figure 3.15. For simplicity, assume that the FLN consists of one VFE layer. A VFE layer contains three components as shown in Figure 4.2:
1. A fully connected layer followed by a ReLU and BN layer, which transforms each point $\hat{p}_j^{(i)}$ in the voxel to a point-wise feature vector, denoted by $f_j^{(i)} \in \mathbb{R}^M$.

2. An element-wise maxpooling layer, which applies the maximum operations across all point-wise feature vectors $f_j^{(i)}$ to get locally aggregated feature, $f_{\text{max}}^{(i)} = \max (f_1^{(i)}, ..., f_R^{(i)}) \in \mathbb{R}^M$.

3. A point-wise concatenate, which concatenates each point-wise feature vector $f_j^{(i)}$ with the locally aggregated feature $f_{\text{max}}^{(i)}$ and obtains the output of VFE layer $f_{\text{out}}^{(i)} = \{ (f_j^{(i)}, f_{\text{max}}^{(i)})^T \} \in \mathbb{R}^{2M}$.

To obtain the final voxel representation $V_{\text{out}}^{(i)} \in \mathbb{R}^C$, the output of the last VFE-$n$ layer $f_{\text{out}}^{(i)}$ is fed to a fully connected layer followed by a ReLU and BN layer, and finally through an element-wise maxpooling layer (see Figure 4.2). The element-wise maxpooling layer makes the FLN invariant to $T!$ permutations of the points. See [5] and [40] for more details.

By feeding all non-empty voxels $V_{\text{in}}^{(1)}, ..., V_{\text{in}}^{(R)}$ to the FLN, a list of feature vectors will be obtained $V_{\text{out}}^{(1)}, ..., V_{\text{out}}^{(R)}$. This list can be represented as a sparse 4D tensor of size $(C \times D' \times H' \times W')$. After reshaping it to a 3D tensor of size $(H' \times W' \times C \cdot D')$, it passes through the YOLO network.

![Figure 4.2: Voxel Feature Encoding (VFE) Layer. [20](image)]
4.1.2 YOLO Network

We design a new CNN architecture base on YOLOv2 [39] to detect 3D objects in real-time, called 3DNet. It has 14 convolutional layers and 3 maxpooling layers, as shown in Figure 4.3. After each convolutional layer, there is a LReLU and BN layer, except in the last layer.

The output of 3DNet is a tensor of size $(H'/8 \times W'/8 \times B \cdot (8 + K))$, where $B$ is the number of the anchors and $K$ is the number of classes. Each cell in the feature map grid $(H'/8 \times W'/8)$ predicts $B$ bounding boxes, confidence scores for each of them and $K$ class scores $p_1, ..., p_K$ (see Figure 4.3). Predicted bounding boxes are parameterized as refined anchors. A refined anchor is a vector $(t_x, t_y, t_z, t_w, t_h, t_l, t_\theta)$, where $t_x, t_y, t_z$ are the offset center coordinates, $t_w, t_h, t_l$ are the offset dimensions, and $t_\theta$ is the offset rotation angle. Predicting offset relative to an anchor instead of box coordinates makes the learning problem easier and faster [39]. Given a refined $(t_x, t_y, t_z, t_w, t_h, t_l, t_\theta)$ to an anchor parameterized as $(x_a, y_a, z_a, w_a, h_a, l_a, \theta_a)$, the bounding boxes coordinates are calculated as

![Figure 4.3: 3DNet is the second network of 3D YOLO pipeline which has 14 convolutional layers and 3 maxpooling layers.](image-url)
\[ b_x = t_x d_a + x_a \]
\[ b_y = t_y d_a + y_a \]
\[ b_z = t_z d_a + z_a \]
\[ b_w = e^{t_w} + w_a \]
\[ b_h = e^{t_h} + h_a \]
\[ b_l = e^{t_l} + l_a \]
\[ b_\theta = t_\theta + \theta_a \]
\[ b_\theta = t_\theta + \theta_a \]
\[ c = \sigma_{\text{sigmoid}}(c) \]
\[ p_i = \sigma_{\text{softmax}}(p_1, \ldots, p_K)_j, \quad j = 1, \ldots, K \]

where \( d_a = \sqrt{l_a^2 + w_a^2} \) is the length of the diagonal of the anchor, which is used as a normalization term to constrain the predicted bounding box center location. Our experiments have shown that the location constrain makes the network more stable, compared to YOLOv2, which instead of predicting offset to anchors location, predicts location coordinates relative to the location of the grid cell [39].

### 4.2 Loss function

Our loss function is based on the YOLO multi-part loss for RGB images (3.2), which is introduced in YOLO [2] and YOLOv2 [39]. We extend the loss by adding the third dimension to handle the 3D bounding boxes, which has the following form:

\[
L_{\text{3DYOLO}} = \lambda_{\text{coord}} \sum_{i=1}^{G} \sum_{j=1}^{B} \mathbb{1}_{i,j}^{\text{obj}} \left[ (t_x^{(ij)} - \hat{t}_x^{(ij)})^2 + (t_y^{(ij)} - \hat{t}_y^{(ij)})^2 + (t_z^{(ij)} - \hat{t}_z^{(ij)})^2 \right] \\
+ \lambda_{\text{coord}} \sum_{i=1}^{G} \sum_{j=1}^{B} \mathbb{1}_{i,j}^{\text{obj}} \left[ (t_w^{(ij)} - \hat{t}_w^{(ij)})^2 + (t_h^{(ij)} - \hat{t}_h^{(ij)})^2 + (t_l^{(ij)} - \hat{t}_l^{(ij)})^2 \right] \\
+ \sum_{i=1}^{G} \sum_{j=1}^{B} \mathbb{1}_{i,j}^{\text{obj}} (c^{(ij)} - \hat{c}^{(ij)})^2 + \lambda_{\text{noobj}} \sum_{i=1}^{G} \sum_{j=1}^{B} \mathbb{1}_{i,j}^{\text{noobj}} (c^{(ij)} - \hat{c}^{(ij)})^2 \\
+ \sum_{i=1}^{G} \sum_{j=1}^{B} \mathbb{1}_{i,j}^{\text{obj}} K \sum_{k=1}^{K} (p_k^{(ij)} - \hat{p}_k^{(ij)})^2
\]
where $G = H' / 8 \cdot W' / 8$ is the number of cells in the grid. $1_{ij}^{\text{obj}}$ is an indicator function the $j$-th bounding box predictor in cell $i$ is the candidate for that prediction, while $1_{ij}^{\text{noobj}}$ denotes the bounding boxes that do not contain an object. Following YOLOv2, during the training, only one candidate bounding box is assigned for each object, which is the anchor with the highest overlapping the ground-truth box. The two parameters $\lambda_{\text{coord}}$ and $\lambda_{\text{noobj}}$ are used to increase the loss from bounding box coordinate and to decrease the loss from confidence predictions for boxes that do not contain objects, respectively. The hat symbol indicates the ground truth values, which are obtained from the training dataset.

### 4.3 Inference

In only one forward pass through 3D YOLO, the network predicts $B$ bounding boxes in each cell of the grid of size $H' / 8 \times W' / 8$, which is many boxes. Even after keeping only the bounding boxes with highest confidence score as, there will be many duplicate boxes. One of the common approaches to eliminate duplicate 2D bounding boxes is the NMS algorithm. NMS keeps the most confident box if the boxes overlap, which can be summarized as follows:

1. Project 3D bounding boxes to 2D representation: $[b_x, b_y, b_w, b_h, b_{\theta}, c, p_1, \ldots, p_k]^\top$
2. Remove all bounding boxes with confidence score $c \leq 0.6$
3. While there are any remaining boxes:
   (a) Select the box with the largest confidence score $c$ and output that as prediction
   (b) Remove any remaining box with $\text{IoU} \geq 0.5$ with the output of step 3 (a).

For efficiency, we calculate the overlap of the boxes in the bird’s eye view by projecting the 3D bounding boxes to 2D boxes. Next, we need to calculate class scores for the remaining boxes, by using the conditional class probabilities and the individual box confidence predictions:

$$
\Pr(\text{Class}_j) = \begin{cases} 
\Pr(\text{Class}_j|\text{Object}) \cdot \Pr(\text{Object}) = p_j \cdot c, & \text{if } c \geq 0.5 \\
0, & \text{otherwise}
\end{cases}
$$

where the index $j$ ranges over all different possible classes.
4.4 Training Details

4.4.1 Dataset

The proposed model is trained using a dataset created from KITTI Vision Benchmarking Suite [41]. KITTI datasets are captured by driving a car around the city of Karlsruhe in Germany. The recording platform is a Volkswagen Passat B6, equipped with the following sensors:

- 1 Inertial Navigation System (GPS/IMU): OXTS RT 3003
- 1 Laser scanner: Velodyne LiDAR HDL-64E
- 2 Grayscale cameras, 1.4 Megapixels: Point Grey Flea 2 (FL2-14S3M-C)
- 2 Color cameras, 1.4 Megapixels: Point Grey Flea 2 (FL2-14S3C-C)
- 4 Varifocal lenses, 4-8 mm: Edmund Optics NT59-917

The LiDAR laser scanner spins at 10 frames per second, capturing approximately 100k points per $360^\circ$ scan, which is known as a point cloud. The camera images are cropped to 1382 x 512 pixels. The cameras are triggered at 10 frames per second by the laser scanner (when facing forward) with dynamically adjusted shutter time. [41]

The 3D object KITTI benchmark provides 3D bounding boxes for object classes such as cars, vans, trucks, pedestrians, cyclists and trams, which are labelled manually in 3D point clouds based on the camera information. To avoid the false positives detections, objects outside the camera plane are unlabelled.
KITTI 3D object dataset is comprised 7481 training point clouds (and images) with labels which used for training and validation and 7518 point clouds (and images) without labels which are used for testing. KITTI also provides three detection evaluation levels: *easy*, *moderate* and *hard*, according to the object size, occlusion state and truncation level. The minimal pixel height for easy objects is 40px, which approximately corresponds to vehicles within 28m. For moderate and hard level objects are 25px, corresponding to a minimal distance of 47m [17].

### 4.4.2 Network Details

The specifications of our network details suitable with KITTI LiDAR setup. The point cloud has range $[-3, 1], [-40, 40], [0, 70.4]$ all in meters along the $Z, Y, X$ axes of LiDAR respectively. We choose a smaller voxel size and a lower-dimension vector representation comparing to VoxelNet implementation. The voxel size has set to $v_W = 0.1m$, $v_W = 0.1m$ and $v_D = 0.2m$, with maximum number of points in each voxel $T = 8$. The voxel grid would be of size $D' = 20$, $H' = 800$ and $W' = 704$. The vector representation of voxels has dimension $V_{\text{out}}^{(i)} \in \mathbb{R}^{16}$. Our FLN consist of three VFE layers, VFE-1(7,16), VFE-2(16,16) and VFE-3(16,16), where in VFE(a, b), we define a to be the input layer dimensionality and be to be the output layer dimensionality. Figure 4.5 illustrated this network.

After transforming all non-empty voxels to feature vectors representation in the FLN, we will get a sparse 4D tensor of size $(16 \times 20 \times 800 \times 704)$. We reshaped it to a tensor of size $(800 \times 704 \times 320)$ and feed the tensor to the 3DNet. The architecture of 3DNet describes in details in Table 4.1.
Table 4.1: The detailed architecture of 3DNet that is used for the training

<table>
<thead>
<tr>
<th>Layer</th>
<th>Filters</th>
<th>Size</th>
<th>Stride / Padding</th>
<th>Extension</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolutional 1</td>
<td>3x3</td>
<td>64</td>
<td>1/1</td>
<td>LReLU, BN</td>
<td>800 × 704 × 320</td>
<td>800 × 704 × 64</td>
</tr>
<tr>
<td>Convolutional 2</td>
<td>3x3</td>
<td>128</td>
<td>1/1</td>
<td>LReLU, BN</td>
<td>800 × 704 × 64</td>
<td>800 × 704 × 128</td>
</tr>
<tr>
<td>Maxpooling 1</td>
<td>2x2</td>
<td>2</td>
<td></td>
<td></td>
<td>800 × 704 × 128</td>
<td>400 × 352 × 128</td>
</tr>
<tr>
<td>Convolutional 3</td>
<td>3x3</td>
<td>128</td>
<td>1/1</td>
<td>LReLU, BN</td>
<td>400 × 352 × 128</td>
<td>400 × 352 × 64</td>
</tr>
<tr>
<td>Convolutional 4</td>
<td>1x1</td>
<td>64</td>
<td>1/0</td>
<td>LReLU, BN</td>
<td>400 × 352 × 128</td>
<td>400 × 352 × 64</td>
</tr>
<tr>
<td>Convolutional 5</td>
<td>3x3</td>
<td>128</td>
<td>1/1</td>
<td>LReLU, BN</td>
<td>400 × 352 × 64</td>
<td>400 × 352 × 128</td>
</tr>
<tr>
<td>Maxpooling 2</td>
<td>2x2</td>
<td>2</td>
<td></td>
<td></td>
<td>400 × 352 × 128</td>
<td>200 × 176 × 128</td>
</tr>
<tr>
<td>Convolutional 6</td>
<td>3x3</td>
<td>256</td>
<td>1/1</td>
<td>LReLU, BN</td>
<td>200 × 176 × 128</td>
<td>200 × 176 × 256</td>
</tr>
<tr>
<td>Convolutional 7</td>
<td>1x1</td>
<td>128</td>
<td>1/0</td>
<td>LReLU, BN</td>
<td>200 × 176 × 256</td>
<td>200 × 176 × 128</td>
</tr>
<tr>
<td>Convolutional 8</td>
<td>3x3</td>
<td>256</td>
<td>1/1</td>
<td>LReLU, BN</td>
<td>200 × 176 × 256</td>
<td>200 × 176 × 256</td>
</tr>
<tr>
<td>Maxpooling 3</td>
<td>2x2</td>
<td>2</td>
<td></td>
<td></td>
<td>200 × 176 × 256</td>
<td>100 × 88 × 256</td>
</tr>
<tr>
<td>Convolutional 9</td>
<td>3x3</td>
<td>512</td>
<td>1/1</td>
<td>LReLU, BN</td>
<td>100 × 88 × 256</td>
<td>100 × 88 × 512</td>
</tr>
<tr>
<td>Convolutional 10</td>
<td>1x1</td>
<td>256</td>
<td>1/0</td>
<td>LReLU, BN</td>
<td>100 × 88 × 512</td>
<td>100 × 88 × 256</td>
</tr>
<tr>
<td>Convolutional 11</td>
<td>3x3</td>
<td>512</td>
<td>1/1</td>
<td>LReLU, BN</td>
<td>100 × 88 × 256</td>
<td>100 × 88 × 512</td>
</tr>
<tr>
<td>Convolutional 12</td>
<td>1x1</td>
<td>256</td>
<td>1/0</td>
<td>LReLU, BN</td>
<td>100 × 88 × 512</td>
<td>100 × 88 × 256</td>
</tr>
<tr>
<td>Convolutional 13</td>
<td>3x3</td>
<td>512</td>
<td>1/1</td>
<td>LReLU, BN</td>
<td>100 × 88 × 256</td>
<td>100 × 88 × 512</td>
</tr>
<tr>
<td>Convolutional 14</td>
<td>3x3</td>
<td>512</td>
<td>1/0</td>
<td></td>
<td>100 × 88 × 256</td>
<td>100 × 88 × B · (8 + K)</td>
</tr>
</tbody>
</table>

In general, an important factor in choosing a specific network design or configuration is the input and output resolution. Choosing a grid size is a trade-off between the accuracy and efficiency. Regarding automated driving, efficiency is much more important. In our task, the output resolution needs to be higher than image-based detection, since the point cloud space ranges over a much larger area than RGB images.

After we tried different configurations and design, a reasonable trade-off is a grid size of 100 × 88, where each cell has size 0.8 × 0.8 squared meter (comparing to YOLOv2 where each cell in grid 7 × 7 has size 32 × 32 pixels, see Table 3.1).

4.4.3 Anchors

An anchor can be considered as a prior (initial beliefs) for the size of a detected object. Anchors with multiple scales and aspects make the network more stable and enable faster convergence for the learning algorithm. For efficiency reasons we use only two anchors for each class:

- Car anchors:
  1. \( h_a = 1.6m, w_a = 1.6m, l_a = 4m \) and \( \theta_a = 0^\circ \)
  2. \( h_a = 1.6m, w_a = 1.6m, l_a = 4m \) and \( \theta_a = 90^\circ \)

- Pedestrian anchors:
1. \(h_a = 1.7\text{m}, w_a = 0.5\text{m}, l_a = 0.7\text{m}\) and \(\theta_a = 0^\circ\)
2. \(h_a = 1.7\text{m}, w_a = 1.5\text{m}, l_a = 0.7\text{m}\) and \(\theta_a = 90^\circ\)

- Cyclist anchors
  1. \(h_a = 1.6\text{m}, w_a = 0.7\text{m}, l_a = 2\text{m}\) and \(\theta_a = 0^\circ\)
  2. \(h_a = 1.6\text{m}, w_a = 0.7\text{m}, l_a = 2\text{m}\) and \(\theta_a = 90^\circ\)

### 4.4.4 Framework

3D YOLO has been implemented using Python library Tensorflow [42]. The training and testing were run on a DGX Station with a NVIDIA Tesla P100 GPU 16 GB. A part of the implementation is based on [43] and [44].

### 4.5 Evaluation Metrics

#### 4.5.1 Precision and Recall

Precision and recall are the most common performance measures for classification and object detection tasks. Precision can be defined as

\[
\text{precision} = \frac{\#\text{true positive}}{\#\text{true positive} + \#\text{false positive}}.
\]

The precision metric measures how many of the predicted objects that were true positive. High precision relates to a low false positive rate. A precision score of 1.0 for a class indicates that every object that predicted to be in that class, is classified correctly.

Recall metric is the fraction of correct positive predictions among the all positive observations in actual class given by

\[
\text{recall} = \frac{\#\text{true positive}}{\#\text{true positive} + \#\text{false negative}}.
\]

The recall metric measures how many of the true positives that were found. High recall relates to low false negative rate. A recall score of 1.0 for a class indicates every object in that class has been found and predicted correctly.

In object detection tasks, true positive and false positive is defined via IoU (see Eq. 3.1). It is worth mentioning that a predicted bounding box is considered as true positive if IoU is greater than a given threshold, otherwise it is considered as a false positive.
4.5.2 Average Precision

A model with zero mistakes has precision and recall equal to 1, which is hard to achieved. In practice precision-recall curve is used to observe what the best balance between precision and recall. It shows the trade-off between both metrics for different thresholds, as shown in Figure 4.6.

![Precision-Recall Curve](image)

Figure 4.6: Precision-recall curve shows the trade-off between precision and recall. The closer the curve to (1,1), the higher the performance.

Average Precision (AP) is a numerical summary for the shape of the precision-recall curve, and it is defined as the mean precision over \( N \) discrete values of recall \( \{r_n\}_{n=1}^N \), whose values lie between 0 and 1 [45] defined as

\[
AP = \frac{1}{N} \sum_{n=1}^{N} \max_{n \leq i \leq N} p(r_i)
\]

where \( p(r_i) \) is the measured precision at recall \( r_i \).
Chapter 5 | Results

This chapter presents 3D YOLO results on the KITTI validation set.

5.1 KITTI Evaluation Protocol

We evaluated 3D YOLO on the challenging KITTI object detection benchmark which has 7,481 training and 7,518 test point clouds (frames), comprising a total of 80,256 labeled objects. The ground-truth labels of the test set is not available, but it is possible to evaluate it over the KITTI server \(^1\) [41]. KITTI also provides a common evaluation protocol for object detection that is used in many research papers, where the performance is measured by the Average Precision (AP) and the IoU threshold is 0.7 for Car class (at least 70% overlap with ground truth).

Since the test server access is limited, we evaluate the performance of our method on a validation set. Following [19], we split the training frames into a training set (3,712 frames) and a validation set (3,769 frames), and ensure that frames from training and validation set do not come from the same video sequences.

This training split and KITTI setting is a fare comparison with other object detection methods that used the same setting, such as [18, 19, 17, 15, 20].

5.2 Runtime

We compare runtime of 3D YOLO with MV3D [15] and VoxelNet [20], presented in Table 5.1. Since VoxelNet has been tested on a Nvidia Titan X GPU and source code is unavailable, we compared our runtime with an unofficial Tensorflow implementation [43] of VoxelNet on a Nvidia Tesla P100 GPU. The table shows our model is 1.64× faster than VoxelNet.

\(^1\)When writing this report, the access to the server was limited, to three submissions per month.
<table>
<thead>
<tr>
<th>Method</th>
<th>Data</th>
<th>Frames Per Second (fps)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nvidia Titan X GPU</td>
<td>Nvidia Tesla P100 GPU</td>
</tr>
<tr>
<td>MV3D [15]</td>
<td>LiDAR+Mono</td>
<td>2.8</td>
</tr>
<tr>
<td>VoxelNet [20]</td>
<td>LiDAR</td>
<td>4.3</td>
</tr>
<tr>
<td>3D YOLO</td>
<td>LiDAR</td>
<td>16.1</td>
</tr>
</tbody>
</table>

Table 5.1: Runtime (in fps) for MV3D, VoxelNet and 3D YOLO. * denotes runtime for an unofficial Tensorflow implementation [43] of VoxelNet.

## 5.3 KITTI Evaluation On Validation Set

We evaluate our model only on the Car class across all three difficulty levels (easy, moderate and hard) and compare it with several state-of-the-art 3D object detectors, including Mono3D [18] and 3DOP [19] which use image data only, VeloFCN [17] and VoxelNet [20] which use LiDAR data only and MV3D [15] which was both images and LiDAR data.

We compute two evaluation metrics: localization average precision ($AP_{loc}$) and 3D bounding box detection average precision ($AP_{3D}$). $AP_{loc}$ is measured on the bird’s eye view (2D ground plane), while $AP_{3D}$ is measures on the world space (3D space). Since $AP_{loc}$ considers only 2D location and orientation of the bounding boxes, it should yield higher value than $AP_{3D}$.

<table>
<thead>
<tr>
<th>Method</th>
<th>Data</th>
<th>Easy</th>
<th>Moderate</th>
<th>Hard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mono3D [18]</td>
<td>Mono</td>
<td>5.22</td>
<td>5.19</td>
<td>4.13</td>
</tr>
<tr>
<td>3DOP [19]</td>
<td>Stero</td>
<td>12.63</td>
<td>9.49</td>
<td>7.59</td>
</tr>
<tr>
<td>VeloFCN [17]</td>
<td>LiDAR</td>
<td>40.14</td>
<td>32.08</td>
<td>30.47</td>
</tr>
<tr>
<td>MV3D [15]</td>
<td>LiDAR</td>
<td>86.18</td>
<td>77.32</td>
<td>76.33</td>
</tr>
<tr>
<td>MV3D [15]</td>
<td>LiDAR+Mono</td>
<td>86.55</td>
<td>78.10</td>
<td>76.67</td>
</tr>
<tr>
<td>VoxelNet [20]</td>
<td>LiDAR</td>
<td>89.60</td>
<td>84.81</td>
<td>78.57</td>
</tr>
<tr>
<td>3D YOLO</td>
<td>LiDAR</td>
<td>82.99</td>
<td>73.52</td>
<td>65.10</td>
</tr>
</tbody>
</table>

Table 5.2: A comparison of the localization performance of 3D YOLO with the state of the art 3D object detectors on KITTI validation set. The evaluation metric is Average Precision ($AP_{loc}$) (in %) of bird’s eye view bounding boxes 0.7 IoU.
<table>
<thead>
<tr>
<th>Method</th>
<th>Data</th>
<th>3D IoU=0.7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Easy</td>
<td>Moderate</td>
</tr>
<tr>
<td>Mono3D [18]</td>
<td>Mono</td>
<td>2.53</td>
</tr>
<tr>
<td>3DOP [19]</td>
<td>Stero</td>
<td>6.55</td>
</tr>
<tr>
<td>VeloFCN [17]</td>
<td>LIDAR</td>
<td>15.20</td>
</tr>
<tr>
<td>MV3D [15]</td>
<td>LIDAR</td>
<td>71.19</td>
</tr>
<tr>
<td>MV3D [15]</td>
<td>LiDAR+Mono</td>
<td>71.29</td>
</tr>
<tr>
<td>VoxelNet [20]</td>
<td>LiDAR</td>
<td>81.97</td>
</tr>
<tr>
<td><strong>3D YOLO</strong></td>
<td>LiDAR</td>
<td><strong>73.80</strong></td>
</tr>
</tbody>
</table>

Table 5.3: A comparison of the 3D detection performance of 3D YOLO with the state of the art 3D object detectors on KITTI validation set. The evaluation metric is Average Precision ($AP_{3D}$) (in %) of 3D bounding boxes 0.7 IoU.

On the validation set (see Table 5.3), our model outperformed all LiDAR-based methods except VoxelNet on both easy and moderate difficulty levels. We see in the tables and Figure 5.1 what is expected, the average position for bird-eye-view has higher value than the 3D world.

![3D Detection](image1)

![Bird’s Eye View Detection](image2)

Figure 5.1: Precision-recall curve of car detection in the 3D world and in bird’s eye view. The closer the curve is to (1,1), the higher performance has the model.
(a) All vehicles are detected.

(b) One vehicle is missed.

(c) One vehicle is unlabeled by KITTI, but is detected by 3D YOLO.

(d) All vehicles are detected.

Figure 5.2: Qualitative examples of 3D object detection results on KITTI validation set. The ground truth bounding boxes are shown in green and the predicted bounding boxes are shown in red. Note that 3D YOLO is based only on LiDAR data. For better visualization the 3D bounding boxes are projected on to the bird’s eye view and the images.
(a) Misclassification example.

(b) Misclassification example.

(c) Three vehicles are missed.

(d) Five vehicles are missed and one bounding box has wrong orientation.

Figure 5.3: Relevant examples of false negative on KITTI dataset. The ground truth bounding boxes are shown in green and the predicted bounding boxes are shown in red. Note that 3D YOLO is based only on LiDAR data, for better visualization the 3D bounding boxes are projected on to the bird’s eye view and the images.
Chapter 6  |  Discussion

This chapter presents a discussion about the obtained results, our conclusion and future work.

6.1 Result Analysis

Our results show that the end-to-end models outperform the models that use hand-crafted features. The comparison models except VoxelNet [20] use hand-crafted features, while 3D YOLO and VoxelNet follow the end-to-end fashion. Even though MV3D [15] and 3DOP[19] use a pre-trained network for weight initialization, 3D YOLO and VoxelNet, who are trained from scratch, yield better results. The main advantage of working directly with point clouds that is provides a flexible and scalable geometric representation that allows our model to be applied in many applications.

The proposed model results have been as expected: 3D YOLO outperforms all LiDAR-based models in efficiency, but not in accuracy. As seen in Table 5.1, the inference time of 3D YOLO is 16.1 fps, which satisfies the real-time requirements. The tables show that VoxelNet has higher AP than 3D YOLO but at the same time it also slower.

The efficiency-accuracy trade-off can be adjusted by the size of the output resolution. A higher precision can be achieved by increasing the output resolution, but it would also decrease the efficiency. Since our study aims to detect big objects such as cars and vans in real-time, we designed our model with a relative low output resolution in order to increase the efficiency. This has been achieved as seen in the results; VoxelNext use twice the output resolution than our model but with a lower fps. However, for detecting small objects, the output resolution needs to be higher. Also note that increasing the output resolution implies increasing the input resolution, which means that the architecture of the model needs to be reconfigured.

There is no clear theory on how to design an ANN model. Designing ANN architectures requires both experiments and human experience. As seen in Chapter 3, 3D YOLO is based on two state-of-the-art models, but some details of the design is based on our intuition and trial and error approaches, such
as choosing the number of kernels in each convolutional layer and selecting the activation functions. Therefore, the network details can be changed to suit other applications.

6.2 Dataset

Our results are based on KITTI LiDAR setup. The KITTI vehicle is equipped with a high resolution LiDAR sensor that generates a large amount of measurement points that provides more descriptive information of the objects. The high resolution dataset improved the precision and recall of our model by decreasing the sparsity of the point cloud.

A trained 3D YOLO on KITTI dataset cannot be used in vehicles with other LIDAR setup: it needs to be retrained for the new setup. Data collected from a sensor that is placed on the lower parts of a vehicle can scan the lower parts of a detected object, while if the sensor is placed higher, it can scan the top part of an object. Thus, they will have different perspective.

As mentioned in the Chapter 1, LiDAR sensors are sensitive to weather phenomena such as rain or snow. Since KITTI dataset misses cloudy weather data, our trained model probably has lower precision in these phenomena. A possible trick to overcome this limitation is adding a small amount of noise to the points cloud to create a rain/snow like dataset and retrain the model.

6.3 Future Work

For future work it is recommended to use 3D convolutional layers which consider the spatial and temporal locality of the point cloud. The challenge is that 3D convolutional operation is computationally expensive.

Using additional sensors, e.g. camera, would also be beneficial to give more shape descriptive information to the objects. This could potentially be done by projecting the image pixels on the point cloud.

Recently, several works have proposed new models for feature representation of point cloud data. PointCNN[46] is a general framework for feature learning from point cloud which showed very impressive results. By replacing the FLN with the PointCNN, the performance can be improved.

Another further step is testing the performance on the categories pedestrians, cyclists and trams, and investigating how the model performance with noisy dataset.
6.4 Conclusion

During the project process, the main challenge was the training session for the models. When the architecture behind the model is changed, the model must be trained from scratch. Therefore, a deep understanding is required to make accurate changes before running each test. Each training session takes roughly one week. To be efficient, we combined the best features from the different models. After trying various architecture and experiments, we came up with 3D YOLO.

Our conclusion is that the 3D YOLO is a real-time end-to-end model, based on LiDAR data only, that outperforms all end-to-end LiDAR-based models in efficiency. Our model is 1.64 times faster than VoxelNet with the following $AP_{3D}$ loss: 8.17\% for the easy level, 4.27\% for the moderate level and 8.32\% for the hard level. The average precision of 3D YOLO could be increased by increasing the output resolution but there will be a trade-off with the efficiency. We sacrificed little on the average precision in order to achieve a real-time model that can be a good candidate to be deployed on autonomous vehicles.
Bibliography


