Efficient Convolutional Neural Networks for Low-power Automotive Processors

by

Rytis Verbickas

Thesis submitted to the University of Ottawa
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Declaration of Authorship

I hereby certify that this thesis is entirely my own original work except where otherwise indicated. I am aware of the University of Ottawa regulations concerning plagiarism, including those regarding consequent disciplinary actions. Any use of the works of any other author, in any form, is properly acknowledged at their point of use.

Abstract

The last decade has seen a significant expansion of the application of ConvNet models to many important tasks, especially for applications where it is necessary to perceive and categorize the surrounding environment as scanned by sensors. This includes areas of application such as drones, mobile devices, and especially autonomous driving. In autonomous driving, the detection of vulnerable road users such as cyclists and pedestrians is critical for the safe operation of vehicles. These vehicles may be given a degree of control over their driving inputs. Additionally, detecting other road vehicles is essential for the safe operation of vehicle features. The availability of increasingly capable devices (especially GPUs) for developing such models has meant that power limits and computational complexity are less of a concern than outright detection capability. However with compute-restricted platforms, this becomes an issue since restrictive power limits or computational capability place tight restraints on the allowable ConvNet models that can be employed. We introduce two models that look at pedestrian detection in 2D, and car and cyclist detection in 3D, running on embedded platforms with restrictive power constraints. We show how they can be optimized to their respective task and outperform competing models while being significantly faster. In the case of the 3D model, we show how competing approaches on the same model misidentify the weaknesses of the model, that we leverage as strengths. Our improvements allow the model to consume point clouds in real-time with higher detection performance and a far faster rate, on an embedded platform, than directly competing models. Our work does not preclude low-level optimizations such as precision calibration (quantization), layer fusion or memory optimization. We then extend the well known KITTI dataset by hand, providing fine semantic segmentations for its Car class while using the data to perform an analysis of an existing LiDAR and image fusion model. This analysis shows how the choice of feature fusion for our target architecture can drastically alter model performance. We leverage these insights to propose a further modification to PPslim called PPslimg. This updated model fuses ground plane estimations and exceeds the performance of a competing approach for the same model, that relies on fusing computationally expensive semantic segmentation features.

Acknowledgements

A big thank you to my family who are a constant source of positivity and support. A special thank you to my wife who continues to be a bright beacon for me.

Thank you to Professor Laganière for his very supportive role as my supervisor and to Md Atiqur Rahman for his help and insights during our time as graduate students in VIVAlab.

Credit to Professor Wail Gueaieb at the University of Ottawa, whose LaTeX template was used as a starting point for writing this thesis.

Dedication

This thesis is dedicated to the memories of Professor Anthony Whitehead and Jonathan Blanchette.

Professor Whitehead, thank you for your tireless and deeply insightful guidance through the world of computer vision. You were kind enough to share your bright spark with your students, and I was lucky to have been one of them.

Jonathan, thank you for your insightful discussions about the brain and for your insights into mathematics. Wherever you are, I hope you have finally found peace.

Table of Contents

List of Tables

List of Figures

Abbreviations

ADAS Advanced Driver Assistance Systems 1

AEB Automatic Emergency Braking 101

AMCW Amplitude Modulated Continuous Wave 2

ANNs Artificial Neural Networks 8

BEV Birds Eye View 23, 38

BN Batch Normalization 23

ConvNet Convolutional Neural Network 3, 10

CVAT Computer Vision Annotation Tool 115

DSOD Deeply Supervised Object Detector 23

DSSD Deconvolutional Single Shot Detector 23

FC Fully Connected 9

FCN Fully Convolutional Network 26

FLOPs Floating Point Operations 9

FMCW Frequency Modulated Continuous Wave 2

FOV Field Of View 3

IMU Inertial Measurement Unit 49

IOU Intersection Over Union 31

L4T Linux for Tegra 57

LAMR Log Average Miss Rate 67

LiDAR Light Detection and Ranging 2

MAC Multiply-Accumulate Operations 9

mAP Mean Average Precision 24

MEMS Microelectromechanical systems 3

MSE Mean Squared Error 13

NAS Neural Architecture Search 4

NiN Network in Network 16

NMS Non-Maximum Suppression 27

OPA Optical Phased Array 3

OS Operating System 57

PCA Principal Component Analysis 126

PVSA Point-Voxel Feature Set Abstraction 80

R-CNN Regional CNN 23

R-FCN Regional Fully Connected Network 28

RADAR RAdio Detection And Ranging 2

ReLU Rectified Linear Unit 9

RGB Red/Green/Blue Colored 38

RoI Region of Interest 25

RPN Region Proposal Network 26

SoC System on a Chip 56

SVD Singular Value Decomposition 126

SVM Support Vector Machine 24

VOC Visual Object Classes 24

YOLO You Only Look Once 23

Chapter 1

Introduction

1.1 Overview

An embedded system is defined as a computer system (combination of processor, memory and input/output peripherals), with software, that is designed to perform a dedicated function [111]. Such systems are today ubiquitous in mobile phones, drones & Advanced Driver Assistance Systems (ADAS) while being characterized by their resource constraints [5] [58]. These constraints include limitations on total power consumption which ultimately come down to constraints on computational power (both circuitry & capability of that circuitry) and memory. To enable more capabilities & features on such systems, these systems must run carefully optimized algorithms, models of the problem or environment and software packages so as to minimize power consumption, computational requirements and memory footprint. For example in the case of ADAS [120], features can include lane departure, blind spot warnings as well as collision detection & avoidance software that requires rapid processing to achieve safe operation. In the case of phones and drones, battery life is an additional constraint while trying to be lightweight and at the same time controlling movement, navigation, managing communications as well as processing inputs received from attached sensory devices.

The expansion of features and capabilities has come with the development of more capable sensors for scanning the surrounding environment. The use of sensor depends on the specific application, as well as costs, but the sensors themselves can broadly be categorized as:

- **Cameras**: One (monocular) or multiple (stereo) cameras are now quite common on

phones, drones and vehicles. Their ubiquity is in large part to their cheap cost and the rich amount of information they can provide about the immediate environment. Their main drawback when extending to 3D use is their low accuracy relative to other possible sensors (such as Light Detection and Ranging (LiDAR)).

- **RAdio Detection And Ranging (RADAR)**: RADAR operates by emitting radio wave signals and measuring their return based on time. Various ranges of RADARs exist depending on application with short range RADARs useful for collision avoidance (ranges in the tens of meters) over a wide area. A useful property of RADARs is the ability to measure object speed due to the Doppler effect of reflected waves, as well as penetrating fog, rain and snow where camera systems may not be able to see [56].

- **LiDAR**: Very capable for highly accurate, sparse, spatial localization of the surrounding environment. They work by emitting laser light and capturing its reflection from the scene through a receiver. The received light is typically represented as a point cloud, indicating at least the spatial position of each return. The dimensionality of the LiDAR can vary from 1D to 3D, however more recent "4D" LiDAR produces not only a point cloud, but also velocity information (so called Doppler LiDAR) [2]. A wide variety of LiDAR types exist [92], and there are various ways to classify them. We could for example classify them based on the measurement strategy of its imaging system, where LiDARs are broadly based on the Time of Flight/ToF principle of determining distance based on the measurement of travel time between emitter and receiver. However the specific measurement strategy may not explicitly rely on time of flight to determine range. The measurement strategies include:

  - **Pulsed**: Measures the roundtrip delay between source and target to determine distance (explicitly relying on traditional time of flight). A signal is typically modulated with a pattern during transmission so that it can be checked for arrival back at the receiver. A typical range resolution of 1.5cm can be observed and is well suited for outdoor environments.
  - **Amplitude Modulated Continuous Wave (AMCW)**: Works by measuring the phase shift between a reflected and emitted signal to determine distance. AMCW LiDARs work well in indoor environments.
  - **Frequency Modulated Continuous Wave (FMCW)**: Relies on measuring a frequency difference or beat frequency, between outgoing and incoming beams. This difference is proportional to range. An advantage of the FMCW approach is its ability to measure velocity of non stationary targets, based on the same

frequency difference coupled with additional measurements of the difference between the transmitted and detected signal. Another benefit of this strategy is its excellent depth resolution relative to the other measurement strategies.

An alternate way to classify LiDARs is by scanning strategy, including:

- **Mechanical**: These LiDARs use a rotating assembly to produce one or more rotating beams around a mechanical axis. They provide a wide Field Of View (FOV), covering up to 360°. However their mechanical construction and moving parts mean possible reliability issues in addition to size and expense.
- **Solid-state**: These LiDARs have no rotating mechanism, allowing for increased reliability in harsher environments. In contrast to rotating scanners, these usually have a more limited FOV and possibly a reduced range.
  - **Microelectromechanical systems (MEMS)**: These scanners adjust beam position using tiny mirrors that vary in orientation with stimulus, allowing for the resulting beam to be directed to specific scene points.
  - **Optical Phased Array (OPA)**: Uses an array of light-emitting elements with varying phase to steer the direction of the resulting laser beam.
  - **Flash**: Generates an optical pulse that floods the scene, with the returning light captured by a 2D receiver array. The resulting 3D point cloud can be obtained from this pulse. The measurement range depends on emitter power level.

- **Acoustic**: This category includes ultrasonic and sonar sensors. Sonar sensors use sound propagation and its reflections for performing detection and determining ranges, usually underwater. In the case of sonar at frequencies above audible (20 kHz+), ultrasonic sensors can also be used for object detection and distance measurements. These sensors are typically found for parking and short range applications.

Complex applications serviced by resource constrained embedded systems with multi-sensor inputs necessitates the use of highly optimized models that provide users the best performance, safety capabilities and features within the envelope of the systems capacity. To describe the effectiveness of a model executing on such a system to achieve a task, it is useful to consider the efficiency of that model. For this thesis, this consideration is specifically within the context of a special class of models called a Convolutional Neural Network (ConvNet) [63] applied to object detection. These models can be trained end-to-end, which means given inputs and desired outputs, they can be adjusted using a straightforward algorithm to better represent the association between inputs and outputs

through a process called supervised learning [7]. This is attractive in the case of models for use on embedded systems where there are potentially many constraints on the model and algorithm used. Within a particular compute and parameter envelope, maximizing resource usage for the same overall architecture would usually be expected to lead to higher model capacity. However reducing compute and parameters while showing resilience to input variation, maintaining or improving performance and all without removing the wrong components of the model (by misidentifying its limitations) is a challenging task with no well-defined recipe in the search for more computational efficiency.

The term efficiency is a fundamental concept in the sciences and engineering. As a noun it is defined as "the ratio of the useful energy delivered by a system to the energy supplied to it" [77]. So we can see efficiency, essentially, as a measure of how effectively a system converts its input to generate an output. In our case we focus specifically on computational efficiency, where for less input (power), our level of output (detection performance) is a result of a better use of available resources. ConvNet's are models that we wish to physically implement in an efficient way to solve a task such as classifying its inputs or localizing some object of interest in the same input. To understand what we mean by "an efficient way" we must consider the implementation of such a network. Any physical implementation of such a model has practical considerations whose deployment will be limited by power consumption, computational complexity (the use of available circuitry to perform basic mathematical operations or move blocks of bits) and memory usage (mainly quantity of volatile memory used directly for executing operations but also non-volatile long term storage). Lower levels of expenditure of power, computational complexity or memory usage to achieve the same level of performance will therefore mean a more efficient neural network model.

What remains then is to define "level of performance", which we can do by referring to established measures of performance in the tasks we're interested in; a higher level of performance means a better value in the respective metric (either higher or lower depending on the metric and context). In the context of this thesis, the task of interest is object detection and the "levels of performance" we focus on are how well using an established metric the resulting model finds objects of interest. Then for models which have lower computational complexity, use less memory or requires less power while maintaining the same level of output (better value with respect to a performance metric) are considered to be more efficient. We then must show this efficiency is durable and can be achieved with resilience in model input, unlike some competing approaches that remove portions of the same model in the quest to also improve efficiency or, merge expensive to compute inputs in a way that does not fully leverage the information content in those inputs.

While automated searching of efficient ConvNet models is possible using Neural Archi-

tecture Search (NAS) [121], its application has a high computational cost. This still leaves room for manual ConvNet architecture optimization to try to maximize efficiency.

1.2 Problem Statement

ConvNet-based detector models tend to be large and computationally expensive to push state-of-the-art boundaries in performance. While such advances are important, they sacrifice speed on embedded platforms meaning they run slowly or not at all depending on the target platform. The high computational cost of automated searching for new architectures means there is still room for manual modification of existing architectures for higher computational efficiency while maintaining detection performance. This is especially true if some existing approaches misidentify bottlenecks of existing models. The focus of this thesis is on proposing modifications to existing 2D and 3D neural network-based object detectors to make them more efficient for use on embedded systems.

1.3 Contributions

The contributions of this thesis can be summarized as follows:

- The proposal and evaluation of a low computational complexity, 2D image-based object detector that performs coarse pedestrian detection and compares favourably with competing approaches but also being able to operat on a low power automotive processor.
- The proposal and evaluation of extensions to an existing 3D object detector, PointPillars [62], that drastically reduces its computational complexity while maintaining its performance. The proposed model, PPslim, achieves state of the art performance on the KITTI dataset [37] for its level of computational complexity and operating speed relative to competing approaches, while running on an embedded platform.
- The construction of detailed semantic segmentation annotations for the Car and Truck classes in the KITTI dataset. The constructed annotations are used to evaluate an alternative feature fusion approach for the PointPillars model, that is experimentally shown to outperform a competing fusion approach proposed by the PointPainting [110] modification to the PointPillars model.

- The proposal and evaluation of an extension to the original PointPillars model, PPslimg, that performs feature fusion to improve 3D object detection performance on KITTI. This extended model exceeds the detection performance of PointPainting, while being much simpler computationally and maintaining the speed of our proposed PPslim model.

1.4 Outline

The chapters of this thesis are arranged as follows:

- Chapter 2: Provides an introduction to neural networks (specifically convolutional neural networks), their optimization and important applications (like object detection). More specific details on building more efficient network structures, especially in the context of 2D detectors (and also 3D), are explained to allow understanding of proposed ideas and explanations in later chapters.

- Chapter 3: Proposes a modification to the SqueezeNet 2D object detection model, focused on fast execution on a low powered device on a standard pedestrian benchmark. This work was published in a workshop for IEEE CVPR [109].

- Chapter 4: Proposes a modification to the PointPillars 3D object detection model, PPslim, that focuses on fast execution on a low powered device for the KITTI Cyclist and Car classes. The proposed model maintains performance with extreme computational complexity reduction even before standard optimization techniques are applied. For its computational complexity, the model exceeds competing models in terms of its detection performance and drastically reduces their runtime on an embedded platform. The model is also resilient to input resolution changes, unlike competing PointPillars modifications that remove specific components in an attempt to improve model speed.

- Chapter 5: Describes an extension of the KITTI dataset which provides detailed semantic segmentation annotations for the Car and Truck KITTI classes. This data is used to show via experiments a better fusion strategy for image and LiDAR data fusion compared to an extension model for the PointPillars architecture. Based on this analysis, an extension to the PointPillars architecture is proposed that performs feature fusion, while improving its performance for the classes of interest, outperforming the PointPainting architecture even with that model having a significant

advantage of additional complexity in computing semantic segmentations from RGB images and fusing them with LiDAR data. Based on these results, an extension to our PPslim model is proposed that performs ground plane fusion in order to obtain a detection performance improvement with almost no additional cost in execution speed.

- Chapter 6: Concludes the thesis with a summary of concepts and contributions while describing areas of future research.

Chapter 2

Background

This chapter presents background material necessary as context for the work in this thesis. A progression is maintained starting with first principles regarding neural networks and their optimization. The discussion then shifts to the construction of efficient structures, which leads into the utilization of structures for specific purposes such as classification, segmentation and detection. An emphasis is made on the background for 2D detection due to its prominence in application for 3D detection. The chapter concludes by looking at details of the datasets and hardware utilized for the work in this thesis.

2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are biologically inspired models that emerged from the work of McCulloch and Pitts on simplified neural models [76]. ANNs importance in the field of machine learning was solidified with the later development of the backpropagation algorithm [93], which allows their weights to be adapted to a target problem, and the universal approximation theorem, that showed them to be universal approximators capable of representing any function (with certain assumptions on activation function) using a hidden layer of units or neurons. In practice, these networks contain several layers of hidden units (shown in Figure 2.1) and use non linear activation functions. The activation functions are necessarily non-linear to allow representing complex non-linear relationships from input to output.

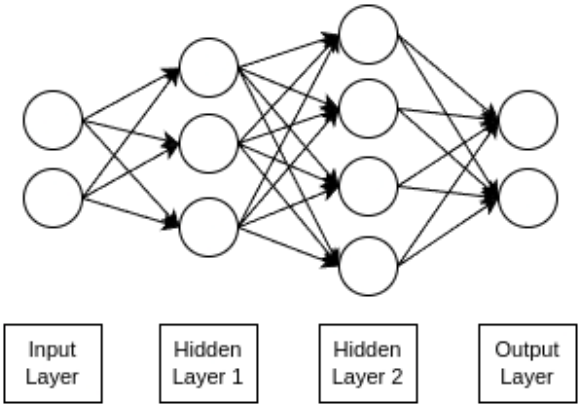

Figure 2.1: Fully Connected Neural Network

Each connection has an associated real or complex valued parameter called a weight. The values at input units depend on the current input, but the value at each hidden or output layer unit depend on multiplying the activation of a unit from the previous layer (L-1) with the weight of the connection leading to the unit in the current layer (L). If we represent the inputs at layer L-1 as vector x_{L-1}, connections weights as matrix W_{L-1}, activations at layer L+1 as vector a_L and activation function with vector input σ_L, we obtain the layer output vector y_L:

$$a_L = W_{L-1}x_{L-1} \tag{2.1}$$

$$y_L = \sigma_L(a_L) \tag{2.2}$$

The input includes an extra unit with constant value 1 that multiplies by a weight unit called the bias to adjust the activation. The Fully Connected (FC) nature of the network means for layers of size N, there will be N^2 parameters per layer and $2 * N^2$ Floating Point Operations (FLOPs) (equivalently N^2 Multiply-Accumulate Operations (MAC) where each MAC is two FLOPs).

The choice for activation unit can depend on application, for example due to certain non-linearity properties (tanh) or the ease of network quantization (Swish [87]), but will generally be non-linear. One of the most commonly used activation functions due to its simplicity is the Rectified Linear Unit (ReLU), where the output is given by:

$$\text{ReLU}(x) = \max(0, x) \tag{2.3}$$

2.2 ConvNets

ConvNet models (or CNNs) are a special type of ANN popularized by LeCun et al. [64] and were inspired based on the earlier work of Fukushima on the Neocognitron [35]. The Neocognitron model was based on the biological idea of organized, alternating structures of cells in the visual cortex where simpler cells extract features and more complex cells pool those features. With increasing depth, more complicated features could then be represented based on a combination of simpler features from the previous layer.

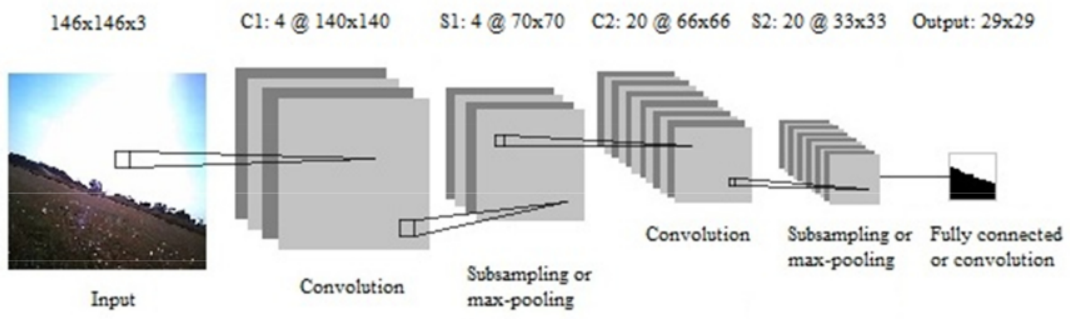

Figure 2.2: Example ConvNet With Image Input Layer, Intermediate (Convolutional (C) or Subsampling (S) Layers) and Output Layer [107]

With ConvNet's (an example is shown in Figure 2.2), their structure consists of a sequence of feed-forward layers of feature maps with varying numbers of channels. One can think of these feature maps as a variation of the fully connected networks described in Section 2.1 that, for C units in a single layer, could interpret their dimension as being 1x1xC. If we add more units separately around each individual unit the feature maps become HxWxC where H & W are the feature maps height and width respectively. Operationally, the main difference when comparing with fully connected networks are the shared weights used by ConvNet's where each feature map in a convolutional layer uses a separate set of weights to connect to all of the feature maps at the previous layer (what is called a receptive field). The same set of weights (or kernel) for a feature map is used at all positions within that feature map (weight sharing), as shown in Figure 2.3 and Figure 2.4. This allows for a significant reduction is model parameters relative to a fully connected network while the increase in dimensionality produces an increase in computational complexity (more multiplications and additions needed). Other layer types are possible in addition to convolutional layers. For example max-pooling layers compute the maximum activation

within their receptive field to produce their output while average-pooling layers compute an average. These pooling layers typically include a stride size greater than 1 (step size of the receptive field, where a stride greater than 1 means downsampling) which makes the network more resilient to small variations in the input (both at the individual layer level and at the network input level).

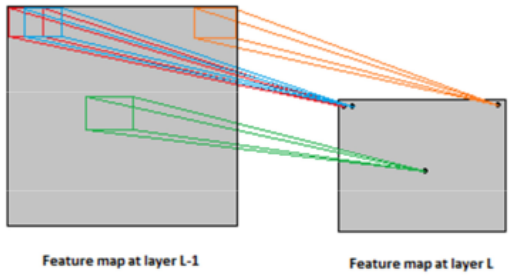

Figure 2.3: A position at a feature map at layer L has a receptive field over an area in all feature maps at previous layer L-1 [107]

Figure 2.4: Each receptive field is actually just connections to discrete positions in feature maps at the previous layer [107]

Since each feature map has its own set of weights, each can learn its own spatial filters to apply to the previous layer. The earlier layers are larger (in size, typically not in channels) and represent low level, more fine-grained details of the input. Since layers deeper in the network input from previous layers, the representation at each layer builds from the previous. To keep computational complexity reasonable and to allow for more

input variation without changing the output, deeper layers typically downsample their input where adjacent outputs in a feature map represent a step size greater than one in the previous layer. This means at the same time, the spatial resolution decreases. That means additional feature maps must be added to help preserve the information from the previous layer, leading to smaller feature maps but more numerous in number.

The spatial filters or kernel (E) for a feature map are typically M*N*K + 1 in size where M & N are the height and width of the filter and K is the number of feature maps at the previous layer. The extra parameter corresponds to a bias term that gets added to the operation. For a feature map at layer L that produces output O and takes its input from layer L-1, the discrete convolution is defined as:

$$O(i,j,l) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \sum_{k=0}^{K} I(i+m, j+n, k) \cdot E(m,n,k,l) \tag{2.4}$$

In practice, the weights in the kernels throughout the network are learnable parameters that are adjusted during training on a dataset composed of input images and corresponding ground truths. The specific task determines what kind of ground truths are necessary and also determines the structure of the loss layers (layers that compare the predictions generated by the network to the ground truth) of the network. For example cross-entropy may be used for classification or mean squared error for regression. In the case of classification, cross-entropy (CE) loss is the more appropriate measure since it gives a measure of how well the predicted probability distribution aligns with the true distribution (it expresses the amount of "surprise" given the assignment of probabilities to actual labels). The expression for CE is shown in Equation (2.5) for a single example "i". There y_i is the ground truth label while $\hat{y}_i$ is the predicted label:

$$L_{\text{CE,i}} = -(y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)) \tag{2.5}$$

We can see that in the case of a negative target label, even moderate predicted probability values ($\hat{y}_i$) will mean that when there are many negative examples, the sum of the total CE contribution from negative examples will be large. Even if those examples should not be weighed exactly equally in terms of importance (positive examples like objects versus negative such as background) or in terms of their difficulty (positive or "rare" examples as "difficult" versus negative or background examples as "easy"). For this reason, a modified version of CE called Focal Loss [67] exists specifically for object detection and, is designed for scenarios of unequal weighting between positive and negative examples as well as heavy imbalance between easy and hard examples (or put another way between numerous and

rare classes respectively). For a binary classifier, we can use the equation shown in Equation (2.6). There is a focusing parameter γ, α is a balancing factor for the true class, $\hat{y}_i$ is the estimated probability of the true class label for that example. Here γ is meant to adjust the loss based on the example difficulty while α is for the relative weighting between positive and negative examples. The affect in the case of, for example, anchor boxes as used by a 2D or 3D object detector is to reduce the focal loss for negative and/or easy examples depending on the selection of α or γ. For example with values of α=0.25 and γ=2, a low probability easy, negative example can have its CE loss reduced by a factor of 500 or more from the total loss contribution, allowing more such examples to be tolerated. If such an example turns out to have high probability, it would still have a large CE (albeit reduced). In the case of α=1 and γ=0, the expression reduces to the original expression for CE given in Equation (2.5).

$$L_{FL,i} = -(y_i \alpha (1-\hat{y}_i)^{\gamma} \log(\hat{y}_i) + (1-y_i)(1-\alpha)(\hat{y}_i)^{\gamma} \log(1-\hat{y}_i)) \tag{2.6}$$

Traditionally, binary focal loss is stated for the i-th example as:

$$L_{FL} = -\alpha_t (1-p_t)^{\gamma} log(p_t) \tag{2.7}$$

$$p_t = \begin{cases} \hat{y} & \text{if } y = 1 \\ 1-\hat{y} & \text{if } y \neq 1 \end{cases} \tag{2.8}$$

$$\alpha_t = \begin{cases} \alpha & \text{if } y = 1 \\ 1-\alpha & \text{if } y \neq 1 \end{cases} \tag{2.9}$$

On the other hand, the prediction of continuous values means a more appropriate loss function is Mean Squared Error (MSE) (also called L2 loss), where larger deviations from the ground truth are penalized a lot more. The expression is given by Equation (2.10):

$$L_{\text{MSE}} = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{2.10}$$

The degree of the power used in the loss equation can be represented by p with the loss type written as Lp. L1 loss for example is less sensitive to outliers compared to L2 loss. For some models, Smooth L1 loss [40] (shown in Equation (2.11)) is preferred when performing bounding box regression. Smooth L1 tries to combine the advantages of both L1 and L2 loss. So outlier sensitivity is reduced beyond a threshold (keeping error from

getting out of hand by too much) while maintaining it for values within the threshold. The function is also not discontinuous like L1 meaning some special checks don't need to be added in the gradient computations for smooth L1 while its gradients will vary smoother without suddenly jumping between +/-1.

$$\text{Smooth}_{L1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise} \end{cases} \tag{2.11}$$

The loss function can be thought of as creating an error surface over the weight space of the network. If the network weights are represented as a real vector, then given the training set (or a subset of it) the loss function computes a real valued number for those weights. The set of all losses computed over any possible combination of weight values is the error surface, with hills, valleys and plateaus similar to a landscape (albeit in a much higher dimensional space). The gradient at a point on this surface is then a way to move in the weight space such that the error decreases the most. The computations at the loss layer are used to compute these necessary changes for all network weights, using back-propagation. The resulting weight changes can then be applied to the original weights in the negative direction (direction of maximum decrease of the loss) as a form of hill-climbing called gradient descent. By selecting small batches of examples for back-propagation, instead of the entire dataset, we can reduce computational complexity while maintaining the efficiency of performing batch computations (as well as obtaining a better estimate that just a single sample for the true gradient).

Various approaches exist to compute more effective gradients depending on the task. The simplest approaches are to scale by a constant learning rate η or a learning rate $\eta(e)$ that changes with training epoch e, with the rate decreasing as the network nears convergence. To avoid traversal issues in error space areas having high curvature or small gradients, a momentum term can be added that balances the gradient from the previous step with that of the current step [83]. More computationally complex approaches have also been proposed and include first order approaches such as Adam, RMSProp and AdaGrad, as well as more computationally complex second order approaches such as the Levenberg-Marquardt algorithm.

- **Adaptive Gradient (AdaGrad)** [27] focuses on adjusting the scale of learning rate updates depending on frequently (small updates) and infrequently (large updates) changing parameters. It is better suited for sparse data due to its quick shrinking of gradients (at which point the model stops learning).

- **RMSProp** [50] uses an (exponentially decaying) moving average of squares of past gradients to perform a normalization of the learning rate for each weight. This is in an attempt to make the step sizes in each direction of the weight space more uniform, and to resolve one of the major shortcomings of AdaGrad.
- **Adam (Adaptive Moment Estimation)** [59] is a very popular optimizer. It combines an exponentially decaying average of standard gradients (like what can be used with regular gradient descent through momentum) with an exponentially decaying average of squared gradients (as seen with RMSProp).
- **Levenberg-Marquardt algorithm** [41] adjusts the learning rates based on the curvature along each dimension of the weight space, switching between gradient descent or alternatively either a Hessian (second order derivatives of the loss with respect to the weights) or a Gauss-Newton approximation ($H \approx J^T J$) of the Hessian depending on a trust region around the current location in weight space. In practice, the Gauss-Newton approximation is preferred over computing the actual Hessian due to the Hessian's computational complexity.

2.2.1 ConvNet Model Structures

This section outlines the historical development of basic ConvNet structures and ideas to organization that have been developed over time and are seen in object detection models that use ConvNet's (some of which are investigated in this thesis).

The original AlexNet network proposed by Krizhevsky et al. [61] for the 2012 ImageNet competition was the first successful application of a ConvNet to a large scale, realistic dataset for classification. With over a million training examples and 1000 object classes, the ImageNet competition was a significant step forward in terms of dataset scale. This competition had been running for 2 years prior with very modest improvements each year. This was due to winning entries relying mainly on hand crafted solutions which were not trained end to end. In 2010 and 2011, the top-5 classification error rates (in terms of accuracy) were 28.19% and 25.77% respectively (5 allowed predictions per example) which AlexNet lowered to 16.42% in 2012. However, the resulting model was quite large with 62M parameters and a model size of 250MB. Since then it has been shown that AlexNet was extremely over-parameterized [43]. This over-parameterization is due to the models 3 fully connected neural layers which comprise 94% of the total number of parameters in the model. It has been experimentally shown that approximately 90% of these fully connected parameters can be ignored while achieving the same level of classification performance.

Although AlexNet was revolutionary, its large size was very much the result of a reliance (at the time) on fully connected layers to perform classification. This pattern had originated in 1998 with Yann LeCun's LeNet-5 network [63] which performed the same task but at a much smaller scale in the context of image classification. However, in 2013 the Network in Network (NiN) architecture [66] introduced a specialized structure combined with global average pooling layers as a replacement for fully connected layers. Fully connected layers, while computationally cheap relative to convolutional layers, are extremely storage intensive due to the lack of weight sharing as compared to convolutional layers. The idea with global average pooling is to create a single feature map per target class for classification. Thus, for each of K NxM feature maps in a layer of size NxMxK, we compute an average of each feature map to build a new layer which is 1x1xK. The introduction of global average pooling, combined with the innovation of using 1x1 convolutional layers (feature pooling) meant the resulting model only needed 7.5M parameters (29 MB) and achieved almost the same Top-1 ImageNet performance as AlexNet (39.2% vs 40.7% top-1 error respectively) while taking half as long to train. This was the first significant result in the improvement of model efficiency since AlexNet.

In each subsequent year, the error rate fell dramatically as candidate submissions in the ImageNet competition switched to some form of end-to-end trained, convolutional based models. In 2014, the winning entry was GoogLeNet, or as it's also known Inception V1 [104], which leveraged the innovations from NiN with some of its own to lower the top-5 error to 6.67% when used in an ensemble (the base model had a top-5 error of 10.07%). Inception V1 was 22 layers deep and used branches which forked off the main network and contained fully connected layers, but these were only used during training in order to stabilize it and to help mitigate the problem of vanishing gradients. This is shown in Figure 2.5 with the right-most branch, beginning with an average pooling layer. After training, these branches were removed.

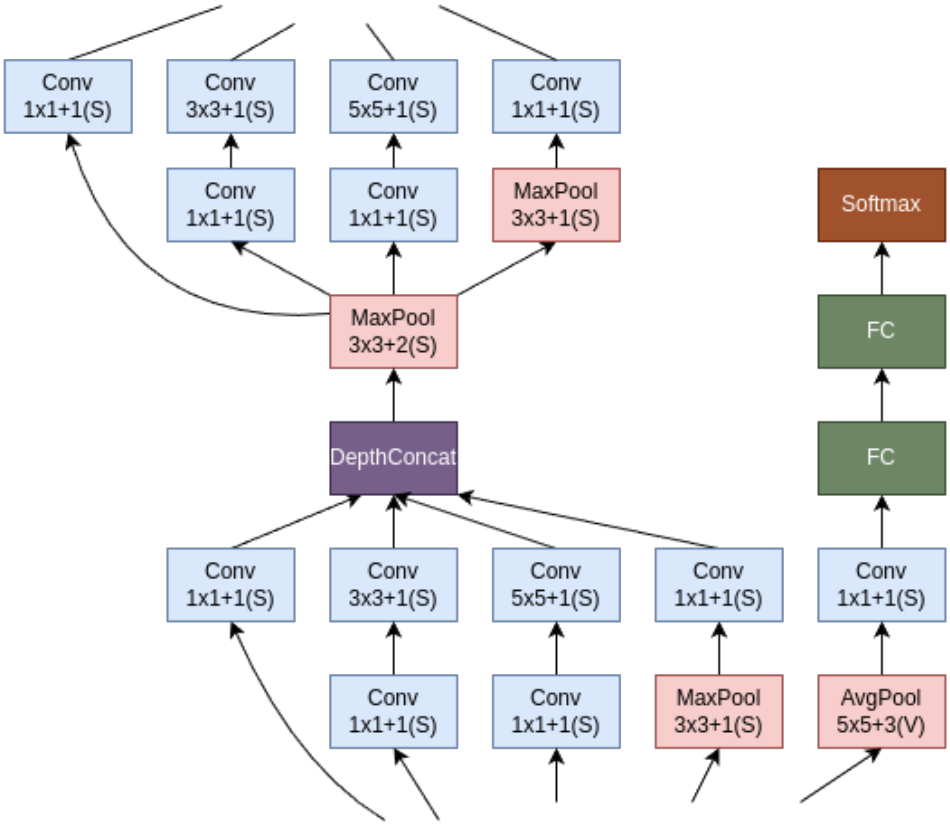

Figure 2.5: Inception V1 Architecture with branches

It is important to distinguish here between linear and non-linear fully connected layers. Linear fully connected layers have a linear activation function at their output. This means there is no benefit to stack such layers, since the transformation they represent can be collapsed into a single transformation layer. However, by adding a non-linearity at the layer output, we can introduce a non-linear dependence between a layers inputs and outputs, meaning stacking successive layers does make a difference. The sole fully connected layer in Inception V1 is a linear fully connected layer (Figure 2.6) and was used to help in transfer learning to new datasets. However this came at the cost of an extra 1M parameters and the original paper noted that replacing this layer with an average pooling layer actually improved performance by 0.6%.

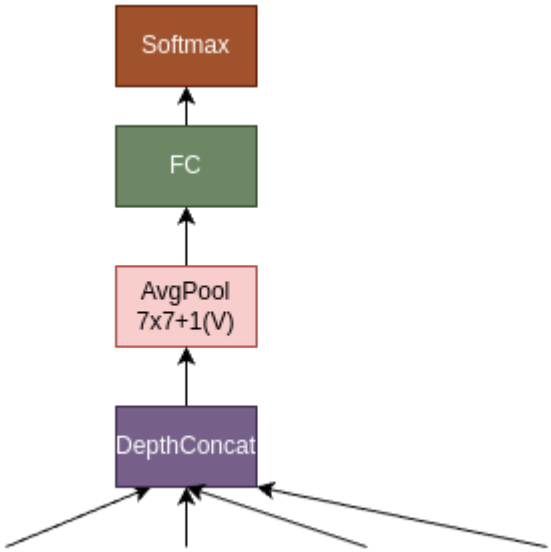

Figure 2.6: Output stage for Inception V1

An additional novelty introduced by Inception V1 was the Inception module, which borrowed the idea to an extent from the modular architecture of NiN. The core of this idea was to save parameters and computational cost by using small filter kernels (1x1, 3x3 and some 5x5). Especially numerous was the use of 1x1 kernels to 'bottleneck' the representation which was fed through 3x3 and 5x5 filters. A bottleneck occurs when the number of filters used by a layer is more than the number of channels in that layer. By applying 1x1 convolutions with a relatively small channel dimension to the input of the module, the representation from the previous layer was in a way 'squeezed' from a larger number of channels to a smaller one. Applying a subsequent 3x3 or 5x5 kernel was then less expensive because it was over a smaller number of channels. The stacking of Inception modules (Figure 2.7) of a similar structure reduced the need to hand tune the layout of each layer.

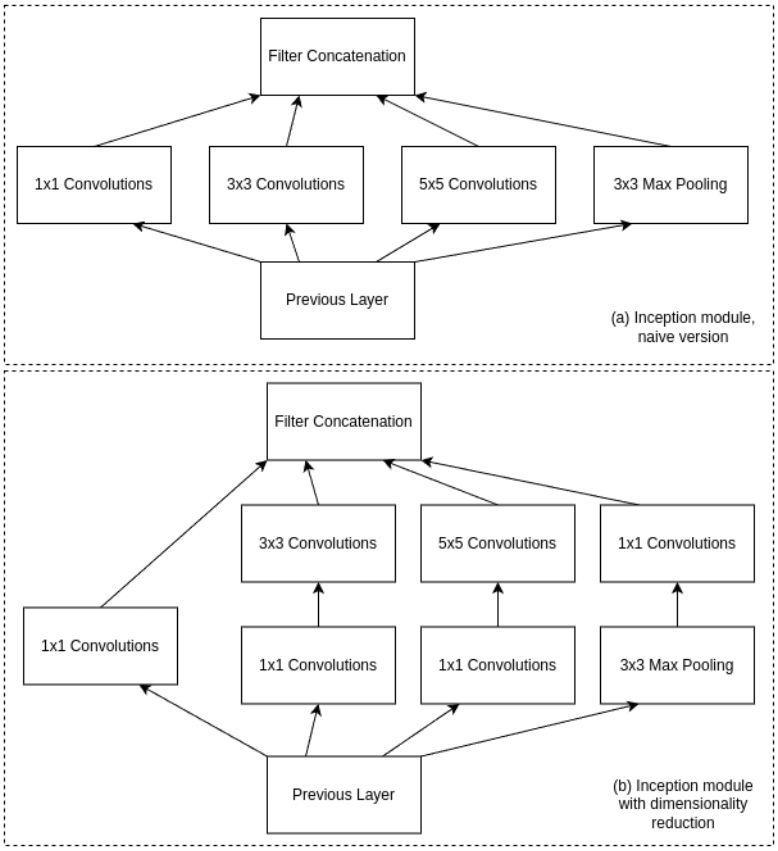

Figure 2.7: Inception Naive (a, top) and Efficient (b, bottom) Module Stacking

A similar concept of a channel bottleneck and repeatedly stacked modules was shown in the SqueezeNet model [53]. This model has 22 layers, the majority of which was a stack of 8 'fire' modules (Figure 2.8) with the same structure. This structure had 2 layers, the first of which was a 'squeeze' layer which applied 1x1 convolutions to the module input but contained a smaller number of channels than the input. This created a 'bottleneck' for the incoming representation, which occurs by reducing the number of channels seen by the subsequent layer in the module. The 'squeeze' layer was followed by an 'expand' layer which used both 1x1 and 3x3 kernels producing a minimum of 4x more channels for both. The output layer was a global average pooling layer to conserve parameters. This structure produced a relatively small model of only 4.8MB, with 1.25M parameters (50x smaller than AlexNet) and matched or exceeded the performance of AlexNet on ImageNet.

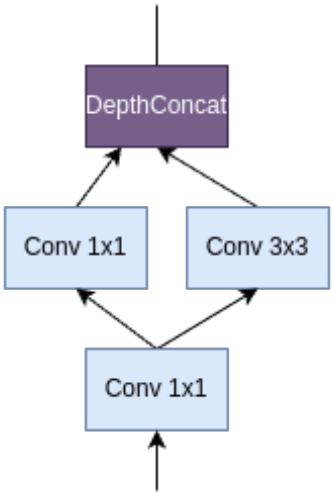

Figure 2.8: SqueezeNet Fire Module

The Inception architecture has also pioneered the exploration of separable filters in CNNs [105]. Separable spatial filters are spatial filters whose structure can be expressed as the product of 2 1-dimensional filters. This product produces a spatial filter which is constrained in its rank (the matrix expressing the filter has rank 1) however there is a savings of parameters in the process. This savings is because 2 1D filters (for example 1xN and Nx1) require 2N parameters while an NxN 2D filter requires N^2 parameters. The 1D filters are applied in sequence, first convolving the input with the 1xN filter followed by a convolution with the Nx1 filter. This combination gives an effective receptive field of NxN after both filters are applied.

The receptive field of a larger convolution can be approximated by repeated convolutions of smaller receptive field sizes. For example, a 5x5 convolution can be replaced with two 3x3 convolutions. Another variant of convolutional layers is that of depthwise separable convolutions, as shown by the Xception [17] and MobileNet style [15] networks. Here 'separable' is not in the previously mentioned spatial sense but along the depth dimension of a layer. A depthwise separable convolution consists of 2 stages (layers); a depthwise convolution followed by a point-wise convolution (regular 1x1 convolution). The depthwise convolution in the first stage has the same number of output channels as input channels, and each output channel applies 1 convolutional filter to 1 channel in the input (at the corresponding depth). Thus for an input to a layer of HxWxC, if we take C as the number of input channels and D as the number of output channels, the depthwise convolution has

a KxKx1xD kernel (each KxK is applied only to 1 corresponding channel in the input by each of the D output channels) instead of the KxKxCxD kernel found in a regular convolutional layer. This is a considerable saving in parameters and computation, and constitutes the 'depthwise separable' part of the name of this operation since each channel only 'sees' one channel from the previous layer (hence separable). Depthwise convolutions themselves are simply a special case of the more general group convolutions [61] [54] where a filter kernel of dimension KxKxCxD is only applied to C/N input channels. Thus each output channel D is computed from C/N channels in the input. In the case where the number of groups equals the number of channels (C=N), each output channel is computed from only 1 channel of the input and a group convolution reduces to a depthwise convolution. Group convolutions have been used to improve the efficiency of a popular type of convolutional model called Residual Networks.

Further improvements on ImageNet performance have come as a result of refinements to the Inception architecture [105] [103] and the introduction of feed forward connections with the ResNet architecture [46] which allows one to bypass the problem of vanishing or exploding gradients when build neural networks with many layers [51]. These refinements have focused mainly on classification performance and not on the reduction of overall computational cost (Table 2.1), although the exploration of group convolutions (especially to ResNet models) has created the ResNeXt variant of these architectures [115] which offer the same performance at a reduced computational cost and increased performance.

Model	Task	Input	Mparams	GFLOPs
AlexNet	Cls	341x341	58.25	1.56
InceptionV2	Cls	336x336	12.75	3.6
ResNet-18	Cls	336x336	11.25	4.2
ResNet-50	Cls	336x336	26.75	7.8
ResNet-101	Cls	336x336	47.25	15
RexNeXt-50-32x4d	Cls	336x336	26.25	15
MobileNetV2	Cls	336x336	3.5	1.05
SqueezeNet	Cls	336x336	1.25	0.64
DenseNet121	Cls	336x336	7.75	6.5
SSD-MobileNet	Det	336x336	5.5	1.5
RFCN-ResNet50	Det	300x425	30.5	15.6
YOLOv3	Det	416x416	61	65
YOLOv4	Det	416x416	64	61.5
FCN16s	Seg	384x384	514	125
DeepLabV2-ResNet-101	Seg	336x336	505	155

Table 2.1: Summary of various classification (Cls), detection (2D) and segmentation networks (Seg) with their model parameters (expressed in millions of params or Mparams) and computational complexity in GFLOPs.

When considering a neural network, there are various types of optimizations that can be added that help improve a network or its training process. This includes, for example, the optimization approaches described in Section 2.2 such as adaptive first & second order learning rates, learning rate schedules (to work in conjunction with the optimizer), gradient clipping (for stability), network initialization (better training starting point) or using mini batches. However there are some other very important techniques that merit mention:

- **Regularization** is a broad set of techniques to help reduce over-fitting and improve the generalization ability of a network. By introducing a penalty, for example to the loss, a constraint is imposed on the learning process. This can include for example L1 and L2 regularization through weight decay [7] to ensure weights either decay to 0 as a way to enforce sparsity (L1 regularization) or to shrink weights more evenly but not necessarily abring them to 0 (L2 regularization). Other forms of regularization include early stopping of training to prevent over-fitting or dropout [101], where connections, activations or entire feature maps are randomly deactivated during training to prevent co-dependence and to improve generalization.

- **Batch Normalization (BN)** [55] performs a normalization of the input (previous) layers inputs by learning and applying a mean and variance over the batch dimension of the input. These running statistics are meant to stabilize and speed up training by allowing higher learning rates and possibly mitigating the impact of bad initialization. This can have a regularizing affect to reduce over-fitting by introducing noise into the training process but this is not its primary function. Various other types of normalization exist such as layer normalization [4], instance normalization [106] and group normalization [114].

2.3 ConvNet Model Applications

This section looks at applications that leverage insights from the construction of efficient or specialized structures of ConvNet models (as described in the previous section). These applications include object detection (both 2D and 3D), point cloud processing and semantic segmentation. In our review of applications, we do emphasize 2D detection since the PointPillars model that is a major focus of this thesis, operates largely in the 2D Birds Eye View (BEV).

2.3.1 2D Object Detection

To understand 3D ConvNet detection models, it is important to first review 2D ConvNet detection models as 2D models are broadly used within 3D detectors and also are one of the central parts of the 3D PointPillars model.

2D detectors can generally be categorized based on whether they use a proposal network directly for classification and detection (so called single shot detectors) or use a multi stage detection process with a proposal network to select regions interest for further post-processing. Multi-stage detectors include approaches such as OverFeat [96], Regional Fully Connected Network [19], Faster R-CNN [91] (along with its predecessors Regional CNN (R-CNN) [39] and Fast R-CNN [40]) and Mask R-CNN [47]. Mask R-CNN is mentioned for completeness but is not investigated as part of this thesis since its main purpose is to add semantic segmentation capability to Faster R-CNN models. Approaches which don't have separate stages, called single-shot detectors, include Single Shot Detector (SSD) [70] and its variants Tiny SSD [112], Deconvolutional Single Shot Detector (DSSD) [34] and Deeply Supervised Object Detector (DSOD) [97] as well as You Only Look Once (YOLO) [88] and

its variants YOLOv2 [89], YOLOv3 [90] and YOLOv4 [8].

The earliest successful attempts at ConvNet detectors were multi-stage architectures and appeared around the same time in 2013. The two most significant early approaches appeared around the same time in 2013. OverFeat [96], proposed by Sermanet et al., won the ImageNet 2013 visual recognition challenge in the localization task while Region-based CNNs (R-CNNs), proposed by Girshick et al. [39], improved the state of the art Mean Average Precision (mAP) on the Pascal Visual Object Classes (VOC) dataset [30] by over 30%. The OverFeat model performs classifications on a sliding window, obtaining confidences of object classes which are then used to predict bounding box locations relative to the sliding window. OverFeat was quickly superseded by variants of R-CNN (it was outperformed by over 7% in mAP by R-CNN using a similar training and evaluation process in a post-competition test) in terms of speed and performance.

In the case of an R-CNN, the computation is summarized in Figure 2.9. The starting point is to generate crops of the input image (region proposals) using a selective search algorithm. This algorithm generates window crops by grouping pixels in the image based on intensity, color and other features. A large number of regions proposals can be generated by this algorithm per image, with thousands of proposals needed by R-CNN to obtain good performance. Proposals are then warped to a standard size and run through a feature extractor CNN network up to a desired layer (for example a fully connected layer after the convolutional layers). The resulting activations are then used as a feature vector for that proposal. The feature vectors for each proposal are then classified using a class-specific (including a background class) linear Support Vector Machine (SVM), and the proposal region position and size are adjusted using a separate (class specific) linear regressor for localization. The addition of an SVM meant it was not end-to-end trained. The feature extractor, SVM and localization regressor were not end to end trained. This meant that training the SVM classifier and bounding box regressors required significant storage cost (100's of GB) since region proposals for images in the training set needed to be precomputed first. Due to the large number of proposals, there would be a high degree of overlap between them. Since computation was not shared, a separate inference would occur for each proposal meaning that the inference time for a single image was several seconds even on a high end GPU.

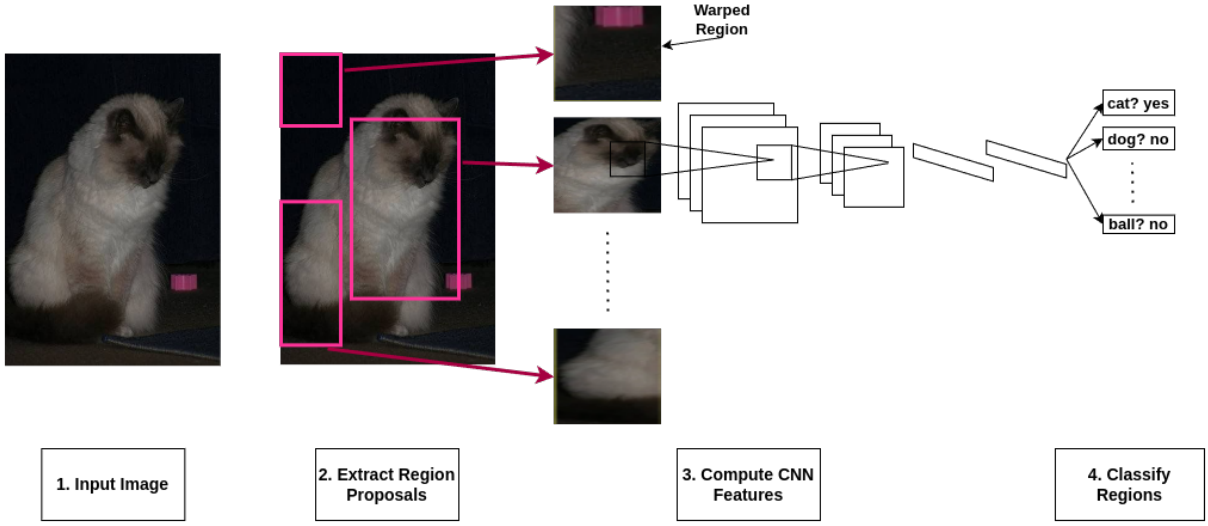

Figure 2.9: R-CNN detection pipeline [39] (scene image from [30])

The two main shortcomings of R-CNN, that proposals were generated using a hand crafted algorithm and that the detector was not end-to-end trained were addressed by two subsequent papers. First, Fast R-CNN [40] made the pipeline (with the exception of the proposal stage) end-to-end trainable by using the concept of Region of Interest (RoI) pooling as shown in Figure 2.10. A layer in a feature extractor is first selected to be the layer where the RoI pooling is applied and a fixed grid size HxW is chosen. Given a region proposal on the input image, we project the proposal region to the feature layer in our feature extractor. We then overlay an HxW grid on this region of the feature layer and apply max pooling to the activations inside each gridbox. This yields a fixed HxW output grid regardless of the size of the proposal regions. The remaining task is to apply additional layers to generate a final RoI feature vector which we feed to 2 separate output branches; classification and localization regression.

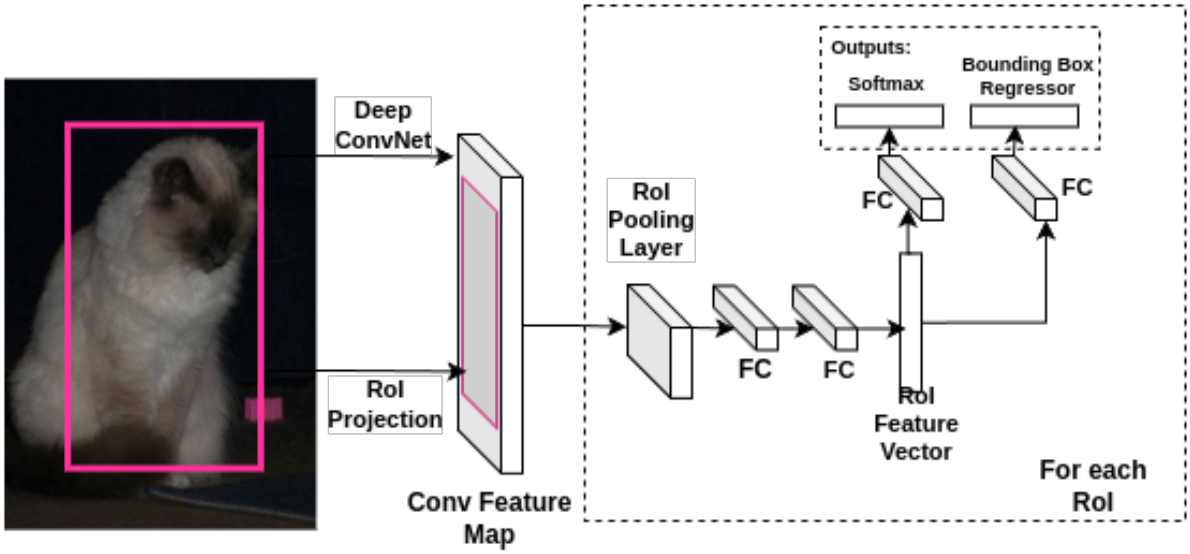

Figure 2.10: Fast-RCNN detection stage with RoI pooling [40] (scene image from [30])

The classification stage is simply a softmax, having replaced the R-CNN SVM while the localization is performed with a regression layer which generates (per-class) offsets and scales for the corresponding RoI regions offset and scale relative to the top left corner of the image. A multi-task loss was introduced to simultaneously optimize classification and regression with a single loss function, allowing (almost) complete end-to-end training. These optimizations brought a training speedup of 9-18x and inference speed-ups of 80-146x for the networks tested (AlexNet variant and variants of VGG16) on a high end GPU. Softmax for classification was shown to outperform the SVM for Fast R-CNN (but not for R-CNN), allowing the authors to drop the SVM. The multi-task loss function allowed optimization of classification and detection simultaneously, without having to resort to separate optimizations for each (SVM for classification and least squares for localization).

The proposal of Faster R-CNN by He et al. [91] overcame the remaining limitations of Fast R-CNN, with the introduction of the Region Proposal Network (RPN) to replace the selective search function. The RPN in this paper is at the output of a Fully Convolutional Network (FCN), which is a type of CNN popular for semantic segmentation [71]. The RPN is responsible for generating and ranking 'anchors' (rectangular regions) to propose the ones which most likely contain an object. The network structure downstream of the

proposed regions remains the same as Fast R-CNN, namely the ROI pooling and branches for classification and regression. The RPN begins by looking at the feature map activations at the last convolutional layer of the FCN. Each position in the feature map (the layer has dimensions HxWxC) can be seen as corresponding to a position in the original input image. Adjacent positions in the feature map will not be adjacent in the input image, because of the size difference between the input and the feature map, and there will be a scaling factor q which represents the relative difference. The input image size will then be qH x qW x 3, and q will represent the stride between each spatial position on the input image which corresponds to adjacent, respective positions on the feature map. The first stage of the RPN uses the HxWxC layer activations and applies an n x n x p kernel to it to generate an intermediate layer (in the paper a 3x3 kernel is used with p varying depending on the architecture; 256 for ZF and 512 for VGG-16). Depending on the structure of the FCN, including its depth and stride sizes, different effective receptive field sizes are possible based on the size on the n x n kernel. For example, the original VGG-16 network used by the authors, using a 3x3 kernel meant an effective receptive field on the input image of 228 pixels.

Each position in the input image is treated as the center of a set of anchor boxes. Anchor boxes are selected based on size and aspect ratio of the objects one expects to find in the ground truth dataset. For example car and pedestrian bounding boxes will have different sizes (car bounding boxes will tend to be bigger at the same distance) and aspect ratios (car box aspect ratios will typically be > 1 while person boxes will tend to be < 1). Implicitly, the selection of anchors will select for a default distribution of possible physical distances of classes within the scene. The selection would typically occur based on the training dataset (assuming the dataset is representative of what you are trying to detect), for example by clustering box sizes into one or more groups per class and then selecting a representative box for each cluster. In the original paper, the authors proposed 3 scales and 3 aspect ratios, giving $k = 9$ anchor boxes for each position. The intermediate layer is then branched. One branch is dedicated for classification and, for each of the k anchor boxes, two feature maps are computed using a 1x1 convolutional kernel. The two feature maps represent the 'object' and 'not object' classes. The second branch is for localization and is meant to compute the manner in which the anchor box is to be deformed to obtain the 'true' bounding box for an object. For each of the k anchor boxes, four feature maps are computed representing the horizontal & vertical shift as well as change in height and width for an anchor box. The result is that there is a (4+2)*k channel output following the intermediate layer which is responsible for determining classification and localization regression for the input relative to a set of default anchor boxes. In order to eliminate proposals with a high degree of overlap, a Non-Maximum Suppression (NMS) stage is applied

to proposal regions based on the 'objectness' class scores and using an IoU threshold of 0.7. The result is a set of N proposals ranked in terms of their 'objectness'. The user can select the number of proposals they desire based on run-time and performance constraints. The work in the Faster R-CNN paper showed as few as 300 proposals could be used at run-time to obtain the best results and subsequent comparative work with other detectors discovered as few as 30 proposals [52] could be used, while maintaining only a very small drop in mAP on the PascalVOC and COCO datasets.

Regional Fully Connected Network (R-FCN) were proposed by Dail et al. [19] to overcome what the authors called an unnatural pairing between the ResNet architecture [46] and Faster R-CNN. Unlike the architectures used for the original Faster R-CNN paper, ResNet is almost a fully convolutional architecture, with pooling operations being very rare relative to convolutional operations. This is to preserve the spatial dimension of feature maps for residual connections, however it also introduces a problem when considering ResNet for detection. In a CNN, the convolutional layers of the network are equivariant w.r.t the network input because the convolutional operator itself is equivariant to translation. On the other hand, the max pooling operations in a CNN allow for the network to become translation invariant to some degree since they are invariant to small deformations within their receptive field. With the removal of most pooling operations (ResNet), a problem may be encountered when building a detector since a detector needs to have a degree of translation variance so that it is not too sensitive to position changes (but not so insensitive as to have too much translation variance, in a sense a kind of sweet spot). In such a case moving an object inside a candidate proposal box should produce a changing response related to the change in overlap between the object and the proposal box.

The solution up until this point had been to insert the RoI pooling at some intermediate layer within the ResNet architecture and continue with training the remaining stages of Faster R-CNN. However this meant removing layers which contribute to ResNet's excellent performance and introduces a number of layers which must now use available computation for region specific tasks. To alleviate these issues, the authors proposed R-FCN, which introduces a layer of feature maps after the last convolutional layer in the network (in the paper an additional 1x1 convolution layer was added with half the number of channels) that compute position sensitive scores for detection (Figure 2.11).

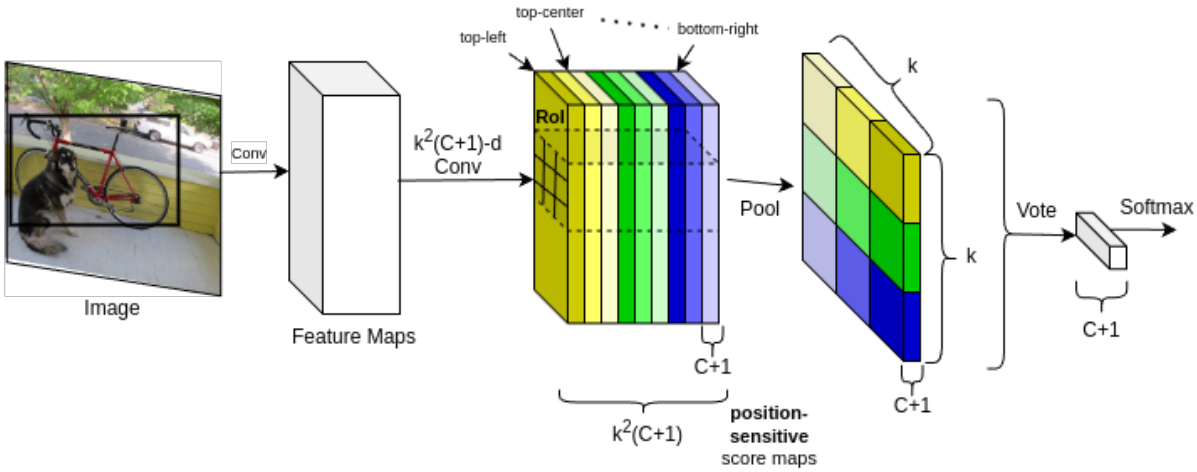

Figure 2.11: R-FCN for object detection, including the score maps and 3x3 (k x k in general) RoI pooling for C+1 classes (+1 for background) [19]

The score maps are position sensitive, in that for each class there are k^2 feature maps with each feature maps responsible for a corresponding region within the 2D k x k area. The RoI pooling operation is then applied per class over the RoI region (size h x w) in the score maps, pooling the activations by averaging them over the h x w region into a k x k grid (each gridbox is approximately (h/k) x (w/k)). This gives an RoI pooled output of k x k for each of the C+1 classes, which is then averaged (for simplicity) for each class and followed by a softmax for classification. This operation is performed for each region proposal. Regression in R-FCN is handled in a similar way, but is class independent, with $4k^2$ score maps which are RoI pooled in the same manner. The k^2 are then averaged producing a 4 dimensional vector t = (tx, ty, tw, th) which parameterizes the proposed bounding box. The authors show that when applied to ResNet-101 and Pascal VOC, a performance increase of 7.7 mAP can be achieved relative to Faster R-CNN and a similar result to Faster R-CNN on the MS COCO dataset but running 2.5x faster. We can see that the primary difference between Faster R-CNN and R-FCN are the positive sensitive score maps. Unlike Faster R-CNN, which generates proposals and extracts features for RoI pooling from the same layer (to save computation), R-FCN adds intermediate layers between the feature extraction layer used for proposal generation and the layer where crops are taken (score maps). This can have an advantage in cases where the number of classes is reasonable.

In the case of single-shot detectors, SSD and YOLO are the most popular and their variants are state of the art in single-shot detection. Due to the lack of an explicit region proposal stage, these detectors instead only use a collection of regular grids over multiple scales to perform classification and localization. This is different from R-CNN and R-FCN based detectors, which use a proposal stage which first decides an 'objectness' score for proposed regions and for those regions which meet a minimum 'objectness' score will post-process those regions further to determine their actual class and true location. In the case of single shot detectors, classification and detection is performed directly from the outputs of the feature extractor. In this sense, YOLO is a little bit different from SSD since it does compute an 'objectness' score at the same time as the location and class scores over a regular grid.

YOLO was proposed by Redmon et al. [88] around the same time as Faster R-CNN, and although there are some similarities between these methods, YOLO is a single-shot detector. With YOLO the input image is split into an S x S grid (Figure 2.12), and each position in the grid is assigned B bounding boxes. A bounding box is simply 5 predictions, consisting of the (x,y) coordinates of the center of the box (relative to its gridbox), the width and height, which are normalized to be between 0 and the width & height of the image respectively, and finally a confidence score. The confidence score is in a way similar to the object/non-object classification performed by an RPN which performs object/non-object classification for proposal regions. In the YOLO paper, the authors train on Pascal VOC and used a 7x7 grid (S=7) with B=2 bounding boxes per grid cell. Since Pascal VOC has 20 classes, there are 5*B+20 feature maps to predict, which gives a 7x7x30 output.

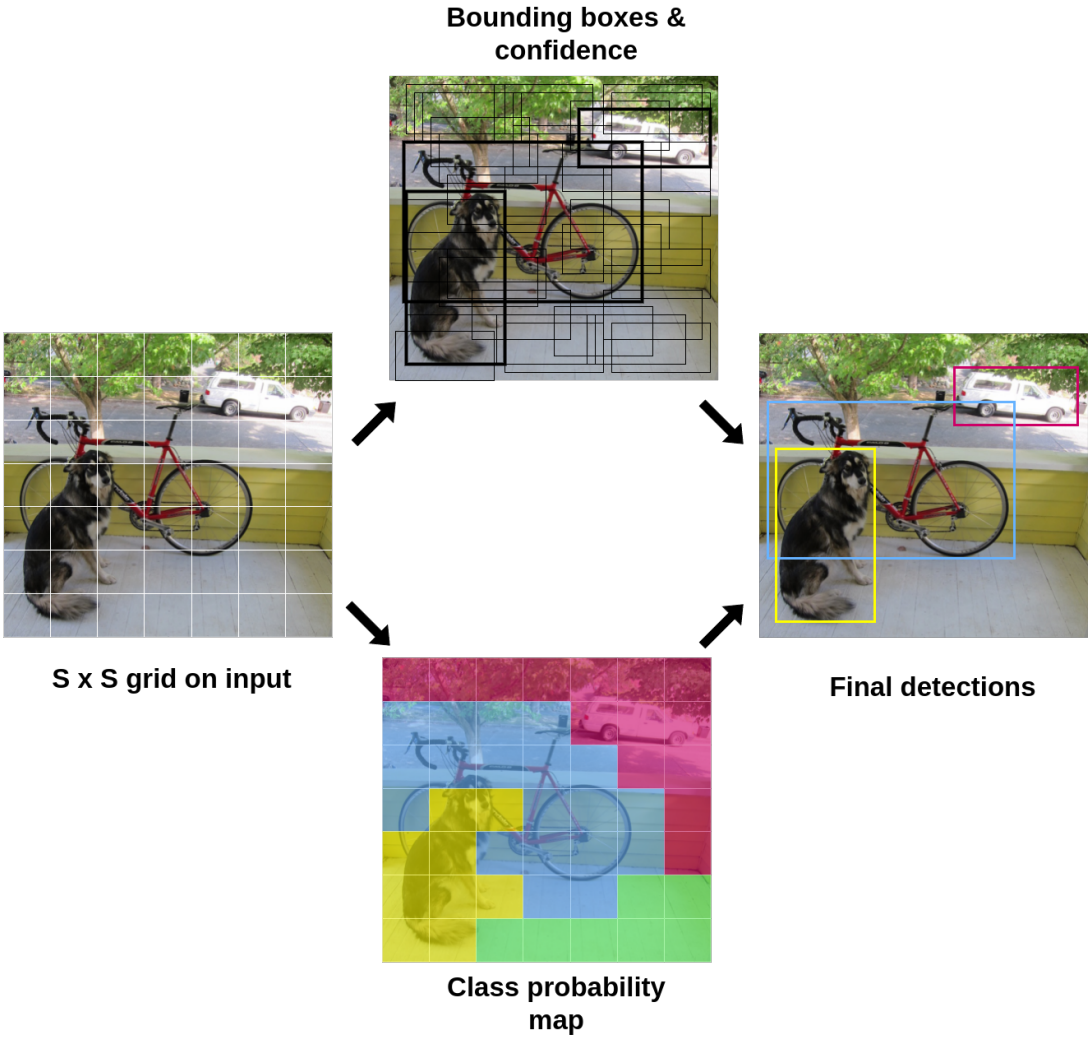

Figure 2.12: Detection process for the YOLOv1 architecture (scene image from [30])

The confidence score is defined as $Pr(Object) * IOU_{pred}^{truth}$ where the Intersection Over Union (IOU), also called the Jaccard Overlap, is computed between a ground truth bounding box and the predicted box. This modification to $Pr(Object)$ occurs either at training time or at "test time" (as stated by the authors), when the ground truths are known and the true IOU can be computed between prediction and ground truth. At inference time, the raw value of $Pr(object)$ is used without the IOU modification. A ground truth box is assigned to a gridbox based on whether the center of the ground truth is in that gridbox,

with the target confidence score set to the IOU value (as the probability of an object) or 0 if no ground truth box is centered in that gridbox. This means YOLO has difficulty distinguishing many objects close together (ex. a crowd of people) when the selected grid is too coarse, requiring careful tuning of the grid relative to the task. Finally, each grid cell (regardless of the number of boxes) has C class conditional probabilities $Pr(Class_i|Object)$ for each target class. Multiplying the confidence with the class conditional probabilities gives $Pr(Class_i|IOU^{truth}_{pred})$, the class specific confidence scores for each box. The confidence and class conditional probabilities together encode the probability of a class being in the box and how well the bounding box for a cell predicts an object.

For training the YOLO detection model the authors first construct a custom classification network which has 20 convolutional layers, followed by max-pooling, a fully connected layer and softmax. This network is similar to NiN (and also to subsequent VGG and SqueezeNet networks), with alternating 1x1 and 3x3 convolutional layers. It is also similar to Inception V1 (GooLeNet) in that pairs of 1x1 followed by 3x3 layers are stacked repeatedly in parts of the network (a module style approach). The network is trained in a framework called Darknet and so the network is called simply Darknet. After pre-training and removal of classification related layers, Darknet has an additional 4 convolutional layers added, followed by a fully connected layer, followed by a YOLO specific layer which is 7x7x30. The overall model is shown in Figure 2.13.

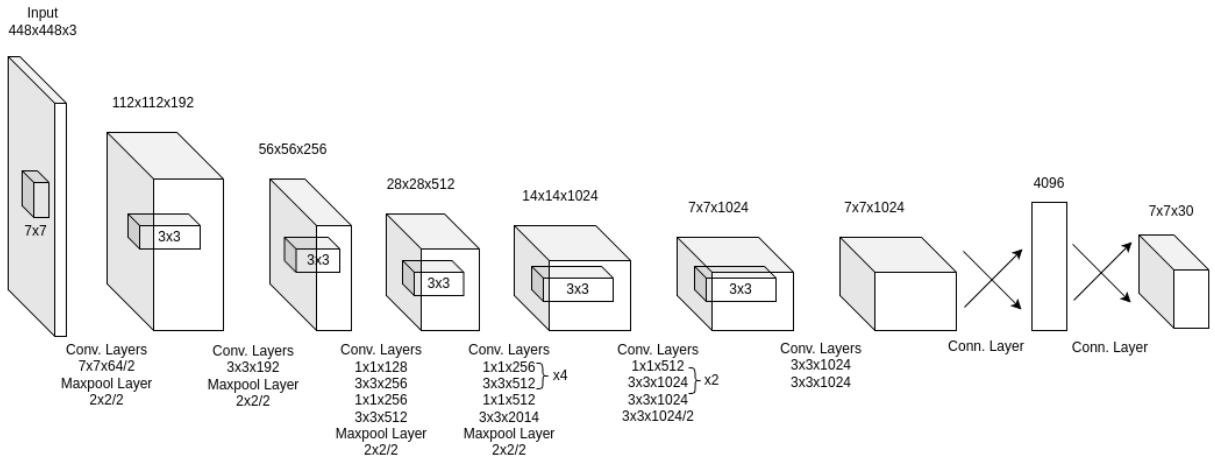

Figure 2.13: YOLOv1 Architecture [88]

The Darknet classifier is trained at 224x224, however the detector is trained at a higher resolution (448x448). The subsequent YOLOv2 paper shows that several mAP points

of performance can be gained by fine-tuning the classifier at 448x448 for a short time before training the detector. During detector training, the authors use a multi-part loss similar to Fast and Faster R-CNN which computes loss for classification and box regression simultaneously.

Several variants of YOLOv1 have appeared, starting with YOLOv2 which adds a number of iterative changes to improve YOLOv1. These include a modified classification architecture, Darknet-19, batch normalization for all layers and the mentioned higher resolution fine-tuning for detection. However, the most noticeable change is that of anchor boxes similar to that used with other detectors, including R-CNN variants as well as SSD (YOLOv1 did not use anchor boxes and regressed box dimensions directly). However, the authors explicitly investigate determining a more optimal selection of anchor boxes with which to initialize the detector. In particular, they use k-means clustering of the Pascal VOC and COCO datasets to find anchors. They cluster based on IOU between a centroid and ground truth boxes, defining the distance:

$$d(box, centroid) = 1 - IOU(box, centroid) \tag{2.12}$$

This distance is different than the standard Euclidean distance, where larger boxes generate larger errors than smaller boxes. The authors found that, as expected, clustering by IOU does improve the average IOU of the resulting clusters. For 5 clusters this is an improvement of 2 IOU relative to Euclidean distance which for 9 clusters, there's an improvement of almost 7 IOU relative to the hand chosen anchors used in Faster R-CNN. The number of clusters still needs to be hand chosen, which they fix at 5, which means there are 5 anchors per position in the detection feature map.

Another improvement in YOLOv2 is that of multi-scale training. For YOLOv2, the authors briefly fine-tune the classifier at 448x448 however they train the detector at various resolutions. Every 10 batches, the authors random change the image dimension following multiples of 32 between 320 and 608; the smallest image is 320x320 and the largest 608x608. The final network is therefore trained at a number of resolutions instead of a single resolution, allowing one to select the input resolution based on a speed vs. accuracy tradeoff. Following YOLOv2, Redmon introduced YOLOv3 which was an incremental improvement over YOLOv2. This included a different base architecture with residual connections (Darknet-53) and a feature pyramid for prediction at 3 different scales (feature maps of different sizes). A YOLOv4 variant was subsequently released [8] that again improved the Darknet backbone with CSPDarknet53, a different activation function from the standard ReLU and some additional training augmentations. The relative complexity of YOLO variants (Table 2.1) and their specialization for feature-rich image detection tasks

means they were not relevant for this thesis, although we do discuss them further in the context of 3D detection in Section 2.3.3.

In addition to YOLO and its variants, SSD has been the other dominant one-stage CNN detector architecture. The "SSD"-ness of the approach mainly refers to the selection of feature maps and how they are grouped together to make a prediction (classification of a box and regression of its position and size). SSD is used as part of the baseline PointPillars model. In the case of the original SSD [70], the detector is constructed by selecting a set of intermediate layers in a feature extractor CNN. The number of layers selected depends on the scales one is interested in detecting objects over (it is dataset and target task dependant). For example, selecting a layer where feature maps are 19x19 means a finer scale (finer grid of boxes) than selecting a layer where feature maps are 2x2. Therefore, the selection is largely dependant on the expected input size (for example 300x300). An example is shown below in Figure 2.14, with an input image having 2 ground truth bounding boxes. Two grids, 8x8 and 4x4, are shown for perspective and demonstrate the importance of scale where the finer scale is more suited for smaller objects and the coarser scale is more suited for larger objects. This also contrasts SSD with the original YOLO algorithm, which used a single scale feature map for detections.

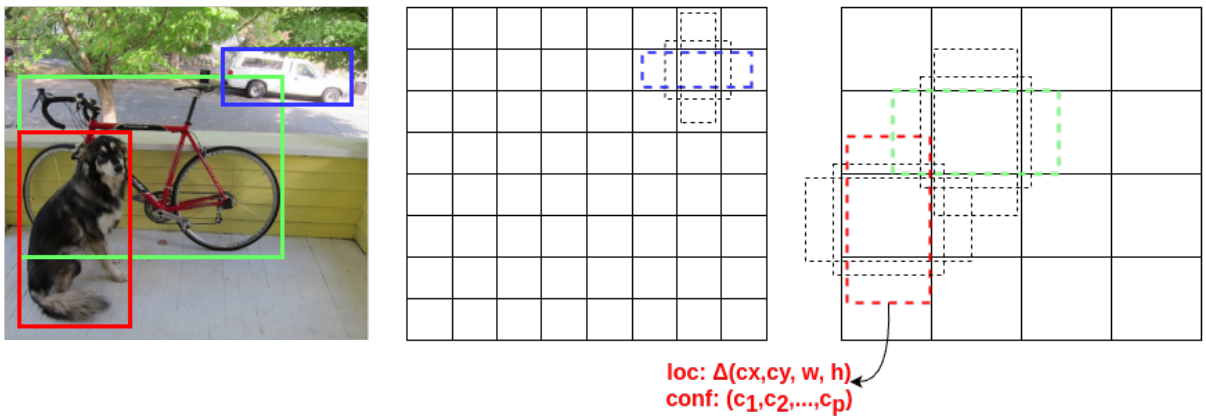

Figure 2.14: SSD grids (8x8 and 4x4) overlaid on an image with 2 ground truth boxes (scene image from [30])

Having selected a set of layers, a position in any feature map in a particular layer corresponds to the center of one of the boxes in the grid. Thus the (i,j)'th activation in an 8x8 feature map at layer L is the (i,j)'th grid box center in the 8x8 grid. Each of these grids has a number of default anchor boxes applied to it, which vary in their scale and

aspect ratio. If there are C classes (+1 for background) to be detected and there are K anchor boxes for a single feature layer, then for each of the K anchor boxes we will need to predict C+1 classes and 4 box offsets (2D position, width and height) relative to the original anchor box. That means for an MxN feature map, K(C+1+4) channels (P) with a total of MNK(C+1+4) activations will occur at a single scale. Each of the selected feature extractor layers is branched from and a convolutional layer having a 3x3xP kernel (that layer has P channels) is applied to generate the SSD detection layer for that particular grid size. The anchor boxes in SSD are similar to Faster R-CNN anchors but are applied to feature maps at different depths (resolutions) in the network. An example of these branches is shown below in Figure 2.15, where the filter kernel dimensions for each detector branch are shown (with the background dimension omitted). Note that additional layers for coarser grids can be added post-hoc, meaning they need not have been in the original base feature extractor. The final stage after generating classifications and box regressions is to apply NMS to combine predictions.

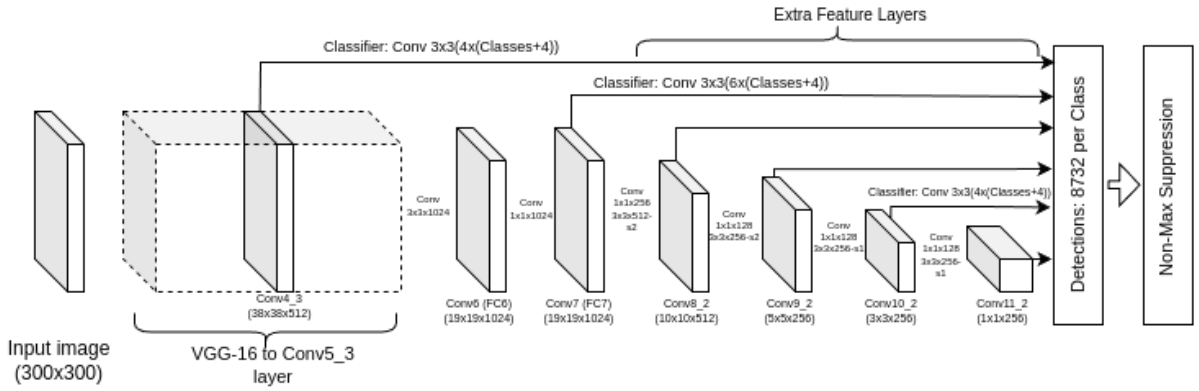

Figure 2.15: SSD VGG-16 architecture with detection branches [70]

The training strategy for an SSD network is to match each ground truth box with its set of default boxes across all detection branches. The default box with the highest IOU is matched first, followed by any ground truth boxes with a Jaccard overlap greater than 0.5. This allows multiple, overlapping default boxes to participate in the computation of the loss function and also to find the best fit for a object. However it also means multiple detections will likely occur per object, which necessitates the used of NMS to select a single 'best' prediction. Default boxes which don't match any ground truth boxes are

assigned to the 'background' class. During training, most default boxes will be negative (will not overlap with any ground truth boxes in an image). This creates a significant imbalance between positive and negative examples for each frame and must be balanced with hard negative mining. As such a ratio between positive and negative examples must be picked (the original paper used a ratio of 3:1 for negative to positive examples). Negative examples are sorted in decreasing confidence and then selected based on the number of positive examples.

2.3.2 Point-Cloud Processing

A cloud of 3D points can be represented as a size N list of tuples with tuple having 3 real coordinates representing the XYZ position of that point. The resulting list of tuples can be seen in 3D as a sparse point cloud, representing locations where some return was observed from the environment (for example a return from a LiDAR sensor). Prior to the development of end-to-end machine learning models for feature extraction from such clouds, these representations were largely processed to produce either hand crafted features (Spin Images (1999) [57]), visual similarity measures (2003) [12], shape classification (2007) [69], Point Feature Histograms (2009) [94], and volumetric representations that included early 3D ConvNet models (2015) [74].

The PointNet model, shown in Figure 2.16, was proposed by Qi et. al. [85] as a way to avoid expensive 3D convolutions, while encoding geometric data from a point cloud in a direct way, allowing global and localized classification of points while preserving the permutation invariance of the clouds. This invariance comes from the fact that point clouds are inherently unordered, and a permutation of the points yields the same structure. When input to a network, it is desirable to have the network treat the permutations in an identical way. A key insight for PointNet is the use of the max-pool function which, regardless of the point order, produces the same output at that layer. This was the key to the authors work, where they mathematically prove that PointNet with max-pooling can act as universal approximators for set functions. Given any function that inputs a set of points and produces a fixed-size output, there is a PointNet (or PointNet-like) architecture that uses max pooling that can approximate this function to any desired level of accuracy. The original PointNet inputs an Nx3 matrix of points and passes these points through several fully connected layers to finally compute a global feature representing the input. The original PointNet has 2 distinct outputs: k global scores as a prediction for the entire cloud (using the global feature result) and a separate branch of NxM scores where M is the number of per point class scores for each of the N input points. This latter output provides a kind of localized or segmentation score of each point in the cloud.

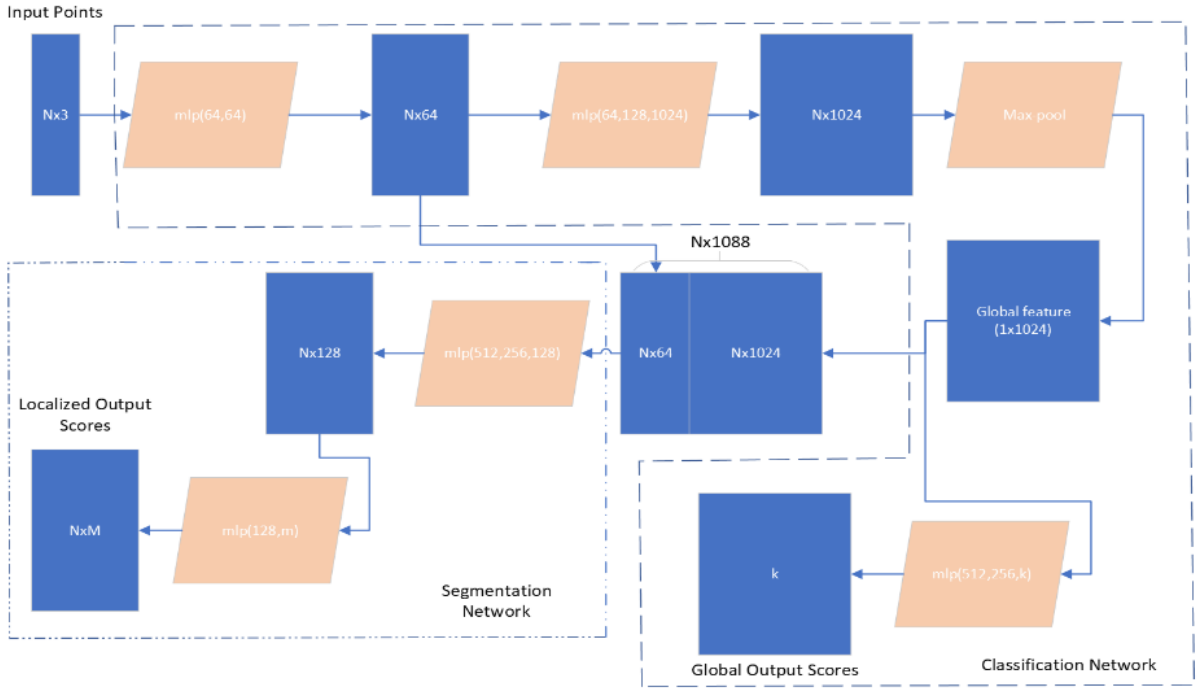

Figure 2.16: Overview of the PointNet architecture. The input to PointNet is an Nx3 point cloud (N 3D points). Here "mlp(X_1,X_2,...,X_J)" are J fully-connected (multi-layered perceptron) layers. There are k total "global output scores" that represent object classes that we want to generally classify for a cloud. There are an additional M, "localized output scores" that are set for each of the N points in the cloud. These localized scores represent part classes, where each of the M components classifies a different part.

To extend the PointNet model to capture local structure, the PointNet++ model was proposed [86] which applied PointNet on nested partitions of the input cloud. This introduced a hierarchical approach to processing the point cloud and allowed the simultaneous capture of local and global features. Although an improvement to PointNet, for our purposes, only a part of the PointNet model is sufficient to encode input features while minimizing the amount of computational complexity. Therefore, only a stripped down version of the "Classification Network" portion of PointNet is used within PointPillars, retaining one or two FC layers, the critical max-pooling operator and one more FC layer after max-pooling.

2.3.3 3D Object Detection

In this section we cover 3D detectors that rely on the usage of ConvNets. The task of 3D detection is not necessarily restricted by the sensor used (Section 1.1). For example, it can be performed by multi-camera setups that estimate depth to feature points or in more extreme (but rare) scenarios by single camera (monocular) setups [16]. Estimated depth can then be projected to a 3D cloud, in an approach called Pseudo-LiDAR [117]. Other sensor readings, such as from RADAR or ultrasonics, can also be processed to yield point clouds, although with ultrasonics this is more rare due to their range and specific application. This means that models that are shown in this thesis to be trained on LiDAR-based data are not necessarily restricted to being trained and/or tested only with LiDAR-based point clouds.

3D Object Detection models have evolved significantly in the last decade, mainly driven by the increasing use of ConvNets and end to end training pipelines. These developments have been driven in part due to the increasing maturity of 2D detectors as described in Section 2.3.1 where 2D detectors first matured to give reasonable 2D performance (R-CNN, SSD) and were then repurposed for 3D detection. When moving to 3D however, the dense image based features available for purely 2D-based detection are replaced with sparse point cloud data that must be represented and processing efficiently to prevent possible artificial limitations of sensor ranges or reducing feature resolution , below acceptable levels, in an already sparse modality. Prior to the ConvNet era, 3D detection relied on hand-crafted features [32] [95] extracted from images & depth maps (and to some extent point clouds) possibly with a combination of traditional classifiers such as SVMs or decision trees. With hand crafted features, the feature extraction may be less computationally intensive and the interpretability may be higher, however scaling to larger datasets and real world scenarios becomes harder due to varying scenes, levels of lighting (in the case of camera usage for 3D detection), object occlusions (and variability); this is especially true for unseen, out of sample data. With the use of ConvNets, hand crafted features became much less important (with the exception of hand crafted meta parameters).

MV3D [15] was one of the early successful ConvNet-based approaches that fused LiDAR point clouds in the BEV with Red/Green/Blue Colored (RGB) images. This was followed by VoxelNet [119] which grouped point clouds into a dense 3D grid of regions called voxels and used 3D convolutions for feature extraction on this 3D grid giving a large jump in performance (at high computational cost due to the expensive 3D convolutions). To mitigate the cost of 3D convolutions, specialized sparse convolutions were introduced by SECOND [116] to offset some of the computational cost of 3D convolutions, where relative spatial positioning of points was maintained to allow selection of neighboring

points for convolutional operations. However, the introduction of PointNet (described in Section 2.3.2) to represent point clouds in a learnable way opened new research areas by providing a method that didn't require hand crafting features to encode the information in a point cloud. That meant it would be up to the network to find better representations for groups of cloud points as part of the learning process. This paved the way for the introduction of models like PointPillars [62]. This approach uses "pillars" where only 1 subdivision in the vertical direction is made, meaning a voxel will extend indefinitely in the vertical direction (within an overall predefined bounding volume). The traditional voxel, of which there may be many in the vertical direction along a column, therefore becomes a "pillar" and there are no voxels/pillars stacked on top of each other as there would be with methods like VoxelNet. The pillars are still packed side-by-side which when viewed from above the scene (above the point cloud), which from that view can be seen as a 2D grid. With a PointNet encoder deciding on the best feature representation for input 3D (LiDAR) points, the resulting 2D grid meant a 2D detector could now be used (such as SSD) in the BEV to generate 3D box predictions instead of a specialized 3D detector.

The sparse feature density of point cloud data also has implications regarding feature map resolution and the density of anchor boxes. Sparse features mean a higher feature map resolution is needed to ensure proper localization of predicted 3D boxes. The larger feature maps in turn need to contain enough ground truth anchor boxes to regress to. In the case of SSD, the use of focal loss to help with dense anchor box predictions makes SSD ideal for a feature-sparse detection task where a less dense representation, such as with YOLO, may in fact struggle due to its sparser grid representation. For example, the Complex-YOLO model [100], including its YOLOv3 [38] and YOLOv4-based [75] variants, struggles even in the BEV-only 3D task on the KITTI dataset. This is even with those models large total computational complexity and parameter advantage.

2.3.4 Semantic Segmentation

Semantic segmentation involves labelling all pixels in an input image with one or more classes. Initial work in this area focused on image processing techniques (edge detection, thresholding) but gave way to hierarchical approaches using ConvNets. For example, Fully Convolutional Networks (FCNs) [71] applied pixel wise segmentation over an input image, with high parameter savings by using only convolutional layers (no FC layers). The authors introduced upsampling using transpose convolutions and skip connections for retaining low level features while at the same time having limited context of the overall scene. The DeepLab series of models [13] [33] [14] were a noticeable improvement that addressed capturing larger fields of view and information from multiple scales, with the introduction of

atrous convolutions (also referred to as dilated convolutions, where the "dilation" involves spreading the convolutional kernel over a larger area of the input) and a novel Atrous Spatial Pyramid Pooling layer. The use of CRFs for post processing was computationally intensive, trading speed for increased segmentation performance.

2.4 Computer Vision

This section describes the background material for understanding the sensor configuration and available data structures for the KITTI dataset, discussed in Section 2.5.2. For completeness, additional details are presented regarding stereo calibration and correspondence, as they help frame the explanation regarding the transformation of KITTI data across coordinate frames.

2.4.1 Camera Calibration

Cameras are complex systems, with lenses and image sensors that must be modelled in a manageable geometric form to allow for inferring its details. The ideal pinhole camera model, shown in Figure 2.17, is an idealized model meant to provide a simplified view of the image formation process. For a 3D point P in the camera coordinate frame given by $[X, Y, Z]^T$ (relative to center of projection O), we can take the point and scale it using a non-negative parameter λ such that it falls onto an image plane at point $q = [u, v, 1]^T$. Any 3D point with $Z \neq 0$ can be projected in this way, and we have:

$$P = \lambda q \tag{2.13}$$

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \lambda \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \tag{2.14}$$

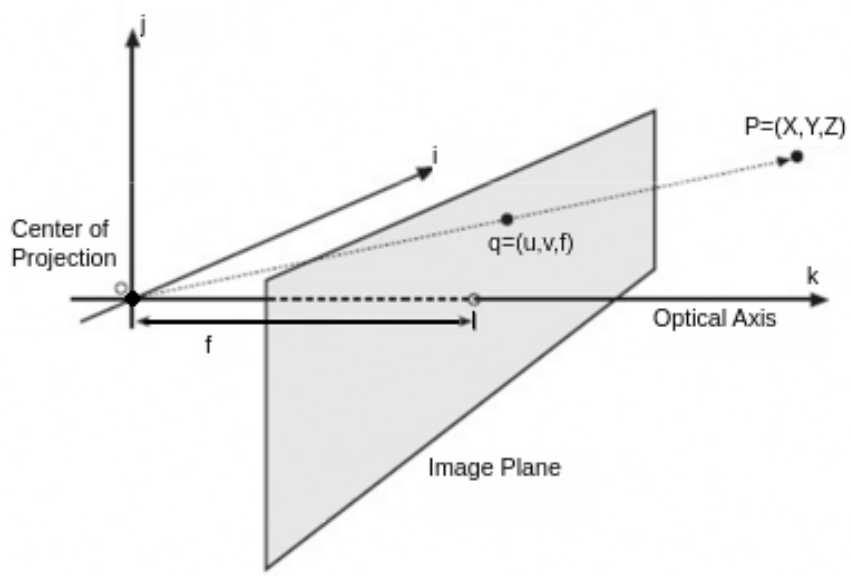

Figure 2.17: An illustration of the pinhole camera model for a 3D point P as it projects to point q on the image plane

With the ideal pinhole model, the world coordinate frame also has its origin at the center of projection O. The coordinates u, v are given by $u = f\frac{X}{Z}$ and $v = f\frac{Y}{Z}$. However, the camera and world coordinate frames need not correspond in general. Under the general pinhole camera model, these frames can vary. There the point P can be seen as point P_W in the world frame and P_C in the camera frame, where under a rigid body transformation determined by rotation $R \in \mathbb{R}^{3x3}$ and translation $T \in \mathbb{R}^3$, we have the relationship:

$$P_C = RP_W + T \tag{2.15}$$

Since the point P_C is relative to the camera frame, we can expand it using Equation (2.13) to give:

$$\lambda q = RP_W + T \tag{2.16}$$

Here the rotation R and translation T define the extrinsic parameters of the camera that orient it with respect to the world frame. Unlike the ideal pinhole model, the generalized model doesn't assume that the center of projection O coincides with the center of the image plane. It also doesn't assume the measurement scale along the image plane is the same as

the world frame. These differences (and others) can be accounted for by a set of parameters called the intrinsic parameters that can be summarized in a matrix $K_{intrinsic} \in \mathbb{R}^{3x3}$ as shown in Equation (2.17), with Equation (2.16) rewritten as Equation (2.18).

$$K_{intrinsic} = \begin{pmatrix} \frac{f}{e_u} & s & c_u \\ 0 & \frac{f}{e_v} & c_v \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} f_u & s & c_u \\ 0 & f_v & c_v \\ 0 & 0 & 1 \end{pmatrix} \tag{2.17}$$

$$\lambda q = K_{intrinsic}(RP_W + T) \tag{2.18}$$

The parameters shown in the intrinsic matrix as part of Equation (2.17) can be summarized as follows:

- Imaging sensor pixel sizing (e_u and e_v). These are the horizontal and vertical physical sizes of a unit cell of a sensor (dimension of each pixel, typically in μm), allowing different pixel densities along each sensor dimension.
- Physical focal length (f). This is the distance from the lens to the image sensor. It determines the field of view and magnification of resulting images. When the sizing terms e_u and e_v are applied, we end up with 2 different effective focal lengths (f_u and f_v) measured in pixels along with image width and height.
- Skew (s). This value helps adjust when the x & y axes of the image sensor are not perpendicular (some shear angle is present between them).
- Principal points (c_u and c_v). The principal points are the location of intersection for the optical axis of the camera (determined by its lens system) with the imaging sensor. They're important since depending on the camera manufacturing process (and quality control), the optical axis may not be at the center of the imaging sensor.

The components of the intrinsic matrix can be determined through a calibration process [44] [118], where a pattern with known geometry, such as the regular grid pattern of a checkerboard, is captured from the camera at a number of ranges and poses. These variations are then analyzed to determine the intrinsic parameters through an optimization process where the observed corners of the checkerboard pattern are compared to the camera models projections based on the estimation of the intrinsic and extrinsic parameters.

Depending on the camera system, there may also be some degree of distortion present in the recorded image. This distortion can come in several forms. One type of such

distortion, called radial distortion, can be visualized as shown in Figure 2.18 and modelled using a function of distance from the optical center (relative to the principal points) that relates the undistorted points to the initial distorted points as given in Equation (2.19) and Equation (2.20). There are other models for correcting radial distortion, such as the one used by OpenCV, where the principal points are not considered. The use of a particular model depends on application and level of radial distortion involved.

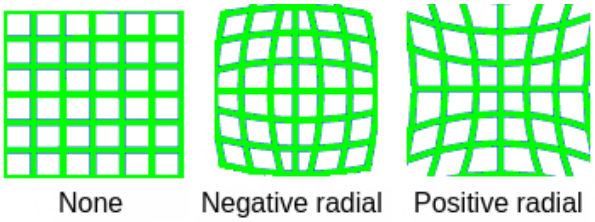

Figure 2.18: Common camera system distortions, where parallel lines have no distortion (left), are curved in a negative radial pattern (center, "barrel" distortion), or are curved in a positive radial pattern (right, "pincushion" distortion)

$$u_{corrected}^{radial} = c_u + (u - c_u)(1 + \kappa_1 r_d^2 + \kappa_2 r_d^4 + e_u) \tag{2.19}$$

$$v_{corrected}^{radial} = c_v + (v - c_v)(1 + \kappa_1 r_d^2 + \kappa_2 r_d^4 + e_v) \tag{2.20}$$

The e_u and e_v terms can be expanded further using a similar geometric progression with κ_3 and r_d^6 if it is expected that they won't be too small (for example with wide-angle lenses). The radial distance term is defined as $r_d = \sqrt{(u - c_u)^2 + (v - c_v)^2}$. The undistortion is applied before calibration of the camera. There is also a second distortion that may occur, which is tangential distortion. This distortion occurs when the imaging plane incurs a tilt relative to the lens system. The corrected coordinates are given by:

$$u_{corrected}^{tangential} = u + 2p_1uv + p_2(r^2 + 2u^2) \tag{2.21}$$

$$v_{corrected}^{tangential} = v + p_1(r^2 + 2v^2) + 2p_2uv \tag{2.22}$$

So we need at least 3 radial coefficients and 2 tangential coefficients for modelling and correcting the most common types of camera distortions. The calculation of these coefficients is typically handled by the camera calibration process [118].

2.4.2 Stereo Calibration

When multiple cameras are present, where images are captured in a temporally synchronized manner such that overlapping parts of a scene will be visible, a stereo calibration allows the views from different cameras to be related mathematically to each other.

The spatial transformations between cameras can be obtained using a stereo calibration algorithm [118] that uses a reference pattern with known geometry (a checkerboard pattern) that is visible from both cameras as shown in Figure 2.19. The corners of the boxes in the checkerboard grid are easily detectable in both the left (left camera) and right (right camera) images. Since the number of corners and their spatial layout relative to each other is known, point correspondences can be established between the images.

A point correspondence is given by Equation (2.23), where p_l is a point $(u_l, v_l, 1)^T$ in the left image and p_r is $(u_r, v_r, 1)^T$ is the same point in space but seen in the right image. The matrix F, a 3x3 rank 2 matrix, is known as the Fundamental Matrix.

$$p_l F p_r = 0 \tag{2.23}$$

This matrix encodes the geometry between the 2 views, however it is ambiguous since it actually does not map point to point but rather point to line. The expression $Fp_r = 0$ is actually a line, and any solution kp_r satisfies the equation $p_l F(kp_r) = 0$. Physically this means a point in 3D, $P \in \mathbb{R}^3$, that projects to the point p_r on the right camera image can correspond to any point along a line that can be seen in the left camera image, and one of the points on this line is our correspondence p_l. Since we already have correspondences this is not an issue, but for any new point selected in the right image (for example after calibration), the corresponding point in the left image must be found by a matching procedure along a line in the left image. We can see an image of the corresponding geometric setup in Figure 2.20, known as the epipolar geometry for the stereo pair of cameras. The point P relative to the respective left and right camera origins (O_l and O_r) can be seen as 3D points P_l and P_r, that respectively map to points p_l and p_r. There is also the line T that connects the 2 optical centers, and intersects both image planes at the epipoles; e_l and e_r respectively for the left and right images. An epipole is the second cameras optical center as seen from the first camera. The line in the left image that e_l and, the true correspondence point, p_l both fall on is called an epipolar line. This line is a view of the line from O_r to P_r as seen from the left image. In practice for a known point p_r in the right image, we don't know its 3D point or its corresponding point p_l. All we know is that in the left image it falls somewhere on the epipolar line, which is what a matching process would look for (more on this in Section 2.4.3).

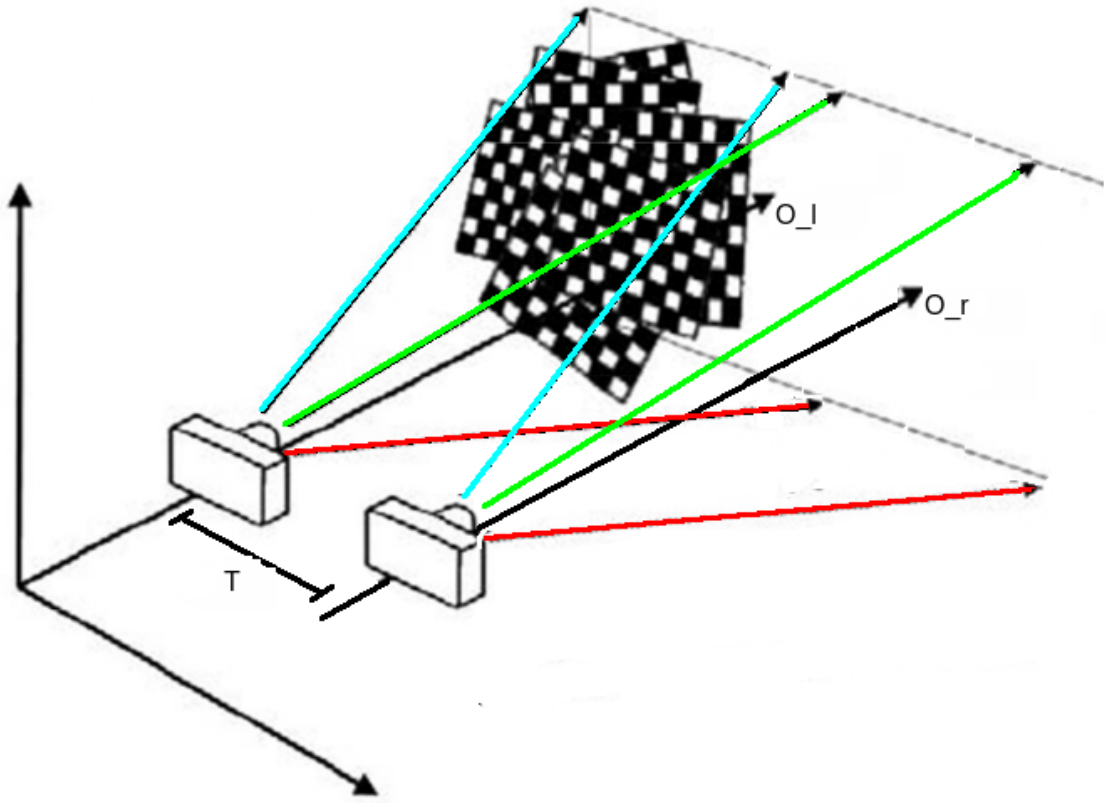

Figure 2.19: A stereo camera pair calibration setup. The left camera with optical center O_l and right camera with optical center O_r are shown a mutually visible reference pattern with known geometry. The camera optical axes (black lines) are aligned but the cameras are horizontally separated by a distance B. The teal, green and red line pairs define mutually parallel lines that make up part of the view frustum of each camera (added for clarity).

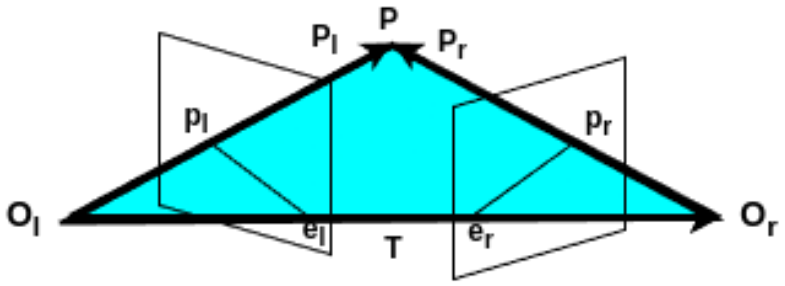

Figure 2.20: The epipolar geometry for a stereo pair of cameras. A point P in $\mathbb{R}^3$ is visible by both cameras (as 3D points P_l and P_r respectively by the left and right cameras). This point projects to respective points p_l and p_r. A line T (the baseline) connects the optical centers of the left and right cameras, denoted by O_l and O_r respectively.

When only point correspondences are known, the Fundamental Matrix allows us to encode the epipolar geometry between two cameras in a stereo configuration. However, it may be the case that additional information is known about the cameras, such as the intrinsic parameters, which would allow us to compute the Essential Matrix (E). This matrix is a more direct relationship for the rotation and translation between the two camera views.

2.4.3 Stereo Rectification

The stereo matching alluded to in Section 2.4.2 is usually not done directly on the raw image frames. If both cameras (and therefore image planes) were placed side by side (this configuration is called "frontal parallel") as shown in Figure 2.19, the matching could be much simpler. In that case, epipolar lines would be horizontal and we could move horizontally to the adjacent pixel from a starting pixel in the left image to find the best correspondence for a pixel in the right image; a drastic simplification from the general case. To allow for such a configuration (without necessarily even requiring physical alignment), the problem is usually simplified through a process called rectification. This is a process that mathematically aligns the original camera views of the scene so that they become frontal parallel. This is possible because moving the physical cameras means physical rotations and translations that we can estimate to achieve the same affect, producing the setup shown in Figure 2.21.

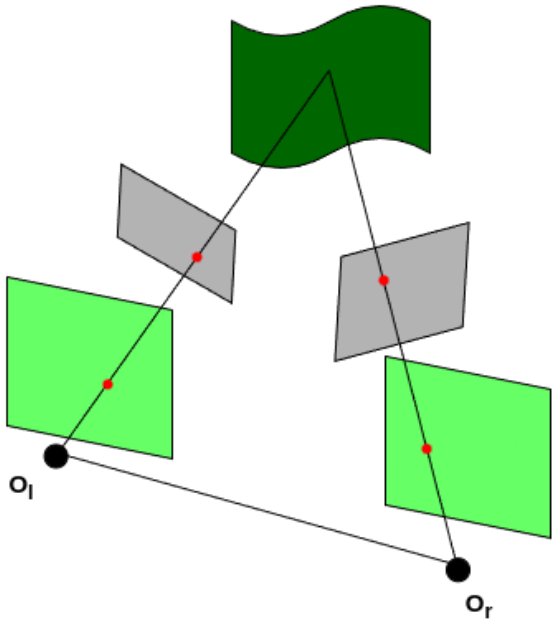

Figure 2.21: Stereo rectification process. The true camera image planes (grey rectangles) are adjusted to be frontal parallel (yellow rectangles). The center of projection for the left and right cameras (O_l and O_r respectively) are also shown.

Depending on the circumstances, this process can be helped by the experimenter. For example, if the placement of cameras can be adjusted, they can be arranged so that they're physically as close to frontal parallel as possible. The rectification process also typically does not need to be re-implemented by hand (except possibly in more specialized cases such as unique calibration objects or special camera configurations). Instead standard implementations of popular rectification algorithms [72] can be used either through the Matlab Computer Vision toolbox [73], OpenCV [9] or other libraries (depending on the programming language used). The same applies for stereo calibration [118]. The result of a rectification can be visualized as in Figure 2.22.

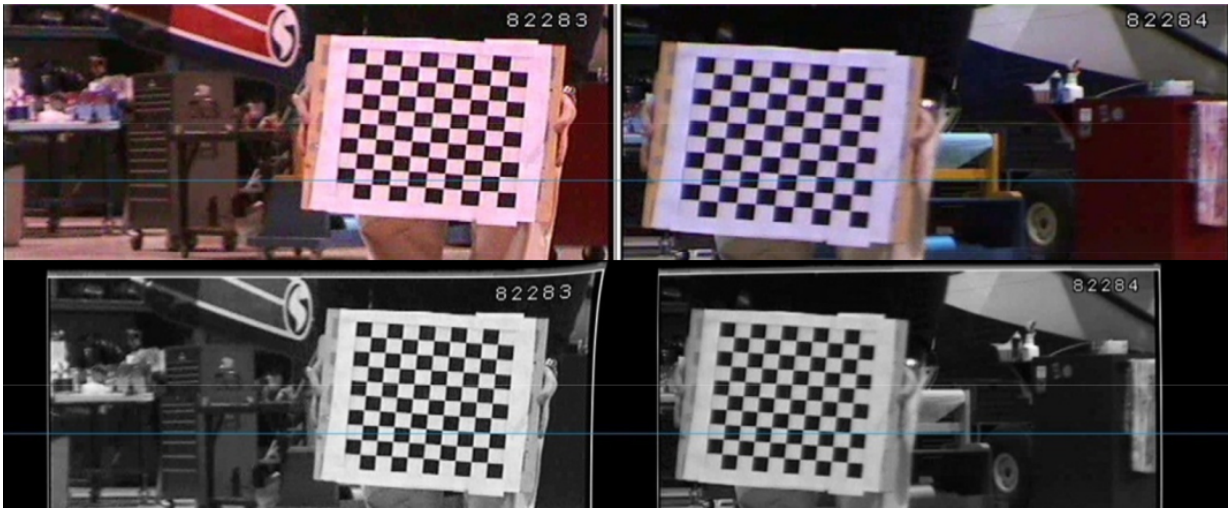

Figure 2.22: Visualization of a rectification from a stereo pair of images. The top image pair represents respectively the left and right image pair. The bottom pair is the rectified result. In the top pair, the blue line clearly intersects completely different points on the checkerboard. For the bottom pair, the same feature points are intersected between the 2 images along the blue line (the top left of the 3rd square from the bottom right being the most discernible). [107]

2.5 Datasets

Modern research into ConvNet models has benefited greatly from the gradual accumulation of more and more datasets targeting various tasks such as classification (ImageNet [22]), as well 2D & 3D detection (KITTI [37], NuScenes [10], Waymo [102]) and segmentation (MS-COCO [68]). The multiplicity of datasets has grown with their complexity as well as the availability of sensors with which to capture them and the compute hardware on which to train models. Arguably the compute hardware has grown the slowest relative to these other measures, so that without a computer having at least 3-4x high end GPUs, it can take weeks (not days) to train the most performant models. Even the PointPillars model investigated in this thesis takes 3-4 days to fully converge on NuScenes. This limitation is why the focus of this thesis is restricted to smaller datasets (Caltech-USA [26] and KITTI). While a limitation it is still informative since we know smaller datasets can be used to pretrain large models for better large dataset training results while exploration on a smaller dataset can still be helpful if a contribution is provided that adds to the body of annotated data (for example the KITTI dataset did not have segmentation information

for annotated objects). The remainder of this section outlines the datasets used in the chapters in this thesis.

2.5.1 CaltechUSA

The Caltech-USA dataset [26] is a pedestrian dataset with 250k annotated frames containing 350k 2D bounding boxes of 2300 unique pedestrians. The dataset is broken down into short sequences of 640x480 pixel frames sampled at 30Hz from a vehicle driving in an urban environment (Los Angeles metropolitan area in areas known to contain lots of pedestrians). This means not every frame of each sequence has pedestrians; approximately 50% of frames have none while 30% have two or more. Sequences are not shared between the training and test sets. Pedestrians are labelled with full 2D boxes, so that occluded areas are part of the bounding box. The test set is not released to the public, requiring a submission of an algorithms predictions to the authors in order to know its performance on the test set.

2.5.2 KITTI

The KITTI dataset [37] is a widely known and extensively studied computer vision dataset. It consists of left and right camera RGB images (for stereo), sensor calibration, Inertial Measurement Unit (IMU) data, tracklets and LiDAR point clouds in 151 video sequences taken in city, road and campus driving environments during daytime summer. The videos in this dataset have been sampled over time to construct specific benchmarks with which researchers have developed new advances in object detection, segmentation, optical flow and other research areas. The 3D object detection benchmark specifically, uses 7481 training and 7518 test images & point clouds sampled from 141 of these sequences (sequences are mutually exclusive between training and test frames). The result for the overall 3D detection dataset (train and test) are 80256 labelled objects across 9 classes. The standard practice is to use a specific split of the training data creating 2 subsets 'train' (3712 samples) and 'val' (3769 samples), where 'val' is used to validate the model and to allow competing methods a point of comparison. This is necessary because the 3D object detection benchmark is closed to submissions without an accompanying publication (the test data is not released to the public). Hence the evaluations performed in this thesis are all based on the KITTI 'val' subset and compared where possible to other methods evaluated on the 'val' subset.

The training data, in addition to its class and 3D box, also has a per sample categorization of 'difficulty'; the categories are easy, moderate and hard (referred to respectively as 'easy', 'mod' and 'hard' in performance tables throughout the thesis). These 3 criteria, taken together, are how the difficulty of an example is assigned to an object. The criteria are assigned as they're observed in the camera frame in the 2D image of the scene (not the LiDAR). The criteria are summarized below, including Table 2.2.

- **Object 2D height**: The minimum height of a bounding box in pixels.
- **Object occlusion**: The max occlusion level as assigned to the object. Occlusion levels are already part of the annotations for each object, and can be one of 3 integral categories: no occlusion, partial or difficult/heavy occlusion. Occlusions occur due to some other objects in the scene that are in the foreground of the object of interest (and therefore block some amount of the view of the object of interest)
- **Object truncation**: The max level of truncation (light, moderate and heavy) which is the amount of the object that is cut off by the FOV boundary

Criteria \ Difficulty	Easy	Moderate	Hard
Min 2D Height	40	25	25
Max Occlusion	None	Partial	Difficult
Max Truncation	15 %	30 %	50 %

Table 2.2: KITTI object difficulties and criteria for assigning objects to each difficulty

The coordinate frames for the KITTI dataset are shown in Figure 2.23 for both the Camera and LiDAR perspectives as well as the defined directions for each physical box dimension. So a bounding box for a Car, without any rotation applied in the camera frame, would have its length dimension along the X axis and its width along the Z axis; essentially the car would be facing to the right. The camera frame is the coordinate frame for all annotations, including the position of 3D bounding boxes and their dimensions. The 3D box centers are defined differently depending on the coordinate frame, with 3D box positions in the LiDAR frame defined relative to the box center while the position in the camera frame is defined relative to the center of the bottom face (XZ plane) of the 3D box. This is demonstrated in Figure 2.23.

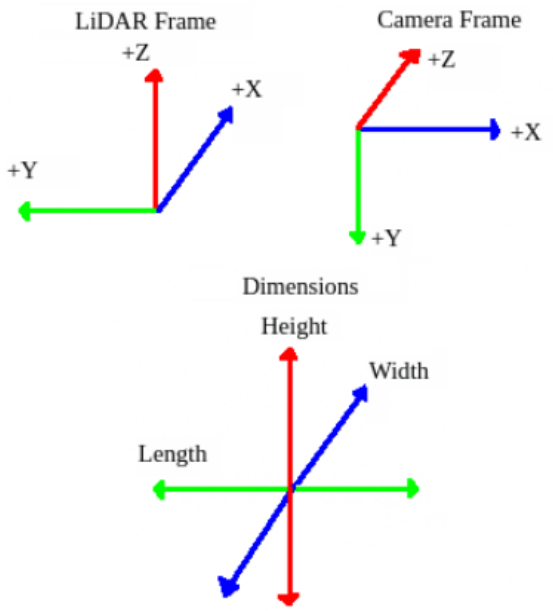

Figure 2.23: Overview of the KITTI Coordinate Frames and the Corresponding Dimension Orientations

Two angles annotate each ground truth object; a yaw angle θ_{yaw}, range of [-π, π], which is the rotation angle about the camera frames Y-axis and an alpha angle α, range of [-π, π], which is an object observation angle. Counter-clockwise rotation is considered positive in the camera frame as viewed from the +Y axis (from below the XZ plane). To understand the alpha/observation angle, it is necessary to understand the bearing angle, $\theta_{bearing}$, which is the angle between two lines emanating from an observer. The two lines are a reference direction (forward or +Z in the camera frame) and a line L extending to a point of interest (in this case an object center). In the context of the right side axes in Figure 2.24, a line L clockwise of +Z has a positive bearing angle and negative counterclockwise from +Z.

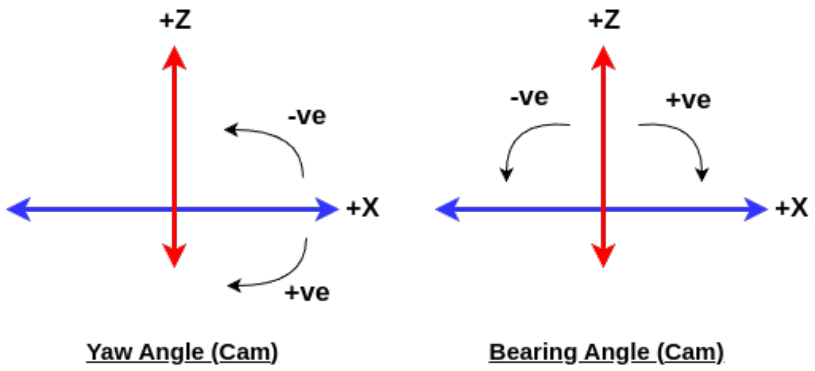

Figure 2.24: BEV view of KITTI rotation directions and zero angle points for yaw and bearing

The angle alpha (α) can then be defined as:

$$\alpha = \theta_{yaw} - \theta_{bearing} \tag{2.24}$$

As yaw varies, a corresponding change in bearing keeps the alpha angle fixed such that a single alpha value represents a particular, consistent view of an object as shown in Figure 2.25. As alpha varies between $[-\pi, \pi]$, a sweep of view directions around an object is performed as shown in Figure 2.26.

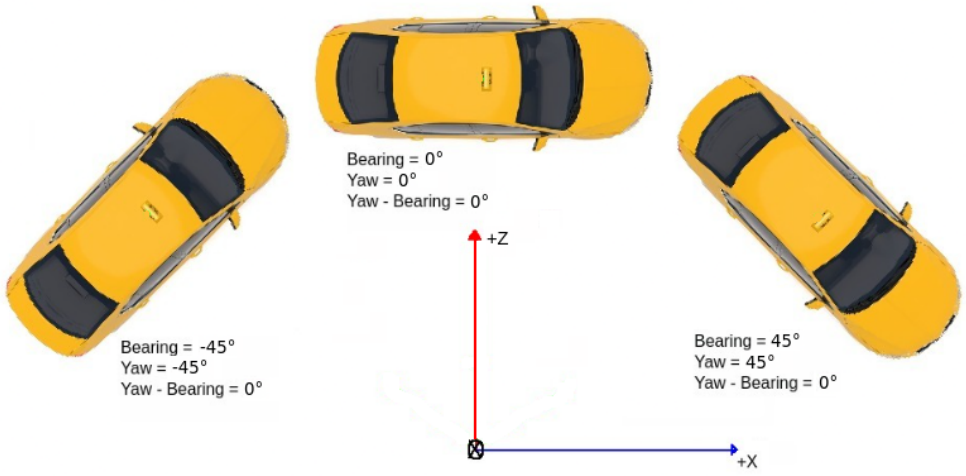

Figure 2.25: Constant alpha angle with varying yaw and bearing in a BEV of the KITTI camera frame

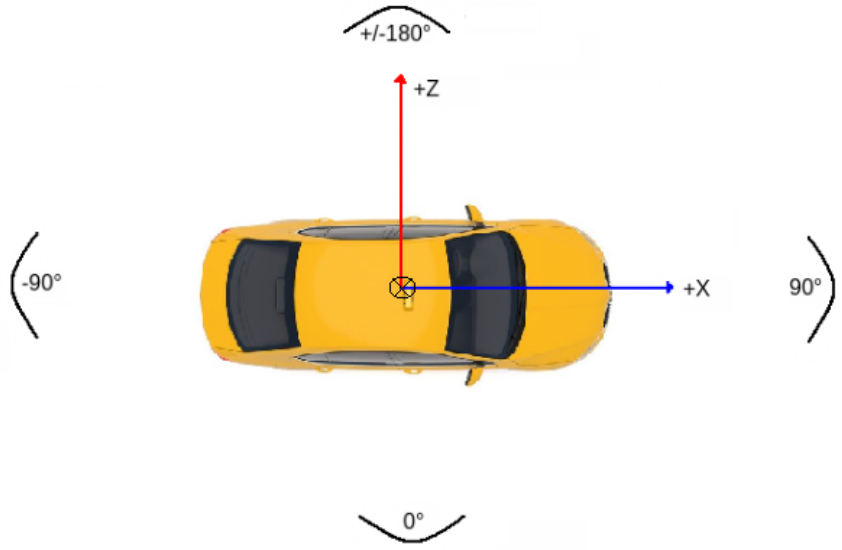

Figure 2.26: Range of alpha values around an object

Every scene/frame in the KITTI dataset comes with a calibration file (text document) consisting of data structures that describe the spatial relationship between the LIDAR, cameras and the vehicle before and during data acquisition [36]. The LiDAR and 4 cameras

(two grayscale and 2 color cameras) were mounted to the roof of the vehicle (a station wagon) as shown in Figure 2.27. The data structures in the calibration file describe the spatial transformations between each sensor, allowing content captured from one sensor to be related to another. There was also another unit (not shown), the GPS/IMU unit, inside the vehicle recording GPS coordinates, orientation, velocities, acceleration and angular rates.

Figure 2.27: Birds eye/top down view of the mounting of sensors to the KITTI data acquisition vehicle. Sensors are color coded (Red: LiDAR, Green: Grayscale camera, Purple: Color camera)

Since there are 4 cameras, the authors provide 4 projection matrices P^i_{rect} for i=0,1,2,3 where the indices correspond to the following cameras:

- Index 0: Leftmost grayscale camera
- Index 1: Rightmost grayscale camera
- Index 2: Leftmost color camera
- Index 3: Rightmost color camera

The expression for the projection matrices is given by the authors as Equation (2.25), for 3D point $\mathbf{x} = (x, y, z, 1)^T$, image point $\mathbf{y} = (u, v, 1)^T$ and $\mathbf{y} = \mathbf{P}^i_{rect}\mathbf{x}$:

$$\mathbf{P}^i_{rect} = \begin{pmatrix} f^i_u & 0 & c^i_u & -f^i_u b^i_x \\ 0 & f^i_v & c^i_v & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{2.25}$$

We can see the upper left 2x3 entries encode the intrinsic parameters of the i'th camera (the focal lengths and principal points). The extra parameter b^i_x is the baseline distance

along the +X axis (right/left) in the camera frame. In practice the other projection matrices in the calibration files have extra, non-zero entries in positions (1,3) and (1,4) (assuming a 0-based row,column indexing). The term at position (0,3) is a "baseline times focal length" term. We can interpret the value at position (1,3) as being the same "baseline times focal length" term but in the +Y direction (down/up). The computed baseline for this direction is quite small in the KITTI calibration data, (for example, 0.000488m for $\mathbf{P}^1_{rect}$) indicating a very small alignment difference in this direction. Likewise, the (1,4) entry is of a similar order. The coordinate $\mathbf{x}$ is assumed to be in rectified camera coordinates. A single rectification matrix, R^0_{rect} is provided with off diagonal entries of the order 10^{-3}, and diagonal entries very close to 1. This implies a rotation matrix close to an identity rotation meaning the physical placement of the cameras is very closely aligned relative to each other. Finally, a 3x4 matrix is provided for encoding the rotation ($\mathbf{R}^{cam}_{velo} \in \mathbb{R}^{3x3}$) and translation ($\mathbf{t}^{cam}_{velo} \in \mathbb{R}^{1x3}$) for the transformation from LiDAR to camera coordinates:

$$\mathbf{T}^{cam}_{velo} = \begin{pmatrix} \mathbf{R}^{cam}_{velo} & \mathbf{t}^{cam}_{velo} \\ 0 & 1 \end{pmatrix} \tag{2.26}$$

The 3D point $\mathbf{x}$ in LiDAR coordinates can then be projected to the i'th camera using the following transformation:

$$\mathbf{y} = \mathbf{P}^i_{rect}\mathbf{R}^0_{rect}\mathbf{T}^{cam}_{velo}\mathbf{x} \tag{2.27}$$

The KITTI LiDAR data consists of 3D points in the LiDAR coordinate frame along with a 4th value per point representing reflectance. This reflectance value is a real value in the range [0,1] and is a measure of the amount of scanning light reflected back to the sensor from the corresponding 3D point in the scene. This value is typically normalized based on the initial power of scanning light emitted by the LiDAR sensor. Higher reflectance values represent a higher power level of the returned scanning light. The recorded reflectance will be influenced both by environmental factors (dust, moisture) and the characteristics of the object on which the point is located. For object characteristics, these involve factors such as surface material (metal materials reflect differently than dark surfaces or vegetation), surface incidence angle (perpendicular incidence results in higher reflectance) and distance (farther distance means more divergence of each LiDAR beam resulting in more energy loss). In the case of the KITTI dataset, the authors use a Velodyne HDL-64E sensor with 64 scan lines. The stated reflectance of this sensor by its manufacturer is 50m for pavement (approximately 0.1 reflectivity) and 120m for cars and foliage (approximately 0.8 reflectivity). In practice, the distance of 120m is extremely optimistic. In practice, the original PointPillars model and our variants don't exceed a 70m distance with performance

of the underlying model degrading well before this distance.

2.6 Hardware

As discussed in Section 2.2.1, ConvNet models that show high performance tend to be computationally complex on the order of 10s to 100s of GFLOPs and millions of model parameters. The training of these models is enabled by powerful GPUs where the training time can be days or weeks, and may be acceptable depending on the task. However, inference times for such models will also be high and use of these models on severely resource constrained hardware will be difficult or impossible without making compromises to the model performance. This optimization is the focus of this thesis and the rest of this section explores the hardware used in this thesis to support this investigation.

2.6.1 S32V234 Automotive Processor

The S32V234 is a high performance, ultra low power, automotive grade processor targeted for industrial applications where it particularly excels in applications for vision and sensor fusion. It includes Quad ARM Cortex-A53 cores @ 1GHz, 4MB of on chip system RAM, an integrated 3D GPU (not used for our work) and Dual APEX2 image cognition (specialized image processing units for handling vision and image data). The APEX cores were the processors used for the work in Chapter 3. An API is available to interface to the capabilities of the platform. It is part of the more general S32V family of processors that target ADAS. It operates within about a 2W power envelope.

2.6.2 Jetson AGX Xavier

The Jetson AGX Xavier (2018) is an "edge AI" compute module released by NVIDIA [79]. This basically means its a specific version of NVIDIA's Jetson line of embedded compute boards; a compute board is a specialized integrated circuit designed for a specific function. The compute board is a Tegra System on a Chip (SoC), that packages the CPU, GPU and some other components on a single chip. This SoC forms the core of the AGX Xavier, and is integrated on a circuit board called a development kit which houses the RAM, eMMC storage, power distribution and IO (PCIe, USB, SATA, Ethernet and others). The GPU has a specific Compute Capability designation, just like discrete NVIDIA GPUs

Component	Specification
CPU	8-core NVIDIA Carmel ARmv8.2 @ 2.26GHz
GPU	512-core NVIDIA Volta with 64 Tensor Cores
Memory	32 GB 256-Bit LPDDR4x, 137.7 GB/s
Storage	32 GB eMMC 5.1

Table 2.3: Summary of Components in the Jetson AGX Xavier

(specifically Volta 7.2 for the AGX Xavier), that describes its specific CUDA features. The components are summarized in Table 2.3 [80].

The SoC can be placed in a variety of power modes that adjust the number of active cores as well as the peak frequencies of the RAM and SoC components like the CPU & GPU. This is summarized in Table 2.4 [80].

Power Mode	Power Budget (Watts)	Online CPUs	Max CPU / GPU / RAM Frequency (GHz)
MAX-N (0)	30	8	2.265 / 1.377 / 2.133
30W-ALL (3)	30	8	1.2 / 0.9 / 1.6
15W (2)	15	4	1.2 / 0.67 / 1.33

Table 2.4: Power Modes of Xavier AGX

On the software side, the board itself runs NVIDIA's Linux for Tegra (L4T) Operating System (OS), which is essentially a fork of the Ubuntu operating system, but adapted to run on NVIDIA Tegra processors. Each L4T version is packaged with a specific version of an SDK called JetPack, that includes the NVIDIA (CUDA, cuDNN) and third party (OpenCV) software libraries needed to effectively use the system.

Chapter 3

SqueezeMap: Fast 2D Pedestrian Detection

3.1 Introduction

In the era of autonomous vehicles and smart assist vehicle computers, a robust and above all safe driving system requires a model which can create an accurate representation of the environment. Such systems rely on sensors for its input, which can include laser, radar or camera based solutions. Camera solutions offer a cheap and simple input source for such systems.

For such camera dependent systems, the data needed to train them is readily available, foregoing the necessity of manual collection. The abundance of data is helpful, since the state of the art models for object detection and classification are generally deep learning models which have a large number of tunable parameters. A subset of these deep models, ConvNets, have been especially successful for a wide range of image processing tasks, including segmentation, classification and detection. However, the majority of research in this area has focused on improving outright performance and exploring the full design space of ConvNet architectures while placing less emphasis on model size.

For an embedded system in an autonomous vehicle, this emphasis is paramount, because models not only have to be small in terms of runtime footprint, and therefore a small footprint in power consumption, they are also small in total model parameters as changes to the model need to be pushed over network links with limited bandwidth. The decrease in footprint can help to reduce the runtime computational requirements, as any pedestrian

detection system needs to ensure a processing speed sufficient for performing avoidance as well. With a decrease in model size also comes the tricky task of optimization to maintain the same level of performance relative to existing, unconstrained models. This is complicated by the fact that autonomous vehicles systems need to maintain high levels of recall and precision in order to be safely deployed.

In this chapter we focus on approximating pedestrian locations using a coarse, grid--based approach. We do this by adding an additional layer to an existing SqueezeNet network. This layer performs a kind of weighted voting scheme across the depth dimension of the previous (convolutional) layer to determine whether a pedestrian is present. Our approach introduces only a small number of new parameters, with a tiny final model size of 3.24MB. This model can run at 72FPS on a GTX Titan X GPU on an RGB input of 681x227 and at 30FPS on dual APEX2 low-power processors [81] with the entire system running within a 2W power envelope. We estimate the accuracy of the approach relative to other models trained on the CaltechUSA dataset for a range of model and evaluation settings and show it has comparable performance in terms of log-average miss rate.

3.2 Related Work

Pedestrian detection is a special case of more general object detection. Investigation in this area has produced a wide range of solutions and a number of benchmark datasets, including CaltechUSA, KITTI [37] and ETH [28] among others. CaltechUISA has a reasonably large set of images and a large number of solutions have been evaluated on it. These solutions include variants of Viola & Jones [78], HOG [20], deformable part models (DPM) [31] and various types of neural networks including ConvNets [6]. While the non ConvNet variants have steadily improved over time, it is the ConvNet-based solutions we are interested in due to their repeating, general structure and operations which can be trained easily end-to-end and consistently achieve state-of-the-art performance.

A number of variants for this task which use a ConvNet as a starting point have been proposed, including Region-based ConvNets (or RCNNs) by Girshick et al. [39] and FCNs, which focus on semantic segmentation, popularized by Long et al. [71] (including variants such as R-FCNs which focus on object detection). The RCNN strategy is to identify regions of interest as a starting point for applying a ConvNet. Faster and more computationally efficient variants are available, such as FastRCNN [40] and FasterRCNN [91], which share the computation time for proposed regions. Overall, these approaches can be slow without powerful GPUs. To address the speed shortcoming, newer approaches such and YOLO (You Only Look Once) [88] have been proposed which combine classification and region

proposals into a single stage while estimating bounding boxes. While these approaches are impressive, they are extremely difficult to run within our target platform constraints. This necessitates a re-think of the type of detections we should consider.

While region proposal based approaches have shown good performance on CaltechUSA, to save computations, we can avoid having to estimate the exact pixelwise locations of bounding box corners and instead have a cluster of activation groups to indicate the approximate extent of a pedestrian (while using a bare minimum of parameters to do so). Thus instead of estimation of bounding box corners or single pixel labeling, we focus on a coarser grid-based level where a gridbox in the grid being "on" indicates a pedestrian feature is present within that gridbox.

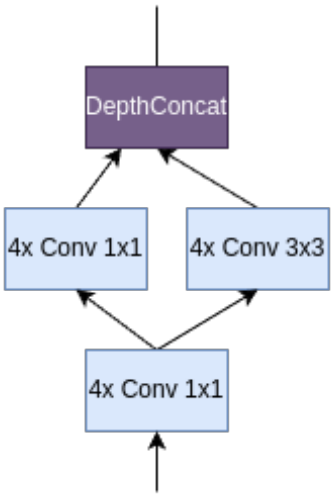

Figure 3.1: A fire module used to construct a SqueezeNet network. The squeeze layer is meant to show 4 FMs which each use 1x1 convolutional filters. The expand layer has 4 FMs which use 3x3 filters and 4 which use 1x1 filters

3.3 Method Description

3.3.1 Network Structure

SqueezeNet is a small ConvNet architecture (building block shown in Figure 3.1) that was designed to emphasize small model size while retaining classification accuracy relative

to popular ConvNet architectures like AlexNet [53]. It is able to do this by following 3 strategies which have either been shown experimentally to work well or work to prevent an explosion of parameters:

1. Placing an emphasis on 1x1 convolutions
2. Restricting the number of input channels to filters larger than 1x1
3. Delayed down-sampling throughout the network

The first 2 strategies are mainly aimed at parameter reduction. By keeping the number of feature maps in squeeze layers small, and with only 1x1 convolutions, strategy 1 and 2 are achieved by saving parameters and by supplying only a small number of channels to the 3x3 "expand" feature maps. Thus as we propagate through consecutive fire modules we're repeatedly squeezing the previous modules (expand layer, fireN) feature map output through a small number of feature maps (squeeze layer, fireN+1), followed by a larger number of feature maps (expand layer, fireN+1). Strategy 3 is based on observations made in prior experimental results [48], which attempts to keep feature maps relatively large (to the input) in an attempt to learn better features. The intuition is that this should give higher accuracy (albeit at higher computational cost and runtime memory footprint). Since fire modules don't decrease the size of their input, repeated propagations through multiple fire modules mean the input is not down-sampled, in the process helping to fulfill strategy 3.

Although SqueezeNet is amenable to compression techniques (pruning, deep compression [43]), we do not investigate this option due to the already small footprint of the network and the overhead introduced in some of these approaches. A summary of the network parameters is shown in Table 3.1, inspired by a similar table in the original SqueezeNet publication. Note the feature map dimensions are slightly different in our TensorFlow implementation than in the reference Caffe model. We use SqueezeNet 1.1 as the base model, pre-trained on the ImageNet 2012 dataset [22]. This model is an extension of v1.0, achieving almost identical performance on ImageNet while reducing the number of computations by 2.4x through a small re-organization of the original model. A summary of the required MAC operations at each layer is shown in Table 3.1.

We remove the conv10 layer entirely, since it was found this does not impact the performance of model to the target task. In the process we save 500k parameters, leaving 722.5k. After the addition of our partially connected layer we have just under 810k parameters or a model about 3.237MB in size. The addition of batch normalization (BN)

Layer Name	Layer Size (HxWxC)	Filter Size / Stride	s1x1	e1x1	e3x3	Num. Params	MACs
Input	227x227x3						/
conv1	113x113x64	3x3/2				1792	22064832
pool1	56x56x64	3x3/2					/
fire2	56x56x128		16	64	64	11408	35323904
fire3	56x56x128		16	64	64	12432	38535168
pool3	27x27x128	3x3/2					/
fire4	27x27x128		32	128	128	45344	32845824
fire5	27x27x256		32	128	128	49440	35831808
pool5	13x13x256	3x3/2					/
fire6	13x13x384		48	192	192	104880	17651712
fire7	13x13x384		48	192	192	111024	18690048
fire8	13x13x512		64	256	256	188992	31842304
fire9	13x13x512		64	256	256	197184	33226752
hm0	13x13x512	1x1x512				86697	86528
			Total Parameters			809193	
			Total Size (MB)			3.237	
			Total MACs				266.1M

Table 3.1: Summary of network parameters

after fire9 introduces 2 parameters for each output in fire9, resulting in 810,217 parameters (a final size of 3.24MB). The choice of base model is open and is not mandated by our approach. The only other change we investigate to the base SqueezeNet model is swapping the ReLU units after each convolutional layer to exponential ReLU units, which decay exponentially when the argument is less than 0, but found no noticeable effect to ReLU's.

Our primary goal is similar to SqueezeNet, which is to maintain a small number of parameters, while obtaining a coarse estimate of pedestrian location and size. A standard way to do this would be to introduce a fully connected layer that has the same width and height as fire9, allowing an estimate of the presence/absence of a pedestrian at each pixel of the FM. However fully connected layers, while much less computationally burdensome than convolutional layers, are generally wasteful in terms of storage. State of the art networks which have large fully connected layers can usually be pruned of about 90% or more of their parameters [43]. In addition, unless the scale of the pedestrian is large enough to fill a large portion of the fire9 activations (the pedestrian is too close), our intuition is to keep the number of connections across the width and height of fire9 small and instead

focus on the depth dimension in fire9. The output of the partially connected layer can (but doesn't need to) match the height and width of the convolutional layer (given by H and W respectively) it is connected to, connecting only across the depth dimension (D). For a convolutional layer, each activation A_{hwd} can be indexed by (h,w,d) where each is in a range from 0 to H, W, D respectively.

For an output in the heatmap layer, (h,w), its input is computed as:

$$\sum_{d=0}^{D-1} w_{hwd} * A_{hwd} + b_{hw} \tag{3.1}$$

The number of parameters is reduced from H*W*(H*W*D+1) to H*W*(D+1). That means only 86,528 additional parameters for this layer in our case.

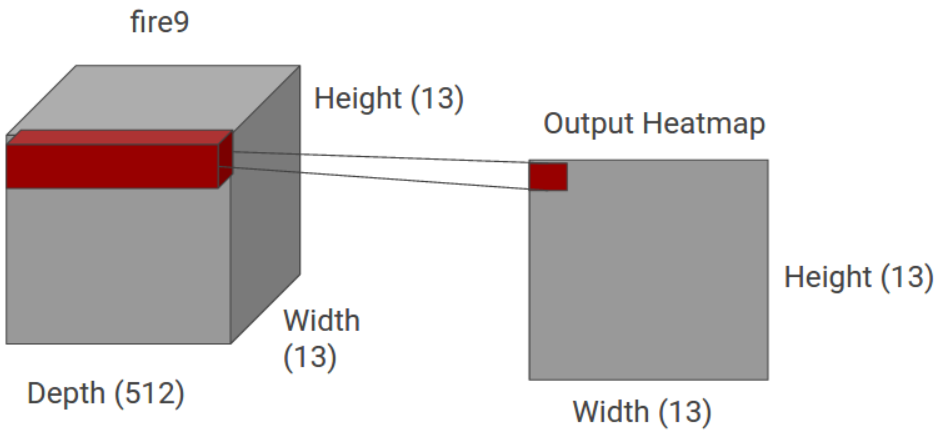

Figure 3.2: Connectivity along the depth dimension between fire9 and the output layer

This scheme is essentially performing a weighted voting along the depth dimension, for every pixel in the partially connected layer (Figure 3.2). We could ask why we would want to have different weight sets for voting at each output pixel instead of applying the same set of weights (essentially a kind of depth convolution). Taking this further, majority pooling could be performed which will activate an output if a majority of the inputs are "on" (using some fixed threshold) saving even more weights. The benefit of having only one set of weights is tested by experiment but we leave examining more complex pooling operations (such as rank-pooling) for future work.

We add dropout between the fire9 and heatmap layers, although we investigate its placement at other locations within the SN network, and experiment with various dropout rates <= 50%. The effect of BN after fire9 is also investigated, including placement before or after the dropout layer. At the output layer, we use tanh as our activation function although we did experiment with exponential linear units (ELU's) at the output layer but were not able to obtain consistent or comparable performance. Preliminary experiments using the output of layers upstream of fire9, followed by a heatmap layer, have shown a decrease in performance although the extent has not been fully investigated. For example, it may be possible to further trim the network depth while remaining close to the reference performance level when using the fire9 layer.

3.3.2 Data Generation

Figure 3.3: A grid overlaid on an image patch with ground truth bounding boxes shown (left) and the resulting target gridboxes after using a 50% threshold (original image from [26])

For generating the training dataset we overlay a set of 227x227 pixel patches over 640x480 pixel images. Each overlay has a 13x13 box grid embedded inside. To make the 13x13 box grid fit neatly we discard 3 columns of pixels on the left and right sides of the image (resulting in 17x17 pixel gridboxes). A ground truth bounding box, when intersected with the grid in a patch will intersect with a set of gridboxes, S. We leave the intersection percentage between the gridbox and the ground truth bounding box as a tunable parameter

when generating our dataset. We experiment with values in the range of 35% to 75%. An example of an overlaid grid with ground truth bounding boxes is shown in Figure 3.3. The adjacent image shows the resulting gridboxes which are deemed to be active after applying a 50% threshold. Some bounding boxes, due to an insufficient overlap with gridboxes, do not produce activations in Figure 3.3. The adjustment of this overlap is discussed as part of the evaluation process in Section 3.3.4.

We experiment with various ways to overlay the 227x227 pixel input over the original 640x480 pixel images. The most important region of the input images is just above ($\sim$200 pixels from the top of the image) the horizontal centerline of the image as described in [24]. We focus on the following sampling regimes for our training data:

- Sample 3 227x227 pixel regions along the horizontal centerline.
- Sample 6 227x227 pixel regions, 3 along the centerline and 3 samples offset upwards by $\frac{1}{4}$ of the sampling region height

We allow one column of gridboxes to overlap when transitioning horizontally between one grid to the next. An example of sampling for Case 1 is shown in Figure 3.4, with the extent of the 3 regions overlaid (red, green and blue). When generating the evaluation dataset, we follow the process detailed in [24], of generating samples every 30 frames (starting with the 30th). We investigate a number of sampling schemes for building the evaluation dataset and report performance for all:

- Sample 3 227x227 pixel regions along the horizontal centerline (sampling scheme named "3pos")
- Sample 6 227x227 pixel regions, 3 horizontally spaced starting at the top left of the image and 3 spaced similarly starting at (227,0) (named "6hilo")
- Sample 9 227x227 pixel regions, 3 along the horizontal centerline, 3 offset upwards to (0,0) which places 3 sampling regions along the top of the image and 3 offset downwards to (411,0) which places the last 3 regions along the bottom of the image (named "9pos")

Figure 3.4: Example of sampled regions for "3pos" (image from [26])

The first scheme focuses on our main area of interest along the horizontal centerline. The second scheme attempts to cover as much of the image as possible starting along the top and is closest to achieving unique coverage over the original 640x480 pixel image. The third scheme attempts to maintain the horizontal centerline while also covering as much of the image as possible.

3.3.3 Training Procedure

We use an L2 loss function to train our network. Although we experiment with L2 regularization, we found that skipping it generally produces better performing networks. During training, we apply exponential averaging to the network weights and to the computed loss, using rates of 0.9999 and 0.9 respectively. We experimented with a number of optimizers, including vanilla gradient descent, Adam, Adagrad and RMSProp. Overall, we found RM-

SProp to work best (in terms of resulting performance and also for speed of convergence) and most consistently.

3.3.4 Evaluation

The evaluation scheme for the Caltech Pedestrians dataset is extensive but focuses on bounding boxes as the unit of measure for the extent of pedestrians. Although our approach specifically foregos bounding box estimation, we still want to estimate our performance relative to other approaches applied to this dataset.

To do this we use the ground truth bounding boxes for an input image and, with the output from the heatmap layer, perform a voting procedure to estimate the bounding box predictions. This process allows us to estimate the true positive (TP), false negative (FN) and false positive (FP) rates for pedestrian detection, which then allows us to estimate the Log Average Miss Rate (LAMR) of our method. The log average miss rate is a measure of miss rate (false negative rate or "1 - recall") versus the false positives per image (FPPI) over a range of FPPI values (evenly spaced in the log domain between 0.01 and 1). This scale is meant to emphasize small rates of false positives and the overall metric is meant to be used in cases where FPs have an upper limit regardless of how many objects are present in the scene [24]. To obtain a set of miss rate values, we apply a range of thresholds over the range [-1, 1] to our output layer.

By introducing a number of parameters to control this estimation, we can obtain a kind of performance envelope with the 2 extremes corresponding to a difficult evaluation and an easy one. These parameters include:

- P_{OBB}: The percentage overlap between a gridbox and a ground truth bounding box
- P_{ACT}: Percentage of gridboxes belonging to a ground truth bounding box which must be on to consider the bounding box detected
- B_{INDIV}: Whether to count activations not within ground truth bounding boxes individually or to use 8-way connected components
- B_{OCC}: Whether to use the full bounding boxes or the unoccluded portion of each for the ground truth
- B_{IG}: Whether to use "ignore" ground truth bounding boxes which fulfill certain criteria, including a minimum height of 20px for ground truth bounding boxes, filtering "people" and "person?" annotations and ignoring bounding boxes truncated by image boundaries.

Ideally, the most difficult parameter setting should yield performance comparable to the best performers on the Caltech benchmark. The evaluation process proceeds as follows (illustrated with concrete dimensions for clarity). For each desired activation threshold at the output layer, $T \in [-1, 1]$, and the next 640x480 pixel input image, I:

- Extract the desired set of 227x227 grid overlays from I, as in Figure 3.4, to get a set of overlay regions, R.
- For each overlay region, R_k, remap the ground truth bounding boxes in I relative to each overlay region. This produces a set of bounding boxes, $GTBB_k$ for each overlay region.
- Intersect the 13x13 grid in each overlay region with the remapped ground truth bounding boxes. Keep the gridboxes whose percent overlap with each ground truth bounding box exceeds P_{OBB}. These are the ground truth target gridboxes, $GTTG_{k,n}$, and each overlay, k, will have some number of sets of resulting target gridboxes (one set for each bounding box in $GTBB_k$) whose number we index as a single "n" here for simplicity.
- If B_{IG} is "on", the bounding boxes which don't satisfy the mentioned criteria and are meant to be ignored have their target gridboxes computed in a similar manner and are aggregated into a set, GTIG. Otherwise we leave GTIG empty.
- For each overlay region, k, apply the current threshold T to our network output to produce a set of gridboxes of where pedestrians are predicted to be. Call this set, ACT_k.
- For each overlay, k, iterate each set of target gridboxes in $GTTG_{k,n}$ and compute the set intersection with ACT_k. This is computing which gridboxes for each ground truth bounding box are predicted "on" by our network. If the percentage is greater then P_{ACT} we count this as a TP. Otherwise it is a FN. Remove the current set of target gridboxes from ACT_k.
- The remaining gridboxes in ACT_k, are those for which our network activated but were not in a target bounding box. We remove entries from this set which are in GTIG to yield $ACT2_k$. These are all of the gridboxes which the network activated for that were either incorrect predictions or were in an ignore region.
- If counting individual entries (B_{INDIV} is "on"), the size of set $ACT2_k$ is the number of FP's otherwise we use connected components to group the activations and count the number of components as the number of FPs.

Note that after removing each set of target gridboxes, $GTTG_{k,n}$, from ACT_k in step 6, we may be left with a number of gridbox activations along the perimeter of the ground truth bounding box. The more our model confines its output to each bounding box region the less of a problem this kind of overestimating of pedestrian location will be. How much of an impact this has can be controlled by whether connected components are used or by controlling P_{ACT}; both when performing the evaluation and also when we are generating the training data and determining the target activations for an input image. As our approach is not meant to distinguish between occluded and unoccluded pedestrians, we must be careful when comparing the result to the benchmark. This is because the evaluation is based on the full bounding boxes and not just the unoccluded portions. To be fair we evaluate performance on both sets of bounding boxes (using the B_{OCC} parameter) and show a small performance difference between the 2 cases.

3.4 Experimental Results

3.4.1 Runtime Performance

This section details the run-time evaluation we performed on our models. We use TensorFlow [1] for our model training and benchmark our model on 3 different systems, one of which is an embedded architecture:

- A laptop with an i3-6100U @ 2.3GHz, 16GB of RAM and a Samsung 850 EVO SSD
- A desktop with an i7-4790 @ 3.6GHz, 24GB of RAM, a GTX Titan X and a Samsung 850 EVO SSD
- S32V234 automotive processor

The S32V234 is a high-performance, ultra low power, automotive grade processor which supports a range of applications in vision and sensor fusion. It includes Quad ARM Cortex-A53 cores @ 1GHz, 4MB of on chip system RAM, 3D GPU and Dual APEX-2 image cognition processor cores. We evaluate and report the overall frame-rate of our model on the APEX-2 processors. In Table 3.2, we give the observed performance on the laptop and desktop when forward propagating 3 samples (from the "3pos" sampling scheme) which is effectively a 681x227 pixel input region. Note that there is overhead in our unoptimized implementation. As such we show both the raw frames/sec which is the time

for forward propagating the 681x227 pixel region through our full network and also the full frames/sec which include overhead processing time of extracting and preparing frames and other overhead of the processing code.

	Laptop	Desktop	Dual APEX-2
Time (ms) / 3 patches	96.4	4.9	/
Raw Frames / sec	10.3	205	/
Time (ms) / frame	115.1	13.8	33
Full Frames / sec	8.6	72.4	30

Table 3.2: Summary of performance across hardware, including time per patch group & time per frame (to distinguish processing overhead and batch effect speedups) as well a frames per second

When evaluating the "3pos" scheme, with a non BN network, on the APEX-2 processor, we observe a frame rate of 15FPS on one processor and 30FPS on both processors. The addition of BN is expected to have a negligible impact on performance, allowing for the same framerate. For our application, the entire S32V234 SoC requires about 2W with approximately 800mW for both APEX processor cores. This power is achieved because all ConvNet inference computations are performed by the APEX processors at 30 FPS, with minimum control by the ARM core, and the GPU is not in use in this case.

With regard to automotive safety, the current implementation is not "aliasing" any intermediate tensor buffers. For processing a single patch, the overall required memory is 1.86 MiB which fits into the 4 MiB SRAM of on-chip memory. We note the APEX vector processing unit usage is 86%. Our GPU implementation was tested with 32-bit floating point weights and activations while our S32V234 implementation was quantized to 8-bit for everything (input, output and weights) except the tanh activations which were left as floating-point. If the desired threshold is small enough, the tanh could be substituted with a linear approximation allowing it to be 8-bit as well.

3.4.2 Detection Performance

Our target reference point is around 65% LAMR, putting us in the middle of the "Overall" benchmark on the Caltech Testing data as shown in Figure 3.5. This includes, for example, RPN+BF at a LAMR of 65% which is a region proposal network followed by a boosted forest. Although our approach is not yet able to match the performance of SA-FastRCNN

(a scale aware fast-RCNN) at 63% [65] and MS-CNN (multi-scale ConvNet) at 61% [11], we note that these approaches rely on significantly larger models with orders of magnitude more parameters and framerates lower by several multiples.

In general we found that increasing the dropout rate (we experimented with 0-50%) mainly helps as we began to sample positive and negative frames more frequently to build the training dataset (positive frames $<=$ every 5th frame and negative frames $<=$ every 10th frame). This makes sense as an increase in data redundancy should be helped by a larger dropout rate. However, in general the performance when oversampling in this manner was generally poor regardless of whether BN was used or not. The effects of large batch sizes were observed to be negative. Going up from 32 samples per batch generally meant a small increase in training time while increasing batch sizes beyond 64 (up to 128) meant increasing the time to convergence and decreasing the performance of the best model achieved by about 8%.

Without BN, the training is sensitive to class imbalance between positive and negative frames. The number of negative frames in this case should be about 1/2 to 2/3 of the number of positive frames. This may have to do with the large number of target gridboxes which end up being off with increasing numbers of negative frames. We don't observe this class imbalance issue when using BN. When using shared weights at the partially connected layer, the best network was found to be just over 1% higher LAMR than the best network trained without shared weights.

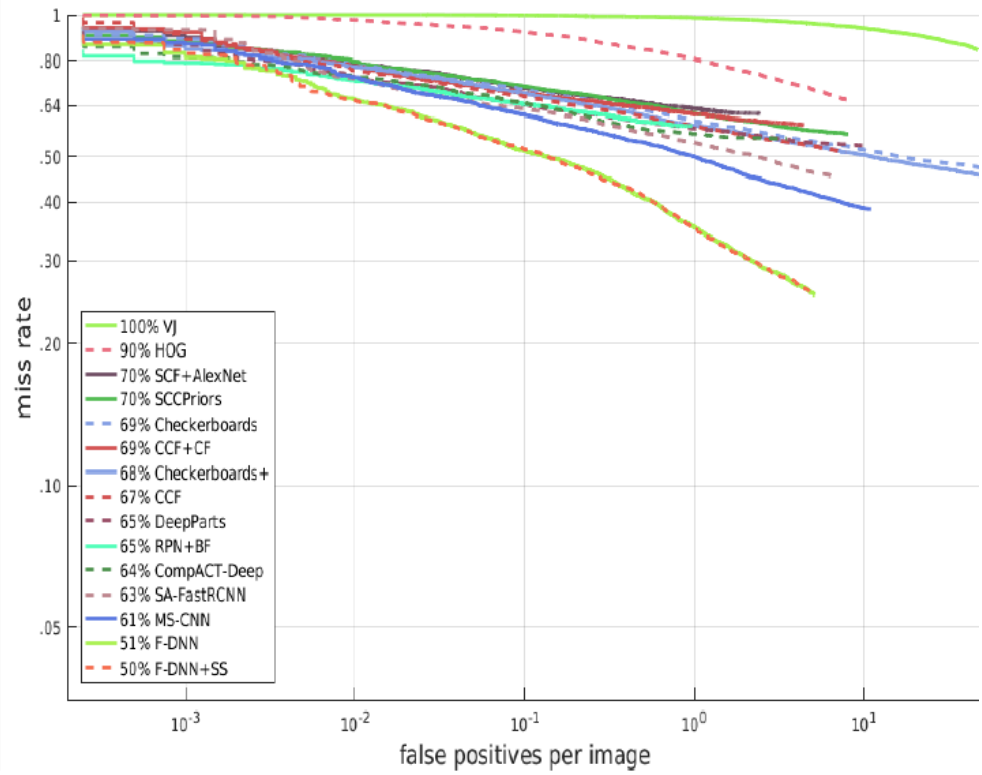

Figure 3.5: Top 13 (+ VJ and HOG) Results on the "Overall" Caltech Pedestrian Testing Dataset. Graph generated using visualization code and select available data [23] [25]

We generally found training to be fast, requiring no more than 10 epochs to converge (training time of 10-20min) when using 45K 227x227 pixel training patches. In general, the training time when using BN was 3-5X faster than without, also allowing for a wider set of learning schedules to be used. We observe the performance curve shown in Figure 3.6 when running a batch normalized network on the test set, trained using:

- A dropout rate of 40
- A dataset generated using a positive frame sample delay of 6 frames, a negative frame delay of 20 (sampling scheme 2 for training data) and a 50% P_{OBB} value when generating the data
- An exponential learning rate of 0.02 decayed at 0.9 every epoch

- Using the full ground truth bounding boxes (instead of taking the unoccluded subregions) as our target
- Sampling scheme "6hilo"

We can see in both cases, as we restrict P_{OBB} to be larger, a smaller number of gridboxes cover the perimeter of the ground truth bounding boxes. Note that switching to the unoccluded subregions of ground truth bounding boxes as the target results in a decrease of the LAMR by 0.01 for each P_{OBB} value. Thus using the harder case of full bounding boxes does not have much impact on performance. For P_{OBB}=2/3, with B_{INDIV} set to True, we have a LAMR of 0.712. In general, decreasing P_{ACT} below 50% gives an increasing boost in performance. For example, decreasing to 35% for the same network with P_{OBB}=2/3 and B_{INDIV} on (original LAMR of 0.721) gives a LAMR of 0.7009 while at 25% it is 0.6764. We aim to keep this threshold at 50% in the spirit of the evaluation benchmark. When using the 3pos or 9pos sampling schemes, at P_{OBB}=2/3 (B_{INDIV} on) we see a LAMR value of 0.7281 and 0.7238 respectively. A performance curve is shown in Figure 3.7 for P_{OBB}=2/3 and B_{INDIV} on.

We show some examples in Figures 3.8, 3.9 and 3.10, including results for the test dataset and a frame from the (unseen) ETH dataset for sequence LOEWENPLATZ [29]. Note the variation in scale which seems to be captured quite well. Overall, we notice consistent performance in negative regions of the image, showing non pedestrian patterns are captured well. We do notice some moderate difficulty in regions such as trees or at the edges of cars (where people getting out of cars tend to be). The blue region is the sampling region generated by the "3pos" sampling scheme and the red boxes are the active gridboxes which have detected a pedestrian.

3.4.3 Conclusion

In this chapter, we introduce an extension to the SqueezeNet architecture for performing coarse pedestrian detection. This extension uses partially connected neurons to mimic a weighted voting scheme for the location of pedestrians, resulting in a kind of heatmap of their location. In the process, we end up with a small 3.24MB model which can run in real-time at 30FPS on an automotive processor operating within a 2W envelope. The performance of the model is comparable to state of the art models on the Caltech pedestrian testing dataset.

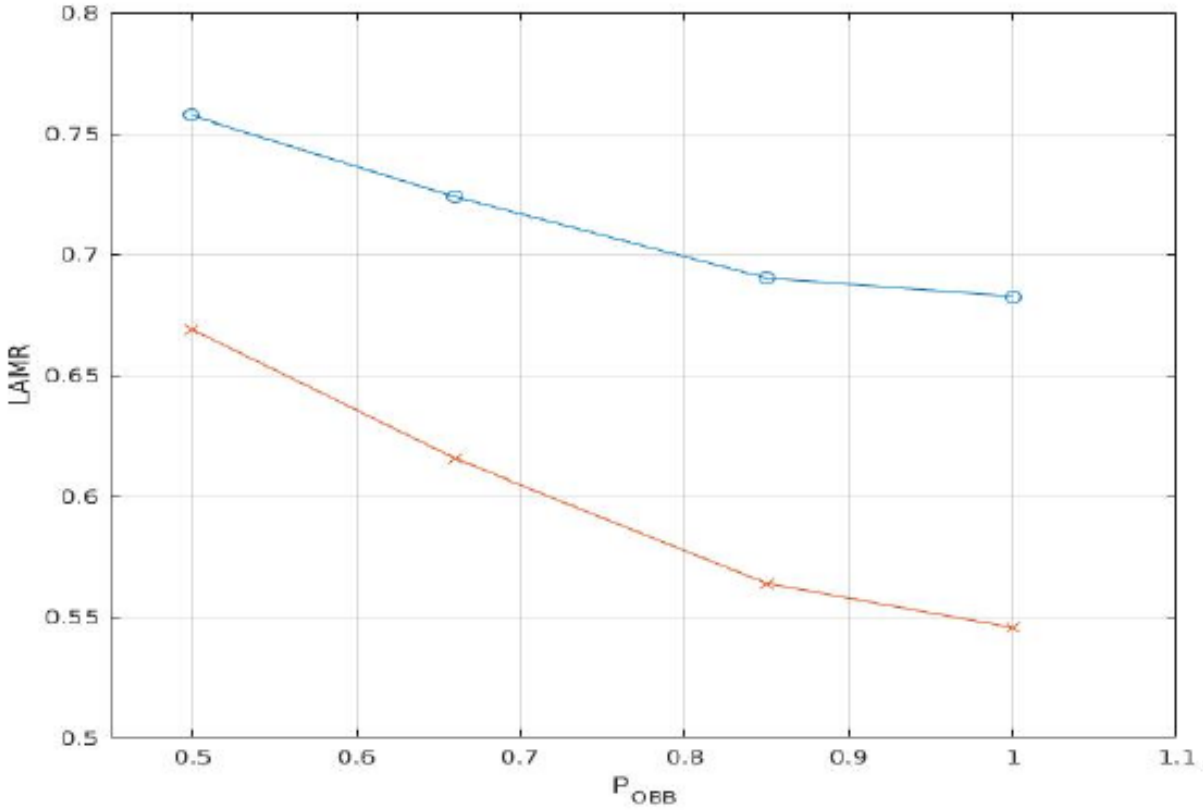

Figure 3.6: Plot of P_{OBB} values and the corresponding LAMR rate observed on the testing set with B_{INDIV} on (top/blue line) and using connected components/B_{INDIV} off (bottom/red line) for sampling scheme "6hilo"

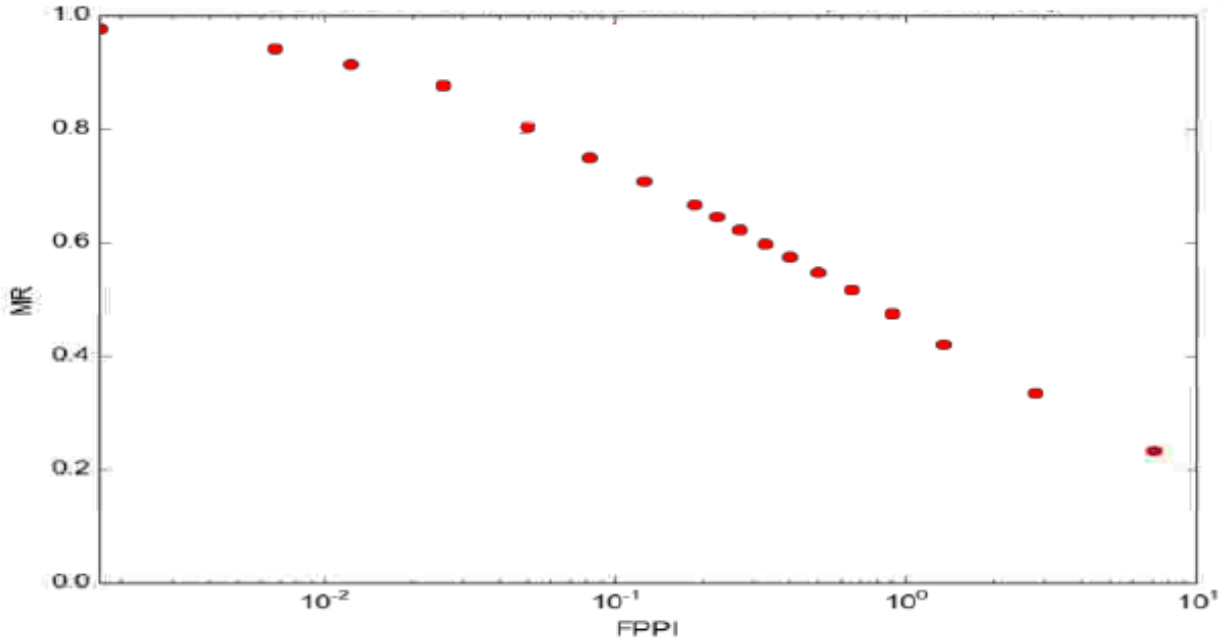

Figure 3.7: Example FPPI versus Miss Rate performance curve showing result points using a range of thresholds and, with $P_{OBB} = 2/3$ and B_{INDIV} on

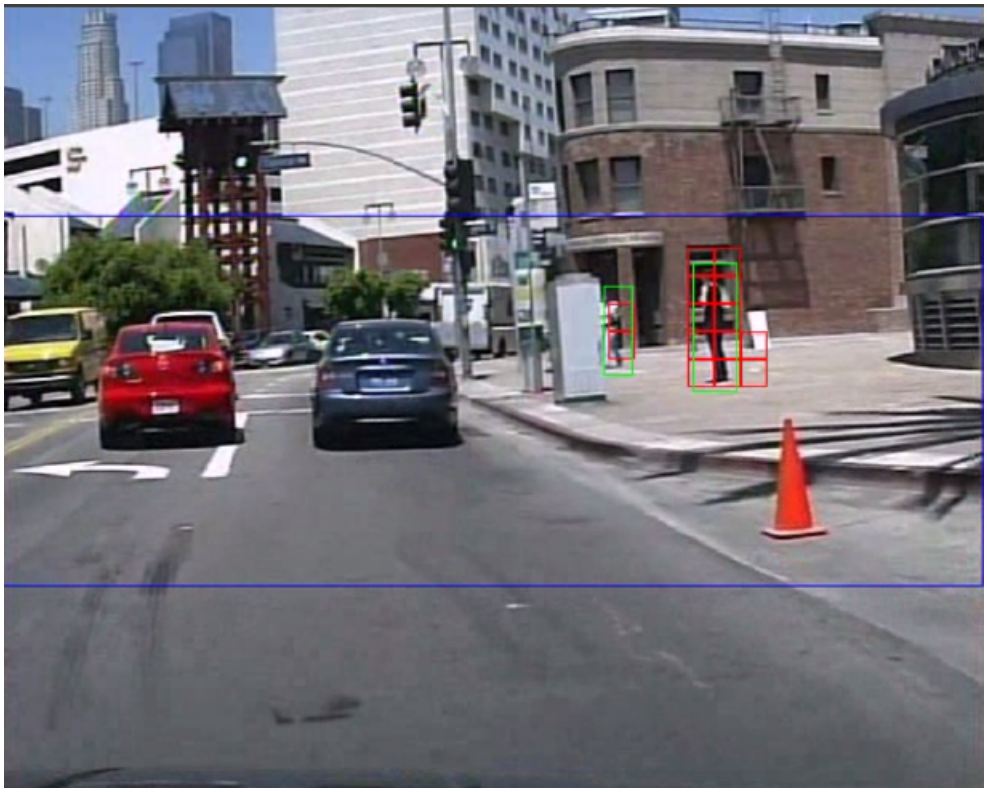

Figure 3.8: Sampled frame from the Caltech-USA testing dataset [26], including ground truth boxes (green) and predictions (red). This frames shows an occluded pedestrian at long range and an over-estimation that occurs due to the approximation of arbitrary bounding box sizes with fixed size gridboxes.

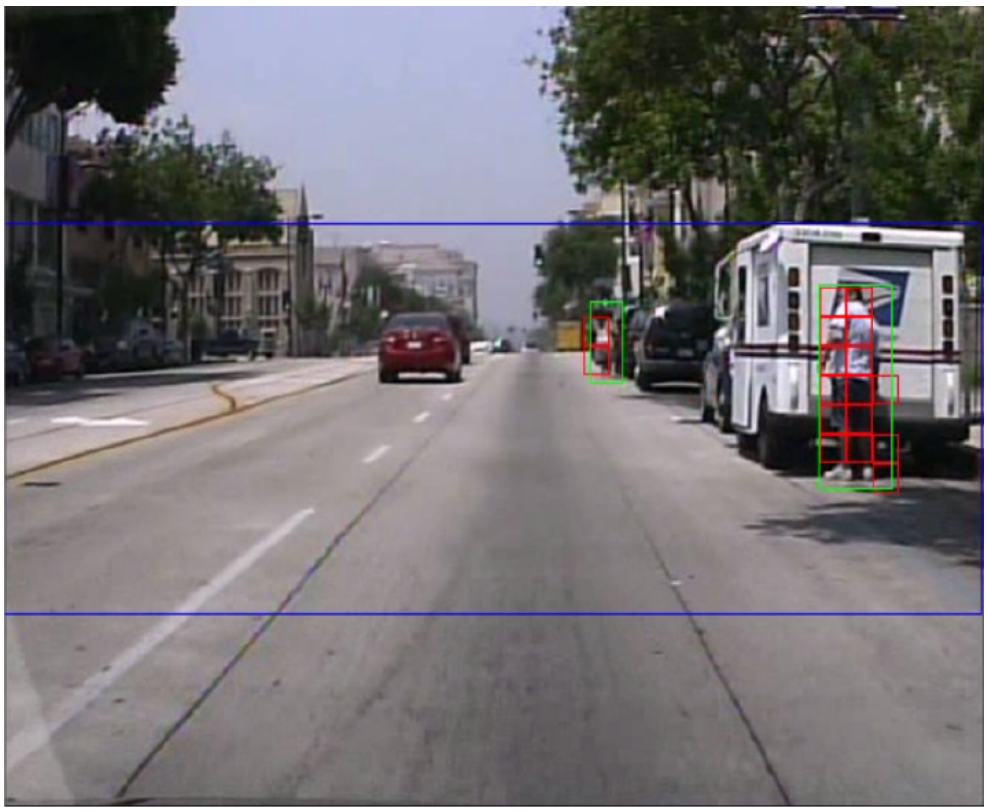

Figure 3.9: Sampled frame from the Caltech-USA testing dataset [26], including ground truth boxes (green) and predictions (red). This frame shows a long range detection at the approximate targeted visibility distance for which the 13x13 grid was selected for.

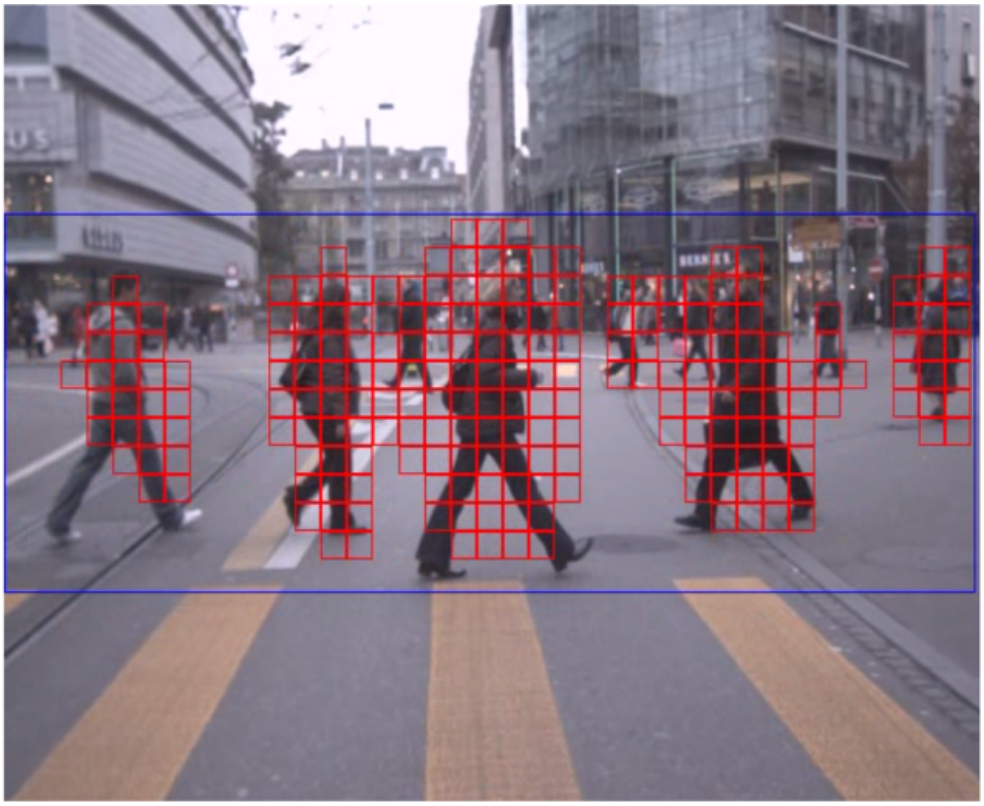

Figure 3.10: Sampled detection results for a frame from an out of sample dataset ETH LOEWENPLATZ [29]. A busy scene, in a different season, with lots of background (very far) pedestrians (missed) and intermediate & short range pedestrians (detected).

Chapter 4

PointPillars Slim: Fast 3D Object Detection

In this chapter, we present a novel adaptation of the PointPillars architecture that reduces the computational complexity (GFLOPs) of the original model and its parameter count by around an order of magnitude, while maintaining almost the same level of performance. Experimental results on the KITTI validation set indicate our model outperforms competing models for the same task while being drastically faster on embedded hardware, even to the degree that our model can run on an embedded system (the Jetson AGX Xavier) faster than some of the other models can on a desktop GPU. Experimental results are obtained on the KITTI dataset which, even though it has been supplanted by larger datasets, is still useful when training on limited capacity GPUs. It is important to note that none of the optimizations in this chapter involve the use of post-training optimizations that can generally be applied to any ConvNet (for example TensorRT or quantization). Such approaches are expected to further boost our models computational speed.

4.1 Introduction

Object detection on 3D point clouds has become a cornerstone task in areas such as autonomous driving and robotic navigation. This application has been aided by the rapid development of LiDAR sensors and a reduction in their cost, where their capabilities in quickly and accurately obtaining a scan of the surrounding environment means they can be effectively paired with low cost embedded systems for achieving object detection. This

development has in turn driven the need, and helped the development, of high performing object detection models that can run on limited power embedded systems.

Models such as VoxelNet [119], SECOND [116], PointRCNN [99] and PV-RCNN [98] represent competing architecture types to the PointPillars architecture, which has proven to be one of the leading solutions, providing an adept blend of detection capability and efficiency. Nevertheless, deploying PointPillars in real-world, resource-constrained environments requires further optimization in terms of computational demand and model size, and some existing extensions to PointPillars misidentify the necessary modifications to the model in order to do that. This chapter introduces a modified versions of PointPillars, which addresses these concerns without compromising detection performance. We demonstrate how a modification of the encoder stage & a reduction in network depth allows for a wide adjustment of the input grid resolution. These adjustments allow the network complexity to be reduced drastically and when combined with necessary training optimizations, allow for drastic reductions in GFLOPs and parameters, making the architecture much more suitable for deployment on embedded systems and for real-time applications. We show with our modifications that we can exceed the performance of directly competing models on the KITTI Car and the more difficult Cyclist classes.

4.2 Related Work

The field of object detection from 3D point clouds has seen a multitude of advancements, with methods often leveraging the rich geometric and spatial information contained within LiDAR data. VoxelNet is a 3D convolution based model with a reliance on 3D convolutions that leads to high computational costs. Compared to PointPillars, which eliminates the need for computationally expensive 3D convolutions by projecting features into a 2D plane, VoxelNet is more computationally heavy and less efficient. SECOND [116] improves on VoxelNet using a sparse convolution operator, reducing computation while maintaining the benefits of 3D convolutions. The use of sparse convolutions does increase complexity, and sacrifices speed. PointRCNN uses a different approach with a two-stage architecture similar to a traditional RCNN framework but adapted for point clouds. A first stage generates candidate object proposals directly from the raw point cloud, and the second stage performs the final bounding box regression. While PointRCNN offers high accuracy, its two-stage mechanism inherently involves more complexity and computational load. This makes PointRCNN less suitable for applications requiring real-time inference. PV-RCNN combines the advantages of voxel-based and point-based approaches by utilizing a Point-Voxel Feature Set Abstraction (PVSA) layer. The method shows significant improvements

in detection performance but with increased complexity (high FLOP counts and larger number of model parameters) due to using both voxel features and raw point features. SA-SSD [45] is a Structure Aware SSD modification that incorporates structural information of objects into the learning process. This comes in the form of auxiliary blocks of convolutional layers that are added to the network to help the it focus on learning parts of object shapes. The auxilliary part is necessary to guide the training to a "better" location in the weight space of the actual model but is not required for inference. However the initial network stages still rely on a complex and expensive sparse 3D convolution. Finally, RAD [3] and FA3D [21] are very similar approaches to PointPillars that focus on speeding up the model. Point cloud occupancies are scattered to a similar representation to PointPillars but the point cloud space is quantized using an occupancy cuboid with 2 separate resolutions in the horizontal direction and a vertical subdivision (instead of a single pillar) while also using 2D convolutions for the backbone. Upsampling is performed at similar branch points (to PointPillars) with a regression and classification head that uses class averages as anchors for the regression. The published results of these models on the Xavier AGX platform and the close similarity to the PointPillars model mean they can be directly compared to our model (on the same hardware).

4.3 PointPillars Architecture

The original PointPillars architecture is shown in Figure 4.1. It's novelty relative to previous work was the elimination of dense 3D voxels (where most voxels would be empty of any cloud points) through the definition of "pillars", the elimination of hand crafted features to encode point cloud data and the use of a 2D detector (possible due to the "pillar" representation) without special convolutional operations (such as SECOND) for the detection stage. This architecture can be broken down into 3 broad stages, listed below in the order they are executed in (overall execution flow visualized in Figure 4.1):

- **Feature Encoder**: This stage encodes the raw point cloud into a form usable by the feature extractor stage

  Voxelization: This stage uses a target grid size (HxW) and a 2D grid box size in physical dimensions to group the points in the input cloud. Grid boxes are 2D because they're pillars (not voxels) meaning they can be viewed as a rectangle in the BEV. Each pillar can have at most N cloud points (additional points are discarded). At most P pillars are allowed (additional pillars will be discarded). All dimensions are zero padded (points per pillar and pillars).

PointNet Encoding: The PointNet generates a single, encoded, multi-dimensional feature per pillar. When the original point cloud is grouped into pillars, a pillar can have many points within it all having 4 features (3D position and reflectance). The encoder converts the many points into a single higher dimensional "point" for each non empty pillar.

- **Backbone**: This stage is the bulk of the ConvNet and scatters the encoded features back to their original grid square positions but now into a feature map which has enough channels (C channels in total) to receive the encoded feature. This is also called a pseudo-image since it is essentially a BEV view of the PointNet encoded features. The remaining convolutional layers then perform the feature extraction needed by the subsequent detection head to classify and localize objects. Feature maps are added deeper into the network as spatial resolution decreases. At specific branch points, upsampling is performed using transpose convolutions so that the feature maps from each branch are the same H/2 x W/2 size.

- **Detection Head**: The detection head applies SSD to extracted features from the backbone to predict 3D bounding boxes. This stage includes the Non-Max Suppression stage that selects the final network predictions.

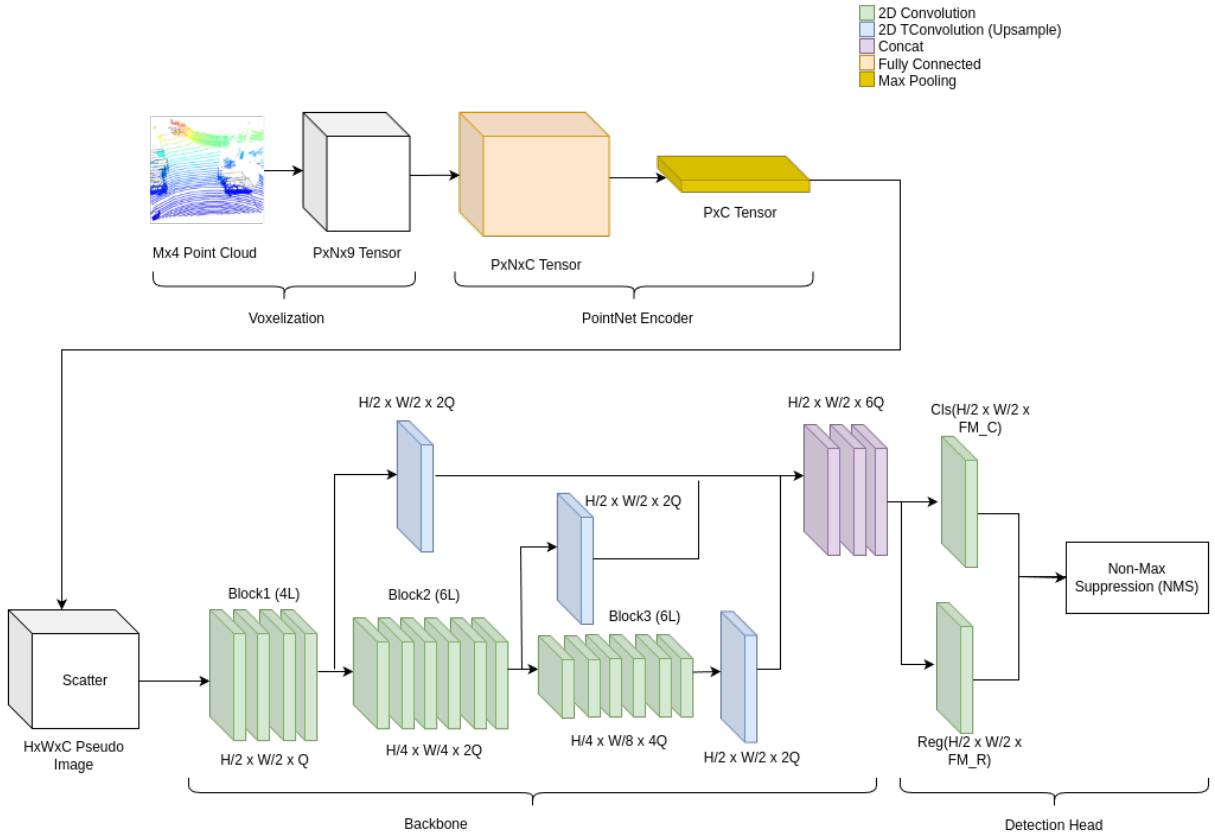

Figure 4.1: PointPillars architecture

If we assume the KITTI LiDAR coordinate frame (described in Section 2.5.2) (+X: Fwd, +Y: Left, +Z: Up) then we can define pillar dimensions P_X, P_Y, P_Z that tile over the physical range (for the purposes of this thesis all physical dimensions are in meters) defined by the cartesian product of:

$$[X_{min}, X_{max}] \times [Y_{min}, Y_{max}] \times [Z_{min}, Z_{max}] \tag{4.1}$$

These physical bounds define a 3D volume given by a respective width (meters) x length (meters) x height (meters) of $[X_{max} - X_{min}]$ x $[Y_{max} - Y_{min}]$ x $[Z_{max} - Z_{min}]$, with points outside this volume clipped/thrown away. Note that since our voxels are actually pillars we have $Dim_Z = \frac{Z_{max} - Z_{min}}{P_Z} = 1$, where Dim_Z is the number of grid positions along the vertical direction/the direction orthogonal to the BEV; hence the term pillar as a special

case of a voxel. Also note the ground plane of the scene and objects themselves need not align with $Z = 0$. Indeed the terrain in the general case may be far from a plane as well. The LiDAR sensor itself is considered the origin for the LiDAR coordinate frame and is about 1.73 meters above the ground (so the ground at the origin is approximately at $Z = -1.73$).

The dimensions of the grid as seen from the BEV are defined as shown in Equation (4.2). The number of grid squares is 1 in the direction orthogonal to the BEV grid, hence the more specific term pillars instead of voxels.

$$W = \frac{X_{max} - X_{min}}{P_X}, H = \frac{Y_{max} - Y_{min}}{P_Y} \tag{4.2}$$

The origin of this grid as viewed from the LiDAR frame is the back and right-most position in the volume. In the KITTI 3D Object Detection dataset, all GT objects are located within the view frustum of the camera. Since the LiDAR and camera frames are quite similar that means the bounds for Equation (4.1) are typically defined similar to [0, 70] x [-40, 40] x [-3, 3], where X_{min} is 0. Depending on the farthest expected distance one expects to see the classes of interest, the bounds can be set smaller or larger (the LiDAR sensor may not be able to "see" the objects/classes of interest beyond a certain point). The grid resolution $[P_X, P_Y]$ is a user-defined parameter that has a sizable impact on the number of FLOPs performed by the network and needs to be set with this consideration as well as the density of points the LiDAR sensor will produce (a sensor with fewer scan lines means having a dense grid is wasteful). The pillar construction means we can view this 3D detection problem in the context of a 2D detection problem, allowing us to use 2D detectors (with extra parameters to predict the vertical dimension).

An Mx4 dimension real-valued tensor is used as input to the network with the first dimension (M) representing the number of points in the input point cloud and the second dimension (4) representing the 3 spatial dimensions and reflectance value of the corresponding point. The voxelization portion of the first stage converts the Mx4 point cloud tensor into a PxNx4 tensor where P is the number of pillars and N is the number of points per pillar. Here the number of pillars and points per pillar are user set parameters that control for complexity and must be set based on the characteristics of the LiDAR sensor (point density) and desired model complexity. The tensor is zero padded such that missing values have no impact. Additional features are added for each point in a pillar such as the 3D distance of each point from the pillar mean as well as the 2D distance from the pillar center (in the BEV); a total of 5 more features.

The PointNet encoder at the input of the network takes this updated PxNx9 tensor

and applies the encoder, which is essentially just FC layers that in their last layer has C channels, followed by a max-pooling operation. The max-pooling operation is performed over the second dimension (N), leaving P pillars each having a C dimensional feature vector. Each pillar has a corresponding physical location, so the C dimensional feature vectors are scattered back to those locations in the pseudo-image. This image is the input to the backbone network. The backbone is applied to the pseudo-image and at specific locations, the feature maps are taken and upsampled using a transpose convolution to common dimensions to allow concatenation. This allows detection at the finer scale needed for higher IOUs of predictions with the GT since with a larger feature map you have more box predictions per region at a finer scale of prediction (double the resolution in each dimension when a 2x upscaling is used).

The final stage in the network for PointPillars uses SSD as described in Section 2.3.1 but will be explained in detail below in the context of PointPillars. From Figure 4.1 we see the height and width of the feature maps at the head will typically be half the resolution of the input, but the number of channels is controlled in a specific way. The number of feature maps devoted to classification is actually a sum of "classification feature maps" F_C which are for predicting the probability of an object at a particular anchor and "directional classification feature maps" F_D which are a subdivision of arc from 360 degrees in the BEV that coarsely bin the object yaw angle:

$$FM_{classification} = F_C + F_D = N_{anchors} * N_{classes} + N_{anchors} * D_{dir} \tag{4.3}$$

Here $N_{classes}$ is the number of ground truth classes while $N_{anchors}$ is the number of anchor boxes located at each position in the feature maps. D_{dir} is the number of directional bins to use per anchor, in a class agnostic way. In the case of PointPillars, 2 directional bins are used, so the classifier computes the probability that the object yaw coarsely falls within one of two bins each covering 180 degrees of arc. The fine grained adjustment described below in Equation (4.4g) is then added to the center of the bin with higher probability to create the final prediction.

Each class has a set of anchor boxes, usually defined based on the average dimension of 3D boxes for that class on the training set. For example, we can compute the average box size in KITTI for the classes we're interested in:

Class	Average Size (meters)
Car	3.88 x 1.53 x 1.63
Cyclist	1.76 x 1.74 x 0.6
Pedestrian	0.84 x 1.76 x 0.66

Table 4.1: Summary of Average KITTI Box Sizes (LHW ordering based on Figure 2.23)

An anchor of a particular size is tiled across the entire grid, but the centers of the anchor can be anywhere. However generally, they're placed centered at each pillar center for simplicity (since we want each anchor to make predictions spatially "close" to itself). Note how the anchor dimensions used can be much larger than the pillar dimensions (this depends on the class) but selecting the average dimensions should work well because the feature maps are actually predicting a difference between selected anchors and the ground truth boxes in the scene; the raw output of the regression feature maps is an "encoded" representation of the true prediction, which needs to be decoded to obtain the actually predicted parameter values. There are various encoding/decoding schemes, however the one used by PointPillars is as follows, with the following assumptions for the groups of equations 4.4 and 4.5:

- A particular class c of the set of classes being predicted for
- Anchor k for class c. Each class can have a different number of anchors.
- $d_a^{k,c} = \sqrt{(l_a^{k,c})^2 + (w_a^{k,c})^2}$ is a term describing the combination of length (l_a) and width (w_a) of anchor k for class c. The anchor height is h_a.
- The remaining regression features are the box position (x, y, z) in the LiDAR frame and its yaw angle θ
- $()_p$ each feature of the raw feature map predictions (decoding) or the expected feature map activation from a ground truth (encoding)
- $()_d$ each feature of the decoded predictions or, during encoding, each feature of the encoded anchors
- $()_a$ as the anchor parameters

The equations to decode the activations of the feature maps in the detection head are then:

$$x_d^{k,c} = d_a^{k,c} * x_p^{k,c} + x_a^{k,c} \tag{4.4a}$$
$$y_d^{k,c} = d_a^{k,c} * y_p^{k,c} + y_a^{k,c} \tag{4.4b}$$
$$z_d^{k,c} = h_a^{k,c} * z_p^{k,c} + z_a^{k,c} \tag{4.4c}$$
$$l_d^{k,c} = l_a^{k,c} e^{l_p^{k,c}} \tag{4.4d}$$
$$w_d^{k,c} = w_a^{k,c} e^{w_p^{k,c}} \tag{4.4e}$$
$$h_d^{k,c} = h_a^{k,c} e^{h_p^{k,c}} \tag{4.4f}$$
$$\theta_g^{k,c} = \theta_a^{k,c} + \theta_p^{k,c} \tag{4.4g}$$
$$\tag{4.4h}$$

When a ground truth box is placed at a certain location, its information for the loss computation at the detection head can be encoded by the equations:

$$x_p^{k,c} = \frac{x_d^{k,c} - x_a^{k,c}}{d_a^{k,c}} \tag{4.5a}$$
$$y_p^{k,c} = \frac{y_d^{k,c} - y_a^{k,c}}{d_a^{k,c}} \tag{4.5b}$$
$$z_p^{k,c} = \frac{z_d^{k,c} - z_a^{k,c}}{h_a^{k,c}} \tag{4.5c}$$
$$l_p^{k,c} = log(\frac{l_d^{k,c}}{l_a^{k,c}}) \tag{4.5d}$$
$$w_p^{k,c} = log(\frac{w_d^{k,c}}{w_a^{k,c}}) \tag{4.5e}$$
$$h_p^{k,c} = log(\frac{h_d^{k,c}}{h_a^{k,c}}) \tag{4.5f}$$
$$\theta_p^{k,c} = \theta_g^{k,c} - \theta_a^{k,c} \tag{4.5g}$$
$$\tag{4.5h}$$

The 2 angles typically used for anchor regression are 0 & $\frac{\pi}{2}$ rad since the most common yaw angles observed in driving scenarios tend to be of cars in lanes going in the same

or incoming direction (without accounting for front/back, both are a box in the KITTI LiDAR frame rotated by $\frac{\pi}{2}$) and vehicles at crossings/intersections where vehicles boxes would be rotated by 0 or π degrees. After box predictions are decoded, the highest scoring M predictions having a minimum threshold are grouped and NMS is applied to prune overlapping, redundant predictions.

During the training phase, the predictions don't need to be decoded. Instead, the ground truth 3D boxes need to be encoded relative to the anchors so the loss computations can proceed. To perform this encoding, the anchors for a class must be matched to its ground truth boxes using the IOU between them as a similarity measure. A minimum IOU is required to consider a match as positive. Its not uncommon to have multiple anchors matching to a single ground truth box since anchors are so dense (each grid box/feature map position is the location of one or more anchors). For PointPillars, IOU is computed using 2D boxes in the BEV (ignores box vertical position and height) and boxes are axis aligned before comparison by finding the angle closest to the smallest possible integer multiple of $\frac{\pi}{2}$ rad. The IOU matching of ground truths to anchors are why statistics from the training dataset are a good initialization for anchor sizes instead of random dimensions or unrelated sizes. At the end of the matching process there will be 3 categories of labels that can be assigned to an anchor:

- A positive label (1) indicating the anchor was matched to a ground truth box.
- A negative label (0) indicating no match of an anchor to a ground truth box. The vast majority of these will be background.
- An ignore label (-1) indicating this anchor is ambiguous and we ignore it so as not to introduce additional noise in the loss.

The labels assignment strategy can be summarized as follows, assuming 2 thresholds $T_{matched}$ and $T_{unmatched}$ with $T_{matched} > T_{unmatched}$. For a single class, all anchors with either a max IOU with a GT box or with an IOU $>= T_{matched}$ have their label set to 1 (foreground/positive anchor). All anchors whose IOU is below $T_{unmatched}$ have their labels set to 0 (negative anchor). All remaining (ambiguous) anchors have their labels set to -1 (ignore) since their status is ambiguous.

The assignment of labels raises the issue of imbalance between positive and negative anchors. For regression this is more straightforward since only positive anchors are used for loss computation (negative anchors have no ground truth box to regress to). This can still introduce 'dominance' of specific anchors or anchor groups, which may need additional

balancing in the loss term. However, in practice the heavy reliance of PointPillars on sampling augmentation helps mitigate this issue (omitting sampling augmentation drops performance by 15-20% AP). This is because individual ground truths (as well as samples inserted into a scene) not only get rotated, translated & scaled relative to their positions in the scene, the scene itself is rotated, translated & scaled as well. To summarize, the loss in the context of regression is focused on refining predictions given that we know there is a higher chance that there is something at the location of an anchor (since it's a positive anchor). Classification, on the other hand, is focused on identifying whether a particular anchor actually has a higher chance of having an object at that location. This is why the classification loss is not focused only on positive anchors, using both the positive and negative anchors that by default will be highly imbalanced, so it needs to leverage the focal loss described in Equation (2.6). The values used in the focal loss are typically $\alpha = 0.25$ and $\gamma = 2$ and work quite well in practice. For example, with a typical PointPillars model we would observe around 50-75 positive anchors and around 50000-75000 negative anchors (an imbalance of about 1000 to 1), with no convergence issues. Note that this imbalance of anchors is different from class imbalance, where we may want to have similar total numbers of instances of each class in the scene.

4.4 Network Improvements

Method	FPS	Car 2D		Car BEV		Car 3D	
		mod	hard	mod	hard	mod	hard
PV-RCNN	12.5	94.47	92.28	91.11	88.93	84.43	82.69
SA-SSD	25	95.9	93.57	92.79	90.32	84.54	81.71
PointRCNN	10	91.99	89.93	88.01	86.1	78.83	77.58
SECOND	20	89.59	88.72	88.20	86.98	78.78	77.41
RAD	84	91.94	89.34	88.21	85.70	75.8	72.65
FA3D	57.83	/	/	88.1	85.49	75.47	71.97
PointPillars*	50	89.43	88.26	88.31	86.96	77.65	75.56
PPslim	94	89.51	88.18	87.74	86.56	77.5	75.24

Table 4.2: Average Precision at IOU threshold of 0.7 for our method and comparison methods on the Car class of the KITTI val dataset. We mark PointPillars with a * as this is our best performing implementation of this model.

In Table 4.2 we compare the best models from the improvements presented in this

chapter to reported results from the KITTI benchmark on the validation set. We don't currently have results on the test set since this requires a publication submission to allow evaluation on the KITTI test set. The results show that substantially more complex models (lower speeds) do bring an increase in performance however our approach does beat our nearest direct competitor RAD in both speed and performance. The results we report for PointPillars contains our own result for this model (in terms of speed and performance) since experimentally it was difficult to replicate the best reported result on the Car class (validation set). Nonetheless even with this slightly lower result, the baseline PointPillars model and our modification still outperforms RAD by a noticeable margin while being on par or significantly faster, respectively, in execution on an embedded platform.

4.4.1 Voxelization and Encoder Optimization

In this section, we show how a modification of the encoder structure can yield a reduction in the total GFLOPs at this stage. When reducing the network complexity overall, we want to ensure a high number of channels in the encoder stage to maintain a rich feature encoding however we don't want to maintain the corresponding increase in FLOPS. We can achieve this by taking advantage of the max-pooling operation in the encoder, which removes a large dimension from the input tensor (the number of points per pillar). If we squeeze the number of features that are part of the input tensor by reducing the dimensions of the FC layers before the encoder, then apply wider FC layers after the encoder (once the large dimension has been eliminated), we show how we can drastically reduce encoder complexity.

In the original PP model, the authors use a large 12k ceiling for the maximum number of pillars at a 0.16m grid resolution. As we can see from Figure 4.2, if we plot for a 0.16m (p16) grid resolution the distribution of the number of voxels we expect to see across the entire KITTI training dataset, this is not necessary. The maximum number we expect to see is 9180. The large spike at the lower end represents a handful of sequences of the reference vehicle, parked outside a building with very little change in occupancy. As we decrease the grid resolution, we expect more LiDAR points to fit per pillar while having less non-empty pillars/voxels (Table 4.3). This number is scanner dependent (the Velodyne scanner produces ∼1M points per second of which at most ∼20.78k fall into the view frustum) and may need to be accounted for if using a LiDAR sensor with a drastically different point density other than the one the model was trained on. In our case we will show experimentally that the model is not particularly sensitive to grid resolution (unlike competing models like RAD), even while reducing the model complexity.

Due to the construction of PointPillars, the voxel limitation is not necessarily an issue. This is because the complexity of the model only scales with the number of occupied voxels and not the number of points per voxel. The original tensor is zero initialized, meaning if we initialize it to a size of (9000,100,4) with a maximum of 9000 voxels, 100 points per voxel and 4 features per point. Any unfilled voxels or unfilled points per voxel will just be ignored in the computations (only occupied voxels are used from the tensor) and the encoder applies the max-pool operation along the second axis (points per voxel). The feature maps in the backbone of the network have 1 location per grid square regardless of how many points are in that grid square/pillar. This also means that drawing a random cloud from the dataset to compute GFLOPs will generate a noisy estimate of the model complexity since the number of cloud points varies between 12.8k and 20.78k and the number of voxels will vary across the dataset. For our models we assume a maximum of 9000 voxels for all models with coarser resolutions than 0.16 and assume the authors original 12000 for any 0.16 models.

Grid (m)	Max Voxels	Max Points Per Voxel	Mean Points Per Voxel	Sdev. Points Per Voxel	Max % Occupied Voxels
0.16	9180	599	3.3	5.5	4.1
0.22	7159	795	4.4	8.1	6.1
0.24	6590	992	4.8	8.99	6.7
0.26	6135	856	5.2	9.9	7.3
0.28	5762	1174	5.64	10.8	8.1
0.32	5048	1235	6.48	12.72	9.1

Table 4.3: Comparison of grid size with voxel limits and statistics of points per voxel

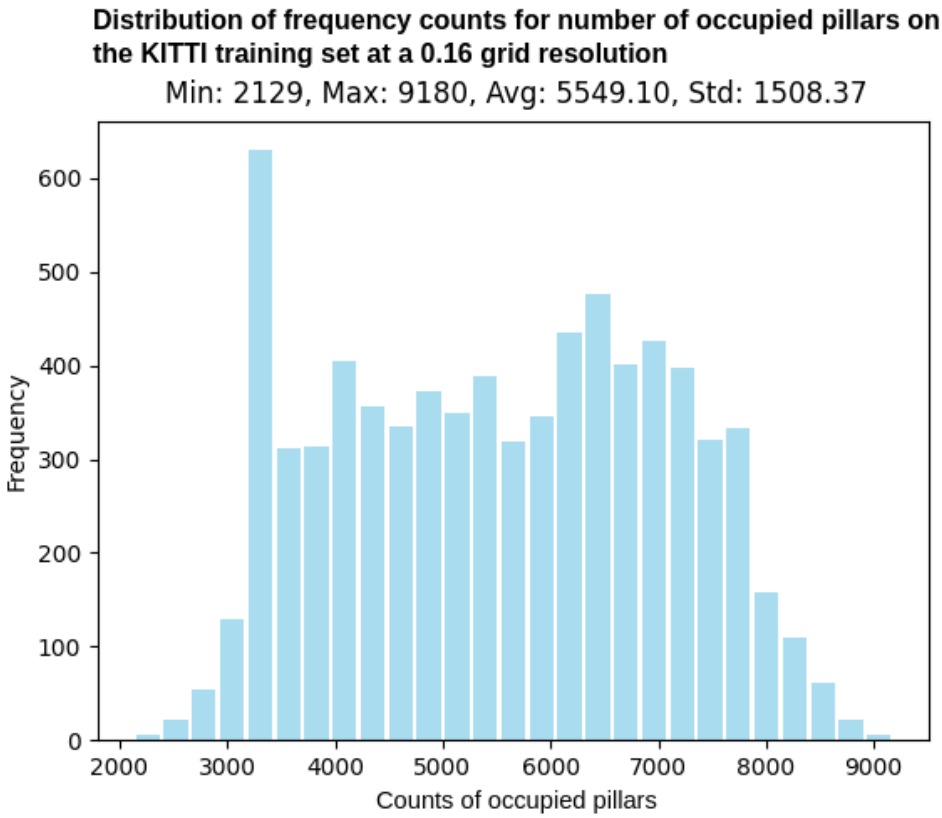

Figure 4.2: Histogram of number of non-empty voxels (voxel/pillar occupancies) at a 0.16 (p16) grid resolution over the entire KITTI training dataset. Summary statistics are shown in the title.

For the encoder, it is illustrative to use an example. Consider the following:

- A batch (size B) of point clouds as input
- A limit to the number of pillars P and points per pillar N, at 0.28x0.28 grid resolution P=8000 and N=100.
- The encoder expects D input features (augmented LiDAR points) and produces C output features. For the original model, D=9 and C=64.
- All point clouds in the batch hit the P & N limits

This represents a worst case scenario computationally (since we must process the maximum limits of voxels and points). Without loss of generality, we can assume B=1. This is because for B>1, we can simply take the B, (P, N, D) tensors and stack them along the first dimension, giving a new tensor of size (B*P, N, D). This is equivalent to an image of size (B*P, N) with D channels. If each point cloud has less than P pillars, the size of the first dimension will be less than B*P. A 1x1 convolution is then applied across the (B*P, N, D) 'image' to get a (B*P, N, C) output. Applying a max across the 2nd axis gives a (B*P, C) output from the encoder stage.

The computational cost (in terms of MACs) of a 2D convolution (neglecting bias) depends on the kernel size (K), the input channels (C_{in}), output feature map size (H_{out} and W_{out}) and output channel count (C_{out}):

$$K * K * C_{in} * H_{out} * W_{out} * C_{out} \tag{4.6}$$

For the PointNet encoder this means a cost of the following (w/ B=1):

$$D * B * P * N * C \tag{4.7}$$

Using B=1, D=9, N=100 and C=64 we have the following MAC counts:

Grid Size	Maximum Pillars (P)	Encoder GMACs
0.12^2	16000	0.921
0.16^2	12000	0.691
0.28^2	8000	0.461

Table 4.4: Encoder MAC Counts With Varying Grid Size

Specific to the encoder, reducing the number of output channels is the most direct way to reduce the number of operations, for example cutting the output channels (C) in half. However this can lead to a loss in representation ability in the downstream backbone part of the network. Instead we can squeeze the channel count in the encoder, followed by an expansion through a 1x1 convolutional layer to a larger number of channels (M). The 1x1 would be applied after the max layer, after the point per pillar dimension (N) has been reduced, allowing for much more efficient computation. Assuming B=1, we now have the following MAC counts, with C*P*M the cost of the additional 1x1 layer:

$$D * P * N * C + C * P * M = C * P(D * N + M) \tag{4.8}$$

Grid	Maximum Pillars (P)	Output Encoder Channels (C)	Output Expanded Channels (M)	Encoder GMACs
0.12^2	16000	64	/	0.921
0.12^2	16000	32	64	0.494
0.12^2	16000	16	64	0.247
0.16^2	12000	64	/	0.691
0.16^2	12000	32	64	0.370
0.16^2	12000	16	64	0.185
0.28^2	8000	64	/	0.461
0.28^2	8000	32	64	0.2468
0.28^2	8000	16	64	0.1234

Table 4.5: Encoder Configurations With Compute Complexity

The ratio of new to original MACs is then given by (take C_1 to be the original PointNet output channels and C_2 to be the new):

$$\frac{P * C_2(D * N + M)}{(D * P * N * C_1)} \tag{4.9}$$

If we take M=C_2 (we expand out to the size of the original encoder through the extra layer) then we have:

$$\frac{C_2}{C_1} + \frac{C_2}{D * N} \tag{4.10}$$

Since C_2 will be small relative to D*N, the biggest impact is how much we can shrink the PointNet encoder while still maintaining the same level of performance. We can summarize the improvements as shown in Table 4.5. In fact combining both a modified encoder and a reduction in grid resolution by a moderate amount should bring an even larger reduction in model complexity. The biggest gains are at higher grid resolutions because they require a higher limit on the number of pillars (P). Other encoder variations are possible, for example a narrow number of channels C, followed by a wide number of channels (128), followed by a narrow number of channels (say M=32 or 64).

4.4.2 Backbone and Detection Head

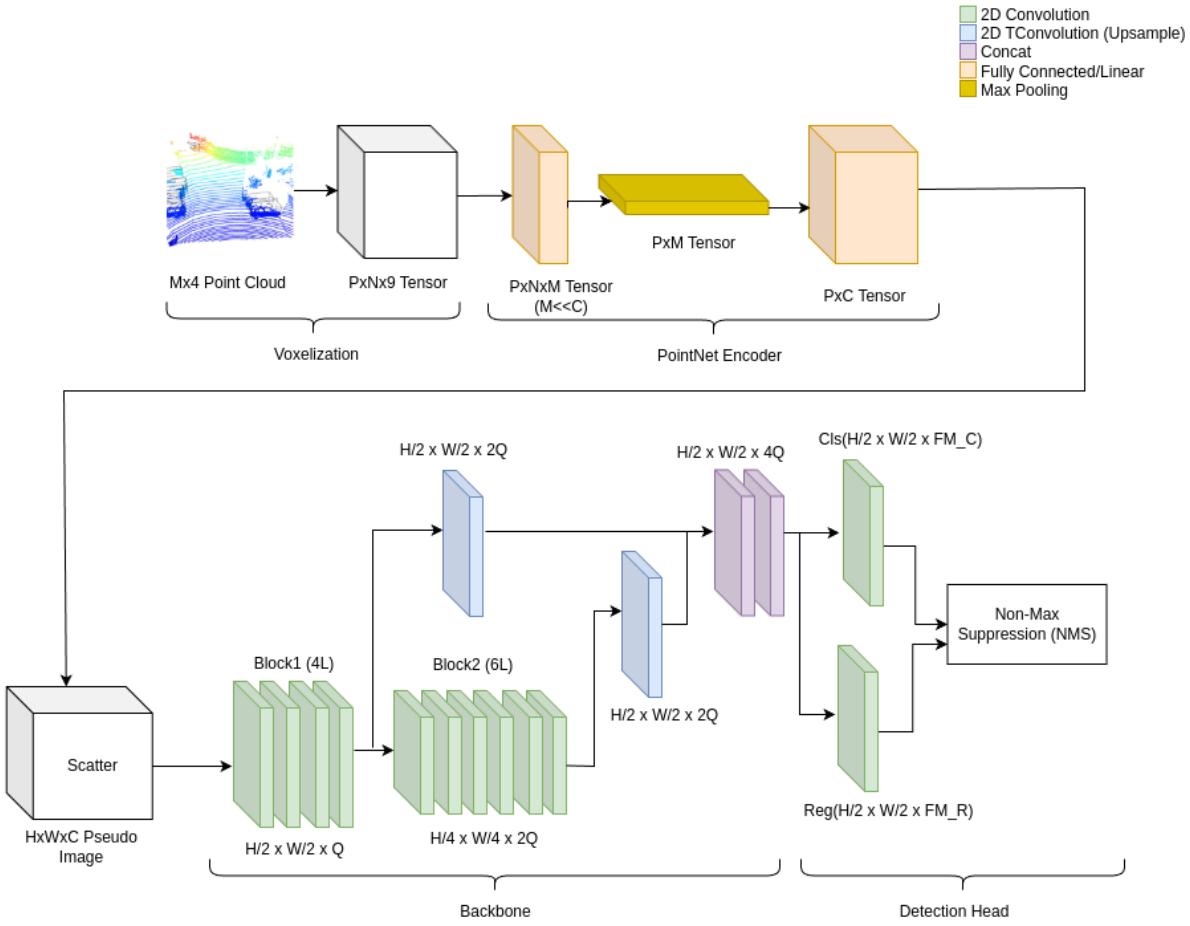

Figure 4.3: PPslim model

Figure 4.3 shows the modified PPslim model after applying the encoder optimization proposed in Section 4.4.1. We see in the encoder the original PxNxC tensor, that is reduced to PxC by a max-pooling layer, is now PxNxM where the feature dimension M is smaller than our expected final number of channels C. After the max-pooling reduction, we finally apply another, larger, fully connected layer. This larger layer is less expensive computationally at this stage since we've eliminated the large point dimension (N) though the max-pooling layer and we observe a further savings since the input to the max-pooling was reduced in dimension itself.

Following the encoder optimization, there is a subsequent reduction in the depth of the backbone where the last block of feature maps is removed. The reasoning for the backbone reduction comes from the fact that down-sampling feature maps an additional time (each block starts with a down-sampling) produces feature maps that are too small to be useful. With the additional Cyclist class that's already significantly smaller than Car, the earlier feature maps would be most relevant to retain the extra information. This was evaluated in comparison with cutting the number of channels in each layer by a fraction up to 1/2, however it was found to be less repeatable to consistently train to the same level of performance. In general, such an approach would have to know what fractional level to set the number of channels to, while the alternate approach that we settled on focuses on removing an entire block which is at too low of a resolution (since its at the deepest parts of the network) for the smallest expected class is a more straightforward solution. We validate in Section 4.4.3.2 that the performance decrease using the removal of feature maps is mainly relevant at distances where there are already too few points for the extra layers to matter.

When reducing the network depth and grid resolution in a way that removes so many parameters and convolutional feature maps, we see a need to train the resulting model for over twice the duration of the original unmodified model (approximately 600-800 epochs instead of 300-400 epochs). The learning rate that generally works is an exponential one starting at 0.0003 with a decay factor of 0.8 applied in a staircase pattern. We use the Adam optimizer (no other optimizer seems to converge anywhere near as well as Adam, for example RMSprop) with an initial weight decay (unfixed) of 0.0001. We notice that the model performance converges in a familiar pattern that can be used to judge the progress of training. PPslim models at the point of convergence show an oscillatory jump in performance, where before the jump, the models AP is 3-4 points from its final performance. One could stop training at this point but ramping the learning rate back to its starting rate and repeating the process generally helps to achieve another 1-2 points of performance. We explain this phenomenon by reasoning that given sufficient time, the model settles into a "good" overall region in the loss landscape and starts to settle into a specific part of it. Ramping the learning rate back up allows further exploration in this "good" overall region and a final settling.

As we show in Section 4.4.3.1, regression plays a larger role in the degradation of our models performance than classification. Even though we use a scaling factor of 2:1 in favor of regression to account for this, we also have the option to scale the loss of individual regression features. We find that an elevated weighting of 4:1 for the predicted vertical box position and box height relative to the other features works best. This makes sense, as this scaling allows the model loss to focus on the part of our model where its contribution

to the error is largest.

In Appendix A, we provide a more detailed parameter breakdown for our model setup in the case of a 0.22m pillar grid.

4.4.3 Analysis

The first additional performance results that needs to be added to Table 4.2 is the models comparison for the 2nd class of interest (Cyclist) that addresses Vulnerable Road Users (VRUs), which is shown in Table 4.2. An analysis of KITTI VRU-type classes such as Pedestrian and Cyclist in terms of the LAMR metric from Chapter 3 is not available for comparison purposes from other authors as the LAMR metric is a much less common metric than AP (for example on Pascal VOC, COCO, KITTI, NuScenes, Waymo). So we focus our analysis based on the standard metric used for the KITTI dataset instead much like other popular object detection datasets mentioned above. As most papers focus solely on reporting their performance on Car for the validation set, we mainly focus on comparing ourselves to RAD which is our direct competitor. Here we see the original PointPillars model consistently outperforms RAD by a wide margin and our slimmed model PPslim generally does too.

Method	Grid (m)	Cyclist 2D		Cyclist BEV		Cyclist 3D	
		mod	hard	mod	hard	mod	hard
RAD	0.16	71.27	68.23	65.53	61.79	62.03	58.28
FA3D	0.16	/	/	66.29	62.89	60.23	56.86
PointPillars*	0.16	71.75	68.45	68.58	62.9	65.6	61.16
PPslim	0.22	71.1	69.14	67.15	62.88	64.39	59.76

Table 4.6: Average Precision at IOU threshold of 0.5 for our method compared to RAD on the Cyclist class of the KITTI val dataset. Note PointPillars* is our best performing implementation of this model.

Given these performance results we can also address the question of runtime performance (summarized in Table 4.7). For benchmarks, we measure FPS on both a Jetson AGX Xavier embedded board and a Core-i7 CPU & RTX 2080Ti GPU for comparison. On the Xavier board, we benchmark at several power settings as well as core configurations (as described in Section 2.6.2).

Method	Grid (m)	Device / Power Setting	Voxelization (ms)	Backbone (ms)	Head (ms)	FPS
SECOND	/	MAXN	/	/	/	4.5
Point-RCNN	/	MAXN	/	/	/	1.3
RAD	0.16	MAXN	0.62	83.20	7.85	10.91
FA3D	0.16	MAXN	0.62	125.26	11.6	7.27
PointPillars*	0.16	MAXN	3.66	32.8	75.8	8.9
PPslim	0.22	MAXN	2.49	31.56	16.6	19.7
PPslim	0.22	15W	4.24	56.7	30.4	10.95
PPslim	0.32	MAXN	2.13	21.8	13.5	**26.7**
PPslim	0.32	15W	3.58	39.36	24.4	14.84
RAD	0.16	GPU	0.48	9.79	1.57	84.46
FA3D	0.16	GPU	0.48	14.34	2.47	57.83
PointPillars*	0.16	GPU	1.64	6.06	12.2	50.1
PPslim	0.22	GPU	1.33	6.38	3.6	88.2
PPslim	0.32	GPU	1.26	4.53	9.6	**104**

Table 4.7: Runtime comparison for competing approaches compared with PointPillars (we mark it with a * to indicate as our implementation of the original model) & PPslim on the Jetson Xavier AGX platform

We see that except for the voxelization and some of the head post-processing, the GPU is the main driver of increasing the FPS on the PC for PointPillars since the backbone is by far the biggest consumer of MACs. The RAD paper reports an FPS of 42 for their PointPillars implementation on a 2080 Ti while the original PointPillars network was reported to run at 42 FPS on a 1080 Ti. The difference in inference speed between the 1080 Ti and 2080 Ti should be about 1.3x to 1.4x. This significant difference in performance between the 2 GPUs explains why, when we run the original PointPillars model on a 2080 Ti, we obtain an FPS of 50 and on the Xavier an FPS of 8.9 (slightly different from the FPS of 6.15 reported by RAD). So the true runtime speed of the original PointPillars model (on the AGX) at 8.9 FPS is already quite comparable to the 10.91 FPS speed reported for the optimized RAD model. We note that our fastest model, on a GPU, runs at almost exactly the same speed that the authors original PointPillars model runs at after TensorRT optimization (ours is not TensorRT optimized).

Its natural to ask what level of impact the grid quantization has to network performance.

A priori, we expect that a finer quantization means higher compute requirements since the resulting feature maps will be larger but that should bring an increase in performance. The question though is the level of sensitivity in performance to these changes in grid quantization. If a model only works best at one particular grid quantization but drastically worse at all others this does not bode well for the model for use with/in various LiDAR scanners and environments. For example, with RAD, the authors show that the models performance degrades immediately at other grid sizes than 0.16m (they look at 0.12m and 0.2m). We can look at the model performance over a range of grid sizes, from the original 0.16 to a quarter the grid resolution at 0.32m, for our PPslim model. From Table 4.8 we can see that the performance shows little impact even with extreme reduction in grid resolution unlike RAD. In fact for the 3D challenge and hard BEV challenge, our model at a quarter resolution still exhibits higher performance than the original RAD model at full resolution and runs faster at the highly restricted 15W setting than RAD does at the maximum Jetson power setting. Note that these results are from a few training sessions per grid resolution (to confirm the result was consistent) & is not exhaustive. It is likely possible to achieve higher performance at each resolution through a more exhaustive fine-tuning process (especially for the BEV results), however such an investigation would likely result in overfitting to the task. These runs are enough to illustrate a general pattern, although it is clear the performance does slowly degrade as the grid resolution becomes too coarse.

Method	Grid (m)	GFLOP	Params (M)	Best Enc.	Car 2D		Car BEV		Car 3D	
					mod	hard	mod	hard	mod	hard
FA3D	0.1	/	/	/	/	/	88.1	85.49	75.47	71.97
RAD	0.16	/	/	/	91.94	89.34	88.21	85.70	75.80	72.65
RAD	0.2	/	/	/	88.94	88.23	86.14	84.43	73.54	69.54
PPslim	0.22	5.5	0.4	(16,32),64	89.51	88.18	87.74	86.56	77.5	75.24
PPslim	0.24	5.06	0.39	(16,32),64	89.1	87.81	87.01	86.51	77.01	74.72
PPslim	0.26	4.6	0.37	(16,32),64	89.15	87.64	87.52	86.69	76.82	74.62
PPslim	0.28	4.3	0.35	(16,32),64	88.9	87.31	86.76	86.17	76.45	74.4
PPslim	0.32	3.9	0.32	(16,32),64	88.85	87.45	86.75	86.34	76.47	74.36

Table 4.8: Comparison of FA3D and RAD versus PPslim AP performance at 0.7 IOU on the KITTI Car class while varying grid resolution. We highlight the complexity of the resulting PPslim model and its parameter count. Also included are the best encoder (Best Enc.) used for this performance level, where the first number is the FC unit count before the encoder max-pooling and the second number is the count afterwards as described in Figure 4.3.

We notice that in the analysis for both RAD and FA3D, the authors focus excessively on 2 topics that in practice we argue doesn't make sense when evaluating the best possible performance and, in fact, will severely mask such performance. This is the focus on using a lower IOU threshold of 0.5 for Car, and the second is using an averaged F1 score over each of the 3 difficulties. While a looser IOU threshold can be used to get an idea of model performance, the resulting AP of good models is in the low to mid 90%. Even models that exhibit a large performance increase relative to PointPillars suddenly appear near identical to it when considered at the far easier IOU. The same applies for F1 score, where the score on the easy challenge is generally in the high 80s/low 90s for 3D and in the mid to high 90s for BEV and 2D. That means models performing poorly in the 3D challenge suddenly appear better in their average F1 score. We argue that results using the lower IOU threshold or average F1 score are almost never reported by authors for this reason; they're not very informative on top-end model performance (for comparison) and simply make models with lower performance (relative to state of the art) seem more competitive. For completeness however, we do also present F1 score for the moderate 3D challenge on the KITTI Car class in Section 4.4.3.2. Furthermore, we now elaborate more on the distinction between 0.5 and 0.7 IOU in practical terms.

Consider two 3D bounding boxes B_{GT} and B_{pred} that have identical size LxHxW and therefore identical magnitude volumes V_{GT} and V_{pred}. Using the KITTI camera frame, the coordinates corresponding to each size dimension are X, Y and Z based on Figure 2.23. With the boxes exactly aligned, we move the prediction box along one axis (the +X axis) by an increment Δx as shown in Figure 4.4. The combined length of both boxes along the +X axis has increased from L to $L + \Delta x$, while the length along this axis that is the intersection has shrunk from L to $L - \Delta x$. The IOU of the 3D boxes can be written as the ratio of the intersection of their volume divided by its union as in Equation (4.11).

$$IOU(B_{GT}, B_{pred}) = \frac{V_{GT} \cap V_{pred}}{V_{GT} \cup V_{pred}} = \frac{(L - \Delta x) * H * W}{(L + \Delta x) * H * W} \tag{4.11}$$

For an IOU of 0.5, that means $\Delta x = \frac{L}{3}$. So a 0.5 IOU allows for an error of up to $\frac{1}{3}$ of the length of the class assuming the deviation is concentrated only in 1 dimension. This follows for the other 2 dimensions giving a 0.5 IOU error range of $(\Delta x, \Delta y, \Delta z) = (\pm\frac{L}{3}, \pm\frac{H}{3}, \pm\frac{W}{3})$ along each separate dimension. At a stricter 0.7 IOU, these bounds become $(\Delta x, \Delta y, \Delta z) = (\pm\frac{L}{5}, \pm\frac{H}{5}, \pm\frac{W}{5})$. Using our averages from Table 4.1, we can summarize the ranges as shown in Table 4.9. It is important to note that this analysis is regarding an error in one dimension only. When two or three are involved in the error, the error distance in each dimension will be reduced. We can see that using the lower 0.5 IOU threshold would give an average error

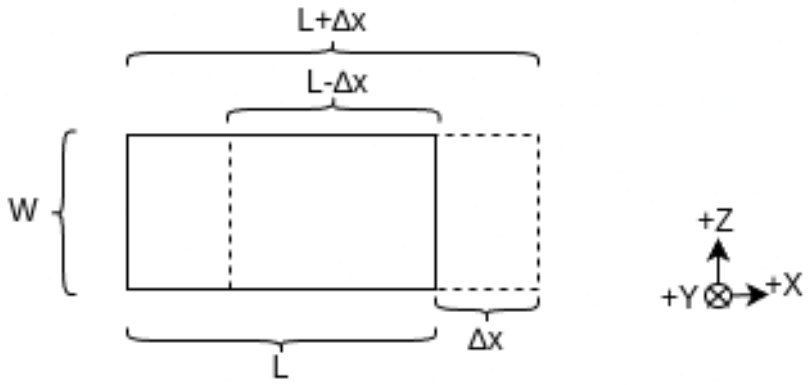

Figure 4.4: BEV of a 3D bounding box (solid) with dimensions LHW and an equally sized prediction box (hatched) translated by Δx with intersection and union extents along the length axis annotated (3D camera frame axes marked)

as high as 1.29 meters along the vehicle length axis. Depending on the application this can have a significant impact. For example with Automatic Emergency Braking (AEB), every centimetre means additional braking time that can mean the difference between near-miss, damage or death. In the case of parking assistance, even a 0.5m distance can mean the difference between inconvenience and property damage. Even in the context of lane keeping, where North American and EU lane widths typically range from 2.5 to 3.7m (8 to 12ft) depending on urban, motorway, construction zones & speed reduction zones (such as schools) or others environments [113], the car to car distance with narrow lanes can be as low as 0.5m assuming average exactly centered cars. Contrary to accepted traffic engineering practices, in the US at the national level, it has recently been reported in [42] (supporting earlier studies [84]) that the number of crashes does not significantly change with a reduction of lane widths from 10 or 11ft down to 9ft (2.74 meters). The conclusions drawn from this report were that narrow travel lanes in cities may be safer due to a heightened sense of alert by drivers operating on narrower lanes. However this report did not account or control for autonomous vehicles or driver assistance features that have been implemented in the last decade to enhance driver awareness. Especially at high speeds or adverse weather conditions, an incorrect perception of lane intrusion leading to avoidance or braking could have serious repercussions on those involved.

Class	IOU	$\pm\Delta x$ (Length)	$\pm\Delta y$ (Height)	$\pm\Delta z$ (Width)
Car	0.5	1.29	0.51	0.54
Cyclist	0.5	0.59	0.58	0.2
Pedestrian	0.5	0.28	0.59	0.22
Car	0.7	0.776	0.306	0.326

Table 4.9: KITTI average per class allowable distance error in each dimension at 0.5 & 0.7 IOU (0.7 only for Car)

In the case of model scaling, we note that with KITTI we are dealing with 90 degree FOVs in front of the vehicle. When moving to 360 degree FOVs surrounding the entire vehicle, the biggest increase in computational complexity from the 4x extra occupied voxels will be the backbone and not the encoder which at worst will only increase in delay by a few milliseconds. Realistically, our 3D volume bounds are quite extensive and practically our model would only require approximately doubling feature map sizes since we already process close to 50 meters left/right. So we would need to open up the feature map from 0 to 70 meters in the forward direction to -70 to 70 meters. Such a move will necessarily require a significant complexity decrease in the model, the simplest of which is a grid reduction that we can show could be handled by our model (not so for FA3D or RAD). The caveat would be when a LiDAR scanner with a drastically lower point density is used, but in such a case one could argue image fusion may be necessary to get good estimates at a reasonable distance -or- since we have the capability to slightly increase our encoder complexity, introduce more of the PointNet encoder into our encoder to perform a finer encoding of the input pillars.

4.4.3.1 Feature Analysis

The presence of both regression and classification at the output and loss computation raises the question of which has the biggest impact on the model performance. Both are computed using separate loss terms and, in the loss function, what works well involves weighting them before adding together using a 2:1 factor, where regression/localization loss is weighed more heavily. But which has a bigger impact on final performance? We opt to perform this analysis in a greedy fashion, where each of 8 features predicted by the model (class probability, XYZ position, LWH box size and yaw angle) can be replaced individually & systematically with the actual ground truth, to see the impact. After finding the highest impacting feature, this feature can be replaced in our predictions and

the processes repeated with the remaining features until none are left. We start with looking at fixing classification.

We start by probing the classification head. Consider a model applied to an input frame that produces predictions. The predictions will be true positives (TPs), false positives (FPs) and false negatives (FNs) as computed by the KITTI evaluation process at a given IOU. Predictions will be those selected for a threshold closest in distance to the 'optimal' solution on the Average Precision plot. By using the true labels for the model predictions, we end up adding false negatives to the predictions and false positives are removed from consideration. We can see the result in Table 4.10 where the performance increases slightly but not by a large margin. The remainder of the difference can be made up by predicting the regression parameters better. But which ones? A priori, since our model operates in the BEV, we expect the vertical direction in which we have no subdivision to yield the largest improvement (as well as object height).

Method	Car BEV		Car 3D	
	mod	hard	mod	hard
Original	88.31	86.96	77.65	75.56
GT Cls	91.4	89.99	80.75	79.41

Table 4.10: Performance improvement when replacing class probability predictions with ground truths for Car class using the standard 0.7 IOU

If we apply the described greedy analysis, we can indeed see that our guess was correct. The biggest regression gains come from the LiDAR z (vertical) direction and the box height. This helps to explain why the model does do slightly worse than competing models on the 2D challenge that relies on an accurate 3D box height to compute the 2D bounding box projection. The slightly elevated yaw error is expected relative to the x,y,l & w parameter since we expect them to be predicted quite well in the BEV. These results also help to explain the observations from Section 4.4.2 where we noted that an increase in loss scaling factor to the vertical box position, bounding height and yaw gave an increase in performance by forcing the model to, in a sense, pay more attention to the error in those predictions.

We show in the next chapter how we can leverage this analysis with insights from feature fusion to improve the performance of the PPslim model.

Iter	0	1	2	3	4	5	6	7
Param	cls	z	h	yaw	x	y	l	w
Delta	+3.1	+8.3	+6.1	+2.5	+1.1	+0.8	+0.25	+0.2

Table 4.11: Incremental AP improvements based on greedy analysis with successively increased number of parameters replaced with ground truth. Starting AP of 77.65 on Car 3D and 0.7 IOU. Coordinate axes are in the KITTI LiDAR frame.

4.4.3.2 Range Analysis

When comparing PPslim to PointPillars*, we see that the detection performance is quite similar in summary (in terms of AP). However in practical terms, this summary number does not tell us how either model performs across different conditions. Here, condition can mean observed object orientation angles or, more critically, in terms of distances to the object. In the case of objects in the Car class, this could represent immediate danger of car-to-car collisions for short distances or possible danger for intermediate distances. In the case of Cyclists, this potential danger would be even more immediate (at short distances) due to vehicles posing an even greater risk to cyclists (VRUs).

For our analysis, we can start by computing the components of precision/recall (TPs, FPs and FNs) assuming the 'best' threshold is used for our box predictions with each model. This threshold can be selected based on the point on the AP curve closest to the optimal classifier (perfect precision and recall) on our precision/recall curve, where the curve is already computed to obtain a summary AP estimate of performance. In our case we can further group each of the components as a function of distance (meters) and shown in Figure 4.5. We expect model performance to decrease with increasing distance since less points will be seen by a LiDAR return for objects farther away. Indeed this is the pattern we observe, although the graphs for both models look very similar. It's important to note that in Figure 4.5 (and all similar subsequent figures), the cumulative counts of TP's and FNs will not match the class instance counts listed in Table 5.1. This is because each 'difficulty' filters for a specific subset of samples from the validation set for the purpose of evaluation.

To compare how similar the graphs are, we summarize the performance per distance bin using a harmonic mean of precision and recall (F1 score). In the absence of additional information regarding the penalty of type I (false positive) and type II (false negative) errors, the F1 score gives an equal weighting for both recall and precision. Additional information could include, for example, mandating a higher cost of FN's versus FPs, where we want to miss less positives while being able to tolerate some incorrect positive predictions

(for example when a more vulnerable road user such as a Cyclist is present). This could be done with a different F-score to take this difference into account. In our case, the result is an F1 score per distance bin as summarized in Table 4.12. We take advantage of the fact that to compute F1 score, we don't need to compute precision or recall directly (which may otherwise be undefined), instead relying on Equation (4.12).

$$F1_{score} = \frac{2 * TP}{2 * TP + FP + FN} \tag{4.12}$$

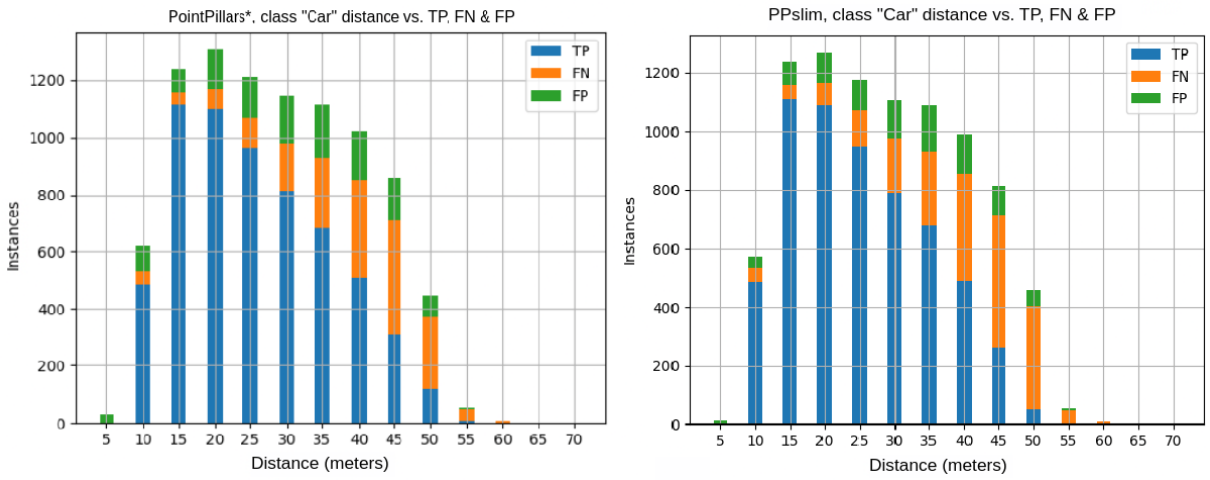

Figure 4.5: For the "Car" class, using a 0.7 IOU at the moderate difficulty, (left) the PointPillars* model and (right) the PPslim model, summarizing TP, FN and FP counts as a function of distance (meters). Distance bucket labels indicate (non-inclusive) end-points of each bucket, so the first bucket is [0,5) meters, the second is [5,10) meters and so forth. A distance of up to 70m is shown since the 3D volume bounds for the point cloud extends out to that distance.

Distance (m)	$F1_{score}$ PPslim	$F1_{score}$ PointPillars*
5	0.0	0.0
10	91.9	87.5
15	94.6	94.7
20	92.6	91.5
25	89.1	88.5
30	83.2	82.9
35	76.6	75.7
40	66.1	66.5
45	48.6	52.9
50	20.0	42.2
55	0.0	21.8
60	0.0	0.0
65	0.0	0.0
70	0.0	0.0

Table 4.12: Comparison of F1 score between PPslim and PointPillars* as a function of distance based on Figure 4.5

We see that the F1 scores are quite similar across distances up to 45 meters. The scores quickly drop well below 85% once we go beyond 30 meters, which is the score for our best performing (but much slower) competitor SA-SSD as shown in Table 4.13. To obtain a better understanding of samples at this range, we can plot histograms of ground truth point counts within each of the distance buckets shown in Figure 4.5, as shown in Figure 4.6.

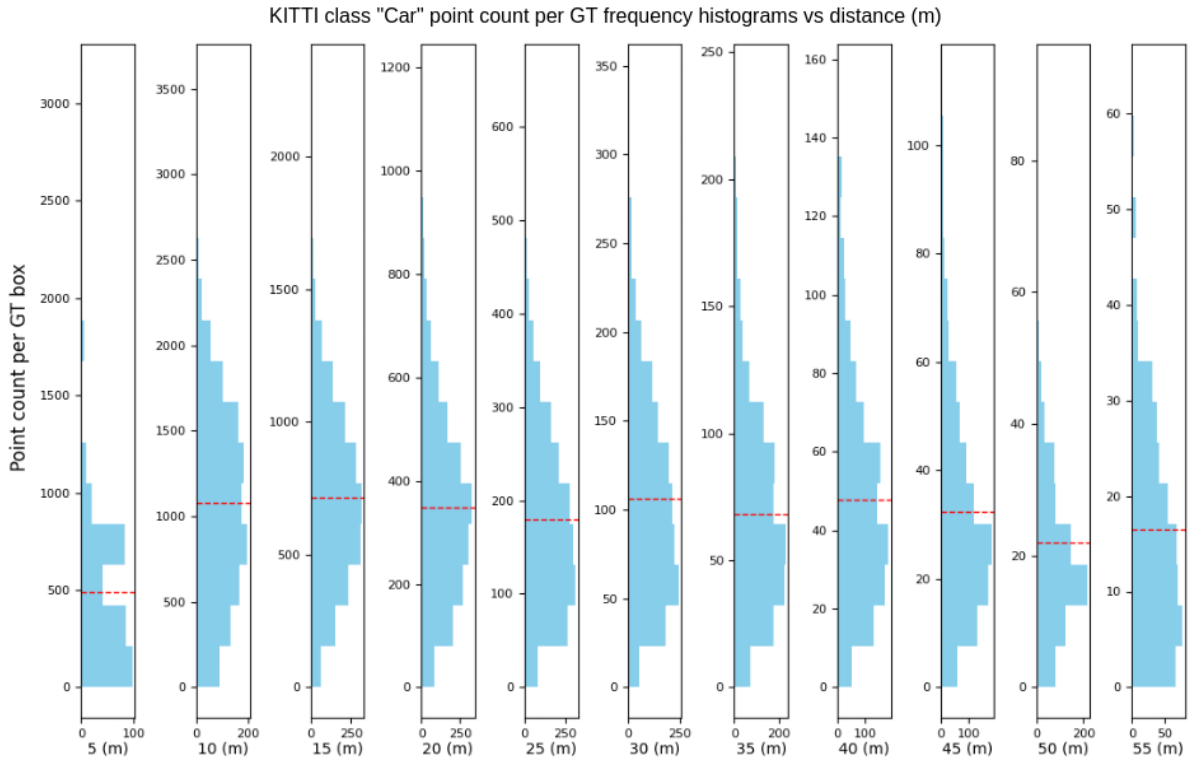

Figure 4.6: Plot of GT point count histograms for each distance bucket shown in Figure 4.5 (cut off above 50m for display clarity) for the 'Car' class. The main horizontal axis is each distance bucket and for each subplot is frequency/count of numbers of samples. The vertical axis is the number of points per GT box sample at the given distance range/bucket. The red dashed lines show the mean number of points for the GT samples in that bucket. A fixed number of bins (15) is used for all histograms.

Between 40-50 points per GT sample, the performance of both models rolls off drastically. This is an upper bound since points per box capture ground points under and in the immediate vicinity of each 3D box. If we use the segmentation masks described in Chapter 5, we can further filter the cloud points per box as only those belonging to the Car class as shown in Figure 4.7. This shows about 30-40 points per sample are where both models begin to drastically drop off in performance. If we compare this to the performance distribution for the Cyclist class, as shown in Figure 4.8 and Figure 4.9, we see the same roll-off in performance but around a closer distance of about 35m due to the smaller object size for Cyclist (relative to Car), at which point on average 38 points are seen per sample (would be lower with a segmentation mask). There is an oddity in the Cyclist data

between [20,25) meters in Figure 4.8, where there is a spike of false negatives that proportionately is higher than the other range buckets (a spike of 30-40 FNs). To answer why, we start by seeing in Figure 4.9 (the 25m bucket) a corresponding spike of 35 instances of GT boxes that have about 35 points/sample. This spike of instances is in stark contrast to the distribution of the remaining samples in this range bucket (other samples at this range more typically have between 60-80 points per 3D box). We know our performance drops significantly when only 30-40 points are available per sample and the spike of FN causing samples is approximately 35 points/sample (if we had a mask this would likely filter down to 30 or less points/sample). Indeed, the anomaly occurs precisely because in a particular sequence, the reference vehicle was stopped at an intersection and sampled a heavily occluded cyclist multiple times as shown in Figure 4.10. The other objects in the scene cycle through the FOV quickly, but this particular object remains static. This affect would be less noticeable with a larger sample size of Cyclists whereas the size of the 'val' subset for the Cyclist class is only 893. The result is observed with both PPslim and PointPillars*.

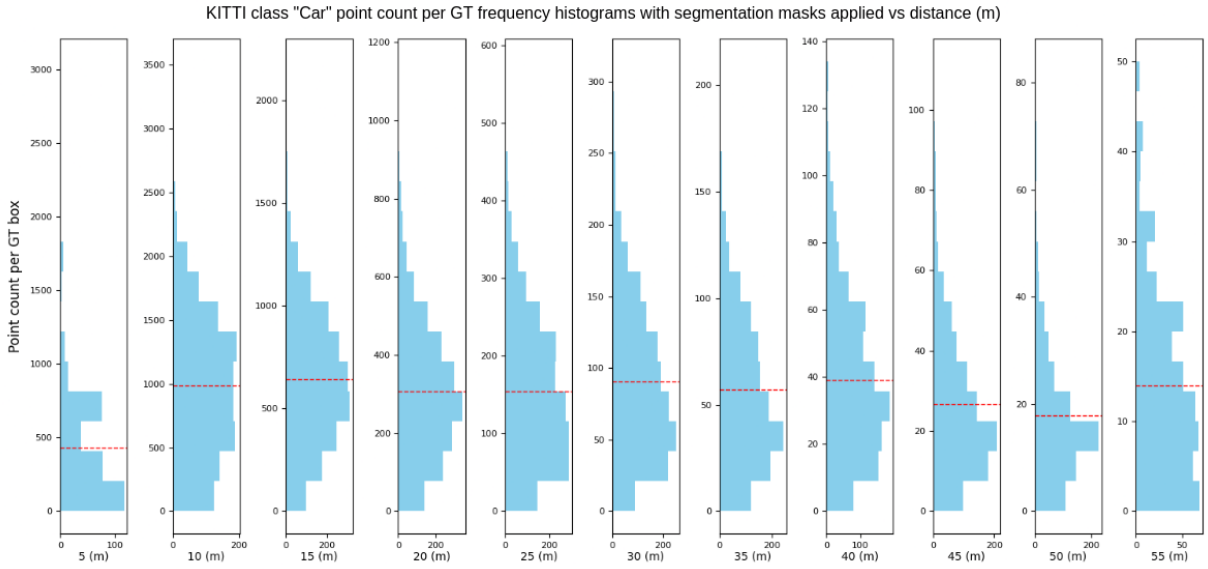

Figure 4.7: Plot of GT point count histograms for each distance bucket shown in Figure 4.7 (cut off above 55m for display clarity), for the 'Car' class, including post filtering to 3D points that fall within the respective segmentation mask. The main horizontal axis is each distance bucket and for each subplot is frequency/count of numbers of samples. The vertical axis is the number of points per GT box sample at the given distance range/bucket. The red dashed lines show the mean number of points for the GT samples in that bucket.

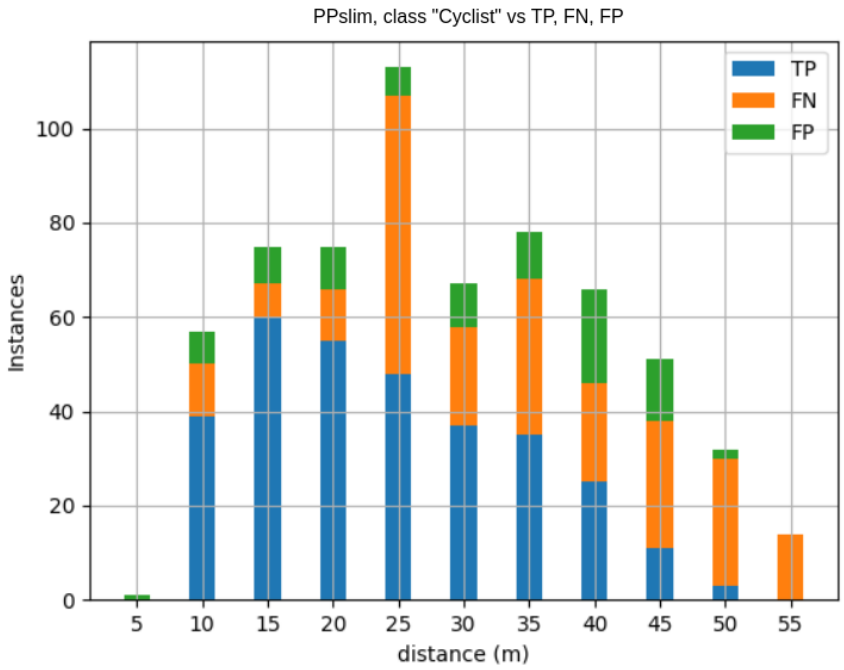

Figure 4.8: For the "Cyclist" class, using a 0.5 IOU at the moderate difficulty with a PPslim model, summarizing TP, FN and FP counts as a function of distance (meters). Distance bucket labels indicate (non-inclusive) end-points of each bucket, where the first bucket is [0,5) meters, the second is [5,10) meters and so forth.

Returning to the 'Car' class, if we compute the differences in F1 score between PPslim and PointPillars* at each distance, we can see they're quite similar in terms of which model "wins" in each bucket. At farther distances (above 40 meters) PointPillars* does have a clear advantage, however at these distances its performance has already dropped significantly in an absolute sense (53%), due to the low point cloud density of ground truth objects.

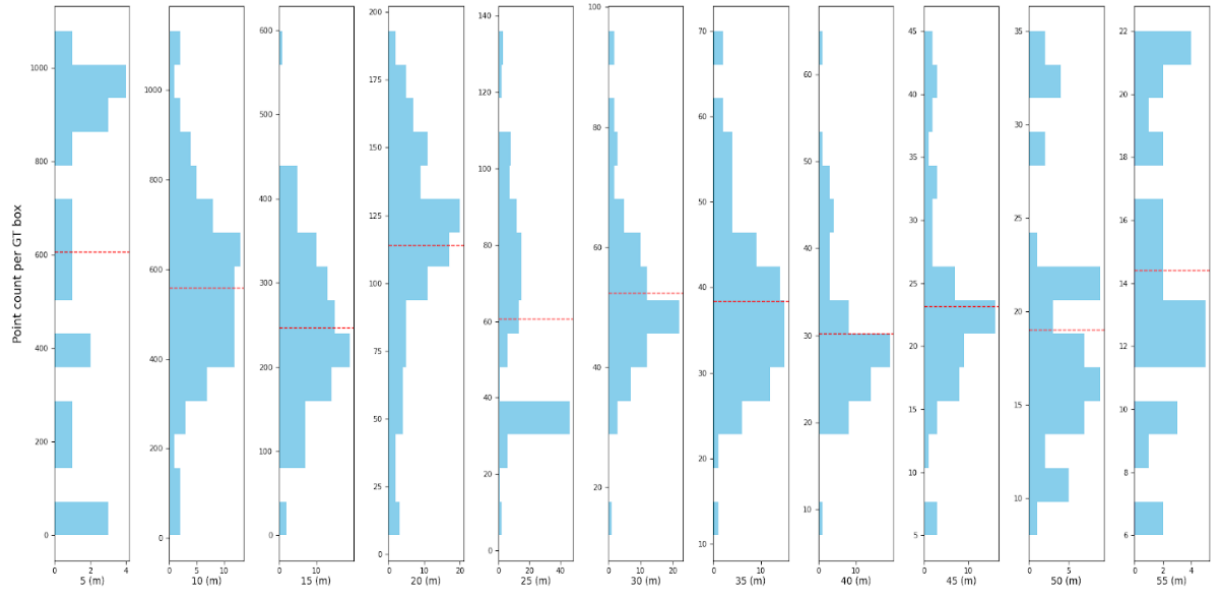

Figure 4.9: Plot of GT point count histograms for each distance bucket shown in Figure 4.8 (cut off above 55m for display clarity) for the 'Cyclist' class. The main horizontal axis is each distance bucket and for each subplot is frequency/count of numbers of samples. The vertical axis is the number of points per GT box sample at the given distance range/bucket. The red dashed lines show the mean number of points for the GT samples in that bucket.

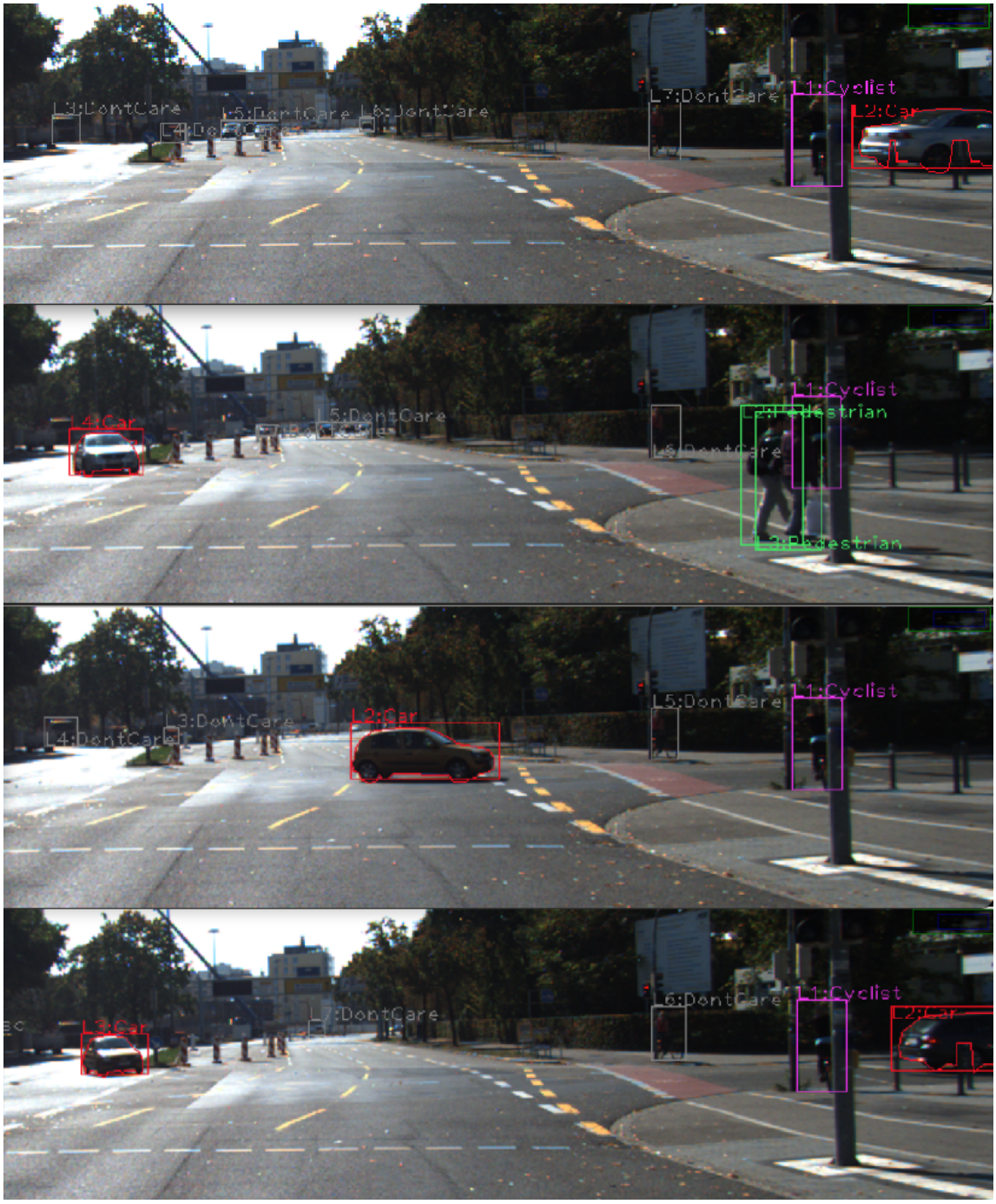

Figure 4.10: Heavily occluded static cyclist example with an average 33 points per box within a range bucket of [20, 25) meters (images from [37]).

Model	$F1_{score}$
SA-SSD	85.29
Point-RCNN	82.91
PointPillars*	81.1
PPslim	80.35
RAD	78.79

Table 4.13: Summary of maximum F1 scores for models on the KITTI Car class (3D moderate challenge @ 0.7 IOU)

4.5 Conclusion

We showed in this chapter the original PointPillars model can be significantly slimmed in computational complexity and parameters (over an order of magnitude) while retaining almost the same level of performance on the KITTI dataset. More importantly our slimmed model, PPslim, still outperforms our most similar, immediate, competitors and is drastically faster than they are on our target platform (an embedded system). Through an analysis, we showed how to train such models and identified key areas for future improvement.

Chapter 5

Augmenting KITTI: Semantic Segmentation and Ground Plane Feature Fusion

The baseline PointPillars model and the modified PPslim model described in Chapter 4 perform well on KITTI, however it is natural to ask what the simplest network modification is to obtain even better performance. The natural way to do this is to incorporate extra information from the scene as a more detailed description of each LiDAR point. This can be done by leveraging the 2D information available for each scene, which has a simultaneous image captured from a camera. This chapter looks at the way in which a fusion of such new information could be achieved within the PointPillars architecture, focusing on the shortcomings of the existing PointPainting extension to PointPillars and looks at a better (more performant) way to achieve the same fusion of information. To support this analysis, detailed semantic segmentations are added for the KITTI Car class (as a primary class of focus of experiments) and also for the Truck class. These segmentations are used to demonstrate a different semantic segmentation fusion approach to PointPainting, which is in a way that allows a large increase in detection performance. The proposed fusion approach is then used to implement an extension of PPslim from Chapter 4, named PPslimg, with ground plane feature fusion instead of image or segmentation features. The resulting model adds little computational impact to PPslim but improves its performance relative to PPslim and the competing PointPainting model which relies on the expensive computation of semantic segmentation.

5.1 Motivation

The KITTI dataset has been around since 2012 and some extensions & additions have been contributed to the parts of dataset that involve the KITTI leaderboard challenges (for example 3D detection, depth estimation, optical flow) including ground planes and depth maps. However, to the best of our knowledge no detailed semantic segmentations were available publicly at the time we completed our semantic segmentations. Since then, a dataset has been released publicly by another researcher group [49].

While dated and, now, superseded by larger datasets such as NuScenes and Waymo, the KITTI dataset can still be useful as a pre-training dataset for models that need to be trained on these larger datasets (much like pre-training on CIFAR [60] is still useful for training on ImageNet). In addition, it also provides a dataset that is much more accessible for researchers who are limited in their available compute capability (for example without access to high-end GPUs or cloud compute). There is an extensive codebase around the KITTI dataset and a history of active research (especially close to a decade of published results and an active leaderboard online) which means even today, new models or those aiming for state of the art will publish their models performance on KITTI (both validation and test sets). Therefore the dataset is still relevant today and, with segmentation data missing from the available information, this endeavour seems quite relevant for aiding future research. The resulting segmentations are available for download with example Python code to read the annotation data [108].

5.2 KITTI Semantic Segmentations

As described in Section 2.5.2, KITTI has 7481 training and 7518 test frames. The goal of our work was to segment the 7481 training frames. Since KITTI is still a relevant dataset today, we were motivated to annotate the largest class in this dataset (Car) as well as one additional class (Truck). The size of the task can be seen in Table 5.1, which summarizes the number of individual instances of each class in the dataset (broken down based on membership in the training or validation subsets of the full training set). The reason to annotate the extra additional class was related to sampling augmentation experiments that required using larger objects as occluders for smaller objects. The result for the overall 3D detection dataset (train and test) are 80256 labelled objects across 9 classes. A breakdown is shown in Table 5.1 for the training subset. The 'useful' samples are those that omit the DontCare or Misc classes (DontCare in particular are bounding boxes masking whole 2D regions that should not be considered for evaluation).

Class	Instances	Train Instances	Validation Instances
Car	28742	14357	14385
Pedestrian	4487	2207	2280
Cyclist	1627	734	893
Van	2914	1297	1617
Truck	1094	488	606
Tram	511	224	287
Person_sitting	222	56	166
DontCare	11295	5399	5896
Misc	973	337	636
Total	51865	25099	26766
Total Useful	39597	19363	20234

Table 5.1: KITTI class breakdown on the training set for the 'train' and 'val' subsets

To perform the annotation, we leverage Computer Vision Annotation Tool (CVAT) [18]. CVAT was originally developed by Intel but is now free and open-source. This tool supports various types of annotations (2D as well as 3D point cloud labelling) where the 2D labelling includes bounding boxes support for semantic segmentation. The tool has a web-based interface as shown in Figure 5.1, where imported batches of examples (a batch can be, for example, the frames of a single KITTI sequence that are also part of the 3D Object Detection challenge) can be shown as tasks to complete (one task for all the detection frames of a single sequence), with information on numbers of annotated & remaining frames, which user owns which task and the ability to mark completion of tasks. While not strictly necessary, having a tool to organize and mark which tasks are complete and which are remaining help speed up the annotation process.

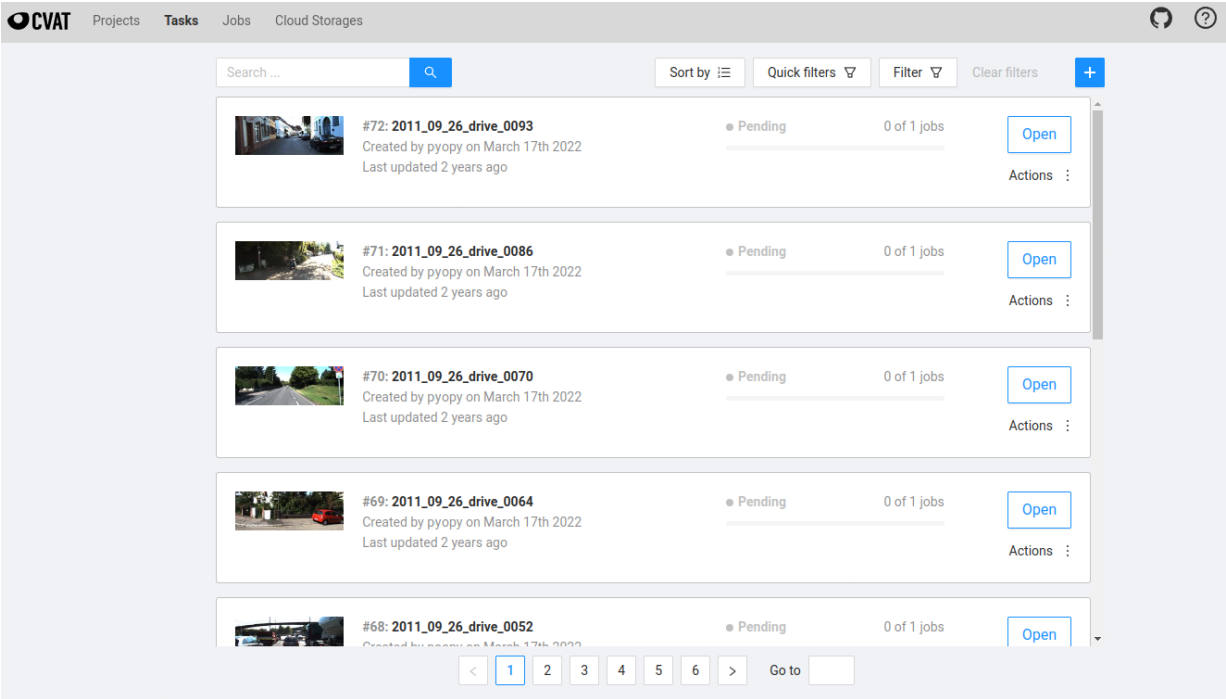

Figure 5.1: Example of CVAT 'tasks' list where each task is the annotation of all 3D Object Detection related frames from a particular KITTI sequence

When annotating each frame for segmentation, the process involves selecting point-by-point a polygon that encompasses the boundary of each object of interest in the scene (specifically the 'Car' and 'Truck' classes) as shown in Figure 5.2. A separate viewer tool was written to show KITTI ground truth boxes and object information to ensure the correct objects were then annotated in CVAT.

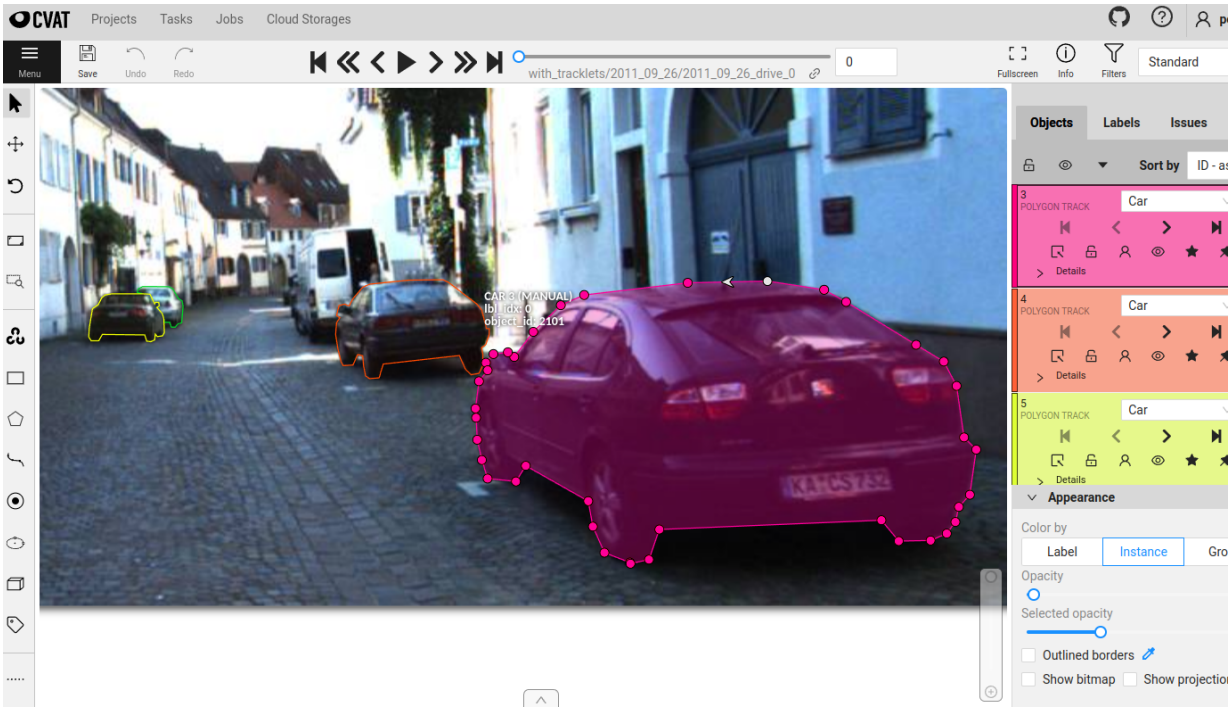

Figure 5.2: Example polygons of vehicles from the 'Car' class for a KITTI frame (image from [37])

An additional feature added to CVAT by this work was to add additional OpenCV image pre-processing options. The original KITTI frame when loaded through the web renderer showed each image as darker than it appears making it much harder to annotate the underbody of vehicles or areas with low lighting. Since OpenCV has useful image transforms (or the ability to combine several together), added a custom pre-processor for frames made the annotation of some objects more accurate. For each frame, every car having a ground truth bounding box was given a tight segmentation boundary, given an object identifier (ID) which is consistent across frames in the KITTI dataset as well as a label identifier that corresponds to the 0-based index of the object in the KITTI label file for the associated frame. Label files are how KITTI annotations are stored for each training frame, with each row giving the annotation for one object in the frame/scene including its 2D and 3D bounding box. Since the object IDs persist across frames for the same Car, these annotations can be used for evaluating object tracking algorithms in addition to segmentation algorithms. Indeed, KITTI already provides what it calls tracklets for a limited number of frames in the training set. This limited number of frames comes from a

small number of sequences where every every object is given a unique, trackable ID across every frame in the sequence where it shows up. The information from those frames was used to cross-reference our own annotations and ensure complete consistency. In some cases, objects had multiple visible parts that meant multiple polygons had to span the object such as shown in Figure 5.3. In such a case the same label index and object ID would be entered for the parts, and in the exported results after the completion of segmentation the resulting data would just need to be post-processed to group the polygons (a list of points) into a list representing all regions for an object.

Figure 5.3: Example of a split segmentation where an object is spanned by multiple disconnected regions (image from [37])

Some examples of annotated frames are shown in Figure 5.4 (zoomed for clarity) with the segmentation boundary displayed for the car class. The annotation process took approximately 2 months to complete.

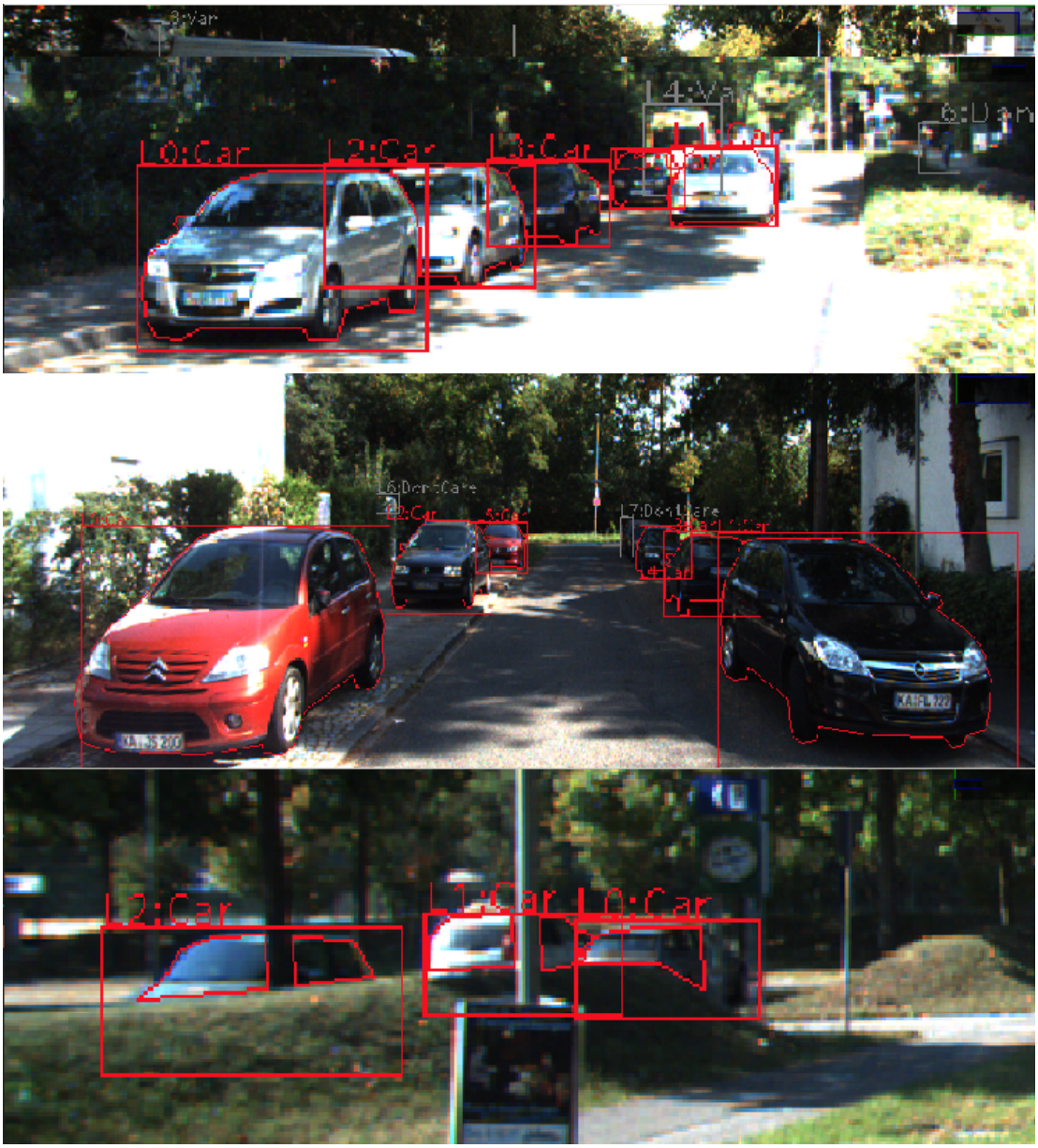

Figure 5.4: KITTI 'Car' class semantic segmentation annotation examples (red boundaries) along with ground truth bounding boxes (images from [37])

5.3 Image Feature Fusion

A proposed extension to the PointPillars model is to incorporate knowledge from the 2D images associated with each LiDAR frame. Vora et al. propose the PointPainting extension to PointPillars [110], that modifies the encoder with semantic segmentation of classes in the scene to help with detection. This relies on first applying a model to perform semantic segmentation of the scene so as to label every pixel in the 2D image of that scene. The LiDAR points can then be projected to the image and the projected LiDAR point can then be assigned the label at the integer coordinate image pixel it was projected to (or None if the segmentation yielded no labelling at that location). This approach requires obtaining segmentation information from a separate 2D segmentation stage (in their case pre-trained) adding significant computational burden to the network (especially when considering the possible computational complexity of such a segmentation model to the complexity of a PPslim model). Since both the segmentation and 3D detector must be run in sequence, the computational cost cannot be easily hidden. In their work, the resulting model shows no improvement for the Car class, a 5 AP improvement on the Pedestrian class and $<$1 AP improvement on Cyclist. Even when merging near ground truth segmentation information for NuScenes, the results suggested the mAP performance does not approach "perfect" performance. If we consider that this occurs even with the use of ground truth, then it suggests that the fusion of image features through the encoder may not be the best strategy for PointPillars.

Since we have the ground truth segmentation maps for the Car class, this allows us to train the model using the ground truth data as the input. The idea is to see how well the encoder learns in the presence of ground truth information. We expect that with this information, the performance should jump noticeably if the network is able to effectively use the information. It's worth emphasizing that this fusion is for analysis purposes only, as using the ground truth as an input to the model is not realistic.

There are 2 basic ways to perform the fusion. The first is to take the 2D grid in the BEV and to intersect it with the ground truth bounding boxes for that scene (both original and inserted boxes) as shown in Figure 5.5. This approach can be called "GT Box". Any 2D gridbox that intersects the GT box will have a constant C as an activation at that gridbox. For our experiments, we found that it is sufficient to keep activations in one feature map regardless of class and to use a value of 1.0 for the activation (0 elsewhere).

The alternate approach that can be called "GT Seg", which is similar but doesn't use ground truth boxes, relies only on the segmentation information so as to be more fair in comparison with PointPainting. This approach projects 3D LiDAR points to the

2D image and, for 2D points that fall into a ground truth segmentation mask, marks the corresponding feature map position for the pillar/grid box the 3D point came from as a 1.0 similar to Figure 5.5. The caveat here is that scene objects undergo random transformations and we don't know what the segmentation mask will look like in the image afterwards. Likewise for objects inserted into the scene. So instead we first project 3D LiDAR point to the image, and when a point falls within a mask we instead mark the 3D point as being part of a segmentation mask for a class. Object clouds inserted from other scenes have their individual clouds pre-marked as being part of a mask (if it fell within one) in the scene the object came from. This allows us to apply arbitrary augmentations to the point cloud, including object insertions, and only after those augmentations are complete compute which grid boxes/pillars are occupied with marked cloud points. This strategy allows us to avoid complicated augmentation strategies where visual consistency in the corresponding 2D image has to be maintained while augmentations are performed to the 3D point cloud.

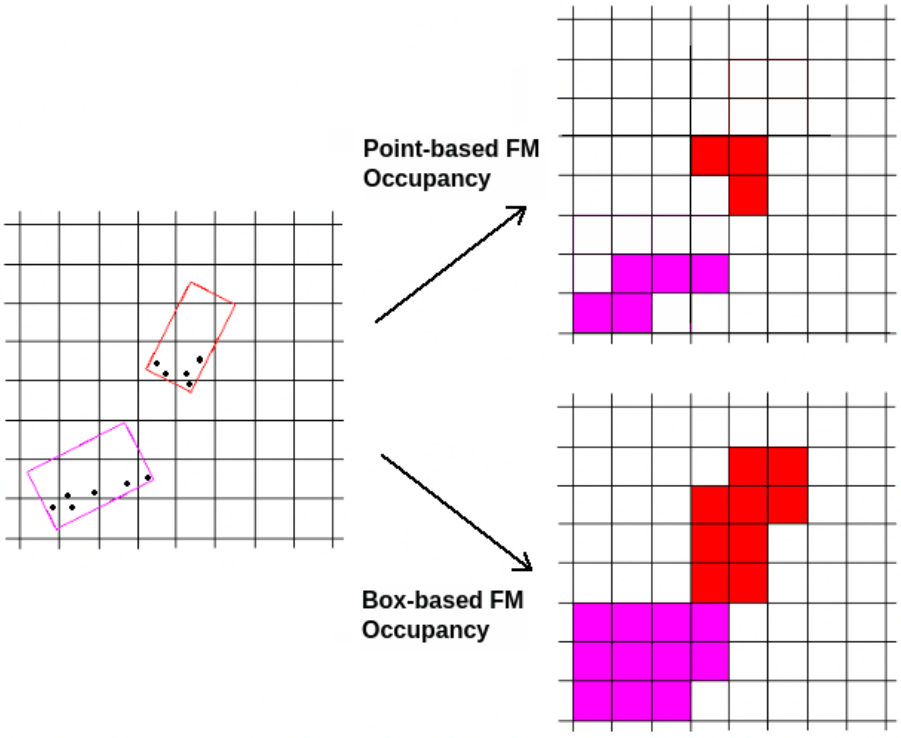

Figure 5.5: Converting KITTI GT bounding boxes to feature map activations

It's useful to define a dropout probability p_{drop} per object to get an idea of how much

information from a GT is needed to affect performance. Once the grid boxes occupied by an objects mask are computed, this probability is used to decide how many actual grid boxes to keep (or 'light up' in the spirit of Figure 5.5). Furthermore, the layer at which to fuse (D_{fuse}) the information is important since fusion at earlier layers should mean the extra information would contribute more to the final output. The fusion of extra feature maps can be achieved using concatenation of the new channels with the input or output feature maps at particular layer.

Starting with an original reference PointPillars model we can introduce the ground truth fusion for the same model, using different values of p_{drop} and D_{fuse}. The results are show in Table 5.2. We can see that even as we increase p_{drop} to high levels of dropout (over 95%), the model maintains high levels of performance. The first row of Table 5.2 shows the result for our implementation of PointPainting just with ground truth segmentation data. The remainder of the row is empty since we don't have ground truth segmentations for the other classes. For the remaining rows, the Car result columns use segmentation data while the result columns for the other 2 classes use ground truth box projections as described earlier in this section.

Segmentation Merge Method	p_{drop}	D_{fuse}	Car @ 0.7			Cyclist @ 0.5			Pedestrian @ 0.5		
			3D	BEV	2D	3D	BEV	2D	3D	BEV	2D
Predicted Seg Via Encoder (Authors)	/	/	76.77	87.65	88.73	64.18	68.76	74.29	66.15	72.41	61.1
None (Baseline PPslim)	/	/	77.5	87.74	89.51	64.39	67.15	71.1	54.1	62.88	57.9
GT Seg Via Encoder	/	/	78.24	89.46	90.26	/	/	/	/	/	/
GT Seg+GT Box Via FM Fusion	0.9	2	86.35	98.19	89.21	82.75	80.72	82.78	76.73	79.75	75.36
GT Seg+GT Box Via FM Fusion	0.9	7	78.1	89.33	89.85	74.57	75.9	76.97	68.02	78.02	65.43
GT Seg+GT Box Via FM Fusion	0.99	2	88.27	98.77	88.95	81.64	81.82	82.4	69.92	79.38	60.74

Table 5.2: Summary of performance at the moderate difficulty level across classes at a given IOU and for the corresponding challenge. The first row is the original PointPainting model. The second row is our PPslim model, with an unoptimized Pedestrian result. The third row is our result of fusing ground truth segmentation data through the PPslim encoder. The remaining rows show a combination of "GT Seg" (Car) and "GT Box" (Cyclist+Pedestrian) fusion and their results with various drop probabilities and fusion depths. All base models (except the author reported result for PPaint which uses a 0.16m grid) are PPslim with a coarser 0.22m grid.

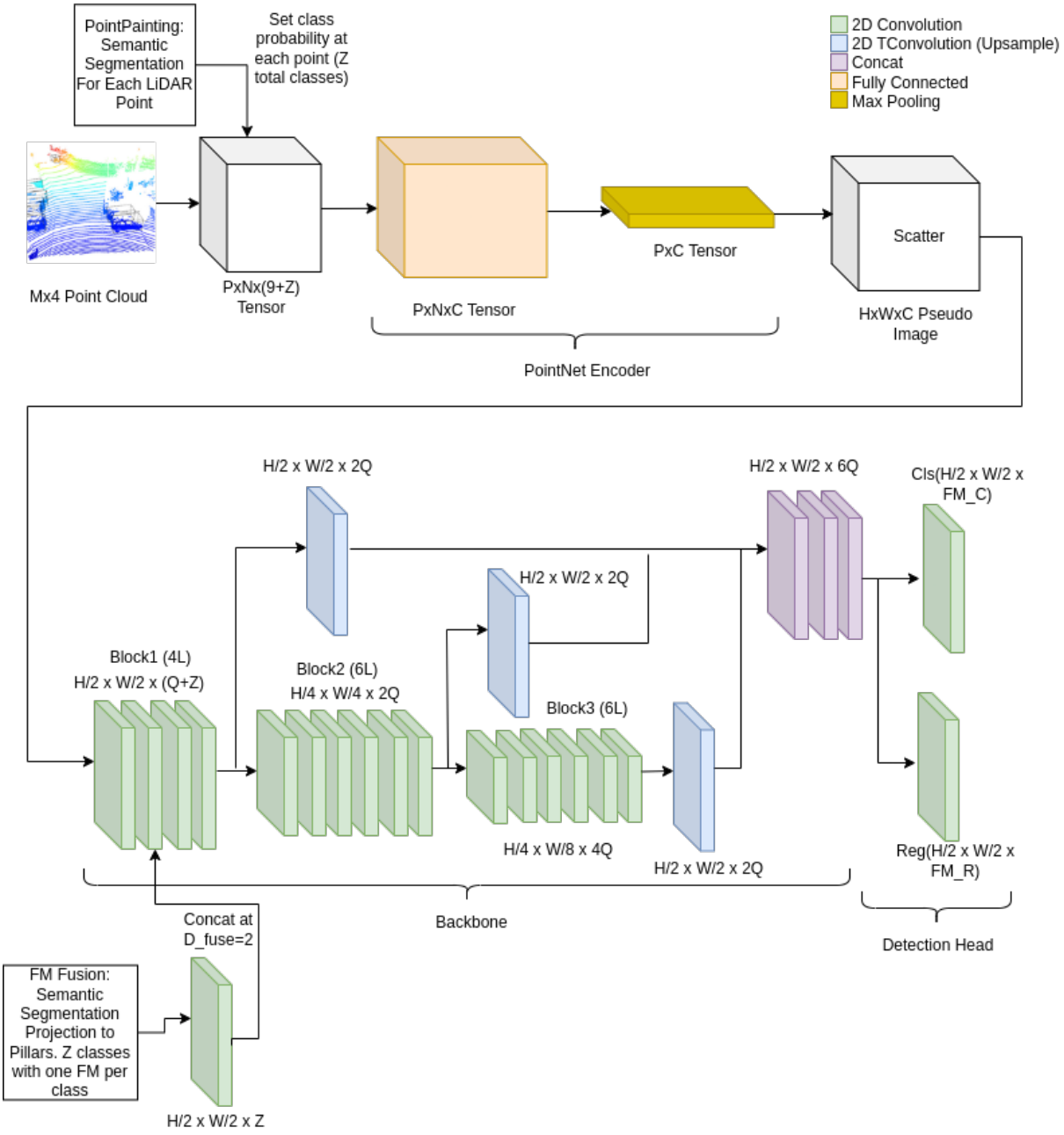

Figure 5.6: Visually comparing the fusion location between PointPainting and "FM Fusion".

We can modify the architecture diagram for the original PointPillars model in Figure 4.1 to the diagram shown in Figure 5.6. The PointPainting approach fuses the segmentation information at the beginning of the network by concatenating to the input tensor fed to the encoder. Our comparison approach, "FM Fusion", is shown in the figure as being after the encoder with the computed feature map activations (built as shown in Figure 5.5) concatenated at convolutional layer D_{fuse} from the start of the backbone portion of the network. To fuse deeper in the network (larger values of D_{fuse}), we just need to build a coarser grid or build a finer grid then down-sample before concatenating (we found building the fine grid and using stride 2 convolutions to down-sample works best).

We expect that the fusion described should improve the BEV performance of the model the most because we're essentially providing a strong cue of the location of object locations in the BEV. The 3D performance (and the 2D performance which largely depends on the 3D) should also increase but to a lesser extent. Indeed, this is exactly what we observe where even with an aggressive value for p_{drop} of 99% (using a small PPslim model), the performance increase is large relative to the original model. If we use the first described strategy of using GT boxes for the Pedestrian and Cyclist classes (shown in Figure 5.5) and use the segmentation mask strategy for the Car class, we obtain a similar jump in performance for the non-Car classes. These results support our hypothesis that this fusion strategy is preferred to fusing the same information at the encoder stage. Fusion of features deeper into the network yields smaller performance increases to the point where performance begins to revert back to the performance of the original model since the fusion is too late in the network and at too low a resolution to make an impact.

5.4 PPslimg: PPslim Augmentation Using Ground Plane Fusion

In Section 5.3, we showed how different fusion routes of feature information with the PointPillars model can make a difference in performance. If we consider the analysis from Section 4.4.3.1, we see that where the model performance can be most improved are the "vertical" predictors of the 3D boxes (box vertical position and box vertical height). To address this improvement, we propose providing additional cues to the model in the form of feature maps encoding vertical offset information (in the spirit of Section 5.3) that bypass the encoder stage and are fused directly with convolutional feature maps. But which vertical offset information? The encoder stage already uses the components of the average 3D position of pillar points as input features, so its possible those could be used as

separate inputs (one dimension per feature map). However we instead focus on estimating a feature that the network would already learn implicitly, which is the ground plane.

To generate an estimate for the ground plane, we can use a similar approach to Principal Component Analysis (PCA) to obtain the principal axes that capture the variance in the point cloud data for a pillar. Assume a pillar grid G having individual physical pillar dimensions CxCxE, where C is a scaled up value (by an integer) of the corresponding pillar grid dimension of the underlying network, to allow gathering more points for the ground plane approximation. For a 0.22m grid in the underlying network, a grid with C=0.66m works well for example. For a pillar grid position (i,j) in G, the cloud points falling within the pillar can be grouped as an Nx3 matrix $D_{i,j}$.

$$D_{i,j} = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \\ \vdots & \vdots & \vdots \\ x_{N,1} & x_{N,2} & x_{N,3} \end{bmatrix} \tag{5.1}$$

The mean position of the points is then $\bar{d}_{i,j} = \begin{bmatrix} \bar{x}_1 & \bar{x}_2 & \bar{x}_3 \end{bmatrix}$ which we can copy across rows as:

$$\bar{D}_{i,j} = \begin{bmatrix} \bar{x}_1 & \bar{x}_2 & \bar{x}_3 \\ \vdots & \vdots & \vdots \\ \bar{x}_1 & \bar{x}_2 & \bar{x}_3 \end{bmatrix} \tag{5.2}$$

We can then subtract both matrices to produce the matrix $M_{i,j} = D_{i,j} - \bar{D}_{i,j}$ on which we can apply the Singular Value Decomposition (SVD). This allows us to represent M as follows, where in our case U is Nx3, Σ is 3x3 and V^T is 3x3.

$$M_{i,j} = U_{i,j} \Sigma V_{i,j}^T \tag{5.3}$$

The mean centering is necessary since we want to perform the analysis relative to the center of the cloud rather than in the direction of it. Performing PCA would mean constructing the covariance matrix using M (specifically the construction of $\frac{1}{N-1} M^T M$) followed by an eigendecomposition. From applying SVD, the matrix Σ represents singular values (σ_1, σ_2 and σ_3 ordered from highest to lower) that are the square roots of the eigenvalues that would have been obtained for the covariance matrix and its eigendecomposition. The right singular vectors V (having columns V_1, V_2 and V_3) represent the principal

axes from PCA, and are mutually orthonormal vectors. For the purpose of ground plane estimation, our interest is mainly in the smallest singular value and its relationship with the other two along with its corresponding principal axis.

If the current grid square points indeed form a ground plane, the two principal axes from the SVD with the most variance should capture most of this cloud variation in a plane spanned by these axes while the remaining axis would be normal to this plane. The square of the corresponding singular value σ_3 (also $\sigma_{min}(M) = \sigma_3$, the minimum singular value for M) is then proportional to the variance of the points in this direction:

$$Variance = \frac{1}{N-1}\sigma_3^2 \tag{5.4}$$

For point clouds that are increasingly planar, M will be increasingly ill-conditioned. To decide whether the estimate is "good enough" as a plane estimate, we can perform a number of checks of the SVD result to decide whether to keep or ignore the estimate (other pre-checks such as requiring a minimum number of points, P, in the pillar to apply the SVD are also required, that we take as 7). We first check whether smallest singular value ($\sigma_{max}(M)$) is non-zero and check the condition number (Equation (5.5)) to ensure it is greater than a constant T_1. The condition number is the ratio of largest to smallest singular values for M, with larger values indicating a more ill-conditioned M. We also check the ratio of the second largest singular value σ_2 to σ_3 to ensure it is greater than a constant T_2. We use a value of 25 for the thresholds based on checks against the training set. Finally, we check the resulting vector V_3 (which is the normal vector) to ensure its dot product with the vertical vector in the LiDAR frame (Y=[0 0 1]) is within J degrees (a cone around the the vertical direction). Using a value of J=10 degrees is quite lenient in our experiments, where for KITTI only yaw angle is provided for ground truth or expected to be predicted. In practice, AASHTO regulations [82] in the US recommend maximum road slopes of 3 to 6% in urban and suburban environments (up to 10% in extreme cases for an angle of 6 degrees).

$$\kappa(M) = \frac{\sigma_{max}(M)}{\sigma_{min}(M)} \tag{5.5}$$

For candidate normals that remain after filtering, we apply nearest neighbors to spread its values among its 8-connected neighbors for F iterations (taken to be F=3). The value spread is the location of the original ground plane estimate so that the new square can compute its own ground plane. We want to spread the strictly filtered values to locations that may have objects to provide a ground plane estimate there. We can check if this

is occurring by applying our CxC grid G to all KITTI training frames and intersecting ground truth boxes with this grid. We can then measure on average (per frame) how many grid squares intersect with an object only (no ground plane estimate), how many intersect with only a ground plane estimate and how many have both an object and a ground plane estimate. The same check can be applied after the nearest neighbor check. We see from Table 5.3 that initially, on average, very few grid square positions contain ground plane estimates. After nearest neighbors, many more object positions now have a ground plane estimate "under them" and two sets of grid squares now exist; seed squares that started with a normal and neighbor squares that be populated from seed squares. We can visualize the results of this process in Figure 5.7.

Condition	Mean Grid Squares Object Occupied (per frame)	Mean Grid Squares GP Occupied (per frame)	Mean Grid Squares Object+GP Occupied
Pre-Nearest Neighbors	104 ± 79	273 ± 111	2 ± 1
Post-Nearest Neighbors (3 steps)	57 ± 52	1107 ± 379	50 ± 46

Table 5.3: For the KITTI training data, the number of 0.66m grid squares on average (mean ± standard deviation) that intersect with a ground truth box only, contain a ground plane estimate only or, intersect with an ground truth box and have a ground plane estimate. Both pre and post nearest neighbors is shown. Using 0.66m grid squares, minimum of 7 points and maximum of 16 points per voxel, 3 steps of 8-connected nearest neighbors, maximum ground plane angle to vertical of 10 degrees and maximum condition number of 25.

Figure 5.7: Visualization of "seed" grid squares (0.66m grid) for 3 scenes. The top row shows the scene. The middle row shows a point cloud and the initial seed squares (green) on the ground plane for each corresponding scene (above). The bottom row shows the locations where seed squares have spread after applying nearest neighbors (N=3).

Having spread the ground plane information, each "occupied" grid square will point back to the original seed square for its ground plane estimate $n_{i,j}$. The last step is to compute the actual ground plane at the original seed locations using the point mean $\bar{d}_{i,j}$ in those seed grid squares, then compute the ground plane heights at both seed and non seed squares. The plane normal $n_{i,j}$ must be orthogonal to a vector from any point on the plane x (which is in the LiDAR frame) and the mean grid square point $\bar{p}_{i,j}$:

$$n_{i,j}(x - \bar{p}_{i,j}) = 0 \tag{5.6}$$

This can be expanded to give the equation of the line:

$$n_1 x + n_2 y + n_3 z - n_1\bar{p}_1 - n_2\bar{p}_2 - n_3\bar{p}_3 = 0 \tag{5.7}$$

$$n_1 x + n_2 y + n_3 z + Q = 0 \tag{5.8}$$

$$Q = -(n_1\bar{p}_1 + n_2\bar{p}_2 + n_3\bar{p}_3) \tag{5.9}$$

The plane "height" within a grid square is taken to be the center of the 0.66m grid square in LiDAR coordinates. We can benchmark on the Jetson, the latency in computing this estimate takes at various power modes to see how it may impact PPslim inference speed. We can see from Table 5.4 there is a small impact due to the sparsity of the overall LiDAR data and, assuming we use a default of a maximum of 24 points per grid square to compute the SVD. If we compute this estimation in parallel with the voxelization and encoder stages, we can full amortize the computation time to remove any impact on the PPslim backbone. This is made possible since we are fusing this new information through the feature maps and not through the PPslim encoder.

Jetson Power Setting	Max Points Per Pillar	Average Time (ms)
MAXN	16 / 24 / 32	7.01 / 8.6 / 10.8
15W	16 / 24 / 32	13.5 / 14.1 / 17.9

Table 5.4: Comparison of Jetson AGX Xavier runtime per KITTI frame for computing all SVDs and final 8-connected nearest neighbor densification using different maximum point counts per 0.66m pillar (from 0.22m base pillar size)

When comparing the best observed PPslimg model to PPslim and PointPainting, we see that we generally exceed the detection performance of PPslim while outperforming PPaint in most difficulties and challenges for Car and Cyclist, as summarized in Table 5.5, without incurring the cost of the semantic segmentation extraction stage. We can further compute our summary F1 score for the 3D moderate challenge, as summarized in Table 5.6, and although we don't quite match PointPillars*, we do improve relative to PPslim.

Method	Cyclist 2D		Cyclist BEV		Cyclist 3D		Car 2D		Car BEV		Car 3D	
	mod	hard	mod	hard	mod	hard	mod	hard	mod	hard	mod	hard
RAD	71.27	68.23	65.53	61.79	62.03	58.28	**91.94**	**89.34**	88.21	85.70	75.8	72.65
FA3D	/	/	66.29	62.89	60.23	56.86	/	/	88.1	85.49	75.47	71.97
PP*	71.75	68.45	68.58	62.9	**65.6**	**61.16**	89.43	88.26	**88.31**	**86.96**	**77.65**	**75.56**
PPslim	71.1	69.14	67.15	62.88	64.39	59.76	89.51	88.18	87.74	86.56	77.5	75.24
PPaint	74.29	69.85	**68.76**	63.99	64.18	60.79	88.73	87.4	87.65	85.56	76.77	70.25
PPslimg	**75.05**	**70.75**	68.17	**64.89**	65.49	60.81	89.81	88.67	88.13	86.68	77.52	75.33

Table 5.5: Average Precision on the KITTI Car & Cyclist classes for each of the 3 challenges (2D, BEV and 3D) and 2 difficulties (moderate & hard). Include methods are our PPslimg method as compared to our PPslim model, PointPillars* (PP*) as our implementation of PointPillars and the PointPainting (PPaint) image fusion approach. Also shown are the competing RAD and FA3D methods. We see that PPslimg generally exceeds PPslim performance and exceeds the performance of PointPainting even with its advantage of fusing semantic segmentation information.

Model	$F1_{score}$
SA-SSD	85.29
Point-RCNN	82.91
PointPillars*	81.1
PPslimg	80.98
PPslim	80.35
RAD	78.79

Table 5.6: Extension table to Table 4.13. Summary of maximum F1 scores for models on the KITTI Car class with PPslimg included (3D moderate challenge @ 0.7 IOU)

5.5 Conclusion

In this chapter we described an extension to the KITTI 3D Detection dataset that provides fine semantic segmentation annotations for the Car and Truck classes. The resulting annotations are to be made publicly available for download. We use these new annotations to analyze an existing extension to the PointPillars model, PointPainting, that performs a fusion of semantic segmentation information to increase performance. We show that the model performance can be drastically improved using a different fusion strategy (with the same data) that bypasses the encoder stage and instead creates an occupancy grid for concatenating with backbone feature maps. Since the difference in computational complexity

between both approaches is negligible, our fusion approach should be preferred since in the limit of better semantic segmentations, the model will converge to the performance it achieves using ground truth data.

We leverage this realization to propose an extension to our modified PPslim model from Chapter 4, that takes localized ground plane estimations (and LiDAR reflectance values) and fuses them by bypassing the input encoder. This fusion is performed in the same way that was discovered to improve the fusion of semantic segmentation ground truths, relative to the PointPainting model. The resulting model, PPslimg, maintains the fast inference speed of PPslim while improving performance relative to it on both the Cyclist and Car classes while exceeding the performance reported on PointPainting in most comparison categories.

Chapter 6

Conclusion and Future Work

In this thesis, we explored the improvement in efficiency for 2D & 3D ConvNet object detection models for power restricted embedded platforms.

In Chapter 3 we propose using a heatmap-style approach to perform coarse pedestrian detection. This approach relies on a simplification of a popular ConvNet architecture, SqueezeNet, that is already built with a focus on parameter reduction. This extension uses partially connected activations in a weighted voting scheme for the location of pedestrians, resulting in a kind of heatmap of their location in the scene. In the process, we end up with a small 3.24MB model which can run in real-time at 30FPS on an automotive processor operating within a 2W envelope on an ADAS targeted embedded platform while showing that the evaluation process for our model puts it reasonably close to good models that can't run on the same hardware.

In Chapter 4, we extended the existing PointPillars 3D ConvNet detector by modifying its input encoder and remaining parts of the network that by analysis, showed little to no performance benefit on the KITTI dataset (while using valuable computation). The resulting model was named PPslim. The encoder optimization for PPslim relied on a squeeze and expand pattern, whose application was possible due to the max pooling operator used in the encoder to remove a large dimension from the input tensor to the encoder. The input to the network (pre max pooling) could then be squeezed in dimension while the post max pooling stage could be expanded in dimension. Following these complexity saving modifications and having retained the encoder instead of removing it, meant that we could decrease the resolution of the input grid to the network, allowing us to further reduce complexity of all parts of the network (including the SSD-based detection stage). By keeping the encoder, we showed the network maintains an excellent resilience to large changes

in the grid resolution and, is something directly competing modifications for PointPillars cannot do partly due to a misidentification of the networks weaknesses that we leverage as strengths. We found experimentally that our modifications mean we can run our model far faster than directly competing models. Experimentally it was also found that the resulting model needed to be trained for far longer, likely due to the drastic decrease in parameters.

In Chapter 5, we looked to further extend the PointPillars and PPslim models through the fusion of additional information. Our standard was the PointPainting model, which performs a computationally expensive, sequentially executed with the 3D detector stage, semantic segmentation of image corresponding to a 3D point cloud of a scene. To facilitate our analysis, we first annotate the Car class of the KITTI dataset with detailed semantic segmentations (along with an additional class). However, our approach does not fuse 2D image features (in the form of semantic segmentations) during the encoder stage of the PointPillars model. Instead, we compute a projection of LiDAR to the 2D segmentation mask and use the knowledge of the resulting points spatial locations to mark a corresponding feature map in the BEV with an indication of object presence. This feature map was then fused with the convolutional layers of the PointPillars network at various depths through experiments to show its effectiveness towards reaching ideal detection performance on the KITTI benchmark. The realization that pushing semantic information through the PointPillars encoder may create a performance bottleneck led us to extend the PointPillars architecture by fusing an estimate of the ground plane in the same manner as our comparison approach to PointPainting. This fusion bypassed the encoder stage while also encoding ground level estimates to give an additional cue to the detection head regarding the placement of 3D boxes. While the improvement for the Car class was small, a more significant improvement was noted for the Cyclist dataset. The resulting model experienced a small decrease in speed relative to PPslim while exceeding the evaluation performance of PPaint without needing complex image feature fusion. We will release our segmentation annotations publicly to aid future research.

6.0.1 Future Work

The drastic reduction in model complexity & increase in speed all the while maintaining performance (and exceeding competing models) means that it's possible in the future to implement our PPslim model as a two-stage detector. With such a model, we could predict a ground plane along with the original predictions at the first stage. The ability to estimate a ground plane could be helpful for guarding against the increased performance loss in the prediction of 3D box vertical size and vertical position (as revealed by our parameter analysis). In the second stage, the proposed predictions could be merged and re-weighted to

produce the final predictions. The first stage of such a network could omit the computation of NMS (a threshold could still be applied to filter the first stage proposals), meaning that a significant portion of the computational cost could be deferred to the second stage of the network. With a two stage architecture, the feature merging approach described in Chapter 5 would be more feasible since the addition of extra network layers would mean a later fusion would be possible. This would allow a 2D image feature extractor (such as a segmentation network) to execute in parallel with the two-stage network and to perform a late fusion of features (without the drawback of having to run sequentially as in the case of PointPainting). Even if our model speed is halved it would still run faster than its nearest competing model. Some initial tentative work has also shown some promise in applying our PPslim model to the NuScenes dataset, with a similar drastic reduction in model parameters while maintaining detection performance.

Given the evaluations that we've performed on the Jetson using competing models, it would be beneficial to evaluate those detectors in terms of their range performance including minimum number of LiDAR points to maintain a certain performance threshold. We could contrast that analysis with power consumption on the Jetson along with comparisons of model complexity and parameter counts. It would give an opportunity to incorporate some additional, newer, detectors from the last few years such as those utilizing transformers, for example.

Future work following from Chapter 5 can be broken down into 2 main tasks, with the first having a number of independent subtasks. The first is regarding the segmentation annotations. Since it was recently discovered that competing segmentation annotations were released for KITTI [49] after our work was complete, a number of useful extensions could be applied to our data. The first is to validate the competing annotations for correctness, as several pending issues were raised on the project page regarding data quality. In our case, the annotations were manually verified and verified through automation. Automation verification included confirming existing tracking information, for the few sequences where it was available, matches our labels and also verifying 2D box bounds fully enclose segmentation regions.

But in addition to this annotation subtask, another subtask would be to adjust KITTI 3D box bounds (and in turn the 2D box bounds) to give much tighter results based on a combination of segmentation information and the visibility indicator already assigned to each instance (fully visible, partially occluded, heavily occluded). This is due to the fact that there are a noticeable number of KITTI 3D ground truth boxes with very large over estimations of box bounds, likely caused by the Mechanical Turk nature of the annotations. The sides of the 3D bounding boxes can be adjusted for fully visible sides of cars, especially if they're within a certain range (cloud point density). As an optional step, since KITTI

cars are generally symmetric, points could be mirrored across the longitudinal axis of each vehicle. Then, a registration across all separate point clouds for each unique car in KITTI can be performed since we provide tracking information for each car, allowing one to identify all point clouds belonging to the same vehicle across scenes. This would give a more dense cloud for a car than any single point cloud and can be combined with the knowledge of occlusion levels for each instance. With a dense cloud and knowledge which cloud sides were fully visible around the object, the 3D box bounds could be moved towards the cloud, in a negative direction to the outward pointing normal to each box face. Consistency of the new 3D box could be checked by projecting to 2D and comparing to the segmentation mask bounds in all frames where the car is fully visible. Furthermore, one more subtask could be the investigation of whether utilizing the segmentations to pre-train semantic segmentation models, that then train on a larger dataset such as NuScenes would obtain even better performance.

The second task from Chapter 5 would be to investigate ground plane estimation as part of the PointPillars model. The feature analysis we performed in Section 4.4.3.1 showed that after we controlled for classification (which had a small impact on overall performance), a large performance increase was observed through a better knowledge of the vertical position of 3D boxes. If the PointPillars model was trained to predict the ground plane along with 3D predictions, and the ground plane was then used to adjust the 3D predictions at its most confident ground plane locations or locations with "low" object scores, it would be interesting to see the detection performance impact from such an adjustment. Additional segmentations could be added for the drivable area within the scene where a definitive ground plane ground truth could be obtained from enclosed cloud points. The reason to do this is that even though ground plane estimation approach seems reasonable, an end-to-end trained approach is generally preferable to a hand-crafted approach.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[2] AEVA. Aeva introduces atlas – the first automotive-grade 4d lidar sensor for mass production automotive applications. https://web.archive.org/web/20240428041833/https://www.aeva.com/press/aeva-introduces-atlas-the-first-automotive-grade-4d-lidar-sensor-for-mass-production-automotive-applications/, 2024. Accessed: 28-Apr-2024.

[3] Hamed H. Aghdam, Elnaz J. Heravi, Selameab S. Demilew, and Robert Laganiere. Rad: Realtime and accurate 3d object detection on embedded systems. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 2869–2877, 2021.

[4] Lei Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *CoRR*, abs/1607.06450, 2016.

[5] Peter Barry and Patrick Crowley. *Modern Embedded Computing: Designing Connected, Pervasive, Media-Rich Systems*. Elsevier, 2021.

[6] Rodrigo Benenson, Mohamed Omran, Jan Hendrik Hosang, and Bernt Schiele. Ten years of pedestrian detection, what have we learned? In *ECCV Workshops (2)*, volume 8926 of *Lecture Notes in Computer Science*, pages 613–627. Springer, 2014.

[7] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, New York, 2006.

[8] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.

[9] Gary Bradski, Adrian Kaehler, et al. Open source computer vision library. https://opencv.org/, 2023. Accessed: April, 2022.

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

[11] Zhaowei Cai, Quanfu Fan, Rogerio S Feris, and Nuno Vasconcelos. A unified multi-scale deep convolutional neural network for fast object detection. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV 14*, pages 354–370. Springer, 2016.

[12] Ding-Yun Chen, Xiao-Pei Tian, Yu-Te Shen, and Ming Ouhyoung. On visual similarity based 3d model retrieval. In *Computer graphics forum*, volume 22, pages 223–232. Wiley Online Library, 2003.

[13] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE transactions on pattern analysis and machine intelligence*, 40(4):834–848, 2017.

[14] Liang-Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. Encoder-decoder with atrous separable convolution for semantic image segmentation. In *Proceedings of the European conference on computer vision (ECCV)*, pages 801–818, 2018.

[15] 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 (CVPR)*, July 2017.

[16] Yongjian Chen, Lei Tai, Kai Sun, and Mingyang Li. Monopair: Monocular 3d object detection using pairwise spatial relationships. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12093–12102, 2020.

[17] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[18] CVAT.ai Corporation. Computer vision annotation tool (cvat), April 2024.

[19] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-fcn: Object detection via region-based fully convolutional networks. In *NIPS*, pages 379–387, 2016.

[20] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 886–893 vol. 1, 2005.

[21] Selameab S Demilew, Hamed H Aghdam, Robert Laganière, and Emil M Petriu. Fa3d: Fast and accurate 3d object detection. In *Advances in Visual Computing: 15th International Symposium, ISVC 2020, San Diego, CA, USA, October 5–7, 2020, Proceedings, Part I 15*, pages 397–409. Springer, 2020.

[22] Jia Deng, R. Socher, Li Fei-Fei, Wei Dong, Kai Li, and Li-Jia Li. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition(CVPR)*, volume 00, pages 248–255, 06 2009.

[23] P. Dollar. Caltech pedestrian benchmark. http://www.vision.caltech.edu/Image_Datasets/CaltechPedestrians/. Accessed 2017.

[24] P. Dollar, C. Wojek, B. Schiele, and P. Perona. Pedestrian detection: An evaluation of the state of the art. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 34(4):743–761, 2012.

[25] Piotr Dollár. Piotr's Computer Vision Matlab Toolbox (PMT). https://github.com/pdollar/toolbox.

[26] Piotr Dollár, Christian Wojek, Bernt Schiele, and Pietro Perona. Pedestrian detection: A benchmark. In *2009 IEEE conference on computer vision and pattern recognition*, pages 304–311. IEEE, 2009.

[27] John C. Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. In *COLT*, pages 257–269. Omnipress, 2010.

[28] Andreas Ess, Bastian Leibe, and Luc Van Gool. Depth and appearance for mobile scene analysis. In *ICCV*, pages 1–8. IEEE Computer Society, 2007.

[29] Andreas Ess, Bastian Leibe, Konrad Schindler, and Luc Van Gool. A mobile vision system for robust multi-person tracking. In *2008 IEEE conference on computer vision and pattern recognition*, pages 1–8. IEEE, 2008.

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

[31] Pedro F. Felzenszwalb, Ross B. Girshick, David A. McAllester, and Deva Ramanan. Object detection with discriminatively trained part-based models. *IEEE Trans. Pattern Anal. Mach. Intell.*, 32(9):1627–1645, 2010.

[32] Sanja Fidler, Sven J. Dickinson, and Raquel Urtasun. 3d object detection and viewpoint estimation with a deformable 3d cuboid model. In *NIPS*, pages 620–628, 2012.

[33] L-CCGP Florian and Schroff Hartwig Adam. Rethinking atrous convolution for semantic image segmentation. In *Conference on computer vision and pattern recognition (CVPR). IEEE/CVF*, volume 6, 2017.

[34] Cheng-Yang Fu, Wei Liu, Ananth Ranga, Ambrish Tyagi, and Alexander C. Berg. Dssd : Deconvolutional single shot detector. *CoRR*, abs/1701.06659, 2017.

[35] Kunihiko Fukushima and Sei Miyake. Neocognitron: A new algorithm for pattern recognition tolerant of deformations and shifts in position. *Pattern Recognit.*, 15(6):455–469, 1982.

[36] A Geiger, P Lenz, C Stiller, and R Urtasun. Vision meets robotics: The KITTI dataset. *The International Journal of Robotics Research*, 32(11):1231–1237, August 2013.

[37] A. Geiger, P. Lenz, and R. 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, Providence, RI, June 2012. IEEE.

[38] Deepak Ghimire. Complex-YOLOv3: An implementation of yolov3 detector for 3d point clouds. https://github.com/ghimiredhikura/Complex-YOLOv3, 2019. Accessed: 2022-04-27.

[39] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *2014 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 00, pages 580–587, June 2014.

[40] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

[41] Martin T Hagan and Mohammad B Menhaj. Training feedforward networks with the marquardt algorithm. *IEEE transactions on Neural Networks*, 5(6):989–993, 1994.

[42] Shima Hamidi, Reid Ewing, Ebrahim Azimi, Mohamad Tayarani, Bahar Azin, Justyna Kaniewska, Dong ah Choi, Hassan Ameli, and Duman Bahrami-Rad. A national investigation on the impacts of lane width on traffic safety: Narrowing travel lanes as an opportunity to promote biking and pedestrian facilities within the existing roadway infrastructure. Research report, Center for Climate-Smart Transportation (CCST), November 2023.

[43] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In *ICLR*, 2016.

[44] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, New York, NY, USA, 2nd edition, 2003.

[45] Chenhang He, Hui Zeng, Jianqiang Huang, Xian-Sheng Hua, and Lei Zhang. Structure aware single-stage 3d object detection from point cloud. In *CVPR*, pages 11870–11879. IEEE, 2020.

[46] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, June 2016.

[47] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[48] Kaiming He and Jian Sun. Convolutional neural networks at constrained time cost. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5353–5360, 2015.

[49] Jonas Heylen. Kitti3d instance segmentation development kit. https://github.com/HeylenJonas/KITTI3D-Instance-Segmentation-Devkit, 2023. Accessed on 2022-05-10.

[50] Geoffrey Hinton. Overview of mini-batch gradient descent. Accessed: December, 2022.

[51] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In *A Field Guide to Dynamical Recurrent Neural Networks*. IEEE Press, 2001.

[52] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, and Kevin Murphy. Speed/accuracy trade-offs for modern convolutional object detectors. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3296–3297, 2017.

[53] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and 1MB model size. *CoRR*, abs/1602.07360, February 2016.

[54] Y. Ioannou, D. Robertson, R. Cipolla, and A. Criminisi. Deep roots: Improving cnn efficiency with hierarchical filter groups. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5977–5986, July 2017.

[55] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr, 2015.

[56] SH Jeong, JE Lee, SU Choi, JN Oh, and KH Lee. Technology analysis and low-cost design of automotive radar for adaptive cruise control system. *International Journal of Automotive Technology*, 13:1133–1140, 2012.

[57] Andrew E Johnson and Martial Hebert. Using spin images for efficient object recognition in cluttered 3d scenes. *IEEE Transactions on pattern analysis and machine intelligence*, 21(5):433–449, 1999.

[58] A.B. Jones and C.D. Smith. Trends in embedded systems technology for mobile phones and drones. *Journal of Embedded Systems*, 45(3):123–134, 2020.

[59] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR (Poster)*, 2015.

[60] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[61] 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.

[62] Alex H Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. Pointpillars: Fast encoders for object detection from point clouds. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12697–12705, 2019.

[63] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[64] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[65] Jianan Li, Xiaodan Liang, ShengMei Shen, Tingfa Xu, Jiashi Feng, and Shuicheng Yan. Scale-aware fast r-cnn for pedestrian detection. *IEEE transactions on Multimedia*, 20(4):985–996, 2017.

[66] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, December 2013.

[67] 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.

[68] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*, pages 740–755. Springer, 2014.

[69] Haibin Ling and David W. Jacobs. Shape classification using the inner-distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29:286–299, 2007.

[70] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14*, pages 21–37. Springer, 2016.

[71] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[72] Charles T. Loop and Zhengyou Zhang. Computing rectifying homographies for stereo vision. In *CVPR*, pages 1125–1131. IEEE Computer Society, 1999.

[73] MathWorks. *Computer Vision Toolbox User's Guide*, 2022. Version 9.4.

[74] Daniel Maturana and Sebastian A. Scherer. Voxnet: A 3d convolutional neural network for real-time object recognition. *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2015.

[75] N. Maudzung. Complex-YOLOv4-Pytorch: An implementation of yolov4 for 3d object detection in point clouds using pytorch. https://github.com/maudzung/Complex-YOLOv4-Pytorch, 2020. Accessed: 2022-04-27.

[76] Warren Mcculloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147, 1943.

[77] Merriam-Webster. Definition of efficiency. Merriam-Webster Dictionary Online, 2023. Accessed: 25-12-2023.

[78] Ajmal S. Mian. Robust realtime feature detection in raw 3d face images. In *WACV*, pages 220–226. IEEE Computer Society, 2011.

[79] NVIDIA Corporation. Nvidia jetson agx xavier. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/. Accessed 2023.

[80] NVIDIA Corporation, https://docs.nvidia.com/jetson/archives/l4t-archived/l4t-325/index.html. *NVIDIA Jetson Linux Developer Guide*, 32.5 Release. Accessed: 21-12-2023.

[81] NXP Research. S32v whitepaper. http://cache.freescale.com/files/automotive/doc/white_paper/S32V230WP.pdf, 2015. Accessed 2017.

[82] Abishai Polus, J Craus, and Moshe Livneh. Determination of longitudinal grades on rural roads. *Transportation research circular*, (E-C003):21–1, 1998.

[83] Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr computational mathematics and mathematical physics*, 4(5):1–17, 1964.

[84] Ingrid B. Potts, Douglas W. Harwood, and Karen R. Richard. Relationship of lane width to safety on urban and suburban arterials. *Transportation research record*, 2023(1):63–82, 2007.

[85] Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation. *CVPR 2017*, December 2016.

[86] 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*, volume 30. Curran Associates, Inc., 2017.

[87] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions. In *ICLR (Workshop)*. OpenReview.net, 2018.

[88] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[89] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

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

[91] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28, 2015.

[92] Santiago Royo and Maria Ballesta-Garcia. An overview of lidar imaging systems for autonomous vehicles. *Applied Sciences*, 9(19), 2019.

[93] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, October 1986.

[94] Radu Bogdan Rusu, Nico Blodow, and Michael Beetz. Fast Point Feature Histograms (FPFH) for 3D Registration. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), Kobe, Japan, May 12-17*, 2009.

[95] Radu Bogdan Rusu, Nico Blodow, and Michael Beetz. Fast point feature histograms (fpfh) for 3d registration. In *ICRA*, pages 3212–3217. IEEE, 2009.

[96] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. In *ICLR*, 2014.

[97] Zhiqiang Shen, Zhuang Liu, Jianguo Li, Yu-Gang Jiang, Yurong Chen, and Xiangyang Xue. Dsod: Learning deeply supervised object detectors from scratch. In *Proceedings of the IEEE international conference on computer vision*, pages 1919–1927, 2017.

[98] Shaoshuai Shi, Chaoxu Guo, Li Jiang, Zhe Wang, Jianping Shi, Xiaogang Wang, and Hongsheng Li. Pv-rcnn: Point-voxel feature set abstraction for 3d object detection. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10529–10538, 2020.

[99] Shaoshuai Shi, Xiaogang Wang, and Hongsheng Li. Pointrcnn: 3d object proposal generation and detection from point cloud. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 770–779, 2019.

[100] Martin Simony, Stefan Milzy, Karl Amendey, and Horst-Michael Gross. Complex-yolo: An euler-region-proposal for real-time 3d object detection on point clouds. In *Proceedings of the European conference on computer vision (ECCV) workshops*, pages 0–0, 2018.

[101] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[102] Pei Sun, Henrik Kretzschmar, Xerxes Dotiwalla, Aurelien Chouard, Vijaysai Patnaik, Paul Tsui, James Guo, Yin Zhou, Yuning Chai, Benjamin Caine, et al. Scalability in perception for autonomous driving: Waymo open dataset. In *Proceedings of the*

IEEE/CVF conference on computer vision and pattern recognition, pages 2446–2454, 2020.

[103] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 31, 2017.

[104] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[105] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[106] Dmitry Ulyanov, Andrea Vedaldi, and Victor S. Lempitsky. Instance normalization: The missing ingredient for fast stylization. *CoRR*, abs/1607.08022, 2016.

[107] Rytis Verbickas. Convolutional Neural Network-based Vision Systems for Unmanned Aerial Vehicles. Master's thesis, Carleton University, Canada, 01 2013.

[108] Rytis Verbickas and Robert Laganiere. Kitti car and truck instance segmentations. https://www.site.uottawa.ca/research/viva/projects/kitti_instance_seg/index.html, 2024. Accessed: 2024-01-10.

[109] Rytis Verbickas, Robert Laganiere, Daniel Laroche, Changyun Zhu, Xiaoyin Xu, and Ali Ors. Squeezemap: fast pedestrian detection on a low-power automotive processor using efficient convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 146–154, 2017.

[110] Sourabh Vora, Alex H Lang, Bassam Helou, and Oscar Beijbom. Pointpainting: Sequential fusion for 3d object detection. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 4604–4612, 2020.

[111] Elecia White. *Making Embedded Systems: Design Patterns for Great Software*. O'Reilly Media, 2011.

[112] Alexander Womg, Mohammad Javad Shafiee, Francis Li, and Brendan Chwyl. Tiny ssd: A tiny single-shot detection deep convolutional neural network for real-time embedded object detection. In *2018 15th Conference on computer and robot vision (CRV)*, pages 95–101. IEEE, 2018.

[113] Kun-Feng (Ken) Wu and Tong Lin. Investigating the effects of travel lane configuration and lane width on traffic safety where powered-two-wheelers (ptws) share with larger vehicles: A micro perspective. *Accident Analysis and Prevention*, 172:106682, 2022.

[114] Yuxin Wu and Kaiming He. Group normalization. In *Proceedings of the European conference on computer vision (ECCV)*, pages 3–19, 2018.

[115] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1492–1500, 2017.

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

[117] Yurong You, Yan Wang, Wei-Lun Chao, Divyansh Garg, Geoff Pleiss, Bharath Hariharan, Mark E. Campbell, and Kilian Q. Weinberger. Pseudo-lidar++: Accurate depth for 3d object detection in autonomous driving. In *ICLR*. OpenReview.net, 2020.

[118] Z. Zhang. A flexible new technique for camera calibration. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 22, pages 1330–1334, 2000.

[119] 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.

[120] Adam Ziebinski, Rafal Cupek, Damian Grzechca, and Lukas Chruszczyk. Review of advanced driver assistance systems (adas). In *AIP Conference Proceedings*, volume 1906. AIP Publishing, 2017.

[121] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2017.

APPENDICES

Appendix A

PPslim Training Parameter Configuration

The following tables break down PPslim training parameters into groups depending on whether they are applicable for model building, training or augmentation.

Parameter	Setting	Description (if applicable)
Pillar Volume Bounds	[0.0, -40.48, -3.0, 70.4, 40.48, 1.0]	The min and max XYZ bounds given as XYZ_{min} and XYZ_{max}
Pillar Dimension	[0.22, 0.22, 4]	The dimension of a single pillar (meters)
Downsample Strides	[2, 2]	For initial convolution in each block
Encoder pre-max pool FC units	[16,32]	The number of FC units per layer before max-pooling
Encoder post-max pool FC units	[64]	The number of FC units per layer after max-pooling
Backbone Block Channel Counts	[32,64]	The per-block channel counts
Backbone Block Layers	[3, 5]	Number of convolutional layers in each block
Upsample Block Channels	[128, 128]	Number of upsample channels per block
Upsample Block Stride Size	[1, 2]	When upsampling block outputs
Max Pillars	8000	Maximum pillars per scene
Max Points	125	Maximum points per pillar

Table A.1: Voxelization, input encoder and backbone parameter for training an example PPslim model from Chapter 4. Bounds and coordinates are relative to the KITTI LiDAR frame (XYZ where X:forward, Y:left, Z:up)

Parameter	Setting	Description (if applicable)
Car Anchor Size	[1.6, 3.9, 1.56]	In meters
Cyclist Anchor Size	[0.6, 1.76, 1.73]	In meters
Cyclist and Car Anchor Angles	$[0, \frac{\pi}{2}]$	Values as a list (in radians)
NMS Score Threshold	0.3	Cutoff to consider for NMS
NMS IOU Threshold	0.01	IOU for rejecting overlap
Rotate NMS	Yes	Type of NMS
Pre-NMS Max Samples	1000	
Post-NMS Max Samples	300	
Region Similarity	IOU-based	
Car Matched Threshold	0.6	Positive cutoff for SSD
Car Unmatched Threshold	0.45	Negative cutoff for SSD
Cyclist Matched Threshold	0.4	Positive cutoff for SSD
Cyclist Unmatched Threshold	0.25	Negative cutoff for SSD
Focal Loss α	0.25	Positive/negative balance factor
Focal Loss γ	2.0	Easy/hard focusing factor
Activation Function	Swish	For entire network
Smooth L1 Localization σ	3.0	
Smooth L1 Localization Loss Weights	[1.0, 1.0, 4.0, 1.0, 1.0, 4.0, 3.0]	Applied respectively to the XYZ position, LWH and yaw angle loss
Classification Weight	1.0	Multiplier for respective loss
Localization Weight	2.0	Multiplier for respective loss
Loss Normalization	Positives Only	Which samples to count to normalize loss
Gradient clipping	15	L2 clipping
Batch Size	2	
Optimizer	Adam	With 0.0001 weight decay
Learning Rate	Exponential	0.0003 initial, decay 0.1 by 0.8
Learning Rate Policy	300+300+300	Epochs

Table A.2: SSD detection head (including NMS) and loss parameters for training an example PPslim model from Chapter 4. Bounds and coordinates are relative to the KITTI LiDAR frame (XYZ where X:forward, Y:left, Z:up)

Parameter	Setting	Description (if applicable)
Car samples	20	Count per scene
Car Min Points	5	Count per object
Cyclist samples	8	Count per scene
Cyclist Min Points	5	Count per object
Scene Rotation Noise	$\frac{\pi}{2}$	In radians
Scene Scale Noise	[0.95, 1.05]	Min and max factor
Scene Translation Noise	0.2	For each XYZ, in meters
Scene Random Flips	Yes	
Object Rotation Noise	$\frac{\pi}{20}$	In radians
Object Scale Noise	[0.95, 1.05]	Min and max factor
Object Translation Noise	0.1	For each XYZ, in meters

Table A.3: Sampling augmentation parameters for training an example PPslim model from Chapter 4. Bounds and coordinates are relative to the KITTI LiDAR frame (XYZ where X:forward, Y:left, Z:up)