Multimodal Fusion Detection System for Autonomous Vehicles

by

Michael Person

Bachelor of Science
Engineering Physics
Colorado School of Mines
2016

A thesis
submitted to the College of Engineering at Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Mechanical Engineering

Melbourne, Florida
April, 2018

We the undersigned committee
hereby approve the attached thesis

Multimodal Fusion Detection System for Autonomous Vehicles by Michael Person

---

Matthew Jensen, Ph.D.
Assistant Professor
Mechanical Aerospace Engineering
Committee Chair

Anthony Smith, Ph.D.
Assistant Professor
Electrical and Computer Engineering
Outside Committee Member

Hector Gutierrez, Ph.D.
Professor
Mechanical Aerospace Engineering
Committee Member

Hamid Hefazi, Ph.D.
Professor and Department Head
Mechanical Aerospace Engineering

ABSTRACT

Title:

Multimodal Fusion Detection System for Autonomous Vehicles

Author:

Michael Person

Major Advisor:

Matthew Jensen, Ph.D.

In order for autonomous vehicles to safely navigate the road ways, accurate object detection must take place before safe path planning can occur. Currently there is a gap between models that are fast enough and models that are accurate enough for deployment. We propose Multimodal Fusion Detection System (MDFS), a sensor fusion system that combines the speed of a fast image detection CNN model along with the accuracy of a LiDAR point cloud data through a decision tree approach. The primary objective is to bridge the trade-off between performance and accuracy. The motivation for MDFS is to reduce the computational complexity associated with using a CNN model to extract features from an image. To improve efficiency, MDFS extracts complimentary features from the LIDAR point cloud in order to obtain comparable detection accuracy. MFDS achieves 3.7% higher accuracy than the base CNN detection model and is able to operate at 10 Hz. Additionally, the memory requirement for MFDS is small enough to fit on the Nvidia Tx1 when deployed on an embedded device.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I am thankful for my personal funding that comes from the SMART Scholarship, which is funded by: USD/R&E, National Defense Education Program / BA-1, Basic Research. I would also like to thank both Magna and Polaris for funding Florida Tech's IGVC team. I would like to thank Aaron Schuetz for introducing me to the reward one gains after working hard in order to push past our self imposed barriers. Lastly, I would like to thank Mark Ollis for teaching me how to take theoretical computer vision knowledge and build real world systems with it.

Chapter 1

Introduction

1.1 Background

1.1.1 Autonomous Vehicles

Vehicles are an integral part of the world as we know it today. They transport goods between factories, shipping ports, and stores and also are a primary mode of transportation for many populations of the world. Vehicles may be very beneficial; however, their usefulness does not come without a cost. Every year roughly 30,000 are killed and over 2 million are injured in the US in automobile accidents [3]. Therefore, safety is a large concern not only for government regulators but also for automotive manufacturers. Keeping their passengers safe is a top priority for automotive manufacturers but there is also an added incentive the large profits that can be had by constantly researching and upgrading their safety features. However, with human drivers comes the opportunity for human error, increasing the risk associated with using vehicles. At best, humans have limited reaction time

and sensory inputs to make optimal decisions and at worst it is easy to become distracted while driving by either passengers, phones, or inebriation just to name a few.

It is desired to have vehicles that are capable of driving themselves, since they would never be distracted and could therefore be a huge step forward in vehicle safety. Ever since Stanley, the robotic vehicle from Stanford under Sebastian Thrun, shown in Figure 1.1, won the DARPA Grand Challenge in 2005, there has been a push in the research and commercial fields to work towards the development of autonomous vehicles. This rush for autonomy was increased when Boss, from Carnegie Mellon University won the DARPA Grand Urban Challenge in 2007, shown in Figure 1.2. Most vehicles now come equipped with Advanced Driver Assistance Systems (ADAS), such as Adaptive Cruise Control, that are classified as National Highway Traffic Safety Administration (NHTSA) level 1 or 2 autonomy (partial autonomy). However, one of the largest challenges in designing level 4 or 5 autonomous vehicles (highly or completely autonomous) is accurate perception in all environments of the world around the vehicle [4].

Figure 1.1: Stanley, winner of the 2005 DARPA Grand Challenge, [73]

Figure 1.2: Boss, winner of the 2007 DARPA Grand Urban Challenge, [19]

1.1.2 Perception

At the heart of perception lies computer vision. The vehicle must take in raw data from sensors such as cameras, LiDAR, and RADAR and then process it to form a meaningful representation of the world around it. These different modalities of data each provide their own unique benefits, but also have their own shortcomings and failure points. These different streams of data must be analyzed in order to gather the important information from the environment around the vehicle and must be passed to later stages of the autonomy system such as path planning. Cameras are the most common sensor used for perception in both commercially available and developmental systems due to their low cost and information density. Modern monocular cameras produce high resolution pictures for very low cost.

An autonomous vehicle's perception stack must perform three major tasks; detection, classification, and localization of objects in the world surrounding the vehicle. First, an object must be detected in the scene, meaning the vehicle must

recognize that there is something there. Second, the object must be classified, which in the terms of autonomous driving can include but is not limited to stop lights, signs, other vehicles, cyclists, and pedestrians. Lastly, in the perception stack after objects have been detected and classified, the agent must localize the object in some relevant coordinate frame for the path planner to safely route the vehicle. Normally algorithms will perform detection and classification and will then localize the classified detections, if any are found. Not only do these three tasks need to be done accurately for the safety of the planned path of the vehicle, they must also be done quickly due to the speed of the vehicle. If either the accuracy or speed of the perception system is low, then the autonomous vehicle will be dangerous and therefore unusable.

Not only are there multiple objects to look for in a scene, there are many different environments in which these objects will appear. For example, a white vehicle is easier to detect at noon when the lighting is uniform than when the sun is low and provides glare on the camera lens near the vehicle. There are a near infinite number of scenarios similar to the previous example and is impossible to write code with a limited number of sensors for every single situation the vehicle will encounter, which means the vehicle should be capable of learning, similar to a human, for generalization to situations the vehicle has not yet been provided direct instruction.

1.1.3 Image Detection

A naive approach for object detection and classification with camera data would be to select certain sized regions of interest (ROI) at different locations in the image and extract hand selected features such as histogram of oriented gradients

(HOG) or a histogram of color values. These histograms would be used to form feature vectors that have smaller dimensionality, yet representationally equivalent to each ROI. Next the feature vectors would be passed through linear classifiers such as a support vector machines (SVM) for every class of object you are looking for. The SVM would return a boolean true or false whether the extracted feature vector was a certain object or not. If the decision boundary that you are trying to find is nonlinear however, you would need to transform your feature vector to a different, often higher dimensionality, vector space in order for the linear SVM to perform nonlinear transforms in the original feature space. The designer must know beforehand what transform to perform and what features to extract, making this method very domain specific and makes both system design highly complicated and time consuming.

A modern approach for object detection and classification with camera data is to use a convolutional neural network (CNN). The CNN can be be passed an image and will output both bounding boxes and object labels. Using a CNN has two major advantages over hand crafted feature extraction and linear classification; learned feature extraction and highly nonlinear classification. The CNN learns which features to extract by optimizing its convolutional kernels by means of backpropogation and optimization for every example the CNN is trained on. Each layer of the CNN has a nonlinearity in it and modern CNN's usually have no fewer than 10 layers; meaning the decision boundary that the CNN can form is extremely rich in its classification power. This class of algorithms yields high accuracy and generalizable detections.

1.1.4 LiDAR Detection

There are many approaches to detect and classify objects in a scene using LiDAR data. There are grouping methods, hypothesis verification methods, sample consensus methods, segmentation methods, and even some CNNs just to name a few. Even though the data dimensionality is different, camera and LiDAR data analysis all have the common themes of feature extraction and classification. What is most advantageous about using LiDAR algorithms is that they reason directly with the 3D world instead of image based algorithms, which use a 2D projection. Directly manipulating 3D data allows detections to be used for path planning in the next stage of an autonomous vehicle, which would not be possible with the 2D detections of an image algorithm. Unfortunately, detection accuracy is traditionally worse for LiDAR based algorithms than camera based ones, making LiDAR only algorithms unsuitable for autonomous vehicles [21].

1.1.5 Detection Fusion

In order to overcome the weaknesses of these independent modes of data, it is possible to fuse the data together. The two defining types of data fusion between image and range data (LiDAR) are decision and feature level fusion. Decision level fusion is performed by fusing detections from the different modes of data together to form superior detections to the individual detections alone. An example of this would be to take camera only detections and LiDAR only detections and see if they were spatially close enough to be considered the same detection. Feature level fusion means to extract and combine features from the different modes of data in a way that leads to better detections than if the individual types of data

were analyzed independently. Both of these fusion techniques have their own weaknesses, as feature level fusion is difficult to find equivalent representations for each type of data and decision fusion can be less accurate since you are rely on each sets of features independently. The benefits though are increased detection quality over LiDAR only algorithms and 3D detections which can not be obtained by monocular camera algorithms thus making fusion algorithms the final output of and autonomous vehicle's perception system.

1.2 Objective

Image based algorithms provide higher accuracy detections than LiDAR based algorithms, however highly accurate image algorithms are too slow and take up too much memory to deploy in current real world systems and algorithms that are fast enough are not accurate enough [10][57][56]. The primary problem is that the accuracy gains from larger, more complex algorithms are much smaller than the increased computational complexity that come with them [31]. The Multimodal Fusion Detection System (MFDS) that was developed in this thesis attempts to solve this problem of memory consumption, accuracy, and speed by using information from different types of sensors to efficiently augment one another, rather than greatly increase the computational cost of a single sensor for marginal benefit. In addition, this algorithm has the capability of reasoning about the 3D world instead of a 2D projection of it, allowing objects to be detected in both spaces, which is not yet possible even in the most computationally expensive image only algorithms.

1.3 Importance of Study

Autonomous vehicles rely on an accurate and fast perception system to allow for safe path planning. The algorithm proposed in this thesis is the foundation for such a perception system; fast, accurate, and localized detections.

The purpose of this thesis is to study the fusion of image and range data in order to provide accurate, fast object detection and classification. Many modern detection algorithms have high levels of accuracy or low latency, however they are not as accurate as slower algorithms. Therefore, it is important to build algorithms that bridge the gap to small, fast, and accurate algorithms. This is easier to exploit using data fusion rather than building extremely computationally intense algorithms for a single type of data. The importance of size, accuracy, and speed matters even more so when developing an algorithm for use on an embedded system which have less computational power and memory than the desktop computers the algorithms are built on. The target deployment platform for this algorithm is the Nvidia Tegra series, specifically the Tx1.

Chapter 2

Literature Review

2.1 Datasets

For supervised learning algorithms, datasets with inputs and corresponding labels must be provided. The different types of supervised learning tasks that were researched were classification and detection tasks. Image classification is the task of determining the class of object that an image belongs to, illustrated in Figure 2.1. Image detection is the task of detecting objects in a given image and also determining the objects class, illustrated in Figure 2.2. Image segmentation is the task of assigning every single pixel in an image a class label, illustrated in Figure 2.3.

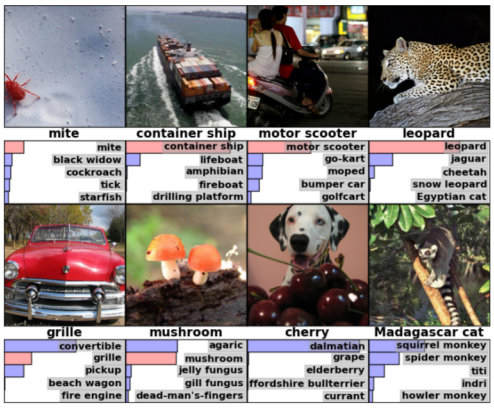

Figure 2.1: Classification Task, [37]

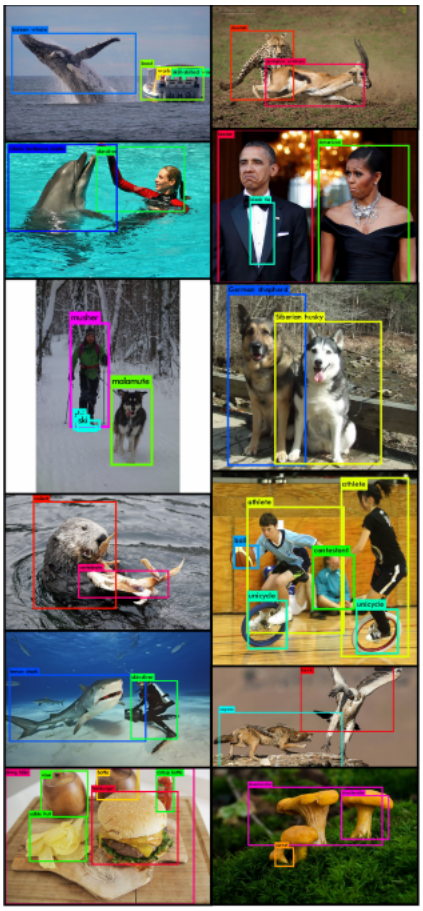

Figure 2.2: Detection Task, [56]

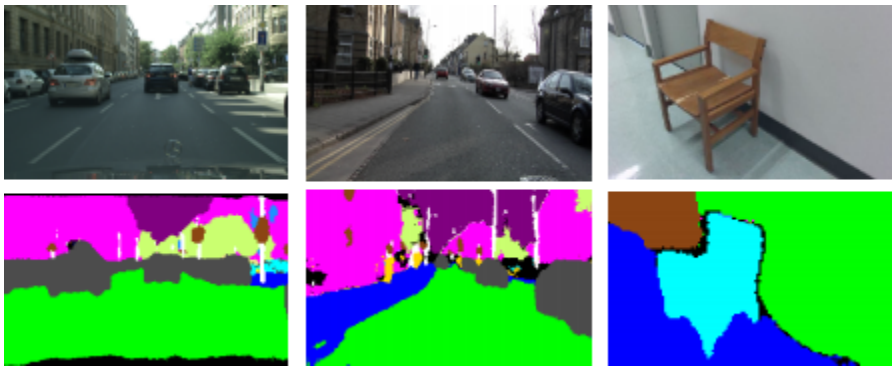

Figure 2.3: Segmentation Task, [51]

2.1.1 MNIST

MNIST is a dataset comprising of 60,000 training and 10,000 testing images of handwritten digits. The images are 28 x 28 black and white pixels spanning a total of 10 different classes, seen in Figure 2.4. MNIST was used for early image detection research due to the dataset's relatively low complexity.

Figure 2.4: Example from MNIST Dataset, [38]

2.1.2 CIFAR10

The CIFAR10 dataset consists of 60,000 images, each are 32 x 32 color pixels [36]. There are 10 classes and the total 60,000 images are split with 50,000 being for training and 10,000 for testing. This dataset is useful for determining the merits of a classification network since the images are so small, it is possible to perform training quickly in order to test a networks final accuracy. The CIFAR10 dataset was used extensively in this research in order to save time while testing out different network configurations.

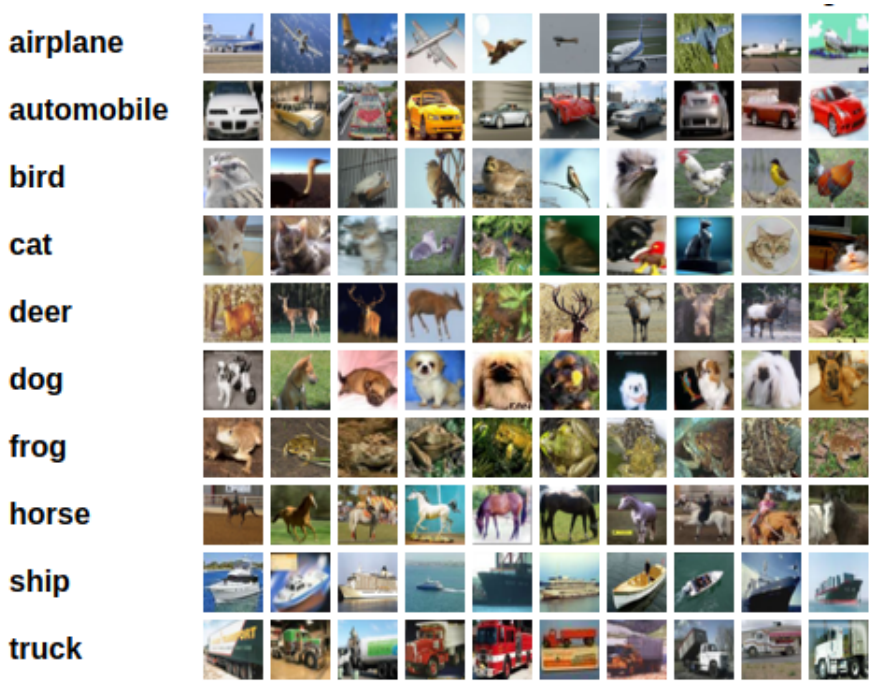

Figure 2.5: Sample of CIFAR10 Images spanning all Classes, [36]

2.1.3 Imagenet

The ImageNet dataset consists of roughly 1,287,000 images, each of which have 224 x 244 color pixels spread across 1000 classes [17]. The used dataset is the ImageNet

dataset from ImageNet Large Scale Visual Recognition Challenge 2011's (ILSVRC) classification task, which is only a small subset of the entire ImageNet dataset. The 2011 dataset is used because the classification challenge was largely eliminated after this year and when the classification challenge was brought back a new dataset was not introduced; therefore, the 2011 dataset has become the standard of all classification networks. ImageNet was the largest dataset until recently with the release of Open Images, which consists of roughly 9 million images. However, Open Images is focused on the detection task rather than classification [35].

ImageNet is the most commonly used classification training dataset because of the number of examples it contains, which allows for large models to be trained to very high accuracies, but with little to no loss in generalization. This allows for classification networks to learn general, robust features to extract in order to classify an image. However, these general extracted featured can also be used to learn a task that is more difficult than only classification, such as detection of segmentation. This process of learning classification and moving to another task is called transfer learning.

2.1.4 Common Objects in Context

The Common Objects in Context (COCO) dataset has 91 object types and 328, 000 images [42]. COCO has labels for detection, keypoint recognition, segmentation and captioning. It is one of the standard training datasets to use while training a detection CNN due it its large size and variability in classes.

Figure 2.6: Example of COCO's Segmentation Task, [42]

2.1.5 KITTI

The KITTI dataset is the largest and foremost vehicle related dataset that is publicly available [21]. There are 7481 synchronized images from stereo cameras and LiDAR point clouds within the dataset used for 2D and 3D object detection, segmentation, optical flow, depth prediction, odometry, and tracking. Each image is 360 x 1240 color pixel and contains classes of Pedestrians, Cyclists, and Vehicles. Since KITTI is the most relevant dataset to autonomous driving, it is common to pretrain a model on COCO due to its large number of training examples and then perform transfer learning for the model to be fine tuned on KITTI.

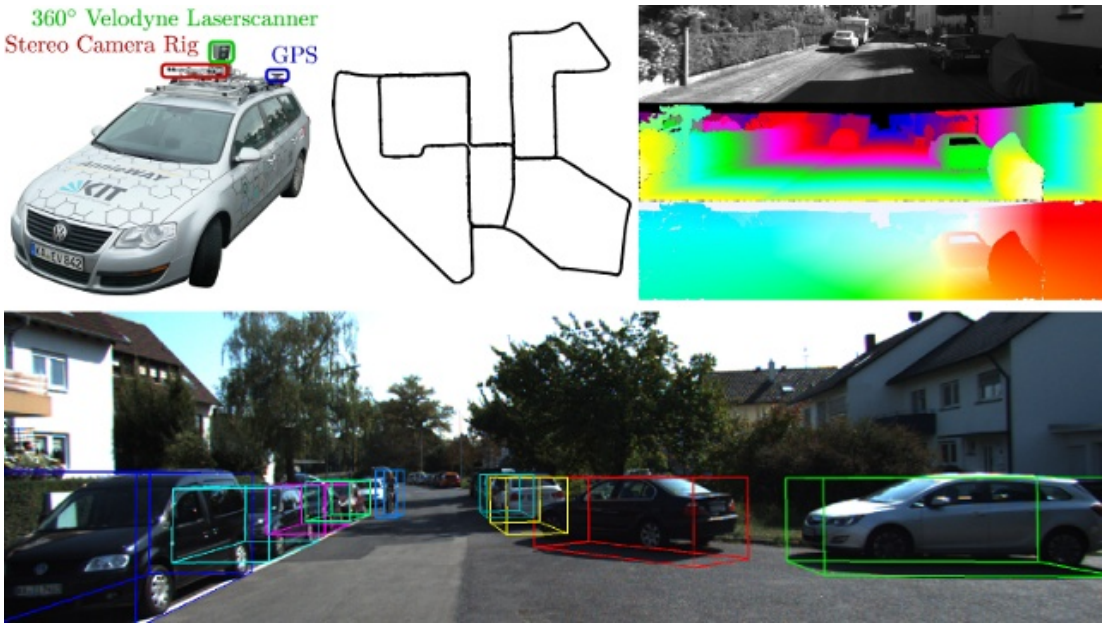

Figure 2.7: Examples of KITTIs Multiple Tasks, including Odometry, Depth Prediction, and Detection, [21]

2.2 Image Classification Convolutional Neural Networks

Convolutional Neural Networks (CNNs) have seen great success in image classification. Classification is the simplest task for a neural network and therefore was the first problem that was undertaken with CNNs. The difficulty of the task can mathematically be determined by the loss function that is required to be optimized, as learning is defined as the optimization of a model given a loss function. For the classification task, a standard cross entropy function is used:

$$L_{class} = -\frac{1}{N}\sum_{i=1}^{N}(y_i * log(\hat{y}_i) + (1 - y_i) * log(1 - \hat{y}_i)) \tag{2.1}$$

The predicted class is $\hat{y_i}$ and the label is y_i.

CNNs have also proven successful in a variety of other computer vision tasks including, but not limited to, detection and segmentation. Although classification, detection, and segmentation tasks are all different from one another, neural networks are called universal approximators and are able to learn how to perform each of them with high levels of accuracy [7]. Each task requires specific network architectures for optimality but due to their generality, many improvements to CNNs in one task also prove to be improvements in the others.

By themselves Multi-layer Perceptrons (MLPs), commonly referred to as neural networks, have enough representational power to be used to classify small datasets such MNIST, discussed in Section 2.1.1. However, as datasets that include more classes and/or larger images, the number of parameters in the model increases rapidly. This increase in parameters makes the optimization problem much more difficult and therefore, models converge to local optima that yield low accuracy, therefore achieving poor results. In order to solve this, a parameter sharing technique needed to be developed. The most successful parameter sharing technique so far is the 2D convolution. The convolution operation allows for a small number of parameters to form a large doubly block circulant matrix, which allows for large network inputs while restricting the number of parameters needed to be learned [24].

2.2.1 LeNet5

The first CNN used for image classification was developed by LeCun et al in 1998 and was used to recognize the hand written digits of MNIST [39]. The main differences between LeNet5 and its MLP predecessor was the use of the modern

backpropogation algorithm and the use of 2D convolution. Before LeNet5, a type of backpropogation was used that created artificial targets for each layer so that a gradient could be computed for each layer. However, LeNet5's backpropogation algorithm computed only the loss at the output of the network and propagated the gradients backward through every layer of the network. This combination of the modern backpropogation algorithm and 2D convolution yielded much stronger classification accuracy on MNIST, albeit at the then large cost of 24 Megabytes of runtime memory.

2.2.2 AlexNet

It was not until 2012 when AlexNet, built by Krizhevsky et al, won ILSVRC 2012 and achieved over 10% higher accuracy than the previous state of the art that CNNs started to be more widely used for image classification [37]. Although AlexNet was groundbreaking, it would not have been possible without the prior work of Geoffrey Hinton leading up to the development of AlexNet to achieve such dramatic results [27], [49]. AlexNet had many firsts that caused it to be such a powerful image classifier. These included, but were not limited to, a very aggressive data augmentation scheme including a Principal Component Analysis (PCA) image whitening technique along with more traditional crops, rotations, translations, and mirroring, a technique called local response normalization, and a multi-gpu training regime with the CUDA programming language. However, the largest contributor was the use of the RELU activation function instead of the standard sigmoid or hyperbolic tangent function. Lastly dropout was employed, which boosted the networks accuracy. AlexNet was an eight layer deep network with five convolutional layers and a total of 60 million parameters, seen in Figure

2.8.

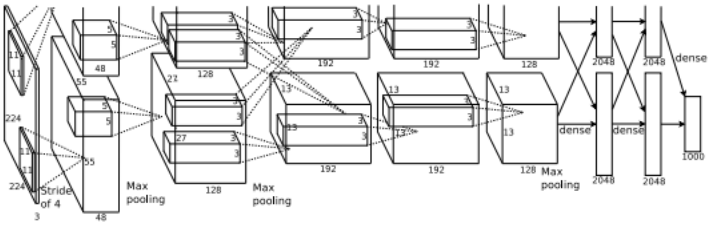

Figure 2.8: AlexNet Architecture, [37]

2.2.3 VGG

After AlexNet, research in CNNs accelerated rapidly and many papers soon followed. However, there are several significant papers that have shaped the way CNNs are used today. One such network was VGG, published in 2014, but did not do anything necessarily noteworthy besides make a 16 and 19 layer network, seen in Figure 2.9 [62]. VGG achieved state of the art performance, has provided inspiration to other networks, is commonly used as a comparison point for other networks, and has been used in conjunction with modern networks.

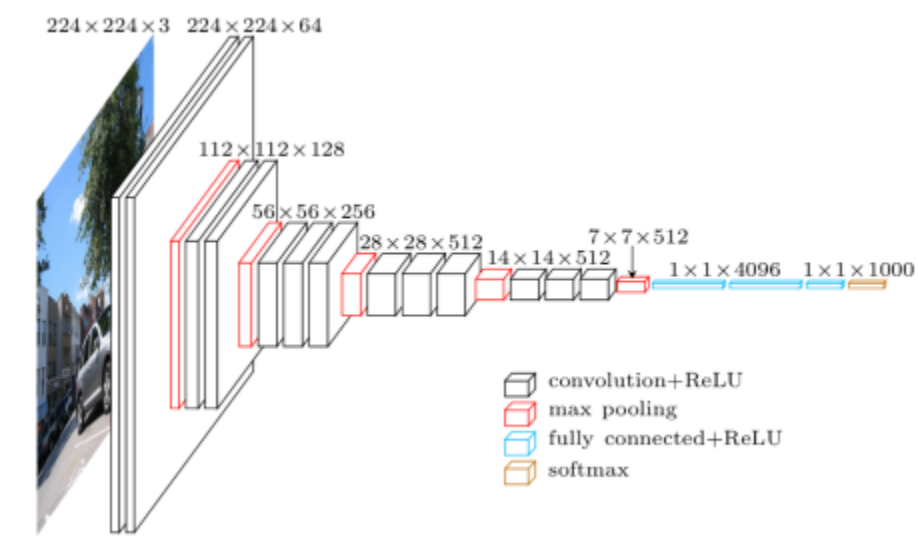

Figure 2.9: VGG Architecture, [62]

2.2.4 GoogLeNet

GoogLeNet, whose name pays homage to LeNet5, introduced the concept of the Inception module in 2015 [69]. The Inception module allowed networks to be wider while maintaining the same depth and also to be computationally efficient via the use of 1 by 1 bottleneck convolutions, shown in Figure 2.10. In addition, they were the first to address the issue of gradient backflow by adding on two auxiliary classifiers to expose early layers to a full gradient that was not washed out from the non-linearities.

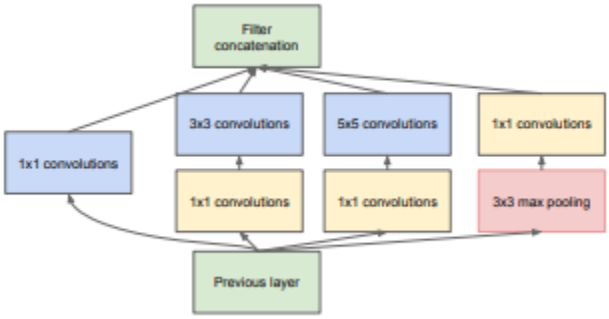

Figure 2.10: The Inception Module, [71]

2.2.5 Residual Networks

Another major improvement made in 2015 was Residual Networks (ResNets) [25]. ResNets introduced residual connections between layers, allowing for gradients to backflow through many layers and enabled very deep neural networks to be trained. To emphasize the increased level of depth that was achieved with residual connections, GoogLeNet was considered very deep with 22 layers but ResNet was 152 layers deep. A residual connection is an identity mapping from one layer to the next and can be thought of as a skip connection. The output of one layer is added to the output of the next layer, seen in Figure 2.11.

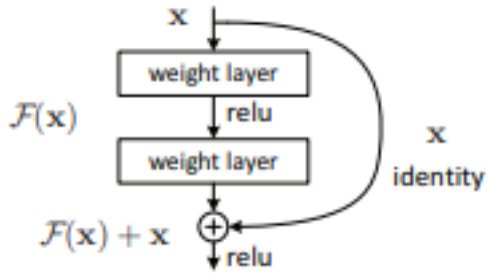

Figure 2.11: A Residual Connection, [25]

2.2.6 MobileNets

By 2017, image classification had reached the necessary accuracy levels for real world performance; however, to be useful for most applications, models would need to take up less memory and perform inference faster. The first major paper to address this problem of model deployment was MobileNets, which developed an accurate, small, fast classification network and was shown to perform well at transfer learning to more complicated tasks, seen in Table 2.1 [29]. MobileNets was based upon the idea of the depthwise seperable convolution, which was shown to be nearly equivalent to normal convolution, as shown in Figure 2.12. Depthwise seperable convolution is faster than standard convolution because they are optimized for the General Matrix to Matrix (GEMM) function call from within the CUBLAS library, which is utilized by the cuDNN library, the importance of these libraries will be discussed in Section 2.7.

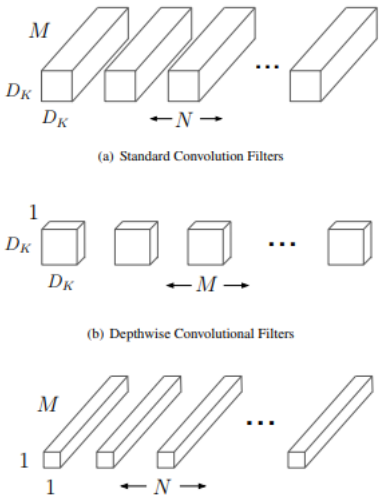

Figure 2.12: Depthwise Seperable Convolution, [29]

Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNets-224	70.6%	569	4.2
GoogLeNet	69.8%	1550	6.8
VGG 16	71.5%	15300	138

Table 2.1: MobileNets Comparison, [29]

2.2.7 DenseNets

Lastly, in 2017 a publication called DenseNets proposed identity shortcuts from every previous layer, which is the extreme generalization of residual connections that only occur every few layers, shown in Figure 2.13. Although this network has slightly lower accuracy than other modern networks, its reuse of features allows the network to have very few parameters. For comparison, an 110 layer ResNet has 1.7 million parameters while an 100 layer DenseNet has only .8 million parameters with an error rate that is half as large [30].

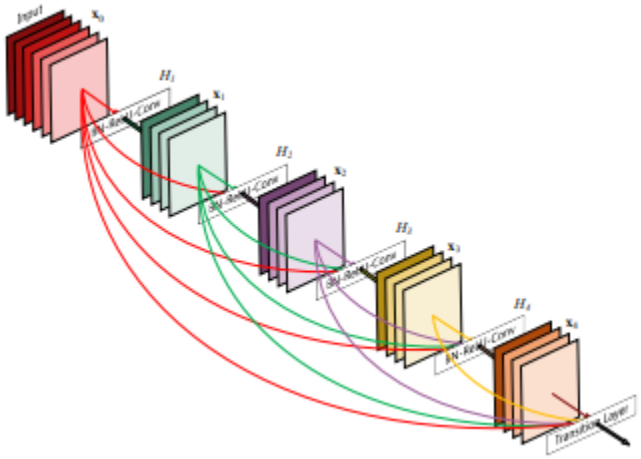

Figure 2.13: A DenseNet connection scheme, [30]

2.3 Image Detection

Image detection is the next step for computer vision after image classification. Instead of being provided an image with only one object that needs to be classified, an image with potentially many objects in it needs to classify each object but also locate it within the image. The output of a detection task are rectangular bounding boxes where each detection has six numbers; a class label, a confidence, and four coordinates for the bounding box. This increase in algorithmic complexity means that for the loss function to be optimized, a combination of class loss and position loss must be considered. This new detection loss function is written as the following, taken from [31]:

$$L_{det} = \frac{1}{N}\sum_{i=1}^{N}\Big(\lambda_{loc} * \mathbb{1}(i) * L_{loc}(\hat{f}_i, f_i) + \lambda_{class} * L_{class}(\hat{y}_i, y_i)\Big) \tag{2.2}$$

A predicted bounding box class is $\hat{y}_i$ while the closest corresponding labeled box's class is y_i. The predicted bounding box coordinates are $\hat{f}_i$ and the closest corresponding labeled box's coordinates are f_i. The $\mathbb{1}(i)$ function is a binary indicator depending on whether a label is found for the predicted bounding box. The λ terms are weighting parameters for how much the loss function should be focused on either optimizing object localization or object class. L_{loc} is the localization loss which varies widely algorithm to algorithm and the L_{class} loss is generally Cross Entropy.

2.3.1 Non Maximal Suppression (NMS)

A post processing step in virtually all detection algorithm is NMS, which eliminates overlapping detections. Detectors output many different detections, most of which

will be overlapping objects with varying levels of confidence, seen in Figure 2.14. NMS operates by checking the Intersection over Union (IOU) of all detections against all other overlapping detections, NMS then eliminates all but the most confident detection if the pair's IOU is below a given threshold. The IOU metric is computed by first computing the intersection area of the two detections, then by dividing by the area of union, seen in Figure 2.15. After NMS has been applied, there should be a single detection covering the area of a single object, seen in Figure 2.16. NMS is a hand crafted, greedy clustering algorithm and recent attempts have been made to perform the same task of detection suppression by either learning using stand alone networks, built in Recurrent Neural Networks (RNNs), or varying the threshold [45][5][28].

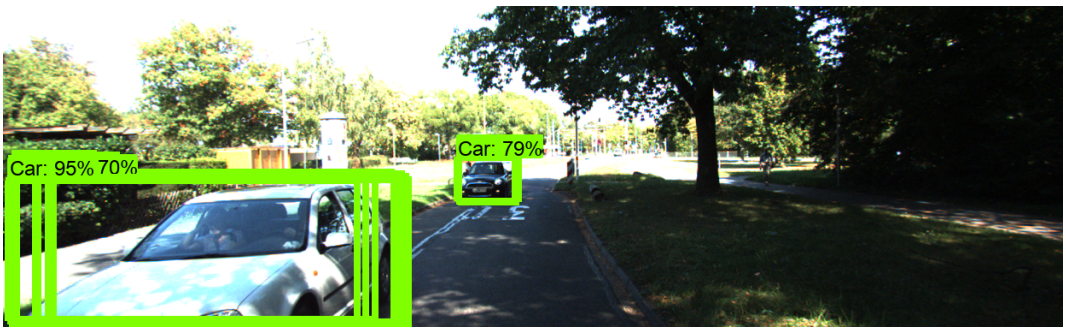

Figure 2.14: Example of Detections Before NMS

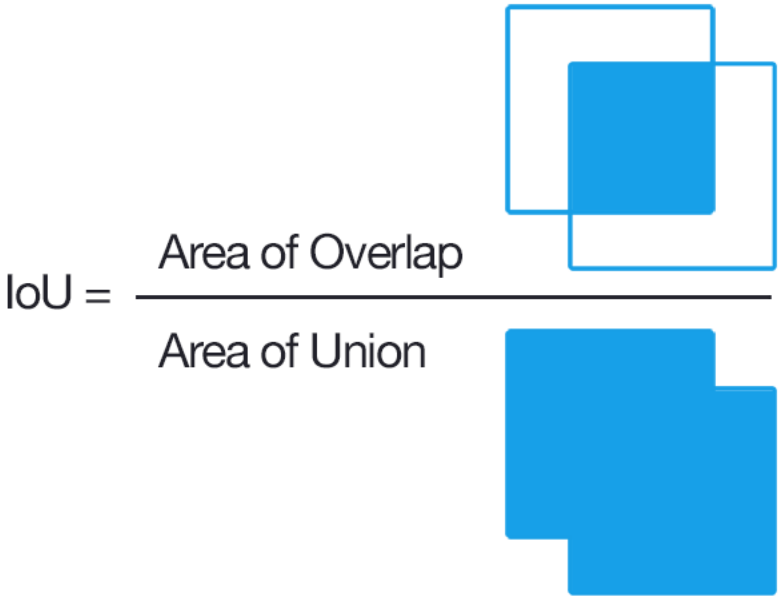

Figure 2.15: Visualization of IOU, [58]

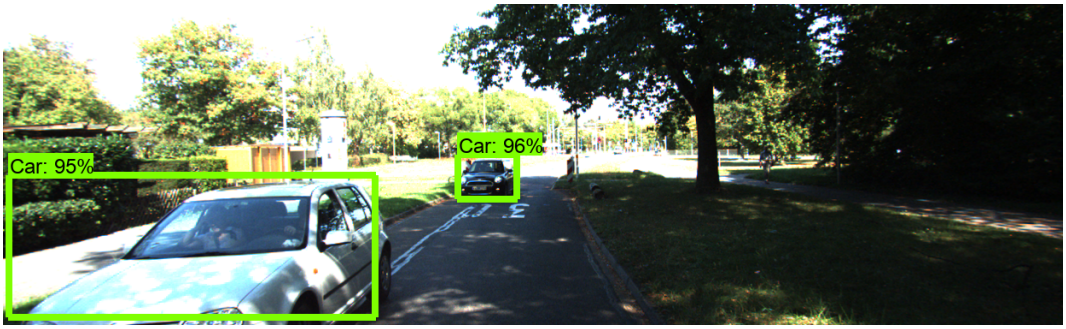

Figure 2.16: Example of Detections After NMS

2.3.2 Classical Detection

Classical Detection techniques refer to models that do not rely on CNNs. Generally speaking they were very domain specific algorithms and were not capable of generalizing outside their single domain as they relied on hand picked features that provided the largest amount of statistical separation between groups. There

were many different techniques to perform classical detection; however, a common algorithm would be as follows.

First a set of Region of Interests (ROIs) in varying locations and size, seen in Figure 2.17, are selected. Then manual features would be selected and extracted such as Histogram of Oriented Gradients (HOG), seen in Figure 2.18. Next, some sort of linear classifier such as a Support Vector Machine (SVM) would be used to classify each feature vector, seen in Figure 2.19 [14][47]. Lastly the ROI's that were grouped into a particular class of interest would be combined, seen in Figure 2.19, to form a final detection, seen in Figure 2.21. These systems did not scale well since classes needed to be individually separated, resulting in an intractable number of classifiers to train as the number of classes increased.

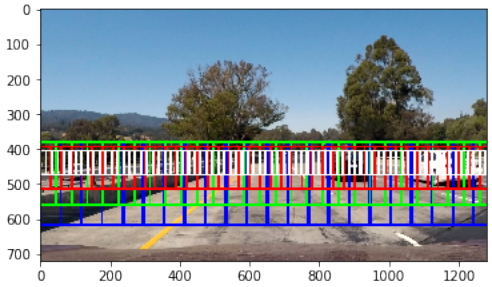

Figure 2.17: Generated ROIs

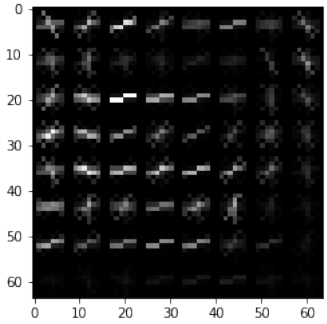

Figure 2.18: HOG Feature Extraction Output

Figure 2.19: ROIs that were correctly classified

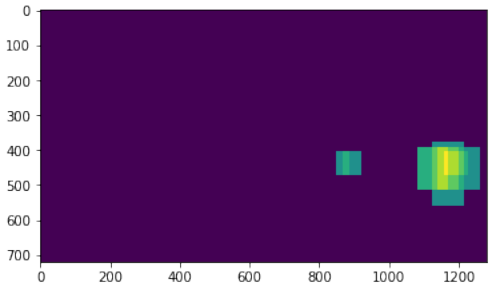

Figure 2.20: Heatmap used to combine detections

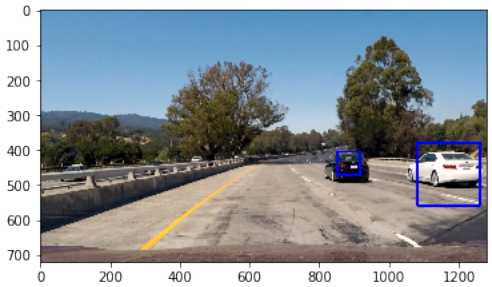

Figure 2.21: Classical Detection Output

2.3.3 Regional Convolutional Neural Network Family

The first major breakthrough in generalized object detection using CNNs came in 2013 with Regional Convolutional Neural Network [23]. The primary difference between RCNN and previous object detection techniques was that the extracted

features were learned features from a CNN instead of hand written. Many different ROIs were passed into the CNN, which extracted features, which were then classified using SVMs. RCNN achieved a 50% increase over the previous state of the art detection algorithm; however, it was very slow due to extracting features multiple times. Also, training was difficult since it was not a unified pipeline as the CNN and all SVMs needed to be trained separately.

The next iteration in the RCNN family is Fast RCNN, which greatly improved the inference speed and ease of training over its predecessor [22]. The speed increase came from extracting features over the entire image once, followed by extracting ROI's from the features which then classified and localized each ROI using two more fully connected layers. Performing feature extraction and using fully connected layers for classification greatly sped up the algorithm and allowed for differentiability, meaning the algorithm could be trained end to end thus eliminating the problem of a multiple training stages.

The last iteration in the RCNN family is Faster RCNN (FRCNN), which only increased inference speed over Fast RCNN [57]. The primary bottleneck in Fast RCNN was the region proposal algorithm used to generate the ROIs, which was addressed in FRCNN by using a novel Region Proposal Network (RPN), seen in Figure 2.22. RPN regressed objectness and offsets using anchor boxes, which are bounding boxes chosen beforehand. These offsets and objectness scores were then used by the ROI pooling of the extracted features. This lead to increased accuracy and a faster algorithm. Multiple RPNs have since been proposed that have either differed slightly from the original RPN or built upon it's principles.

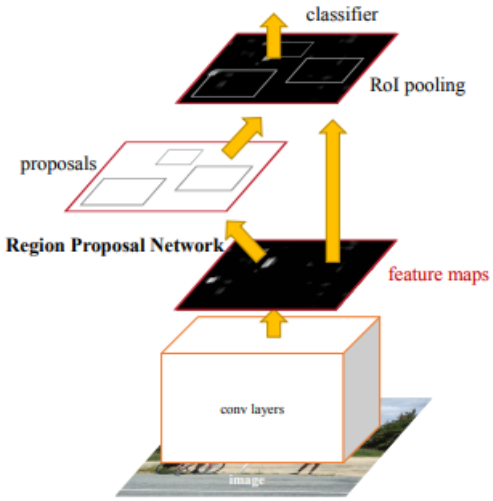

Figure 2.22: FRCNN Network, [57]

2.3.4 You Only Look Once Family

FRCNN used two CNNs, a RPN and a feature extraction CNN which You Only Look Once (YOLO) combined into a single CNN [55]. YOLO was classified as a unified object detector, which both classified and localized objects in a single network pass. This was achieved by discretizing the image into a grid and then regressing the center of each bounding box as an offset from the center of the grid cell as well as the size of each bounding box, shown in Figure 2.23. This greatly increased the speed of the algorithm while still maintaining high accuracy.

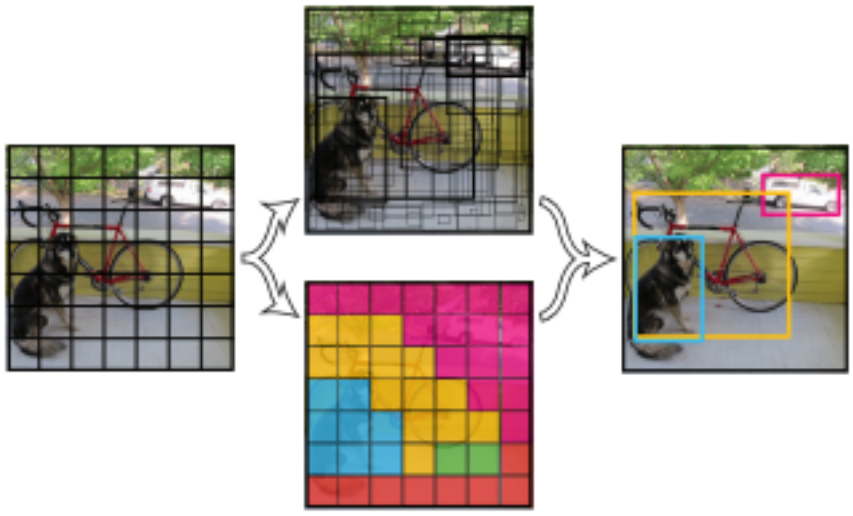

Figure 2.23: YOLO discritization, [55]

It was found that YOLO suffered from large localization error and also exhibited low recall, therefore YOLOv2 was developed [56]. YOLOv2 brought in ideas such as Batch Normalization, which will be discussed in Section 2.4.3, additional fine tuning learning stages, and the use of anchor boxes similar to those in FRCNN. Batch Normalization was added after every convolution allowing the network to converge to a better optima. The additional fine tuning allowed for the network to make a better transition from classification to detection, and the use of anchor boxes decreased localization errors since the network only needed to regress bounding box offsets instead of outright coordinates. The anchor boxes were also intelligently chosen by using kMeans clustering on the dataset's labels. All of these changes yielded a far superior algorithm, which rivaled the accuracy of FRCNN, but was able to operate in real time on a desktop machine.

The YOLO detection networks were developed and released on a custom deep learning framework called Darknet, which is written in C, CUDA, and CUDNN

making it extremely efficient. However it has no ability to perform model optimization after a model has been trained, the importance of which will be discussed in Section 5.4.

2.3.5 Single Shot MultiBox Detector

The Single Shot MultiBox Detector (SSD) was created in between YOLO and YOLOv2 [43]. It was a substantial improvement over YOLO, but performed worse than YOLOv2. Its main improvement over YOLO was that detections of different sizes came from different feature map resolutions. Instead of having a single output to encompass all detection sizes, there are multiple outputs from different stages of the network, each focusing on a different scale of object. An example of this can be seen in Figure 2.24 where the blue bounding box's offsets are determined at a higher feature map resolution than the red bounding box of the dog.

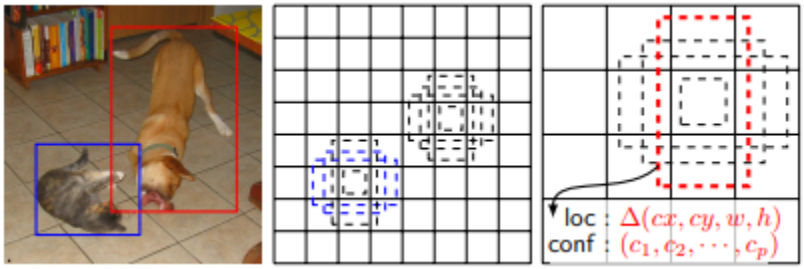

Figure 2.24: SSD Detection Offsets, [43]

2.3.6 Regional Fully Convolutional Network

Regional Fully Convolutional Network (RFCN) made an improvement over FR-CNN by reducing the amount of recomputed features and therefore, performed inference faster [13]. The authors argue that the use of ROI pooling in a detection network acts as a trade off between the feature extractor classification networks

need for translation invariance and a detection networks need for translational variance. Therefore, a large improvement could be made utilizing modern fully convolutional classification networks such as ResNet101 while greatly reducing the unnecessary computation of recomputing features over subnetworks for each ROI. This was achieved by placing the ROI pooling at the end of the network, after all convolutions have been applied, instead of placing convolutional layers after a ROI pooling layer which creates ROI subnetworks. RFCN was able to achieve comparable accuracy with Faster RCNN but with a much higher speed [31].

2.4 Optimization Methods

2.4.1 Momentum Methods

The most common optimizer used is Stochastic Gradient Descent (SGD), which is often augmented with Nesterov Momentum [67]. SGD computes the instantaneous gradient of the current batch and moves in that direction. Momentum is an accumulation of these instantaneous gradients over multiple batches and helps reduce the directional variation of the instantaneous gradient, yielding faster convergence rates than standard SGD. Nesterov Momentum differs from standard Momentum by taking a large step in the old gradient then compute the latest accumulated gradient and apply a small correction step rather than first computing the accumulated gradient and making a large step. The motive being that it is better to correct a mistake after you have made it rather than preemptively correct for a mistake you are about to make. The introduced hyperparemeters that need to be hand tuned are the learning rate and the momentum value.

2.4.2 Adaptive Methods

A large problem with Nesterov Momentum is hand tuning the learning rate and momentum value. Therefore, there are two main adaptive optimization techniques to automatically tune these rates at each iteration for faster convergence. The first technique is the Adam optimizer, which builds upon SGD but with an adaptive learning rate [34]. Adam is a powerful optimizer that yields good accuracy and generally converges more quickly than SGD and converges to a better optima. However the authors of the Yellowfin optimizer claim that a momentum based optimizer with hand tuned parameters will converge faster than Adam, which does not take momentum into account [81]. Therefore Yellowfin was created to not only adaptively change the learning rate but also the momentum value. Yellowfin yields much faster convergence rates on Residual-like networks.

2.4.3 Batch Normalization

While training a neural network, the gradients of each layer are computed with the backpropogation algorithm. However, backpropogation computes instantaneous gradients of each layer assuming that all other layers remain constant. This proves to be problematic because all layers are changed simultaneously according to their gradient. As layers stack on top of one another and are changed constantly, the inputs to the nonlinear activation functions have constantly varying distributions. The variances in these distributions are called internal covariate shift and make learning difficult. Therefore, batch normalization reparameterization is computed, which makes the new mean and standard deviation of the activation's input learnable parameters rather than dependent on all the previous layers [33].

Batch normalization greatly increases the learning process and has been called by LeCun, “”one of the most exciting recent innovations in optimizing deep neural networks” [24].

The batch normalization reparameterization is done by replacing the activation input $\mathbf{X}$ with $\left(\frac{\gamma(\mathbf{X}-\mu)}{\sigma}+\beta\right)$ where γ, μ, σ, and β are the learnable parameters. This normalization step can be combined into a single matrix operation called fused batch normalization. Fused batch normalization refers to the process of combining the multiple kernels that need to be applied in series into one single kernel, while this increases memory usage, it reduces computation time.

2.5 LiDAR Detection

2.5.1 Point Net

Point Net does not perform a detection task, but rather performs a joint classification and segmentation. However, there is no standard for LiDAR processing so the methods described still holds meaning to the detection task [53]. Point Net uses an already detected object's point cloud that has been extracted from the whole cloud, seen in Figure 2.25. The inputed cloud is passed through a MLP to extract a learned feature vector. The feature vector is then run through a symmetric function followed by a transform, which then represents a global signature vector of the point cloud. The global signature vector allowed for a novel 1D representation of point cloud data, which was then combined with each individual point to perform reasoning with both local and global information. This allows for the entire object to be classified and each point to be segmented relative to the entire object.

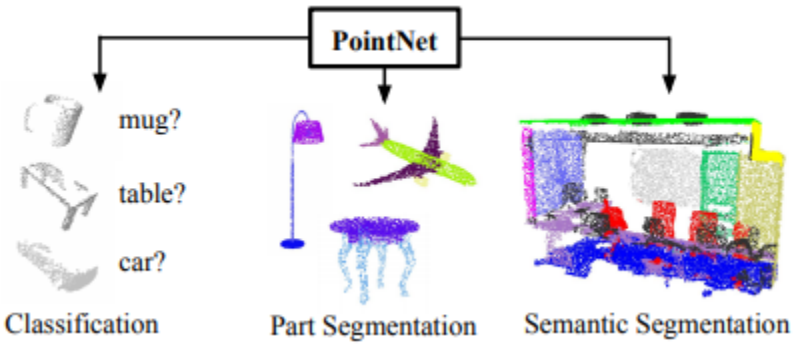

Figure 2.25: Point Net Example, [53]

2.5.2 Voxel Net

Unlike Point Net, Voxel Net performs a detection task on an inputted point cloud and outputs 3D bounding boxes, unlike image detection with outputs 2D rectangles [82]. Voxel Net starts off in the common fashion of forming a voxel grid and then grouping points into their corresponding voxel cell. Voxel Net's novelty is their Voxel Feature Encoding layers, which utilize a combination of hand crafted feature engineering and learned feature extraction on all occupied voxel cells. After each voxel cell has an associated feature vector, a max pooling operation is performed before being passed into 3D convolution and then finally into a custom RPN, seen in Figure 2.26. The 3D convolution is not intractable because 90% of all the voxel cells are empty, making the tensors sparse which allows for more efficient computation to be performed.

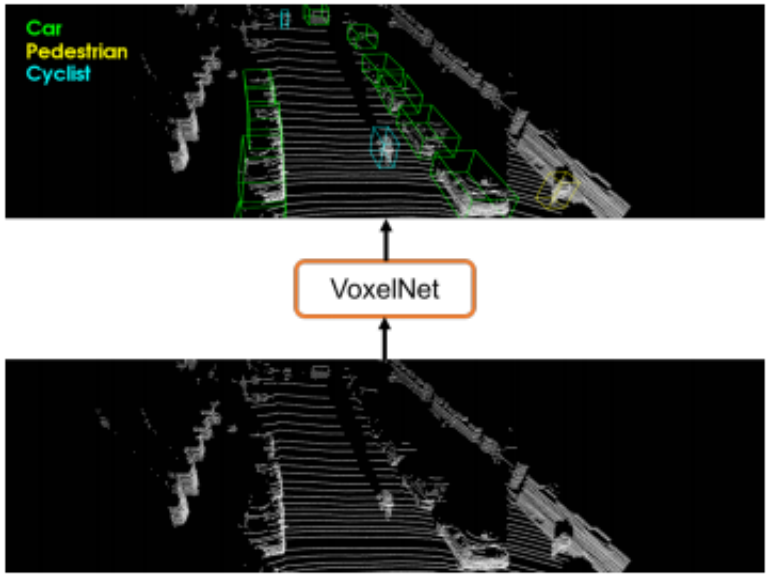

Figure 2.26: Voxel Net, [82]

2.5.3 Vote Family

Vote3D and Vote3Deep both start off in the same manner and utilize the same core voting scheme operation, seen in Figure 2.27 [74][18]. Both algorithms start off by computing a 3D voxel grid, groups points into each cell, and then hand crafted features are extracted from the points in each cell. After the 3D grid of feature vectors is created, most of which are empty sets, the voting scheme which was shown to be equivalent to sparse convolution is employed. Vote for Vote developed the voting scheme, but only performs it a single time. Vote3Deep builds upon the idea, by applying the voting scheme over and over again, which they argue is analogous to image CNNs applying convolution over and over again. This allows for more representational power and lead to more accurate detections.

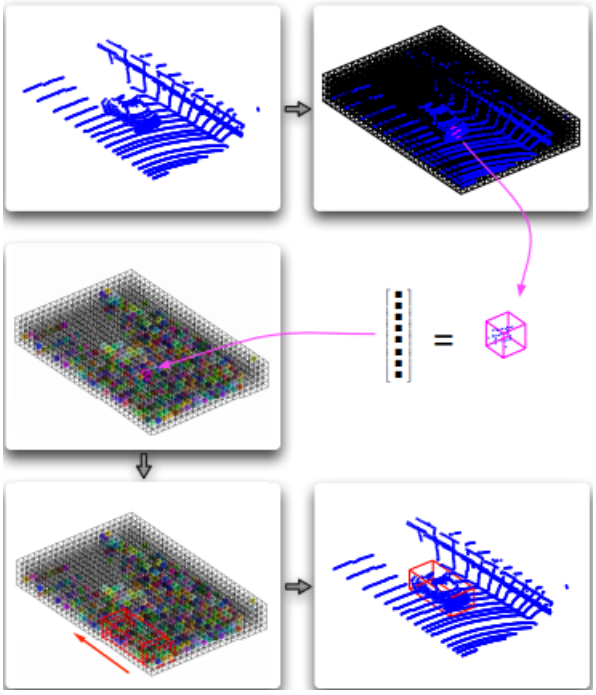

Figure 2.27: Voting Scheme Example, [74]

2.5.4 Sliding Shapes Family

The Sliding Shapes detection algorithm processes a stereo depth map instead of a LiDAR point cloud, however the detection process is extremely similar in this different medium [63]. The algorithm's methodology is straightforward; move a ROI around the depth map and perform feature extraction within that window, after which the feature vector is classified using a set of Exemplar SVMs. Each Exemplar SVM in the set was trained on a single viewpoint of a single object, which means that the size of the set is very large. Although the number of linear classifiers was large and hand crafted features were extracted, it achieved a 1.7% increase in accuracy over other top models in 2014 when it was published.

The next iteration in the Sliding Shapes family, called Deep Sliding Shapes, uses

the more modern approach of a completely learned process, seen in Figure 2.28 [64]. They introduced the novel idea of a 3D RPN, which had been constrained to 2D and also a Object Recognition Network (ORN) that learned both 3D geometric and 2D color features to extract in order to perform classification. Deep Sliding Shapes was 200 times faster than the original algorithm and achieved state of the art performance by over 13.8 mean Average Precision (mAP).

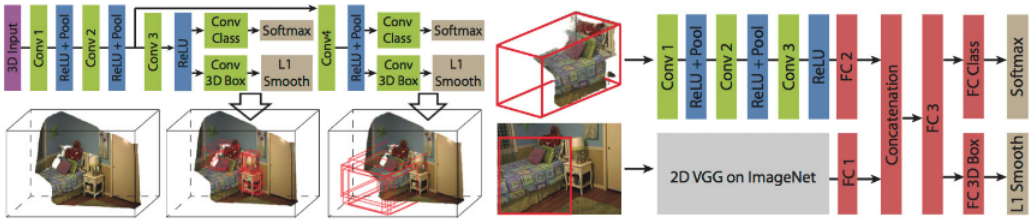

Figure 2.28: Network Used in Deep Sliding Shapes, [64]

2.6 Fusion Methods

2.6.1 Multi-View 3D

Multi-View 3D (MV3D) is a deep learning fusion of LiDAR and camera data [10]. The LiDAR point cloud is projected into both top down views and a front facing view. There are multiple top down view slices, each at different height values, to make an image like tensor different and the front facing view is accordingly projected into a single channel image. The top down view tensor is used to compute object proposals using 2D convolutions. The output of these convolutions are then averaged with features extracted from the front LiDAR view image and the camera image, which are also extracted with their own set of 2D convolutions. These two sets of features combined with the proposals are fused using an element-wise mean, which they call a deep fusion. This network scheme can be seen in Figure 2.29. In

order to combat overfitting they perform dropout on both convolutional paths and also on entire data inputs. In addition, they use auxiliary paths and losses similar to those in GoogLeNet to aid learning [69].

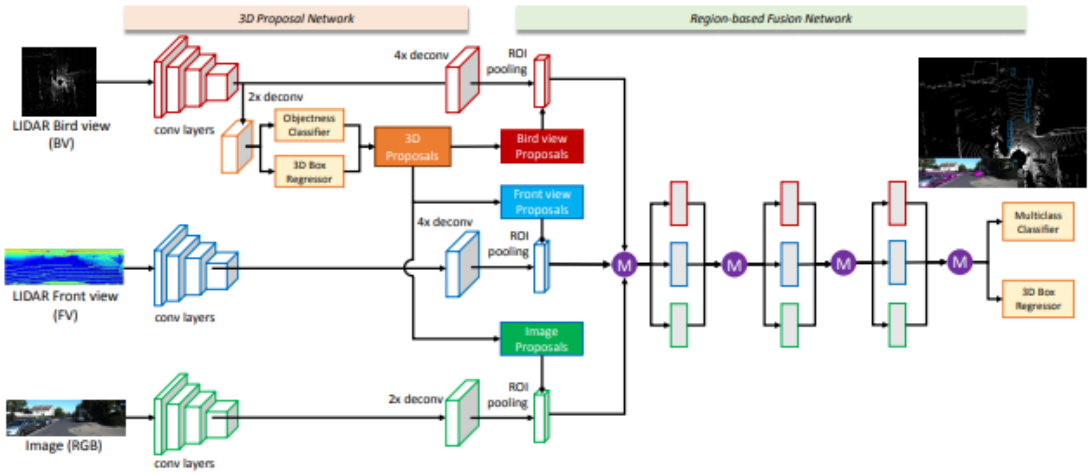

Figure 2.29: MV3D, [10]

2.6.2 PointFusion

PointFusion operates similarly to MV3D, however builds upon it by performing feature extraction on the native 3D point cloud instead of operating on different 2D projections, the network architecture of which is seen in Figure 2.30 [80]. The point cloud has native 3D features extracted by PointNet, discussed in Section 2.5.1; however, all batch normalization layers were removed. These layers were removed because they argue batch normalization removes the effective scale and bias of all features by removing internal covariate shift. However the absolute values of the geometric points are important when processing 3D data. The image is cropped according to the outputs of Faster RCNN, discussed in Section 2.3.3, and then features are extracted from this crop using ResNet101, discussed in Section 2.2.5. The Dense Fusion network then fuses the spatially and visually computed

features via a concatenation followed by a MLP. The Dense Fusion network then predicts 3D bounding boxes relative to known anchor boxes, as is done in 2D object detectors.

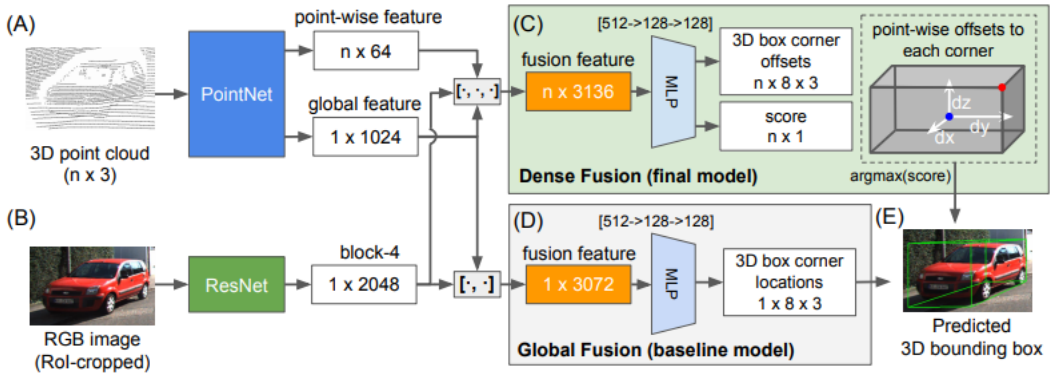

Figure 2.30: PointFusion, [80]

2.6.3 Decision Level Fusion

A Decision Level Fusion algorithm was proposed that combines proposals from the image outputted from a classical segmentation model and supervoxel proposals from the voxelized point cloud outputted from Voxel Cloud Connectivity Segmentation (VCCS), seen in Figure 2.31 [50]. The outputs from each sensor pipeline are processed through independent CNN's, whose outputs are associated using Basic Belief Assignment (BBA), and finally combined using Yagers Combination Rule.

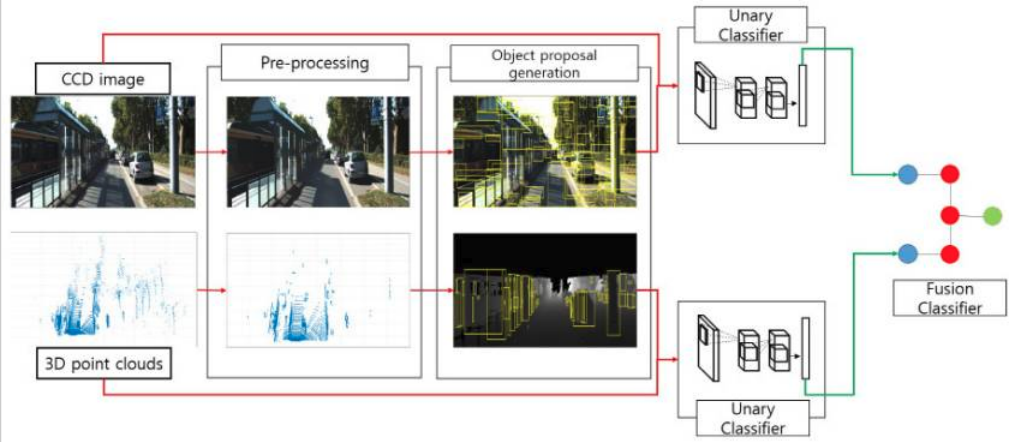

Figure 2.31: Decision Level Fusion, [50]

2.6.4 FusionNet

FusionNet does not operate on LiDAR point clouds but on 3D CAD models. However, it performs a fusion of the 3D voxelization and 2D projection of the models which makes it relevant [26]. There are three different CNNs followed by FusionNet, which combines the outputs of all three previous CNNs, seen in Figure 2.32. The first V-CNN1 performs 3D convolution on the voxelized data, the second V-CNN2 again performs 3D convolution, however concatenates tensors together internally similar to the Inception Module, and finally the third CNN MVCNN, which is largely similar to AlexNet, operates on different projected 2D viewpoints of the model [66]. The FusionNet portion of the network is a fully connected layer that uses the outputs of all previous CNNs as inputs. The fully connected layer acts as a feature fusion and outputs the models classification.

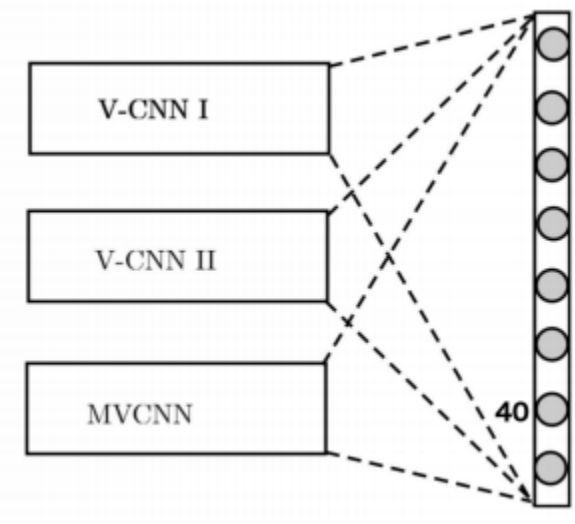

Figure 2.32: FusionNet, [26]

2.7 Software Implementation

2.7.1 CUDA and cuDNN

CNNs are computationally intensive tasks; for example even classification, which is the simplest of tasks, can range from 9 to 31 billion floating point operations [56]. Therefore, it is imperative to exploit 2D convolution in order to make both training and performing inference on CNNs tractable. 2D convolution can be represented as a special kind of matrix called a doubly block circulant matrix and can therefore be run through specific linear algebra routines [24]. CUDA, the general purpose GPU programming language provided the ability for deep neural networks to become reality; however, CUDA could only go so far [37]. cuDNN was developed as a GPU library, built in CUDA, specifically for deep neural networks with specific memory allocation and kernel routines that allowed for up to a 36% reduction in training times [12]. cuDNN was designed to act as a drop in replacement for

previous convolution calls and has become the practical standard for deep learning applications.

2.7.2 Tensorflow

Tensorflow is a machine learning library produced by Google that was released in 2015 after they upgraded their system from DistBelief, which is the platform GoogLeNet was developed in [69][20][16]. Tensorflow was built around scalability and the ability to perform a wide variety of tasks. It allows for symbolic differentiation to reduce compute time, was written in C++ for easy deployment, scales across many machines, and is compatible with both CUDA and cuDNN. Tensorflow relies on computational graphs to represent all operations and sessions to store the computational graph's variable's values. There are a multitude of features that Tensorflow includes such as an execution timeline, model optimizations, and powerful network debugging tools such as Tensorboard. Due to the power and scale of Tensorflow, it has become the most widely used CNN developing tool.

2.7.3 Robotic Operating System

One of the foremost problems in the field of Robotics was the ability to share code. Even the simplest of tasks would require a large amount of effort in order to build up the required codebase, which was time consuming and prohibited researchers from easily sharing their results with one another. Robotic Operating System (ROS) provided the solution to this problem and was built to standardize robotic software development and to facilitate easy code reuse [54]. ROS features a universal build system, a message passing interface, and powerful diagnostic and visualization tools. ROS is based around modularity in order to allow for many

features of code to be reused between projects. This results in many Unix processes being required for any robotic task, the management of which is seamlessly handled by the ROS master, even across a network over many machines.

2.7.4 Point Cloud Library

The Point Cloud Library (PCL) was largely built during Radu Rusu's doctoral dissertation for use on point cloud data and later finished at the company that produces ROS [60]. PCL is a library that contains functions for processing and manipulating different types of point clouds and is the point cloud analog to image processing's OpenCV. Since point clouds are very representationally powerful, it is common that robots will have sensors that generate point clouds such as stereoscopic cameras or LiDARs. Therefore, ROS comes precompiled with it's own version of PCL and it's other packages interface seamlessly with the library.

2.7.5 Object Detection API

As another piece of the Tensorflow library, Google published an object detection API in order to directly compare different detection models [31]. As stated in the paper, many detection networks were developed on different platforms and written in different libraries, making it difficult for direct comparisons between different algorithms. Therefore a team at Google built a unified detection platform using Tensorflow and were able to train and test many different networks and variations. They were able to determine the most optimal combination of parameters in the detection networks, which culminated in building an ensemble of these currently available networks and achieved state of the art performance.

Chapter 3

Multimodel Fusion Detection System Overview

3.1 Overview of Proposed Algorithm

The proposed Multimodel Fusion Detection System (MFDS) is a decision-feature fusion, seen in Figure 3.1 and formally described in Algorithm 1. At the start of the algorithm, an image based CNN performs detection to output a set of possible objects represented as bounding boxes with classes and probabilities. After which, the synchronized point cloud is transformed into the same coordinate frame as the camera, the ground plane is removed, and the remaining points are separated into clusters based upon Euclidean distance. Then image detections and clusters are associated and paired together. Next a vector of features is extracted from the cluster of each pair, if there are any, and run through a MLP to regress class probability, object length, distance to the object's center, and orientation of the object. The pairs confidence's are adjusted, or could potentially be removed, based

upon the output of the MLP. The output of the fusion algorithm are confident 3D localized detections.

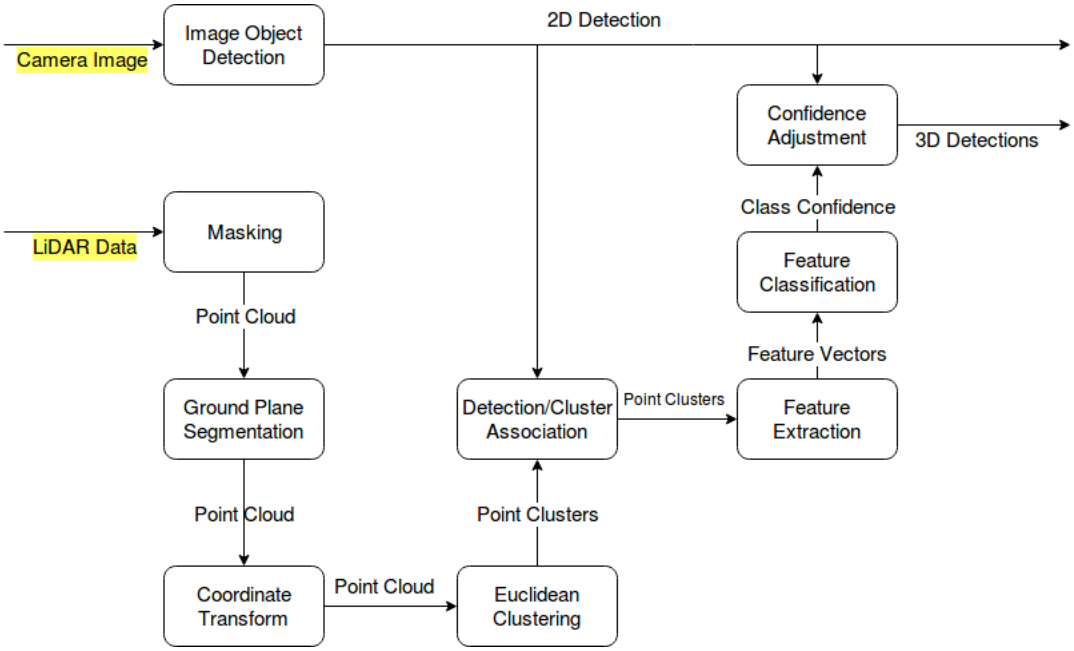

Figure 3.1: System Diagram for Fusion Algorithm

3.2 Creation of Fusion Models

The first step was acquiring a pretrained image detection CNN model, specifically SSD, and then performing transfer learning on them to be fine tuned on the KITTI dataset. The KITTI dataset was split to create training and validation sets, out of the 7481 images and point clouds, 5611 were used for training and 1870 were used for validation. Next, the LiDAR processing pipeline was built up to the feature extraction of identified clusters stage. The converted LiDAR dataset was then created by running every LiDAR point cloud in the KITTI dataset through the processing pipeline to extract features of every cluster, which were then compared to KITTI's labels and saved to disk if they matched. After the converted dataset was formed, the classifying MLP was built and trained.

3.3 Detection Fusions

3.3.1 Association Problem

In order to perform the fusion algorithm after trained models were created for the image detection CNN and LiDAR classifying MLP, image detections and clusters needed to be associated together since each are potentially noisy sensor readings detecting the same object. The projection matrix from 3D LiDAR space to 2D camera image space is provided for every timestep in the KITTI dataset. The projection matrix allows to compute the pixel location for each 3D point within the cloud, seen in Figure 3.2. The projection can be computed below:

Algorithm 1: Fusion Algorithm

Result: Set $\mathbf{D} \; \forall \; \mathbf{D}_i =$
$\{\text{Class}_i, \text{Confidence}_i, x_i, y_i, z_i, l_i, w_i, h_i, \{x_{1,i}, y_{1,i}, x_{2,i}, y_{2,i}\}\}$

Initialization: Image CNN model $\equiv \mathbf{F}(x)$, Point Cloud Feature MLP model $\equiv \mathbf{G}(x)$;

Input: Image, Point Cloud;

begin

- Set $\mathbf{B} = \mathbf{F}(\text{Image}) \; \forall \; \mathbf{B}_i =$ $\{\text{Class}_i, \text{Confidence}_i, \text{Bounding Box}_i = \{x_{1,i}, y_{1,i}, x_{2,i}, y_{2,i}\}$;
- Mask(Point Cloud);
- Planar RANSAC Segmentation(Point Cloud);
- Coordinate Transform(Point Cloud);
- Clusters $\mathbf{C}$ = Euclidean Clustering(Point Cloud);
- Set $\mathbf{X} = \{\}$, $\mathbf{D} = \{\}$;
- **for** $b \in \mathbf{B}$ **do**
  - **for** $c \in \mathbf{C}$ **do**
    - x_c, y_c, z_c = Centroid(c);
    - u, v = Projection(x_c, y_c, z_c);
    - u_c, v_c = Centroid(b);
    - **if** $\sqrt{(u - u_c)^2 + (v - v_c)^2} < Threshold$ **then**
      - Append {b, c} to $\mathbf{X}$;
    - **end**
  - **end**
- **end**
- **for** $x \in \mathbf{X}$ **do**
  - $b = x.b$;
  - $c = x.c$;
  - f = Feature Extraction(c);
  - Class, Confidence, z, $l = \mathbf{G}(f)$;
  - **if** $b.Class = Class$ **then**
    - b.Confidence += .5;
    - **if** $b.Confidence > 1.0$ **then**
      - b.Confidence = 1.0;
    - **end**
    - x, y, w, h = Projection(b);
    - Append {Class, b.Confidence, $x, y, z, l, w, h, \{b.\text{Bounding Box}\}$ to $\mathbf{D}$;
  - **end**
- **end**
- Non Maximal Suppression($\mathbf{D}$);

end

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \mathbf{P} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{3.1}$$

$$\mathbf{P} \in \mathbb{R}^{3,4} \tag{3.2}$$

After the matrix multiplication has been computed, the output vector needs to be normalized by the last element so it becomes a 1 and the projection from 3D LiDAR to 2D camera image space is complete. After the projection had been developed, clusters mean x, y, and z coordinates were projected into image space, along with forming the centroids of each of the CNN detection's bounding box. With the two sets of projected cluster centroids and bounding box centroids, a simple Euclidean distance comparison in pixel space was performed. Any pair of centroids that were under a certain threshold were considered to to be a pair of potentially the same object. The threshold that was used was 75 pixels due to the large variance of the cluster centroid's means.

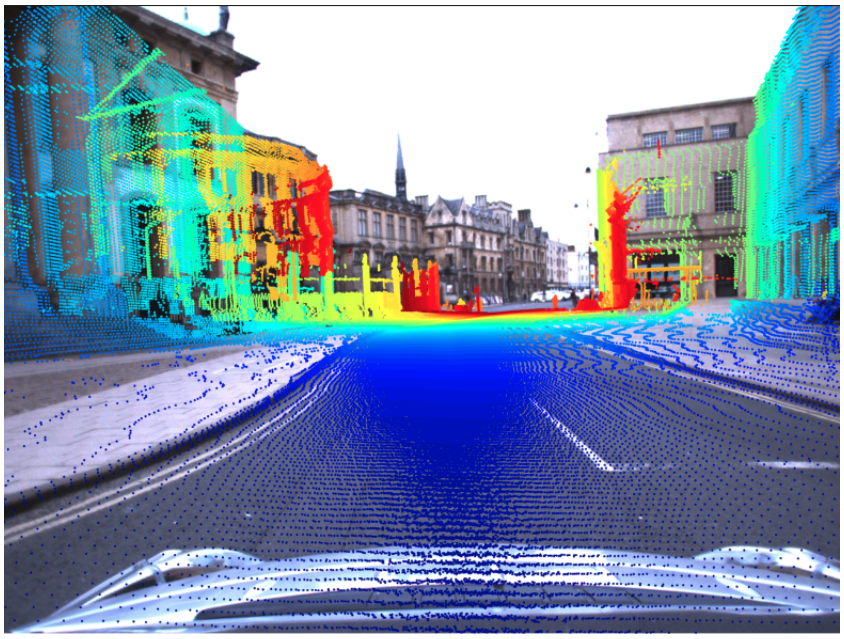

Figure 3.2: LiDAR Projection to Image Pixel Space, [46]

3.3.2 Confidence Adjustment

After each pair's cluster had been classified, a check was performed between the pair's image detection class and cluster classification class. If the classes were not the same then the detection was eliminated, however if the classes were the same then the detections confidence was increased by 50% because half of the confidence comes from each detection method. If a detections confidence was above 1 it was set to 1 to maintain the laws of probability. This confidence adjustment allowed for uncertain image detections, which would be eliminated, to gain the required confidence to be considered true detections.

3.4 MFDS Deployment Details

The fusion algorithm was be jointly implemented in Python and C++ using ROS as the communication and build platform. ROS serves multiple purposes in this code; first to allow for messages to be passed with between the two languages, second for the use of powerful debugging tools such as topic monitoring and it's visualization package RVIZ, and third because MFDS needs to be easily deployable on robotic systems.

MFDS requires two ROS nodes to operate concurrently, seen in Figure 3.3; the first of which is a Python node to perform image detection and the second is a C++ node to perform point cloud manipulation, classification, and fusion. The Python node is subscribed to a Compound Data Message (CDM) containing both an image and a synchronized point cloud. Once the CNN finishes computing detections, another compound message is formed containing the detections and the CDM, which is published to the C++ node. The C++ node performs all point cloud preprocessing, cluster formation, cluster detection pairing, feature extraction, feature classification, and fusion. The output of the C++ node is the final set of fused 3D localized detections. Python was chosen the image detection CNN because of Python's easy access to the Tensorflow library eliminating the need to write the network inference code. C++ was chosen for the fusion node because PCL only operates in C based languages.

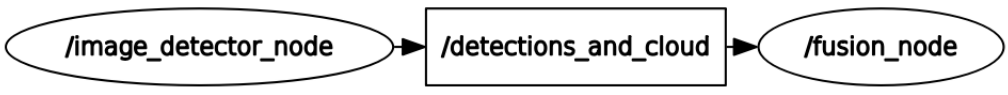

Figure 3.3: Fusion Algorithm's Compute Graph

Since the LiDAR classifing MLP was trained with Tensorflow in Python but inference was performed in C++, the MLP needed to be ported over. Since the network involved no convolutions and only a simple MLP, the cuDNN library did not need to be called. A MLP is as a series of matrix multiplications followed by nonlinear activation functions so naturally an efficient, large scale matrix multiplication library was required. CUBLAS was decided upon over Eigen due to the size of the matrices that needed to be multiplied. CUBLAS is a library of CUDA that operates similarly to the Basic Linear Algebra Subprograms (BLAS) library in C++ however CUBLAS performs the same operations on a GPU instead of a CPU. The CUBLAS function call Sgemm that is used to implement the MLP is the same function call that MobileNets optimized their depthwise seperable convolutions around [29].

Chapter 4

Classification Network Construction

4.1 Network Inspiration

A substantial amount of time must go into crafting a small in memory, fast network that has a high level of accuracy for the classification task. This means that not only is the depth of the network constrained, but so is the width of each layer. Dense and residual networks have seen extreme success not only in classification, but also detection tasks; making them the ideal starting point for a classification network. However, as is the standard for such nets, they are extremely deep and therefore needed to be approached with caution since their inference speed is very slow. Due to the large amount of training time required, all initial classification prototype testing was performed on CIFAR10 to greatly increase the number of network variants and ideas that could be tried in a reasonable amount of time.

4.2 Training Platform and Method

The machine that training was performed on had 32GB of RAM for the CPU to access and a set of three Nvidia 1080 GPU's, each with 8GB of VRAM for an effective memory of 24GB. However, due to the multiple GPUs the training pipeline needed to be altered to allow for the network to learn properly. Normally a Tensorflow graph will store the parameters on whatever the default device is, which is the first GPU if one is installed. However, this would mean that each GPU had an individual set of weights or variables which is not optimal. Therefore, a single set of weights was stored on the CPU. A copy of the set of variables was received by each GPU at the beginning of each iteration when an individual batch was pushed from the CPU to GPU. After the batch and fresh weights were received, each GPU performed inference, computed the loss, and backpropogated the gradients. After the gradients on each GPU had been computed, they were copied back to the CPU, averaged, and then were used to perform a single update, shown in Figure 4.1. The use of multiple GPUs is seen as increasing the effective batch size, which allows for the use of higher learning rates and accelerated training.

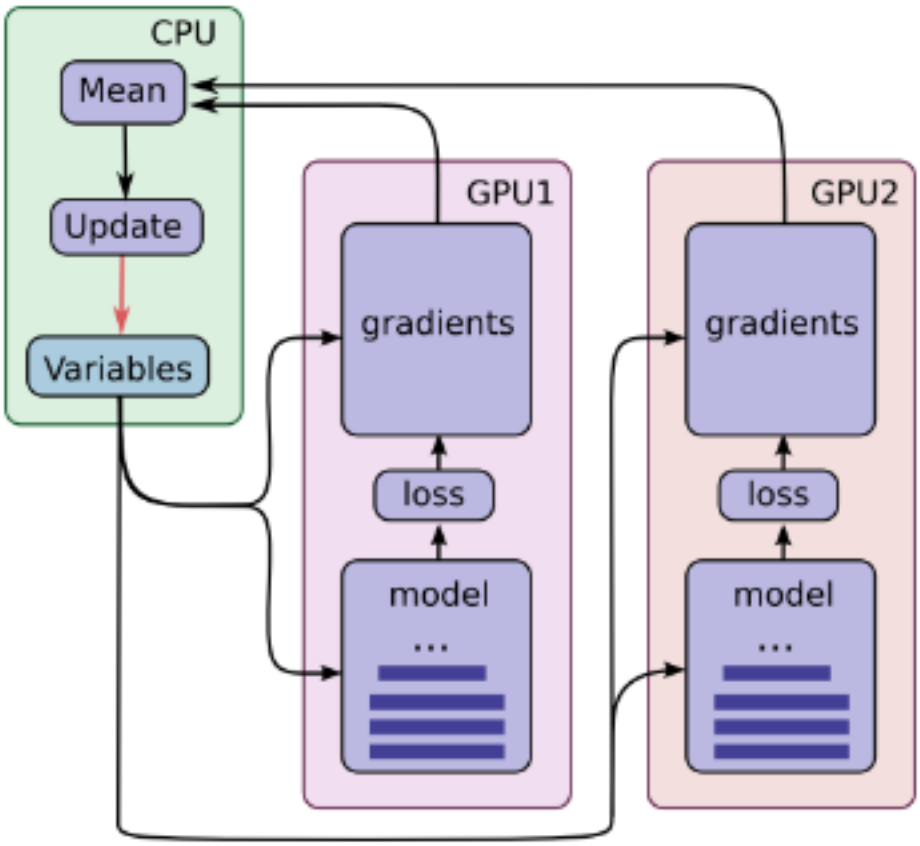

Figure 4.1: Example of Asynchronous Model Training, [2]

4.3 First Network Iteration

DenseNets is currently one of the most recently influential classification networks and was therefore used as the starting point for first iteration of the classification network's development, rather than a residual network [30]. The first classifier was heavily based upon DenseNet40, growth rate k=12 and as a parallel to Inception-ResNet-v2, the Inception module from GoogLeNet was substituted for every convolutional layer [68] [69]. The Inception module was very successful with residual connections and therefore similar success could be seen with Inception modules

and dense connections. The depth output of each dense block was 24, 36, and 48. Since there are concatenations of feature maps within both the Inception modules and the dense connections, the depth of the tensor passing through the network drastically increased in size. Therefore, the number of learned dense layers needed to be dropped from 40 to 20 in order to fit into memory. Even though the width was greatly increased, the reduced depth severely limited the representational power of the network by only achieving 76.85% accuracy.

4.4 Second Network Iteration

A major difference between DenseNets and residual networks is the way their features are combined. DenseNets perform a concatenation of feature tensors while residual networks perform an addition. Addition instead of concatenation would reduce the tensor size but still allow for similar benefits of dense connections. However since the tensor depth would be drastically reduced, the base growth rate needed to be increased from 12 to 16. Also, an idea from the YOLO architecture was borrowed in order to increase the depth of the tensor [55] [56]. The idea is to transform a tensor by taking its spacial features and reorganizing them into depth features. This change spatially reduces the size of the tensor by a factor of 2 in both dimensions and increases the depth by 4. This resulted in needing the growth rate to be quadrupled, instead of doubled; the tensor's depth during each dense block was 64, 256, and 1024 respectively. This increased tensor depth made the memory footprint the same as the previous network's, meaning that a deeper or wider network had not been created. This network variant ended up performing similarly to the previous discussed network and converged to only a slightly higher

maximum accuracy. However, this network iteration converged in roughly 10 times fewer training iterations than the previous network iteration. This was consistent with the claims made by the authors of Inception ResNet v2 that the primary role of residual connections is actually not increased accuracy, but decreased training time [68].

4.5 Third Network Iteration

Although YOLO was a successful detection network, the spacial-to-depth transformation has not been used any where else, therefore its benefit to the networks accuracy was questionable. This was addressed in the third network iteration and the spatial transformation operations were replaced with transition layers similar to those of DenseNets, an average pooling and convolution. However, the operations were reordered in the transition layer to be average pooling first, followed by the convolution due to the feature maps being spatially reduced by a factor of 2 before the depth projection is performed. A reduction in computation is seen due to this change. The depth projection in the transition layer increased the incoming depth by a factor of four in order to act as a drop in for the spacial to depth transformation. In addition to all these changes, each convolution was replaced with depthwise separable convolutions used in MobileNets due to their decreased computation [29], seen in Figure 2.12. However, the depthwise convolutions increased training time and were abandoned and replaced with standard convolution. This network variant was able to converge to a slightly higher accuracy than the previous version.

4.6 Fourth Network Iteration

The previous network iteration had performed reasonably well on CIFAR10, achieving roughly 82% accuracy, which was deemed accurate enough to train on the ImageNet dataset for the first time. A greater explanation of the ImageNet dataset can be found in Section 2.1.3. Since the image area increased by a factor of 49, the memory demands of the network were significantly increased, which needed to be compensated for by reducing the batch size to 4. This reduction allowed the network to fit into memory. This network achieved poor performance after training for roughly 24 hours and the training operation was terminated. It was assumed that the small batch size was the primary problem with the poor accuracy, so a fourth dense block was added, as was done in the DenseNets paper, allowing for another average pooling layer to be included in the network. This allowed the feature maps to decrease in spatial dimensions and reduced the memory footprint enough to use a batch size of 16 even with the added filters. This first network iteration on ImageNet achieved roughly 24% top 1 accuracy, over 10% better than the three block counterpart, but was still not acceptable. The poor accuracy was believed to be caused by the very small number of feature maps being outputted after each convolution.

4.7 Training Pipeline and Network Inspection

Since the network had such poor accuracy on ImageNet, training was reverted back to CIFAR10. One of the largest problems with the different iterations up until this point was that multiple aspects of the network were changed at a time, making it difficult to see the exact effect of each change individually. Therefore

an overhaul of many different aspects were performed one by one to determine the results of each change. These included the number of GPU's used in training to change the effective batch size per iteration, the input image data range (0:1 or -1:1 for each pixel value), different preprocessing techniques, using different optimizers, using dropout, and fusing the batch normalization parameters. Tensorboard, Tensorflow's main debugging tool was utilized extensively during this period, seen in Figures 4.2, 4.3, and 4.4.

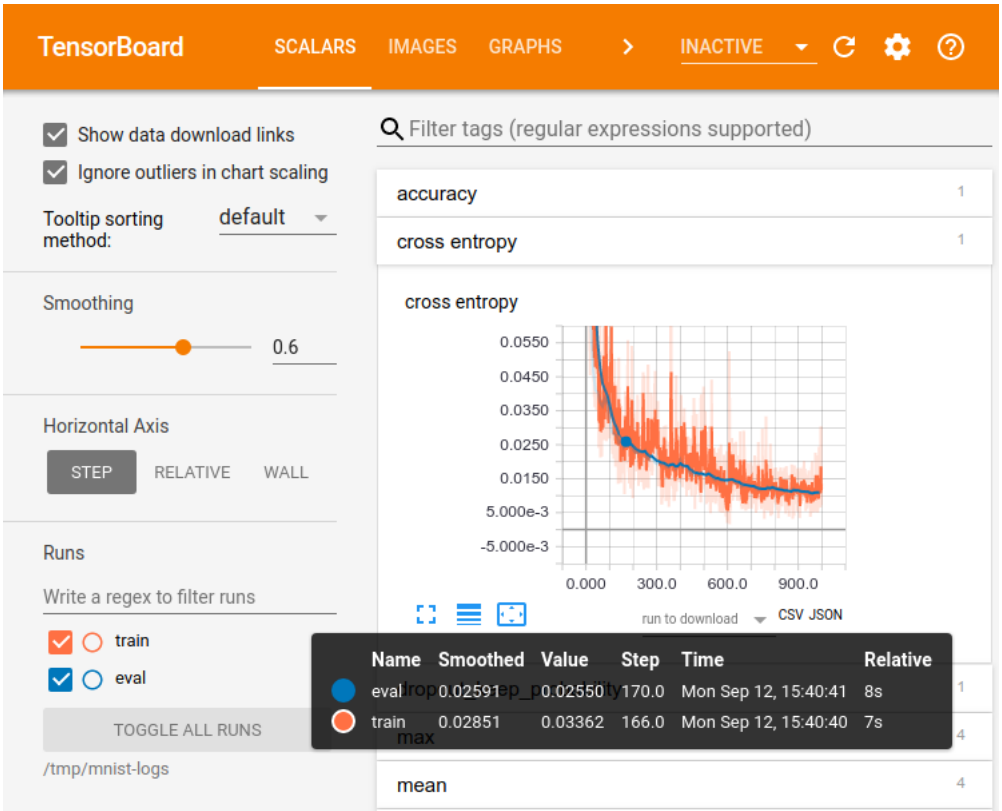

Figure 4.2: Example of Tensorboard's Scalar Tracker

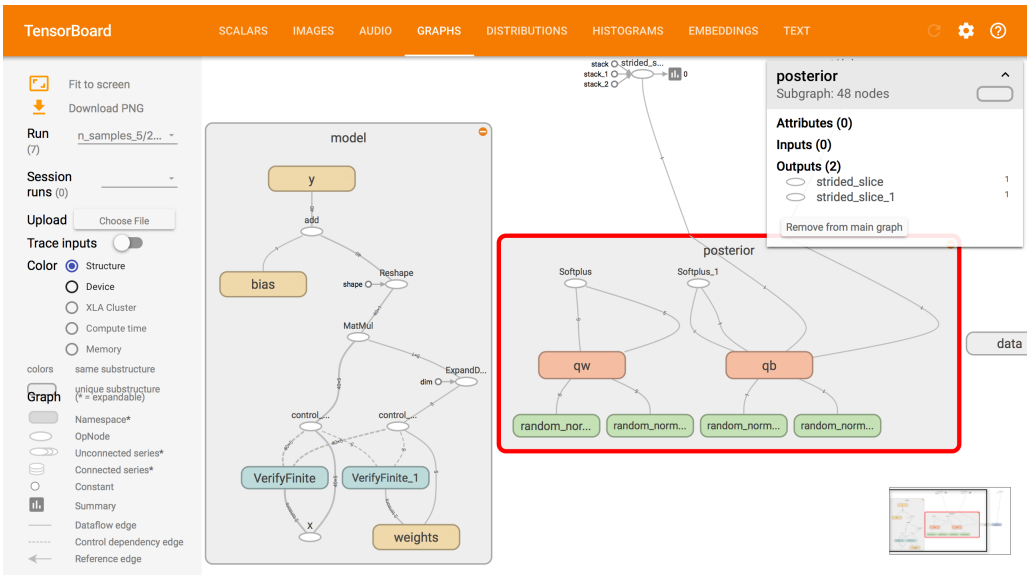

Figure 4.3: Example of Tensorboard's Graph Visualizer

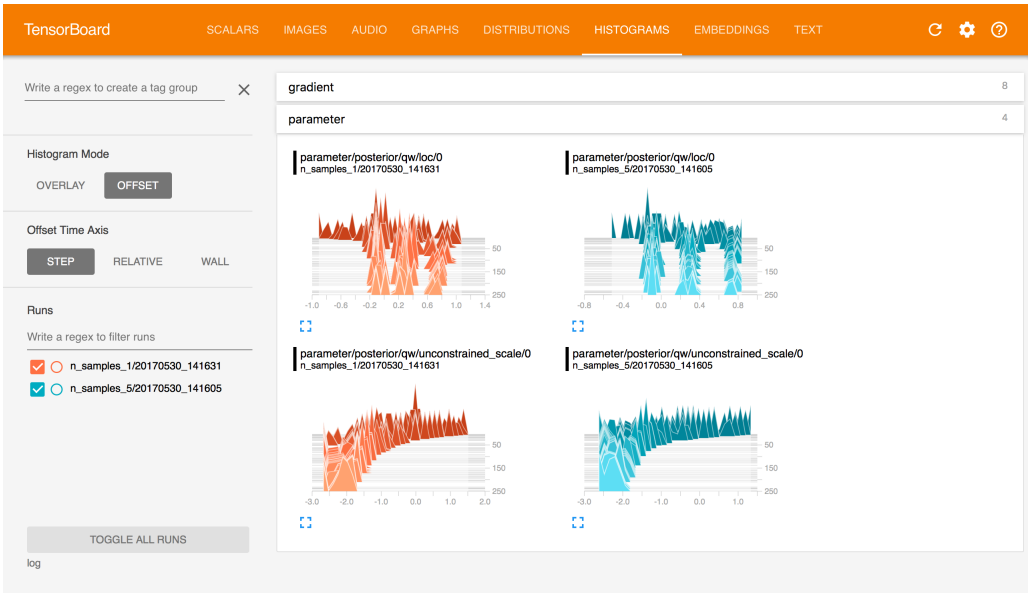

Figure 4.4: Example of Tensorboard's Variable Tracker shown as distributions

The number of GPU's used during training varied between one and three and was found to have no effect on accuracy. Therefore, three were used for the remaining duration as it reduced the training time by a factor of three. The accuracy

increased by 4% when the input image range 0:1 was used. Performing a per image standardization was also 4% worse than performing a dataset standardization, which was used moving forward. A comparison between the supposed superior Yellowfin optimizer and standard stochastic gradient descent (SGD) was performed and SGD converged to 17% higher accuracy. As a result, Yellowfin was abandoned. This is most likely due to the the claim that Yellowfin is targeted at improving residual networks, which is slightly different than this network's architecture [81].

Testing different dropout probabilities resulted in slight accuracy increases, however this could be matched by performing more data augmentation [27]. Therefore the dropout probability was set to 0, meaning no path would be eliminated. This also resulted in removing the ensembling effect of dropout from every layer in the network. Performing fused batch normalization greatly reduced training time due to the more effective kernel usage, even though more memory was consumed and a smaller batch size needed to be used [33].

Next, Tensorflow's timeline object was consulted, which shows the computation time associated with each operation on each device, seen in Figure 4.5. It was found that computing the gradients associated with the dropout operation took a very large amount of time relative to an entire training iteration. Therefore, since dropout was already not being used but was still in the computational graph, all dropout operations were removed along with a few other spurious operations. Not only did this reduce the training time per iteration, but also drastically reduced the memory requirements and allowed batch sizes of an order of magnitude larger than previously used to fit into memory during training. This change greatly reduced the training time.

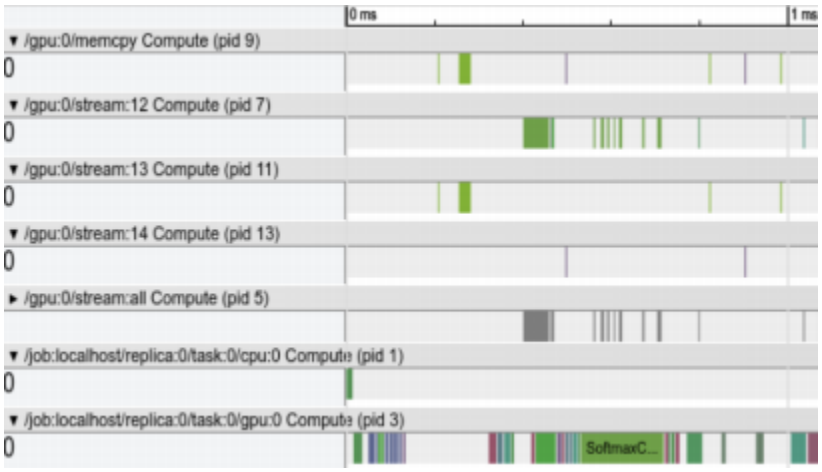

Figure 4.5: Tensorflow's Timeline Object

The greatest reduction in training speed came from the use of asynchronous preprocessing and transfer to the GPU's on the CPU. The training pipeline was set up to synchronously load each batch into memory, perform preprocessing, push the batch onto the GPU's sequentially, perform forward inference on the batch, backprogate error through the network, transfer the gradients from the GPU's back to the CPU, average the gradients for each kernel over all the GPU's, perform kernel optimization and then finally load another batch of images into CPU memory.

The greatest reduction in training speed came from the use of asynchronous preprocessing and transfer to the GPU's on the CPU. The training pipeline described in Section 4.2 was completely sequential requiring one operation to be completed before the next could begin, which was an extreme bottleneck. The training pipeline was first altered to use a system of queues that were asynchronously filled by the CPU using multi-threading, seen in Figure 4.6. The queues acted as buffers to again asynchronously preprocess images. The preprocessed images were then dequeued to form batches that were ready to be pushed to the GPU as soon as the previous weight update was performed. These batches were able to pushed to all

GPU's simultaneously due to Tensorflow's multi-threading capability. This switch from all sequential to all asynchronous CPU processing increased the training speed by over a factor of two.

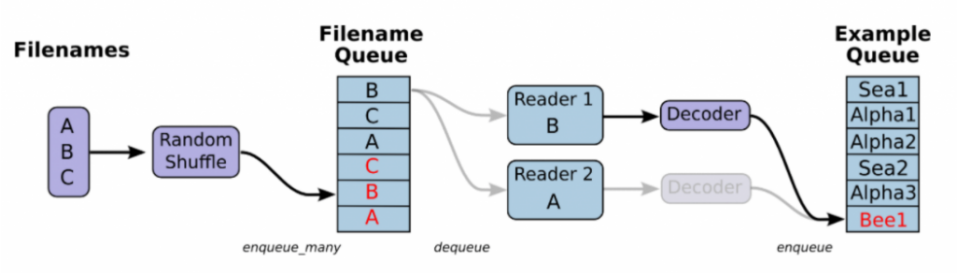

Figure 4.6: Tensorflow Queue and Threading, [1]

After all these improvements were made, the network was able to achieve 88% accuracy, only a 6% improvement than before on CIFAR10 but the expected training time on the ImageNet data for 150 epochs dropped from slightly over a month to only 13 days.

4.8 Last Network Iteration

With the new training pipeline and network architecture, training was returned to ImageNet. Rather than use a batch size of four images per GPU, 64 images per GPU were used and each training iteration was faster than before, despite the substantial increase in batch size. This network achieved a 59% top 1 accuracy and 80% top 5 accuracy after training it for 13 days for the full 150 epochs. Although this was a tremendous step forward, it still fell short of AlexNet's top 1 and top 5 error rates by roughly 3% [37]. The poor results were due to the lack of depth of the network. A deeper network was attempted by increasing the number of layer by a factor of 2 with a corresponding reduction in width; however, the batch size

needed to be reduced, which made training time intractable.

The concatenation operation is extremely memory inefficient. This requires very small batch sizes to be used and made training times intractable. The authors of DenseNets had two solutions to this problem. The first was to train these networks with a larger server setup with dozens of GPUs training asynchronously on multiple CPUs. The second was to publish a second paper about a memory efficient implementation which achieved state of the art accuracy [52]. The current version of Tensorflow has an especially large problem with memory allocation of concatenation and is a current area of improvements for future version updates. It should be noted that the Authors built DenseNets in both PyTorch and Caffe, where the memory allocation problem is not as significant.

It was decided to abandon a novel classification network and instead use a pretrained model. This was decided upon because of time constraints on the research as building a novel classification network is not mandatory for the fusion algorithm. The pretrained classification networks that were decided to be used were MobileNets, Inception ResNet v2, and ResNet101, all of which were trained on ImageNet [29][68][25]. These three networks were chosen because they encompassed a wide variety of accuracy and inference speed tradeoffs with MobileNets being the fastest and least accurate, Inception ResNet v2 was the slowest and most accurate, and ResNet 101 being in middle of the two. All of these networks were acquired in the publicly available Tensorflow Slim model zoo. Tensorflow Slim is a streamlined higher level library within Tensorflow's Contrib library, which was built to eliminate most of its boilerplate code.

Chapter 5

Detection Network Development

In addition to using pretrained classification networks, it was decided to also use a pretrained detection network. This was deemed acceptable since the detections outputted from any CNN is usable in the fusion algorithm and does not affect the novelness of the fusion. SSD was the detection network that was chosen to be used in MFDS and RFCN and FRCNN were included as references to test MFDS' ability to utilize different CNN networks. Pretrained versions of these networks were found in the publicly available Object Detection API from Google.

5.1 Selected Networks

Similarly to the classification networks, these detection networks were chosen because they span a range of accuracies and inference speeds. SSD is the fastest and least accurate, while FRCNN is the slowest but most accurate and RFCN is in the middle of the two.

Each of these detection networks had a different feature extractor or region pro-

posal generator depending on the detectors architecture. The SSD model used the MobileNets classification network, the RFCN model used the ResNet101 classification network, and the FRCNN model used the Inception ResNet v2 classification network, [68]. This pairing was chosen to align networks with similar speed and accuracy with one another, seen in Figure 5.1.

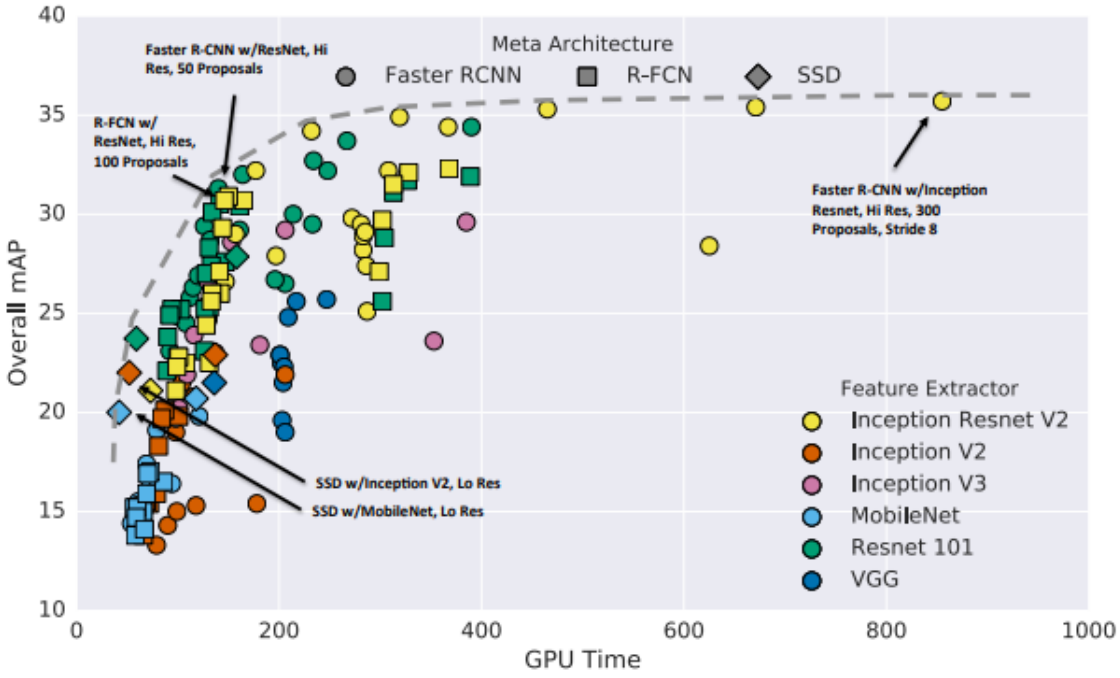

Figure 5.1: Comparison of Detectors, [31]

The detection networks in the model zoo are trained to detect 80 different classes in the COCO dataset, much of which are irrelevant to roadways and would therefore perform very poorly for this fusion algorithm as it was designed specifically for an autonomous vehicle's perception system. However, the convolutional kernels in the networks have been trained to pick up general visual cues that can be retrained to detect other features quickly and with high accuracy through transfer learning. Each network was retrained on the KITTI dataset, which allowed for fast

training on each network while retaining high accuracy due to transfer learning.

5.1.1 Training Details

The training details were all set to the defaults of the Object Detection API [31]. The association of a predicted detection and an anchor box was performed by a many-to-one matching strategy and Jaccard Overlap, or IOU. Box encodings were performed by the function:

$$\phi(b_a, a) = [10\frac{x_c}{w_a}, 10\frac{y_c}{h_a}, 5log(w), 5log(h)] \tag{5.1}$$

The Loss function that was optimized was defined in Equation 2.2. L_{loc} was set to be the Smooth L1, or Huber, loss and L_{class} was set to Cross Entropy [32]. Both of the loss weights, λ_{loc} and λ_{class}, were set to 1 and hard negative mining was also used. Each network trained for 84 thousand iterations and outputted 50 detections in order to reduce computation by reducing the dimensions of the output tensor.

5.2 SSD Model

The model used was a MobileNets feature extractor combined with the SSD detection network. To fine tune this model, a batch size of 12 was split across three GPUs. The optimization method used was RMSProp. Data augmentation was performed by random horizontal flips.

The loss curve shows that the SSD model was challenged to be able to perform accurate detection on the KITTI dataset, seen in Figure 5.2. It is believed that

the primary reason for this is that unified models such as SSD have a difficult time detecting small objects, meaning relatively far enough away objects will most likely not be detected well. Generally speaking, a detection will occur in the correct location and the correct class, but the confidence in the detection will be very low. The low confidence will be wiped out by the threshold placed to eliminate potential false detections, a common threshold value is 60%. This effect is readily visualized in Figure 5.3 where the farther away the object of interest appears, the lower the confidence becomes.

The bounding box colors are determined by what class the detection is in Figures 5.3, 5.5, and 5.7. Blue boxes represent cars, green represents pedestrians, and red cyclists. Training loss for each network is reported in Figures 5.2, 5.6, and 5.4 with training iteration on the x axis and loss on the y. Since a burn in period was not used, the loss at the start of training is large and showing it on the graph would take away from the importance of later iterations so it is omitted.

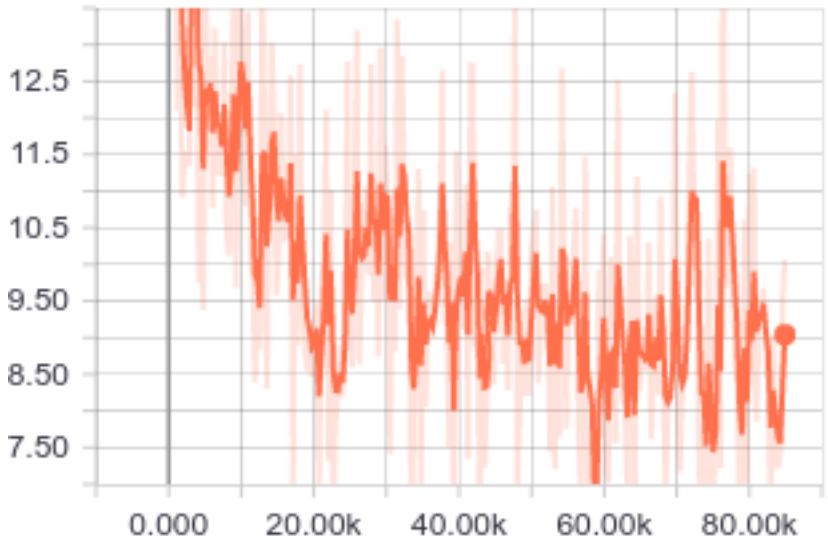

Figure 5.2: Total loss of SSD Model

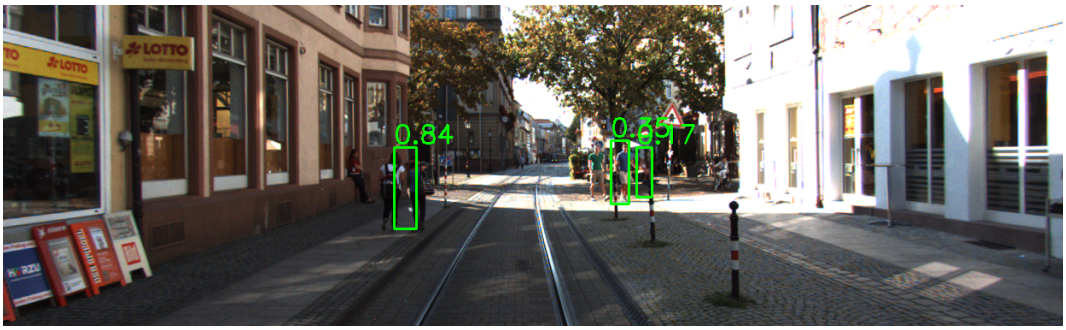

Figure 5.3: Example Output of KITTI Trained SSD Model

5.3 Reference RFCN and FRCNN Models

The RFCN model used was a ResNet101 feature extractor combined with an additional detection portion of the network. To fine tune this model, a batch size of 9 was split across three GPUs. The optimization method used was Nesterov Momentum and the initial learning rate was 0.0003 and momentum value of 0.9. Data augmentation was performed by random horizontal flips.

The loss curve shows that the RFCN model was able to sufficiently learn the KITTI dataset, seen in Figure 5.6. The model had enough representational power to detect most objects correctly and with high confidence. RFCN is not a unified model and therefore does not struggle with distant objects like SSD does, seen in Figure 5.5.

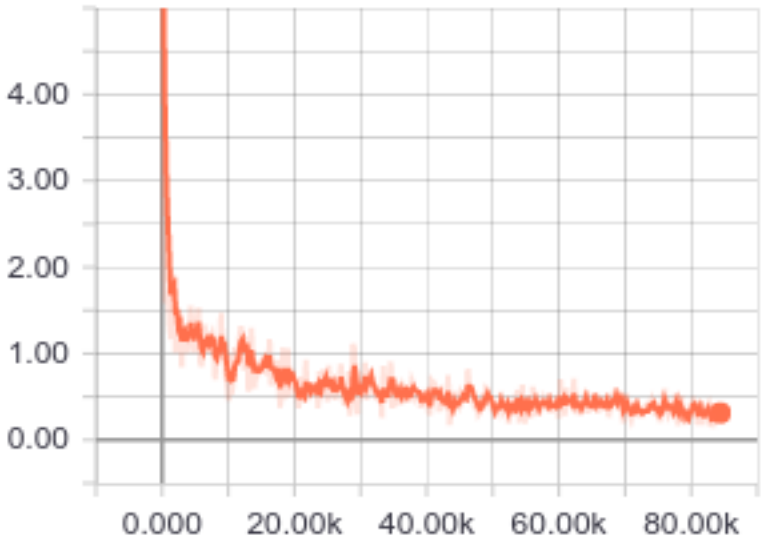

Figure 5.4: Total loss of RFCN Model

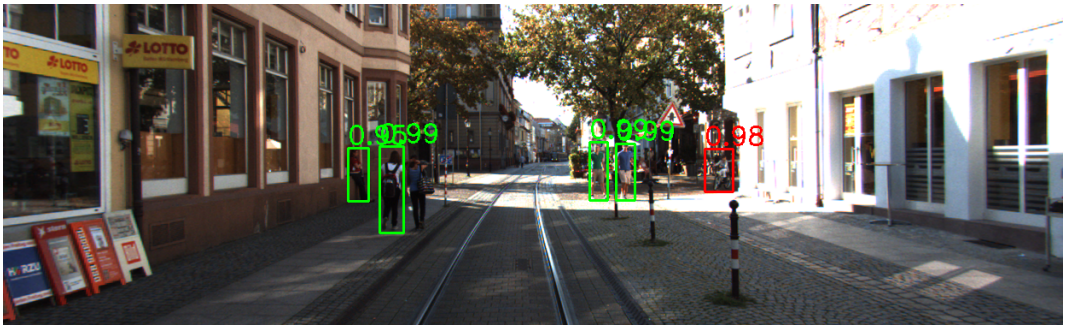

Figure 5.5: Example Output of KITTI Trained RFCN Model

The FRCNN model used was an Inception ResNet v2 with atrous, or dilated, convolution feature extractor combined with an additional detection portion of the network [9]. To fine tune this model, a batch size of 3 was split across three GPUs. The optimization method used was Nesterov Momentum, the initial learning rate was 0.0003 and momentum value of 0.9. Data augmentation was performed by random horizontal flips.

The loss curve shows that the FRCNN model did not having trouble learning

the KITTI dataset, seen in Figure 5.4. The model had enough representational power to detect most objects correctly, with a formal definition of correctness provided in Section 7.2, and with high confidence. FRCNN is not a unified model and therefore does not struggle with distant objects like SSD does, seen in Figure 5.7.

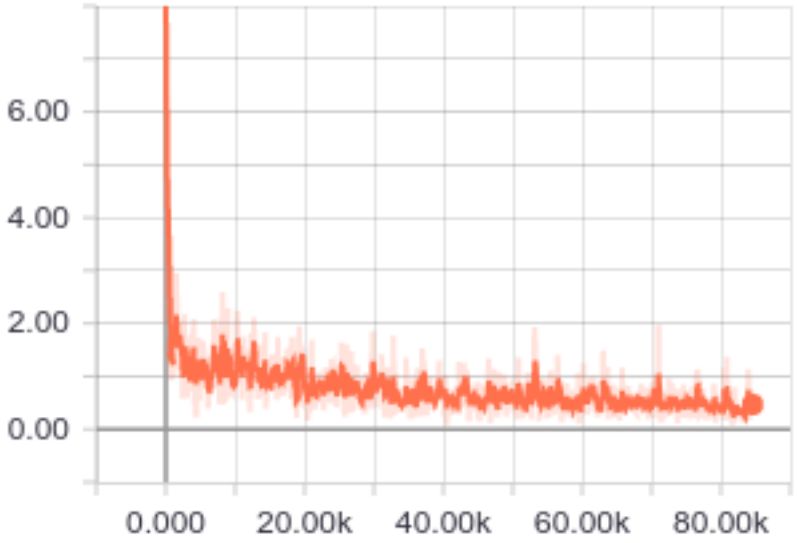

Figure 5.6: Total loss of Faster RCNN Model

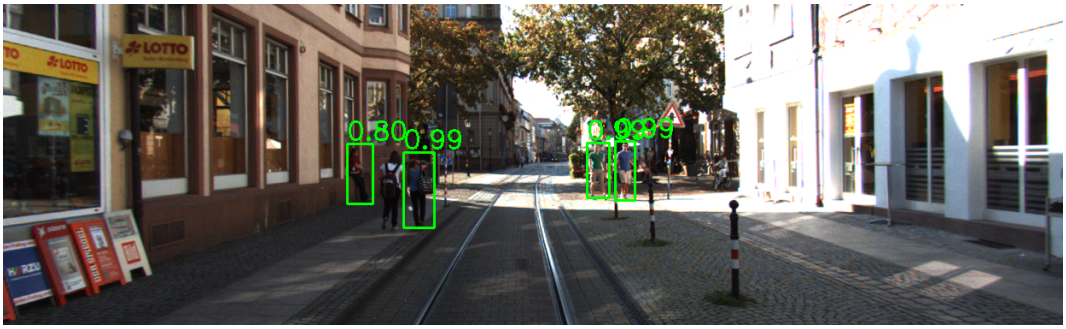

Figure 5.7: Example Output of KITTI Trained FRCNN Model

5.4 Model Optimization

In order to reduce inference time and reduce the memory footprint of the trained models, various model optimization techniques were applied after the training process was completed. These optimization techniques are especially important for model deployment on an embedded device that has significantly less memory and computational resources than a desktop computer.

5.4.1 Freezing

After a Tensorflow graph has been constructed and used to train a model, it is saved to disk to be used only for inference. However, the model parameters are still defined as variables even though their values will not be changed and there are also many operations in the graph that were only required during the training process. Freezing a Tensorflow graph and model first eliminates all operations that are not required for the inference process, which reduces the necessary computation and model size, and then converts all variables into constants which allows for faster computation.

5.4.2 Fusing

After a frozen graph has been created, the data flow can be optimized even more by performing operation fusion, which was also performed in Section 4.7 for batch normalization fusion. Take for example a batch normalization operation, followed by a RELU activation, finished with a convolution. Each operation's kernel must wait for the previous kernel to finish which creates a small bottleneck. Model fusion would combine the batch normalization, RELU, and convolution kernel's

into a single kernel that increases the data flow through the frozen graph, seen in Figure 5.8.

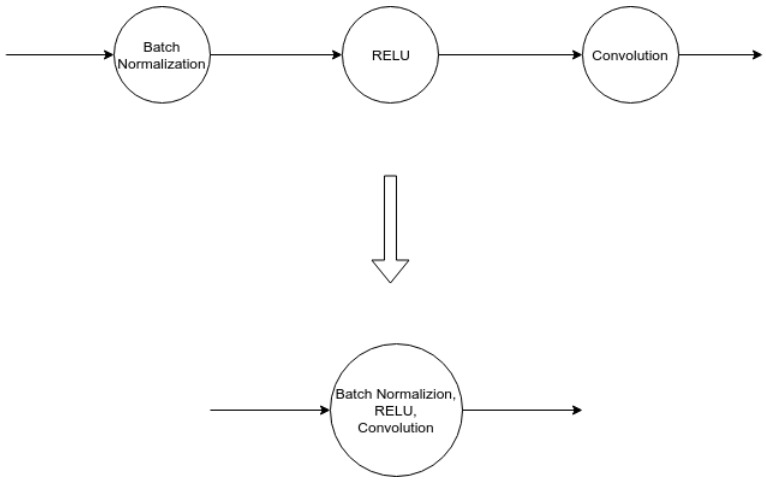

Figure 5.8: Example of Model Fusion

5.4.3 Quantization

The last optimization that can commonly be performed on trained models is quantization. Quantization was not performed on the models used in MFDS because quantized graphs are not supported in the the version of Tensorflow that was used to develop MFDS however, quantization would be an important step for deploying MFDS. During training, model parameters are stored as single precision 32 bit floating point numbers. This level of parameter precision is required while the weights undergo slight updates after each iteration. However, after a model has been trained these weights can be reduced to either half or quarter precision values, 16 or 8 bit floats respectively. This allows for for a smaller memory footprint but also for faster inference speed since memory retrieval takes less time [77]. This large drop in computational precision does not lead to a large degradation in accuracy because the model has been trained to be resilient to noise. Quantization

reduces the size of the the value's significand digits and is therefore, introducing a kind of roundoff error which is just another source on noise meaning the accuracy of the trained model is not significantly effected.

Chapter 6

LiDAR Point Cloud Detection

6.1 Cluster Formation

6.1.1 Masking

Since the point cloud was a full 360 degree view, which extends farther than the image view (Figure 6.1), there was a large percentage of the points that there was no need to reason about and were therefore removed. A $\frac{\pi}{2}$ portion of the cloud was extracted centered at 0 radians and can be seen in Figure 6.2. This range was chosen empirically by matching the outputted point cloud to the field of view of the image.

Figure 6.1: Raw inputted LiDAR Point Cloud

Figure 6.2: Masked LiDAR Point Cloud

6.1.2 Ground Plane Segmentation

The next step in the processing pipeline was to remove the ground plane is order to isolate the clusters worth reasoning about. This was done using a Planar Random

Sample Consensus (RANSAC) model in the PCL library. The model coefficients were chosen to be optimized and the distance threshold that was used was 0.2 centimeters. The RANSAC model returned a set of indices's that were believed to fall in the range of the ground plane. These points were then removed from the point cloud and left all other points within the cloud, seen in Figure 6.3.

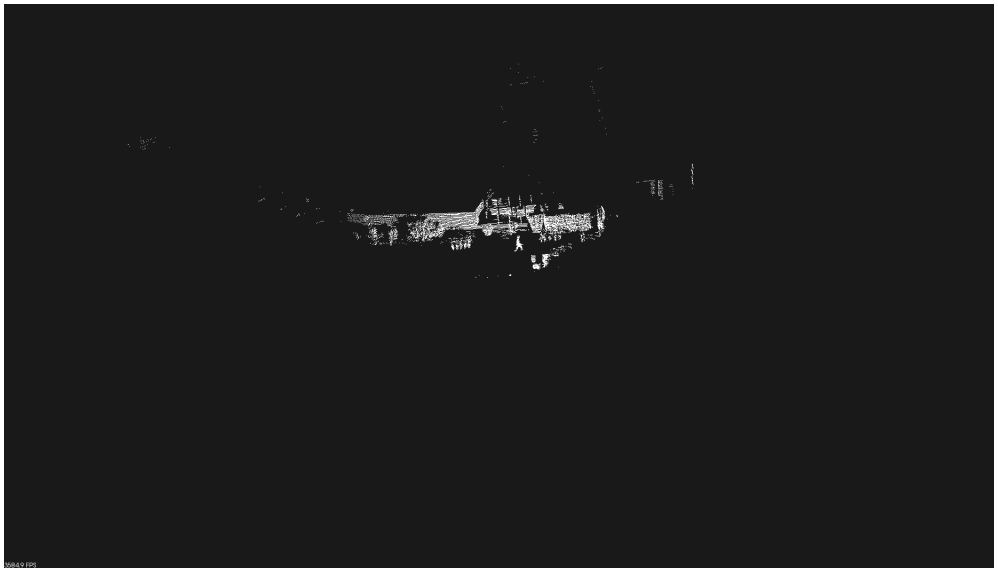

Figure 6.3: LiDAR Point Cloud with after Ground Plane Segmentation

6.1.3 Coordinate Transform

After the ground plane points had been removed from the cloud, the next step was to transform the point cloud from the LiDAR sensor coordinates to camera coordinates. A coordinate transform matrix with translation and rotation components were included in the KITTI dataset for every synchronized pair of camera images and point clouds. An averaging of all 5611 matrices was performed to have a general transformation matrix however there were some extreme outliers that skewed the average very far from what the true transform was, therefore they

were removed and then averaging was performed again this time to obtain the true general transform.

After the average transform was computed, the transform was applied to each point in the cloud after a 1 was appended onto the x,y,z point which is mathematically shown below. The matrix vector multiplication was done using the Eigen library.

$$\mathbf{p}_{\text{cam}} = \mathbf{T}_{\text{velo to cam}}\mathbf{p}_{\text{velo}} \tag{6.1}$$

6.1.4 Euclidean Clustering

Euclidean clustering was then performed on the cloud using a Kd Tree within PCL, seen in Figure 6.4. The cluster tolerance was set to 25cm and the maximum cluster size was set to 25,000 points. At training time the minimum allowable cluster size was 50 points while at inference time the minimum was lowered to 5 in order to help the network pick up on objects that were farther away and were represented with fewer points. Training was performed with a minimum cluster size of 5 however this led to inferior results and therefore was abandoned for the previous training approach of at least 50 points per cluster. A vector of of a vector of points was returned representing the different clusters that had been formed. All points outside of these clusters were no longer reasoned about.

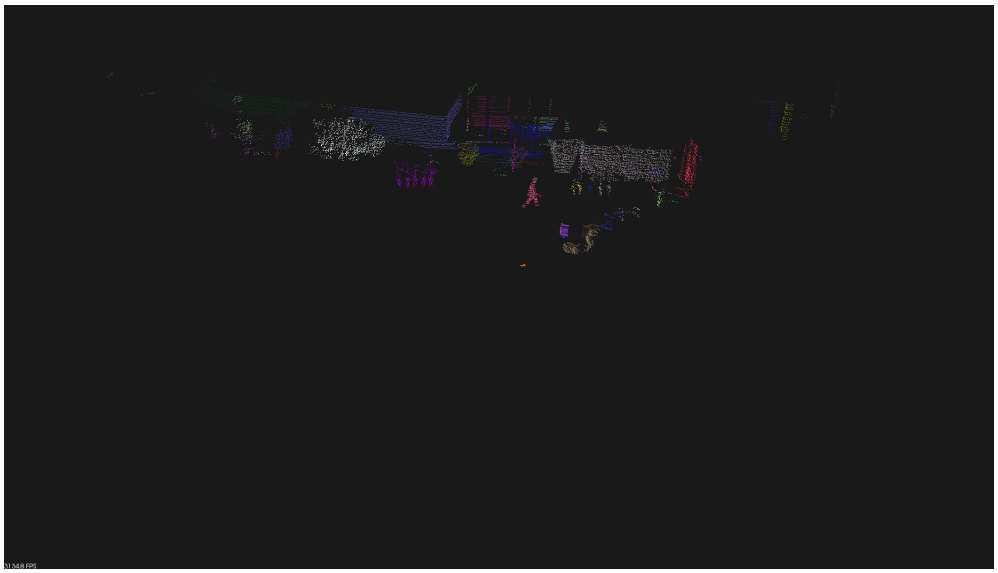

Figure 6.4: Clustered LiDAR Point Cloud

6.2 Cluster Classification

6.2.1 Feature Extraction

After point cloud clusters had been formed, the last step before the MLP was to perform feature extraction. The features that were decided upon was the sample mean and standard deviation of the clusters x, y, and z coordinates, ranges for all 3 dimensions, as well as ratios of x to y, x to z, y to x, y to z, z to x, and z to y. Although these ratios were redundant and included inverses, they proved to be beneficial as the MLP which did not learn as well if they were removed. It is believed that this is because this slight redundancy slightly reduced the complexity of the nonlinear decision boundary that the network needed to learn. These features were chosen because they represent important geometric information about the point cloud cluster while being computationally cheap to compute. The geometric

data of the point cloud is used to augment the color intensity data stored in the camera image.

6.2.2 Dataset Creation

After the LiDAR point cloud pipeline had been created, the dataset needed to be created. To form the dataset, each point cloud in the separated training portion of KITTI was processed with the same pipeline that was used during inference which included masking, ground plane extraction, transformation, and clustering. Each cluster was checked against each labeled cuboid to determine if that cluster represented one of the labeled objects. Since the labels were not perfect, a cluster was considered a labeled object if no more than 5% of the points existed outside the label cuboid. In addition to the KITTI labels of Vehicle, Pedestrian, or Cyclist, clusters with no class label were included in the final dataset in order for the network to learn to be able to reject objects that were not relevant to be detected which meant that the MLP would output a score for four classes instead of three. Therefore, the final classes used were Don't Care, Vehicles, Pedestrians, and Cyclists. In total there were 17,382 clusters, 17% was other, 69% was vehicles, 10% was pedestrians, and 3% was cyclists, seen in Figure 6.5. Therefore, cyclists were the most difficult class to classify since they had the least number of training examples. After the datasets clusters were identified, each cluster had their features extracted and their class, distance to center, length, orientation, and features saved to disk. 75% of the point clouds were used for training and the remaining 25% were used for validation.

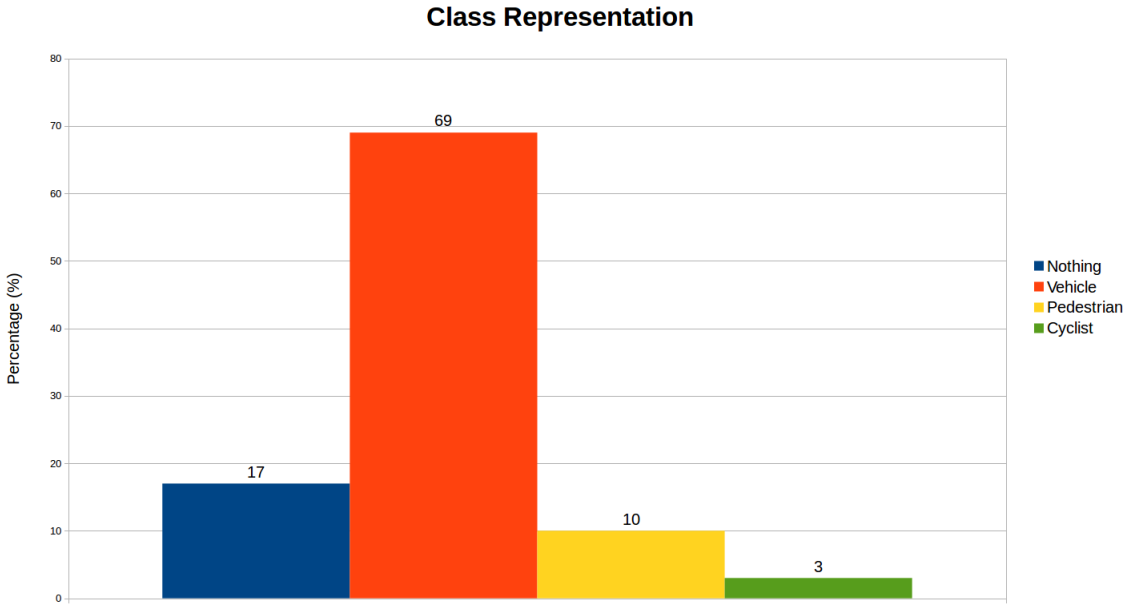

Figure 6.5: Class Representation in Dataset

6.2.3 Mutli Layer Perceptron Architecture

The MLP to classify point cloud clusters was built in Python with the Tensorflow library. The MLP's jobs were to predict the class of the object's cluster, the distance to the center of the object from the LiDAR sensor's origin, the size of the object along the z axis, and the rotation of the object given the cluster's features and the trained model. The class probability distribution is an unknown nonparametric distribution. The MLP is used to form an estimate of the posterior distribution given the trained model and input feature vector. The class probability is mathematical stated in Equation 6.2.

$$\hat{P}(C_i|\mathbf{f}_i, \Theta) \tag{6.2}$$

C_i represents the class probability for the ith feature vector and Θ represents the trained MLP model parameters. The distance to the center of the object, z_i, the size of the object, l_i, and the rotation of the object, α_i, are all estimated values.

The MLP needed one output layer for every value that needed to be regressed, therefore the MLP had four different output layers. The MLP architecture is seen in Figure 6.6 but should be noted that the nodes are not draw to scale according to their size. The input layer for the feature vector had a size of 15 since that was the length of the feature vector, the class output layer had a size of 4 to be able to regress each class probability, and distance, size, and rotation layers each had a size of 1.

The number and shape of the MLP's hidden layers needed to be determined. The number of layers tested varied from 1-7 and the number of neurons in each layer was varied between 10-200. The most hidden layer configuration that converged to the best accuracy had a single hidden layer with 150 neurons. The final MLP architecture can be seen in Figure 6.6.

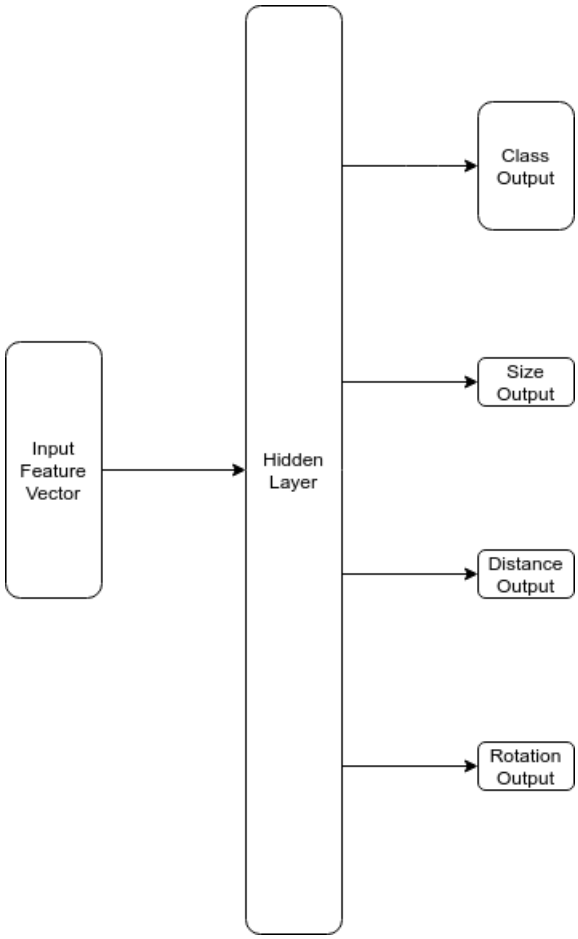

Figure 6.6: Point Cloud Cluster MLP

A truncated normal distribution with a standard deviation of 0.01 was sampled from to initialize all parameters. As is standard for detection networks, the distance, length, and rotation was not predicted outright, an encoding was used instead. The entire dataset was analyzed and the maximum distance was 50m as was the maximum length. The maximum length was this large because of random clusters that were included in the dataset. Therefore, the distance and length did not need to be predicted but instead and encoding, which was easier for the network to learn, was predicted. In addition, the rotation value can only assume

values between $-\pi$ and π so the natural scaling factor used was π. The encodings are seen below:

$$\phi(z_{pred}, l_{pred}, \alpha_{pred}) = \left[\frac{z_{pred}}{50}, \frac{l_{pred}}{50}, \frac{\alpha_{pred}}{\pi}\right] \tag{6.3}$$

The activation function that was used for the hidden layer was a RELU, the class output layer was a softmax, the distance and length output layers were both sigmoids, and the rotation layer was a hyperbolic tangent. The class layer used the softmax to squash all class probabilities between 0 and 1 and to make the sum of the probabilities equal 1. The distance and length layers used the sigmoid because the predictions fell between 0 and 1 since neither of the values could be negative and also the scaled value could never be over 1. The rotation layer used a hyperbolic tangent since the values fell between -1 and 1.

6.2.4 Training

In order to train the MLP to regress all four values simultaneously, a multi-part loss function was created:

$$L = L_{reg} + \frac{1}{N_{total}} \sum_{i=0}^{N_{total}} (\lambda_{class} L_{class}) + \frac{1}{N_{care}} \sum_{j=0}^{N_{care}} ((\lambda_{dist} L_{dist}) + (\lambda_{size} L_{size}) + (\lambda_{rot} L_{rot})) \tag{6.4}$$

L_{class} was Cross Entropy Loss and L_{dist}, L_{size}, L_{rot} were Smooth L1 Loss, L_{reg} was the regularization loss, and $\lambda_{class} = .8$, $\lambda_{dist} = \lambda_{size} = \lambda_{rot} = \frac{1-.8}{3}$. The class loss weight was set so much higher than the other three was because that output

was the most important and needed to be emphasized to learn to the highest degree of accuracy possible. The reasons why the summations were not over the same number of values is that there were no labels for length, size, or rotation of the clusters that were labeled Don't Care and therefore needed to be avoided when computing loss which would impact the gradients and therefore degrade the performance of the MLP.

The batch size was set to 12 on each GPU so there was an effective batch size of 36 and the network was trained for 50,000 iterations. Both Nesterov Momentum, with learning rate of .1 and momentum of .9, and Adam however Adam converged to a higher class accuracy and was therefore used. Each of the individual terms in the Loss Equation and the Total Loss can be seen in Figures 6.7, 6.8, 6.9, 6.10, 6.11, and 6.12.

Due to the unknown shape of the error surface, the MLP accuracy was extremely sensitive to initial conditions used for the learnable parameters. Since the MLP was so sensitive, the network was trained 75 times and the network that scored the highest classification accuracy on the validation set was selected to be the final network.

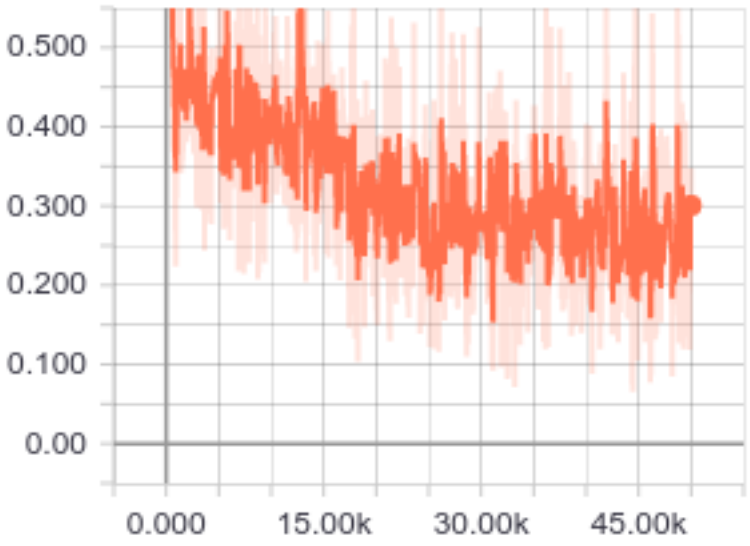

Figure 6.7: MLP Total Loss

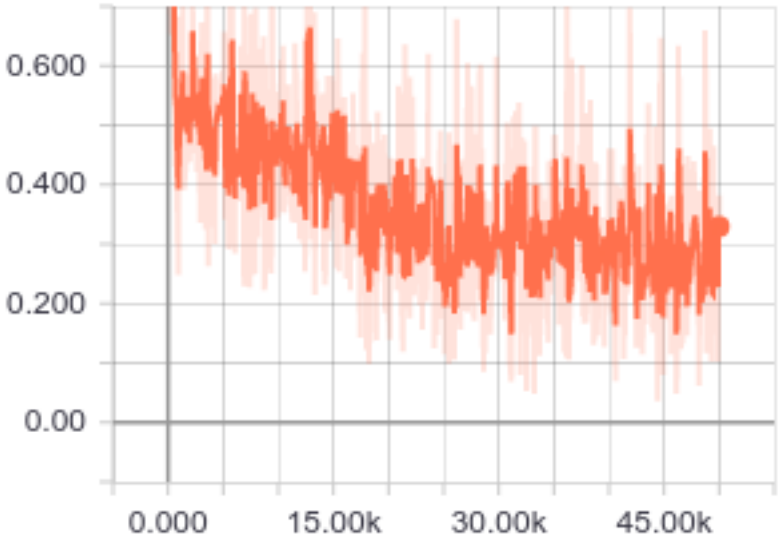

Figure 6.8: MLP Class Loss

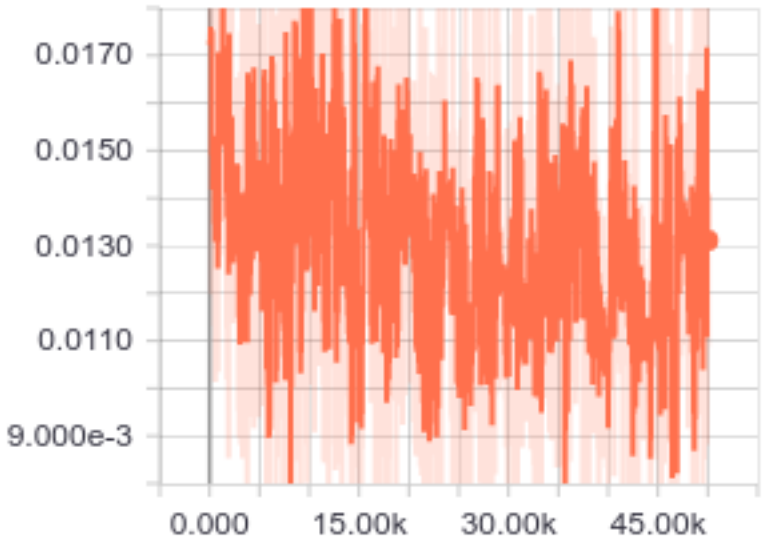

Figure 6.9: MLP Distance Loss

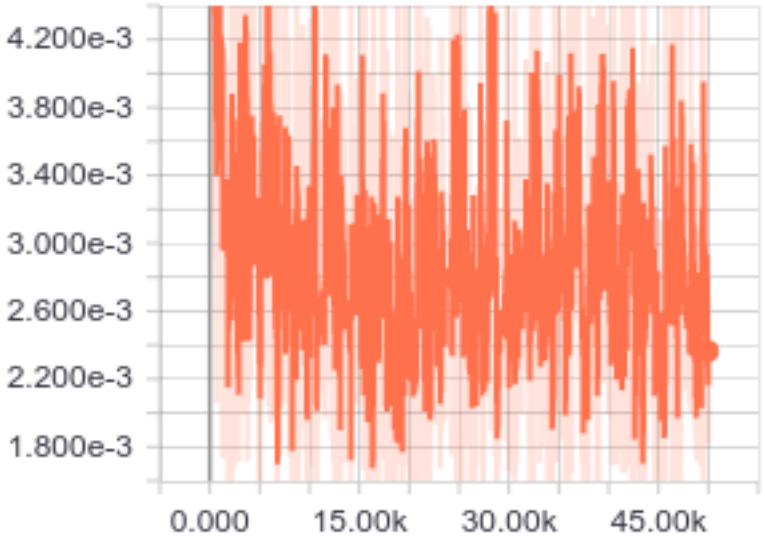

Figure 6.10: MLP Length Loss

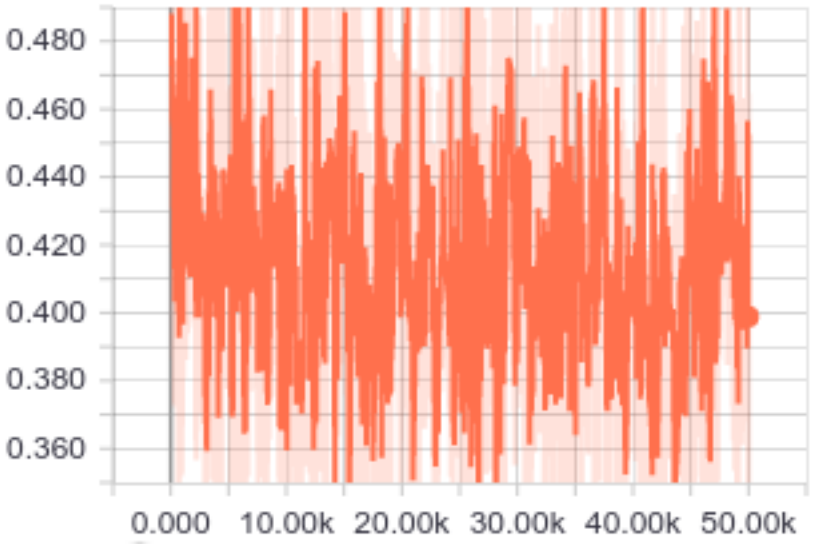

Figure 6.11: MLP Rotation Loss

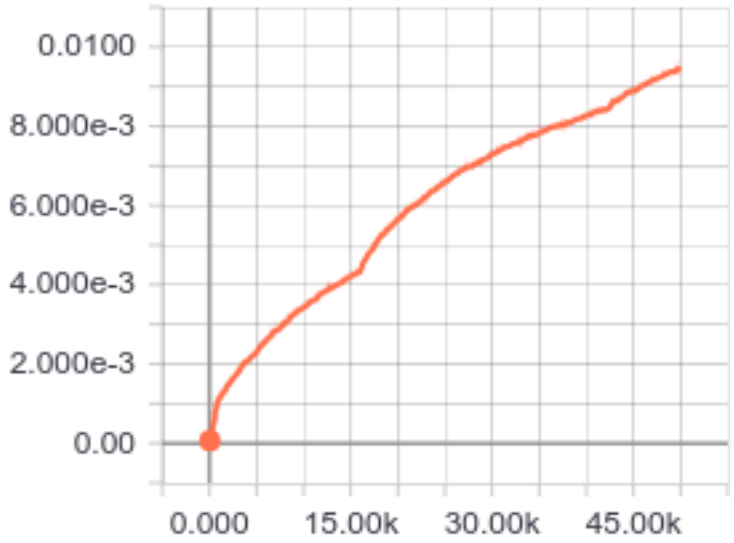

Figure 6.12: MLP Regularization Loss

Chapter 7

Results

7.1 LiDAR Cluster MLP

The MLP was able to learn to classify each cluster with 90% accuracy, on average, without the need for complicated feature extraction. The MLP struggled with classifing the nothing, or don't dare, class as well as cyclists. It is believed that classifying the nothing class was difficult due to the large variance in shapes and sizes, making it difficult for the network to form a relationship between the high variance features and the class label. Cyclists were difficult to classify due to the relatively low number of training examples to learn from.

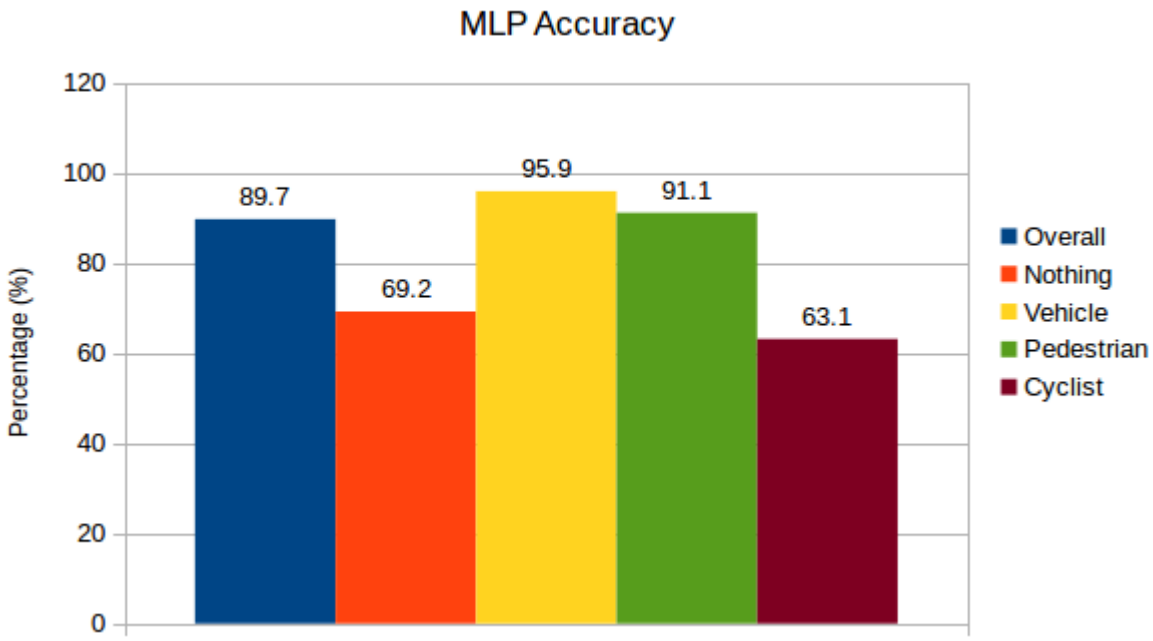

Figure 7.1: MLP Accuracy

All distance, length, and rotation errors presented are after rescaling the MLP's outputs and labels back to dimensional values from their nondimensional outputs. The MLP output of predicted distance to the object had a Mean Square Error (MSE) of 1-2 square meters, which is a much better predictor than the centroid of the cluster (MSE of 39.52 square meters), but not an optimal value. The MLP distance prediction is an order of magnitude more accurate than the naive analysis of the point cloud. The length of the object's prediction had an MSE of approximately a quarter of a meter squared when the average length of vehicles, pedestrians, and cyclists were 18.3 meters. Unlike distance, there is no easy way to predict object length or rotation without making strong assumptions about each class's shape. Therefore, the MLP provides access to reasonable predictions for these values very quickly at the expense of some accuracy.

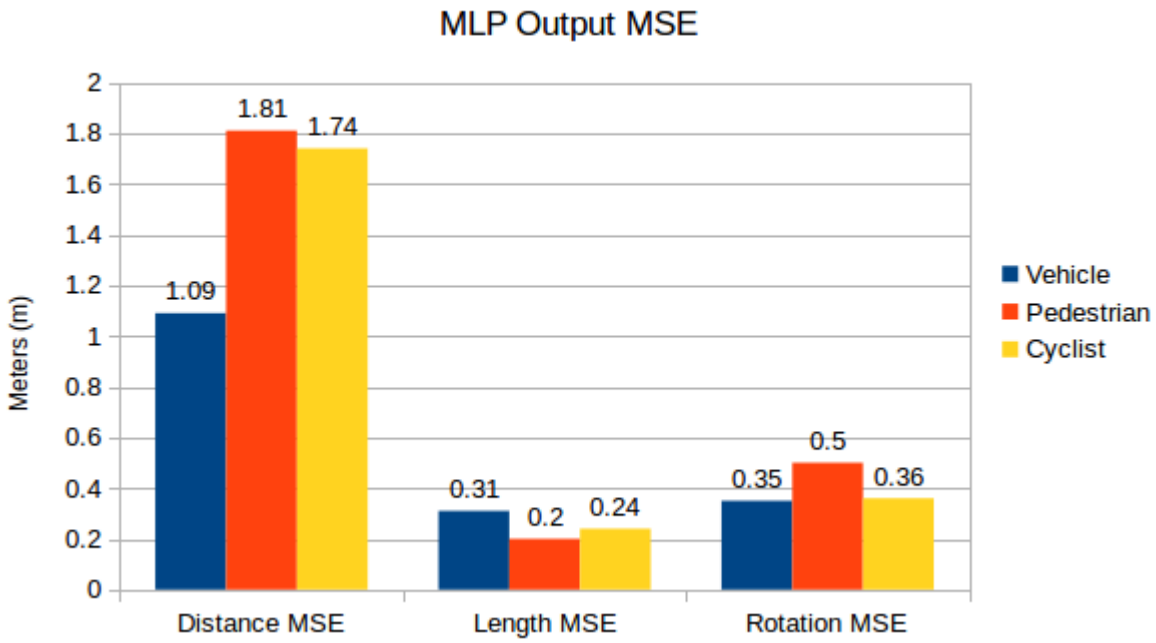

Figure 7.2: MLP Output MSEs

As the MLP was progressing through its 75 separate training initializations, the network appeared to converge to only three different levels of accuracy. It became apparent that these three levels of accuracy were proportional to each classes representation in the dataset. The network would converge to roughly 79%, 87%, or 90%. The networks that converged to 79% had not learned how to predict pedestrians or cyclists and instead only predicted class labels of vehicles and nothing since they made up the majority of the training examples. 87% convergence represented not learning cyclists and 90% represented learning how to predict all classes. This theory was tested by tallying each of the outputs of the test set and it was confirmed that the network never outputted the corresponding class labels.

7.2 MFDS and Image Detection CNN

Testing was performed on the remaining 1870 images and point clouds that were set aside in the test set. The image only detection method was analyzed in order to see how well the primary SSD model, as well as the RFCN and FRCNN reference models, performed on the KITTI dataset, as well as to act as a benchmark to see how much improvement MFDS provided. The KITTI dataset uses the mean Average Precision (mAP) metric for reporting results and is computed by finding the area under the precision recall curve. KITTI defines a true positive as a detection that scores over 70% IOU for cars and 50% for pedestrians and cyclists. However, mAP is a poor indicator of MFDS's detection quality since it is dependent on how many detections an algorithm can output.

Recall is a measure of how completely the detector's output detection set covers all labeled objects, while precision is a measure of how few incorrect detections are in the output set. Recall is largely dependent on how many detections a detector can output, generally in the range of 10 to 300 [31]. This variable number of detections, at varying confidence levels, allows for increased recall. As more detections are outputted, the likelihood of covering all labeled objects increases. MFDS operates in direct opposition to the idea of a variable number of detections with different confidence levels and works to only output as many detections as necessary, each with high confidence. This difference in output ideology means that recall is not a good evaluation metric for MFDS and as a result, neither is mAP. As the few high confidence outputs of MFDS are used to compute the systems recall, the precision recall curve drops to zero when there are no more available detections to cover the labels, seen in Figure 7.3. Therefore a more applicable metric to evaluate the improvement of MFDS over the base SSD image detection

CNN was used.

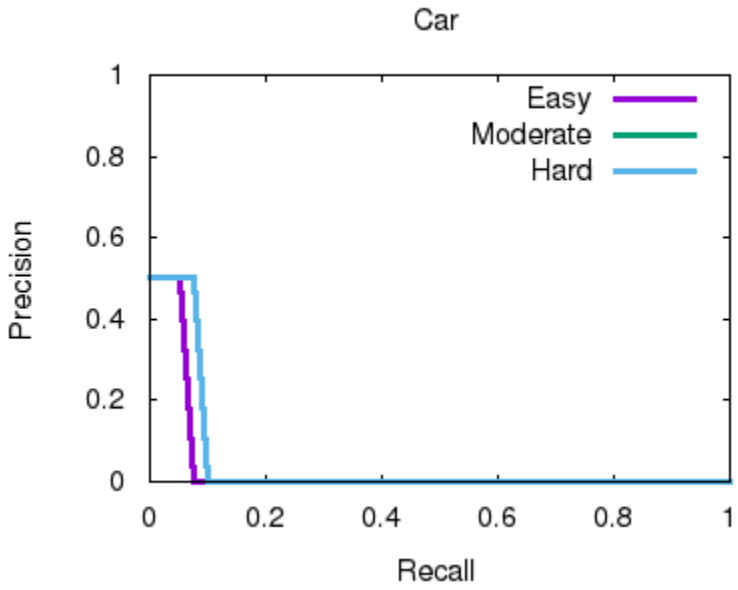

Figure 7.3: MFDS and SSD Precision and Recall Curve for the Car Class

In order to determine the viability for deployment of the image detection CNN model, three main metrics were considered; the memory footprint on the CPU and GPU, the inference time, and the adjusted accuracy. Adjusted accuracy, Equation 7.1, is defined as the percentage of accurate detections with a penalty for incorrect detections, divided by the total numbed of labeled objects. In addition to these three metrics, it was beneficial to view the distribution of different types of detections in order to see the room for improvement that MFDS could add. The different types of detections were based upon varying combinations of confidence and correctness. Correctness was defined as the detection predicting the correct class with appropriate IOU. Confidence was defined as the confidence MFDS had in the detection's class, and a miss was defined as the detection CNN placing a detection around no labeled object. MFDS's main objective was to reduce the

number of unconfident but correct detections and also reduce the number of confident misses. Therefore, MFDS is best suited to be used in conjunction with a CNN detection model that has a high number of unconfidently correct detections, a high number of confident misses, and performs inference at a high rate of speed.

$$\text{Adjusted Accuracy} = \frac{\text{True Positives} - \text{False Positives}}{\text{True Positives} + \text{False Negatives}} \tag{7.1}$$

The command line tool nvidia-smi was used to find each model's GPU memory consumption and the command line top was used to find each model's CPU memory consumption, which included all memory required by the process. The memory consumption is much larger than the size of the trained models because the memory consumption includes the amount of memory for storing temporary values and all additional required libraries for performing inference.

The SSD model was chosen for its small memory footprint, fast inference time, and ideal detection type distribution. Roughly 11% of SSD detection's fell within the categories of unconfidently correct and confidently incorrect, seen in Figure 7.7, as the targeted types of detections for MFDS to eliminate. MFDS takes the small, lightweight SSD model and increases the confidence of it's detections in order to output high confidence, correct detections with a minimal increase in memory demands while still being able to operate at 10 Hz, seen in Table 7.1. Figures 7.4, 7.5, and 7.6 show that MFDS increases the confidence of detections that are able to become confident enough to become final detections. MFDS' output is 96% high confidence, correct detections, which is roughly a 50% improvement over the base SSD model, seen in Figure 7.7. MFDS suffers from a slight, roughly 0.8%,

increase in confidently incorrect detections and misses. With the increase in the number of confident detections, MFDS was able to increase the adjusted accuracy by 3.7%.

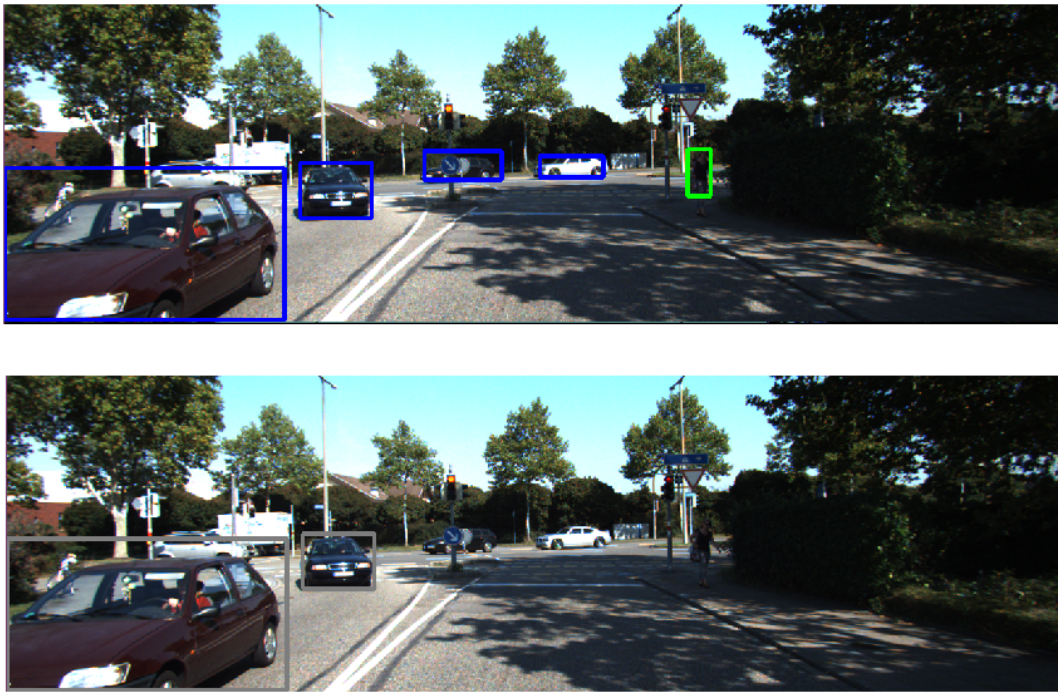

Figure 7.4: SSD to MFDS Comparison; MFDS above, SSD below

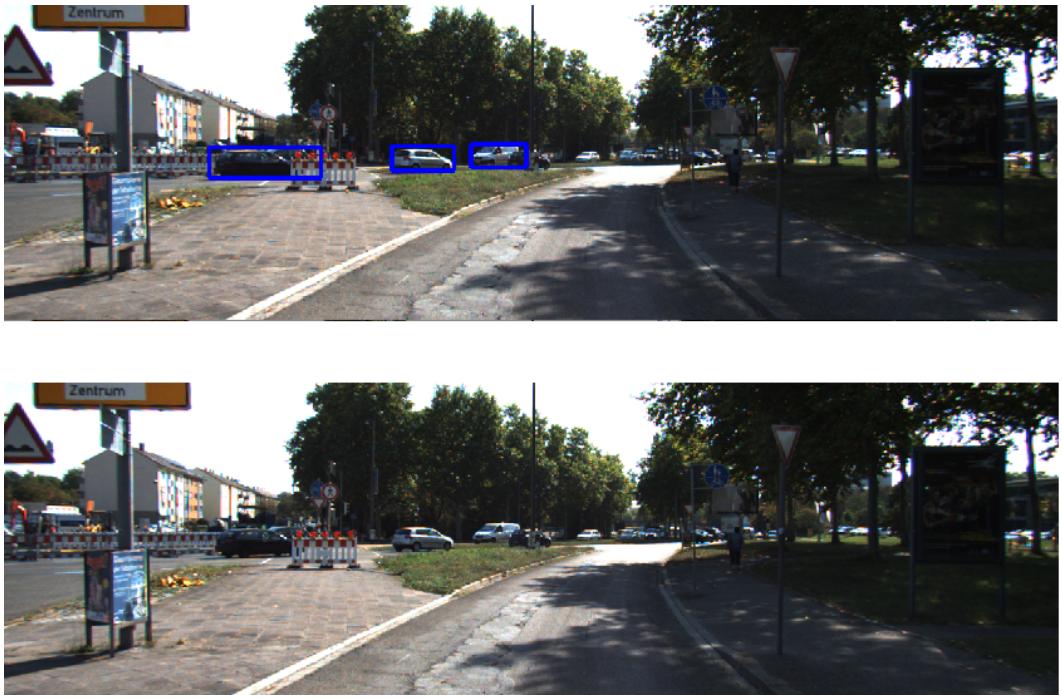

Figure 7.5: SSD to MFDS Comparison; MFDS above, SSD below

Figure 7.6: SSD to MFDS Comparison; MFDS above, SSD below

	SSD	MFDS
CPU Memory (GB)	1.824	2.176
GPU Memory (GB)	0.554	0.703
Inference Time (s)	0.0167	0.1083
Adjusted Accuracy (%)	37.18	40.89

Table 7.1: SSD Results in MFDS

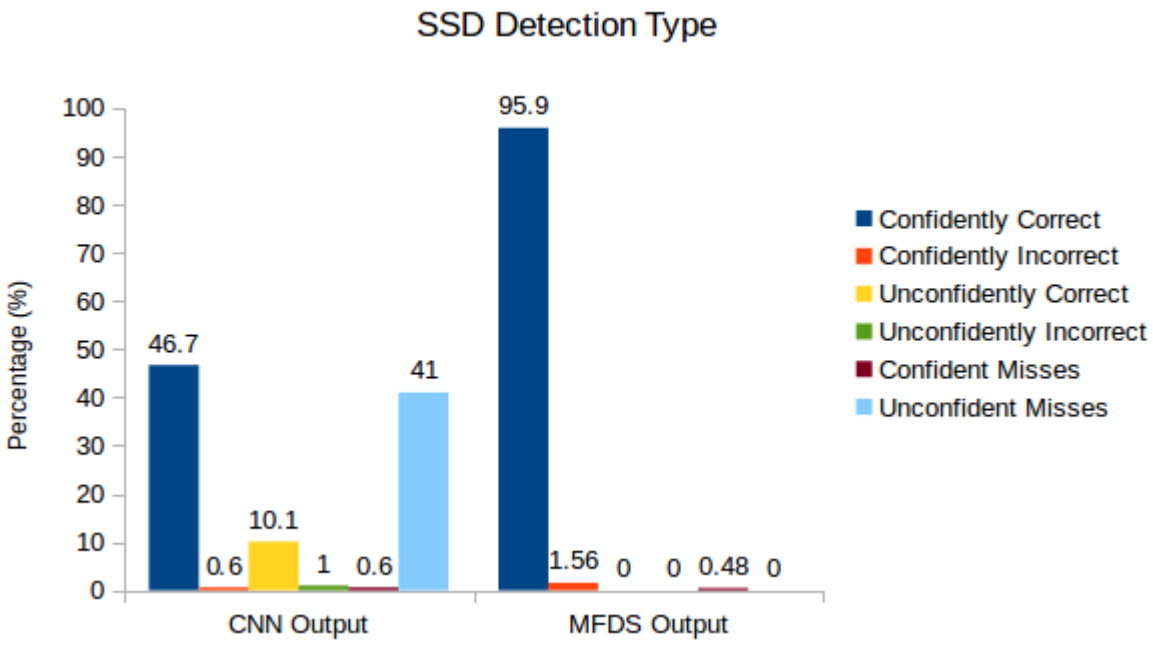

Figure 7.7: SSD Model Detection Type

$$\text{SSD Confusion Matrix} = \begin{bmatrix} 0.9905 & 0.0084 & 0.0011 \\ 0.0236 & 0.9477 & 0.0317 \\ 0.0316 & 0.0772 & 0.8912 \end{bmatrix} \tag{7.2}$$

$$\text{MFDS + SSD Confusion Matrix} = \begin{bmatrix} 0.9908 & 0.0076 & 0.0016 \\ 0.0284 & 0.913 & 0.0586 \\ 0.0518 & 0.0098 & 0.9385 \end{bmatrix} \tag{7.3}$$

In order to evaluate the inference speed of MFDS, each function in MFDS was timed, shown in Table 7.2. Cluster formation took up roughly half of the entire MFDS inference time with the other half evenly spread out amongst the other functions. Although the MFDS processing is roughly as fast as the RFCN and faster than the FRCNN inference time, they are not directly comparable because the CNN detection models operate on the GPU and MFDS operates on the CPU.

Task	Time (s)
CNN	.0167
Masking	0.014
Segmenting Ground Plane	0.009
Coordinate Transform	0.015
Cluster Formation	0.039
Detection/Cluster Association	0.001
Feature Extraction	0.0004
Classification	0.008
Confidence Adjustment	0.0001
Total	0.1083

Table 7.2: Time Analysis of MFDS Inference

One of the main sources of error in MFDS proved to be the detection-cluster association. A known problem is when objects appear very close to one another, their bounding boxes will be nearly on top of one another and will be erased by NMS [5]. MFDS is very susceptible to this problem due to the way it associates detections and clusters together. An example of this can be seen in Figures 7.8 and 7.9 where a cyclist partially occludes a pair of pedestrians. There is only a single bounding box after NMS, due to their high IOU, however, there are two different clusters that are paired with it, visualized as the pink and blue points in Figure 7.9. There will be two final detections with the same bounding box, but with different 3D localized values which, by KITTIs definition, is one true positive and

one false positive since labels can only have one detection after which all detections are considered errors.

Figure 7.8: Occluded Objects from the Image Viewpoint

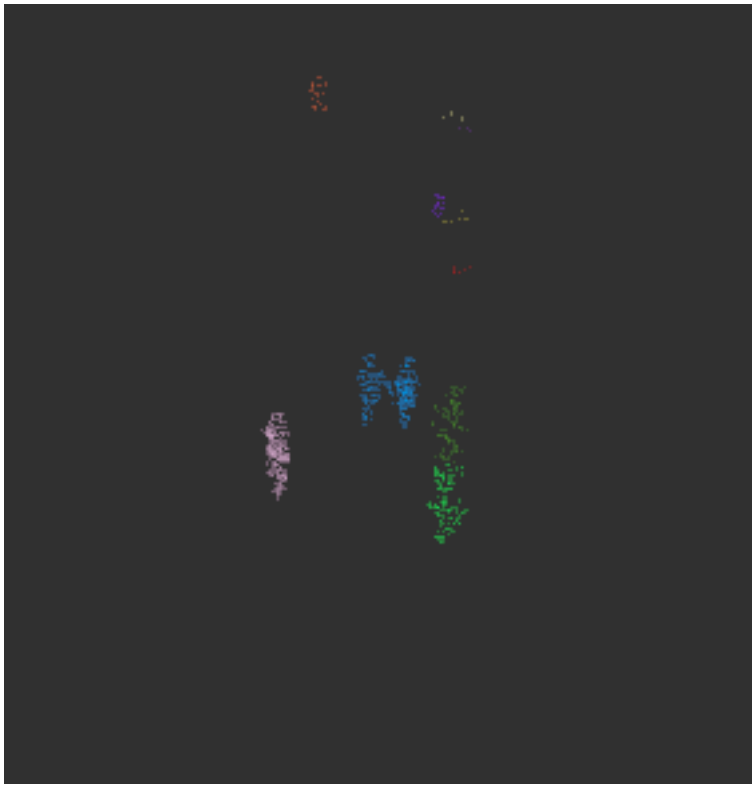

Figure 7.9: Occluded Objects from the Point Cloud Cluster Viewpoint

Another source of error for MFDS was the false positive detection rate due to confident misses. Since many clusters are generated from a point cloud and are

then paired with image detections with MFDS's association method, non object clusters make it to the cluster classification stage. The classifying MLP has a classification accuracy of 90%, which means that for every non object to reach the MLP, 10% will be viewed as confident detections due to false cluster classification and a poor confidence image detection. An example of this can be seen in Figure 7.10 where the treetop canopy's cluster is falsely detected as a car.

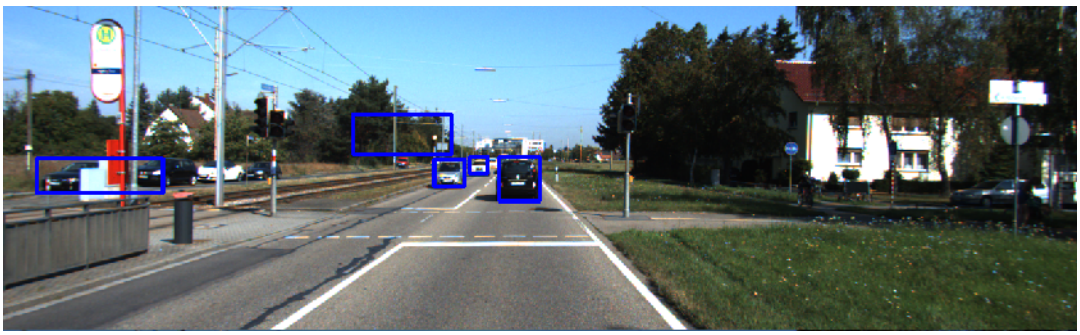

Figure 7.10: False Detection Much Higher than Ground Plane

7.2.1 Reference RFCN and FRCNN Models

MFDS with RFCN and FRCNN both showed roughly a 10% decrease in adjusted accuracy over the standalone CNN detection model, shown in Tables 7.3 and 7.4. This decrease in accuracy is caused by the CNN detectors outputting more detections than the point cloud captures due to LiDAR's comparatively low resolution. Since both an image detection and LiDAR cluster are required to form a pair, many correct image detections are erased since they have no corresponding cluster, creating the drop in adjusted accuracy. Although the adjusted accuracy drops for these networks when MFDS is added, the detection type distribution is reformed to be similar to the final MFDS system with SSD, seen in Figures 7.11 and 7.12.

The time of the base RFCN model is comparable to the final MFDS system

and the base FRCNN model takes roughly five times as long as the final MFDS system. The memory requirements for both the RFCN and FRCNN models are large compared to MFDS. Although this does not pose a problem on the desktop computer used to run the evaluation, on an embedded system this would prove to be detrimental to the inference speed of the network. Take for instance Nvidia's Jetson Tx1 which has 4GB of shared CPU and GPU memory. Tensorflow pre-allocates memory for intermediate tensors within a graph to speed up inference, and without these preallocations, inference takes longer. Both RFCN and FRCNN are larger that the memory capacity of the Jetson Tx1, which would result in an increase in inference time due to the lack of preallocated tensors.

	RFCN	RFCN + MFDS
CPU Memory (GB)	2.592	2.944
GPU Memory (GB)	1.962	2.111
Inference Time (s)	0.073	0.1646
Adjusted Accuracy (%)	0.5811	0.4847

Table 7.3: RFCN Results in MFDS

	FRCNN	FRCNN + MFDS
CPU Memory (GB)	2.816	3.168
GPU Memory (GB)	7.876	8.025
Inference Time (s)	0.54	0.6316
Adjusted Accuracy (%)	0.5784	0.4799

Table 7.4: FRCNN Results in MFDS

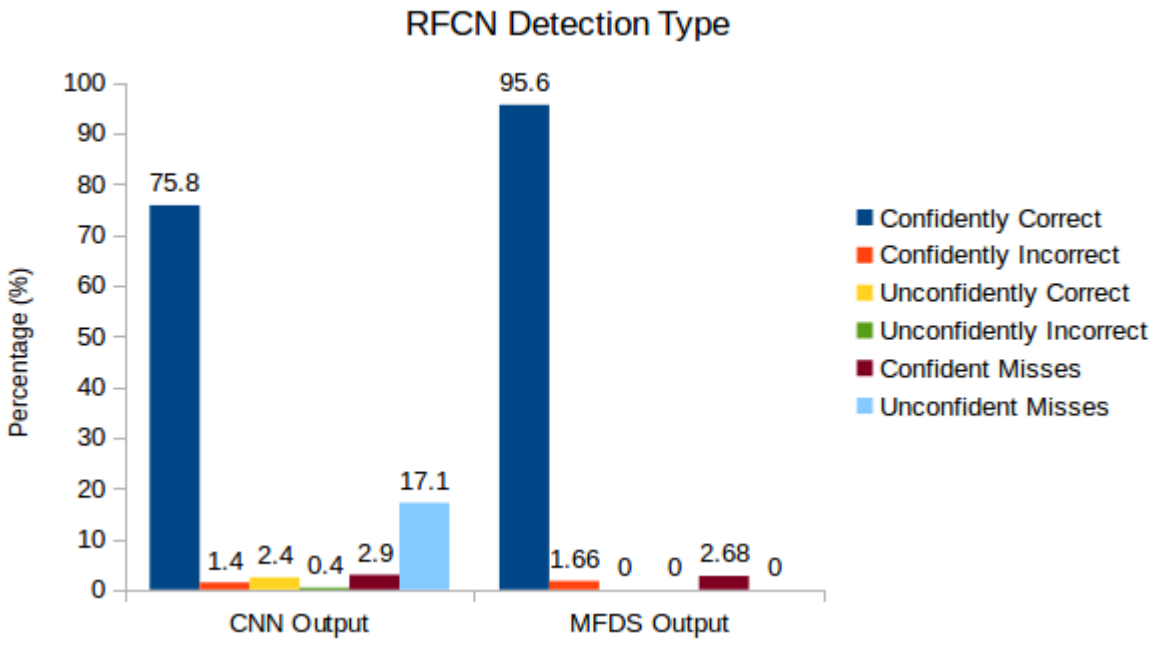

Figure 7.11: RFCN Model Detection Type

$$\text{RFCN Confusion Matrix} = \begin{bmatrix} 0.9904 & 0.0083 & 0.0013 \\ 0.0132 & 0.9606 & 0.0261 \\ 0.0167 & 0.0354 & 0.9479 \end{bmatrix} \tag{7.4}$$

$$\text{MFDS + RFCN Confusion Matrix} = \begin{bmatrix} 0.9916 & 0.0071 & 0.0013 \\ 0.0223 & 0.9372 & 0.0405 \\ 0.0398 & 0.0374 & 0.9228 \end{bmatrix} \tag{7.5}$$

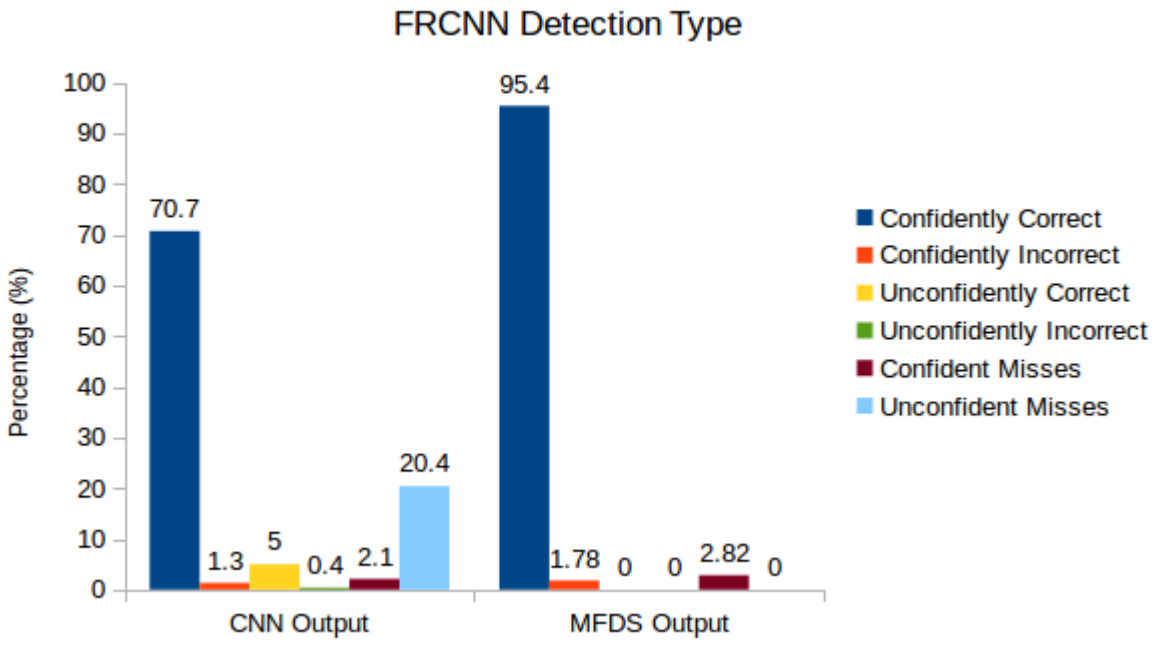

Figure 7.12: FRCNN Model Detection Type

$$\text{FRCNN Confusion Matrix} = \begin{bmatrix} 0.9912 & 0.0077 & 0.0011 \\ 0.0146 & 0.9546 & 0.0308 \\ 0.0203 & 0.0274 & 0.9523 \end{bmatrix} \tag{7.6}$$

$$\text{MFDS + FRCNN Confusion Matrix} = \begin{bmatrix} 0.9919 & 0.0066 & 0.0015 \\ 0.0246 & 0.925 & 0.0504 \\ 0.0372 & 0.0322 & 0.9306 \end{bmatrix} \tag{7.7}$$

7.2.2 LiDAR Processing

Due to MFDS's association problem, no reliable errors can be reported for it's 3D localized outputs. MFDS's detections are comprised of multiple of the same bounding box, each with different 3D localized coordinates provided by cluster analysis. This makes comparing outputted detections to 3D labels impossible without biasing the results. There is no reliable way of associating 3D detections to their true labels without unfairly biasing the association, which would lead to reporting MSE values that are less than their true values. Therefore the most accurate and unbiased MSE results for MFDS's localization that can be reported are the values obtained from the LiDAR cluster test set in Section 7.1.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

An object detection system for autonomous vehicles was discussed in this thesis. The importance of a detection system that can operate in real time and provide 3D localization was discussed in Chapter 1 as well as the history of autonomous vehicles. A review of current object detection systems and the required components to build these systems was discussed in Chapter 2. A gap between fast and inaccurate detectors and slow and accurate detectors was found in the literature. Chapter 3 included the proposal and outline for the Multimodal Fusion Detection System (MFDS). Explanation was provided for how MFDS would bridge the gap and be deployed on an autonomous vehicle as well as a detailed explanation of how MFDS would be developed and built.

A classification network for MFDS's image detector was discussed in Chapter 4 although it was not used due to poor accuracy. Instead, a prebuilt image detection CNN, Single Shot Multibox Detector (SSD), was chosen to be used due to fast

inference speed and low memory usage making it ideal for MFDS. The fine tuning of SSD for autonomous driving was discussed in Chapter 5 along with comparisons to other image detection CNNs. Trained model optimization techniques for the detector CNN to perform inference faster were discussed. The point cloud processing pipeline and how to fuse the LiDAR point cloud data with the image CNN detections in MFDS was discussed in Chapter 6. The creation and training of the MLP used to output point cloud detections and learned 3D localized values was covered.

In Chapter 7 the final results of MFDS were discussed. MFDS was able to increase the adjusted accuracy by 3.7% over SSD while providing detections at 10 Hz. This increase in adjusted accuracy was achieved by changing unconfident into confident detections by performing an analysis on the corresponding point cloud cluster. MFDS performed inference comparably or faster than reference image detectors, took up significantly less memory, and provided 3D localized detections. MFDS was able to take unconfident detection proposals from the image CNN and use LiDAR data to add enough confidence for the detection proposals to be considered true detections. Problems discovered with MFDS and how they negatively impacted accuracy were discussed. A timing and memory analysis of MFDS was completed to provide insight on how MFDS would perform on a Tx1 embedded system for deployment.

MFDS was a step towards a deployable object detection system for autonomous vehicles. It fused information from multiple sensors to produce outputs directly usable by the path planning module of an autonomous vehicle. Although there are limitations to MFDS; the benefits and information that MFDS produces outweigh the problems it faces. Given MFDS's limitations, there are new avenues of research

to improve the detection systems capabilities even farther than this thesis did.

8.2 Future Work

There are multiple avenues for improving the MFDS algorithm. The first, and most important, is creating a better association method for points in the point cloud to image CNN detections. The second is a set of algorithmic improvements that speed up inference time. Lastly there are additional features that can be added to create a more complete perception system instead of only a detection system.

8.2.1 Association Method

Currently the LiDAR point cloud is analyzed by performing a clustering process, followed by matching clusters with image CNN detections. An association technique could be developed that does not rely as heavily on the point cloud in order to output a detection. Since LiDAR has a very low resolution compared to cameras, objects that are far away could rely more heavily on the image based CNN detection instead of the LiDAR point cloud; since the point cloud on current LiDAR sensors is increasingly sparse, the farther away objects are from the sensor.

8.2.2 Algorithm Improvements

There are multiple points at which the current algorithm could be made faster by exploiting parallelism. Besides the image detection CNN and classification done by the MLP, none of MFDS' functions are implemented on the GPU. CUDA kernels could be written to perform masking and transformations as well as compiling

PCL with GPU support in order to use a GPU varient of Euclidean Clustering which consumes the most time. In addition, each cluster's feature vector is run through the MLP in sequence however, it is possible to form a matrix out of every feature vector and run this matrix through the MLP in a single pass, which can be seen below. As there are more numbers of clusters to classify, the benefits of this operational shift will increase.

$$\mathbf{y}_i = \mathbf{W}\mathbf{x}_i \qquad i \in N \tag{8.1}$$

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & ... & \mathbf{x}_N \end{bmatrix} \tag{8.2}$$

$$\mathbf{Y} = \begin{bmatrix} \mathbf{y}_1 & \mathbf{y}_2 & ... & \mathbf{y}_N \end{bmatrix} \tag{8.3}$$

$$\mathbf{Y} = \mathbf{X}\mathbf{W} \tag{8.4}$$

The last major source of improvement via parallelism to the current algorithm would be to perform all point cloud preprocessing and the image detection CNN at the same time.

In MFDS' current state, the image detection CNN and LiDAR point cloud analysis happens in sequential order. However, there is no reason that the point cloud preprocessing in order to form clusters can not be completed in parallel with the image detection CNN. Therefore, as soon as the image CNN detections are computed, they can immediately be associated with the clusters to have features extracted and classified. This however will be a daunting task. The first challenge that this poses is that Tensorflow will need to be compiled with ROS in a CMake file. This will require porting the Bazel build commands used for all other Ten-

sorflow executables and also setting variables to point to the correct places in the Tensorflow build location. Once this has been achieved and it is possible to run the detection network inside C++ and a multi-threaded executable must then be written to perform the two tasks of image detection and point cloud preprocessing simultaneously.

Another potential path for performing detection inference and point cloud preprocessing in parallel would be to use TensorRT instead of Tensorflow for inference. This eliminates the complex CMake port since TensorRT does not require Bazel and is easy to be compiled. However, TensorRT's C++ API only supports NVCaffe models and not Tensorflow models so the Tensorflow model must be converted into a UFF format in Python and then the UFF serialized graph will be imported in C++. After this is completed, the multi-threaded executable would still need to be built.

8.2.3 Additional Features

Since this is the start of a perception system there are many features that should be added. The easiest improvement to be made would be to add reasoning about detections that appear a significant distance above the ground plane. Right now incorrect image and cluster detections that match and slip through both detectors will be verified as a correct detection, an example of which can be seen in Figure 7.10, where a set of tree's canopy is detected as a vehicle.

The next most important feature that should be added is temporal filtering. Currently a fresh analysis is performed at each timestep without utilizing any of the information computed from previous timesteps. This means that there is a large amount of computation that is performed searching the whole scene instead

of starting and focusing the search around where previous detections were found. This means that with the use of previous knowledge, a truly real time system should be able to be developed. However, temporal filtering creates two different problems that need to be solved. It creates a tracking problem and also a data association problem. Not only would previously detected objects need to be tracked to future timesteps, but detections from the current timestep would need to be associated to the tracked detections to make sure that the detections are indeed there and not merely an error at a single timestep and to ensure that new detections enter into the tracker. Most likely the algorithm that is chosen for the tracker should be some variant of a particle filter due to its nice scalability over nonlinear Kalman filters [48].

A major step forward for this perception system would be to form a dynamic world model using the collected point cloud and stereo cloud data. Being able to identify which points in the world were obstacles would allow for more complex reasoning to be performed about paths the vehicle could take. This world model could potentially be either corresponded to or turned into a map for future use that would allow the vehicle to navigate to known locations when directed to, only needing to avoid obstacles along the way.

Part of the world model would also require knowledge about where the lanes of the road are. Therefore a lane detection module would need to be implemented that detected lanes in image space and then was able to map the image space location to world space. These lanes could then be added to the world model as drivable space for paths to be planned. The vehicle would also then be able to localize itself in the model given pose information from an INS system and visual information.

Bibliography

[1] An introduction to tensorflow queuing and threading, 2017.

[2] Convolutional neural networks, 2018.

[3] Nation Highway Traffic Safety Administration. Traffic safety facts 2015, a compilation of motor vehicle crash data from the fatality analysis reporting system and the general estimates system. 2015.

[4] C. Berger and B. Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. *ArXiv e-prints*, September 2014.

[5] Navaneeth Bodla, Bharat Singh, Rama Chellappa, and Larry S. Davis. Improving object detection with one line of code. *CoRR*, abs/1704.04503, 2017.

[6] Z. Cai, Q. Fan, R. S. Feris, and N. Vasconcelos. A Unified Multi-scale Deep Convolutional Neural Network for Fast Object Detection. *ArXiv e-prints*, July 2016.

[7] Gerald H. L. Cheang. Approximation with neural networks activated by ramp sigmoids. *J. Approx. Theory*, 162(8):1450–1465, August 2010.

[8] H. Chen, Q. Dou, L. Yu, and P.-A. Heng. VoxResNet: Deep Voxelwise Residual Networks for Volumetric Brain Segmentation. *ArXiv e-prints*, August 2016.

[9] 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. *CoRR*, abs/1606.00915, 2016.

[10] X. Chen, H. Ma, J. Wan, B. Li, and T. Xia. Multi-View 3D Object Detection Network for Autonomous Driving. *ArXiv e-prints*, November 2016.

[11] Xiaozhi Chen, Kaustav Kundu, Ziyu Zhang, Huimin Ma, Sanja Fidler, and Raquel Urtasun. Monocular 3d object detection for autonomous driving. In *IEEE CVPR*, 2016.

[12] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cuDNN: Efficient Primitives for Deep Learning. *ArXiv e-prints*, October 2014.

[13] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-FCN: object detection via region-based fully convolutional networks. *CoRR*, abs/1605.06409, 2016.

[14] Navneet Dalal. *Finding People in Images and Videos*. Theses, Institut National Polytechnique de Grenoble - INPG, July 2006.

[15] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1 - Volume 01*, CVPR '05, pages 886–893, Washington, DC, USA, 2005. IEEE Computer Society.

[16] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1223–1231, USA, 2012. Curran Associates Inc.

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

[18] M. Engelcke, D. Rao, D. Zeng Wang, C. Hay Tong, and I. Posner. Vote3Deep: Fast Object Detection in 3D Point Clouds Using Efficient Convolutional Neural Networks. *ArXiv e-prints*, September 2016.

[19] Christopher Urmson et al. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics Special Issue on the 2007 DARPA Urban Challenge, Part I*, 25(8):425–466, June 2008.

[20] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[21] A Geiger, P Lenz, C Stiller, and R Urtasun. Vision meets robotics: The kitti dataset. *Int. J. Rob. Res.*, 32(11):1231–1237, September 2013.

[22] R. Girshick. Fast R-CNN. *ArXiv e-prints*, April 2015.

[23] Ross B. Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR*, abs/1311.2524, 2013.

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

[25] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. *ArXiv e-prints*, December 2015.

[26] V. Hegde and R. Zadeh. FusionNet: 3D Object Classification Using Multiple Data Representations. *ArXiv e-prints*, July 2016.

[27] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012.

[28] Jan Hendrik Hosang, Rodrigo Benenson, and Bernt Schiele. Learning non-maximum suppression. *CoRR*, abs/1705.02950, 2017.

[29] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[30] G. Huang, Z. Liu, K. Q. Weinberger, and L. van der Maaten. Densely Connected Convolutional Networks. *ArXiv e-prints*, August 2016.

[31] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama, and K. Murphy. Speed/accuracy trade-offs for modern convolutional object detectors. *ArXiv e-prints*, November 2016.

[32] Peter J. Huber. Robust estimation of a location parameter. *Ann. Math. Statist.*, 35(1):73–101, 03 1964.

[33] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *ArXiv e-prints*, February 2015.

[34] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[35] Ivan Krasin, Tom Duerig, Neil Alldrin, Vittorio Ferrari, Sami Abu-El-Haija, Alina Kuznetsova, Hassan Rom, Jasper Uijlings, Stefan Popov, Andreas Veit, Serge Belongie, Victor Gomes, Abhinav Gupta, Chen Sun, Gal Chechik, David Cai, Zheyun Feng, Dhyanesh Narayanan, and Kevin Murphy. Openimages: A public dataset for large-scale multi-label and multi-class image classification. *Dataset available from https://github.com/openimages*, 2017.

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

[37] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[38] Yann Lecun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L.D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[39] Yann LeCun, Lon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, volume 86, pages 2278–2324, 1998.

[40] B. Li. 3D Fully Convolutional Network for Vehicle Detection in Point Cloud. *ArXiv e-prints*, November 2016.

[41] B. Li, T. Zhang, and T. Xia. Vehicle Detection from 3D Lidar Using Fully Convolutional Network. *ArXiv e-prints*, August 2016.

[42] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.

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

[44] Suresh K. Lodha, Edward J. Kreps, David P. Helmbold, and Darren Fitzpatrick. Aerial lidar data classification using support vector machines (svm). In *Proceedings of the Third International Symposium on 3D Data Processing, Visualization, and Transmission (3DPVT'06)*, 3DPVT '06, pages 567–574, Washington, DC, USA, 2006. IEEE Computer Society.

[45] Liangzhuang Ma, Xin Kan, Qianjiang Xiao, Wenlong Liu, and Peiqin Sun. Yes-net: An effective detector based on global information. *CoRR*, abs/1706.09180, 2017.

[46] Will Maddern, Geoff Pascoe, Chris Linegar, and Paul Newman. 1 Year, 1000km: The Oxford RobotCar Dataset. *The International Journal of Robotics Research (IJRR)*, 36(1):3–15, 2017.

[47] Tomasz Malisiewicz, Abhinav Gupta, and Alexei A. Efros. Ensemble of exemplar-svms for object detection and beyond. In Dimitris N. Metaxas, Long Quan, Alberto Sanfeliu, and Luc J. Van Gool, editors, *ICCV*, pages 89–96. IEEE Computer Society, 2011.

[48] Michael Montemerlo, Sebastian Thrun, Daphne Koller, and Ben Wegbreit. Fastslam 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges. In *In Proc. of the Int. Conf. on Artificial Intelligence (IJCAI*, pages 1151–1156, 2003.

[49] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 807–814, USA, 2010. Omnipress.

[50] Sang-Il Oh and Hang-Bong Kang. Object detection and classification by decision-level fusion for intelligent vehicle systems. *Sensors*, 17(1), 2017.

[51] Adam Paszke, Abhishek Chaurasia, Sangpil Kim, and Eugenio Culurciello. Enet: A deep neural network architecture for real-time semantic segmentation. *CoRR*, abs/1606.02147, 2016.

[52] Geoff Pleiss, Danlu Chen, Gao Huang, Tongcheng Li, Laurens van der Maaten, and Kilian Q. Weinberger. Memory-efficient implementation of densenets. *CoRR*, abs/1707.06990, 2017.

[53] Charles Ruizhongtai Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. *CoRR*, abs/1612.00593, 2016.

[54] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[55] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You Only Look Once: Unified, Real-Time Object Detection. *ArXiv e-prints*, June 2015.

[56] J. Redmon and A. Farhadi. YOLO9000: Better, Faster, Stronger. *ArXiv e-prints*, December 2016.

[57] S. Ren, K. He, R. Girshick, and J. Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *ArXiv e-prints*, June 2015.

[58] Adrian Rosebrock. Intersection over union (iou) for object detection, 2016.

[59] Radu Bogdan Rusu. *Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments*. PhD thesis, Computer Science department, Technische Universitaet Muenchen, Germany, October 2009.

[60] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.

[61] Varuna De Silva, Jamie Roche, and Ahmet M. Kondoz. Fusion of lidar and camera sensor data for environment sensing in driverless vehicles. *CoRR*, abs/1710.06230, 2017.

[62] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[63] Shuran Song and Jianxiong Xiao. *Sliding Shapes for 3D Object Detection in Depth Images*, pages 634–651. Springer International Publishing, Cham, 2014.

[64] Shuran Song and Jianxiong Xiao. Deep Sliding Shapes for amodal 3D object detection in RGB-D images. 2016.

[65] H. Su, S. Maji, E. Kalogerakis, and E. Learned-Miller. Multi-view Convolutional Neural Networks for 3D Shape Recognition. *ArXiv e-prints*, May 2015.

[66] Hang Su, Subhransu Maji, Evangelos Kalogerakis, and Erik G. Learned-Miller. Multi-view convolutional neural networks for 3d shape recognition. *CoRR*, abs/1505.00880, 2015.

[67] Ilya Sutskever, James Martens, George E. Dahl, and Geoffrey E. Hinton. On the importance of initialization and momentum in deep learning. In *ICML (3)*, volume 28 of *JMLR Workshop and Conference Proceedings*, pages 1139–1147. JMLR.org, 2013.

[68] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. *ArXiv e-prints*, February 2016.

[69] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going Deeper with Convolutions. *ArXiv e-prints*, September 2014.

[70] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the Inception Architecture for Computer Vision. *ArXiv e-prints*, December 2015.

[71] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[72] Jun Tan, Jian Li, Xiangjing An, and Hangen He. Robust curb detection with fusion of 3d-lidar and camera data. In *Sensors*, 2014.

[73] Sebastian Thrun, Mike Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, Kenny Lau, Celia Oakley, Mark Palatucci, Vaughan Pratt, Pascal Stang, Sven Strohband, Cedric Dupont, Lars-Erik Jendrossek, Christian Koelen, Charles Markey, Carlo Rummel, Joe van Niekerk, Eric Jensen, Philippe Alessandrini, Gary Bradski, Bob Davies, Scott Ettinger, Adrian Kaehler, Ara Nefian, and Pamela Mahoney. Stanley: The robot that won the darpa grand challenge: Research articles. *J. Robot. Syst.*, 23(9):661–692, September 2006.

[74] Dominic Zeng Wang and Ingmar Posner. Voting for voting in online point cloud object detection. In *Proceedings of Robotics: Science and Systems*, Rome, Italy, July 2015.

[75] J. Wang, Z. Wei, T. Zhang, and W. Zeng. Deeply-Fused Nets. *ArXiv e-prints*, May 2016.

[76] U. Weiss, P. Biber, S. Laible, K. Bohlmann, and A. Zell. Plant species classification using a 3d lidar sensor and machine learning. In *2010 Ninth International Conference on Machine Learning and Applications*, pages 339–345, Dec 2010.

[77] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized convolutional neural networks for mobile devices. *CoRR*, abs/1512.06473, 2015.

[78] Z. Wu, S. Song, A. Khosla, F. Yu, L. Zhang, X. Tang, and J. Xiao. 3D ShapeNets: A Deep Representation for Volumetric Shapes. *ArXiv e-prints*, June 2014.

[79] Yu Xiang, Wongun Choi, Yuanqing Lin, and Silvio Savarese. Data-driven 3d voxel patterns for object category recognition. In *Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition*, 2015.

[80] Danfei Xu, Dragomir Anguelov, and Ashesh Jain. Pointfusion: Deep sensor fusion for 3d bounding box estimation. *CoRR*, abs/1711.10871, 2017.

[81] J. Zhang, I. Mitliagkas, and C. Ré. YellowFin and the Art of Momentum Tuning. *ArXiv e-prints*, June 2017.

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