Philip Huang (1003196917)
Allan Liu (1003479354)
Word Count: 1986

December 2, 2018

Final Report: Hawkeye

Introduction

Self-driving cars are the buzz. Uber, Google, and automobile giants like GM and Tesla all want a part of the action and for good reason. Successful automation of cars can significantly reduce congestion and traffic accidents, potentially altering the transportation industry as much as the introduction of cars did a century ago.

The most challenging aspect of self-driving is teaching the car how to understand its surroundings, known as perception. A successful perception system should be able to localize the car on a map, identify useful roads, understand different signs and track dynamic objects. Without a confident understanding of the surrounding environment, no self-driving car is able to decide where, when and how to drive.

Hence, this project tackles the important problem of **real-time, 3D object detection** as applied to self-driving cars**.** Using a convolutional neural network, we will exploit the Bird's Eye View (BEV) representation of LIDAR-generated point clouds to detect objects (i.e. cars, pedestrians and cyclist) in 3D spaces. The goal is to incorporate HawkEye into the pipeline of University of Toronto's self-driving car Zeus (Figure 1).

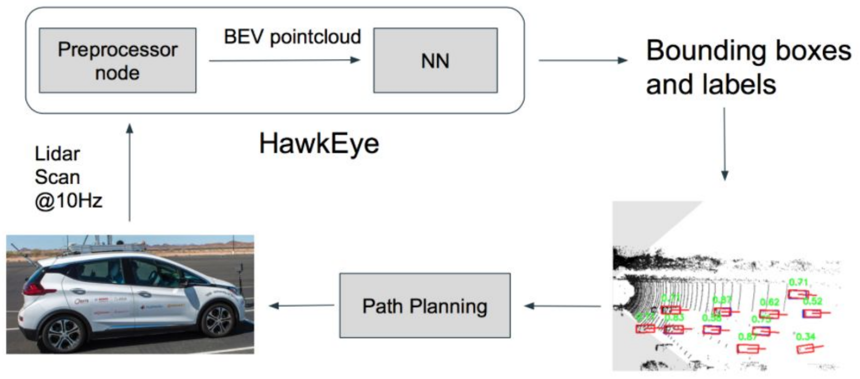

Figure 1: HawkEye's role in a self-driving car's full pipeline

Overall Software Structure

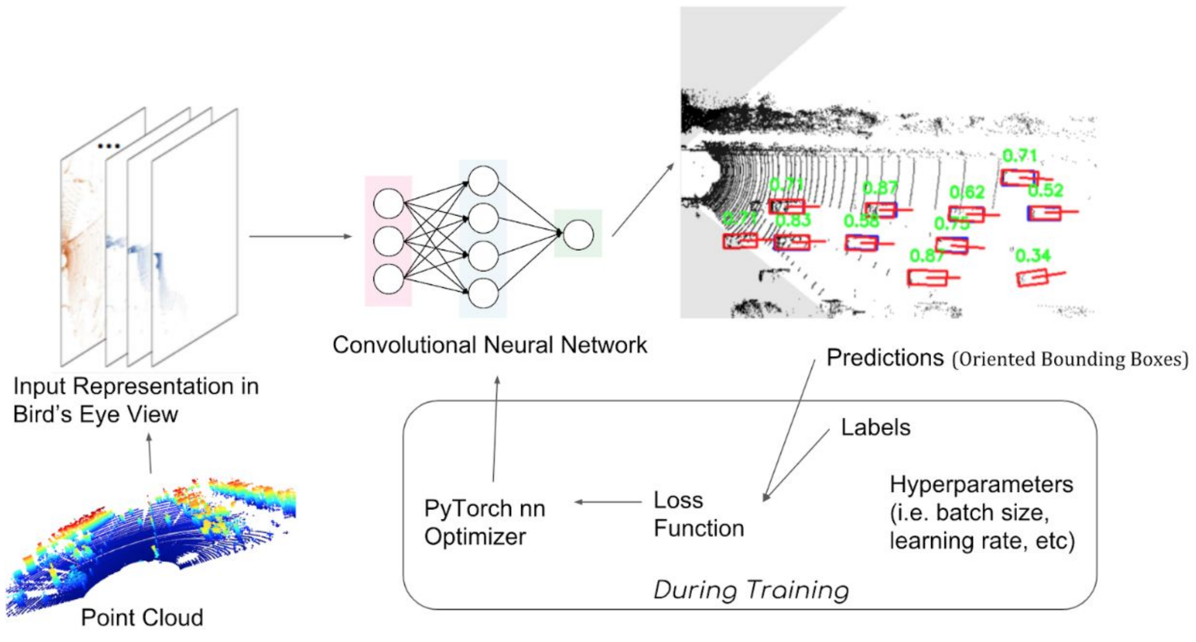
Input Representation in Bird's Eye View
Point Cloud
Convolutional Neural Network
Predictions (Oriented Bounding Boxes)
Labels
Loss Function
PyTorch nn Optimizer
Hyperparameters (i.e. batch size, learning rate, etc)
During Training

Figure 2: Block diagram of HawkEye architecture.

Hawkeye takes in sensor data and outputs bounding boxes of predicted cars . The software for HawkEye contains 4 main modules:

1) Pointcloud preprocessing & dataset wrapper

2) CNN model

3) Post processing & visualization

4) Training and evaluation

Pointcloud Preparation

Sources of Data

1. KITTI Object Detection Dataset for Training and Validation

   The KITTI Dataset [1] is an object detection benchmark containing **7481** labeled frames. Images contain common sceneries like highways, urban centres, and intersections. Each raw point cloud is stored as a list of LIDAR points (see Figure 3), where each point holds (x, y, z) spatial coordinates and a reflectance value between [0, 1]. The LIDAR finishes a scan in 100ms (10Hz) and each scan generates around 120,000 points on average.

2. Custom Dataset for Testing

   Thanks to aUToronto, University of Toronto's self-driving team, we have access to a Chevrolet Bolt (Zeus) mounted with a Velodyne HDL LIDAR sensor. We recorded data containing a sequence raw Velodyne packets and converted them to a sequence of pointclouds in the same format as the KITTI dataset.

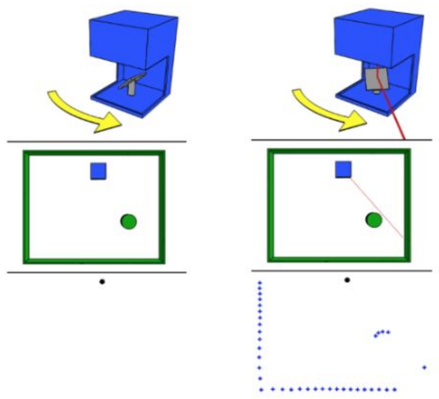

Figure 3: Visualization of a LIDAR scan generating a pointcloud [2]

Labeling Procedure

We used Oxford's VIA tool [3] to label the test data. To label 3D pointclouds, we project the lidar occupancy map onto BEV and compare with 2D camera images as shown in Figure 4. A custom script then processes the generated labels to a format similar to KITTI labels.

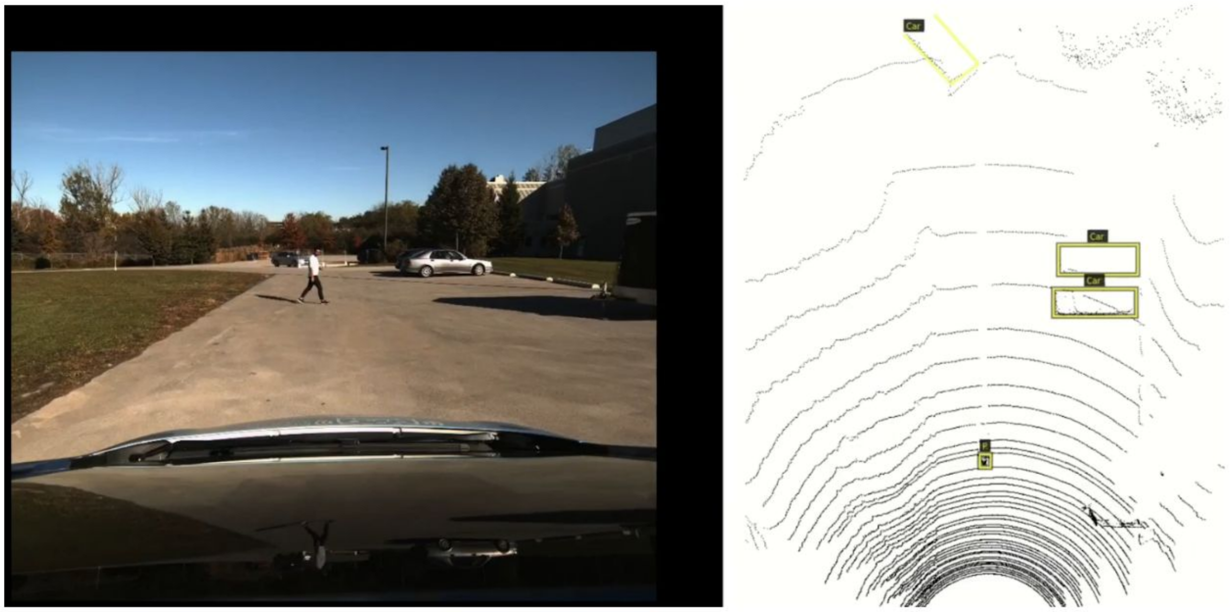

Figure 4. Labeling interface with camera view on the left and BEV of pointcloud with labelled objects on the right.

Lidar Preprocessing

Traditionally, a CNN requires an input image in the form of 3-dimensional tensors.To modify a point cloud into this form, we project all LIDAR points onto slices of the Birds' Eye View (BEV) at different heights. We choose BEV because most road objects lie on the ground plane for the purposes of autonomous driving.

To do this, we first define a scene of L * W * H (**[0m, 70m] * [-40m, 40m] *[-2.5m, 1m]** in our experiments) with respect to the lidar coordinates and eliminate all points that lie outside our region of interest. The 3D volume is then discretized into a cell with resolution of dL * dW * dH (**0.1m * 0.1m * 0.1m** in our experiments). The resultant input tensor has the shape (L/dL, W/dW, H/dH). This input tensor is then treated similar to an RGB image, where 2D convolution is applied on the BEV-plane and height is treated as input channels. Our final BEV representation has a shape of **[800, 700, 36]** with an extra channel to represent the average reflectance in the BEV.

Our training set is composed of 3712 velodyne frames from the KITTI dataset, where each scan contains about 120,000 points. To avoid storing processed inputs on disk memory, we created an efficient C++ preprocessing script that voxelizes the pointcloud at 8ms/frame during runtime.

Neural Network Model

The neural network is inspired by a research paper from Uber ATG [4] and is composed of two core modules: a ResNet backbone that extracts features and a header network that predicts bounding boxes and computes the loss.

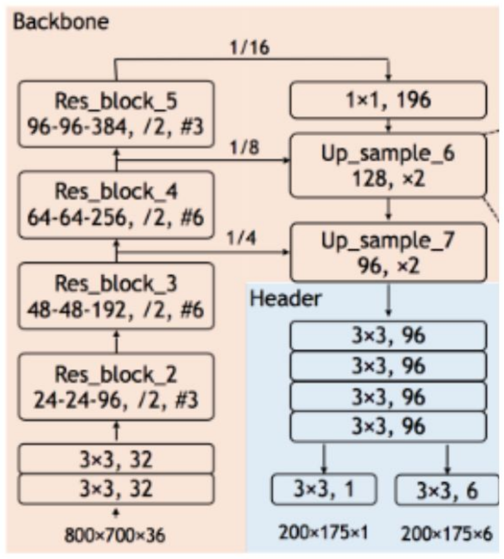

Figure 5: Neural network structure

ResNet Backbone

The ResNet backbone is built with many ResNet blocks. A central argument for this architecture is a deeper CNN should at least perform on par than a shallower one if we simply copy all parameters from the shallow network and apply identity mapping to any deeper layers. However, deeper network are harder to train through back-propagation because of the “vanishing gradient problem”. ResNet solves this issue by adding identity mapping (on the right branch of Figure 6) and forces the left blocks to learn a “residual mapping”. As a whole, using ResNet reduces overfitting and allow a very deep model to train more effectively. The ResNet backbone for HawkEye has 59 convolution layers with over 1.7 million parameters, allowing for very fine adjustments to feature extraction.

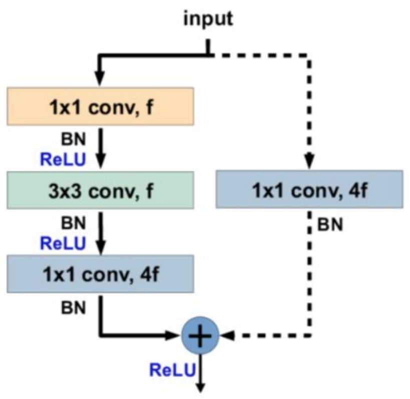

Figure 6: ResNet Block

Header Network and Loss

The header network takes feature maps generated by the backbone and performs 2D convolution to generate prediction boxes for each pixel (Figure 7) and a confidence value. The loss is computed as a combination of classification error and regression error. Binary cross entropy (BCE) is used for the classification error and a smooth L1 distance is used for the regression between labelled bounding box corners and output bounding box corners.

- Header Network
  - Binary Classifier
    - Conv2D -> Sigmoid -> Confidence
  - Regressor
    - Conv2D -> [cosθ, sinθ, dx, dy, log(w), log(l)]
- Loss = *smooth_l1*(regression) + *BCE* (classification)

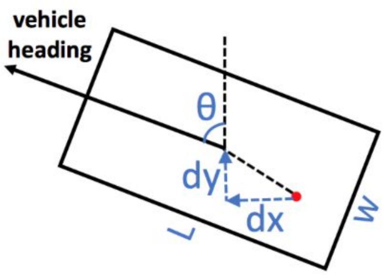

Figure 7: Outputs of header network

Experiments

Training Setup

We trained our model using stochastic gradient descent with momentum of 0.9. We used a NVIDIA Tesla V100 GPU on Google Cloud which can fit a batch size of 6 frames (each frame costs about 2.5 GB of memory). Our initial learning rate is 0.01, and we decay learning rate by a factor of 10 at epoch 20 and 30. Training ends after 34 epochs. We use a L2 weight penalty of 0.0001 to reduce overfitting.

Post Processing

In object detection, non-max suppression (NMS) is the standard technique to remove duplicate detections [5]. During training, we select the top 64 bounding boxes based on their confidence score and perform NMS. However, during inference we kept the all detections with a confidence score greater than 0.5 as positive detections and ran NMS on all of these detections.

Evaluation Protocol

In contrast to binary classification, our label cannot directly tell if an prediction is correct. Thus, we applied the following algorithm to evaluate our detection pipeline :

1) Sort the bounding boxes within each image based on confidence.
2) For each predicted bounding boxes
    a) Calculate the Intersection over Union (IOU) score with each ground truth prediction.
    b) Find the best matching ground truth match (i.e. highest IOU) among those has not been matched with another predicted bounding box.
    c) If the highest IOU is smaller than the decision threshold (0.5 in our experiments), the predicted bounding box is a **false positive.**
    d) If this ground truth box has never been selected, we match this pair and the prediction is a **true positive.**
3) Repeat step 1-2) for every image in the dataset until every prediction is marked as a TP or FP.
4) Calculate Precision, Recall and Average Precision as for the binary classification.

Training Results

Table 1: Results of HawkEye Model on KITTI dataset.

	AP (car)	Precision	Max Recall	F1
Training	71.5%	89.1%	73.2%	80.4%
Validation	60.7%	75.4%	67.2%	71.0%

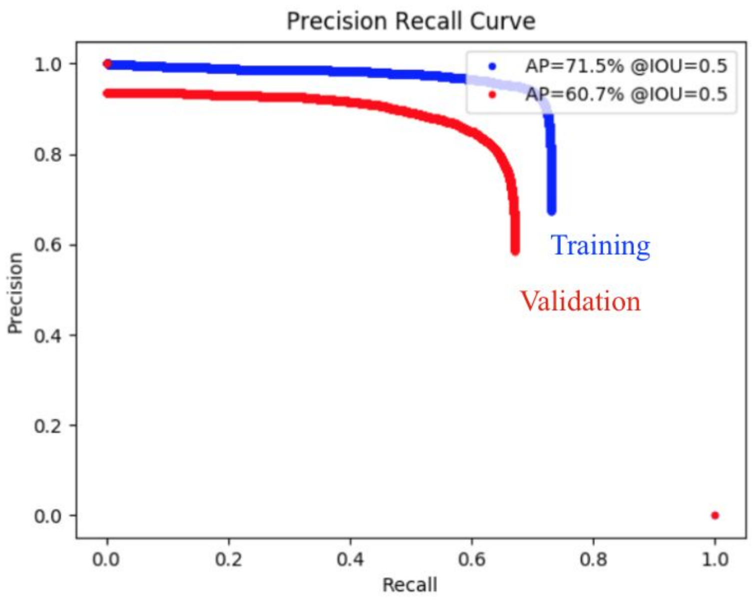

Figure 8. Precision-Recall Curve evaluated on the KITTI training and evaluation dataset

On the KITTI dataset, our model was able to generate very precise detection results with very few false positives on both training and validation datasets (Figure 8). However, our method suffers from a suboptimal recall value, which means the model yields many false negatives (undetected objects). This can be caused by the imbalanced data as there are far more "background pixel" than cars. Furthermore, the model fails to detect objects that are far away due to the sparsity of Lidar pointcloud at long distances.

The model is overfitting on training data and data augmentation could be used to reduce this.

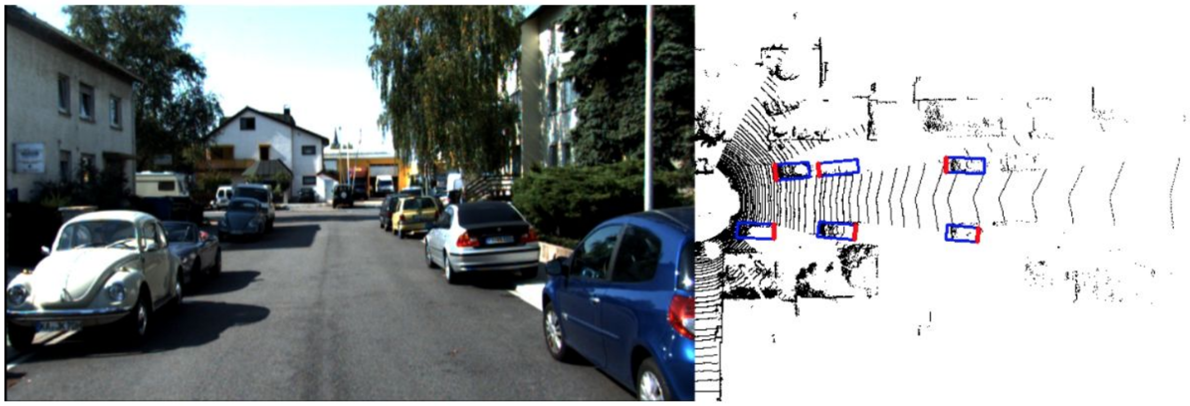

Figure 9. Sample detection on the KITTI dataset

Testing

Testing was performed on data collected from University of Toronto's self-driving car. A qualitative analysis of the results showed that the trained model did not generalize well.

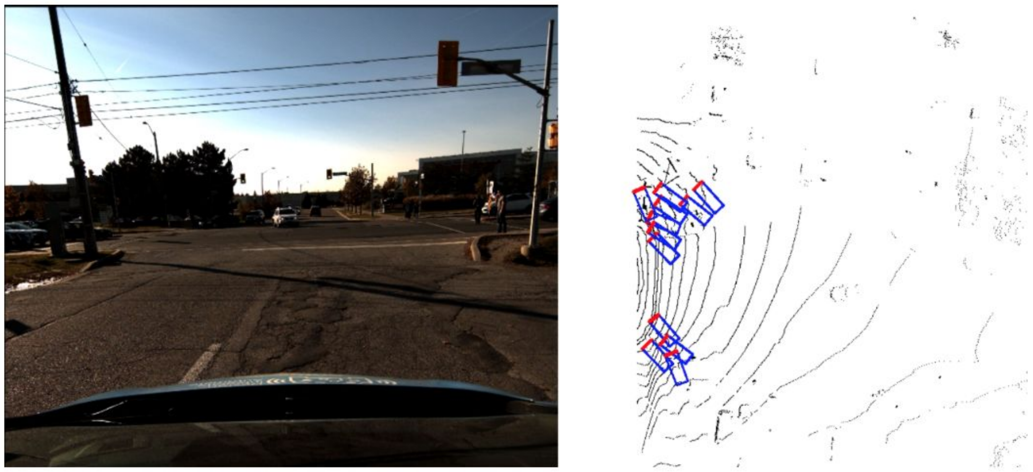

Figure 10. Sample detection on the custom testing set

There are a few reasons for this. There could simply be a bug in the software or hardware, as the complicated pipeline means that there are many potential sources of error. The more salient issue is that the test set was collected on a different car and is very different from the distribution of the KITTI data. This was by design, as ideally the same model would work for different cars without having to retrain the model. The poor test results suggest that it might be necessary to mix in the test data into training and validation datasets.

Ethical Issues

Building a self-driving car poses many ethical challenges. There is the canonical discussion of the “trolley” problem [6], which raises the question of statistics vs. morality. Cars will inevitably get into accidents where a decision has to be made about damage control. Should a car avoid a crowd if it means hitting a bystander? Should a car distinguish between billionaires and homeless persons? Is it always necessary to protect life over objects? What about non-human life forms? Given the state of self-driving car development, it is hard to even consistently distinguish between cars and people and naturally many self-driving car engineers are not as concerned with these ethical problems. However, mindful engineers should design with ethics in mind.

Most pertinent to Hawkeye is the detection of cars. First, we should be aware of where the training data (KITTI) comes from. For example, a model trained on data collected Germany might not generalize well to Chinese roads where vehicle density and/or surrounding buildings may be drastically different. Another consideration is whether Hawkeye is biased against types of cars. A “fair” model should not prioritize expensive cars over family cars or vice versa. Finally, if Hawkeye is generalized to other objects such as bicycles or people, we have to be cognizant of the model’s ability to predict without discrimination.

Key Learnings

Software Engineering

Developing large scale projects requires writing scalable code. Throughout the project, we created a very flexible pipeline allowing us to easily configure hyperparameters and compare results. Some notable implementations in our code that we found very useful are: multi-CPU and GPU support, early-stopping/resume, unit testing and easily adjustable/comparable hyperparameters in config files.

Experiment Organization and Training

Since experiments take a long time to complete, it is essential to have some way of quickly evaluating an experiment in its early stages. However, training and validation was performed remotely on Google Cloud and University of Toronto's dev server and it was difficult to visualize and assess how well an experiment was going. TensorBoard is a great tool to address this problem: by storing log files generated an experiment, TensorBoard allows developers to remotely monitor the progress of an experiment.

Proper Scoping of Project

Collecting usable test data is a multiple-stage process that required too much time given the timeline of this project. It would have been better to scope down this project and test on KITTI data only. Furthermore, integrating pedestrian detection into this model was too complicated and re-scoping the project to only car detection was appropriate.

References

[1] A. Geiger, P. Lenz, C. Stiller and R. Urtasun, "Vision meets robotics: The KITTI dataset", *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1231-1237.

[2] "Lidar", *En.wikipedia.org*, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Lidar#/media/File:LIDAR-scanned-SICK-LMS-animation.gif. [Accessed: 27- Oct- 2018].

[3] A. Dutta, "VGG Image Annotator (VIA)", *Robots.ox.ac.uk*, 2018. [Online]. Available: http://www.robots.ox.ac.uk/~vgg/software/via/. [Accessed: 02- Dec- 2018].

[4] B. Yang, W. Luo, R. Urtasun, “PIXOR: Real-Time 3D Object Detection From Point Clouds”, *In proceedings of the IEEE Conference on Computer Vision and Pattern Recognition 2018,* pp. 7652-7660

[5] A. Ng, "Non-max Suppression - Object detection | Coursera", *Coursera*, 2017. [Online]. Available: https://www.coursera.org/lecture/convolutional-neural-networks/non-max-suppression-dvrjH. [Accessed: 03- Dec- 2018].

[6] S. Bakewell, "Clang Went the Trolley", *Nytimes.com*. [Online]. Available: https://www.nytimes.com/2013/11/24/books/review/would-you-kill-the-fat-man-and-the-trolley-problem.html. [Accessed: 27- Oct- 2018].