Object Detection In An Urban Environment

Ashwin Vaidya

October 2021

1 Project overview

Vision arguably is the single most important part of an autonomous vehicle. If a car has to navigate through an Urban street, it needs to know all the obstacles that can hinder its navigation. While LIDAR is a good source for getting an idea of the environment, it is not as rich as a camera. A failure to detect other vehicles, pedestrians, or cyclists can lead to serious consequences. The goal of this project is to build a vision system that can accurately detect different objects in the scene.

The project begins with an exploration of the dataset used for training the detector. Section 3 goes over this exploration. A few statistics of the data are presented. Based on this analysis, Section 3.2 suggests an approach to split the data into train, validation, and test splits. The success of the vision system relies on the quality of splits. A poor split would mean that the model does not perform well when deployed even though the training performance is good. The analysis also informs the augmentation strategy used for training the model. Details of model training are presented in Section 4. A reference baseline model is trained as control and experiments are performed using different augmentation strategies, optimizers, and neural network backbone. The final model based on EfficientDet gives a mean average precision of 0.2156.

2 Set up

The code in this repository runs on the docker image provided. An additional file called the *devcontainer.json* is introduced which makes it easier to work on the docker image with Visual Studio Code. The *requirements.txt* has been moved from the build folder to the root folder of the directory, and `seaborn` and `pycocotools 2.0.2` have been added to the requirements. If Visual Studio Code has been correctly installed with the `Remote-Containers` extension, then getting started is as easy. Once the folder is opened in Visual Studio Code, it will prompt the user that a *devcontainer.json* is present and the folder can be opened in a container. Proceeding by clicking on `next` will automatically trigger the building of the image and mounting the file system with persistent

storage. It should be noted that `gcloud sdk` and the dataset are not set up automatically. The instructions for setting these up are on the course page.

3 Dataset

3.1 Dataset analysis

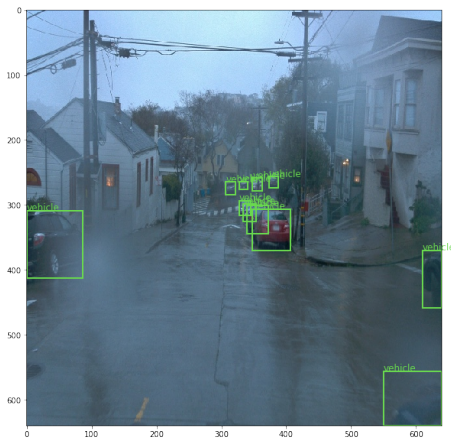
3.1.1 General Overview

Figure 1 shows an instance from the Waymo Dataset with the three classes (pedestrian, vehicle, and cyclist) annotated in different colors. The dataset also contains a fourth class (signs) but is not shown here. This is because the downloaded dataset does not contain the *signs* class.

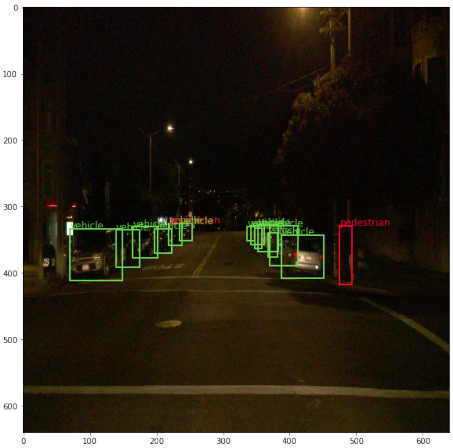
Figure 1: Example of an Image from the Waymo Dataset Showing the Pedestrian, Vehicle and Cyclist Classes

Even a brief exploration of the dataset quickly identifies a few potential issues. The dataset contains occluded and very small objects as can be seen in Figure 2c. Additionally, the environmental conditions vary across samples. Conditions such as fog (Figure 2a) decrease the visibility while rain falling on the aperture of the camera distort the image. These might affect the quality of the

detection later in the pipeline. Figure 2b shows driving night driving scenarios. The limited visibility in the night should (ideally) mean that a cautious approach be taken by the driving policy. Adapting to input variation is necessary as the safety of the pedestrians on the street and the passengers in the autonomous car depends on the accuracy of the recognition system. This accuracy should be consistent across different environmental conditions. An incorrect prediction might lead to a fatal accident.



(a) Foggy/Rainy Condition.



(b) Night Driving Scenario.

(c) Small and Occluded Classes.

Figure 2: Images from Training Data Showing Different Environmental/Weather Conditions.

A closer look into the dataset is needed to ensure better model performance and avoid potential accidents.

3.1.2 Class Distribution

First, it is a good idea to know the distribution of the classes in the datsaet. Figure 3 shows the distribution of bounding box labels for the first 25,000 frames from each file in the test dataset. It can be seen that there is a strong imbalance between the classes. The majority of the classes belong to the vehicle category with the fewest for cyclists category. While the entire dataset takes time to process, we can assume that this subset of the population approximates the actual distribution of classes. This means that a classifier trained on the annotated regions might get biased towards predicting vehicles. Methods such as minority oversampling can be used to ensure the classifier does not get biased. However, these are not explored as the models used in this project are single-shot detectors and sub-sampling detected boxes is challenging.

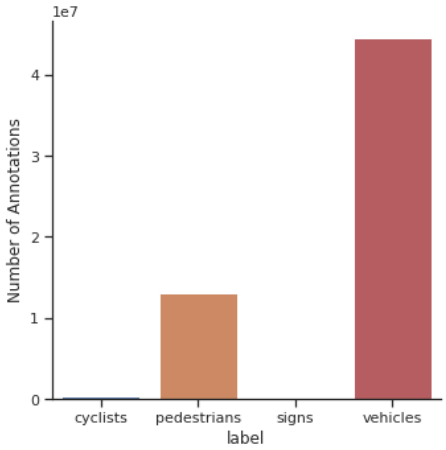


Figure 3: Distribution of Class Annotations for the First 25,000 Frames from Each File in the Test Dataset. There are a Total of 57,845,187 Bounding Boxes in these Images. 322,817 for the Class Cyclists, 13,079,723 for the Class Pedestrians, 0 for Signs and 44,442,647 for the Class Vehicles.

3.1.3 Distribution of Bounding Boxes

Second, as discussed in Section and shown in Figure 2c some of the instances are very small. Thus, it would be interesting to know the trend of the distribution of bounding box sizes in the dataset. Figure 4 shows a histogram for the distribution of the bounding box sizes.

The plot indicates that there are bounding boxes whose area is less than 10 pixels squared. This can be observed in Figure 5. This is concerning as such small values are not optimal for the numerical stability of the regressor.

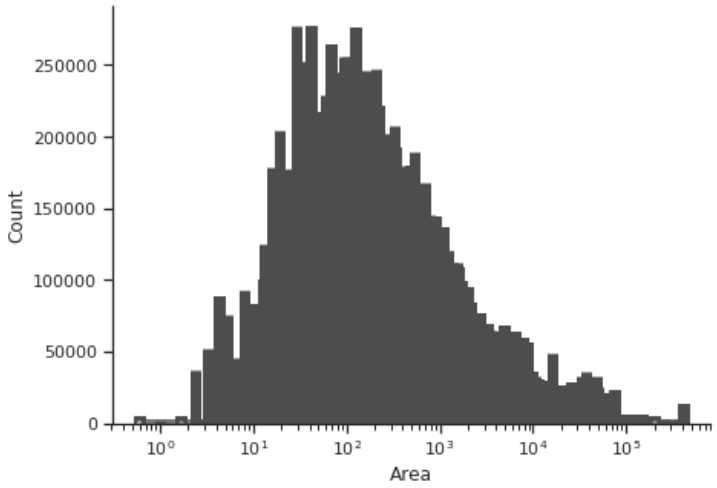


Figure 4: Distribution of the Bounding Box Areas (Pixels Squared in with area in log scale with base 10) for the First 25,000 Frames from Each File in the Test Dataset

These small values are probably introduced due to resizing the original image to 640x640 pixels. This issue however is not addressed when creating the splits and all the boxes are considered for detection regardless of their size.

3.2 Cross validation

By going over a few images in each `tfrecord`, it can be observed that each `.tfrecord` contains labels from a single driving instance. For example, file `segment-12200383401366682847_2552_140_2572_140_with_camera_labels` contains only a night driving instance. If this is used as a test or a validation set then the model will not give good results. The first reason for the expected poor performance is that the distribution of the training set will be different from the test or validation set. And, the second reason will be that if all similar (in terms of environment, weather conditions, or daylight) conditions end up getting distributed in an unequal ratio between the sets, it will lead to class balance. To get accurate results, we have to ensure that the distribution of images in the test dataset is similar to the training data. Hence, a simple approach such as moving a few files to the test and validation folder while retaining the most to the training folder will not be ideal. So to split the data, the *split* function reads a single file and writes each entry to the respective set.

With the lack of the total number of data points, it is difficult to get a correct split but if the assumption is made that the data points are in a multiple of tens then we can reserve 8 for testing, 1 for test, and 1 for validation for each batch of 10 data points. Hence we might achieve the test-val-train split of 80%-10%-10%.

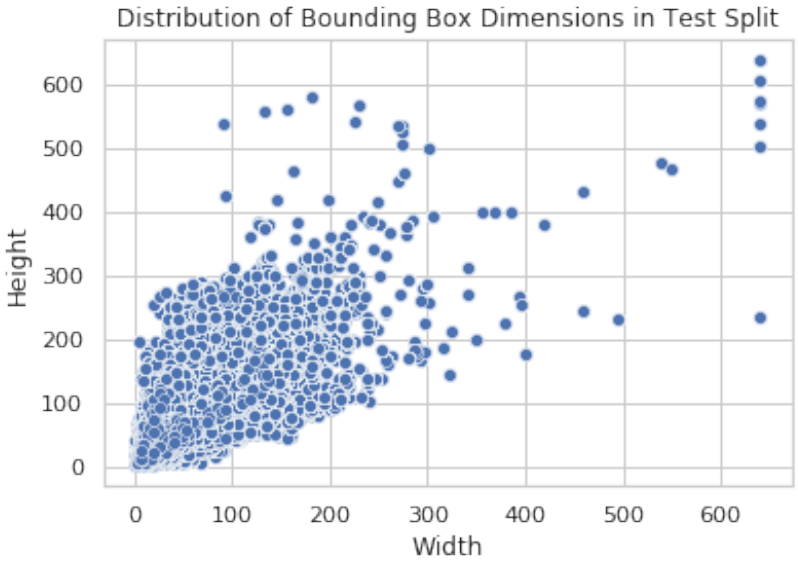


Figure 5: Scatter Plot of Width and Heights Bounding Boxes in the First 25,000 Frames from Each File in the Test Dataset

4 Training

4.1 Reference experiment

The reference experiment uses the default configuration provided with the repository. The model is Single-Shot Detector (SSD) with a ResNet50 architecture. It has two image augmentation options: random horizontal flip and random crop image. It uses the momentum optimizer with cosine learning rate decay. All models tested for this project are trained for a maximum of 25,000 steps with a batch size of 4. The mean average precision of the model is tabulated in Table 1 and the average recall in Table 2. It can be observed that this model performs better than all the models except for the one in Experiment 10. This result is explained in Section 4.2.

Figure 6 shows only the validation loss for the reference model as the training loss was not available on TensorBoard. By looking at the classification and localization loss we can see that running the model for a greater number of steps might lead to further reduction of loss. However, we can see that the regularization loss has saturated. This means that either it has reached a local minima thus needing an increase in learning rate or that the model requires greater data. We can either gather data or augment the current data to introduce variations in the data we have. Section 4.2 goes over a few experiments where improvements to model performance are explored through data augmentation.

	mAP	**mAP L**	**mAP M**	**mAP S**	**mAP@0.5IOU**	**mAP@0.75IOU**
Reference	0.1093	0.4066	0.2947	0.0412	0.2074	0.1021
Exp 0	0.05796	0.2324	0.1613	0.01856	0.1154	0.05393
Exp 1	0.09365	0.3783	0.2514	0.03547	0.1815	0.08763
Exp 2	0.04551	0.1925	0.1315	0.01264	0.09094	0.04224
Exp 3	0.09498	0.3714	0.2512	0.03627	0.181	0.09038
Exp 4	0.05781	0.2303	0.1723	0.0187	0.1132	0.05421
Exp 5	0.04546	0.1886	0.1338	0.01533	0.08687	0.04334
Exp 6	8.73e-3	0.07055	0.01873	1.24e-3	0.02463	4.58e-3
Exp 7	0.09663	0.3683	0.2912	0.03081	0.2132	0.07859
Exp 8	0.01668	0.07752	0.04341	5.06e-3	0.04663	7.57e-3
Exp 9	0.02838	0.08776	0.08949	0.01101	0.06165	0.02363
Exp 10	**0.2156**	**0.6857**	**0.602**	**0.08714**	**0.3992**	**0.2001**

Table 1: Mean Average Precision on the Validation Set for Each Experiment

	AR@1	**AR@10**	**AR@100**	**AR@100 L**	**AR@100 M**	**AR@100 S**
Reference	0.02607	0.1145	0.1827	0.5703	0.4306	0.1046
Exp 0	0.0154	0.0618	0.1074	0.329	0.2518	0.05529
Exp 1	0.02426	0.1018	0.1624	0.5255	0.3884	0.08984
Exp 2	0.01392	0.05297	0.09569	0.2896	0.2363	0.04617
Exp 3	0.02454	0.1017	0.159	0.4843	0.3621	0.09148
Exp 4	0.01628	0.06209	0.1052	0.3165	0.2581	0.05137
Exp 5	0.01533	0.05617	0.1056	0.3259	0.2573	0.05139
Exp 6	5.29e-3	0.01594	0.04918	0.216	0.1277	0.01766
Exp 7	0.02636	0.1079	0.1674	0.5049	0.4573	0.08297
Exp 8	6.89e-3	0.02734	0.0648	0.2531	0.1706	0.02575
Exp 9	0.01252	0.0445	0.09124	0.2717	0.2278	0.04367
Exp 10	**0.0428**	**0.1973**	**0.2807**	**0.7603**	**0.6535**	**0.1713**

Table 2: Average Recall on the Validation Set for Each Experiment

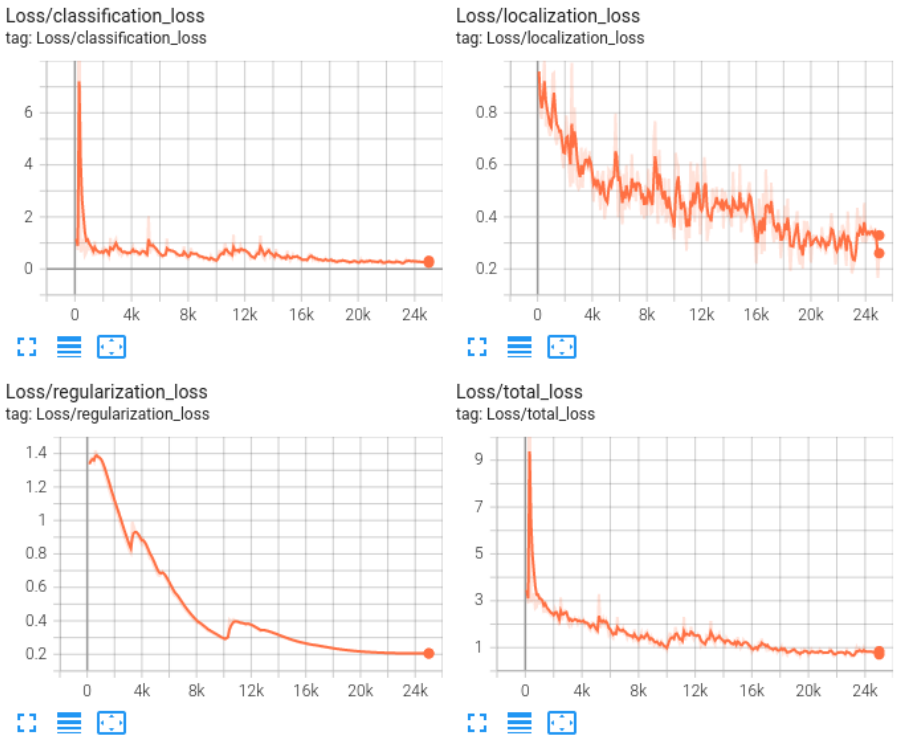


Figure 6: Reference Model Validation Loss

4.2 Improve on the reference

Figure 7a shows a single sample image with the default augmentations present in the reference pipeline configuration file. These are random horizontal flip, and random crop image.

The robustness of the final object detection model depends on the data it is trained on. As mentioned in Section 3.1.3 the dataset consists of a lot of variations in weather conditions and daytime/nighttime scenarios. To make the model robust to different light conditions, in Experiment 0, hue, saturation and brightness transforms are added. To handle different weather conditions, hue and random distort color are introduced. The results of this experiment are visible in Figure 7b. However, from Table 2 and Table 1 we can see that the performance drops across all metrics.

One reason that might explain the drop in performance can be that in some scenes, the brightness might be too high. Hence, in Experiment 1, all the augmentation options are kept the same except for adding max delta 0.1 to random adjust brightness. The sample of this can be seen in Figure 7c. The sample is of an image with a greater effect of color distortion. We can see from the Tables 2 and 1 that this improves the performance across the metric.

(a) Reference (b) Experiment 0 (c) Experiment 1

(d) Experiment 2 (e) Experiment 3 (f) Experiment 4

(g) Experiment 5 (h) Experiment 6 (i) Experiment 7

(j) Experiment 8 (k) Experiment 9 (l) Experiment 10

Figure 7: Examples of a Sample with Augmentations for Different Experiments

Since limiting the brightness helped improve performance, the next question is what is the effect of color distortion on the performance. Figure 7d shows a sample with the same augmentations as in Experiment 1. In Experiment 2, random distort color is removed from the augmentations. However, the model ends up giving a poor performance on the validation set. This means that color distortion is a good contributor to the model performance.

Experiment 3 explores whether limiting the change in hue leads to increasing the performance. A max delta of 0.1 is added to the random adjust hue. We can see from Table 2 and Table 1 that this does indeed improve the performance. A sample with this limited hue distortion is visible in Figure 7e.

The default pipeline config already uses the random crop augmentation. However, the documentation mentions SSD random crop as well. Experiment 4 replaces random crop image with SSD random crop. Figure 7f shows an example of the result produced. However, Table 2 and Table 1 shows that this does not work well with the rest of the configuration as the performance drops. The is probably because a lot of the data is lost if the crop ends up being poor. And, as can be observed in Figure 7f, some of the bounding boxes and the classes get truncated as well.

Experiment 6 explores the Adam optimizer. The results show that it gives poor performance. From Table 2 and Table 1 we can see that the average precision and average recall scores are in the range of 10^{-3}. While fine-tuning the optimizer might help improve the performance. This tuning is not explored in this project. Another difference is that the warm-up steps are set to 100 instead of the default 2000.

To get an idea of why this model does not perform well, we can look at the loss curves shown in Figure 8. Except for the classification loss, all other losses have higher initial values. The is probably because of fewer warm-up steps. We can also see that regularization loss and total loss quickly saturate. One idea to solve this issue can be to change the initial and the final base learning rate. Additionally, Adam parameters such as epsilon need to be fine-tuned. Looking at the localization loss, it seems that the training is unstable. To solve this issue the batch size can be increased and the learning rate can be reduced.

While one value for epsilon is explored in Experiment 8, further fine-tuning is not explored in this project. The augmentations remain the same and the sample is visible in Figure 7h.

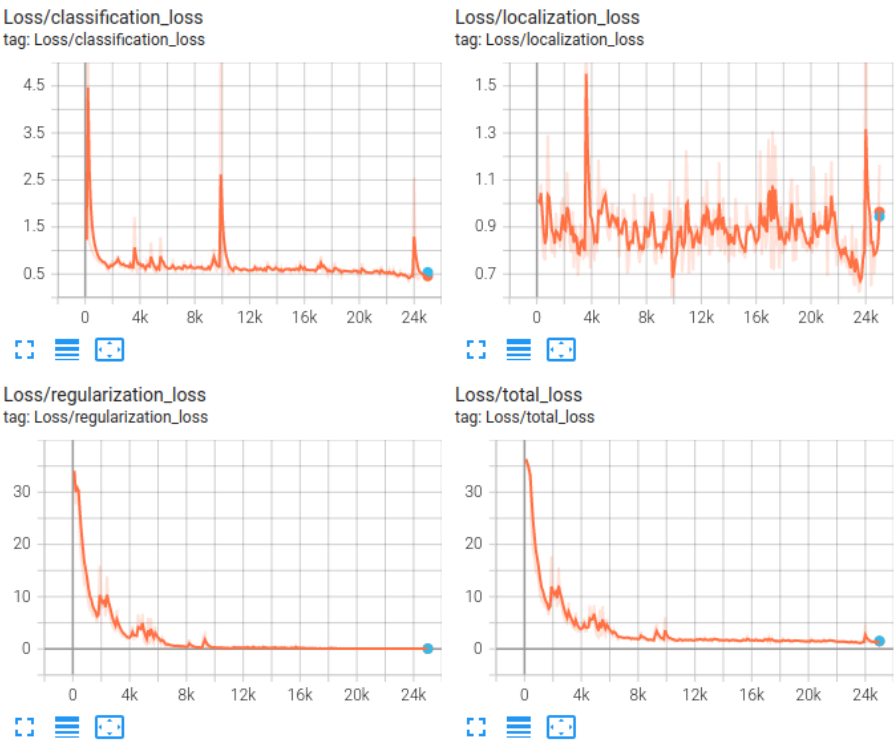


Figure 8: Experiment 6 Model Validation Loss

Due to the lack of performance with Adam, it is replaced in Experiment 7 with RMSprop optimizer with warm-up steps to 2000. While this performance is better than the Adam model, it still gives lower performance than the baseline. Table 2 and Table 1 shows the result. The augmentation options remain the same for this experiment and the resulting image can be seen in Figure 7i.

Since the Adam experiment (Experiment 6) gave a poor performance, this experiment attempts to improve the performance using the epsilon parameter. It is set to $1e-07$ following an example in the TF Object Detection API repository on GitHub. Like Experiment 7, the augmentation options remain the same and the sample can be seen in the Figure 7j. However as can be seen from Table 2 and Table 1 the results for this experiment are still in the rage of 10^{-3}.

Experiment 10 was run twice. First, the backbone is changed from SSD ResNet50 to SSD EffiicientNet b1 BiFPN. During the first run, it gave NAN loss. This is probably due to the small batch size with a larger learning rate. In the second run, the batch size is increased to 16. The learning cosine learning rate parameters are changed to the default values given in the EfficientDet config. Warm-up steps are increased to 2500 while the warm-up learning rate is decreased to 0.001 and the base learning rate is increased to 0.0799. The use

of bfloat 16 is set to true to match the EfficientDet's config however, this is just to suppress the warnings. Since my device does not have a TPU, this does not matter. The random crop image is changed to random scale crop and pad to square. While the official configuration of SSD EfficientDet uses it, there seems to be some over-saturation issue with it and is visible in Figure 7l. This might be related to this issue https://github.com/tensorflow/models/issues/9226 on GitHub. However, this is kept as the model gives a good performance. We can see in Table 2 and Table 1 that this model gives the best performance across metrics and even outperforms the baseline.

While the performance can be improved further, due to the long training time, the model from Experiment 10 is submitted as the final model, and the video of the predictions is available at https://youtu.be/prhFJJD3qQU (EfficientDet). The predictions for the reference can be viewed at https://youtu.be/din5EHcCCDE.

Note: The animation script is also modified as it uses a default of 198 frames from a single file. I have multiple frames across different files in my test set as explained in Section 3.2. So, the animation script goes over all the files in the test set to create the animation. The resulting video is about 12 minutes in duration.