# Object Detection Project Report

Li Yu

December 2021

## 1 Project overview

This project is about object detection in an urban environment, which uses part of the data from Waymo Open Dataset and is fulfilled with Tensorflow Object Detection API. The task of object detection in images is locating the target objects and classify its class. It is very important in autonomous driving because we want to detect any vehicles, pedestrians, and any other obstacles that may do harm to driving. The detection should be accurate and fast such that the autonomous car is able to take the right decision as to either change lane, stop, speed up or down. This project works on a real-world dataset and aims to build an object detection model that can be used in autonomous driving.

## 2 Set up

I completed my project on my local computer, which contains a GeForce 1660 GPU. I didn't use a docker to run the experiments but instead created a conda environment and manually installed all related packages, including tensorflow, object detection api, ray, matplotlib, etc.

I tried to download the tfrecord files myself but found it take too long to finish, so instead I chose to copy the files in the workspace to my local computer. It has 97 processed tfrecords for training and validation, and 3 tfrecords for testing. Since the tfrecords are already processed, I didn't run the 'download_process' python file.

## 3 Dataset

### 3.1 Dataset analysis

As mentioned above, I copied the processed tfrecords from the workspace. My dataset analysis was based on the 97 files for training and validation. The figure 1 below shows a typical frame from the dataset, with bounding boxes marked in different colors.

As I looked over some images with bounding boxes, I found that different image contains different class and number of objects. The location of vehicles, pedestrians, cyclists also vary from image to image. I decided to get the statistics of average number of objects per class in each image and also the object size and location distributions. To get the statistics, I sampled 10 frames each from 97 tfrecords. On average, there are 18 vehicles, 5.3 pedestrians, and 0.1 cyclists in an image. The distribution of bbox size and location are plotted in figure 2.

### 3.2 Cross validation

Since the tfrecords were copied from the workspace, there are 97 files for training and validation, and 3 files for testing. I split 97 files in a 80% vs. 20% ratio into a training set and a validation set. The


Figure 1: An image with bounding box.

reason behind 20% validation split is I don't have sufficient training data. Validation dataset has to be large enough to cover the cases in training.

# 4 Training

## 4.1 Reference experiment

I followed the provided instructions in the tutorial to run training and validation. There are 77 files for training and 20 files for validation. All the hyperparameters such as number of steps, learning rate, data augmentation are set at default values.

The training loss curves and learning rate curve are displayed in figure 3 and 4. From the 'pipeline_new.config' file, I know the number of steps is 25000 and there is a warm-up period of 2000 steps. We can expect the learning rate at first 2000 steps to be big and decrease gradually after, as visualized in figure 4. The decaying curve is a cosine function as defined as the optimizer. From figure 3, I find the final total loss is around 8. It is very high versus the optimum value of 1. It is easy to spot that regularization loss contributes the largest portion to the final loss and the shape of its loss curve dominates the final loss curve.

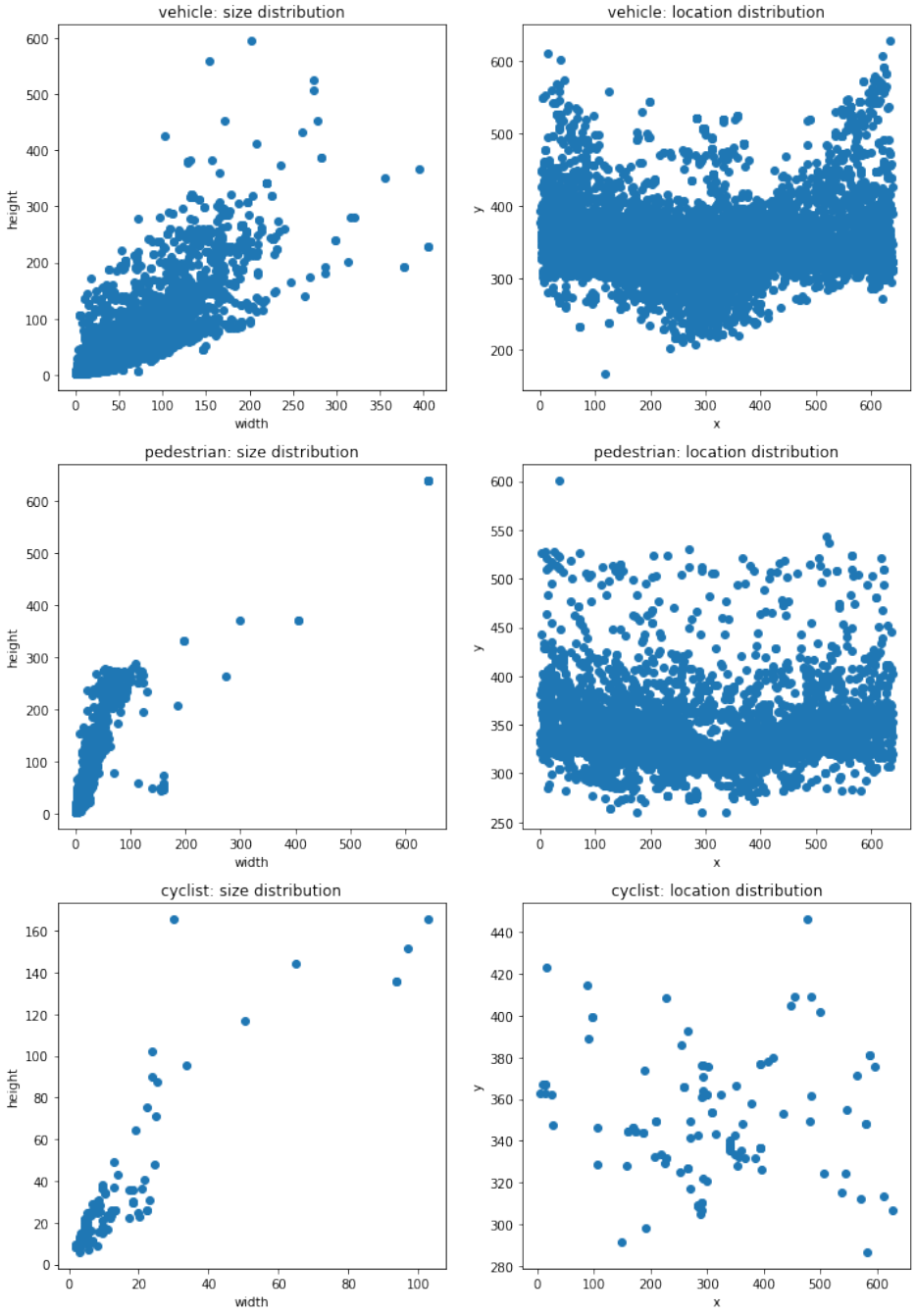
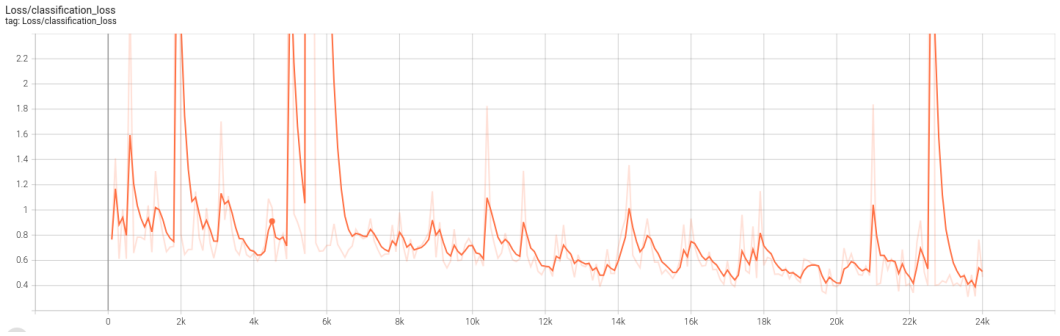


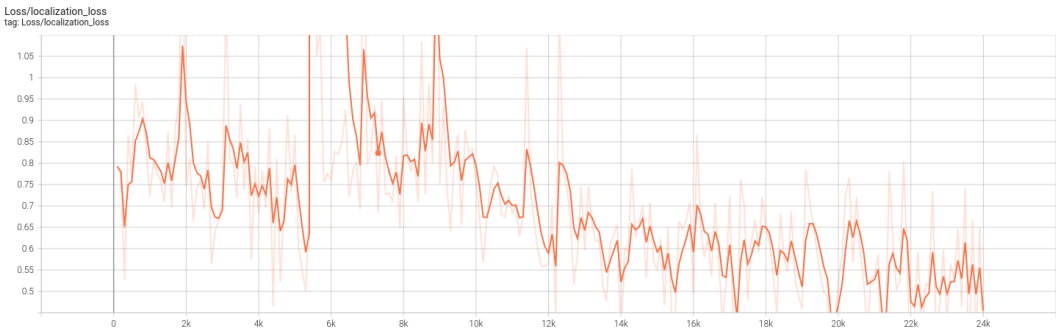

Figure 2: Bounding box size and location distribution.

## 4.2 Improve on the reference

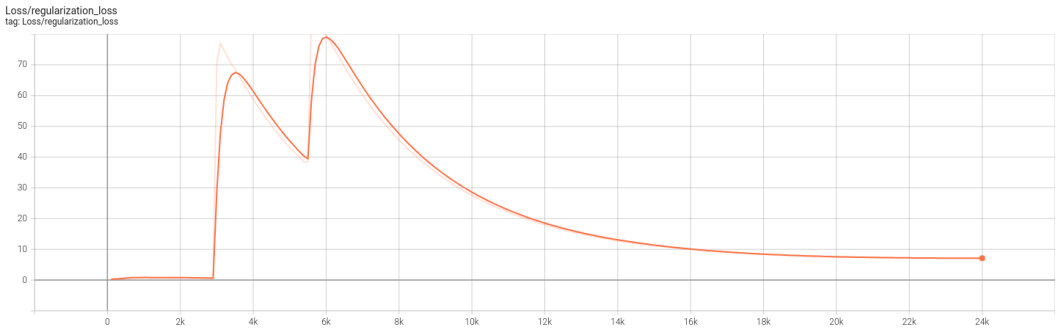
From the reference we get to know that the final total loss stays at 8, which is still a big loss. I explored two strategies to try to improve the performance.



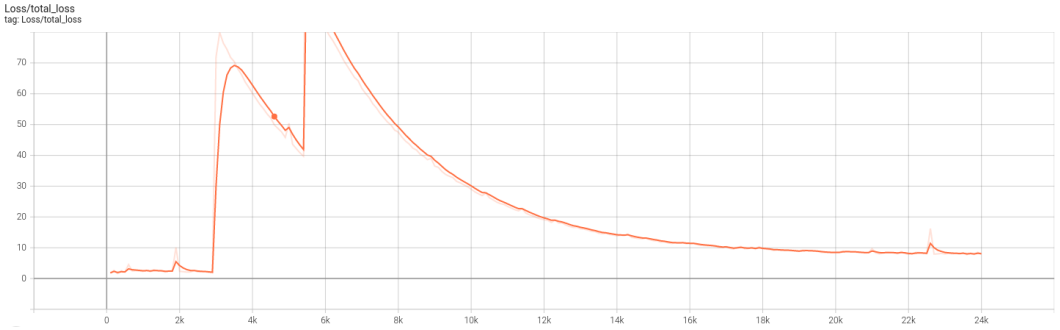

(a) Reference classification loss




(b) Reference localization loss




(c) Reference regularization loss




(d) Reference total loss

Figure 3: Reference training loss curves

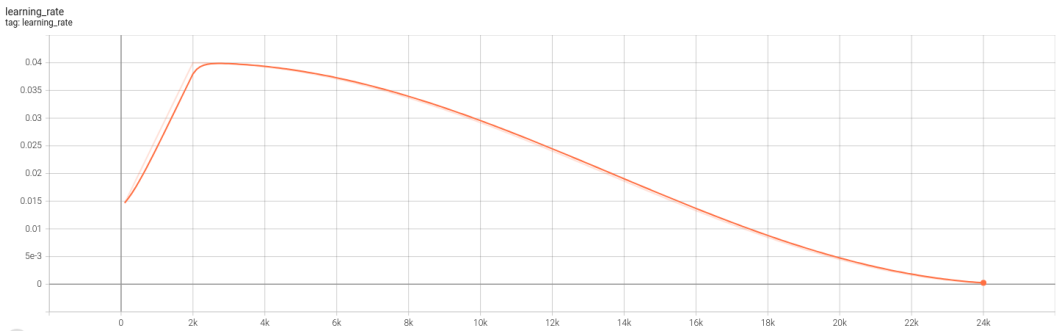



Figure 4: Reference learning rate.

#### 4.2.1 Data augmentation

The reference only used random crop and horizontal flip. There is no augmentation on the color space. As we know from the data, images can be captured at bright or dark environment, clear or blurry condition. We need to apply some augmentation to generate fake images that cover more such scenarios and make the model more robust against brightness and color variations. Making use of the *Explore augmentations.ipynb* and referencing some posts on *Udacity*, I experimented with some common color variation augmentations and decided to add random brightness, contrast variation and also color to gray-scale conversion. The resulted images are listed in figure 5. We can tell that the augmented images have more variations than the original images, which can introduce more robustness to the model.

As a result, the total loss after adding these new data augmentations greatly reduced from 8 to around 1.3. Figure 6 plots the total loss curve at 25000 steps. It is clear that the total loss decreased more rapidly than that in the reference training. We can even expect it will keep decreasing below 1.0 if we run for more steps.

To verify that the new augmentations indeed help in model performance, I evaluated both the reference and new-aug models, using the evaluation code provided in the instructions. The quantitative results are listed in table 1 and 2. From table 1, new-aug model seems to work better than the reference model. All loss values are reduced, especially the regularization loss. However if we compare the mAP results in table 2, new-aug model has smaller mAPs over all metrics, which is kind of interesting.

Table 1: Loss values of reference (default aug) and new-aug models

| Models | localization | classification | regularization | total |
|---|---|---|---|---|
| Reference | 0.638423 | 0.766206 | 7.111167 | 8.515795 |
| New-aug model | 0.622851 | 0.824731 | 0.389706 | 1.837288 |

Table 2: mAP values of reference (default aug) and new-aug models

| Models | mAP | mAP@.50IOU | mAP@.75IOU | mAP (small) | mAP (medium) | mAP (large) |
|---|---|---|---|---|---|---|
| Reference | 0.027682 | 0.060849 | 0.022513 | 0.006454 | 0.097579 | 0.129818 |
| New-aug model | 0.014509 | 0.032250 | 0.010634 | 0.000896 | 0.049091 | 0.073470 |

#### 4.2.2 Use different architecture

From its config file, the reference model is **SSD ResNet50 V1 FPN 640x640**, which has a reported mAP of 34.3 on COCO dataset, according to the official model zoo of the object detection api. I tried

(a) Brightness variation

(b) Brightness variation

(c) Contrast variation

(d) gray-scale conversion

Figure 5: Sample images from data augmentation

to use the other model **EfficientDet D1 640x640**, which has a reported mAP of 38.4, in the hope to boost the performance. I downloaded the config file from the official model zoo and modified it to adapt to our custom training dataset. However the training loss curves don't seem to work out well, as can be seen in figure 7. It fluctuates in a decreasing trend in the first 7000 steps while stays stable in the following steps. It is not a good sign of valid training so I won't post the loss values and mAP results on the test dataset here.

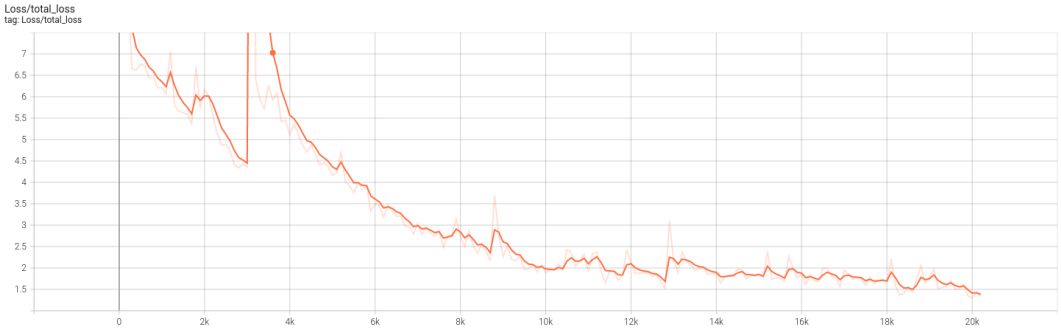



Figure 6: Total loss curve after introducing new data augmentation.

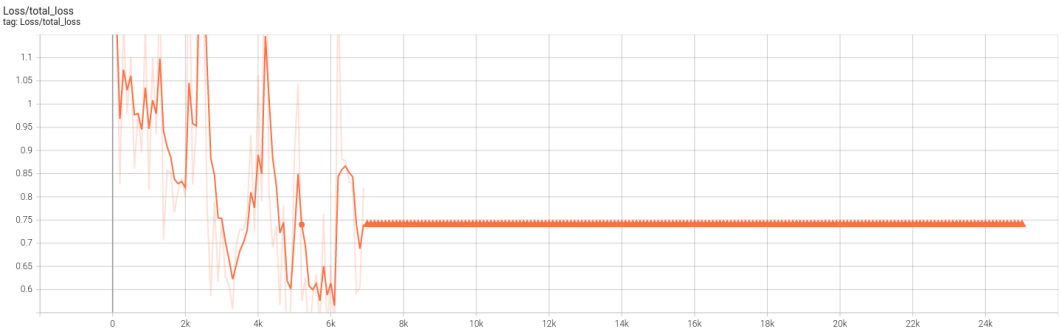



Figure 7: Total loss curve of EfficientDet D1 640x640 model.