Project overview

This project focuses on the detection of agents (e.g. vehicles, cyclist, and pedestrians) in urban environments. The detection is made using a trained neural network (resnet). The goal of the project is to train a neural network such that it will be capable of detecting the agents that are present in the field of view of the camera.

Dataset

Dataset analysis

A representative sample set of the driving conditions for the present project is shown in Figures 1, 2 and 3. These images were extracted from the list of trips contained in the tfrecord files. These files are located in the directory */data/waymo/*.

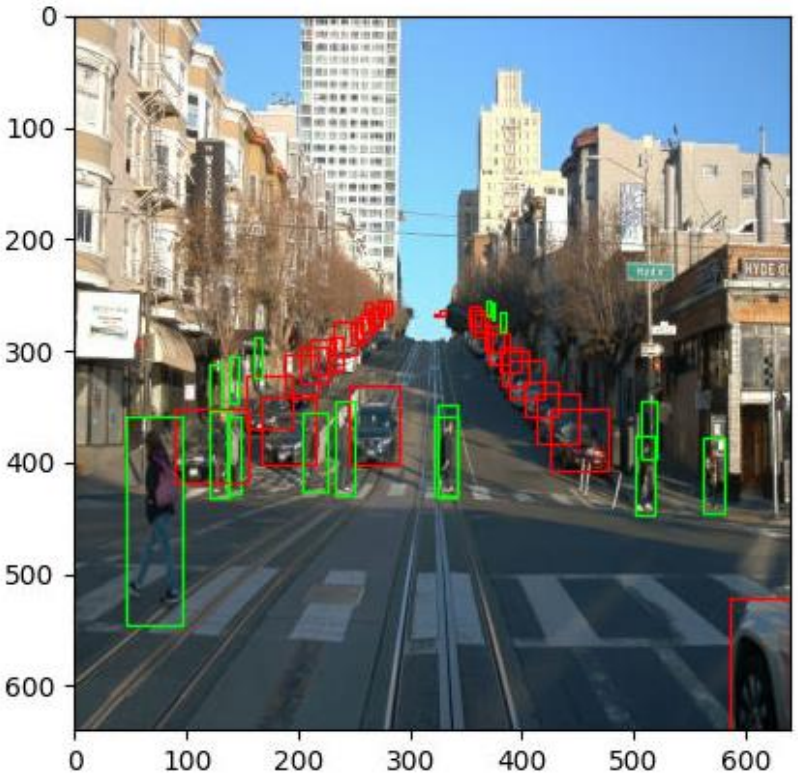

Figure 1. Urban Driving Conditions.

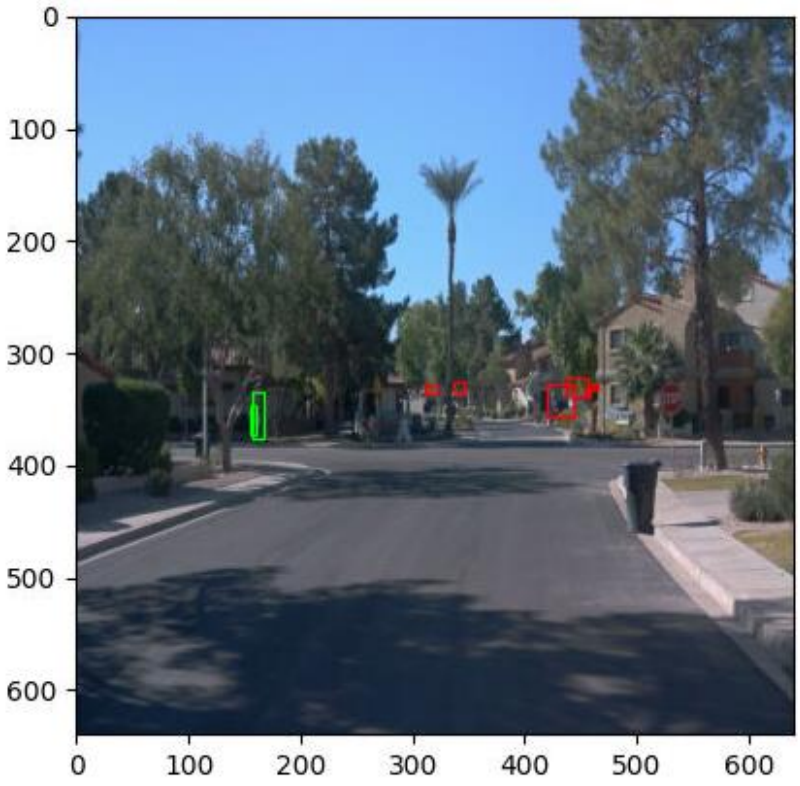

Figure 2. Suburban Driving Conditions.

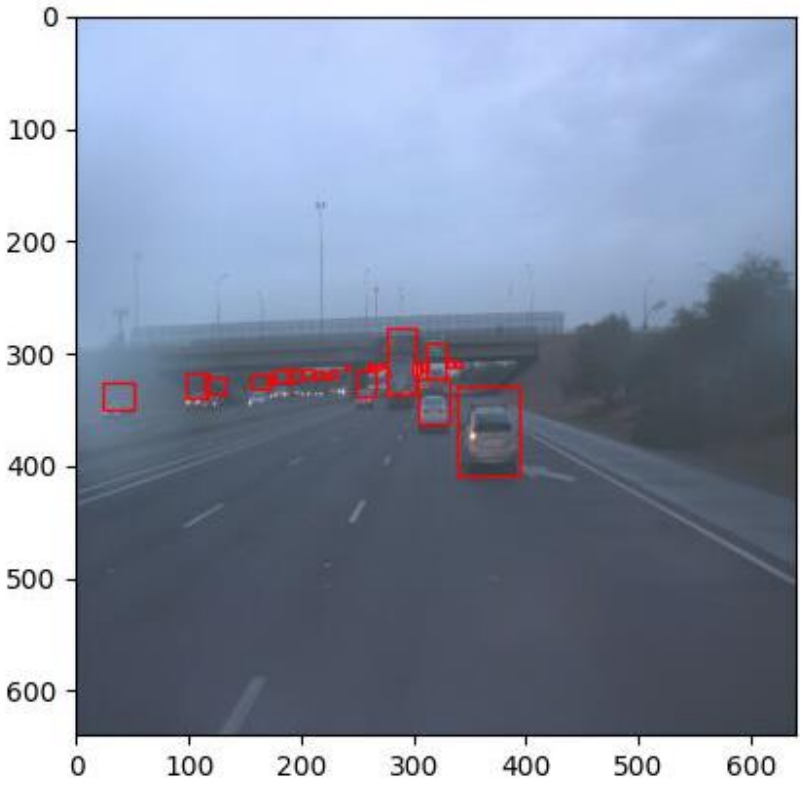

Figure 3. Highway Driving Conditions.

The trips that were provided in this project were divided into different driving scenarios; city roads (Figure 1) which contained a high density of agents (vehicles, pedestrians, or cyclists), suburban roads (Figure 2) where the agent density was not as high as in the city, and highways (Figure 3) where the agent density was not as high as in city roads but usually higher than on the suburban roads.

It was also observed that the city roads had a high density of traffic signals, for example; pedestrian crossing marks, stop signs, speed limit signs, etc. Suburban roads had a lower density of traffic signs (some of the streets didn't even have lane dividers). Finally, highways had a medium density of traffic signs.

The driving conditions that were observed in the images of the trips were mostly for day driving conditions but it was also observed that some images included nighttime driving conditions. Furthermore, most of the driving conditions corresponded to good/fair weather conditions and there were some bad (rain only) conditions. No snow conditions were present in the provided dataset.

A sample set of the driving conditions can be observed in Figure 4.

Figure 4. A Sample of Driving Conditions.

Based on the driving conditions, it was decided to divide the into three sections; object density, light, and weather. The driving conditions and its categories are listed in Table 1.

Table 1. Driving Conditions.

Driving Condition	**Categories**	
Object Density	Low (Suburban/Highway)	High (City)
Light	Day	Night
Weather	Fair	Bad (Rain/Adverse)

The driving conditions are explained below:

- Object density. This condition refers to the number of cars/pedestrians/bicycles that appear in the image. There are two categories assigned to this driving condition.
    - Low. This category is usually assigned to suburbs and highways.
    - High. This category is usually assigned to city driving.
- Light conditions. Related to the light conditions of the images. The categories are; day and night driving.
- Weather conditions. This is related to the visibility of the driving conditions. The categories here are; fair which is related to good visibility conditions and bad which relates to low visibility due to rain or other conditions that could affect the visibility.

All the trips that were provided in the tfrecord files were inspected and proper labels were assigned to each one of them. The statistics for each of the driving conditions are shown in Figures 5, 6, and 7.

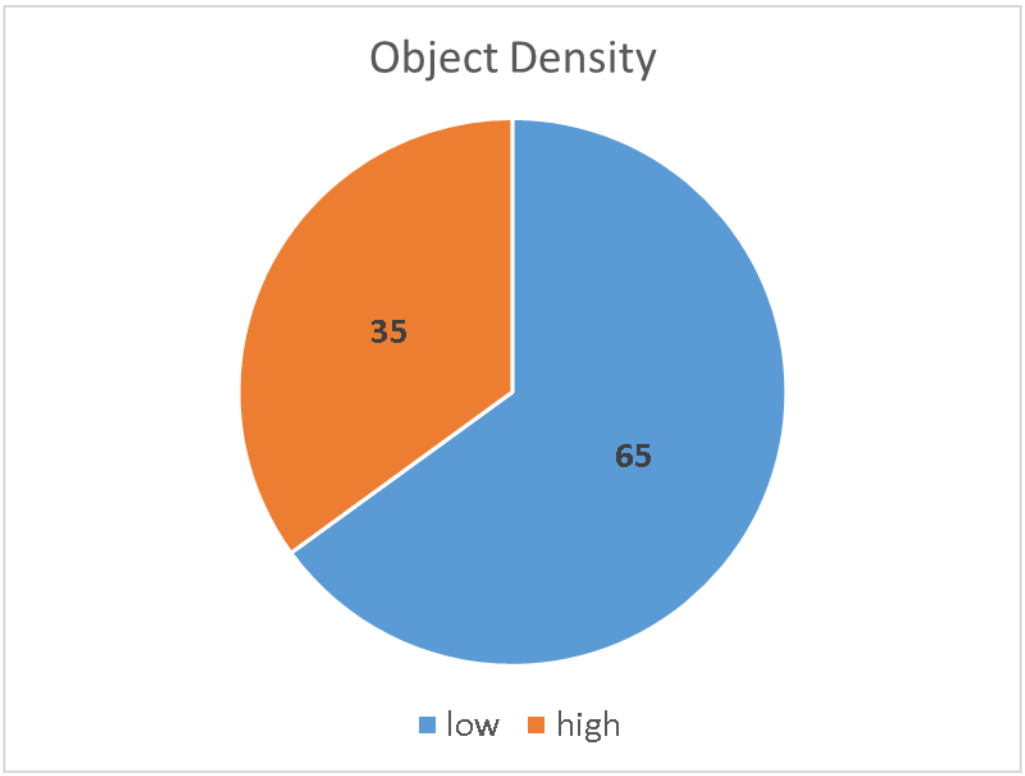

Figure 5. Object Density Trip Classification.

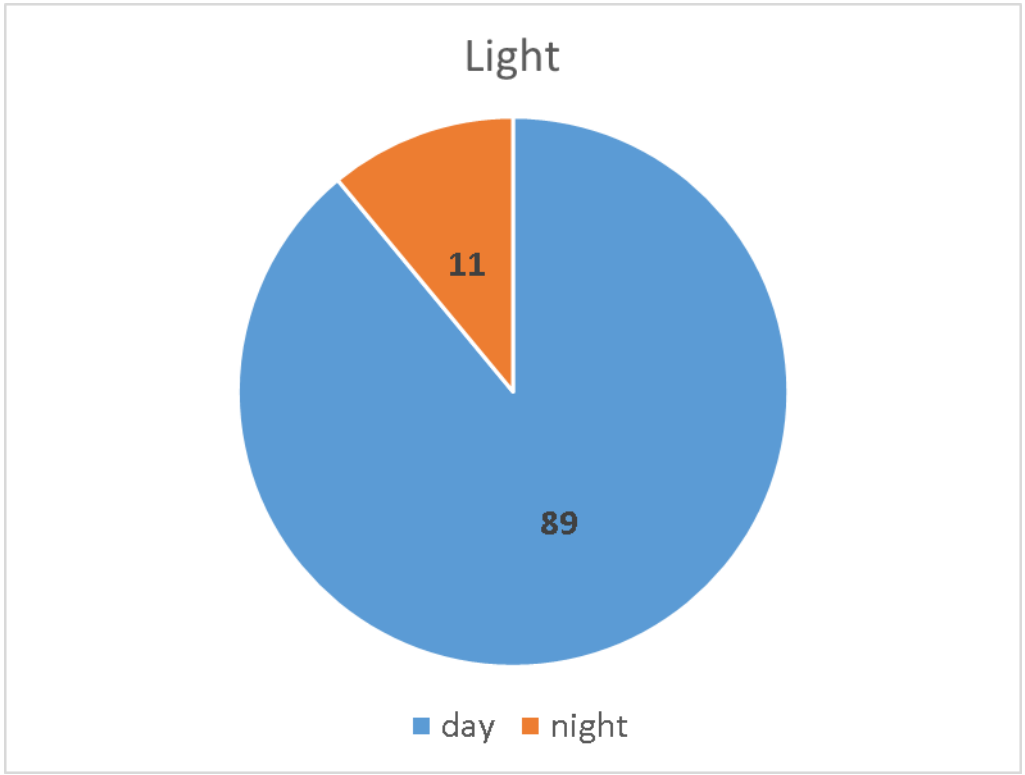

Figure 6. Light Conditions Trip Classification.

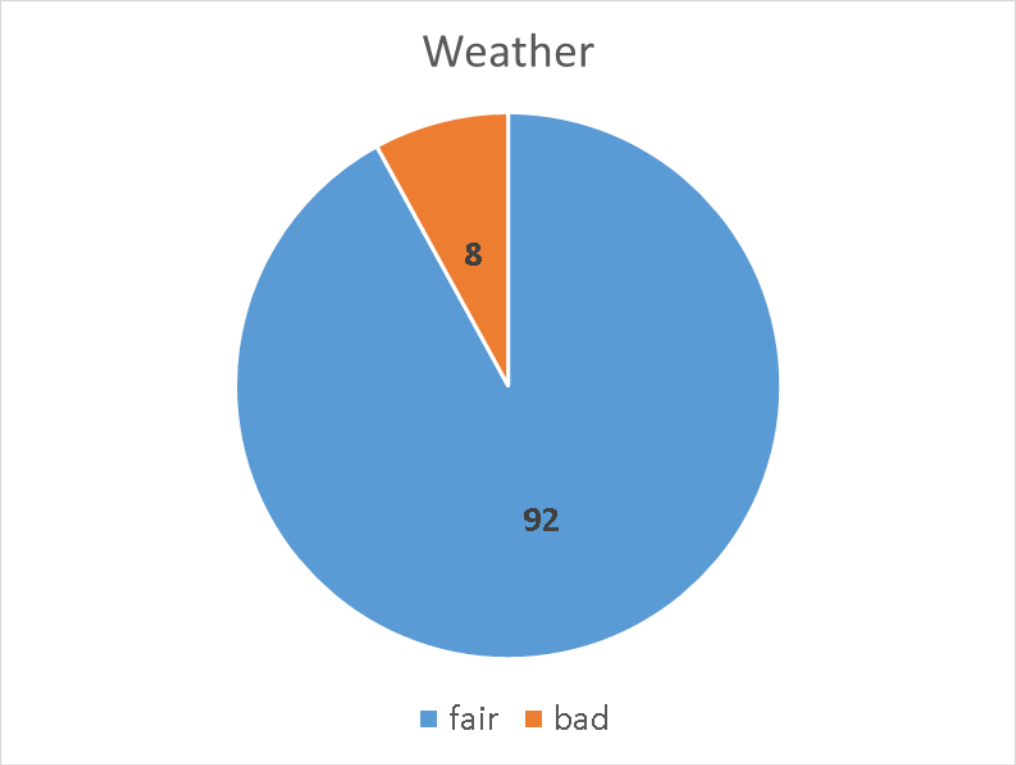

Figure 7. Weather Conditions Trip Classification.

The data labels used in Figures 5, 6, and 7 correspond to the number of trips that met the conditions for each one of the driving categories. There were 100 trips in total. The classification was performed by inspecting one frame for each of the trips (tfrecord file inspection). This process was carried out using a python script located in the following location *my_scratch_files/file_splits_v2.py*. The script can be launched using the following command *python file_splits_v2.py*. The execution of the script generates an output file that contains the trip name (tfrecord file name) along with the driving conditions. The name of the output file is *trip_description.txt*.

Trip Split Generation

It was decided to create the trip splits using the following fraction of the total number of trips; test splits = 0.1, validation splits = 0.15, and training splits = 0.75.

It was also decided that the splits should be created using the information obtained from the exploratory analysis, shown in Figures 5, 6, and 7. This process was performed by running the script *trip_split.py* located in *my_scratch_files* directory (*python trip_split.py*). The procedure to create the splits is explained below:

- The process starts with the test split. Here the number of trips that belong to this split is equal to 0.1*100 = 10. Then python script *trip_split.py* looped throughout the driving conditions in the following order; first weather conditions, then light conditions, and finally object density conditions. The script retrieved the total number of trips for each driving condition and multiplied it by the fraction of the test split. For instance, Figure 7 shows that for the test split we needed 0.1*8 = 0.8 ~ 1 trip for bad weather conditions, Figure 6 required that we multiplied 0.1*11 = 1.1

~ 1 trip for night driving conditions, and Figure 5 told us that we needed 0.1*35 = 3.5 ~ 4 trips for high object density driving conditions.

- The previous point tells us that one trip was needed for the bad weather and night driving conditions for the test split. Then the script *trip_split.py* randomly selected a trip from the main trip list. Once the trip was selected it then was checked for whether it satisfied the following three driving conditions (weather=bad, light=night, object density = high). If the trip satisfied these three conditions then it was added to the test split and the driving conditions counters were updated as follows: number of trips for bad weather = 1-1=0, night driving conditions = 1-1 = 0, high object density = 3-1 = 2.
- Then two more trips were randomly selected and it was checked that the following driving conditions were satisfied; weather=fair, light=day, object density = low. If the checks were true then these two trips were removed from the main trip list and added to the test split. The counters were all updated to zeros, bad weather = 0, night driving conditions = 0, high object density = 0.
- The last step in the test split was to randomly select 10 – 3 = 7 trips with the driving conditions weather=fair, light=day, object density = low. The trips were also removed from the main trip list.
- The procedure described for the test split is repeated for the validation split but the fractions were updated, instead of 0.1 a 0.15 fraction was used. All the selected trips were removed from the main trip list as well.
- Finally, the trips left in the main trip list were assigned in the training split.

The aforementioned procedure was implemented in the *trip_split.py* script that is located in the *my_scratch_files* folder. It should be mentioned that the *trip_split.py* script created the main trip (the list that contains all the available trips, tfrecord files) list from the file description file *trip_description.txt*. When executing the script *python trip_split.py* we can see the following output in the terminal:

```
“””
********** weather **********************
test split
Number of trips in weather with bad conditions for test dataset = 1
validation split
Number of trips in weather with bad conditions for validation dataset = 1
********** light **********************
test split
Number of trips in light with night conditions for test dataset = 1
validation split
Number of trips in light with night conditions for validation dataset = 2
********** density **********************
test split
Number of trips in density with high conditions for test dataset = 4
validation split
Number of trips in density with high conditions for validation dataset = 5
Number of trips in test split is = 10
Number of trips in validation split is = 15
Number of trips in training split is = 75
********* TEST *****************
Object density info
Number of trips with high object density = 4
Light conditions info
Number of trips during night = 1
weather info
Number of trips with bad weather  = 1
```

********* VALIDATION **********
Object density info
Number of trips with high object density = 5
Light conditions info
Number of trips during night = 2
weather info
Number of trips with bad weather = 1
********* TRAINING **********
Object density info
Number of trips with high object density = 26
Light conditions info
Number of trips during night = 8
weather info
Number of trips with bad weather = 6
“””

The *trip_split.py* script generated three output files that contained the name of each trip and the split that they belong, for example; *test_split.txt*, *validation_split.txt* , and *training_split.txt*.

Finally, the script *create_splits.py* was modified in order to read the information from the files; *test_split.txt*, *validation_split.txt*, and *training_split.txt* in order to create the appropriate split directories.

Training

The training process was performed using the default contents of the *pipeline_new.config*. The output of the tensor board GUI is shown in Figures 8 and 9.

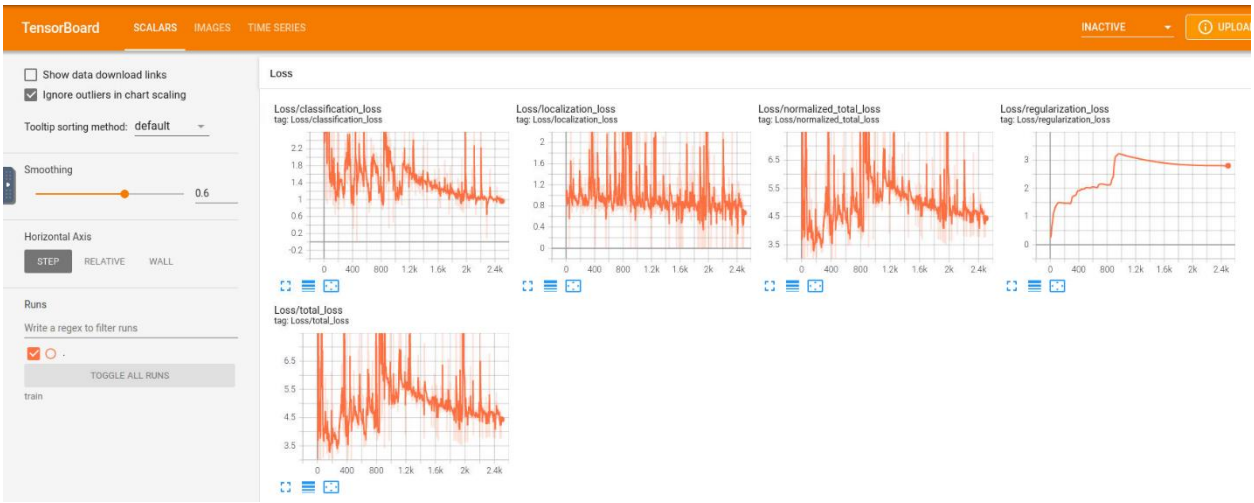

Figure 8. Tensorboard Output – Default Model.

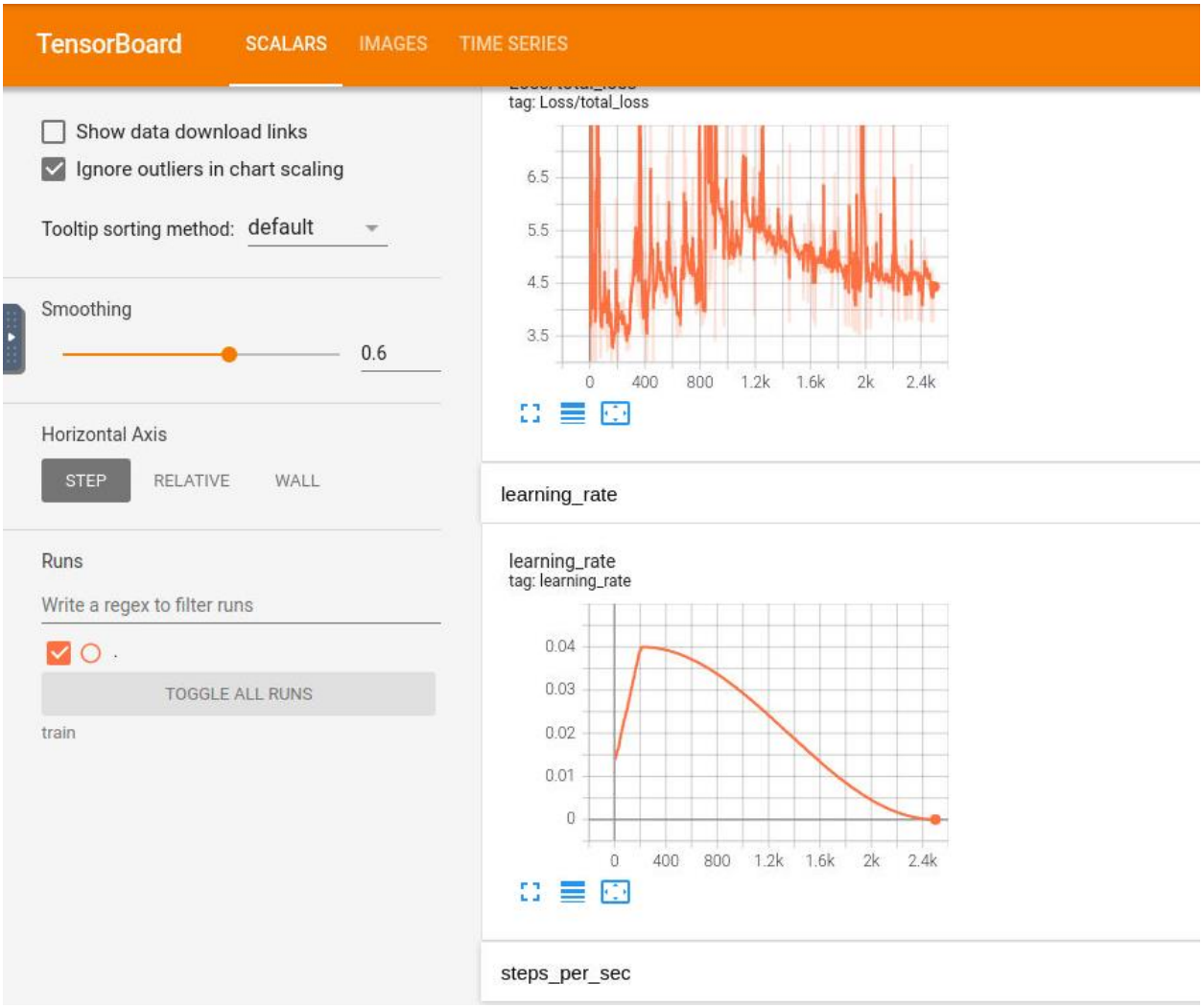

Figure 9. Tensorboard Output – Learning Rate Default Model.

It can be seen in Figure 8 that the default settings for training the detection networks are not optimal since the total loss is around 4.5. In order to have a fair performance detection, the total loss should be between 1 and 2.

There was an attempt to run the evaluation process while the training was happening in the Udacity workspace but it was unsuccessful due to an out of memory issue. Other users have reported this issue in the past as it is documented in the knowledge center. The knowledge center of this Nanodegree recommends using a smaller batch size to avoid the out of memory issue but it was decided for the present study that tracking of the total loss during training should be enough to decide if the detection network has reached enough accuracy.

Augmentations.

The following augmentations were explored and included in the training of the final version of the detection network; random_crop_image, random_rgb_to_gray, random_adjust_contrast,

random_adjust_brightness, random_black_patches. A sample set of the augmentations is shown in Figures 10 and 11.

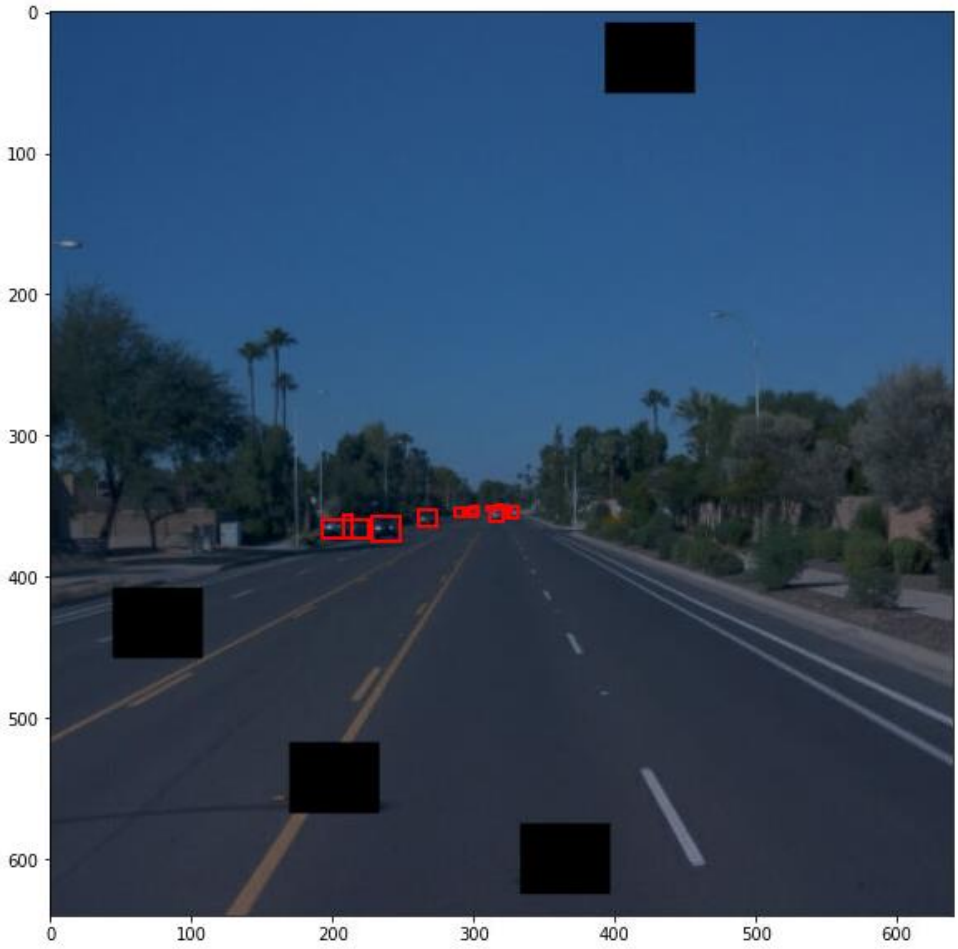

Figure 10. Augmentations Sample 1.

Figure 11. Augmentations Sample 2.

The main goal of the augmentations is to create more variations for the training of the detection network without the need to generate more measurements (more trip images). The variations in the images that the detection network uses for training will help to improve the detection performance of the network.

Improvement of training parameters.

It was decided to reduce the default value of the learning rate to improve the training of the detection network. Other modifications include changing the optimizer from momentum_optimizer to rms_prop_optimizer, the learning rate variation was decreased from 2500 steps to 200 steps. The modification were included in the *pipeline_new.config* file that is located in the directory experiments/exp4. The Tensorboard results of the training of the modified network are shown in Figures 12 and 13.

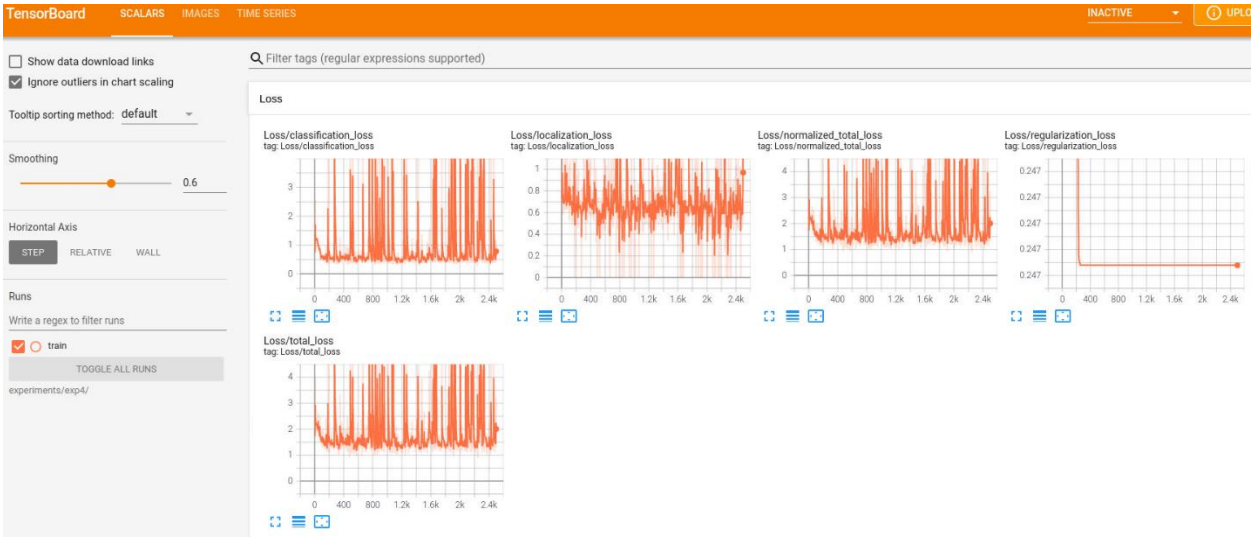

Figure 12. Tensorboard Output – Modified Model.

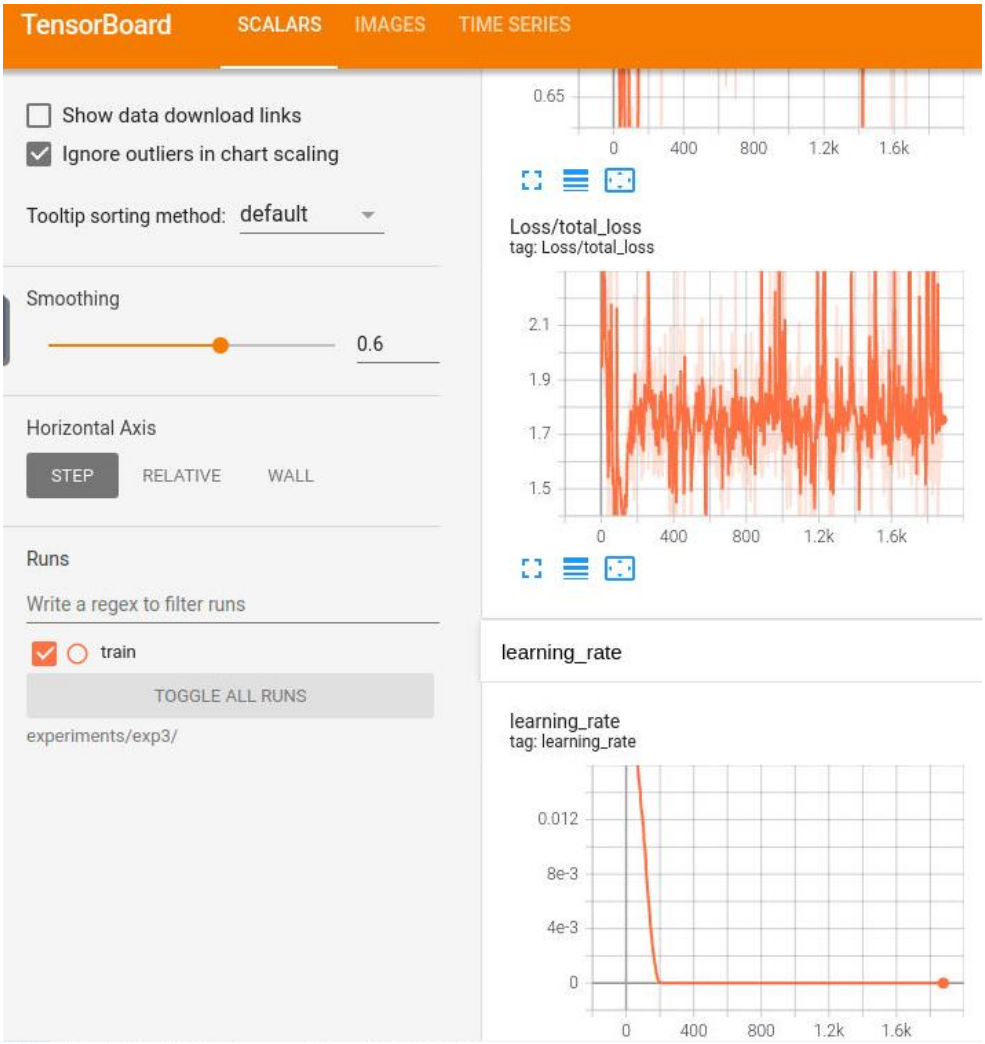

Figure 13. Tensorboard Output – Learning Rate Modified Model.

Detection Animation Comments

An animation of the trained detection network was created and uploaded in the workspace directory. The file name is *animation4.gif*. This animation file can be also be found in the Git hub project that is included in the current submission for the Udacity evaluators.

Conclusions

The submitted detection network reached a total loss within the acceptable range (between 1 and 2) but it can certainly be improved. Other strategies to be tested would be to use other architectures such as; AlexNet, VGG, Inception, etc.

It is proposed in this study to create three driving conditions and use the information to generate the splits; test, validation and training. Other driving categories could be proposed but it is believed that the three proposed categories were a good start and they cover most of the driving conditions of the trip dataset. Other areas of opportunity include the automation of labeling of the trips according to the driving conditions (weather, light, and object density).

Finally, the augmentation can be further improved by adding other strategies such as motion blurring. It was interesting to notice that the motion blurring capability is part of the albumentations python library but it is not available in the *preprocessor.proto* list. Further efforts can be focused in how to include the albumentations from the python library in the *pipeline_new.config* file.

Further Comments

I couldn't finish this final project right after I finished the materials of the Computer Vision section because of several issues that I had with the Udacity workspace. I think that the Udacity workspace is a great feature of this learning platform but it needs more attention from Udacity technical support. There are other workspace issues that were reported by several users months ago but the issues are still present up to this day. For example:

- Jupyter notebooks constantly crashing due to a Firefox issue. Wouldn't be better to install Chrome browser and run the notebooks with Chrome? You already mention this topic in the project instructions so why not install Chrome and make it the default browser when running the notebooks?
- Not enough disk space. If the problem is the lack of disk space, why not tell the students that instead of moving tfrecord files to the split directories that they should create soft links to the files? This will avoid the disk space limitations of the workspace.
- Tensorboard alias does not exist. There is no alias to run the tensorboard to babysit the training of the network. The lack of the tensorboard alias was reported by several users in the knowledge center.
- Out of memory issue when trying to run the network training and evaluation at the same time. A warning note should be included in the project instructions so the user know how to prevent this issue (batch size decrease).

- Furthermore, the instructions are not clear about how to launch the validation process. The project instructions give the impression that the student should run the training exercise first and then launch the validation process. It is in the knowledge center where this issue is discussed by other students.

Since this is the first project of the nanodegree, I think that the students should spend most of their time working in the data analysis and the training of the network. I think that the aforementioned issues cause distractions in the completion of the project.