Udacity Self-Driving Vehicle (System Integration Final Project) Report and README

TEAM MEMBER	EMAIL ADDRESS
ROY, ABHISHEK	aroy24@ford.com
CHOE, KYU	kchoe@ford.com
PONTISAKOS, CHRISTOPHER	cpontisa@ford.com

Video File Recording

A video recording was captured since latency issues exist in the workspace which are larger than the latency issues on our own virtual workspaces. In the same folder as this readme, there will be a video labeled:
/home/workspace/CarND-Capstone/self_driving_car_nanodegree_program 8_26_2019 5_31_27 PM.mp4

This video shows the performance of the car under our own workspaces. This performance is still degraded from when we run with the camera off. The camera latency adds additional negative performance on our workspaces. This has been discussed with mentor who said this issue would be eliminated on the tester's machines. We expect better performance than even this video shows because of this.

In-Vehicle Run Instructions

The traffic light detector model in code is for the simulation use. **For running in vehicle, please replace the model** found in:
…\tl_detector\light_classification\graph_optimized.pb
with the model in the subfolder:
…\tl_detector\light_classification\converted_optimized_mobilenet_carla_alex_model\graph_optimized.pb

ROS integration

In the file 'src/tl_detector.py/tl_detector.py', the function 'get_light_state', was updated and instead of returning fixed traffic light state from a config file, it was updated to provide traffic light state in real time. The image from camera was sent to the function 'def get_classification' in the file 'src/tl_detector.py/light_classification/tl_classifier.py'. Next, using our trained traffic light classifier model, each of the input image was processed to predict state of the traffic light and the information is passed back to the function 'get_light_state' in the file 'src/tl_detector.py/tl_detector.py'.

Traffic Light Detection Model

There are number of choices team considered building a traffic detection module:

- Create a brand new network from scratch
- Retrain traffic sign classifier developed in previous lesson with traffic light data set
- Retrain pre-trained published models on web

The first approach involves trial and error development process with performance benchmarking with existing pre-trained network models. This approach would be attractive only if enough time is granted.

The second approach is the fastest way to deploy already developed network for our purpose, but it is not a true traffic light "detector" since it processes the whole image for a single object.

Second Method: (The model/approach below was only briefly assessed and was therefore discarded for the third method described above)

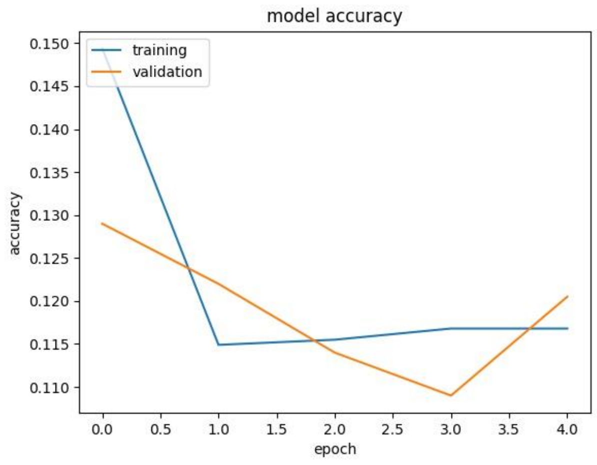
<u>traffic light classification by ".h5" model</u>
Training data was collected by running simulation in autonomous mode, and saving each of the images (received from the callback function 'image_cb' in the file 'src/tl_detector.py/tl_detector.py') in a folder along with corresponding traffic light state in a csv file. Using this data, the model previously designed for the behavior cloning project, was retrained from scratch. The summary of relevant parameters are:

Total no. of data images collected : 425
Total no. of images filtered : 85
Total no. of remaining data images : 340
Total no. of training images : 272
Total no. of validation images : 68
Total no. of epoch for training : 5
Total no. of steps per epoch for training : 100
Total no. of data items in a batch per step per epoch for training : 100
Total no. of epoch for validation : 1
Total no. of steps per epoch for validation : 200
Total no. of data items in a batch per step per epoch for validation : 10
Final training accuracy : 0.1168
Final validation accuracy : 0.1205

The training & validation accuracy was found quite low and thus, when the generated model was used for testing with simulation in real-time traffic light classification, the performance was found to be poor.

Plan for improvement:
a) Train model with improved parameters for epochs, steps per epoch and data items per epoch
b) better data collection, i.e, drive manually in simulation mode and capture most of images
- Adjust the ratio of image data with traffic lights compared to without traffic lights.
- Adjust the angle of the car when collecting traffic light training images
c) Improve model by adding additional complexity with additional layers and dropouts

Summary of results of this trained model which was discarded



Third method for Classifier Training (BEST – Used for submission)

We chose to implement the 3rd approach for best accuracy achievable in given time frame. Also tested was the second approach which will be later described. We tested inception V2 and mobilenet V2 models from TensorFlow model zoo that were pre-trained on COCO database. Both network worked fine in our applications (simulator and Udacity test track), but the latency issue with "camera on" during simulation forced us to choose mobilenet V2 for our project for faster performance.

The largest issue we encountered for re-training the ready-made networks is the software version requirements from Udacity. TensorFlow Object Detection API requires minimum tensorflow v1.4 at least and most of the networks developed recent requires tensorflow v1.12. It could have been helpless unless a former Udacity student commented that inception and mobilenet converted to tensorflow v1.4 also worked with tensorflow v1.3. Following his guide, the model was retrained in tensorflow v1.12 environment and then exported from tensorflow v1.4 that is implemented in our project.

DataSets for Training

Preparing training data set could be one of the most time consuming during model developing process if done manually from scratch. We used multiple pre-labelled data sets available on various websites. To merge multiple data sets from different sources, the label input file format needs to be consistent to be processed by a customized Tensor Flow Record writing utility. Total 1,555 test site samples were collected and were split by 3:1 ratio for training and evaluation. Total 422 samples were split into 316 and 106 samples for training and evaluation respectively.

Dataset	Training	Evaluation
Simulation	633	212
Test Site (Carla)	1593	531

Performance

Each re-trained network for real world has been tested manually with 12 selected images that were not used in training/evaluation. Both Inception V2 and mobilenet V2 trained on our dataset had good accuracy (100% correct predictions). For speed, we chose mobilenet V2 due to latency issue with steering control. Also the size of mobilenet v2 model (13.4 MB)was significantly less than inception model (50.7 MB).

Model	Accracy
Inception V2	100%
Mobilenet V2*	100%

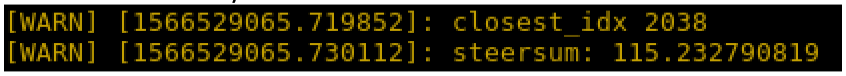
* implemented in project

Twist Controller and PID

When camera turns on, our simulator shows 250+ ms of latency. This led us to believe the yaw-controller was insufficient for steering control since the car could not properly follow the waypoints. It wasn't until after PID was implemented and tuned (PID tuning was able to follow the road to perform laps of the testrack with this latency, but it would often cut into adjacent lanes) that we decided to disable the camera subscription. Suddenly we saw no latency, the yaw controller worked pretty well, and the following investigation was performed to determine if PID still provided significant advantage over the simple yaw controller. We have no way of knowing if the PID tuning works with the camera on since our simulators are not capable of executing without this latency. We have received feedback that this latency does not exist on tester machines.

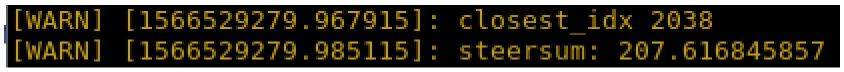
Assumption: Following the path with the least steering is most optimal.
Metric Sum of squared steering inputs [deg] until third light (waypoint 2038). This penalizes larger steering angles more than smaller steering angles

Yaw Controller Only:



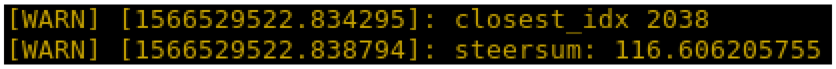
```
[WARN] [1566529065.719852]: closest_idx 2038
[WARN] [1566529065.730112]: steersum: 115.232790819
```

[kp , kd , ki] = [1.55 , 0.30 , .05]
Yaw Controller + PID



```
[WARN] [1566529279.967915]: closest_idx 2038
[WARN] [1566529279.985115]: steersum: 207.616845857
```

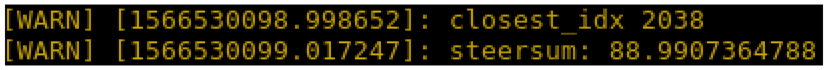
[kp , kd , ki] = [1.55 , 0.30 , .05]
Yaw Controller + PID + LPF



```
[WARN] [1566529522.834295]: closest_idx 2038
[WARN] [1566529522.838794]: steersum: 116.606205755
```

The additional PID + LPF has not added benefit over the yaw controller and so re-tuning PID is in order. Lowering the proportional term and integral term will result in vehicle performing fewer minute adjustments.

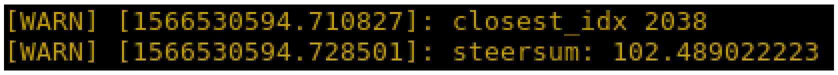
[kp , kd , ki] = [0.50 , 0.25 , .01]
Yaw Controller + PID + LPF



```
[WARN] [1566530098.998652]: closest_idx 2038
[WARN] [1566530099.017247]: steersum: 88.9907364788
```

The numbers look great and show a 22.7% improvement over the Yaw Controller Only baseline. The problem is that the relaxation of the proportional and integral controller often results in the vehicle cutting over onto adjacent lanes. Finding middle ground for the proportional control may fix this.

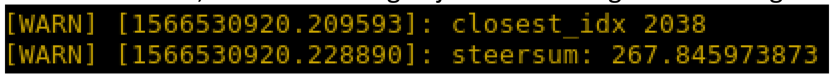
[kp , kd , ki] = [0.70 , 0.40 , .01]
Raise proportion constant to middle ground. Car stays well centered in lane!
Yaw Controller + PID + LPF



```
[WARN] [1566530594.710827]: closest_idx 2038
[WARN] [1566530594.728501]: steersum: 102.489022223
```

We see a more modest 11% improvement over the baseline yaw controller!

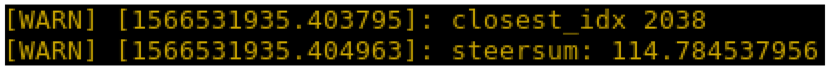
Without the LPF, minute steering adjustments integrate for a large steering squared sum.



```
[WARN] [1566530920.209593]: closest_idx 2038
[WARN] [1566530920.228890]: steersum: 267.845973873
```

LPF values used for steering were tau = 20, ts = 5

It is worth testing without PID if the Yaw Controller + LPF improves results
Yaw Controller + LPF



```
[WARN] [1566531935.403795]: closest_idx 2038
[WARN] [1566531935.404963]: steersum: 114.784537956
```

Only a 0.5% improvement. This shows that the LPF acts mostly on the PID terms and has little effect on the Yaw Controller direct output.