Capstone project

Daisuke Miyazaki

Here is my writeup for this last project in Udacity Self-Driving-Car nanodegree program.

Overall workflow

In the simulator provided from Udacity, the goal for this project is to drive the car autonomously. There're also traffic signals, so the car should behave accordingly such as stopping by a red sign and throttling by a green sign.

Overall, the software is mostly written base on ROS (Robotic operation system) in python. Below I listed 4 nodes of code that I wrote for this project. I mainly followed the provided walkthroughs from Udacity lectures.

Waypoint updater node

In this node, it subscribes to /Base_waypoint once and obtains the entire trajectory that the car is supposed to follow in the simulator. It also subscribes to /Current_pose to obtain the vehicle's coordinates in real time.

Bases on these data, it generates a number of waypoints – or the vehicle's future trajectory – at the front of the vehicle. This number corresponds to LOOKAHEAD_POINTS variable. With this, it publishes the waypoints to *waypoint follower node.*

DBW node

In this node, it publishes brake, steering, and throttle commands to the simulator.

Among electrically operated vehicles, a self-driving feature should be used during safe operation in the vehicle. In another word, this should disengage for safety reason if the signals for controlling the vehicle has some issues. For this reason, this node subscribes to the /vehicle/dbw_enabled topic, where the node receives a Boolean signal. This value is true when the vehicle drives automatically or false when it's manually driving.

Traffic light detection node

In this node, it takes images from the simulator, processes them, and publishes upgraded

waypoints to waypoint updater node.
Vehicle/traffic_lights topic includes all the coordinates of signals in the simulator. With get_closest_waypoint function, it searches for the closest waypoint to the vehicle at the time and publishes it to waypoint updater node. To distinguish the state of the signal, I wrote up a get_light_state function in tl_decetor.py file.

Here, it's possible to receive data about the state of the signal – whether it is being red, green and so forth through /trafficLight topic. Also, it can confirm the state by using an image from the camera mounted at the front of the vehicle. These approaches can be described as 2 distinct methods, and I will discuss it later with its following results.

Waypoint updater node

In the end, this node subscribes to traffic light detection node and updates the original waypoints.

Building a Classifier

At this point, tl_classifier wants to receive the traffic light state which comes from the closest to the vehicle(needless to say, this traffic light should be at the front of the vehicle). To return this value, there're 2 different approaches; getting the state directly from the signals or recognizing the state of the signals through camera images by processing the images.

Detecting traffic signals by processing

Using image data from the camera enables me to process them to classify the traffic light. More precisely, on the simulator I drove a car and turned on the camera button, and obtained the images through /image_color topic. Converting the image from RGB to HSV type which comes at 30 Hz , I set up thresholds for red, green and yellow colors. Scanning the images one by one allows to detect any images that exceed the threshold, yielding the type of signal the vehicle is facing. As a result, this code can return the state of the target light. The code is in /CarND-Capstone/ros/src/tl_detector/Previous_work/tl_classifier_5.py.

One drawback is that this method only works within this simulator. For example, in a case of driving in real world, red advertisement signs or posts can be misinterpreted as a red signal. Therefore, it's critical to consider the shape of traffic lights as well.

Detecting traffic lights by using Machine learning

With the described method above, I detected the color of the signals by using the camera. Although, this time I built a model for machine learning and attempt to detect the traffic lights. This enables me to classify the shape of the signals and it becomes more applicable to real use cases outside of simulation as well. The whole codes are in /CarND-Capstone/train_nn directory.

Collecting data

** With this submission, I'm not including the data.

First, I store image data to files by using cv2.imwrite function. At the same time, I labelled each type of picture with its color and distinct numerical value; 0 for RED; 1 for YELLOW; 2 for GREEN; and 3 for UNKNOWN.

Then I loaded the images through their absolute path, and stored them to traffic_light_data.csv file using csv module.

I ended up collecting about ~500 images as dataset for the training.

Model architecture

After the steps above, I build a neural network architecture to be trained with the data. The following is the architecture that I used for this project.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 600, 800, 3)       0
_________________________________________________________________
lambda_1 (Lambda)            (None, 96, 96, 3)         0
_________________________________________________________________
lambda_2 (Lambda)            (None, 96, 96, 3)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 94, 94, 8)         224
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 47, 47, 8)         0
_________________________________________________________________
dropout_1 (Dropout)          (None, 47, 47, 8)         0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 45, 45, 16)        1168
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 22, 22, 16)        0
_________________________________________________________________
dropout_2 (Dropout)          (None, 22, 22, 16)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 20, 20, 32)        4640
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 10, 10, 32)        0
_________________________________________________________________
dropout_3 (Dropout)          (None, 10, 10, 32)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 8, 8, 64)          18496
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 4, 4, 64)          0
_________________________________________________________________
dropout_4 (Dropout)          (None, 4, 4, 64)          0
_________________________________________________________________
dense_1 (Dense)              (None, 4, 4, 128)         8320
_________________________________________________________________
flatten_1 (Flatten)          (None, 2048)              0
_________________________________________________________________
dense_2 (Dense)              (None, 4)                 8196
=================================================================
Total params: 41,044
Trainable params: 41,044
Non-trainable params: 0
_________________________________________________________________
Epoch 60/60
11/12 [==========================>...] - ETA: 0s - loss: 0.0178 - acc: 1.0000Epoch 00059: val_acc did not improve
12/12 [==============================] - 6s - loss: 0.0176 - acc: 0.9974 -
```

val_loss: 0.0254 - val_acc: 0.9857

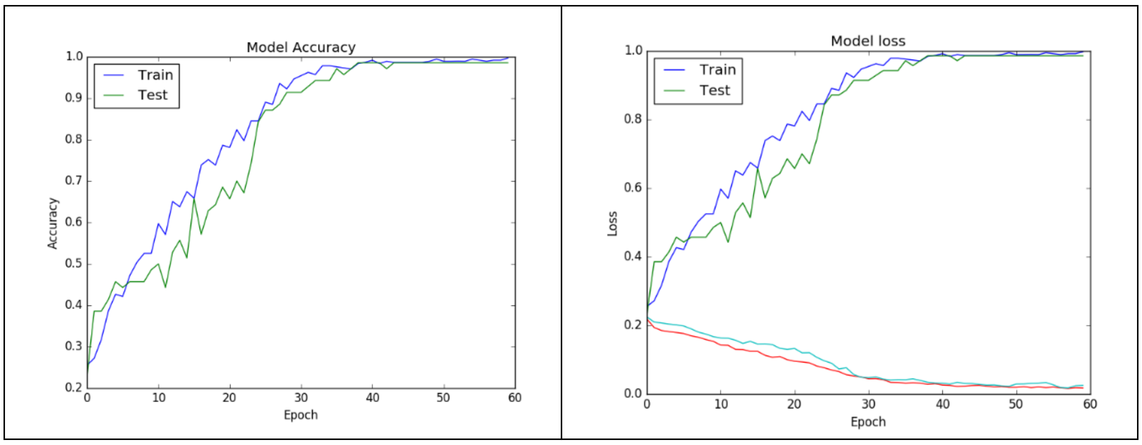
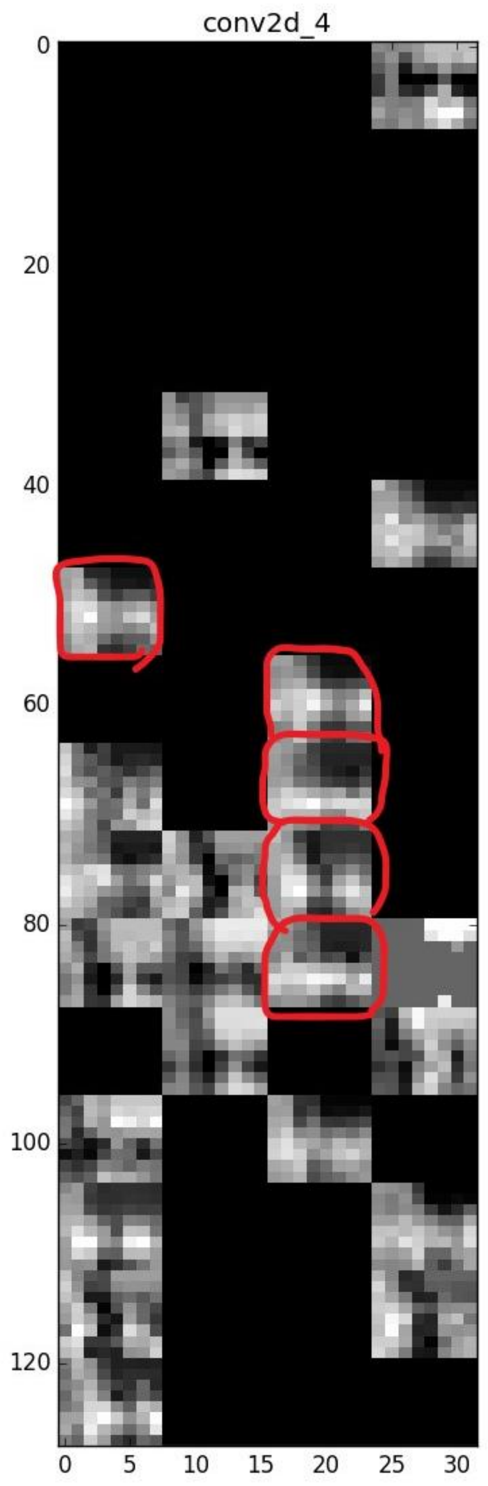
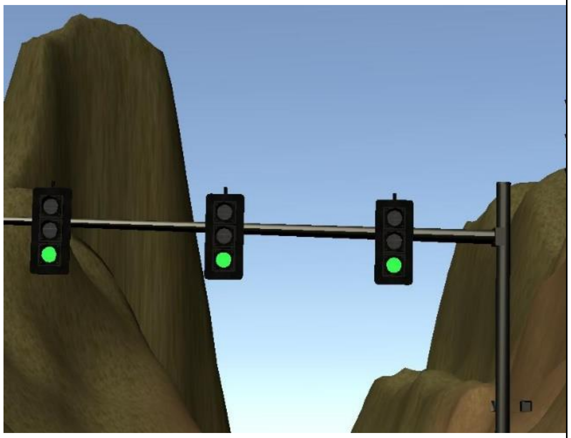


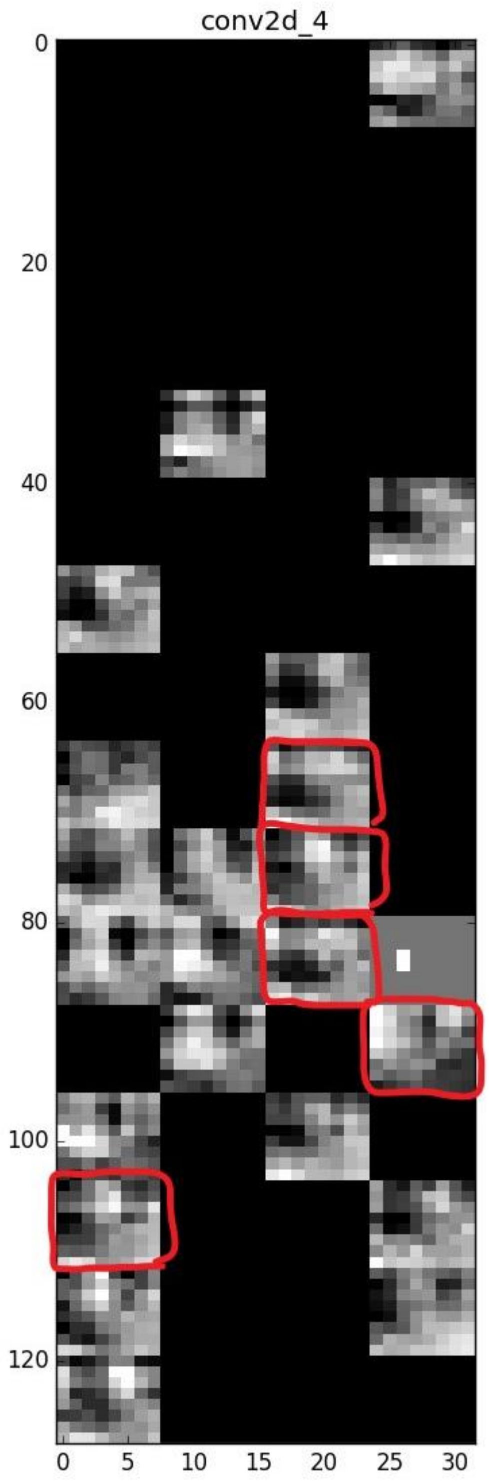
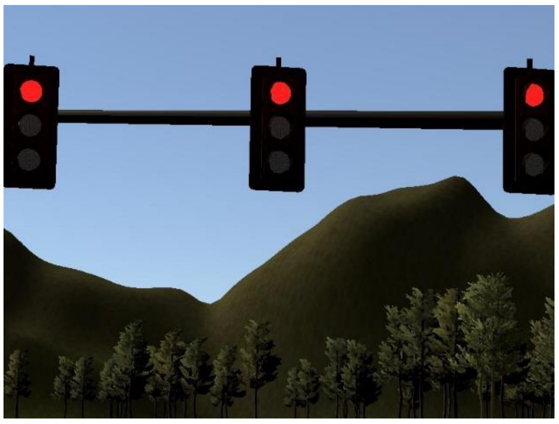
Figure 1 plot on neural network's training and testing

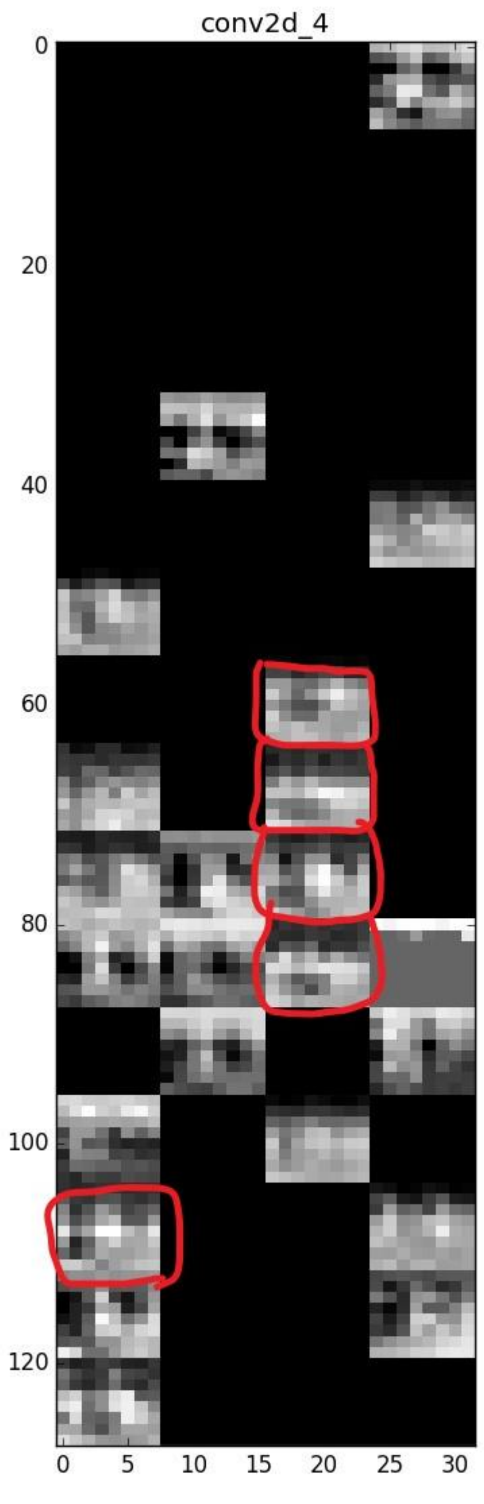
To begin with, I tried to go for a transfer learning and used VGG16 model for this classification task, though it took about 30 minutes to train per epoch so it was computationally quite expensive. Instead, I used the architecture above.

When building the neural network, I visualized all the channels in each layer to investigate what feature each channel captures. As the neural network increases its layers, channels in the layers take more details while decreasing the width and height size of each tensor in each channel. The 4th convolution layer has 64 channels, and I observed some pixels in some channels(marked with red) that are activated, which may indicate that the neural network took them as the corresponding color of the signal. (Figure below)

I'm going to display some images of the visualization; 64 channel in 4 x 16 grids on the left and the corresponding input image on the right.

conv2d_4 0 20 40 60 80 100 120 0 5 10 15 20 25 30	

conv2d_4 0 20 40 60 80 100 120 0 5 10 15 20 25 30	



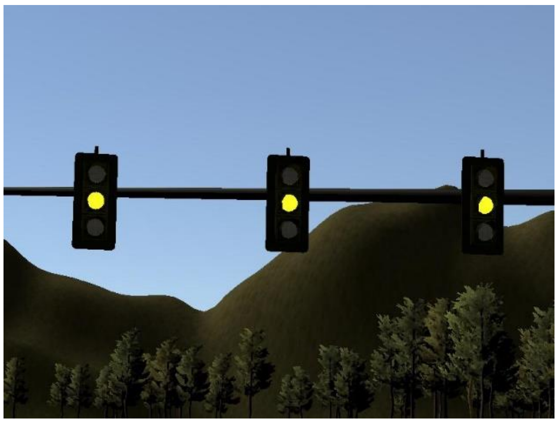


Figure 2 visualization of each channel in the 4th convolutional layer on each color random image

Meanwhile, it may be too naïve to estimate that these pixels in specific channels are directly corresponding to the targeted color and object and hence leading the network to a successful detection, as they're only recognized subjectively by myself. I could argue that 3 channels in common across the images above – 3 channels in the same column on top of each other – are maybe the corresponding channels for this model to recognize the signals.

It can be said that, for example, inhibited biases of data contain more dominant effect for the network on deciding which color the traffic signal is, not the color and shape features that the images share across the data. Taking samples over more images, 1000 images for example, and focusing on the same layer and comparing each channel across the images, may visualize more compelling activated neurons in common such that it'd be more reassuring and certain about the relationship between activation of certain channels and successful image recognition.

Ideally speaking, these channels should have more width and heights such that the visualization would be clearer, making it easy to tell which pixel(neuron) is activated when the network is given some input image. However, the subjectivity on this issue still remains. On the other hand, this increases the size of the network, which requires more computation resources.

As for this project, due to the latency issue that I'll describe after this session, as well as the small number of traffic sign labels, I concluded that further visualization or effort of sort to investigate the relationship wouldn't directly contribute to this submission, and the accuracy and practical performance of the model should be prioritized. I used this architecture to test whether the model can detect the traffic signs correctly.

As a result, the validation rate scored at 98.5 % and I used this as a classifier. I saved the model to a path and loaded it on initialization.

Testing the model

With the model being built, the car in the simulator was successfully able to classify the signals. Also while driving, classifying the signal alone was successful. Please refer to the videos below for the visual reference. Note that I'm not activating autonomous mode due to the latency issue that I'll describe in the following session(due to this, there's also latency on this recognition of images alone) . Instead, I only activated camera when the car is stationary and checked if the model recognizes the corresponding color properly.

- /capstone_NN@Start
- /capstone_NN@Stop
- /capstone_NN@DifferentPostions

Latency issue when the camera button is on

The machine learning approach makes more sense than just image processing with certain pixels considering a real world driving situation. Although in this project, there was a significant latency issue; not only is the data guest OS receives from the simulator (host OS) is heavy; but also, the workload of the simulator is heavy.

This latency causes the vehicle to be in uncontrol of its timings for steering, throttle and brake commands. In fact, when I lowered the frequency of the nodes between DBW_node and the simulator(through /vehicle/brake_cmd, /vehicle/steering_cmd, vehicle/throttle_cmd topics) from 50 to 0.1 Hz, I confirmed the similar issue arose. This indicates that slowing down the communication speed worsens the control of vehicle.

Moreover, the communication through /image_color topic happens between master server and traffic light detection node. Even if I disengaged either of the communication sources, the latency issue occurs as soon as I turn on the camera button in the simulator and the vehicle behavior worsens, resulting in uncontrol itself.

For this reason, the problem isn't maybe within the code, but would be the lack of the specification of the host OS machine. **In short, when the camera button is activated this latency issue arises.** This is an inevitable constrain for me and I will not be able to solve this issue unless I have access to a better hardware, if I need to run this vehicle with the camera on in the simulator.

Detecting traffic lights by getting state from the traffic light

Other workaround to autonomously navigate the vehicle in this project is to get the state of the signals from /trafficlight topic. This dose not involve in activating the camera, therefore there's no latency issue here.

As a conclusion, I submit my result, provided that my code will be run without turning on the camera button in the simulator by the reviewer. With my current machine, my code can successfully navigate the vehicle around the whole track with the camera button being off on its way.