Udacity Capstone Project

Team SDC Fun

Traffic light detection

Traffic light detection required several steps to accomplish, the preliminary parts being visible in the `/data_science/` directory, the latter parts in the main project. The first aspect that was addressed was collecting training data. The full images were classified manually after loading the `ROSBAG` and were output as `h5` for simpler later processing. The traffic lights were extracted from this collection of frames by color selection and bounded using k-means clustering. This resulted in labels for training the network as well as identification of the specific regions where the lights exist in each image.

The available data was split into training, test and validation sets. A pre-trained VGG16 model was used as a starting point with the classifier being re-trained for our specific problem. The full images were used for training, with most images in the training set containing no light, followed by examples of red and green lights, and the fewest training examples being yellow lights. The training data was a combination of simulator images and the provided Udacity track images so that we would have the sense that the performance we saw in the simulator would generalize to the real track since the training data contained both sets. The total number of training features used was 2703. Augmentation was used and included the following transformations:

- Rotation
- Width shift
- Height shift
- Shear
- Zoom
- Horizontal image flip

The new classifier applied to VGG16 is a fully connected network with 256 neurons connected to 4 output neurons with 50% dropout and softmax output. This was trained

using the RMSprop optimizer with categorical cross-entropy loss for 30 epochs using a batch size of 32. The training and validation accuracies and losses were evaluated to ensure the model was improving but not overfitting. Ultimately, the accuracy was about 99% on the validation set. The model was fine tuned from this point by allowing the last three convolutional layers to be trained. This improved the validation accuracy to about 99.5%. The model then evaluated random image samples visually ensure it performed as expected.

The traffic light detector employs this model, classifying images that are exposed by the /vehicle/traffic_lights topic. First, the image is classified and the predicted state is determined by taking the argmax of the four resulting classification vectors. Inference speed is about 15ms per frame using a GTX 1060. If there is evidence of a traffic light in the frame, the nearest traffic light waypoint in front of the car is used as the waypoint. Both the traffic light state and the waypoint of the traffic light are then passed to the waypoint updater to apply logic to control the acceleration and braking of the car, though only if the traffic light detected has been experienced STATE_COUNT_THRESHOLD times in a row to prevents false positives.

Motion control

Motion control depends on knowledge of the waypoints. In this case, the waypoints are sufficiently close together to not require interpolation. We first changed waypoint_loader.py to publish a single, latched read of the waypoints, as it had initially been set to publish remarkably frequently and wasted a lot of valuable resources. This was done with latching a single publish() because the use of the topic is essentially to avail a static data set to various objects that use the data as a property. Doing this simply improved performance, though this was fixed later in the Udacity repo.

The waypoint_updater.py code is critical to the project as it performs two main tasks. First, it takes the current pose and other runtime properties, as well as the static waypoints, and produces a Lane topic that consists of a subset of the waypoints starting with the first waypoint in front of the car, with a twist linear velocity set in the axis of travel. Second, the traffic light control is performed, which is the topic of the next section.

We first identify which waypoint index is nearest in front of the car, as well as ensuring the heading of the waypoint while determining the nearest. Next, depending on the control from the traffic light, we generate a velocity profile for waypoints in front of the

car. This profile calculates velocities for the forward waypoints based on an acceleration and deceleration models. This provides smooth, controlled acceleration and braking. The calculated list is integrated into a Lane topic and published.

The waypoint follower implements what is known as the pure pursuit algorithm, which returns a TwistStamped topic that is the computed best move for the car based on the algorithm. dbw_node.py subscribes to the TwistStamped topic produced by the waypoint follower. Here we seek to control the throttle, brake and steering of the car using the desired linear and twist velocities and the current velocity. The main loop in dbw_node.py simply creates a convenient data structure that is passed to an instance of our Controller class to achieve the desired control.

With respect to the throttle and brake, we implemented a single PID controller that has a split output for the throttle and brake. This conditionally allows scaling for the throttle in the [0,1] range while allowing scaling to maximum torque as defined by the car properties for the braking. The PID controller was tuned manually. The scaling function for both acceleration and braking is a variant of soft-step. The reasons for this are smooth transition , bounding due to the asymptotic nature of the function, and the ability to easily change the degree of control that will saturate the output. To avoid excessive toggling between braking and accelerating, we implemented three control zones: one for acceleration, a dead zone where acceleration is set to zero, and finally the braking. To eliminate brake chatter when coming to a stop at a light, a low pass filter was implemented with the braking, and a minimum braking torque was set so that the resulting braking action was smoothly varying and longer acting than the direct PID output. To avoid excessive throttling, particularly from stops, we limited the rate that the throttle value could increase during each update period. Control is responsive to the dbw_enable topic and resets the integral accumulator of the PID controller when this is toggled.

For steering control we simply used the provided yaw controller. We tried a variety of techniques mentioned in more detail later in this document, but found some of the best general results arise from the plain yaw controller. There is a slight oscillation at low speeds using this technique, but they are well within the bounds of the problem in terms of acceleration and jerk.

How we managed work in our team

One important aspect of working on a team to complete this project was defining a good schedule with milestones for completion. We wanted to have an MVP ready to turn in by September 18, two full weeks prior to the project deadline, so that if there were any problems, unmet requirements or new ideas there would be plenty of time to accommodate so that we would be able to have our code run on the actual car. When we initially formed the team we decided this accelerated schedule was desirable since we all wanted to achieve this shared goal.

In terms of work division, this project has two main components, training a model to identify the waypoint associated with the nearest red light in the direction of travel of the car, and waypoint management and motion control. These tasks can be divided into a sequence of two asynchronous tasks followed by integration and testing.

The members of our team selected what aspect to work on initially based on personal preference, knowing that as the project evolved we would all be involved in a wider view of it. James and Nick predominantly worked on the model for the traffic light detector while Colin and Nimish predominantly worked on the motion control. By the weekend of September 9 the whole team started working on integration to meet our MVP milestone.

We did have a functioning MVP by September 18, but as a team we did not think that it was good enough. We started reviewing all of the components of the car, questioning aspects that we had previously constructed, and tried a variety of new approaches to obtain better performance. In the end this paid off and we were able to demonstrate better performance. However, time was running out and we had to prioritize what aspects we needed to work on more than earlier in the project.

Our team predominately used Slack to facilitate communications, augmented by e-mail correspondence and a few video conference calls used to meet each other and check in on progress during the integration phase. We opted to use a single GitHub repo with privileges granted all team members. Over the course of the project we created new branches for new features. As branches became outdated we deleted them to reduce repository clutter, focusing mainly on the branch we would turn in.

Lessons learned and areas for improvement

There are a number of lessons that were learned, both related to the technical aspects of the project as well as the way that we worked together to accomplish the goal.

It turned out to be very useful to be involved in a team early. This allowed for accelerated milestones and helped keep the project on track. Early on we divided the work between the deep learning aspect and the motion control aspect, with the expectation that as these were completed we would all work across the entire project. As we began, those working on deep learning became familiar with the ROSBAG and other aspects of the project germane to training our model. At the same time, those working on the motion control were exposed to other ROS features and the general control system for the simulator and car.

There was a lot of discussion early on about the model and how to generate the training data. As well, there was a lot of discussion about how to use the various motion control aspects that were provided. Since we were one of the first groups to have solid progress, at least that we could tell, we had to make a number of decisions by simply trying many potential solutions. As it turned out, the two most troubling issues ended up being how to control the steering and how to generate and control the signal for braking and accelerating the car.

Two ideas were put forward regarding the steering. One was using the stock yaw controller, which works very well except in the case of slowing down quickly, as the difference between the desired and current velocity affects the steering. The other solution was to rework the yaw controller so that it was more compatible with a PID controller, namely by having two yaw controllers, one for the desired yaw and one for the current yaw, and then applying a PID controller to this error term. This was more complex, deviated from the supplied code, and required using low pass filters for smoothing the resulting steering. This also had negative properties, as the low pass filter made performance in the tighter turns not as ideal. We put forth models having both of these so that as a team we could look at the relative performance and consider what would be the best solution for the specific problem we are working on.

Regarding the control of the motion from the traffic light detection, there were three main ideas. One was creating a minimum jerk velocity profile for future waypoints and letting the PID controller attempt to track this. A second was to simply apply constant braking with magnitude determined by the stopping distance and the characteristics of the car (mass, velocity, wheel radius), and brake at the appropriate time before the light. A third mechanism was presented where the PID controller was still in the loop, and

utilized a soft-step mapping with asymptotic behavior so that effectively it would brake hard when stopping and have a deterministic effect based on desired stopping distance, but for low speed behave gently. In the end we used some of the best features from these differing initial approaches.

Much of this competitive solution optimization was performed after we had an MVP on the date that we expected to have the MVP. As a team we were simply not satisfied with the results and opted to work on opportunities for improvement that were consistent with the vision we had for making the improvements. Since everyone had their own ideas on this, we developed the ideas we liked best and then started evaluating what worked best.

There are plenty of opportunities for improvement, such as:

- More training data
- More types of augmentation
- Interpolation of waypoints
- Better tuning of PID controllers
- Better methods of steering control
- Better methods of motion planning
- Better estimation of traffic light for images

While there are a lot of opportunities for improvement, we are also happy about all of the parts of the project that we solved within the time allowed for completion and within the scope of the objectives that needed to be met for the solution. While some of the aspects of the project were actually simpler than prior assignments, due to the larger, integrated scope of this project, as well as the requirement of working in a team, we were about to gain a better sense of the effort required to solve real world problems in the autonomous vehicle space.

Thank you so much, Udacity, for presenting us with the various problems over the last year, and for helping us become more familiar with the practical challenges that go into the development of self-driving vehicles. It has been awesome.