The goal of this week’s sprint was to research and explore reinforcement learning algorithms and approaches as outlined in Sutton and Barto’s Reinforcement Learning book. This book covered many algorithms generally separately, for the sake of the reader’s intuition, and then would (often) later combine and generalize them. From this book I’ve come to understand reinforcement learning in much greater detail, and this background knowledge will allows us to make better decisions in which algorithms we choose to use and implement.

For this current sprint, we implemented a simple tabular Q-learning method. The policy is greedy and one-step, so it chooses the action that has the best value for the given state. Learning is done by bootstrapping the value function: we update the value of the chosen action based on the reward and the discounted value estimation of the following state. For this sprint’s model, the learning was done both on our manually collected data, and some exploration of an epsilon-greedy policy—a policy which with some probability epsilon does not act optimally and instead chooses an action randomly (leading to exploration of states that the model does not think are good but may have untapped potential). I think, given more training time, this model would perform relatively well but it has some problems that could be corrected to give us a more robust algorithm.

The first problem is the small number of states. This was necessary to allow a tabular method to work, but it means that many sequential frames will be classified as the same state in spite of the follower’s action choice. This is also a symptom of the high frame rate of the follower camera, but if we seek a model with good performance this seems relatively nonnegotiable. Regardless, if we consider the environment as a finite state machine, the follower would see a “loop” transition occurring almost all of the time, making it even harder for it to discern the outcomes of its actions—and it would make training much slower! Thus it would be preferable to have a much more granular representation of states, and such a desire leads us to a value function using function approximation rather than a table lookup method. This is what we seek to incorporate in the following sprint.

Training the approximation function for value occurs much the same way as supervised learning. As in a supervised case, we have a loss function which determines how much our output value differs from our desired output value. The desired value is determined by the reinforcement learning algorithms and principles: this is the same as the tabular case. However, there is an additional issue that, unlike in the tabular case where policy improvement is improved with every update step, updating a function which evaluates all states (or action-states) will necessarily affect all other value approximations. Thus, it becomes necessary to value the losses from different states differently—in particular the loss from states we see commonly is more important than the loss incurred in states that we only very rarely see. This leads to additional problems in the off-policy case, which is the case we would prefer, since collecting data (learning in the on-policy case) is more time expensive than iterating through collecting data. This is a problem that will have to be handled differently depending on which algorithm we ultimately choose.

The most natural extension of our current approach would be to use the TD(λ) method with eligibility traces, which is a generalization of the bootstrapping TD(0) which only updates values based on one step, and the Monte Carlo method which updates all values based on a final end state cumulative reward (λ = 1). This carries a clever optimization over n-step TD, namely that it uses a backwards-looking approach that allows for updating values on each step. The main drawbacks are twofold. Firstly, this method does not easily translate to an off-policy method. Probably the best mitigation for this is to consider learned weights from saved data as good initial weight values, as in transfer learning, but allow each unique model to explore and exploit in the real world to refine those weights. Thus, hopefully, due to the good initial weights the model will converge to good behavior quickly enough to be practical. A second problem is due to the simple greedy policy. This policy is of course easy to implement and theoretically viable, but it does require a policy to be epsilon greedy or epsilon soft in order to explore. We could decay epsilon as the car gets more data, in order to better approximate our “optimal” policy. Then however our exploration decays, so we would need to somehow “alert” our car if it goes to a new environment, in order to tell it to begin being more exploratory again. A greedy policy can also lead to sharp changes in policy based off of very slightly different values, a “problem” (this is probably not desirable behavior) which would not occur with policy gradient methods.

The main alternative to this approach is to use a Policy Gradient method, the main advantage of which is that policy is not simply a greedy function but is learned, with some function like softmax in order to determine the probabilities of picking certain actions. A further advantage of this method is that it allows selection of continuous action values, for instance we could represent our outputs as “throttle” and “direction” instead of as some cross product of sets of action and directional enumerated values. Whether or not this would actually lead to better policy for our use case, however, is not intuitively obvious.

With these methods outlined, it seems the most logical step is to first try TD(λ) since it is more closely associated with our current implementation. Whether or not we go on to implement a policy gradient method would depend on the performance of the aforementioned model.

A final big question is what features can and should we extract from our input image. Unfortunately, since the performance of a neural network on raw data seems infeasible, it is left to us to decide how to extract and select features for use in the (linear) value function approximation. Natural ones to choose are an approximation of distance, angle, some combination of the two. Additionally, it seems important that we keep some amount of past state in the current state, in particular for knowing which way to turn when the leader car turns sharply causing us to lose line of sight or for when motion blur renders the leader car temporarily unrecognizable. With tabular methods this was less feasible, because keeping old state information around causes the number of states to blow up very quickly, but with function approximation methods we should have more liberty in deciding what information we want to keep around. As for other features, it isn’t exactly clear what they should be, but perhaps further experimentation will bring some to mind.