Teach a smartcab to drive

Project 4 for the Udacity Machine Learning Nanodegree

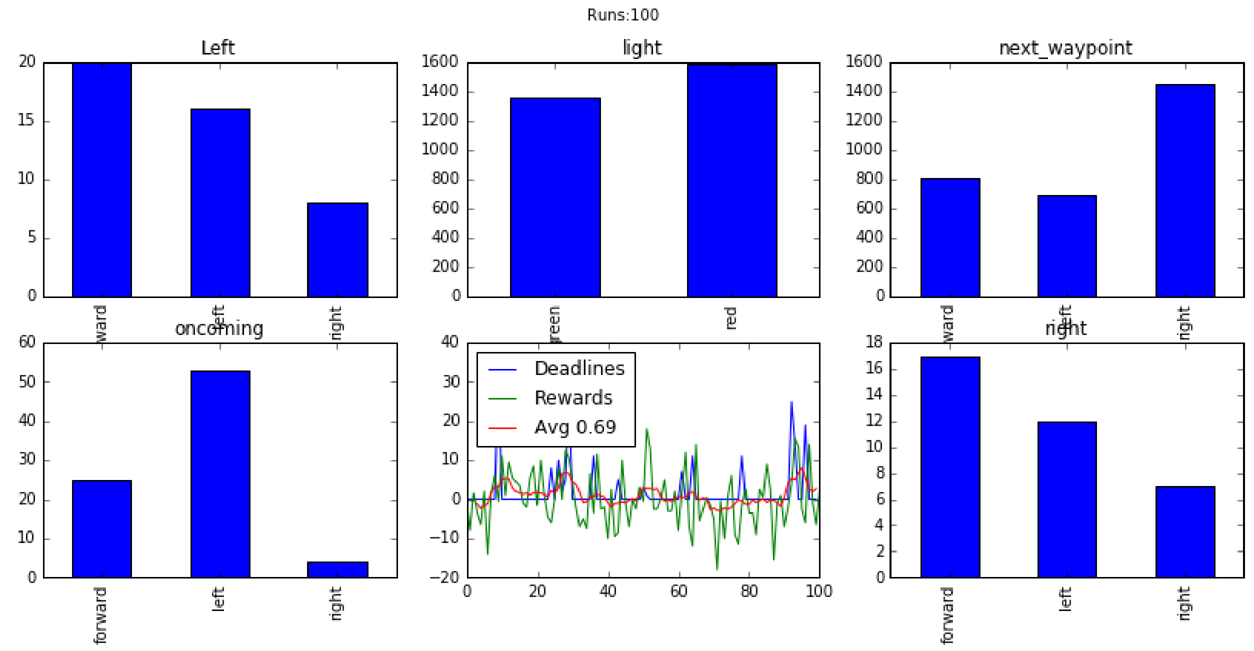
In this project, we will use Q-learning to train a smartcab to follow traffic rules and reach it's destination in a timely manner.

All code was developed in the accompanying Jupiter notebook, and exported directly to the agent.py, located in the smartcab directory. The project report may be viewed with or without code included inline, by choosing the appropriate pdf, or directly from the ipynb file as a notebook.

Implement a basic driving agent

When we run an agent with a random action policy, we see that it will move about the board with no pattern, and will eventually reach the destination, in most cases. If we allow the use of deadlines, we see that the agent rarely reaches the destination in time, although it may still occur.

Identify and update state



When we sense our environment, we perceive 5 variables, with several possible states These include: left, light, next_waypoint, oncoming, and right. We can see right away that light and next_waypoint contains new information at every poll, while the others usually have no value.

It's not readily apparent that the direction of travel information of the other cars (described by left/right/oncoming) is relevant to our agent. A case could be made to remove the direction information, and only retain information about another car being present at the light. This would have the benefit of reducing the number of possible states, increasing the learning speed of the agent. This may be a valuable approach in resource constrained environments.

The downside is that the agent may pick an action that causes a longer trip. Early in the learning phase, it could also pick an action incorrectly. For instance, by proceeding through a light when the opposite car is turning left. In this case, it may have previously seen a positive reward for moving through the light, because the opposite car was not turning. This time through, it will receive a negative reward, and in the future when a car is at the oncoming light, it will always wait till the intersection is clear.

In the interest of correct action, we will choose to use the state as returned from the sensor, with the addition of the next_waypoint.

While I have tracked the deadline, it is not apparent that it will provide useful information to the agent. It is useful to note that the agent does not see any change in the deadline value yet. We may expect this to adapt as we implement learning.

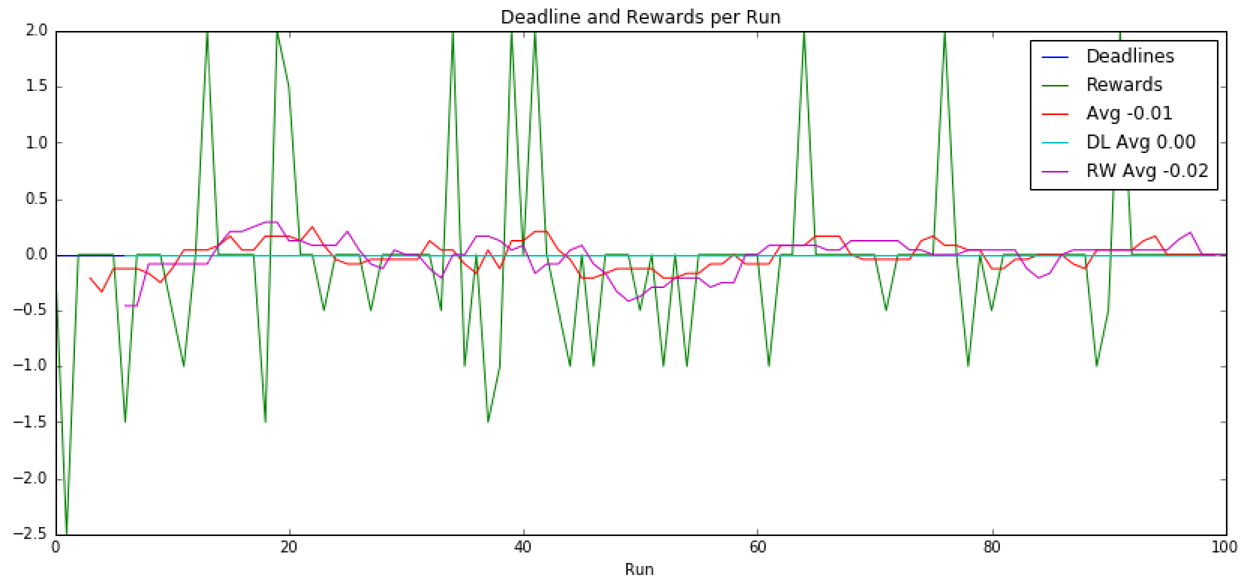
Implement Q-Learning

With a basic Qlearning algorithm, we note that the agent quickly learns a set of rules that allow the agent to move toward the objective. Generally speaking the agent, is moving to the destination, but does not always make optimal choices. We see the agent begin to obey some traffic rules and make moves toward its destination. We can see that now the reward in each run is generally positive.

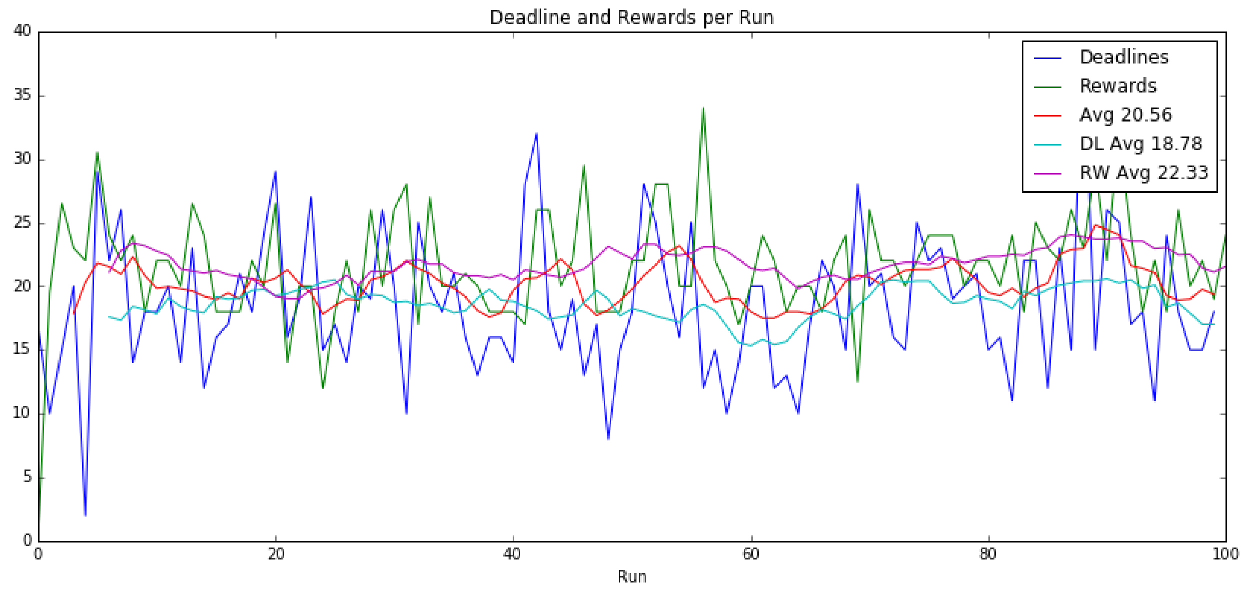
We don't see a large continued increase in the agent speed toward the destination after the initial runs. We still see strange behavior such as repeated right turns back to the original destination, or staying in a no action state for extended periods, without much change in pattern through the run. With Epsilon set to 0, we don't see the agent select new behaviors, and with Gamma at 0 we don't consider our future state.

One interesting note is the difference made by re-ordering the available action list. Placing the 'None' value at the end plays to the bias of the 'max' function, in which if all values are the same, will select the first match. Since we initialize the list to zero, if 'None' is first in the list, the agent tends to take no action, and never reaches the goal. By reordering the list, we bias the agent toward action, allowing the agent to perform significantly better.

Agent w/ ‘none’ bias:



Agent w/ ‘forward’ bias



Enhance the driving agent

We immediately see the agent begin learning when we begin using epsilon to explore new states. The addition of gamma provides many of the same benefits, and we see that the agent learns to reach the destination as quickly as the first or second run. Following this, the agent will quickly begin to reach it's destination well before the deadline, with a positive score, the majority of times. This is despite biasing the action list to 'None', as in the previous example. Even at the end of the run, we do still see odd behaviors, as the agent tries new methods according to epsilon, or encounters new states.

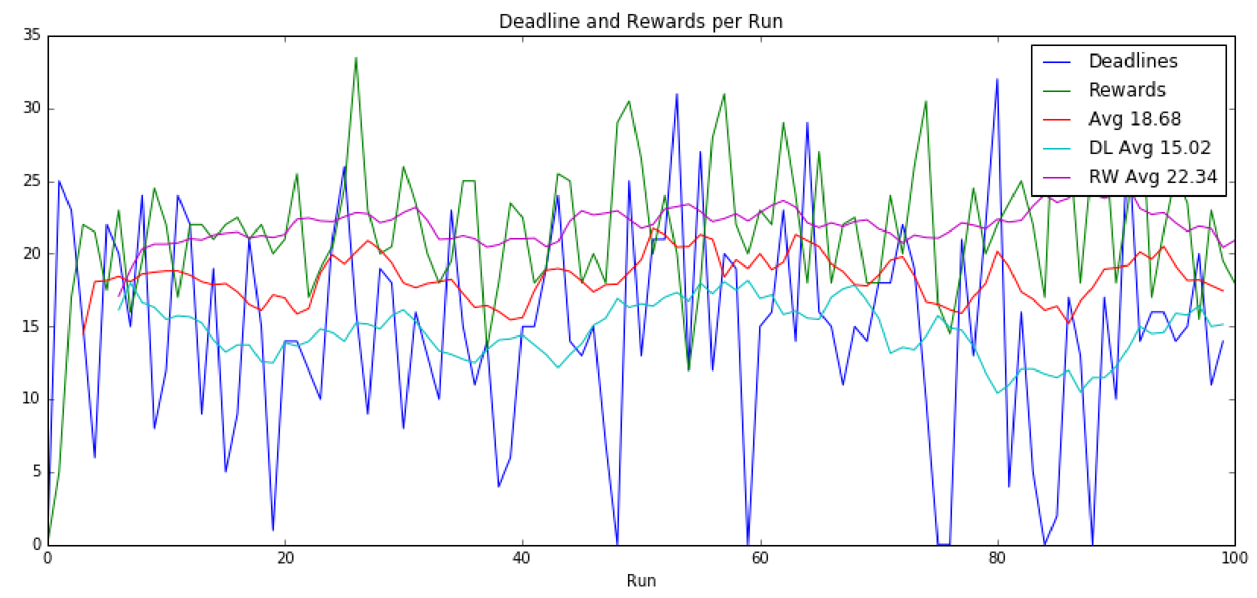
In addition to tuning the final epsilon and gamma, I have added the ability for each to adjust the amount they affect the outcome over time. Initially we want to prefer a random action, as our table is now initialized with random values, and we want to explore the available states, and record their effects. The opposite is true for gamma. In it's case, we would like to highly discount any initial knowledge initially, and over time grow to trust what we know. My implementation allows us to control the rate of change over time of each. I have selected starting/ending values and rates based on experimentation, and optimized these numbers further by implementing a grid search to test the values over multiple runs, etc. These were found to be:

alpha = 0.374625 epsilon = 0.125 gamma= 0.125 success_counts_avg = 91.92

We could also implement methods to adjust these rates based on factors other than time, for example based on the difference between the current and new values in the Q table for the state, but my initial attempts haven't been successful in finding a model that learns as fast as the current implementations.

I have also changed the Q-table initialization to use random values from -1 to 1. This is intended to assist early phase learning of each state taking random actions, increasing state exploration.

It should be noted that the change that most affects success from the initial algorithm was to re-order the selection of available actions, such that forward is preferred over inaction. This led to an agent that was successful in reaching it's goal at least as often as any other enhancement, usually reaching the goal more than 96% of the time. This may be a quirk of the environment, and not something that is true generally, especially when the learning is moved to environment's that may have a larger state space, or with a more complex reward system. We do see that the final model has a higher average reward per run.



Conclusion

We can see that the fully implemented agent has learned a set of rules governing road travel, including stop light behavior, and how to follow the route planner. Our agent quickly approaches an optimal policy state, with the caveat that it retains the ability(via epsilon settings) to learn changes. Since we have concentrated on 100 run trials,we will not see all possible states, and so can't reach a fully optimized policy, wherein the agent would follow all rules perfectly and always take the shortest path that those rules allow. We currently see behaviors, such as circling around a block, that maximize the total reward, but don't serve any other purpose. The possibility exists to save the policy and continue to learn, eventually having an entry for each possible state, and minimizing the mistakes. Until then, our agent is able to perform in a very acceptable manner for this environment.