P4 Train a smartcab report

May 16, 2016

1 P4 : Train a Smartcab to Drive

1.1 Implement a basic driving agent

If the code provided was run without modification, a simulation ensues with the agent ('red') that does not move at any step, regardless of the state of the traffic, traffic light, or way point direction. By defining 'action' as the random.choice of the 4 viable actions, the agent will attempt to move forward, left, right, or not at all in each step. However, due to the rules of the simulator, when an action that is illegal (e.g. running a red light) no movement occurs, though a negative reward (penalty) is recorded. This mode of policy rarely navigates the agent to the destination.

If one were to set the action to copy the way point provided by the planner.py program, then the agent will progress as directly as possible to the destination. However, in this case the agent fails to heed any traffic laws or the presence of other agent cars at the intersections, frequently incurring penalties.

It is clear that a policy for optimal driving will need to incorporate the state of the agent at each step, along with the way point, in order to both reach the destination, obey traffic laws, and avoid collisions with other agents.

1.2 Identify and update state

Within the simulation, the possible variables that the agent can sense are the next way_point ('forward', 'left', or 'right') which serve as a heading towards the destination, the traffic light signal ('red' or 'green'), or the heading of any traffic that is oncoming, approaching from the left, or approaching from the right (if no traffic is approaching from a direction, the variable is listed as None).

For the purposes of my learning agent, I chose to monitor a state composed of four variables:

```
next way_point ("Directions");

the traffic light ("light");

traffic that is oncoming ("oncoming");

traffic that is approaching from the left ("left")
```

I did not choose to monitor traffic from the right, as theoretically this information is redundant with the traffic light color. Assuming an oncoming agent obeys traffic laws and does not run a red light, there is no situation that could be altered by the presence of a car from the right. For instance, if my agent's light is green, the approaching agent will have a red light and must give right of way if turning right. If my agent's light is red, then the only viable MOVEMENT is a right turn which would be unaffected by a car approaching from the right. Thus inclusion of this variable in the state would not provide any value, but would exponentially increase the number of learning steps needed for Q-learning convergence.

When implemented with 'action' set to random.choice as described above, the current state is measured and displayed in the simulation header during each step, though the sensed state has no link to the agent's choice of action. The link between policy and exploration was created in the next step, via implementation of the Q-learning algorithm.

1.3 Implement Q-Learning

Q-learning is the iterative process of using observed rewards from state-action pairs to learn a policy of best actions for a state space. Typically, an initial Qˆ is chosen, and then updated as a state space is explored by a learning agent, via the Q-learning equation:

Q(s,a) := Q(s,a) + alpha * [r(s,a) + gamma * (argmax(Q(s',a') - Q(s,a)]

In its most basic form, the Qˆ (current Q-value as it converges towards the true Q-value) is updated based on the reward gained from taking a nominated action from a specific state. Several parameters are included in this process, which include the learning rate (alpha), the Q-values from potential future actions (argmax(Q(s',a'))), and a discount factor (gamma) that transcribes the level of importance the agent should attribute to future rewards versus current rewards.

Within the the framework of the curent agent.py program, incorporating the Q-learning algorithm required a rearrangement of program structure, due to the fact that certain aspects of the updated state would always be random (e.g. traffic light color), and thus could never be known in real time. To address this, the state, action, reward for each step were briefly stored for a cycle, and utilized in the subsequent step when the next state had been randomly generated.

From there, I needed to set the various parameters for the Q-learning iteration. As the initial Qˆ value for every state-action pair, I chose to set equal to 0. I chose zero because it transected negative penalties for traffic infractions and the positive rewards for following traffic rules and not crashing. The next parameter needed for Q-learning was the learning rate, alpha. For deterministic environments, such as this, an alpha set to 1 is most appropriate. Without fear that the 'best' action would incur a penalty due to random chance, there is no need to taper the learning rate. However, in environments where the rewards for state-actions are probabalistic, then alpha may need to be tuned.

For gamma, I initially chose a value of 0. This selection coerces the learning algorithm to ignore future rewards and update based solely on the current state-action reward. There are several aspects of this simulation that warrant a total discount of future rewards. The first is that subsequent states after implementing an action are either entirely random or unable to be forseen. The traffic lights, presence of oncoming traffic, can never be known prior to the action being taken. So while my program could indeed generate the discounted reward from the 'best' future action, there is no real way to know what the particular state would be in the next action, thus making this term irrelevant. Additionally, because the agent is egocentric (e.g. no global positioning system or knowledge where exactly the destination resides), there is no way to reward 'getting close' to the destination over simply following the next way point. Finally, intuitively there is no state where taking a negative reward would increase the overall Q-value for the entire journey. This would change if certain streets were closed or if traffic lights were on a fixed pattern, etc, but again this would require an allocentric view from the agent where such global knowledge was accessible. In this simulation as it stands, the only senses the agent experiences are immediate.

To program this process, the first step is to initialize and maintain a master Q value table for policy reference. Instead of generating all possible states and actions a priori, the agent simply checks the master Qtable for the presence of the current state. If it does not exist in the table, it is added and all possible actions given an initial Qˆ value of 0 as discussed above. If the state has been encountered before, then the program will choose the action with the maximum Qˆ for the state. In situations where multiple actions have the same Q value which is the maximum, then one of the 'tied' states is chosen randomly.

This version of the program did run without error, however the learning agent frequently got stuck in a policy that was clearly not optimal. Compared to breaking traffic rules or going the wrong direction, the action of not moving (i.e. selecting "None" for action) is preferable. Thus, in the first instance where a state was encountered, if the first random action explored was to remain stationary, a local Qˆ optimum would be created. In subsequent occurences of the state, the agent would interpret the 'best' action as remaining still. While this policy avoids traffic violations but also avoids exploring the state action space for the most rewarding potential action (which obviously would be to follow the way_point if the traffic rules and presence of other agents allowed).

Nevertheless, this initial implementation of Q-learning did train the agent to follow traffic rules as negative rewards were quickly relegated to penalized actions.

1.4 Enhance the Driving Agent

One method for controlling the trade-off between exploration and exploitation of knowledge gained is to add an epsilon clause within the program. The epsilon clause is a way to introduce exploration of the state-action space, even if a 'best' action does exist for the specific state. In brief, a random float is generated, and compared to epsilon value. If the float was less than epsilon, the 'best' action would be taken according to the policy. If the random float eclipsed epsilon, then a random action would be taken. This feature allows the agent to explore more of the state and avoid being trapped in a sub-optimal policy that had found a local minimum Qˆ. I initially set epsilon to 0.95, meaning that 1/20 actions occuring from a known state would be a random decision. The reward observed for this random action would always break the local minimum problem, given enough occurences. While this strategy did help the learning agent converge to the optimal policy quickly, it also maintained a randomness that was not necessary during the end stages of learning. The epsilon rate of 1/20 meant that in just about every journey, one incorrect random step would be taken. Therefore I edited epsilon to serve as a function of the number of training steps. Using a logistic function that was limited at 0.75 to 0.99, the epsilon value increases gradually from the early stages of learning into the later stages when the optimal policy has already been found. To note, when the logistic function was implemented for alpha, I reverted the Qˆ intialization back to zero. This made encountering a new state at any time more equivalent.

I implemented a similar strategy for alpha (learning rate) such that the overall learning rate decreases gradually during the simulation. This is not really important for this particular simulation, but if the rules were changed to something probabalistic, this slight modification should help performance of the learning agent.

Finally, I adjusted the gamma (discount) value for the Q-learning equation. Because the agent is egocentric (in that it does not have a global knowledge of where itself, the destination, or other agents currently reside), it does not make theoretical sense to increase the gamma value. The agent learned at similar rates whether the gamma was 0 or 0.1, but performed worse when gamma was increased to levels approaching 1.

Another way to avoid this sub-optimal policy generation would have been to change the inital Qˆ to a large positive integer value. By setting the Qˆ 'optimistically', the learning agent will explore all actions that have not been tested before, as every action (even the 'best') will end up lowering the Q value from the initial optimistic Qˆ. After all actions from a state have been explored, the best action will be evident as the highest remaining Q value. Importantly, this type of strategy is usually only suitable for agents acting in a relatively small state-action space, though given infinite training steps, should converge to the optimal policy (assuming alpha is constant and not a function of training steps). To test this approach, I set Qˆ to 13, alpha to 1 (total replacement), epsilon to 1 (no random decisions), and gamma to 0. For this simple simulation, this 'optimistic' strategy found the optimal policy more efficiently than the parameter optimization strategy discussed above.

1.5 Discussion

Intuitively, the optimal policy for the agent would be as follows: 1) obey traffic laws, and then 2) move in the direction of the way_point provided. If traffic laws occlude the way_point direction, do not move.

The Learning Agent generated from the agent.py program accomplishes this policy through Q-Learning iteration. I used two strategies to increase the efficiency by which the agent learned the optimal policy. The first version initialized Qˆ to zero, and then employed monotonic functions for epsilon and alpha to control the rates at which the agent explored the the state-action space versus exploiting knowledge gained. The discount value for future reward was found to be optimal near the limit of 0. This is due to egocentric nature of the agent, as well as the random aspects of state that cannot be predicted ahead of time. Were the simulation changed to an allocentric view, where the agent could sense where it was in relation to the destination, etc, then gamma term would be a more important parameter for optimal performance and policy generation.

This program version generally took ˜20 trials to converge to an optimal policy. If one knew the simulation would incorporate new features at some point, I would choose to use this version of the program, as it is most adaptable to increased state-action space.

However, for the simulation as it stands, the strategy that appeared most efficient was to set the Qˆ initialization to an overly optimistic value that exceeded any possible single action reward. This change, concomitant with subsequent changes to make alpha constant (set to 1) and a full discount factor (gamma = 0) generated a highly efficient learning agent that almost always found the destination within 5-7 trials. Notably, this strategy was only successful because of the limited state-action space that existed in the simulation. Were I to have included more inputs for the state, or additional variables were added to the simulation, it is likely that this strategy would take exponentially longer to converge to optimal policy. Also helping the efficiency of this strategy is that certain states were much more likely to be encountered than others, e.g. no traffic leaving just traffic light and way point direction. If there were more traffic, then many more training examples would need to be observed by the agent before optimal actions were consistently evident.

In conclusion, I have used various strategies to build a self-learning smartcab that is able to obey traffic rules and efficiently reach a destination. It is reliant on the ability to sense the immediate intersection environment, and is also dependent on information provided by the way point planner. Given these variables an optimal policy for self-driving was generated using Q-learning iteration. The best performing strategy was to initialize Qˆ values at an unobtainable reward level, however it is probable that the addition of features or the incorporation of probabalistic outcomes would require a more fine tuned strategy similar to first strategy discussed here.

```
In [ ]:
```