Training a Smart Cab to Drive

Preface

The object of this project is to train a smartcab to drive. Notice this is train, not teach or tell the smartcab to drive, but train it. We want our agent, a smartcab to get from point A to point B in duration (elapsed time or turns) x. The smartcab is operating in a gridworld of streets, traffic lights, and other agents, where the other agents are other cars. The perspective of the cab is egocentric. That is, it can only observe what is going on in its immediate environment. So, how do we get the cab to go from point A to point B? There are three main options:

1. **Tell:** We could write a set of rules using flow control to force the cab to make certain decisions in every case. For example, this is what you do at a red light with opposing traffic, or without opposing traffic, etc. etc. The challenge with this approach is that it doesn't take very much to create an environment that is so complex it would be daunting to think of every possibility and tell the cab what to do. We would also have to hard code in new elements encountered into the environment.

2. **Teach:** Teaching would be akin to a supervised learning approach. We could have agents drive around the virtual grid world and experience things and then use the record of those experiences to teach a new agent what to do. This would be daunting for even fairly simple real-world applications and would not necessarily account for all new environmental elements

3. **Train:** Training is just what it sounds like. The agent drives around and experiences things. Some actions turn out great, some not so great. The agent stores these experiences and uses them to form a policy of behavior. That is, given a situation, what is the best action to take?

Approach and Resources

So how do we do this? Through reinforcement learning. Specifically we use Markov decision processes and unsupervised learning to train our agent. The approach I have chosen for this task is Q learning. There are many resources that explain Q learning, but these are the three I found most useful for building intuition as well as implementing Q learning in code.

1. The Study Wolf Blog blog contains a great overview of reinforcement learning and Q-learning and even a python code implementation This implementation was modified for use in this project see q_learn.py.

2. This great tutorial shows how Q-learning tables could be applied to teach an agent to move from a randomly selected room, out of a house.

3. Demystifying Deep Reinforcement Learning is another great resource with explanations of how Q learning, reinforcement learning, and Markov decision processes relate.

The resources listed above do a comprehensive job of explaining the principles and approach to reinforcement and Q learning but I paraphrase below, per my understanding.

Essentially we have an agent who is operating on a grid or network, where locations are nodes and paths are edges. In the case of a smartcab each intersection would be a node and each road an edge. The primary agent has a goal, in our case to go from point A to the destination. In this gridworld there are states. The states in our case are made up of the following elements [traffic signal, position of other agents relative to our agent, and position of our agent relative to the next waypoint]. Where traffic signal has the states [red, green] and a car can be to our left, right, or in front us. Our actions can be to stay put, turn left, turn right, or go forward. From the gridworld perspective our agent can choose to stay put or to advance in one of the four cardinal directions.

Each state, action combination (s,a) is associated with a value, which acts as a reward or penalty. So if our car is like pac man, just running around gobbling up rewards and trying not to hit other agents/ghosts, what is the motive force that moves us to the goal? The rewards can be thought of as the immediate gratification for taking a particular action in a particular state. To learn the agent must also consider the future impact of an action. This is where Q comes in. Q is a measure of the quality of an action taken in a state. It is defined by the Bellman equation

> Q(s,a)= r + (gamma * (max)a'Q(s',a'))

where s and a are the current state and action and s' and a' are the next state and the action taken in that state. (max selects for the maximum Q for all possible a' in s') gamma is a value between 0 and 1 that discounts the future reward. If gamma is 0 then we only have the immediate reward whereas if gamma = 1 we add in the maximum future reward. A value of 1 would only make sense in a deterministic environment (no random events), but since this is a stochastic environment values of .5 to .9 are typical. So the Q value for each state and action combination is affected by the maximum utility of the next state and the best possible action in that state, which in turn is affected by the all possible states and actions linked to that state, etc. etc. In other words, the value of Q is back propagated from the goal.

So, our agent starts out with a reward matrix R with state as the index (rows) and actions as columns. Each intersection of state and action has a reward. At the start of the learning process we have a Q matrix as well, which is initialized to 0. As the agent samples states and actions the Q matrix is filled in. Eventually this leaves us with a mapping of every state and action to a Q value. The states and actions that are more likely to lead us to the goal have a higher Q than those who are less likely to lead us to the goal. In some ways this reminds me of chemotaxis. That is the propensity to move towards a target by sensing the strength of a chemical signal. As the organism gets closer to the source of the chemical, the signal gets stronger. So, it follows a chemical gradient in a similar way as our agent follows a Q gradient.

The practical form of updating Q is:

> Q(s,a) += alpha * (actual reward - expected reward)

Where the actual reward is

> reward(s,a) + Q(s',a')

and the expected reward is:

> Q(s,a)

and alpha is the learning rate. Basically alpha and gamma are used to attenuate, gamma on the contribution of Q(s',a') to Q and alpha on the magnitude of the Q update and therefore the rate at which Q changes.

Expressing Q learning in Psuedo Code

Resource 2 above

expressed this as:

```
The Q-Learning algorithm goes as follows:

1. Set the gamma parameter, and environment rewards in matrix R.

2. Initialize matrix Q to zero.
```

```
3. For each episode:

Select a random initial state.

Do While the goal state hasn't been reached.

Select one among all possible actions for the current state.
Using this possible action, consider going to the next state.
Get maximum Q value for this next state based on all possible actions.
Compute: Q(state, action) = R(state, action) + Gamma * Max[Q(next state, all actions)]
Set the next state as the current state.
End Do

End For
```

Resource 3

expressed this as:

```
initialize Q[numstates,numactions] arbitrarily
observe initial state s
repeat
    select and carry out an action a
    observe reward r and new state s'
    Q[s,a] = Q[s,a] + α(r + γmaxa' Q[s',a'] - Q[s,a])
    s = s'
until terminated
```

Exploration

At some point we will have a lookup table where state action combinations are mapped to a Q value, so our agent simply needs to lookup Q to determine what action to take in any state. What if our Q is not the result of complete sampling or what if there are better options in the future? That is, how do we generalize our policy to prepare for future states and actions or states and actions we have yet to discover? We can add a wildcard term, epsilon, that is compared to a randomly generated number if the number is less than epsilon, explore, otherwise exploit (follow the Q gradient). There are two main methods of exploration that I am aware of:

1. In the explore mode choose a random action. This is typically coupled with a reduction of epsilon per step, so that the agent explores less as time goes on.
2. In the explore mode randomly add values to Q values for that state scaled by the maximum Q value for this state. In this way the exploration action is still based on Q rather than a completely random choice.

Python Code Implementation

How do we implement this in code, give an environment (for the smartcab the environment was developed using pygame, it is a grid with traffic lights, streets, intersections, and other cars)?

Let's fill in what we need to turn the psuedo-code into actual code. A reasonable approach is to create a Q class.

```
The Q-Learning algorithm goes as follows:

1. Set the gamma parameter, and environment rewards in matrix R.

'''
Complete: Environment rewards matrix is part of the environment code
TODO: Set gamma, alpha, and epsilon. These values will be initialized with each
new instance of the class.
'''

2. Initialize matrix Q to zero.

#TODO: Initialize an empty dictionary with each Q class instance
```

```
3. For each episode:

   Select a random initial state. #Completed as part of the agent class

Do While the goal state hasn't been reached.

    Select one among all possible actions for the current state.
    Using this possible action, consider going to the next state.
    #TODO: Build selection method as part of the Q class

    Get maximum Q value for this next state based on all possible actions.
    Compute: Q(state, action) = R(state, action) + Gamma * Max[Q(next state, all actions)]
    Set the next state as the current state.
    #TODO: Create a function or functions to calculate Q via the Bellman equation
    and update Q for each state action combination.

End Do

End For
```

Report

Implement Basic Driving Agent

The basic driving agent was implemented by adding the following code to the update method.

```
valid_actions = [None, 'forward', 'left', 'right']
action = random.choice(valid_actions)
```

QUESTION: *Observe what you see with the agent's behavior as it takes random actions. Does the smartcab eventually make it to the destination? Are there any other interesting observations to note?*

As consistent with the implementation the agent wanders about the grid world randomly. It infrequently reaches the destination. The most pertinent observations are:

- The agent does not learn, that is, it never becomes a better driver.
- The agent has no regard for the rules of the road, and if the gridworld was subject to the physical limitations of the real world, it wouldn't last very long.

Inform the Driving Agent

QUESTION: *What states have you identified that are appropriate for modeling the smartcab and environment? Why do you believe each of these states to be appropriate for this problem?*

Inputs and states are implemented in the following manner:

```
# Gather inputs
self.next_waypoint = self.planner.next_waypoint()  # from route planner, also displayed by simulator
inputs = self.env.sense(self)
deadline = self.env.get_deadline(self)

# Update current state
self.state = (inputs['light'], inputs['oncoming'], inputs['left'], inputs['right'], self.next_waypoint)
```

The states consist of:

- Lights: red or green
- Oncoming traffic: None, forward, left, or right
- Traffic left: None, forward, left, right
- Traffic right: None, forward, left, right
- Next Waypoint: None, forward, left, right

Condition Name	Alpha	Gamma	Epsilon

These states are a comprehensive set of states available from the environment and represent a set of common states experienced by actual drivers in the real world.

QUESTION: *How many states in total exist for the smartcab in this environment? Does this number seem reasonable given that the goal of Q-Learning is to learn and make informed decisions about each state? Why or why not?*

The number of states in this environment are 2 * 4 for each other input included since traffic light has two possible values and each additional input has 4 possible values. So, for the inputs chosen for self.state we have num_states= 2 * 4 * 4 * 4 * 4 which yields 512. So the qtable would represent a 512 X 4 (4 valid actions for each state) matrix. Since this is represented as a dictionary, this results in a dictionary with 2048 distinct keys.

This number does see fairly large to sample and learn relevant states in 100 trial. if the smartcab does have trouble learning the best option would likely to remove input['right'] since a car on the right does not represent the complex traffic situations that can result from a car on the left, particularly at intersections. This would reduce the number of states to 128, which is a more reasonable number to sample in 100 trials.

Implement a Q-Learning Driving Agent

QUESTION: *What changes do you notice in the agent's behavior when compared to the basic driving agent when random actions were always taken? Why is this behavior occurring?*

At first I didn't think my implementation was working because the initial rounds appeared random. Soon the agent changed from driving aimlessly to honing in on the target. The agent's driving approved dramatically in a short time period and then the improvement began to level off.
This behavior is occurring because with each sampled state, action combination we update the q-table, adding values to inform the action decision of the agent.

The one very unusual behavior I noticed in from the driving agent was driving in circles. When confronted with a red light the agent often chose to drive around the red light rather than to simply wait until it turned green. This may work well in the grid-world but it would be considered odd in the real world. Why is this occurring? The highest quality state for all of the conditions studied, for a red light, is to turn right. Any action other than turning right on a red (no matter the other elements of the state) has a q value <= 0.

Improve the Q-Learning Driving Agent

The starting parameters for training the agent, which I refer to as base parameters are alpha (learning rate)= 0.1, gamma (discounted future reward)= 0.9, epsilon= 0.1

The alpha value is fairly low which helps prevent changes in q from occurring too rapidly. The gamma value is close to what we would choose in a deterministic instead of a stochastic environment. This is value I was most interested in changing to see the impact from the base case. Epsilon is low to allow for exploration. I was less interested in this value because of the way exploration is implemented, that is if a random value is > epsilon with add a random value to q. This is a more gradual way of introducing exploration, since exploration is not purely dependent on whether we "roll" less than or great than epsilon.

QUESTION: *Report the different values for the parameters tuned in your basic implementation of Q-Learning. For which set of parameters does the agent perform best? How well does the final driving agent perform?*

Parameters tuned (alpha, gamma, epsilon), their values, and the results for Key metrics, are tabulated and plotted, below. Parameters were sampled in a range where the intent is to fill in the parameters space between ranges if any particular setting demonstrates a promising trend.

Condition and Parameter Key

Condition Name	Alpha	Gamma	Epsilon
alphap1_gammap9_epsp1	0.10	0.90	0.10
alphap5_gammap9_epsp1	0.50	0.90	0.10
alphap1_gammap25_epsp1	0.10	0.25	0.10
alphap1_gammap9_epsp25	0.10	0.90	0.25

Condition Name	Alpha	Gamma	Epsilon	[illegible]
alphap1_gammap5_epsp1	0.10	0.50	0.10	
alphap25_gammap9_epsp1	0.25	0.90	0.10	
alphap1_gammap9_epsp5	0.10	0.90	0.50	

Global Success Rate (successes/round)

Condition Name	Alpha	Gamma	Epsilon	Global Success Rate
alphap1_gammap9_epsp1	0.10	0.90	0.10	78%
alphap5_gammap9_epsp1	0.50	0.90	0.10	76%
alphap1_gammap25_epsp1	0.10	0.25	0.10	68%
alphap1_gammap9_epsp25	0.10	0.90	0.25	68%
alphap1_gammap5_epsp1	0.10	0.50	0.10	67%
alphap25_gammap9_epsp1	0.25	0.90	0.10	59%
alphap1_gammap9_epsp5	0.10	0.90	0.50	59%

- Success rate is measured as successes/round
- The best success rates (78%, 76%) corresponds to:
  - [alpha: 0.10, gamma: 0.90, epsilon: 0.10]
  - [alpha: 0.5, gamma: 0.90, epsilon: 0.10]
- The remaining conditions are 10 - 19% lower than the top conditions.

Success Count

Condition Name	Alpha	Gamma	Epsilon	Number of Successes
alphap5_gammap9_epsp1	0.50	0.90	0.10	98
alphap1_gammap9_epsp1	0.10	0.90	0.10	97
alphap1_gammap25_epsp1	0.10	0.25	0.10	97
alphap25_gammap9_epsp1	0.25	0.90	0.10	95
alphap1_gammap9_epsp25	0.10	0.90	0.25	95
alphap1_gammap5_epsp1	0.10	0.50	0.10	93
alphap1_gammap9_epsp5	0.10	0.90	0.50	92

- The overall number of successes per 100 trials are only marginally different per condition.
- Combined with the success rate data that indicates that the agent is able to reach it's destination reliably, independent of the conditions, and that the primary difference is how many rounds this takes.
- It's not clear if this is a function of agent parameters or random variation in the distance of the destination from the agent origin.

Q-table Length

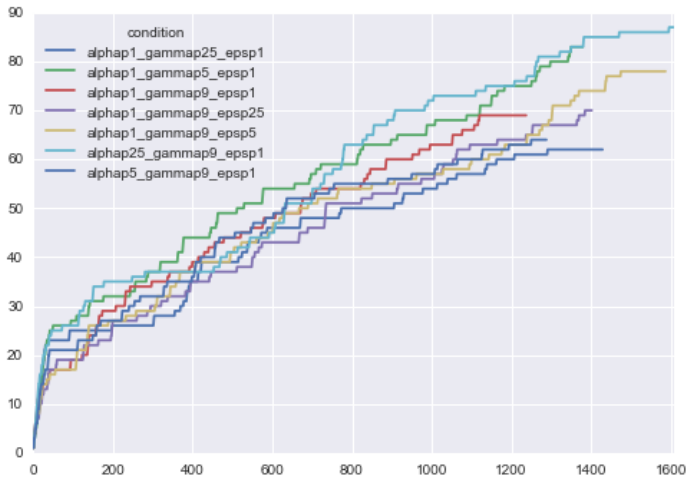
Condition Name	Alpha	Gamma	Epsilon	Table Length
alphap25_gammap9_epsp1	0.25	0.90	0.10	87
alphap1_gammap5_epsp1	0.10	0.50	0.10	83
alphap1_gammap9_epsp5	0.10	0.90	0.50	78

Condition Name	Alpha	Gamma	Epsilon	Table Length
alphap1_gammap9_epsp25	0.10	0.90	0.25	70
alphap1_gammap9_epsp1	0.10	0.90	0.10	69
alphap5_gammap9_epsp1	0.50	0.90	0.10	64
alphap1_gammap25_epsp1	0.10	0.25	0.10	62

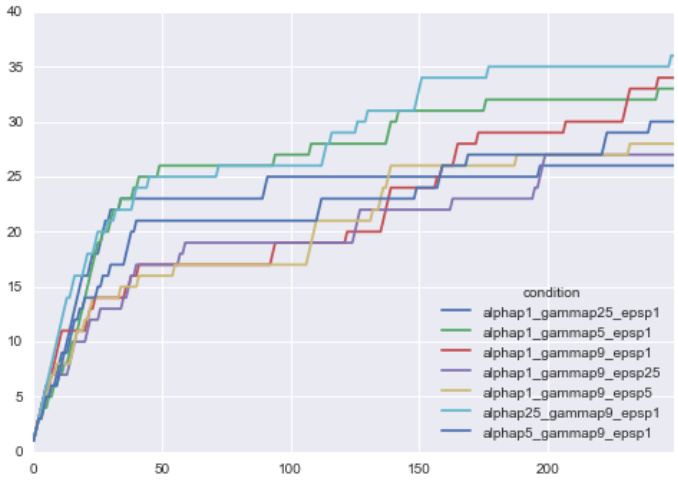
- Q-table length represents the number of distinct states encountered by the agent and assigned a Q-value.
- The most samples state is [alpha: 0.25, gamma: 0.90, epsilon: 0.10]
- Followed by [alpha: 0.10, gamma: 0.50, epsilon: 0.10]
- and [alpha: 0.10, gamma: 0.90, epsilon: 0.50]
- These settings also correspond to the lowest 3 success rates, so once again it is challenging to say whether or not this is truly a function of the parameters or the stochastic nature of the environment.

Plots

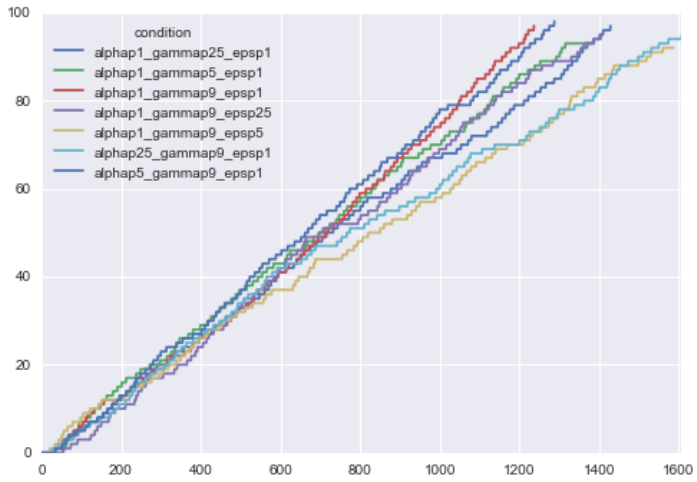
Length of Q-table by round



Length of Q-table for first 200 rounds



Success Rate



Agent with Largest Q-table vs. Agent with Highest Success Rate

- The agent with settings that generate the largest Q-table [alpha: 0.25, gamma: 0.90, gamma: 0.10] has 95 successes, a success rate of 59%, and a q table with 87 (state, action) combinations
- The agent with settings that generate the best success rate [alpha: 0.25, gamma: 0.90, gamma: 0.10] has 97 successes, a success rate of 78%, and a q-table with 69 (state, action) combinations
- There are 52 unique (state, action) combinations for largest - highest
- There are 34 unique (state, action) combinations for highest - largest

  Largest Q (state, action) combinations in largest not in highest

```
[((('red', 'left', None, None, 'right'), 'right'), 7.823596515641668),
 ((('red', None, None, 'left', 'right'), 'right'), 2.0),
 ((('green', None, 'right', None, 'right'), 'right'), 2.0),
 ((('green', None, None, 'left', 'right'), 'right'), 2.0),
 ((('green', None, None, 'right', 'right'), 'right'), 2.0),
 ((('red', None, None, 'forward', 'right'), 'right'), 2.0),
 ((('green', None, None, 'right', 'forward'), 'forward'), 9.649999999999999),
 ((('green', None, 'left', None, 'right'), 'right'), 2.0)]
```

- Right hand turns are incentivized for most of the differences

  Largest Q (state, action) combinations in highest not in largest

```
[((('green', None, 'forward', None, 'forward'), 'forward'), 12.0),
 ((('green', None, None, 'forward', 'right'), 'right'), 2.0),
 ((('red', 'forward', None, None, 'right'), 'right'), 2.0),
 ((('green', None, None, 'left', 'forward'), 'forward'), 12.1592),
 ((('green', None, None, 'forward', 'forward'), 'forward'), 2.162),
 ((('green', None, 'right', None, 'left'), 'left'), 2.0)]
```

- Other light behavior (left and forward) incentivized for differences

For which set of parameters does the agent perform best?

It's challenging to determine which set of parameters yields the best performance. In terms of overall successes per 100 trials, there is only a marginal differences for agents give the tested parameter sets. In terms of rate and explored state action combinations, there are larger differences. All things considered I chose the agent with the highest success rate, partially since the unique (state, action) combinations for the agent with the largest Q-table incentivize behaviors best avoided (see below).

- The agent with settings that generate the best success rate [alpha: 0.25, gamma: 0.90, gamma: 0.10] has 97 successes, a success rate of 78%, and a q-table with 69 (state, action) combinations

QUESTION: *Does your agent get close to finding an optimal policy, i.e. reach the destination in the minimum possible time, and not incur any penalties? How would you describe an optimal policy for this problem?*

I do think my agent comes close to reaching the destination in the minimal possible time without incurring penalties. The final policy for the base case (alphap1_gammap9_epsp1) shows that only actions that follow driving rules are high quality whereas those that don't are low quality. The only thing that I find suboptimal is that right turns are highly incentivized. The end result is that the agent tends to drive around in circles. This may play out well in the grid world but in the real world it would be odd at best and at worst it would waste gas and potentially time, as traffic conditions and delays from making a circle are not as predictable as the grid world (gridworld is more deterministic).

I would describe an optimal policy as one in which the destination is reached in the smallest time frame given that traffic rules are followed and unusual driving behavior is avoided.

Q-tables

Final Q-table for the base case (alphap1_gammap9_epsp1) represented as a python dictionary:

```
{(('green', None, None, None, 'forward'), None): 0.0,
 (('green', None, None, None, 'forward'), 'forward'): 6.132283212465348,
 (('green', None, None, None, 'left'), 'left'): 8.43767377788923,
 (('green', None, None, None, 'right'), 'left'): -0.5,
 (('green', None, None, None, 'right'), 'right'): 6.809170982132021,
 (('green', None, None, 'forward', 'forward'), 'forward'): 2.162,
 (('green', None, None, 'forward', 'left'), 'right'): -0.5,
 (('green', None, None, 'forward', 'right'), 'right'): 2.0,
 (('green', None, None, 'left', 'forward'), None): 0.0,
 (('green', None, None, 'left', 'forward'), 'forward'): 12.1592,
 (('green', None, None, 'left', 'forward'), 'left'): -0.5,
 (('green', None, None, 'left', 'forward'), 'right'): -0.5,
 (('green', None, None, 'left', 'left'), 'right'): -0.5,
 (('green', None, None, 'right', 'forward'), 'left'): -0.5,
 (('green', None, 'forward', None, 'forward'), 'forward'): 12.0,
 (('green', None, 'left', None, 'forward'), None): 0.0,
 (('green', None, 'left', None, 'forward'), 'forward'): 12.0,
 (('green', None, 'left', None, 'forward'), 'right'): -0.5,
 (('green', None, 'left', None, 'left'), 'right'): -0.5,
 (('green', None, 'left', None, 'right'), 'forward'): -0.5,
 (('green', None, 'right', None, 'forward'), 'left'): -0.5,
 (('green', None, 'right', None, 'forward'), 'right'): -0.5,
 (('green', None, 'right', None, 'left'), 'left'): 2.0,
 (('green', 'forward', None, None, 'forward'), None): 0.0,
 (('green', 'forward', None, None, 'right'), 'left'): -1.0,
 (('green', 'left', None, None, 'forward'), None): 0.0,
 (('green', 'left', None, None, 'forward'), 'forward'): 11.582030313240434,
 (('green', 'left', None, None, 'forward'), 'left'): -0.5,
 (('green', 'left', None, None, 'forward'), 'right'): -0.5,
 (('green', 'right', None, None, 'forward'), 'right'): -0.5,
 (('green', 'right', None, None, 'right'), 'forward'): -0.5,
 (('red', None, None, None, 'forward'), None): 0.0,
 (('red', None, None, None, 'forward'), 'forward'): -1.0,
 (('red', None, None, None, 'forward'), 'left'): -1.0,
 (('red', None, None, None, 'forward'), 'right'): -0.5,
 (('red', None, None, None, 'left'), None): 0.0,
 (('red', None, None, None, 'left'), 'forward'): -1.0,
 (('red', None, None, None, 'left'), 'left'): -1.0,
 (('red', None, None, None, 'left'), 'right'): -0.10283403044109152,
 (('red', None, None, None, 'right'), 'left'): -1.0,
 (('red', None, None, None, 'right'), 'right'): 6.218542206861301,
 (('red', None, None, 'forward', 'forward'), None): 0.0,
 (('red', None, None, 'forward', 'right'), None): 0.0,
 (('red', None, None, 'forward', 'right'), 'forward'): -1.0,
 (('red', None, None, 'left', 'forward'), None): 0.0,
 (('red', None, None, 'left', 'forward'), 'right'): -0.5,
 (('red', None, 'forward', None, 'forward'), 'forward'): -1.0,
 (('red', None, 'forward', None, 'forward'), 'left'): -1.0,
 (('red', None, 'forward', None, 'forward'), 'right'): -1.0,
 (('red', None, 'left', None, 'forward'), None): 0.0,
 (('red', None, 'left', None, 'forward'), 'forward'): -1.0,
```

```
(('red', None, 'right', None, 'forward'), None): 0.0,
(('red', 'forward', None, None, 'forward'), None): 0.0,
(('red', 'forward', None, None, 'forward'), 'forward'): -1.0,
(('red', 'forward', None, None, 'forward'), 'left'): -1.0,
(('red', 'forward', None, None, 'forward'), 'right'): -0.5,
(('red', 'forward', None, None, 'left'), 'forward'): -1.0,
(('red', 'forward', None, None, 'left'), 'left'): -1.0,
(('red', 'forward', None, None, 'right'), 'right'): 2.0,
(('red', 'left', None, None, 'forward'), None): 0.0,
(('red', 'left', None, None, 'forward'), 'forward'): -1.0,
(('red', 'left', None, None, 'forward'), 'left'): -1.0,
(('red', 'left', None, None, 'forward'), 'right'): -0.5,
(('red', 'left', None, None, 'left'), 'forward'): -1.0,
(('red', 'left', None, None, 'left'), 'left'): -1.0,
(('red', 'right', None, None, 'forward'), None): 0.0,
(('red', 'right', None, None, 'forward'), 'forward'): -1.0,
(('red', 'right', None, None, 'forward'), 'right'): -0.5,
(('red', 'right', None, None, 'left'), None): 0.0}
```

Final Q-table for largest Q-Table (alphap25_gammap9_epsp1) represented as a python dictionary:

```
{(('green', None, None, None, 'forward'), None): 0.0,
 (('green', None, None, None, 'forward'), 'forward'): 11.473222212188928,
 (('green', None, None, None, 'forward'), 'left'): -0.5,
 (('green', None, None, None, 'forward'), 'right'): -0.5,
 (('green', None, None, None, 'left'), 'left'): 11.858519166007213,
 (('green', None, None, None, 'right'), None): 0.0,
 (('green', None, None, None, 'right'), 'forward'): -0.5,
 (('green', None, None, None, 'right'), 'left'): -0.5,
 (('green', None, None, None, 'right'), 'right'): 11.5790139087193,
 (('green', None, None, 'forward', 'right'), 'forward'): -0.5,
 (('green', None, None, 'left', 'forward'), 'right'): -0.5,
 (('green', None, None, 'left', 'right'), 'right'): 2.0,
 (('green', None, None, 'right', 'forward'), 'forward'): 9.649999999999999,
 (('green', None, None, 'right', 'forward'), 'left'): -0.5,
 (('green', None, None, 'right', 'right'), 'forward'): -0.5,
 (('green', None, None, 'right', 'right'), 'right'): 2.0,
 (('green', None, 'forward', None, 'forward'), 'right'): -0.5,
 (('green', None, 'forward', None, 'left'), None): 0.0,
 (('green', None, 'forward', None, 'left'), 'forward'): -0.5,
 (('green', None, 'forward', None, 'left'), 'right'): -0.5,
 (('green', None, 'forward', None, 'right'), 'forward'): -0.5,
 (('green', None, 'left', None, 'forward'), None): 0.0,
 (('green', None, 'left', None, 'forward'), 'forward'): 2.0,
 (('green', None, 'left', None, 'forward'), 'left'): -0.5,
 (('green', None, 'left', None, 'forward'), 'right'): -0.5,
 (('green', None, 'left', None, 'left'), None): 0.0,
 (('green', None, 'left', None, 'left'), 'forward'): -0.5,
 (('green', None, 'left', None, 'left'), 'right'): -0.5,
 (('green', None, 'left', None, 'right'), None): 0.0,
 (('green', None, 'left', None, 'right'), 'right'): 2.0,
 (('green', None, 'left', 'right', 'right'), 'left'): -0.5,
 (('green', None, 'right', None, 'right'), 'right'): 2.0,
 (('green', 'forward', None, None, 'left'), None): 0.0,
 (('green', 'left', None, None, 'forward'), 'forward'): 6.0249028008791345,
 (('green', 'left', None, None, 'forward'), 'right'): -0.5,
 (('green', 'left', None, None, 'right'), None): 0.0,
 (('green', 'left', None, None, 'right'), 'forward'): -0.5,
 (('green', 'left', None, None, 'right'), 'left'): -0.5,
 (('green', 'right', None, None, 'forward'), None): 0.0,
 (('green', 'right', None, None, 'forward'), 'left'): -1.0,
 (('green', 'right', None, None, 'right'), None): 0.0,
 (('red', None, None, None, 'forward'), None): 0.0,
 (('red', None, None, None, 'forward'), 'forward'): -1.0,
 (('red', None, None, None, 'forward'), 'left'): -1.0,
 (('red', None, None, None, 'forward'), 'right'): 8.842588452559566,
 (('red', None, None, None, 'left'), None): 0.0,
 (('red', None, None, None, 'left'), 'forward'): -1.0,
 (('red', None, None, None, 'left'), 'left'): -1.0,
 (('red', None, None, None, 'left'), 'right'): 10.273587054351314,
```

```
    (('red', None, None, None, 'right'), None): 0.0,
    (('red', None, None, None, 'right'), 'forward'): -1.0,
    (('red', None, None, None, 'right'), 'left'): -1.0,
    (('red', None, None, None, 'right'), 'right'): 13.034483252238077,
    (('red', None, None, 'forward', 'forward'), None): 0.0,
    (('red', None, None, 'forward', 'forward'), 'left'): -1.0,
    (('red', None, None, 'forward', 'left'), 'forward'): -1.0,
    (('red', None, None, 'forward', 'right'), 'left'): -1.0,
    (('red', None, None, 'forward', 'right'), 'right'): 2.0,
    (('red', None, None, 'left', 'left'), 'right'): -0.5,
    (('red', None, None, 'left', 'right'), 'right'): 2.0,
    (('red', None, None, 'right', 'forward'), 'left'): -1.0,
    (('red', None, None, 'right', 'right'), None): 0.0,
    (('red', None, None, 'right', 'right'), 'left'): -1.0,
    (('red', None, 'forward', None, 'left'), 'forward'): -1.0,
    (('red', None, 'forward', None, 'right'), 'right'): -1.0,
    (('red', None, 'left', None, 'forward'), None): 0.0,
    (('red', None, 'left', None, 'forward'), 'left'): -1.0,
    (('red', None, 'left', None, 'forward'), 'right'): -0.5,
    (('red', None, 'right', None, 'forward'), 'left'): -1.0,
    (('red', 'forward', None, None, 'forward'), None): 0.0,
    (('red', 'forward', None, None, 'forward'), 'left'): -1.0,
    (('red', 'forward', None, None, 'forward'), 'right'): -0.5,
    (('red', 'forward', None, None, 'left'), 'right'): -0.5,
    (('red', 'left', None, None, 'forward'), None): 0.0,
    (('red', 'left', None, None, 'forward'), 'forward'): -1.0,
    (('red', 'left', None, None, 'forward'), 'left'): -1.0,
    (('red', 'left', None, None, 'forward'), 'right'): -0.5,
    (('red', 'left', None, None, 'left'), None): 0.0,
    (('red', 'left', None, None, 'left'), 'forward'): -1.0,
    (('red', 'left', None, None, 'left'), 'left'): -1.0,
    (('red', 'left', None, None, 'left'), 'right'): -0.5,
    (('red', 'left', None, None, 'right'), None): 0.0,
    (('red', 'left', None, None, 'right'), 'forward'): -1.0,
    (('red', 'left', None, None, 'right'), 'left'): -1.0,
    (('red', 'left', None, None, 'right'), 'right'): 7.823596515641668,
    (('red', 'right', None, None, 'forward'), 'forward'): -1.0,
    (('red', 'right', None, None, 'left'), 'forward'): -1.0}
```

References and Resources

1. The Study Wolf Blog blog contains a great overview of reinforcement learning and Q-learning and even a python code implementation. This implementation was modified for use in this project see q_learn.py.
2. This great tutorial shows how Q-learning tables could be applied to teach an agent to move from a randomly selected room, out of a house.
3. Demystifying Deep Reinforcement Learning is another great resource with explanations of how Q learning, reinforcement learning, and Markov decision processes relate.