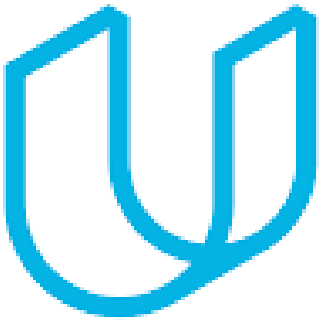
Reinforcement Learning

Train a Smart cab to Drive

Udacity Machine Learning Engineer Nanodegree - Project 7

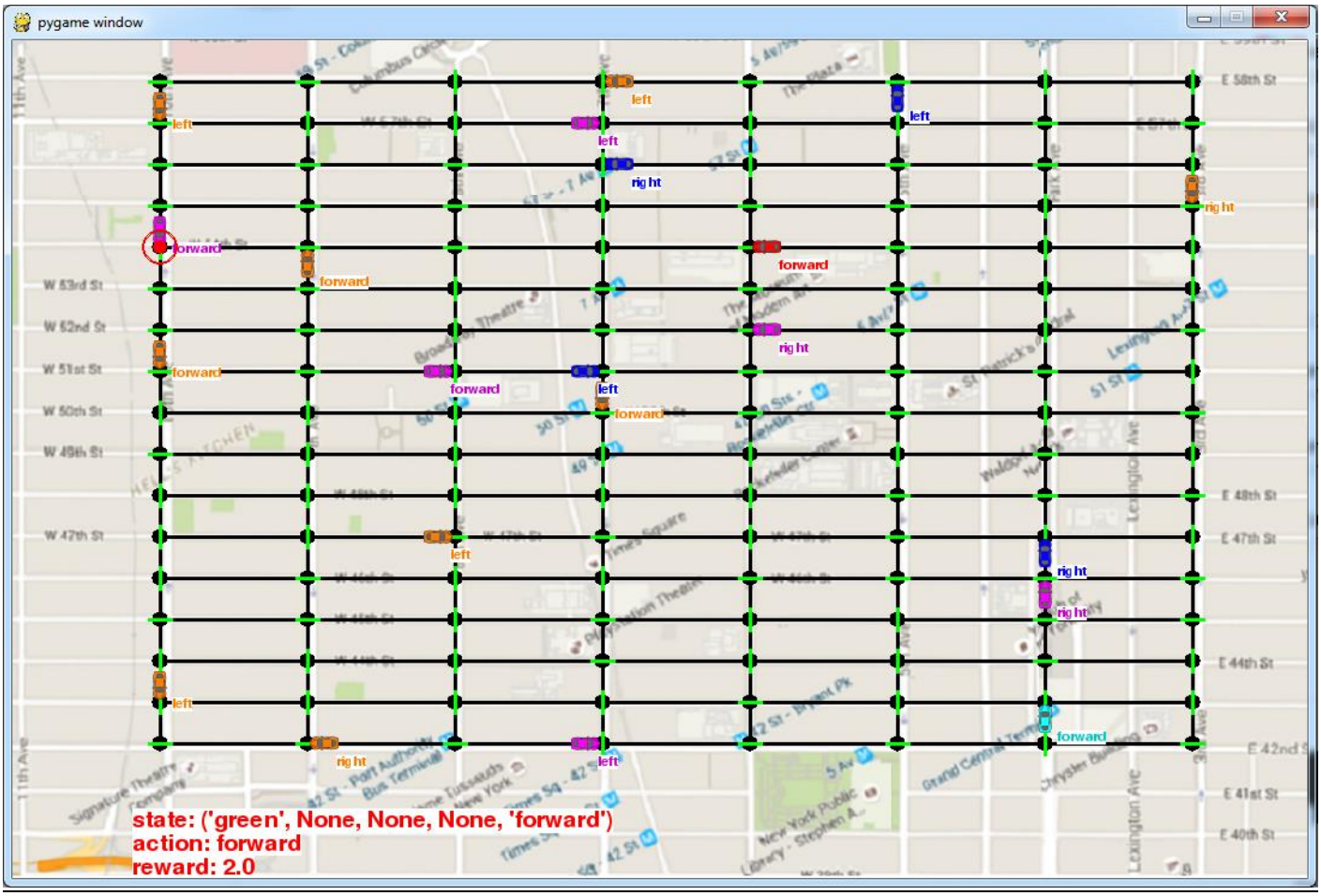
Diego Menin – June 2016



UDACITY

The Environment:

- The Red Car is the Learning Agent;
- The Red dot is the destination;
- All other cars are Dummy Agents who randomly mover around the map;
- The map is from Manhattan in NY city, but it serves only as background, it won't affect the algorithm nor I implemented Manhattan traffic rules, so the agents can move UP and DOWN, EAST and WEST on any direction;



Task 1: Implement a basic driving agent

Implement the basic driving agent, which processes the following inputs at each time step:

- Next waypoint location, relative to its current location and heading,
- Intersection state (traffic light and presence of cars), and,
- Current deadline value (time steps remaining),

And produces some random move/action (None, 'forward', 'left', 'right'). In your report, mention what you see in the agent's behaviour. Does it eventually make it to the target location?

The implementation was made on the "*update*" method of the "**LearningAgent**" class on the apent.py file. That method uses a function called "get_next_waypoint_given_location" from the Agent class, that, given the agent's locations, it randomly select the next action. The only caveat is that I wrote this function to prevent the agents from crossing from one side to the other of the map, so if the agent is facing north at position (1, 3), its only available options are left and right. This function is also shared with the dummy agents.

Also, to avoiding code duplication, I moved the code that checks if an action is ok from the "*update*" method on the "**DummyAgent**" class to a function called "*check_if_action_is_ok*" inside the "**Agent**" class, so I can use it on the "*update*" method on the "**LearningAgent**" class.

As expected, the agent moves around the map randomly. Eventually it may reach the destination but it is only a matter of chance.

Task 2: Identify and Update States

Identify a set of states that you think are appropriate for modelling the driving agent. Justify why you picked these set of states, and how they model the agent and its environment.

We have a few options to use as states. The most obvious are the result from the "sense" function, which the agent uses to "feel" the environment. It includes the traffic light state and whether we have left, right and oncoming traffic. We also have the "next_waypoint" defined by the planner and the "deadline" to get to the destination.

Due to their high importance, I chose to use all of variables above, with the exception of the "deadline" and the reason for that is that choosing to include it, means that I'd have to treat states separately for each deadline on the QTable – which doesn't make sense from my point of view. The deadline variable can take blows up the state space into a size that cannot be feasibly explored by the agent. Furthermore, I know that in order to learn the value of each action, the agent will need to visit every state many times over. If we have too many states it will take a very long time to learn, but also, if you have too few states it may be unable to distinguish between some actions.

Maybe, it would be a good idea to treat the deadline as a bucket or a percentage, so the agent would learn different actions if it is "too late" or not, but I didn't try this approach.

There is also an important point to be made related to the "oncoming traffic from the right". The project scope mentions that according to "circulation rules in US, the traffic coming from the right does not matter", which would be a good reason to leave that input out from the states, but I feel that it is a terrible decision because there may be circumstances where that is irrelevant, like, let say there is an ambulance, a fire truck or a police car for whatever reason coming for the right side, or even a drunk driver or any other driver that is simply wrong, willing to do it or not.

I feel that the agent should be able to sense the environment to avoid a collision on any of the situations mentioned above.

Task 3: Implement Q-Learning

Implement the Q-Learning algorithm by initializing and updating a table/mapping of Q-values at each time step. Now, instead of randomly selecting an action, pick the best action available from the current state based on Q-values, and return that. Each action generates a corresponding numeric reward or penalty (which may be zero). Your agent should take this into account when updating Q-values. Run it again, and observe the behaviour. What changes do you notice in the agent's behaviour?

To do this, I created a "QLearningAgent" class that also inherits from "Agent". It contains a "QLearn" object where all the learning happens. The "update method" implements a simple Q-Learning approach:

1) Sense the environment (see what changes naturally occur in the environment)
2) Take an action - get a reward
3) Sense the environment (see what changes the action has on the environment)
4) Update the Q-table (Learn)
5) Repeat

And the learning is calculate using the formula bellow:

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

Where:

- Q(s,a) = current reward of the previous state
- alpha = learning rate
- R(s,a) = reward received for the last move
- Gamma = value of future reward
- Argmax = best of all possible follow up actions from the state the agent is after the move

The agent now will start by taking random moves because the QTable starts empty - it hasn't learned anything yet – so by trying to get the best reward (they are all zero) it randomly picks an action among those, but as the agent moves, it learns the environment and starts using the QTable to decide which action to take.

This is how my QTable looks like, where the two first columns representing the state – just separated them for a better visualization, under the hood is one big tuple – and the last column is the reward:

```
QTable:
+----+-----------------------------------------+----------+----------+
|    | State (light / oncoming / right / left) | action   |   reward |
+====+=========================================+==========+==========+
|  0 | ('green', None, None, None, 'right')    | forward  | -0.375   |
+----+-----------------------------------------+----------+----------+
|  1 | ('red', None, None, None, 'forward')    | right    | -0.25    |
+----+-----------------------------------------+----------+----------+
```

There is also a third variable that is not on the above formula, called pRandomMove (also known as *epsilon*), which defines the percentage of time the agent will take a random (exploration) move, regardless of the QTable.

Task 4 - Enhance the driving agent

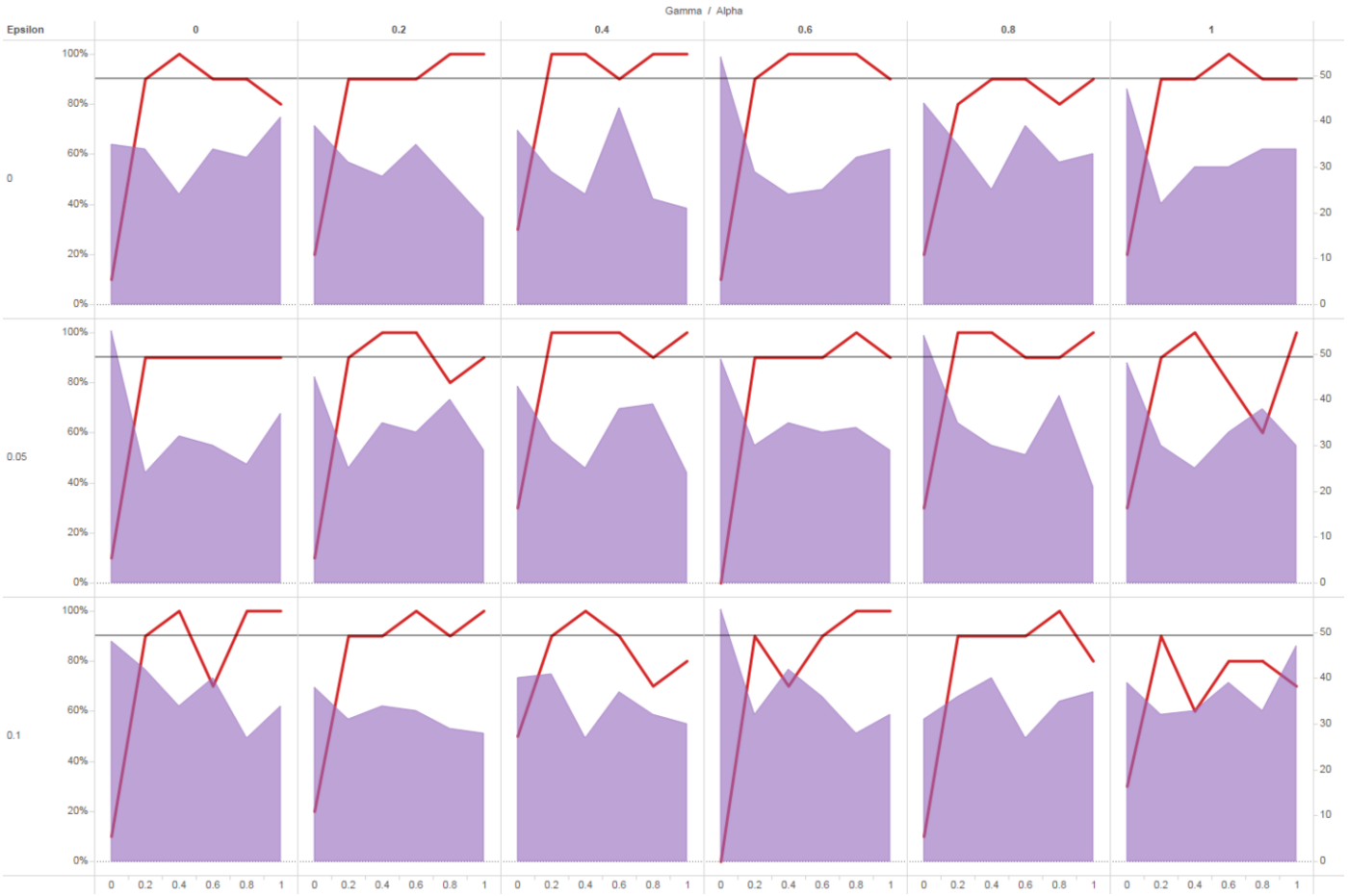
Apply the reinforcement learning techniques you have learnt, and tweak the parameters (e.g. learning rate, discount factor, action selection method, etc.), to improve the performance of your agent. Your goal is to get it to a point so that within 100 trials, the agent is able to learn a feasible policy - i.e. reach the destination within the allotted time, with net reward remaining positive.

Now that the Agent was learning, I went through the task of finding the best set of parameters (epsilon, alpha and gamma) where it performs better. I first tried by running 10 trials with each combination. Alpha and gamma 0 to 1 in in 0.2 steps and epsilon 0, 0.05 and 0.1 (because I don't think the agent should randomly explore more than 10% of the time).

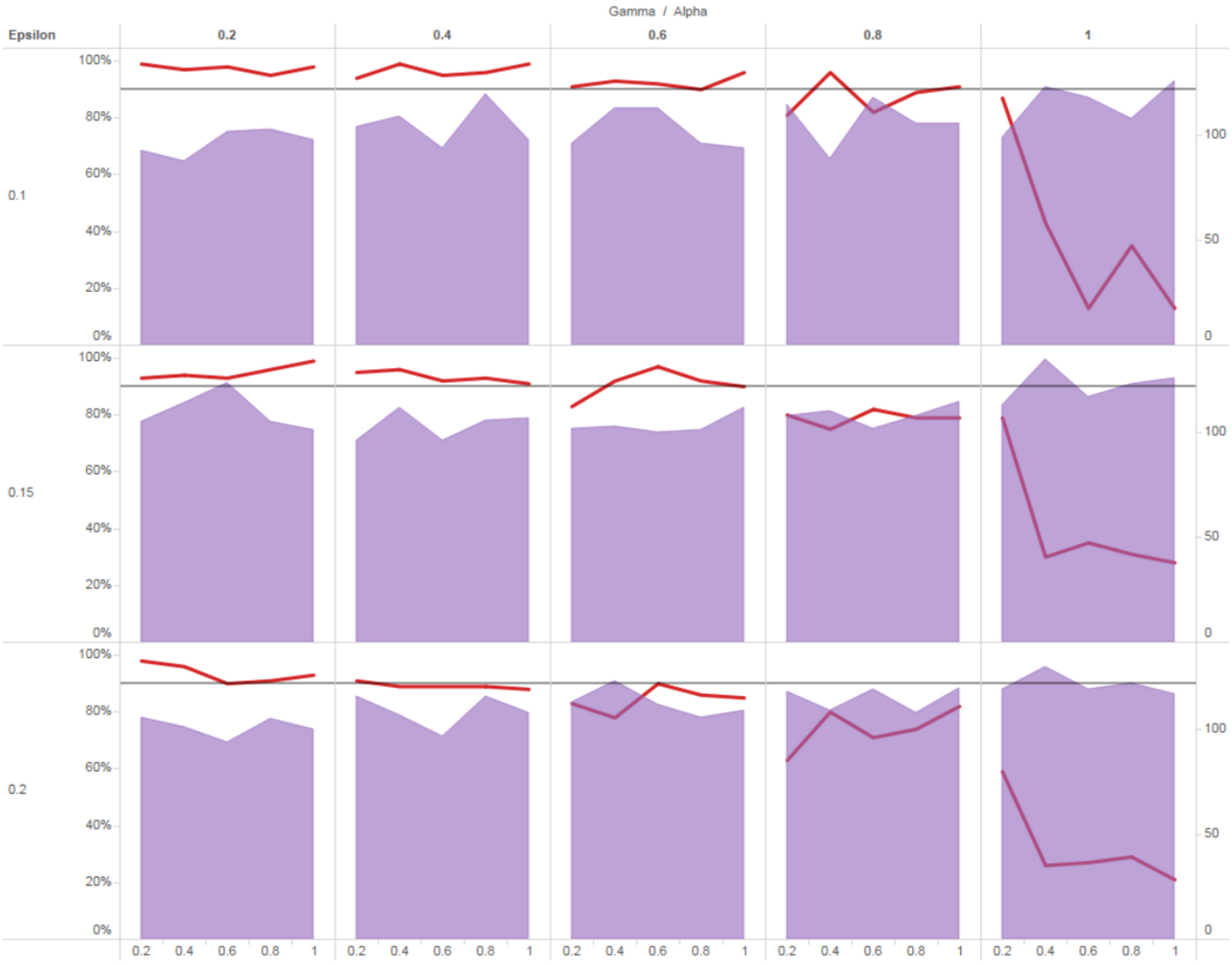
Below is a graph that outputs the result. The red line shows the percentage of the time it got to the destination (so if it is 100%, it got it right 10 out of 10 times) and the purple area, the size of the QTable on the last step of the trial (which is there only as a matter of curiosity). The grey line is the 905 threshold.

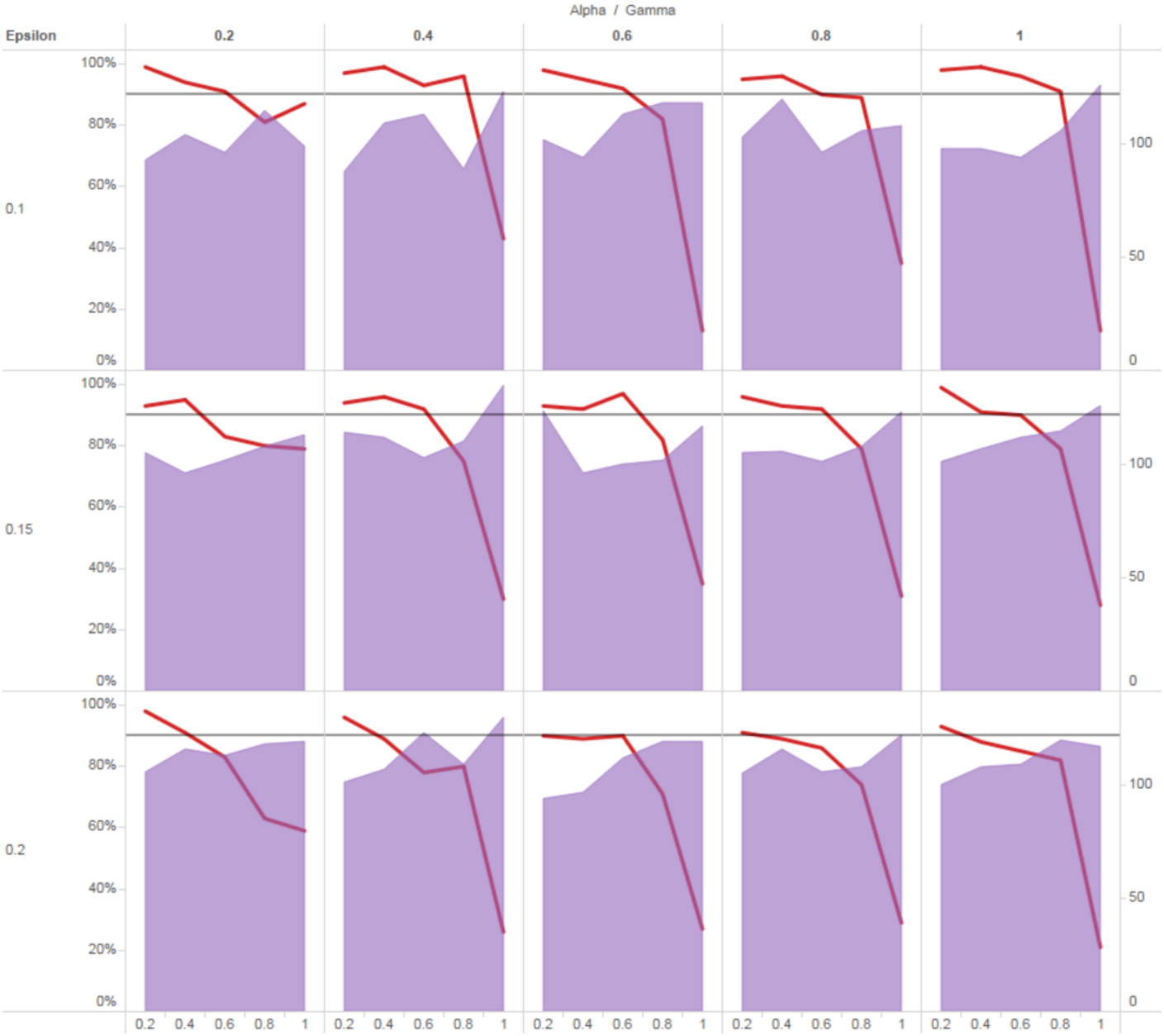
Each data point represents 10 trials with a particular combination of epsilon, alpha and gamma and was executed with an empty QTable; The result is the sum of times the agent got to the destination, so if we look at the first picture on the top left, with epsilon, gamma and alpha all being 0, it got to the destination only once; On the second data point (still same top left picture), epsilon =0, gamma =0.2 and alpha =0 , it got to the destination 9 times using a smaller QTable.

BTW, I understand that the images bellow may not look the best on this report, so I deployed this version of the dashboard to [this](#) link.



After some feedback on the forms I realized that 10 trials weren't enough, so I ran the exercise again with 100 trials (on a 0.2 step at this time) and the result follows bellow (this time I'm showing the graph plotted as Gamma\Alpha and Alpha\Gamma):


Gamma / Alpha
Epsilon
0.2
0.4
0.6
0.8
1
0.1
0.15
0.2
100%
80%
60%
40%
20%
0%
100
50
0
0.2
0.4
0.6
0.8
1

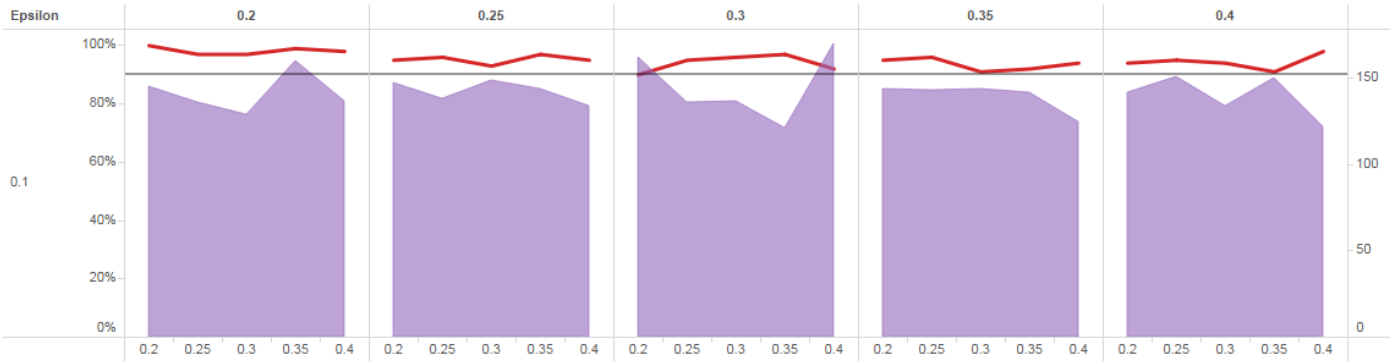


Looking at these plots I can see that the best combination is around epsilon = 0.10 and gamma\alpha between 0.2 and 0.4 where the cab got to the destination 99% of the time.

		Gamma				
Epsilon	**Alpha**	**0.2**	**0.4**	**0.6**	**0.8**	**1**
0.1	0.2	**99%**	94%	91%	81%	87%
	0.4	97%	**99%**	93%	96%	43%
	0.6	98%	95%	92%	82%	13%
	0.8	95%	96%	90%	89%	35%
	1	98%	**99%**	96%	91%	13%
0.15	0.2	93%	95%	83%	80%	79%
	0.4	94%	96%	92%	75%	30%
	0.6	93%	92%	97%	82%	35%
	0.8	96%	93%	92%	79%	31%
	1	**99%**	91%	90%	79%	28%
0.2	0.2	98%	91%	83%	63%	59%
	0.4	96%	89%	78%	80%	26%
	0.6	90%	89%	90%	71%	27%
	0.8	91%	89%	86%	74%	29%
	1	93%	88%	85%	82%	21%

To complicate things a little, I ran the test once more with epsilon 0.1 and alpha\gamma between 0.2 and 0.4 with a step of 0.05. I have also increased the map size to have 17 rows and 8 columns (that's driven by the Manhattan map on the background☺) and increase the traffic to 20 dummy cars.

The results bellow confirm what alpha 0.2 is indeed the best option, so is gamma, which besides having a very good result on 0.4\0.4, is also pretty good on 02. The agent even got 100% correctness on 0.2\0.2.



		Gamma				
Epsilon	**Alpha**	**0.2**	**0.25**	**0.3**	**0.35**	**0.4**
0.1	0.2	**100%**	95%	90%	95%	94%
	0.25	97%	96%	95%	96%	95%
	0.3	97%	93%	96%	91%	94%
	0.35	**99%**	97%	97%	92%	91%
	0.4	98%	95%	92%	94%	**98%**

So, I believe that the agent is behaving optimally. If I look at my log tables at the end of the simulation, I can see a bigger difference between the distance to the destination and the number of steps taken.

Important note: *the number of steps are represented in time units which it is not equal to the distance, because if the agent stops at a red light, the steps keep increasing, so the only way of getting Distance = Steps is to go straight to the destination and to get all green lights).*

That being said, simulations like 83, 86, 86 and 88 are good examples of learning.

First trials:

	A	B	C	D	E	F	G	H	I	J
1	Trial	Distance	HeadLine	Sucess	Steps	Epsilon	Alpha	Gamma	QTableSize	
2	0	15	75	Y	54	0.1	0.2	0.2	21	
3	1	14	70	Y	55	0.1	0.2	0.2	26	
4	2	8	40	Y	18	0.1	0.2	0.2	30	
5	3	10	50	Y	21	0.1	0.2	0.2	31	
6	4	19	95	Y	57	0.1	0.2	0.2	47	
7	5	10	50	Y	33	0.1	0.2	0.2	53	
8	6	8	40	Y	19	0.1	0.2	0.2	57	

Final Trials:

	A	B	C	D	E	F	G	H	I
1	Trial	Distance	DeadLine	Sucess	Steps	Epsilon	Alpha	Gamma	QTableSize
83	81	7	35	Y	16	0.1	0.2	0.2	134
84	82	7	35	Y	12	0.1	0.2	0.2	135
85	83	4	20	Y	6	0.1	0.2	0.2	135
86	84	13	65	Y	24	0.1	0.2	0.2	135
87	85	6	30	Y	12	0.1	0.2	0.2	135
88	86	7	35	Y	12	0.1	0.2	0.2	135
89	87	14	70	Y	39	0.1	0.2	0.2	138
90	88	7	35	Y	9	0.1	0.2	0.2	139
91	89	18	90	Y	30	0.1	0.2	0.2	139
92	90	10	50	Y	21	0.1	0.2	0.2	143
93	91	5	25	Y	10	0.1	0.2	0.2	143
94	92	4	20	Y	11	0.1	0.2	0.2	144
95	93	8	40	Y	15	0.1	0.2	0.2	144
96	94	10	50	Y	28	0.1	0.2	0.2	144
97	95	6	30	Y	12	0.1	0.2	0.2	145
98	96	16	80	Y	27	0.1	0.2	0.2	145
99	97	9	45	Y	15	0.1	0.2	0.2	145
100	98	10	50	Y	21	0.1	0.2	0.2	145
101	99	14	70	Y	36	0.1	0.2	0.2	145
102									

Final QTable:

1		State (light / oncoming / right / left)	action	reward
2	114	('green', None, None, 'forward', 'forward')	forward	3.809513
3	42	('green', None, 'right', None, 'forward')	forward	3.57947
4	16	('green', None, None, None, 'forward')	forward	3.106151
5	73	('green', None, None, None, 'right')	right	2.492602
6	93	('red', None, None, None, 'right')	right	2.393461
7	105	('green', None, None, 'left', 'forward')	forward	2.335391
8	86	('green', None, None, None, 'left')	left	2.281955
9	119	('green', None, 'forward', None, 'forward')	forward	2.207758
10	63	('green', 'left', None, None, 'forward')	forward	2.196221
11	30	('green', None, 'left', None, 'forward')	forward	2.117873
12	15	('red', 'forward', None, None, 'right')	right	1.388258
13	78	('green', 'left', None, None, 'right')	right	1.35749
14	138	('green', None, None, 'left', 'left')	left	1.23164
15	72	('green', None, None, 'forward', 'right')	right	1.213777
16	115	('green', None, None, 'forward', 'left')	left	1.185467
17	40	('red', None, None, 'left', 'right')	right	0.894611

134	140	('red', 'left', None, None, 'left')	left	-0.2
135	142	('red', None, 'right', None, 'forward')	left	-0.2
136	143	('green', 'right', None, None, 'left')	left	-0.2
137	12	('green', None, None, None, 'left')	forward	-0.244
138	91	('red', None, None, None, 'right')	left	-0.248
139	100	('green', None, None, None, 'forward')	right	-0.32448
140	22	('red', None, None, 'right', 'forward')	left	-0.36
141	27	('red', 'forward', None, None, 'forward')	forward	-0.36
142	19	('red', None, None, None, 'right')	forward	-0.40951
143	43	('red', None, None, None, 'left')	left	-0.488
144	32	('red', None, None, None, 'left')	forward	-0.67232
145	104	('red', None, None, None, 'forward')	left	-0.98559
146	122	('red', None, None, None, 'forward')	forward	-0.9941

Helpful links:

- What happens when gamma is too high: https://classroom.udacity.com/nanodegrees/nd009/parts/0091345409/modules/540405889375460/lessons/5453599987/concepts/6512308800923
- Short-term reward X long term reward (delayed rewards): https://classroom.udacity.com/nanodegrees/nd009/parts/0091345409/modules/540405889375460/lessons/5453599987/concepts/6512308820923
- Epsilon: https://www.udacity.com/course/viewer#!/c-ud728-nd/l-5446820041/m-634899065
- Q-Learning tutorial (good gamma explanation): http://mnemstudio.org/path-finding-q-learning-tutorial.htm
- QLearn Implementation: https://discussions.udacity.com/t/please-someone-clear-up-a-couple-of-points-to-me/45365
- Q-Learning tutorial: https://studywolf.wordpress.com/2012/11/25/reinforcement-learning-q-learning-and-exploration/
- Installing pygame on Anaconda: https://dmenin.wordpress.com/2016/06/14/how-to-install-pygame-using-anaconda/