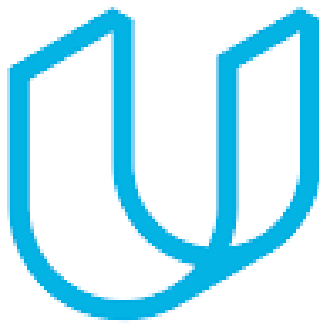


Reinforcement Learning

Train a Smart cab to Drive

Udacity Machine Learning Engineer Nanodegree - Project 7

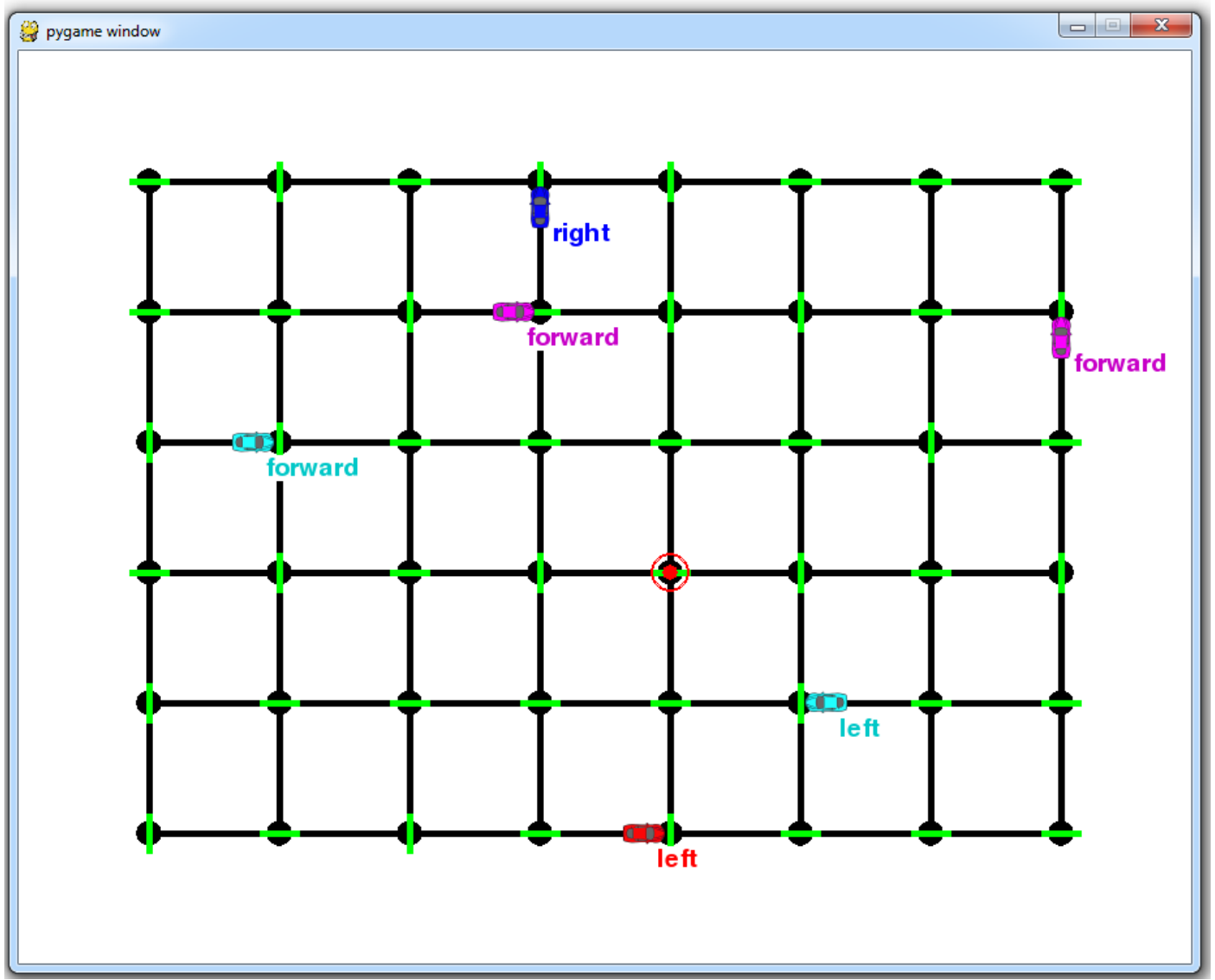
Diego Menin – June 2016



UDACITY

The Environment:

- Red Car is the Learning Agent
- All other cars are Dummy Agents who randomly mover around the map



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. 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:

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

Where:

- $Q(s,a)$ = current reward of the previous state
- α = learning rate
- $R(s,a)$ = reward received for the last move
- γ = 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

Task 4 - Enhance the driving agent

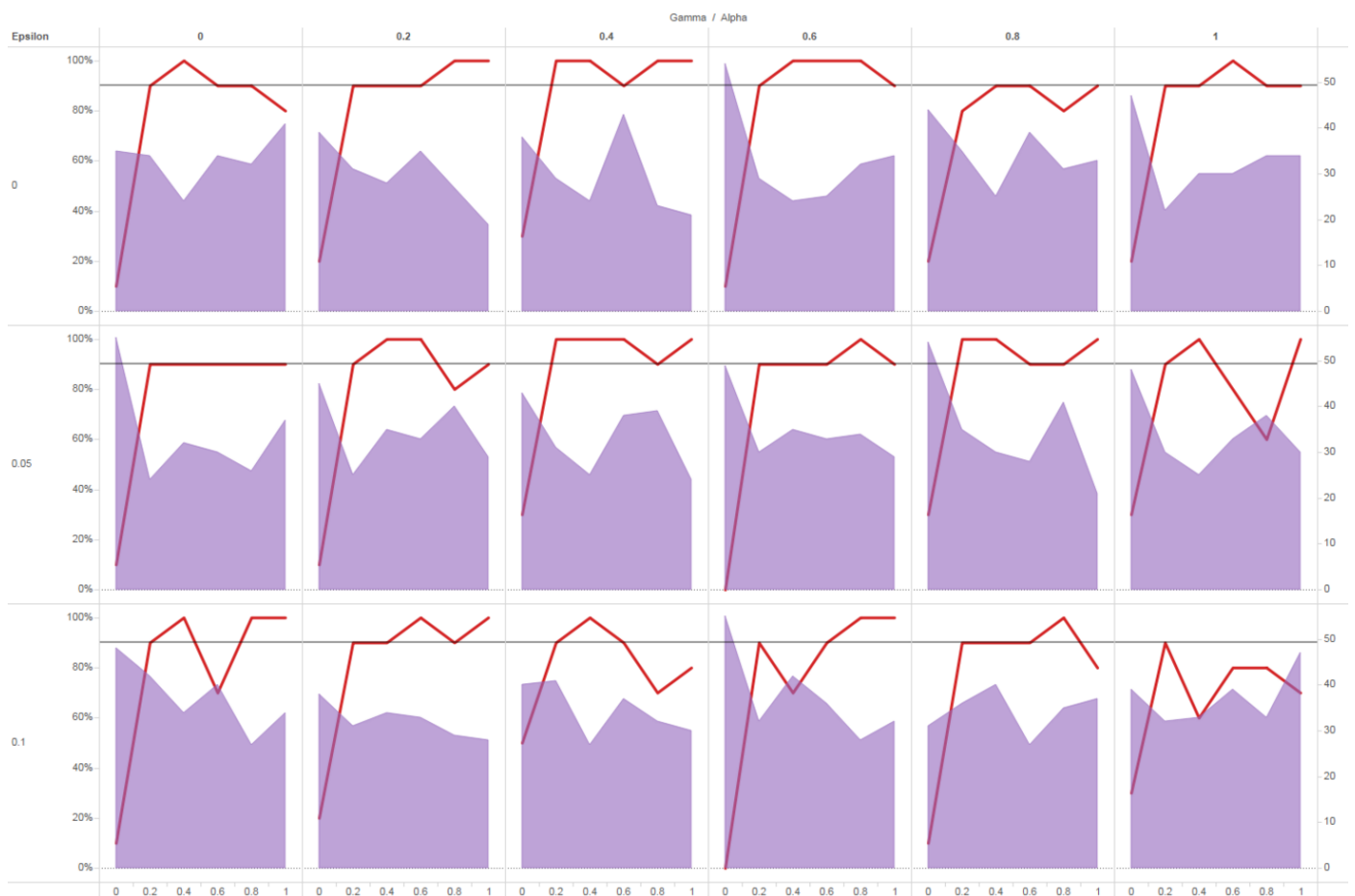
Apply the reinforcement learning techniques you have learnt, and tweak the parameters (e.g. learning rate, discount factor, action selection method, etc.), to improve the performance of your agent. Your goal is to get it to a point so that within 100 trials, the agent is able to learn a feasible policy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

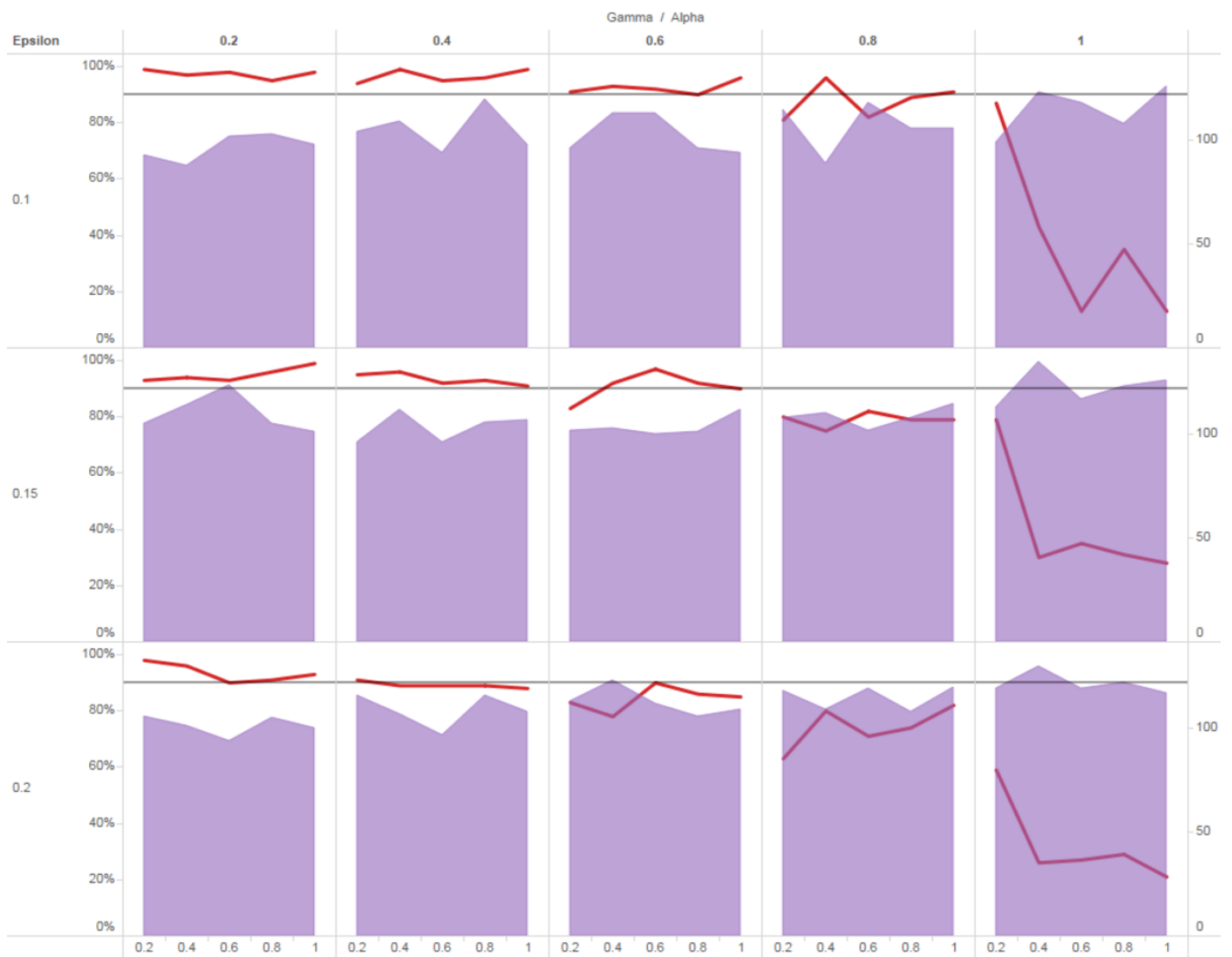
Below is a graph that outputs the result. The red line shows the percentage of the time it got to the destination (so if it is 100%, it got it right 10 out of 10 times) and the purple area, the size of the QTable on the last step of the trial (which is there only as a matter of curiosity). The grey line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

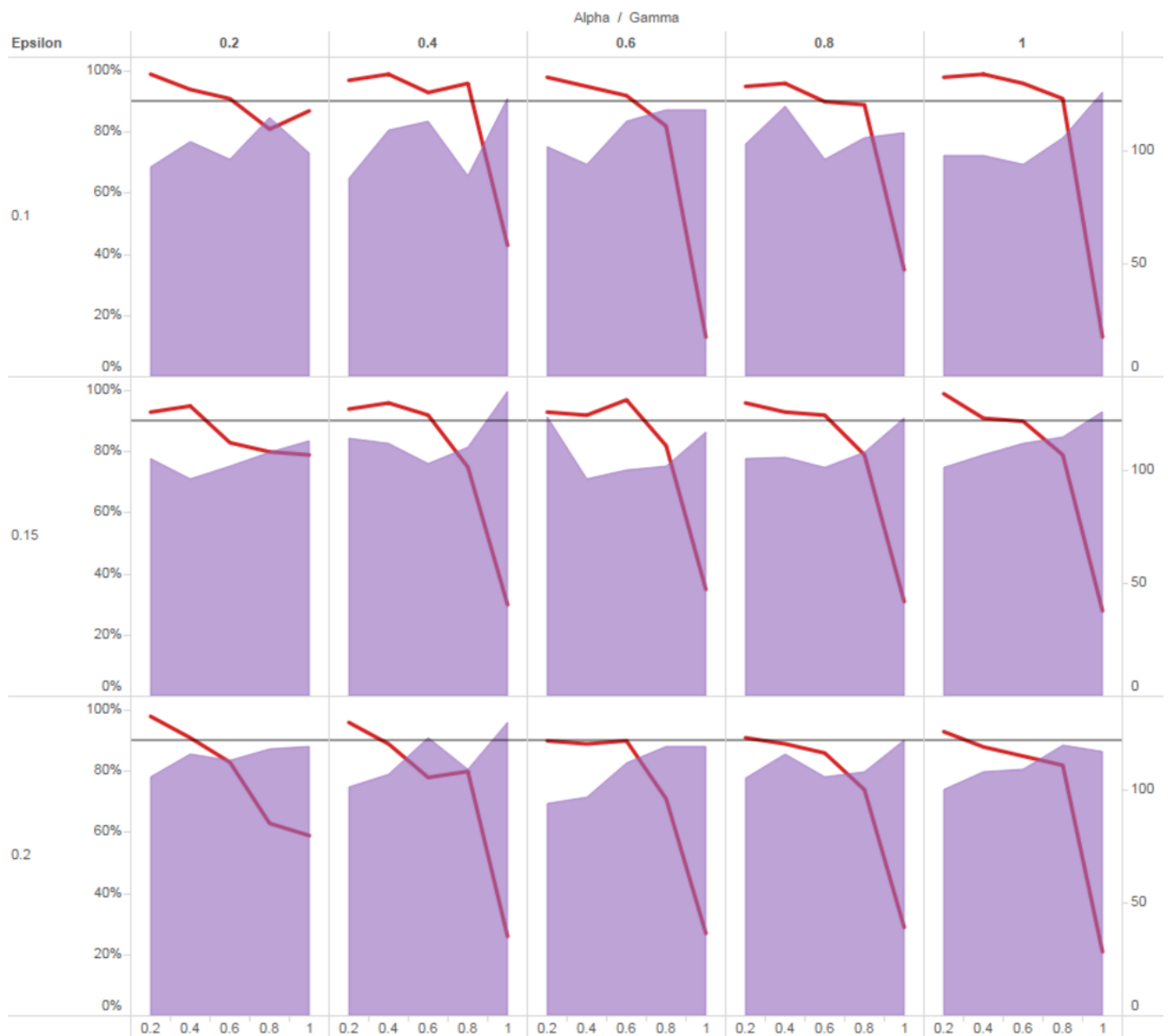
Each data point represents 10 trials with a particular combination of epsilon, alpha and gamma and was executed with an empty QTable; The result is the sum of times the agent got to the destination, so if we look at the first picture on the top left, with epsilon, gamma and alpha all being 0, it got 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, but I will provide a link to a dashboard for the last and most important graph.



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):

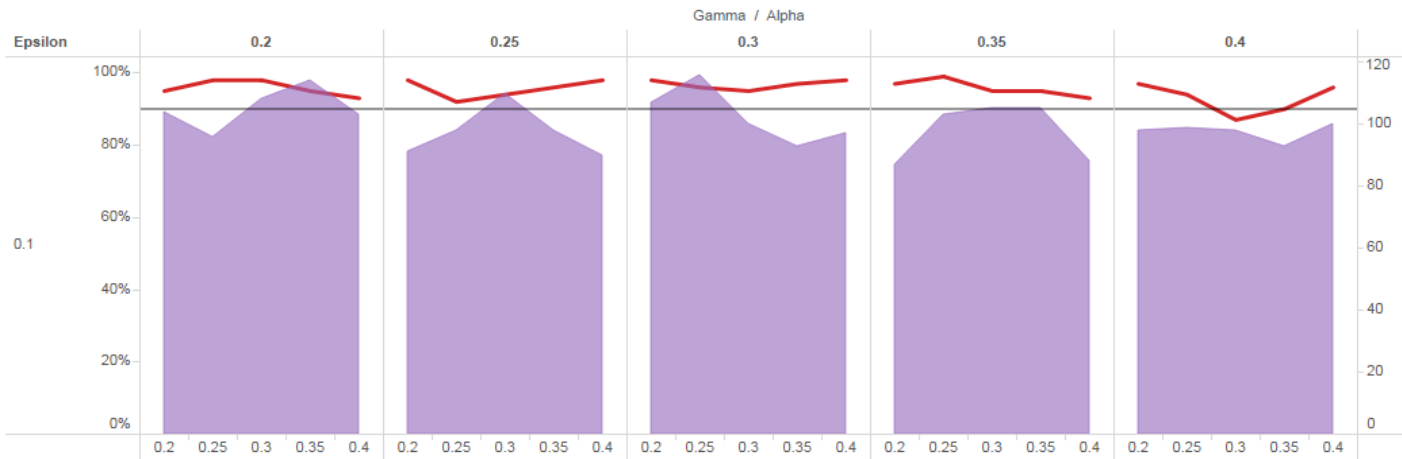




Looking at these plots I can see that the best combination is around epsilon = 0.10 and gamma\alpha between 0.2 and 0.4 where the cab got to the destination 99% of the time.

Epsilon	Alpha	Gamma				
		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%

Just to be more specific, 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. This time, the best combination was alpha = 0.25 and gamma = 0.35:



		Gamma				
Epsilon	Alpha	0.2	0.25	0.3	0.35	0.4
0.1	0.2	95%	98%	98%	97%	97%
	0.25	98%	92%	96%	99%	94%
	0.3	98%	94%	95%	95%	87%
	0.35	95%	96%	97%	95%	90%
	0.4	93%	98%	98%	93%	96%