Project report: smartcab

***Read before reading**: I implemented Q-learning algorithm in agent.py by adding more methods and more variables. To monitor the performance of algorithm, I also added more codes in agent.py and environment.py. The performance monitoring interface will be explained in later description.*

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')`. Don't try to implement the correct strategy! That's exactly what your agent is supposed to learn.

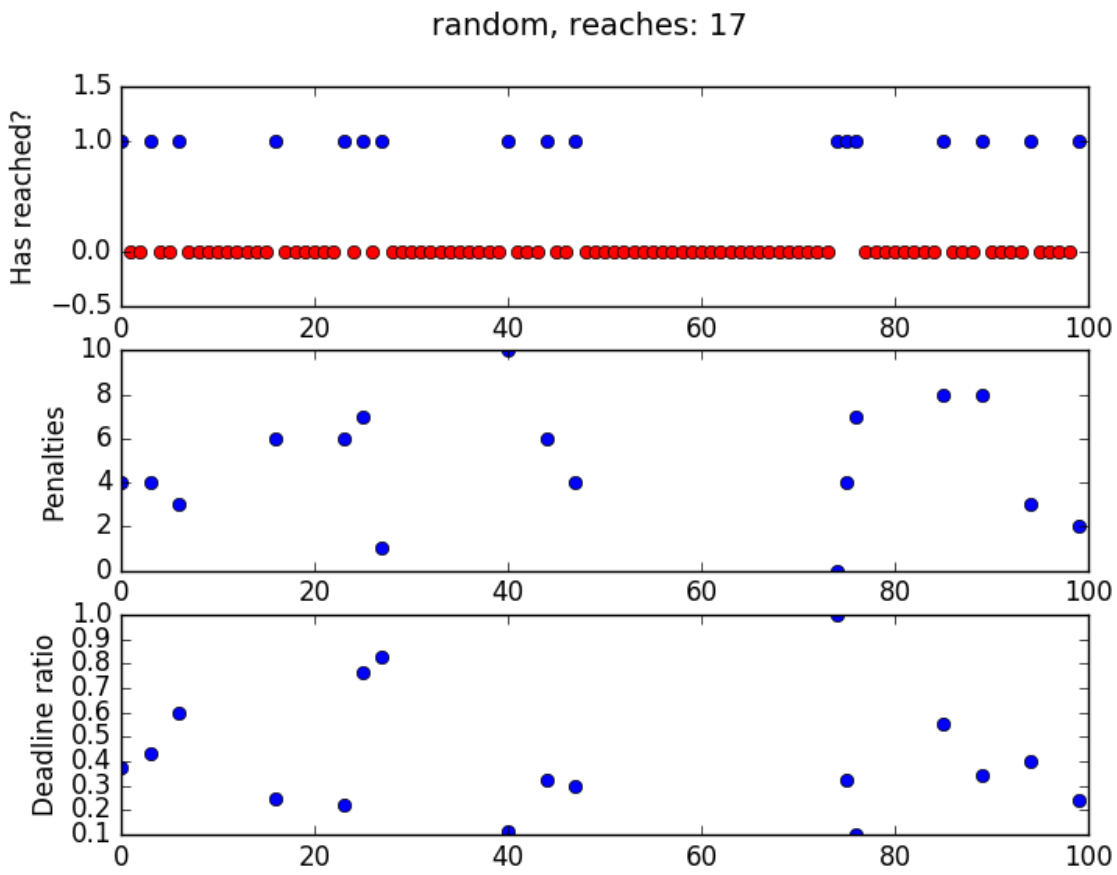
Run this agent within the simulation environment with `enforce_deadline` set to `False` (see `run` function in `agent.py`), and observe how it performs. In this mode, the agent is given unlimited time to reach the destination. The current state, action taken by your agent and reward/penalty earned are shown in the simulator.

In your report, mention what you see in the agent's behavior. Does it eventually make it to the target location?

Answer:

When the actions are totally random among valid actions, the agent takes a long time to reach the destination. This is expected. When the actions are totally random, there is no exploitation but only exploration. So the agent can not learn from its experience and its next step is totally blind. For the agent, the opportunity of reaching the destination is proportional to the number of intersections. That's why it took so long for the agent to reach the destination.

I also tried to set `enforce_deadline` to be `True`, and this what I found:



This figure contains 3 subplots. The first one is showing if the agent reaches the destination in trial n. In this random action case, the agent reached the destination 17 times. The second subplot shows how many times the agent breaks the traffic law in trial n, assuming it reached the destination in that trial. The smaller this value is the better it is. (Since reaching destination is much more important than avoiding penalty in the assumption, it makes little sense to show the penalties when the agent does not reach the destination). And from this subplot, we see the agent gets a lot of penalties (compared to the Q-learning agent which will be discussed later). The third plot shows how fast the agent can reach the destination. Since the initial distance from the agent to the destination is random for each trial, measuring how many steps it takes to reach the destination is not very reasonable. Instead I use (remaining deadline / initial deadline) to measure how fast the agent getting to the destination. And the higher this value is, the faster the agent reaches the destination. And this subplot shows the agent usually moves slow to the destination.

Identify and update state

Identify a set of states that you think are appropriate for modeling the driving agent. The main source of state variables are current inputs, but not all of them may be worth representing. Also, you can choose to explicitly define states, or use some combination (vector) of inputs as an implicit state.

At each time step, process the inputs and update the current state. Run it again (and as often as you need) to observe how the reported state changes through the run.

Justify why you picked these set of states, and how they model the agent and its environment.

Previous Answer:

I picked inputs['light'], inputs['oncoming'], inputs['left'] and the next_waypoint which the planner indicates to the agent to be the state.

I first thought about using the current location, the heading of the agent and the current inputs to be the state. Current location should work well for a fixed destination. But later I realized the destination changes in each trial. So the location is not proper to be an element in the state.

Then according to the US traffic law, inputs['right'] is not necessary to be included in the state. Because if light is red, the agent can do nothing to collide with the right car. And if light is green, the right car can not go before the agent. So this is no chance these two cars will collide due to the agent's fault. So not all of the current inputs are included in the state.

The heading information is not necessary either. I select the next_waypoint to be in the state. The next_waypoint already contains implicitly the information of heading.

So inputs['light'], inputs['oncoming'], inputs['left'] and the next_waypoint model the agent and its environment. For an agent which tries to get to the destination, what it needs are obeying the traffic law, avoiding collision and getting indications from the planner. So this state setup will work for the agent in this environment.

Renewed Answer:

The candidates for the state are: input, location, next_waypoint, deadline. I will express how did I select the state elements which I think are reasonable.

First, we can not select location. For each trial, the destination changes. Recording the location in one trial gives no good to the agent in the next trial.

Second, the inputs are good candidates for a state selection. However, not all of them are necessary. The input['light'] is necessary, because it tells if a car can move or not. Previously, I thought inputs['oncoming'], inputs['left'] were also necessary.

(inputs['right'] is not necessary because it does not induce collision supposing a car from right obeys the traffic law). But according to the currently provided setup in the assignment, there is no penalty for collisions with other cars. If collision is concerned in the future setup, inputs['oncoming'] and inputs['left'] should be included in the state.

Third, the deadline itself is not a good candidate because the original deadline in the beginning of each trial is not constant. Moreover, the deadline has a very long range, which could bring the "curse of dimensionality". So deadline could not be directly added into the state. But deadline does provide some information about if the agent should move to the destination as soon as possible. So in the codes, I define another variable, "close_to_end" and add it into the state. "close_to_end" is 1 if the deadline is smaller than half of the deadline in the beginning of a trial and 0 if larger. This variable does improve the performance of the agent.

Fourth, next_waypoint should be included in the state. It gives indications to the agent so that the agent knows where the destination is.

So inputs['light'], 'close_to_end', and next_waypoint are combined as the state to model the agent and its environment in this assignment. But as I said above, if collision with other cars has been concerned in the future assignment, inputs['left'] and inputs['oncoming'] should also be added into the state.

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 behavior.

What changes do you notice in the agent's behavior?

Attention: *the following answer is using a fixed epsilon. Later the model has been improved by changing epsilon to be a decay value according to the comments.*

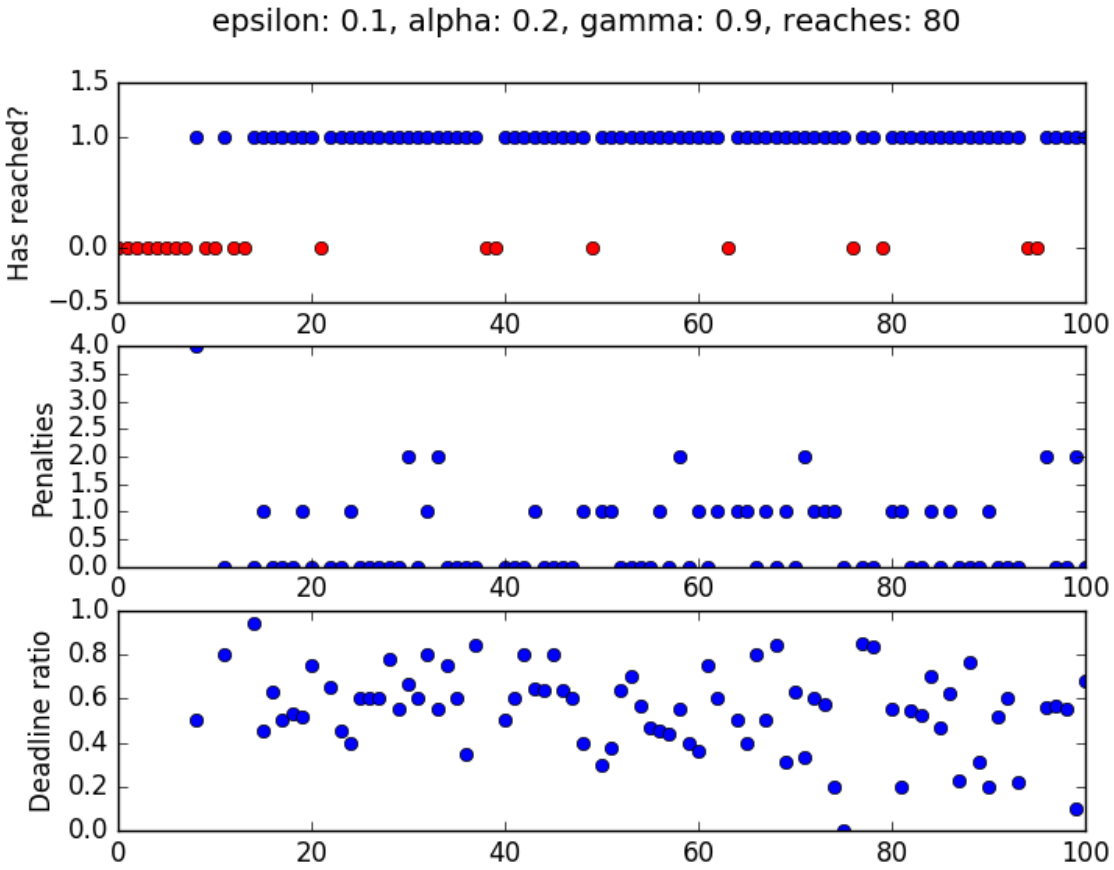
Answer:

The Q-learning codes can be found in agent.py.

I notice by applying the Q-learning algorithm, the agent seems taking random actions at first, but after several trials, it moves more and more "consciously" to the

destination, and the possibility for the agent to reach the destination before deadline is higher and higher.

The following figure shows the performance of the Q-learning with starting parameters: alpha=0.2, gamma=0.9. Compared to the agent with random action, this agent learns from prior experience how to reach the destination, to avoid penalties, and to get the destination fast.



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.

Report what changes you made to your basic implementation of Q-Learning to achieve the final version of the agent. How well does it perform?

Previous Answer:

I changed three parameters to optimize the Q-learning model: the trade-off value between exploration and exploitation--epsilon, the learning rate--alpha, and the discount

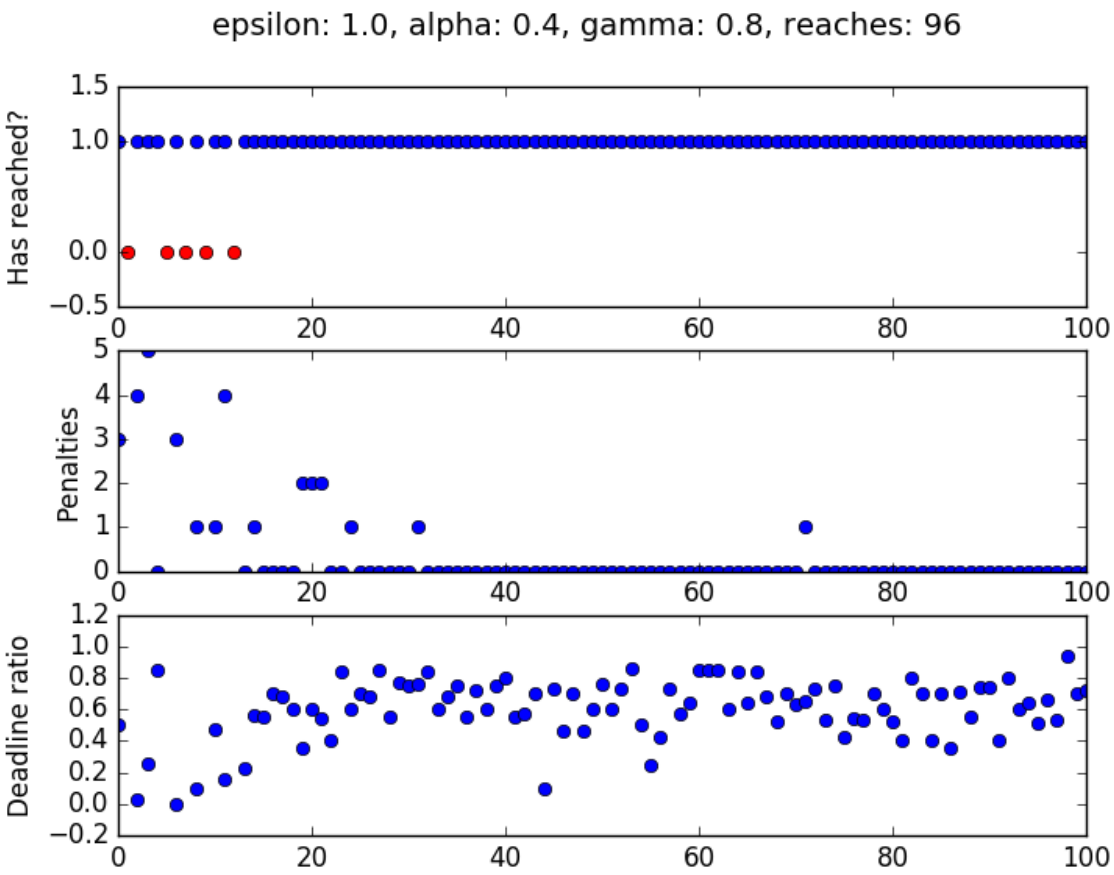
factor--gamma. I tried about 40 combinations of these values to see how the algorithm performs. These performance figures could be found in the directory "plots" and the title of the each figure means how these three parameters are combined.

Renewed answer:

First, I changed epsilon from a fixed value to a decaying value. And the decay speed is 1 / (# of trials). So in the beginning of training, the agent has a big chance to explore. But later, as it learns more and more about the environment, it tends to exploitation. This is a big improvement. I was thinking about using dynamic epsilon before doing the first submission. But I'm not very clear how to apply my thought. Thanks to the comments. Now I know better how to balance exploration and exploitation of the agent.

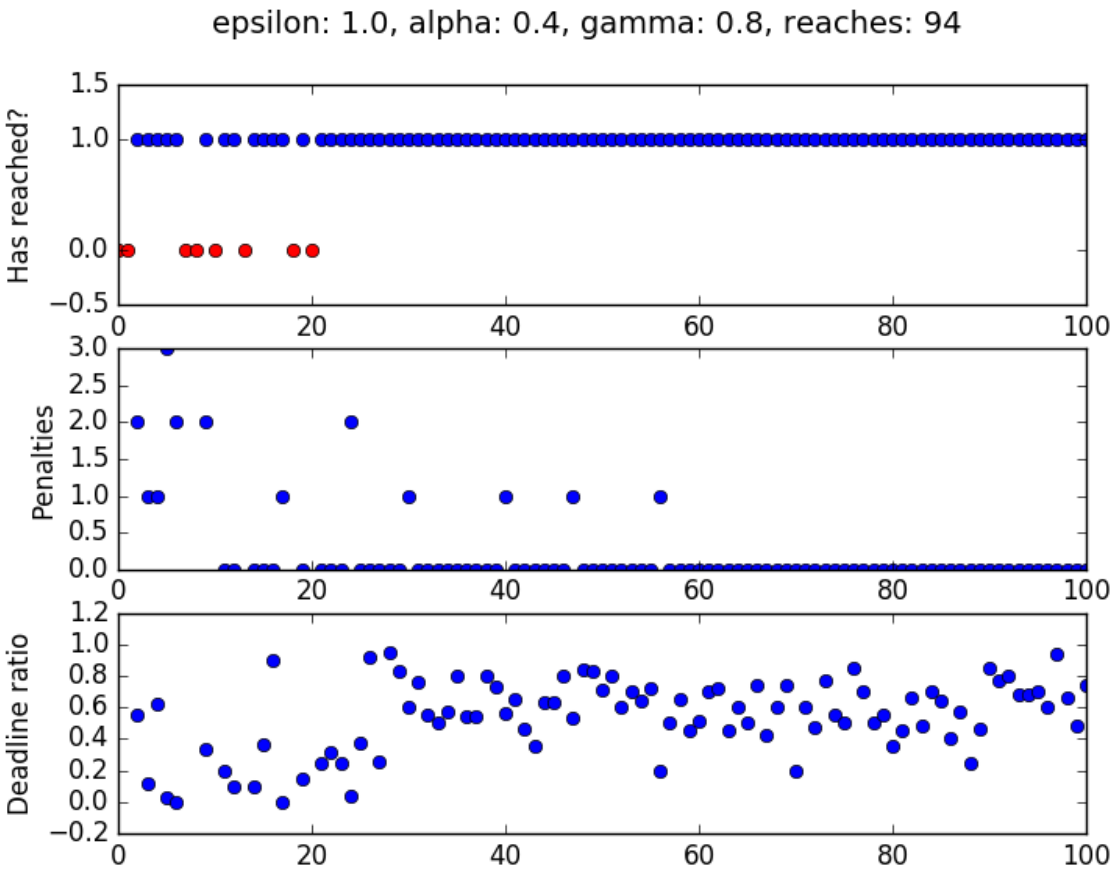
Second, I tried a series of combinations of gamma and alpha to get the best performance. Those comparison figures are in the directory "plots". And I found when alpha = 0.4 and gamma = 0.8, my Q-learning algorithm gives the best result.

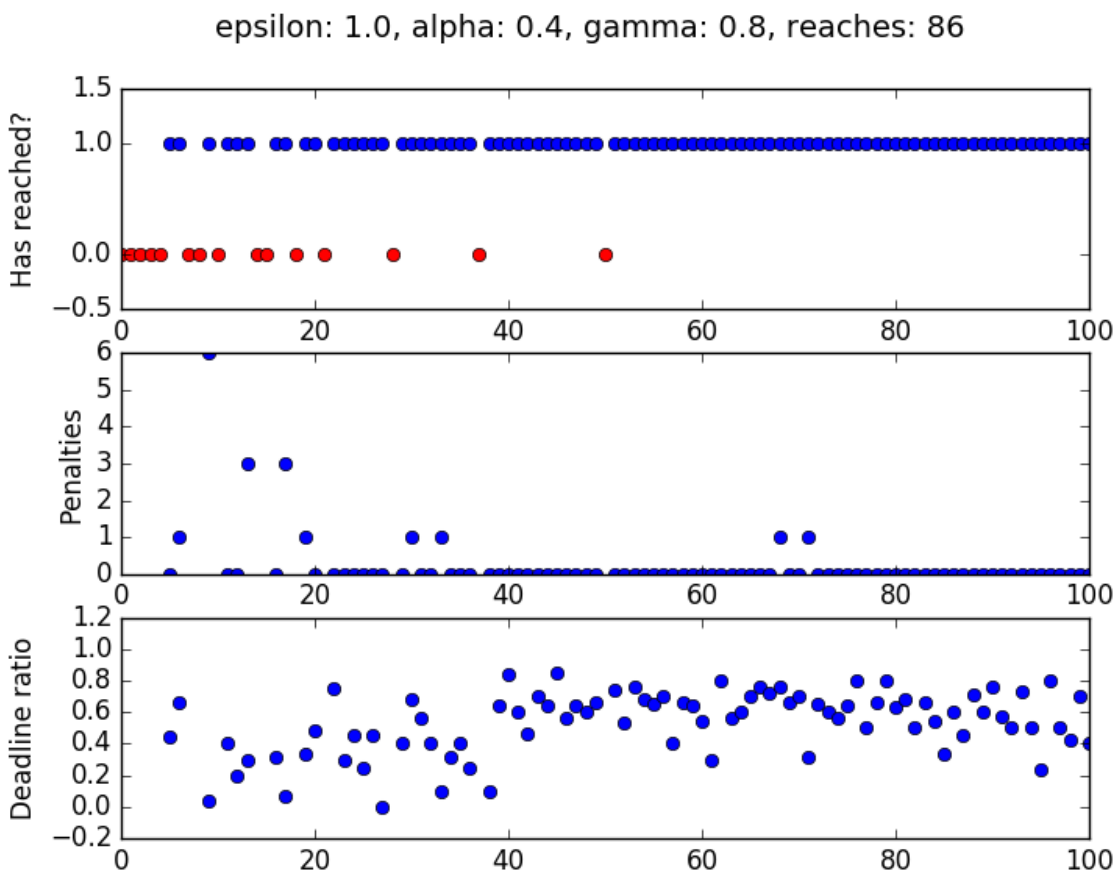
The following figure shows how it performs.

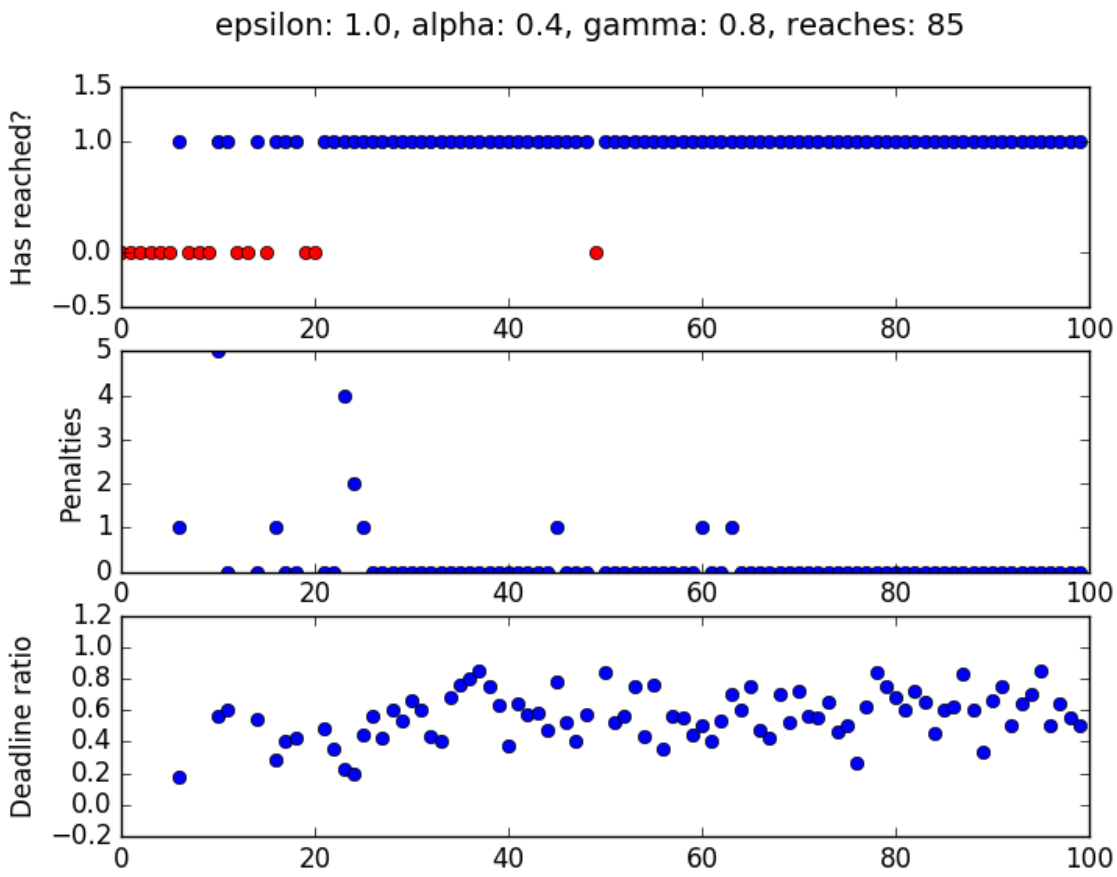


From this figure we see the agent learns pretty fast. After around 20 runs, it has already learned how to reach the destination in time. And we also see the penalties value is decreasing and converging to 0 after the first 40 trials. After 80 trials, it never gets any penalty. Moreover, the deadline ratio is increasing and converging. The ratio keeps high ~0.6 for the last 40 trials. It means the agent does not waste time in making circles.

What's more, the robustness of this agent is pretty good. I tested it several times and the performance is pretty consistent. (Refer to following figures)


epsilon: 1.0, alpha: 0.4, gamma: 0.8, reaches: 94
Has reached?
Penalties
Deadline ratio


epsilon: 1.0, alpha: 0.4, gamma: 0.8, reaches: 86
Has reached?
Penalties
Deadline ratio



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?

Previous Answer:

There are a lot of combinations of the three parameters mentioned above. I will just post a few of them to show how I get the optimal policy.

For epsilon = 0.1, alpha = 0.2 and gamma = 0.6, the agent can be trained to reach the destination in allotted time. However, it gets too many penalties in its trip. This agent needs to do better to decrease the penalties.

For epsilon = 0.02, alpha = 0. 9, gamma = 0.9, the agent gets less penalties compared to the above figure. However, it still has too many penalties.

The optimized agent or the agent close to the optimal may be this one: epsilon = 0.02, alpha = 0. 3, gamma = 0.9. Although it takes more trials to train, it has much less penalties. I also tested this models several times (figure_35, figure_41, and figure_42). This agent is pretty robust. It has a very big possibility that after the training, the agent will reach the destination in time and will never get penalty.

Renewed answer:

I'm not very sure if the agent gets the optimal policy. From my tests mentioned above, as the trial number increases, the performance of the agent does converge to a high level. For the last 40 trials, it always reaches the destination in time and the deadline ratio, which is a measure of effectiveness of route selection, is higher and robust. From my observation of the agent's movement, after a few tens of trials' training, the agent always selects the shorted route to the destination. It does not make circles. That does not necessarily mean the total time is minimum because the light can stop the agent. But since the light is pretty random, one can not predict the light in the next intersection. In such a case, taking the shorted route is the best solution. And after 20 trials, it never gets any penalty. I could not say this is the best agent among all possible agents, but I think it's close to the best.