- **Project – Training a smart cab to drive - Manoj Ramachandran – Monday, April 25, 2016**

  Last Updated – Friday, May 27, 2016 4:30 p.m.

  These are the references I looked up to understand about Q-Learning implementation.

- #ref: https://studywolf.wordpress.com/2012/11/25/reinforcement-learning-q-learning-and-exploration/
- #https://github.com/e-dorigatti/tictactoe
- #https://gist.github.com/fheisler/430e70fa249ba30e707f
-

Train a Smartcab to Drive

Implement a basic driving agent

The LearningAgent class runs in simulator. It picks up a random action from the available actions and computes reward based on a particular action.

The run method changes the simulation run by these three parameters:

- deadline (enforce_deadline),
- delay in updating simulation run (updated_delay) and
- number of trials (n_trials)

Identify and update state

What I understand from the meaning of the 'state' is that the state is a unique combination of attributes that can help locate (or identify) where the cab is at any given point of time and helps it to make the next step.

In my own driving, say I am sitting at a stop sign at an intersection, my current state with my other cars at the intersection determines the actions I could take depend upon. The light at the intersection, whether I want to turn left or right and upcoming traffic state determines my decision what to do next. So, I will use three for determining states. They are

a. status of the light - I can't move if it is red in first place.
b. action of the 'oncoming traffic' whether they are turning right, or going forward
c. and traffic on the left – whether they are going straight and I want to make right. I have to wait.

After much though, the traffic on the right doesn't' affect much since most of their actions happen when it is red and they don't determine whether I could turn right.

Thoughts:

Q-learning with discretised states and actions scale poorly. As the number of state and action variables increase, the size of the table used to store Q-values grows exponentially.
http://users.cecs.anu.edu.au/~rsl/rsl_papers/99ai.kambara.pdf

with two light states (red and green), three oncoming traffic states (straight, right or left), and three states for left traffic (straight, right or left) and where we want to go to the next step, we already have 2 times 3 times 3 times 3 totaling 54 states and will grow exponentially as we add more states.

Increasing states increases memory and time to compute the next best possible action. For speeding up dynamic programming and decision, we need to be able ignore those variables that are irrelevant to the current decision. This is called safe state abstraction (https://www.aaai.org/Papers/AAAI/2002/AAAI02-019.pdf). If we included variables like **deadline and destination** with very specific but numerous possibility of discrete values, we will end up with numerous state spaces which are not only possible to easily trace but becoming useless in providing feedback to update Q-values. The resulting Q-matrix then becomes very sparse and not reliable to able to consistently learn or it might take a less than reasonable time period to learn that we can use for making decisions.

The main reason for not including **destination** is that the destination could be very well change for each iteration. We are not trying to teach a cab to go to one destination in the entire simulation. We are enabling to drive it safely at each of the cab's decomposed state towards the prescribed destination and therefore influence of the destination variable is indirectly affected by the waypoint that we have included.

Implement Q-Learning

I have created a QLearningAgent using the same class template of LearningAgent which instantiates a QLearningPlayerObject with three parameters

We are updating the previous state-action using the equation (corrected wth feedback):

Q(s,a) += alpha* (rewards(s,a) + **gamma*(max(Q(s',a'))** - Q(s,a)) –

I hope this matches with what I see in https://en.wikipedia.org/wiki/Q-learning - where t is the time period

$$Q_{t+1}(s_t, a_t) \leftarrow \underbrace{Q_t(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha_t(s_t, a_t)}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_a Q_t(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q_t(s_t, a_t)}_{\text{old value}} \right)$$

Where

- s is the previous state
- a is the previous action
- s' is the current state
- a' is the current action

there are three main parameters of interest

- epsilon – determines how much to search randomly or exploration. As a default, we say there is a 20% chance that the agent will choose to explore. We want this to be low since we just don't want the agent to always explore but exploit Q values it is updating on every move
- alpha – learning rate – tells how much of the newly acquired information overrides the old information. setting alpha to zero turns of Q-learning and agent will not learn anything. Setting to 1 would make the agent completely forget the old information and uses only the new information.
- gamma – discount factor – determines the importance of future rewards. it affects the importance of long term or short term rewards

the implementation stores, updates, fetches the q value for state action pair. For choosing the best action, we use max of Q values and then index function to pull the best action out. If there is a tie in fetching the max Q value for state action pair combination, then randomness is applied to make the determination.

Between trials, last_move and last_state are reset in the start_game function.

Changes in behavior explained:

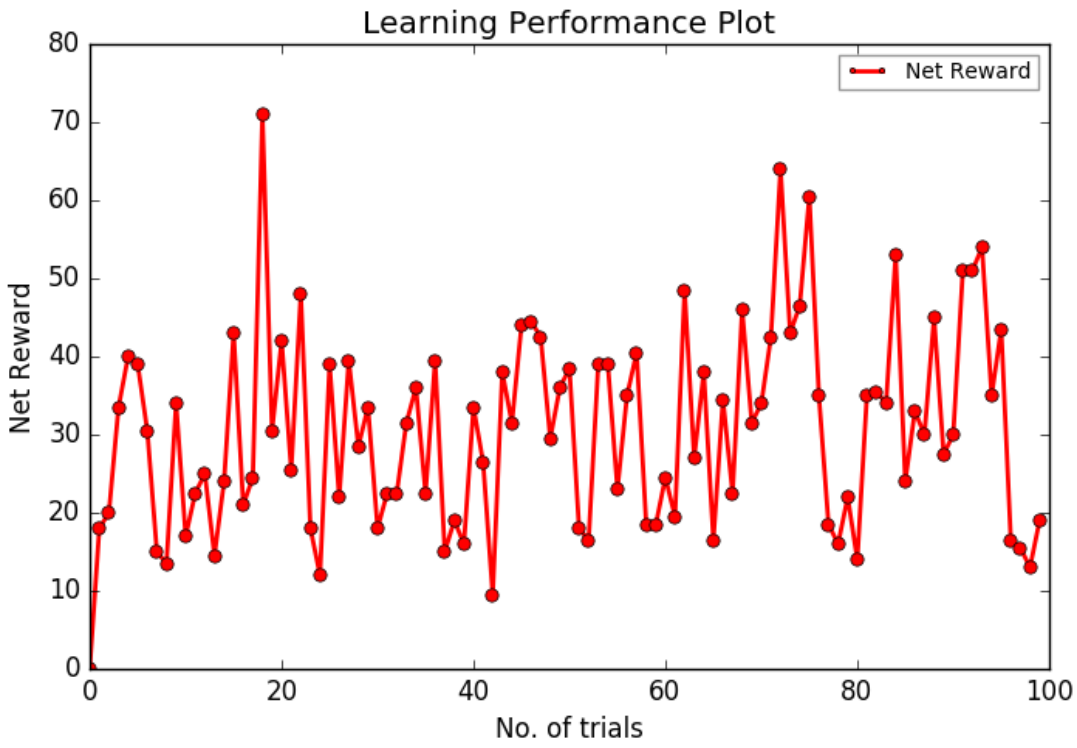
In simple Learning Agent, we did a random action at every state while guided by the planner to reach the destination. However, because of Q-learning, the scored our past action by the reward we receive at every step. This builds a very robust model where we let epsilon, alpha and gamma parameter values play a great role in taking action at every state.

In other words, there is no 'learning' in the LearningAgent without Q-learning. Though the states are updated, we don't look back and evaluate the effect at t+1 time period of every single move the cab makes at time period (t) .Also, the agent is not working towards any optimal policy. The performance of the cab at any instance is completely independent of the actions it took previously.

Agent with q-learning (learning rate = 0.8) reached destination 83% of the time (83 out of 100) and 80% of the time in the last 10 runs (8/10). Agent without q-learning reaches destination only 29% of the time and this ratio could vary very much. Basically, there is no learning and therefore no expected performance improvement in an agent without any learning. Q-Learning however helps remember the state and action we took to maximize reward and helps the cab to make the best decision when it arrives to the same state next time – thus becoming an intelligence agent.

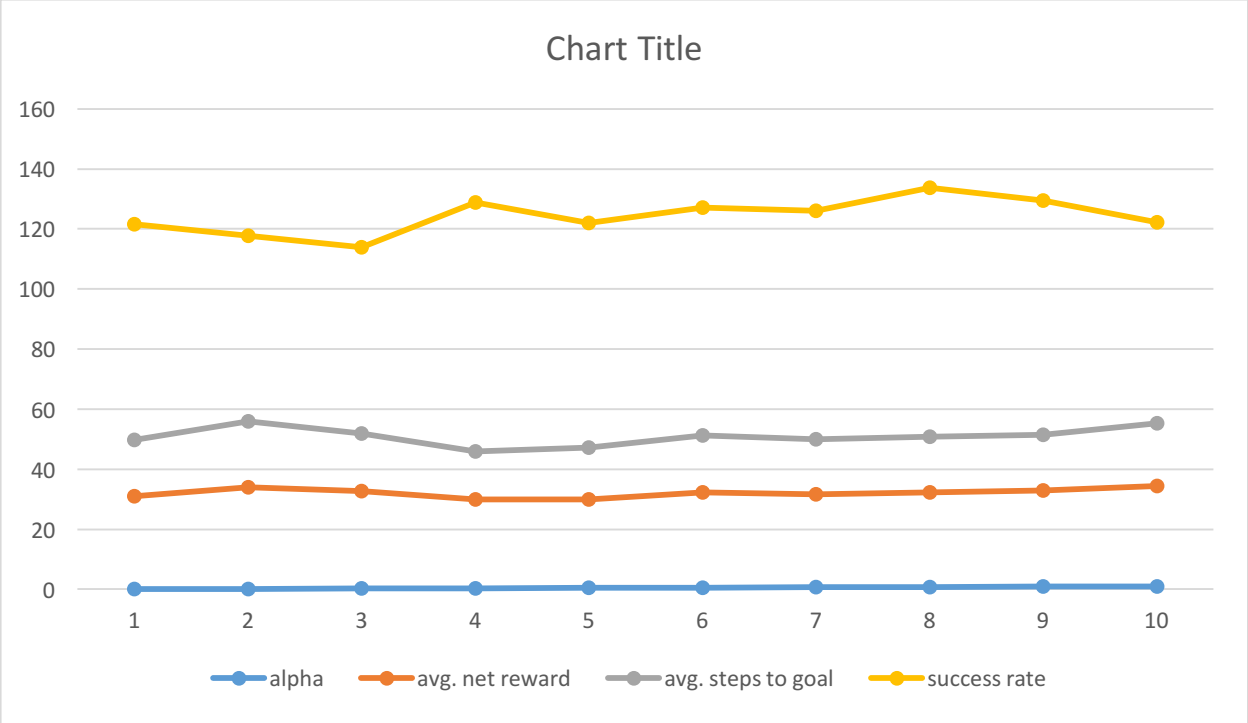
Enhancing the Agent:

1. **Agent learns a feasible policy within 100 trials:** Yes. I could see the Agent consistently reaches the destination. The net reward stays positive.



2. **Improvements reported** -
   a. We want the Agent to explore less at random as it gains solid Q-values for states. So, we want the epsilon to be dynamically decreasing as the iterations increase. See line 91
   b. I have attempted to iterate the simulation run with different learning rates [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.9,1.0], I had to create two global arrays that helped to see the variation of Learning Rate vs Avg. No. of Steps toward Goal and Learning Rate vs Avg/ Net Reward. I got some inputs from this paper - http://www.jmlr.org/papers/volume5/evendar03a/evendar03a.pdf. I am leaning towards picking the policy that reaches the goal in less number of steps while increasing the avg. net reward.
   c. Please see the table value and the graph below. Learning rate of 0.8 gives the best success rate with higher net reward. Though you get the same success rate with alpha = 0.4, I tend to go with one that yields higher avg. net reward given the difference in the avg. steps to goal between those learning rates (0.4, and 0.8) is relatively small.

alpha	avg. net reward	avg. steps to goal	success rate
0.1	30.86	18.72	72
0.2	33.695	21.95	62
0.3	32.395	19.25	62
0.4	29.525	15.95	83
0.5	29.525	17.07	75
0.6	31.77	18.78	76
0.7	30.905	18.47	76
0.8	31.515	18.43	83
0.9	31.925	18.66	78
1	33.375	20.86	67



d. Based on the above parameter tuning, I found LearningRate = 0.8 maximizes the reward though the number of steps is relatively higher when compared to learning rate of 0.4.

e. So, running with the optimal LearningReate = 0.8, the Agent consistently reaches the destination (only two misses in the last 10 runs). I wish it would have been easier to compute different metrics easily without having to manually change the parameters. I just had to copy the outcome in excel, do a filter and count the number of records for success that said “Primary Agent has reached the destination”

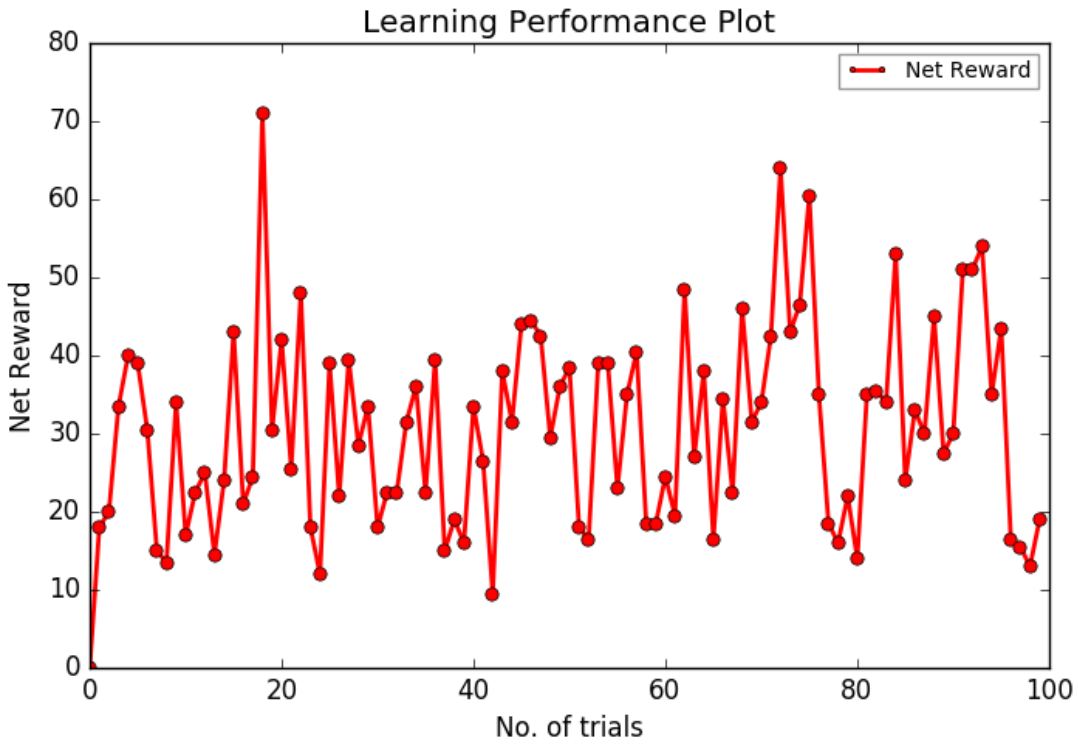
From the last run (after I corrected the simulation run to 100), I got the following which tells me that only three runs in the last 10 did not reach destination within the deadline. I did this many times and am getting the same or lesser number of misses.

Simulator.run(): Trial 89
Environment.reset(): Trial set up with start = (8, 5), destination = (5, 3), deadline = 25
Environment.act(): Primary agent has reached destination!
Simulator.run(): Trial 90
Environment.reset(): Trial set up with start = (8, 5), destination = (4, 3), deadline = 30
Environment.act(): Primary agent has reached destination!

```
Simulator.run(): Trial 91
Environment.reset(): Trial set up with start = (5, 5), destination = (1, 1), deadline = 40
Environment.reset(): Primary agent could not reach destination within deadline!
Simulator.run(): Trial 92
Environment.reset(): Trial set up with start = (4, 2), destination = (1, 1), deadline = 20
Environment.act(): Primary agent has reached destination!
Simulator.run(): Trial 93
Environment.reset(): Trial set up with start = (5, 1), destination = (5, 6), deadline = 25
Environment.act(): Primary agent has reached destination!
Simulator.run(): Trial 94
Environment.reset(): Trial set up with start = (6, 5), destination = (7, 1), deadline = 25
Environment.act(): Primary agent has reached destination!
Simulator.run(): Trial 95
Environment.reset(): Trial set up with start = (5, 6), destination = (2, 3), deadline = 30
Environment.act(): Primary agent has reached destination!
Simulator.run(): Trial 96
Environment.reset(): Trial set up with start = (4, 4), destination = (2, 2), deadline = 20
Environment.act(): Primary agent has reached destination!
Simulator.run(): Trial 97
Environment.reset(): Trial set up with start = (3, 1), destination = (1, 4), deadline = 25
Environment.act(): Primary agent has reached destination!
Simulator.run(): Trial 98
Environment.reset(): Trial set up with start = (6, 2), destination = (1, 3), deadline = 30
Environment.reset(): Primary agent could not reach destination within deadline!
Simulator.run(): Trial 99
Environment.reset(): Trial set up with start = (4, 2), destination = (2, 6), deadline = 30
Environment.reset(): Primary agent could not reach destination within deadline!
```

3. Final Agent Performance:

    a. An optimal policy – for a cab driving automatically, the optimal policy will be to reach the destination within the deadline and at the same time, most importantly, not making any wrong moves and getting into an accident.
    b. I can see the cab is still making "wrong move" while at the same time has fully learned to arrive at the destination most of the time which could be understood that it is approaching at the optimal policy but not just quiet. A longer training period could help it achieve the optimal policy. However, I do infer from the outputs that the number of "wrong moves" in a trail and the frequency of occurrences in trials considerably comes down as the number of iteration increases. 23 (23% of) trials made at least one wrong move. We want the cab to learn not to make any wrong move at all.
    c. With the optimal learning rate of 0.8, the net reward stays at 31.5 and most of the time avg . number of steps toward goal is around 18.5
    d. Also, the graph drawn below (the code to produce graph is commented out) clearly shows the net reward stays positive and in fact is slowly moving upwards which means the agent is trying to make decisions that maximizes the reward.



    e. .