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

  Last Updated – Friday, May 20, 2016

  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.

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.

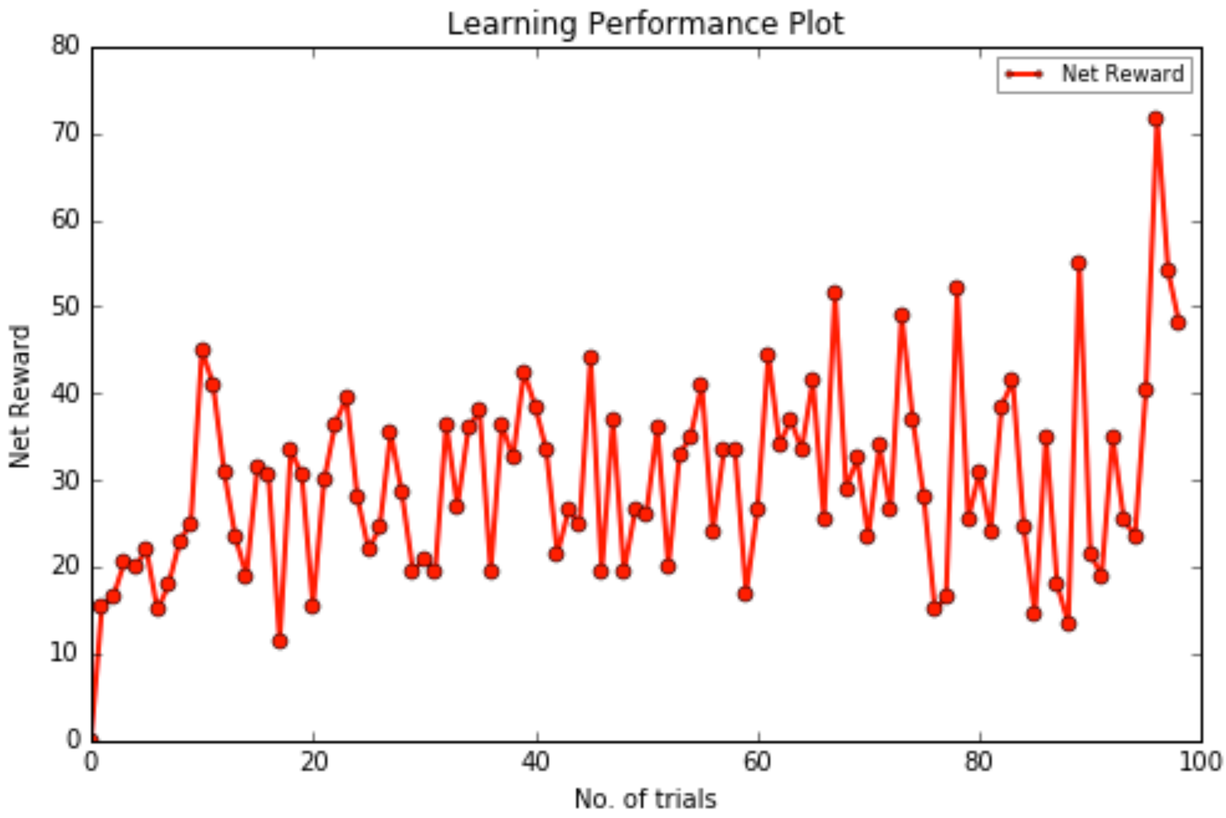
Between trials, last_move and last_state are reset in the start_game function.

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.

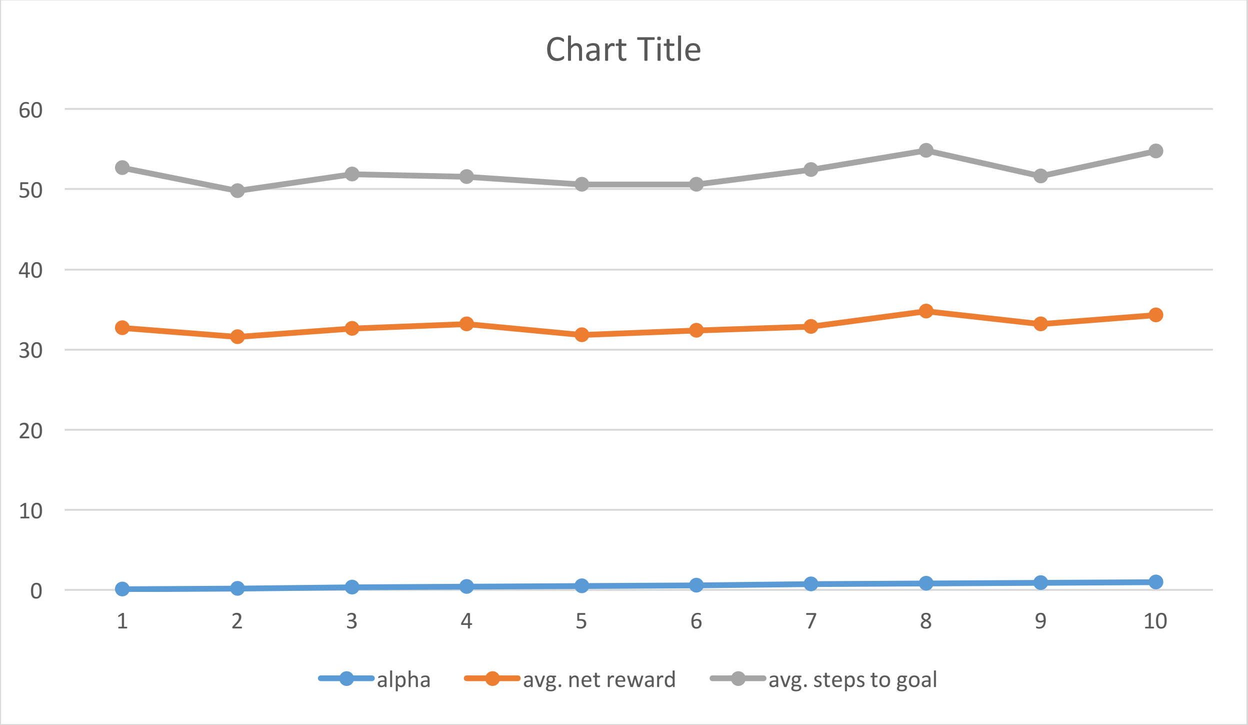
Enhancing the Agent:

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. Based on the above parameter tuning, I found LearningRate = 0.8 maximizes the reward though the number of steps is relatively high. Though it appears from the graph that it took the most number of steps, the difference between the min and the max of 'avg. number of steps' doesn't vary much. So, the best parameter for 'learning rate' seems to be 0.8.

d. So, running with LearningReate = 0.8, the Agent consistently reaches the destination. Other than the print statement that says "Agent has reached the destination", I am not able to find any attribute of the agent that tells whether the agent has reached the destination or not and hence unable to create a "precision" metric.

3. Final Agent Performance:
   a. Referring http://artint.info/html/ArtInt_267.html, we can tell that the the optimal policy is the one that maximizes the reward with minimum number of steps. So, if we draw a graph of Accumulated Reward vs Total No. of Steps the Agent takes.

Then it is fair to say the “one policy dominates the other if its plot is consistently about the other” which will then become the optimal policy.

b. I can see from the graph I drew at the end of 100 trials that the agent tends to keep the net-rewad as positive consistently indicating me that it has learned and very close to the stated optimal policy

c. For some reason that I am not certain, the net reward hasn’t gone below zero at any time which makes me wonder whether it was a loose implementation of the reward rule or whether I am doing something wrong. If there was a zero-crossing as it described on the referenced site, then it will tell us how long the algorithm or policy takes to recoup the cost of learning.