* Project – Training a smart cab to drive - Manoj Ramachandran – Monday, April 25, 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 a intersection, my current state with my other cars at the intersection, whether I can make a left,right or drive straight determines the following action I could take.

The agent updates the state using the statement – ‘self.state = ’ statement. However, in the LearningAgent class, I don’t see the basic Agent taking ‘state’ into the consideration to produce the next action. The original LearningAgent does random action and is not influenced by state.

QLearningAgent, which I extended, does take the state into consideration.

So, naturally I took the next way point the planner wants me to go to, and the direction to the front, left and right as those that define my state with which I make a decision to take the next step.

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:

Q(s,a) += alpha ( rewards(s,a) + gammamax(Q(s’) - Q(s,) )

Where

* s is the previous state
* a is the previous action
* s’ is the current state
* alpha is the discount factor

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 acton, 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:

Between the original LearningAgent and QLearningAgent, we expect the random decisions decrease by 80% (by setting the epsilon to be at 0.2). The system now takes a methodical approach to update its knowledge on what action to take given a particular state to go towards destination.

Implement a basic driving agent

| CRITERIA | MEETS SPECIFICATIONS |
| --- | --- |
| Agent accepts inputs | Student is able to implement the desired interface to the agent that accepts specified inputs. |
| Produces a valid output | The driving agent produces a valid output (one of None, ‘forward’, ‘left’, ‘right’) in response to the inputs. |
| Runs in simulator | The driving agent runs in the simulator without errors. Rewards and penalties do not matter - it’s okay for the agent to make mistakes. |

Identify and update state

| CRITERIA | MEETS SPECIFICATIONS |
| --- | --- |
| Reasonable states identified | Student has identified states that model the driving agent and environment, along with a sound justification. |
| Agent updates state | The driving agent updates its state when running, based on current input. The exact state does not matter, and need not be correlated with inputs, but it should change during a run. |

Implement Q-Learning

| CRITERIA | MEETS SPECIFICATIONS |
| --- | --- |
| Agent updates Q-values | The driving agent updates a table/mapping of Q-values correctly,  implementing the Q-Learning algorithm. |
| Picks the best action | Given the current set of Q-values for a state, it picks the  best available action. |
| Changes in behavior explained | Student has reported the changes in behavior observed, and  provided a reasonable explanation for them. |

Enhance the driving agent

| CRITERIA | MEETS SPECIFICATIONS |
| --- | --- |
| Agent learns a feasible policy within 100 trials | The driving agent is able to consistently reach the destination within allotted time, with net reward remaining positive. |
| Improvements reported | Specific improvements made by the student beyond the basic Q-Learning implementation have been reported, including at least one parameter that was tuned along with the values tested. The corresponding results for each value are also reported. |
| Final agent performance discussed | A description is provided of what an ideal or optimal policy would be. The performance of the final driving agent is discussed and compared to how close it is to learning the stated optimal policy. |