Implement a Basic Driving Agent

QUESTION: *Observe what you see with the agent's behavior as it takes random actions. Does the* ***smartcab*** *eventually make it to the destination? Are there any other interesting observations to note?*

By choosing random action from 4 viable actions (None, left, right, forward), the agent moved around in the environment. Since agent didn't take into account any of the traffic rules, it received negative reward whenever the random action resulted in traffic violation reported by environment. Also, agent didn't follow any strategy to reach destination so random action at each intersection resulted agent moving randomly in the environment ignoring goal to reach the target.

By NOT enforcing simulation deadline agent many times hit hard time limit (100 iterations) and aborting simulator run. Sometimes agent did make to destination but way past deadline but again there is no pattern here.

It is now clear that agent needs a strategy or policy to reach the destination guiding to choose right action at each interaction. This policy will need to take into account on how to reach the destination optimally given the traffic laws (avoiding collisions) , learning reward influence to take proper action at each intersection.

Inform the Driving Agent

QUESTION: *What states have you identified that are appropriate for modeling the* ***smartcab*** *and environment? Why do you believe each of these states to be appropriate for this problem?*

OPTIONAL: *How many states in total exist for the* ***smartcab*** *in this environment? Does this number seem reasonable given that the goal of Q-Learning is to learn and make informed decisions about each state? Why or why not?*

Within the given simulation, agent is given route direction guidance as next_waypoint (forward, left or right) which serves as direction

towards reaching the target , agent can sense traffic light signal (‘red or ‘green’) and also traffic from different directions (oncoming, left, right) as inputs. Also, deadline is given which serves as the time limit to reach the destination. Agent is allowed to take 4 possible set of actions to move {None, Left, right, forward} at each intersection. For any action taken by Agent, Simulator asses the taken action to be compliant with traffic laws or rules. If taken action found to be in violation of simulation rules, Simulator penalizes with negative reward. A positive reward is given for reaching towards the destination without any violations and final reward for reaching goal.

This is classic reinforcement problem where agent is given set of inputs, possible actions with a target goal (rewards and penalties for various actions). Our goal is to learn optimum policy (series of actions) to reach destination within the given deadline. To learn the optimum policy, first we have to analyze the state of each possible inputs, possible actions and outcomes.

Here agent state can be captured based on following possible inputs at any intersection. I have chose to omit Simulation’s ’deadline’ part of the input state which explodes state dimension considerably making it hard to learn or explore the states matching with time limit. For example, with deadline taking into account, agent heading ‘forward’ with ‘red’ light turned on, with ‘none’ oncoming traffic, ‘none’ left, ‘none’ right traffic can be with with variable time limit. Making our policy state modeling really expensive and really doesn’t benefit the given problem. By decoupling deadline limit from environment, in given grid agent states can be modeled using following inputs

Next_waypoint = {forward, left, right}

Light = {green, red}

Oncoming Traffic = {None, forward, left, right}

Right Traffic = { None, forward, left, right}

Left Traffic = {None, forward, left, right}

Although we can eliminate some of the values I have choose to model states with all possible inputs in mind. In this case,we can state uniquely represent any combination of above possible values.

For example, at one intersection at any some specific time, we may have Agent heading ‘forward’ (based on router planner), with a ’red’ light, ‘none’ oncoming traffic, ‘forward’ right traffic and ‘none’ left traffic. So we can represent this state as combination of these values (Next_Waypoint + Light + Oncoming + Right + Left) : ‘ForwardRedNoneForwardNone’ or more simply ‘FRNFN’ or ’frnfn’.

With above reasoning we can have total of 384 different states (3 (waypoints) X 2(lights) X 4(oncoming) X 4(right) X 4(left) = 384) for this kind of grid of any size with given this set of input conditions. Yes, this seems like reasonable number given the state machine can be applied to any grid size. With Q-learning we wanted to learn best possible (state, action) pair for any given state and this state will capture all unique state representing grid intersections.

Also, by choosing the state based on above combinations, I don’t to worry about changed traffic rules (as the state capture fully all possible values) in simulation with given set of possible values in each traffic. This provides nice residence and extensibility at the cost increased possible state values.

Implement a Q-Learning Driving Agent

QUESTION: *What changes do you notice in the agent's behavior when compared to the basic driving agent when random actions were always taken? Why is this behavior occurring?*

The basic approach for implementation is

1. Initialize the Q-values table (Q_learn) for each possible (state,

action) observed : Q(s, a). I have chosen 0. has initial value
2. Observe the current state, s.
3. Choose an action, a, for that state based on one of the action selection policies (example: Greedy-Epsilon).
4. Take the action, and observe the reward, r, as well as the new state, s'.
5. Update the Q-value for the state using the observed reward and the maximum reward possible for the next state. The updating is done according to the formula outlined and parameters described above.
6. Set the state to the new state, and repeat outlined steps from 1 to 6.

The step5, following equation summarizes the process of Learning

Q(s, a) = Q(s,a) + alpha [r(s,a) + gamma (argmax(Q(s', a') - Q(s,a)]

Where,

alpha : learning rate: determines to what extent the newly acquired information will override the old information

gamma (discount factor): the weight agent should attribute to future reward vs current reward.

argmax(Q(s', a') : Q-values from potential future actions

In step 3, I have chose Greedy-Epsilon policy for action selection: In this policy, the action is selected greedily with respect to the Q-value estimates a fraction (1–ϵ) of the time (where ϵ is a fraction between 0 and 1), and randomly selected among all actions a fraction ϵ (epsilon - exploration rate) of the time. Policy that has some randomness promotes exploration of the state space.

With this implementation, i have run the agent with few scenarios with learning and policy parameters set to their extremes to understand agent behavior,

1) With with basic set of parameters set to values that with no learning or no exploration to observe the behavior(Exploration rate to 0, future

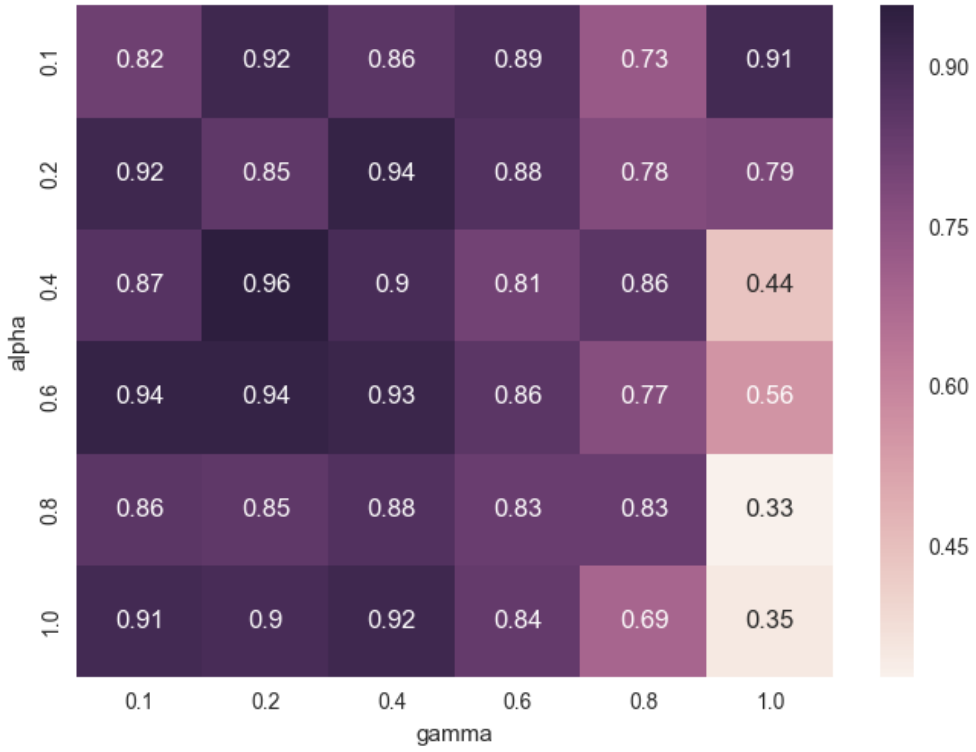
discounts to 0, learning rate to 0). As suspected, agent didn't reach destination most of the time before the deadline, confirming that agent policy is choosing maximum explored action but with no chance to exploring the new states (local maximum problem)

2) With exploration rate to 1, future discounts to 1, learning rate 1 (always learning but always exploring without using learning data). In this case, as expected agent didn't reach destination most of the time within deadline but did better than previous scenario. In this case, agent policy is always taking random actions without looking into learned rewards. This is pretty much like random actions at each step.

3) With exploration rate to 0 (which means greedy all the time), future discounts 1, learning 1 agent did poorly again. As expected, agent policy is choosing maximum of explored (state, action) pair values runs into (local maximum) problem. And didn't converge to destination.

With this observation, found the need to fine tune learning rate and policy choices (parameters: alpha, gamma, epsilon) to be able to reach destination optimally.

By implementing 'heatmap' analysis as suggested (wonderful way to understand the parameter tuning), which nicely summarizes the hotspots for alpha and gamma combinations. Found that gamma less than 0.4 proves nice hotspots (agent reaching destination 90% of the trials) for different set of alpha values. Since our simulation is deterministic world, we can chose alpha to be anywhere from 0.1 (like in practice) or constant 1.



Agent is NOT really coded with rules of the road, instead it is learning actions to be taken any given state by exploring and exploiting all possible actions and their rewards. Agent needs to explore many possible actions from given state to understand best action possible in given state. Also, at the same time agent needs to take best action (greedy) from explored actions to maximize chances of reaching the destination.

With heat map helping to fine-tune alpha, gamma values, I have tried various epsilon greedy strategy to understand policy combinations, greedy strategy 80% of the time (exploit) and 20% chance to explore randomly(exploit). Greedy strategy 70% and 30% random. Found that Overall 90%-10% has better success rate.

However, it is still not optimized policy, as negative rewards are observed after many iterations (> 10,000) due to random actions (not

choosing to be greedy). And also, with being more greedier (90%) without exploration caused agent suffering local maxima problem.

For example, following results show the problem(for 1000 iterations), of choosing action randomly causing the negative penalty.

LearningAgent.update(): iter= 999, self.t = 2, deadline = 23, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, waypoints =left, action = right, greedy_type = random, reward = -0.5, q_values = {'forward': -0.5000001622016008, 'None': 0.09586581338838779, 'right': -0.08399467513116637, 'left': -0.8232010665989513}

LearningAgent.update(): iter= 999, self.t = 3, deadline = 22, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'left': None}, waypoints =right, action = forward, greedy_type = random, reward = -0.5, q_values = {'forward': -0.08002581502416123, 'None': 0.5000245428906631, 'right': 12.095865813388388, 'left': -0.49986666661148604}

Improve the Q-Learning Driving Agent

QUESTION: *Report the different values for the parameters tuned in your basic implementation of Q-Learning. For which set of parameters does the agent perform best? How well does the final driving agent perform?*

QUESTION: *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? How would you describe an optimal policy for this problem?*

In order to improve, I have implemented epsilon-decay policy. Which allows exploration (0.8) to be higher (choosing random actions to explore many states) in the beginning causing agent to choose random actions (80%) forcing agent to learn from failures. As the agent explored enough with failures, With epsilon decay reduced

exploration at each iteration (epsilon_decay) to rely more on exploitation(greedy policy). This worked really well as agent comes to rely on learned actions as iterations increases (epsilon decaying to less than 0.000001) and considerable drop in negative rewards.

Of course, there were occasional negative rewards due to state not being explored but never or rarely due to random action.

Example results (946 successful journey out of 1000 iterations),

Q_learning table (state = WayPointLightOnComingLeftRight)

{'rgNNN': {'forward': -0.0833333333191601, 'None': 0.4967936, 'right': 2.1158709756586673, 'left': -0.016638977648995024},

'fgfNN': {'forward': 0.0, 'None': 0.4166718358186877, 'right': 0.0, 'left': -0.5000010666011303},

'lgNNr': {'forward': 0.0, 'None': 0.4193341866640461, 'right': 0.0, 'left': 0.0},

'frflN': {'forward': -0.5000000065667081, 'None': 0.0, 'right': 0.0, 'left': 0.0},

'rrNNl': {'forward': -0.6, 'None': 0.0, 'right': 2.416670933398869, 'left': 0.0},

'rglNN': {'forward': -0.08332820650719075, 'None': 0.499205375983637, 'right': 4.416666922597851, 'left': 0.0},

'frrNN': {'forward': -0.9159938560682667, 'None': 0.41692799146723375, 'right': -0.080134186650934 47, 'left': -0.8239743996040526},

'frlNN': {'forward': -1.0, 'None': 0.4198666666666667, 'right': -1.0, 'left': -1.0},

'lgNNl': {'forward': 0.0, 'None': 0.0, 'right': -0.0033330791082631372,

'left': 2.0839936},

'frNlN': {'forward': -0.9833333334016, 'None': 0.019334604733918792, 'right': 0.0, 'left': -0.5128000425957827},

'rrNNf': {'forward': -0.5033661522051123, 'None': 0.49986666666668944, 'right': 0.0, 'left': 0.0},

'fgNNN': {'forward': 2.083871787005272, 'None': 0.5032245756586676, 'right': -0.5, 'left': -0.08319898617184251},

'lgNrN': {'forward': 0.0, 'None': 0.0, 'right': 0.0, 'left': 2.480133367466562},

'fgNNr': {'forward': 0.0, 'None': 0.4166668796586843, 'right': 0.0, 'left': -0.0830719917424596},

'rrlNN': {'forward': -1.0, 'None': 0.49999466666666675, 'right': -1.0, 'left': -0.504},

'rgNNl': {'forward': 0.0, 'None': 0.0, 'right': 12.49997418666669, 'left': -0.08000639180527347},

'rgNrN': {'forward': -0.40333337427968524, 'None': 0.49666708309606905, 'right': 0.0, 'left': 0.0},

'frrlN': {'forward': 0.0, 'None': 0.0, 'right': 0.0, 'left': 0.0},

'lrlNN': {'forward': -0.5833587203408963, 'None': 0.48400512164102233, 'right': -0.11600518143999983, 'left': -1.0},

'fgNrN': {'forward': 2.0800053756592125, 'None': 0.0, 'right': -0.01935999999999999, 'left': -0.016775680000000015},

'lrNNr': {'forward': -1.0, 'None': 0.0, 'right': 0.0, 'left': -0.5837120426644813},

'lrNfN': {'forward': 0.0, 'None': 0.17679996721834842, 'right': 0.0, 'left':

-0.583232},

'frNfN': {'forward': -1.0, 'None': 0.4838666240157286, 'right': -0.0032246169736525054, 'left': -1.0},

'fgrNN': {'forward': 12.083334195131734, 'None': 0.0, 'right': 0.0, 'left': 0.0},

'lgNfN': {'forward': -0.41666667485866316, 'None': 0.0, 'right': -0.08332906625709274, 'left': 2.0840053326512127},

'lrNlN': {'forward': 0.0, 'None': 0.09999978699174272, 'right': 0.0, 'left': 0.0},

'fgNNf': {'forward': 2.1799741866940563, 'None': 0.0, 'right': -0.4999936017066667, 'left': -2.612020838393958e-05},

'rgfNN': {'forward': 0.0, 'None': 0.0, 'right': 0.0, 'left': -0.599872},

'lgNNN': {'forward': 0.3998666734796802, 'None': 0.41666688155663123, 'right': -0.01666666530143812, 'left': 2.579999745719774},

'rrfNN': {'forward': 0.0, 'None': 0.0, 'right': 2.416794666682833, 'left': -1.0},

'rgNNr': {'forward': -0.08335462237470803, 'None': 0.0, 'right': 0.0, 'left': -0.01666666527470384},

'rrrNN': {'forward': 0.0, 'None': 0.41984512, 'right': 0.0, 'left': 0.0},

'fgNNl': {'forward': 2.5031946666672127, 'None': 0.0, 'right': 0.0, 'left': 0.39666667683512324},

'lgNlN': {'forward': -0.5, 'None': 0.0033853869392929054, 'right': -0.0032266666640670127, 'left': 0.0},

'frNrN': {'forward': 0.0, 'None': 0.10322661377374076, 'right': -0.5,

'left': -0.9006666649736577},

'lrNNl': {'forward': -0.9166658474011307, 'None': 0.08412266674585617, 'right': -0.083201237333333329, 'left': -0.4833651613689456},

'frNNl': {'forward': 0.0, 'None': 0.020665386693317966, 'right': 0.0, 'left': -0.9000000017066833},

'lglNN': {'forward': -0.5, 'None': 0.0, 'right': 0.0, 'left': 4.4967946666994347},

'lrNNN': {'forward': -0.080800213333334164, 'None': 0.4833592299383893, 'right': -0.083333129898688852, 'left': -0.5833280426770643},

'lrNrl': {'forward': 0.0, 'None': 0.0, 'right': 0.0, 'left': -0.9198664106829677},

'rgrNN': {'forward': 0.0, 'None': 0.499967999999999997, 'right': 2.8967946666671915, 'left': 0.0},

'lrfNN': {'forward': -0.6, 'None': 0.4198402134010751, 'right': -0.32061337804526866, 'left': -0.50333332488192},

'frNNf': {'forward': -1.0, 'None': 0.0014141951317333742, 'right': 0.0, 'left': 0.0},

'rgNlf': {'forward': 0.0, 'None': 0.0, 'right': 2.579333375931733, 'left': 0.0},

'lrrNN': {'forward': 0.0, 'None': 0.0, 'right': -0.08399999999999996, 'left': 0.0},

'fgNfN': {'forward': 2.0206346666666843, 'None': 0.0, 'right': 0.3969280389938983, 'left': 0.0},

'rrNlN': {'forward': 0.0, 'None': 0.42332287991808004, 'right': 0.0, 'left':

-0.5161599573224107},

'lgrNN': {'forward': 0.0, 'None': 0.0, 'right': -0.4841866236067884, 'left': 0.0},

'rgNfN': {'forward': 1.99997312, 'None': 0.0, 'right': 2.4167997866672084, 'left': 0.0},

'fgNfl': {'forward': 0.0, 'None': 0.0, 'right': 0.0767948475591348, 'left': 0.0},

'frNNr': {'forward': 0.0, 'None': 0.087813367466687 65, 'right': 0.0, 'left': -0.9833229226664526},

'rrNNN': {'forward': -0.5839744003304106, 'None': 0.5159933866666667, 'right': 2.0833344425847464, 'left': -0.02013332488126 4847},

'rrNfN': {'forward': 0.0, 'None': 0.0, 'right': 2.0833343914797737, 'left': -0.5006410243964929},

'lgfNN': {'forward': 0.383876915134442 33, 'None': 0.0, 'right': -0.06412906666655843, 'left': -0.1001331282025703 7},

'frNNN': {'forward': -0.5833333326670724, 'None': 0.4167743574010543, 'right': -0.019834871480215233, 'left': -0.9835455589619021},

'lgNrl': {'forward': -0.08320000006553618, 'None': 0.0, 'right': 0.0, 'left': 0.0},

'frNrf': {'forward': 0.0, 'None': 0.0, 'right': 0.0, 'left': -1.0},

'lgNNf': {'forward': 0.0, 'None': 0.0, 'right': 0.0, 'left': 12.416794837399},

'fgINN': {'forward': 12.09935978668021, 'None': 0.0, 'right': -0.4008,

'left': -0.016666877968273 93},

'rgNff': {'forward': 0.0, 'None': 0.0, 'right': 2.4166666666796646, 'left': 0.0},

'rgNNf': {'forward': -0.0007997866120532 66, 'None': 0.4966666673355731, 'right': 2.499999959039891, 'left': -0.5},

'rrNrN': {'forward': 0.0, 'None': 0.0, 'right': 2.0966666670714185, 'left': 0.0},

'fgNlN': {'forward': 2.099333374293966, 'None': 0.0, 'right': -0.480032, 'left': -0.00400000000000 0036},

'lrNrN': {'forward': 0.0, 'None': 0.4833333674666667, 'right': 0.0, 'left': 0.0},

'rgrrN': {'forward': 0.0, 'None': 0.0, 'right': 2.4992012714666845, 'left': 0.0},

'fgNll': {'forward': 0.0, 'None': 0.083358974309172 05, 'right': 0.0, 'left': 0.0},

'rgNlN': {'forward': 0.0, 'None': 0.0, 'right': 0.0, 'left': -0.0007999999999999119},

'lrNNf': {'forward': 0.0, 'None': 0.4839946666667926, 'right': 0.0, 'left': -0.919328002048},

'frfNN': {'forward': -0.6, 'None': 0.4999946673355708, 'right': -0.099333162667190 72, 'left': -1.0}}

Following are some of the penalties iterations where negative rewards accused due to policy exploration still not being optimal.
However no random action penalities compared to fixed epsilon strategy,

[492, {'light': 'red', 'oncoming': None, 'right': 'forward', 'left': None},

'forward', 'right', -0.5, {'forward': -1.0, 'None': 0.0, 'right': 0.0, 'left': 0.0}]
[492, {'light': 'red', 'oncoming': None, 'right': 'forward', 'left': None}, 'forward', 'right', -0.5, {'forward': -1.0, 'None': 0.0, 'right': 1.9166978047317338, 'left': 0.0}]
[552, {'light': 'red', 'oncoming': None, 'right': 'forward', 'left': None}, 'forward', 'left', -1.0, {'forward': -1.0, 'None': 0.0, 'right': -0.0032246169736525054, 'left': 0.0}]
[586, {'light': 'red', 'oncoming': None, 'right': None, 'left': 'left'}, 'left', 'left', -1.0, {'forward': 0.0, 'None': 0.0, 'right': -0.083201237333333329, 'left': 0.0}]
[642, {'light': 'red', 'oncoming': None, 'right': None, 'left': 'left'}, 'left', 'forward', -1.0, {'forward': 0.0, 'None': 0.0, 'right': -0.083201237333333329, 'left': -0.4833651613689456}]
[924, {'light': 'red', 'oncoming': None, 'right': None, 'left': 'right'}, 'left', 'left', -1.0, {'forward': -1.0, 'None': 0.0, 'right': 0.0, 'left': 0.0}]
[944, {'light': 'green', 'oncoming': 'forward', 'right': None, 'left': None}, 'left', 'forward', -0.5, {'forward': 0.0, 'None': 0.0, 'right': -0.06412906666655843, 'left': -0.1001331282025703 7}]

Given the grid model, and router planner strategy, the best or ideal policy would be follow the planner guidance (way_point) as ‘action’ at each intersection with ’None’ action if there is chance of collision or violating traffic rules. This would avoid any penalties and reach destination optimally.

The policy parameters (alpha 0.1 or 1, gamma 0.2 to 04, epsilon=0.8 with decay_rate= 0.991) my agent performed better close to optimal policy.
My agent is close to this behavior most of the time after policy is learned but occasionally suffered failure or negative rewards due to policy states not being explored completely.