Train a Smartcab How to Drive

**Project 3: Reinforcement Learning**

* A smartcab is a self-driving car from the not-so-distant future that ferries people from one arbitrary location to another. This project utilizes reinforcement learning to train a smartcab how to drive. Design the AI driving agent for the smartcab. It should receive the above-mentioned inputs at each time step t, and generate an output move. Based on the rewards and penalties it gets, the agent should learn an optimal policy for driving on city roads, obeying traffic rules correctly, and trying to reach the destination within a goal time.

**Environment**

* The smartcab operates in an idealized grid-like city, with roads going North-South and East-West. Other vehicles may be present on the roads, but no pedestrians. There is a traffic light at each intersection that can be in one of two states: North-South open or East-West open. US right-of-way rules apply: On a green light, you can turn left only if there is no oncoming traffic at the intersection coming straight. On a red light, you can turn right if there is no oncoming traffic turning left or traffic from the left going straight. At any instant, the smartcab can either stay put at the current intersection, move one block forward, one block left, or one block right (no backward movement). The smartcab gets a reward for each successfully completed trip. A trip is considered “successfully completed” if the passenger is dropped off at the desired destination (some intersection) within a pre-specified time bound (computed with a route plan). It also gets a smaller reward for each correct move executed at an intersection. It gets a small penalty for an incorrect move, and a larger penalty for violating traffic rules and/or causing an accident.

**Inputs**

* Assume that a higher-level planner assigns a route to the smartcab, splitting it into waypoints at each intersection. And time in this world is quantized. At any instant, the smartcab is at some intersection. Therefore, the next waypoint is always either one block straight ahead, one block left, one block right, one block back or exactly there (reached the destination). The smartcab only has an egocentric view of the intersection it is currently at (no accurate GPS or global location). It is able to sense whether the traffic light is green for its direction of movement (heading), and whether there is a car at the intersection on each of the incoming roadways (and which direction they are trying to go). In addition to this, each trip has an associated timer that counts down every time step. If the timer is at 0 and the destination has not been reached, the trip is over, and a new one may start.

**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)
  + 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 the 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.

**Question 1: Observing the agent’s behavior. Does it eventually make it to the target location?**

* Given the valid actions the smartcab can take `None,'forward','left','right' and the valid\_inputs {'light','oncoming','left','right'}, the goal for the agent is to reach the final destination by picking the optimal actions. In this instance we see the agent's actions are completely at random regardless of the conditions in the environment. For example, there are numerous examples throughout the iterations where the agent does not learn running a red light results in a negative reward, thus this action should not be repeated. Another case would be when the light turns green and there is oncoming traffic, the action the agent takes is 'None', translating into a negative reward. Majority of the time the smartcab does not reach the destination within the deadline, but in few cases it does. Calculating the success rate (number of times the agent reaches the destination per number of trials), only 77% of the time the agent does not reach the destination by the deadline [ntrials=100, episodes=1]. Furthermore, the number of penalties (negative rewards) per each move the smartcab made was 26.12% (723/2716). As we see, the performance of the smartcab is rather poor as expected given the actions were at random, thus this will act as our benchmark for comparing how much the agent improved by implementing Q-learning.

**Identify and update state**

* Identify a set of states that are appropriate for modeling the driving agent. The main sources 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.

**Question 2: Justify why you picked these set of states, and how they model the agent and its environment.**

* The following states are considered in the model:
  + {`light`: green, red}
  + {`oncoming`: None, 'forward', 'left', 'right'}
  + {`left`: None, 'forward', 'left', 'right'}
  + {`right`: None, 'forward', 'left', 'right'}
  + {`way\_point`: None, 'forward', 'left', 'right'}
* These four states (`light, oncoming, right, left, next\_waypoint`) provide a good start for the agent to begin learning the surrounding environment and traffic laws by observing numerous variation of the environment at each intersection. For instance, the smartcab needs to know if the light is red/green and whether it should proceed through the intersection or stop. If the light is red, the cab can turn right if there is no oncoming traffic. Alternatively, if the light is green, the cab can turn left (assuming no turn signal) if there is no oncoming traffic. Hence the importance of these four states being included in the model so that the cab can learn and obey the traffic laws. The agent will learn by visiting every state multiple times in order to learn the value of each action. However, if too many states are present, the time it will take for the agent to learn will be significantly longer. On the other hand, if there are only a few states then it may be unable to determine between actions and which one to take.

**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.

**Question 3: What changes do you notice in the agent’s behavior?**

Enhance the driving agent

* Apply the reinforcement learning techniques 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.
* QLearning is an algorithm that is used during reinforced learning where the agent attempts to learn what the optimal policy is from its history of interacting with the environment. The history of the agent is just a sequence of the state-action-reward-next state <s,a,r,s`>. The algorithm stores a value for each state-action pair Q(s,a) in either a matrix or a table. Initially, all of the Q-values are set equal to zero. Just as a note, this can require a large amount of memory given a situation where there are numerous states and actions. In every state the Q-Learning algorithm visits it will determine what is the best action to take, rather than choosing randomly. Next, the maximum of these future values are selected and is incorporated in to the current Q value. The caveat to Q-Learning is that you have to explore enough of the grid world to eventually reduce the learning rate over time while in the meantime, using what the agent already knows to get a high rewards, which is referred to as exploitation. In this code, I implement the ε-greedy (ε = epsilon) method for exploration. This method flips a coin at every time step and then decides to make a random action or the optimal policy given the following inequality: if epsilon is greater than the random integer (0<x<1), the agent will then act randomly; meaning a random action is made a fraction of the time, else follow the current best Q value. The equation is given below for how the Q-values are updated at a given time. [Source: “*Q-Learning for a Simple Board Game*”, Arvidsson and Wallgren, 2010]

**Equation:**

*where:*

= Discount Factor; future rewards are worth less than the current reward; 0<<1, as -> 0, the agent ignores all future rewards

= Learning Rate; closer to 1, means the agent will only consider future rewards with greater weight. As gamma approaches 0, the agent will consider only immediate rewards.

= Feedback; the action gives feedback from the environment (positive or negative reward); negative reward can be called punishment or penalty. If the agent reaches the destination on time, a reward of 10 is assigned; if the agent makes a penalty, such as running the red light , a reward of -1 is given.

= Expected discounted feedback

= Old Q-Value

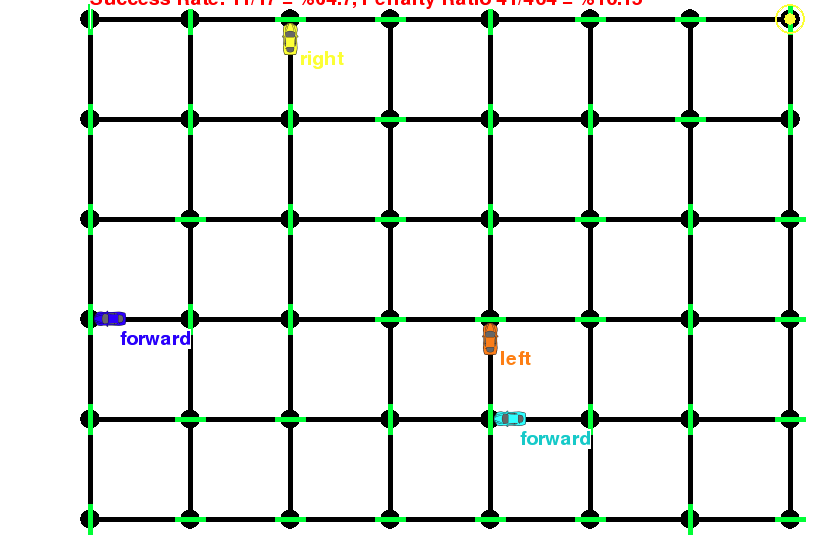
= Max future Q-value

* After implementing Q-Learning algorithm to the code, the results have significantly improved in contrast to the initial configuration where the agent was selecting actions at random. The results show that after running n\_trials = 100,

Epsilon is set at 0.30, which gives

**Question 4: 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?**

**Question 5: 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?**

Screenshot of grid: 5X7