**TRAIN A SMARTCAB TO DRIVE**

A smartcab is a self-driving car from the not-so-distant future that ferries people from one arbitrary location to another. In this project, you will use reinforcement learning to train a smartcab how to drive.

**About the Problem**

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 (sorry, no accurate GPS, no 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.

**Install and How to run the program**

This project need Python 2.7 and Pygame library

<https://www.pygame.org/wiki/GettingStarted>

Make sure you are in the top-level project directory *smartcab/* (that contains this README). Then run:

***python smartcab/agent.py***

or:

***python -m smartcab.agent***

**Task 1 - The basic driving agent**

I built a basic driving agent, these agent choice randomness of the allowed actions (None, Forward, Left, Right ). The definition of random choice coded in python is shown above:

|  |
| --- |
| **Def** choiceRandomAction(self, state):  Actions = self.mapActions(state)  bestAction = random.choice(Actions)  **return** bestAction |

In every trial, the agent choice one random action. To test the random agent, I made it run through a total of 100 trials to check how well it was.

This basic agent was implemented in the file *agent.py*. I did in the code a switch (or variable) to change between the random and learning agent, the *self.Learning* variable is this switch. If the *self.Learning* is False, the random agent will be active, and if is True the Learning agent will be active. In terms of a rough algorithm, this is what it actually performs:

|  |
| --- |
| State = Actual State of agent.  If Learning if False:  Action = random choice of (None, Forward, Left, Right)n and set reward variable  Perform Action and set reward variable  New State = Actual new State of agent. |

The results is present in *CabResults\_Random.txt*, the agent when

does not fails, he takes a several steps to reach the destination. The random agent fails to reach the destination 85 times in 100, in other words the agent reaches only 15%. Because of this agent has the random behavior there are times when the agent performs better, sometimes it does not. The results from this run is not surprising.

NOTE: The results would be a good metric to compare with Q-Learning Cab.

**Task 2 – The Q-Learning Agent.**

*“In Reinforcement Learning an agent without a priori knowledge learns by its interaction with the environment, which retrieves a feedback, commonly called reward, as a result of the action performed by the agent. The basic assumption of RL is that an agent can learn a policy to choose actions that will maximize its revenue in the long term only by experiencing the environment. Consider an autonomous agent interacting with its environment via perception and action. On each interaction step the agent senses the current state s of the environment, and chooses an action a to perform. The action a alters the state s of the environment, and a scalar reinforcement signal r (a reward or penalty) is provided to the agent to indicate the desirability of performing the chosen action a when in the observed state s, leading to the resulting state s´. The goal of the agent in a RL problem is to learn na action policy that maximizes the expected long term sum of values of the reinforcement signal, from any starting state.”*

The Text above was taken from the article “*Knowledge Transfer in Heuristic Reinforcement Learning: An Early Investigation*” and explain very well what is Reinforcement Learning and what need to do in the task 2. Building one of most known algorithm of Reinforcement Learning, the Q-Learning.

### 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.

Apply the reinforcement learning techniques you have learnt, 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.

**States of the Problem**.

I had some of possible candidates to use as states of the problem (deadline, next waypoint by the planner, traffic light, traffic data of oncoming, left, right etc.). I could use as many percepts I want, but if I use too much I believe the time to algorithm Q-Learning converge in an optical policy can be more than 100 trials. My assumption is based in how the QL works, because if I have too much different states, the time to learn the variation of all states and the size of Q table can be so big that the time to learn can take more than 100 trials, in other words can be a problem to find an optical policy. To solve this dilemma, I choice to use the state variables - **next waypoint** and **traffic light**.

I found these two variables a good combination to reach a useful performance in the Learning, because in a grid world is important to learn what need to do in the next grid, by grid in grid and next waypoint is good choice to do next. The traffic light helps in the areas of the oncoming traffic, training the cab to perform legal moves.

**Building the Q-Learning Agent.**

The Q-Learning agent use the basis the Q-Learning strategy, the code that I implemented is present in the file *agent.py*.

The code below describes the overall process involved in the q-learning agent

|  |
| --- |
| **self.state = self.mapState(inputs)** *=> receive in state parameter (traffic\_light,next\_waypoint) – This is the actual state of the agent.*  **action = self.choiceAction(self.state)** *= > choice the action by the policy learning.*  **reward = self.env.act(self, action)** *=> Execute action and get reward*  **self.NEWstate = self.mapState(inputs)** *=> After the action, the agent is in new state, is necessary to know where is.*  **self.QLupdate(reward, action, self.state, self.NEWstate)** *=> Update the QL with the reward of action and states* |

The action is choice by the policy that gets the best Q-value for every state where the agents goes. The policy uses exploitation (make the best decision given current information) and exploration (gather more information) to learn how to choice the best action. After that, the action is taken and gets the reward and the agent goes to another state.

The learning of agent is going through several iteration to learn how to take best action and reach of the best policy of the problem.

NOTE: The exploration helps the algorithm to don't be stuck in a minimum local.

**Results for the Q-Learning agent.**

The parameters used in the experiment are exploration/exploitation = 0.1, gamma =0.9, alpha =0.2, the value of the gamma and alpha was found by iterations over many possible values for alpha and gamma over all the runs. The initial Q-value was set to a value of 20, this value is more than the highest possible positive reward. The value lets us use policy decision each iteration without having to do with the problem of exploration or exploitation. Initially we would be doing random decisions.

The Q-learning agent with this variables was ran 100 runs. The results can see in the file CabResults\_QL.txt. The cab reached the destination 93 times over the 100 runs and this performance can be checked by executing *agent.py*.

To show how the initial Q-value in 20 improve the learning, I did three tables, in all the tables I compare with and without the set table value. All results of the tables represents the number of times that the cab reached the destination.

**Table 1: Gamma = 0.9, alpha=0.2 and change epsilon**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **epsilon** | **0.1** | **0.2** | **0.4** | **0.6** | **0.8** | **1.0** |
| **Initial Q-value =0** | 76 | 71 | 57 | 42 | 32 | 19 |
| **Initial Q-value = 20** | 93 | 76 | 65 | 50 | 22 | 20 |

The information on Table 1 is not surprising, because when a high value of epsilon more randomness is the algorithm and worse is the performance, only thing we can note here is the difference with the initial Q-value with 20.

**Table 2: Gamma=0.9, epsilon =0.1 and change alpha value**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **alpha** | **0.1** | **0.3** | **0.4** | **0.6** | **0.8** | **1.0** |
| **Initial Q-value = 0** | 77 | 75 | 71 | 66 | 58 | 52 |
| **Initial Q-value = 20** | 93 | 87 | 85 | 84 | 75 | 68 |

The Table 2, show the variation of alpha value, with a small value of alpha (0.1) the agent will be slow to converge, but the estimates will not fluctuate very much due to randomness, again, it can see the differences between the initials value.

**Table 3: epsilon =0.1, alpha=0.2 and change gamma value**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **gamma** | **0.1** | **0.4** | **0.6** | **0.8** | **0.9** | **1.0** |
| **Initial Q-value = 0** | 80 | 78 | 82 | 77 | 76 | 40 |
| **Initial Q-value = 20** | 86 | 83 | 80 | 86 | 93 | 60 |

In the table 3 the parameter gamma was changed between 0.1 to 1.0, when the gamma is close to “0” the agent tends to consider only immediate values of reward. If the values are very close to 1, the agent considers the future rewards with the most weight. One interested thing when the value of gamma is one is about the performance, they drop substantially, probably considers only the future rewards with the most weight is not a good choice.

It is common in the Q-Learning the initials values to be random, this strategy helps the agent to “explore” the environment and avoid to pick up action with same initial value, but to do that in the smartcab I need to run all the states and all the actions and in my opinion it is not productive for this kind of simulation.

“What happens if the agent is navigating/learning from another location or in a state/country with another setting of traffic laws and regulations?”

In reinforcement learning when the environment change or the rules (law/regulations) changes is a problem, the agent need to learn again, probably he will do a lot of things wrong and receive a lot of bad rewards and update the learn table to learn the news rules, but it takes time.

“Changes in behavior of your agent and how the Q-Learning algorithm affects this behavior”

When the agent start, the actions are still random, because the Q-tables is basically a lot of number without significant values, the Q-table will be update solely with future rewards when the agent reache the bojetive. Action are chosen in accordance with the maximum Q-value of the neighboring states.

Take more than 20 runs to agent start to show something smart and the agent start to reach the objective more frequently tha the random agent, but he still show problem with clockwise loops, about 50~60 runs the agent show a descente aprendizado, apesar de ainda escolher muitas vezes nenhum acao. Somente superior a 80 runs ele parece ter resultados inteligente e faz o esperado.

Great analysis on the improvements and behaviors of the agent. Please provide the different alpha and gamma values that were used to tune the model. Also please provide the success rate for them and the utility/rewards.

I would recommend to use a table for this, as visuals are a great way to tell a story. It also makes stats easier to read!

**Q-Learning Agent *versus* Random Agent *versus* SARSA**

When we compare the agents, we need to pay attention in two important aspects: Agent that got destination with less time, and agent to reach the destination with a cumulative positive reward, even with negative reward steps. Just looking the files, we can see that the Q-Learning agent always reaches the destination with good accumulative reward and at most of the time with few steps, this is happening because the agent learn an optimal policy, learning to take the right movers and smallest route.

I built a third algorithm to compare, the SARSA (State-Action-Reward-State-Action) algorithm

“*This name simply reflects the fact that the main function for updating the Q-value depends on the current state of the agent "****S****1", the action the agent chooses "****A****1", the reward "****R****" the agent gets for choosing this action, the state "****S****2" that the agent will now be in after taking that action, and finally the next action "****A****2" the agent will choose in its new state. Taking every letter in the quintuple (st, at, rt, st+1, at+1) yields the word SARSA* “Wikipedia

The result with SARSA was worse, the cab reached the destination 83 times over the 100 runs (with Q-Learning was 93 and with random was 15). The different between these algorithms is because Sarsa learns the policy and sometimes takes an optimal actions and explores other actions, while Q-Learning takes optimal (estimated) action and learn about the policy that does not explore. Sarsa will learn to be careful in an environment where exploration is costly, Q-learning will not. These differences made the Q-Learning have a better performance than SARSA in the CAB problem.

Note: I have run a lot of trials, and the Q-Learning agent reaches the destination over 90% all the times, but I believe with uses of heuristic (I found several articles about it) it is possible to get a better results I just don´t did that because in my opinion it is over the scope of this work and probably I need to change more than just *agent.py*, but I will do by my own curiosity.