Train a smartcab how to drive

Reinforcement Learning Project for Udacity's Machine Learning Nanodegree.

Install

This project requires Python 2.7 with the pygame library installed:

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

The Current Features

This project currentl uses two types of agent:

- The frst type of agent is what I call a random/initiated type of agent. It takes action in the direction of more reward. It also switched back to random mode when It finds that it performs bad when it is initiated mode.
- The second type of agent uses Q-learning method to learn the true values of each state and perform actions accordingly.

The following line can be used to toggle between the Qlearning agent and The normal learning agent:

```
    a = e.create_agent(QLearningAgent)  # create agent
```

How to run the program

Make sure you are in the top-level `smartcab` directory. Then run:

```
python smartcab/agent.py
```

OR:

```
python -m smartcab.agent
```

Task 1

Implement a basic driving agent

Download smartcab.zip, unzip and open the template Python file agent.py (do not modify any other file). Perform the following tasks to build your agent, referring to instructions mentioned in README.md as well as inline comments in agent.py.

Also create a project report (e.g. Word or Google doc), and start addressing the questions indicated in italics below. When you have finished the project, save/download the report as a PDF and turn it

in with your code.

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), and,
- 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 your 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. In this mode, the agent is given unlimited time to reach the destination. The current state, action taken by your agent and reward/penalty earned are shown in the simulator.

In your report, mention what you see in the agent's behavior. Does it eventually make it to the target location?

Implementing the basic driving agent

I went on to create the initial driving agent with a greedy approach in mind. I felt it could do better than complete randomness. My initial go at the basic driving agent is present in `agent.py` and is called as the `LearningAgent`.

Working of naive driving agent

It implements a naive greedy strategy and picks a random legal action to go ahead. Once it performs this action, it later checks if this action gives it a good reward or no. If it gives a bad reward, it simply performs a random action in the next move, else it performs an initiated action. In terms of a rough algorithm this is what it actually performs:

```
States :=> 'Random', 'Initiated'

Initally:
    State :=> Random
    lastAction => None
    lastReward => None

While Cab has not Reached Destination do:
    if State == Random:
        actionPerformed :=> Pick a Random Legal Action
                            from ['Forward','Left',
                               Right', None]

        Perform Action and set reward variable
    else if State == Initiated:
        actionPerformed => lastAction

    # Update State
```

```
    If reward > lastReward
        State = Initiated
    else
        State = Random

    # Set Last Action performed and last reward gained
    Set:
        lastAction :=> actionPerformed
        lastReward :=> reward
```

The algorithm works in a very naive way by finding a path that maximises the reward and follows it. In the above pseudocode, the algorithm only perfoms legal action and hence does not pick a negative reward.

How well does it perform ?

I ran test trials of 10 with enforced deadline and the following table represents the tabular results of each trial:

Trial #	Result
1	Primary agent could not reach destination within deadline!
2	Primary agent could not reach destination within deadline!
3	Primary agent could not reach destination within deadline!
4	Primary agent could not reach destination within deadline!
5	Primary agent could not reach destination within deadline!
6	Primary agent could not reach destination within deadline!
7	Primary agent has reached destination!
8	Primary agent could not reach destination within deadline!
9	Primary agent could not reach destination within deadline!
10	Primary agent could not reach destination within deadline!

As you can see, the naive agent was able to only reach the destination once out of 10 test runs.

Task 2

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.

What changes do you notice in the agent's behavior?

Enhance the driving agent

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.

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?

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?

Identifying the states

The Q-learning agent I implemented follows the same algorithm as prescribed in the class. The first step towards designing the q-learning agent was to identify the possible states. At any given point, the cab can sense many percepts out of which we need to select the values that best define the current state of the cab. Here are some of the possible percepts that were candidates for selection: - location - destination - next_waypoint by the planner - traffic light - traffic data of oncoming, left, right etc.

Among all these inputs, I choose to model the following two state variables: - next waypoint - traffic light

I went ahead with these two variables for two specific reasons and they are explained below.

Performance

In terms of performance both these variables in combination proved to be really useful in performance. Although, among each possible runs using next waypoint seems pointless but it best serves as an input to model the cab w.r.t destination. Some of the ideal candidates in this case would've been the destination variable itself or maybe even the location of the cab. But, here are some of the issues that could come up when using these: - destination: Destination changes at each interval, therefore encoding it as a state would defeat the purpose and the agent would not be able to learn properly. - location: This is another variable that should not be used to model the state as it would cause problems due to the sheer size of the grid. This would mean that we will have plenty of states and for the q values to converge, it would definitely take more than 100 trials.

Thus, in order to model the destination as a form of the state, I went ahead with next waypoint.

Train the cab to perform legal moves

This was another reason that I went ahead with one of my state variables being traffic light. This helps in modelling the traffic rule and also helps in training the cab to perform legal moves. One of the areas to improve upon would be to model the oncoming traffic to act legally using right of way rules.

Working of the Q-learning agent

The Q-Learning agent works on the basis of the Q-Learning strategy. The code for such an agent is present in the file `QLearning.py`.

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

```
States :=> (traffic_light,next_waypoint)

learning_variables :=> alpha,gamma,epsilon

While cab has not reached destination:

    execute makeStateVariable
    execute getBestAction
    execute GetReward
    execute updateQTable
```

The following is the pseudo code for getting the action based on q-value

```
getBestAction
    execute flipACoin
        perform RandomAction
    else:
        perform policyAction
```

The policy action gets the best q-value for the given state. Once tha action is performed, we get the reward which is used to update the q-values for the states. Thus, after plenty of iteration, we will have the true q-values for each state. This will help in training the cab appropriately.

Test Results for the Q-learning agent

Here is the initial configuration for my Q-learning agent. after having brute forced out the parameters alpha, gamma and epsilon, I decided to use the following run configurations:

- Inital Q-value was set to a hypothetical value of 20, which is more than the highest possible positive reward. This 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.
- alpha value was set to 0.9 for a high learning rate, this value was found by iterationg over many possible values for alpha over all the runs.
- Gamma value was set to 0.35 as once agin it provided a really good performance enhancement when testing over all possible range of values.

After having configured these variables, I ran the q-learning agent and it learnt to drive efficiently after 10-15 runs. You can see that effect in the `testresults.txt`. The cab reached the destination almost 85 times over the 100 runs. This performance was comsistent and can be checked by

executing agent.py