Thomas Tilli: Train a smart cab 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 currently uses two types of agent:

- The first 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 Q-learning agent and The normal learning

agent:

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

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?

Implementation of the basic driving agent

I implemented the following naive strategy:

- Pick a random legal action to go ahead.

The implementation looks like:

```
current_env_state = self.env.sense(self)
        # TODO: Update state
        #Environment State  is a set of possible states:
         # 'location': 'heading','destination', 'deadline'


        action = None
        # Set of possible actions Environment.valid_actions[1:]
        possible_actions = []
        if(current_env_state['light'] == 'red'):
            if(current_env_state['oncoming'] != 'left'):
                possible_actions = ['right', None]
        else:
            # traffic ligh is green and now check for oncoming
            #if no oncoming
            if(current_env_state['oncoming'] == 'forward'):
                possible_actions = [ 'forward','right']
            else:
                possible_actions = ['right','forward', 'left']


        # TODO: Select action according to your policy
        if possible_actions != [] :
            action_int =  random.randint(0,len(possible_actions)-1)
            action = possible_actions[action_int]


        # Execute action and get reward
        reward = self.env.act(self, action)
        # TODO: Learn policy based on state, action, reward
        # No learning strategy!
```

How well does it perform?

In order to test the random, agent - I made it run through a total of 100 trials to check how well it will perform. With parameter *enforce_deadline=False* the destination was reached always (see file NaiveAgent_enforce_false.txt). With parameter *enforce_deadline=True* the destination was reached in 30 of 100 trials (see file NaiveAgent_enforce_true.txt).

One of the important things to notice in the result of the random agent is that there are no significant areas of improvement. If we slice over the runs on 10 trial basis, we get random results. There are times when the agent performs well, sometimes it doesn't. This is because of the way the agent has been programmed.

Task 2: Q-learning

Identify and update state

Identify a set of states that you think are appropriate for modeling the driving agent. The main source 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.

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:

- destination
- next waypoint by the planner
- traffic light
- traffic data of oncoming, left, right etc.
- location

Destination changes with every run, therefore it is not a good candidate. Location could not be sensed and it would require to implement a grid as state variable with a large size and many states which would lead to slow learning.

Traffic light and next waypoint are sensed by the planner and are easy to use and are very useful for drive the car and for q-learning. The traffic rules can be learned too using these state variables. Therefore, among all these inputs, I choose the following two state variables:

- traffic light
- next waypoint

Implementation of the Q-Learning agent

The central method of the Q-Learning agent is the update function:

```
def update(self, t):
```

```
        self.next_waypoint = self.planner.next_waypoint()
        ## this is my current state
        self.state = self.createState(self.env.sense(self))
        ##get the current best action based on q table
        action = self.getAction(self.state)
        ##perform the action and now get the reward
        reward = self.env.act(self, action)
        ## in case of initial configuration don't update the q table, else update q table
        if self.previous_reward!= None:
             self.updateQTable(self.previousState,self.previousAction,self.state,
                               self.previous_reward)
        # store the previous action and state so that we can update the q table on the next iteration
        self.previousAction = action
        self.previousState = self.state
        self.previous_reward = reward
        self.cumulativeRewards += reward
```

The method getAction selects the best possible action based on the q-value in the q-table:

```
def getPolicyAndValue(self, state):
        """
        Compute the best action to take in a state.
        input: state
        output: best possible action and qvalue(policy maps states to action)
        Evaluate all legal actions and return the one with the best Q value.
        """
        legalActions = self.getLegalActions(state)
        bestAction = None
        bestQValue=-1000000000
        for action in legalActions:
            if(self.getQValue(state, action) >= bestQValue):
                bestQValue = self.getQValue(state, action)
                bestAction = action
        return bestQValue,bestAction


    def getValue(self, state):
        value,action=self.getPolicyAndValue(state)
        return value


    def getPolicy(self, state):
        value,action=self.getPolicyAndValue(state)
        return action
```

```
    def getAction(self, state):
        # Pick Action
        legalActions = self.getLegalActions(state)
        action = None
        action = self.getPolicy(state)
        return action
```

The q-table is updated in the method updateQTable:

```
  def updateQTable(self, state, action, nextState, reward):
        if((state, action) not in self.qTable):
            self.qTable[(state, action)] = 30.0
        else:
            self.qTable[(state, action)] = self.qTable[(state, action)]
            + self.alpha*(reward + self.discount*self.getValue(nextState)
                                  - self.qTable[(state, action)])
```

Test Results for the Q-learning agent

Initial configuration:

- Initial Q-value was set to a hypothetical value of 20, which is more than the highest possible positive reward.
- Initial alpha value was set to 0.1.
- Initial gamma value was set to 0.1

With this configuration the cab reaches the destination within deadline in 78 of 100 runs and outperforms the random agent by a large extent.

Enhancing the Q-learning agent

I optimized the Q-learning agent by parameter tuning. The following table shows the results of this tuning (random agent included for comparison):

Agent	alpha	gamma	Success	Runs	
Naive Agent			30	100	enforce_deadline=True
			100	100	enforce_deadline=False
Q-learning	0.1	0.1	78	100	
	0.2	0.1	87	100	
	0.3	0.1	92	100	
	0.4	0.1	93	100	
	0.5	0.1	95	100	
	0.6	0.1	97	100	
	0.7	0.1	98	100	
	0.7	0.3	97	100	
	0.8	0.1	98	100	
	0.9	0.1	97	100	

The table shows that the parameter selection is not very critical. Parameter alpha can be in the range (0.5…0.9), gamma=0.1 and in more than 95 iterations the destination is reached within deadline after 3 to 5 initial iterations. The Q-learning agent is very robust and learned quite fast the optimal policy.