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 or None. Due to circulation rules in US, traffic oncoming from the right does not matter.
- Location
- deadline

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. Deadline encodes the distance to destination (deadline = 5*distance), but in my opinion it would not be helpful to include the deadline into the state.

For this model I used the data available at intersections. This includes the traffic light (red/green). The traffic data of oncoming could also be used, but I want keep the state space low, therefore I decided to use only the traffic light and I assumed that the agent will learn the traffic rules either – this have to be proofed!

Furthermore, I used the information of next waypoint. The agent can use this information to act properly to reach the destination in an efficient manner. In my opinion traffic light, next waypoint shall be sufficient to learn the traffic rules and to learn to reach the destination. Location and destination could not be sensed by the agent; therefore,

we don’t lose anything by not including this items into the state. Traffic data of oncoming could be helpful to learn the traffic rules, but traffic light and the rewards earned shall be sufficient to learn the traffic rules too. Since adding traffic data of oncoming to the state will increase the state space it could take longer to learn the traffic rules with this additional information. Including deadline would increase the state space too without adding a great (if any) benefit.

Therefore, I used only traffic light and next waypoint in my state model and in my opinion this is sufficient to learn the traffic rules and to reach the destination within deadline. Since the state space is quite small with this two information I expect that the agent will learn quite fast.

Important note: omitting the traffic data of oncoming seems to make it impossible to learn the correct traffic rules. But in the method act of class Enviroment the reward is calculated as:

```
    light = 'green' if (self.intersections[location].state and heading[1] != 0) or ((not
self.intersections[location].state) and heading[0] != 0) else 'red'


        # Move agent if within bounds and obeys traffic rules
        reward = 0  # reward/penalty
        move_okay = True
        if action == 'forward':
            if light != 'green':
                move_okay = False
        elif action == 'left':
            if light == 'green':
                heading = (heading[1], -heading[0])
            else:
                move_okay = False
        elif action == 'right':
            heading = (-heading[1], heading[0])


        if action is not None:
            if move_okay:
                location = ((location[0] + heading[0] - self.bounds[0]) % (self.bounds[2] -
self.bounds[0] + 1) + self.bounds[0],
                            (location[1] + heading[1] - self.bounds[1]) % (self.bounds[3] -
self.bounds[1] + 1) + self.bounds[1])  # wrap-around
                #if self.bounds[0] <= location[0] <= self.bounds[2] and self.bounds[1] <=
location[1] <= self.bounds[3]:  # bounded
                state['location'] = location
                state['heading'] = heading
                reward = 2 if action == agent.get_next_waypoint() else 0.5
            else:
                reward = -1
        else:
            reward = 1
```

Therefore, violating the traffic rules gives a reward=-1. This makes it possible that the agent learns the traffic rules. To validate that the agent code includes a method checkAction:

```
  def checkAction(self,action,current_env_state):
        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']
        if  not (action==None) and not(action in possible_actions):
            print "illegal action: ",action, "for state",current_env_state
            self.illegalMoves+=1
```

As it is shown below the agent can learn the traffic rules quite fast.

Implementation of the Q-Learning agent

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

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

```
def update(self, t):
        self.steps+=1
        self.next_waypoint = self.planner.next_waypoint()
        ## this is my current state
        current_env_state = self.env.sense(self)
        self.state = self.createState(current_env_state)
        ##get the current best action based on q table
        action = self.getAction(self.state)
        self.checkAction(action,current_env_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
```

```
        if reward>=10: #destination reached
            self.destinationReached+=1
            self.cumulativeSteps+= self.steps
            self.totalRewards+=self.cumulativeRewards
            print "steps to reach destination: ",self.steps," reward: ",self.cumulativeRewards
            print "destination reached: ",self.destinationReached,
                  " illegal moves ",self.illegalMoves,
                  "cumulative steps: ", self.cumulativeSteps,
                  " average number of steps: ",
                  1.0*self.cumulativeSteps/self.destinationReached," average rewards: ",
                  1.0*self.totalRewards/self.destinationReached
            print
        self.previousAction = action
         self.previousState = self.state
        self.previous_reward = reward
        self.cumulativeRewards += reward
```

This method also prints a summary of the results achieved when the agent reaches the destination.

The method getAction selects the next action of the agent:

```
def getAction(self, state):


       # Pick Action
        legalActions = self.getLegalActions(state)
        action = None
        action = self.getPolicy(state)
        p=math.exp(-self.trials/self.epsilon)
        if (self.tossCoin(p)):
        #    print "random choice"
            action = random.choice(legalActions)
        else:
        #    print "policy choice"
            action = self.getPolicy(state)
        return action


def tossCoin(self, p ):
        r = random.random()
        return r <= p
```

The method calculates a p-value which decays over the number of iterations. If the value of epsilon is almost zero, this p value is almost zero. This p value is used by tossing a coin. If the coin returns true, the agent makes a random move, which means he is in explorative mode. Otherwise he is in exploitation mode.

The method getPolicy is implemented as follows:

```
def getPolicy(self, state):
        """
        Compute the best action to take in a state.
        input: state
        output: best possible action(policy maps states to action)
        Working:
        From all the legal actions, return the action that has the bestQvalue.
        if two actions are tied, flip a coin and choose randomly between the two.
        """
        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
            if(self.getQValue(state, action) == bestQValue):
                if(self.tossCoin(.5)):
                    bestQValue = self.getQValue(state, action)
                    bestAction = action
        return bestAction
```

The method best action to take in a state. From all the legal actions, return the action that has the best Q-value. If two actions are tied, flip a coin and choose randomly between the two.

Test Results for the Q-learning agent

Initial configuration:

- Initial Q-value was set to a hypothetical value of 30, 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
- Initial epsilon value was set to 0.001 – this means the agent works only in exploitation mode.

With this configuration the cab reaches the destination within deadline in 75 of 100 runs and outperforms the random agent by a large extent. The number of illegal moves (violating traffic rules) were 179. The analysis of the log file shows, that the agent needs a lot of iterations to learn the traffic rules.

Enhancing the Q-learning agent

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

Agent	alpha	gamma	epsilon	Success	Illegal Moves	average number of steps	average rewards	Runs	
Naive Agent				30				100	enforce_deadline=True
									enforce_deadline=False

Q-learning	0.1	0.1	0.0	74	179	14.31	18.35	100	
	0.2	0.1	0.0	87	91	13.31	17.88	100	
	0.3	0.1	0.0	91	66	13.67	17.18	100	
	0.4	0.1	0.0	93	48	12.86	19.44	100	
	0.5	0.1	0.0	95	39	12.84	16.29	100	
	0.6	0.1	0.0	96	34	12.95	17.97	100	
	0.7	0.1	0.0	97	27	14.09	17.68	100	
	0.8	0.1	0.0	96	26	12.9	17.33	100	
	0.9	0.1	0.0	98	28	13.03	16.47	100	
	1.0	0.1	0.0	97	23	13.61	18.35	100	
	0.8	0.01	0.0	98	22	13.54	17.10	100	
	0.8	0.2	0.0	97	24	11.21	16.65	100	
	0.8	0.1	0.1	98	24	12.15	16.09	100	
	0.8	0.1	0.2	96	23	11.78	16.99	100	
	0.8	0.1	0.3	96	28	13.83	16.88	100	
	0.8	0.1	1.0	97	30	13.26	16.93	100	
	0.8	0.1	5.0	98	40	13.6	16.67	100	
	0.8	0.1	10.0	93	54	13.52	17.41	100	
	0.8	0.1	0	994	31	12.08	17.544	1000	
	0.9	0.1	0	997	29	13.10	16.54		
	0.8	0.1	0.1	995	34	13.28	16.85	1000	
	0.8	0.1	0.5	993	36	13.01	16.95	1000	
	0.8	0.1	1.0	995	29	12.31	17.26	1000	

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 (epsilon=0.0) and in more than 95 iterations the destination is reached within deadline after 3 to 5 initial iterations. Furthermore, the Q-learning agent learns the traffic rules quite fast with this parameter setting. The average number of steps to reach the destination is between 12 to 14 steps, which seems quite good.

For alpha= 0.8 and gamma=0.1 I evaluated also the parameter epsilon. Interestingly for the given problem at hand more explorative behavior (larger epsilon) makes the agent not better. He needs more iterations to learn the traffic rules and to reach the destination.

Based on this trial and error method of parameter optimization I suggest that the best parameter combination is:

- alpha =0.8
- gamma=0.1
- epsilon <0.5

but the exact values are not critical as the results in the table above shows. For the parameter settings above the log file shows, that the agent learns to reach the destination and the traffic rules within a small number of iterations:

```
Simulator.run(): Trial 0

Environment.reset(): Trial set up with start = (3, 3), destination = (8, 5), deadline = 35

RoutePlanner.route_to(): destination = (8, 5)

illegal action:  left for state {'light': 'red', 'oncoming': None, 'right': None, 'left': None}

illegal action:  forward for state {'light': 'red', 'oncoming': None, 'right': None, 'left':
None}

illegal action:  left for state {'light': 'red', 'oncoming': None, 'right': None, 'left': None}
```

```
illegal action:  forward for state {'light': 'red', 'oncoming': None, 'right': None, 'left': None}

illegal action:  left for state {'light': 'red', 'oncoming': None, 'right': None, 'left': None}

illegal action:  forward for state {'light': 'red', 'oncoming': None, 'right': None, 'left': None}

illegal action:  forward for state {'light': 'red', 'oncoming': None, 'right': None, 'left': None}

illegal action:  left for state {'light': 'red', 'oncoming': None, 'right': None, 'left': None}

illegal action:  left for state {'light': 'red', 'oncoming': None, 'right': None, 'left': None}

Environment.reset(): Primary agent could not reach destination within deadline!

Simulator.run(): Trial 1

Environment.reset(): Trial set up with start = (5, 1), destination = (4, 4), deadline = 20

RoutePlanner.route_to(): destination = (4, 4)

illegal action:  forward for state {'light': 'red', 'oncoming': None, 'right': None, 'left': None}

illegal action:  left for state {'light': 'red', 'oncoming': None, 'right': None, 'left': None}

illegal action:  left for state {'light': 'red', 'oncoming': None, 'right': None, 'left': None}

illegal action:  forward for state {'light': 'red', 'oncoming': None, 'right': None, 'left': None}

illegal action:  forward for state {'light': 'red', 'oncoming': None, 'right': None, 'left': None}

Environment.reset(): Primary agent could not reach destination within deadline!

Simulator.run(): Trial 2

Environment.reset(): Trial set up with start = (6, 1), destination = (8, 4), deadline = 25

RoutePlanner.route_to(): destination = (8, 4)

illegal action:  left for state {'light': 'red', 'oncoming': None, 'right': None, 'left': None}

illegal action:  left for state {'light': 'red', 'oncoming': None, 'right': None, 'left': None}

Environment.act(): Primary agent has reached destination!

steps to reach destination:  11  reward:  11.0

destination reached:  1  illegal moves  16 cumulative steps:  11  average number of steps:  11.0
average rewards:  11.0


Simulator.run(): Trial 3

Environment.reset(): Trial set up with start = (2, 2), destination = (8, 2), deadline = 30

RoutePlanner.route_to(): destination = (8, 2)

illegal action:  forward for state {'light': 'red', 'oncoming': None, 'right': None, 'left': None}

illegal action:  forward for state {'light': 'red', 'oncoming': None, 'right': None, 'left': None}

illegal action:  left for state {'light': 'red', 'oncoming': None, 'right': None, 'left': None}

illegal action:  left for state {'light': 'red', 'oncoming': None, 'right': None, 'left': None}

illegal action:  left for state {'light': 'red', 'oncoming': 'right', 'right': None, 'left': None}

illegal action:  forward for state {'light': 'red', 'oncoming': None, 'right': None, 'left': None}
```

```
Environment.reset(): Primary agent could not reach destination within deadline!

Simulator.run(): Trial 4

Environment.reset(): Trial set up with start = (1, 6), destination = (7, 5), deadline = 35

RoutePlanner.route_to(): destination = (7, 5)

illegal action:  left for state {'light': 'red', 'oncoming': None, 'right': None, 'left': None}

illegal action:  left for state {'light': 'red', 'oncoming': None, 'right': None, 'left': None}

illegal action:  left for state {'light': 'red', 'oncoming': None, 'right': None, 'left': None}

illegal action:  left for state {'light': 'red', 'oncoming': None, 'right': None, 'left': None}

Environment.act(): Primary agent has reached destination!

steps to reach destination:  31  reward:  29.0

destination reached:  2  illegal moves  26 cumulative steps:  42  average number of steps:  21.0
average rewards:  20.0
```

After only 3 iterations the agent has learned to reach the destination and after only 5 iterations the agent has learned the traffic rules. After the initial learning phase the agent violates the traffic rules only 2 times in 95 iterations:

```
Simulator.run(): Trial 19

Environment.reset(): Trial set up with start = (1, 1), destination = (7, 1), deadline = 30

RoutePlanner.route_to(): destination = (7, 1)

illegal action:  right for state {'light': 'red', 'oncoming': 'left', 'right': None, 'left':
None}

Environment.act(): Primary agent has reached destination!

steps to reach destination:  16  reward:  22

destination reached:  18  illegal moves  23 cumulative steps:  223  average number of steps:
12.3888888889  average rewards:  15.1666666667
```

```
Simulator.run(): Trial 21

Environment.reset(): Trial set up with start = (2, 5), destination = (7, 3), deadline = 35

RoutePlanner.route_to(): destination = (7, 3)

illegal action:  left for state {'light': 'red', 'oncoming': None, 'right': None, 'left': None}

Environment.act(): Primary agent has reached destination!

steps to reach destination:  16  reward:  21.5

destination reached:  20  illegal moves  24 cumulative steps:  252  average number of steps:
12.6  average rewards:  15.625
```

This means that the agent had learned the traffic rules very well.

The analysis of the logfiles shows that in several cases the agent reaches the destination quite fast, for example:

```
Simulator.run(): Trial 14

Environment.reset(): Trial set up with start = (7, 2), destination = (4, 1), deadline = 20

RoutePlanner.route_to(): destination = (4, 1)
```

```
Environment.act(): Primary agent has reached destination!

steps to reach destination:  7  reward:  9
```

Since the deadline is 5 times the L1 destination between the starting point and the destination, the distance here is 5 and the steps are 7. Quite good. Many other examples like this can be found in the logfile. But in several cases like this one:

```
Simulator.run(): Trial 44

Environment.reset(): Trial set up with start = (7, 5), destination = (1, 2), deadline = 45

RoutePlanner.route_to(): destination = (1, 2)

Environment.act(): Primary agent has reached destination!

steps to reach destination:  25  reward:  34
```

the distance is 9 but the agent needs 25 steps to reach the destination. I observed the behavior of the agent and sometimes the agents seem to cycle, which explains that sometimes the agent needs a quite large number of steps to reach the destination. Therefore, the agent learnt the traffic rules very well but he is not optimal since he does not always reach the destination with a minimal amount of steps. But since the agent consistently reaches the destination within deadline after a very short initial phase in more than 97% of all cases the results are quite good.