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.

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.

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.createStateComplex(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 191. 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	75	191	15.4	18.35	100	
	0.2	0.1	0.0	87	101	14.33	17.88	100	
	0.3	0.1	0.0	92	71	13.45	17.18	100	
	0.4	0.1	0.0	89	60	14.76	19.44	100	
	0.5	0.1	0.0	95	42	12.38	16.29	100	
	0.6	0.1	0.0	95	45	14.28	17.97	100	
	0.7	0.1	0.0	96	35	13.63	17.68	100	
	0.8	0.1	0.0	96	34	13.32	17.33	100	
	0.9	0.1	0.0	97	42	12.67	16.47	100	
	1.0	0.1	0.0	91	31	13.92	18.35	100	
	0.8	0.01	0.0	95	31	13.18	17.10	100	
	0.8	0.2	0.0	97	37	12.98	16.65	100	
	0.8	0.1	0.1	97	37	12.29	16.09	100	
	0.8	0.1	0.2	98	29	13.32	16.99	100	
	0.8	0.1	0.3	99	42	1.63	16.88	100	
	0.8	0.1	1.0	96	34	13.15	16.93	100	
	0.8	0.1	5.0	94	44	13.2	16.67	100	
	0.8	0.1	10.0	90	79	13.65	17.41	100	
	0.8	0.1	0	991	106	13.37	17.544	1000	

	0.9	0.1	0	995	98	12.51	16.54		
	0.8	0.1	0.1	993	113	12.81	16.85	1000	
	0.8	0.1	0.5	995	117	12.81	16.95	1000	
	0.8	0.1	1.0	994	104	13.18	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 1

Environment.reset(): Trial set up with start = (1, 6), destination = (7, 2), deadline = 50

RoutePlanner.route_to(): destination = (7, 2)

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:  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}

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

Simulator.run(): Trial 2

Environment.reset(): Trial set up with start = (4, 2), destination = (5, 5), deadline = 20

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

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

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

Simulator.run(): Trial 3

Environment.reset(): Trial set up with start = (3, 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:  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 4

Environment.reset(): Trial set up with start = (1, 6), destination = (4, 1), deadline = 40

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

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.act(): Primary agent has reached destination!

steps to reach destination:  37  reward:  37.5

destination reached:  2  illegal moves  24 cumulative steps:  38  average number of steps:  19.0
average rewards:  18.75


Simulator.run(): Trial 5

Environment.reset(): Trial set up with start = (7, 2), destination = (5, 5), deadline = 25

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

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

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

steps to reach destination:  20  reward:  26.0

destination reached:  3  illegal moves  25 cumulative steps:  58  average number of steps:
19.3333333333  average rewards:  21.1666666667


Simulator.run(): Trial 6

Environment.reset(): Trial set up with start = (5, 5), destination = (2, 1), deadline = 35

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

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

steps to reach destination:  12  reward:  17
```

```
destination reached:  4  illegal moves  25 cumulative steps:  70  average number of steps:  17.5
average rewards:  20.125


Simulator.run(): Trial 7

Environment.reset(): Trial set up with start = (3, 6), destination = (2, 2), deadline = 25

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

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

steps to reach destination:  5  reward:  6

destination reached:  5  illegal moves  25 cumulative steps:  75  average number of steps:  15.0
average rewards:  17.3


Simulator.run(): Trial 8

Environment.reset(): Trial set up with start = (4, 1), destination = (2, 6), deadline = 35

RoutePlanner.route_to(): destination = (2, 6)

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:  14  reward:  15

destination reached:  6  illegal moves  27 cumulative steps:  89  average number of steps:
14.8333333333  average rewards:  16.9166666667


Simulator.run(): Trial 9

Environment.reset(): Trial set up with start = (7, 4), destination = (5, 2), deadline = 20

RoutePlanner.route_to(): destination = (5, 2)

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

steps to reach destination:  16  reward:  20

destination reached:  7  illegal moves  27 cumulative steps:  105  average number of steps:
15.0  average rewards:  17.3571428571


Simulator.run(): Trial 10

Environment.reset(): Trial set up with start = (1, 3), destination = (3, 5), deadline = 20

RoutePlanner.route_to(): destination = (3, 5)

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

steps to reach destination:  7  reward:  9

destination reached:  8  illegal moves  27 cumulative steps:  112  average number of steps:
14.0  average rewards:  16.3125


Simulator.run(): Trial 11

Environment.reset(): Trial set up with start = (3, 4), destination = (7, 3), deadline = 25

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

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

steps to reach destination:  8  reward:  11
```

```
destination reached:  9  illegal moves  27 cumulative steps:  120  average number of steps:
13.3333333333  average rewards:  15.7222222222


Simulator.run(): Trial 12

Environment.reset(): Trial set up with start = (1, 1), destination = (6, 3), deadline = 35

RoutePlanner.route_to(): destination = (6, 3)

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

steps to reach destination:  16  reward:  17.5

destination reached:  10  illegal moves  27 cumulative steps:  136  average number of steps:
13.6  average rewards:  15.9
```

After only 5 iterations the agent has learned to reach the destination and after only 7 iterations the agent has learned the traffic rules.