UDACITY

PROJECT

Train a Smartcab to Drive

A part of the Machine Learning Engineer Nanodegree Program

PROJECT REVIEW

CODE REVIEW 3

NOTES

smartcab/agent.py 3

```
import random
import math
from environment import Agent, Environment
from planner import RoutePlanner
from simulator import Simulator

class LearningAgent(Agent):
    """ An agent that learns to drive in the Smartcab world.
        This is the object you will be modifying. """

    def __init__(self, env, learning=False, epsilon=1.0, alpha=0.5):
        super(LearningAgent, self).__init__(env)     # Set the agent in the evironment
        self.planner = RoutePlanner(self.env, self)  # Create a route planner
        self.valid_actions = self.env.valid_actions  # The set of valid actions

        # Set parameters of the learning agent
        self.learning = learning # Whether the agent is expected to learn
        self.Q = dict()          # Create a Q-table which will be a dictionary of tuples #to do
        self.epsilon = epsilon   # Random exploration factor
        self.alpha = alpha       # Learning factor

        ###########
        ## TO DO - DONE ##
        ###########
        # Set any additional class parameters as needed
        self.n_trial = 0


    def reset(self, destination=None, testing=False):
        """ The reset function is called at the beginning of each trial.
            'testing' is set to True if testing trials are being used
            once training trials have completed. """

        # Select the destination as the new location to route to
        self.planner.route_to(destination)

        ###########
        ## TO DO DONE ##
        ###########
        # Update epsilon using a decay function of your choice
        # Update additional class parameters as needed
        # If 'testing' is True, set epsilon and alpha to 0
        # else epsilon will be negative


        if testing == True:
            self.epsilon = 0
            self.alpha = 0
        else:
            self.n_trial +=1
            self.epsilon = self.alpha ** self.n_trial
            #self.epsilon -= 0.05 #epsilon for question6

        return None

    def build_state(self):
        """ The build_state function is called when the agent requests data from the
```

```
            environment. The next waypoint, the intersection inputs, and the deadline
            are all features available to the agent. """

        # Collect data about the environment
        waypoint = self.planner.next_waypoint() # The next waypoint
        inputs = self.env.sense(self)           # Visual input - intersection light and traffic
        deadline = self.env.get_deadline(self)  # Remaining deadline

        ###########
        ## TO DO DONE ##
        ###########
        # Set 'state' as a tuple of relevant data for the agent

        # this is my state

        light = inputs['light']
        left = inputs['left']
        oncoming = inputs['oncoming']

        state = (waypoint, light, left, oncoming) #check the env.sense() waypoint left right or

        return state


    def get_maxQ(self, state):
        """ The get_max_Q function is called when the agent is asked to find the
            maximum Q-value of all actions based on the 'state' the smartcab is in. """

        ###########
        ## TO DO DONE ##
        ###########
        # Calculate the maximum Q-value of all actions for a given state
        maxQ_key = max(self.Q[state], key=lambda i:self.Q[state][i])
        maxQ = self.Q[state][maxQ_key]

```

SUGGESTION

Could also simply do `maxQ = max(self.Q[state].values())`

```
        return maxQ


    def createQ(self, state):
        """ The createQ function is called when a state is generated by the agent. """

        ###########
        ## TO DO - DONE ##
        ###########


        if self.learning == True:

            # When learning, check if the 'state' is not in the Q-table
            # If it is not, create a new dictionary for that state
            #   Then, for each action available, set the initial Q-value to 0.0
            if state not in self.Q.keys():
                 self.Q[state] = {None : 0.0, 'left': 0.0, 'right': 0.0, 'forward': 0.0 }
                 print 'createdQ'
```

AWESOME

Awesome! Very precise!

```

        return


    def choose_action(self, state):
        """ The choose_action function is called when the agent is asked to choose
            which action to take, based on the 'state' the smartcab is in. """

        # Set the agent state and default action
        self.state = state
        self.next_waypoint = self.planner.next_waypoint()
        action = None

        ###########
        ## TO DO DONE ##
        ###########
        # When not learning, choose a random action
        if self.learning == False:
                action  = self.valid_actions[random.randint(0, 3)]


```

```
        elif self.learning == True and (self.epsilon * 100 > random.randint(1, 100)):

        # When learning, choose a random action with 'epsilon' probability
        #   Otherwise, choose an action with the highest Q-value for the current state

            action = random.choice(self.valid_actions)
        else:
            ##  Max Q
            #action = self.get_maxQ(state)

            maxQ_value = self.get_maxQ(state)
            candidateQ = {}

            print 'maxQ_value', maxQ_value

            for key, value in self.Q[state].iteritems():
                if value == maxQ_value:
                    candidateQ.update({key:value})

            print 'candidates', candidateQ
            print 'length',len(candidateQ)
            print 'choice', random.choice(candidateQ.keys())

            if maxQ_value is None:
                print 'error'

            if len(candidateQ) == 1:
                action = candidateQ.keys()[0]
            elif len(candidateQ) > 1:
                action = random.choice(candidateQ.keys())
```

SUGGESTION

Although this might be a bit obsessive to use a dictionary here, it definitely does work! Nice job. Might be a bit simpler to use a list comprehension.

```
maxQ = self.get_maxQ(state)
action = random.choice([action for action in self.valid_actions if self.Q[state][action] == maxQ])
```

```


        return action


    def learn(self, state, action, reward):
        """ The learn function is called after the agent completes an action and
            receives an award. This function does not consider future rewards
            when conducting learning. """

        ###########
        ## TO DO  ##
        ###########
        # When learning, implement the value iteration update rule
        #   Use only the learning rate 'alpha' (do not use the discount factor 'gamma')
        if self.learning == True:
                currentQ = self.Q[state][action]
                self.Q[state][action] = reward * self.alpha + currentQ* (1-self.alpha)
        return


    def update(self):
        """ The update function is called when a time step is completed in the
            environment for a given trial. This function will build the agent
            state, choose an action, receive a reward, and learn if enabled. """

        state = self.build_state()          # Get current state
        self.createQ(state)                 # Create 'state' in Q-table
        action = self.choose_action(state)  # Choose an action
        reward = self.env.act(self, action) # Receive a reward
        self.learn(state, action, reward)   # Q-learn

        return


def run():
    """ Driving function for running the simulation.
        Press ESC to close the simulation, or [SPACE] to pause the simulation. """

    ##############
    # Create the environment
    # Flags:
    #   verbose     - set to True to display additional output from the simulation
    #   num_dummies - discrete number of dummy agents in the environment, default is 100
    #   grid_size   - discrete number of intersections (columns, rows), default is (8, 6)
    env = Environment()

    ##############
```

```
    # Create the driving agent
    # Flags:
    #   learning   - set to True to force the driving agent to use Q-learning
    #    * epsilon - continuous value for the exploration factor, default is 1
    #    * alpha   - continuous value for the learning rate, default is 0.5
    agent = env.create_agent(LearningAgent, learning = True, alpha = 0.25, epsilon=0.05)
    #agent = env.create_agent(LearningAgent, learning = False)

    ##############
    # Follow the driving agent
    # Flags:
    #   enforce_deadline - set to True to enforce a deadline metric
    env.set_primary_agent(agent, enforce_deadline = True)

    ##############
    # Create the simulation
    # Flags:
    #   update_delay - continuous time (in seconds) between actions, default is 2.0 seconds
    #   display      - set to False to disable the GUI if PyGame is enabled
    #   log_metrics  - set to True to log trial and simulation results to /logs
    #   optimized    - set to True to change the default log file name
    #sim = Simulator(env, update_delay = 0.01, log_metrics = True)
    sim = Simulator(env, update_delay = 0.001, log_metrics = True,  optimized = True)

    ##############
    # Run the simulator
    # Flags:
    #   tolerance  - epsilon tolerance before beginning testing, default is 0.05
    #   n_test     - discrete number of testing trials to perform, default is 0
    sim.run(n_test = 10, tolerance = 0.10)


if __name__ == '__main__':
    run()

```

▸ visuals.py

▸ smartcab.html

▸ smartcab/simulator.py

▸ smartcab/planner.py

▸ smartcab/environment.py

▸ smartcab/__init__.py

▸ README.md

▸ project_description.md

▸ logs/sim_improved-learning.txt

▸ logs/sim_default-learning.txt