PROJECT

Train a Smartcab to Drive

A part of the Machine Learning Engineer Nanodegree Program

PROJECT REVIEW

CODE REVIEW 6

NOTES

agent.py 6

```
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
        self.epsilon = epsilon   # Random exploration factor
        self.alpha = alpha       # Learning factor

        ###########
        ## TO DO ##
        ###########
        # Set any additional class parameters as needed
        self.iter = 1
        self.a = 0.05
        self.b = 0.05


    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 ##
        ###########
        # 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
        if testing:
            self.epsilon = 0
            self.alpha = 0
        else:
            #self.epsilon -= 0.05
            self.epsilon = math.cos(self.a*self.iter)
            if self.alpha > 0.2:
                self.alpha = max( math.cos(self.b*self.iter), 0.2)
            self.iter += 1

        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

        is_red_light = True if inputs['light'] == 'red' else False
        #left = inputs['left']
        #right = inputs['right']
        #oncoming = inputs['oncoming']
        can_go_left = True if inputs['oncoming'] != 'forward' and inputs['oncoming'] != 'right' else False
```

SUGGESTION

We really should NOT be self creating states here. As the agent should be able to learn your self created states based solely on its process.

(https://discussions.udacity.com/t/number-and-identity-of-states/186637)

The goal of reinforcement learning is to learn these rules without them being hard-coded into the algorithm. Try to include these states directly and see that Q-matrix will converge such as your agent will be able to learn these rules on its own. But I will let this slide, since these are actually the correct way way to do this. But in the future, please don't do this in reinforcement learning.

```
        can_go_right = True if inputs['left'] != 'forward' else False

        ###########
        ## TO DO ##
        ###########
        # Set 'state' as a tuple of relevant data for the agent
        state = (waypoint, is_red_light, can_go_left, can_go_right) # left, right, oncoming)

        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 ##
        ###########
        # Calculate the maximum Q-value of all actions for a given state

        actions = self.Q[state]

        maxQ = None
        max = float('-inf')

        for act in actions.keys():
            if actions[act] > max:
                maxQ = act
                max = actions[act]

        return maxQ
```

REQUIRED

You should be returning the maximum Q-value for the state, not necessarily the action associated for it. As your function right now return something like `right`, as we should be returning a number such as `0.837`. The reason for this is in the `choose_action()` function.

```


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

        ###########
        ## TO DO ##
        ###########
        # When learning, check if the 'state' is not in the Q-table
```

REQUIRED

Make sure you Q-table is only initialized when Learning == True

```
        # 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 self.learning:
            if state not in self.Q:
                self.Q[state] = dict()
                for act in self.valid_actions:
                    self.Q[state][act] = 0.0

```

```
        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 ##
        ###########
        # When not learning, choose a random action
        if not self.learning:
            action = self.valid_actions[random.randint(0,len(self.valid_actions)-1)]
        # When learning, choose a random action with 'epsilon' probability
        #   Otherwise, choose an action with the highest Q-value for the current state
        else:
            if self.epsilon > random.random():
                action = random.choice(self.valid_actions)
            else:
                action = self.get_maxQ(state)
```

REQUIRED

Your agent should be choosing a random action from a choice of actions that have the highest Q-value. For example, since all actions are initialized with a reward of zero, it's possible that all four actions are considered "optimal". Not having the agent choose a random action from this would imply that it always chooses, perhaps, the first available option. This is incorrect behavior:

```
STATE X:
-- 'forward' : 0.00
-- 'left'    : 0.00
-- 'right'   : -1.023
-- 'None'    : 0.00
```

The agent should choose one of 'forward', 'left', or 'None' with equal probability, since they are all considered optimal with the current learned policy. Using a list would be a good idea.

```

        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
```

REQUIRED

Make sure you Q-Learning algorithm is only updated when Learning == True

```
        #   Use only the learning rate 'alpha' (do not use the discount factor 'gamma')
        self.Q[state][action] = (self.Q[state][action] * (1 - self.alpha)) + (reward * self.alpha)
```

AWESOME

Good work with your Bellman equation!

```

        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


```

```python
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=1)

    ##############
    # 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, 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)


if __name__ == '__main__':
    run()
```

▸ report.html

▸ logs/sim_improved-learning.txt

▸ logs/sim_default-learning.txt

Learn the best practices for revising and resubmitting your project.

RETURN TO PATH

Rate this review

Student FAQ