agent.py 4

```
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 environment
        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.train_no = 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 ##
        ###########
        # 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

        import math
        self.train_no = self.train_no + 1
        self.epsilon = math.e**(-0.01 * self.train_no)
        self.alpha = math.e**(-0.001 * (self.train_no))
        if testing:
            self.epsilon = 0
            self.alpha = 0

        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 ##
        ###########
        # Set 'state' as a tuple of relevant data for the agent
        state = (waypoint, inputs['light'], inputs['left'], inputs['oncoming'])
```

需要修改

请参照项目审阅，修改你的代码～

```

        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

        maxAction = max(self.Q[state], key = self.Q[state].get)

        return self.Q[state][maxAction]
```

建议

- 这边可以用 `max(self.Q[state].values())` 直接搞定~

```

    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
        # 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 self.Q.get(state, None) is None:
                self.Q[state] = {}
                for action in self.valid_actions:
                    self.Q[state][action] = 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()


        ###########
        ## TO DO ##
        ###########
        # When not learning, choose a random action
        # When learning, choose a random action with 'epsilon' probability
        #   Otherwise, choose an action with the highest Q-value for the current state

        if not self.learning:
            action = random.choice(self.valid_actions)
        else:
            if random.random() < self.epsilon:
                action = random.choice(self.valid_actions)
            else:
                maxQ = self.get_maxQ(state)
                maxActions = []
                for action_key in self.Q[state]:
                    if self.Q[state][action_key] == maxQ:
                        maxActions.append(action_key)
```

棒极了

对具有相同最大值的action进行了随机选择～

```
                action = random.choice(maxActions)
        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:
            self.Q[state][action] = self.Q[state][action] * (1 - self.alpha) + reward * self.alpha
```

棒极了

公式正确！

```

        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()
    #verbose = True  #added by GardenHo
    env = Environment(verbose=True)

    ##############
    # 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 = 0.5)
    agent = env.create_agent(LearningAgent, learning = True)

    ##############
    # Follow the driving agent
    # Flags:
    #   enforce_deadline - set to True to enforce a deadline metric
    #env.set_primary_agent(agent)
    #enforce_deadline = True #added by GardenHo
    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)
    #display = True #added by GardenHo
    #sim = Simulator(env, update_delay=0.1, display=True, log_metrics=True, optimized=False)
    sim = Simulator(env, update_delay=0.01, display=True, 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()--GardenHo
    #sim.run(n_test=10, tolerance=0.01)
    sim.run(n_test=50, tolerance = 0.1)


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

logs/sim_improved-learning.txt

logs/sim_default-learning.txt