Project 4: Reinforcement Learning

Train a Smartcab How to Drive

Implement a basic driving agent

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?

Answer:

Run the `agent` and see the results

```
LearningAgent.update(): deadline = 11, inputs = {'light': 'green', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 0.0
LearningAgent.update(): deadline = 10, inputs = {'light': 'green', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 0.0
LearningAgent.update(): deadline = 9, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 0.0
LearningAgent.update(): deadline = 8, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 0.0
```

Update state

According to the requirements, I use the following as the attributes:

```
self.state = (
            self.next_waypoint,
            inputs['light'],
            inputs['oncoming'],
            inputs['right'],
            inputs['left'],
            deadline)
```

Random action

In order to producing random action, I create a possible action states array, and use the random fuction to give the random action.

```
# TODO: Select action according to your policy
        possible_action_states = [None, 'forward', 'left', 'right']
        action = possible_action_states[random.randint(0, 3)]
```

Set the deadline to false

```
e.set_primary_agent(a, enforce_deadline=False)  # specify agent to track
```

Speed up

In order to speed up the action, I set the update_delay to 0.01 and select display to false.

```
sim = Simulator(e, update_delay=0.01, display=False)  # create simulator (uses
pygame when display=True, if available)
    # NOTE: To speed up simulation, reduce update_delay and/or set display=False
```

Results

I run the code, and find 3 possible situations:

- case 1: Trial aborted because agent hit hard time limit

```
LearningAgent.update(): deadline = -100, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = left, reward = -1.0
Environment.step(): Primary agent hit hard time limit (-100)! Trial aborted.
```

- case 2: Agent reached destination but out of the deadline

```
LearningAgent.update(): deadline = -82, inputs = {'light': 'green', 'oncoming':
None, 'right': None, 'left': None}, action = forward, reward = 2.0
Environment.act(): Primary agent has reached destination!
```

- case 3: Agent reached destination within the deadline

```
LearningAgent.update(): deadline = 3, inputs = {'light': 'green', 'oncoming': None,
'right': None, 'left': None}, action = left, reward = 2.0
Environment.act(): Primary agent has reached destination!
```

And I run some tries, most possible is case 1 and some times is case 2, seldom is case 3.

The random results I run and save results in `random_action_results.txt`, and the case 3 only 19 times, which the possibility is only **19/100=19%**.

```
python smartcab/agent.py >> random_action_results.txt
```

So the conclusion is that, without Q-learning, the possibility of reaching target locations is very low if only random.

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.

Justify why you picked these set of states, and how they model the agent and its environment.

Answer

According to the below, I think possible states can be used for modeling the driving agent, are the followings:

- Deadline
- Next_waypoint by planner: {None, 'forward', 'left', 'right'}
- Traffic light: {'red','green'}
- Oncoming: {None, 'forward', 'left', 'right'}
- Left: {None, 'forward', 'left', 'right'}
- Right: {None, 'forward', 'left', 'right'}

I will pick the all states except deadline:

Reasons

1. Deadline is a judge of driving, but not a state of the car driving, so I will drop it. And with practise, the performance is bad if I add the deadline into the state.
2. Others are important state of car driving, so I will use all of them to build the Q[(state,action)] table.

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?

- Implement Q-Learning agent by a new class named `QLearningAgent`.
- Initialize the Q values table,

```
for action in self.possible_actions:
            if(state, action) not in self.Q:
                self.Q[(state, action)] = 100
```

- Caculate the best value for next action, and using softmax function,

```
        Q_best_value = [self.Q[(state, None)], self.Q[(state, 'forward')],
self.Q[(state, 'left')], self.Q[(state, 'right')]]
        # Softmax
        Q_best_value = np.exp(Q_best_value) / np.sum(np.exp(Q_best_value), axis=0)
```

- Choose the largest Q value action between the possible actions,

```
        action = self.possible_actions[np.argmax(Q_best_value)]
```

- Update the Q value by reward, state, action and learning rate.

```
self.Q[(state,action)] = reward * self.learning_rate + ( 1 - self.learning_rate ) *
self.Q[(state,action)]
```

Results

After Implement the Q-Learning agent, and change the agent from random agent to this agent, we can run and save the results in `implement_q_learning.txt`:

```
python smartcab/agent.py >> implement_q_learning.txt
```

And the result which reached the destination is **86/100=86%**. So with Q Learning the car can smart driving according to the environments.

Reasons of changes in behavior observed

- In the random agent, the car do the action according to random of state, so it don't take the environments into consideration, and the accuracy is very low, and the performance will don't be better after more training.
- In the Q-learning agent, the car do the action according to choosing the best next step state Q value, which is calculated by the Bellman equation. And after some training steps, the Q value will tend to stable.

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?

Case 1: Add the deadline to state

After Implement the Q-Learning agent, and add the deadline to the state, we can run and save the results in `enhance_1.txt`:

```
python smartcab/agent.py >> enhance_1.txt
```

And the result which reached the destination is **23/100=23%**. So add the deadline is a very bad choise. It will not enhance the driving agent.

Case 2: Remove the other cars' states

After Change the Q-Learning agent, and remove other car's states, we can run and save the results in `enhance_2.txt`:

```
python smartcab/agent.py >> enhance_2.txt
```

And the result which reached the destination is **87/100=87%**. So this action don't change the result much, and a little bit better, according to Occam's Razor, it make the Q table more simple, which can be applied to enhance.

Case 3: Change the initial Q values

I change the initial Q value from 100 to 20, and run & save in `enhance_3.txt`,

```
python smartcab/agent.py >> enhance_3.txt
```

And the result which reached the destination is **98/100=98%**. Which has a very high rate to reach the destination within time. So when we change the initial Q value to a suitable value, it can raise the accuracy very much!

Case 4: Change the learning rate

I change the learning rate from 0.9 to 0.8, and I run and save the resultsin `enhance_4.txt`,

```
python smartcab/agent.py >> enhance_4.txt
```

And the result which reached the destination is **95/100=95%**. Which changed from 98% to 95%, so it does worse.

Conclusion

After trying different parameters, we can get a optimal policy. We change the initial Q value from 100 to 20, and the accuracy raise to 98%, And the last 10 tries are all reached the destination. And if the actions break the traffic laws, which will cause the penalties, so according to choosing the max Q value, it will not break the traffic laws.