Train a smartcab how to drive

Reinforcement Learning Project

Implement a basic driving agent

Implement the basic driving agent, which processes the following inputs at each time step:

1. Next waypoint location, relative to its current location and heading,
2. Intersection state (traffic light and presence of cars), and,
3. 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.

1. Does it eventually make it to the target location?

> Yes, it does (well past the deadline of course). The agent behavior appears to be random, as intended. For example, there are time steps when it chooses to stay in place, even though the light is green and no other cars are in the intersection. Several times the agent was very close to the destination, but chose actions that moved it farther away.

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.

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

> The model uses a combination of the 'light' and 'oncoming' outputs of function Environment.sense() and the next direction to proceed from RoutePlanner.next_waypoint(). This produces a vector containing info about the traffic light, oncoming traffic at the intersection as a boolean, and the direction of the destination from the agent's current location.
>
> - The intersection data will give the agent a concise view of the current environment and will allow it to learn the general traffic rules for each variation of environment it observes.
> - The direction provided as 'next_waypoint' will serve as guide for the direction of the destination, which the agent can learn to use as its next action (or not) in the context of the current intersection environment.

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.

3. What changes do you notice in the agent's behavior?

> With Q-values initialized to 0, alpha 0.2, gamma 0.8 -- The agent chooses the first action in the Q table during each time step when the first valid action is None (valid actions in order: [None, 'forward', 'left', 'right']).
>
> Changing the order of valid actions to ['forward', 'left', 'right', None], the agent learns to simply go forward at green and turn right at red. This leads to some wins, but the agent doesn't really learn optimal actions. It operatates merely at the whim of each traffic light it encounters, believing that going forward at every green light and going right at every red will maximize its rewards/Q-values.

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.

4. 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?

The final version of the agent changes the discount factor gamma and the default Q-value initialization (initialize Q-values to 3, alpha 0.2, gamma 0.5).

During the first few times that a specific state is encountered, these parameter settings will decay the Q-value for any action that doesn't return a reward of 2 (the max reward possible, not counting the reward for reaching destination).

This allows the agent to explore the state-action space and not get stuck on the first possible action that generates a positive reward. The agent quickly learns the optimal action at each time step, and succeeds during almost 100% of trials after just 10-20 trials.

Prior to settling on this final version, other parameter tweaks were attempted, including:

- **E-greedy Exploration**: Used a percentage value epsilon to determine if the agent takes a random action or maximum value action at each time step. Started with large epsilon (.5 to .9) to encourage a lot of random action during first few trials to explore the state-action space, then slowly decayed epsilon during later trials. Also decayed epsilon within each time step to create sense of urgency as deadline counted down. This method did actually cause the agent to learn a near-optimal policy, but took many trials (>50) to do so.
- **Learning rate alpha**: Experimented with alpha 0.1 and 0.9, but in isolation of other tweaks this did not improve the agent's learning much.
- **Discount factor gamma**: Reduced gamma to 0.5 to put less weight on the max Q-value of the next state. This seemed to improve the agent but was not super useful until tuned in coordination with the default Q-values.
- **Q-value initialization**: Tried out large initial values, but during the first few time steps and trials it caused actions with reward 2 to produce a lower Q-value. After trial & error and tinkering to find an initial value that would always increase default Q-values when choosing actions with reward 2 and decrease for anything else, an initialization of Q=3 and gamma=0.5 was settled on.

5. 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?

Defining the optimal action at each time step as 'next_waypoint' if the action is allowed and None if it isn't, the agent learns close to an optimal policy.

For example, this run achieved 47 successes in 50 trials, taking optimal actions 73% of the time...

Results: list of (Win/Failed, deadline) for each trial

- [('Win!', 25), ('Win!', 45), ('Win!', 25), ('Win!', 40), ('Win!', 45), ('Failed', 35), ('Win!', 20), ('Win!', 20), ('Win!', 35), ('Failed', 35), ('Failed', 25), ('Win!', 35), ('Win!', 60), ('Win!', 20), ('Win!', 35), ('Win!', 20), ('Win!', 25), ('Win!', 35), ('Win!', 20), ('Win!', 25), ('Win!', 25), ('Win!', 20), ('Win!', 20), ('Win!', 20), ('Win!', 20), ('Win!', 20), ('Win!', 20), ('Win!', 20), ('Win!', 45), ('Win!', 25), ('Win!', 25), ('Win!', 20), ('Win!', 40), ('Win!', 25), ('Win!', 55), ('Win!', 35), ('Win!', 20), ('Win!', 20), ('Win!', 25), ('Win!', 20), ('Win!', 25), ('Win!', 35), ('Win!', 20), ('Win!', 25), ('Win!', 20), ('Win!', 30), ('Win!', 30), ('Win!', 35), ('Win!', 20), ('Win!', 40)]

Optimal policy used: list of (optimal action taken, total actions) for each trial

- [(5, 16), (13, 17), (15, 18), (11, 17), (10, 11), (3, 4), (7, 9), (10, 14), (5, 7), (13, 21), (11, 17), (7, 9), (9, 12), (5, 5), (12, 15), (7, 8), (11, 15), (13, 17), (6, 8), (5, 6), (11, 14), (10, 12), (9, 12), (9, 10), (8, 12), (8, 13), (8, 9), (9, 13), (10, 12), (25, 33), (7, 9), (11, 15), (9, 15), (7, 10), (13, 16), (10, 15), (12, 14), (5, 7), (14, 21), (5, 8), (9, 11), (6, 8), (14, 22), (12, 14), (23, 33), (7, 8), (9, 12)]

Q table: key is state (light, oncoming, left, right, next_waypoint), value is actions [None, 'forward', 'left', 'right']

- Q: {('green', 'oncoming', 'left'): [3, 3, 3, 3],
  ('green', 'oncoming', None): [3, 3, 3, 3],
  ('green', 'oncoming', 'right'): [3, 3, 3, 3],
  ('red', 'oncoming', None): [3, 3, 3, 3],
  ('green', 'no_oncoming', None): [3, 3, 3, 3],
  ('red', 'oncoming', 'forward'): [2.729000000000001, 2.5000000000000004, 2.5000000000000004, 2.739851073729147],
  ('red', 'oncoming', 'right'): [2.9000000000000004, 2.5000000000000004, 2.5000000000000004, 3.847420224361293],
  ('red', 'no_oncoming', 'right'): [2.9000000000000004, 2.5000000000000004, 2.5000000000000004, 4.374941309831393],
  ('green', 'oncoming', 'forward'): [2.9000000000000004, 3.2153700621398165, 3, 3],
  ('green', 'no_oncoming', 'left'): [2.9000000000000004, 2.8000000000000003, 5.949170798811015, 3],
  ('red', 'no_oncoming', None): [3, 3, 3, 3],
  ('red', 'oncoming', 'left'): [2.9000000000000004, 2.5000000000000004, 2.5000000000000004, 2.9531088015190305],
  ('green', 'no_oncoming', 'right'): [2.9000000000000004, 2.8000000000000003, 2.8000000000000003, 4.50839678065863],
  ('red', 'no_oncoming', 'forward'): [2.1094189891315134, 2.0500000000000007, 2.0500000000000007, 2.8536557119728854],
  ('green', 'no_oncoming', 'forward'): [2.9000000000000004, 6.964849941005833, 3, 3],
  ('red', 'no_oncoming', 'left'): [2.254186582832901, 2.0500000000000007, 2.0500000000000007, 2.68903986447919]}

In [49]:
```
# plot the pct of optimal actions per trial: optimal action / total actions
optimal_moves.plot().axis(v)
```

Out[49]: [0, 47, 0, 1]