Machine Learning Engineer Nanodegree

Project 4: Train a Smartcab to Drive

Jou-ching Sung

Implement a basic driving agent

I implemented a basic driving agent that takes in the default information about state, and just outputs a random action: one of None, forward, left, right.

Effectively, the agent performs a near-random walk ("near-random" because it cannot move backwards). The agent will eventually reach the destination, via randomness.

Identify and Update State

I used a combination of the waypoint and the environment inputs (light, oncoming, left, right) to represent the state, with the following optimizations:

- Ignore traffic on the right
    - If the agent has a green light, all traffic on the right must yield to the agent. If the agent has a red light, the traffic on the right has no effect on the agent's valid actions.
- If light is green, ignore traffic on the left
    - If light is green, then set inputs['left'] to None
- If light is red, ignore oncoming traffic
    - If light is red, then set inputs['oncoming'] to None

The above logic is implemented in the function compress_sa() within the LearningAgent class.

With the above state compression logic, we can represent our valid state space with a smaller Q-table, allowing more efficient and faster learning. For example, by ignoring all traffic on the right for our state representation, our Q-value table can be reduced by 4X (there are 4 possible values for inputs['right']).

Implement Q-Learning

After implementing Q-learning, I noticed the agent gradually learns to move towards the objective, and gradually learns traffic rules. Early in the learning process, the agent performs

random walks, but the probability of random actions decreases as the agent gains more "experience" through the Q-learning process.

Enhance the Driving Agent

In addition to the basic Q-learning algorithm, below are the additional enhancements I made:

- Added tracking of global time for the agent, i.e. how long the agent has been "alive"/operational
    - Tracking the agent's total alive-time is more relevant than tracking the time of each individual trial, since the agent learns over its entire lifetime, rather than only over a single trial
- Epsilon-greedy exploration: Epsilon, defined as the probability of taking a random action, decreases sigmoidally as a function of global time
    - Epsilon = 1 - sigmoid(RATE*global_time - OFFSET)
- Implemented learning rate decay
- Tweaked the relevant parameters to allow agent to learn near-optimal policy within 100 trials

Implementing the above enhancements, the final agent performs well. After 100 trials, the final 40-50 trials are always consistently successful. Success is defined as reaching the objective destination within the deadline, and not incurring any negative rewards. As expected, many of the initial trials failed, but the agent learned over time and improved its performance at the end.

Note in agent.py, we print the net reward and number of penalties incurred for each trial.

Based on the above, we can conclude the agent does get close to finding the optimal policy. The final 40-50 trials always has the agent reaching the destination within the deadline, and no penalties (negative rewards) are incurred.