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 optimization:

- 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. Thus we can ignore traffic on the right, assuming the other agents follow traffic rules.

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. 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
- Searched through the relevant parameter combinations to allow agent to learn near-optimal policy within 100 trials
    - In 'smartcab/find_hyper_parameters.py', we can find the code to sweep the following parameters: sigmoid_offset, sigmoid_rate, alpha_decay, gamma
    - I modified the constructor of the 'LearningAgent' class in 'smartcab/agent.py', to allow custom hyper-parameter values
    - I also modified 'smartcab/environment.py' and 'smartcab/simulator.py' to add a "score" metric (percentage of successful trials over the last 30 trials), and propagate the score value appropriately
    - The range of values I chose to sweep through for each hyper-parameter were determined by first manually running the 100 trials for random values of the hyper-parameters, and finding a rough range of values that produces qualitatively good results
    - The final results of the hyper-parameter sweep are located in 'hyper_params.csv'
    - Choosing the final set of hyper-parameters, there are many combinations of hyper-parameters which have a perfect score of 1. I chose a set of hyper-parameters that resulted in the most randomness in the early trials (maximize exploration while still achieving a high score), i.e. high sigmoid_offset, low sigmoid_rate, low alpha_decay. Doing so would result in the most consistent scores across multiple 100-trial runs.
    - The final hyper-parameters I chose are: sigmoid_offset=8, sigmoid_rate=0.01, alpha_decay=0.1, gamma=0.5

Implementing the above enhancements, the final agent performs well. After 100 trials, the final 30 trials are always consistently successful (28+ successful trials in the final 30 trials). 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.

To determine if the agent finds the “optimal” policy, we can slowly step through the agent’s actions, using the agent’s final learned parameters. In order to accomplish this, I modified ‘smartcab/simulator.py’ to force update_delay=1.0 in the final three trials, so we can slowly observe the agent’s behavior, and pause/resume if necessary.

From observing the agent’s final learned behavior, I see the agent does *not* learn the optimal policy, even though it learns a “good enough” policy to reach its destination within the allotted time, most of the time. The agent is unable to learn the optimal policy due to (1) limitations of the simplistic route planner, and (2) the reward structure which rewards the agent for following the route planner while not breaking traffic rules, while *not* rewarding any actions that deviate from the route planner but may reduce the manhattan distance to the agent’s destination. To elaborate:

1. *Limitations of the route planner*: If the destination is located to the southeast of the agent and the agent is facing west, the agent will perform a right-U-turn then head east, and finally south. However, the optimal policy in this scenario is to make a *left*-U-turn in the beginning. The planner’s hard-coded logic to make a right-U-turn causes the agent to make non-optimal decisions in this case.
2. *Reward structure*: The reward structure influences the final policy the agent learns. The current reward structure pushes the agent to learn a policy that follows the planner’s waypoints while not breaking traffic rules. However, there are cases where the optimal policy involves deviating from the planner’s waypoint. For example, if the destination is located southeast of the agent and the agent is heading east, but there is a red light, the planner’s waypoint will still be “east”. In this case, the agent’s optimal action would be “right” if the traffic allows, to reduce the manhattan distance to the destination. However, the learned policy will cause the agent to stay put and wait for the red light.

Based on the above, potential enhancements can be made to the planner and/or reward structure, to allow the agent to learn a policy that is closer to the “optimal” one.