SmartCab Reinforcement Learning Report

QUESTION: Observe what you see with the agent's behavior as it takes random actions. Does the smart cab eventually make it to the destination? Are there any other interesting observations to note?

When taking turns randomly, the car does not reliably reach its destination. Sometimes it gets lucky, but it usually runs out of turns before successfully reaching the destination. After watching the visual simulation in Pygame, it appears rare for the car to encounter traffic from the left or right. These inputs are unlikely to be useful when implementing the learning algorithm.

After running 100 iterations (enforce_deadline=True) of the trial with random actions, only 18 (18%) reached the destination.

QUESTION: What states have you identified that are appropriate for modeling the smart cab and environment? Why do you believe each of these states to be appropriate for this problem?

I set the state to a tuple with the with the indices representing Light, Waypoint, and Oncoming traffic. In total, this provides 24 possible combinations for the state. I chose this pattern because it maximizes information while minimizing complexity. For instance, I could have added left and right traffic to the tuple to bring it to 384 possible combinations. It likely that most of the information would be useless to the driver and add unnecessary complexity to the Q mapping.

```
example_state = ( light, waypoint, oncoming )
>>> ('red', 'left', None)
```

- Light: Necessary for training the cab to obey traffic rules.
- Oncoming: Necessary for avoiding traffic accidents. Of the three traffic inputs, oncoming appears to contain the most information.
- Waypoint: Necessary for providing a general sense of direction. I imagine this input like a passenger in the back seat giving left/right/forward directions to the driver. Without this data, the driver would still be randomly searching for the endpoint.

OPTIONAL: How many states in total exist for the smart cab in this environment? Does this number seem reasonable given that the goal of Q-Learning is to learn and make informed decisions about each state? Why or why not?

Total features can be determined by multiplying the total count of possible values from each input.

inputs = light(2), waypoint(3), oncoming(4), left(4), right(4)

With the inputs above, possible state combinations are 2x3x4x4x4 = 384.

If you add another state for each number of turns left in deadline(50), possible states go up to 19,200.

With 100 turns to learn, 19,200 is not reasonable. The driver would not enter many of the possible states during the course of the trials. Most of this information should be consolidated into a simplified structure.

QUESTION: What changes do you notice in the agent's behavior when compared to the basic driving agent when random actions were always taken? Why is this behavior occurring?

Updating the Q table with the current reward value creates a much more effective driver. The algorithm does not consider the impact of future moves, but it still has a much higher success rate compared to complete randomness. Visually, the smartcab appears to be more patient and deliberate with its actions. It drives in straight lines and is not afraid to wait at traffic lights. After 100 trials, the driver failed to reach the destination only 10 times for a 90% success rate.

QUESTION: Report the different values for the parameters tuned in your basic implementation of Q-Learning. For which set of parameters does the agent perform best? How well does the final driving agent perform?

- α (alpha) is the learning rate. A higher alpha value will make the algorithm more sensitive to changes in rewards.
- γ (gamma) is the discount rate on future rewards. A higher gamma will make the algorithm more sensitive to rewards from future actions.
- ε (epsilon) is the exploration rate. A higher epsilon will cause the algorithm to make more random choices.

I ran experiments with the tuning parameters listed below. Each experiment ran with 100 trials.

α=0.5, γ=0.5, ε=0.5 (Random/Explorer Learner)

Setting all parameters to 0.5 resulted in a 31 failed runs, or a success rate of 69%. I suspect the epsilon value of 0.5 is making the algorithm too random. Only half of the decisions are being made with the Q table, while the other half are random exploratory moves.

α=0.5, γ=0.5, ε=0.2 (Balanced Learner)

An epsilon value 0.2 improves learning performance by a significant margin. The failed runs were reduced to only 12, for a total success rate of

88%. The balanced learner puts the same amount of weight on the reward in the current move as it does for the future move.

α=0.8, γ=0.1, ε=0.1 (Short-Term/Greedy Learner)

Short term learning worked well, with only 5 failed runs or a 95% success rate. The low epsilon value makes this a greedy combination that does make exploratory moves often. Comparatively, this combination leads to a policy with more negative Q value rewards, or discouragement from performing bad moves.

α=0.2, γ=0.9, ε=0.1 (Long-Term/Greedy Learner)

This policy leads to larger positive Q values, or encouragement for successful moves. It failed 6 runs, for a 94% success rate. By the end of the 100 trials, this policy reaches the destination quickly and rarely incurs negative rewards with its choices.

QUESTION: 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? How would you describe an optimal policy for this problem?

Towards the end of the 100 trials, the agent is far less likely fail and the total reward received was always greater than 0. Most failed runs are found within the first 20 attempts. In fact, it is common for the driver to avoid all negative rewards during many of the later trials.

The agent learns very quickly with the input states provided, which leads me to believe that a short term policy is optimal. It only takes about 5 to 10 turns for the Q table to be filled with effective knowledge about the most common turns. There does not appear to be any intricate patters between the states that justify a more complex model. In other words, the learning rate (α) should be high, while the gamma (γ) and epsilon (ε) should be low. This combination will force the algorithm to weigh its decisions mostly on the current move. Only during times when the next move has a significant reward will the resulting action be influenced.