P4: Train a Smartcab 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?

The basic agent was implemented in **b_agent.py**. The following pseudo code describes the logic of how it determines what action to take.

```
Initialise list of valid actions to empty

if light is GREEN and ONCOMING traffic are going FORWARD
    valid actions are 'Forward' or 'Right'
otherwise
    valid actions are 'Forward', 'Right' or 'Left'

if light is RED and ONCOMING traffic are turning LEFT or traffic to the LEFT is moving FORWARD
    valid actions are 'None', 'Forward' or 'Left'
otherwise
    valid actions are 'None', 'Forward', 'Left', or 'Right'

If we have exceeded the DEADLINE and the WAYPOINT is a valid action
    then choose the WAYPOINT as the ACTION
otherwise
    if there are one or more valid actions to choose from, then select one at random
catchall
```

if there are no valid actions then choose the previous action used

'None' was not considered a valid action when the light is green, as you cannot sit in the middle of the road when you have the right of way and do nothing (unless broken down). The state was set to either of 'waypoint', 'random' or 'lastaction' as a means to document the action decision process.

To the code, I also added some counters, used to count the number of trials that complete within the deadline and the number of trials that exceed the deadline. I also count the number of iterations taken for each successful and unsuccessful trial to allow calculation of averages (see table below).

I ran the basic agent (**b_agent.py**) with differing number of trials, specifically 100x, 500x and 1000x, and observed the behaviour for each. The following table displays some summary statistics about each run:

Statistics from 100x Trial Run:
agent::run() Statistics: Trip successes=20 Number of Moves=320 Average number moves per Trip=16.0
agent::run() Statistics: Trip failures=80 Number of Moves=3647 Average number moves per Trip=45.5875

Statistics from 500x Trial Run:
agent::run() Statistics: Trip successes=125 Number of Moves=2225 Average number moves per Trip=17.8
agent::run() Statistics: Trip failures=375 Number of Moves=3647 Average number moves per Trip=45.2746666667

Statistics from 1000x Trial Run:
agent::run() Statistics: Trip successes=239 Number of Moves=4375 Average number moves per Trip=18.3054393305
agent::run() Statistics: Trip failures=761 Number of Moves=34372 Average number moves per Trip=45.1668856767

As can be seen, in all cases the SmartCab reached its destination, however it could only achieve a successful trip approximately 20%-25% of the time. A successful trip is defined as arriving at the destination before the deadline is exceeded. See attached file **b_agent_log.txt** for the log of the full 500x trial run.

Unsurprisingly the SmartCab appears to move in a random manner, and more often than not it gets to the destination by accident rather than by design. Many times the SmartCab can be observed getting to within one intersection of the destination and then turn away and explore another area before eventually finding its way to the destination.

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.

At the start I choose possible states from all information available when the agent.update() function receives control, these include the following:

- **traffic light state** – either 'red' or 'green'
- **oncoming state** – one of “None”, “forward”, “left” or “right”
- **right state** - one of “None”, “forward”, “left” or “right”
- **left state** - one of “None”, “left”, “right” or “forward”
- **waypoint state** - one “forward”, “left” or “right”
- **lastaction** - one of “None”, “forward”, “left” or “right”
- **starting position**
- **current location**

In addition to those, I also experimented with some states I created:

- a static constant – eg. 99
- some random states – random binary number, either 0 or 1
- the number of iterations in the current trial
- deadline of the trial and whether it has been exceeded or not
- the difference between the deadline and current number of iterations

For non-numeric states I converted the values into a single digit numeric state, the states were then combined into a simple numeric “state string”. This would then serve as the key in the Q-Matrix, thus allowing a simple single value for the key, rather than a long list of words and numbers.

My **createState** function (see **ReinforcementLearning.py**) could be updated easily by commenting/uncommenting various combinations of states in-order to form new “state strings” to allow testing of various combinations.

For example, if I were to select the following states to make up my “state string” and they have the indicated values:

- traffic light state - “green”

- right state - “left”
- left state - “forward”
- lastaction state - “right”

I would get a “state string” of “2324” for the indicated input values. The numeric value assigned to represent each state value is the index of the state value from a list of all possible values for the state (index values start at 1).

After running numerous trials with different combinations of states, typically 6-10 states making up the “state string”, I found I was not getting more than 30% of trials completing before the deadline. So I decided to start reducing the size of my “state string” ie. use less states in-order to reduce the size of the overall state space and thereby potentially learn quicker and better by needing less iterations in-order to explore the state space.

Eventually after many trials I found a combination of just two states offered the best success rate, they being the **traffic light state** and the **waypoint state**. This gives rise to a state space size of 24 (2 (possible traffic light states) * 3 (possible waypoint states) * 4 actions). As the state space is quite small, all possible combinations of state and actions can be seen very quickly. After this point any future action chosen will be based upon the result of past actions (rewards received) rather than a random action, thus the most optimal action can be chosen given the agents knowledge at that particular point in time.

These two states model the agent in the most fundamental way, in that, the traffic light state indicates whether or not the SmartCab is moving or not and the waypoint state points in the general direction of the destination, although it might not be the optimal direction to the destination. (that's for the agent to find out).

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?

I have implemented the following form of the Q-Learning algorithm:

- **$Q(s, a) \leftarrow (1 - alpha) * Q(s, a) + alpha * (r + gamma * \max_{a'} Q(s', a'))$**

Where:

- **alpha** – the learning rate, set between 0 and 1. A setting of 0 means that Q-Values are never updated hence nothing is learned. Setting a high value >0.9 means that learning can occur quickly.
- **r** – the current reward received
- **gamma** – the discount factor, set between 0 and 1. This models the fact that future rewards are worth less than immediate rewards. If equal 1, the agent will value future rewards just as much as current rewards. A value of 0 will cause the agent to only value immediate rewards.
- **$\max_{a'} Q(s', a')$** – the estimate of optimal future value.
- **Q(s, a)** – current Q-Value for the given state and action pair.

I used alpha=0.90 and gamma=0.20 to have the Q-Learner learn quickly and place more value on current rewards.

I have created a function called **chooseActionUsingMaximumQValue** (see **ReinforcementLearning.py**) which is used to select the action for a given state that has the largest Q-Value.

Compared to the basic agent (**b_agent.py**) which picked actions at random, the Q-Learning agent moves the SmartCab in what appears a much more deliberate manner. It looks like the agent has an idea of how to get to the destination, the movements don't appear as random and arriving at the destination doesn't look like it got there by accident. It tends to travel a route in a straight line longer before taking turns, whereas the random agent rarely moved more than one intersection in a straight line, most of the time it just takes random turns. The Q-Learning agent tends to make turns towards the destination, rather than taking turns for the sake of random exploration.

As the Q-Learning agent moves about, it gets feedback (rewards) after each move including when it makes errors, so over time as it explores the state space it "learns" from the feedback and is able to pick the most optimal action/move.

Output from a 100x trial run, choosing an action using Maximum Q-Value can be found in the following file **Q_Learning_Maximum_QValue_log.txt**. Edited sample output can be seen below:

```
========== RESTART: G:\ML_Nanodegree\smartcab\smartcab\agent.py ==========
---> ReinforcementLearning::__init__() Q initialised.
---> ReinforcementLearning::__init__() Initialised with the following parameters:
---> ReinforcementLearning::__init__() Actions=[None, 'forward', 'left', 'right'].
---> ReinforcementLearning::__init__() alpha=0.90.
---> ReinforcementLearning::__init__() epsilon=0.25.
---> ReinforcementLearning::__init__() gamma=0.20.
---> ReinforcementLearning::__init__() alpha decay=False.
---> ReinforcementLearning::__init__() temperature=0.275.
.
---> ReinforcementLearning::save_obj() As Requested. No Save Done.
---> agent::run() Number of Entries in Q Matrix=24
---> agent::run() Q Matrix Contents...
(11, None) 1.8855973138
(11, 'forward') -1
(11, 'left') -1
(11, 'right') 0.5
(12, None) 1
(12, 'forward') -1
(12, 'left') -1
(12, 'right') 2.60228606868
(13, None) 1
(13, 'forward') -1
(13, 'left') -0.695081635236
(13, 'right') 1.47522601835
(21, None) 1
(21, 'forward') 0.5
(21, 'left') 2.58252612439
(21, 'right') 0.5
(22, None) 1
(22, 'forward') 0.5
(22, 'left') 0.5
(22, 'right') 2.8680844537
(23, None) 1
(23, 'forward') 2.55616474534
(23, 'left') 0.5
(23, 'right') 0.5

---> agent::run() Statistics: Trip successes=45 Number of Moves=655 Average number moves per
Trip=14.5555555556
---> agent::run() Statistics: Trip failures=55 Number of Moves=1595 Average number moves per
Trip=29.0
```

Comparing the above to the basic agent we can see a significant increase in the number of successful trips (45 compared to 20) and a reduction in the average number of moves taken in a successful trip (14.5 compared to 16.0).

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?

To the original implementation of Q-Learning I have made the following enhancements:

- Tuned the alpha parameter – the learning rate
- Tuned the gamma parameter – the discount factor
- Implemented a function to decay the learning rate on each iteration
- Implemented three action selection algorithms

Parameter Tuning

Tuning the alpha and gamma parameters was done by running the agent multiple times with different combinations of these parameters and observing the results, ie. did the number of successful trips get better or worse. Generally I found changes in alpha and gamma did not make significant differences in the success rate of my 100x trials. Anecdotally I have different values of alpha/gamma improve the average number of moves taken to successfully reach the destination. I have chosen to use **alpha=0.75** and **gamma=0.333** for all subsequent testing.

Learning Rate Decay

I also experimented with decaying the learning rate (alpha), through manipulating the **decay=True/False** parameter of my learner. The learning rate (alpha) was decayed using the following formula:

- **1 / 1 + counter**, where counter is the number of the iterations in the trial so far

From my experiments I observed no significant difference when decaying the learning rate and not decaying the learning rate. This is probably due to the fact that the agent is able to quickly learn the relatively small state space. I choose to decay the learning rate for all subsequent experiments.

Action Selection Algorithms

I implemented three action selection algorithms to determine which would give me the best results, ie. which would assist in achieving a higher trip success rate (a trip success is defined as the SmartCab reaching the destination within the deadline. I consider arriving at the destination on exactly the deadline as a success).

The three action selection algorithms tested were:

1. **Epsilon Greedy** – the epsilon greedy strategy is to select the "greedy" action, one that maximises Q(s, a) all but epsilon of the time and to select a random action epsilon of the time, independent of the action value estimates, where 0 <= epsilon <= 1.

2. **Maximum Q-Value** – for a given state all associated Q-Values for the corresponding actions are found. The action with the largest Q-Value is chosen, if more than one action has the largest Q-Value then an action is chosen at random from among them.

3. **Softmax** – Uses a Gibbs/Boltzmann distribution, where the probability of selecting action **a** in state **s** is proportional to **e**Q(s, a)/T**, that is, in state **s**, the agent selects action **a** with probability:

    - **(e**Q(s, a)/T) / (Sum$_a$ e**Q(s, a)/T)**

    Where T>0 is the temperature specifying how randomly values should be chosen. When T is high, the actions are chosen in almost equal amounts. As the temperature is reduced, the highest value actions are more likely to be chosen, and in the limit as T → 0, the best action is always chosen.

Overall the Epsilon-Greedy action selection method performed more poorly than did either of Softmax and Maximum Q-Value, this is not surprising as this method has a higher degree of randomness and thus a higher chance of selecting the wrong action. This can be seen by looking at a log of a 100x trial run using the Epsilon-Greedy action selection method, see **Q_Learning_Epsilon_Greedy_log.txt**. It shows that within each trial, one or more iterations received a negative reward. Whereas when using Softmax or Maximum Q-Value, iterations stopped receiving negative rewards after the first few trials (see **Q_Learning_Softmax_log.txt**), in fact in most runs negative rewards are not seen after the first trial.

As can be seen in the table below, as epsilon in increased, the number of successful trips decreases, it also takes longer for a trip to complete as the average number of moves per trip can be seen to increase (for both successful and unsuccessful trips).

Epsilon	Success Rate	Average number moves/successful trip	Average number moves/unsuccessful trip
0.10	73%	14.6	23.5
0.25	60%	15.4	26.7
0.50	44%	15.6	28.4

Note: (1) The average number of moves for unsuccessful trips will be an underestimate when enforce_deadline is True.

(2) Figures in the table are averages for 10x runs of 100x trials each.

(3) Values for alpha and gamma were constant throughout the trials.

The converse is also true ie. as epsilon is decreased, the success rate increases, this is expected as the chance of a random action decreases and the success rate will trend towards that of Maximum Q-Value, as that selection method is used when Epsilon-Greedy chooses a non-random action.

My testing using action selection methods Softmax and Maximum Q-Value revealed little or no difference between the two, although, both were significantly better than Epsilon-Greedy.

See table below:

Action Selection Method	**Success Rate**	**Average number moves/successful trip**	**Average number moves/unsuccessful trip**
Maximum Q-Value	92%	13.46	21.5
	90%	12.48	18.9
	92%	12.3	22.1
	92%	12.96	20.25
	91%	13.74	20.88
Average	**91.4%**	**12.988**	**20.726**
Softmax	91%	13.1	19.7
	87%	12.6	20.15
	94%	13.2	23.3
	94%	12.9	19.16
	91%	13.4	19.7
Average	**91.4%**	**13.04**	**20.402**

An observation I can make when comparing Softmax and Maximum Q-Value is that from my testing the Softmax success rate is more variable than that of Maximum Q-Value, I've observed Softmax success rates in the range high 80%'s to 97%, whereas the Maximum Q-Value success rate tends to stay in the low-mid 90%'s.

When using either of Softmax or Maximum Q-Value, we only see negative rewards within the first few trials, and a lot of the time they only occur in the first trial. Indicating that the Q-Learning agent has learn't the correct moves very quickly when selecting actions based upon experience rather than just picking actions randomly. This would appear to be a function of the relatively small state space, in that most states/actions are visited very quickly and appropriate lessons learned.

My code in agent.py issues the following message when a negative reward is received:

- ---> agent::update() Received a Negative Reward=-nn

The following message is issued when all states in the state space have been explored at least once:

- ---> ReinforcementLearning::selectLearner() State space complete in nn.0 iterations.

Overall Performance

Overall my agent performs reasonably well, for most 100x trials it guides the SmartCab to its destination successfully within the deadline over 90% of the time and does so consistently. See **Q_Learning_Softmax_log.txt**. Comparing information from the log to the basic agent we can see a significant increase in the number of successful trips (94 compared to 20) and a reduction in the average number of moves taken in a successful trip (13.75 compared to 16.0).

In general the only trips that the agent sometimes fails on are trips that have the smallest deadlines ie. 20 or 25.

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?

I believe my agent does get close to finding an optimal policy after 100x trials, as the values in the Q-Matrix for the 100x trial run are very similar to those of a 10,000x and a 50,000x trial run, see below:

Q-Matrix after 100x trial run:

---> agent::run() Q Matrix Contents...
(11, None) 1.58080361298
(11, 'forward') -1
(11, 'left') -1
(11, 'right') 0.5
(12, None) 1
(12, 'forward') -1
(12, 'left') -1
(12, 'right') 2.64421369568
(13, None) 1.6244014978
(13, 'forward') -0.936042129073
(13, 'left') -1
(13, 'right') 0.5
(21, None) 1
(21, 'forward') 0.5
(21, 'left') 2.76510957008
(21, 'right') 0.5
(22, None) 1
(22, 'forward') 0.5
(22, 'left') 0.5
(22, 'right') 2.64254646747
(23, None) 1.1
(23, 'forward') 2.53823622544
(23, 'left') 0.5
(23, 'right') 0.5

Q-Matrix after 10,000x trial run:
---> agent::run() Q Matrix Contents...
(11, None) 1.63040502539
(11, 'forward') -1
(11, 'left') -0.888109775458
(11, 'right') 0.5
(12, None) 1.0928117745
(12, 'forward') -1
(12, 'left') -0.724039026486
(12, 'right') 2.81102163981
(13, None) 1.56655341133
(13, 'forward') -1
(13, 'left') -1
(13, 'right') 0.54304958691
(21, None) 1
(21, 'forward') 0.5
(21, 'left') 2.74203105888
(21, 'right') 0.5
(22, None) 1.12405639639
(22, 'forward') 0.5
(22, 'left') 0.595994775271
(22, 'right') 2.83179608146
(23, None) 1.02626482213
(23, 'forward') 2.65759361313
(23, 'left') 0.5
(23, 'right') 0.5

Q-Matrix after 50,000x trial run:
---> agent::run() Q Matrix Contents...
(11, None) 1.65994144405
(11, 'forward') -1
(11, 'left') -1
(11, 'right') 0.5
(12, None) 1
(12, 'forward') -1
(12, 'left') -1
(12, 'right') 2.50571874812
(13, None) 1.56899847267
(13, 'forward') -1
(13, 'left') -1
(13, 'right') 0.5
(21, None) 1.08190379174
(21, 'forward') 0.5
(21, 'left') 2.79691673238
(21, 'right') 0.5
(22, None) 1
(22, 'forward') 0.520664615233

```
(22, 'left') 0.5
(22, 'right') 2.83473120601
(23, None) 1.04527641813
(23, 'forward') 2.51180600947
(23, 'left') 0.5
(23, 'right') 0.5
```

My agent's consistency in achieving similar results time after time also demonstrates that it does get close to converging to an optimal policy.

As so few negative rewards are received, the agent always completes successful trips with net rewards remaining positive. For example, see the following messages in log **Q_Learning_Softmax_log.txt**:

- ---> agent::reset() iterationCount=n Deadline=dd totalReward=rr.r

Even trials that fail to complete within the deadline will finish with net rewards remaining positive as they generally arrive at the destination within a maximum 5-10 further iterations (this confirmed by running some trials with **enforce_deadline=False**). Its interesting to compare this with trials run with the basic agent, where some trials would exceed the deadline and go on for hundreds or thousands of iterations before arriving at the destination.

Note:

1. My agent and learners messages are prefixed with "**--->** ".
2. The **agent.reset()** function does not receive control from **environment.py** once the deadline has been reached so my counters don't include the result of the last trial (So I run 101x trials instead).
3. The **iterationCount** on unsuccessful trips only includes iterations up to and including the deadline when **enforce_deadline=True**.