Train a Smartcab to Drive Report

In this project we aim at training a **smartcab** (may be referred to as **smart agent**) to move on a 8 by 6 grid, with randomly chosen starting point and destination (minimal distance: 4). The **environment** consists of the **grid**, three other **randomly moving agents** and **traffic lights** on every intersection, switching with different time intervals.

Our goal is to get our agent to the destination within the set deadline, and as fast as possible.

Note that most if not all of our code mentioned in this report will be in `agent.py`. We may make minor changes to other files for logging purposes only.

Setting up the Baseline

Random Walk Mode

Before we start implementing any machine learning algorithm, it is important that we know from what point we are optimizing from. And for problems like this, very often we can choose the random walk as our baseline.

This can be easily implemented with one line of code (or two lines, if you count the import):

```
import random
action = random.choice(Environment.valid_actions)
```

where `Environment.valid_actions = [None, 'forward', 'left', 'right']`.

In [1]:
```
from __future__ import division
import numpy as np
import pandas as pd
from IPython.display import display

random_mode = pd.DataFrame({'Number of Trials' : 100,
                    'Success Rate' : pd.Series([0.24, 0.24, 0.18, 0.16, 0.21]),
                    'Number of Penalized Trials' : pd.Series([96, 98, 97, 96, 9
7]),
                    'Avg. Reward Rate' : pd.Series([0.728, 0.672, 0.641, 0.753,
 0.675]),
                    'Max Reward Rate' : pd.Series([5.0, 3.167, 6.25, 10.5, 4.0]
),
                    'Min Reward Rate' : pd.Series([0.038, 0.0, -0.069, -0.071,
0.0]),
                    'Mode Reward Rate' : pd.Series([(0.0, 19), (0.0, 15), (0.0,
 26), (0.0, 24), (0.0, 16)])})

display(random_mode)
```

	Avg. Reward Rate	Max Reward Rate	Min Reward Rate	Mode Reward Rate	Number of Penalized Trials	Number of Trials	Success Rate
0	0.728	5.000	0.038	(0.0, 19)	96	100	0.24
1	0.672	3.167	0.000	(0.0, 15)	98	100	0.24
2	0.641	6.250	-0.069	(0.0, 26)	97	100	0.18
3	0.753	10.500	-0.071	(0.0, 24)	96	100	0.16
4	0.675	4.000	0.000	(0.0, 16)	97	100	0.21

Above are the data for each test run in random walk mode. The KPIs are defined as below:

- **Success Rate:** $\frac{\text{Number of times the agent reaches the destination}}{\text{Number of trials}}$
- **Number of Penalized Trials:** Number of trials which get any negative reward
- **Reward Rate:** $\frac{\text{Net reward}}{\text{Number of steps taken to get to the destination}}$
- **Avg. Reward Rate:** Mean Reward Rate of all the successful trials
- **Max Reward Rate:** Maximum Reward Rate
- **Min Reward Rate:** Minimum Reward Rate
- **Mode Reward Rate:** Mode of Reward Rates rounded to the first decimal point

Since our goal is to get to the destination as fast as possible while getting as much reward as possible (following traffic rules and not getting penalized), the most important factor here is the **Reward Rate** (the higher the better). This, however, doesn't tell us everything. Our smartcab may learn to "be smart" and try to get to the destination fast at all cost, breaking the traffic rules as a tradeoff. So in addition to this, we also want to monitor the **Number of Penalized Trials** (the lower the better) to make sure our smartcab learns to follow the traffic rules (Note: this is not the case for the current world model).

In addition to this, we can also look at the **Success Rate** which monitors the same behavior but is defined to give us a more macroscopic view.

With the KPIs set up we can see that the random walk performs rather poorly on getting our cab to its destination. And we will improve that with a better algorithm.

Optimize Against the Baseline

Identify the States

Our environment assumes the US right-of-way rules.

1. On a green light, you can turn left only if there is no oncoming traffic at the intersection coming straight.
2. On a red light, you can turn right if there is no oncoming traffic turning left or traffic from the left going straight.

Although violating some of the rules are not penalized for the current version, we would like to take those features into account and make them into our states so that the code is ready for future change (this will of course expand our feature space and therefore we will need more trials for the agent to learn the optimal policy).

Below are the inputs that our smartcab can take:

1. Next waypoint
2. Traffic light status
3. Status of oncoming agent
4. Status of agent on the left
5. Status of on the right
6. Deadline

As said earlier, although the smartcab doesn't get penalized for violating the right-of-way rules (which means we don't really need to let the smartcab sense the statuses of other agents), for now we still take them into account and observe how the smart agent would learn.

Next waypoint

This is the essential state that we need to take into account, since the destination is different for each new trial, representing the absolute location and heading is not useful. The only way for our smartcab to ever find the destination is to follow the **next waypoint**. The smartcab gets a reward of 2 each time it correctly follows the next waypoint, and a reward 0.5 when it doesn't.

Traffic light status

Status of the traffic light plays an important role here as well. The reward/penalty works as follows:

1. When the traffic light is 'red' and the agent goes 'forward' it gets penalized by -1
2. When the traffic light is 'red' and the agent goes 'left' it gets penalized by -1
3. In other cases the agent gets reward as stated in the Next waypoint section.

Statuses of other agents

As said ealier, statuses of other agents don't really play any role here. We include them here so that our code is ready for future change. The feature space is slighly expanded by including them so we may need to take them out if our smart agent fails to learn the optimal policy. Yet, we may not need to.

Deadline

Deadline is not represented as a status explicitly for now, because including it can make the state space too sparse for the Q values to converge. However, it does get implicitly represented by γ, since the less steps it takes for the agent to get to the destination, the more rewarding of reaching the destination would be, which in turn makes the policy that gets to the destination faster more favorable. We may take this into account later if the performance of the smart agent is not satisfying.

States used in Q-Table

To sum up, we will go with the following states to start with, and make adjustments as it rolls if need be.

- Next waypoint

In [2]:
```
qd_perf = pd.DataFrame({'Number of Trials' : 100,
                    'Success Rate' : 0.76,
                    'Number of Penalized Trials' : 54,
                    'Avg. Reward Rate' : 2.06,
                    'Max Reward Rate' : 5.5,
                    'Min Reward Rate' : 0.558,
                    'Mode Reward Rate': [(2.2, 9)]})

display(qd_perf)
```

	Avg. Reward Rate	Max Reward Rate	Min Reward Rate	Mode Reward Rate	Number of Penalized Trials	Number of Trials	Success Rate
0	2.06	5.5	0.558	(2.2, 9)	54	100	0.76

This is huge improvement! In this quick dirty version of Q-learning we start with complete randomness and decrease the probability of going random by 1% (which means we increase the probability of following the current best policy learned with Q-learning) each time. The decrease stops at when the probablity of going random is at 20%.

We won't address all of other factors here, such as the learning rate α or the discount factor γ. Also since the driving policy is very much random so we can't say that we can get constant good results like this, but nonetheless this is a good start.

Q-learning formula

The Q value of each state-action pair is given by:

$$Q(s,a) = R(s) + \gamma \cdot max_{a'}\ Q(s',a')$$

where the parameters/variables are defined as below:

- s: states
- a: action
- s': next states
- a': next valid actions
- Q: Q value
- γ: discount factor
- R: reward as function of s

However, since we don't know the exact Q value of any state, we'll use an estimated version of the formula, given by:

$$\hat{Q}(s,a) \overleftarrow{\alpha} R(s) + \gamma \cdot max_{a'}\ \hat{Q}(s',a')$$

where $\hat{Q}$ is our estimated Q (defaulted 0), and α is the learning rate ($V \overleftarrow{\alpha} x \equiv (1-\alpha)V + \alpha x$). α is defined by time step t: ($\alpha(t) = \frac{1}{t}$), where t is incremented by 1 each time $\hat{Q}$ gets updated.

So in any state, our policy is just choosing the action a that maximizes the Q values of that state (we call the max Q value the utility of that state: $U(s) = max_a\ Q(s,a)$), that is, $\pi(s) = augmax_a\ Q(s,a)$.

Training-testing

Our goal is to learn a feasible policy within 100 trials. And to test how well the agent has learned, we let it run another about 400 trials following the learned policy and inspect the statistics (we decrease the probability of going random by $\frac{1}{100}$ for each trial until it reaches 0. So from Trial 100 our agent would start completely following the policy it's learned from the previous 100 trials).

first_perf = pd.DataFrame({

```
                   'Success Rate': 0.913,
                   'Fail Trials': [(107, 108, 114, 128, 146, 149, 162, 169, 170, 221,
   260, 263, 272, 275, 281, 294, 295, 318, 328, 331, 350, 353, 362, 366, 405, 406, 409
   , 425, 428, 431, 452, 477, 486, 487, 494)],
                   'Avg. Distance': 4.388,
                   'Avg. Steps': 14.635,
                   'D/S': 0.300,
                   'Penalty Rate': 25 / 401,
                   'Penalized Trials': [(106, 117, 145, 149, 162, 175, 191, 214, 220,
   222, 226, 260, 286, 303, 307, 320, 353, 354, 362, 390, 393, 401, 426, 456, 465)],
                   'Avg. Reward Rate' : 2.702,
                   'Mode Reward Rate': [(2.1, 36)],
                   'SD Reward Rate': 1.112})
```

display(first_perf)

One thing to note here is that here we have made some minor changes to our KPIs, where we only get our statistics from our testing trials (rounds that strictly follow the policy the agent's learned). We have also added average steps needed to get a better understanding of how efficient our agent is. SD is also present to help us understand the consistency.

Q-Learning Optimiazation

KPI review

It appears that we are not so far away from our optimal policy now, given the Success Rate at 91.3% and Penalty Rate at 6.23%. Let's again review what KPIs we have and decide what we should do next.

Primary KPIs

The most important KPIs are of course:

- Success Rate
- Penalized Trials

If the smartcab doesn't even get to the destianation, then it is not really that smart and it doesn't really matter how much reward it collects. Also out of our 400 test-drive trials, about 6% of trials still get penalized.

Let's address these issues.

Parameters to consider

Take a look at our Q-learning equation and epsilon-greedy algorithm, we can see there are parameters that define the nature of our smart agent. They are:

- ϵ: the probability of our smart agent going random
- Learning rate α: parameter that determines how fast our Q values converge
- Discount factor γ: parameter that determines how much a "future" Q value is worth "now".

ϵ: Exploration-exploitation dilemma

Whenever our smart agent takes an action, it's either trying to explore the unknown or following what is known to it. The reasonable course of action, is therefore that if our agent knows a lot, it should just do what it knows is best. On the other hand, if our agent doesn't know much, it should just try to see the unseen and learns from it. When our smart agent first set out, the entire environment was completely unknown to it, so it should spend more time on exploration, rather than just following what the really limited "knowledge" of its. On the other hand as it learns more and more from the environment, it should start exploiting what it knows, rather than randomly exploring.

To implement this, we can decrease ϵ as we fill the Q table of the smartcab, so it would start out having a higher probability of going random and gradually shift to following the learned policy.

Learning rate α

As defined earlier, α as a function of t decreases as time elapses. The change rate of α signifies how much we believe in what we've learned now. If decreases fast, it means we believe we can learn the true Q value in just a few steps. On the other hand, if decreases slow, it means we believe there are a lot more to it then we have already learned. We can think of an agent with steeper learning rate as being more conservative (believing more strongly in what it's learned in the past), and more open if it has a less steep learning rate. The upside of of having a less steep learning rate is that it can always take new information into account, while the downside may be that it has a less stable policy and may converge more slowly.

Discount factor γ

A larger discount factor discourages future value and focuses more on immediate rewards. The good thing about a large discount factor is that our agent is more eager to get to the final destination faster so that the reward at the goal is more attractive to it. The downside with large discount factor is that if it discounts future value too much the agent may lose its

In [3]:
```
second_perf = pd.DataFrame({
                    'Success Rate': [0.913, 0.990],
                    'Fail Trials': [(107, 108, 114, 128, 146, 149, 162, 169, 17
0, 221, 260, 263, 272, 275, 281, 294, 295, 318, 328, 331, 350, 353, 362, 366, 4
05, 406, 409, 425, 428, 431, 452, 477, 486, 487, 494), (105, 149, 193, 200)],
                    'Avg. Distance': [4.388, 4.695],
                    'Avg. Steps': [14.635, 13.3325],
                    'D/S': [0.300, 0.352],
                    'Penalty Rate': [25 / 401, 33 / 401],
                    'Penalized Trials': [(106, 117, 145, 149, 162, 175, 191, 21
4, 220, 222, 226, 260, 286, 303, 307, 320, 353, 354, 362, 390, 393, 401, 426, 4
56, 465), (118, 122, 143, 150, 174, 177, 178, 186, 199, 216, 241, 242, 244, 287
, 306, 310, 342, 343, 347, 364, 372, 385, 389, 395, 397, 406, 417, 431, 457, 46
0, 468, 469, 495)],
                    'Avg. Reward Rate' : [2.702, 2.653],
                    'Mode Reward Rate': [(2.1, 36), (2.0, 133)],
                    'SD Reward Rate': [1.112, 1.557]})

display(second_perf)
```

	Avg. Distance	Avg. Reward Rate	Avg. Steps	D/S	Fail Trials	Mode Reward Rate	Penalized Trials	Penalty Rate	SD Reward Rate	Success Rate
0	4.388	2.702	14.6350	0.300	(107, 108, 114, 128, 146, 149, 162, 169, 170, ...	(2.1, 36)	(106, 117, 145, 149, 162, 175, 191, 214, 220, ...	0.062344	1.112	0.913
1	4.695	2.653	13.3325	0.352	(105, 149, 193, 200)	(2.0, 133)	(118, 122, 143, 150, 174, 177, 178, 186, 199, ...	0.082294	1.557	0.990

This is tremendous improvement! If we take a closer look, the success rate has increased to 99.0% and D/S has also increased 17.33%! This means that not only our smartcab is now more reliable, but it also gets to the destination in shorter time (reduced to $\frac{1}{1+0.1733} = 0.8522 = 85.22\%$).

There are still four cases where our smartcab faild to make to the destination. We put these cases in the Appendix. We can see that in all four cases our smartcab shows that it has the capability of taking detours and go around red lights. It wasn't able to make it in these four cases could be that the cases were just too extreme.

For now we are happy with the result. Let's move on to address the penalty problem.

Penalized trials

There seems to be quite a few trials in which our agent got penalized, both in where $\gamma = 0.9$ (penalty rate: 6.23%) and $\gamma = 0.8$ (penalty rate: 8.23%).

Let's take a closer look at the trials in the case where $\gamma = 0.9$. Below are the trials that got penalized.

[106, 117, 145, 149, 162, 175, 191, 214, 220, 222, 226, 260, 286, 303, 307, 320, 353, 354, 362, 390, 393, 401, 426, 456, 465]

From the code we can see, the penalty only occurs when our smartcab tries to violate the traffic rules. Our environment is set up in a way that not only it penalizes our agent by assigning to it negative rewards, but also it doesn't allow the smartcab to even move. This means violating the traffic rules does not give the smartcab benefits of any sort, long term or short term.

This makes the analysis a lot easier, since it wouldn't be the case that our agent was trying to make some tradeoff, trying to get to the destination faster, or just simply trying to collect more rewards. The only possible explanation for this that the smartcab didn't follow the rules is that it didn't see enough data of those particular states.

Let's look at some trials to verify our hypothesis.

```
Simulator.run(): Trial 465
Environment.reset(): Trial set up with start = (4, 6), destination = (6, 4), deadli
ne = 20
RoutePlanner.route_to(): destination = (6, 4)
LearningAgent.update(): deadline = 11, inputs = {'light': 'red', 'oncoming': None,
'right': 'right', 'left': None}, action = left, reward = -1

Simulator.run(): Trial 286
Environment.reset(): Trial set up with start = (4, 3), destination = (3, 6), deadli
ne = 20
RoutePlanner.route_to(): destination = (3, 6)
LearningAgent.update(): deadline = 15, inputs = {'light': 'red', 'oncoming': None,
'right': 'right', 'left': None}, action = left, reward = -1

Simulator.run(): Trial 145
Environment.reset(): Trial set up with start = (2, 4), destination = (7, 1), deadli
ne = 40
RoutePlanner.route_to(): destination = (7, 1)
LearningAgent.update(): deadline = 39, inputs = {'light': 'red', 'oncoming': None,
'right': 'right', 'left': None}, action = left, reward = -1
```

Bingo! In all of the three situations, our agent was in exactly the same state: `inputs = {'light': 'red', 'oncoming': None, 'right': 'right', 'left': None}` and took the action of `action = left`. Apparently the smartcab didn't really learn about what to do in this situation from the data collected from the first 100 runs.

As we run the simulation, we can see that in most situations, our smartcab is running on its own, not seeing any other agents coming nearby. And this makes it very difficult for our smart agent to learn a good policy for situations when there are other cars around.

We may just increase the number of dummy agent to solve this issue. The problem is, in the present representation, there

Appendix

Training-Testing data

Trial 494 (failed)

```
Simulator.run(): Trial 494
Environment.reset(): Trial set up with start = (1, 1), destination = (8, 2), deadli
ne = 40
RoutePlanner.route_to(): destination = (8, 2)
LearningAgent.update(): deadline = 40, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 39, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 38, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 37, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 36, inputs = {'light': 'green', 'oncoming': None
, 'right': None, 'left': None}, action = left, reward = 2
LearningAgent.update(): deadline = 35, inputs = {'light': 'green', 'oncoming': None
, 'right': None, 'left': None}, action = forward, reward = 2
LearningAgent.update(): deadline = 34, inputs = {'light': 'green', 'oncoming': None
, 'right': None, 'left': None}, action = forward, reward = 2
LearningAgent.update(): deadline = 33, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = right, reward = 0.5
LearningAgent.update(): deadline = 32, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 31, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 30, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 29, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 28, inputs = {'light': 'green', 'oncoming': None
, 'right': None, 'left': None}, action = left, reward = 2
LearningAgent.update(): deadline = 27, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = right, reward = 0.5
LearningAgent.update(): deadline = 26, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 25, inputs = {'light': 'green', 'oncoming': None
, 'right': None, 'left': None}, action = left, reward = 2
LearningAgent.update(): deadline = 24, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = right, reward = 0.5
LearningAgent.update(): deadline = 23, inputs = {'light': 'green', 'oncoming': None
, 'right': None, 'left': 'forward'}, action = forward, reward = 0.5
LearningAgent.update(): deadline = 22, inputs = {'light': 'green', 'oncoming': None
, 'right': None, 'left': None}, action = left, reward = 2
LearningAgent.update(): deadline = 21, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = right, reward = 0.5
LearningAgent.update(): deadline = 20, inputs = {'light': 'green', 'oncoming': None
, 'right': None, 'left': None}, action = left, reward = 2
LearningAgent.update(): deadline = 19, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 18, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 17, inputs = {'light': 'red', 'oncoming': None,
```

Second test data ($\gamma = 0.8$)

Trial 200

```
Simulator.run(): Trial 200
Environment.reset(): Trial set up with start = (8, 3), destination = (4, 5), deadli
ne = 30
RoutePlanner.route_to(): destination = (4, 5)
LearningAgent.update(): deadline = 30, inputs = {'light': 'green', 'oncoming': None
, 'right': None, 'left': None}, action = left, reward = 2
LearningAgent.update(): deadline = 29, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 28, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 27, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 26, inputs = {'light': 'green', 'oncoming': None
, 'right': None, 'left': None}, action = forward, reward = 2
LearningAgent.update(): deadline = 25, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 24, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 23, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 22, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 21, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 20, inputs = {'light': 'green', 'oncoming': None
, 'right': None, 'left': None}, action = forward, reward = 2
LearningAgent.update(): deadline = 19, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 18, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 17, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 16, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 15, inputs = {'light': 'green', 'oncoming': None
, 'right': None, 'left': None}, action = forward, reward = 2
LearningAgent.update(): deadline = 14, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = right, reward = 0.5
LearningAgent.update(): deadline = 13, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = right, reward = 2
LearningAgent.update(): deadline = 12, inputs = {'light': 'green', 'oncoming': 'lef
t', 'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 11, inputs = {'light': 'green', 'oncoming': 'lef
t', 'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 10, inputs = {'light': 'green', 'oncoming': 'lef
t', 'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 9, inputs = {'light': 'red', 'oncoming': 'left',
 'right': None, 'left': None}, action = right, reward = 2
LearningAgent.update(): deadline = 8, inputs = {'light': 'green', 'oncoming': None,
 'right': None, 'left': None}, action = right, reward = 2
LearningAgent.update(): deadline = 7, inputs = {'light': 'red', 'oncoming': None, '
right': None, 'left': None}, action = right, reward = 0.5
LearningAgent.update(): deadline = 6, inputs = {'light': 'green', 'oncoming': None,
```

Final test data

Trial 170 (failed)

```
Simulator.run(): Trial 170
Environment.reset(): Trial set up with start = (1, 4), destination = (3, 6), deadli
ne = 20
RoutePlanner.route_to(): destination = (3, 6)
LearningAgent.update(): deadline = 20, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = right, reward = 0.5
LearningAgent.update(): deadline = 19, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 18, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 17, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 16, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 15, inputs = {'light': 'green', 'oncoming': None
, 'right': None, 'left': None}, action = forward, reward = 2
LearningAgent.update(): deadline = 14, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 13, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 12, inputs = {'light': 'green', 'oncoming': None
, 'right': None, 'left': None}, action = forward, reward = 2
LearningAgent.update(): deadline = 11, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 10, inputs = {'light': 'red', 'oncoming': None,
'right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 9, inputs = {'light': 'red', 'oncoming': None, '
right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 8, inputs = {'light': 'green', 'oncoming': None,
 'right': None, 'left': None}, action = forward, reward = 2
LearningAgent.update(): deadline = 7, inputs = {'light': 'green', 'oncoming': None,
 'right': None, 'left': None}, action = forward, reward = 2
LearningAgent.update(): deadline = 6, inputs = {'light': 'red', 'oncoming': None, '
right': None, 'left': None}, action = None, reward = 1
LearningAgent.update(): deadline = 5, inputs = {'light': 'green', 'oncoming': None,
 'right': None, 'left': None}, action = forward, reward = 2
LearningAgent.update(): deadline = 4, inputs = {'light': 'red', 'oncoming': None, '
right': None, 'left': None}, action = right, reward = 0.5
LearningAgent.update(): deadline = 3, inputs = {'light': 'green', 'oncoming': None,
 'right': None, 'left': None}, action = right, reward = 2
LearningAgent.update(): deadline = 2, inputs = {'light': 'red', 'oncoming': None, '
right': None, 'left': None}, action = right, reward = 2
LearningAgent.update(): deadline = 1, inputs = {'light': 'red', 'oncoming': None, '
right': None, 'left': None}, action = right, reward = 2
LearningAgent.update(): deadline = 0, inputs = {'light': 'green', 'oncoming': None,
 'right': None, 'left': None}, action = left, reward = 2
Environment.reset(): Primary agent could not reach destination within deadline!
```