P4 Smartcab

1. Implement a Basic Driving Agent

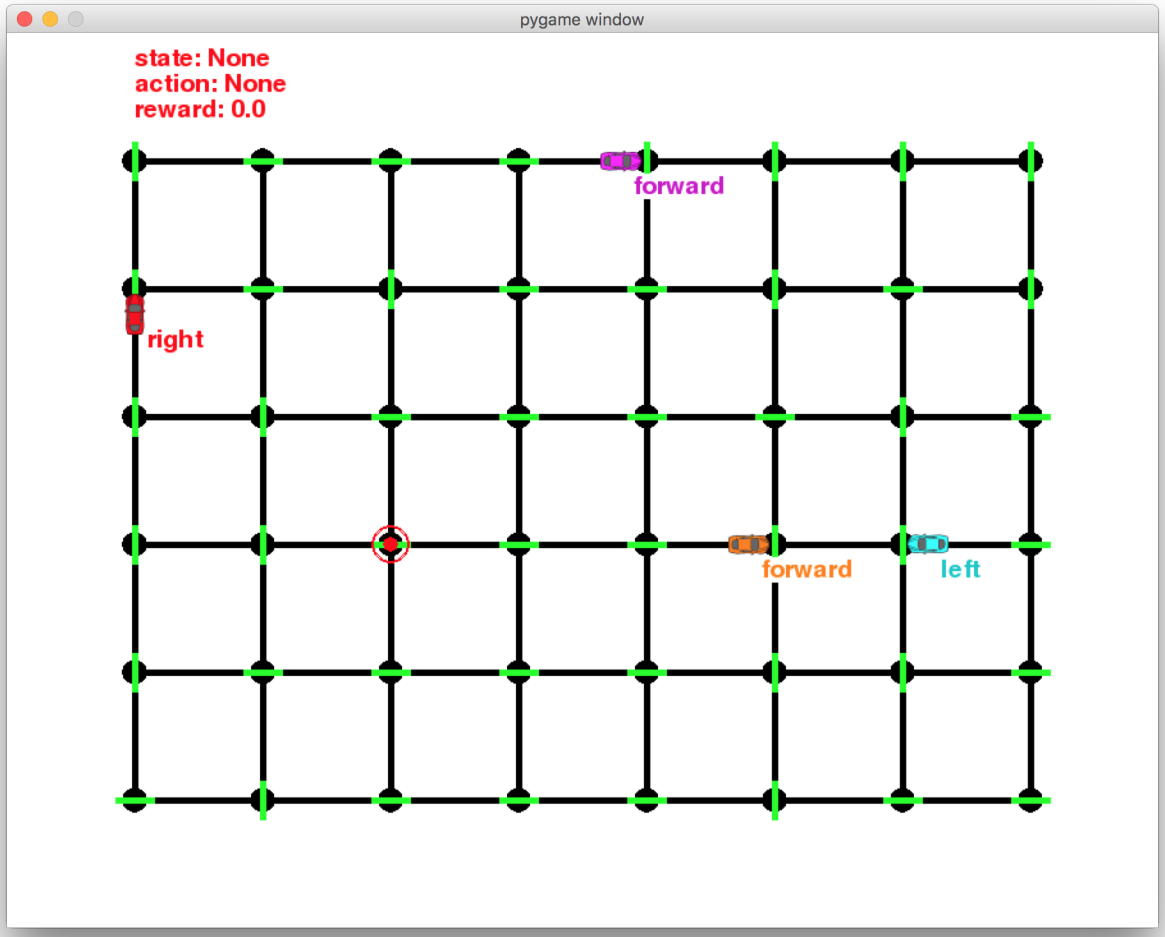
Process

Understanding the game The first thing I did was run the simulation to get an understanding of the game. I did not alter any code before I did this, so `enforce_deadline` was set to `True`.

In [28]:

```
from IPython.display import Image
Image(filename='img/grid.png')
```

Out[28]:



Changing Code I then (1) modified `action` within the `LearningAgent`'s `update` function in `agent.py` to make the car choose actions randomly instead of not at all:

```
    action = random.choice([None, 'forward', 'left', 'right'])
```

and (2) set `enforce_deadline=False`.

QUESTIONS:

1. Observe what you see with the agent's behavior as it takes random actions.
   - The agent can be blocked at one intersection for multiple turns because it wants to e.g. turn right at a green light, which is not allowed. (Reward = -0.5 for that turn)
   - The agent often goes around in loops.
2. Does the smartcab eventually make it to the destination?
   - It did in Trial 1. It did not in Trial 2: it hit the hard time limit (-100) and the trial aborted. (See console output below)
3. Are there any other interesting observations to note?
   - The agent can move through the rightmost side of the grid to end up on the left side of the grid. Similarly, it can move through the topmost side of the grid and reappear at the bottom and vice versa.

In [ ]:

```
# Console output for Trial 2 - Did not make it to destination

Simulator.run(): Trial 2
Environment.reset(): Trial set up with start = (7, 1), destination = (6, 6),
deadline = 30
RoutePlanner.route_to(): destination = (6, 6)
LearningAgent.update(): deadline = 30, inputs = {'light': 'green',
'oncoming': None, 'right': 'forward', 'left': None}, action = None, reward
= 0.0
LearningAgent.update(): deadline = 29, inputs = {'light': 'green',
'oncoming': None, 'right': 'forward', 'left': None}, action = right, reward
= 2.0
...
LearningAgent.update(): deadline = 1, inputs = {'light': 'green',
'oncoming': None, 'right': None, 'left': None}, action = right, reward =
2.0
LearningAgent.update(): deadline = 0, inputs = {'light': 'red', 'oncoming':
None, 'right': None, 'left': None}, action = left, reward = -1.0
LearningAgent.update(): deadline = -1, inputs = {'light': 'red', 'oncoming':
None, 'right': None, 'left': None}, action = forward, reward = -1.0
...
LearningAgent.update(): deadline = -99, inputs = {'light': 'green',
'oncoming': None, 'right': None, 'left': None}, action = None, reward = 0.0
LearningAgent.update(): deadline = -100, inputs = {'light': 'green',
'oncoming': None, 'right': None, 'left': None}, action = left, reward =
-0.5
Environment.step(): Primary agent hit hard time limit (-100)! Trial
aborted.
```

2. Inform the Driving Agent

QUESTIONS:

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

- Why do you believe each of these states to be appropriate for this problem?

States:

Attribute	**Why it's appropriate**	**Info source**	**Possible values**	**Number of possible values**
Where we want to go next to get to our destination	If there were no traffic lights or other cars, next_waypoint would give us the optimal next action we should take to reach our destination. That somewhat models our ideal action.	self.next_waypoint	None, 'forward', 'left', 'right' (Though if it's `None` we'll have reached our destination and won't care)	4 (3 without `None`)
Traffic light	Traffic lights will give part of the constraints that determine whether or not taking certain actions will be effective and what rewards they will receive.	inputs['light']	green, red	2
Oncoming (cars)	Similar to traffic lights, oncoming cars (straight ahead) are a constraint on our actions. We don't want to crash into another car. We want the algorithm to learn that crashing is bad.	inputs['oncoming']	None, 'forward', 'left', 'right'	4
What the car immediately to the left wants to do	If the car to the left is going to turn right, you don't want to turn left and crash into it.	inputs['left']	None, 'forward', 'left', 'right'	4
What the cars immediately to the right wants to do	Similar to inputs['left'].	inputs['right']	None, 'forward', 'left', 'right'	4

OPTIONAL:

- How many states in total exist for the smartcab 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 number of states: 4^4 * 2 = 512 states.

The minimum 'deadline' is `minimum distance` x 5 = 4 x 5 = 20 and the maximum is 12 x 5 = 60.

If we suppose that the car takes an average of at least 20 turns per trial and that each state has the

same likelihood of being visited, that means **each state will be visited an average of** 20 turns x 100 trials / 512 states i.e. about **4 times**. This is **reasonable but is still quite low**.

This low number is **why I did not include further state attributes** I considered (see boloew) because that would only reduce the number of visits to each state even further.

States that I considered:

Attribute	**Info source**	**Possible values**	**Number of possible values**
Deadline	deadline	• Impossible: if compute_dist < deadline. • Possible: if compute_dist >= deadline	2
Location relative to destination	Primary agent coordinates, destination coordinates	8*6-10=38 coordinate pairs, discounting rotational and reflective symmetries.	38

Glossary:

- Coordinate grid: (1,1) is top left and increases to the right and down. Bottom right is (8,6).
- Headings: Directions car is facing. [(1,0),(0,-1),(-1,0),(0,1)] # ENWS
- inputs['right'] means that there is a car to the primary agent's immediate right. (See image below) The attribute value indicates what that car wants to do.

In [ ]:

```
Image(filename="img/input_right.png")
```

3. Implement a Q-Learning Driving Agent

Q-learning algorithm The crux of the Q-learning algorithm is

```
new_q = old_q*(1 - self.alpha) + self.alpha*(reward + self.gamma * m
ax_state2_q)
```

in the `learn_q` function in `agent.py`.

Choosing actions The agent chooses the best action to take by choosing the action with the maximum Q-value for the corresponding state. It chooses a random action if there are multiple actions where the resulting Q-value = maxQ.

Exploration The agent also chooses a random action with probability epsilon. This allows it to escape if it 'gets stuck' in some suboptimal local optima.

Decaying learning rate (1/t) The initial learning rate is high at Alpha=1 so the agent learns quickly. As time goes on, the agent becomes more confident with what it's learned and is less persuaded by new information.

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?

The agent is **more likely to take actions corresponding to `next_waypoint`** when those are viable because it receives a reward of 2.0 if it can execute that action without penalty and is learning to behave in ways that **maximise total expected reward**.

The agent is **less likely to take actions tha result in penalties** (crashing into cars, making illegal moves or making moves that are legal but are not equal to `next_waypoint` so they are further from the destination) because those usually do not maximise reward. It does learn to make legal but 'incorrect' moves rather than crash into a car.

The agent **reaches the destination more frequently**: after implementing Q-learning, it reaches the destination in time over 80% of the time, whereas while it was moving randomly it reached the destination in time less than 10% of the time. The agent does not just move randomly in loops any more.

As the agent **gains experience**, it is less likely to go around in loops or get penalised for going against traffic rules or crashing into other cars.

Notes: Debugging 'Implementing a Q-Learning Driving Agent'

1. I realised the agent wasn't acting because the `count` variable was defined wrongly:
    - `count` was used to see there were multiple actions with `q-value = maxQ` for that state.
    - If `count` > 1, we would randomly choose one action out of the set of actions where `q-value = maxQ`.
    - `count` was wrongly defined as `len([maxq])`, which is always equal to one since it is an array with a float in it.
    - It should've been `len([i in q if q[i] == max_q])` instead.
    - Because it was defined wrongly, the agent kept choosing the first of all the actions that had the same q-value.
    - This meant the agent often chose `None`.
2. I'd forgotten to incorporate `next_waypoint` into my state. Pretty silly.
3. I wanted to print results after every turn for debugging purposes and put `self.results` in TrafficLight instead of Environment.

4. Improve the Q-Learning Driving Agent

4.1 Planning

Procedure:

1. Run each configuration 50 times (50 sets of 100 trials)
2. Write metrics into separate file
3. Convert to summary statistics over 50 sets

4. Observe statistics
5. Alter list of configurations as appropriate and repeat until satisfied

The **metrics considered** were

- **Total number of successful outcomes** (out of 100) because unsuccessful outcomes (Trial aborted because car did not make it in time) indicates **inefficiency**.
  - **Average buffer** (Time left / Initial deadline) -> Indicates how efficient the driving agent was.
- **Average number of incorrect actions per trial** (penalties of -1.0) because this indicates an action was **unsafe**.

The parameters considered were

- Exploration rate Epsilon (epsilon)
- Discount rate Gamma (gamma)
- Learning rate Alpha (alpha)
- Default Q value (if one did not exist before (default_q)

4.2 Optimising

4.2.1 Optimising for Epsilon

epsilon	gamma	alpha	default_q	successes	avg_buf	avg_penalties
0.20	0.50	'1.0/t'	0.0	87.78	0.5179	1.0810
0.10	0.50	'1.0/t'	0.0	94.20	0.5709	0.5732
0.05	0.50	'1.0/t'	0.0	96.50	0.5709	0.3664
0.01	0.50	'1.0/t'	0.0	98.36	0.5829	0.1926

Observation It seems like the smaller Epsilon is, the greater the proportion of successes, the greater the average buffer and the smaller the average number of penalties.

Interpretation This makes sense: the learning does not 'get stuck' often in ways that we need random actions to get it back on track, so random actions only decrease performance. It seems intuitive enough that we don't have to try a variety of epsilon values over some other combination of gamma and alpha.

Next actions For the rest of the trials we will experiment with epsilon values of 0.01 and 0.05 for robustness. We want an algorithm that will drive safely even if it goes haywire sometimes.

Once we have chosen our gamma and alpha, we will optimise for epsilon.

4.2.2 Optimising for Gamma (and Alpha)

Gamma takes values between zero and one, so I first tested gamma values from four quartiles to get an idea of how our metrics changed with Gamma.

epsilon	gamma	alpha	default_q	successes	avg_buf	avg_penalties

0.05	0.01	'1.0/t'	0.0	98.00	0.5705	0.3694
0.05	0.25	'1.0/t'	0.0	97.18	0.5726	0.3538
0.05	0.50	'1.0/t'	0.0	96.50	0.5709	0.3664
0.05	0.75	'1.0/t'	0.0	94.02	0.5573	0.3822
0.05	0.99	'1.0/t'	0.0	75.30	0.5399	0.6030

Observations

- It seems that the number of successes are higher if Gamma is lower. and buffer are higher if gamma is lower.
- The average penalty decreases slightly as Gamma increases from 0.01 to 0.25 before increasing again at Gamma=0.5.
- Likewise, the average buffer increases slightly as Gamma increases from 0.01 to 0.25 before decreasing again at Gamma=0.5.

Next actions This motivates us to try more Gamma values in the range (0,0.5).

4.2.3 Pre-emptive checking for robustness

We need to test if our observations for Gamma hold for different values of Alpha. I tested gamma values from the four quartiles with Alpha=1/t^2 instead of Alpha=1/t and obtained similar results:

epsilon	gamma	alpha	default_q	successes	avg_buf	avg_penalties
0.05	0.01	'1.0/(t^0.5)'	0.0	98.06	0.5709	0.3710
0.05	0.20	'1.0/(t^0.5)'	0.0	97.76	0.5767	0.3568
0.05	0.25	'1.0/(t^0.5)'	0.0	97.68	0.5722	0.3636
0.05	0.50	'1.0/(t^0.5)'	0.0	96.62	0.5696	0.3616
0.05	0.75	'1.0/(t^0.5)'	0.0	93.76	0.5539	0.3888
0.05	0.99	'1.0/(t^0.5)'	0.0	69.60	0.5312	0.7028

Observation The **trend differences** were that average penalties continued to decrease as Gamma was increased up to 0.50.

Decision-making and next actions The decrease in average penalty from Gamma=0.25 to Gamma=0.50 was smaller than 0.01 but came at the cost of a decrease in average successes of over 1 as well as a decrease in the average buffer, so I decided it was **not worthwhile to experiment with increasing Gamma beyond 0.25**.

4.2.4 Continue optimising for Gamma

Alpha = '1.0/t'

epsilon	gamma	alpha	default_q	successes	avg_buf	avg_penalties
0.05	0.01	'1.0/t'	0.0	98.00	0.5705	0.3694

0.05	0.25	'1.0/t'	0.0	97.18	0.5726	0.3538
0.05	0.50	'1.0/t'	0.0	96.50	0.5709	0.3664

Alpha = '1.0/(t^0.5)'

epsilon	gamma	alpha	default_q	successes	avg_buf	avg_penalties
0.05	0.01	'1.0/(t^0.5)'	0.0	98.06	0.5709	0.3710
0.05	0.25	'1.0/(t^0.5)'	0.0	97.68	0.5722	0.3636
0.05	0.50	'1.0/(t^0.5)'	0.0	96.62	0.5696	0.3616

Observations: For Alpha = 1/t, we have Gamma=1.0 having a higher average number of successes, the highest average buffer and a lower average penalty than Gamma=0.01 but higher than Gamma=0.2. There is thus an argument for **setting Gammma to be around 0.1**.

For Alpha = 1/(t^0.5), we still have Gamma=0.01 with the marginally highest average successes. There is still a tradeoff between (1) average successes and (2) average buffer (up to Gamma=0.2) and decreasing average penalties (up to Gamma=0.25). It is less clear where we'd want to set Gamma to be in this case if we do not make value judgements about which metrics matter more. If we do make value judgements, I would argue that **setting Gamma to be around 0.01** would be appropriate because an increase in successes of 0.3 trials (per 100) is more important than decreasing average penalty by 0.008 per trial.

Next Actions: It seems that we have a general idea of where the optimal Gamma is going to be and that the precise optimal gamma may depend on our choice of Alpha. Thus we should begin to optimise Alpha before we further narrow down our choice of Gamma.

We will try various Alpha = 11/(t^exp) where exp=0.01, 0.25, 0.5, 0.75 and 1, with Gamma=0.01, 0.10 and 0.20.

4.2.5 Optimising for Alpha

epsilon	gamma	alpha	default_q	successes	avg_buf	avg_penalties
0.05	0.01	'1.0/(t^0.01)'	0.0	97.72	0.5761	0.3618
0.05	0.01	'1.0/(t^0.25)'	0.0	98.06	0.5722	0.3608
0.05	0.01	'1.0/(t^0.5)'	0.0	98.06	0.5709	0.3710
0.05	0.01	'1.0/(t^0.75)'	0.0	97.84	0.5713	0.3718
0.05	0.01	'1.0/t'	0.0	98.00	0.5705	0.3694

epsilon	gamma	alpha	default_q	successes	avg_buf	avg_penalties
0.05	0.10	'1.0/(t^0.001)'	0.0	97.72	0.5733	0.3616
0.05	0.10	'1.0/(t^0.01)'	0.0	98.34	0.5737	0.3634
0.05	0.10	'1.0/(t^0.25)'	0.0	98.06	0.5723	0.3608
0.05	0.10	'1.0/(t^0.5)'	0.0	97.98	0.5682	0.3638

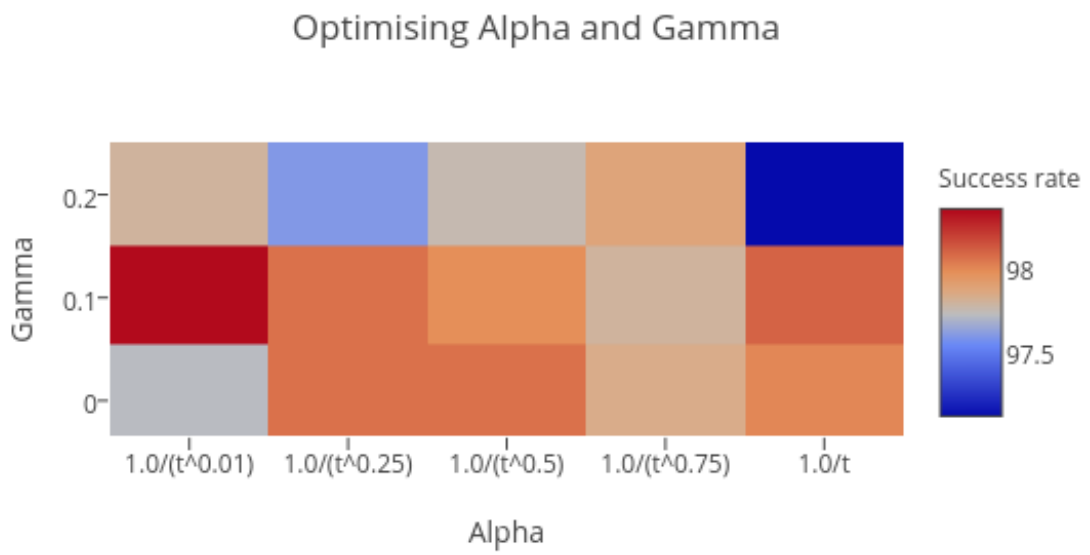
0.05	0.10	'1.0/(t^0.75)'	0.0	97.80	0.5707	0.3788
0.05	0.10	'1.0/t'	0.0	98.10	0.5747	0.3604

epsilon	gamma	alpha	default_q	successes	avg_buf	avg_penalties
0.05	0.20	'1.0/(t^0.01)'	0.0	97.80	0.5653	0.3730
0.05	0.20	'1.0/(t^0.25)'	0.0	97.60	0.5724	0.3606
0.05	0.20	'1.0/(t^0.5)'	0.0	97.76	0.5767	0.3568
0.05	0.20	'1.0/(t^0.75)'	0.0	97.88	0.5694	0.3632
0.05	0.20	'1.0/t'	0.0	97.12	0.5685	0.3834

In [29]:

```
Image(filename='img/heatmap-alpha-gamma.png')
```

Out[29]:



(Alpha=0.01 was doing so well I decided to try Alpha=0.001. The heat map is not continuous which seems strange.)

For Gamma=0.01, the difference between different alphas seems insignificant with the exception of exponent=0.75.

- Pick exp=0.25

For Gamma=0.1, it seems like performance plotted against the exponent of (1/t) is shaped like x^2, where exponent = 0.25 and 1 perform best.

- Pick exp=0.01

For Gamma=0.2,

- Pick exp=0.75

Overall: pick Gamma=0.1, Alpha=1/(t^0.01).

4.2.6 Optimising Epsilon

epsilon	gamma	alpha	default_q	successes	avg_buf	avg_penalties
0.000	0.10	'1.0/(t^0.01)'	0.0	98.70	0.5861	0.1706
0.000001	0.10	'1.0/(t^0.01)'	0.0	98.90	0.5885	0.1728
0.000005	0.10	'1.0/(t^0.01)'	0.0	98.96	0.5869	0.1686
0.00001	0.10	'1.0/(t^0.01)'	0.0	99.12	0.5926	0.1640
0.001	0.10	'1.0/(t^0.01)'	0.0	98.98	0.5963	0.1692
0.01	0.10	'1.0/(t^0.01)'	0.0	98.66	0.5884	0.2058
0.05	0.10	'1.0/(t^0.01)'	0.0	98.34	0.5737	0.3634

Choose epsilon = 0.00001.

4.2.7 Optimising default Q-value

epsilon	gamma	alpha	default_q	successes	avg_buf	avg_penalties
0.00001	0.10	'1.0/(t^0.01)'	0.0	99.12	0.5926	0.1640
0.00001	0.10	'1.0/(t^0.01)'	0.5	98.66	0.5889	0.1760
0.00001	0.10	'1.0/(t^0.01)'	1.0	98.88	0.5886	0.1844
0.00001	0.10	'1.0/(t^0.01)'	2.0	99.12	0.5912	0.1848
0.00001	0.10	'1.0/(t^0.01)'	2.5	98.48	0.5827	0.1974

successes 98.480000 avg_buffer 0.582687 avg_penalties 0.197400

A default Q-value of 0.0 has the best performance, though a default Q-value of 2.0 interestingly comes very close. For more robust results, I would try smaller increments of Q with larger 100-trial sets.

It seems that *moderate optimism in the face of uncertainty* is a less optimal assumption here.

(Note: I only tested different Q-values on this particular set of epsilon, gamma and alpha values. It is possible higher Q-values will work better for other epsilon, gamma and alpha.)

QUESTIONS:

Parameters chosen:

Exploration rate Epsilon	Discount rate Gamma	Learning rate Alpha	DefaultQ
0.00001	0.1	1/(t^0.01)	0.0

Discussion: How well does the final driving agent perform?

- An optimal policy would be successful in almost all trials (the exceptions being where it is

- An optimal policy would be successful in almost all trials (the exceptions being where it is impossible for some reason). That is' it would attain close to 100 successes per 100 trials.
- It would be efficient and thus approach the theoretical maximum buffer of 0.8 (since deadline = compute_dist * 5)
- It would maxmise net reward and thus likely incur close to zero -1.0 penalties.

Comparing our driving agent to the optimal policy

Policy	**Avg successes per 100 trials**	**Average buffer (proportion) per trial**	**Number of -1.0 penalties**
Our agent	99.12	0.5926	0.1640
Optimal policy	100	Close to 0.8 (approaching from below)	Likely 0

- Judging by the the Average Successes per 100 trials, our policy is close to the optimal policy.
- As noted before, it's hard to know what the optimal policy's average buffer would be so although there is a gap, we don't know how great the gap between our policy and the optimal one is.
- There are still a significant number of penalties occurring (violations of traffic rules or crashing). This is suboptimal.

Penalties that occurred in the last 10 trials in a set:

Trial 94:

- next_waypoint: forward
- q: [0.0, 0.0, 0.0, 0.0]
- max_q: 0.0
- action: forward
- LearningAgent.update(): deadline = 32, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': 'left'}, action = forward, reward = -1.0

Trial 99:

- next_waypoint: forward
- q: [0.0, 0.0, 0.0, -0.48971014879346336]
- max_q: 0.0
- action: forward
- LearningAgent.update(): deadline = 26, inputs = {'light': 'red', 'oncoming': None, 'right': 'forward', 'left': None}, action = forward, reward = -1.0

The penalties occur because the agent has had little (99) or no (94) previous experience in this state. These are usually states where `oncoming`, `right`, or `left` are not blank.

We then conclude that **our policy is efficient but not nearly as safe as it could be** .