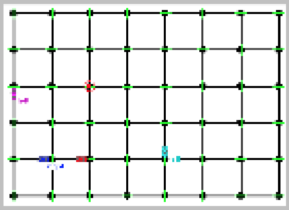
Implement a basic driving agent

In your report, mention what you see in the agent's behavior. Does it eventually make it to the target location?

1. When **None** is assigned to the action variable, the **agent virtually never make it** to the target location given it doesn't appear directly on it.
2. When **random.choice** is assigned to the action variable, the **agent sometime make it** to the target location but looking at the following example, even that close to its target, the probability the car makes it to the location in two moves is of 1/16 given it can achieve an unlawful move. The probability it does without burning any light is of 1/64.



3. When a **Q table protocol** is assigned to the action **at the very beginning**, the agent take the first state related action as the maximum since it's the only one and therefore the maximum. The **agent ends up in loop-like behaviour** and **hardly do it to the target**. In fact, it is even worse than random.
4. When we take a **100 random trials while uploading the Q table then take Q table protocol** as our action definer, the **agent ends up finding its way** to the target most of the time and happen to **break less rules**.

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. Justify why you picked these set of states, and how they model the agent and its environment.

The **self.state variable is a tuple** containing the following substates from *inputs*: **light, oncoming, left**. We reduced the substates number since

We add the 'next_waypoint' to self.state therefore allowing the agent to understand where it needs to go given a substate of (light, oncoming, left) which means that in order to find the optimal policy for a state, the agent needs to visit it at least 4 times.

When the **light is green** the only substates we need to take into account **'light', 'oncoming' and 'next_waypoint'** since we only need to look up if someone is coming onward in case we need to turn left. All other substates are irrelevant since **no 'legal' danger** can come from it. We're assuming the **other agents are moving perfectly**, hence there's no need to look right or left when the light is green since no dummy agent is suppose to burn a red.

When the **light is red** the only substates we need to take into account are **'light', 'left' and 'next_waypoint'** since we only need to look up if someone is coming left in case we turn right. All other substates are irrelevant since **no 'legal' danger** can come from it. We're assuming the **other agents are moving perfectly**, hence there's no need to look oncoming when the light is red since no dummy agent is suppose to burn a red.

As we see there is **no need to include traffic from the right** since if the light is green, we are allowed to turn right whatever and if the light is red, we are only interested in the traffic from the left.

In both cases, we add the 'next_waypoint' to allow the agent to understand where it needs to go given a state which means that in order to find the optimal policy for a state, the agent needs to visit it at least 4 times.

We **didn't include the deadline** substate since it **make the dimensionality way bigger** for no significative contribution. If deadline was included in self.state, two similar states would only similar if taken at the very same **deadline count**. It also makes the **Q table way bigger** and affects the program **efficiency**. The agent may need **way more trials** to makes its conclusions for few more good trips.

Implement Q-Learning

What changes do you notice in the agent's behavior?

Since we've chosen a 100 random trials as describe in question 1, visual improvements aren't continuous but sudden at the 101 trials!

At first when all the inputs were used as state, the dimensionality was so big that the agent wasn't able to understand all the states and happen to repeat non optimal moves. The agent must visit all the state 4 times which implies a tremendous quantity of move it needs to make until it find optimal -

Then, when pseudo vectors were introduced, the dimensionality was reduced and the agent happen to behave more efficiently but happen to make many bad moves.

Enhance the driving agent

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? (Here you should give a detailed list of parameter tested and report the results obtained).

List of upgrades and reasons
Remove the +10 reward associated with a good trip ending.
Led the agent to think a bad move made just before ending was a very good move with +9 reward and that a mediocre move was a even better one with +9.5 or +10 -
Add a decaying alpha
A decaying alpha helps the agent to trust less and less what it is learning to compensate a long-term learning with huge numbers leading to less efficient table search.
Add decaying epsilon probability of taking a random move
Avoid getting **stuck into the right-right-right-right loop** forever and allow a proportion of learning when the agent is passed its learning state of 100 random trials. We chose **decaying epsilon** against a static one since as we approaches **perfect policy** the randomness happens to hurt more than it helps. On the other hand, at first stage of Q-drived trial, we need some randomness to avoid the loop problem in case we learned something bad. We choose 8 as at 100 trials we have a 92% chance of taking a Q-driven action.

*We need to keep in mind that the **randomness** of the learning leads to variety of answers for a same set so the **answer must be taken more as a magnitude than a performance**!*

It worth pointing out that the alpha decay variable is not the same as alpha: alpha = alpha decay/(alpha decay + move counter) where counter is +1 at each move.

It also worth pointing out that the epsilon decay variable is not the same as epsilon: epsilon = epsilon decay/ trial counter where counter is +1 at each trial.

List of parameters with associated performances ; gamma

gamma	alpha decay	epsilon decay	good trips	bad moves	optimal moves
.0001	5	8	96.0%	0.44%	52.92%
.05	5	8	89.0%	0.1%	43.3%
.1	5	8	88.0%	1.1%	45.3%
.2	5	8	93.0%	1.0%	47.1%
.3	5	8	89.0%	1.1%	44.2%
.4	5	8	91.0%	0.4%	56.7%
.5	5	8	91.0%	0.5%	43.9%

List of parameters with associated performances ; alpha decay

gamma	alpha decay	epsilon decay	good trips	bad moves	optimal moves
.05	1	8	89.0%	0.4%	42.0%
.05	2	8	93.0%	0.47%	45.0%
.05	3	8	99.0%	0.3%	45.2%
.05	5	8	84.0%	0.3%	41.2%
.05	7	8	95.0%	0.3%	57.45%
.05	9	8	100.0%	0.5%	51.8%
.05	50	8	72.0%	0.36%	43.2%

We've found the best overall tuple is near **(0.5,5,8), we'll use this as a first guess** in an optimisation protocol. We'll use the **scipy.optimize.minimize** protocol which is a greedy multi-approach optimizer. The files used to do the optimization are : **OverAnalyzer.py and agentLooper.py** .

scipy.optimize.minimize on 1-proportion of +2 moves, 100 trials, hardcoded Q table

Parameters	Initial Guess	Range	Best Parameters
gamma	0.5	[0.0001,1]	1.0
alpha decay	5	[0.0001,20]	20.0
epsilon decay	8	[0.0001,20]	20.0

scipy.optimize.minimize on proportion of bad trips, 100 trials, hardcoded Q table

Parameters	Initial Guess	Range	Best Parameters
gamma	0.5	[0.0001,1]	0.049999940255 5
alpha decay	5	[0.0001,20]	5.0
epsilon decay	8	[0.0001,20]	0.949998

scipy.optimize.minimize on proportion of -1 moves, 2 trials, hardcoded Q table

Parameters	Initial Guess	Range	Best Parameters
gamma	0.5	[0.0001,1]	0.05
alpha decay	5	[0.0001,20]	5.0
epsilon decay	8	[0.0001,20]	0.95

We had to reduce the number of looping trials in the parameters search minimizer for -1 moves since the process was very expensive and last too long.

We see that in order **to maximize good move we should be never exploring or taking random move** since generally ¾ of those come in contradiction with choosing the optimal one. We see that, in order to reduce bad trips and bad moves the **optimal parameters are the pretty much the same as we guessed** with hand-run grid search. Since we mostly want to **avoid breaking the law and finish in time**, we weigh more on **minimizing proportions of bad moves and bad trips** than optimizing good moves. Hence the parameters are **(gamma = 0.05, alpha decay = 5, decay epsilon = 8)** .

*Since the **epsilon is threatened by 2 levels of randomness**, one from the very definition of the problem, the other from epsilon itself, the result of optimality as a measurements will be **irrelevant** even in a magnitude sense!*

In a qualitative sense, we can say that for epsilon less than 90, the agent is moving dangerously and for **epsilon bigger than 99, it have chance to get caught in** the right-right-right-right **loop** for a painful time. We chose a dynamic decaying epsilon to avoid this problem without paying full randomness price of taking foolish action instead of perfect one.

Enhance the driving agent

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? (Here you should observe and analysis the behavior of the agent, does it always go in an optimal path and follow traffic laws and if not under what specific circumstances does it not follow the optimal policies).

Average behaviour of agent over many run on best parameters (0.05, 5, 0.95)

Average of good moves (+2)	40%
Average of mediocre moves (+0 or +0.5 or +1)	50%
Average of bad moves (-1)	10%
Average of good trips	85%

Given it's **impossible to get a +2 at each steps** (since a problem can cross our optimal path to our target and force us to wait or take alternative paths which are non-optimal), we are suppose to be near an optimal policy.

Near half the move we make are optimal moves which is related to the 50% ratio of red light and negligible ratio of dummy agents crossing meaning we are definitely near an optimal solution.

The agent is doing better than 80% on ending trips before forced deadline which means our agent is following near optimal policy.

Although it is ***near optimal* it is not *optimal*** since 20% of trials ends with deadline, an **optimal policy should end very near 100%** -

An optimal policy would always take the +2 direction when its possible, otherwise it chose the **optimal legal alternative** action toward target. **Optimally**, the agent virtually never break the law. In practice, it can only happens given the epsilon is going toward 0.

The agent don't take the most optimal route but it is getting there. When a **red light is crossing its next_waypoint path**, the agent tends to **stop and wait** in order to get the +1 reward associated. Since it has no sense of deadline, even if the light is **stuck red**, the agent prefers to wait +1 than to **turn right** +0.5 and go on. In further implementations, we could give the agent a sense of deadline by reserving more aggressive reaction as the deadline goes by.

Computation and Problem

I had problem with the Pycharm installation on my osx so I used a Linux Weezy vm machine on Google Compute Engine with a VNC viewer on which the installation was far more easier.

When I created the OverAnalyzer.py program, I have ran into the computation problem. I needed my computer to run 200 steps of heavy code for approximately a 100 of times which I was hardly able to do on my osx, so I upgrade my CPU's use on GCE and turned my vm into a 8 CPUs one. This made the computation far more easier.

The only drawback was the time I needed to set-up the vm, install a desktop, install a VNC viewer, create a firewall rule, etc.

Further Possible Implementations

Implementations which are not known as legal in the problem context and complex implementations.

1. Use a defined learning stage where we assured that all states are visited in the 4 possible ways. This could make sure we are always finding the perfect policy.
2. Loop over OverAnalyzer.py a great number of times to assure a near-perfect average. Since I'm using a trial version of Google Compute Engine, I can't use the parallel CPU distribution, so this could be very long even with 8 high grade CPUs.
3. Upgrade the Environment and Simulator code to allow a visual free version and therefore lower the heaviness of the code by a great proportion.
4. Allow a dynamic version of deadline and next_waypoint to allow the agent to run through traffic (with many dummy agents).