Udacity MLND P4: Train a Smartcab

Implementing 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?

What I notice in agent's behavior is that its very random. There are lot's of -1 rewards which suggest that it is certainly not performing the way its intended to. In some of the cases it does make up to the location but its just pure luck. In most of the trials the end result is
"Primary agent hit hard time limit (-100)! Trial aborted."

Identify and update state

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

I think the state of the agent is described by the next_waypoint as well as the inputs which our environment is sensing.
Thus the state comprises of following key value pairs

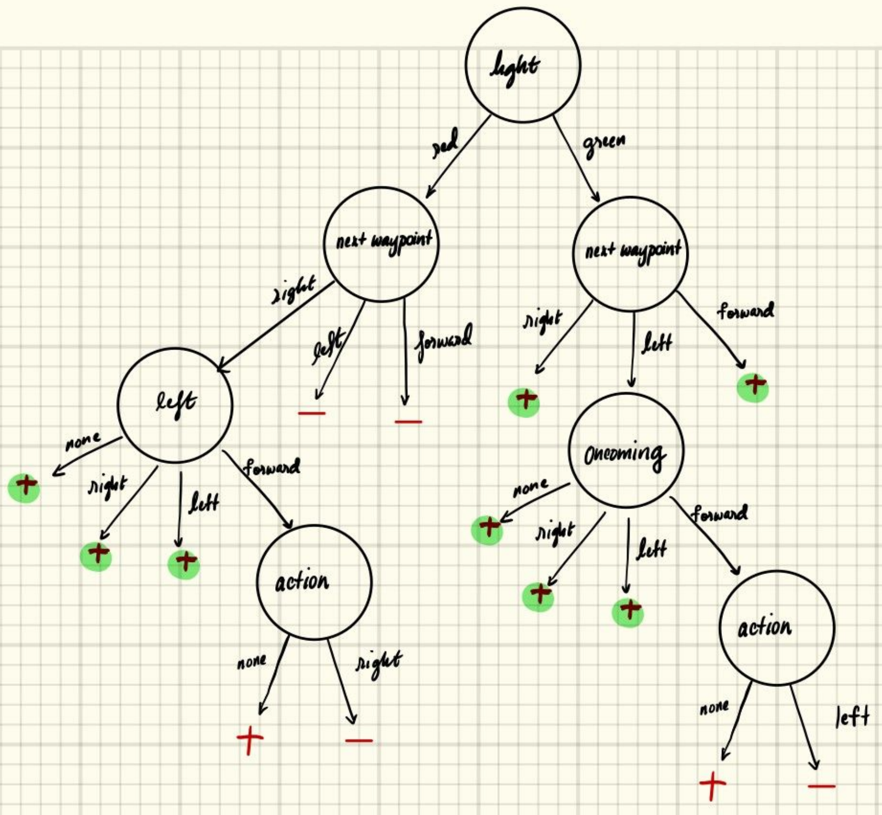
```
    {
        'Left' : ['left', 'right', 'forward', 'none'],
        'Right' : ['left', 'right', 'forward', 'none'],
        'Oncoming' : ['left', 'right', 'forward', 'none'],
        'Light': ['Red' , 'green' ],
        'Next_waypoint' :  ['left', 'right', 'forward', 'none']
    }
```

So the state space comprises of 4x4x4x4x2 = 512 states which is finite but too large. However, by making some smart guess we can cut down the state space. If you notice the significance of 'Right' input, there's nothing much it conveys we can ignore it. Which reduces our state space to 4x4x4x2 = 128 states.
'Deadline' can also be one more component of the state space. But I will omit it because I would prefer my agent to reach late instead of making wrong moves trying to reach to the destination as fast as possible.
So our final states consist of following 4 keys

    ('Left', 'Oncoming', 'Light', 'Next_waypoint')

Out of these next_waypoint plays an important role in helping the agent reach his destination whereas the other three are consequences of rules and good driving habits.

Below is a good decision tree representation of the reward system based of US-way of traffic rules that I found online.



+ denotes an implicit action node where there are two options:
① Take the next waypoint, which is positively rewarded.
② Take no action (i.e. None), which is also positively rewarded
Since they are both positively rewarded, and they don't give as interesting information I chose to write them as an implicit action node.

As you can notice depriving the state of any of 'Left' , 'Oncoming' and 'Light' can lead to violation of traffic rules and can even put passeneger's life into risk. That's why we are taking it into consideration. Also following the next waypoint when also following the U.S. right-of-way rules results in bigger rewards than just following the U.S. right-of-way rules but meandering aimlessly.

Implement Q-Learning

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

After implementing Q-learning and making choices based on q_values, the performance of agent improved significantly. Without deadline, the agent reached destination 100% of the time.
Also with deadline, after 100 trials, the agent was able to reach destination in time without taking any wrong step most of the time.
There were some penalties but those were mostly due to either a random choice(because of epsilon) or the states which accounted for penalties were unseen before by our agent so we didn't have q-values for them and thus ended up taking random decision.

So long story short, with q-learning agent commits errors initially but once it has explored enough(seen most of the states) it can act wisely maximizing the rewards making smart 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?

So I made a comprehensive_search function (similar to grid search) that selects the parameters that would result best best reward/time ratio. After getting the optimal parameters the agent was performing faster and the errors (-ve rewards) also reduced slightly.
Alpha and gamma are for Q-value updating, while epsilon is indeed the exploration/exploitation dilemma. Ideally speaking, all of the three should decrease over time because
As the agent continues to learn, it actually builds up more resilient priors. This means that, ideally, alpha should decrease as you continue to gain a larger and larger knowledge base. Similarly, for gamma, as you get closer and closer to the deadline, your preference for near term reward should increase, as you won't be around long enough to get the long term reward, which means your gamma should decrease.
For epsilon, as you develop a policy, you have less need of exploration and get more utility from actually following your policy, so as trials increase, epsilon should decrease.

I couldn't find a good way of deciding by how much these values should decrease after each iteration so I assumed them to be static. So the values which I received from my grid search was the values which we are assuming constant all over the simulation. But still the results are very good.

The metric I decided to chose for the best combination of parameters in reward/(no. of steps). This is because we would want to choose parameters such that we get maximum reward as fast as possible. Also I decided to track the no. of penalties as this could also be a deciding factor (we don't want agent to violate rules at the cost of reaching faster).

Here are the results of my grid search sorted by the ratio. The key-tuple is of the form (alpha, gamma, epsilon)

```
[((0.3, 0.1, 0.01), {
   'penalties': 3,
   'ratio': 1.558340217969466
}), ((0.3, 0.1, 0.1), {
   'penalties': 45,
   'ratio': 1.5581813138610796
}), ((0.3, 0.3, 0.1), {
   'penalties': 39,
   'ratio': 1.5580637050394854
}), ((0.3, 0.7, 0.01), {
   'penalties': 1,
   'ratio': 1.5580321587193515
}), ((0.3, 0.3, 0.01), {
   'penalties': 5,
   'ratio': 1.5579624305986932
}), ((0.3, 0.7, 0.1), {
   'penalties': 51,
   'ratio': 1.5578915112668181
}), ((0.3, 0.5, 0.01), {
   'penalties': 1,
   'ratio': 1.5578670606931486
}), ((0.3, 0.5, 0.1), {
   'penalties': 40,
   'ratio': 1.557815610866924
}), ((0.5, 0.1, 0.01), {
   'penalties': 8,
   'ratio': 1.5578021276528837
}), ((0.3, 0.7, 0.2), {
   'penalties': 75,
   'ratio': 1.5576539753021834
}), ((0.3, 0.1, 0.2), {
   'penalties': 98,
   'ratio': 1.5576378958408064
}), ((0.5, 0.7, 0.01), {
   'penalties': 2,
   'ratio': 1.5576235792718167
}), ((0.3, 0.5, 0.2), {
   'penalties': 101,
   'ratio': 1.5576080696671961
}), ((0.5, 0.7, 0.1), {
   'penalties': 46,
   'ratio': 1.557574476263694
```

```
}), ((0.3, 0.3, 0.2), {
    'penalties': 86,
    'ratio': 1.5575525088772708
}), ((0.5, 0.5, 0.1), {
    'penalties': 35,
    'ratio': 1.557488743706304
}), ((0.5, 0.1, 0.1), {
    'penalties': 61,
    'ratio': 1.5574645757719299
}), ((0.5, 0.3, 0.01), {
    'penalties': 1,
    'ratio': 1.5572828843007207
}), ((0.5, 0.5, 0.01), {
    'penalties': 1,
    'ratio': 1.5572669562830257
}), ((0.5, 0.3, 0.1), {
    'penalties': 41,
    'ratio': 1.5572528597221205
}), ((0.5, 0.5, 0.2), {
    'penalties': 91,
    'ratio': 1.5572498220376803
}), ((0.7, 0.1, 0.01), {
    'penalties': 3,
    'ratio': 1.557050595329191
}), ((0.7, 0.1, 0.1), {
    'penalties': 50,
    'ratio': 1.5569902571620549
}), ((0.5, 0.1, 0.2), {
    'penalties': 90,
    'ratio': 1.556988256424859
}), ((0.5, 0.7, 0.2), {
    'penalties': 100,
    'ratio': 1.5568849097720954
}), ((0.7, 0.3, 0.1), {
    'penalties': 31,
    'ratio': 1.5567784707602517
}), ((0.7, 0.3, 0.01), {
    'penalties': 5,
    'ratio': 1.556768728215714
}), ((0.5, 0.3, 0.2), {
    'penalties': 88,
    'ratio': 1.5567525487252878
}), ((0.7, 0.5, 0.01), {
    'penalties': 0,
    'ratio': 1.5566981975451821
}), ((0.7, 0.7, 0.1), {
    'penalties': 35,
    'ratio': 1.5566032792579152
}), ((0.7, 0.5, 0.1), {
    'penalties': 38,
    'ratio': 1.5566028638927727
}), ((0.7, 0.1, 0.2), {
    'penalties': 89,
    'ratio': 1.556599577449127
}), ((0.7, 0.7, 0.01), {
    'penalties': 4,
    'ratio': 1.556436926787852
}), ((0.7, 0.3, 0.2), {
    'penalties': 97,
    'ratio': 1.556229641878613
}), ((0.7, 0.5, 0.2), {
    'penalties': 85,
    'ratio': 1.5561841662257707
}), ((0.9, 0.5, 0.01), {
    'penalties': 0,
    'ratio': 1.5560981782493477
}), ((0.9, 0.3, 0.1), {
    'penalties': 30,
    'ratio': 1.5560570846858615
}), ((0.9, 0.5, 0.1), {
    'penalties': 30,
    'ratio': 1.556055974488178
}), ((0.9, 0.7, 0.1), {
    'penalties': 50,
    'ratio': 1.5560326527039152
}), ((0.9, 0.7, 0.01), {
    'penalties': 3,
    'ratio': 1.5559439537080084
}), ((0.9, 0.1, 0.01), {
    'penalties': 6,
    'ratio': 1.5558978722389696
}), ((0.9, 0.3, 0.01), {
    'penalties': 1,
    'ratio': 1.5558029534771827
}), ((0.7, 0.7, 0.2), {
    'penalties': 120,
    'ratio': 1.55573855432301
}), ((0.9, 0.1, 0.1), {
    'penalties': 49,
    'ratio': 1.555732803670374
}), ((0.9, 0.3, 0.2), {
    'penalties': 94,
   'ratio': 1.555711567008516
}), ((0.9, 0.5, 0.2), {
   'penalties': 89,
   'ratio': 1.5556531305206354
}), ((0.9, 0.7, 0.2), {
   'penalties': 95,
   'ratio': 1.5556314872471146
}), ((0.9, 0.1, 0.2), {
   'penalties': 90,
   'ratio': 1.5553576354796703
})]
```

As you can see the ratios are almost same, so the no. of penalties will be my deciding factor. Here are the results sorted on the basis of penalties

```
[((0.7, 0.7, 0.2), {
   'penalties': 109,
   'ratio': 1.5883420513156112
}), ((0.7, 0.3, 0.2), {
   'penalties': 101,
   'ratio': 1.5885315640640563
}), ((0.9, 0.7, 0.2), {
   'penalties': 99,
   'ratio': 1.5883058043727878
}), ((0.7, 0.1, 0.2), {
   'penalties': 98,
   'ratio': 1.588649488392016
}), ((0.5, 0.7, 0.2), {
   'penalties': 92,
   'ratio': 1.5887381425966522
}), ((0.9, 0.5, 0.2), {
   'penalties': 90,
   'ratio': 1.5887556071891482
}), ((0.9, 0.3, 0.2), {
   'penalties': 88,
   'ratio': 1.5887471355047147
}), ((0.7, 0.5, 0.2), {
   'penalties': 88,
   'ratio': 1.5887387125142098
}), ((0.3, 0.3, 0.2), {
   'penalties': 86,
   'ratio': 1.58776230824755
}), ((0.5, 0.1, 0.2), {
   'penalties': 86,
   'ratio': 1.5884341761782883
}), ((0.3, 0.5, 0.2), {
    'penalties': 82,
    'ratio': 1.5880535596910192
}), ((0.5, 0.3, 0.2), {
    'penalties': 80,
    'ratio': 1.5887048507533583
}), ((0.5, 0.5, 0.2), {
    'penalties': 77,
    'ratio': 1.5888648033022073
}), ((0.3, 0.7, 0.2), {
    'penalties': 76,
    'ratio': 1.5883953437632352
}), ((0.9, 0.1, 0.2), {
    'penalties': 74,
    'ratio': 1.588548148797849
}), ((0.3, 0.1, 0.2), {
    'penalties': 72,
    'ratio': 1.5880466123827035
}), ((0.9, 0.7, 0.1), {
    'penalties': 47,
    'ratio': 1.5889507190858516
}), ((0.7, 0.5, 0.1), {
    'penalties': 43,
    'ratio': 1.589110560643034
}), ((0.5, 0.3, 0.1), {
    'penalties': 43,
    'ratio': 1.588992688032784
}), ((0.5, 0.1, 0.1), {
    'penalties': 42,
    'ratio': 1.5886313643864327
}), ((0.5, 0.7, 0.1), {
    'penalties': 42,
    'ratio': 1.5893707054027717
}), ((0.5, 0.5, 0.1), {
    'penalties': 40,
    'ratio': 1.5891893554013012
}), ((0.7, 0.7, 0.1), {
    'penalties': 39,
    'ratio': 1.5889334063650498
}), ((0.7, 0.3, 0.1), {
    'penalties': 38,
    'ratio': 1.5888913446335369
}), ((0.3, 0.3, 0.1), {
    'penalties': 37,
    'ratio': 1.588054204251921
}), ((0.3, 0.7, 0.1), {
    'penalties': 36,
    'ratio': 1.5887779898904286
}), ((0.7, 0.1, 0.1), {
    'penalties': 35,
    'ratio': 1.5890906177165245
}), ((0.3, 0.1, 0.1), {
    'penalties': 32,
    'ratio': 1.5884310191432014
}), ((0.9, 0.3, 0.1), {
    'penalties': 32,
    'ratio': 1.5893200680338724
}), ((0.3, 0.5, 0.1), {
    'penalties': 32,
    'ratio': 1.588294142214711
}), ((0.9, 0.5, 0.1), {
    'penalties': 31,
    'ratio': 1.5892276833494248
}), ((0.9, 0.1, 0.1), {
    'penalties': 29,
    'ratio': 1.5888371655901787
}), ((0.9, 0.5, 0.01), {
    'penalties': 7,
    'ratio': 1.5891417171656568
}), ((0.7, 0.3, 0.01), {
    'penalties': 4,
    'ratio': 1.5888452984251675
}), ((0.9, 0.1, 0.01), {
    'penalties': 4,
    'ratio': 1.588725700172252
}), ((0.3, 0.1, 0.01), {
    'penalties': 3,
    'ratio': 1.588271049484363
}), ((0.9, 0.7, 0.01), {
    'penalties': 3,
    'ratio': 1.5891253575223903
}), ((0.3, 0.3, 0.01), {
    'penalties': 3,
    'ratio': 1.5881468847916933
}), ((0.9, 0.3, 0.01), {
    'penalties': 3,
    'ratio': 1.5889068829577218
}), ((0.7, 0.7, 0.01), {
    'penalties': 3,
    'ratio': 1.5890733545272577
}), ((0.5, 0.7, 0.01), {
    'penalties': 2,
    'ratio': 1.5892600611369205
```

```
}), ((0.5, 0.1, 0.01), {
   'penalties': 2,
   'ratio': 1.5888118023988433
}), ((0.7, 0.5, 0.01), {
   'penalties': 2,
   'ratio': 1.5890486964016375
}), ((0.5, 0.5, 0.01), {
   'penalties': 2,
   'ratio': 1.5890892195952193
}), ((0.7, 0.1, 0.01), {
   'penalties': 1,
   'ratio': 1.5890203342735654
}), ((0.3, 0.5, 0.01), {
   'penalties': 1,
   'ratio': 1.588067485906639
}), ((0.5, 0.3, 0.01), {
   'penalties': 1,
   'ratio': 1.5886671734649738
}), ((0.3, 0.7, 0.01), {
   'penalties': 1,
   'ratio': 1.5885221831286698
})]
```

Observation - As the epsilon value increases penalties increase which is obvious because we are exploring and making random decisions. But one thing to notice is among all the values of epsilon, the (alpha,gamma) pair with highest ratio is (0.3,0.1)

So the optimal parameters that I ended up chosing are

Alpha = 0.3
Gamma = 0.1
Epsilon = 0.1

I left epsilon to be 0.1 only because during the trials we should explore also instead of always trusting on prior knowledge.

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?

Yes I think that the agent was indeed able to find the optimal policy by reaching as soon as possible and taking care of the penalties.
So initially while exploring the agent makes quite a few errors because agent has no prior knowledge yet but as the iterations increase error rate comes down. It comes out that after training with 100 trials and then setting epsilon to 0. The next 100 trials had 1-2 penalties only. And I noticed that these penalties are due to the rare states which our agent didn’t

encounter while it was training, so he ended up making random choice resulting in a bad move.
Here’s one typical example of such a situation where multiple cars interact

```
{
        'light': 'red',
        'oncoming': 'forward',
        'right': None,
        'left': 'right',
         action = forward,
        reward = -1,
        next_waypoint = forward
}
```

So after exploring enough it tends to figure out the optimal policy which is “Reaching the destination before the deadline without violating any traffic rules” and makes very few errors.