Matthias Wettstein
Udacity - Machine Learning Engineer
Smartcab Project Report

Implement A Basic Driving Agent

Code

agent_basic.py. My comments are indicated with ’’’ <> ’’’.

Two steps are implemented in the code:

1. Processing of the inputs
2. Choosing and performing of a random action out of all possible actions
3. Measuring and displaying the share of successful trials

Observation of the agent’s behaviour

When performing 10 experiments with 100 trials each the agent in average reaches destination in 14.7 trials per experiment. During the whole 1'000 trials, it did not manage to perform a single trial without negative reward.

Random Action			
Experiment	Performed Trials	Successful Trials	Successful Trials w/o Negative Rewards
1	100	16	0
2	100	11	0
3	100	19	0
4	100	9	0
5	100	13	0
6	100	14	0
7	100	20	0
8	100	15	0
9	100	15	0
10	100	15	0
Mean Success		14.7	

The agent acts not according to the state it is in, but instead randomly.

This leads to rewards ranging from -1 for violation of traffic rules or even accidents, to 2.0, when it accidentially drives the way indicated by the planner without misconduct.

Although it has plenty of time (100 time fragments plus the deadline) for each trial, and traffic in the basic set-up is not dense (3 cars on a grid of 8x6), it barely ever reaches target. Only by bare luck, it will manage to perform a trial without getting any negative rewards.

The agent does not learn: It falls into the same traps over and over again. This is visible through the reward structure over trials and time steps per trial: negative rewards keep happening until the very end of trials. If we take 30 as average deadline per trial, this means that 13'000 moves are not making the agent any wiser.

Identify And Update State

The full set of inputs for every state is the following:

1. Planner waypoint indication
2. Intersection state, which consists of
    a. the traffic light state
        i. Red
        ii. Green
    b. the presence and direction of cars:
        i. Presence:
            1. (Coming from the) left
            2. (Coming from the) right
            3. Oncoming
        ii. Directions:
            1. None
            2. Forward
            3. Left
            4. Right

In general, the agent should follow the planner's instruction unless it thereby will get a negative reward by violating traffic rules or not following the planner's instructions.

In order to reach this goal, I picked the inputs indicated in green to be the relevant elements of the current state. They are directly determining the calculation of the best action:

- Following the planner waypoint indication is the crucial element for reaching the destination. Not following the planner's instruction will lead to a negative reward.
- Complying with the intersection state is crucial for the agent if it should not violate traffic rules or provoke an accident, and thus getting a negative reward. Albeit, only some of the intersection states are relevant inputs:
    - With a green light (traffic light state), the agent is allowed to move in whatever direction, except to the left, if oncoming traffic is headed forward.
    - With a red light, the agent is not allowed to move forward or left. It is allowed to move right, if oncoming traffic is not headed left, or traffic coming from the left is not headed forward

These four inputs build the states, on which the agent learns and builds up experience with action-reward functions. Acting on this state will enable the agent to eventually reach the destination without getting any negative reward in the trial.

The presence of cars from the right never plays a role for correct driving, given the agent follows the traffic light. The same is true for cars not moving at all, or cars coming from whatever direction, and moving to the right. Their presence is superfluous in the state, since not relevant for the agent's action.

Other inputs like time *t*, the *deadline*, or the *number of trials* could - in a different experimental setting - affect the parameters *alpha, gamma*, and *epsilon*, which of course at the end determine the selection of the best action indirectly (see *Outlook*).

At one point during my work with the Q-learning algorithm, I decided to take them into the state. Especially the *number of trials* was killing the learning, since the states are differing by this input trial by trial. The state space thus is blown up, and this is not allowing the agent to update Q values during more than one trial. All of a sudden, my agent did near-to-random moves again, and I was wondering - for quite a while - why.

The other two inputs *t*, the *deadline* are not as forbidding, but still are blowing up the state space, which I also had to experience at one point.

Implement Q-Learning

agent_qlearning.py. My comments are indicated with ’’’ <> ’’’.

The initial guesses for the *Q values* learning rate *alpha*, discount factor *gamma*, and epsilon-greedy parameter *epsilon* are arbitrary:

- The *Q values* are initialized to 0. In complete absence of any knowledge, a neutral initial value is reasonable. It guarantees, that even minor rewards and punishments afterwards are showing immediate effect onto learning.
- *Alpha* is initialized to 0.1. This is weak learning, but all the same, it's learning.
- *Gamma* is initialized to 1.0. This guarantees that the agent will seek for long-term high reward, which seems to be a good strategy to me.
- *Epsilon* is initialized to 0.1. This guarantees exploration in 10% of all moves. At this point, I do not want to be too sure of the policy the agent is learning, so I allow for deviation.

The following steps are implemented in the code:

1. Initialize a dictionary containing Q values for states and actions (*Q-matrix*)
2. Initialize unknown Q-values
3. Initialize the learning rate *alpha*
4. Initialize the discount rate *gamma*
5. Initialize the epsilon-greedy action selection parameter *epsilon*
6. For Performance Measurement, set up counters for the number of trials, the number of successfully terminated trials, and the number of successfully terminated trials without any negative reward.
7. Update the state at t_0 as described above-standing.
8. Memorize the state at t_0 for the latter updating of the Q values at t_0
9. Select the optimal action according to the action-reward functions for the state, or select a random action if the state is not yet in the Q matrix.
10. Collect the reward for the action
11. Learn the policy based upon reward, current Q value, and subsequent Q value

12. Display performance metrics after each successful trial

Observation of the agent's behaviour

When performing 10 experiments with 100 trials each, the agent in average reaches destination in 85.7 trials per experiment. In average, it manages to perform 5.3 trials without negative reward (6.2%).

Alpha 0.1	Gamma 1.0	Epsilon 0.1		
Experiment	Performed Trials	Successful Trials	Successful Trials w/o Negative Rewards	Share
1	100	87	5	5.7%
2	100	88	5	5.7%
3	100	86	3	3.5%
4	100	86	3	3.5%
5	100	79	3	3.8%
6	100	95	6	6.3%
7	100	81	6	7.4%
8	100	84	6	7.1%
9	100	89	7	7.9%
10	100	82	9	11.0%
Mean Success		85.7	5.3	6.2%
Standard Dev.				2.3%

The agent now reaches destination in 86% of all trials, which is a dramatic improvement compared to the random action! It even *learned the optimal policy*, since it performs some successful trials during which it does not accumulate any negative reward. This performance is also stable across the experiments.

In general, since there are not many cars on the grid, it happens often that at the beginning learning takes place for empty intersections. The agent there has only to take into consideration the traffic light and the planner's instructions, so act upon a reduced state space. At some point these policies are learned. But as soon cars start appearing at the intersections, they do not cover the situation. Instead, the new state is taken into the Q matrix, and fresh learning must take place. This traffic-related learning tends to be slower than with empty intersections, since the occurrance of crowded grids in this setting is relatively rare.

It seems also that *epsilon* is chosen too large: Too many actions deviate from learned policies, especially in the context of rare crowded intersections and thereupon slow learning.

Having tracked this as an issue, and to check for the real learning patterns at this point, I reduce epsilon to 0 and compare actions, rewards, and Q-values whenever there is a reward of smaller than 0.

Some prominent observable patterns briefly explained:

- As foreseen, there are some new states arising with crowded traffic at some points during the experiments, when the random action for unknown state is potentially producing negative rewards.
- In general, with red light, situations happens, where the agent randomly chooses to go forward or left, is punished, has still 3 options set to initial value 0, next time chooses right, is maybe punished for not following the planner, eventually takes forward, and is punished again.
- Again with red light, an agent might have been rewarded several times for taking the right (and following the planner). Another time, when the planner indicates non-right in this state, the agent will faithfully stick to the right-turn policy, potentially over and over, until the high Q value for action right has eventually melted away. So, some states related to red traffic light are tricky for the agent to learn, whereas green light related states are much more straight forward and quicker learned.

Enhance the Driving Agent

Report the Changes / Does the Agent Find An Optimal Policy

The chosen implementation seems to be a good base for further exploration. It finds the optimal policy, i.e. *it reaches the destination in the allotted time while not incurring any penalties*.

-- V2 -- START
By stating this, I do not say that the agent is *always* reaching the destination in time, without ever accumulating negative rewards.

Merely, I want to point out that there already *are* some trials where these conditions are met. These trials do not necessarily have to be located at the end of an experiment, by when the agent is expected to have learned a lot already.

In oder to show which pattern are responsible for receiving negative rewards at the end of an experiment / with a mature learner, I will focus on the last 10 trials of another experiment with the given parametrization (*alpha* 0.1, *gamma* 1.0, *epsilon* 0.1).

agent_qlearning.py. My comments are indicated with ''' < ... V2 ... > '''.

The additional functionality in the code creates a matrix, where for each of the iterations during the last 10 trials the following information is collected:

1. Trial
2. Iteration
3. Relevant Inputs
4. Action
5. Reward
6. Q-value at t0
7. The Updated Q-value
8. Q-value for "None" agent movement

9. Q-value for "forward" agent movement
10. Q-value for "right" agent movement
11. Q-value for "left" agent movement

The nested list is printed, but it preferably should be saved as a csv when reproducing the steps taken here.

The result table and analysis is stored in *smartcab_scrutiny.xlsx*.

For each record with a negative reward I then perform a manual assessment of the reason behind the action (tab *List*). The crosstabulation is located in tab *Analysis.*

Results:

1. The agent learned to fully avoid negative rewards of -1
2. Dominant strategies are responsible for 81% of the negative rewards, and red light accounts for 91% of dominant strategies. With red light, turning right is the *single dominant strategy for multiple states*. The Q-value for action 'right' is so high, that it is prevailing until the end of the experiment. This strategy is mostly responsible for the 'merry-go-round' clockwise movement of the agent when it encounters multiple red lights in a row.
3. 17% of the negative rewards are caused by exploration.
4. 1 single action (2%) was due to an initialization situation, where the agent chose a random move out of the actions with action-reward functions initialized to 0.

Wrap-Up:

1. The domination of turning right at a red light is unquestionable. The hypothesis at this point is, that it probably arises from situations where 'right' is rewarded with 2, whereas the safety option 'None' is never giving any positive reward. To fully uncover the root causes of this effect, I would need to observe red light states during a whole experiment.
2. Exporation is sometimes causing negative rewards.
3. Learning "never stops", new states can arise until the end of an experiment.

-- V2 -- END

In order to increase the agent's performance, I will follow a hill climbing-like approach

- tune *gamma*
- find a best performing configuration for the parameter, ceteris paribus
- tune *alpha*
- find a best performing configuration for the parameter, ceteris paribus
- tune the initial *Q-value*
- find a best performing configuration for the parameter, ceteris paribus
- tune *epsilon*
- find a best performing configuration for the parameter, ceteris paribus

This approach bears the risk of getting stuck in local minima. Also, the number of experiments is too small for a well-founded analysis.

Thus, this is about getting a first idea on how to tune a Q-learner, and to get an idea of a potentially well-working configuration.

Starting Point

I already have identified that too large an *epsilon* is decreasing the agent's performance. So, as already discussed, I lowered *epsilon* to 0.

The result is striking: the average share of successful trials without negative rewards increases by more than 60%:

Alpha 0.1	Gamma 1.0	Epsilon 0.0		
Experiment	Performed Trials	Successful Trials	Successful Trials w/o Negative Rewards	Share
1	100	63	28	44.4%
2	100	100	76	76.0%
3	100	99	54	54.5%
4	100	97	78	80.4%
5	100	98	81	82.7%
6	100	57	34	59.6%
7	100	100	80	80.0%
8	100	98	74	75.5%
9	100	100	80	80.0%
10	100	98	53	54.1%
Mean Success		91	63.80	68.7%
Standard Dev.				14.0%

Increase *Gamma* to 1.1

From this promising start, I now have two ways to go with *gamma*. First, I increase it to *gamma* = 1.1, and get a similar performance as with *gamma* = 1.0, although with higher variance in the results:

Alpha 0.1	Gamma 1.1	Epsilon 0.0		
Experiment	Performed Trials	Successful Trials	Successful Trials w/o Negative Rewards	Share
1	100	91	9	9.9%
2	100	99	79	79.8%
3	100	99	83	83.8%
4	100	98	52	53.1%
5	100	97	82	84.5%
6	100	99	69	69.7%
7	100	61	26	42.6%
8	100	100	84	84.0%
9	100	99	75	75.8%

10	100	96	51	53.1%
Mean Success		93.9	61.00	63.6%
Standard Dev.				24.2%

Decrease *Gamma* to 0.9

The other way looks more promising: better average as with *gamma* = 1.0, similar variance in the results:

Alpha 0.1	Gamma 0.9	Epsilon 0.0		
Experiment	Performed Trials	Successful Trials	Successful Trials w/o Negative Rewards	Share
1	100	98	83	84.7%
2	100	99	83	83.8%
3	100	99	81	81.8%
4	100	100	77	77.0%
5	100	99	76	76.8%
6	100	99	48	48.5%
7	100	98	85	86.7%
8	100	96	56	58.3%
9	100	98	75	76.5%
10	100	99	72	72.7%
Mean Success		98.5	73.60	74.7%
Standard Dev.				12.3%

Decrease *Gamma* to 0.8

So, why not further decrease *gamma*? And yes, this looks as good as with *gamma* = 0.9:

Alpha 0.1	Gamma 0.8	Epsilon 0.0		
Experiment	Performed Trials	Successful Trials	Successful Trials w/o Negative Rewards	Share
1	100	98	55	56.1%
2	100	100	82	82.0%
3	100	99	80	80.8%
4	100	99	82	82.8%
5	100	95	46	48.4%
6	100	98	77	78.6%
7	100	99	80	80.8%
8	100	98	88	89.8%
9	100	100	76	76.0%
10	100	99	87	87.9%
Mean Success		98.5	75.30	76.3%
Standard Dev.				13.4%

Decrease *Gamma* to 0.7

This third decrease seems to have it stretched too far. At least, there is no indication of significant improvement:

Alpha 0.1	Gamma 0.7	Epsilon 0.0		
Experiment	Performed Trials	Successful Trials	Successful Trials w/o Negative Rewards	Share
1	100	96	71	74.0%
2	100	100	86	86.0%
3	100	99	80	80.8%
4	100	100	74	74.0%
5	100	97	59	60.8%
6	100	98	85	86.7%
7	100	100	47	47.0%
8	100	96	59	61.5%
9	100	99	81	81.8%
10	100	57	33	57.9%
Mean Success		94.2	67.50	71.0%
Standard Dev.				13.5%

Return to *Gamma* = 0.8, Increase *Alpha* to 0.2

I will stay with *gamma* = 0.8, and now increase *alpha.* The result is a deterioration in average and variance, so I abandon this idea and leave *alpha* at 0.1:

Alpha 0.2	Gamma 0.8	Epsilon 0.0		
Experiment	Performed Trials	Successful Trials	Successful Trials w/o Negative Rewards	Share
1	100	85	2	2.4%
2	100	99	77	77.8%
3	100	96	79	82.3%
4	100	98	76	77.6%
5	100	99	77	77.8%
6	100	69	47	68.1%
7	100	94	50	53.2%
8	100	91	46	50.5%
9	100	99	72	72.7%
10	100	98	53	54.1%
Mean Success		92.8	57.90	61.6%
Standard Dev.				23.9%

Return to *Gamma* = 0.8, *Alpha* = 0.1, setting the initial Q value to the first reward

Although the idea of setting a completely unknown Q-value to the neutral initial value 0 makes very much sense, I want to try out the idea of initializing the Q-values to the first achieved reward per trial (Inspiration: https://en.wikipedia.org/wiki/Q-learning).

The result shows that this makes little sense, in this setting:

Alpha 0.1	Gamma 0.8	Epsilon 0.0		
Experiment	Performed Trials	Successful Trials	Successful Trials w/o Negative Rewards	Share
1	100	96	10	10.4%
2	100	99	84	84.8%
3	100	35	19	54.3%
4	100	56	4	7.1%
5	100	32	16	50.0%
6	100	0	0	
7	100	95	12	12.6%
8	100	71	13	18.3%
9	100	93	8	8.6%
10	100	31	13	41.9%
Mean Success		60.8	17.90	32.0%
Standard Dev.				27.2%

Return to the neutral Q-value initialization, correcting the reward for learning

The idea crossed my mind at one point: The agent is given additional 10 points for reaching the target within deadline. Thinking like a human being, this is a great incentive. But it actually makes no difference to the agent, event if we summed up cumulative rewards per trial, since every successful trial is given the same additional reward.

On the other hand, the additional reward biases the overall reward structure for learning. Imagine the agent turning to the left and thereby a) reaching the destination b) producing an accident with an forward coming car from the oncoming direction. Overall, the agent got an biased reward of 10 - 1 = 9, the unbiased penalty for the move would be -1. I thus decide to correct the reward in case of reaching the destination.

The result sets the new benchmark:

Alpha 0.1	Gamma 0.8	Epsilon 0.0		
Experiment	Performed Trials	Successful Trials	Successful Trials w/o Negative Rewards	Share
1	100	100	81	81.0%
2	100	100	86	86.0%
3	100	98	70	71.4%
4	100	100	89	89.0%

5	100	100	88	88.0%
6	100	99	76	76.8%
7	100	99	84	84.8%
8	100	99	81	81.8%
9	100	99	82	82.8%
10	100	100	81	81.0%
Mean Success		99.4	81.80	82.3%
Standard Dev.				5.3%

Corrected rewards, allowing for 1% Exploration

It remains to play around with *epsilon* in order to again allow for some exploration along the way. 10% exploration was obviously too much deteriorating the agent's performance, but how about at least one random move per trial, in average?

Again, the result is evidence that the performance does not improve, on the contrary:

Alpha 0.1	Gamma 0.8	Epsilon 0.01		
Experiment	Performed Trials	Successful Trials	Successful Trials w/o Negative Rewards	Share
1	100	96	28	29.2%
2	100	70	9	12.9%
3	100	95	51	53.7%
4	100	100	76	76.0%
5	100	99	66	66.7%
6	100	97	48	49.5%
7	100	97	38	39.2%
8	100	95	52	54.7%
9	100	94	54	57.4%
10	100	97	41	42.3%
Mean Success		94	46.30	48.1%
Standard Dev.				18.3%

Summing Up the Parameter Tuning Experience

With the stated evidence, I would suggest to set *gamma* to 0.8 or 0.9, *alpha* to 0.1 and *epsilon* between 0 and 0.01, and correcting the reward for learning purposes. Further evidence with many more experiments will be needed to fully verify, but at the moment I am content with the agent's performance. I have tested this set-up also with more than three agents, and I still performs.

Outlook

In order to increase the testing set-up, I would embed the simulation in a loop, which performs 100 - 500 experiments for combinations of multiple parameter values, initial Q-value (0, reward, constant negative, constant positive) and corrected learning (Y/N), and store the results in a table.

As a plus, I would include variations of number of dummy agents on the streets, from 3 to 80.

Finally, I would include variations of decreasing parameters:

- Let *alpha* decrease linearly with every trial, e.g.
    - *alpha = 1 / math.sqrt(t + 1)* or *alpha = 1 / (t + 1)*
- Let *gamma* decrease non-linearly with every trial, e.g.
    - *gamma = self.gamma - math.exp(t) / math.exp(deadline + t)*
    - *if gamma < 0:*
    - *gamma = 0*
- Let *epsilon* decrease linearly with every trial, e.g.
    - *epsilon = self.epsilon / math.sqrt(self.number_of_trials)*