Train a Smartcab to Drive

Joshua Cook

Abstract

A smartcab is a self-driving car from the not-so-distant future that ferries people from one arbitrary location to another. In this project, you will use reinforcement learning to train a smartcab how to drive.

The task at hand is a well-posed learning problem. A well-posed learning problem is defined in the following way:

> Definition: A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E. (Mitchell, T.M., 1997. Machine learning. WCB.)

For this case, the computer program is the Learning Agent defined in the file `agent.py`.

E is the experience of navigating the grid system defined in `environment.py`.

T is the ability to safey navigate this grid system which includes red lights and other agents with which our agent must not collide.

In order to complete a clear definition, we must define P.

This task is a well-established and solved problem. ALVINN (Autonomous Land Vehicle In a Neural Network) was trained to "effectively follow real roads under certain field conditions" during tests at Carnegie Mellon (Pomerleau, D.A., 1989. Alvinn: An autonomous land vehicle in a neural network (No. AIP-77). CARNEGIE-MELLON UNIV PITTSBURGH PA ARTIFICIAL INTELLIGENCE AND PSYCHOLOGY PROJECT).

The Learning Task

The task as it is defined here is a Markov Decision Process.

> In a Markov decision process (MDP) the agent can perceive a set S of distinct states of its environment and has a set A of actions that it can perform. At each discrete time step t, the agent senses the current state, s_t, chooses a current action, a_t, and performs it. The environment responds by giving the agent a reward, $r_t = r(s_t, a_t)$ and by producing the succeeding state, $s_{t+1} = \delta(s_t, a_t)$. (Mitchell, 1997)

Environment

Your smartcab operates in an idealized grid-like city, with roads going North-South and East-West. Other vehicles may be present on the roads, but no pedestrians. There is a traffic light at each intersection that can be in one of two states: North-South open or East-West open.

US right-of-way rules apply: On a green light, you can turn left only if there is no oncoming traffic at the intersection coming straight. On a red light, you can turn right if there is no oncoming traffic turning left or traffic from the left going straight.

To understand how to correctly yield to oncoming traffic when turning left, you may refer to this official drivers' education video, or this passionate exposition.

Inputs

Assume that a higher-level planner assigns a route to the smartcab, splitting it into waypoints at each intersection. And time in this world is quantized. At any instant, the smartcab is at some intersection. Therefore, the next waypoint is always either one block straight ahead, one block left, one block right, one block back or exactly there (reached the destination).

The smartcab only has an egocentric view of the intersection it is currently at (sorry, no accurate GPS, no global location). It is able to sense whether the traffic light is green for its direction of movement (heading), and whether there is a car at the intersection on each of the incoming roadways (and which direction they are trying to go).

In addition to this, each trip has an associated timer that counts down every time step. If the timer is at 0 and the destination has not been reached, the trip is over, and a new one may start.

Outputs

At any instant, the smartcab can either stay put at the current intersection, move one block forward, one block left, or one block right (no backward movement).

Rewards

The smartcab gets a reward for each successfully completed trip. A trip is considered "successfully completed" if the passenger is dropped off at the desired destination (some intersection) within a pre-specified time bound (computed with a route plan).

It also gets a smaller reward for each correct move executed at an intersection. It gets a small penalty for an incorrect move, and a larger penalty for violating traffic rules and/or causing an accident.

Goal

Design the AI driving agent for the smartcab. It should receive the above-mentioned inputs at each time step t, and generate an output move. Based on the rewards and penalties it gets, the agent should learn an optimal policy for driving on city roads, obeying traffic rules correctly, and trying to reach the destination within a goal time.

Tasks

Setup

You need Python 2.7 and pygame for this project: https://www.pygame.org/wiki/GettingStarted

For help with installation, it is best to reach out to the pygame community [help page, Google group, reddit].

Download

Download smartcab.zip, unzip and open the template Python file `agent.py` (do not modify any other file). Perform the following tasks to build your agent, referring to instructions mentioned in `README.md` as well as inline comments in `agent.py`.

Also create a project report (e.g. Word or Google doc), and start addressing the questions indicated in italics below. When you have finished the project, save/download the report as a PDF and turn it in with your code.

Implement a basic driving agent

Implement the basic driving agent, which processes the following inputs at each time step:

1. Next waypoint location, relative to its current location and heading,
2. Intersection state (traffic light and presence of cars), and,
3. Current deadline value (time steps remaining),
4. And produces some random move/action `(None, 'forward', 'left', 'right')`.

Don't try to implement the correct strategy! That's exactly what your agent is supposed to learn.

Run this agent within the simulation environment with `enforce_deadline` set to `False` (see `run` function in `agent.py`), and observe how it performs. In this mode, the agent is given unlimited time to reach the destination. The current state, action taken by your agent and reward/penalty earned are shown in the simulator.

Implementation

To implement this random movement strategy, I added the following to the `update` method in the `LearningAgent` class.

```
action = random.sample([None, 'forward', 'left', 'right'], 1)[0]
```

Analysis

With a totally random method for selecting the next action, it is mere chance that our Learning Agent will arrive at his destination.

The `reset` method within the `Environment` class ensures that the destination will not be too close to our agent.

```
    # Ensure starting location and destination are not too close
    while self.compute_dist(start, destination) < 4:
        start = random.choice(self.intersections.keys())
        destination = random.choice(self.intersections.keys())
```

Let's assume that by random chance the destination was selected to be 4 moves away from our Learning Agent. In this case, we have a 1 in 81 (3^4) chance of arriving at our destination. This is the *best* possible odds we will have in randomly arriving at our destination. Furthermore, each incorrect move increases our changes by a factor of 3. If we start 4 moves away and make an incorrect choice, we now have a 1 in 243.

With the `enforce_deadline` attribute set to `True`, it is highly unlikely we will arrive at our destination within the target time period. With the `enforce_deadline` attribute set to `False` there is still a hard ceiling to moves of 100 and it is still unlikely we will randomly arrive at our destination.

This was confirmed in five trials, during none of which did our agent arrives it the destination.

Note: this method has been removed from the Learning Agent.

Troubleshooting

While attempting to assess the problem several methods were written and used in order to study the problem. All of these have been removed for submission.

Using the Keyboard to Control the Agent

The following method was used to assign the agent to keyboard control.

```
actions = {'j':'left','k':'forward','l':'right'}
action = actions[raw_input()]
```

Using `self.next_waypoint` to Control the Agent

```
action = self.next_waypoint
```

Embed `IPython` into the

The following `import` statement was added to the head of the file.

```
from IPython import embed
```

Then the following was added just before the end of the `run` method.

```
embed()
```

The Learning Agent

The Learning Agent learn a policy, π mapping a state to an action

$$\pi : S \to A$$

Considering that the agent is being rewarded for its actions, we can consider the cumulative value of the rewards that it receives for a given policy, π

$$V^{\pi} \equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

This is also called the **discounted cumulative reward**.

The task of the Learning Agent is to identify the *optimal policy*

$$\pi^* \equiv \operatorname*{argmax}_{\pi} V^{\pi}(s)$$

with

$$V^*(s) = V^{\pi^*}(s)$$

Given that π is a mapping from a state to an action, the optimal policy in terms of the optimal action can be stated as

$$\pi^* = \operatorname*{argmax}_{a} \left(r(s,a) + \gamma V^*(\delta(s,a))\right)$$

Let us define the substance of the argmax

$$Q(s,a) = r(s,a) + \gamma V^*(\delta(s,a))$$

so that

$$\pi^* = \operatorname*{argmax}_{a} Q(s,a)$$

And now we have clearly stated what we are after a mapping from state to action, now stated solely in terms of state and action.

We can also note that the optimal discounted cumulative reward can be rewritten as

$$V^*(s) = \max_{a'} Q(s,a')$$

Now, we have the recursive function

$$Q(s,a) = r(s,a) + \gamma \max_{a'} Q(\delta(s,a),a')$$

We seek an approximation to this, using iterarion rather than recursion

$$\hat{Q}(s,a) \leftarrow r(s,a) + \gamma \max_{a'} \hat{Q}(s',a')$$

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.

At each time step, process the inputs and update the current state. Run it again (and as often as you need) to observe how the reported state changes through the run.

Modeling State

The `namedtuple` data structure was selected to model state.

```
from collections import namedtuple
```

Next, a `namedtuple`, `State` was defined with a list of attributes.

```
State = namedtuple("State", \
    ["oncoming", "light", "left", "right", "next_waypoint"])
```

A method was defined in order to build a `State` object:

```
def _form_state_object(self, inputs):
    this_state = State(oncoming=inputs['oncoming'],
                       light=inputs['light'],
                       right=inputs['right'],
                       left=inputs['left'],
                       next_waypoint=self.next_waypoint)

    return this_state
```

Then at each time step, the Learning Agent runs

```
 self._form_state_object(inputs)
```

I decided to use these attributes because of the lazy, egocentric sensing abilities of the Learning Agent. Because it can only see what is directly in front of it, it does not make sense to store other values such as destination, location, distance from destination, or remaining time. `oncoming`, `left`, and `right` are important to know in order to prevent a collision. `light` is important to know in order to follow safety laws. A time based variation was not selected because of the impact it might have on the size of our state space. Given that a single trial maay last up to 100 turns, we would need to multiply the dimension of our state space by 100 to accomodate this. As it is not necessary to solve the task, such a parameter was not used.

Implement Q-Learning

Implement the Q-Learning algorithm by initializing and updating a table/mapping of Q-values at each time step.

The Q-values table was initialized as follows as part of the `__init__` method for the Learning Agent.

```
valid_actions = [None,'left','right','forward']
self.Q = {
          (State(oncoming=on,
            light=lt, left=lf,
            right=rt, next_waypoint=nw), action) : 0
          for hd in [(1, 0), (0, -1), (-1, 0), (0, 1)]
          for on in valid_actions
          for lt in ['red','green']
          for lf in valid_actions
          for rt in valid_actions
          for nw in valid_actions
          for action in valid_actions
         }
```

Of note:

- a dictionary comprehension method was used to build out the 8192 `State` permutations.
- the keys of the dictionary correspond to a state,action pair signifying $Q(state, action)$

Now, instead of randomly selecting an action, pick the best action available from the current state based on Q-values, and return that.

The following was added to the `update` method for the Learning Agent.

```
possible_Q = { (self.current_state, action): self.Q[(self.current_state,action)]
               for action in valid_actions }
max_Q = [ key for key, value in possible_Q.items()
           if value == max(possible_Q.values()) ]
choose_one_max = random.sample(max_Q,1)[0]
action = choose_one_max[1]
```

The first line creates a dictionary of Q values possible for the `current_state` give the list of `valid_actions`.

The second line creates a list of keys (`(State,action)` tuples) corresponding to the maximum possible Q in the `current_state`.

The third line randomly chooses one `(State,action)` tuple, necessary because several `(State,action)` tuples may have the same maximal value.

The fourth line assigns just the `action` component from the `(State,action)` tuple to the variable `action`.

Learning Method

Each action generates a corresponding numeric reward or penalty (which may be zero). Your agent should take this into account when updating Q-values. Run it again, and observe the behavior.

The following was added to the `update` method for the Learning Agent.

```
self.next_state = self._form_state_object(self.env.sense(self))
possible_Q = { (self.next_state, action): self.Q[(self.next_state,action)]
                for action in valid_actions }
max_Q = [ key for key, value in possible_Q.items()
          if value == max(possible_Q.values()) ][0]

update = self.learning_method(1, possible_Q, max_Q)
self.Q[(self.state,self.action)] = update
```

The first line identifies the `State` after taking the current `action`.

The second line again creates a list of keys (`(State,action)` tuples) corresponding to the maximum possible Q in the `current_state`.

The third line selects the first `(State,action)` tuple from a list of maximal valued `(State,action)` tuples. It is not neccessary to randomly select a tuple because we are only interested in the Q value.

Finally, we need an Algorithm for learning Q based upon these values. We will first use

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}\left(s',a'\right) \qquad \text{(Learning Method 1)}$$

```
def learning_method(self, method, possible_Q, max_Q):
    if method == 1:
        return self.reward + self.gamma * possible_Q[max_Q]
```

Analysis

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

Of primary import is that the agent is now successful at completing the task. In 20 trials, the agent only failed to reach his goal on the first attempt. All subsequent attempts were successful.

When the agent begins the trials it travels randomly. As it learns from its reinforcement system, it gets better at navigating the system and moving toward its destination.

Convergence

It is important to consider the convergence of our algorithm, in particular because we are using an iterative approach. We are **counting** on the convergence in a reasonable time to give us a workable estimate. Mitchell states that our algorithm *will* converge given a few assumptions:

1. the system is a deterministic Markov Decision Process
2. the immediate reward values are bounded

$$\exists c \text{ such that } |r(s,a)| < c$$

1. the agent selects actions in such a fashion that it visits every possible state-action pair infinitely often

The first we know from the definition of the problem. The second, while not explicitly stated, we have to assume from the included `environment.py` file.

The third, however, is not guaranteed by our method. Consider this line of code:

```
max_Q = random.sample([key for key, value in possible_Q.items()
    if value == max(possible_Q.values())],1)[0]
```

There is a `random` method being used here, but it is only being used to differentiate methods with the **exact** same Q value, typically after initialization. Consider the following interaction

Let

s_0 = `State(oncoming=None, light='red', left=None, right=None, next_waypoint='forward')` $$

and,

`possible_Q = {(s_0, None): 0, (s_0, 'left'):0, (s_0, 'right'):0, (s_0, 'forward): 0 }`

Note, that no state is the lone max. The "correct" choice is obvious to the rational human thinker. `forward` is open, therefore the agent should move `forward`. Suppose, by chance, however, the learner chooses left. The value of $Q(s_0, \texttt{'left'})$ will be updated based upon the reward received and the maximum possible Q from the subsequent state. Suppose that this value is positive (no matter how small). Henceforth, the learner will turn left, even though the ideal move is to go forward. In other words, we must guarantee that

> the agent selects actions in such a fashion that it visits every possible state-action pair infinitely often

Exploration vs Exploitation

One way to ensure that all possible routes are explored is to choose an action probabalistically. Mitchell suggests the following

$$\mathbb{P}(a_i|s) = \frac{k^{\hat{Q}(s,a_i)}}{\sum_j k^{\hat{Q}(s,a_i)}}$$

In order to implement a probabilistic selection, we will need a Pythonic method. I started here, and wanting to keep it simple, went with the linear approach.

```
import random
possible_Q = {'left': 0.1, 'right': 2, 'forward': 0.4, None: 0 }

def probabilistically_select_action(k,possible_Q):
    weights = { key: k**value for key,value in possible_Q.items() }
    weights = { key: int(1000*weight/sum(weights.values()))
                for key, weight in weights.items() }

    print(weights)
    actions = []
    for key,value in weights.items():
        actions += [key]*value
    return random.sample(actions,1)[0]

print(probabilistically_select_action(0.1,possible_Q))
print(probabilistically_select_action(2,possible_Q))
```

```
{'right': 4, 'forward': 180, 'left': 360, None: 454}
None
{'right': 541, 'forward': 178, 'left': 145, None: 135}
forward
```

Note, k has determines the strength of the relationship between $\hat{Q}$ and the probability. A small k leads to a larger chance for a low $\hat{Q}$ and vice versa.

We added this new method to our Learning Agent.

```
possible_Q = {(self.state, action): self.Q[(self.state,action)]
    for action in valid_actions }
self.action = self.probabilistically_select_action(self.k**self.trial,
                                                   possible_Q)[1]
self.current_Q = possible_Q[max_Q]
```

Note that we are increasing the value of k with each trial. If $k > 1$ this means that high values of Q will be selected more often the further into the trials we go.

Revision to the Q-Learning Algorithm

Mitchell presents the following revision to the Q-learning algorithm

$$\hat{Q}(s,a) \leftarrow (1-\alpha_n)\hat{Q}(s,a) + \alpha_n \left[r + \gamma \max_{a'} \hat{Q}\left(s', a'\right) \right] \quad \text{(Learning Method 2)}$$

Here α is a learning rate.

We add this as another learning method

```
def learning_method(self, method, possible_Q, max_Q):
    if method == 1:
        return self.reward + self.gamma * possible_Q[max_Q]
    elif method == 2:
        return (1 - self.alpha)*self.current_Q + \
               self.alpha*(self.reward + self.gamma*possible_Q[max_Q]
    else:
        raise ValueError('Invalid method')
```

Enhance the driving agent

Apply the reinforcement learning techniques you have learnt, and tweak the parameters (e.g. learning rate, discount factor, action selection method, etc.), to improve the performance of your agent. Your goal is to get it to a point so that within 100 trials, the agent is able to learn a feasible policy - i.e. reach the destination within the allotted time, with net reward remaining positive.

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?

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?

The formulas for updating Q-values can be found in this video.

At this point we have two learning methods and three parameters we might tweak in order to tune our learner. In order to assess different values, we now need a metric for measuring the success of our learner. We use 100 trials as the baseline. Each trial has a deadline associated with it. I intend to measure how much of the deadline was used in each trial as a percentage of the initial deadline given. I will run this for different values of $\alpha \in [0,1]$, $\gamma \in [0,1]$, and $k \in (1,1000)$, as well as for our two learning methods.

Measuring Success as a Function of Time Used

At this point we have two learning methods and three parameters we might tweak in order to tune our learner. In order to assess different values, we now need a metric for measuring the success of our learner. We use 100 trials as the baseline. Each trial has a deadline associated with it. I intend to measure how much of the deadline remains in each trial as a percentage of the initial deadline given. I will run this for different values of $\alpha \in [0,1]$, $\gamma \in [0,1]$, and $k \in (1,1000)$, as well as for our two learning methods.

An Optimal Policy

I believe that an optimal policy will use as little of the given time as possible. Considering that not all trips are of the same distance, I am normalizing time remaining upon trip completion by dividing by the initial time alloted.

$$\text{success metric} = \frac{\text{deadline}}{\text{initial deadline}}$$

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
%matplotlib inline
```

Basic Plot of Collected Data

This plot shows trial versus time used (deadline/initial deadline)

```
plt.figure(figsize=(20,6))
fig = plt.plot(
        np.genfromtxt('../results/test-alpha-1-gamma-0.1-k-1.1-method-1.txt',
        delimiter=',')[:,1])
fig = plt.plot(
        np.genfromtxt('../results/test-alpha-1-gamma-0.1-k-1.2-method-1.txt',
        delimiter=',')[:,1])
```

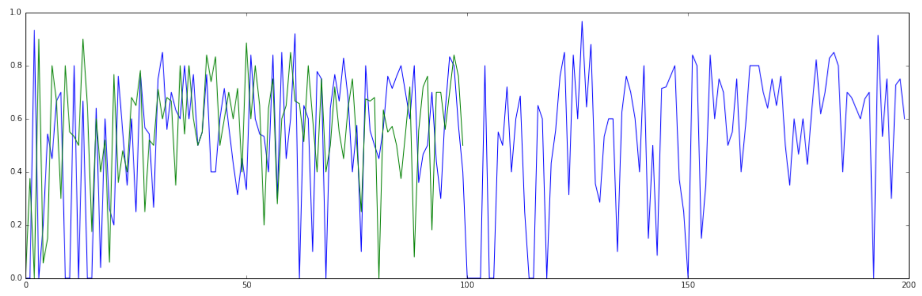

Figure 1: png

```
def define_test_vector(alpha,gamma,k,method):
    file = '../results/test'
    file += '-alpha-'+str(alpha)
    file += '-gamma-'+str(gamma)
    file += '-k-'+str(k)
    file += '-method-'+str(method)
    file += '.txt'
    try:
        mean = np.genfromtxt(file, delimiter=',')[:,1].mean()
    except OSError:
        return [alpha, gamma, k, method, None]
    return [alpha, gamma, k, method, mean]
```

```
method_1 = [define_test_vector(1,gamma,k,1)
              for gamma in [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]
              for k in [1.001,1.01,1.03,1.05,1.07,1.1,1.2]
]
method_2 = [define_test_vector(alpha,gamma,k,2)
              for alpha in [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]
              for gamma in [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]
              for k in [1.001,1.01,1.03,1.05,1.07,1.1,1.2]
]
```

```
tuning_data_method_1 = pd.DataFrame(method_1,
    columns=['alpha','gamma','k','method','mean_time_remaining_on_completion'])
tuning_data_method_2 = pd.DataFrame(method_2,
    columns=['alpha','gamma','k','method','mean_time_remaining_on_completion'])
```

Method 1 peak performance

```
tuning_data_method_1.sort(['mean_time_remaining_on_completion'],
                              ascending=[0]).head(10)
```

-	alpha	gamma	k	method	mean_time_remaining_on_completion
27	1	0.4	1.2	1	0.590374
41	1	0.6	1.2	1	0.566127
6	1	0.1	1.2	1	0.560794
13	1	0.2	1.2	1	0.556146
48	1	0.7	1.2	1	0.541316
34	1	0.5	1.2	1	0.538649
5	1	0.1	1.1	1	0.531257
20	1	0.3	1.2	1	0.524682
19	1	0.3	1.1	1	0.517925
55	1	0.8	1.2	1	0.515909

Method 2 peak performance

```
tuning_data_method_2.sort(['mean_time_remaining_on_completion'],
                               ascending=[0]).head(10)
```

-	alpha	gamma	k	method	mean_time_remaining_on_completion
132	0.3	0.1	1.2	2	0.591882
195	0.4	0.1	1.2	2	0.567945
496	0.8	0.8	1.2	2	0.567771
482	0.8	0.6	1.2	2	0.567520
146	0.3	0.3	1.2	2	0.557671
20	0.1	0.3	1.2	2	0.557002
356	0.6	0.6	1.2	2	0.555718
321	0.6	0.1	1.2	2	0.553543
194	0.4	0.1	1.1	2	0.552575
335	0.6	0.3	1.2	2	0.551680

These results signify that when tuned, we are approaching 60% in `mean_time_remaining_on_completion`.

It is of note that a high k value (1.2) yields the best results. This would encourage us to "exploit" rather than explore. As a result a few more tests were run.

Method 1 peak performance

```
method_1 = [define_test_vector(1,gamma,k,1)
              for gamma in [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8]
              for k in [1.001,1.01,1.03,1.05,1.07,1.1,1.2,1.3,1.4]
]
tuning_data_method_1 = pd.DataFrame(method_1,
    columns=['alpha','gamma','k','method','mean_time_remaining_on_completion'])
tuning_data_method_1.sort(['mean_time_remaining_on_completion'],
                           ascending=[0]).head(10)
```

-	alpha	gamma	k	method	mean_time_remaining_on_completion
33	1	0.4	1.2	1	0.590374
61	1	0.7	1.3	1	0.580744
26	1	0.3	1.4	1	0.579742
53	1	0.6	1.4	1	0.576483
17	1	0.2	1.4	1	0.575397
35	1	0.4	1.4	1	0.571008
51	1	0.6	1.2	1	0.566127
7	1	0.1	1.3	1	0.560990
6	1	0.1	1.2	1	0.560794
62	1	0.7	1.4	1	0.560190

Best Value for Method 1

Note that we have added higher k values and yet are still not able to beat our original best value. At this point, we recommend the following values as our "best model" for Method 1:

- $\gamma = 0.4$
- $k = 1.2$

$$\hat{Q}(s,a) \leftarrow r + 0.4 \max_{a'} \hat{Q}\left(s', a'\right) \tag{Learning Method 1}$$

With probabalistic selection method

$$\mathbb{P}(a_i|s) = \frac{1.2^{\hat{Q}(s,a_i)}}{\sum_j 1.2^{\hat{Q}(s,a_i)}}$$

Method 2 peak performance

```
method_2 = [define_test_vector(alpha,gamma,k,2)
              for alpha in [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]
              for gamma in [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]
              for k in [1.001,1.01,1.03,1.05,1.07,1.1,1.2,1.3,1.4]
]
tuning_data_method_2 = pd.DataFrame(method_1,
    columns=['alpha','gamma','k','method','mean_time_remaining_on_completion'])
tuning_data_method_2.sort(['mean_time_remaining_on_completion'],
                           ascending=[0]).head(10)
```

-	alpha	gamma	k	method	mean_time_remaining_on_completion
539	0.7	0.6	1.4	2	0.608558
168	0.3	0.1	1.2	2	0.591882
206	0.3	0.5	1.4	2	0.585489
494	0.7	0.1	1.4	2	0.583304
439	0.6	0.4	1.3	2	0.582343
196	0.3	0.4	1.3	2	0.581254
449	0.6	0.5	1.4	2	0.580798
512	0.7	0.3	1.4	2	0.577619
259	0.4	0.2	1.3	2	0.577227
89	0.2	0.1	1.4	2	0.576319

Best Value for Method 2

Here we were able to beat our original best value. At this point, we recommend the following values as our "best model" for Method 1:

- $\alpha = 0.7$
- $\gamma = 0.6$
- $k = 1.4$

$$\hat{Q}(s,a) \leftarrow 0.3\hat{Q}(s,a) + 0.7\left[r + 0.6\max_{a'}\hat{Q}\left(s',a'\right)\right] \qquad \text{(Learning Method 2)}$$

With probabalistic selection method

$$\mathbb{P}(a_i|s) = \frac{1.4^{\hat{Q}(s,a_i)}}{\sum_j 1.4^{\hat{Q}(s,a_i)}}$$

Analysis of Optimized Run

We ran 100 trials against our optimized learning algorithm and have here imported the state and action data.

```
import pandas as pd
trial_data = pd.read_csv('100_optimized_runs.txt',
    names=['trial','oncoming','light','left','right','next_waypoint',
    'action','current_Q','update_Q'])
```

```
trial_data.head()
```

	trial	oncoming	light	left	right	next_waypoint	action	current_Q
0	0	None	green	None	None	right	left	0.0
1	0	None	green	None	None	right	None	0.0
2	0	None	green	None	None	right	forward	0.0
3	0	None	red	None	None	right	right	0.0
4	0	None	red	None	None	right	left	0.0

We were interested in the number of times that the agent was assigned a negative Q value.

```
trial_data[trial_data.update_Q<0].count()
```

```
trial            66
oncoming         66
light            66
left             66
right            66
next_waypoint    66
action           66
current_Q        66
update_Q         66
dtype: int64
```

Of these negative Q values, how many were for not selecting the `next_waypoint` and how many were for violating the rules of the road and/or causing a collision? The number of negative Q updates is fairly small so we examined them by hand.

Analyzing negative Q update for traffic violations

```
negative_Q_update = trial_data[trial_data.update_Q<0]
negative_Q_update[trial_data.trial==0]
```

	trial	oncoming	light	left	right	next_waypoint	action	current_Q
0	0	None	green	None	None	right	left	0.0000
2	0	None	green	None	None	right	forward	0.0000
4	0	None	red	None	None	right	left	0.0000
5	0	None	red	None	None	right	forward	0.0000
6	0	None	red	None	None	right	forward	-0.1120
7	0	None	red	None	None	right	left	-0.1120
9	0	None	red	None	None	forward	right	0.0000
11	0	None	green	None	None	left	forward	0.0000
12	0	None	green	None	None	left	right	0.0000
15	0	None	red	None	None	right	left	-0.1456
19	0	None	red	None	None	right	forward	-0.1456
21	0	None	red	None	None	forward	right	-0.3500
22	0	None	green	None	None	left	forward	-0.3500
23	0	None	red	None	None	left	left	0.0000

In the first trial (trial 0), the agent broke the law on indices 4, 5, 6, 7, 15, 19, and 23. 7 times!

```
negative_Q_update[negative_Q_update.trial==1]
```

	trial	oncoming	light	left	right	next_waypoint	action	current_Q
28	1	None	red	None	None	forward	right	-0.455
29	1	None	green	None	None	left	forward	-0.455
30	1	None	red	None	None	left	right	0.000
34	1	None	green	None	None	forward	right	0.000
35	1	forward	red	None	None	left	left	0.000
36	1	forward	red	None	None	left	left	-0.700
37	1	forward	red	None	None	left	right	0.000
44	1	None	green	None	None	forward	right	-0.350
47	1	None	red	None	None	forward	forward	0.000
48	1	None	red	None	None	forward	left	0.000
49	1	None	red	None	None	forward	forward	-0.700
52	1	None	red	None	None	forward	left	-0.700
53	1	None	green	None	None	forward	left	0.000

In the second trial (trial 1) the agent broke the law on indices 35, 36, 47, 48, 49, and 52. 6 times.

```
negative_Q_update[negative_Q_update.trial==2]
```

	trial	oncoming	light	left	right	next_waypoint	action	current_Q
56	2	None	red	None	None	left	left	-0.700
57	2	None	red	None	None	left	forward	0.000
58	2	None	red	None	None	left	left	-0.910
60	2	None	red	None	None	left	forward	-0.700
61	2	None	green	None	forward	left	forward	0.000
62	2	None	red	None	None	left	left	-0.973
64	2	None	red	None	None	left	forward	-0.910

In the third trial (trial 2) the agent broke the law on indices 56, 57, 58, 60, 62, and 64. 6 times.

```
negative_Q_update[negative_Q_update.trial>2][negative_Q_update.trial<23]
```

	trial	oncoming	light	left	right	next_waypoint	action	current_Q
86	4	None	red	None	None	left	left	-0.991900
128	7	None	red	None	None	left	right	-0.350000
169	8	None	green	None	left	forward	right	0.000000
189	9	None	red	None	None	forward	right	0.452997
218	11	None	red	None	None	forward	right	0.075716
278	16	None	red	None	None	forward	right	0.051186
292	17	None	red	None	None	forward	right	-0.313146
310	18	left	red	None	None	forward	right	0.000000
345	20	left	red	None	None	right	left	0.000000
347	20	left	red	None	None	right	forward	0.000000
371	22	forward	red	forward	None	forward	right	0.000000

From here, it showed marked improvement. It broke the law on trial 4, trial 7.

It looks as though it may have caused a collision on trial 22, index 371.

```
negative_Q_update[negative_Q_update.trial>22][negative_Q_update.trial<53]
```

	trial	oncoming	light	left	right	next_waypoint	action	current_Q
422	25	None	red	None	right	forward	right	0.000000
473	29	None	red	None	None	left	right	0.073588
506	32	None	red	None	None	forward	right	0.597147
507	32	None	red	None	None	left	right	0.176201
540	33	None	red	None	None	left	right	-0.223135
545	33	None	red	None	None	forward	right	0.393551
567	34	None	green	left	None	forward	right	0.068910
623	39	None	red	None	None	forward	right	0.164834
659	41	None	red	None	None	forward	right	-0.231320
715	45	None	green	left	None	forward	right	-0.122148
723	45	left	green	None	None	forward	left	0.462902
770	48	None	red	left	None	forward	right	0.000000
824	52	left	red	None	None	forward	left	0.000000
825	52	left	red	None	None	forward	forward	0.000000

It did not break the law again until trial 52, index 824. Unfortunately, it may have caused a collision at this time as well.

```
negative_Q_update[negative_Q_update.trial>52]
```

	trial	oncoming	light	left	right	next_waypoint	action	current_Q	update_Q
950	61	None	green	right	None	forward	left	0.000000	-0.350000
1173	79	None	green	forward	None	left	forward	0.000000	-0.350000
1309	86	None	red	None	forward	forward	left	0.000000	-0.700000
1347	89	left	green	None	None	forward	right	0.158286	-0.302514
1359	90	forward	red	None	None	forward	left	0.000000	-0.700000
1360	90	forward	red	None	None	forward	forward	0.000000	-0.700000
1483	98	forward	red	None	None	forward	right	0.000000	-0.350000

From there it did not break the law again until trial 86, then again on trial 90, and trial 98.

Considering that the agent broke the law more times in the first three turns then in the rest of the trials combined, it is clear that the agent is learning.