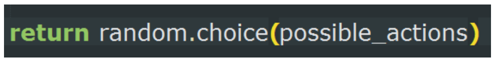
Udacity Machine Learning
6/23/2016

SmartCab Project Report

By David Rose

Initial Agent's Behavior:

The project begins by implementing a random action to take, and ignoring the preset deadline to reach the destination

- At first I direct it to choose from a random direction [None, forward, left, right], and set '*enforce_deadline* = False' to see how it performed.
    - 
```
return random.choice(possible_actions)
```
- Most of the time it fails to reach the destination before the timer hits 0, but sometimes it will still reach its destination in time despite moving randomly through the grid.
- The green/red light status and traffic information is being displayed for each turn, but is not yet considered by the agent in its actions.

Identify and Update State

Next a set of states will need to be identified for the agent process so that it can learn from the optimal actions in the future once the learning algorithm is implemented

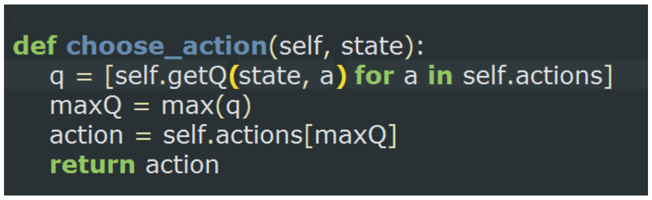
- Add the possible states for the agent to process while deciding actions:
    1. Traffic light: ***inputs['light']***
    2. Time left: ***self.env.get_deadline(self)***
        - Knowing the time left could aid in making different choices at different points in the trip.
    3. Directions of traffic: ***inputs['oncoming'], inputs['left'], inputs['right']***
        - This will give me the adjacent location of the cars relative to the agent, as well as their direction of travel.
    4. Optimal direction for the agent: *self.next_waypoint*
- Soon after though I realized knowing the traffic to the right of me should not have any affect upon the agents actions, so I had this removed
- To compute all the possible states at any point in time, we will multiply all of the combinations of input states together that I consider useful to the model: Light, Cars (left, oncoming), & Waypoint. ***2 x 4 x 4 x 3 = 96***
    1. You could also possible add *deadlinel* to figure out the time left so that the model can learn to ignore smaller errors closer to deadline, and prioritize finishing the trip in time over running a red light. But I find this to increase the state size too much to make it worth it, to only help in the edge cases where it may not make it on time. **96 x deadline [20, 30, 40] = 1920, 2880, 3840 (*Wow that's a lot!)***
    2. So knowing this information, we now will have 96 states (*forgoing deadline)* for the agent to learn from during the course of its driving trials.

Udacity Machine Learning
6/23/2016

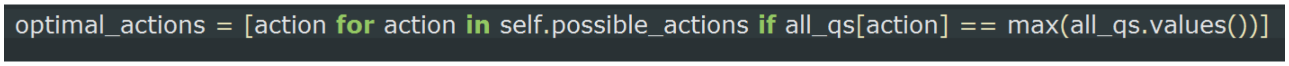
Implement Q-Learning

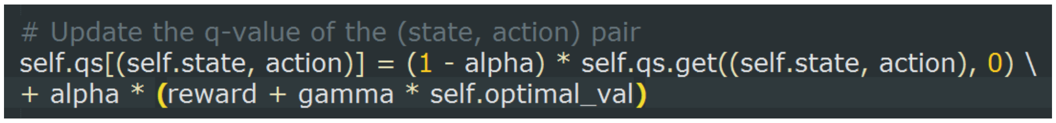
The guide for this project states that we must compute the most optimal action based on our current state, so that means running an exploitive-style model of learning where it acts more on short term gain, giving up possible better rewards that could be discovered by exploring more

- After reviewing the lessons on Q-Learning I initially had some trouble taking it from the abstract to putting code down. This source: https://studywolf.wordpress.com/2012/11/25/reinforcement-learning-q-learning-and-exploration/ helped me lay the foundation to get my first draft of the Q-Learning algorithm working.
- Once I had the basic idea I tried implementing various versions of code:



```
def choose_action(self, state):
    q = [self.getQ(state, a) for a in self.actions]
    maxQ = max(q)
    action = self.actions[maxQ]
    return action
```

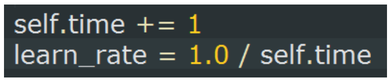
- Trying a similar type of this code above was one method, but my attempt to grab the key related to highest value in the dict would not seem to work correctly with my data, so I checked other sources.
- I eventually found the answer in this method:
    - 

      ```
      optimal_actions = [action for action in self.possible_actions if all_qs[action] == max(all_qs.values())]
      ```
    - And updating the values using this formula:
    - 

      ```
      # Update the q-value of the (state, action) pair
      self.qs[(self.state, action)] = (1 - alpha) * self.qs.get((self.state, action), 0) \
      + alpha * (reward + gamma * self.optimal_val)
      ```
    - which now seemed to run correctly, as well as perform much better than my initial attempt at going random directions.
    - This will run through all the possible moves from the current state and pick the one with the highest Q value.
- Now that the model is taking into account it's previous states rewards, it soon knows the consequences of them and learns that it should not perform actions such as running red lights and hitting other cars. Now that the basic Q-Learning model is in play, the cab only has at most a few missed deadlines.

Enhance the driving agent

Now I began to alter it further, using the reinforcement learning techniques from the lessons, mainly adding the learning rate (alpha) that changes dynamically over the course of each trip. This helped to improve the performance slightly more again.

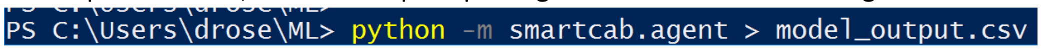
- 

  ```
  self.time += 1
  learn_rate = 1.0 / self.time
  ```
- The code above was added to create a dynamic learning rate that used an incrementing time value to aid the learner quickness in the beginning and further enhance the performance of the model.

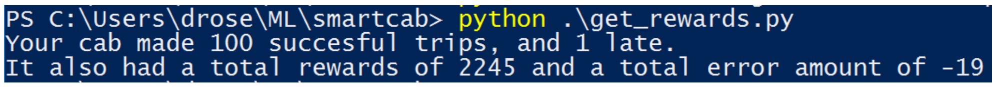
Udacity Machine Learning
6/23/2016

Measuring Performance

While I can passively just view the trips as they are taken and note that it is succeeding most of the time, I would like something more rigorous to measure how often I am exactly making the trip successfully.

- Without altering other parts of the code outside of *Agent.py* I was not sure how best to measure the overall model performance, so I ended up outputting the shell to a CSV file using the command:


```
PS C:\Users\drose\ML> python -m smartcab.agent > model_output.csv
```

- Once it is in there I then wrote a script to comb through the CSV file and iterate over it line by line. If it detects a successful trip, it then it records a **pass**. If the trip is not successful, it records a **fail**. I then add it all up to get an objective measure of how often it made it.
- **Total Rewards:**
    - Along with Pass/Fail, I add up all the rewards gained throughout the trips and report the sum. Unfortunately it seems that this is biased towards longer trips that have more opportunity to give more positive rewards.
    - Possibly an attempt to track the deadline of each trip when it starts and them divide the total rewards by the steps or deadline total to get a reward ratio would be more helpful.
- **Tracking Errors:**
    - A problem I noticed while only tracking the pass rate is that most of the time not a single trip fails to meet the deadline, so it was a bit hard to exactly measure the performance changes over time.
    - So I decided to also keep track of all the errors (negative rewards) and sum them up at the end of the 100 trips.
- After running the agent then parsing through the output with my script **get_rewards.py** I would end up with this message:


```
PS C:\Users\drose\ML\smartcab> python .\get_rewards.py
Your cab made 100 succesful trips, and 1 late.
It also had a total rewards of 2245 and a total error amount of -19
```

- The total code for this script is found on the page below.
- **Learning Rate (Alpha)**: Altering the rate did not change the model performance as much as I had thought. I ran 100 trials 100 times in a row with each rate, to smooth out some of the noise (10,000 runs total), and overall the lower numbers seemed to work best. But the variation even between runs of 10,000 at a time were too large to make any use of.
    - Here is an example of what I encountered (**100 trips run 20 times per rate**), So it seems outside of a 0 rate, nearly everything is equal as far as performance.

Learning Rate	Missed Trips (out of 10,000)	Total Rewards	Total Errors
0	1628	203	-23715
0.001	4	44492	-55
.01	6	44327	-56
.1	4	44991	-55
.5	7	44377	-57
.9	7	44402	-61
1	8	44573	-56

- **Gamma:**
    - I noticed a similar pattern here. The last reviewer mentioned that as this exercise is a relatively simple one, there is not much variation for the agent once they learn the basic examples of what to avoid and the direction to go.
    - So outside of very exteme and non-sensical values like 0, it all performs moderately well, altering both alpha and gamma values.

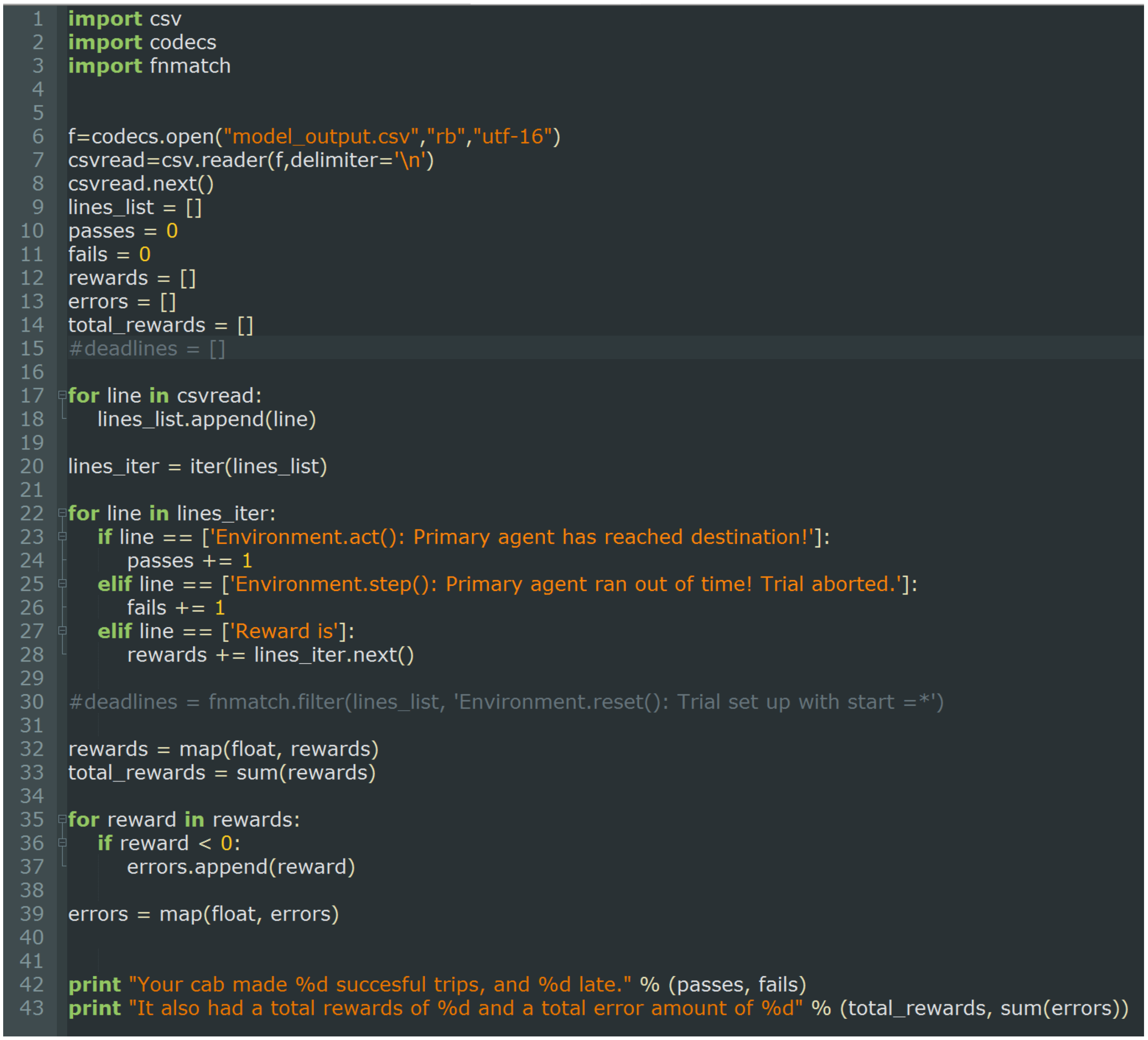
Udacity Machine Learning
6/23/2016

Optimal Policy for the agent

- The agent is not perfect amd still has some problems occasionaly. As it is not keen on exploring unknown choices, once it sees another immediate reward option it will always do that first, which works relatively well in a simple situation such as this.
    - One problem is that the agent will continue to perform negative actions until it learns them firsthand (as in it does not know it can't turn left on red until it tries is). That means it's guaranteed to get multiple errors throughout the simulation as it learns every new situation.
    - In the previous trial I ran, it took up until the 92nd trip for the agent to realize it can not turn left on green when there is oncoming traffic. So while it will eventually learn all negative state/actions, it can take longer than you would like in an ideal world
    - Now in this simple game, it is not a big deal, but you can imagine how much more dangerous that would be in a real life automated car, **"Car's never encountered a speebump before, so it just goes along and hits it at 30mph, ouch!"**
- If ignoring the rules of the project one can insert a fairly simple policy coding in the rules of the road combined with following the next ideal waypoint towards the goal.
- Just following the next waypoint, while specifically obeying the rules such as: don't cross another car's pass and don't turn left or go straight on red, the agent would get the optimal amount of rewards while avoiding any negative amounts.

Code for reviewing the performance of the model on page below

Udacity Machine Learning
6/23/2016



```python
import csv
import codecs
import fnmatch


f=codecs.open("model_output.csv","rb","utf-16")
csvread=csv.reader(f,delimiter='\n')
csvread.next()
lines_list = []
passes = 0
fails = 0
rewards = []
errors = []
total_rewards = []
#deadlines = []

for line in csvread:
    lines_list.append(line)

lines_iter = iter(lines_list)

for line in lines_iter:
    if line == ['Environment.act(): Primary agent has reached destination!']:
        passes += 1
    elif line == ['Environment.step(): Primary agent ran out of time! Trial aborted.']:
        fails += 1
    elif line == ['Reward is']:
        rewards += lines_iter.next()

#deadlines = fnmatch.filter(lines_list, 'Environment.reset(): Trial set up with start =*')

rewards = map(float, rewards)
total_rewards = sum(rewards)

for reward in rewards:
    if reward < 0:
        errors.append(reward)

errors = map(float, errors)


print "Your cab made %d succesful trips, and %d late." % (passes, fails)
print "It also had a total rewards of %d and a total error amount of %d" % (total_rewards, sum(errors))
```