Training a Smartcab to Drive

May 18, 2016

1 Training a Smartcab to Drive

By Miguel Morales mimoralea@gmail.com

```
In [1]: import pandas as pd
        import numpy as np
        df = pd.read_csv("analysis-1463027538.csv", index_col=0)
```

1.1 Project Description

In this project we will be creating an agent capable of learning basic traffic rules by using Reinforcement Learning. This document will briefly discuss the thought process we went through get this agent to an over 90% accuracy.

1.2 From Random to Expert System

The first impulse when I opened this project is to change the algorithm to make the car drive. Even if the project description warns over this impulse, it is just natural to want to see the car accomplish the goal as soon as possible. So, this is the first thing I added to the source code. A simple, even suboptimal Expert System that 'knows' how not to crash while following the directions provided by the planner. The code looks something like this:

```
In [ ]: ## NOT TO BE RAN
        @staticmethod
        def get_planned_action(inputs_tuple):
            light, left, right, oncoming, planner = inputs_tuple
            # None, 'forward', 'left', 'right'
            # valid_actions = [None, 'forward', 'left', 'right']

            if light == 'red' and oncoming != 'left' and left != 'oncoming' and planner == 'right':
                return LearningAgent.valid_actions.index('right')

            if light == 'red' and oncoming != 'right' and right != 'oncoming' and planner == 'left':
                return LearningAgent.valid_actions.index('left')

            if light == 'green' and oncoming is not None and planner == 'left':
                return LearningAgent.valid_actions.index(None)

            return LearningAgent.valid_actions.index(planner)
```

I added this as a static method and does nothing much but avoid a collision. The car arrived to the destination, so now we need to work on automation this decision making.

1.3 From Expert System to Automated

For this, we had to implement Q-Learning, at first basic Q-Learning seemed like it wouldn't be enough, speacially since we are being asked to have the agent learn within 100 iterations. Usually, Q-Learning algorithms are to be executed in thousands of episodes, so we implemented a random action filter with decay, as well as a planned action with decay, followed by the Q-Learning decision. In the end the agent performs very well since it does lots of random exploration initially to avoid suboptimal actions long term, and it also exploits immediately by randomly relying on the planner, then the algorithm run on pure Q-Learning with a very high success rate.

Let's discuss the deatils implemented:

1.4 Implementing Q-Learning

Implementing the algorithm itself was not necessarily difficult, however, selecting the appropriate parameters was more of a challenge. For this, I decided to collect data from the executions on lots of variations of the parameter, for example, alpha and gamma [0.2, 0.5, 0.8], with and without randomization and planning, and various combination of values for initializing the q-table [-5, -2, 0, 2, 5].

1.4.1 State Representation

The best state representation I came up with was to initialize the Q table the following way:

```
In [ ]: # NOT TO BE RAN
        self.Q = np.random.uniform(self.minv, self.maxv, size=(2, 4, 4, 4, 3, 4))
```

With 2 indeces for the light ('red', 'green'), 3 for the 4 possible traffic values (None, 'forward', 'left', 'right'), 3 different values for the planner ('forward', 'left', 'right') and finally the 4 actions the agent is able to take (None, 'forward', 'left', 'right').

However, I considered multiple other state representations, including one where the agent would keep track of the time step remaing. This in particular didn't worked that well probably because the state space was so huge the agent was not able to learn on time.

Additionally, instead of indexing the values I attempted to discritize the state into a single number. This however, was more of a problem since several of the inputs, like for example the state of the light, have a very small number of states. Then, allowing a whole decimal space for 2 values was wasteful.

In the end, concatenating the indeces into a string was the most efficient method, thus the one used for this project and it performed well.

1.4.2 Learning Rate

For the learning rate, as metioned above we tried values ranging from 0.2 to 0.8, however, there weren't significant differences between the two alone. For example, in both, the maximum number of successes was above 90, though overall the higher the learning rate the higher the median and mean successes.

```
In [18]: df.loc[(df.alpha == 0.20) & (df.plr == 0.0) & (df.rar == 0),['total_reward','successes']].desc
```

Out[18]:

	total_reward	successes
count	74254.000000	74254.000000
mean	1309.439397	8.117771
std	920.177121	15.846170
min	-33.500000	0.000000
25%	482.500000	0.000000
50%	1220.000000	1.000000
75%	2031.000000	8.000000
max	3864.000000	93.000000

```
In [19]: df.loc[(df.alpha == 0.80) & (df.plr == 0.0) & (df.rar == 0),['total_reward','successes']].desc
```

Out[19]:

	total_reward	successes
count	73369.000000	73369.000000
mean	1329.322275	9.150009
std	908.638518	16.104411
min	-17.500000	0.000000
25%	507.000000	0.000000
50%	1270.500000	2.000000
75%	2063.500000	11.000000
max	3584.500000	99.000000

Interestingly, the more successes didn't necessarily imply more total rewards, and this is something that some agents had problems with. They would prefer stay in the same spot and avoid traffic accidents at all costs instead of exploring and attemptting to get higher future rewards. This is where the discount factor had its effect.

1.4.3 Discount Factor

Modifying the discount factor allows us to give more importance to later rewards than short term. What we found, as mentioned above, is that the short-sighted agent would prefer to stay put and avoid collissions. It would start exploring but shortly after a few mistakes, it would always return the best policy as "None". We found however, that a higher alpha did not necessarily imply higher amount of successes as we can see on the tables below:

```
In [55]: df.loc[(df.gamma == 0.20) & (df.plr == 0.0) & (df.rar == 0),
                ['total_reward','successes', 'at-none', 'at-forward', 'at-left', 'at-right']].describe(
```

Out[55]:

	total_reward	successes	at-none	at-forward	at-left
count	72030.000000	72030.000000	72030.000000	72030.000000	72030.000000
mean	1317.609482	10.437831	933.907816	143.062571	126.274205
std	924.777093	17.876505	770.715044	199.158170	265.870670
min	-30.500000	0.000000	0.000000	0.000000	0.000000
25%	483.125000	0.000000	282.000000	11.000000	10.000000
50%	1236.000000	2.000000	762.000000	30.000000	27.000000
75%	2039.000000	11.000000	1434.000000	245.000000	124.000000
max	3454.500000	99.000000	3085.000000	1096.000000	2104.000000

	at-right
count	72030.000000
mean	183.292059
std	298.764670
min	0.000000
25%	15.000000
50%	68.000000
75%	208.000000
max	2108.000000

```
In [56]: df.loc[(df.gamma == 0.80) & (df.plr == 0.0) & (df.rar == 0),
                ['total_reward','successes', 'at-none', 'at-forward', 'at-left', 'at-right']].describe(
```

Out[56]:

	total_reward	successes	at-none	at-forward	at-left
count	74387.000000	74387.000000	74387.000000	74387.000000	74387.000000
mean	1401.325359	7.880638	846.825050	196.127173	104.206246
std	857.246758	13.956613	699.224302	276.559838	151.592018
min	-6.500000	0.000000	0.000000	0.000000	0.000000
25%	667.500000	0.000000	306.000000	14.000000	12.000000
50%	1369.500000	2.000000	663.000000	62.000000	51.000000

75%	2087.000000	9.000000	1223.000000	275.000000	123.000000
max	3662.000000	89.000000	3085.000000	1230.000000	964.000000

	at-right
count	74387.000000
mean	258.644669
std	314.117155
min	0.000000
25%	29.000000
50%	105.000000
75%	414.000000
max	1447.000000

1.5 Pitfalls

We had several pitfalls during this project, as we mentioned before, the state representation was one. Other didn't seem so obvious, like a higher learning rate, or discount factor is necessarily best. We found that intermediate values gave the best combination as shown below.

1.6 Tweaking Q-Learning Parameters

Finding the best parameters was tedious, and we found that overall the bootstrapping method and the randomized action taker was in fact useful. Also, we found that the best performances had an alpha around 0.6 and a gamma a little below 0.5, also, initializing the q-values in a 'positive' matter worked best. That, as explained in the lectures, a higher q-value initilization the more the agent would be willing to explore. In the case of this project I found random values from around 0 to a little less than 5 was best.

```
In [57]: df.loc[(df.successes > 95),
                ['attempt', 't', 'alpha', 'gamma', 'minv', 'maxv', 'plr', 'rar']].describe()
```

Out[57]:

	attempt	t	alpha	gamma	minv
count	2018.000000	2018.000000	2018.000000	2018.000000	2018.000000
mean	99.105055	7.987116	0.623687	0.419871	-1.542616
std	0.952966	6.645484	0.193714	0.211134	1.865433
min	96.000000	0.000000	0.200000	0.200000	-5.000000
25%	98.000000	3.000000	0.500000	0.200000	-2.000000
50%	99.000000	7.000000	0.500000	0.500000	0.000000
75%	100.000000	11.000000	0.800000	0.500000	0.000000
max	100.000000	39.000000	0.800000	0.800000	0.000000

	maxv	plr	rar
count	2018.000000	2018.000000	2018.000000
mean	2.806244	0.005874	0.003336
std	2.130967	0.002631	0.003581
min	0.000000	0.000000	0.000000
25%	0.000000	0.005154	0.000000
50%	2.000000	0.006363	0.000000
75%	5.000000	0.007070	0.007070
max	5.000000	0.013303	0.009698

1.6.1 State Representation

For the state representation we found the least the better. Initially, the idea of adding inputs seems like it would be great, maybe our agent would be able to rush through a red light if the time required it. However, this didn't hold to be given the 100 attempts constraint since by the 100th attempt, the agent with a large state space would still not have learned important and basic driving skills.

1.6.2 Learning Rate

For the learning rate we settle for 0.7, this was an intermediate value that allowed the agent to learn somewhat quickly, without getting stuck in a local optima.

1.6.3 Discount Factor

The best discount rate we found was, at least at first, surprisingly low. Most humans have a bias to think long term means better, for example, investments are better if you go in long term, or even relationships should last long. So initially, I only tried high numbers for the discount factor. However, the agent was best at balancing the rewards and getting higher number of successful deliveries when taking care of short term matters as well. This could be because things as simple as obeying traffic signals was rewarded positively on this environment.

1.7 Additional Techniques

As mentioned before, our agent work best by bootstrapping knowledge and frequently randomizing actions. We indeed added this features as seens in "Machine Learning for Trading" for the randomized, and "Reinforcement Learning" for bootstrapping of knowledge. In the end however, the agent is able to fully learn the best policy which is even different than the planned action. For example, often the agent learns to go around a block when there is traffic in the intersection which is something we found amazing.

1.7.1 Putting Randomness Back

We added the randomness given at the project's skeleton but with a 0.9 decay rate which got applied everytime the agent took a random action. So initially we would see a combination of max action (q-learning), planned actions with random actions, but quickly after the 15th attempt the max actions would be taking over.

1.7.2 Putting Expert System Back

The planned actions serve as a bootstrap of knowledge to the agent, which although it doesn't allow it to learn the actual optimal policy, it allows it to learn where the max actions are quickly enough that we get almost perfect score in the end. It is surely not needed since the agent is able to find a high streak of success eventually anyways, but it indeed makes the agent learn a lot faster. It analogous to training wheels, they might not teach you to actually ride a bike correcly, but they get you going.

1.8 Final Results

Our results were outstanding, the agent is capable of consistently getting higher than 90 successes within the 100 attempts, the best part is that the agent learns, within the 100 attempt, that it is best to circle around a block than to wait on red. This, though, creates odd driving actions, but it ensures to get the highest numbers of successes in the end. Maybe this agent makes a great New York taxi driver.