UDACITY MACHINE LEARNING NANODEGREE

PROJECT 4: Train a Smartcab to Drive

By Stephen Fox

Submitted September 2016

Overview

The objective of this project was to apply reinforcement learning techniques to train a self-driving agent to efficiently reach its destination within the allotted time. Udacity provided four python files for this purpose:

- environment.py: creates the smartcab environment, including dummy agents (i.e. traffic)
- planner.py: provides a high-level planner for the smartcab to follow to reach destination
- simulator.py: provides the simulation and graphical user interface
- agent.py: creates the forum for smartcab reinforcement learning

In this project, only the agent.py file was modified. The initial modification was to get the smartcab to move around randomly within the environment.

After random movement was achieved, the next step was to implement the Q-Learning algorithm, enabling the smartcab to learn how to navigate its environment to avoid accidents, avoid breaking traffic laws and successfully reach its destination within the allotted time. To implement reinforcement learning, a set of possible states was defined for the smartcab. From each state, the smartcab could take four possible actions, namely go left, go right, go forward or do nothing. In order to learn, erroneous actions (e.g. those actions causing an accident or breaking a traffic rule) incurred a penalty (in the form of a negative reward) whereas correct actions earned a reward. Reaching the destination before running out of time earned a relatively large reward. Once the state, action and reward space was defined for the smartcab, it was possible to apply the Q-Learning algorithm to let the smartcab learn. As part of this process, the learning variables (alpha, gamma, and epsilon) were tuned, to find values that gave good performance.

Implement a Basic Driving Agent

The first task was to get the smartcab moving around the environment, without any regard to an optimal driving strategy. This was achieved by setting the ‘action’ variable to a random choice of the four possible actions (none, left, right or forward). After adding the code, I observed that with random actions and an infinite time horizon, the smartcab inevitably makes it to the destination.

However, in the process it often makes illogical moves that take it further from the destination and also sometimes break traffic rules, and it loses points because of these moves.

Inform the Driving Agent

The first major step in implementing reinforcement learning was to define a set of states that are appropriate for modeling the smartcab and environment. The key sources of state variables are the current inputs at the intersection (from environment.py) and also the next waypoint for the smartcab to take to reach the final destination (from planner.py). One challenge when implementing reinforcement learning is to balance the number of states appropriately. If you choose a relatively large number of possible states, learning will take a great deal of time, since the agent in theory needs to visit all states several times. If you choose a relatively small number of states, you might not be properly modeling and capturing the complexities that exist in the environment.

Each of the input and next waypoint variables can take on the following values (with the number in brackets showing the possible number of values a given variable can take):

- Light = red or green (2)
- Oncoming = None, right, left, forward (4)
- Right = None, right, left, forward (4)
- Left = None, right, left, forward (4)
- Next waypoint = right, left, forward (3)

If one considers every single possible combination of these five variables, there are 384 possible states. This is far too many to use for the reinforcement problem here, given that the evaluation number of simulations is only 100 and each simulation consists of several 10s of moves, and also given that some states are far more prevalent than others. One would need to simulate many more moves to ensure that each of the 384 possible states was visited several times by the smartcab. Therefore, the number of states needs to be greatly reduced. Fortunately, by considering the rules of the road, one can determine which combinations of inputs are important for road decisions and thereby greatly reduce the number of possible states that the smartcab will encounter, as demonstrated in the following decision tree (Figure 1):

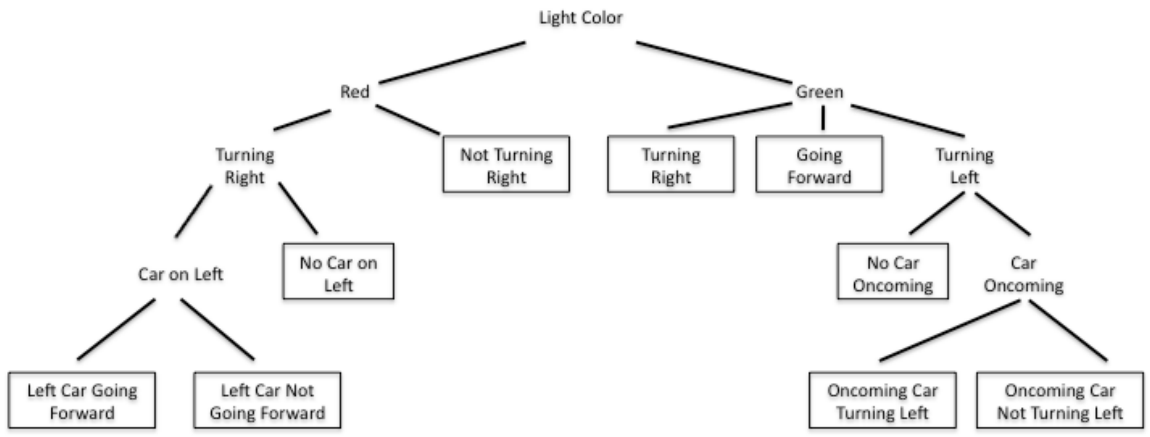

Figure 1: Decision Tree Demonstrating Nine Possible Smartcab States

In the figure, a final state is contained within a rectangular box. To demonstrate how the decision tree works, consider the first left branch. This branch states that the traffic light is red. In such a situation, all that matters to a car is whether they are turning right or not. If the car is not turning right, it should not matter whether the next waypoint is left or forward nor whether there is traffic coming from any of the three directions, since in all cases, waiting at the red light is the proper course of action. Any other action would earn a negative reward, since it breaks a traffic law, and thus all those combinations of inputs can be modeled as a single scenario.

Based on this tree, nine states are adequate for modeling the smartcab in the environment as described for this project. These nine states were used to describe the smartcab state at any given point in the simulation, using the 'self.state' variable. I believe these states are appropriate for the problem at hand, given that they utilize the input and next waypoint variables and properly use groupings to encompass the rules of the road. The number of states (9) seems reasonable given the scope of the simulation. With 100 trips and each trip encompassing 10s of steps, it is reasonably likely that each of these states will be visited several times. In particular, the most common states will be visited on numerous occasions.

Implement a Q-Learning Driving Agent

In order to implement a Q-learning driving agent, the following formula was encoded into the agent.py file:

$$Q(s,a) = (1 - \text{alpha}) * Q(s,a) + \text{alpha} * (\text{reward} + \text{gamma} * Q(s',a'))$$

where s, a, s' and a' are the current state, current action, future state and future action for the agent. The agent was encoded to choose the action with the maximum Q-value, for any given state. The Q-values were all initialized at zero, to ensure there were no initial biases built into the agent's decision. In the case of a tie in Q-values, the agent would randomly choose an action and then update the Q-values accordingly.

For the baseline scenario, I used alpha and gamma values of 0.1. Under these conditions, the agent begins to learn and takes fewer illogical steps than when random actions were always taken. The agent also reaches the destination within the allotted time more frequently. This behavior is occurring because with each step, the agent updates the Q-values and therefore makes slightly more informed decisions on the next step, reducing the randomness and increasing the likelihood of success.

Performance metrics can be used to quantify the difference between the Q-Learning agent and the random agent. I generated the following performance metrics in this project:

- Number of successful trips
- Number of wrong moves
- Fraction of deadline remaining
- Cumulative reward earned
- Last failed trip
- Number of failed trips in the last 10 trips

The 'number of successful trips' measures how often the agent successfully reached the destination before running out of time for any given simulation of a certain number of trials. The 'number of wrong moves' measures how often the agent incurred a penalty (negative reward) for a given move. This could be the result of either an invalid move (e.g. breaking a traffic rule) or an incorrect move (e.g. not moving in the correct direction towards the destination). Note that remaining stationary (i.e. a move of 'none') does not count as a wrong move under this interpretation, even though remaining stationary might not be the correct strategy, since 'none' always generates a reward of zero. The 'fraction of deadline remaining' is a measure of how quickly the agent reaches the destination. For example, if a given trip has a deadline of 50 and the agent reaches the destination in 20 moves, then the 'fraction of deadline remaining' would be equal to 0.60 ((50 – 20) / 50). The 'cumulative reward earned' is the sum total of the reward over any given simulation of a certain number of trials. A higher cumulative reward means the agent is making fewer mistakes and reaching the destination more often, all else equal. The 'last failed trip' measures the last time in any given simulation that the agent failed to reach the destination. For example, on a simulation with 100 trials, if the agent failed to reach the destination on trials 1-5, 12 and 45, then the 'last failed trip' would be equal to 45. All else equal, an agent that is learning from its mistakes will stop failing sooner than an agent that is not learning. Finally, the 'number of failed trips in the last 10 trips' counts how many of the final 10 trips ended with the agent failing to reach the destination. For example, in a simulation of 100 trips, if the agent failed on trips 93 and 95 but passed on the balance of the last 10 trips, then this performance metric would have a value of 2. This performance metric is similar to the 'last failed trip' metric, but is not as prone to random unluckiness occurring on the very last trip and provides a good measure of how the agent would perform under a final assessment, once learning has run for many trips.

Now that the performance metrics used have been defined, lets compare the Q-Learning agent's performance versus a random agent (Table 1). In all cases henceforth, each set of parameters was simulated 10 times (with 100 trips per simulation) and the results shown are the average and the standard deviation (shown as ±) of those 10 simulations:

Table 1: Basic Q-Learning Agent vs. Random Agent (100 Trips; 10 simulations each)

Agent	**# Successful Trips**	**# Wrong Moves**	**Avg. fraction of deadline remaining**	**Cumulative Reward**	**Last Failed Trip**	**# Failed Trips on last 10 trips**
Random (alpha = 0; gamma = 0)	20.4 ± 3.2	1,592 ± 41	0.095 ± 0.022	5 ± 45	99.7 ± 0.5	7.4 ± 1.2
Q-Learning (alpha = 0.1; gamma = 0.1)	82.1 ± 28.9	10 ± 3	0.490 ± 0.179	1,885 ± 501	29.9 ± 48.1	1.8 ± 3.0

The basic Q-Learning agent behaves very differently than the random agent. Instead of moving in random, often illogical ways, the Q-Learning agent begins to exhibit reasonable behavior. This change is occurring because the Q-Table is being updated based on the outcome of each move, and thus is biasing the agent's movements towards moves that earn rewards.

The Q-Learning agent significantly outperforms the random agent in every performance metric, as shown in Table 1. It completes approximately 4 times as many trips, makes far fewer wrong moves and generates a much greater cumulative reward. However, there is clearly an opportunity for optimization, since the Q-Learning agent still failed on average to complete nearly 18% of the trips. It is worth noting that the variance between simulations was significant (as seen by the relatively large standard deviation values). For each of the 10 simulations, the Q-Learning agent either succeeded at or near 100% of trips or it failed on at least 50% of the trips. This suggests that local minima are problematic for this learning environment, as will be addressed during the portion on epsilon greedy learning.

Improve the Q-Learning Driving Agent

In order to improve on the basic Q-Learning agent just presented, several approaches were utilized:

1. Tune the alpha and gamma values
2. Implement ‘epsilon greedy’ learning
3. Add a time decay to epsilon

Each of these three approaches will now be discussed.

Tune the Alpha and Gamma Parameters

The learning rate (alpha) and discount factor (gamma) are critical parameters in reinforcement learning. If alpha is too small, learning takes too long, whereas if alpha is too large, the agent will put most of the emphasis on the most recent action, which might not be desirable. If gamma is too small, too much weight is placed on only the current reward whereas if gamma is too large, too much weight is placed on the future and the present is deemphasized, which can lead to propagation of errors and instabilities in the simulation.

As a first step, various values of alpha were tested while keeping gamma fixed (at 0.1). The performance results of this tuning exercise are presented in Table 2:

Table 2: Alpha Parameter Tuning (Gamma fixed at 0.1; 100 Trips; 10 simulations each)

Alpha Value	# Successful Trips	# Wrong Moves	Avg. fraction of deadline remaining	Cumulative Reward	Last Failed Trip	# Failed Trips on last 10 trips
0*	21.0 ± 3.8	1,575 ± 54	0.101 ± 0.030	2 ± 60	99.9 ± 0.3	7.6 ± 1.3
0.01	75.7 ± 31.1	10 ± 3	0.452 ± 0.177	1,805 ± 562	42.4 ± 49.9	2.6 ± 3.4
0.1	82.1 ± 28.9	10 ± 3	0.490 ± 0.179	1,885 ± 501	29.9 ± 48.1	1.8 ± 3.0
0.2	61.3 ± 26.7	11 ± 2	0.368 ± 0.158	1,558 ± 473	78.6 ± 41.6	2.7 ± 1.9
0.5	66.5 ± 28.3	12 ± 4	0.399 ± 0.163	1,646 ± 509	67.2 ± 47.0	3.5 ± 3.1
1.0	60.7 ± 27.3	13 ± 4	0.367 ± 0.168	1,531 ± 483	69.4 ± 47.9	3.8 ± 3.0

*Statistically different from the base case at the 0.05 significance level

Given the large variance in the results, a one-tailed T-Test was used to determine whether any of the results in Table 2 were statistically different from the base case (alpha = 0.1), based on the number of successful trips. It was determined that only the scenario where alpha was set to zero was statistically different (at the 0.05 significance level) from the base case. Thus, one can conclude that some value of alpha greater than zero is required for learning but that the actual choice of value does not appear critically important in this environment. For the purposes of moving forward, I selected the value with the most favorable mean for each of the performance measurements, which happened to be the alpha value of 0.1 in all cases.

Next, various values of gamma were tested while keeping alpha fixed (at 0.1). The performance results of this tuning exercise are presented in Table 3:

Table 3: Gamma Parameter Tuning (Alpha fixed at 0.1; 100 Trips; 10 simulations each)

Gamma Value	# Successful Trips	# Wrong Moves	Avg. fraction of deadline remaining	Cumulative Reward	Last Failed Trip	# Failed Trips on last 10 trips
0*	54.6 ± 24.5	15 ± 3	0.326 ± 0.143	1,442 ± 430	80.4 ± 40.3	4.3 ± 2.7
0.01*	49.1 ± 19.1	12 ± 4	0.299 ± 0.110	1,319 ± 346	89.3 ± 31.4	4.2 ± 2.1
0.05*	54.4 ± 24.3	13 ± 3	0.327 ± 0.153	1,435 ± 433	79.3 ± 41.5	4.4 ± 2.6
0.1	82.1 ± 28.9	10 ± 3	0.490 ± 0.179	1,885 ± 501	29.9 ± 48.1	1.8 ± 3.0
0.15	67.4 ± 28.2	12 ± 2	0.413 ± 0.175	1,635 ± 494	59.0 ± 50.8	2.6 ± 2.5
0.2*	52.2 ± 25.4	12 ± 3	0.303 ± 0.147	1,360 ± 483	87.4 ± 31.4	5.3 ± 2.9
0.5*	60.1 ± 27.8	27 ± 25	0.353 ± 0.166	1,541 ± 483	73.4 ± 41.9	3.7 ± 3.1

*Statistically different from the base case at the 0.05 significance level

Given the large variance in the results, the one-tailed T-Test was used to determine whether any of the results in Table 3 were statistically different from the base case (gamma = 0.1), based on the number of successful trips. It was determined that very low (0, 0.01) and high (0.2, 0.5) values of gamma were statistically different from the base case. Therefore, a gamma value greater than 0.05 but less than 0.2 appears to be most beneficial for learning. For the purposes of moving forward, I selected the value with the most favorable mean for each of the performance measurements, which happened to be the gamma value of 0.1 in all cases.

Implement Epsilon Greedy Learning

The previously discussed exercise of tuning alpha and gamma demonstrated a very important point about this particular learning environment: given the large variance in simulation results for the same values of alpha and gamma, this environment is prone to local minima. Depending on the initial random movements made by the smartcab, the learning agent is prone to getting stuck in unproductive regions, where learning no longer progresses. Epsilon greedy learning is an excellent technique for forcing the agent out of the local minima, by periodically making random moves, with a probability equal to epsilon.

In order to determine whether epsilon greedy learning would aid learning in this environment, I tested various epsilon values, with the results shown in Table 4:

Table 4: Epsilon Tuning (Alpha & Gamma fixed at 0.1; 100 Trips; 10 simulations each)

Epsilon Value	**# Successful Trips**	**# Wrong Moves**	**Avg. fraction of deadline remaining**	**Cumulative Reward**	**Last Failed Trip**	**# Failed Trips on last 10 trips**
0	82.1 ± 28.9	10 ± 3	0.490 ± 0.179	1,885 ± 501	29.9 ± 48.1	1.8 ± 3.0
0.001	67.2 ± 28.9	11 ± 2	0.392 ± 0.173	1,641 ± 532	56.5 ± 49.1	2.0 ± 2.3
0.005	79.8 ± 21.9	18 ± 4	0.464 ± 0.115	1,894 ± 381	56.2 ± 35.0	0.9 ± 1.9
0.01*	97.2 ± 4.9	19 ± 3	0.571 ± 0.042	2,185 ± 90	23.2 ± 28.5	0.0 ± 0.0
0.02*	92.1 ± 8.4	27 ± 7	0.545 ± 0.041	2,139 ± 180	35.5 ± 32.0	0.1 ± 0.3
0.05*	97.5 ± 2.2	52 ± 10	0.566 ± 0.028	2,202 ± 66	39.7 ± 40.6	0.1 ± 0.3
0.1	96.5 ± 2.0	102 ± 12	0.540 ± 0.016	2,206 ± 39	66.2 ± 34.1	0.4 ± 0.7
0.2	93.3 ± 1.8	200 ± 11	0.516 ± 0.024	2,178 ± 69	85.5 ± 8.6	0.6 ± 1.3
0.5	69.2 ± 3.7	663 ± 37	0.304 ± 0.025	1,830 ± 59	97.2 ± 2.1	2.7 ± 1.2

*Statistically different from the zero epsilon case at the 0.05 significance level

The results in Table 4 demonstrate that epsilon greedy learning had a strong effect on reducing the variance in the number of successful trips. With an epsilon value of zero, the mean is 82.1 trips and the standard deviation is 28.9, meaning many simulations result in a large number of failed trips, whereas with an epsilon of 0.05, the mean is 97.5 trips and the standard deviation is only 2.2, meaning that most simulations resulted in over 90% success rate and that the chance of a simulation resulting in many trips failing was close to zero.

The one-tailed T-Test was used to determine whether any of the results in Table 4 were statistically different from the base case (epsilon = 0), *based on the number of failed trips in the last ten trips*. The number of failed trips in the last 10 trips was identified as important for the purposes of judging the quality of the reinforcement learning, since if the agent has learnt how to navigate the streets, it should rarely fail during simulations 91-100. It was determined that modestly low (0.01, 0.02 and 0.05) values of epsilon were statistically different from the base case, whereas very low (0.001, 0.005) and high (0.1, 0.2, 0.5) values were not. This intuitively makes sense: if the value is too low, there is not enough random movement to facilitate learning. If the value is too high, there are too many random failures still occurring at the end of the simulation to ensure a high chance of success on the final 10 trips.

Add Time Decay to Epsilon

One way to address the problem posed by high values of epsilon causing too many random movements towards the end of a simulation is to let epsilon decay with time. In the first few simulations, a high epsilon will promote lots of random movement and learning whereas in the last few simulations, a low epsilon will discourage random movement and the resulting occasional failures.

Four different types of epsilon decay were tested here, where 't' is the cumulative number of moves:

1. Epsilon = 0.9 / (1 + t / 10)
2. Epsilon = 0.02 / log (t + 2)
3. Epsilon = 0.02 for t < 1,500 and 0 for t ≥ 1,500
4. Epsilon = 0.2 for t < 500 and 0 for t ≥ 500

On a standard simulation with 100 trips, the value of t by the end of the simulation will run into the thousands, since each trip has a deadline of 20 – 50 moves, of which many are often required to complete the trip. The first type of epsilon decay tested is a constant decay. The second type is a logarithmic decay, with a sharper drop off occurring initially, followed by a leveling out. The third and fourth decay use a constant epsilon for an initial number of simulation points and then let epsilon go to zero, thereby turning off epsilon greedy learning in a step change manner. The results for these four types of epsilon decay are compared to the best constant epsilon values (from Table 4) in Table 5.

Table 5: Epsilon Time Decay (Alpha & Gamma fixed at 0.1; 100 Trips; 10 simulations each)

Epsilon Value	# Successful Trips	# Wrong Moves	Avg. fraction of deadline remaining	Cumulative Reward	Last Failed Trip	# Failed Trips on last 10 trips
0.01	97.2 ± 4.9	19 ± 3	0.571 ± 0.042	2,185 ± 90	23.2 ± 28.5	0.0 ± 0.0
0.02	92.1 ± 8.4	27 ± 7	0.545 ± 0.041	2,139 ± 180	35.5 ± 32.0	0.1 ± 0.3
0.05	97.5 ± 2.2	52 ± 10	0.566 ± 0.028	2,202 ± 66	39.7 ± 40.6	0.1 ± 0.3
0.9 / (1+t/10)	97.8 ± 1.9	38 ± 6	0.575 ± 0.026	2,214 ± 50	18.8 ± 28.4	0.0 ± 0.0
0.02 / log(t+2)	81.2 ± 20.2	16 ± 4	0.486 ± 0.111	1,898 ± 369	44.0 ± 37.1	0.6 ± 1.9
0.02 (t < 1,500)*	97.3 ± 3.4	28 ± 7	0.586 ± 0.034	2,186 ± 75	5.5 ± 7.0	0.0 ± 0.0
0.2 (t < 500)	98.2 ± 1.5	70 ± 9	0.571 ± 0.016	2,218 ± 32	12.0 ± 11.5	0.0 ± 0.0

*Statistically different from the epsilon = 0.01 (constant) case at the 0.05 significance level

Both the constant and decaying epsilon scenarios all gave very good results in terms of successful trips (>90%), with the exception of the log-decayed epsilon. Apparently the drop off in epsilon occurs too rapidly to derive the benefits of epsilon greedy learning. The number of failed trips in the last 10 trips was consistently low for all cases too, with a mean value less than 1.0 for all cases.

To differentiate between the different scenarios, the one-tailed T-Test was used to determine whether any of the results in Table 5 were statistically different from the base case (epsilon = 0.01), *based on the last failed trip*. All else equal, one would prefer a system that stops failing sooner rather than later. After applying this test, it was determined that the scenario where a constant epsilon of 0.02 was used for the first 1,500 moves followed by an epsilon of zero was statistically different from the base case. Since this case also gave excellent results on the other important parameters (successful trips, cumulative reward, failed trips in the last 10 trips) it was selected as the final model for epsilon, for the purposes of submitting the project for evaluation. Therefore, my final agent has the following key parameters:

- Alpha = 0.1
- Gamma = 0.1
- Epsilon = 0.02 for first 1,500 moves and 0 thereafter

As a final assessment of my agent, the simulation was run for 10,000 trials (instead of the standard 100) to let the Q-values get closer towards converging. The resulting Q-Table for this simulation is presented in Table 6:

Table 6: Q-Table Final Results

(Alpha & Gamma fixed at 0.1; epsilon = 0.02 for t < 1,500 and 0 for t ≥ 1,500; 10,000 Trips)

State	Action = None	Action = Right	Action = Left	Action = Forward
1: Red Light - Next waypoint NOT right	**0.11**	-0.14	-0.079	-0.45
2: Red Light - Next waypoint right - Clear on the left	0.021	**2.1**	0	-0.10
3: Red Light - Next waypoint right - NOT clear on the left - left car NOT going forward	0	**2.2**	0	0
4: Red Light - Next waypoint right - NOT clear on the left - left car is going forward	**0.21**	-0.079	0	-0.086
5: Green Light - Next waypoint right	0.022	**2.1**	0	0
6: Green Light - Next waypoint forward	0.021	-0.20	-0.028	**2.2**
7: Green Light - Next waypoint left - Clear oncoming	-0.00023	-0.12	**2.1**	-0.050
8: Green Light - Next waypoint left - NOT clear oncoming - oncoming car turning left	0	0	**0.21**	0
9: Green Light - Next waypoint left - NOT clear oncoming - oncoming car NOT turning left	**0.19**	-0.035	0	-0.049

The maximum value for any given state is highlighted in bold. If the highlighted action is the correct action (i.e. the highlighted action obeys the rules of the road for the given traffic scenario), it is colored blue. As the table shows, the agent correctly learned the rules of the road for each state, given that the maximum Q-value corresponded to the correct action.

I believe my agent comes close to finding an optimal policy. I would describe an optimal policy as follows: if the agent can proceed without breaking a traffic rule or causing an accident, then the next move should be equal to ‘next_waypoint’. If the agent cannot proceed without breaking a traffic rule or causing an accident, then the next move should be ‘none’. As shown in the final Q-Table for the simulation of 10,000 trips, the maximum Q-values for each state correspond to the correct (from a traffic law perspective) move for every state. I would find the results slightly more satisfying if there were a few less zero values, which suggest that even at 10,000 simulations, not all the possible state / action pairs are sampled. However, I am satisfied that the agent has learnt the correct action for any given state.