Jonathan Klinger

Abstract

In this assignment I will be using reinforcement learning to guide a race car through a set of tracks. Specifically, I will be showing the difference between how Q-Learning, SARSA and value iteration perform on this type of problem. Overall, value iteration performed the best, with a much lower average turns to finish than the other two. Q learning outperformed SARSA in every test. However, value iteration took a much longer time to run than the others. This was more prevalent on larger tracks, where value iteration often took twice as long to train than its competitors. Performance typically hit a peak at around 250K iterations for q learning and SARSA.

Introduction

Finding the quickest path down a track would be quite easy if the problem was deterministic, however, the problem becomes more difficult when we factor in the idea that planned acceleration for the next state might fail. For this reason, reinforcement learning would be extremely helpful. Value iteration identifies all the optimal moves given a current state, which makes it take a lot longer to train. Q learning and SARSA, on the other hand, learn the optimal moves given a specified amount of random states. While value iteration might be manageable for our problem, in real life there are much larger problems that might make using value iteration computationally intractable. Q learning and SARSA represent a more true form of machine learning where these algorithms do not simply memorize the entire space, but learn from taking different actions in different locations.

My hypotheses going into the project were as follows:
1. Making the problem non-deterministic (by adding a chance that acceleration in the next state could fail) will increase the number of crashes and time to reach the finish line when simulating the race.
2. While value iteration is much slower to train than its peers, it has the advantage of knowing all of the the transition probabilities and rewards. I would expect value iteration to outperform because of this.
3. I foresee issues with dealing with crashes. Due to to the max velocity of x and y ranging from [-5,5], it is possible that the car can move across the barriers and on to the other side of the track. It is also possible that where it crashes is closer to the other side of the track, and it will be revived on that side. (these are comically similar to some famous bugs in Mario Kart)

Implementation of Value Iteration:
Value iteration is the brute force approach to solving this kind of problem. The algorithm starts by initializing a rewards and q table with random uniform values.

The rewards table's dimensions can be found in Equation 1, where y and x represent the vertical and horizontal sizes of the track and v stands for the velocity range of x and/or y. The purpose of this table is to compute the reward for every state possible in the race.

$$Rtable_{size} = (y, x, v_y, v_x)$$

Equation 1: dimensions of rewards table

The q table's dimensions can be found in Equation 2, where y and x represent the vertical and horizontal sizes of the track, v stands for the velocity range of x and/or y, and a stands for the

* results for 750K episodes

number of different accelerations possible. The purpose of this table is to compute the reward for every possible acceleration.

$$Qtable_{size} = (y, x, v_y, v_x, a)$$
Equation 2: dimensions of q table

The algorithm sets all values in the Q table and rewards table to 0, where x and y coordinates correspond to the finish line on the track.

Next, the algorithm iterates through every possible state in given the track and: 1) sets all values for all locations in the rewards table to some large negative value when the location is out of bounds; 2) for each possible pair of accelarations: a) sets the reward to 0 if it has reached the finish line, else -1; b) computes the expected value of the acceleration given a 20% chance that it fails; c) computes the q value for this acceleration by summing the expected value and the reward; 3) sets the reward equal to the max q value of potential accelerations. This process is continued until convergence (changes become extremely small) or until the max number of iterations has been reached. It returns a policy, which identifies the most optimal move for a race car, given its current state.

Implementation of Q Learning:
The Q learning algorithm initializes the Q table (Equation 2) and initializes all values in finish-line states to 0. For a specified number of episodes, the algorithm will choose a random state in the graph an attempt to make movements by picking the most optimal acceleration or a random acceleration (depending on the outcome of the epsilon greedy strategy) for the given state. After the movement, the reward for the acceleration is updated using the Bellman equation (Equation 3), where LR represents the learning rate, Rt+1 represents reward for the move, γ represents the discount rate and max(Q(s',a')) is the optimal move for the next state. This process continues until a wall is hit, the finish line is reached, or the max number of iterations has been reached. This entire process of picking a random state and moving from there until a stopping criteria is met is repeated for the specified number of episodes.

$$Q(s,a)_{new} = (1 - LR) * Q(s,a) + LR(R_{t+1} + \gamma max(Q(s', a')))$$
Equation 3: q table update rule for q learning

Implementation of SARSA
The SARSA algorithm differs from the Q learning algorithm in the sense that it is an on-policy algorithm as opposed to and off-policy algorithm. This means that the existing q table is being used to update the q table. The SARSA algorithm initializes the Q table (Equation 2) and initializes all values in finish-line states to 0. For a specified number of episodes, the algorithm will choose a random state in the graph and the optimal or random acceleration (using epsilon greedy strategy). After this, for a specified number of iterations, the algorithm will make a move from the current state and compute the optimal or a random acceleration (using the epsilon greedy strategy) for the next state. After the movement, the reward for the acceleration is updated using the Bellman equation (Equation 4), where LR represents the learning rate, Rt+1 represents reward for the move, γ represents the discount rate and Q(s',a') is the optimal or random move for the next state. This determined acceleration for the next state is then set to the current acceleration, because for the next iteration, the current state is the previous next state. This process continues until a wall is hit, the finish line is reached, or the max number of iterations has been reached. This entire process of picking a random state and moving from there until a stopping criteria is met is repeated for the specified number of episodes.

$$Q(s,a)_{new} = (1 - LR) * Q(s,a) + LR(R_{t+1} + \gamma Q(s', a'))$$
Equation 4: q table update rule for SARSA

* results for 750K episodes

Experimental approach

Invalid Moves
As I hypothesized, when evaluating the performance of my models, I noticed that the race car was making some invalid moves. These invalid moves took on a few forms: 1) recovering from a crash on the opposite side of the track (Figure 1); 2) at high velocities, crossing over the walls to the opposite side of the track (Figure 2); and 3) jumping from the starting line to the finish line (Figure 3).

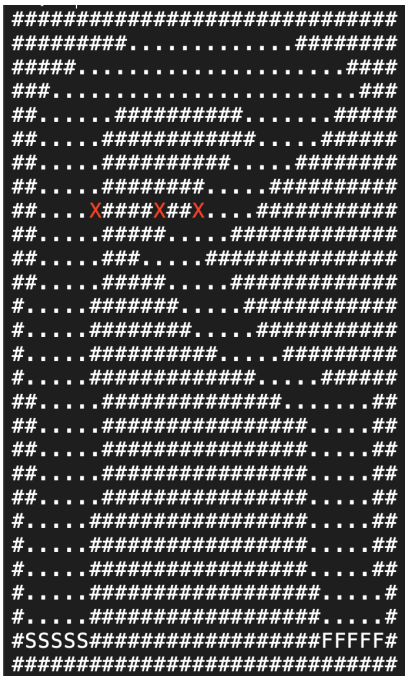

Figure 1

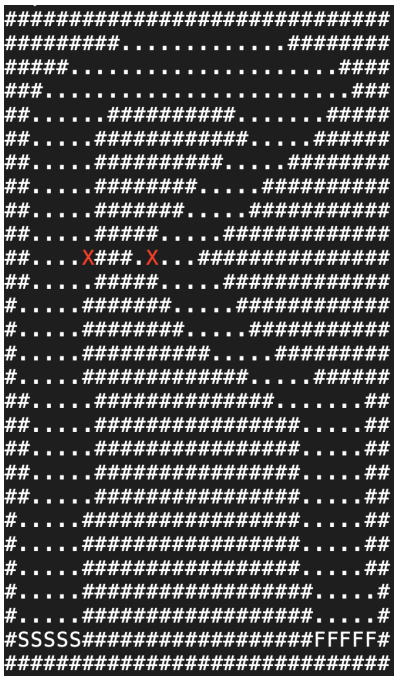

Figure 2

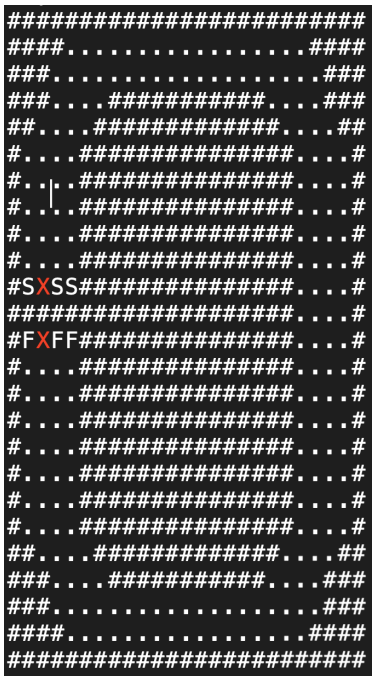

Figure 3

In order to resolve this, I needed to devise a method of identifying and handling invalid moves. When each move is made, the model checks to see if this move was to a valid location or invalid location. If the car is in a valid location, the model uses a recursive function that goes through every possible way the car could have gotten to this point to see if it is possible to get to it without hitting a wall. If this is not possible, the car has jumped over the wall and it resets to its last move. Otherwise it is a valid move and the model continues.
To handle potentially being revived on the opposite side of the track, the fix crash method is modified to only fix in the opposite direction of the last move. This means that a car is unable to revived on the opposite side of the track and skip part of the course.

Epsilon Greedy Strategy

For Q learning, I started out by using a purely exploitation strategy. When I moved to an epsilon greedy approach, I saw a slow in convergence but an increase in performance overall. I did some additional digging on this topic and found my final and best approach, the annealing epsilon greedy strategy. In Towards an Improved Strategy for Solving MultiArmed Bandit Problem, Akanmu mentions that “Epsilon Greedy performed better than Annealing-Epsilon when no of slot machines is 5 while Annealing-Epsilon performed better than Epsilon Greedy

* results for 750K episodes

when no of slot machines is 10. This supports the assertion that Annealing works better in a larger search space." This was exciting to me, because my search space is larger than 10.

$$E = 1/log(time + 0.0000001)$$

Equation 5: Annealing-Epsilon strategy formula

Results:

Performance is expressed as how many turns it took to reach the finish line of the R track over 100 test runs. The results are shown for both fixed crashes and restart crashes. Q learning was run over 750k episodes and 100 iterations for each episode. Value iteration was performed over 10 iterations (any longer than this would have required a significant amount of run time). Table 2 uses the fix crash method to show the results over all tracks.

Results on R Track

	Fix Crash	Restart Crash
Value Iteration	25.34	35.9
Q Learning*	31.82	50.84
SARSA*	36.25	58.37

Table 1: results on R track for all three learning approaches for both crash resolution approaches. Tested over 20 time trials.

Results Using Fixed Crash

	L Track	O Track	R Track
Value Iteration	12.41	25.34	26.2
Q Learning	31.89	14.56	32.28
SARSA	16.17	35.72	36.25

Table 2: results on all tracks all three learning approaches and fixed crash resolution

Value Iteration Performance Over Different Number of Episodes (Fixed Crash)

	Average Turns	Iterations
L Track	12.41	11
O Track	25.34	22
R Track	26.2	27

Table 3: results for value iteration over different numbers of iterations

Q Learning Performance Over Different Number of Episodes (Fixed Crash)

	100K	250K	500K	750K
L Track	15.72	14.56	15.66	14.9
O Track	36.53	31.89	34.27	31.99

* results for 750K episodes

	100K	250K	500K	750K
R Track	51.18	32.84	32.28	31.82

Table 4: results for q learning over different number of episodes

SARSA Performance Over Different Number of Episodes (Fixed Crash)

	100K	250K	500K	750K
L Track	18.03	16.17	17.25	17.32
O Track	40.42	35.72	36.83	35.95
R Track	51.78	37.94	37.79	36.25

Table 4: results for SARSA over different number of episodes

Behavior of Algorithms

Performance Peaks
I found that for q learning and SARSA, there was the biggest jump in performance between 100K and 250K episodes. This was true across all tracks. After 250K, results for q learning and SARSA would improve minutely or even start to perform worse. This highlights the importance of selecting an optimal number of episodes for these algorithms.

More Iterations = More Educated Movers
Something that I found common among all models (but especially so in Q Learning and SARSA) was that the more episodes/iterations that a model was exposed to when creating the q table, the more educated of a mover the car was when simulating the time trial. An example of this was that I found that with more episodes/iterations, the car would move slower around bends and faster when going straight for a while. This is because, when going around a bend, velocity must be changing at a faster rate to avoid crashing. When moving straight for an extended period of time, the car is able to maintain a high velocity in one direction for a period of time without the worry of crashing. This is because of the non-determinism in the problem, so the car is unsure whether an acceleration or deceleration will succeed. For this reason, I found that the better trained cars were able to determine those safer moments to maintain high speed over those less trained.

Fixed Crashes vs. Restart Crashes
When evaluating the difference between fixed crashes and restart crashes, I noticed a few things. First, the fixed crash resolution method showed a much faster performance time, turn count, and stall count. The reason for a faster performance time and turn count is intuitive, as the car does not have to restart from the beginning if it crashes. The lower stall count was interesting, and I assume this was because
One of my hypotheses going into the assignment was that applying fixed crashes will take a lot longer to train than applying restart crashes. This was certainly the case, as I was using a recursive method to identify the nearest space on the track to the crash site. For restart crashes

Stalls
While the performance peaked pretty early on in my testing for q learning and SARSA, the amount of stalling was consistently decreasing. I found that these stalls were less frequent when the number of episodes or iterations were increased, but I was interested in discovering why this was happening. In *End-to-End Deep Reinforcement Learning for Lane Keeping Assist,*

* results for 750K episodes

Sallab discusses the idea of stuck termination: “Stuck Termination condition can help both the basic experiment and out of track experiment, so it is most commonly used with the basic experiment in order to avoid settling in a local minimum either the car found between the track boundaries or the car settles out of the track.” It would make sense that I was finding this issue with q learning and SARSA but not with value iteration, because value iteration is reviewing all of the possible states for a number of iterations (until the change in values becomes very small) and thus has a lower chance of hitting a local minimum.

Conclusion

After all my testing, I have found that q learning seems to be the better of the actual learning algorithms (q learning and SARSA). Q learning outperformed SARSA in every test. I also found that value iteration is the best choice when we are only interested in results and not computation time. This is because value iteration isn’t actually doing ‘learning’, because it is iterating through every possible state in the track several times in order to derive the optimal strategy. Value iteration would be my choice if I was not concerned with time or I was working in a small space. However, if my problem space was large (like R or O tracks) and I did not want to wait a long time to get the most optimal results, q learning would be my go-to choice. Q learning is a strong and versatile approach to solving these problems when there are constraints.

Citations

Sallab, Ahmad El et al. “End-to-End Deep Reinforcement Learning for Lane Keeping Assist.” ArXiv abs/1612.04340 (2016): n. pag.

“Towards an Improved Strategy for Solving Multi-Armed Bandit Problem.” International Journal of Innovative Technology and Exploring Engineering Regular Issue, vol. 8, no. 12, Oct. 2019, pp. 5060–5064., doi:10.35940/ijitee.l2522.1081219.

* results for 750K episodes