Journal of Machine Learning Research 1 (2021) 1-48 Submitted 6/13; Published 0/0

EN.605.649 (81): Machine Learning: Analyzing Efficiency of Various Reinforcement Learning Algorithms on Racetrack Simulations

Kayla Ippongi KIPPONGI@JHU.EDU
Johns Hopkins University

Editor:

Abstract

This assignment outlines the steps taken to build several different reinforcement learning algorithms to train and build a policy that will allow a racecar to travel across a track in a minimal amount of steps.

1. Introduction

We are presented with 3 different racetracks, an L track, an O track and an R track for our racecar to travel in. In this programming assignment, we build and test several factors to drive an optimal race. Firstly we build an environment for the car to travel in, this involves detecting if the car has crossed the finish line or not, or has crashed into a wall. To determine whether a car has crashed into a wall, we use Bresenham's line algorithm to track our movement from point a to b and check if any of those points contains a wall. To determine the closest on-track point to a crash site, we start from the crash site and use breadth-first search via a queue to determine the closest on-track coordinate to the crash site.

In our racecar environment, the states are made up of all possible movements of the car along with its velocities. Thus a state in our racecar environment is a tuple of (x,y,vx,vy). The actions are all possible actions that our racecar can take. Since we are only allowed to control the acceleration of the car, all possible actions include permutations (ax,ay) between the ranges of -1 and 1. Additionally, we introduce some non-determinism by only allowing the car to accelerate 80% of the time. We also introduce exploration for our agent by having the agent choose the maximal action 80% of the time, versus choosing a random action the other 20%

We use 2 differing crash policies, one that resets the car to the nearest track cell with a velocity of zero and another that resets the car back to the starting line. Once we have our track environment set up, we also build several reinforcement learning algorithms. This involves 2 on policy algorithms - Value Iteration and SARSA, as well an off-policy algorithm - Q-learning. Given these factors we develop these hypotheses:

1. Crash version 2 - resetting car to starting line will result in slower race times as this version is much harsher than the first one and will require more steps along the track.

2. The on-policy algorithms, such as value iteration or SARSA will result in better race times as on policy algorithms use the latest learned policy and can exploit the rewards.

1.1 Q-learning

1.1.1 Approach & Assumptions

Under the Q-learning algorithm, we build a Q-table that holds the corresponding values for taking that state-action pair. Thus, when presented with a state, we can consult the Q table to determine which action gives us the maximal reward for that state.

Initially, the Q-table is filled randomly with values between 0 and 1. We follow the given equation to update our Q table accordingly at each iteration, where r is the reward, α is the learning rate, and γ is the discount rate.

$$Q[state][action] = Q[state][action] + \alpha(r + \gamma(maxQ(newState, action)) - Q[state][action])$$

We run several experiments, each with a differing number of iterations to understand how the number of iterations affects our overall time to cross the finish line. Once we have completed the number of iterations, we are left with our filled-out Q-table. We then build our optimal policy by consulting the Q table and associating a state with the action that maximizes our reward.

1.1.2 Analyzation

In figure 1 we see the results of the Q-algorithm on the 3 different racetracks. The learning rate and discount rate were fixed for these experiments, with the learning rate set at 0.3 and the discount rate at 0.8. Future experiments should also tailor these parameters to see how they affect our learning curve.

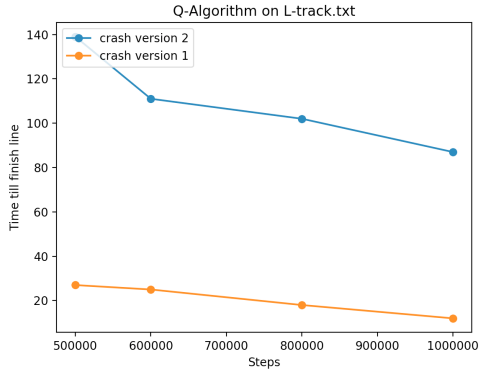
Crash version 1 allows the car to be set back to the closest on-track point, therefore we see that this version is consistently faster at crossing the finish line compared to its much harsher version 2 which puts the car back at the starting line. Thus, we see that our original hypothesis holds to be true in this round of experiments, intuitively this makes sense as setting the car back to the starting position increases the number of steps. Additionally, the Q-algorithm takes many iterations to train in comparison to other algorithms such as value iteration.

1.2 SARSA

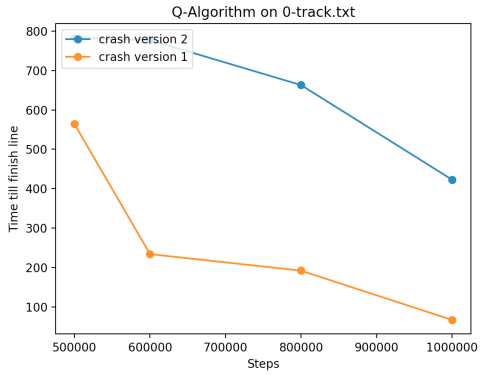
1.2.1 Approach & Assumptions

The SARSA algorithm is fairly close to the Q-algorithm, and the main difference lies in how the Q table is updated after an action is performed. With SARSA, we learn the action values relative to the policy it is following, while the Q algorithm follows the greedy policy approach.

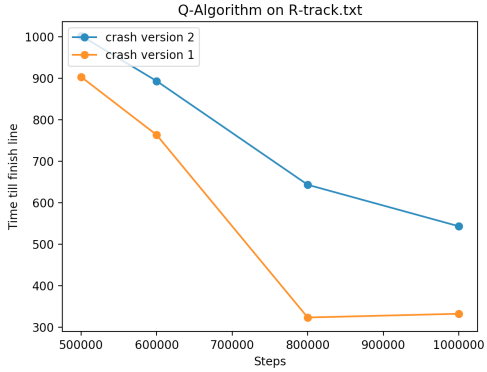
As in the Q-algorithm, the Q-table is filled randomly with values between 0 and 1. We follow the given equation to update our Q table accordingly at each iteration, where r is the reward, α is the learning rate, and γ is the discount rate.



(a) L-track



(b) O-track



(c) R-track

Figure 1: Q-algorithm using LR = 0.3 and discountRate = 0.8

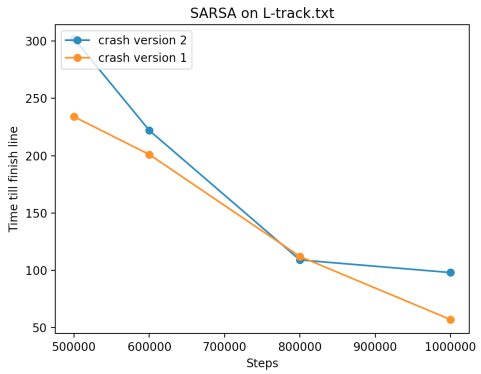
$$Q[state][action] = Q[state][action] + \alpha(r + \gamma(maxQ(newState, newAction)) - Q[state][action])$$

As done in the experiments for the Q-algorithm, we run several with differing number of iterations to see how the learning rate changes as the steps increase.

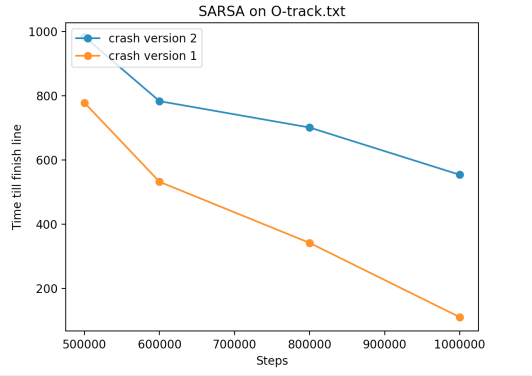
1.2.2 Analyzation

Looking at Figure 2, we see the results for the SARSA algorithm on the 3 racetracks, with the learning and discount rate also kept the same in the previous experiments.

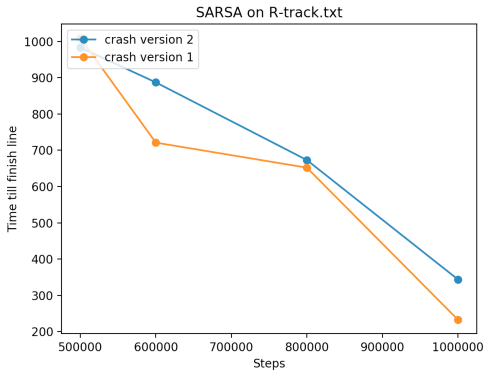
Again we see that our original hypothesis of the crash version 1 being outperforming version 2 still holds in all 3 tracks. In comparison to the Q-algorithm, we see the SARSA algorithm has steeper slopes. This seems to imply that the SARSA algorithm seems to learn or exploit rewards faster than the Q-algorithm does, which can result in our car being able to get to the finish line faster.



(a) L-track



(b) O-track



(c) R-track

Figure 2: SARSA using LR = 0.3 and discountRate = 0.8

1.3 Value Iteration

1.3.1 Approach & Assumptions

Under the value iteration algorithm, we build both a value table and a Q table, where the value table holds the optimal Q(state,action) value for each state-action pair. We iterate through all possible state-action pairs, calculate their value and update the value table. With value iteration, the agent knows both the probabilities of landing in a certain action given a state as well as the reward it can get for taking a specific action. These factors are why we hypothesized for value iteration to outperform the other algorithms.

Calculating transition function, where 0.8 represents the probability of accelerating and 0.2 represents the probability of staying

$$expectedValue = (0.8 * valueNewState) + (0.2 * (valueNewStateFailureToMove))$$

Updating Q table

$$Qtable[state][action] = reward + (discountRate * expectedValue)$$

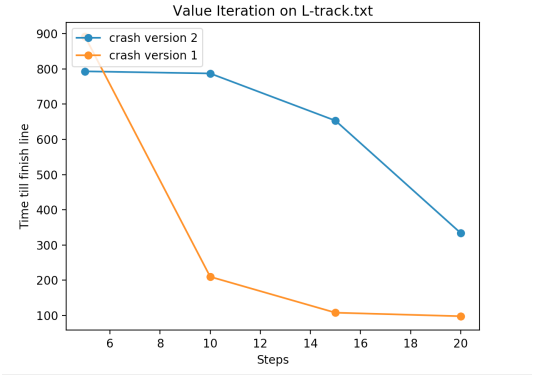
Updating the value table

$$actionWithHighestQ = argmax(Qtable[state])$$
$$Vtable[state] = Qtable[state][actionWithHighestQ]$$

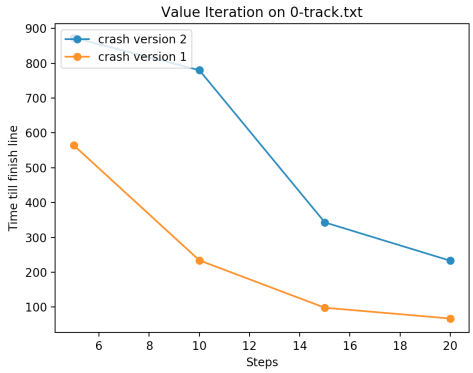
1.3.2 Analyzation

In figure 3 we see the results of the value iteration algorithm on the 3 different racetracks. Like previous experiments, the learning rate and discount rate were fixed for these experiments, with the learning rate set at 0.3 and the discount rate at 0.8. Future experiments should also tailor these parameters to see how they affect our learning curve.

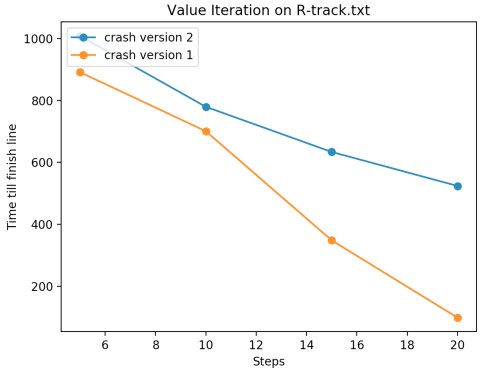
Under value iteration, we see that the times to cross the finish line start off slightly higher than the results from the Q-algorithm, however as the steps increase we see that the time converges faster as well. Looking at the steps to the far-right we see that the time ranges from around 70-200 for all 3 tracks. For more complex race track environments like the R-track, the value iteration seemingly performs best, with a time under 100, in comparison to 220 and 330 for the Q-algorithm and SARSA.



(a) L-track



(b) O-track



(c) R-track

Figure 3: Value Iteration using LR = 0.3 and discountRate = 0.8

2. Conclusion

In this programming assignment, we have built 3 different reinforcement algorithms to run against 3 different racetracks to see their differing learning curves that result from their returned policies. There are several outcomes that we can determine from these runs. Firstly is that reinforcement learning works and is really powerful. It was able to take an unknown environment and successfully learn from its action to reach a specific goal. With that, it is also pretty costly in terms of computing time and can take up a lot of time to train. SARSA and the Q-algorithm took several hours to train for each of the tracks, while value iteration took around 30 minutes. Looking back to our original hypothesis, these experiments validate both of them. We see that the crash version 2 performed a lot more poorly in every experiment and every track. Since that was a harsher crash policy, this was expected behavior. Lastly, the on-policy algorithms - value iteration and SARSA gave us better results overall. For SARSA, the best times for each track ranged from 50-220 and 50-100 for value iteration. Additionally, value iteration trained the fastest out of the three, making it an ideal algorithm for us to use in these types of experiments. Future

improvements or future factors to test, including using a different approach other than BFS to calculate nearest on track point, changing the learning rate and discount rate, and initializing the Q table to zeros rather than randomized. Further understanding how these factors influence our learning curve can help improve the agent reach the finish line faster.