

Training Reinforcement Learning Models to Navigate a Racetrack

Affan Dhankwala

Abstract

Reinforcement learning is a concept which allows a model the capability to understand, learn, and eventually master an environment with little to no prior knowledge. Over the course of this paper, we were tasked to navigate a racecar through a set of tracks starting from the starting line to the finish line while avoiding walls. If the racecar collides with the wall, it will either be reset to the nearest position on the track or at the starting line. At any given point, the agent is only in control of the racecar's acceleration—which is limited to either -1, 0, or 1 in both the x and y direction. With a maximum speed of (5, 5), the agent learns to apply acceleration vectors to both speed up and slow down the racecar. The racecar we designed is capable of being learned to each track via three different models: value iteration, Q learning without SARSA and Q learning with SARSA. All three models excel in different categories and no objective best performance can be set aside. SARSA, however, had a much worse performance than expected and that could be due to a faulty implementation design.

1 Problem Statement

Throughout the course of this project, we have been tasked to design, train, and test three different reinforcement learning strategies upon a race model. The model initializes a track which consists of a starting line, a finish line, walls and road. We are then tasked to initialize a racecar at the starting line and it should reach the finish line by traversing the road and avoiding the walls. The penalty for hitting a wall can be one of two consequences. The racecar could either be placed back onto the track closest to the wall it hit or it could be reset back at the starting line. In both scenarios, the speed of the racecar is set to (0, 0). The agent is responsible for navigating the racecar through the track but is only given access to altering the acceleration of the racecar with the limitation of the acceleration having a magnitude of either 0 or 1 in the positive or negative direction for both x and y coordinates. The velocity of the racecar is capped at (5, 5) in both the positive and negative directions. Therefore, it is up to the agent to learn when to speed up and slow down the racecar.

The three models that will be explored are the value iteration, q learning and SARSA. Assuming that accuracy is defined as minimizing the combination of walls hit by the racecar, moves expended during the race, and race time, we believe that the value iteration algorithm will have the highest performance since it has the most knowledge of the racetrack. By setting up the Q table with respect to all possible states from every state and converging when a certain threshold is reached, the value iteration trained model will have all the information necessary to chart the optimal path through a track. SARSA is expected to have the second highest performance because this algorithm forces the racecar to select a state based on if an action in the next state maximizes the q value of the model. By looking one step ahead, we can provide our model with more information and therefore allow it to learn the track more accurately. Finally, we believe that Q learning will have the worst performance as we will be encouraging the model to adopt a greedy strategy and traverse to the state with the best current reward. Of course, there are some discounting factors to prioritize future rewards but this model is expected to perform worse than our SARSA.

2 Experimental Approach

To efficiently conduct the creation of this model, we utilized an object-oriented base approach where we divided critical components into individual classes with member variables and methods that could be called from one another to improve code readability and minimize redundancy.

Object Oriented Design

As previously stated, we heavily relied on object-oriented principles to design each component of this model to efficiently demonstrate the model's capabilities. The classes that we created were track, racecar, value-iteration, and learning-model. Below, we explain each one more in depth.

Track is a class which is meant to store the physical track. When initialized, it translates the text file provided track into a 2D array maintaining all ordering and symbols. This allows us to quantify each position on the track and easily access its value. Although not stored in any member variables, the track class can identify and return all starting positions of the track. If given a line-vector (a list of 2D coordinates corresponding to positions on the track), the track class has member variables to determine if that line-vector crosses the finish lines or hits a wall and returns appropriate values in each circumstance. These are especially useful methods that are heavily utilized by the following racecar class.

The racecar class is responsible for all the variables of our model's racecar. Not only do we track the racecar's position, velocity, and acceleration, we also track the path it takes and other metrics for telemetry. The racecar is instantiated with a track object member variable which serves as the track that the racecar is assigned to. We cannot assign the racecar to another track after initialization—as there is no need to. By default, once initialized, the racecar's position is set to a random position on the start line with velocity and acceleration set to 0. This is also the same state that the racecar is reset to if we collide with a wall and is required to restart. The racecar can perform the standard legal acceleration applications with all probability and restraints factored in. There are also testing member methods that bypass the legality but these methods are merely for training models rather than traversing the track. These testing methods allow the racecar to not just be the validation tool of the model, but also be a training tool to test if certain training policies are efficient and valid. When testing, the racecar applies an acceleration coordinate at each move. There is 20% fixed chance that the applied acceleration is never implemented and the racer continues moving at the previously noted acceleration. This is known as an acceleration failure and is implemented per project requirements. The methods to test a model's performance call methods that factor this acceleration failure proportion. Methods that assist a model to train bypass this acceleration failure proportion to remove any uncertainty when training.

The value iteration class is, as expected, critically utilized for the training and testing of the value iteration based reinforcement learned model. This class instantiates an instance of the value iteration model by setting up a q table represented via a dictionary. This q table corresponds with the track 2D array and holds the reward for each position. -1000 for walls, 100 for finish line, and -1 for every other point. We can train and test a racecar on a track via this model. We also decrease the reward at each position each time the position is visited by the racer. This is to discourage the racer to visit previously visited positions.

Similar to the value iteration class, we created a learning model class which holds the algorithm for q learning and SARSA. This class also initializes a Q table corresponding to the track with all respective reward. However, during the training process, we train the Q table with a training-based racecar that

bypasses the acceleration failure proportion. This means that each state is updated on when the racecar traverses it. We recall no knowledge of where the finish line and only calculate states' values based on the equation with respect to their neighbor states.

Training

The training differed for all three models as they were all tasked to utilize different algorithms and still manage to create a q table to allow the racecar to traverse the track.

The value iteration model is trained by first initializing the previous stated q table to match its track member variable and hold all rewards at each position. A 'state' is defined as not only a position on the track but also a certain velocity. This means that [4, 5] velocity at [2, 3] is a different state from [4, 4] at [2, 3]. An action at any state is defined by the acceleration that we intend to apply at that state. If the racecar is at [2, 3] and traveling at [4, 5] velocity, applying an acceleration of [-1, 0] is an action. The training of this model requires us to calculate the value of each action at each state. This results in the following number of calculations:

$$\begin{aligned} \text{Number of positions} \times \text{Velocity Combinations} \times \text{Acceleration Combinations} &= N \times 121 \times 9 \\ &= 1089N \end{aligned}$$

This means that this model performs 1,089 calculations for each road position on the track (We don't run the calculations on walls or finish line points because their values are fixated). Each individual calculation for each action per state is the value iteration Equation:

$$V(s, a) = R(s) + \gamma V(s, a, s')$$

$V(s, a)$ corresponds to the value of action a at state s . $R(s)$ corresponds to the reward of the current state. γ is the discount factor and finally $V(s, a, s')$ is the value of the next state if we were to apply a at s . Keep in mind that value and reward are meant to stand for different variables. Reward indicates the base reward for each position and it is a static value minus however many times the racecar visited it. Value represents the calculated benefit of each position and it is stored within the `value_dict`. Once we calculate the value for each action at a particular state, we select the action with the best reward and assign our current state to this value. We update the `value_dict` with this information and compare this value with the previous value (or base reward if we have no previous value). This comparison is known as the delta value, and it is to determine how much the values of each state change within each episode. An episode is complete when we iterate through all states for all road positions of the track. We terminate training once our maximum change (delta) value within an episode becomes smaller than a certain hypertuned threshold. This is when we return the `value_dict` for testing purposes and thus conclude the value iteration training process.

The q learning model and SARSA are both trained in similar manners. First, we instantiate the q table dictionary which holds the base rewards for all positions on the track. Now, instead of working through all possible states at each position, we initialize a racecar object that will traverse our track in a deterministic fashion (we eliminate the possibility of a failed acceleration). After initializing the racecar, we gather its position and velocity and determine whether we plan on exploring or exploiting. Exploring implies that the racecar shall set its acceleration randomly for the purpose of 'exploring' the track. Exploitation implies our racecar shall set its acceleration to the predetermined optimum value—to reach

the next best determined state. Initially our map is completely unexplored, so we set our exploration rate to 100%.

Now, upon selection of exploitation, we consider all acceleration actions and save the acceleration that results in the best reward. Reward implying the base static reward of the position minus visitation count. We also prohibit the static scenario that is generated if our velocity is $(0, 0)$ and our best acceleration is $(0, 0)$.

After determining the next acceleration and its respective base reward (whether that be through exploration or exploitation means), we apply this acceleration to our racecar and log it's finishing metrics. The role of the racecar is now complete in this iteration, but we need to update our table. We need to update the value based on whether SARSA is toggled. If it is off, our q update equation is as follows:

$$Q(s) = Q(s) + \alpha(R(s') + \gamma Q_{\text{MAX}}(s, a, s') - Q(s))$$

If SARSA is enabled, our q update equation is as follows:

$$Q(s) = Q(s) + \alpha(R(s') + \gamma Q(s, a, s', a') - Q(s))$$

$Q(s)$ is the q value at a particular state. α is the learning rate of our model. $R(s')$ is the reward at the next state. γ is again the discount rate. $Q_{\text{MAX}}(s, a, s')$ is the maximum q-value of all successive states. $Q(s, a, s', a')$ is the q-value of the selected successive state. As seen in the equations, the SARSA toggles whether we operate under the assumption of idealistic actions every time or whether we operate with respect to more knowledge of the next step. The reason SARSA does not account for the q-value in the next step at 100% efficiency is due to the explore/exploit ratio. If SARSA calculated that we will explore next round, then we shall select a random position—which is not guaranteed to be the position that SARSA had calculated.

Once we have updated the q-value at the current location, we have now finished this traversal and update the biggest delta value with our state's change in q-value. We keep traversing the track until our racecar reaches the finish line. At this point, we have completed one episode. We keep conducting episodes until our biggest delta is less than a certain threshold or we reach a certain number of maximum episodes. Once the episode terminating condition is executed, our training process concludes and the `value_dict` q table is ready for testing.

Testing

Once we finish training our models, we must test how well they perform. A testing method is engrained within each learning model's class and is called from the test file. When testing, we run through several experiments. An experiment begins with a racecar being initialized at the starting point and concludes once that same racecar passes the finish line. Once each experiment is finished, we gather the run metrics and restart the racecar for the next experiment. This creates a set of metrics for all our experiments that is saved for analysis.

When testing value iteration, we traverse a racecar through the track by retrieving the best acceleration at its current state from the `value_dict`. By factoring in the acceleration failure probability, we are guaranteed some level of randomness that is meant to break possible areas of non-convergence.

When testing the q learning algorithm, SARSA no longer plays a factor as we retrieve all instructions from the value_dict. However, due to the chance that we stumble upon a new state, we must allow the racer to ‘explore’ if there is no optimal policy. There is also a small exploration percentage that is applied within the test and that is to offset any nonconvergence areas. Similar to the value iteration testing sequence, our racecar retrieves the best acceleration at each state from the value_dict and stores each experiments metrics for post-processing.

For ease of visualization, we plot the walls hit, moves expended, and race times of all experiments and plotted them. We also draw out the track and visually represent the location of the racecar. This was for debugging purposes but is still included within the project. The terminal-based track is created within the ‘print_track.py’ file and the sleep time between each track representation can be toggled.

Tuning

Over the course of this model, there were a handful of hyperparameters that were manually tuned for satisfactory results.

The learning rate is set to 0.05 as this was an optimal and somewhat slow learning rate to guarantee convergence but also avoid taking unnecessary time. The discount rate is set to 0.9 to prioritize future rewards. The episode count for Q learning is set to 15,000 cap as this was enough episodes to result in little non-convergence areas but not too long to be tedious to run or run the risk of overfitting. The exploration rate is initially set to 1 to encourage 100% exploration. This value’s decay is set to 0.01% so that we could prioritize exploration heavily within the earlier cycles and, since we were aware of how many episodes we planned to traverse, eventually prioritize exploitation later. The biggest delta threshold is set to 5 as we determined that once we are altering the chance by less than 5, we had a fully functional value_dict Q table.

Presentation of Results

Below are the results from all reinforcement model types for each track. We also made a legend just to shorten the name of each legend.

Legend	
I	Value Iteration without Restart
II	Value Iteration with Restart
III	Q Learning without Restart
IV	Q Learning with Restart
V	SARSA without Restart
VI	SARSA with Restart

L-TRACK						
Metric	I	II	III	IV	V	VI
Training Episodes	24	24	6758	5428	15000	15000
Training Time (s)	78.099	77.654	53.314	42.158	249.107	245.589
Avg Walls hit	0.300	0.600	3.300	4.400	3.300	21.900
Avg Moves	12.200	15.800	34.800	45.100	47.900	224.300
Avg Experiment Time (ms)	2.583	3.123	6.642	7.586	7.616	36.332

O-TRACK						
Metric	I	II	III	IV	V	VI
Training Episodes	24	24	339	619	6724	4507
Training Time (s)	139.768	139.783	1.432	2.342	60.036	36.406
Avg Walls hit	0.100	1.000	0.000	0.000	8.500	2.100
Avg Moves	4.600	7.900	4.400	4.500	79.600	33.100
Avg Experiment Time (ms)	1.311	2.237	1.226	1.275	22.125	9.540

R-TRACK						
Metric	I	II	III	IV	V	VI
Training Episodes	37	37	12682	11322	15000	DNF
Training Time (s)	346.236	346.179	675.302	592.338	1660.054	DNF
Avg Walls hit	2.100	11.000	10.400	17652.500	33.800	DNF
Avg Moves	30.000	139.800	103.200	124889.100	433.000	DNF
Avg Experiment Time (ms)	6.904	33.512	23.853	30613.429	106.984	DNF

W-TRACK						
Metric	I	II	III	IV	V	VI
Training Episodes	24	24	5841	3782	15000	15000
Training Time (s)	143.817	145.168	62.507	32.864	396.103	386.559
Avg Walls hit	0.400	0.600	13.900	1.300	6.700	2.500
Avg Moves	11.400	12.100	122.700	22.800	99.700	46.700
Avg Experiment Time (ms)	2.832	3.046	30.750	5.760	20.994	11.538

Discussion and Conclusion

From these experiment values, we can bring many observations to light. Beginning with simple run time metrics on average, value iteration results in the fastest race times during testing. This is demonstrated on the W, Q, and R tracks. In second, q learning has a variable runtime. Its performance can sometimes be faster than that of value iteration, as demonstrated in the experiment time on the O-track, but it sometimes skyrockets to high values such as those on the R track. Finally, SARSA is the slowest model during

testing. Now for training, value iteration model returns a consistent number of episodes and respective training time. The other two models' training episodes vary with a large range.

The O-track served to be a unique example where our q learning agent performed slightly better than our value iteration agent. From a purely experimental outlook, the 90-microsecond time difference could be attributed to a random luck event. However, it is critical to consider that the training for the Q Learning agent was complete within 1.5 seconds while the value iteration had taken more than 2 minutes. If we consider cumulative time, the q learning was much quicker during the training and testing on this track. This is the only track where we see this performance. On other tracks, the q learner sometimes trains quicker than the value iteration model but its testing performance falters.

A unique concept that we had never considered but can clearly see in our metrics is the concept of ‘restart-sensitive’ tracks—tracks that are friendly or unfriendly for restarts. When testing, if a racecar is forced to restart the track, it will still operate on the same Q table. This means it will, under ideal circumstances, repeat its past action exactly. Now, with the acceleration failure concept, that is not likely. What this creates is the opportunity for a racecar to ‘redo’ the track. Of course, the timer is still running but it saves the racecar from any tight unnecessary areas that it could have possibly been in—such as those on the W track. With that final note, it is worth noting that both the q learning and SARSA models performed much better with the restart flag enabled than their restart-disabled counterparts. The value iteration model performed slightly worse but that can be attributed to some random error. This means that this track is ‘friendly’ to restarts. We believe this is due to the previous concept of rescuing the racecar when it enters those nooks in the wall. On the contrary, the R-track was a restart unfriendly track. With its narrow roads and hairpin turns, our racecar must carefully navigate its premises. Any slippage and the racecar is back at the starting line to redo its attempt. On this track, all models employing restarting enabled tests performed much worse than their restart disabled counterparts. SARSA with restart enabled never made it past the second turn and it ran for long enough to crash the IDE multiple times. We were unsure how this was even possible so, due to time restraint, we left the metrics undefined as DNF.

From these observations, we can go back to our original hypothesis stating that value iteration would be the best performing model followed by q learning and finally SARSA. Best performance comes from a factor of elements. On most tracks, value iteration tests finished faster, bumped into less walls, and expended less moves than the other models. However, q learner was usually the quickest to train on all the tracks. The restart enabled models’ performance versus their restart disabled counterparts was also variable among the tracks. Therefore, it is conclusive to state that there is no ‘optimal’ model. Each model has its strengths.

One issue that is still perplexing us is the relatively low performance of the SARSA model. With training time lacking and testing performance being usually the worst among the three models, this is in complete contrast with our original hypothesis. We debugged the system and didn’t see any glaring errors in the algorithm supporting the SARSA model. However, when the model plans for the next move, it determines if we explore or exploit. Now, explore means selecting a random acceleration value but it does not mean that this is the exact acceleration value that our next iteration will be utilizing. This means that our SARSA model was considering the value of employing actions that we would not run. We were not sure if that was the goal of SARSA or whether SARSA is required to go forth with the action that it had examined. We believe that if we had pursued the later assumption, our SARSA model would be accurately predicting the value of its next state and boosting its performance.