Journal of Machine Learning Research 1 (2020) 1-6 Submitted 11/13; Published 11/15

Reinforcement Learning Algorithms

Brian Loughran BLOUGHRAN618@GMAIL.COM
Statistical Research Department
Loughran Institute
Breckenridge, CO 80424, USA

Editor: Brian Loughran

Abstract

This document describes a series of reinforcement learning algorithms including value-iteration, sarsa and value-iteration as methods of learning generalized tasks. The algorithms described are implemented to complete a racecar time trial, however the algorithms are generalized in a way that can be applied to a variety of tasks. Included in this paper is a problem statement summarizing assumptions made for the algorithms and projections on how the algorithms are expected to perform against a the different tracks. Also discussed is a description of the experimental approach and assumptions made for the reinforcement learning algorithms. Results for each algorithm and track are discussed and finally conclusions are drawn on algorithm learning.

1 Problem statement

Reinforcement algorithms are generally considered to be something in between supervised and unsupervised learning algorithms. Each of the algorithms discussed differ slightly in structure between the value-iteration, q-learning and sarsa algorithms, however each is based around the concept of rewards. The learner is given a reward for each action that results in a positive result (in this case finishing) and a penalty for actions that do not result in a positive result (in this case not finishing). The learner is then incentivized to maximize its reward, which in turn should result in the task at hand being completed in an efficient manner. While each algorithm varies slightly in its approach and underlying assumptions, the general idea prevails throughout all three algorithms.

3 data sets are used to evaluate the learning algorithms. Each data set represents a different track, l track, o track, and r track. Each track is shaped like the capital letter corresponding to the track name, showcasing the flexibility of each of the learning algorithms. Further, crashes can be handled differently for each of the algorithms, with one version of crash behavior representing bumper cars, where the operator is stopped at the wall and its speed is zeroed out, while another version of crash behavior returns the operator to the start of the track. While penalties are not given out explicitly for crashes, this flexible learning model is able to learn and account for the different crash behaviors without being explicitly told the relative benefit of walls between the two crash behaviors

We can look at a few different metrics in order to evaluate each of the learning algorithms. The first and most obvious metric we can look at is the number of moves each of the algorithm takes to complete the track after learning for each of the algorithms. Another metric that we can look at is the learning rate for each of the algorithms, that is how quickly the algorithm is able to reach an asymptote where the learning is unable to progress any further. This learning can be benchmarked against both the number of episode needed to complete learning as well as either algorithmic complexity or wall clock time to reach the conclusion (with the caveat that wall clock time is a bad metric subject to forces outside the program execution). In this paper

we will examine only the number of episodes needed to complete learning, along with a brief discussion on the relative runtimes of each of the learning algorithms.

Based on the success of reinforcement learning algorithms on a variety of real-world tasks, it is expected that each of the learning algorithms proposed in this paper will be able to learn a strategy within 20% of optimum behavior (with "optimum behavior" being the best that a human operator can reasonably achieve). This would be a considerable success considering the relatively small amount of information and processing time/power the learner is given.

2 Experimental Approach

The reinforcement learning algorithms discussed in this paper are well known algorithm, thus the algorithms will not be discussed in detail in this section. Some assumptions are made to streamline the algorithms and those assumptions are discussed in this section, including track and operator behavior, handling rewards, and justification for assigning hyperparameter values.

With the operator moving in both the x and y direction, an algorithm must be chosen to determine the x and y coordinates occupied by the movement of the operator. The Bresenham algorithm seems appropriate in this case as it is an algorithm used to determine the pixels occupied by line in within a two-dimensional space. Assuming linear motion of the operator for each of the turns taken, this would be a good approximation of the spaces occupied by the operator. Using the Bresenham algorithm to determine the spaces occupied by the operator in this sense allows us to determine where and when the operator collides with a wall or crosses the finish line and apply logic accordingly.

For all of the tracks supplied, there are multiple possible starting and finish locations. This would seem to approximate a real race track, where there are not true starting and finishing points, rather starting and finish lines. Nonetheless, the algorithm must account for the multiple possible starting and finishing positions. The starting positions are handled randomly, that is the operator is started at a random position along the array of starting positions. Each of the starting positions and finishing positions are stored as the track is loaded into the track object, thus accessing the positions is trivial after the track is read. Finish locations are dealt with similar to wall collisions, in that once a finish location is detected movement is stopped and the finishing logic is applied immediately once the operator collides with the finish line. In this way, the operator may pass through the finish line with momentum and the finishing logic will apply just as if the operator had crawled its way across the finish line.

As discussed, using the Bresenham algorithm will allow logic for each of the different types of spaces the operator can occupy. Logic can be applied for each of the different types of spaces occupied during a move by the operator. For example, empty space is represented by '.'. Movement will not be stopped for an operator passing through empty space, nor will it be stopped for the operator passing over the starting line (represented by 'S'). However, movement will be stopped if the operator tries to pass through a wall (represented by '#'), or if the operator passes though a finish state (represented by 'F'). If the operator tries to pass through a wall, movement will be stopped and the corresponding collision logic will be applied. If the operator tries to pass through the finish line, movement will be stopped and the operator will have completed the course.

Not every point on the board is able to be occupied by the operator (specifically walls are not able to be occupied). Because of this, we can prune the number of valid states and decrease the runtime of the learning algorithms. Each of the algorithms tracks the valid states that the operator can occupy, where state is the position (x and y position) and speed (x and y speed). Any x and y position that is a wall is invalid, thus these positions does not have to be included in the valid states for the given track.

The above process for tuning the valid states can be extended. For a given position, there are some speeds that are inaccessible for the given position in areas where the board is relatively

tight. It would also be possible to tune these states out of the states array in order to make the algorithms run even faster. While this is possible, it was not done in the implementations of value-iteration, q-learning and sarsa discussed in this report. Because of this, the rewards at these states will approach 0 in the value iteration algorithm, and will remain at 0 for the q-learning and sarsa implementations. Near-0 values for each of the algorithms would indicate that the learner did not explore these states due to them being unreachable. While some unreachable states are expected for each of the algorithms, having an unreasonable number of unreachable states will result in longer than needed learning times.

It was recommended in the problem statement to set the reward for the learner to 0 for the finish states. Through testing, it was found that the learner performed better and learned faster when the reward for finishing was set to a moderate positive number rather than the neutral 0. Manual testing showed that a reward value of 100 worked nicely to balance the benefit of giving a finishing reward and keeping the reward within reasonable limits. While the general idea of low rewards for not finishing and higher rewards for finishing remain, training is appreciably speed up with a positive finishing reward. The algorithms still track the number of moves correctly even with the change in the reward function.

While the goal of the reinforcement learning algorithm is to operate the operator through the track itself, it is helpful to be able to attempt the track manually. This can be helpful for testing, debugging, and setting a baseline against which the learner can compare against. For each turn, the board will be printed, the path the operator takes will be shown, and the result of the movement will be displayed. After completion of the track, the number of moves will be displayed to the user to be used as a baseline.

A number of hyperparameters are used in the learning algorithms in order to facilitate optimal functioning of the algorithms. There are different hyperparameters for each of the algorithms. For value iteration there are just two hyperparameters. The first is a threshold value designating the stopping criteria when learning is completed, and is the difference between the v values of the previous and current learning iteration. This value was set to 0.01 to allow for near full convergence of the value iteration algorithm. The second hyperparameter is the discount factor. This was set to 0.9 to allow for a good learning gradient.

For the q-learning algorithm, there are a different set of hyperparameters. The first is the total number of episodes, indicating how many times the learner should try to navigate the track. This was set to 5000 to balance the learning time and convergence. The second is the epsilon value indicating the percentage of time the learner will select a random action rather than the best known action. This was set to select a random action 10% of the time. Another hyperparameter is the step length taken to update q value estimation, which was set to 0.1. Finally, the discount factor was selected and set to 0.6.

For the sarsa algorithm, there are yet another set of hyperparameters. Similar to the q-learning algorithm, the number of episodes and the epsilon value were set to 5000 and 10% respectively to ensure fair comparisons between the algorithms. Some alterations needed to be made to the step length to ensure expedient learning, and the step length was set to 0.85. Similarly, the discount factor was changed to a value of 0.95.

The hyperparameters chosen were not tuned automatically like in previous machine learning implementations due to the long learning time for each of the algorithms. Instead, the hyperparameters were tuned through a combination of manual tuning and other online algorithm implementations. The hyperparameters chosen help the learner to make appropriate learning steps, however can be further optimized for increased performance if need be.

The v-values in the value iteration algorithm are initialized to zeros, thus there are no random components to the value iteration algorithm. Because of this, each subsequent run of the algorithm will produce the exact same result. In contrast, both q-learning and sarsa have random components in the algorithm. The algorithm will occasionally select a random action to take, thus making the exploration process somewhat random. While the optimal result should converge to

the same optimum, the learning process will differ each time the algorithm is run. Because of this, the q-learning and sarsa algorithms are run multiple times to get an idea of the learning process for each. Logs for each subsequent run are marked with -<index>, and plots for each of the experiments are stored in the plots directory. A number of experiments are performed on each of the tracks and crash behaviors. Discussed results are typically from one representative experiment.

3 Results

Each of the three learning algorithms discussed (value-iteration, q-learning and sarsa) are run on a variety of different tracks to showcase the flexibility of the learning algorithms. The tracks are named l, o and r, and the track shape resembles the capital letter of the track name. There are also two different crash behaviors considered, one where walls stop the operator, and one where walls send the operator back to the starting line. The nomenclature for the data file naming is <algorithm>-<track_name>-<crash_behavoir>.output.txt. Thus, for q-learning on l track with respawn crash behavior, the output file would be named a-learning-l-track-respawn.output.txt.

There is a wealth of information included in the output files, and referencing those files should give greater insight into low-level items not included in this report. For each of the algorithms, the output is obviously different. A summary of the output for the value iteration algorithm is included below:

- The x and y coordinates of each of the valid spaces on the track and the total number of valid spaces on the board
- The state tuple (x coordinate, y coordinate, x speed, y speed) of each of the valid states on the track and the total number of valid states
- The locations on the board that designate a finished state
- For each learning iteration the following information:
    - The current v value array
    - The v value array after the learning iteration
    - The current learning iteration index and the current delta from the previous learning iteration
- The final v values after the learning delta has crossed the specified threshold for each state
- Board navigation demonstration for the learned track (see below for board navigation logging)
- The number of moves the learning demonstration took
- The state-action-state tuples that the learner took for the track

Similarly, there is lower level information for the q-learning and sarsa algorithms stored in the output files. A summary of the information stored in the output files for the q-learning and sarsa algorithms are shown below:

- The x and y coordinates of each of the valid spaces on the track and the total number of valid spaces on the board

- The state tuple (x coordinate, y coordinate, x speed, y speed) of each of the valid states on the track and the total number of valid states
- For the first and last learning episode, the episode number and the board navigation for the track (see below for board navigation logging). Also included in the board navigation is specification for when a random action is selected and a demonstration of the updated q values.
- For all other learning episodes:
    - The episode number
    - The cumulative reward for that episode
- The learned q values at each state
- Board navigation demonstration for the learned track (see below for board navigation logging)
- The number of moves the learning demonstration took
- The state-action-state tuples that the learner took for the track

Board navigation logging information:

- The starting location
- A demonstration of the learner navigating the board including for each move of the operator:
    - The current location and speed as well as reward at the current location
    - The board (with the operator denoted as @)
    - The chosen action
    - Whether the acceleration attempt failed
    - The board spaces the maneuver will pass through, and the state of the board at each of those points
    - A determination of whether the learner crashed, finished or neither

Closely analyzing the results files are a great way to look at the lower level implementation components for the reinforcement learning algorithms discussed. Some useful things that can be done with the information contained includes basic debugging, following along with the algorithm logic, and further analysis into the learning process. The feed-forward neural network outputs relative confidence values for each prediction. Analyzing the rate of learning can be helpful in determining if the learner has completed its learning process. Looking at the state-action-state tuples can be helpful in seeing what was learned by the learner and learning strategies.

We can compare the performance of each of the learning algorithms against the performance of a human to get a sense of how well the learner has learned the track. A summary of the learning of each of the algorithms against each of the tracks is shown below for the bump crash behavior:

	# Moves for a Track Bump Crash Behavior			
Track	Value-Iteration	Q-Learning	SARSA	Human
l	12	34	46	13
o	28	58	55	31
r	26	429	121	36

Table 1: Summary of learning performance with bump crash behavior

Similarly, we can create a similar table for the respawn crash behavior:

	# Moves for a Track with Respawn Crash Behavior			
Track	Value-Iteration	Q-Learning	SARSA	Human
l	13	14	20	14
o	28	167	353	42
r	29	97	741	39

Table 1: Summary of learning performance with respawn crash behavior

Also interesting to examine are the learning curves for each of the algorithms on each of the tracks for both of the different crash behaviors. As discussed, q-learning and sarsa have random components to their learning, so the learning curves may differ across runs. Each plot shown is a representative sample. However, for value iteration the plots will be the same run to run. Below are the learning plots for the value iteration algorithm:

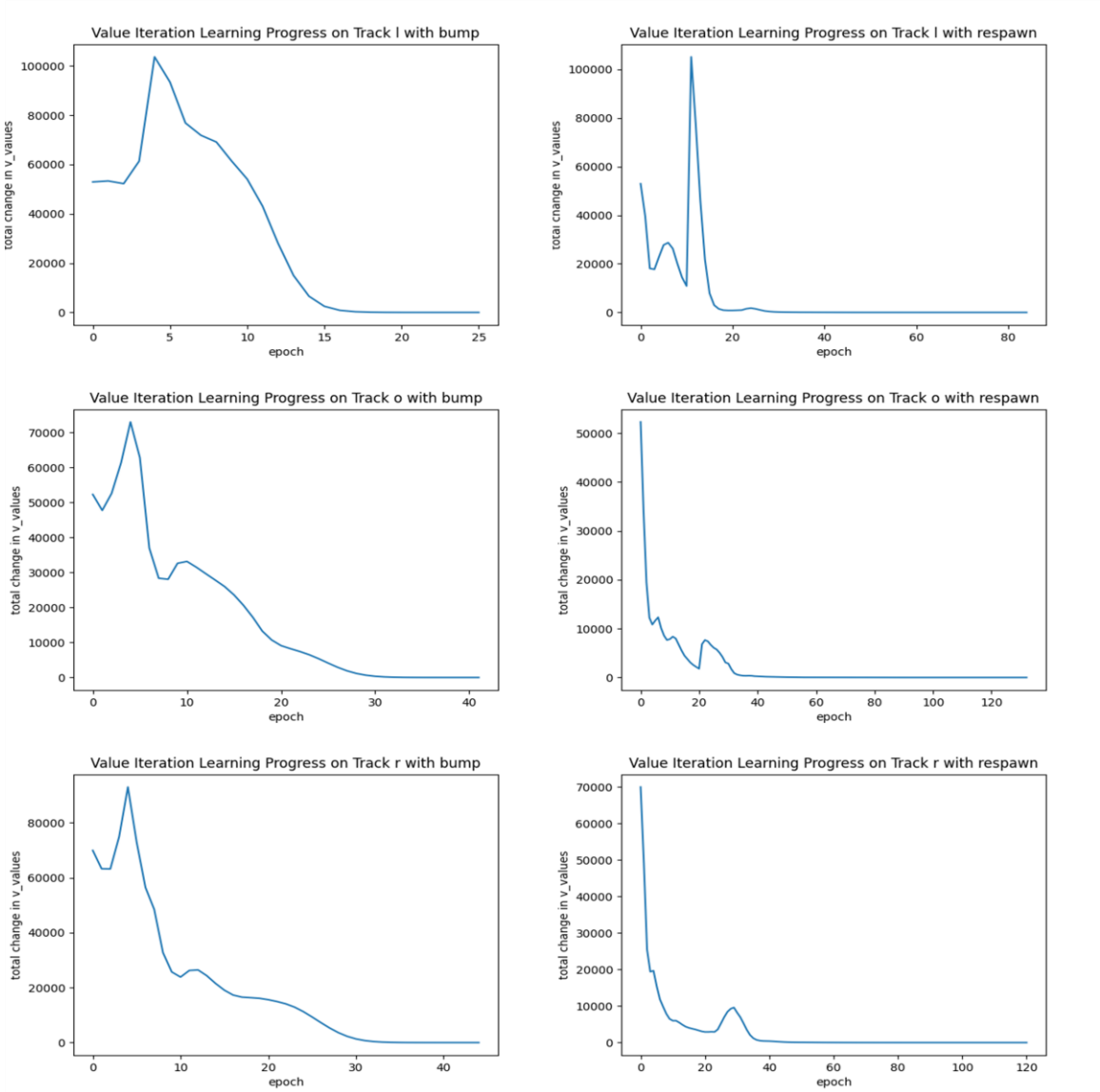

Figure 1-6: Value teration learning curves

We can do the same for the learning curves for q-learning and sarsa. The plots for the q-learning algorithm are shown below in Figures 7-12:

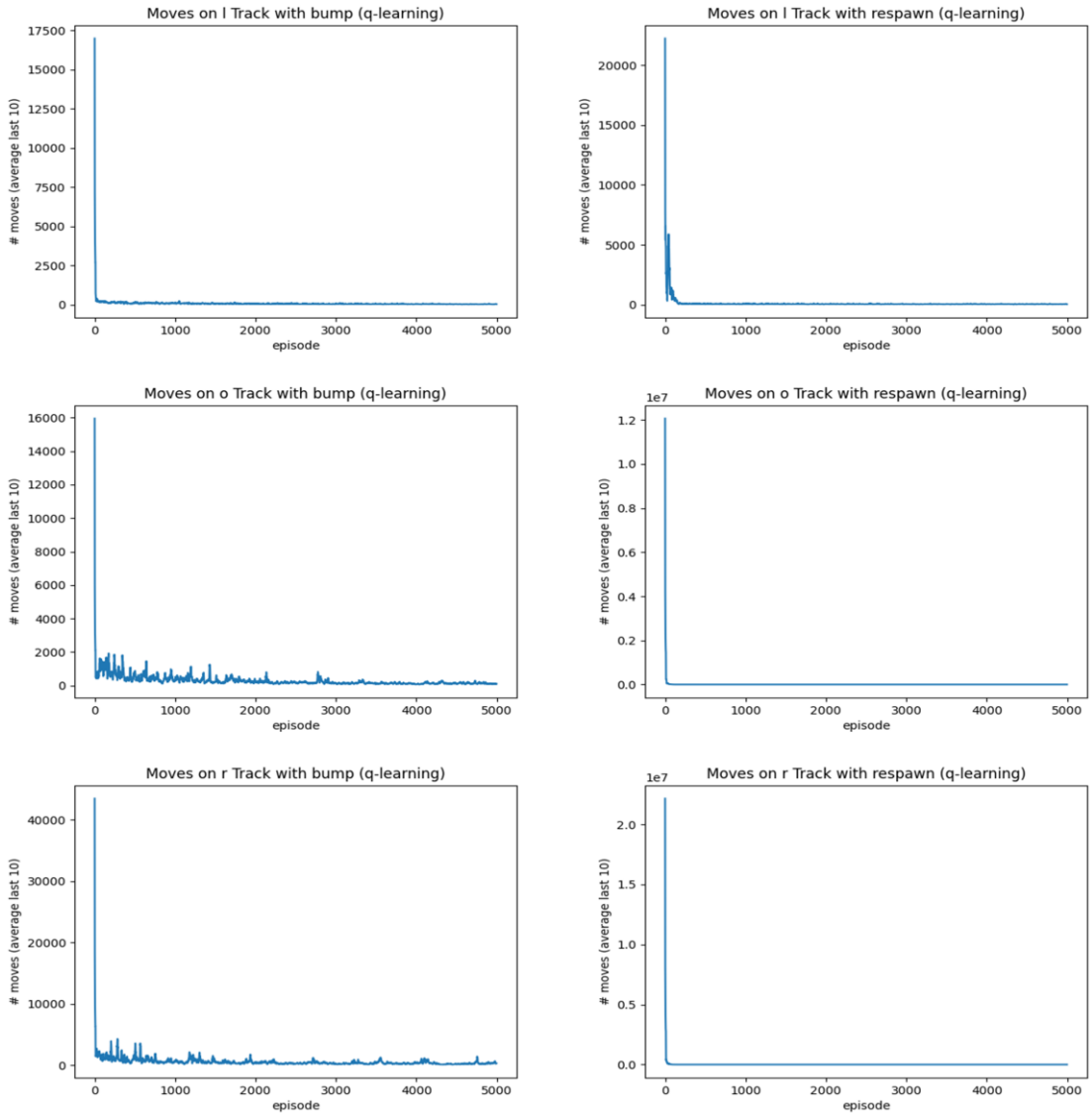

Figure 7-12: Q-learning learning curves

And finally we can do the same for the learning curves for sarsa. The plots for the sarsa algorithm are shown below in Figures 13-18:

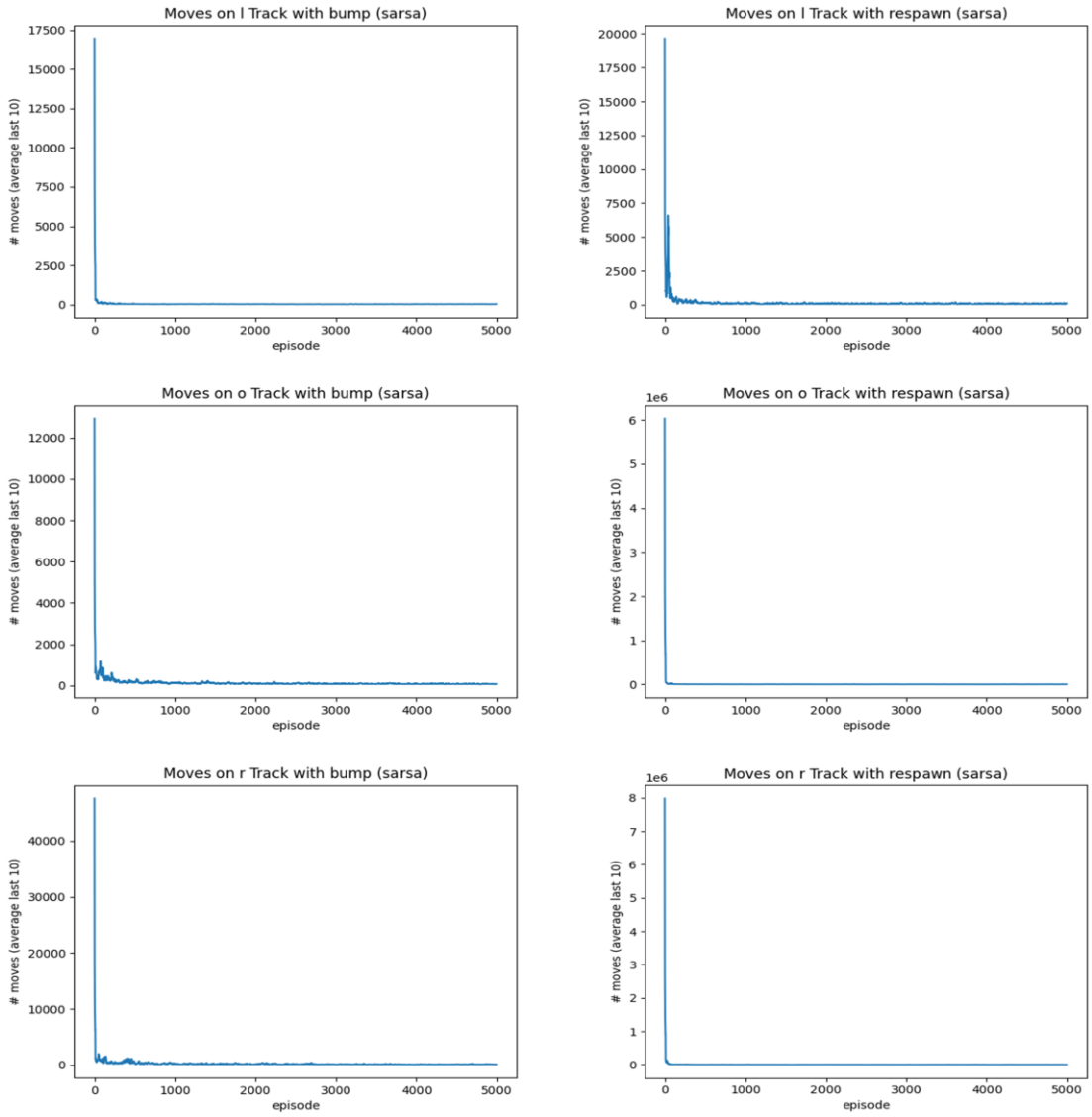

Figure 13-18: Sarsa learning curves

4 Algorithm Behavior

We can make a series of observations from the results presented in the previous section. We can start by comparing the number of moves for each of the algorithms against the human operated run. We note immediately that the value-iteration algorithm performs much better than q-learning and sarsa, especially given the bump crash behavior. This may indicate that q-learning and sarsa have not fully converged for the given tracks. As discussed in the experimental approach, q-learning and sarsa complete the track 5000 times. Increasing the number of episodes should only improve performance (with diminishing returns).

We also note that the respawn crash behavior performs better for the q-learning and sarsa algorithms in some cases. One reason this could be is that the respawn crash behavior will actually spend more time learning the track since it continuously gets sent back to the start. While learning time for these algorithms is much greater, the learning converges much faster (per episode). In the cases that q-learning and sarsa perform worse with the respawn behavior the obvious explanation is that the tracks are more challenging to complete with the respawn

behavior than the bump behavior, especially with the non-deterministic way that acceleration is handled.

Looking at the learning curves for value iteration, it is interesting see that the most learning does not happen on the first iteration. One reason for why this might be is that the finish areas are relatively narrow on each track. As the value iteration algorithm explores more possibilities, the number of moves analyzed expands, which is why most of the curves do not start at the highest learning rate. And of course, after the learning peaks, the learning decreases asymptotically to the threshold value as expected.

Learning curves for q-learning and sarsa can show how quickly the algorithms converge to the optimal solution as well as whether the solution has converged. Clearly q-learning and sarsa converge more quickly on the simpler track (l track) than the more complicated tracks (0 track and r track. Note that for each of the learning plots for q-learning and sarsa the plot is showing the previous 10 episodes in an effort to smooth the learning curves.

5 Conclusion

The problem statement predicted that each of the algorithms would perform within 20% of the optimal performance (human run). This prediction seemed to come true for the value iteration algorithm, which actually outperformed the human runs in most cases. q-learning and sarsa performed generally worse, especially for the bump crash behavior. As discussed previously, this is likely because the learning had not fully converged after 5000 episodes. Increasing the number of episodes or the number of random actions will likely result in better convergence, however would also result in longer learning runtimes.

It should be noted that value-iteration has different information to learn from than q-learning and sarsa. Value-iteration is aware of the structure of the track, while q-learning and sarsa are not. This can give value-iteration an unfair advantage over the other algorithms as model-based learning algorithms. Q-learning and sarsa do not know the state transition probabilities or rewards, and must discover the rewards by taking actions and moving between states. This makes q-learning and sarsa more generalized algorithms than value-iteration, giving them a potential advantage in the case the model is not known. Like many machine learning applications, it is important to know what you are trying to learn in selecting the best possible algorithm.