605.649 Programming Project 6: Reinforcement Learning

Christopher El-Khouri

1. Introduction

Our concluding project for the course takes us to a more advanced topic in machine learning. Reinforcement Learning (RL) is the algorithm design of teaching an agent in an environment on how to make decisions that maximize its reward within that environment to solve a problem. The decisions are represented as actions and each of these actions results in either a penalty or reward. The agent must therefore aim to find the most optimal actions to be taken to solve the problem with the least cost.

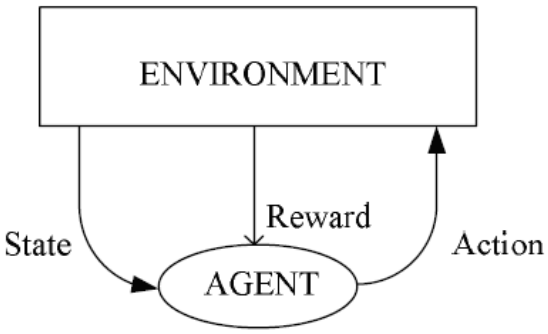

Figure 1: Decision Making Agent in Environment

Figure 1 shows how the decision-making agent interacts with the environment. At each step, the agent decides on the action to take, the environment returns a cost/reward to the agent, and the state of the agent within the environment is modified. The model of the environment is typically considered in Reinforcement Learning is a Markov Decision Process (MDP), credited to Richard Bellman (1957), and is defined by the tuple $< S, A, T, R, \gamma >$ where:

S: The set of states of the environment

A: The set of actions of the agent

T: The state transition function such that $T(s, a, s') = P(s'|s, a)$

R: The reward function which represents the reward received when action a ϵ A is taken

γ: The discount factor, $\gamma \epsilon [0,1]$

Our main focus for this project will be on implementing reinforcement learning using three different algorithms; Value Iteration, Q-Learning, and SARSA.

Value Iteration, credited to Richard Bellman (1957a), is a dynamic programming algorithm used to find the optimal policy of an agent. The algorithm iteratively sweeps through all the states and actions within the environment and updates the values of the state-action policies until the values converge. This algorithm requires a model of the environment to function.

Q-Learning, introduced by Christopher Watkins (1989), is an off-policy RL algorithm. The algorithm works by giving the agent actions and assessing the rewards received based on these actions, then updating the values of the underlying quality function Q. SARSA, an acronym of State-Action-Reward-State-Action, is an on-policy algorithm proposed by G.A Rummery and M.Niranjan (1994). The algorithm works similarly to Q-Learning, however since it is on-policy, it considers the actual next action rather than the optimal. Both Q-Learning and SARSA are considered model-free algorithms, which means that they don't need to know the entire model space in order to optimize.

This project will be solving the racetrack problem, a popular control problem. The goal is to control the movement of a race car (our agent) along a pre-defined racetrack (our environment). The algorithms should enable the race car to get from the starting line to the finish line in a minimum amount of time. At each time step, the state of the agent can be encoded into four variables: $x_t, y_t, \dot{x}_t, \dot{y}_t$. The racetrack is laid out on a Cartesian grid, therefore, x_t and are the coordinates corresponding to the location of the car at time step t. The variables $\dot{x}_t, \dot{y}_t$represent the x and y components of the car's velocity at time t. The control variables for the car are a_x and a_y, which represent the x and y components of an acceleration vector to be applied at the current time step. At any given time step, your car only has active control over the values of a_x and a_y and must use these control variables to influence the car's state. The acceleration values are used to update the velocity values which update the position values as follows:

1. $[a_x, a_y]$ given as an action at time t
2. The velocity values are updated:
$$\dot{x}_t = \dot{x}_{t-1} + a_x$$
$$\dot{y}_t = \dot{y}_{t-1} + a_y$$
3. The position values are updated:
$$x_t = x_{t-1} + \dot{x}_t$$
$$y_t = y_{t-1} + \dot{y}_t$$

Furthermore, our racetracks consist of roads and walls, our algorithms must be capable of detecting the vehicle intersecting with a boundary. We will be assessing two different versions of crashing. The first variant stating that if the car crashes into a wall, it is placed at the nearest position on the track to the place where it crashed, and its velocity is set to 0. The second variant states that when a car crashes, its position is set back to the original starting position, as well as zeroing its velocity.

We will be comparing the performance of each of the algorithms on the racetrack problem. The performance will be measured by assessing the number of iterations each algorithm does and the best track time resulted from that. The experiments will comprise of modifying the discount factor to its effect on convergence, as well as assessing the different crash variants and their effect on racecar performance. Our first hypothesis is that the performance of the race cars using SARSA will have a superior track time to Q-learning. Our second hypothesis is that Q-learning will require a smaller number of iterations to converge than SARSA of the outputs produced will increase with the number of layers used. In the next section, we will describe the technical and theoretical basis of our algorithms and experimental methods, in section 3 we will present and compare the results, and in section 4 we will discuss the significance of the experiments.

2. Algorithms and Experimental Methods

Racetracks

The following racetracks were used in this project:

1. L-Track
2. O-Track
3. R-Track

Each of the tracks will undergo a total of 11 experiments: 3 with Value Iteration, 4 with Q-Learning, and 4 with SARSA. The R-Track, however, will undergo 22 experiments, 11 for each crash variant, resulting in a total of 44 experiments.

Allowable Time

All three algorithms would require a relatively large amount of processing time to optimally converge, therefore, processing time was taken as a factor in our algorithms. Each of the three tracks was given an allowable time to run and this time was inputted as a parameter in our algorithm design so that the programs would break once that time was exceeded. This was implemented in an effort to produce results in a faster way and to give a more relatable measure of performance. Table 1 below shows the allowable times in minutes for each algorithm and each track, the complexity of the tracks, algorithms, and the number of experiments were taken into account when deciding the allowable time values.

Table 1: Allowable Times

Track/Algorithm	**Allowable Time to Run (minutes)**			
	Value Iteration	**Q-Learning**	**SARSA**	**Total**
L	24	48	48	120
O	48	96	96	240
R (without restart)	36	72	72	180
R (with restart)	36	72	72	180
Total (minutes)	144	288	288	**720**

Value Iteration

The Value Iteration algorithm used in this project is based on the pseudocode in Figure 2. The algorithm updates the values of V until the change between the V values from one step to another is less than the error threshold. This algorithm has been modified for our project by giving an allowable time to run as well as batch calculating the value of the summation of the product of the state transfer function with the previous policy value function to improve running time.

```
1: function ValueIteration(MDP, ε)
2:     for all s ∈ S do
3:         V_0 ← 0
4:     end for
5:     t ← 0
6:     repeat
7:         t ← t + 1
8:         for all s ∈ S do
9:             for all a ∈ A do
10:                Q_t(s,a) ← R(s,a) + γ Σ_{s'∈S} T(s,a,s') V_{t-1}(s')
11:            end for
12:            π_t(s) ← arg max_{a∈A} Q_t(s,a)
13:            V_t(s) ← Q_t(s, π_t(s))
14:        end for
15:    until max_{s∈S} |V_t(s) − V_{t-1}(s)| < ε
16:    return π_t
17: end function
```

Figure 2: Value Iteration pseudocode

Q-Learning

The Q-Learning algorithm used in this project is based on the pseudocode shown in Figure 3. For each Q-Learning episode, the algorithm initializes the state, chooses an action, takes the action, and updates the values of Q until s is a terminal state. This algorithm has been modified for our project by giving an allowable time to run, a maximum number of episodes, and an error threshold to decide convergence.

```
Initialize all Q(s,a) arbitrarily
For all episodes
    Initalize s
    Repeat
        Choose a using policy derived from Q, e.g., ε-greedy
        Take action a, observe r and s'
        Update Q(s,a):
            Q(s,a) ← Q(s,a) + η(r + γ max_{a'} Q(s',a') − Q(s,a))
        s ← s'
    Until s is terminal state
```

Figure 3: Q-Learning pseudocode

SARSA

The SARSA algorithm used in this project is based on the pseudocode in Figure 4. This algorithm is similar to Q-Learning but with a key difference, the Q values are updated based on the actual action chosen rather than the optimal. This algorithm has been modified for our project by giving an allowable time to run, a maximum number of episodes, and an error threshold to decide convergence.

Initialize all $Q(s,a)$ arbitrarily
For all episodes
    Initalize s
    Choose a using policy derived from Q, e.g., ϵ-greedy
    Repeat
        Take action a, observe r and s'
        Choose a' using policy derived from Q, e.g., ϵ-greedy
        Update $Q(s,a)$:
            $Q(s,a) \leftarrow Q(s,a) + \eta(r + \gamma Q(s',a') - Q(s,a))$
        $s \leftarrow s',\ a \leftarrow a'$
    Until s is terminal state

Figure 4: SARSA pseudocode

Other Methods

The algorithms above utilize several methods to function properly. The reward function **R** is defined as follows:

$$\boldsymbol{R(s)} = \begin{cases} 0, & s\ in\ Finish\ Line \\ -1, & Otherwise \end{cases}$$

The state transfer function **T** is defined as follows:

$$\boldsymbol{T(s,a,s')} = P(s'|s,a) = 0.8$$

Due to the non-deterministic nature of the project such that the actions given have a 20% chance of not being executed, a value of 0.8 is returned. Since our racetrack has boundary walls, two methods were developed to detect any crashes. The **wallchecker** algorithm described in Figure 5 below takes a state **s** and an action **a**, sets the velocity and position values of the next state **s'** then performs the **LineGen** algorithm between the positions of **s** and **s'** to return the points that form the line, **ps**. **wallchecker** proceeds to iterate through **ps** and depending on the value of **restart**, sets the actual positions and velocities of the next state and returns it.

wallchecker(**s**, **a**, **restart**):
Where:
s: The current state
a: The action to be taken
restart: Whether or not to restart the race car incase of a crash
    $vx = vx_s + a_x$
    $vy = vy_s + a_y$
    $px = x_s + vx$
    $py = y_s + vy$
    $ps =$ **LineGen**(x_s, y_s, px, py)
    for p in ps:
        if p in **walls**:
            if restart = True:
                px, py = Random Starting Line Point
                vx, vy = 0,0
            else:
                px, py = Nearest Crash Point on Track
                vx, vy = 0,0
    return px, py, vx, vy

Figure 5: wallchecker pseudocode

In the **LineGen** pseudocode further below, a straight line is generated between two input points (x1,y1) and (x2,y2) based on the straight-line equation: y=mx+b. The algorithm has 2 main cases upon which to run; x1≠x2, and x1=x2. In the case that x1≠x2, starting from the smaller x, the y value is calculated using the straight-line equation. Due to the racetrack only having integers as coordinates, the calculated y value is rounded to the nearest integer, and the algorithm proceeds to ensure that no y values were skipped. The points are appended into an array **ps** and is returned.

LineGen(**x1**, **y1**, **x2**, **y2**):
Where:
x1, y1: The coordinates of the first point
x2, y2: The coordinates of the second point

```
if(x1 < x2):
    xs = x1
    xb = x2
    y = y1
    m = (y2 − y1) / (x2 − x1)
    b = y1 − m(x1)
elif(x2 < x1):
    xs = x2
    xb = x1
    y = y2
    m = (y1 − y2) / (x1 − x2)
    b = y1 − m(x1)
else:
    ys = min(y1, y2)
    yb = max(y1, y2)
ps = [ ],x = xs
if(x1! = x2):
    while x < xb + 1:
        y = round(mx + b)
        if(y > (last y value in ps) + 1):
            xn = round( ((last y value in ps) + 1 − b) / m )
            ps.append([xn, (last y value in ps) + 1])
        else if(y < (last y value in ps) − 1):
            xn = round( ((last y value in ps) − 1 − b) / m )
            ps.append([xn, (last y value in ps) − 1])
        else:
            ps.append([x, y])
            x+= 1
else:
    while(y < yb + 1):
        ps.append([x1, y])
        y+= 1
return ps
```

The project also utilizes a method called **Qsegmentor**, a method that splits a racetrack before running the Q-Learning and SARSA algorithms. Each of the racetracks has a different splitting mechanism in **Qsegmentor**. The purpose of using this method is to speed up convergence. The figure below shows how **Qsegmentor** was utilized on the L-track.

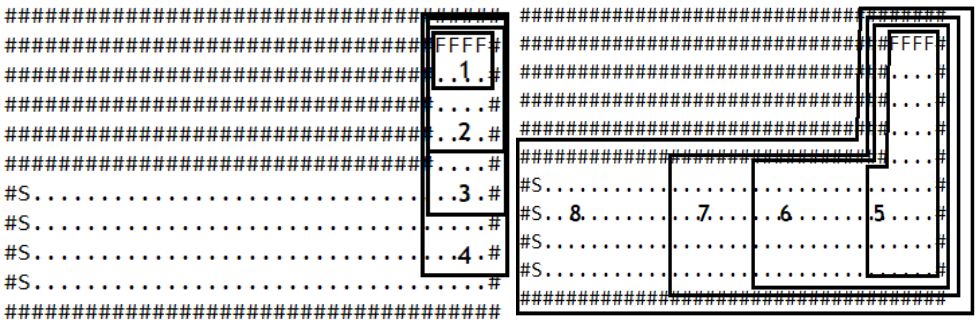

Figure 6: L-Track splitting

The numbers indicated on the figures refer to the order upon how the states were entered into the Q-Learning and SARSA algorithms. The O-track was split into two halves with the first half starting from the finish line and ending halfway along the track. The R-track was not split.

3. Results

The table below shows the results of our Value Iteration algorithm on the different tracks.

Table 2: Value Iteration Experiment Results

Value Iteration Experiment Results							
Track	Experiment	Discount Factor	ε	t	Error	Training Time (minutes)	Steps Needed to Finish Race
L	1	0.1	0.001	3	0.0064	13.6	NA
	2	0.5	0.001	3	0.16	13.7	NA
	3	0.9	0.001	3	0.52	13.7	NA
O	12	0.1	0.001	3	0.0064	19.1	NA
	13	0.5	0.001	3	0.16	19.1	NA
	14	0.9	0.001	3	0.52	19.3	NA
R without Restart	22	0.1	0.001	2	0.08	14.3	NA
	23	0.5	0.001	2	0.4	14.3	NA
	24	0.9	0.001	2	0.72	14.5	NA
R with Restart	33	0.1	0.001	2	0.08	14.7	NA
	34	0.5	0.001	2	0.4	13.9	NA
	35	0.9	0.001	2	0.72	13.6	NA

The table below shows the results of our Q-Learning and SARSA algorithms on the different tracks.

Table 3: Q-Learning and SARSA Experiments Results

Q-Learning and SARSA Experiments								
Track	Experiment	Algorithm	Discount Factor	ε	n	Number of Iterations	Training Time (minutes)	Steps Needed to Finish Race
L	4	Q-Learning	0.1	0.01	1	602	7.1	NA
	5		0.5			3061	10.3	19
	6		0.9			5984	12.1	40
	7					5949	12.1	15
	8	SARSA	0.1			5024	6.55	NA
	9		0.5			5374	6.93	25
	10					7025	8.42	17
	11		0.9			4900	6.92	18
O	15	Q-Learning	0.1	0.01	1	458	17.4	NA
	16		0.5			2759	26	37
	17					1480	32	55
	18		0.9			8841	36.1	37
	19	SARSA	0.1			3088	21.9	NA
	20		0.5			4291	26.4	66
	21		0.9			3236	23.2	NA
R w/o Restart	25	Q-Learning	0.1	0.01	1	85	19	NA
	26		0.5			63	19	261
	27		0.9			93	18.5	120
	28					60	18.7	NA
	29	SARSA	0.1			91	19.6	NA
	30		0.5			65	18.3	NA
	31					86	20	NA
	32		0.9			66	19.3	NA

Figures 7 and 8 below show the plots of the average percentage of optimal actions vs the number of iterations for experiments 7 and 18 respectively.

4. Discussion

Looking at Table 2, although the Value Iteration algorithm used in this project converged the values of V, we were not able to complete a race with it in any of the tracks, it would seem that the training algorithm may have room for improvement.

As for Table 3, we can see that the discount factor plays a role in convergence. The number of iterations increases with gamma in Q-Learning, but seem to decrease with gamma in SARSA. Successful convergence tended to occur in both Q-Learning and SARSA with discount factors of 0.5 and 0.9.

Unfortunately, only 10 out of the 11 experiments for the O-track were completed, and none of the Q-Learning or SARSA algorithms were able to converge with the R-track with restart. I am confident that giving the algorithms more time with the R-track and developing an optimal splitting process would yield good results.

Splitting the track as we did with our Qsegmentor algorithm for the L-track seemed to enhance convergence, this is shown in the figures below. The L-track (Figure 7) started with a higher average percentage of optimal action than the O-track (Figure 8) did, and resulted in a higher average percentage by the end of the runs.

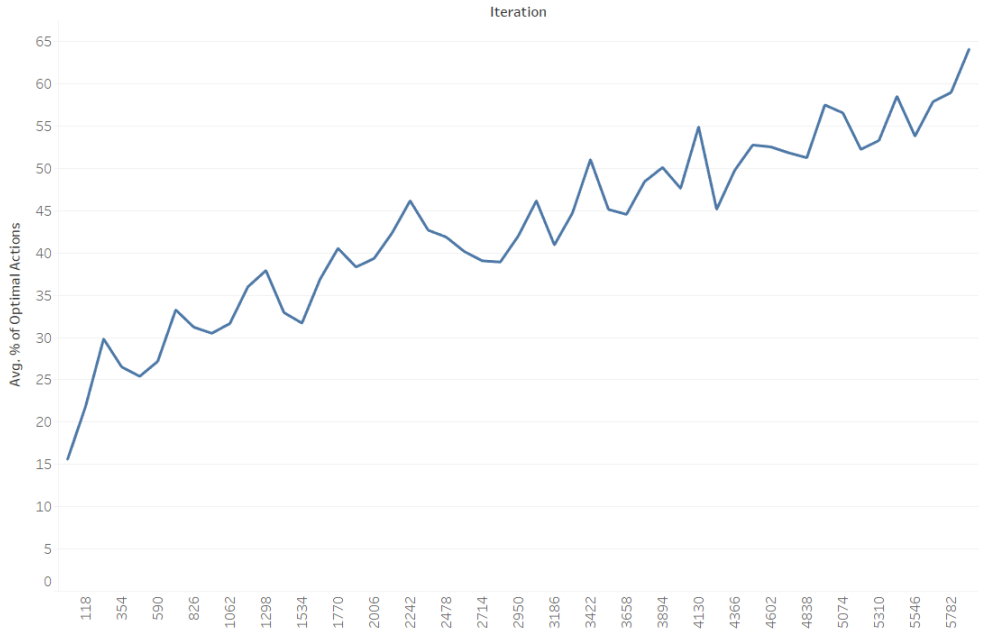

Figure 7: Experiment 7 Learning Curve

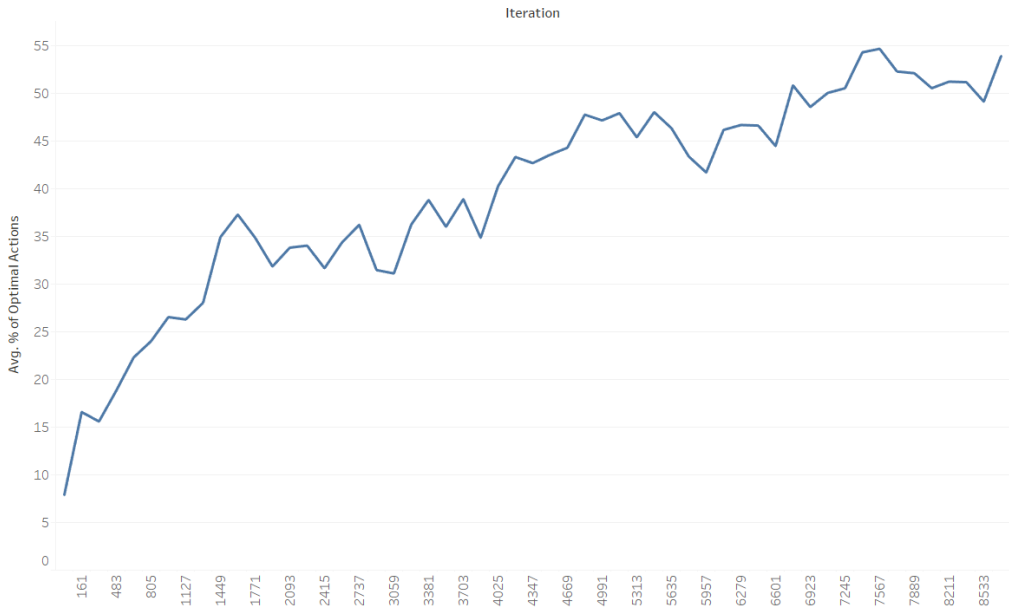

Figure 8: Experiment 18 Learning Curve

5. Conclusion

In conclusion, referring back to our first hypothesis, that the performance of the race cars using SARSA will have a superior track time to Q-learning, our results have shown that this does not stand, at least with the way the experiments have been conducted.

Our second hypothesis, that Q-learning will require a smaller number of iterations to converge than SARSA, seems to stand with all successfully converged Q-learning results requiring fewer iterations than the successfully converged SARSA results. However, I would not conclude the results of either of the hypotheses as I believe that both SARSA and Q-learning would yield different results if more time was allocated in training.

6. References

Richard Bellman.
A Markovian Decision Process.
Journal of Mathematics and Mechanics, 6(5):679-684, 1957

Richard Bellman.
Dynamic Programming.
Princeton University Press, Princeton, New Jersey, 1957a.

Christopher Watkins.
Learning from Delayed Rewards.
PhD thesis, King's College, London, United Kingdom, 1989.

G. A. Rummery & M. Niranjan
On-Line Q-Learning Using Connectionist Systems.
Technical Report 166, Department of Engineering, Cambridge University, Cambridge, United Kingdom, 1994.