Project 2: Reinforcement Learning for Robot Wall Following CSCI-573 Human-Centered Robotics

Casey Duncan
Colorado School of Mines
1500 Illinois St, Golden, CO 80401
caseyduncan@mymail.mines.edu

***Abstract*— In machine learning, the area of Reinforcement Learning (RL) is used to teach a software agent to take specific actions within an environment that maximize it's reward. In this paper, I will use the Q-Learning and SARSA RL algorithms to train a robot to follow a wall and navigate a variety of obstacles. I begin by explaining how the robot will be controlled, and how it's control algorithm will be used to define the RL states and actions. Next, I show that both Q-learning and SARSA prove to be successful at training a robot to successfully navigate these obstacles while following a wall. However, each have their advantages and disadvantages. If you would like for the robot to learn the optimal wall-following policy, choosing Q-learning would be the better of the two. This is because it uses the maximum Q-value for the current state when rewarding the robot in each epoch. However, if you would like the robot to explore a variety of near optimal solutions, selecting SARSA would be the better choice. This is because it learns a near-optimal policy while continuing to explore for a more optimal policy. For a demonstration of each policy, see the YouTube links to their demo videos provided at the end of section III.**

I. INTRODUCTION

The purpose of this project was to design and implement two Reinforcement Learning (RL) algorithms (Q-Learning & SARSA) to teach an autonomous mobile robot to follow a wall and avoid running into obstacles. This was done using a simulation in Gazebo and the robot was programmed in C++ within ROS Melodic on Ubuntu 18.04 LTS. The robot used is an omni-directional mobile robot named Triton, which can be seen in Figure 1, and is controlled using steering and velocity commands.

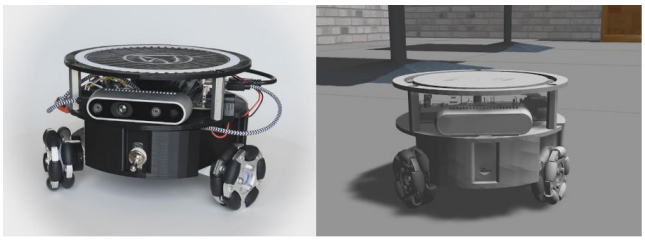

Fig. 1. Triton Robot

To control the steering of the robot, I used the Stanley method path tracking algorithm [1], which kept the robot at a desired distance from the wall and allowed it to navigate the environment obstacles shown in Figure 2. For each RL algorithm, I defined the robot's states as it's heading angle error, which is the difference between where the heading should be versus where the Stanley method calculated it should be. I defined the robot's actions as the angular velocity it would require to drive it's heading angle error to zero. A more detailed explanation to this solution can be seen in section II.

II. APPROACH

In order for the robot to follow a wall, it must first detect the wall using it's sensors. Then it needs to implement a control method to drive straight along the wall and turn when needed at the obstacles shown in Figure 2.

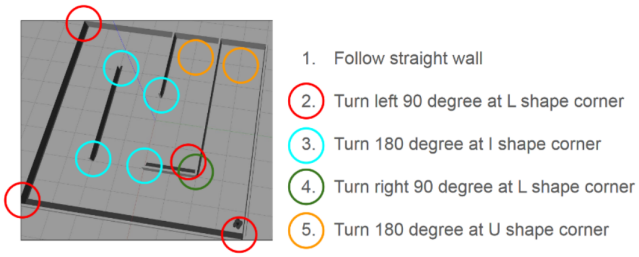

Fig. 2. Environment Obstacles

In order for it to learn to follow the wall, RL is used. In machine learning, the area of RL is used to teach a software agent to take specific actions within an environment that maximize it's reward. In the following sections, I will go over how to implement the sensing (section II-A), control (sections II-B & II-C), and Reinforcement Learning (section II-D) components on the Triton Robot so that it can autonomously learn to follow a wall.

A. Robot Sensing

The Triton robot includes a LIDAR scanner that scans the robot's surroundings in 360 degrees and provides the distance to each obstacle it sees at an angle increment of 1 degree per measurement. It other words, the LIDAR scanner outputs an array of 360 distances. The obstacle closest to the robot can be calculated as the detected obstacle with the smallest distance magnitude. The angle of the obstacle with respect to the robot's heading can be calculated as the index of the distance within the 360 distance array multiplied by the angle increment. For example, if the obstacle with the smallest distance is at index 270 in the array of 360 distances, the

angle of the obstacle with respect to the robot's heading is 270 degrees. For future reference, I will call the distance to the closest obstacle d_{min} and the angle of the closest obstacle with respect to the robot's heading ϕ. Reference Figure 3 for robot coordinate axis and Figure 4 for a robot diagram showing variables d_{min} and ϕ.

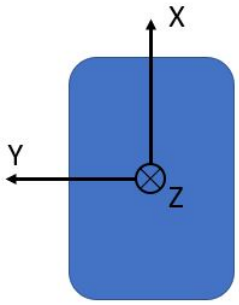

Fig. 3. Triton Coordinate Axis

B. Robot Steering

Since the robot is omni-direction, it is capable of receiving three different commands that allow it to be steered:

- X-Axis Linear Velocity (v_x)
- Y-Axis Linear Velocity (v_y)
- Z-Axis Angular Velocity (ω_z)

To simplify the control algorithm, I chose to steer the robot only using ω_z while supplying a constant v_x of 0.3 meters per second and a constant v_y of 0.0 meters per second.

C. Robot Path Planning Algorithm

In order to ensure that the robot would follow the wall, I implemented the Stanley method path tracking algorithm, which was used by Stanford University's autonomous vehicle entry in the DARPA Grand Challenge [1]. The way this method works is that there is a user-set point in front of the robot at all times and the robot is constantly steered towards that point. For this problem, the user-set point is set at a distance of d_{des} from the closest detected wall and a distance of d_{stan} in front of the robot along it's x-axis (Reference Figure 4). Next, the desired angle that the robot's x-axis should be aligned with is calculated, which is called the Stanley Angle (θ_{stan}). This can be calculated using d_{des}, d_{stan}, and d_{min} as shown in equation 1.

$$\theta_{stan} = \text{atan2}(d_{stan}, d_{min} - d_{des}) \tag{1}$$

Next, the angle error (θ_{err}) between the robot's current x-axis heading and θ_{stan} is calculated as shown in equation 2 (Reference Figure 4).

$$\theta_{err} = |360^\circ - \phi - \theta_{stan}| \tag{2}$$

Additionally, I made sure that θ_{err} was always a value between 0 degrees and 180 degrees by subtracting it from 360 degrees if it was greater than 180 degrees.

Now that the robot knows it's θ_{err}, it can assign a value for ω_z to correct it's angle until the θ_{err} is equal to zero. However, it first must decide whether it should rotate clockwise

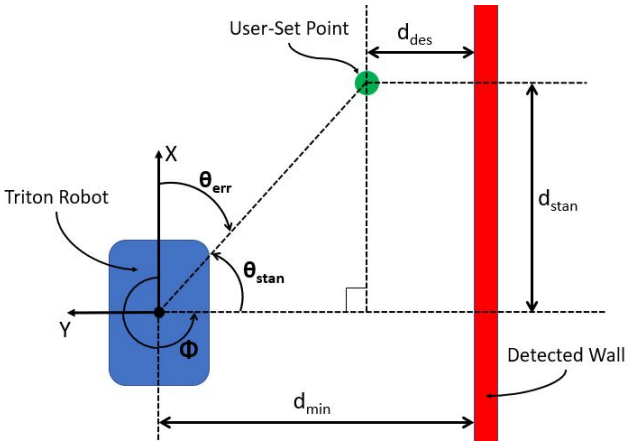

Fig. 4. Stanley Method Control

(CW) or counter-clockwise (CCW). To do this, it determines which direction would require turning the smallest rotation angle and turns that direction. The algorithm for this can be seen in algorithm 1.

Algorithm 1 Rotate CW or CCW

```
if |360° − φ − θ_stan| < π then
    if 360° − φ − θ_stan > 0 then
        Rotate CW.
    else
        Rotate CCW.
    end if
else
    if 360° − φ − θ_stan < 0 then
        Rotate CW.
    else
        Rotate CCW.
    end if
end if
```

Once it has decided whether it needs to turn CW or CCW, the robot assigns a value for ω_z to correct it's angle until the θ_{err} is equal to zero. If it needs to turn CW, ω_z is negative. If it needs to turn CCW, ω_z is positive. When choosing the magnitude for ω_z, it is wise to decrease the magnitude as the robot's θ_{err} approaches zero so that the robot avoids overshooting it's desired angle. This should be kept in mind when designing the reward component of the reinforcement learning algorithm, which I will discuss in section II-D.

D. Reinforcement Learning Implementation

Reinforcement Learning was implemented on the wall following problem so that the robot could train a model, through trial and error, that would allow the robot to follow a wall. The trained model is called a Q-table, which is a matrix of size State quantity $\times$ Action quantity, and the robot is rewarded for how each (state, action) pair perform within the environment. For this project, two different reinforcement learning methods were used, which are the Q-Learning algorithm and the SARSA algorithm. Both algorithms are

temporal difference (TD) algorithms, meaning that they enable the agent, or robot, to iteratively update its Q-table for every single action that it takes. For this project, both algorithms use the same states, actions, and rewards. Using the θ_{err} defined in the Stanley method (section II-C), I chose the states to be angle bins, in degrees, that θ_{err} could fall into. These bins can be seen in equation 3.

$$\begin{aligned} &\text{State 1}: 120^\circ < \theta_{err} \leq 180^\circ \\ &\text{State 2}: 80^\circ < \theta_{err} \leq 120^\circ \\ &\text{State 3}: 40^\circ < \theta_{err} \leq 80^\circ \\ &\text{State 4}: 20^\circ < \theta_{err} \leq 40^\circ \\ &\text{State 5}: 5^\circ < \theta_{err} \leq 20^\circ \\ &\text{State 6}: 0^\circ \leq \theta_{err} \leq 5^\circ \end{aligned} \tag{3}$$

As for the actions, I chose a range of ten different ω_z magnitudes in degrees per second that would be assigned to the robot. These angular velocities can be seen in equation 4.

$$\begin{aligned} &\text{Action 1}: |\omega_z| = 0^\circ/\text{second} \\ &\text{Action 2}: |\omega_z| = 15^\circ/\text{second} \\ &\text{Action 3}: |\omega_z| = 30^\circ/\text{second} \\ &\text{Action 4}: |\omega_z| = 45^\circ/\text{second} \\ &\text{Action 5}: |\omega_z| = 60^\circ/\text{second} \\ &\text{Action 6}: |\omega_z| = 75^\circ/\text{second} \\ &\text{Action 7}: |\omega_z| = 90^\circ/\text{second} \\ &\text{Action 8}: |\omega_z| = 110^\circ/\text{second} \\ &\text{Action 9}: |\omega_z| = 130^\circ/\text{second} \\ &\text{Action 10}: |\omega_z| = 150^\circ/\text{second} \end{aligned} \tag{4}$$

Because I wanted to emphasize turning at specific angular velocities when the robot was in a specific state, I chose to make the reward value into a State quantity $\times$ Action quantity (6×10) matrix, so that each (state, action) pair would be given it's own reward. The reward values can be seen in equation 5.

$$\text{Reward} = \begin{bmatrix} -5 & -4 & -3 & -2 & -1 & 0 & 1 & 2 & 3 & 50 \\ -4 & -3 & -2 & -1 & 0 & 1 & 2 & 3 & 50 & 3 \\ -2 & -1 & 0 & 1 & 2 & 3 & 50 & 3 & 2 & 1 \\ 2 & 3 & 50 & 3 & 2 & 1 & 0 & -1 & -2 & -3 \\ 3 & 50 & 3 & 2 & 1 & 0 & -1 & -2 & -3 & -4 \\ 50 & 3 & 2 & 1 & 0 & -1 & -2 & -3 & -4 & -5 \end{bmatrix} \tag{5}$$

Row 1 corresponds to the rewards for each action within state 1, row 2 corresponds to the rewards for each action within state 2, and so on. Additionally, column 1 corresponds to the rewards for action 1 for each state, column 2 corresponds to the rewards for action 2 for each state, and so on. As you can see, the reword for state 6 ($0^\circ \leq \theta_{err} \leq 5^\circ$) and action 1 ($\omega_z = 0^\circ/\text{second}$) is 50, which is larger than any other reward for state 6. This is because I do not want the robot to turn at an angular velocity of magnitude greater than $0^\circ/\text{second}$ when the θ_{err} is between 0° and 5°.

In addition to the states, actions, and rewards, both Q-Learning & SARSA algorithms use the same learn rate, relative value, and explore rate. These values are defined and provided in the sections for Q-Learning (section II-D.1) and SARSA (section II-D.2).

1) Q-Learning: The Q-learning algorithm implements an off-policy TD control, meaning that it does not follow a policy to find a new action given the state the robot is in. Instead, it chooses the next action, given it's current state, based on the action within the Q-table with the maximum reward. The algorithm for Q-learning can be seen in algorithm 2. Please note that the Q-table is initialized to be a matrix of size State quantity $\times$ Action quantity (in this case, 6×10) where all values in the matrix are zero prior to starting to update the Q-table.

Algorithm 2 Q-learning (off-policy TD control)

Initialize $Q(S,A)$ for all states (S) & actions (A) to be 0.
while Epoch $e < e_{end}$ **do**
  if Epoch $e = 1$ **then**
    Initialize current state (s) using robots θ_{err}.
  end if
  Find current action (a) using ε-greedy policy and s.
  Apply a and observe new state (s').
  Observe reward ($R = R(s,a)$) using s and a.
  Update Q-Table using the following equation:
  $Q(s,a) = Q(s,a) + \alpha(R + \gamma max_a Q(s',a) - Q(s,a))$

  Current state becomes new state ($s = s'$).
  Increase Epoch (e) by 1.
end while

In algorithm 2, an epoch (e) is defined as a single loop through the algorithm and is increased by a value of one epoch each loop until e is equal to the total number of training epochs (e_{end}). Other variables included in the algorithm are the learn rate (α), which is a value between 0 and 1 that puts emphasis on the how much of what has already been learned should be contributed to the Q-table in each epoch, and the relative value (γ), which is a value between 0 and 1 that puts emphasis on the largest Q-value within the robot's new state (s'). Also, it should be noted that the ε-greedy policy is a policy used for selecting an action, which is outlined in algorithm 3.

Algorithm 3 ε-Greedy Policy

Choose an explore rate (ε) between 0 and 1.
Generate a random number (r) between 0 and 1.
if $r < \varepsilon$ **then**
  $a = max_a Q(s,a)$
else
  Choose random action.
end if

2) SARSA: The SARSA algorithm implements an on-policy TD control, meaning that it chooses the action for each state during the learning instead of selecting the next action, given it's current state, based on the action within the Q-table with the maximum reward. The algorithm for SARSA can be seen in algorithm 4. Equivalent to the Q-Learning algorithm, it should be noted that the Q-table is

initialized to be a matrix of size State quantity $\times$ Action quantity (in this case, 6×10) where all values in the matrix are zero prior to starting to update the Q-table.

Algorithm 4 SARSA (on-policy TD control)

Initialize $Q(S,A)$ for all states (S) & actions (A) to be 0.
while Epoch $e < e_{end}$ **do**
  if Epoch $e = 1$ **then**
    Initialize current state (s) using robots θ_{err}.
    Find current action (a) using ε-greedy policy & s.
  end if
  Apply a and observe new state (s').
  Observe reward ($R = R(s,a)$) using s and a.
  Find new action (a') using ε-greedy policy and s'.
  Update Q-Table using the following equation:
  $Q(s,a) = Q(s,a) + \alpha(R + \gamma Q(s',a') - Q(s,a))$

  Current state becomes new state ($s = s'$).
  Current action becomes new state ($a = a'$).
  Increase Epoch (e) by 1.
end while

The variables e, α, and γ are defined the same as they are defined within the Q-Learning section (II-D.1). The only difference is that γ is putting emphasis on the Q-value associated with the s' and a' pair instead of the largest Q-value within the robot's s'.

E. Training

For training the models using both algorithms, I chose for the user set point, shown in Figure 4, to be at a distance of $d_{des} = 0.6$ meters from the wall and set the point distance at $d_{stan} = 0.3$ meters in front of the robot. For each algorithm, I chose the learn rate (α) to equal 1.0, the relative value (γ) to equal 0.5, and the explore rate (ε) to be 0.9. Additionally, I chose to train on each algorithm for 30 minutes with there being 10 epochs (e) per second.

While training I encountered one issue, which was when the robot would get stuck driving into a wall. When this would happen for a long enough amount of time (about 5 seconds), the robot would start rapidly accelerating linearly along its z-axis and leave the simulated environment. After a couple minutes, the robot would drop back down into the environment and proceed with training. Both algorithms learned relatively quickly, so this was rarely an issue and only occurred early on in the training when I was still selecting my rewarding strategy. However, I thought it should be noted just in case this was an issue with the Gazebo environment and the creators wanted to look into it.

Once training was completed, I tested the Q-tables generated by both algorithms and the robot was able to successfully navigate each of the five obstacles (See Figure 2) within the environment without any issues. For a demonstration of each simulation, see the YouTube video links provided at the end of section III.

III. RESULTS

While training the models using both algorithm's, I collected the sum of all Q-table values, or the accumulated reward, and the individual Q-table values for each epoch. Plotting the number of epochs processed versus the accumulated reward for both algorithm's shows how much each model learned per epoch, which can be seen in Figure 5. The Q-Learning algorithm took about 4,000 epochs for it's accumulated reward to begin to flatten out while it took the SARSA algorithm about 1,000 epochs for it's accumulated reward to begin to flatten out. Even though the SARSA algorithm allowed for the model to flatten faster, it's accumulated reward increased by about 750 reward points between epoch 4,000 and 18,000 while the Q-Learning algorithm model only increased it's accumulated reward by about 250 reward points. The reason Q-learning does not change as much towards the end is because the Q-learning algorithm is always learning the optimal policy by using the maximum Q-value for the current state in each epoch. On the other hand, SARSA learns a near-optimal policy while continuing to explore for a more optimal policy, hence why it's reward increases more after beginning to flatten.

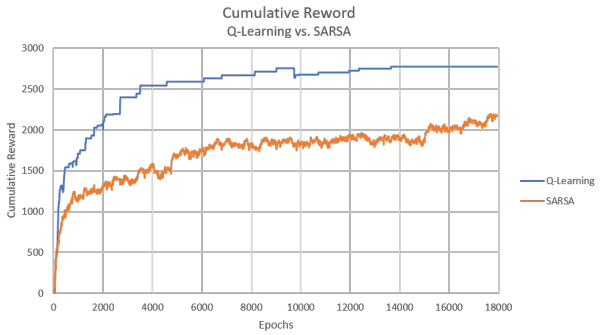

Fig. 5. Accumulated Reward

In order to see which actions are learned for each state as the training progresses, I plotted all of the ten action rewards for each state at each epoch. This can be seen in Figure 6. As you can see, most of the states for Q-learning were able to learn their optimal action within the first 2,000 epochs, and consistently keep these actions as the optimal action throughout the remainder of the training. As stated above, this is because the Q-learning algorithm is always learning the optimal policy by using the maximum Q-value for the current state in each epoch and does not explore new policies. The only state that didn't experience this was state 1 ($120^\circ < \theta_{err} \leq 180^\circ$), which is because the robot is unlikely to have a θ_{err} greater than 90° after starting to follow a wall given the obstacles shown in Figure 2. This is also the case for SARSA. As you can see for SARSA, all of the action rewards for each state dramatically increase and decrease for almost every epoch. As stated above, this is because SARSA learns a near-optimal policy while continuing to explore for a more optimal policy. Even though it continues to search for a more optimal policy throughout the entire training, it's easy

to see that the optimal action is learned early on and remains to have the largest reward throughout the entire training.

For the environment and obstacles shown in Figure 2, the most critical states to learn an action for are states 2 - 6. State 2 is important because obstacles 2 & 4 represent a scenario where the robot experiences a $\theta_{err} = 90^\circ$ and it needs to learn to dramatically increase it's ω_z. As you can see, both algorithms allow it to learn that turning at a $\omega_z = 130^\circ/\text{second}$ would allow it to make these turns successfully. States 3 & 4 are important because obstacles 3 & 5 represent a scenario where the robot experiences a θ_{err} between 20° & 80° and it needs to learn to increase it's ω_z in increments to complete these turns. As you can see, both algorithms allows it to learn that turning at a $\omega_z = 90^\circ/\text{second}$ when in state 3 and $\omega_z = 30^\circ/\text{second}$ when in state 4 would allow it to make these turns successfully. States 5 & 6 are important because obstacle 1 represents a scenario where the robot experiences a θ_{err} between 0° & 20° and it needs to learn to decrease it's ω_z in increments to drive straight. As you can see, both algorithms allows it to learn that turning at a $\omega_z = 15^\circ/\text{second}$ when in state 5 and $\omega_z = 0^\circ/\text{second}$ when in state 6 would allow it to successfully drive straight.

For a demo of each algorithm's learned model, please see the demo videos at the links provided below:

- Q-Learning: `https://youtu.be/LtyQdxRFLvYM`
- SARSA: `https://youtu.be/mWw8RORiZA8`

IV. CONCLUSIONS

As shown above, the Q-learning and SARSA reinforcement learning algorithms both prove to be successful at training a robot to successfully navigate the five obstacles shown in Figure 2 while following a wall. If you would like for the robot to learn the optimal wall-following policy, choosing Q-learning would be the better of the two. This is because it uses the maximum Q-value for the current state when rewarding the robot in each epoch. However, if you would like the robot to explore a variety of near optimal solutions, selecting SARSA would be the better choice. This is because it learns a near-optimal policy while continuing to explore for a more optimal policy. In a real world scenario, forcing the optimal learned policy using Q-learning could become troublesome when the robot encounters an obstacle it has never seen before. This is because the robot has only chooses the best action for the obstacles it has previously learned. If the best action for the learned obstacle most similar to the new obstacle is not sufficient, it could potentially result in the robot becoming damaged. As a result, it is probably better to use the SARSA algorithm in real world situations where the optimal solution may consistently be changing due to new obstacles.

REFERENCES

[1] Sebastian Thrun, Mike Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, et al. Stanley: The robot that won the darpa grand challenge. *Journal of field Robotics*, 23(9):661–692, 2006.

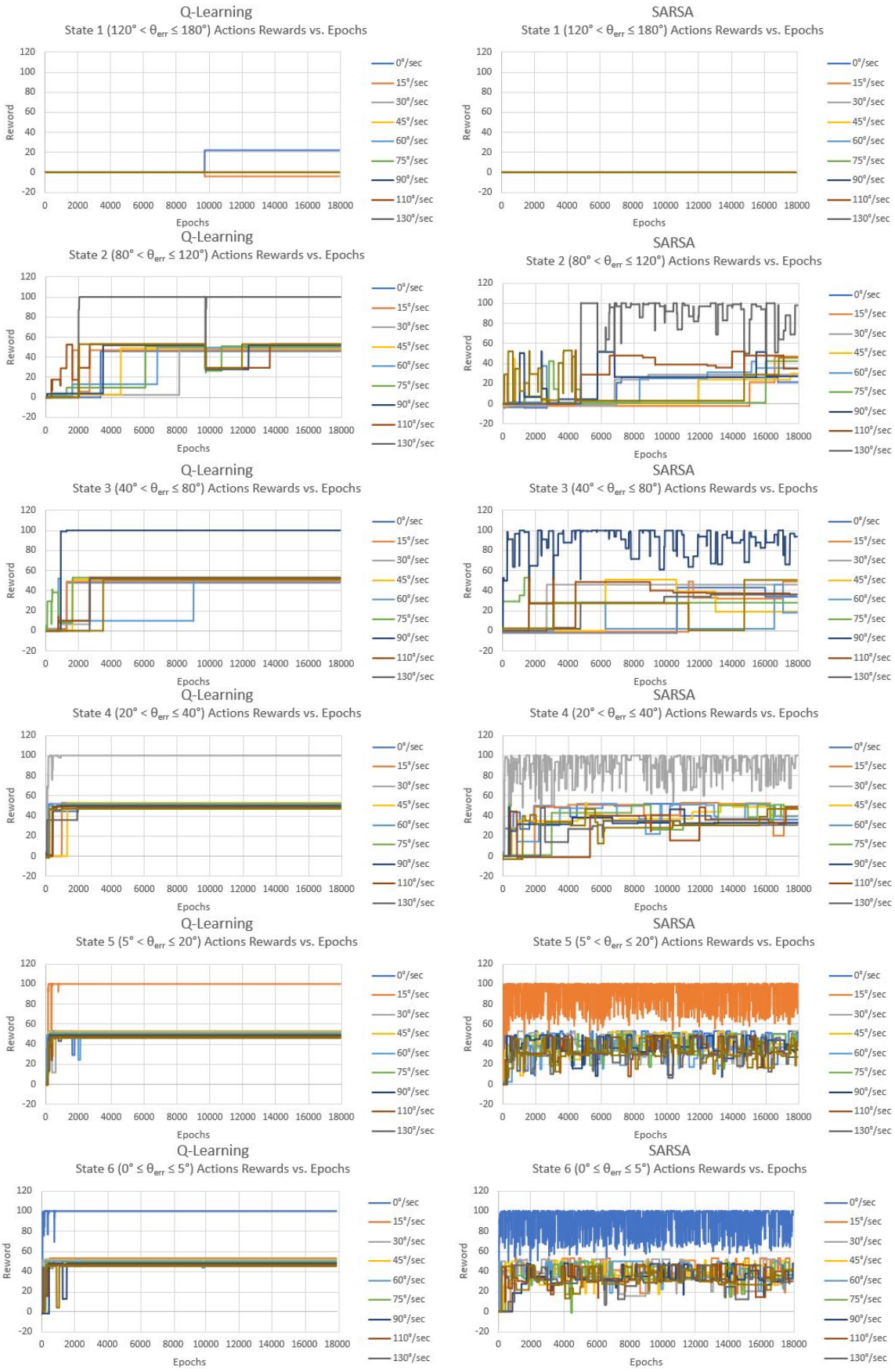

Fig. 6. Action Rewards vs. Epochs for each State using Q-Learning & SARSA