Final Project:
Reinforcement Learning-ROS Damage Recovery: Magni

By:
Abhilesh Borode
Rachel Breshears
Casey Duncan
Emerson Ham
Mohammed Aun Siddiqui

MEGN 545 Advanced Robot Control
6 May 2019

Introduction

As robot autonomy continues to improve, a growing number of robots are being deployed into real-world situations where there is a high probability of hardware damage that can prevent them from performing their designed tasks. Instead of trying to diagnose the damage itself, the robots are adapting to their damage using a trial-and-error approach called Reinforcement Learning (RL). For this Damage Recovery project, we were given the Magni Robot, shown in **Figure 1**, to implement RL using C++ and ROS to "fix" a damaged motor, wheel, or leg depending on the application. Our RL algorithm would learn to correct for "damage" using the position and orientation of the robot as inputs and outputting a prescribed motor velocity, linear in X and angular in Z. Our goals for our RL algorithm was to stay as close to the straight line as possible and increase the learning rate for the speed of convergence.

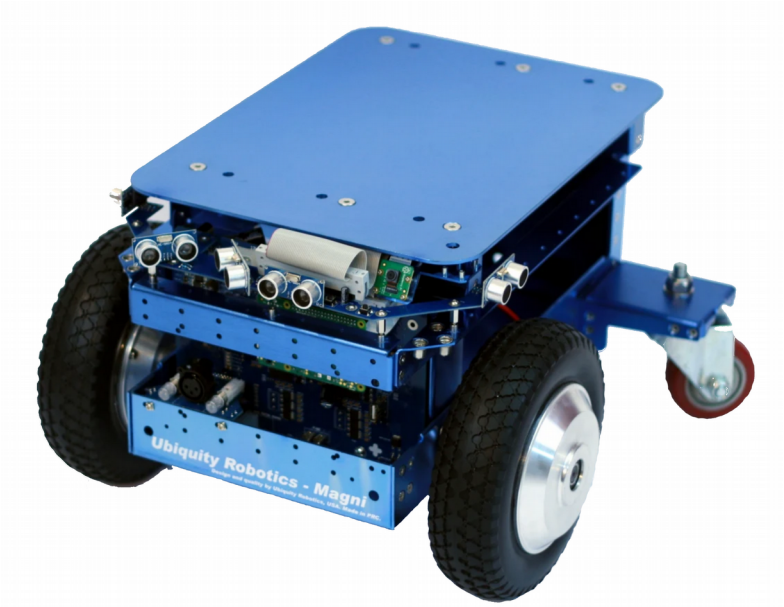

Figure 1: The Ubiquity Robotics Mangi robot used for this project

Working with ROS and C++

We used ROS as an integration platform. ROS (Robot Operating System) is a robotics middleware which allows us to integrate our C++ code with different robot components like the motors and sensors. We decided to use C++ as the programming language because it is a stronger language than its counterparts and ROS is also written in C++. The local Magni robot system consisted of a raspberry pi with ubuntu 16.04 as the operating system. All

our code is compiled in ROS Kinetic distro.

ROS communicates with different components of the robot through different topics as shown in **Figure 2**. We used the /cmd_vel topic to publish the robot velocities that needed to be executed for every action. The /cmd_vel topic has twists as the message type with 3 linear and 3 angular components. Initially we used the /odom topic to read the odometry values. The /odom topic has pose as the message types with 3 linear components and quaternions as the angular components. We decided to move to tf transforms which directly gave us the yaw angle of the robot thus not requiring us to convert the quaternions received from the /odom topic to euler angles.

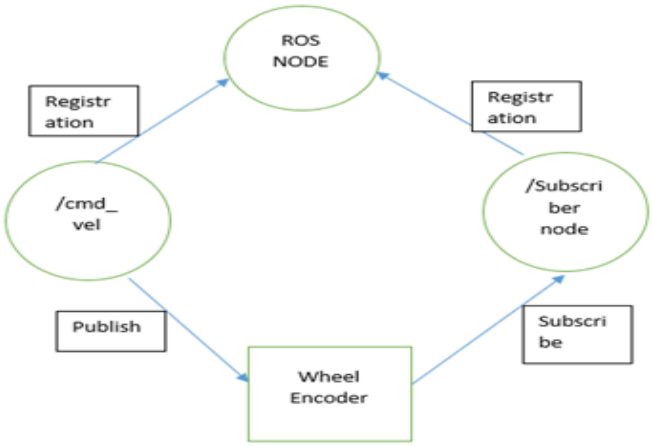

Figure 2: ROS nodes used by the algorithm

Problems with the Robot (Troubleshooting)

Our original problem statement was to drive the damaged robot in a straight line. However after working with the Magni Robot, we found out that the there was an inherent difference in power between both the motors of the undamaged robot which caused the robot to deviate from a straight line.

Apart from the motor problems we also found out that the robot odometry was also poorly calibrated. That is when the undamaged robot was made to go forward it followed a curve like trajectory due to the inherent motor defects. However the IMU on the robot still reported that the robot is moving in a straight line. This was a severe problem as the IMU is the only way we can locate the position of the robot and we had to use the odometry values

as feedback in our Q-learning algorithm.

Lastly when we tried to handicap the robot by changing the power ratio to the motors using the rqt plugins we observed that this also handicapped the odometry topic of the robot which gave us erroneous robot positions.

Taking all of these robot problems into consideration, we firstly wrote our own piece of code that handicapped robot. We did this by offsetting angular velocity of the robot. Now we treat the yellow curved line as shown in **Figure 3** as the ground truth for a straight line, since that's what the odometry topic tells us is a straight line and we implement our Q-learning algorithm with the objective to follow this curve with the handicap.

Figure 3: The “straight line” as determined by poorly calibrated odometry

Basic Reinforcement Learning: Training

Combined Angle

The way our algorithm was set-up was similar to the grid-world homework problem. The input is the position and orientation, and the output was both angular and linear velocities. As both the orientation and position are important for the robot to train, it increases the complexity by making the Q-Matrix three dimensional, where the three axes being for the position (deviation from the trajectory), orientation, and possible actions. There would be too many Q values to explore and training would take a lot of time with

this setup. Instead, we decided to use the Stanley Method [2] that combines both position and orientation into one angle as shown in **Figure 4**. This angle will be referred to as the "combined angle" for the remainder of the report.

The combined angle can be calculated as:

$$Combined\,Angle = \arctan 2(\Delta y, 0.5) - \theta z$$

Where Δy is the offset/deviation from the trajectory, 0.5 is the forward distance along the trajectory towards which the combined angle points, and θz is the orientation angle of the robot in the world frame.

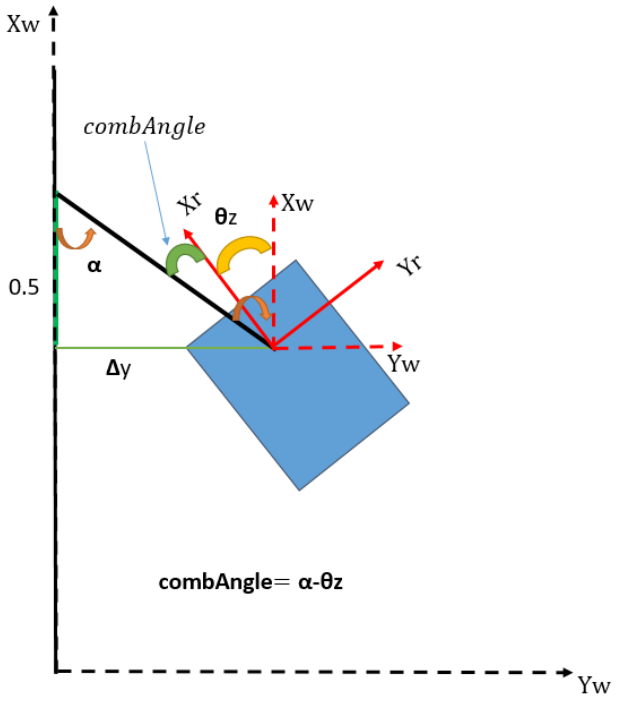

Figure 4: Diagram showing the calculation of the combined angle

States

The Combined Angle was then mapped out to the corresponding state, depending on its value. The way that our bins for our states were defined as:

- Less than -20°
- -20° to -15°
- -15° to -10°
- 0° to 5°
- 5° to 10°
- 10° to 15°
- 15° to 20°

- -10° to -5°
- -5° to 0°
- Greater than 20°

This gives us 10 possible states where the Combined Angle can be in.

Actions

Similarly, our possible angular velocities/actions were -50,-25,-15,-5,0,5,15,25, and 50 (in degrees/sec). The actions are unevenly spaced out because we wanted the robot to have enough angular velocity to be able to traverse through multiple states with just a single action or have a small angular action so that it does not leave the state that it is currently in.

Q-Matrix and Learning Rate

The initial Q-Matrix was initialized to zeros and was of size 10 (possible states) x 9 (possible actions). The learning rate was initialized to 0.5 and then manually decreased periodically after time until it reached 0.1. After that, all other training iterations were done at a learning rate of 0.1. This was done so that Q-Matrix at the beginning does not know what the optimum action is, as we start at a random point in a solution space. We first take big leaps to reach towards the minima and as we reach closer to it, we want to take a smaller step size to reduce noise around the minima. If the learning rate does not decay and starts with a very small value, training takes a lot of iterations to converge. An analogy to this can be made with the game of golf. When the ball is far from the hole, the player hits it very hard to get is as close to the hole as possible. Later when he is near the hole he hits it softer to be more accurate.

Reward

The reward for the action was calculated by this equation:

$$reward = -\left|\left(4.5 - newStateNumber\right)\right|$$

Ideally, each action should lead to a new state close to the trajectory. This reward function is designed to reward actions that do that. Since the new state number can range from zero to nine, and the optimal state is on the line between states 4 and 5, the reward is highest when the robot is properly oriented and lowest when outside of that.

Algorithm

All of these features: states, actions, Q-matrices, etc. all combine into our algorithm. The logic flow is as follows:

1) *Set the reward, learning rate, actions and states*
2) *Initialize Q matrix to 0 if first iteration or read them from the file that saves the Q matrix.*
3) *Till bot reaches 4m:*
    a) *Calculate old state*
    b) *If in extreme state more than 3 times push towards states in between -20 to 20.*
        i) *And do not calculate reward or Q-values.*
    c) *If not in an extreme state more than 3 times*
        i) *Select random action*
        ii) *Calculate new state*
        iii) *Compute reward*
        iv) *Compute reward Q-value using Bellman Equation and update Q-matrix*
4) *Write Q-Matrix to a file for the next iteration*

While training we would perform one implementation for every 2 training iterations so that we could check performance. While implementing we only choose the best actions that were learned from the Q-Matrix. The only difference in this algorithm as compared to the homework is that when the robot is outside the range of -20 degrees to 20 degrees, for more than 3 random actions. We force the robot towards the trajectory so that it can resume useful training since random actions in terrible directions are not useful for training our algorithm.

Basic Reinforcement Learning: Results

Using the basic reinforcement learning method described above proved to be successful, however only some of the time and would also take a really long time for the Q-values to converge. We ran into several issues that would cause failure. One issue found was our system getting stuck in local optima. Bad Q-values would stay in our matrix for a long time because of random updates to the matrix. The robot doesn't know these Q-values are bad, which leads to optimal paths including S-curves or circling around in donuts (Reference **Figure 5**). Another issue encountered was that the noise from trials was just as large as the variance between the Q-values. This noise

would cause our results to be very noisy even once our system had converged.

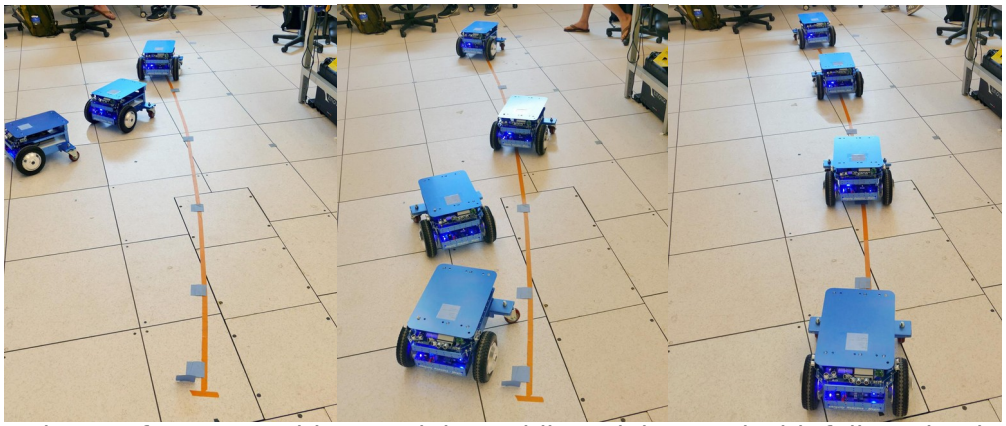

Figure 5: Robot performance with no training, while training, and with fully trained algorithm

Optimizations

Although we had successfully implemented reinforcement learning on the Magni Robot, one of the goals of this project was to optimize the robot to maximize the learning rate of the robot. Several changes were implemented to achieve this goal. The first and most important was creating a smarter exploration algorithm. Instead of just taking random actions every time the robot trained, we updated the robot to keep track of how many times each Q-value had been learned so that the robot could intentionally take the action that had been learned the least. This is beneficial because it helps remove the possibility of poorly learned actions randomly being better than the actual best action. In order to do this, we had to add an additional matrix for keeping track of the number of times each Q-value had been learned, figure out which actions had been learned the least, then choose a random one of those to implement.

Another optimization we implemented was automatically calculating the learning rate as a function of the number of times the Q-value in question had been learned. Originally we selected the learning rate manually, however changing it to this new method is beneficial because it allows us to program it to learn quickly in the beginning so that it approaches the correct value faster, then gradually slow down the learning rate over time so that the noise gradually goes away as well, allowing the Q-value to converge accurately to the correct value of the Q-value with minimal noise. This also allows us to tune the balance between converging faster and overwriting bad values faster, helping us further optimize the reinforcement learning

algorithm.

One other part of the algorithm we were able to optimize was the sizes of the actions and states. Since the robot works by taking actions and seeing how they affect the states, it is important to have them set up so that the actions are able to have a reasonable effect on the state of the robot. If the actions are too small they will not have much effect on the state of the robot, making the different Q-values for each action too similar, which makes it so we have to train much longer to be able to properly observe the difference between them. Conversely if the actions are too large, it will be much harder for the robot to smoothly follow the desired trajectory. Correctly balancing the size of the actions and states maximizes the difference between the Q-values while also allowing the robot to follow the trajectory smoother.

The final optimization we considered was the rate of discounting to the reinforcement algorithm. The theory behind this is that a properly optimized discount rate would add the Q-value of the state the action ends up in as a weight in the Q-value of the previous state and action. This is useful because a poorly learned Q-value for one state and action can be offset by the Q-value of the state the robot ends up in. In other words even though a Q-value might look optimal, if it ends up in a state that’s not as useful the Q-value in question would look less optimal. This helps reduce the effect of poorly learned Q-values, in theory leading to better decision making by the reinforcement algorithm. The new logic flowchart, including optimizations can be seen in **Figure 6**.

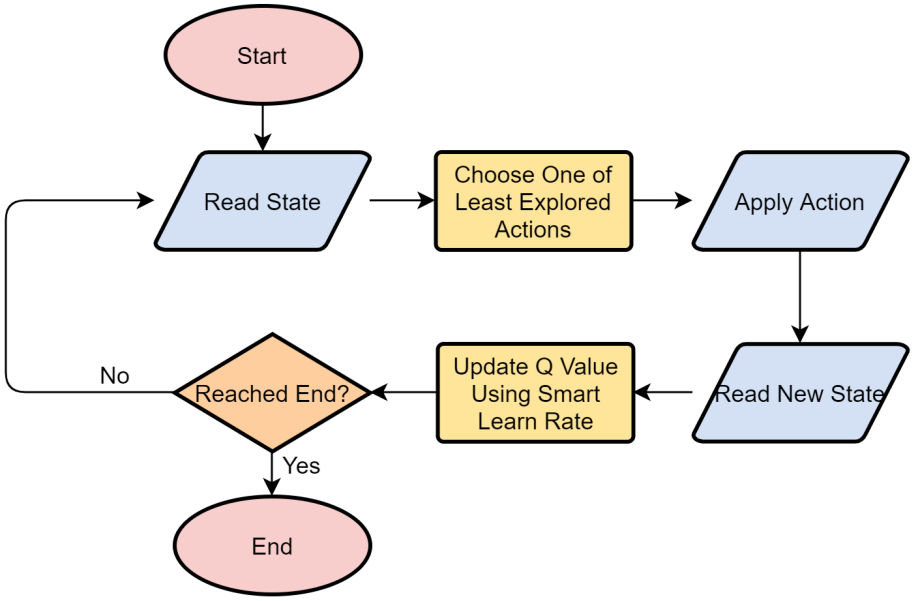

Figure 6: Logic flow of the optimized reinforcement learning algorithm

Optimizations: Results

Using the *Basic RL Reward* method outlined above, we performed the required training to create three different Q-matrices that allowed the robot to smoothly follow the desired trajectory. As seen in **Figure 7**, the average number of trainings required to produce each learned Q-matrix was 6 trainings. Similarly, we followed the same method using the *Optimized RL Reward* method and the average number of trainings required to produce each learned Q-matrix was slightly under 4 trainings. Overall, the *Optimized RL Reward* method allowed the robot to learn to follow the desired trajectory in 39% less trainings than the *Basic RL Reward* method. Additionally, the robot followed the trajectory in smoother movements using the *Optimized RL Reward* method.

Even though both methods allowed the robot to learn to follow the desired trajectory, we had issues with each method when training more than what is shown in the **Figure 7** plot. When continuing to train, the learned Q-matrix would “forget” the optimal path and no longer follow the desired trajectory. Due to time constraints, we were not able to diagnose why this was the case but are confident we would have been able to fix this issue given more training data.

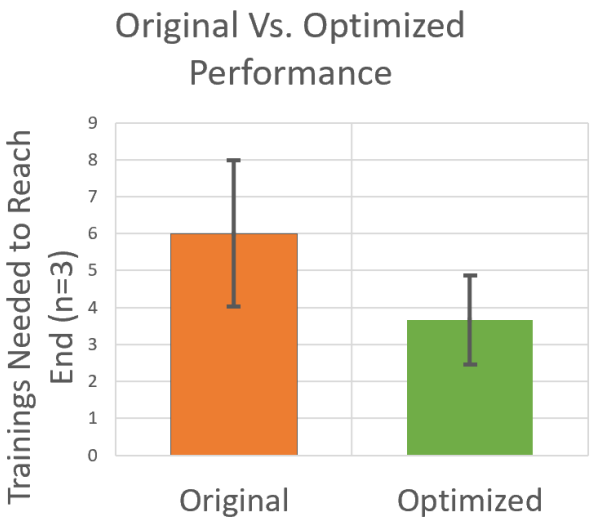

Figure 7: Comparison of performance of original and optimized algorithms

Conclusion

Using both the *Basic RL Reward* and the *Optimized RL Reward* methods, the handicapped robot was successfully able to follow the trajectory regardless of the troubleshooting issues we encountered with the Magni's hardware. The robot learned the optimal actions within a specific state and was fully capable of correcting its biased movements due to the damages motors. Even though the RL algorithm was able to teach the robot to follow the desired trajectory, our method was not as robust as hoped since the learned Q-matrix would "forget" the optimal path and no longer follow the desired trajectory. If we were to improve the RL algorithm in the future, we would change the algorithm to intentionally choose actions that arrive in states we haven't learned yet instead of randomly choosing actions that have not been performed. Also, we would try to diagnose the "forgetting" problem by collecting more data to see which optimizations were most effective.

References:

[1]https://github.com/UbiquityRobotics/magni_robot
[2]https://www.ri.cmu.edu/pub_files/2009/2/Automatic_Steering_Methods_for_Autonomous_Automobile_Path_Tracking.pdf
[3] https://mpatacchiola.github.io/blog/2017/08/14/dissecting-reinforcement-learning-6.html