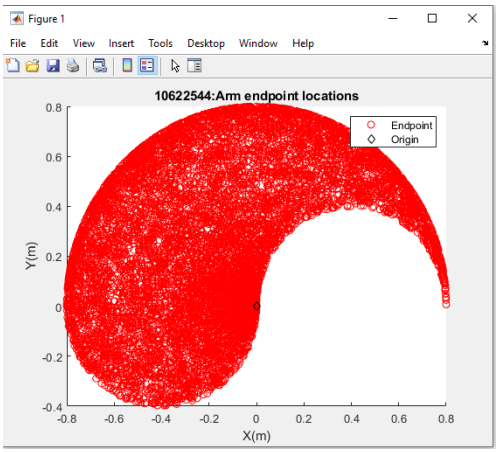
1. ROCO351: 10622544

2. DATA GENERATION

I Started by defining arrays containing the provided values and a set of randomized theta values. I then used min/max to confirm that my randomised theta values were within the specified range.

I then ran these values through the provided forward kinematics function and saved the results to a .mat file.

I then plotted the endpoint location from the forward kinematics function output p2



From this graph we can infer that this simulated arm has an effective range of ±0.8m in the X dimension and between -0.4 and 0.8 in the Y dimension. In the X dimension however, it appears unable to reach most of the positive side.

3. IMPLEMENT A 2-LAYER NEURAL NETWORK

3.1 IMPLEMENTING 2-LAYER NETWORK TRAINING

I started by defining all the values required for the 2 layer sigmoid I included the variable HiddenUnits which when changed increases the number of rows in Weight1 and columns in Weight2, adding additional hidden units.

I then implemented feedforward pass starting with a sigmoid hidden layer and then a linear output layer. I then implemented back propagation calculating the delta and error of the output and using them to update the weights.

3.2 TRAIN NETWORK FOR INVERSE KINEMATICS

I started by importing the armdata from ArmData.mat and assigning its theta values to the target and P2 value to the InputX. I then augmented the Input data with an extra row of 1's.
I also added an extra couple of matrices to track the error progression and added a line to the SGD calculating the sum squared error and a line to save the average error of each iteration to the errorprogression matrix which I then plotted see figure 2 below. I also implemented a couple of timers using Tic/Toc and displayed both the current iteration number and how long it took in the console, this acts as a kind of progress bar which I used to help choose a sensible maximum number of iterations.

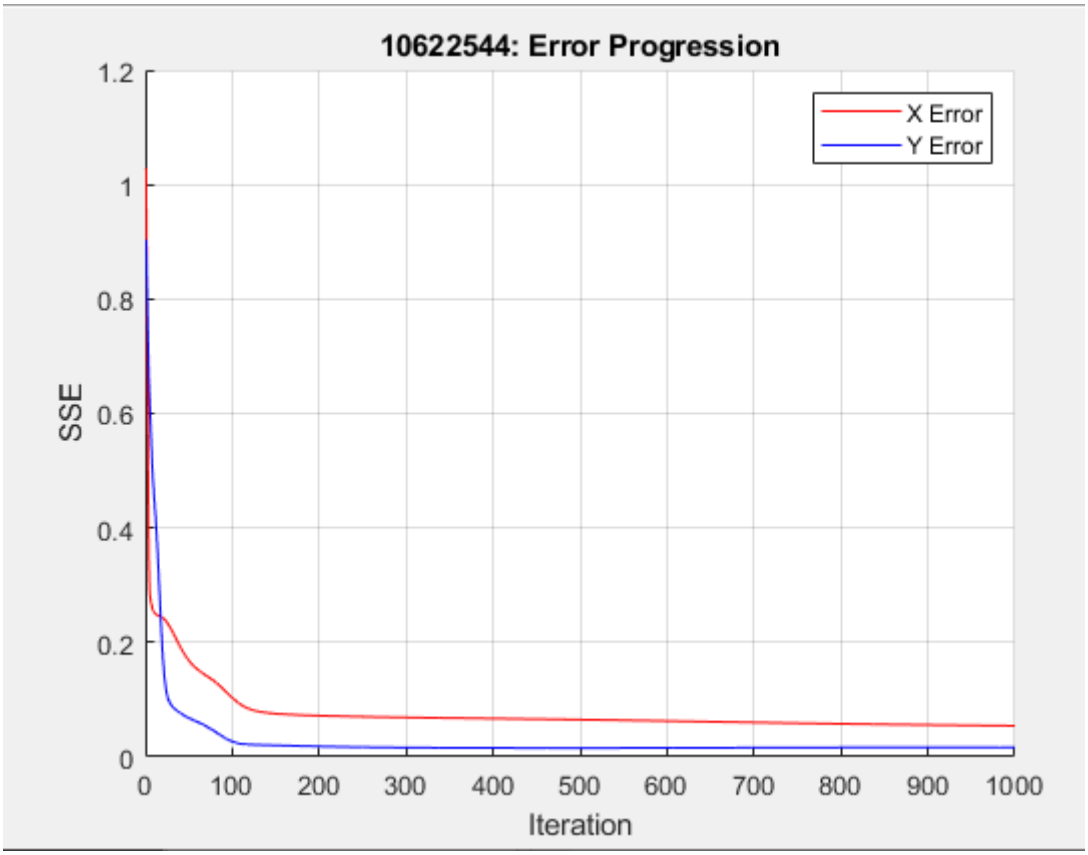


Figure 2. Error progression, This graph plotted using data from a trial using 8 hidden units a learning rate of 0.001 and 1000 iterations.

3.3 TEST AND REFINE THE INVERSE MODEL

I applied the trained weights to the input data by using the forward pass from my training algorithm and stored the results in a matrix. I then used these new inverse theta values as the input for the same forward kinematics function used when generating the arm data and plotted both the inverse theta values and the inverse endpoint. I also plotted the random theta values and the endpoints generated using them first on their own and then on plots with both the randomised and the inverse generated data. These plots make it easy to see how accurate the new inverse data is, especially the overlapping plots.

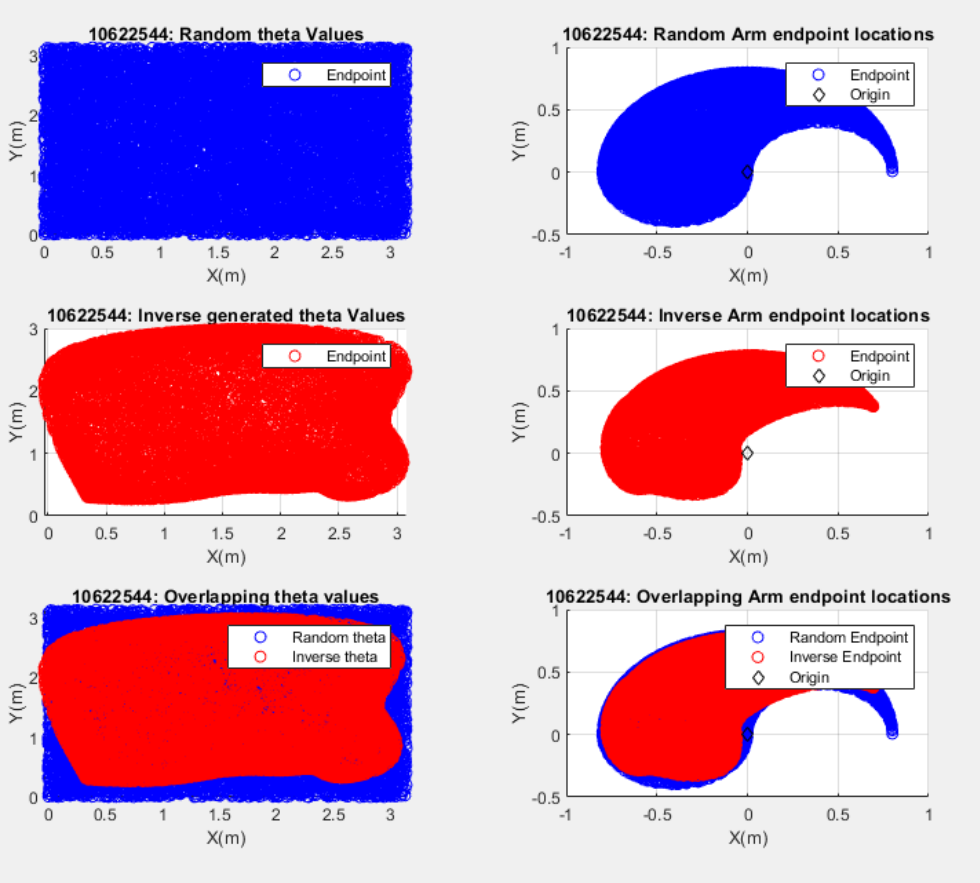


Figure 3. The theta and endpoint graphs from the same trial as figure 2.

I also ran trials with various different parameters and from these trials we can see how each parameter affects the result. Looking at figure 4 which is from a trial with only 4 hidden units we can infer that hidden units are essential for accurately estimating the inverse kinematics and that when you don't have enough the results will have a strange shape and unusual outliers. However adding too many hidden units can also distort the shape of

your results I found during testing that the optimal results were gained from the parameters used in figure.

From Figure 5 where the learning rate was higher we can see that a higher learning rate increases the accuracy of your results without having to increase the number of iterations but also creates more extreme outliers.

And finally, from figure 6 we can infer that increasing the iterations increases accuracy but without creating as many extreme outliers as a higher learning rate, at the expense of taking much longer to run.

I also ran the algorithm using a new set of training data as seen in figure 7.

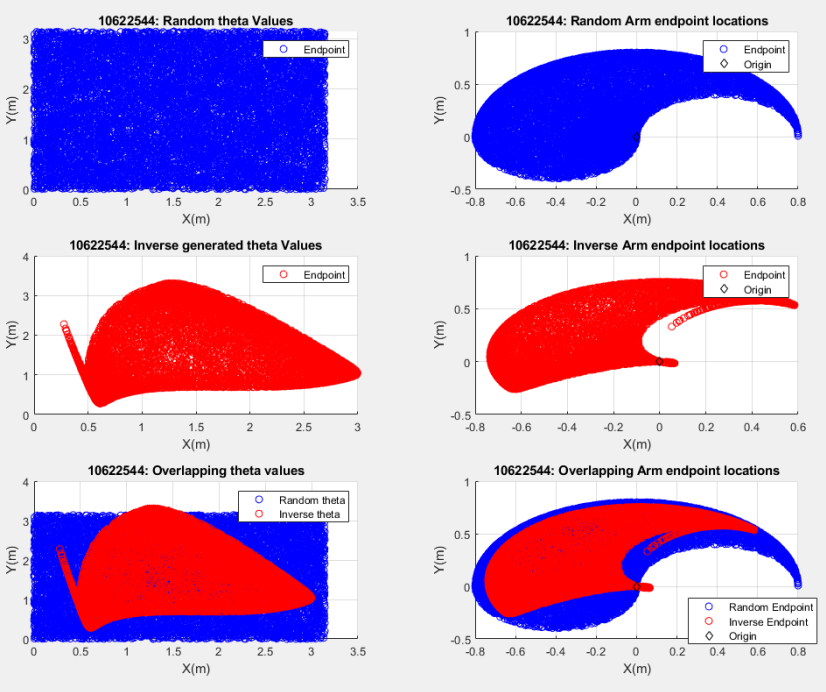


Figure 4. Output graphs of a trial with a learning rate of 0.001, 1000 iterations and only 4 hidden units.

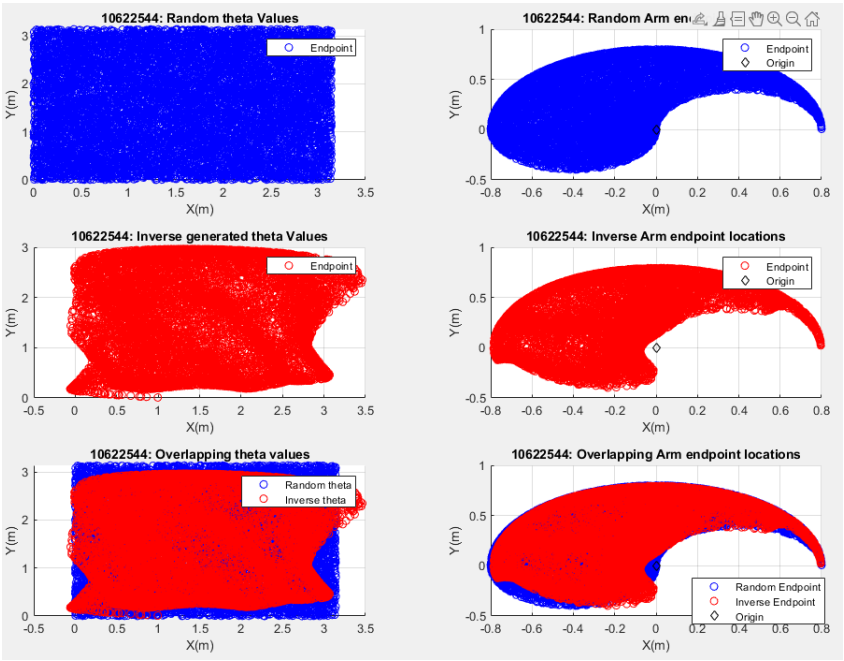


Figure 5. Output graphs of a trial with a learning rate of 0.01, 1000 iterations and 8 hidden units.

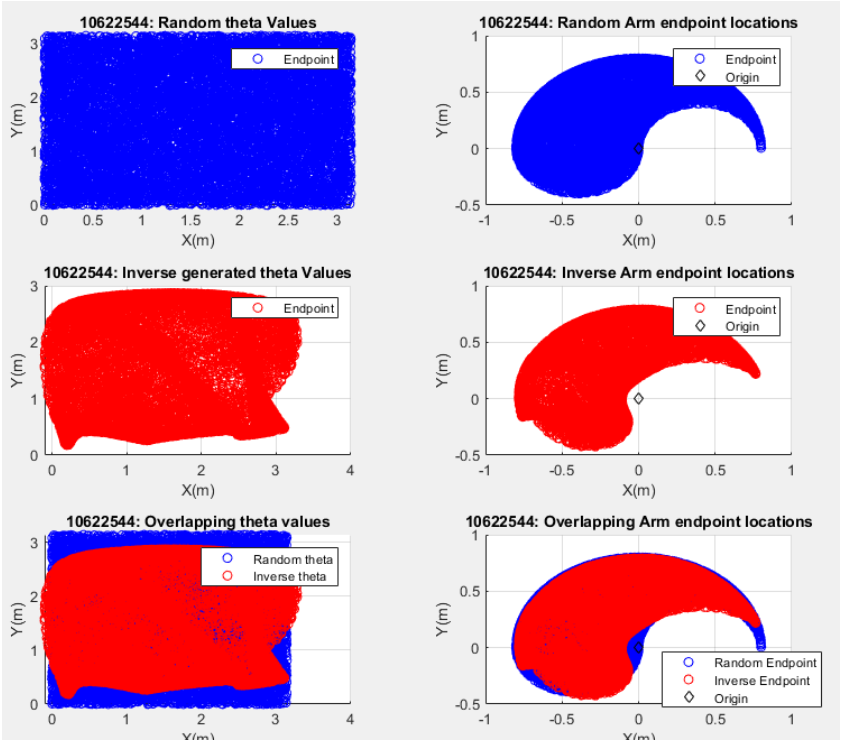


Figure 6. Output graphs of a trial with a learning rate of 0.001, 2000 iterations and 8 hidden units.

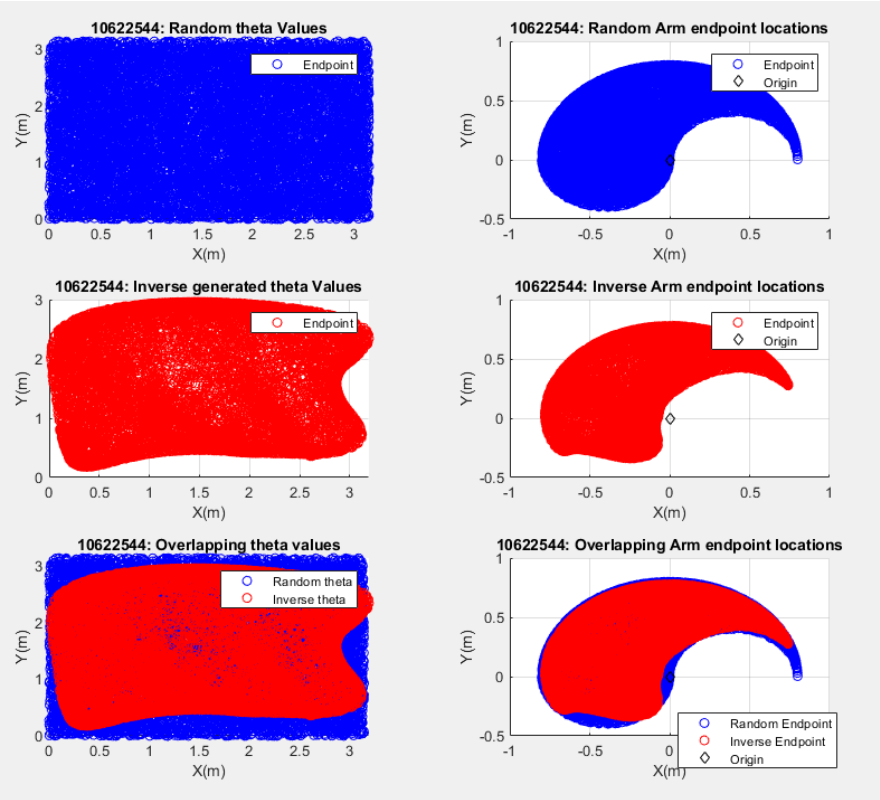


Figure 7. Output graphs of a trial using new training data with a learning rate of 0.001, 1000 iterations and 8 hidden units.

4. PATH THROUGH A MAZE

I started by setting up the maze adding coordinates of the blocked states to f.blocked locations, and adding various other definitions.

4.1 RANDOM START STATE

To select a random start state I start by selecting a random number between 1 and 120, I then round it to the nearest whole number. Next a while loop checks if the random starting state is equal to 121 or if it's a member of f.BlockedID to make sure it is neither the end state or a blocked state. The check to make sure it is not 121 isn't really needed as the random number is between 1 and 120 but its always a good idea to be careful.

Next I used a for loop in the Main_P3 file to generate and store 1000 starting states which I plotted in a histogram to confirm that the random start states were being generated correctly. This histogram figure 8 shows that for the most part the starting states are evenly distributed amongst the non blocked states and that the blocked states are being re-selected as intended.

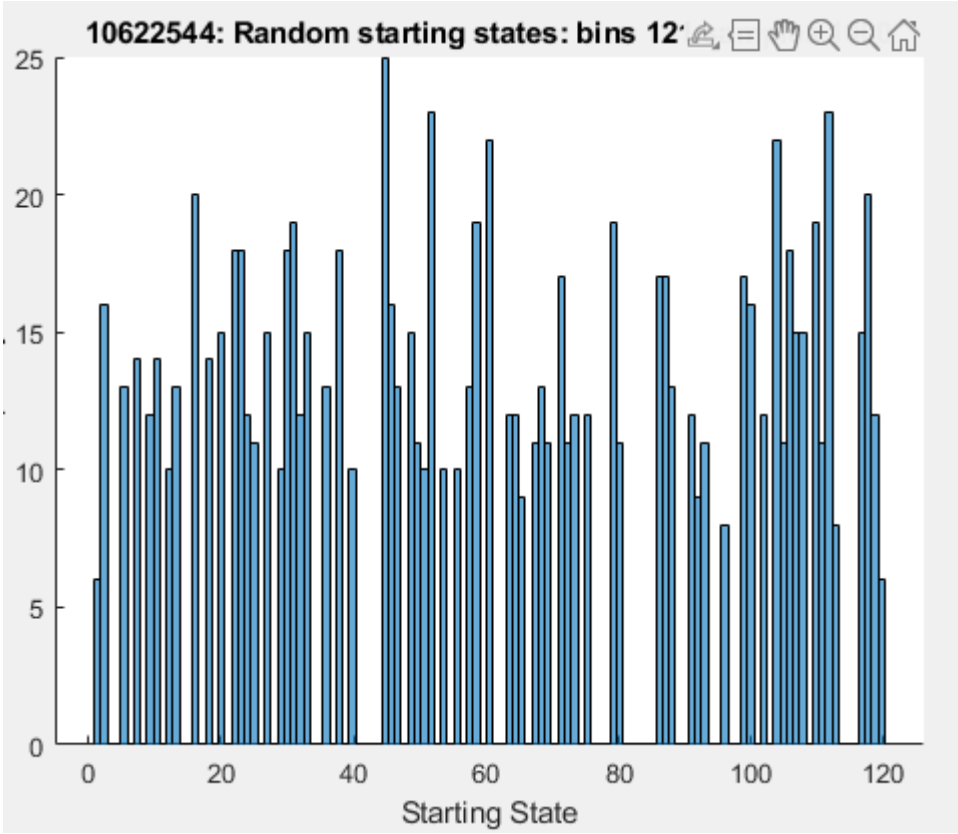


Figure 8. histogram showing the frequency of each random starting state.

4.2 BUILD A REWARD FUNCTION

The Second step was building the reward function. This was a simple task, achieved by simply creating a 121 by 4 reward matrix of zeros and manually setting the reward for the two state/action combinations which result in the next state being the goal state to 10. This means that whenever the reward function is called it produces an output of zero, unless the current state and action would result in the next state being the goal.

4.3 GENERATE THE TRANSITION MATRIX

To create the transition matrix I used two for loops. The first is used for setup and checked which states were blocked and set those that were to 0 in the f.openstate matrix, It also calculated the current state -1 divided by 11 rounded the result and then calculated this value mod11 and stored the result in a matrix. These values are calculated in the first loop as otherwise they are often called out of sequence in the second loop, for example checking if the next state is blocked or not.

The second loop checks each state one at a time checking each action for the current state to make sure that it wouldn't lead outside the grid or into a blocked state.
Actions 1 and 3 were fairly simple, the if else statement just checks if the current state plus or minus 11 is a member of the open state matrix if it is then the transition matrix column 1/3 for the row representing the current state is set to the current state ±11 otherwise its set to the current state.
Actions 2 and 4 were more complicated as I had to check if the current state ±1 was on the same row, to do this I used the modulo calculated previously and compared it to the modulo for the next state, this works because if you divide the state by the number of columns and then mod that value by the number of rows, then the result will be the row that state is on, the exception is the final column where it will think its on the next row up, so to compensate for this I calculated these values for current state -1.

4.4 INITIALIZE Q-VALUES

To initialize the Q-values matrix I simply generated a 121 by 4 matrix of random values between 0.01 and 0.1. I also created the variables Qmin and Qmax which simply find the maximum and minimum Q values, these variables were used to confirm that the randomly generated values were within the specified range.
I used this method as it is incredibly simple requiring only a single line of code.

4.5 IMPLEMENT THE Q-LEARNING ALGORITHM

To Implement the Q-Learning Algorithm I created 4 loops. Two for loops an outer loop which ran until the requested number of trials had been completed, and the inner loop which ran until the requested number of episodes had been completed.
Within the Inner for loop I ran two while loops, the first ran until the state was equal to the goal, the second ran while the state was equal to the goal. The first while loop ran the greedy selection function to choose the next state, the Q-Update function to update the Q-Values and the transition matrix function to update the state lastly it added 1 to a step counter. Once the state is equal to 121 the first while loop ends and the second one starts, this second loop generates a new starting state, stores the step count, resets the step count and sets the state equal to the new starting state, this exits the while loop and starts the next loop of the episode loop.

4.6 RUN Q-LEARNING

I ran 100 trials of 1000 episodes using the specified parameters then calculated the mean and standard deviation of each episode, I then plotted the mean number of steps in each trial with error bars for the standard deviation.
Interestingly unlike the example figure my results only had a large mean/standard deviation on the first trial, the rest of them were incredibly similar

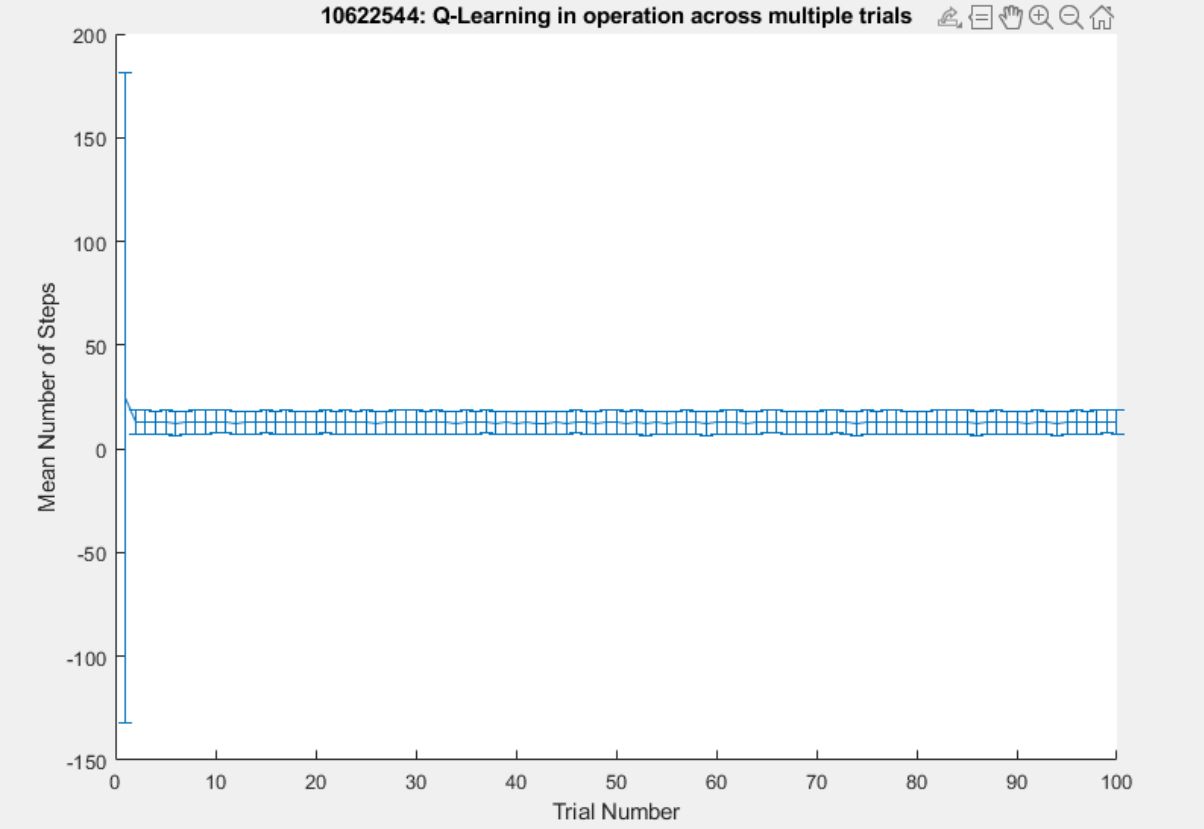


Figure 9. error bar plot of the mean number of steps in each trial

4.7 EXPLOITATION OF Q-VALUES

For my Q-Exploitation I simply set the starting state to one as instructed and copied my Q-Learning “state does not equal 121” while loop replacing Q-Update and GreedySelect with a modified GreedySelect function that always uses the Q-Values rather than taking random actions 90% of the time. On each loop I stored the state visited for plotting and plotted the optimal path through the maze as seen in figure 10.

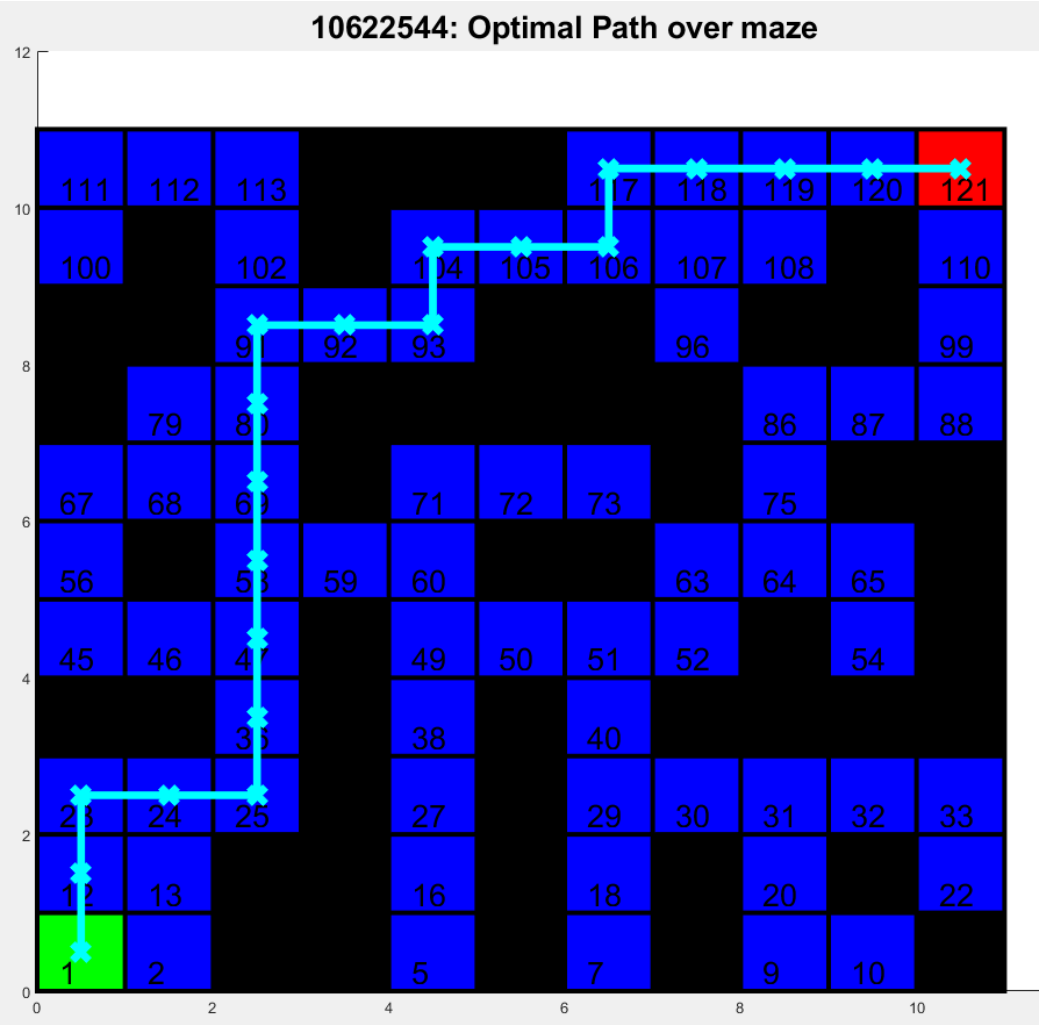


Figure 10. The optimal path through the maze plotted over the maze

5. MOVE ARM ENDPOINT THROUGH MAZE

5.1. GENERATE KINEMATIC CONTROL OF A REVOLUTE ARM

I started by loading all of the needed matrices and files from previous tasks and scaling the limits of the maze to fit within the arm workspace, I plotted the arm workspace over the maze to help with this as seen in figure 11.
I then scaled and augmented the optimal path coordinates from P3 and ran them through the trained weights loop from P2. I then ran the output through the forward kinematics function and plotted the output as seen in figure 12. While doing this I experimented with various parameters for training the neural network to try and produce as accurate an output as possible, in the end I settled on 1500 iterations 40 hidden units and a learning rate of 0.001 as this provided the best shape/output without making each test take too long. Despite my experimentations I the

generated path is still wonky and imperfect as can be seen in figure 12, I believe the best way to solve this might be to add an additional hidden layer to the neural network as I believe I have hit the limit of what can be achieved using this network without significantly increasing the run time.

5.2. ANIMATED REVOLUTE ARM MOVEMENT

I animated the arm moving through the maze using animatedline and addpoints, animation is linked below.

https://drive.google.com/file/d/1aWEyH8sc1SwBkGD-ERcHCZ8VhIBwEm73/view?usp=sharing

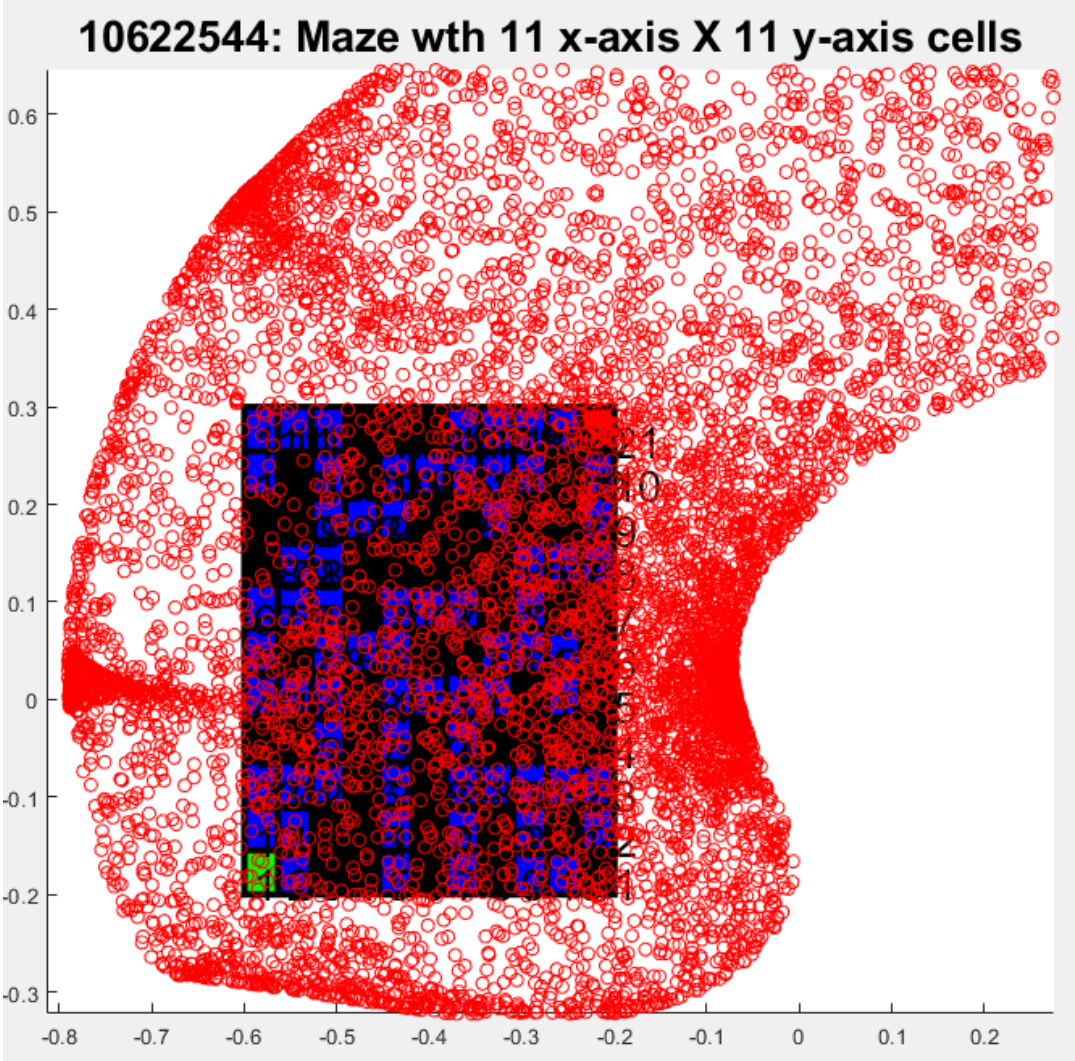


Figure 11. The Arm workspace plotted over the scaled maze

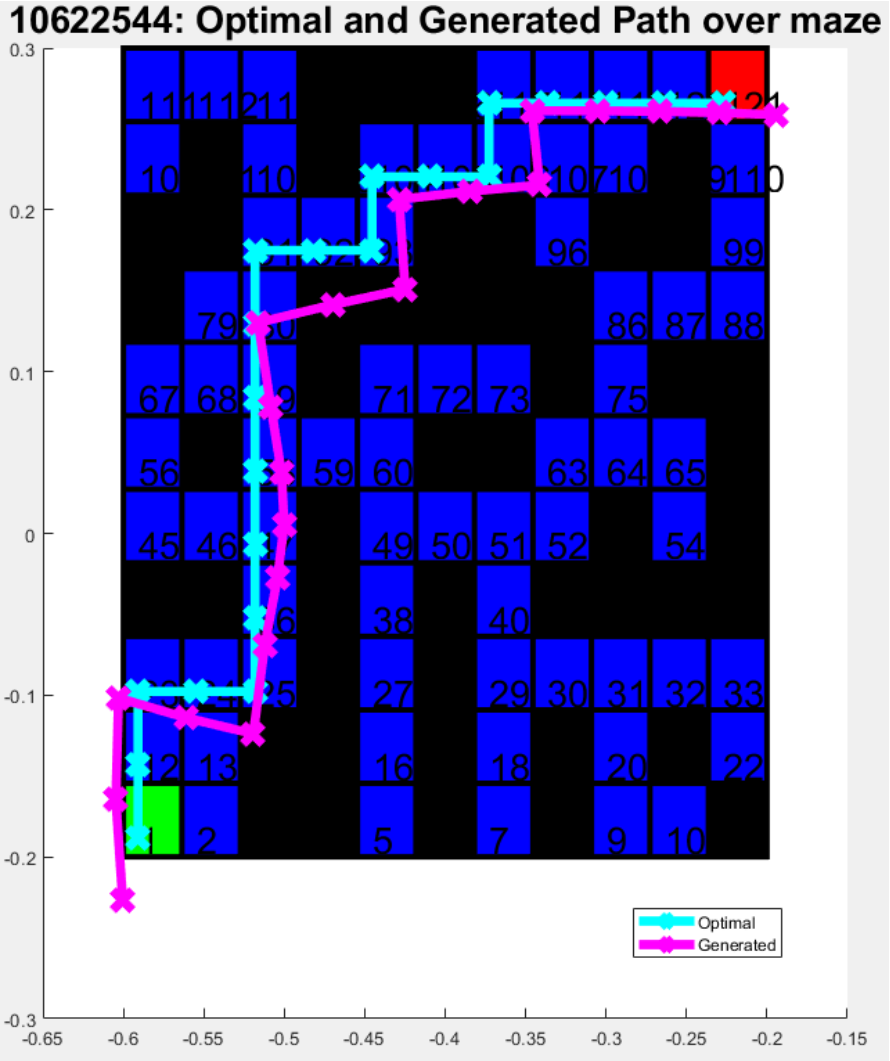


Figure 12. The optimal and generated paths through the maze