AINT351Z Machine Learning - 2022 Coursework
Student Number: 10611816

1. Training Data Generation

1.1 Display workspace of the revolute arm [3 Marks]

Set the arm lengths to 0.4m and the arm base origin to (0, 0).

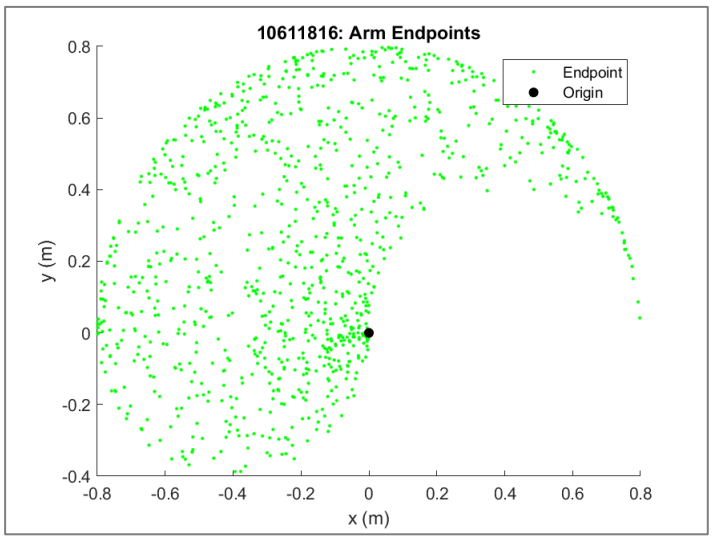
```
armLen = [0.4 0.4];
origin = [0 0];
```

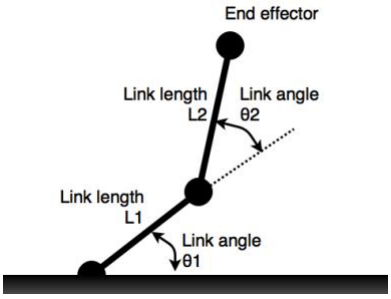
Generate two separate sets of 1000 uniformly distributed set of angle data points in the array theta over the range 0 to p radians using the MATLAB rand command and run the RevoluteForwardKinematics2D function with the specified parameters to generate the corresponding endpoint locations. One set of data will later be used to train a neural network and the other to test it.

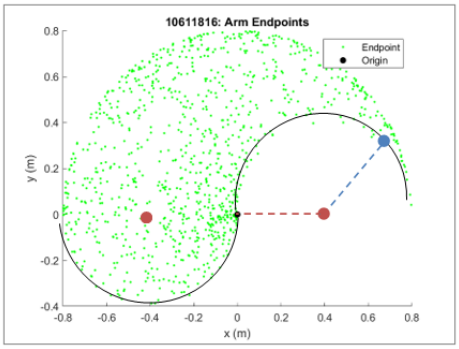
```
samples = 1000;
ang = [0 pi];

% Generate Datasets
% y = offset + range * rand(array size)
theta1 = ang(1) + (ang(2)-ang(1)).*rand(2, samples);
theta2 = ang(1) + (ang(2)-ang(1)).*rand(2, samples);

% Run forward kinematics function on both datasets
[P1_train, P2_train] = RevoluteForwardKinematics2D(armLen, theta1, origin);
[P1_test, P2_test] = RevoluteForwardKinematics2D(armLen, theta2, origin);
```

Plot the endpoint positions and the arm origin position too.

```
plot(P2_train(1,:), P2_train(2,:), 'g.');
plot(origin, origin, 'k.', 'MarkerSize', 20);
```







Discuss the useful range of this arm.

The useful range of this arm is predominantly prescribed by the link lengths and maximum limit for link angles. The elbow of the arm will be either at [0.4, 0] or [-0.4, 0] when at the extremes of its reach. This means that because the end effector can extend up to pi radians anticlockwise from this position the two semi circles shown are formed. The top of the range is limited by the total length of the arm and so again forms a semi circle with a radius of 0.8m.

2. Implement 2-layer network

2.1 Implement 2-Layer Network Training [20 Marks]

You will first need to implement the 2-layer network feedforward pass (with nh hidden units that can be specified as a parameter, and a linear output). Given the weight matrix, this will generate the network output activation for a given input vector.

```
% specify the number of hidden nodes
hN = 11;

% specify learning rate and iterations
learning_rate = 0.0003;
iterations = 10000;
```

- **Then implement backpropagation to train all the weights.**

```
% back propogate to calculate input (lower) layer delta term
W2Hat = W2(:, 1:hN);
delta2 = (W2Hat' * delta3) .* a2 .* (1-a2);

% calculate error gradient w.r.t W1
gradW1 = delta2 * data';

% calculate error gradient w.r.t W2
gradW2 = delta3 * a2Hat';

% update W1 and W2 by moving down towards minumum error
W1 = W1 - alpha*gradW1;
W2 = W2 - alpha*gradW2;
```

- **Make sure you initialize the weights in the network appropriately before running the iterative training.**

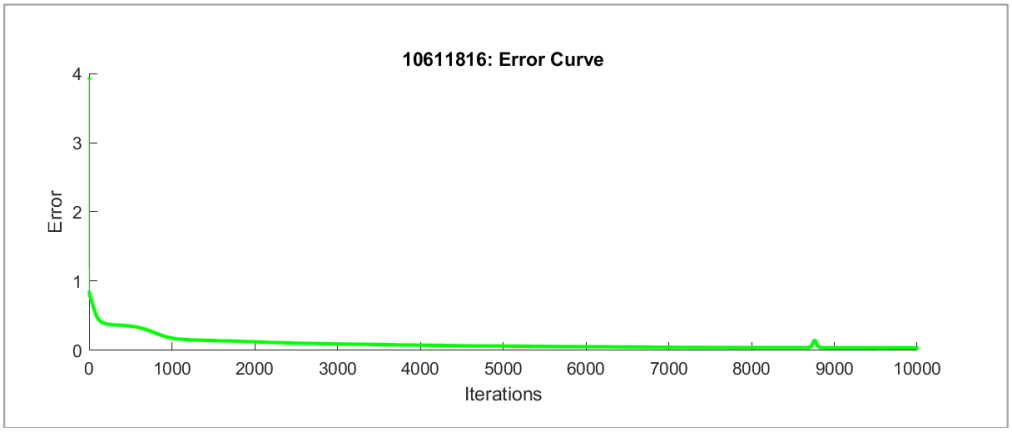
```
% initialise weights according to size of input data and number of nodes
[numRowsX, numColsX] = size(data);
W1 = rand(hN, numRowsX) - 0.5
W2 = rand(2, hN+1) - 0.5;
```

2.2 Train network inverse kinematics [10 Marks]

Now you are required to train your network to perform the inverse kinematics of the planar arm using the 2-layer neural network you developed above. Using the dataset, you generated in section 1.

- **Train your linear output 2-layer network with the augmented end effect positions as input, and the two joint angles as outputs.**

```
%specify the input data and the target data
data = [P2_train; ones(1, length(P2_train))];
target = theta1;
[output, W1, W2, e] = TrainingTwoLayerNetwork(hN, learning_rate, iterations, data, W1, W2, target);
```



- **Plot the error as training proceeds**

```
plot(e, 'g.-');
title(stuNo + "Error Curve");
xlabel("Iterations");
ylabel("Error");
```

2.3 Test and improve the inverse model [13 Marks]

Test your inverse model looking at how the 2-link arm endpoint workspace is mapped back to itself. Do so as follows:

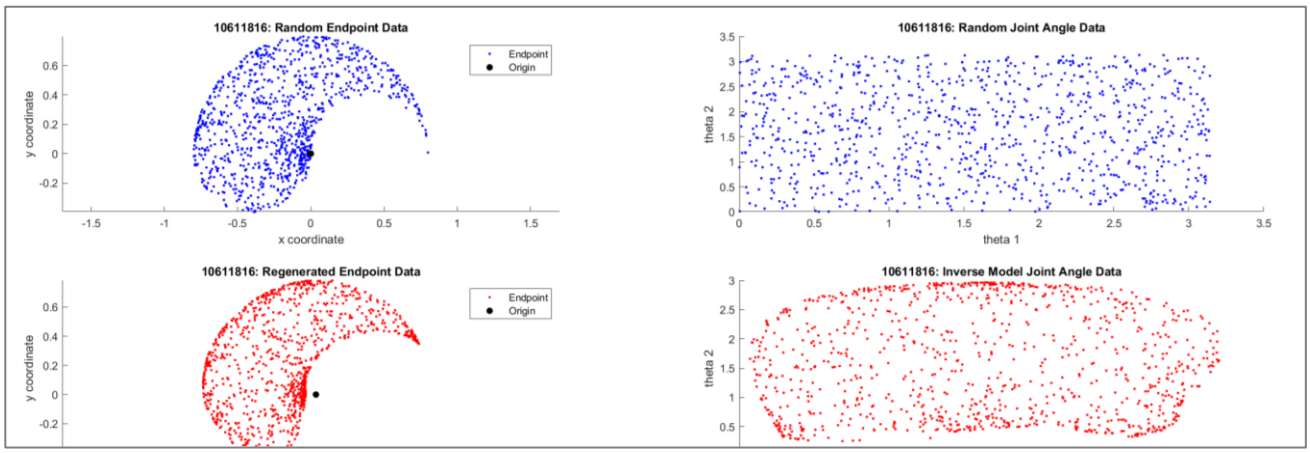
- **Using your testing dataset run a forward network pass on the position data to generate predicted arm angles.**

```
% remind ourselves of the TEST input and target data
in = theta2;
out = P2_test;

% run trained two layer network to generate predicted arm angles
test_data = [out; ones(1, length(out))];
[output_test] = TrainedTwoLayerNetwork(test_data, W1, W2); % output_test = 2x1000
```

- **Then run the forward kinematics function of these predicted angles and generate the corresponding end points**

```
% run forward kinematics to get end points
[regenerated_P1, regenerated_P2] = RevoluteForwardKinematics2D(armLen, output_test, origin);
```

- **Plot the joint angles and endpoints and compare with those you started with. You should get something like the plots shown in Fig. 3.**



- **Discuss the significance of these plots.**

These plots show at a glance how effective the training algorithm has been as we can compare random data with re-generated data. The red dots show the regenerated data and we can see that the algorithm has struggled with the original data in some specific areas, for example the datapoints next to the origin have been regenerated futher left than they should be. We can also see that the tip of the shape has also struggled to be re-generated. This is likely because there are fewer datapoints in this area and therefore less data to learn from. The inverse joint angle data is also interesting because we can see that the area is no longer sqare and there are sections of the graph which have not been reproduced. The random probability at the edges of the blue joint angles cannot be replicated by the algorithm.

- **Experiment with different numbers of hidden units, training times and learning rates.**

I tried many different variations of nodes, learning rate and even iterations, eventually settling for the values seen below. These values most regularly gave me a final error of below 0.04.

```
P2 Task 1 running...
    Hidden Nodes: 12
    Learning Rate: 0.000250
    Iterations: 12000
```

- **Also experiment with different training data sets**

Experimenting with different training sets made it difficult to

3. Path through a maze

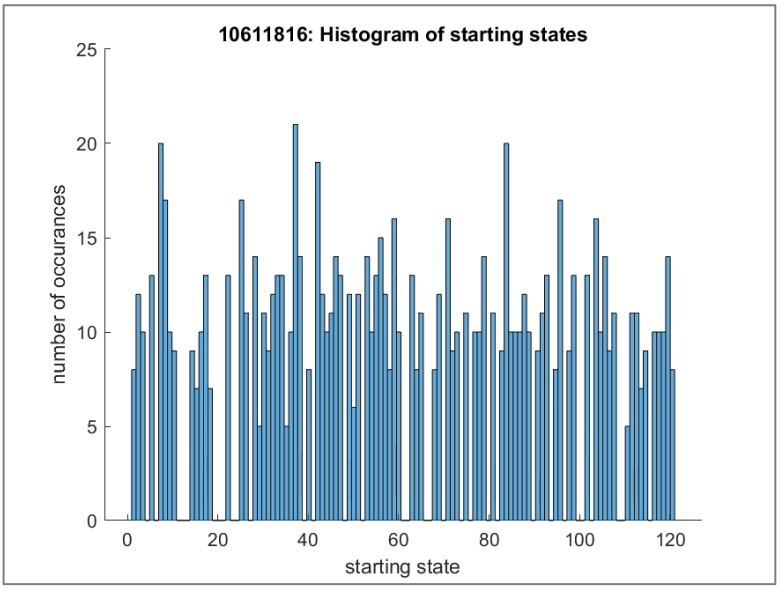
3.1. Random start state [3 Marks]

- **Write a function to generate a random starting state in the maze. NB: The start state may not be a blocked state or the goal state.**

```
function [state] = RandomStartingState(f)
  while(1)
     startLocation = [randi(f.xStateCnt), randi(f.yStateCnt)];
     x = startLocation(1);
     y = startLocation(2);
     if f.stateOpen(x,y) == 1
        break;
     end
  end
  state = f.stateNumber(startLocation(1), startLocation(2));
end
```

- **Generate 1000 starting states and plot a histogram using the MATLAB histogram function to check your function is working. You should end up with a histogram looking something like the one shown in Fig 5.**

```
for i = 1:1000
    [randomStartState(i)] = maze.RandomStartingState();
End

histogram(randomStartState, 121);
title(stuNo + "Histogram of starting states")
```



- **Comment on the histogram.**

As the RandomStartingState function finds a random state between 1 and 121, I have set my histogram to display 121 bins. This means I can tell how many times each individual state has been randomly generated. As the function also avoids all 33 blocked states and the end state, there are 34 bins in the graph showing zero generations. As this function ran 1000 times and there are just under 100 acceptable states that can be generated, the average number of generations for a chosen state is just under 10 times.

3.2 Build a reward function [3 Marks]

- **Specify the reward function for the maze.**

```
reward = maze.RewardFunction(s, a);
```

- **Explain how your reward function work**

```
% init to no reward
reward = 0;
if (stateID == f.stateEndID-1) && (action == 2)
    reward = 10;
else
    reward = 0;
end
```

As there is only one way to reach the goal of this maze (moving East from State 120), this reward function compares the current state and checks if it is state 120 and then compares the current action and checks if it is action 2 (east). If both of these things are true, the reward is set to 10. If either is false, the reward stays at 0.

3.3. Generate the transition matrix [3 marks]

- **Specify a transition matrix for the maze.**

```
maze = maze.BuildTransitionMatrix();
```

- **You should generate it automatically after you represent the maze appropriately using a matrix.**

This transition matrix is found in the workspace under `maze.tm` and is printed to the command window after being generated.

3.4. Initialize Q-values [3 marks]

- **Initialize the Q-values matrix to suitable numbers.**

```
minVal = 0.001;
maxVal = 0.1;
maze = maze.InitQTable(minVal, maxVal);
```

- **Explain your solution.**

```
offset = minVal;
range = maxVal - minVal;
f.QValues = zeros(f.xStateCnt * f.yStateCnt, f.actionCnt);

% YOUR CODE GOES HERE ...
for s = 1:(f.xStateCnt * f.yStateCnt)
  for a = 1:f.actionCnt
     f.QValues(s, a) = offset + range.*rand();
  end
end
```

This function takes the minimum and maximum values passed in and using a uniform distribution creates a matrix of random numbers between these values. The matrix is initialised with zeros initially before being filled with random numbers. It has 4 columns, one for each action, and 121 rows, one for each possible state. This size is calculated using the number of states and actions specified earlier in code and so would adapt and continue working if the maze were redesigned.

3.5. Implement Q-learning algorithm [15 marks]

- **Build the Q-table update function**

```
f.QValues = QTableUpdate(f, state, action, nextState, reward, alpha, gamma);

QT = f.QValues;
maxV = max(QT(nextState,:)); %maximum value in next state

% Q(St, At) = Q(St,At) + alpha(reward + gamma(max value in next state) - previous Q)
QT(state, action) = QT(state, action) + alpha*(reward + gamma*(maxV) - QT(state, action));
```

- **To implement Q-learning you will also need to**
  - **an outer loop to run 100 trials**

  within the Main_P3_RunGridworld2022.m script trials is set to 100.

  ```
  trials = 100;
  ```

  within the function maze.Experiment, the outer loop is run

  ```
  for trial = 1:trials
      [f, stepsAcrossTrials(:, trial)] = Trial(f, episodes, alpha, gamma, epsilon);
      fprintf("Trial %d finished\n", trial)
  end
  ```

  - **a loop to run 1000 episodes**

  within the Main_P3_RunGridworld2022.m script episodes is set to 1000.

  ```
  episodes = 1000;
  ```

  within the function maze.Trial, the inner loop is run

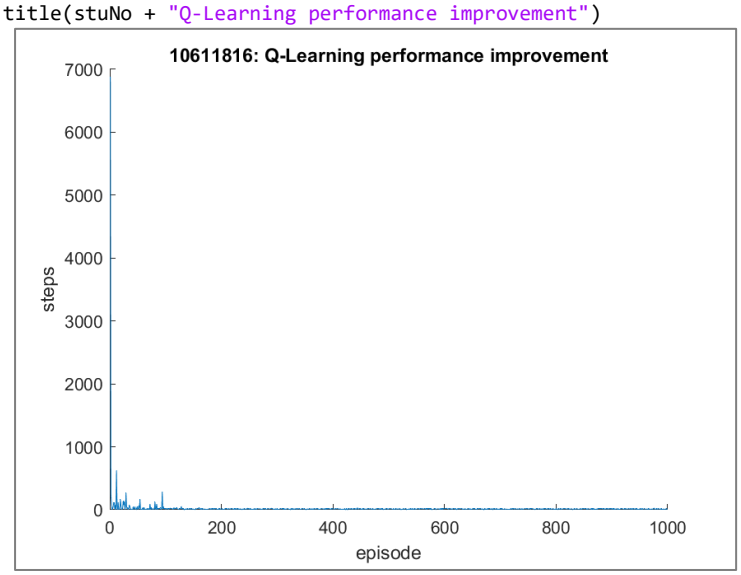
  ```
  for idx = 1:episodes
      [f, stepsAcrossEpisodes(idx)] = Episode(f, alpha, gamma, epsilon);
  end
  ```

  - **Within each episode you will need a loop to run for a given number of states until the goal state is reached**

  ```
  while(running==1)
    action = ActionSelection(f, state, epsilon); % Greedy Action
      nextState = f.tm(state, action);             % next state from action
      reward = RewardFunction(f, state, action);   % reward at next state

      % update Q Table using this info
      f.QValues = QTableUpdate(f, state, action, nextState, reward, alpha, gamma);
      % continue or stop?
      if IsEndState(f, nextState)
          running=0;
      else
          steps = steps+1;
          state = nextState;
      end
  end
  ```

- **Record the number of steps needed in an episode since it is indicative of how good Q-learning policy is becoming, and can be used as an indication of algorithm performance on the training data.**

```
[maze, stepsTr] = maze.Trial(episodes, alpha, gamma, epsilon);
plot(stepsTr)
title(stuNo + "Q-Learning performance improvement")
```



3.6. Run Q-learning [8 marks]

- **Run the Q-learning algorithm using:**
  - **An exploration rate of 0.1**
  - **A temporal discount rate gamma of 0.9**
  - **A learning rate alpha of 0.2.**

```
alpha = 0.2;   % learning rate
gamma = 0.9;   % temporal discount rate
epsilon = 0.1; % exploration rate
```

- **Analyse the performance of your Q-learning algorithm on the maze by running an experiment with 100 trials of 1000 episodes.**

```
episdoes = 1000;
trials = 100;
[maze, stepsEx] = maze.Experiment(episodes, trials, alpha, gamma, epsilon);
```

- **Note you only need to run over 100 trials only to show the variance across complete trials – just a single trial is enough to actually solve the maze.**

```
[maze, stepsTr] = maze.Trial(episodes, alpha, gamma, epsilon);
```

- **Generate an array containing the means and standard deviations of the number of steps required to complete each episode.**

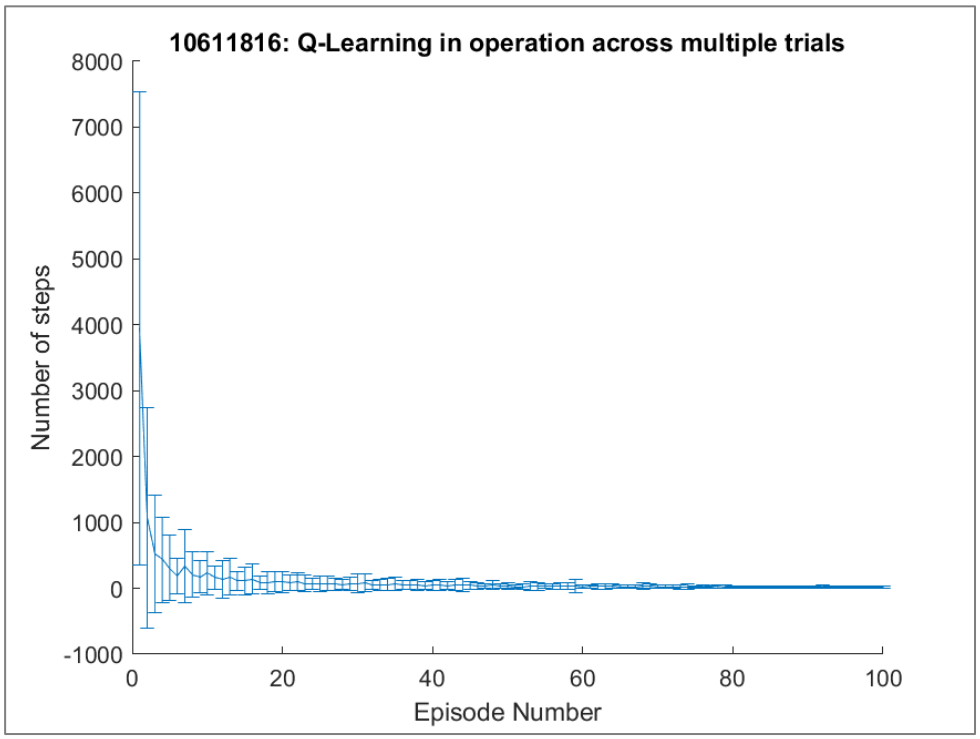
```
for i = 1:trials
    mean_data(i) = mean(stepsEx(i,:));
    std_data(i) = std(stepsEx(i,:));
end
```

- **Plot the mean and standard deviation across trials of the steps taken against episodes. You should get a ploy like that shown in Fig. 6. Describe what you find.**

```
errorbar(mean_data, std_data);
title(stuNo + "Q-Learning in operation across multiple trials");
```

As can be seen in the graph below, across the trials there is a good improvement as the episodes progress. The mean (the curve) shows a quick improvement and by around 10 episodes it is already showing at a mean between 0 and 1000 steps for an episode. The standard variation is also massively improved by this stage and

the error bars show this with the bars reducing in size quickly. There is a small bit of variation visible between 10 and 20 episodes as the mean is not a smooth curve. This is likely due to many similar routes available to the algorithm. By the 20 episode mark these are mostly solved and the error curve and error bars are both hard to see on the large scale graph, close to around 25 steps per episode.



3.7. Exploitation of Q-values [3 marks]

- **Record the Q-table at the end of a training trial.**

This is saved in maze.QValues and can be accessed as needed.

- **Write an exploitation function (or use your Є -greedy function with a zeroexploration probability Є) that makes use of the Q-values and performs greedy action selection without exploration.**

```
epsilon = 0;
path = maze.QExploit(epsilon);
```

- **Select the starting state as the green state shown on the maze.**

```
state = f.stateNumber(1, 1);
```

- **Record the visited states for this episode.**

```
visitedStates(steps) = state;
```

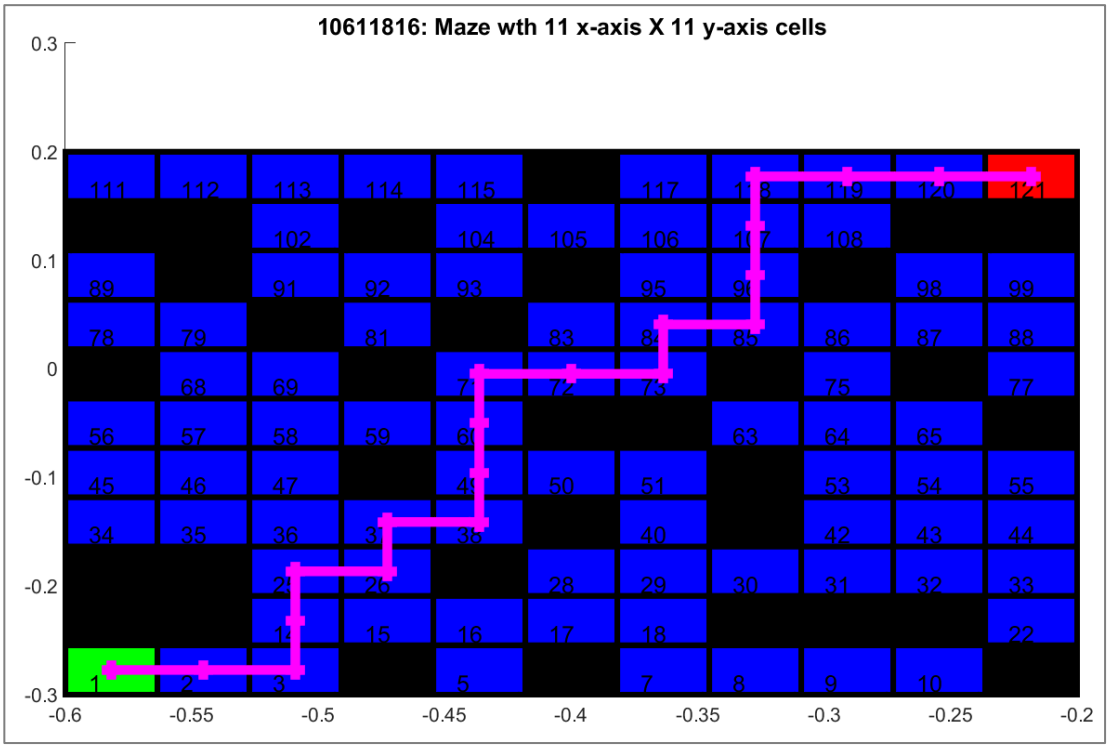
- **Convert the state into a 2-D coordinate that you can plot out in a matrix.**

```
path = zeros(2, steps);
for i = 1:steps+1
    [path(1, i), path(2, i)] = find(f.stateNumber == visitedStates(i));
end
```

- **This should take the form of a 2xN matrix where the first dimension relates to the (x,y) coordinates of the data points, and the second to the N of steps in the episode.**

- **Try to plot out the path over a drawing of the maze.**

```
% plot the path taken over the map
figure(p)
hold on
plot(scalePath(1,:), scalePath(2,:), 'm+-', 'MarkerSize', 10, 'LineWidth', 5)
```



4. Move arm endpoint through maze

4.1. Generate kinematic control to revolute arm [10 Marks]

- **Use the maze path to specify the endpoint trajectory of the 2-joint revolute arm.**

```
scalePathHat = scalePath';
data = [scalePathHat; ones(1, length(scalePathHat))];
```

- **Use the inverse kinematic neural network you trained earlier to generate the arm's joint angles.**

```
theta = TrainedTwoLayerNetwork(data, W1, W2);
```

- **Ensure you have scaled the maze appropriately so that it fits into the workspace of your revolute arm!**

From Main_P3

```
% These define the axis of the maze
limits = [-0.7 -0.2; -0.3 0.2;];
```

From Main_P4

```
axisX = [maze.limitsXY(1, 1) - 0.1, 0.1];
axisY = [maze.limitsXY(2, 1) - 0.1, maze.limitsXY(2,2) + 0.1];
```

This means the origin and the maze will both fit easily onto the workspace.

- **Use the forward kinematic function with these angles as input to calculate the arm elbow and endpoint positions.**

```
[P1, P2] = RevoluteForwardKinematics2D(armLen, theta, origin);
```

- **Plot the endpoint trajectory of the arm on to the maze path you are following to demonstrate how well the inverse model works.**

```
for idx = 1:length(scalePathHat)
    plot(P1(1, idx), P1(2, idx), 'ro', 'MarkerSize', 4);
    plot(P2(1, idx), P2(2, idx), 'co', 'MarkerSize', 4);
    plot([origin(1), P1(1, idx)], [origin(2),P1(2, idx)], 'b.-');
    plot([P1(1, idx), P2(1, idx)], [P1(2, idx),P2(2, idx)], 'b.-');
end
```

4.2. Animated Revolute Arm Movement [6 Marks]

- **Generate an animation of the endpoint of the revolute arm moving through the maze. Also draw the arm as well.**

To generate this animation, I used VideoWriter to capture individual frames as my animation progressed:

```
video = VideoWriter('figures/Task4Animation.avi');
video.FrameRate=2;
open(video);
```

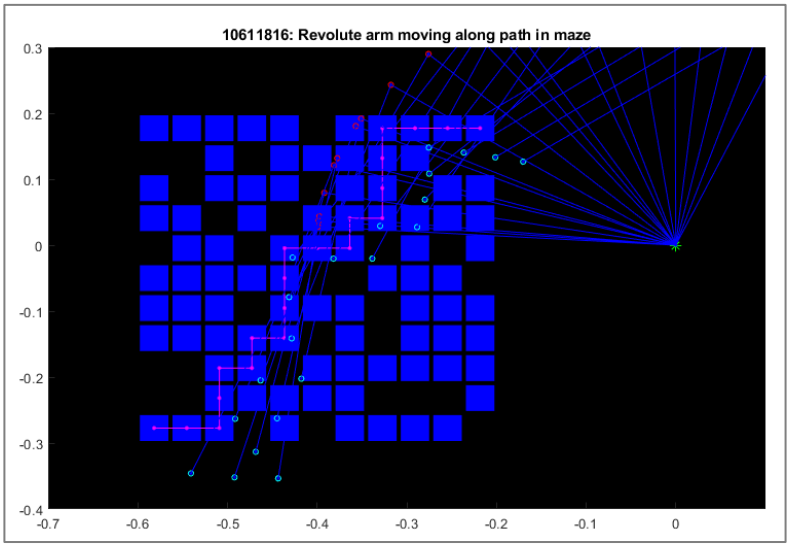
And then I added two lines into the for loop written above:

```
pause(1);
writeVideo(video, getframe(gcf));
```

Finally, I closed the video and it saved as an .avi file.

```
close(video);
```

By adding the pause into my for loop, this not only produced an animation video but also displayed the figure in MatLab as an animation as well.



- **Produce a video of your results and put a link to the video uploaded to YouTube or OneDrive in your report**
  https://youtu.be/sLdH1-F8Tnc