AINT351 – MACHINE LEARNING

10555972 | Computer Science | 12th December 2019

Contents

[1. Training Data Generation 3](#_Toc26178564)

[1.1. Display Workspace of Revolute Arm 3](#_Toc26178565)

[1.2. Configurations of a Revolute Arm 5](#_Toc26178566)

[2. Implement a 2-Layer Network 6](#_Toc26178567)

[2.1. Implement the Network Feedforward Pass 6](#_Toc26178568)

[2.2. Implement 2-Layer Network Training 7](#_Toc26178569)

[2.3. Train Network Inverse Kinematics 8](#_Toc26178570)

[2.4. Test and Interpret Inverse Model 9](#_Toc26178571)

[3. Path Through a Maze Using Q-Learning 12](#_Toc26178572)

[3.1. Generate Random Start State 12](#_Toc26178573)

[3.2. Build a Reward Function 12](#_Toc26178574)

[3.3. Generate a Transition Matrix 12](#_Toc26178575)

[3.4. Initialize Q-Values 12](#_Toc26178576)

[3.5. Implement Q-Learning Algorithm 12](#_Toc26178577)

[3.6. Run Q-Learning 12](#_Toc26178578)

[3.7. Exploitation of Q-Values 12](#_Toc26178579)

[4. Move arm Endpoint Through Maze 12](#_Toc26178580)

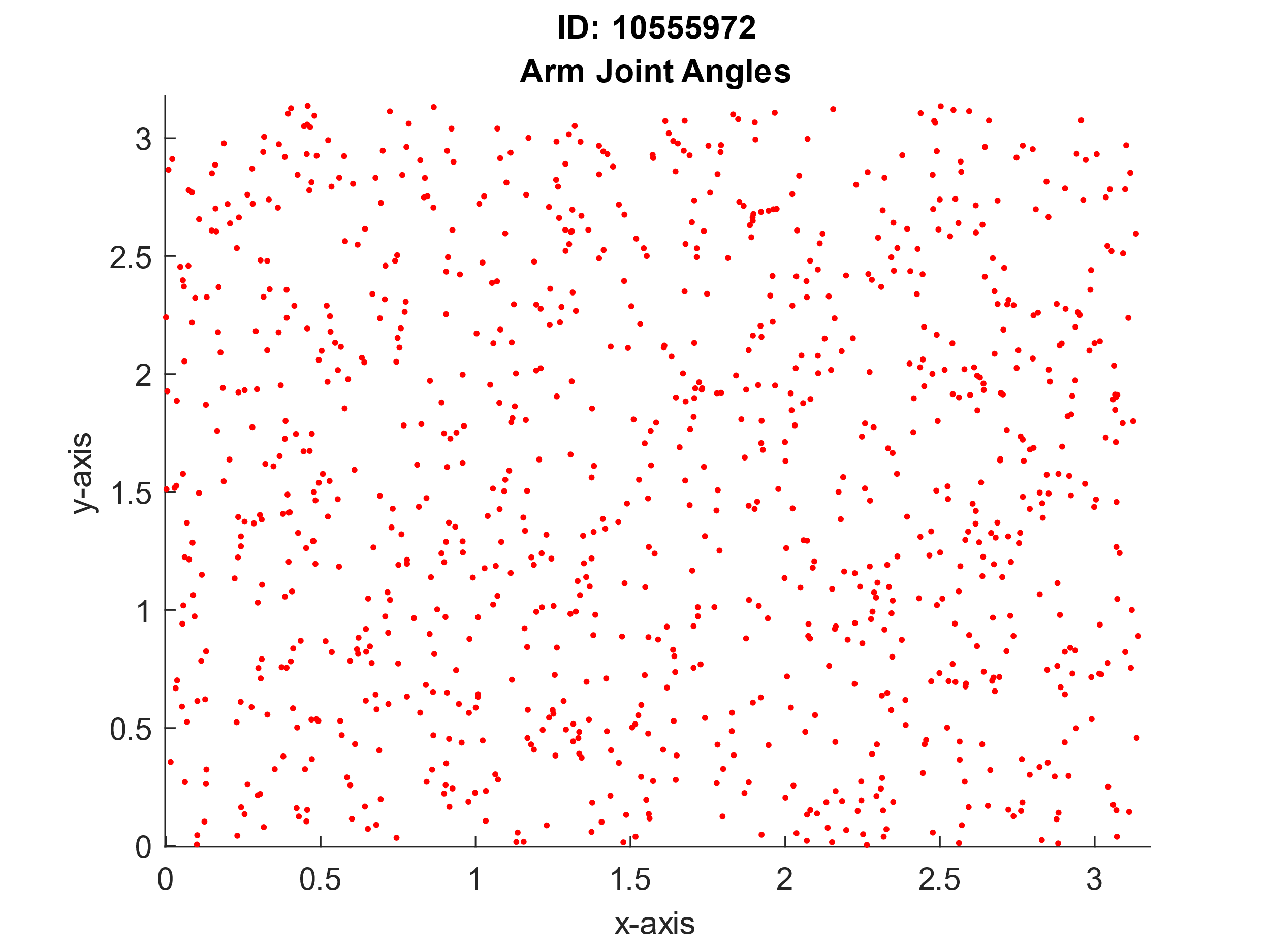
[4.1. Generate Kinematic Control to Revolute Arm 12](#_Toc26178581)

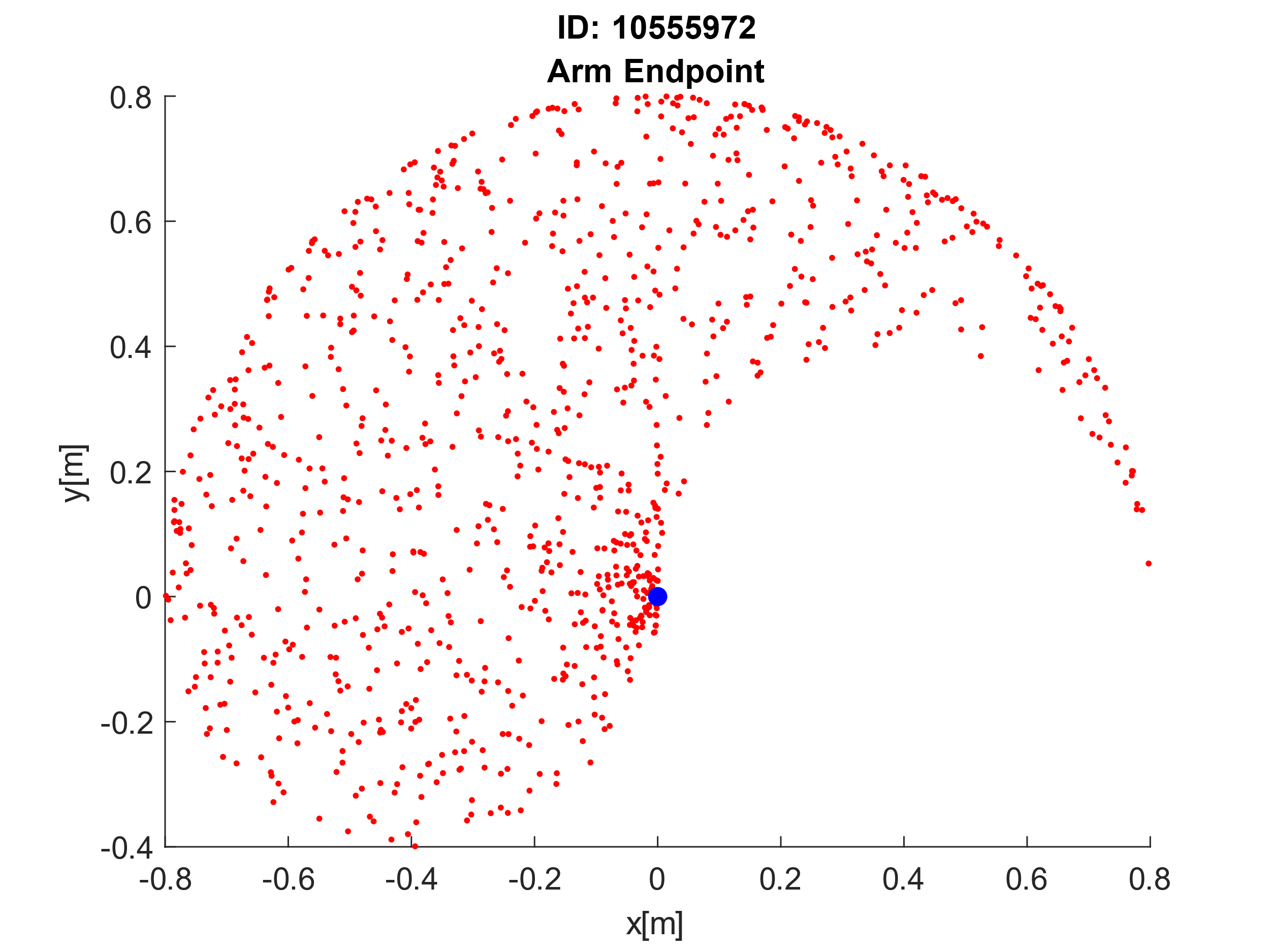
[4.2. Animated Revolute Arm Movement 12](#_Toc26178582)

# Training Data Generation

The provided Revolute Forward Kinematics 2D function is used to be able to output the arm end points by using the arms joint angles, the length of the arms and the base origin of the arms. The end points that are generated from this function will then later be used to train a neural network.

## Display Workspace of Revolute Arm

To display the workspace of the revolute arm I generated a random dataset between the values of 0 and π. The dataset had uniform distribution and contained 2x1000 samples. This dataset contains the angles that will be passed through the forward kinematics to calculate the end points and show the workspace of the arm. I then set the parameters for the Revolute Forward Kinematics function to use, the arm lengths for before and after the elbow were set to 0.4 and the base origin coordinates were set to (0, 0). Passing in these values and the joint angles previously generated, the function produces the correspdoning end points. Due to the arm only having 2 degrees of freedom the useful range of the end points is rather limited. This could be increase by adding a third joint to the arm, allowing it to move freely throughout the plane.

****

% Defining variables

armLength = [0.4;0.4];

baseOrigin = [0, 0];

samples = 1000;

% Generating 2 x samples between 0 - pi

angles = pi \* rand(2,samples);

% Run angles through forward kinematics

[P1, P2] = RevoluteForwardKinematics2D(armLength, angles, baseOrigin);

% Plot randomly generated angles

figure

hold on

title({'ID: 10555972', 'Arm Joint Angles'});

xlabel('x-axis');

ylabel('y-axis');

plot(angles(1,:), angles(2,:), 'r.');

% Plot end points

figure

hold on

title({'ID: 10555972', 'Arm Endpoint'});

xlabel('x[m]');

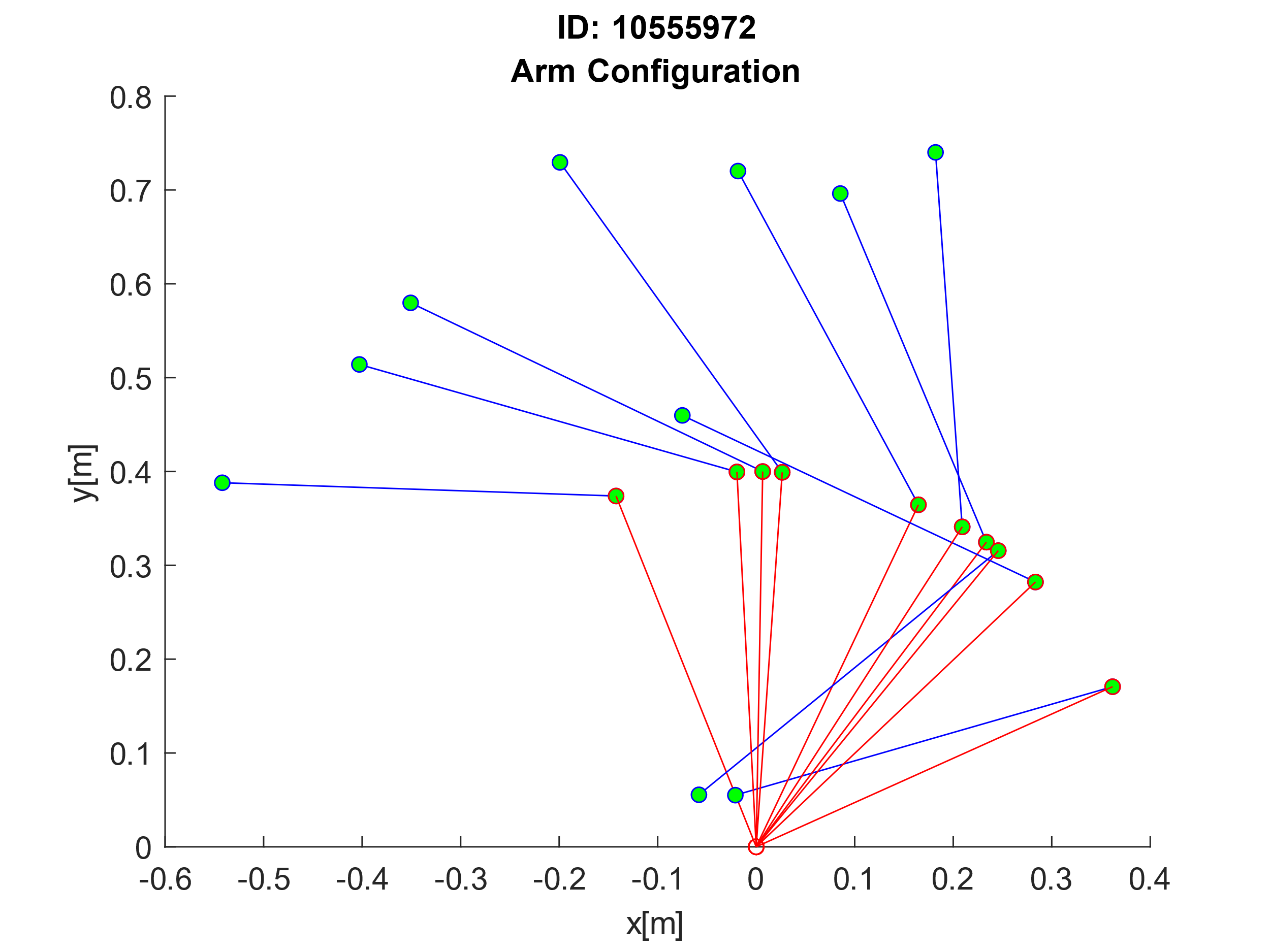
ylabel('y[m]');

plot(P2(1,:), P2(2,:), 'r.')

plot(baseOrigin(1), baseOrigin(2), 'b.', 'MarkerSize', 20);

## Configurations of a Revolute Arm

To help illustrate the arm configurations I have plotted 10 elbow and end points locations and the arm between them. This has been done by using 10 of the randomly generated set of angles previously and running it through the forward kinematics function. This plot gives a greater understanding about the movement of the arm and the range of motion it can have.



% Plot 10 arm configurations

figure

title({'ID: 10555972', 'Arm Configuration'});

xlabel('x[m]');

ylabel('y[m]');

for i = 1:10

hold on

% Plotting from elbow to end of arm

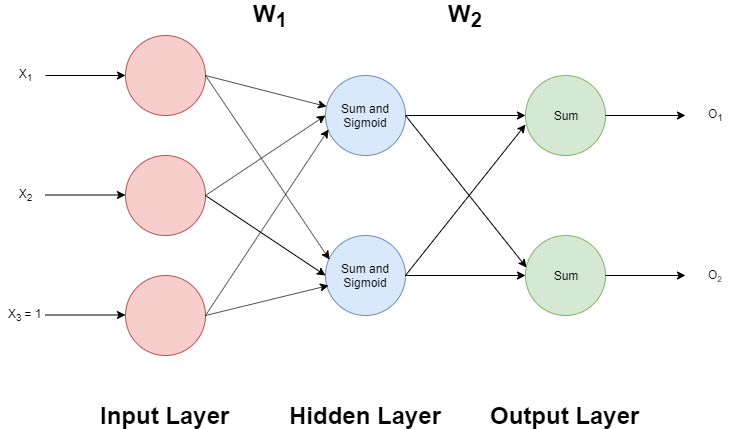
plot([P1(1,i) P2(1,i)],[P1(2,i) P2(2,i)], 'b-o', 'MarkerSize', 5, 'MarkerFaceColor', 'green');

% Plotting from origin to elbow

plot([P1(1,i) baseOrigin(1)], [P1(2,i) baseOrigin(2)], 'r-o', 'MarkerSize', 5);

end

# Implement a 2-Layer Network

The next step is to build a multi-layer neural network which will be used to learn and calculate the robot arm’s inverse kinematics. To do this I will build a network that has 2 inputs (plus a third for the bias), a layer of hidden nodes (the diagram shows 2 but this can be n number of nodes) and two outputs. The network is fully connected by weight matrices on both the first and the second layer and will be passed through a sigmoid function in the hidden layer.

The input data will be the arm endpoints that have been calculated from the forward kinematics function and the output will be the inverse of the kinematics, so in this case it will be the randomly generated dataset between 0 and π. I have chosen to have two outputs for this network instead of one as it will allow me to output both the x and the y coordinates at the same time instead of having to have two separate networks and pass through each one.

## Implement the Network Feedforward Pass

To start I created a feed forward function which takes as parameters the input data and the weight matrices for the network. This function completes a one whole pass of the network to calculate the output which is then returned by the function. I have also created a function which calculates and returns the sigmoid activation of any given input. A sigmoid function is useful because it is non-linear and reduces the range between 0 and 1 whilst keeping continuous values.

% This function is used to carry out a feedforward pass of the network

% given its input data and both weight matrices.

function output = FeedForward(input, W1, W2)

% Add bias to input matrix

input = [input; 1];

% Calculate output from hidden layer

net = W1\*input;

% Sigmoid activation function

a2 = SigmoidFunction(net);

% Adding bias to activation from hidden layer

a2hat = [a2; 1];

% Calculating output from output layer

output = W2\*a2hat;

end

% Function to carry out the sigmoid acivation calculation

function result = SigmoidFunction(net)

result = 1 ./ (1+(exp(-net)));

end

## Implement 2-Layer Network Training

To train a network we must first calculate the error of the output, this is done by finding the difference between the target output and the actual output. Neural networks are a type of supervised learning, so the system requires the target data to be able to calculate the error. To do this for a 2-layer network backpropagation should be used to ensure that the entire network is updated as the second layer takes the first layer’s weights as inputs, so adjusting just the first layer would have a knock-on effect to the second layer. To adjust the weights from the first layer we can then use the delta term calculated from the second layer to be the error term for the first layer.

The function below runs through a feedforward pass and then backpropagates to adjust the weights accordingly. The function takes the input data, target data and both weight matrices, and then returns the updated weight matrices. Delta 3 is equal to the error between the input data and the target data. Delta 2 is equal to the error back propagated from the higher level (the weight matrix has its bias removed before this calculation), multiplied by a scaler due to the sigmoid function in the hidden layer.

The error gradient for both weight matrices is then calculated by using the delta for that layer multiplied by the input of that layer. The weights are then updated and returned by taking away the gradient multiplied by the learning rate.

% Function to train the network given input data, target data and the weight matrix. By calculating the error gradient and updating the weight values.

function [W1, W2] = Train(input, target, W1, W2)

% Setting learning rate

learningRate = 0.01;

% FEEDFORWARD PASS

% Calculate output from hidden layer and add bias

input = [input; 1];

net = W1\*input;

% Sigmoid activation function

a2 = SigmoidFunction(net);

% Adding bias to activation from hidden layer

a2hat = [a2; 1];

% Calculating output from output layer

o = W2\*a2hat;

% BACKPROPAGATION

% Delta 3 is equal to the output error

delta3 = -(target-o);

% Removing bias from weights

for i = 1:size(W2,2)-1

W2Hat(1,i) = W2(1,i);

W2Hat(2,i) = W2(2,i);

end

% Delta 2 is equal to the error from the second layer multiplied by a

scaling factor due to the sigmoid function

delta2 = (W2Hat'\*delta3).\*a2.\*(1-a2);

% Calculating the error gradient

errGradientW1 = delta2\*input';

errGradientW2 = delta3\*a2hat';

% Updating weights using the learning rate and error gradient

W1 = W1 - learningRate\*errGradientW1;

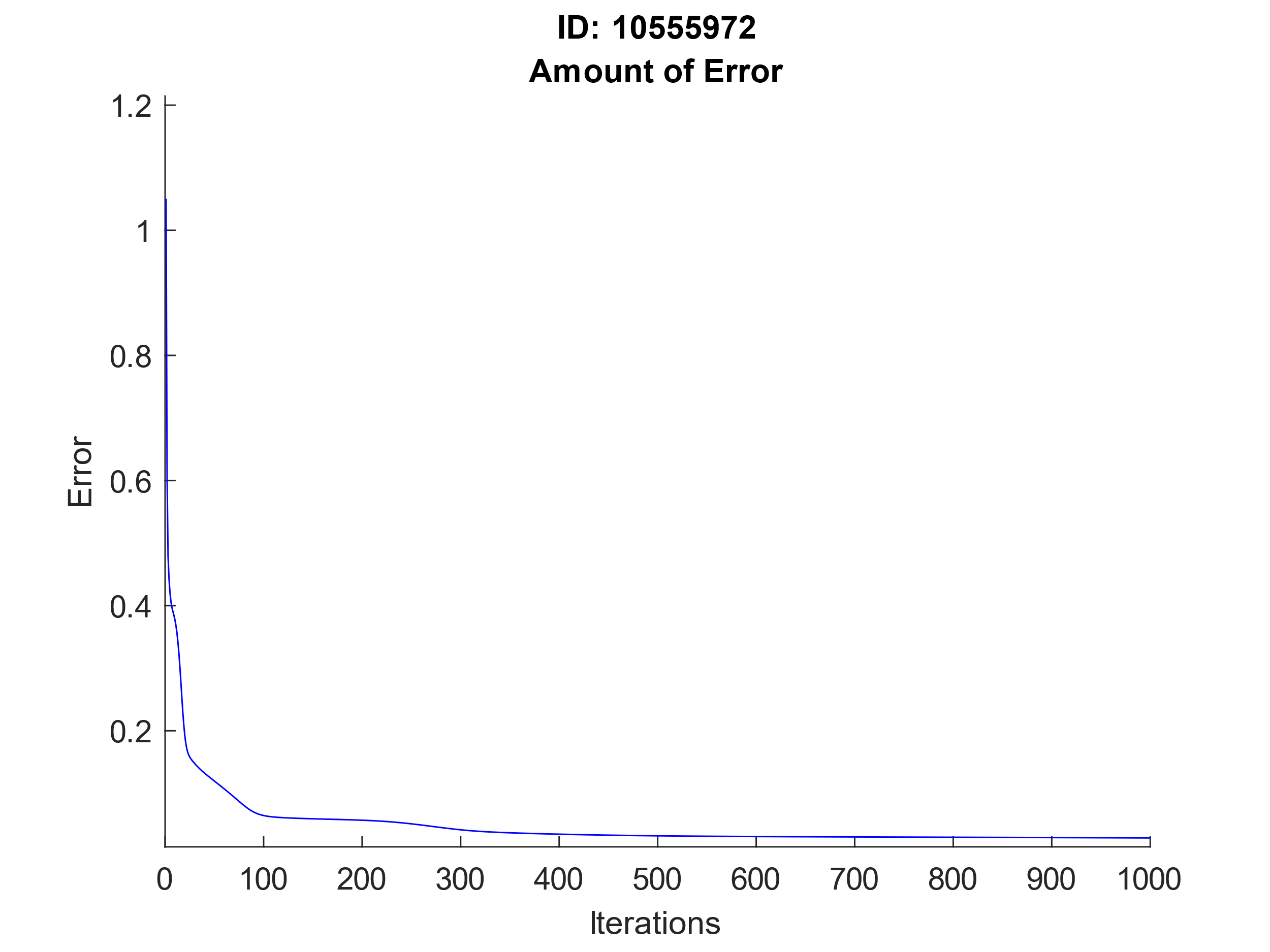
W2 = W2 - learningRate\*errGradientW2;

end

## Train Network Inverse Kinematics

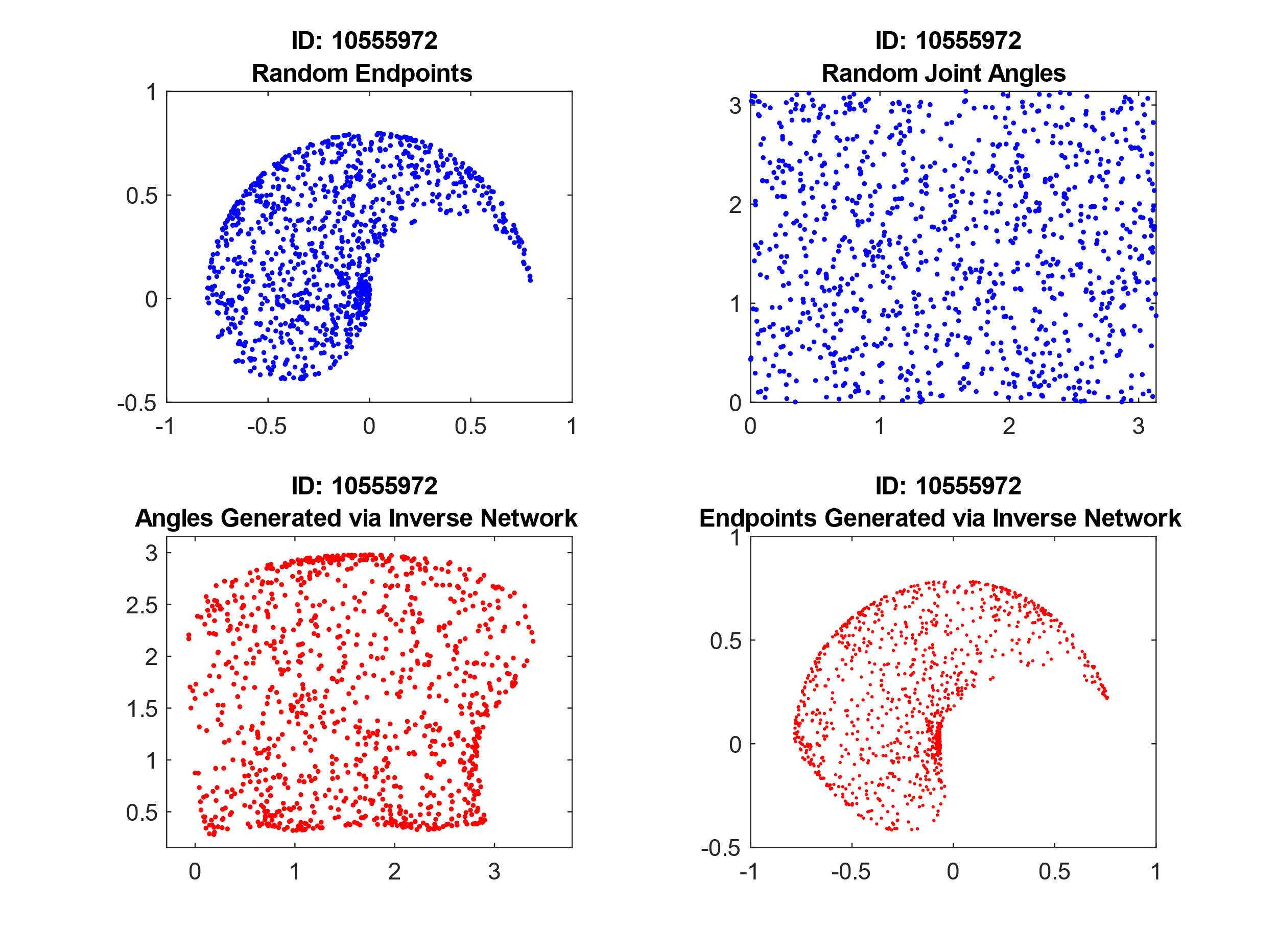
It is possible to calculate the inverse kinematics for a robot arm with only 2 degrees of freedom. However, if this number was greater, then the calculation would become much more difficult. So, in this case we want the network to learn inverse kinematics for us. To do this, we will feed in the arm end points as the input data and some randomly generated angles as the target data. Due to my network having 2 outputs we will only have to train one network.

Below I have plotted the error of the neural network as the training happens. To calculate the error, I used the following formula: . I calculated the error for each data point and then took the mean for each iteration. I ran the training for 1000 iterations and the error tends towards 0.



## Test and Interpret Inverse Model

After the network as been trained, I can carry out feedforward passes on arm end points to get the angles. I have generated a new random dataset and used the Forward Kinematics function to get the endpoints. I have then completed the feed forward pass on this data because it is different data from the data that the network was trained on. As you can see below the results are within the correct range and the output is fairly accurate. For these results I have used 1000 iterations, 10 hidden nodes and a learning rate of 0.01 for training. These values have been chosen to ensure that overfitting does not occur when training my network. Overfitting is when the training error becomes so low on the data that it is being trained on that the error on new data being passed through the network is very large. I have found that these values provide the best results while in turn not taking a considerably long time to train the network.



To better improve the accuracy of a neural network it is sometimes feasible to randomize the order of data when training the network this is so that there is no correlation between data and that the order of the data has an effect on the output of the network. Therefore, I tried randomizing the training data as shown below and this had little to no effect on the training of the network. This could be because the data set it relatively small and also because it is being trained on a random set of data.

% Matrix from 1 to samples in random order

r1 = randperm(samples);

% Training the data for the number of iterations for each data point

for i = 1:iterations

for j = 1:samples

[W1, W2, err(j)] = Train(P2(:,r1(j)), randAngles(:,r1(j)),W1,W2);

end

end

Talk about how the data in extrinsic space is distributed and how to make it more uniform

Speak about scaling data & why you haven’t done it

Due to this network being used to produce a revolute arm to guide its way through a maze, a much more appropriate dataset to train the network on would be to train it specifically between the boundaries of the maze. This is because that the rest of the training data is wasted as the data outside of these boundaries would never be used. In addition, this could potentially allow the network to reduce its training error a lot quicker, in turn reducing training times.

Below is the code used to train the network, complete a feedforward pass on a set of data and then plot the output as well as the original data.

function [W1, W2] = Network()

% Defining variables

armLength = [0.4;0.4]; baseOrigin = [0, 0];

samples = 1000; iterations = 1000;

noOfInputs = 2; noOfHiddenNodes = 10; noOfOutputNodes = 2;

% Generating 2 x samples data between 0 and pi

randAngles = pi \* rand(2,samples);

% Calculating arm end points given angles

[P1, P2] = RevoluteForwardKinematics2D(armLength, randAngles, baseOrigin);

% Initialising random weights, plus 1 used for the bias

W1 = rand(noOfHiddenNodes, noOfInputs + 1);

W2 = rand(noOfOutputNodes, noOfHiddenNodes + 1);

% Training the data for the number of iterations for each data point

for i = 1:iterations

for j = 1:samples

[W1, W2, err(j)] = Train(P2(:,j), randAngles(:,j), W1, W2);

end

end

% Generating new data to feed forward pass through the network

randAngles2 = pi \* rand(2,samples);

[P1, endPoints] = RevoluteForwardKinematics2D(armLength, randAngles2, baseOrigin);

% Passing data through network

for i = 1:samples

outputtedAngles(:,i) = FeedForward(endPoints(:,i), W1, W2);

end

% Using the output angles and getting the arm end points

[P3, P4] = RevoluteForwardKinematics2D(armLength, outputtedAngles, baseOrigin);

% Plot the random angles and endpoints. Then plot the generated

% inverse angles and end points from the network.

figure

hold on

tiledlayout(2,2)

nexttile

plot(endPoints(1,:), endPoints(2,:), 'b.');

title({'ID: 10555972', 'Random Endpoints'});

nexttile

plot(randAngles2(1,:), randAngles2(2,:), 'b.');

title({'ID: 10555972', 'Random Joint Angles'});

nexttile

plot(outputtedAngles(1,:), outputtedAngles(2,:), 'r.');

title({'ID: 10555972', 'Angles Generated via Inverse Network'});

nexttile

plot(P4(1,:), P4(2,:), 'r.', 'markersize',4);

title({'ID: 10555972', 'Endpoints Generated via Inverse Network'});

end

# Path Through a Maze Using Q-Learning

## Generate Random Start State

## Build a Reward Function

## Generate a Transition Matrix

## Initialize Q-Values

## Implement Q-Learning Algorithm

## Run Q-Learning

## Exploitation of Q-Values

# Move arm Endpoint Through Maze

## Generate Kinematic Control to Revolute Arm

## Animated Revolute Arm Movement