HW2
Courtney Smith
Neural Network for learning a better motion model

#1. For this homework, I am aiming to learn a better motion model than the dead reckoning one I used in homework 0. The original model does not account for any error or noise in the system, so after a few seconds the dead reckoning model deviates from the actual robot position. The equations below represent the dead reckoning model.

$$x_t = x_{(t-1)} + v * \cos\left(\Theta_{(t-1)} + w * dt\right) * dt$$
$$y_t = y_{(t-1)} + v * \sin\left(\Theta_{(t-1)} + w * dt\right) * dt$$
$$\Theta_t = \Theta_{(t-1)} + w * dt$$

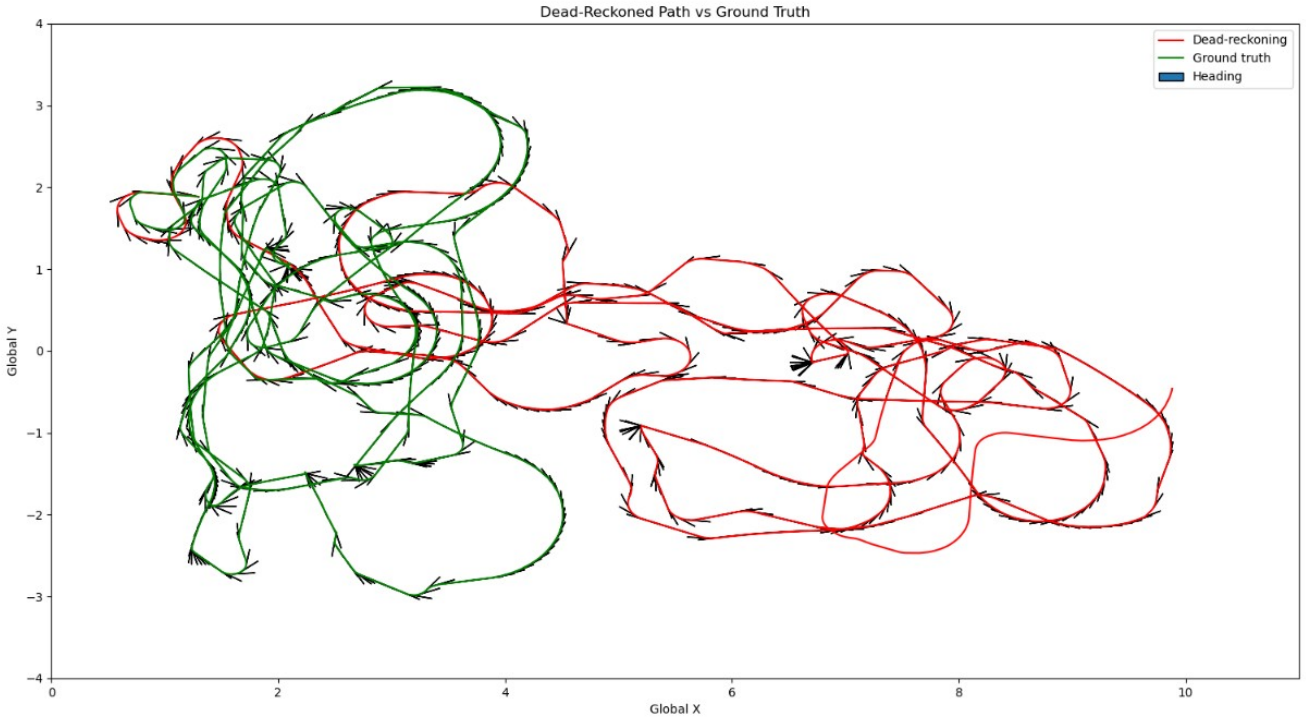



*Figure 1: Error from the Dead Reckoning model in HW0*

To learn a better motion model, I created a dataset that reflects the equations above. The data inputs to the model are dt, the time between 2 positional readings, $x_{t-1}$, $y_{t-1}$, and $\Theta_{t-1}$, the previous robot state, and v and w, linear and angular velocity commanded. These 6 inputs are crucial to determining the next state of the robot because it represents the current state and where the robot started, and how that state changes. The labels of the input data are the true next state of the robot, $x_t$, $y_t$, and $\Theta_t$. The goal of the model is to learn the next state even if the given inputs do not contain the error from noise. Ideally the model will be able to learn how error enters the system via the control inputs and how that affects the true position of the robot.

#2. Utilizing equations and figures from the Machine Learning, by Mitchell, Tom M. McGraw-Hill, 1997, an overview of the math of a neural network is outlined. The main idea behind a neural network is to learn sets of weights and biases that will minimize error through the process of back propagation. The outline of the algorithm is explained below from Table 4.2 in the Machine Learning book.

- Create a feed-forward network with $n_{in}$ inputs, $n_{hidden}$ hidden units, and $n_{out}$ output units.
- Initialize all network weights to small random numbers (e.g., between $-.05$ and .05).
- Until the termination condition is met, Do
  - For each $\langle \vec{x}, \vec{t} \rangle$ in *training_examples*, Do

*Propagate the input forward through the network:*

1. Input the instance $\vec{x}$ to the network and compute the output $o_u$ of every unit $u$ in the network.

*Propagate the errors backward through the network:*

2. For each network output unit $k$, calculate its error term $\delta_k$

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \tag{T4.3}$$

3. For each hidden unit $h$, calculate its error term $\delta_h$

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k \tag{T4.4}$$

4. Update each network weight $w_{ji}$

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

where

$$\Delta w_{ji} = \eta \, \delta_j \, x_{ji} \tag{T4.5}$$

The forward propagation step in the algorithm can be further broken down to a perceptron level illustrated below. The perceptron shows what happens in one layer for a single node where the weights are applied to the inputs and summed before entering an activation function.

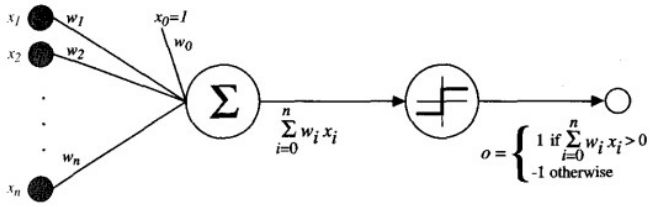



**FIGURE 4.2**
A perceptron.

During the back propagation step, the weights are “learned” and adjusted to minimize error by using an optimization function. In this example, I used a stochastic gradient descent optimizer. Weights are adjusted using the following functions from Ch. 4 of the mentioned texbook.

$$w_i \leftarrow w_i + \Delta w_i$$

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

$$\frac{\partial E}{\partial w_i} = \sum_{d \in D} (t_d - o_d)(-x_{id})$$

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d) \, x_{id}$$

The above equation step through the process of optimizing the weights by looking at the partial derivative of the output error with respect the weight. By using this gradient descent method, the weights can converge quicker. However, with this method, there is a possibility of reaching only a local minima instead of a global minima depending on specific parameters that will be discussed later. Another important point to discuss is what happens during back propagation when there are multiple layers. In the algorithm break down from the book, equation T4.3 and T4.4 show how what is represented by the output error of a layer depends on the error of the previous

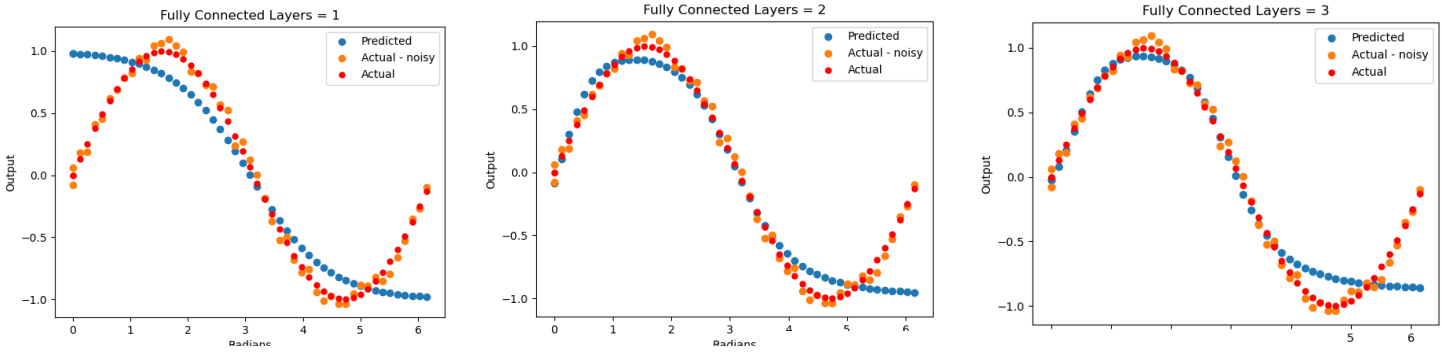
layer. The final layer in the model will back propagate using the total error of $(y_{true}-y_{pred})^2$ whereas the layer before it will back propagate using the output error of the previous layer, and so on for all following layers.

A neural network would work well for a learning problem like this for multiple reasons. First, our robot moves with inconsistent error which cannot be easily seen by the human eye. Neural networks are notorious for finding patterns and forming connection in data which make it a good candidate for learning the motion model whose data contains errors and noise. In addition to the noise, the motion model is nonlinear as shown by the equations written in question 1. Due to the nonlinearity of the learning aim, a neural network is an obvious choice because of it's ability to tune specific parameters to decrease the linearity of the model such as back propagation algorithms and activation functions. Also, while the training time for the model is computationally intensive, once the model is optimized and the weights are set, the model runs very quickly. This means that a real time implementation of this model would be feasible because there would be very little delay in the computation once the model is set. Neural networks have been used in a variety of applications within robotics, and the success of ALVNN and other neural networks gives me confidence the neural net will succeed in learning the motion model.

To optimize the model, I am looking at four parameters:
1. # of Full Connected Layers – also known as # of hidden layers
2. # of inputs/outputs of a layer
3. Learning Rate – how quickly you can change the weights during back propagation
4. Epochs – the number of times you show your model the training data to update parameters

To tune the parameters, I chose initial values for each and then performed a sweep on each of the parameters. Once error was minimized for one parameter, I moved onto the next parameter using the best values known. To begin optimizing for the noisy sine wave, my initial values were FC Layers = 1, inputs/outputs =20, learning rate = 0.01, and epochs = 100. Then I swept over the following intervals: FC Layer: 1,2,3,4, and 5, inputs/outputs: 10,20,30,40, and 50, learning rate: 0.005, 0.01, and 0.05, and epochs: 50,75,100, and 125. After optimizing for the sine wave, I will perform the same optimization on the robot data but with initial values equal to that used for the optimal sine wave.

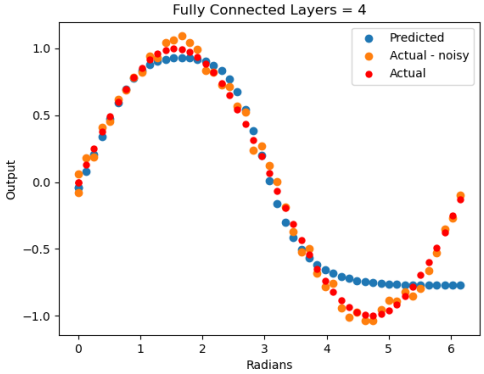
#3. Next I tested my model on a noisy sine wave. Below are 2 sets of graphs. The first set shows the actual sine curve(red), the noisy sine curve(orange), and the model's estimation of the noisy sine curve(blue). The second set of graphs plots the actual noisy sine curve vs the model's estimation, so it is easier to analyze the correlation between the two. The straight line in this graph represents the points where the model would be 100% accurate. The R^2 and MSE of the model's estimation is also shown on the graph.

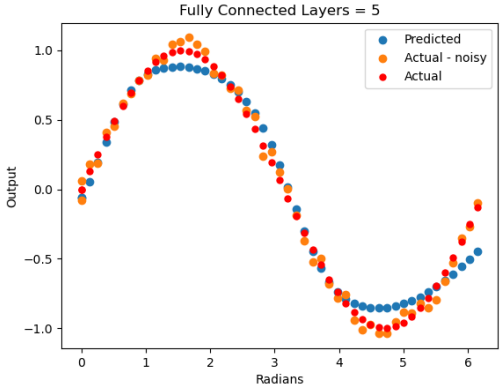


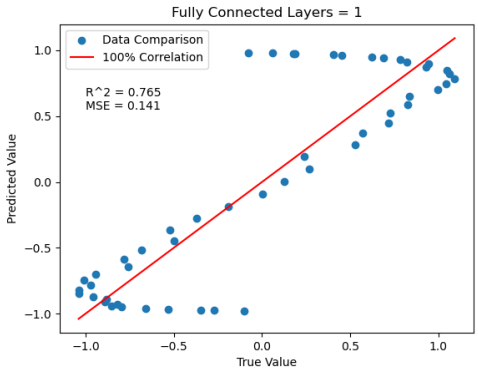


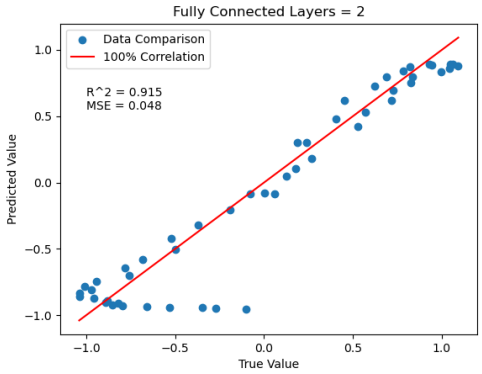


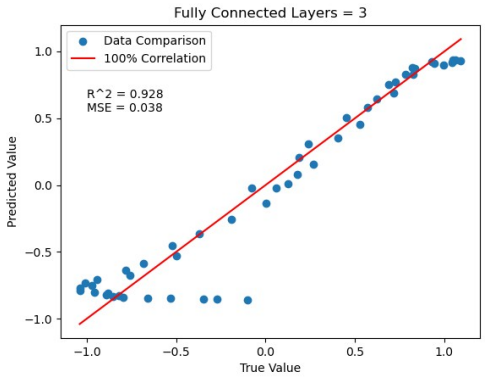


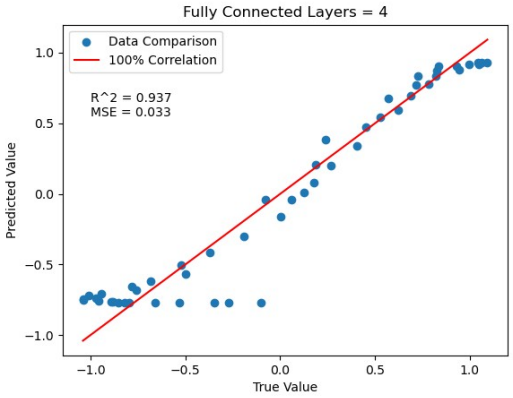



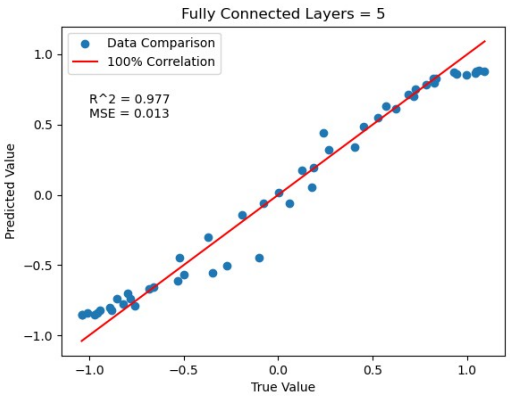





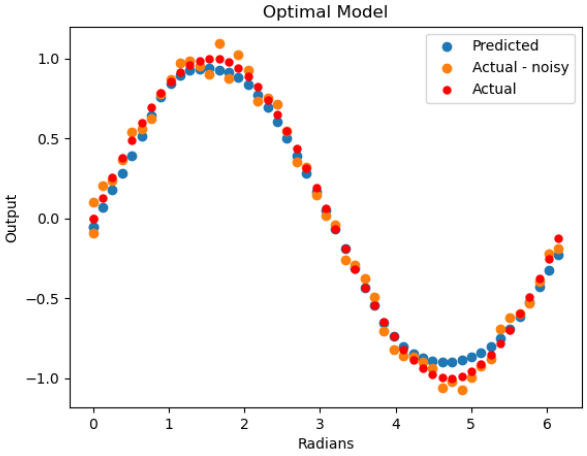



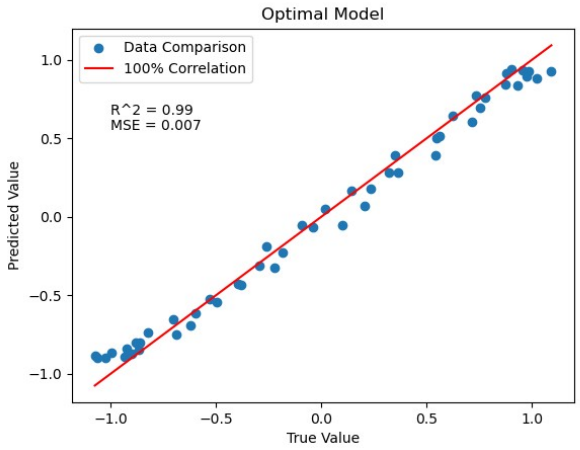




Above are the graphs produced when running a parameter sweep on the number of fully connected layers. In the first 5 graphs, you can see that as you increase the number of fully connected layers, the model's estimation improves. Initially with a smaller number of layers, the model struggles to estimate the edge points and cannot fully bend with the sine curve. However, once you reach 4 or 5 layers, you see better performance on those edge cases with an improved $R^2$ and MSE. From this analysis, I continued optimization with 5 fully connected layers. This process was used for the other parameters of the function until the following optimized parameters were found:

# fully connected layers = 5
# input/output channels = 50
learning rate = 0.005
# epochs = 100



#4. To start the optimization process for the learning aim, I ran the optimized parameters above on the robot data set. The results are shown below:

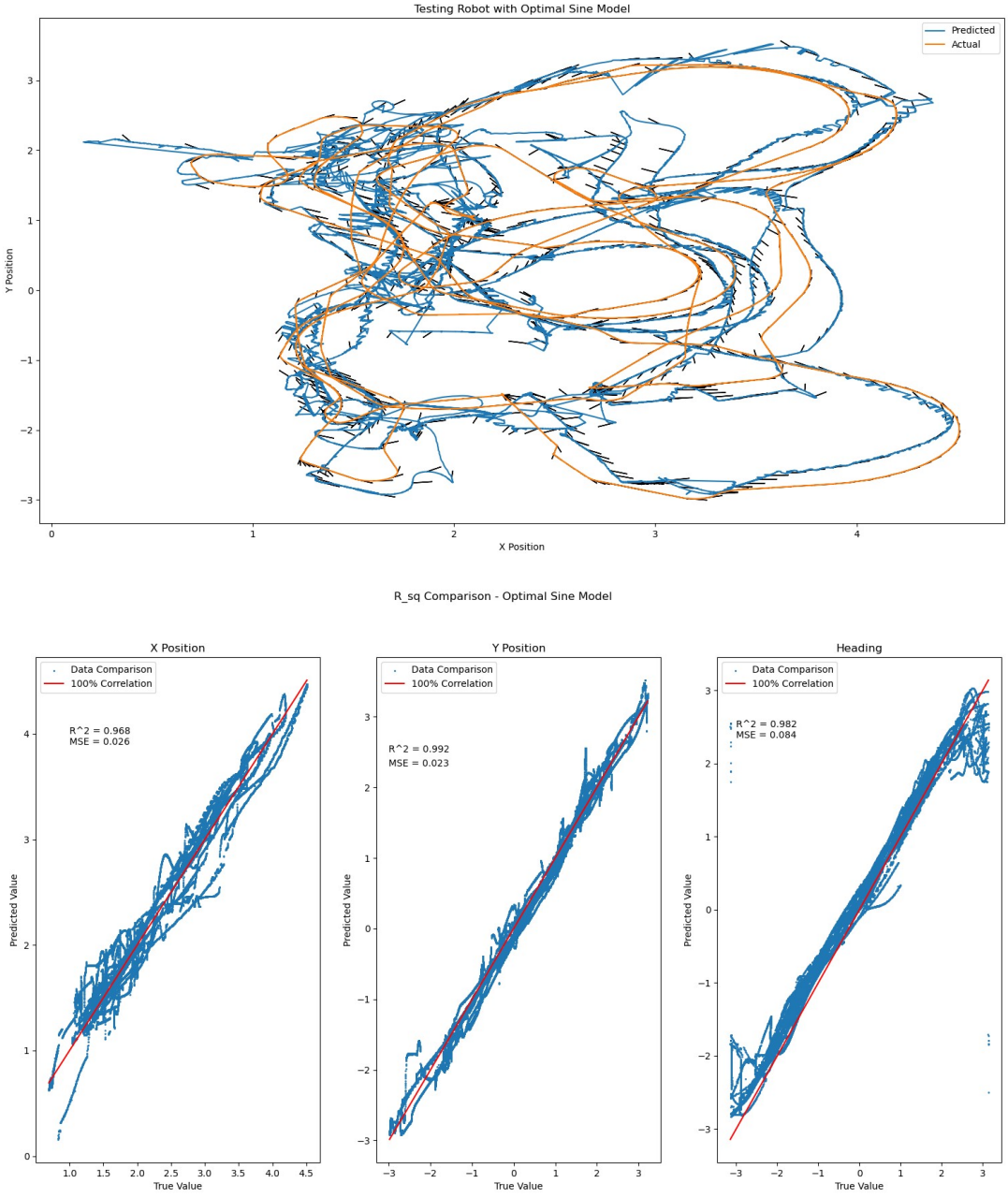



*Figure 2: The parameters optimized for the sine function on the robot data.*

The results above show an $R^2$ for X, Y, and Heading to be 0.968, 0.992, and 0.982 respectively, and the total MSE of X, Y, and Heading predictions were 0.026, 0.023, and 0.084 respectively. The above model performs well with the optimized sine function parameters, but with finer parameter tuning, the model should improve. Specifically, I want to see if the improved model can correct the error seen in the edge cases of the heading. Because the robot dataset is significantly larger and more complex than the sine wave, I am going to try larger input/output sizes, decrease the learning rate to help with some of the noise/jumpiness in the prediction, and increase the number of layers in the model to accommodate the increase in data. I am approaching increasing the number of epochs with caution because I don't want to cause over fitting by over exposing the data to the model.

Initial Parameter Values:

# fully connected layers = 5 (but sweeping first)
# input/output channels = 50
learning rate = 0.005
# epochs = 100

Sweeping Values:

# fully connected layers = (5,6,7,8,10)
# input/output channels = (10, 25, 40, 50, 75, 100)
learning rate = (0.01,0.005,0.001)
# epochs = (75, 100, 125)

Round 1: Sweeping # Fully Connected layers

| # fully connected layers | X Position | | Y Position | | Heading | |
|---|---|---|---|---|---|---|
| | $R^2$ | MSE | $R^2$ | MSE | $R^2$ | MSE |
| 5 | 0.927 | 0.073 | 0.984 | 0.045 | 0.978 | 0.076 |
| 6 | 0.968 | 0.03 | 0.99 | 0.031 | 0.977 | 0.063 |
| 7 | 0.918 | 0.179 | 0.979 | 0.071 | 0.95 | 0.146 |
| 8 | 0.888 | 0.113 | 0.97 | 0.128 | 0.959 | 0.119 |
| 10 | 0.127 | 2.052 | 0.007 | 4.576 | 0.717 | 1.406 |

Round 2: Sweeping # input/output channels

| # input/output channels | X Position | | Y Position | | Heading | |
|---|---|---|---|---|---|---|
| | $R^2$ | MSE | $R^2$ | MSE | $R^2$ | MSE |
| 10 | 0.927 | 0.226 | 0.941 | 0.186 | 0.968 | 0.121 |
| 25 | 0.985 | 0.015 | 0.992 | 0.022 | 0.982 | 0.084 |
| 40 | 0.98 | 0.018 | 0.989 | 0.029 | 0.987 | 0.043 |
| 50 | 0.975 | 0.023 | 0.984 | 0.043 | 0.976 | 0.085 |
| 75 | 0.905 | 0.103 | 0.972 | 0.15 | 0.949 | 0.169 |
| 100 | 0.895 | 0.134 | 0.933 | 0.162 | 0.86 | 0.53 |

Round 3: Sweeping Learning Rate

| Learning Rate | X Position | | Y Position | | Heading | |
|---|---|---|---|---|---|---|
| | $R^2$ | MSE | $R^2$ | MSE | $R^2$ | MSE |
| 0.01 | 0.973 | 0.026 | 0.988 | 0.053 | 0.985 | 0.055 |
| 0.005 | 0.972 | 0.034 | 0.994 | 0.024 | 0.988 | 0.035 |
| 0.001 | 0.99 | 0.012 | 0.997 | 0.007 | 0.991 | 0.031 |

Round 4: Sweeping # Epochs

| # Epochs | X Position | | Y Position | | Heading | |
|---|---|---|---|---|---|---|
| | $R^2$ | MSE | $R^2$ | MSE | $R^2$ | MSE |
| 75 | 0.989 | 0.013 | 0.998 | 0.008 | 0.99 | 0.041 |
| 100 | 0.993 | 0.006 | 0.996 | 0.014 | 0.994 | 0.017 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 125 | 0.99 | 0.013 | 0.997 | 0.006 | 0.993 | 0.021 |

Final Optimized model Parameters:

# fully connected layers = 6
# input/output channels = 25
learning rate = 0.001
# epochs = 100

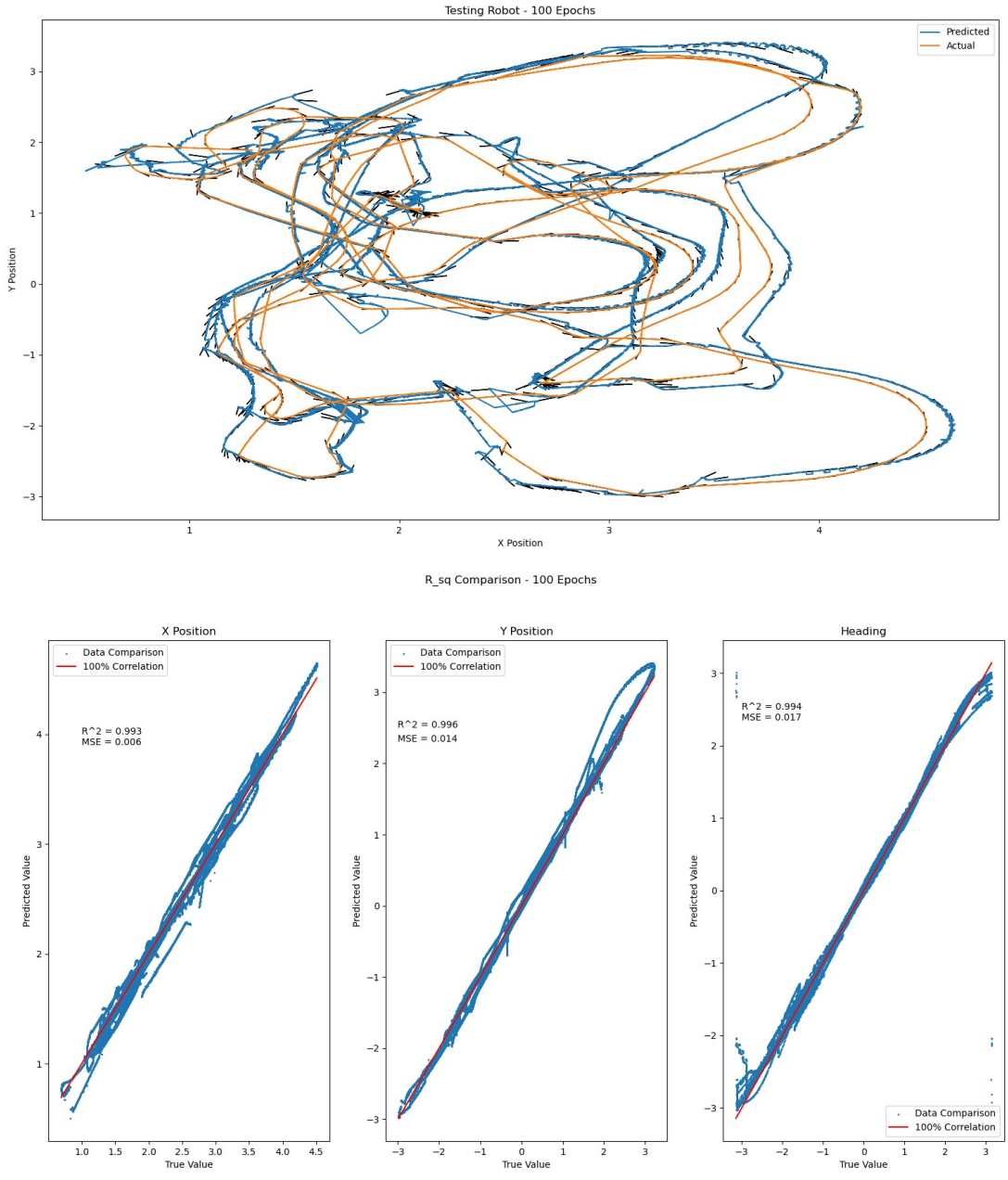



*Figure 3: Optimized parameters specific to the robot data and learning aim*

| # input/output channels | X Position | | Y Position | | Heading | |
|---|---|---|---|---|---|---|
| | $R^2$ | MSE | $R^2$ | MSE | $R^2$ | MSE |

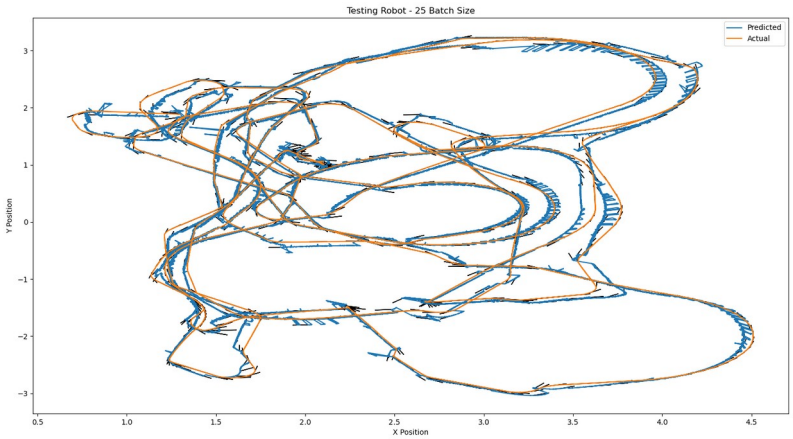
| | | | | | | |
|---|---|---|---|---|---|---|
| Sine Optimal Parameters | 0.968 | 0.026 | 0.992 | 0.023 | 0.982 | 0.084 |
| Aim Optimal Parameters | 0.993 | 0.006 | 0.996 | 0.014 | 0.994 | 0.017 |

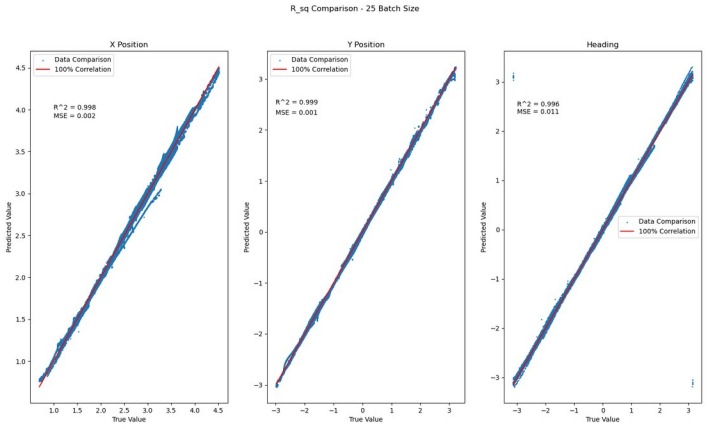
The optimized model results can be seen in the table above. Compared to the Optimal Sine parameters, there is a distinct change in the model's ability to predict the edge cases in the heading which was a goal for reoptimizing the parameters. There is also a change in the model's "noisiness" which resulted in a smoother path in the more optimal path estimated shown above. This observation confirms my hypothesis that a smaller learning rate would allow for smaller steps and lead to less noise. However, I was surprised to find that fewer input and output channels performed better than more when moving towards the larger data set.

While running this optimization, I noticed that run times were longer than I anticipated. While a major disadvantage to neural networks is their computationally expensive training time, there have been different way to combat this, such as mini-batching. This entails back propagating after a set of data points have been passed forward through the model instead of after every sample, reducing the number of computations required.

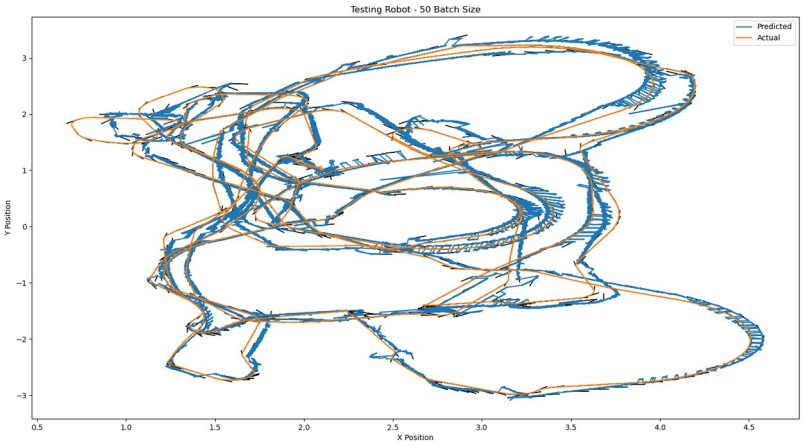
I swept through batch sizes of 25, 50, 75, and 100 and found the following results:

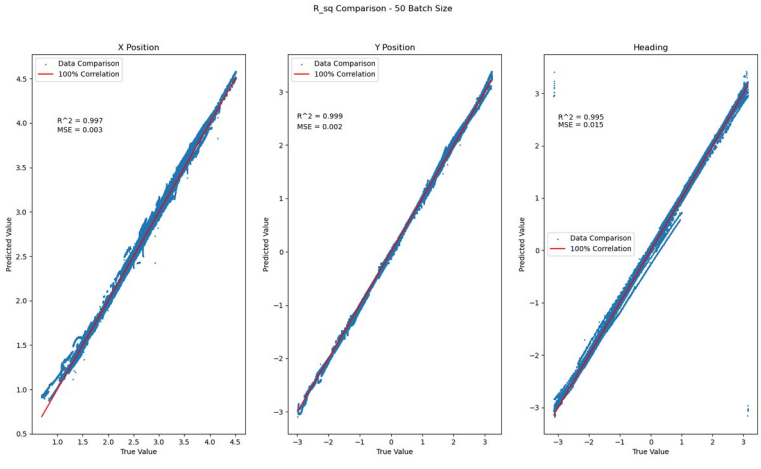
Batch size = 25



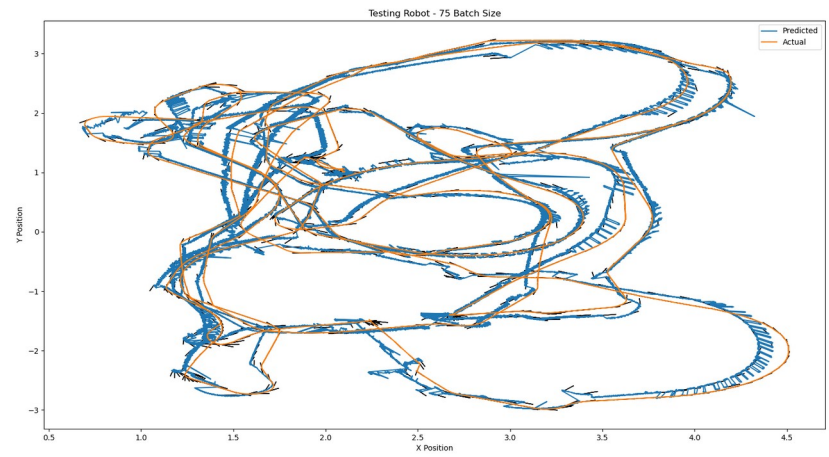



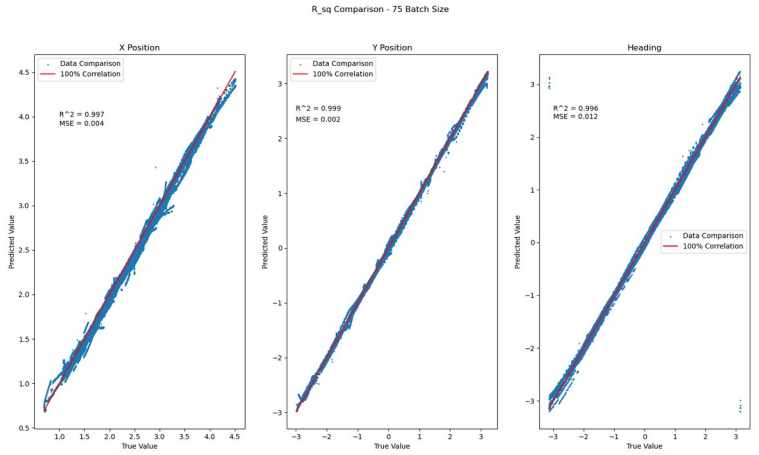

Batch size = 50



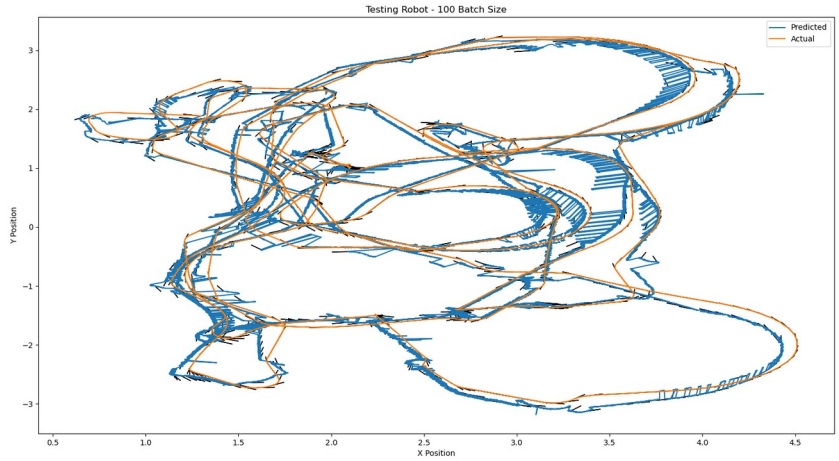



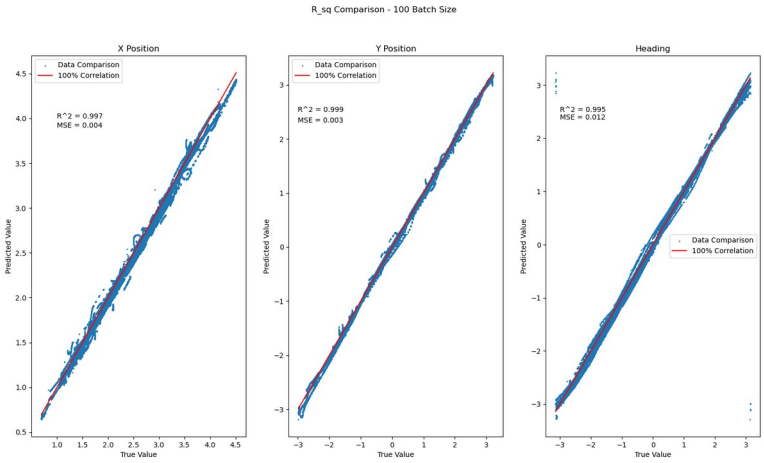
Batch Size = 75




Batch Size = 100




Seen in the figures above, as the batch size increases, there is more noise in the prediction of the model. The most optimal batch size was 25. While this decreased overall model computation and training runtime, there is the trade off between stability of prediction during testing and real runtime.

The model performed better than I anticipated with after tuning. I think other ways to improve the model to make it more generalized would be in the way we give it training data. In research, models are trained by one set of data and tested on data not included in the training dataset through the process of k-folding. This would prevent overfitting the data and inflated MSE and $R^2$ results. Additionally, for a model to be truly stochastic, the given data should be randomized instead of fed in sequentially. These changes to training the model could provide a clearer insight to how well the model can be generalized to the learning aim.