Predicting Chaotic Double Pendulum

Jacob Ryan

12/18/2020

Abstract

A double pendulum is a simple physical system consisting of a pendulum with another pendulum attached to its un-fixed end. However, this simple system can result in chaotic movement that is difficult and computationally expensive to predict. I propose using a physics assistant neural network to predict the movement of a chaotic double pendulum.

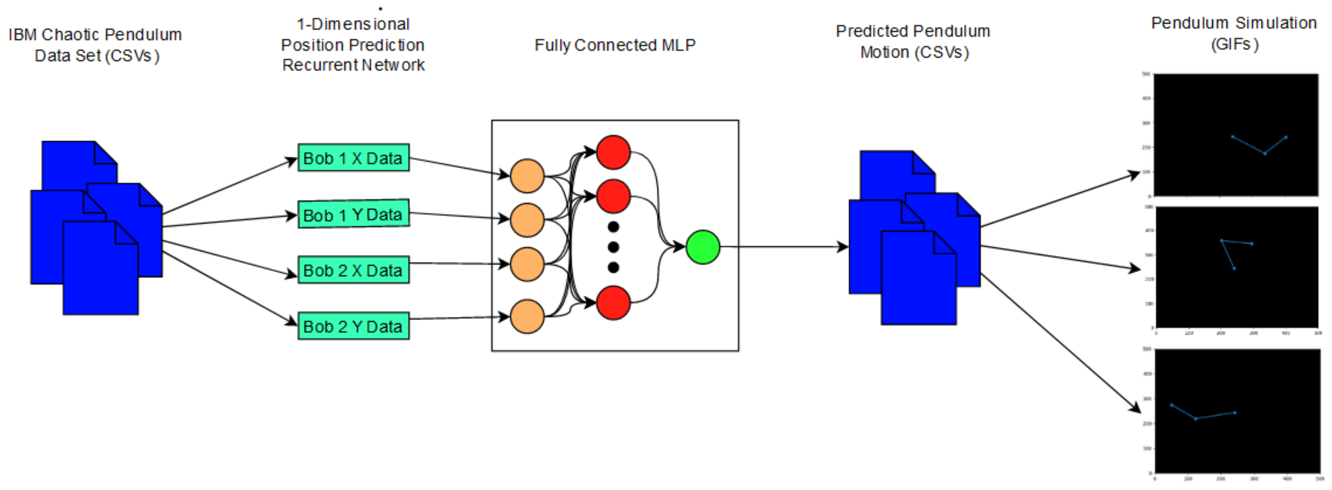
A multi-layer perceptron (MLP) neural network will be trained on a dataset of observed positions of a double pendulum under chaotic motion along with a mathematical model that predicts the motion of the pendulum as well. The network will perceive the x and y coordinate positions of both of the pendulum's bobs for each time step. The MLP shall be fully connected and should observe the position input from a set of 4 recurrent neural networks that predict the position of each bob.

Backpropagation networks have been shown to been able to model functions (I.E. sin waves) much more accurately that previous neural network designs. Applying this type of training to a prediction over time problem proved challenging. The double pendulum provides a simple model that can be visualized easily, but provides a complex and rich movement that proved too challenging for the RNN to accurately predict the motion of bobs over time steps. While prediction of more consistent movement patterns such as a sin wave were possible, the same techniques were not able to predict the movement of a pendulum's bob.

Outline

Backpropagation over MLPs has been shown to be able to model functions with consistent movement such as sinusoidal waves. However, the mapping of these functions does not reflect the temporal nature. In order to aid the MLP in returning an accurate position for the position of both pendulum bobs, two recurrent networks could be used for each bob (a total of 4) to predict the upcoming position of the bob. The output of the RNNs will not take into account the position of their paired X or Y value nor the coordinate position of the other bobs. Ideally, the output from the four RNNs will serve as effective estimates that can be directly passed into the MLP once all the models are trained.

To implement this design, the network is passed a dataset generated from a physics-based model describing the motion of a simulated pendulum for 60 seconds. Once the network has been sufficiently trained on the artificial model, it can be passed the real-life data at a lower learning rate. Real-life data proves to be more chaotic than the output from the physics generated model, and a lower learning rate will prevent outliers in real-life movement from damaging the models ability to predict.

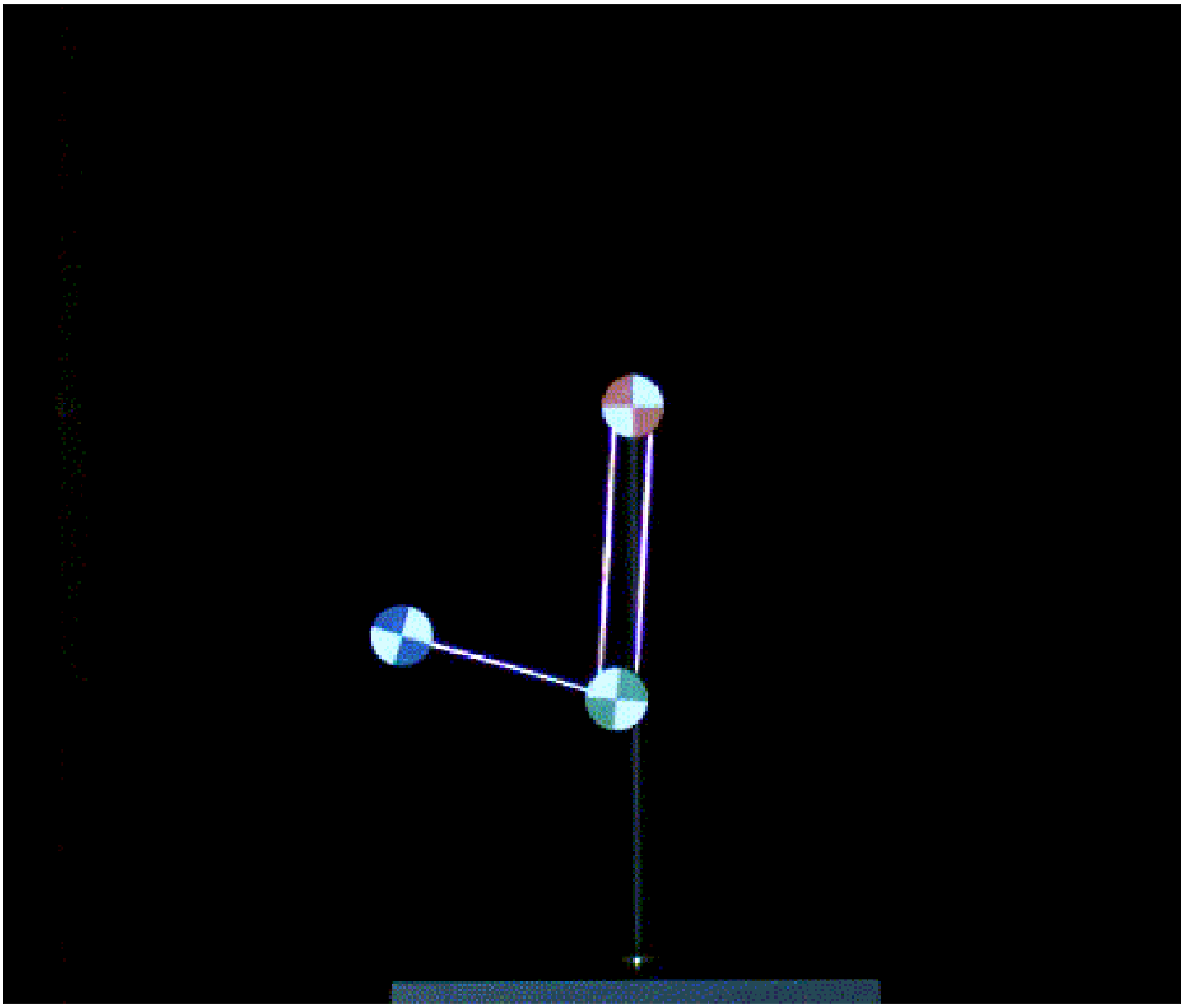

IBM Chaotic Pendulum Data Set (CSVs)
1-Dimensional Position Prediction Recurrent Network
Fully Connected MLP
Predicted Pendulum Motion (CSVs)
Pendulum Simulation (GIFs)
Bob 1 X Data
Bob 1 Y Data
Bob 2 X Data
Bob 2 Y Data

The design is implemented using the following tools and hardware:

> Software:
>
> - Python 3.7.0
> - Jupyter Notebooks 4.6.3
> - Numpy 1.19.4
> - Matplotlib 3.3.3
> - IBM Double Pendulum Chaotic Dataset
>
> Hardware:
>
> - Dell XPS 15 9570
> - Microsoft Windows Pro
> - Intel i7-8760H
> - 16.0 GB RAM

Data Set

The network will be trained using a IBM's "Double Pendulum Chaotic" dataset that features position data measured from various real-life recordings of a double pendulum. A frame from one of these recordings is included below:



> Real time footage is available from IBM's repository linked in the bibliography.

While the data set is generated from a series of videos, the network will be trained using a collection of CSV files containing x and y position data for the pendulum. In addition to this real-life data set, a physics-based model to predict the pendulum has been derived. The physics based model creates a simulated path of a pendulum starting from rest using the initial position for each of the real-life runs.

The network will be trained to minimize the total error between the actual dataset and the error with physics based model. The RNN trains using the actual input to generate accurate weights for each timestep. The MLP trains using outputted data from the RNN and minimizes the difference between the predicted data and the expected position for each timestep.

Deriving a Physics Guided Model

The first week of progress was devoted to generating a set of data that's predicted by a physics based model for the when it comes time to train the neural network. There were multiple elements that prevented generation of a physics based model that approximated the movement of the real-life dataset. While the final model provides accurate motion of a pendulum, it does not follow the path of the real life recordings. The cause of this error will be discussed further below.

As the double pendulum is a simple physical system, very little information is required to derive the anticipated position of the pendulum.

The original dataset provided by IBM captures pixel-position data for the three points on the double pendulum. To calculate how the pendulum will continue moving from any given point we need six values:

1. θ_1: angle between limb 1 and the vertical axis
2. θ_2: angle between limb 2 and the vertical axis
3. M_1: mass of bob 1
4. M_2: mass of bob 2
5. v_1: speed of the center of mass of bob 1
6. v_2: speed of the center of mass of bob 2

To obtain the angles from our original position data, we first obtain the length of both pendulum arms through:

- $L_1 = \text{dist}\{(x_1, x_2), (y_1, y_2)\}$
- $L_2 = \text{dist}\{(x_2, x_3), (y_2, y_3)\}$

From here we can calculate θ_1 & θ_2:

- θ_1: $\arcsin((y_2 - y_1)/L_1) + \Pi/2$
- θ_2: $\arcsin((y_2 - y_1)/L_1)$

Outside of position data, the original dataset does not contain information such as the starting velocity of either bob, nor the distributed mass of each pendulum arm. For this reason, it was not possible to match the path of the simulated dataset to that of the real life model within the time that was available for the completion of the project.

Each simulated run uses a fixed mass (M_1 & M_2) of 1.0 kg for each bob and assumes the pendulum is moving from rest (V_1 & V_2 = 0 m/s^2). Using these variables, simulated movement can be derived using the lagrangian to describe the state of the system at each timestep.

The lagrangian is determined as follows:

- L = Kinetic Energy - Potential Energy
- $= 0.5(v_1^2 + v_2^2) + 0.5*\textit{Inertia}*(\dot{\theta}_1^2 + \dot{\theta}_2^2) - mg(y_1 + y_2)$
- $= (1/6)*ml^2(\dot{\theta}_2^2 + 4\dot{\theta}_1^2 + 3\dot{\theta}_1\dot{\theta}_2\cos(\dot{\theta}_1 - \dot{\theta}_2)) + 0.5 * mgl(3\cos\dot{\theta}_1 + \cos\dot{\theta}_2)$

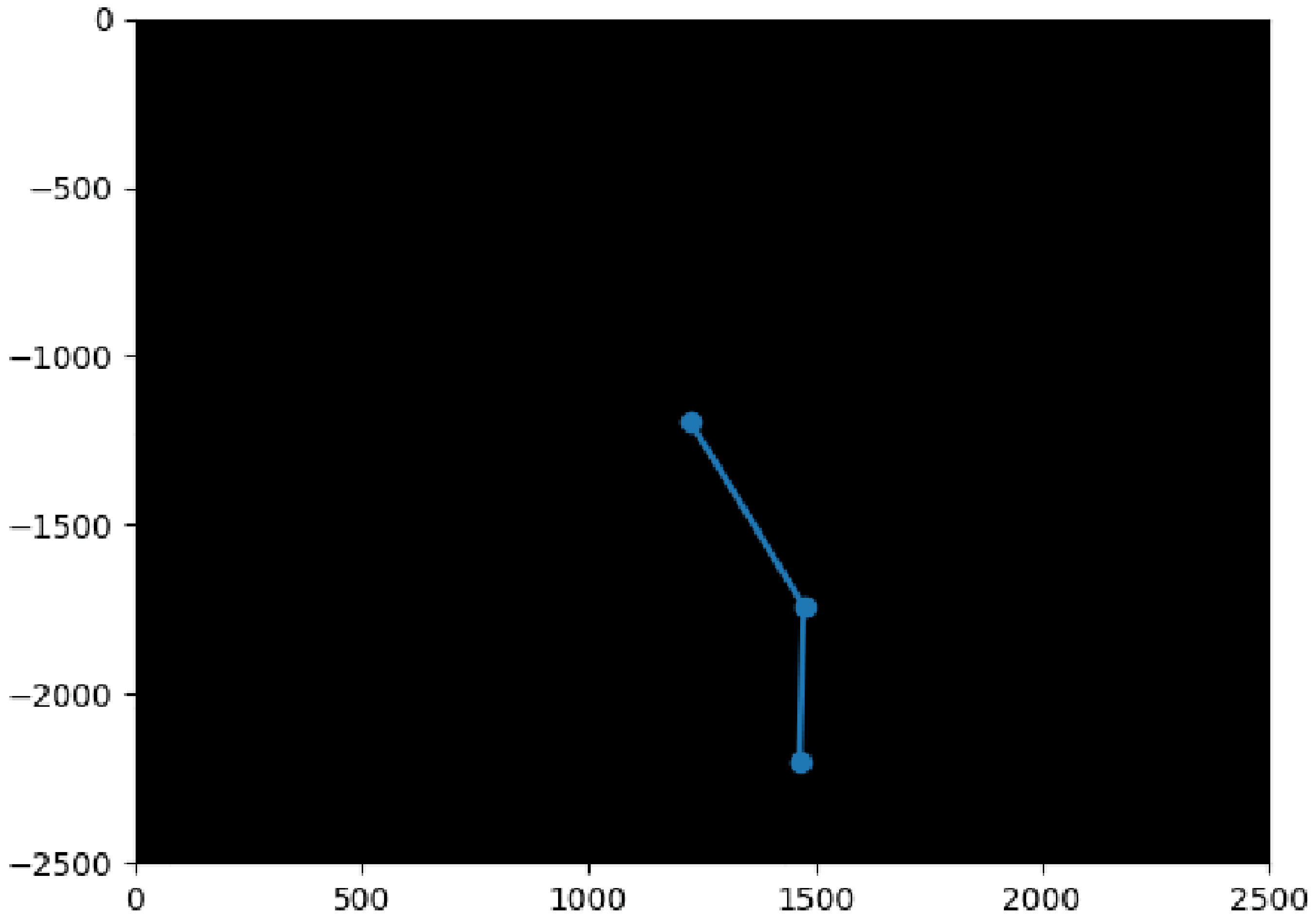
> This level of physics and differential equations is beyond my level of ability, so I'm going to point you towards the Wikipedia page for the continued derivation of the formula. I just wanted to demonstrate that we can get a system of equations that will be solvable to get the new momenta and theta values for each time interval.
>
> https://en.wikipedia.org/wiki/Double_pendulum

To generate the data set that will be used for training the network, we'll use `scipy`'s integrate function to derive the momenta and theta values for each time stamp. After having generated the theta values for each time interval, they can be translated back into x and y values in the same form as the original real life dataset.

The function that generates the equations to be derived can be seen below:

```python
def derivs(state, t):
    dydx = np.zeros_like(state)
    dydx[0] = state[1]

    del_ = state[2] - state[0]
    den1 = (M1 + M2)*L1 - M2*L1*cos(del_)*cos(del_)

    dydx[1] = (M2*L1*state[1]*state[1]*sin(del_)*cos(del_) +
            M2*G*sin(state[2])*cos(del_) +
            M2*L2*state[3]*state[3]*sin(del_) -
            (M1 + M2)*G*sin(state[0]))/den1

    dydx[2] = state[3]
    den2 = (L2/L1)*den1

    dydx[3] = (-M2*L2*state[3]*state[3]*sin(del_)*cos(del_) +
            (M1 + M2)*G*sin(state[0])*cos(del_) -
            (M1 + M2)*L1*state[1]*state[1]*sin(del_) -
            (M1 + M2)*G*sin(state[2]))/den2

    return dydx
```

Here's an example of a double pendulum I generated using this method:



The Network

Originally Proposed Network

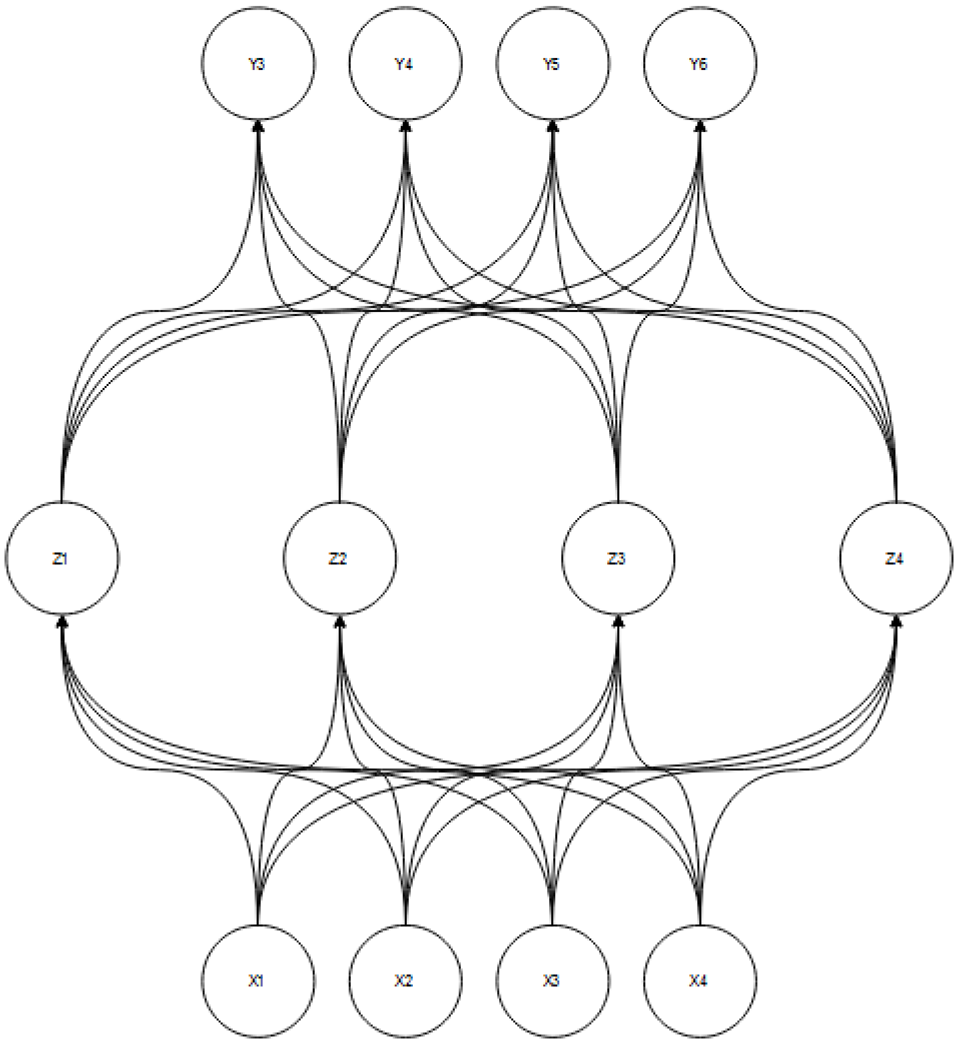
I'll now go into the process of designing the network, the original plan, adaptations that were made, and the final design. The original network design was a multilayer perceptron without iteration of time. Having worked with this type of model in previous work to model the product of two sinusoidal variables, I proposed adapting a similar model to predict the motion of the double pendulum. This original network was also going to be trained using a compound loss function which would combine the difference of the expected position value from the predicted position value with the difference of the expected position value from the physics-guided model.

$$\arg\min_{f} \quad \underbrace{Loss(\hat{Y}, Y) \;+\; \lambda\, R(f)}_{\text{Typical loss function}} + \underbrace{\lambda_{PHY}\; Loss.PHY(\hat{Y})}_{\text{Physical Inconsistency}}$$

> Image Credit to Karpatne, Anuj, et al. - (PGNNs)

The use of a physics guided loss function was proposed as a way to maximize the effectiveness of training by using the existing method of approximating pendulum motion with the ability of the network to train on unlabeled data. (Karpatne)

When it became apparent that it would no longer be possible to model the movement of the real life dataset using the physics-generated data, the compound loss function had to be abandoned for the purpose of this project. Instead, the network would have to learn the motion of the pendulum through the physics based model and refine it's weights further using the real-life data. The physics model does well to anticipate the motion of a non-chaotic pendulum with well-distributed weight, but real-life conditions make creating accurate predictions quite difficult.

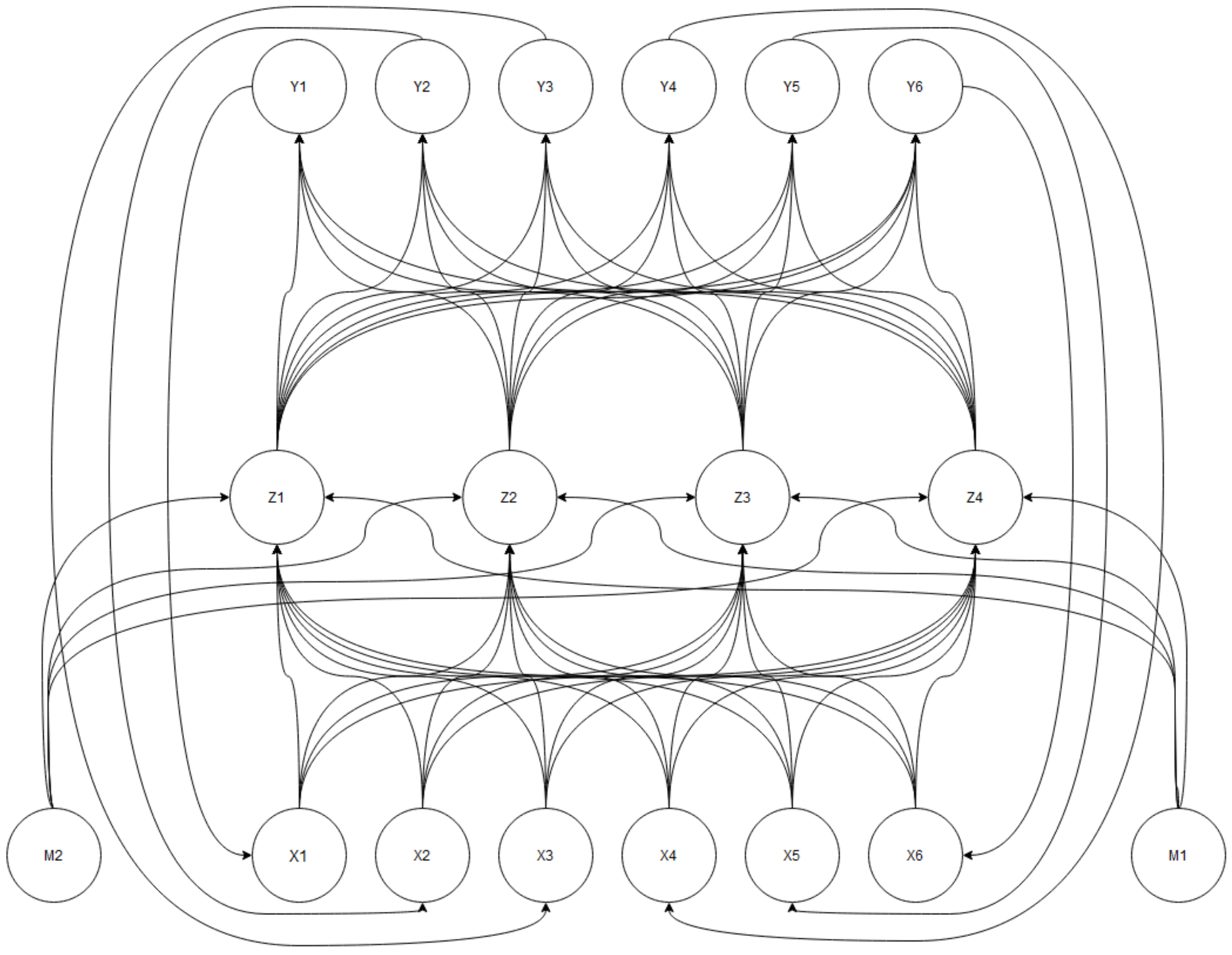


> A diagram describing the model for the MLP can be seen above

Additionally, the requirement that the network should be able to fully generate the arc for a double-pendulum, it became apparent that predicting the next time interval would require output units being passed back to the input units. For this reason, I temporarily stepped away from this design and moved onto the next iteration of this project's design.

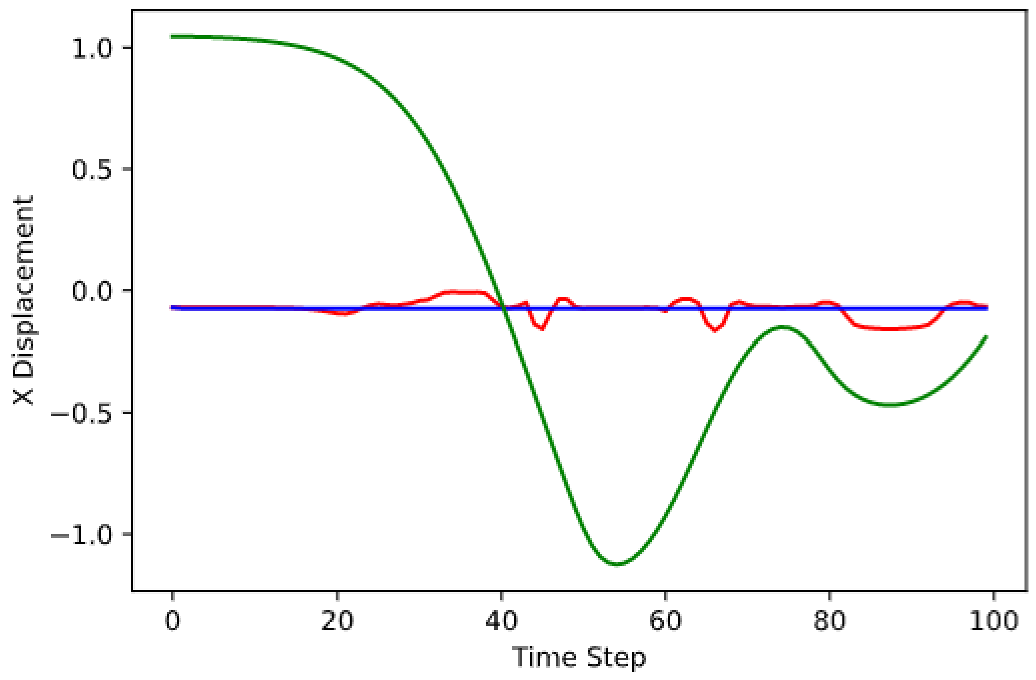
First RNN design

Further reading about backpropagation networks for unique time steps, it became apparent that a recurrent multilayer neural network would be needed for the project. For this network, the output from each time step would be fed to the input units a the next time step (Fausett). This new network designed would also remain fully connected outside of this change. Adapting the MLP to an RNN required calculating the loss over each time step and then applying the weight changes at the end of each epoch. Each time step would share in the same weights matrix (Fausett).



This model had a few issues which I would like to note:

1. Input values for mass were consistent over every run. As they didn't change and couldn't be used for testing on the real-life dataset, they effectively served as biases that would have been removed at later stages in the development process. Inputting M_1 and M_2 added computational strain to the network without adding any value.
2. X_1 and X_2 were inputs representing the X and Y coordinate position of the fixed bob on the pendulum. In the physics generated model, they're value would remain zero at all times. On real life data, imperfections in the tracking of position data from video caused them to slightly drift from the origin. Due to the inter-connected nature of the network, they merely functioned as additional noise to the network and should not have been allowed to impact output.
3. Inputting all the position data never resulted in any kind of learning from the network. Due to how different each output would be at separate time intervals, it was difficult for the network to learn any kind of pattern to anticipate. The network would instead rapidly converge the weights to 0. Below is an outputted graph highlighting this kind of behavior.



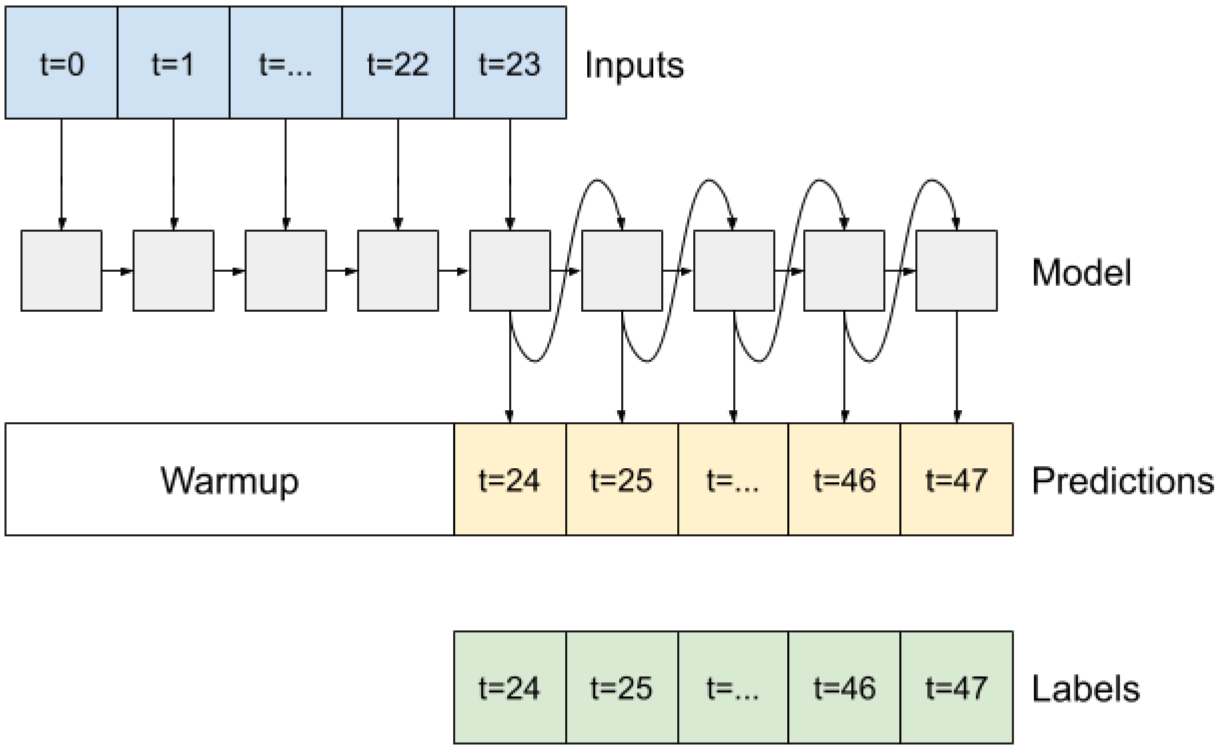
> The blue line indicates the fully predicted recurrent output. The red line is generated output from training data inputed at each time step. Green is the actual graph of the displacement of the second bob along the x axis from the origin.

Inability to train the network to respond with any kind of trend on this configuration led me to build on this design.

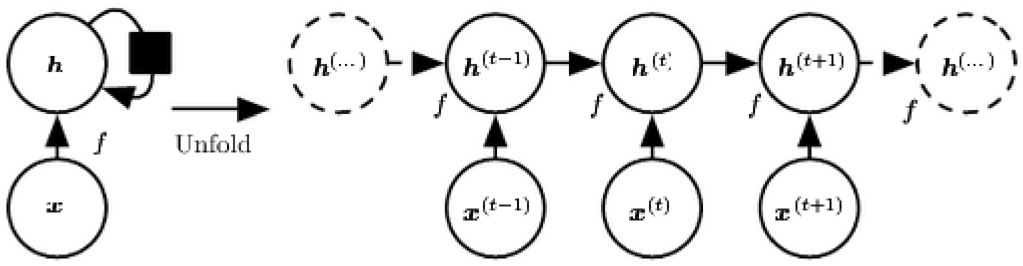
Final Configuration

I took a much more deliberate process to designing the final iteration of this network. Having been completely unsuccessful to produce any kind of result on the previous networks, I began to do a lot of reading into RNNs.

One of the first changes I decided to implement was data windowing. I came upon the concept for this idea while reading about time series forecasting on *Tensorflow*'s websites. While I was not using the *Tensorflow* utility for this project, I was able to learn about better way to leverage the previous timesteps to inform the next prediction. Data windowing entails configuring additional inputs to the network for each time step. These input values will be updated regressively as the network predicts the next time step. Training the network includes a warm-up period of time where the network does not generate any output. Instead, the first set of expected inputs are set for the time window. The input window acts in a FIFO fashion, moving the most recent timestep to the front and removing the oldest.



Further reading in Data science editorial, *towardsdatascience.com*, lead me to realize that the understanding I had of RNNs from reading in Fausett's textbook was incomplete. Their article title "Recurrent Neural Networks - RNN" described a third additional weights array. This new set of weights, U, between the individual nodes in the hidden layer.



The RNN generates the input to each node in the hidden layer in the same way as was described in Fausett's book. However, once the input value to each hidden node is defined, the input is passed through the weights, U, connecting the hidden nodes. The value of h at each time step in the current time window is added to the input value to each h node and passed through the activation function. This serves as the new value that is distributed across the weights connecting the output nodes to the hidden node.

Unlike the other weight matrices for the network, *U* is not fully connected. There is one weight in *U* connecting each hidden node *h* to the hidden node *h*(t+1). This can be seen in the form of the weight matrices when they're initialized for the network in the following code snippet:

```python
def __init__(self, n, p, m):
    self.n = n #Input Nodes
    self.p = p #Hidden Layer Nodes
    self.m = m # Output Layer Nodes
    self.v = [[random.uniform(-2, 2) for j in range(p)] for i in range(n + 1)]
    self.w = [[random.uniform(-2, 2) for j in range(m)] for i in range(p + 1)]
```

```
        # Hidden-to-hidden recurrent weights
        self.u = [random.uniform(-2, 2) for i in range(p)]
        # Short Term Memory for hidden nodes
        self.h = [0 for i in range(p)]
```

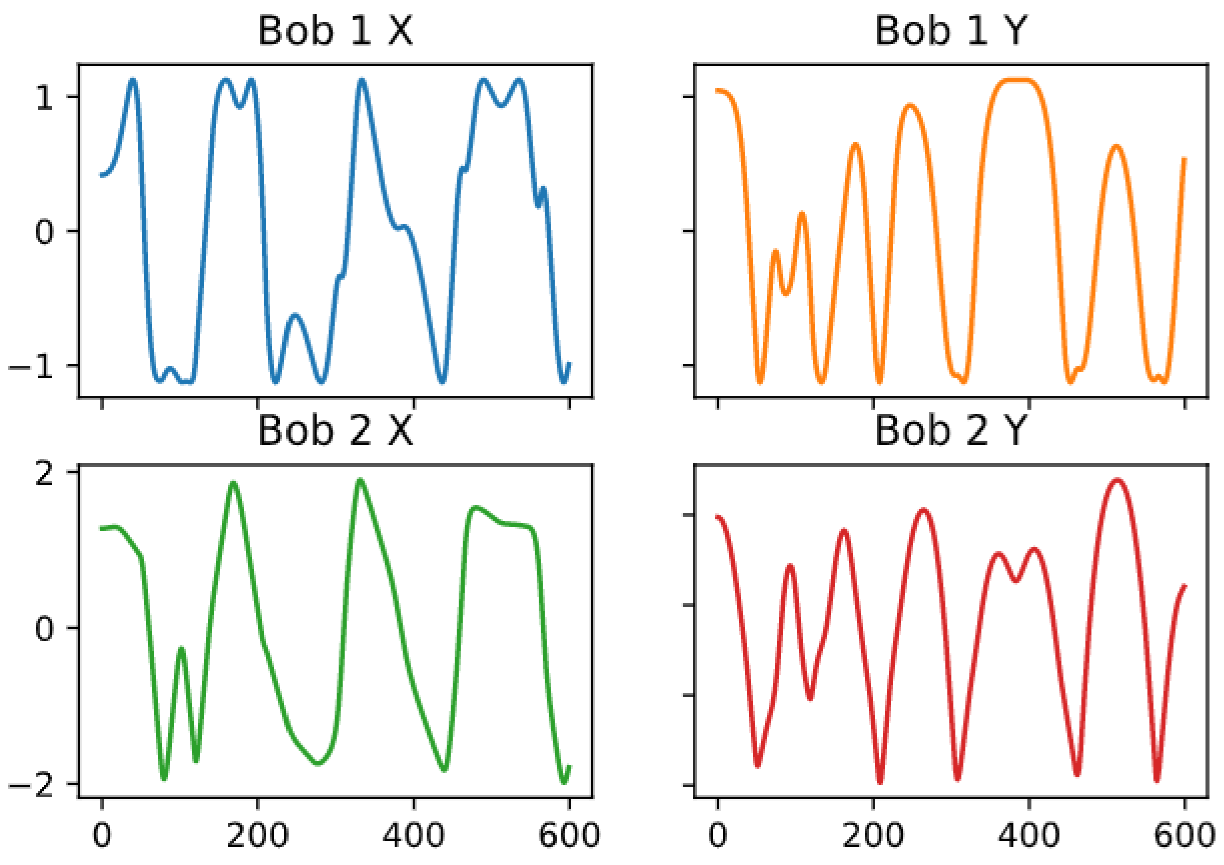
The code snippet from the feed_forward algorithm that implements this step can be seen below:

```
# h_t is the value of the h array at time(t)
# self.h is the value from the previous time step
# z is the array passed from the hidden nodes
h_t = np.add(z, np.dot(self.u, self.h))
h_t = [self.sigmoid(x) for x in h_t]
self.h = h_t
```

Backpropagation for the U weight matrix is done in the following code snippet:

```
 # Step 8 - Distribute error across Hidden Layer Weights
for j in range(1, self.p):
    error_in = error[0]*self.w[j][0]
    error_j = error_in * self.sigmoid_prime(z[j-1])
    # Weight correction term for hidden-to-hidden
    for l in range(1, self.p):
        delta_h[l][j-1] += learning_rate*error_j*h_t[l-1]
    delta_h[0][j-1] += learning_rate*error_j
```

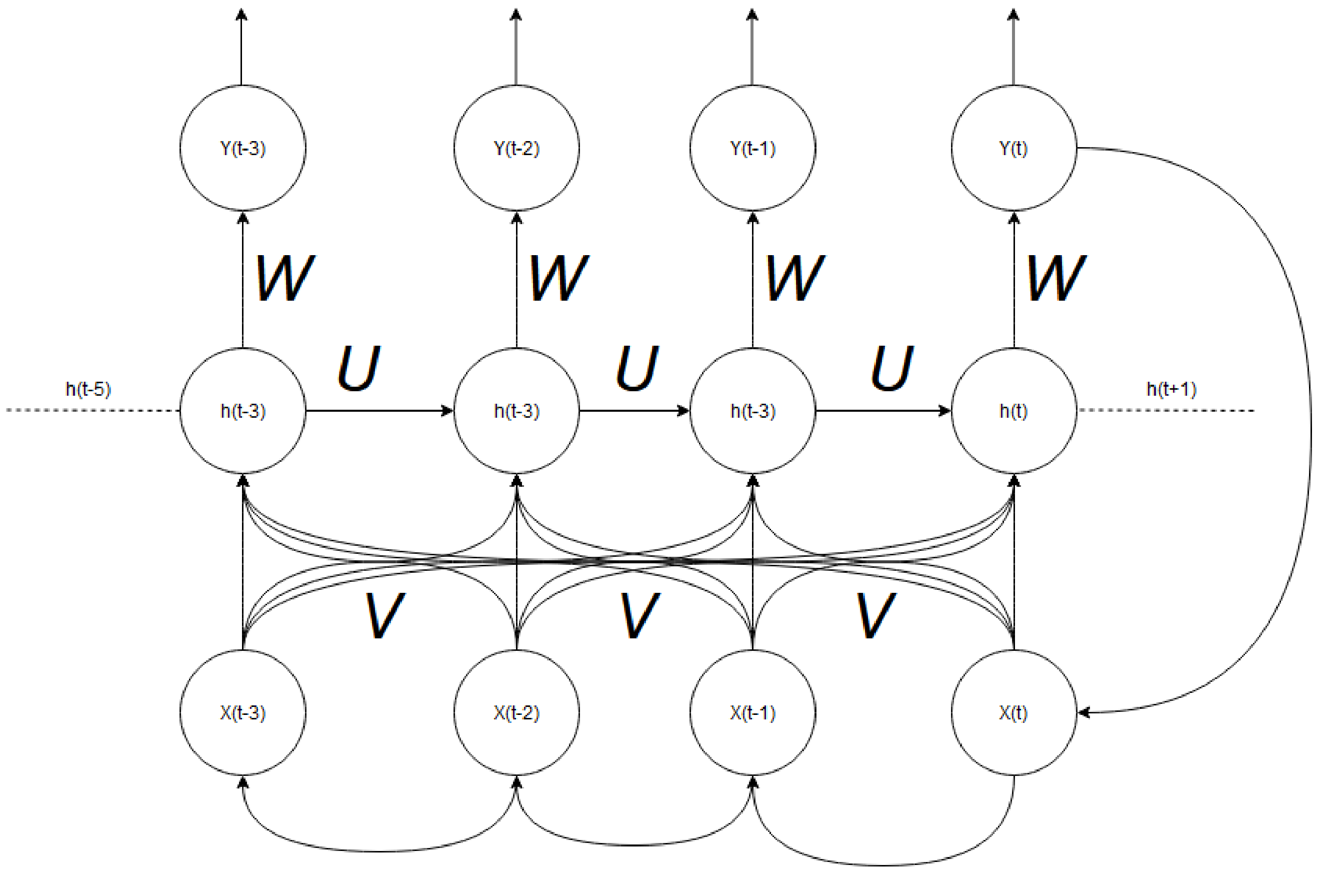
The largest change for this network was to divide the network into multiple RNNs, one for each coordinate. As I noted in the last design, the fully connected design for predicting the position of each network proved too difficult for the network to interpret. While it's difficult to anticipate the position of bobs when looking at all the inputs, each coordinate moves in a very similar pattern. By distributing the task of identifying each coordinate to individual RNNs, we can leverage the rhythmic pattern through which they move. Notice the sinusoidal pattern that the bob's axis move through in the figure below:



The decision to pass the output of these networks into a multi-layer perceptron (MLP) is based in the previously noted ability of an MLP to accurately map the product of two sin functions. Given the similar pattern that these x and y coordinates move, I had anticipated the network being able to correlate the predicted output of the X and Y coordinates together to provide more optimal results.

> It should be noted that this additional portion of the network was never implemented or tested due to time constraints. While it was an original part of the final design, it mostly functions as a way of smoothing any discrepancy that could occur from the 4 RNNs.

A diagram demonstrating the design of each RNN is visible below:



> The same network design is used for each of the 4 coordinate values: X_1, Y_1, X_2, and Y_2.

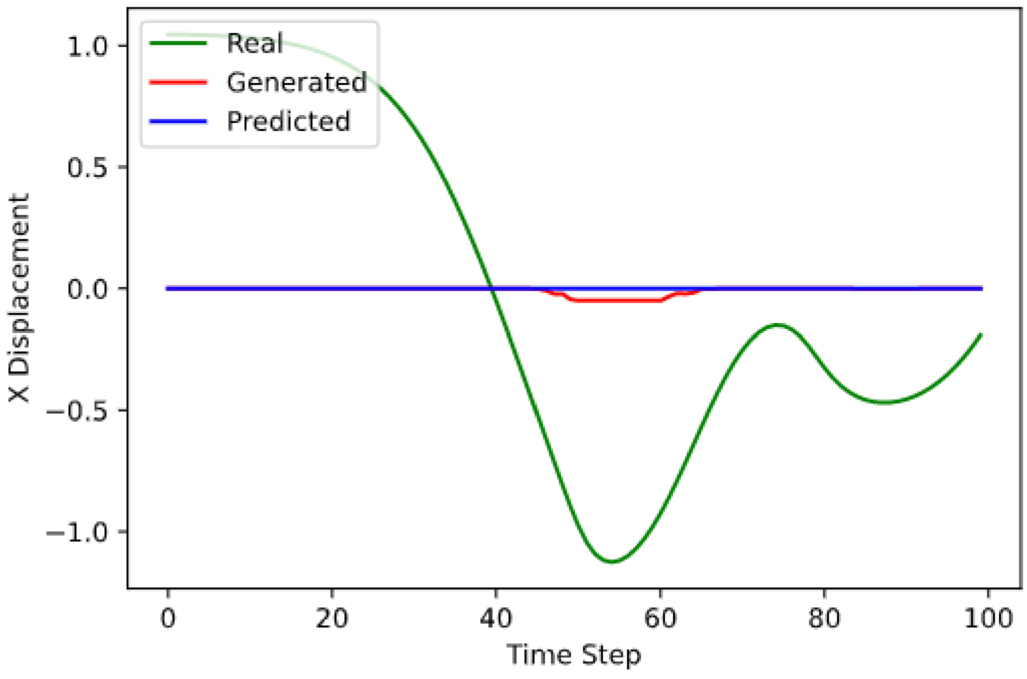
The feedforward algorithm that runs through the design in the diagram above can be read below:

```
def feed_forward(self, x):
    # print(x)
    z = []
    # Forward through hidden layer
    for j in range(self.p):
        z_in = self.v[0][j]
        z_in += sum([x[i-1]*self.v[i][j] for i in range(1, self.n + 1)])
        # Normalize Z_in to bring scale between
        z_in = self.normalize(x, z_in)
        # Apply activation function
        z_j = self.sigmoid(z_in)
        # Append to Z in order to broadcast to next layer
        z.append(z_j)
    # Hidden State Interaction
    h_t = np.add(z, np.dot(self.u, self.h))
    h_t = [self.sigmoid(x) for x in h_t]
    self.h = h_t
    y = []
    # Forward through outputs
    for k in range(self.m):
        y_in = self.w[0][k]
        y_in += sum([h_t[j-1]*self.w[j][k] for j in range(1, self.p + 1)])
        # Normalize Y_in
        z_in = self.normalize(z, y_in)
```

```
            # Apply activation function
            y_j = self.sigmoid(y_in)
            # y_j = y_in
            y.append(y_j)
        # Return position data from hidden & output layers
        return z, y, h_t
```

Results

Training proved to be a long and arduous process for the network, especially early on before use of the weights between nodes in the hidden layer. I began by testing the RNN design on the displacement of the X_2 bob. At this point, the network failed to generate any kind of learning response to the training set.



As you can see from the generated output, the network was unable to model any kind of response resembling the path of the X_2 bob over a training period of 1000 epochs and a learning rate of 0.001.

Originally, I began to think that the issue with the learning came from the path of the pendulum not following a consistent enough arc. In order to provide a training set with more periodic behavior, I generate a sin wave to start debugging.

I began to test the behavior of the network on the sin-wave dataset with different configurations. Originally, I was testing the network on a bounded activation function. The bounded function would output 0 for any output higher or lower than defined min max values. This activation function was easy to backpropagate as the derivative was the same as the activation function. A code snippet of this activation function can be seen below:

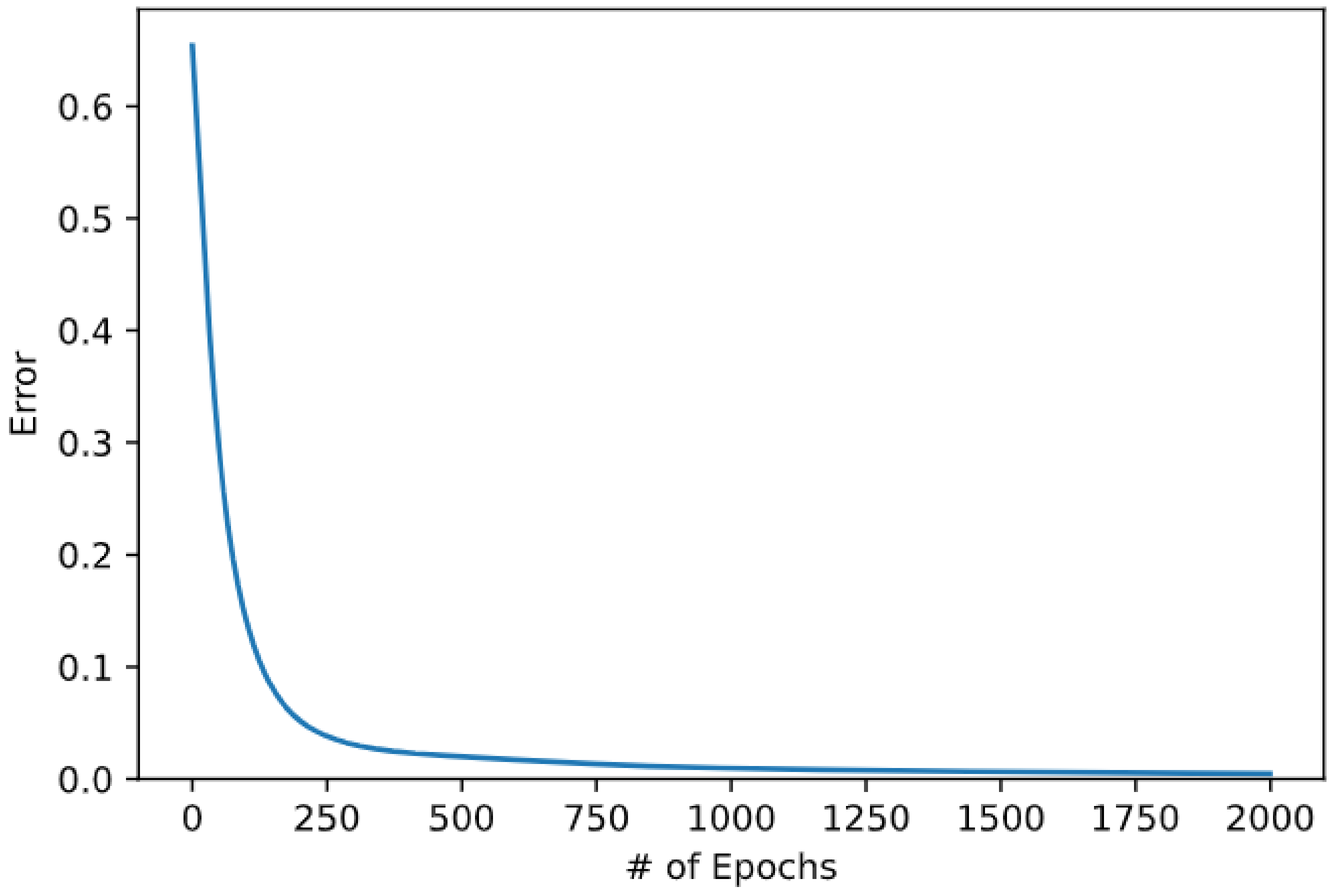
```
def bounded(x, min, max):
    if x > max or x < min:
        return 0
    else x
```

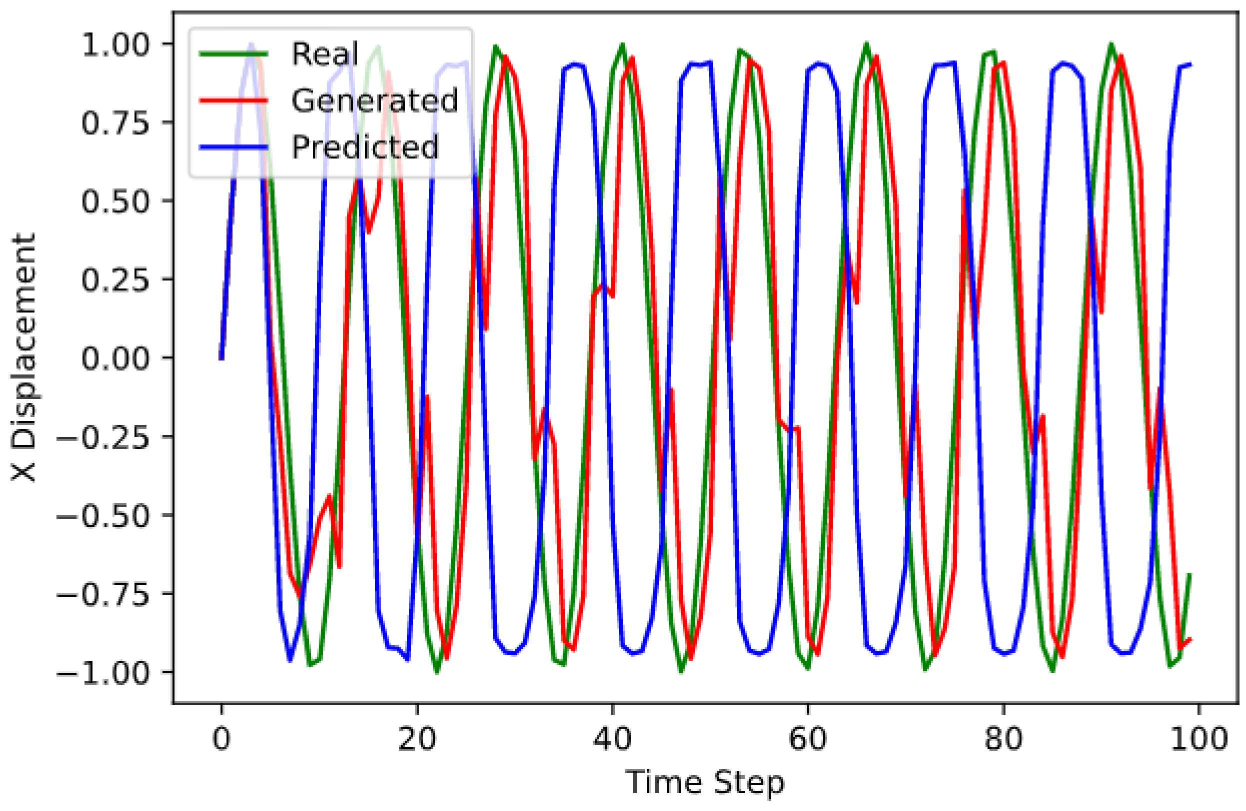
Unfortunately, I was seeing the same result as before. All the weights would rapidly approach zero and minimize any output value. My next attempt was to use a sigmoid function ranging between -1 and 1 for training the network. The sigmoid was the same activation function that I used in training a MLP to learn a sin-wave before, and I had hoped this would perform well for this application. However, I had to normalize my data to this range using a scaling function to take advantage of the sigmoid.

The following code was used for scaling data, and applying the sigmoid function and it's derivative:

```
def scale(value,min_r, max_r, min_t, max_t):
    return (value-min_r)/(max_r-min_r)*(max_t-min_t)+min_t

def sigmoid(self, x):
    x = np.clip(x, -1000, 1000)
    return (2/(1 + np.exp(-x))) - 1

# Expects x to already have been put through the sigmoid function
def sigmoid_prime(self, x):
    return .5*(1 + x)*(1 - x)
```

Using this change, I tried testing the network on the sin based dataset to the following result:

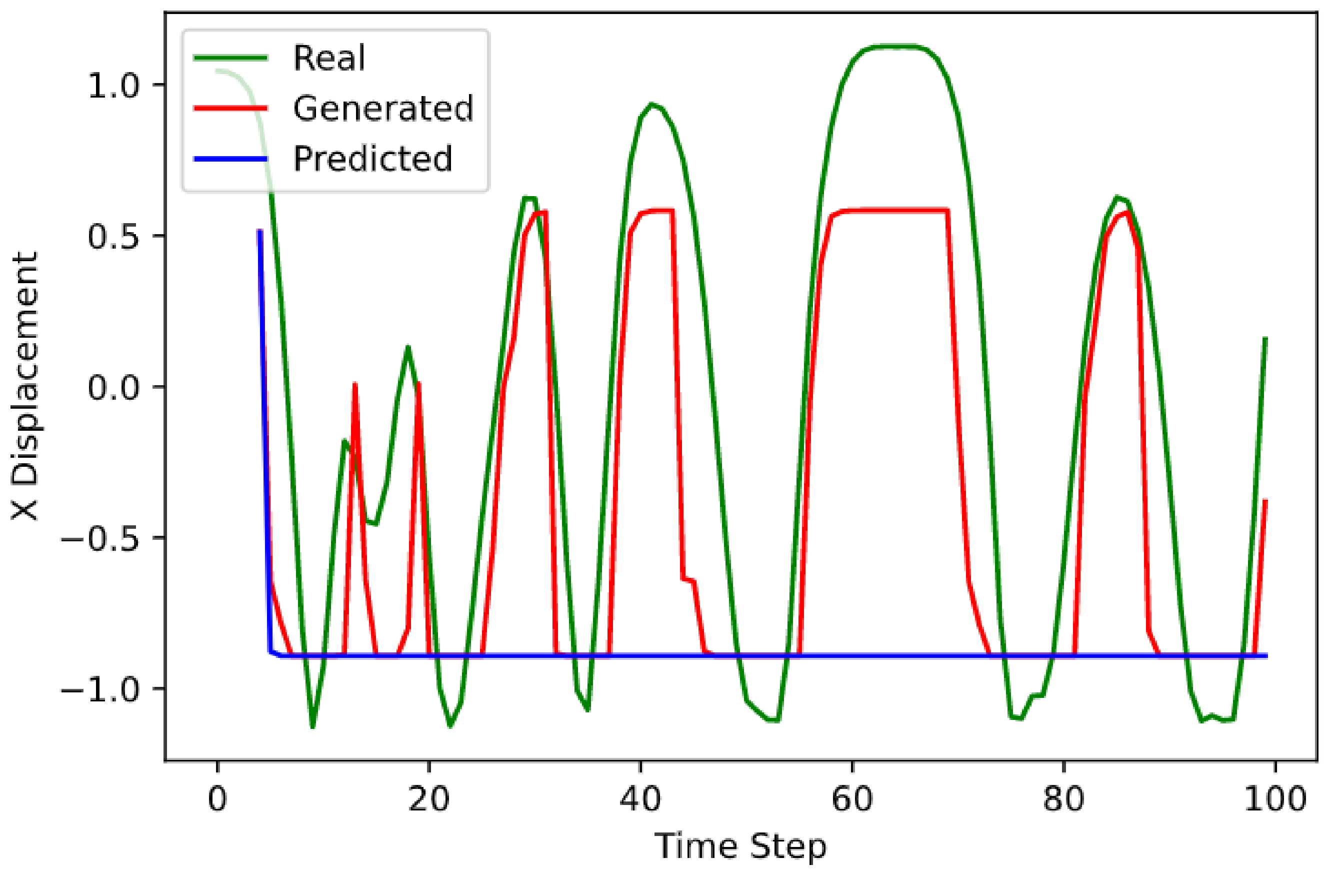




As you can see, the network was able to very accurately model this wave. Final epoch error was down to an MSE of 0.0048 on the 1900th epoch.

> The network was configured using the following configurations:
>
> - Learning Rate: 0.001
> - Time Steps = 100
> - Data Window = 4
> - Epochs = 2000
> - 4 Input Nodes
> - 4 Hidden Nodes
> - 1 Output Node
> - Nguyen-Widrow initialization

Having felt confident from these results, I moved on back to the X coordinate dataset. Unfortunately, the network still struggled to model the less periodic movement of the double pendulum. Additionally, it appears that the data was overfitting the weights. The network was able to return a pattern similar to the original data when inputting the training set data at each time step. However, the network was unable to create a continuous pattern following the pendulum's path when it had to use the output as the next input.

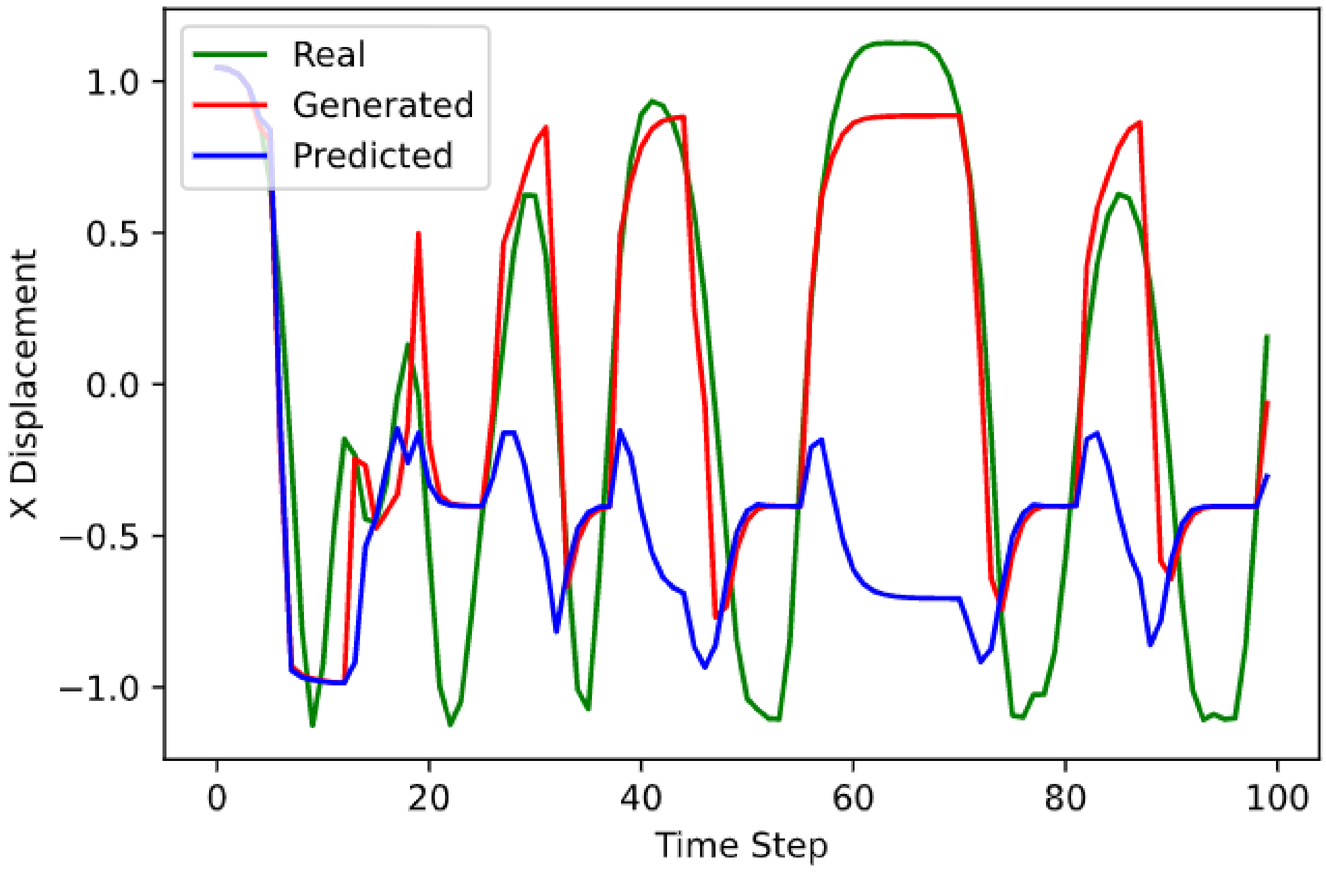


> The network was configured using the following configurations:
>
> - Learning Rate: 0.001
> - Time Steps = 100
> - Data Window = 4
> - Epochs = 2000
> - 4 Input Nodes
> - 2 Hidden Nodes
> - 1 Output Node
> - Nguyen Widrow initialization

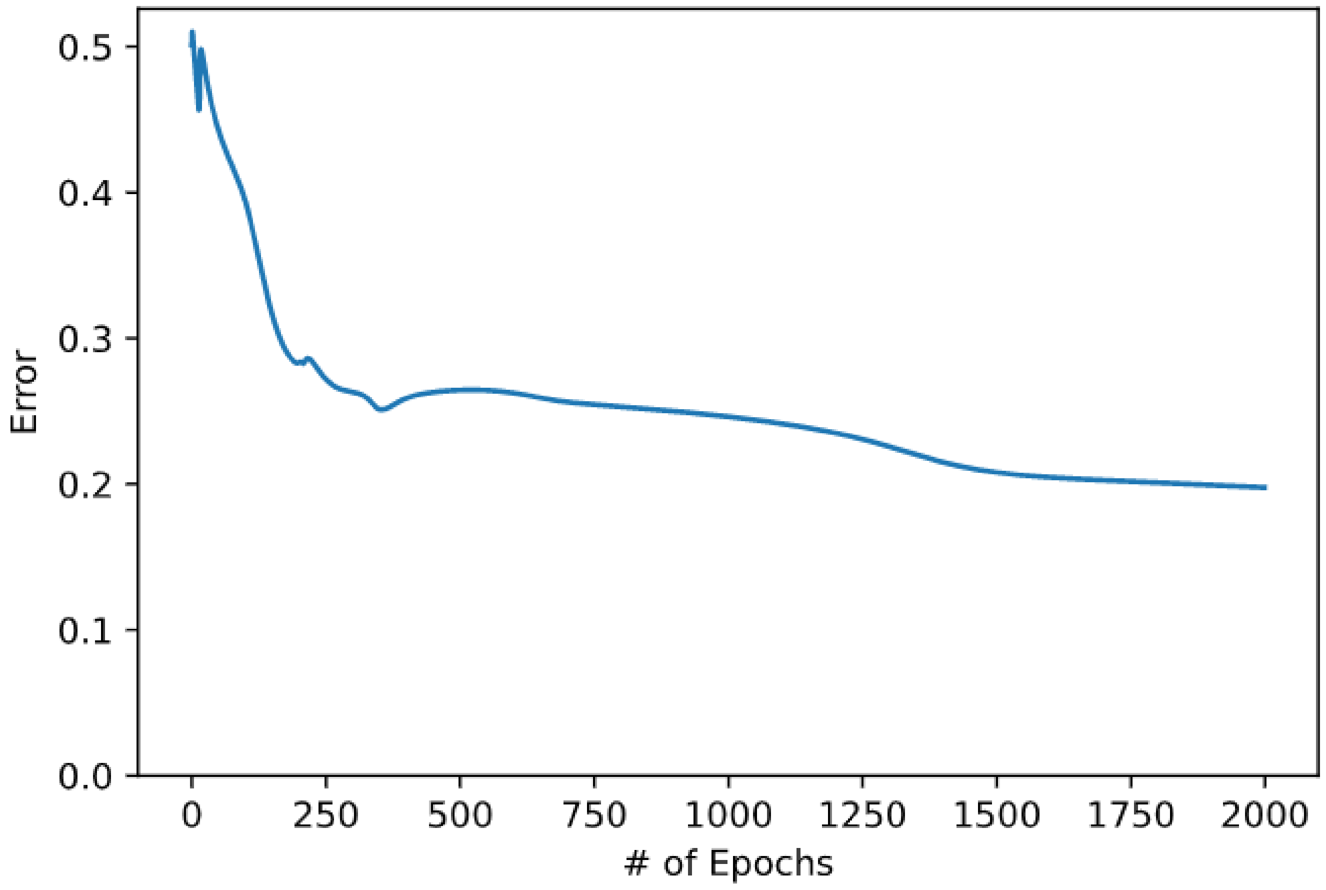
Moving forward from this pattern of result took a long time. For a long period of development, I was under the assumption that something was wrong with my activation function or my backpropagation of error. However, after many days worth of work put into correcting the data, I learned about configuring the network with the *U* weights matrix between each of the hidden nodes.

Fortunately, implementing the RNN structure I described in the "Final Design" section of this report, I saw a lot more success in modeling the data.

Using the same network configuration as described for the previous results, I was able to much more closely model the path of the X_2 bob.



As you can see, the predicted motion was much more akin to the movement of the bob. The network was beginning to learn. Now I need to narrow down the configuration that would provide the best possible results. I was looking to beat the MSE in the graph below:

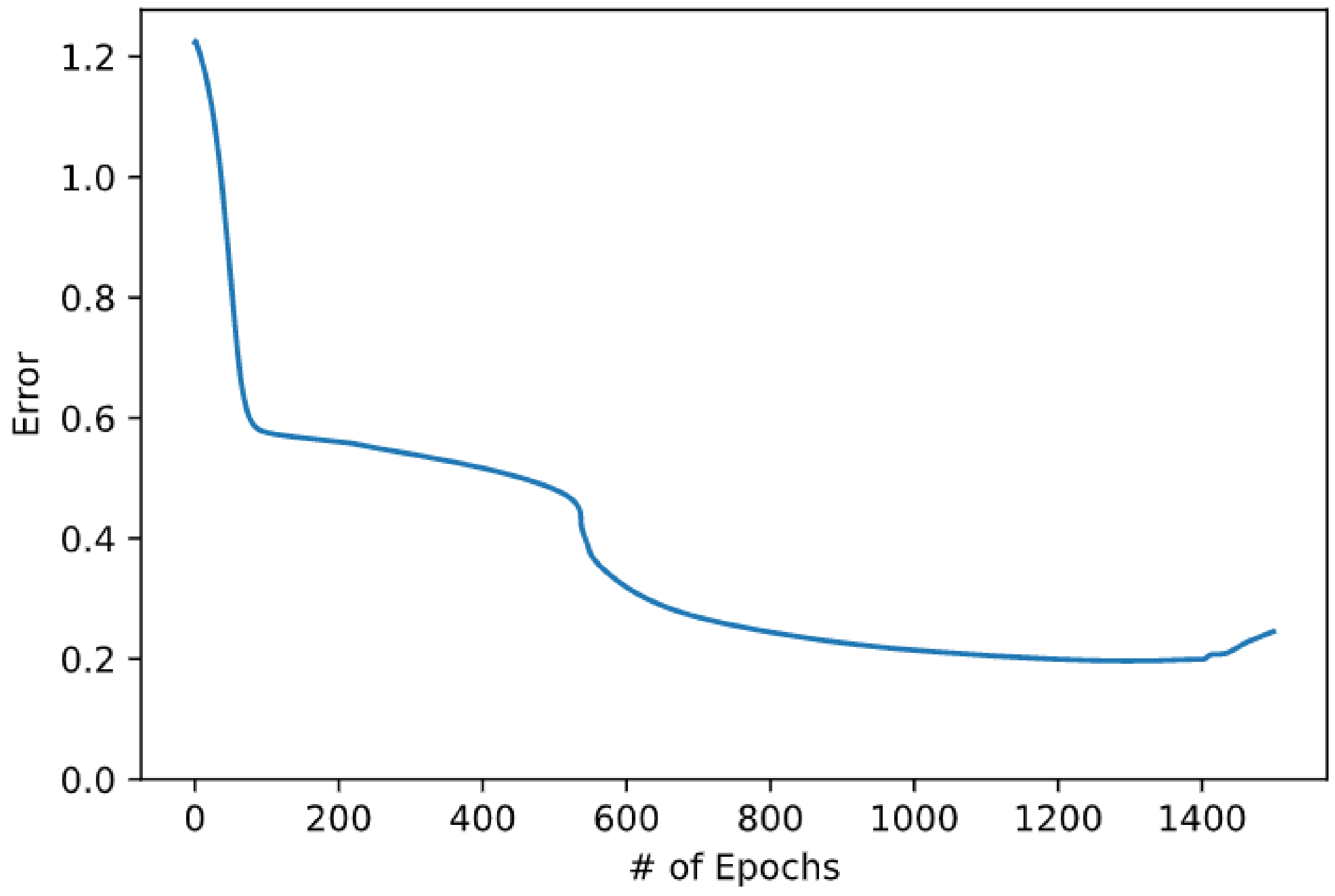


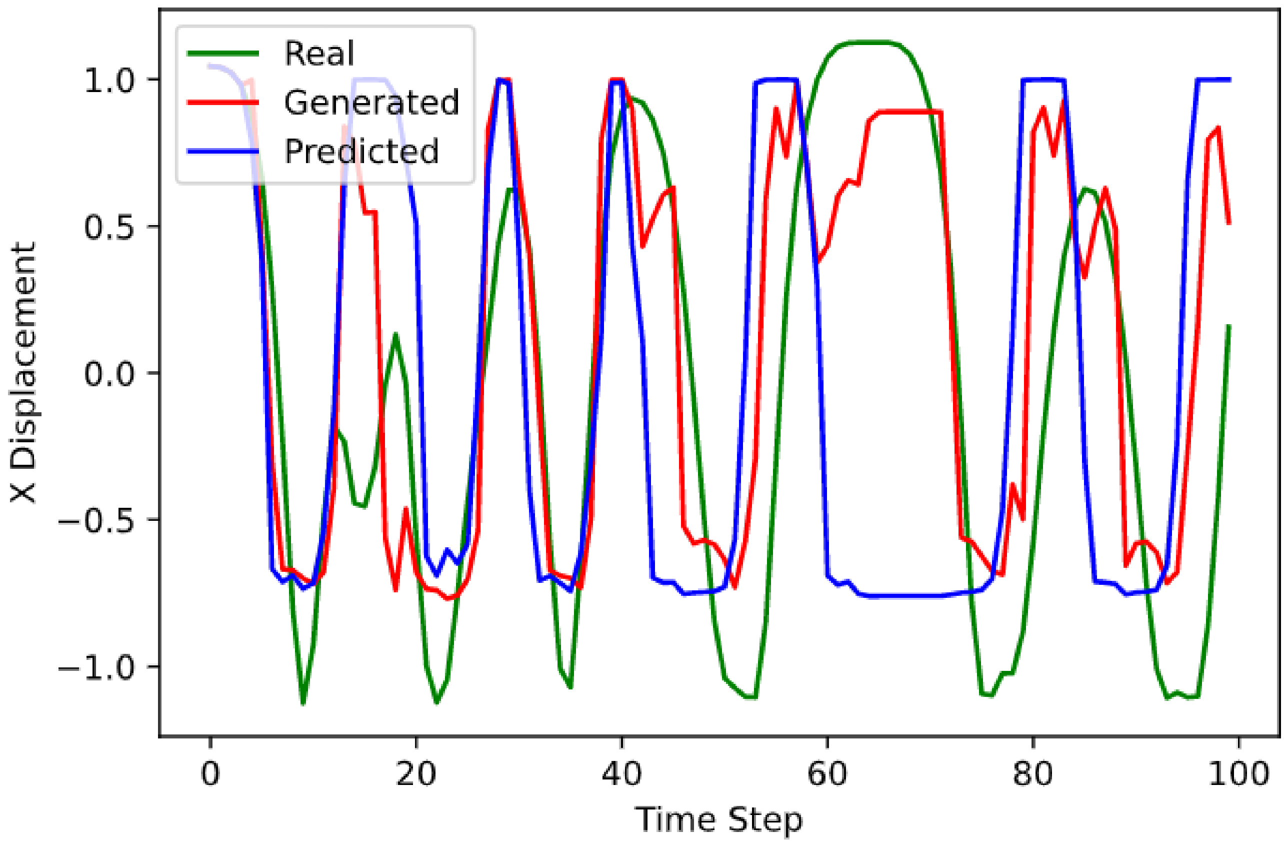
I next began testing the network using an additional third hidden node and allowing the network to run for 1500 epochs. The final epoch error was 0.19. In all the approaches that I tested, this was the most accurate model I was able to find that did not overfit the data and still was able to provide accurate prediction after training. I have tested up to 5000 epochs, and the network error will reduce down to even 0.001 MSE. However, the networks ability to predict motion with only the warmup period being from the testing data decays after 1500 epochs of training in this way.

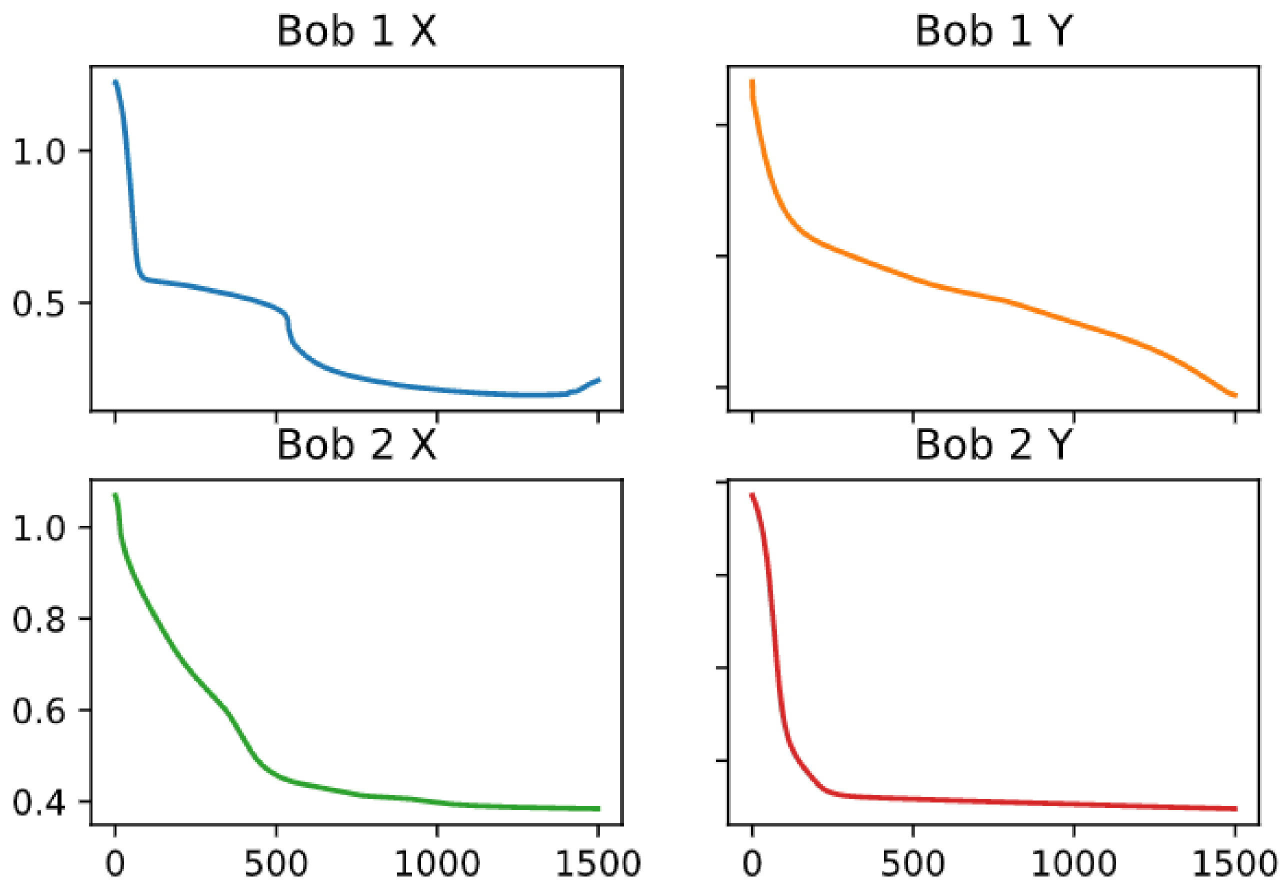
The final optimal network configuration that I was able to find in my testing was:

> - Learning Rate: 0.001
> - Time Steps = 100
> - Data Window = 4
> - Epochs = 1500
> - 4 Input Nodes
> - 3 Hidden Nodes
> - 1 Output Node
> - No Nguyen Widrow initialization

Below is the generated path from the X_1 coordinate, the MSE after training 4 RNNs in this method (one for each coordinate), and the MSE for just training X_1 in this way.







> Notice that all bobs do not converge in the same way. Bob 1 has the least amount of motion on the Y access out of all bobs and trained worse at the beginning. Bob 2 never reaches an MSE as low as Bob 1

> using this configuration.
>
> It is most likely true that each coordinate needs to be tweaked individually to most optimally predict motion of the bobs.

Conclusion

Recurrent Neural Networks have show to have the ability to form powerful models at approximating complex motions. They succeed where earlier MLP designs have struggled at backpropagation over time. The design presented in this paper demonstrated the importance of providing such a network a short term memory through the stored values in the hidden layer. Doing so enables the network to more accurately predict sudden changes in direction and provides more information for adjusting weights during training. Data Windowing provides another powerful technique for time series forecasting. In combination with a robust training set and short term memory, this method could be demonstrated to great effect.

Additionally, the testing and iterative design process described in this paper demonstrates the importance of separating data into relevant groups. Unsupervised learning allows neural networks to provide quickly adapting models, but important care must be put into selecting the relevant data for training.

Unfortunately, this design was not able to be fully implemented due to time limitations. However, early training on the model proved to be promising. With additional optimization, I remain hopeful that the proposed design for predicting the motion of double pendulums will be successful someday.

Bibliography

1. Fausett, Laurene V. Fundamentals of neural networks: architectures, algorithms and applications. Pearson Education India, 2006.Fraser, S.. "Movement Prediction of Three Bouncing Balls." (2018).
2. Karpatne, Anuj, et al. "Physics-guided neural networks (pgnn): An application in lake temperature modeling." arXiv preprint arXiv:1710.11431 (2017).
3. Shi, Xingjian, et al. "Convolutional LSTM network: A machine learning approach for precipitation nowcasting." Advances in neural information processing systems 28 (2015): 802-810.
4. http://www.diva-portal.se/smash/get/diva2:1267392/FULLTEXT01.pdf
5. https://www.tensorflow.org/tutorials/structured_data/time_series
6. https://stats.stackexchange.com/questions/8000/proper-way-of-using-recurrent-neural-network-for-time-series-analysis
7. https://towardsdatascience.com/recurrent-neural-networks-rnns-3f06d7653a85
8. https://en.wikipedia.org/wiki/Double_pendulum
9. https://en.wikipedia.org/wiki/Kinetic_energy
10. https://developer.ibm.com/exchanges/data/all/double-pendulum-chaotic/
11. https://matplotlib.org/3.1.1/gallery/animation/double_pendulum_sgskip.html
12. https://openreview.net/pdf?id=HylajWsRF7
13. http://www.ar.sanken.osaka-u.ac.jp/~inazumi/data/furiko.html