Neural Networks

Ron Wilson

June 14, 2020

Abstract

Neural Network tools are used to analyze three different systems of differential equations in this write-up. The specific differential equations are the Kuramoto-Sivashinky equation, the reaction-diffusion system of equations, & the Lorenz system of equations.

1 Introduction and Overview

Each of the three systems of differential equations will be analyzed separately. Part 1 will be the Kuramoto-Sivashinky (KS) equation, where I will be training a deep learning network for the time-series forecast of the data regressively. I will use the same method for the reaction-diffusion (RD) equations in part 2, however, I will project the data to a low-rank subspace using the Singular Value Decomposition (SVD) to see how time-series forecasting works in the low-rank variables. Finally, for part 3, I will train a neural net for the Lorenz equations to make future state predictions of the data using a feedforward algorithm.

2 Theoretical Background

In this section, I will detail the background information relevant for the training of the three neural nets.

2.1 Regression Network

In the first two parts of this write-up (KS equation & RD equations), we use a method of time series forecasting. We do this by training a sequence-to-sequence regression network, where the responses are the trained sequence shifted by one time-step. This uses a long short-term memory (LSTM) network. LSTM networks are used to take a single sequence of data points and transition them by one time-step at a time, which is essential in advancing an ODE solution from t to Δt.

This network works by inputting the data as a sequence layer. The input layer is then run through the LSTM layers to advance the data in time. We also use a fully connected layer to connect the inputs of each LSTM layer to the activation units of the subsequent layer. This connects all of the inputs to the overall output layer, which is a regression layer. The regression layer computes the error loss of the regression problem. The activation that is used for these problems is the adaptive moment estimation (Adam) optimizer.

2.1.1 Low-Rank Projections Using SVD

The data to be placed as the sequence input layer for the KS equation is just the standardized data from the ODE time-stepper. However, for the RD equations, we want to project the data to a low-rank subspace. This is done using the Singular Value Decomposition (SVD). The SVD of an m-by-n matrix A is defined as:

$$A = U\Sigma V^*. \tag{1}$$

We can compute the economy-size SVD to get results with similar accuracy as the full SVD, but with better execution time. The reduced SVD removes the extra rows or columns of zeros from the matrix Σ, and the corresponding columns in U & V that they would be multiplied by.

The matrix U from SVD is the feature space of the system. In order to project to a low-rank subspace, we can find the rank r of the system and perform the following calculation:

$$X = U(:, 1 : r)^T * A. \tag{2}$$

The data can be projected back into the full-rank space by computing:

$$A = U(:, 1 : r) * X \tag{3}$$

The rank of the feature space is found from the Singular Value Spectrum (SVS). The singular values are the diagonal entries of Σ, and by plotting the singular values we can determine the number of singular values that are the most important in representing the data.

2.2 Feedforward Network

In part 3, we still want to train a neural net to advance a solution (the solution to the Lorenz equations) in time. However, this time we are not regressively cycling through the data. We want to just simply advance the data forward. This is done by using multiple activation functions instead of looping through the same layers as in parts 1 & 2. The activation functions that I will be using are the log-sigmoid (logsig), the radial basis (radbas), and the linear (purelin) transfer functions.

$$f(x) = \begin{cases} \frac{1}{1+e^{-x}} & \text{(logsig)} \\ e^{-x^2} & \text{(radbas)} \\ x & \text{(purelin)} \end{cases} \tag{4}$$

3 Algorithm Implementation and Development

Here, I will explain the algorithms that were used to train the neural nets for each of the three parts (KS equation, RD equations, & Lorenz equations). See Appendix A for the MATLAB functions used here, and Appendix B for the MATLAB code corresponding to these algorithms.

3.1 Part 1: KS Equation

For part 1, we want to advance the solution to the Kuramoto-Sivashinky (KS) equation in time. The KS equation is defined as

$$u_t = -u * u_x - u_{xx} - u_{xxxx} \tag{5}$$

1. First, we need to set the initial & boundary conditions. They are periodic with both sin & cos. Using these parameters, we can run a time-stepping loop to calculate the solution to equation 5 for $n = 64$ data points.

2. Next, we use this data to train the neural net. The data is split into training and testing sets, with 90% of the data in the training set. I also standardized the data by subtracting by its mean and then dividing by the standard deviation. From the standardized data, I further split each set into input and output sets where the input set contains the data points from 1 to $n - 1$ and the output set contains the data from 2 to n (so the input is t, while the output is $t + \Delta t$). Now I can use the input and output of the training set, with the layers & options for a regression network, to train the neural net in MATLAB with the function `trainNetwork(trainInput,trainOutput,layers,options)`.

3. Then, we can predict the evolution of the trajectories for the KS solution by using the MATLAB function `predictAndUpstateState` on the neural net and its input data to update the net, and then using this function again on the output data to continue to update the net to get as accurate as we can with the predictions of the evolution trajectories. Also, once we have the predictions, we can calculate the root-mean-square error of the predictions with the output of the testing set to quantitatively compare the prediction with the truth trajectories.

4. Finally, to evaluate the effectiveness of the neural net, we run the main time-stepping loop for the KS equation again, but with different initial conditions (this time I used only sin, and no cos). Comparing these trajectories with the predicted trajectories can show us how different initial conditions effects the neural net.

3.2 Part 2: RD Equations

In part 2, we want to see how forecasting works on low-dimensional subspaces. We do this by looking at the solutions to the reaction-diffusion (RD) system of equations

$$\begin{aligned} u_t &= \lambda(A)u - \omega(A)v + 0.1(u_{xx} + u_{yy}) = 0 \\ v_t &= \omega(A)u - \lambda(A)v + 0.1(v_{xx} + v_{yy}) = 0 \end{aligned} \tag{6}$$

where the functions λ & ω defined over A are the following:

$$\begin{aligned} A^2 &= u^2 + v^2 \\ \lambda(A) &= 1 - A^2 \\ \omega(A) &= -A^2 \end{aligned} \tag{7}$$

1. Just as we did in part 1, we first need to get the data for the solutions to these equations. This time, we use the MATLAB function `ode45` on equation 6. We define 2π periodic boundary conditions, and periodic initial conditions using cos for u and sin for v.

2. Now, before we train the neural network, we want to work in a low-dimensional subspace. So, we compute the SVD of the data from `ode45` and plot the singular value spectrum. We can see the SVS in figure 1. Here, the first four modes are the most important since they are separated from the rest of the singular values. Thus, I project the data to a low-dimensional subspace by computing equation 2 with rank $r = 4$.

3. Using the low-rank projected data, we train a neural network exactly as we did in steps 2-3 of the algorithm in part 1. Here, however, we need to project the predicted trajectories back into the full-dimensional space to be able to compare the predictions to the truth trajectories. This is done by calculating equation 3 on the predictions.

4. Finally, to evaluate how well forecasting works in low-rank variables, we take a single trajectory from the RD solution and the corresponding predicted trajectory from the neural net to compare. This single trajectory was selected randomly from the testing data.

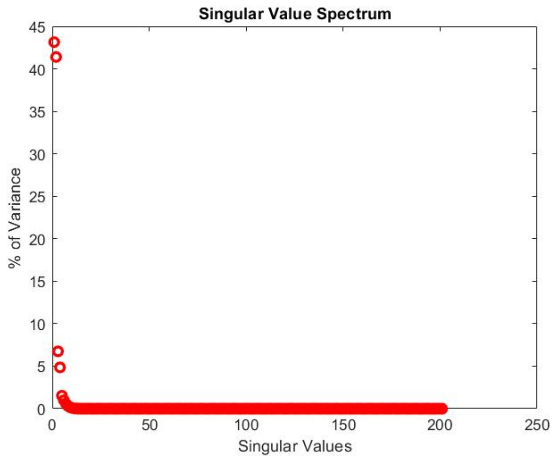

Figure 1: Singular Value Spectrum for the RD equations

3.3 Part 3: Lorenz Equations

Part 3 examines the Lorenz equations, which uses the parameter ρ. Here, we want to advance the solution to the Lorenz equation, but only for three specific choices of ρ (10, 28, & 40), and then see how the neural net works for future state predictions of two other ρ values (17 & 35). We also want to see if the neural net can effectively determine when the trajectories transfer from one lobe to another for a single choice of ρ (28). To do all of this, we need to define the Lorenz equations:

$$\begin{aligned} x_t &= \sigma(y - x) \\ y_t &= \rho x - xz - y \\ z_t &= xy - \beta z \end{aligned} \tag{8}$$

where $\sigma = 10$ & $\beta = \frac{8}{3}$.

1. First, we set the initial conditions and solve the Lorenz equations for each of the three choices of ρ.
2. Then, we train a neural net using the feedforward algorithm. For the data, we collect the solutions for each of the three ρ values into a single matrix to be separated into the input and output sets used to train the neural net.
3. For future state predictions, we need to solve the Lorenz equations again with the new ρ values to get the truth trajectories. We use the net to plug in the initial conditions and get back the future state predictions to compare with the true trajectories.
4. Lastly, we want to see if the neural net can identify transitions from one lobe to another. To do this, we look only at the x and y positions of the $\rho = 28$ Lorenz solutions. Figure 2 shows two lobes in the x, y plane with a line defining the transition between lobes. We can see that the slope of the transition line is 2. Using this fact, we can create a data matrix corresponding to each x, y coordinate point of the Lorenz solution, identifying which lobe each point is in. This allows us to generate labels corresponding to the transition from one lobe to another.
5. Using the transition labels, we can train a neural net with the x, y outputs and their labels. Then, we can take the predicted labels from the neural net and test them against the truth labels that we have pre-defined to observe whether or not the neural net can accurately identify lobe transitions.

4 Computational Results

In this section, I will discuss the results of the neural net training algorithms on the three sets of equations.

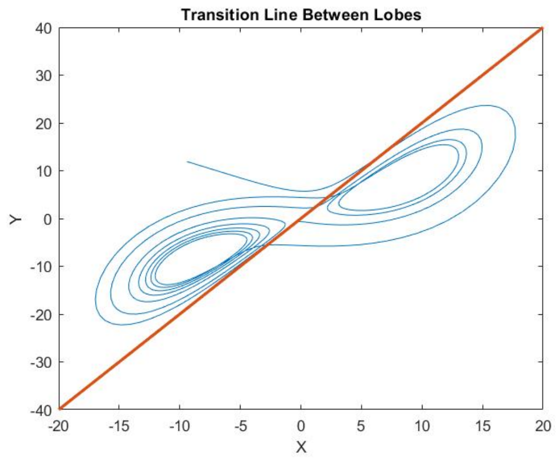

Figure 2: Transition line between two Lorenz lobes

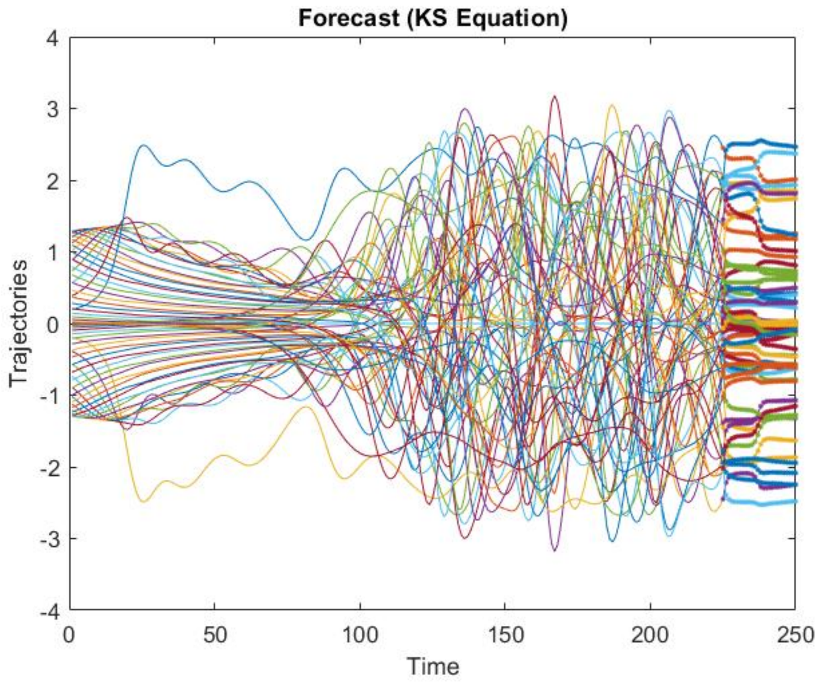

Figure 3: Forecast of the KS equation trajectories

4.1 Part 1: KS Equation

In figure 3 we can see the trained trajectories (approximately the first 225 time series data points) and the neural net forecast trajectories (the remaining time series data) for the KS equation. Comparing the evolution trajectories from the neural net against the ode solver with different initial conditions, we get figure 4. Here, we can see that the structure appears to be the same, but the ode trajectories with different initial conditions are shifted. This is expected since the initial conditions were changed from cos to sin.

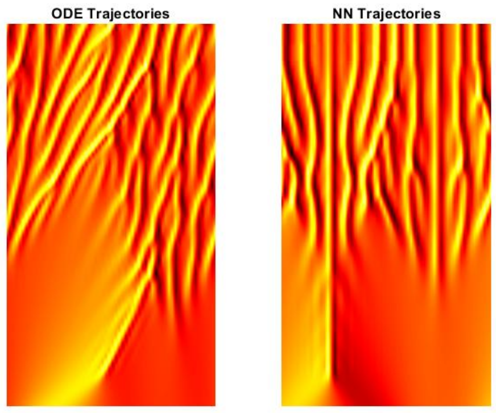

Figure 4: ODE trajectories vs NN Trajectories for KS equation

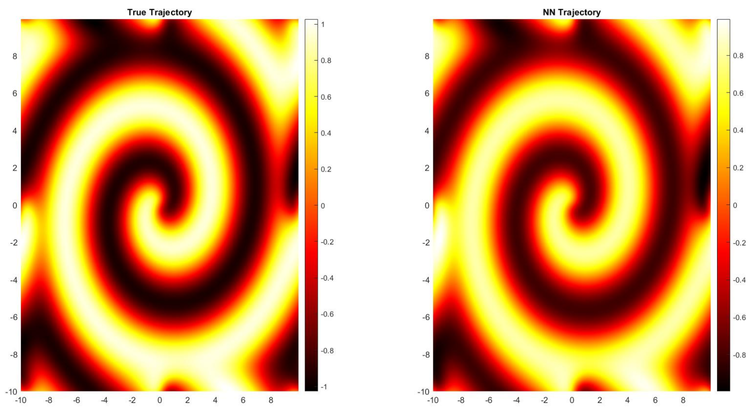

Figure 5: ODE trajectories vs NN Trajectories for RD equations

4.2 Part 2: RD Equations

The solutions to the RD equations were projected to a low-dimensional subspace. In figure 5, we can see the original ”true” trajectories from the ode solutions and the neural net trajectories from training a neural net on the low-rank data. Even with low-rank data, the neural net still got a very accurate prediction of the trajectories. The only difference in the two images is the color; the true trajectory has a spiral with an inner band of white, while the neural net trajectory is mostly yellow. This indicates that the low-rank prediction is missing only a slight amount of information, but it is still a very good prediction.

For the neural nets in both part 1 & part 2, I also looked at the root-mean-square error (rmse) of the network predictions against the true ode solutions. See figures 6 & 7 for the rmse from part 1 & part 2, respectively. For both parts, the error is very low initially, but gets larger with time. For part 1, the increase in error fluctuates, but in part 2, it increases almost linearly. The error in part 2 was likely more linear because of the lower rank subspace we were working in. Note that there are only 4 modes for part 2, while part 1 had 64.

4.3 Part 3: Lorenz Equations

For the Lorenz equations, we trained a neural net on the three ρ values of 10, 28, and 40, and looked at future state predictions for $\rho = 17$ & 35. These future state predictions can be viewed in figures 8 & 9, respectively. In the two figures, the blue spiral is the true trajectory, while the yellow spiral is the neural net predictions. We can see that the predictions were much more accurate for $\rho = 35$ than they were for $\rho = 17$.

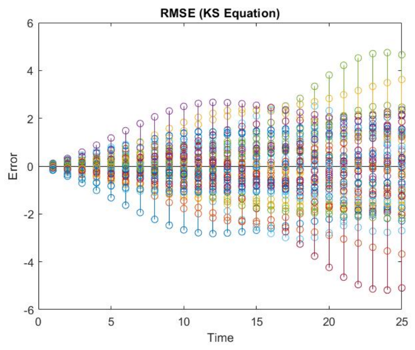

Figure 6: Root-mean-square error for KS equation

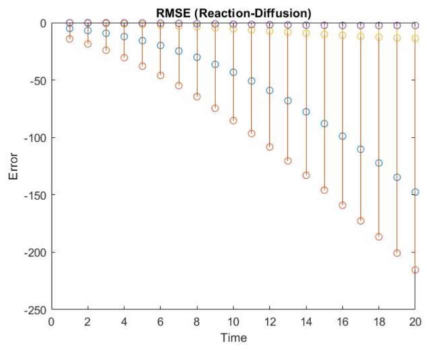

Figure 7: Root-mean-square error for RD equations

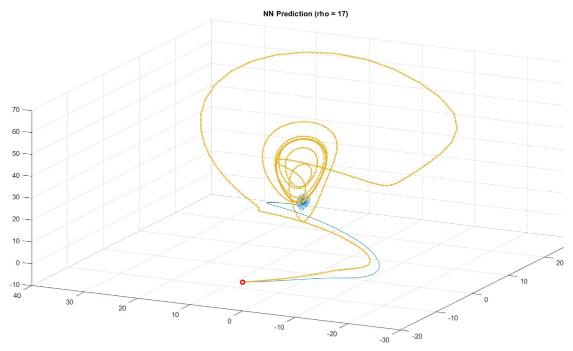

Figure 8: Lorenz NN with rho = 17

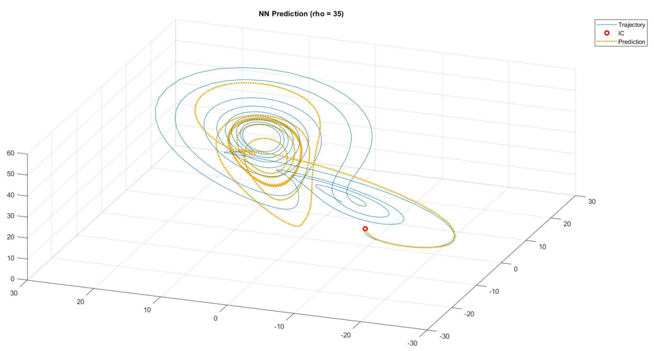

Figure 9: Lorenz NN with rho = 35

This is likely because the feedforward net algorithm had fewer iterations for the $\rho = 17$ problem than it did for the $\rho = 35$ problem, since the future state for $\rho = 35$ is twice as far as that of $\rho = 17$. For a better look at the accuracy's of the future state predictions, see figure 10, which shows the trajectories of each component (x, y, & z). This figure shows even more clearly that the future state predictions for $\rho = 17$ were only accurate initially, but quickly get out of phase. Meanwhile, the predictions for $\rho = 35$ were very accurate for a while before going out of phase.

We are also interested in identifying transitions from one lobe to another. We created labels to identify whether or not the x & y positions transitioned to a new lobe for $\rho = 28$. Training a new neural net against these labels, we generated predictions for the transition between lobes. These predictions can be seen in figure 11. The blue line represents the true labels, while the red is the predicted labels. The x, y trajectories from the Lorenz solutions transition to another lobe when these labels jump from 0 to a larger number. The true labels show 5 lobe transitions total. The predicted transitions were able to accurately identify all 5 transitions. However, they predicted additional transitions that do not occur.

5 Summary and Conclusions

Regression networks are able to accurately forecast time series data, even at low-dimensional subspaces. However, when the initial conditions are changed, you need to expect that the forecast will be shifted. Feedforward networks only give accurate future state predictions when you have larger amounts of time-steps to train on. It is also possible to use a feedforward net to be able to accurately identify lobe transitions, but

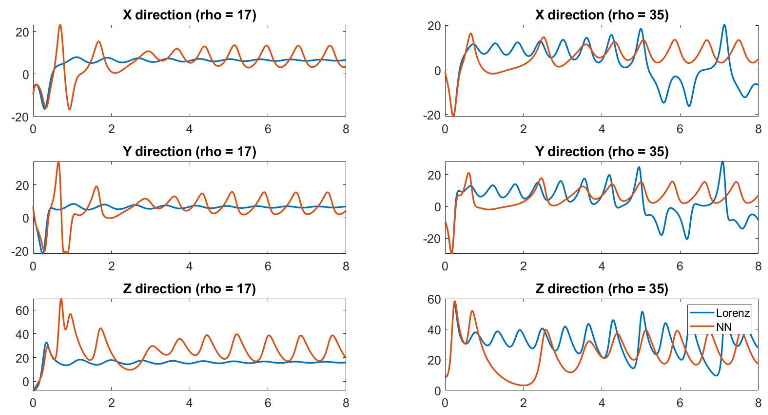

Figure 10: x,y,z directions of NN & Lorenz

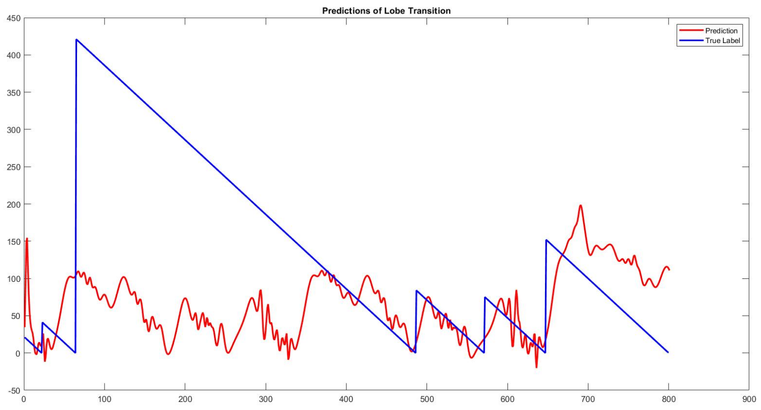

Figure 11: Predictions for lobe transitions

you need to be careful to not over-predict the transitions. The predictions themselves can be made a few time-steps in advance as the prediction curve starts to increase by a large amount.

Appendix A MATLAB Functions

The following is a list of the important MATLAB functions that were used:

- `Y = fft(X)` computes the discrete Fourier transform (DFT) of `X` using a fast Fourier transform (FFT) algorithm.
- `X = ifft(Y)` computes the inverse discrete Fourier transform of `Y` using a fast Fourier transform algorithm. `X` is the same size as `Y`.
- `Y = fft2(X)` returns the two-dimensional Fourier transform of a matrix using a fast Fourier transform algorithm, which is equivalent to computing `fft(fft(X).').'`
- `X = ifft2(Y)` returns the two-dimensional discrete inverse Fourier transform of a matrix using a fast Fourier transform algorithm.
- `layer = sequenceInputLayer(inputSize)` creates a sequence input layer and sets the `InputSize` property.
- `layer = lstmLayer(numHiddenUnits)` creates an LSTM layer and sets the `NumHiddenUnits` property.
- `layer = fullyConnectedLayer(outputSize)` returns a fully connected layer and specifies the `OutputSize` property.
- `layer = regressionLayer` returns a regression output layer for a neural network as a `RegressionOutputLayer` object.
- `options = trainingOptions(solverName,Name,Value)` returns training options for the optimizer specified by `solverName` with additional options specified by one or more name-value pair arguments.
- `net = trainNetwork(X,Y,layers,options)` trains a network for image classification and regression problems. The numeric array `X` contains the predictor variables and `Y` contains the categorical labels or numeric responses.
- `[updatedNet,YPred] = predictAndUpdateState(recNet,sequences,Name,Value)` predicts responses for data in sequences using the trained recurrent neural network `recNet` and updates the network state using additional options specified by one or more `Name,Value` pair arguments.

- `[t,y] = ode45(odefun,tspan,y0,options)`, where `tspan = [t0 tf]`, integrates the system of differential equations `y'=f(t,y)` from `t0` to `tf` with initial conditions `y0` using the integration settings defined by `options`, which is an argument created using the `odeset` function.
- `options = odeset(Name,Value)` creates an options structure that you can pass as an argument to ODE and PDE solvers.
- `[U,S,V] = svd(A,'econ')` produces an economy-size singular value decomposition of matrix `A`, such that `A = U*S*V'`.
- `feedforwardnet(hiddenSizes,trainFcn)` consists of a series of layers. The first layer has a connection from the network input. Each subsequent layer has a connection from the previous layer. The final layer produces the network's output.
- `trainedNet = train(net,X,T)` trains a network `net` according to `net.trainFcn` and `net.trainParam`.

Appendix B MATLAB Code

The MATLAB code used to generate the results of this write-up (`kuramoto_sivashinky.m`, `reaction_diffusion.m`, `reaction_diffusion_rhs.m`, & `nn_lorenz.m`) can be found at the following GitHub repository: `https://github.com/ronwilson016/Data-Science-Projects/tree/master/Machine%20Learning/Neural%20Networks/MATLAB%20Code`