HW3: Neural Networks on Complex Systems

Alexander Van Roijen

May 21, 2019

I Introduction and Overview

Enabled by increasingly powerful computers, Neural Networks have provided us a means to approximate complex systems through large scale transformations. In this paper, I will demonstrate how neural networks can approximate, with various levels of success, complex systems such as the Lorenz attractor, Kuramoto-Sivashinsky(KS) equation, and a reaction diffusion(RD) equation given various alterations to their initial conditions.

II Theoretical Background

Neural Networks can be thought of as solving a system of equations $Ax = b$ where A is our inputs, x is what we seek to learn, and b are our outputs. However, we modify the form of our left hand side to allow for compositions of various A and x. In general, Neural Networks can have multiple layers, each with a different activation function. To generate our predictions, we follow this general algorithm.

1. Given input $n \times d$ X_0, output $n \times y$ Y, and c hidden layers A, we create a Neural Network N that seeks to map $x_i \to y_i$.

   (a) calculate $a_1(x_i \cdot A_{1,a_1}) = x_i^{(1)}$ where a_1 is our first hidden layers activation function

   (b) $\forall A_j \in A$ calculate $a_j(x_i^{(j-1)} \cdot A_{j,a_j}) = x_i^{(j)}$ where $x_i^{(j-1)}$ is the previous layers result.

   (c) return $\hat{y} = A_{c_n,a_n}(A_{c_{n-1},a_{n-1}}(...(A_{c_1,a_1}(x_i))))$

Our weights A within our models are usually learned in an iterative process where we calculate an error of some form $\hat{y} - y$ and use the difference to adjust the weights in each

A_j to reduce the overall error. This method is known as **back propagation**. Again, we are seeking to find our weight vectors x that minimize some error $Ax-b$. We can see that if $c = 1$, $a =$*identity*, and use an error function $||Ax-b||_2^2$, we have a simple Least Squared error calculation. What makes Neural Networks more powerful is that we can stack many A with various dimensions and activation a and create a composition of transformations to learn our system.

III Algorithm Implementation and Development

For all mentionings to data point generation for each complex system, various tools were used developed by other programmers[1,2]. Python's Sklearn MLPRegressor Neural Networks were used to generate our Neural Network weights [3]. Further, the use of an adaptive and constant learning rates were used along with Rectified Linear Units (ReLu) and identity activation functions. To clarify:

- **ReLu activation**: Given input x, returns $max(0, x)$
- **Adaptive learning rate**: Takes constant step size each iteration until consecutive losses in performance, after which it decreases stepsize and continues.
- **Identity activation**: Given input x returns x
- **Constant learning rate**: At each iteration, take the same step size till convergence

In all models highlighted below, the optimal configurations of depth, size of the layers,activation functions, and learning rates, were determined on a case by case basis. Graphical results were generated each time and were used to guide the further development until the final models were determined. For example, all time stepper Neural Networks highlighted below were fed a new initial condition not before seen, and asked to project the initial point x_0, t time steps into the future where t is the length of time steps used in generating one set of data points for the model. It is important to note that **no ground truth** was used outside of the initial starting point for all time steppers when forecasting.

III.I KS Equation Data and Model

We will first begin with attempting to predict future time steps of our KS Equation. I used a resolution of $N = 256$, and created my training data by simulating the equation with five different initial conditions that altered the fluid development over 251 time steps. The equations initial conditions were changed by altering μ in $u = \cos(\frac{x}{\mu}) * (1 + \sin(\frac{x}{\mu}))$ The five different μ were 12,15,16,17, and 19 for training. Since this is a time stepper, we need to offset our inputs and outputs by one time step to achieve $y_i = x_{i+1}$. Thus this

gives us a dataset of size $250 * 5 = 1250 \times N$. I then created a Neural Network where $c = 3$ with sizes $N * 2, N * 3, N * 2$, a penalty $\alpha = 0.0001$, an adaptive learning rate, and Rectified Linear Unit(ReLu) activation in each layer. I then forecasted my model on data where our μ were now 13,16,17,18,20.

III.II RD Data, SVD, and Model

Moving onto the RD problem, we seek again to predict future time points. Thus i generated a training data set of 200 time points, and of resolution $32 \times 32 = 1024$. To make computations and learning more feasible, I used Singular Value Decomposition(SVD) to reduce the dimensions by leveraging numpy's svd library [4]. As we can see in figure 1, a large percentage of the energy is explained by about the first ten singular values. We can reduce the rank of our data thusly by using the first ten columns of the right hand singular vector vh associated with our highest singular values. More formally, let the first ten columns of vh be denoted by z. To reduce the size of an input x, we calculate $x' = x \cdot (z^T)$. To return the data back to our original sized space, we calculate $\hat{x} = x' \cdot (z)$. With our inputs spaced reduced to a 200×10, I train a small Neural network with $c = 1$ with size 100 an $\alpha = 0.00001$, constant step size, and identity activation function.

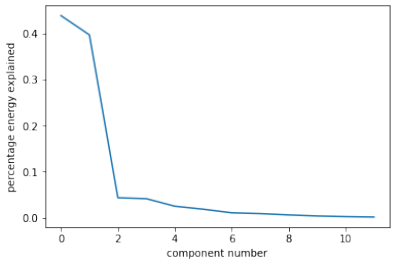


Figure 1: Energy Explained per singular value in descending order

III.III Lorenz Data, and Models

We first want to try and predict future time points for a nonlinear lorenz system of equations with various initial conditions and ρ. To do so, I generated 100 datasets for each $\rho \in [10, 28, 40]$. Each dataset was 8 seconds long recorded at 0.01 time steps. Thus to create our offset where $y_i = x_{i+1}$, we get 799 points per dataset. Thus our total input training data size has dimension $799 * 300 = 239700 \times 4$. The additional fourth column holds the value of ρ associated with the lorenz attractor. Meanwhile the output has dimension$799 * 300 = 239700 \times 3$. Thus when trying to do future time predictions, we append the ρ we are testing onto each output of our model before feeding it back in. With our data organized, I trained a Neural Network with $c = 3$ with sizes 200,300,200 respectively, and an identify activation function with a penalty of 0.00001.

For our final problem, I seek to determine how long will it take for a given point in a lorenz attractor to swap over to the other lobe. Looking at figure 2 we can see that there are two discs separated by a plane around $x = 0$. Thus we want to determine the number of time steps necessary to switch attractors. I generate this data by calculating the lorenz

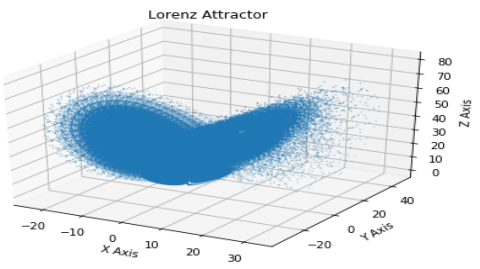


Figure 2: Datapoints for lorenz attractor

points from random initial conditions, just as before, but keeping track of which lobe it starts on and when it switches over. More formally, given an initial condition x_0, y_0, z_0, I take the sign of x_0 and wait till $sign(x_i)! = sign(x_0)$. Once this occurs, I know the number of timesteps between until x_0, y_0, z_0 switches lobes for the first time. Then to generate more of the same simulation, I update the sign to $sign(x_i)$ and wait until $sign(x_j)! = sign(x_i)$ where j is some time δt in the future. Thus each time the signs switch, I have a new set of data points for our model. The only caveat being that I do not hold onto datasets where the point does not transition over the place $x = 0$ more than once. This is because often times an initial condition will simply pass over this plane $x = 0$ on its way to its first attractor. Since I am interested in time steps between attractor swaps for **points on the attractor already**, I only hold onto those that demonstrate that swap at least once.

With the data sorted, I train a simple Neural Network that takes a point x_i, y_i, z_i and outputs a single number k which represents the number of time steps until it switches attractors. This NN was trained on approximately 120,000 points and has a size $c = 3$ with widths 200,300,200 respectively. It further uses a simple identify activation function with a penalty of 0.00001. To assess the validation of this model, I use graphical interpretation where given a new set of points our model has not seen, I ask for each points time till swap k and plot this compared to the truth determined in the same manner as highlighted before.

IV Computational Results and Discussion

IV.I KS Equation

Comparing our forecasts in table one, we see encouraging results when our $\mu = 20$, but seems to act a bit more linear when $\mu = 16, 17$. Further, it isnt able to discern the flat planes too well. However, it does seem to be able to get fit some level of angle when necessary. Overall this model likely needs to be trained with more variations on our μ which I could better determine if I had expertise on the driving equation.

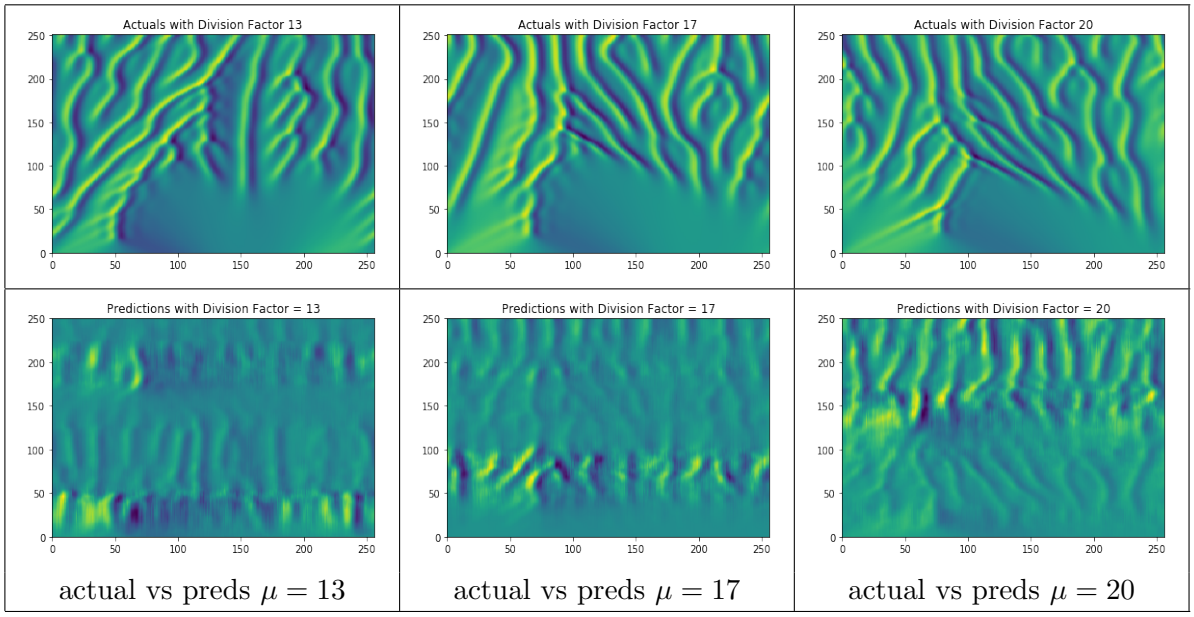


Table 1: Comparative results of our neural network on different initial conditions and their actuals above them respectively.

IV.II RD Equation

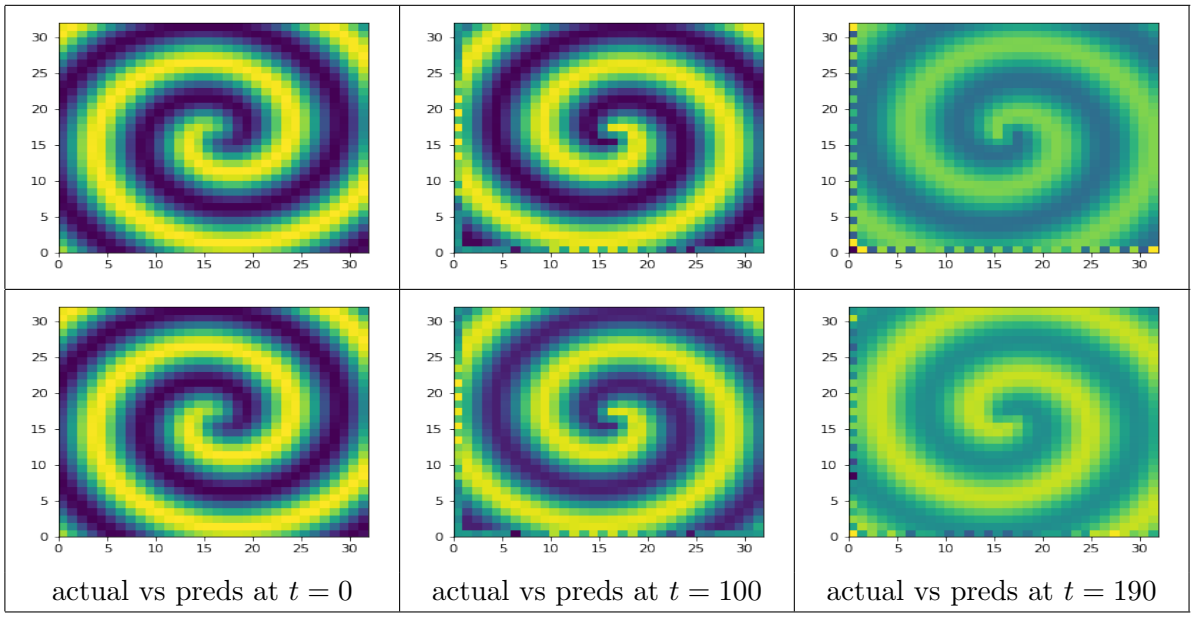
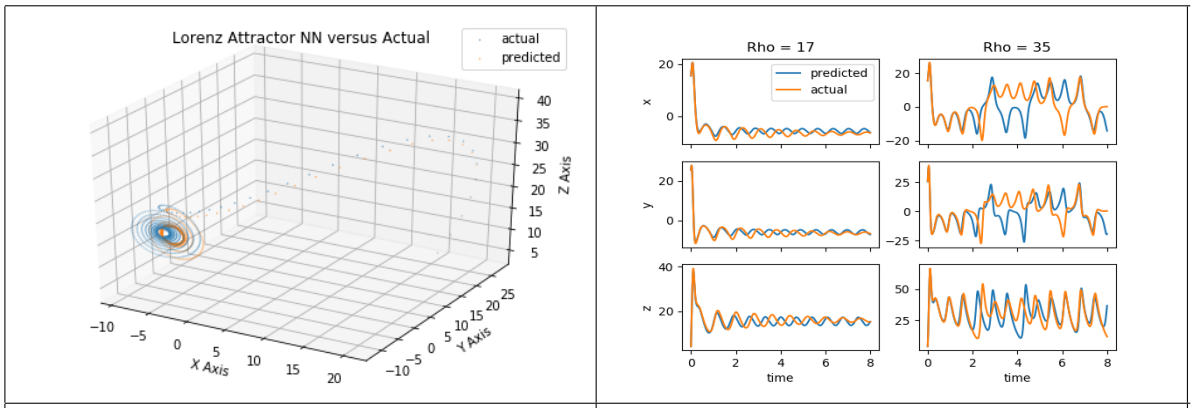


Table 2: Comparative results of our neural networks forecasts and their actuals above them respectively at time slices 0,100,190 out of a total of 200.

Looking at table 2 above of our RD equation forecasts,we get pretty good results. Overall it appears that we stick to the truth pretty well up till about halfway, about 100 time

points in. Furthermore, the reduction to 10 dimensions seems to have encapsulated most of the changes as we are still very close in resulting appearances despite losing some information along the way. However, looking closer to the end we can tell that our forecasts' orientation is either lagging behind or is too far head.

IV.III Lorenz Forecasting

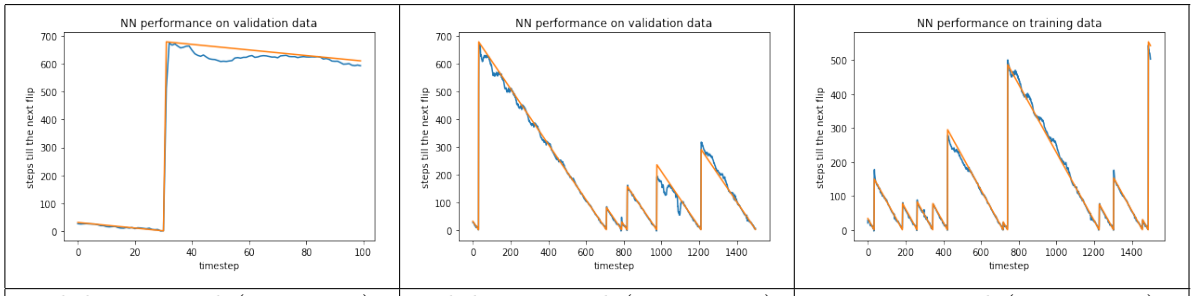


3d projection $\rho = 17$	x,y,z $\rho = 17$ left, $\rho = 35$ right

Table 3: On the left we have a 3d plot of our Neural Network's forecast from some initial point when $\rho = 17$ in orange versus the predicted in blue. The figure on the right breaks down the x y and z components of a forecast when $\rho = 17$ (left) or $\rho = 35$ (right)

Looking at our forecasts for our lorenz transformation in table three, we can see pretty accurate predictions up to about 2 seconds in both forecasts. However, it appears to fall off and follow a slightly different pattern with some similarities later on. Since our data was quite large, I think to achieve better results, more various ρ may be in order.

IV.IV Lorenz Attractor Flip Forecasting



validation preds(100 steps)	validation preds(1500 steps)	training preds(1500 steps)

Table 4: (left to right) Predictions of first 100 data points of test set. Predictions of first 1500 data points of test set. Predictions of first 1500 data points of training set

We can see there is some fluctuation in our models predictions on both the test set and the training set, but overall it does a very good job guessing the general trend. Additionally, the forecast tends to be within 10% of the actual time steps value, peaking at the longer durations. What is even more encouraging is how generalizable it is, as it has visually similar performance to the training data for timesteps large and small between flips. This is likely partly due to the fact that our model only has to worry about one timestep ahead and doesnt rely on previous outputs to determine its next output.

V Discussions and Summary

We have seen how various Neural Network architectures can be used to forecast different parameters of interest of various complex systems. The best success was seen in forecasting the lorenz attractor flips, but we have seen general success in other forecasters up to a certain time step. In most cases, we limited the number of different initial conditions we attempted on our models. However, it should seem intuitive at this point that the more data of various conditions we give our model, the better it can predict. This is because the more possible variations of the data we train on, the less room for error the model can make. For example, I conjecture that for lorenz forecasting, if we had trained on more that just the three values of ρ we would have been able to forecast better on different ρ. The same can be stated for the RD equation as well. However, there are concerns for overfitting which we didnt really see in this assignment, but should be considered in the future.

VI Future Work

Overall, it may be worthwhile to invest some time developing better Neural Networks for some of these systems by expanding the training data as well as the size of our neural networks and allowing for longer training periods.

VII References

- 1: KD Equation and RD equation. Python Scripts May 10, 2019. Electronic. https://www.github.com/someUserKatherineKnows
- 2: matplotlib. Lorenz Attractor. May 11, 2019. Electronic. https://matplotlib.org/examples/mplot3d/lorenz_attractor.html
- 3: Scikit-learn. MLPRegressor. May 14, 2019. Electronic. https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html

- 4: Numpy. SVD. May 20, 2019. Electronic. https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.svd.html

VIII Results Appendix

Code for this assignment can be found in the PDECodes.ipynb attached with this submission.

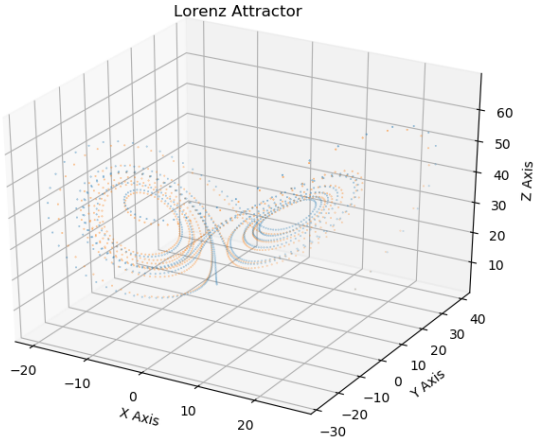


Figure 3: Forecasting from lorenz NN when $\rho = 35$ versus actuals

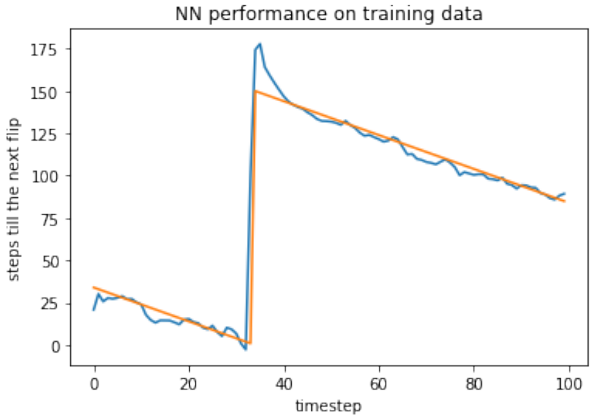


Figure 4: first 100 steps of our lorenz flip model versus actual time till flip

More code,images, and results can be found on my github.