AMATH 563 Homework 04
Neural Networks for Dynamical Systems

Joseph J. Williams

2020 Jun 10

Neural networks (NNs) are applied to the Kuramoto-Sivashinky (KS) equation, reaction-diffusion (RD) equation, and Lorenz equation with the goal of advancing the solutions from t to $t + \Delta t$. In-built MATLAB functions are used to build, train, and test custom NNs. NNs with a single LSTM layer with 50 hidden units and a tanh layer with a variable number of training epochs are trained to make time series predictions for all three equations; the NNs are trained on $\tau \in \{10, 100\}$ simulations with $\nu \in \{10, 50\}$ noise. The NNs are found to make reasonable reconstructions of the data they are trained on and reasonable predictions into the future for all three equations.

Sec. I Introduction and Overview

In this study, we train neural nets (NNs) on three equations or systems of equations: the Kuramoto-Sivashinky (KS) equation, the reaction-diffusion (RD) equation, and Lorenz system. By training a variety of NNs, we seek to produce the most accurate reconstruction and prediction of the solution to the equation or system of equations for a specified time horizon. The equations and system of equations used in this study will be covered briefly at the beginning of § II of this report; however, they are merely the objects of our NN training schemes and not the main interest.

Instead, the main interest of this report is the use and behavior of and results form NNs themselves. Fundamentally, machine learning revolves around optimizations, and NNs are no different, in that they seek to optimize an error function by finding its minimum; this error function comes from comparing the withheld testing data with the prediction made by a NN trained on a similar set of data. The NNs will be trained to produce evolution trajectories for each of the three problems; the predictions will be compared to their numerical solutions and the models will be used to predict the evolutions of systems with different initial conditions (ICs). In the case of the RD equation, the data will be trained and tested only on a spatial subset and using various reconstructions of various rank from the SVD.

A wide variety of training architectures are possible; this study will focus on simple architectures with comprehensive cross-validation techniques. The networks in this study consist on a single LSTM layer and a single tanh layer with Ep $\in \{100, 250\}$ training epochs.

In § II of this report, we will explore the theoretical background and construction of NNs, and in § III, we will discuss their algorithmic implementation. In § IV, we will discuss the results of training various NN architectures on and using the trained models to predict the evolution of the solution for nearby ICs for each of the three problems.

Sec. II Theoretical Background

Before discussing NNs in detail we will first briefly go over the objects of our NN training: the three equations or systems of equations whose solutions at each time step will form the training and testing data for the NN. We cover them in turn:

1. The KS equation models diffusive instabilities in a laminar flame and exhibits chaotic behavior; it is a fourth-order, nonlinear partial differential equation given as:
$$u_t + \nabla^4 u + \nabla^2 u + \tfrac{1}{2}|\nabla u|^2 = 0,$$
where ∇^2 is the Laplace operator and ∇^4 is the biharmonic operator.

2. The RD equation models the changes in space and time of the concentration of a substance both reacting (i.e. being produced or destroyed) and diffusing (spreading out, in some sense, over the domain); it exhibits wave-like phenomena and self-organizing patterns in the solution. It is a semi-linear parabolic differential equation; its one-dimensional, one-component equation is:
$$u_t - Du_{xx} = R(u),$$
where D is the diffusion coefficient and $R(u)$ is the reaction function.

3. The Lorenz system is a simplified mathematical model for atmospheric convection; it also arises in a variety of other scientific applications. Its solution is became known for exhibiting chaotic behavior and for its resemblance, when plotted graphically, to a butterfly. It is a three-dimensional system of nonlinear ordinary differential equations given as:
$$x_t = \sigma(y - x)$$
$$y_t = x(\rho - z) - y$$
$$z_t = xy - \beta z.$$
It is nonperiodic and deterministic. The Lorenz system exhibits a variety of different behaviors for different sets of values of the parameters $\{\sigma, \rho, \beta\}$. The classic values are usually $\{\sigma, \beta, \rho\} = \{10, 8/3, 28\}$ and give rise to chaotic behavior at these and nearby values.

An important but subtle point of chaotic behavior is that, while the KS equation and Lorenz system exhibit chaotic behavior, they are still deterministic – using the exact same ICs, you will achieve the exact same solution. Chaotic behavior does not imply randomness; instead, chaotic behavior means that even small differences in the ICs can lead to unpredictably large changes in the evolution of the solution. Thus, in general, a solution that exhibits chaotic behavior cannot be used to estimate a solution using nearby ICs. We will show that, despite this, well-trained NNs can predict the evolution of chaotic systems for large time horizons using a range of ICs.

We now move to our discussion of NNs. At a high level, a NN, much like the supervised machine learning (ML) algorithms of the previous assignment, accepts inputs, applies various decompositions, filters, or functions to analyze and transform the data, and finally gives as outputs predictions in the form of e.g. image classification, equation or system solution values, or identification of some particular feature of the data. Reference Fig. 1.1 below, borrowed from Brunton & Kutz, for the generic architecture of a multilayer NN.

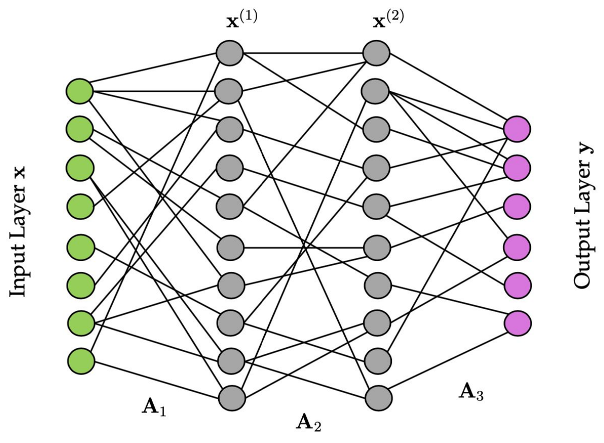

Figure 1.1: Generic Architecture of a Multilayer NN

For a classification task, the dimension of the input space **x** is that of the raw data, e.g. the number of pixels in an image, and the dimension of the output space **y** is the number of different labels, e.g. whether the image is of a dog or a cat (–1 or +1). For equation prediction or state of system prediction, the input space **x** may be all measurement values for a particular range of times, and the output will be the predicted values of those measurements at future times. The layers of the NN consist of the various decompositions, filters, or functions that analyze and transform the function in some way; these functions may be simple or exotic and the filters may break the data up into some small or large number of groups. Each layer may have a different number of nodes, and in general the nodes may have any sort of connection, backwards and forwards, unique or not. While only certain connections and systems of connections have concrete foundations and broad applications, there stills exists a zoo of different NN architectures; for further details and many more architectures in existence today, including basic schematics of each architecture, the reader is referred to pgs. 255–262 of Brunton and Kutz.

There are a number of questions one may ask when viewing Fig. 1.1 and considering the discussion following it:

1. How many nodes should each layer have, i.e. what should be the dimension of each layer?
2. How many layers should there be?
3. How should the nature of the connections between the layers be? e.g. sparse or rich, unique or nonunique
4. Should the mappings between the layers be linear, nonlinear, or a mixture?

Many of these questions will be given treatment in § IV of this report. Moving to the theoretical framework of NNs, in general, NNs optimize over a compositional function

$$\text{argmin}_{\mathrm{A_j}}(f_{\mathrm{M}}(\mathrm{A_M}, \dots \mathrm{f}_2(\mathrm{A}_1,\mathbf{x})) \dots) + \lambda \mathrm{g}(\mathrm{A_j})),$$

where the first term is the objective function and the second term is the regularization. Composition allows one to generate expressive representations of the data, while regularization prevents overfitting. As usual, cross validation is of critical importance, as NNs have significant potential for overfitting to the data. This compositional function is solved using stochastic gradient descent and back propagation algorithms, two algorithms critical for success of NNs, both explored in more detail later. In this compositional function, each matrix A_k contains the weights connecting the kth to the $(k+1)$th layer of the neural network, denoted $\mathbf{x}^{(k+1)}$ and $\mathbf{x}^{(\mathrm{k})}$. The mapping from k to k+1 is given by:

$$\mathbf{x}^{(k+1)} = \mathrm{A}_{k+1}\mathbf{x}^{(\mathrm{k})}.$$

From this, we may obtain the mapping from the input data **x** to the classification **y** by:

$$\mathbf{y} = \mathrm{A}_M\mathrm{A}_{M-1} \dots \mathrm{A}\mathbf{x}.$$

This system is massively underdetermined, in that there are far generally more weights from layer to layer than there are layers themselves. Recall that the initial input is your data – a time series of the numerical solution to a PDE at each spatial node, or the intensities of each pixel in each image in a series. Thus, even a small amount of data will produce many inputs to the NN, which is many nodes at at least the first layer; while having some layers with many fewer nodes than there are inputs is possible and common in some architectures, the "usual" layer will have as many or close to as many nodes as there are inputs. Since the system is massively underdetermined, constraints must be imposed in order to select a unique solution of some desired type (e.g. certain sparsity or minimizing a certain norm). In this study, the number of layers will be on the scale of $\mathcal{O}(10^1 \sim 10^3)$, but the number of variables is on the scale of $\mathcal{O}(10^x \sim 10^y)$.

The backpropagation algorithm (backprop) efficiently frames an optimization problem and produces a formulation amenable to gradient descent optimization. Backprop hinges on the chain rule in differentiation; following Brunton and Kutz, due the compositional nature of the network, when solving the optimization problem to achieve minimum error E, the chain rule for differentiation causes a cascade of derivatives that becomes longer the closer you come to the input layer. This causes the error

The derivatives are solved using the gradient descent algorithm, providing the optimal network weights; however, this leads to the main computational expense: computing all of the derivatives. Although we selected in the first place functions that were easy to differentiate (see Brunton & Kutz, pg. 233), for large data sets, the sheer number of computations we'll have to do for each data point at each iteration quickly grows out of hand. Instead, we use stochastic gradient descent (SGD), which selects at random a point or set of points and evaluates the gradient there. This saves a significant amount of computational effort, the hope being that we will, more often than not, to within some tolerance, be moving towards the global minimum. Some data, however, may be contoured such that the algorithm will get stuck in local minima and never reach the global minima.

This study focuses on the application of NNs to partial differential equations and systems of ordinary differential equations. Cross-validation takes the form of testing the generated models against the numeric solution of the problems for different ICs; a range of parameters will be explored, including variation in number of training epochs *Ep*, the number of hidden units h in the LSTM layer. For the RD equation, there is also the spatial resolution decrease factor *SRD* and rank r reconstruction from the SVD.

In this study, we first experiment with a NN composed of only a single LSTM layer with a variable number of hidden units (usually $h = 50$ and $h = 100$) before experimenting with a wider variety of a combination of LSTM layers and a variety of transfer function layers.

As a form of cross-validation, we run the RD equation a number of times with a small amount of random noise added to the initial conditions each time; the NN is trained over all evolutions and tested against an unseen, sequence of RD solutions with the same ICs but unnoised. We will track $l_2(\varepsilon)$, where the error $\varepsilon = (y - y)$ against time for a variety of NN architectures and parameters.

Sec. III Algorithm Implementation and Development
Running the provided codes for the KS and RD equations and constructing a code for the Lorenz system was straightforward. For constructing and training NNs, MATLAB fortunately provides a very robust toolbox. The net = trainNetwork(X,Y,layers,options) command was used to train a network for deep learning with the specified layers and specified options (including the algorithm to optimize the network weights). MATLAB also provides a variety of tools to post-process and understand the model and use it to forecast future states of the system.

The amount of noise included in the simulations is defined by the parameter ν, which quantifies the noise. Using the KS equation as an example, the provided code initializes the KS equation with the initial conditions:
u = cos(x/16).*(1+sin(x/16))

Random, uniform (not Gaussian) noise is added by:
u = (cos(x/16).*(1+sin(x/16))) .* (ns_lw + (ns_hg-ns_lw).*rand(N,1));

where (ns_lw – ns_hg) * 100 = ν. Thus, for small values of ν, this is only a small variation about the set initial conditions, whereas for large values of ν, there is a large variation about the set of initial conditions. Note that the noise is uniform, as opposed to Gaussian. Another parameter is τ, which is how many times the simulation is run with uniform noise. Larger values of τ are of course more computationally intensive, but training on a larger set of noisy samples should lead to better results.

ns_lw	**ns_hg**	**ν**
0.95	1.05	10
0.75	1.25	50

Table 3.1: Values of ns_lw, ns_hg, and ν used in this study.

Sec. IV Computational Results

Reference Fig. 4.1 below, which shows the l_2 norm of the error between the prediction from the NN and the actual data for the KS equation. Recall that ν quantifies the noise about the initial conditions and τ is how many realizations of the noisy simulation the NN was trained on. The dashed black line at *t* = 175 shows the end of the training period; thus, the NN was not trained past *t* = 175.

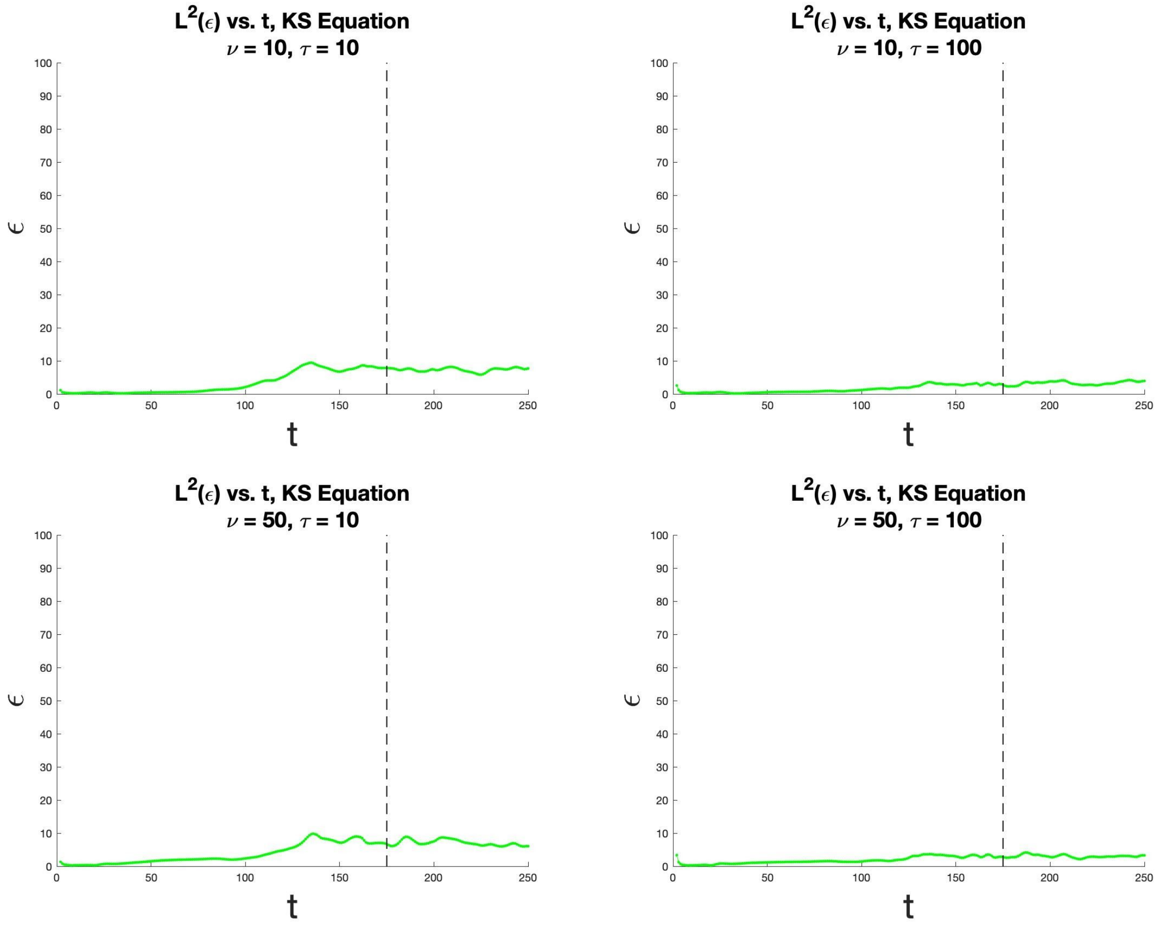

Figure 4.1: l_2 Norm of the Error vs. t for the KS Equation;
ν = 10 (*top*) and ν = 50 (*bot.*); τ = 10 (*left*) and τ = 100 (*right*)

It is expected that, for larger τ, the error is lower, as larger τ means the NN was trained on more noisy samples and is thus better able to learn the algorithm and predict the future state. This is observed. It is also expected that for larger ν, the error would be higher, as larger ν means the samples the NN was trained on had more uniform noise. This, however, is not observed. In fact, it appears as though the errors between ν = 10 (*top*) and ν = 50 (*bot.*) for both values of τ have roughly the same behavior. Finally, one would expect that the error would rise past *t* = 175, as this is the final point in time that the NN trained on; past this point in time, the NN has had no samples to train on. However, the error remains fairly steady past *t* = 175, even appearing to be on a slight downward trend for ν = 50 and τ = 10 (*bot.*, *left*).

Moving from the KS equation to the RD equation, reference Fig. 4.2 on the following page for similar plots showing the l_2 norm of the error between the prediction from the NN and the actual data for the RD equation. For the RD equation, various rank reconstructions from the SVD were used for training and predction; note that *r* = 201 is a full rank reconstruction. Only a single level of noise ν = 10 was used, but two values of *m*, the number of spirals in the initial conditions on the RD equation, were used.

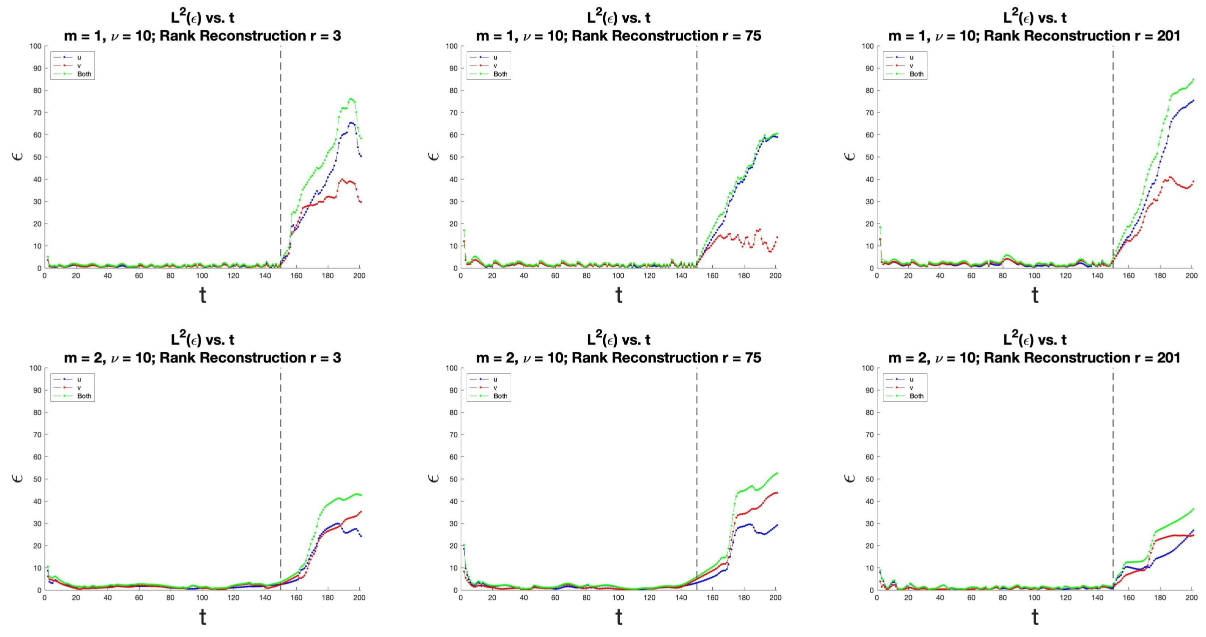

Figure 4.2: l_2 Norm of the Error vs. t for the RD Equation, ν = 10; *r* = 3 (*left*), *r* = 75 (*mid.*), *r* = 3 (*right*); *m* = 1 (*top*) and *m* = 2 (*bot.*)

Clearly, the NN performs better for m = 2 than it does for m = 1 at all rank reconstructions. This is surprising to me, as I would have thought that, as *m* increases, the complexity of the system is increasing and the NN would have a more difficult time training, but this is clearly not the case. Also note that there seems to be no clear pattern between *r* and the error: a full rank reconstruction (i.e. no SVD performed) provides the lowest error for *m* = 2, while *r* = 75 appears to provide the lowest error for *m* = 1. However, using low rank reconstructions certainly sped the training of the NN and is thus still a valuable and useful thing to do. Finally, it is curious to note that there is no clear dominance between the errors of *u* and *v*: in some NN trainings and realizations, the error on *u* is higher, while in others, the error in *v* is higher. Curiously, the error in *v* appears to be quite low for *m* = 1 and *r* = 75 (*top, mid.*).

Moving from the RD equation to the Lorenz system, reference Fig. 4.3 below for similar plots showing the l_2 norm of the error between the prediction from the NN and the actual data for the RD equation. The code for this portion, which again generates a variety of noisy data to train on, was borrowed directly from the textbook, which included code for adding noise to each realization of the Lorenz system; as such, the variable ν is no longer present. Instead, we consider ρ. The NNs are trained on three different values of ρ: 10, 28, and 40. Each trained NN is then tested again data generated from the Lorenz system with ρ = 17 and ρ = 35.

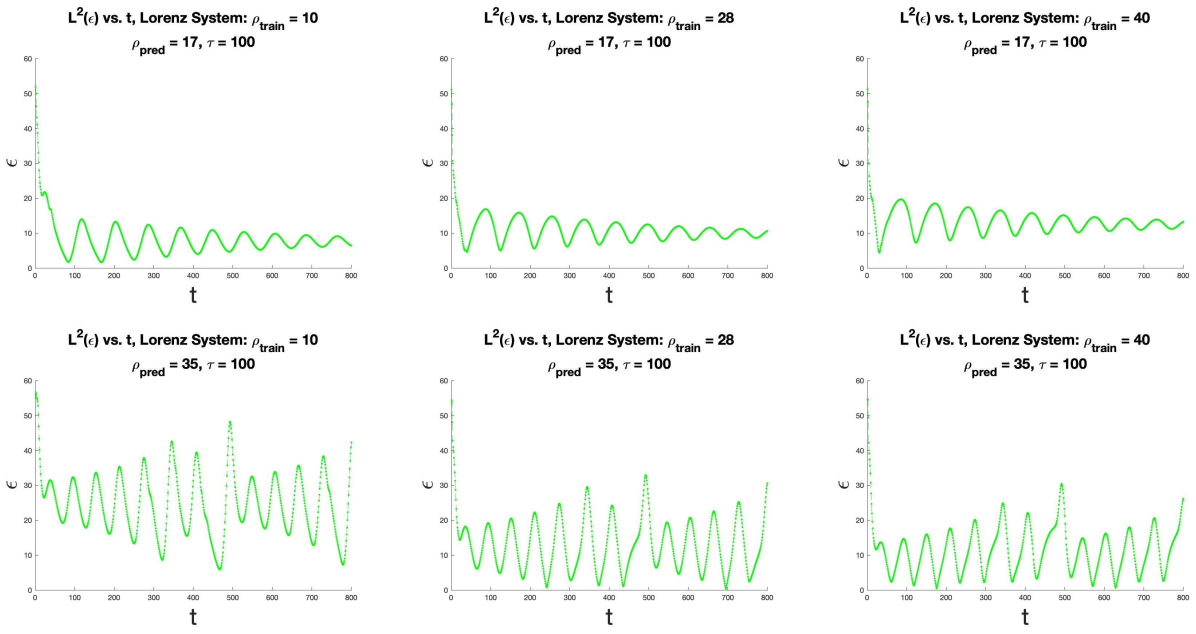

Figure 4.3: l_2 Norm of the Error vs. t for the Lorenz System;
ρ_{train} = 10 (*left*), ρ_{train} = 28 (*mid.*), ρ_{train} = 40 (*right*); ρ_{test} = 17 (*top*), ρ_{test} = 35 (*bot.*)

We see that, in all cases, the error curves have an oscillatory nature; this likely has to do with the solution to the Lorenz system switching lobes periodically. We notice also that, for all values of ρ_{train}, the error is lower for ρ_{test} = 17 than it is for ρ_{test} = 35. This possibly suggests that this particular NN better predicts the time evolution of the system for lower ρ_{pred}; however, with only two values of when ρ_{pred}, we have insufficient evidence to draw this conclusion. We notice also that the error plot generally has smoother oscillatory behavior for ρ_{pred} = 17 than it does for when ρ_{pred} = 35. Finally, curiously, for both values of ρ_{pred}, the error seems to oscillate about a mean, constant, nonzero value; when ρ_{pred} = 17, the oscillations seem to be decaying with at a possibly (negative) exponential rate, while the oscillations when ρ_{pred} = 35 seem to display complex behavior, with the amplitude of the oscillatory curve increasing with time to some value before resetting, much like a sawtooth pattern. It reminds one greatly of plotting a small and finite number of Fourier modes of a curve.

Finally, we turn our attention towards identifying lobe transitions. To be perfectly honest, I did not do this part of the report, as the assignment was turned to credit/no-credit and I sort of lost steam. As I'm turning it in on Friday evening, I know I could accomplish it over the next 48 hours, but I am a bit worn out and would rather just turn it in at this late point in the quarter. First year of grad school has really worn me out, you know?

However, here is how I would do it:

With the architecture to run the above Lorenz simulations and trainings, it would be easy to train a NN in the classification task of identifying lobe transitions. Instead of feeding the NN a 3 x N_{total} matrix, where the 3 rows represent the three components of the solution $\mathbf{x} = [x\ y\ z]$ to the Lorenz system and the N_{total} columns represent different data points in time, we would feed the NN a 4 x N_{total} matrix, where the fourth row is either 0 or 1; 0 represents that a lobe transition is *not* imminent, and 1 represents that a lobe transition *is* imminent. This would require manually viewing the entire solution as making the subjective decision as to when we believe a lobe transition is imminent. Alternatively, we could potentially automatically identify when a lobe transition is imminent by investigating the sign of x and the value of its derivative $\dot{x}$, as I believe a lobe transition occurs when x changes sign.

Once we have identified when the lobe transitions occur, we must choose how far in advance in time (how many time steps in advance, in practice), we want to train the NN to try and predict a lobe transition. It would be trained on some subset of previously unseen data (the withheld testing set), which would also have to be labelled for all time as 0 or 1in the same method as the training data. Cross-validation would be pursued with *k*-fold validation and by increasing the number of epochs the NN is trained for; an error rate would be computed for various NN architectures.

Alternatively, a totally different (and perhaps untenable) idea I had for identifying lobe transitions would be using DMD or SINDy to fit nonlinear functions to the $\varepsilon(t)$ curves, and using the generated curves to identifying lobe transitions. Since the various oscillations in the error curve are likely related to the oscillatory behavior in some way, this would not only help us identify the lobe transitions but also potentially inform *how* to construct and train a neural net to identify lobe transitions in the data.

Sec. V Summary and Conclusions

In this study, we investigated training a neural network to learn various partial differential equations and predict their evolution in time. The PDEs investigated were the Kuramoto-Sivashinky (KS) equation, the reaction-diffusion (RD) equation, and Lorenz system. A simple NN was used, and the l_2 norm of the error between the prediction and the actual output was plotted against time.

While there were a large variety of parameters that could be controlled, including the type and amount of noise to add to training simulations, various parameters specific to the PDEs, and many parameters specific to the NN itself (the type of NN, number and type of layers, training epochs, learning rate, etc.), only a few were systematically varied in this exploratory report. It was found that even these simple NNs are able to adequately learn the PDEs and make reasonable predictions into the future. Future work includes a principled and thorough variation of these parameters in order to develop the most robust neural net.

Appendix A

See GitHub repository:

https://github.com/williamsj0165/AMATH563_HW04.git