AMATH 563 HW 3

Neural Networks as High-Accuracy Time-Steppers

Zachary McNulty
zmcnulty, ID: 1636402

Abstract: Predicting future states in a system is a fundamental challenge within applied mathematics that has applications across all fields. While traditional numerical techniques such as Runge-Kutta schemes are incredibly accurate, they often require the governing equations of the underlying system to be explicitly known. In this paper, we show how neural networks can be used to build model-free data-driven time-steppers for future state predictions.

University of Washington
May 2019

Introduction and Overview

Predicting future states in a system is a fundamental challenge within applied mathematics and data science that has applications across all fields. This kind of forecasting is fundamental to any type of mathematical modeling and can inform future decisions. While traditional numerical techniques such as Runge-Kutta schemes are incredibly accurate, they often require the governing equations of the underlying system to be explicitly known. There are many data-driven techniques including DMD, PCA, and SINDy which might be used to approximate these governing equations. However sometimes, especially in highly complicated system where these techniques may break down or lose interpretability, the future state predictions is all that matters. In this paper, we show how neural networks can be used to build model-free, data-driven time-steppers for future state predictions. As a case study, we will focus on three prominent dynamical systems: the Lorenz System, the Kuramoto–Sivashinsky equation, and the reaction-diffusion system.

Theoretical Background

Neural networks are a form of supervised machine learning that consist of a set of nodes, often arranged in collections called layers, and a set of weighted (one-way) connections between these nodes. Each node in the network takes on a certain activation level that depends on the weighted sum of the activations of all the nodes that have connections to it. Often, this sum is manipulated in some way by an activation function σ to generate the final activation. Below are a few common activation functions:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \qquad sigmoid(x) = \frac{1}{1 + e^{-x}} \qquad RELU(x) = max(0, x) \qquad linear(x) = x$$

This can introduce nonlinearities into the system, improving its ability to capture nonlinearities in the data, and give desirable properties to the activations (i.e. the sigmoid activation function $\frac{1}{1+e^{-x}}$ squishes the activations into $[0, 1]$). Thus, the activation of a given node v_i is:

$$a_i = \sigma\left(\sum_{j=1}^{n} W_{ji} a_j + b_i\right) \tag{1}$$

Data is fed into the network through a set of nodes called the input layer and this defines the initial activations. From here, this information is fed forward[1] to determine all other activations in the network. The goal is to get the final layer, the output layer, to make some kind of prediction like a future state or classification.

Like all kinds of supervised learning techniques, it is important to consider what type of information is best to train on. A variety of preprocessing steps such as the SVD, wavelet/fourier transform, or other dimensionality reduction techniques can be used to change the space in which training occurs. Some spaces are better at capturing key characteristics of the system than others and will thus generate better results following training

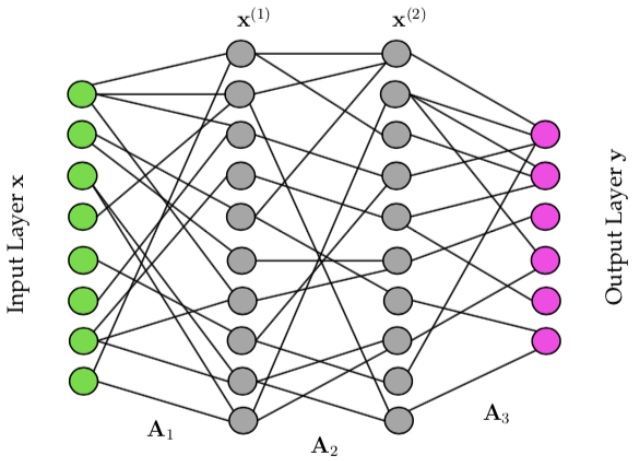

Figure 1: Standard feedforward neural network with an input layer, 2 hidden layers, and an output layer. (Source: image is taken from the class notes)

[1]This is not exactly true. This primarily describes feed-forward neural networks, although there is a huge variety of neural network structures.

so this is an important consideration. In this paper, we used the SVD. The SVD is a diagonalization of the matrix $A \in \mathbb{R}^{m \times n}$ of the form:

$$A = U\Sigma V^*$$

where U, V are unitary matrices and Σ is diagonal nonincreasing. Here, the columns of U, V are orthonormal bases for R^m and R^n respectively. If A is a data matrix whose columns are data measurements and rows are features, then the columns of U are a basis, an orthonormal/uncorrelated coordinate system, for feature space and the corresponding columns in V provide the coordinates of each data sample within this new coordinate system. The corresponding entries in the diagonal of Σ, the singular values, weight how much variance in the data is captured along each of these new coordinates (how "important" each coordinate is). Often, many of the singular values are zero, hinting that the data can be well-approximated in a lower dimensional space by only using the coordinates associated to large singular values. This gives a lower dimensional, uncorrelated representation of the data which is useful for training.

In general, neural networks are hard to train. They are nonconvex so falling into local minima is always a problem and they have so many parameters that overfitting is a huge issue. As we have seen before, the latter case can at least be held in check with regularization. Typically, neural nets are trained using some kind of gradient descent on a cost function so adding an L_1 regularization on the weights in the network can help promote sparsity.

The Lorenz system is a simply model for atmospheric convection that is well-known for its chaotic behavior. It is defined by the system of ODEs:

$$\frac{dx}{dt} = \sigma(y - x)$$
$$\frac{dy}{dt} = x(\rho - z) - y$$
$$\frac{dz}{dt} = xy - \beta z$$

The Kuramoto–Sivashinsky equation is a PDE that describes diffusive instabilities in a laminar flame front. It too is known for its chaotic behavior and is described by the PDE:

$$u_t + \nabla^4 u + \nabla^2 u + \frac{1}{2}|\nabla u|^2 = 0$$

Lastly, the reaction-diffusion equation which model the concentration of a pair of chemical reactants over space and time as the two interconvert. In the case of this paper, we will study a 2D system:

$$\vec{q}_t = D\nabla^2 \vec{q} + R(\vec{q})$$

Algorithm Implementation and Development

Lorenz System

Our first goal is to build a time-stepper that can reliably predict future positions in the Lorenz System. Furthermore, we want it to be able to accurately predict even as we vary the parameter ρ within a reasonable range. To do so, we generate a few hundred eight-second trajectories starting at random initial conditions within the system (using ode45 with absolute tolerance of 10^{-11} and relative tolerance of 10^{-10} to give extremely accurate numerical integration). As the attractor typically stays within $(x, y) \in [-30, 30] \times [-30, 30]$ we choose our initial conditions to be roughly in this range. We repeat this for $\rho = 10, 28, 40$. From there, we train the neural network to take in (x, y, z) at time t and output the point at time $(t + \Delta t)$ by staggering these trajectories in time. 20% of the data points are set aside as part of a validation set during training and cross-validation on this set was used to fine-tune the structure of the network. Once training was complete, we tested the network by generating completely different initial conditions and finding their trajectory through numerical integration. In testing, we also included trajectories from systems with ρ values not present in the training set, $\rho = 17, 35$, to see how well the network generalizes. Ideally, the network-generated trajectory sticks with the numerically calculated one for as long as possible.

Next, we tried to train a network to recognize the time, in iterations, until a transition between the two lobes of the Lorenz attractor will occur. Since the transition times can be easily calculated once the transition iterations are known, we simply trained our neural network to predict whether or not (1 or 0) a transition

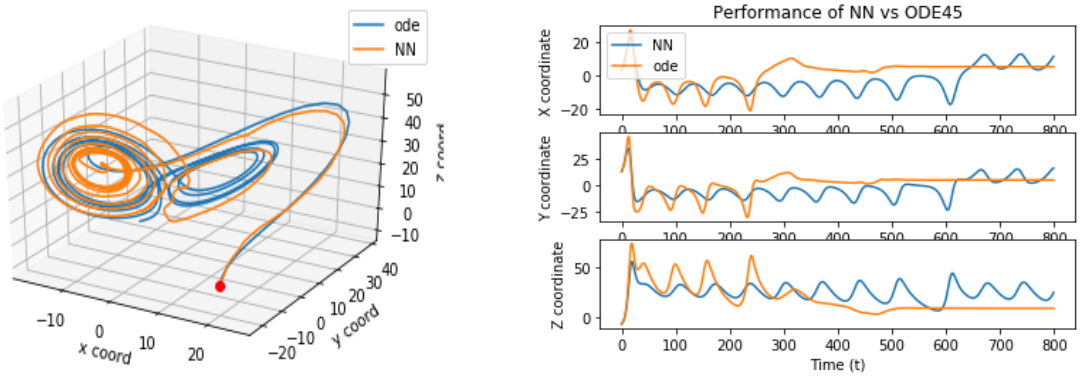

Figure 2: Neural network predictions versus numerical integration with $\rho = 28$ (in training data). The right shows individual coordinates over time.

would occur in the next iteration. Once we knew when transitions occurred, calculating the time to a transition is simple. We saw that the two lobes are roughly separated by the plane $x = 0$ so we looked for times when the trajectory crosses that plane and used that as our transition condition. This time, we fixed $\rho = 28$. Again, we generated trajectories using numerical integration from a number of random initial conditions for our training/validation sample and used a separate set of initial conditions for testing. In both cases, since the Lorenz system is so low-dimensional we found our neural networks did not need very many layers or nodes to get adequate results (code: lorenz_net.ipynb).

Kuramoto-Sivashinsky Equations

Again, we used high-precision numerical integration to generate solutions to this system at a variety of different initial conditions (code: kuramoto_sivashinsky.m). We trained the network to given the wavefront at time t to predict its form at time $(t+1)$, again splitting the data into a training and validation set. In this case, since the system is higher-dimensional we had to use a much larger network than before to get suitable results. Probably due to this, we noticed overfitting in our initial attempts to train the network: the validation error remained high while training error shrunk to zero. To counteract this, we added an L_1 penality ($\lambda = 10^{-4}$) to the weights of our system to help promote sparsity and discourage overfitting. Once the network was trained, we again generated trajectories from completely new initial conditions to test the network's performance and compared its performance against standard numerical integration (code: ks_net.ipynb).

Reaction-Diffusion System

Again, used high-precision numerical integration to generate time-series data for this system (code: reaction_diffusion.m). This time, however, because the system is so high dimensional we projected it into a lower-dimensional space before learning. To do so, we flattened each of the 2D time steps and stored it as a column of our data matrix. Taking the SVD, we chose a low-rank approximation that captured at least 95% of the energy in the singular values. Then, the neural network was trained to move the system forward in the coordinate system defined by these first few principal components. For comparison, we finally project the predicted coordinates back into their original coordinate system (code: reaction_diffusion_net.ipynb).

Neural Network Training

In all networks, we used the Mean-Squared Error as our loss function and used the ADADELTA optimizer which utilizes an adaptive learning rate. This adaptive learning helps with convergence near minima and helps prevent the vanishing gradient issue by scaling the learning rate appropriately.

Computational Results

Lorenz System

The Lorenz system proved the most difficult to simulate due to its famously chaotic behavior. As we see in **Figure 2**, even when $\rho = 28$ was used, one of the systems present in the training samples, to predict the trajectory of new initial conditions, the system struggled to make accurate predictions. We can see the predictions remain accurate at first, but quickly diverge from the true solution.

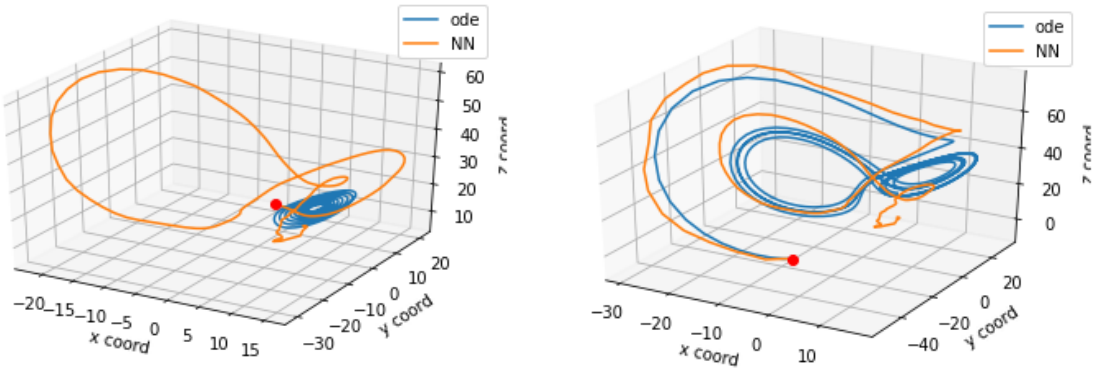

Figure 3: Neural network predictions versus numerical integration with $\rho = 17$ (left) and $\rho = 35$ (right), both outside the training regime. As such, a noticable decrease in performance occurs.

This could be because the chaotic behavior is of the Lorenz system is not adequately captured by the nonlinearities we introduced in our activation functions. As expected, this failure is even more pronounced when we move outside the training regime. In **Figure 3** we see the results of prediction when $\rho = 17$ and $\rho = 35$, two parameters outside our training regime. The network did noticeably worse for $\rho = 17$ likely because the Lorenz system slows down significantly at lower values of this parameter. Lastly, we tried to train a new neural network to predict the transition times. As we can see in **Figure 4** this was very successful, likely due to the simple transition condition ($x = 0$) we used which is clearly linearly separable.

Kuramoto-Sivashinsky Equations

As mentioned above, this system seemed prone to overfitting. The final MSE training error was around 0.03 while the validation error flatlined around 0.75, even after the regularization was added. Consequently, the predictions were more accurate on the training dataset than the testing dataset. As we can see in **Figure 6** where an initial condition from the training sample is used, the neural network seems to capture the overall, somewhat periodic structure of the system quite well. Taking a look at the individual trajectories in the lower figures, we can see the system does a good job of capturing the dynamics in space and a mediocre job in time. However, when we move to the testing data, the performance notably deteriorates. As we can see in **Figure 8**, beyond accurately capturing the initial conditions the neural networks predictions are much less consistent than the true dynamics. This can be seen in the fairly accurate predictions over space (although not all time coordinates chosen generated this accurate of predictions) but poor, chaotic predictions over time. This may suggest our regularization was not enough to prevent overfitting.

Reaction-Diffusion System

After loading the data and transforming it to a matrix as described above, we took its SVD. As we can see in **Figure 10** below, there appears to be around 4 relevant singular values. Thus, we will choose to do a rank 4 approximation. The first four principal components are shown in **Figure 10**. Unsurprisingly, they mainly

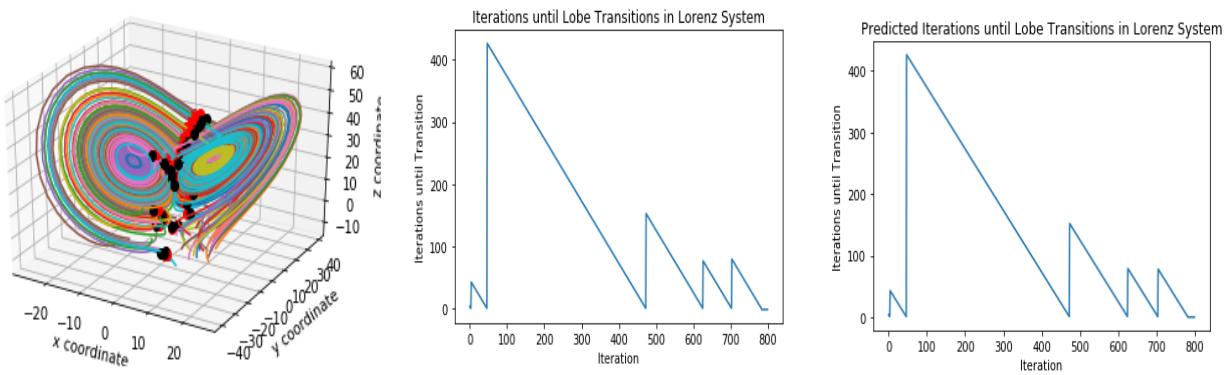

Figure 4: Neural network predictions of lobe transitions. Left highlights the labeling process, where red dots occur right before a transition and black dots immediately following a transition. The middle shows the true predictions of lobe transitions. Right shows the neural networks predictions.

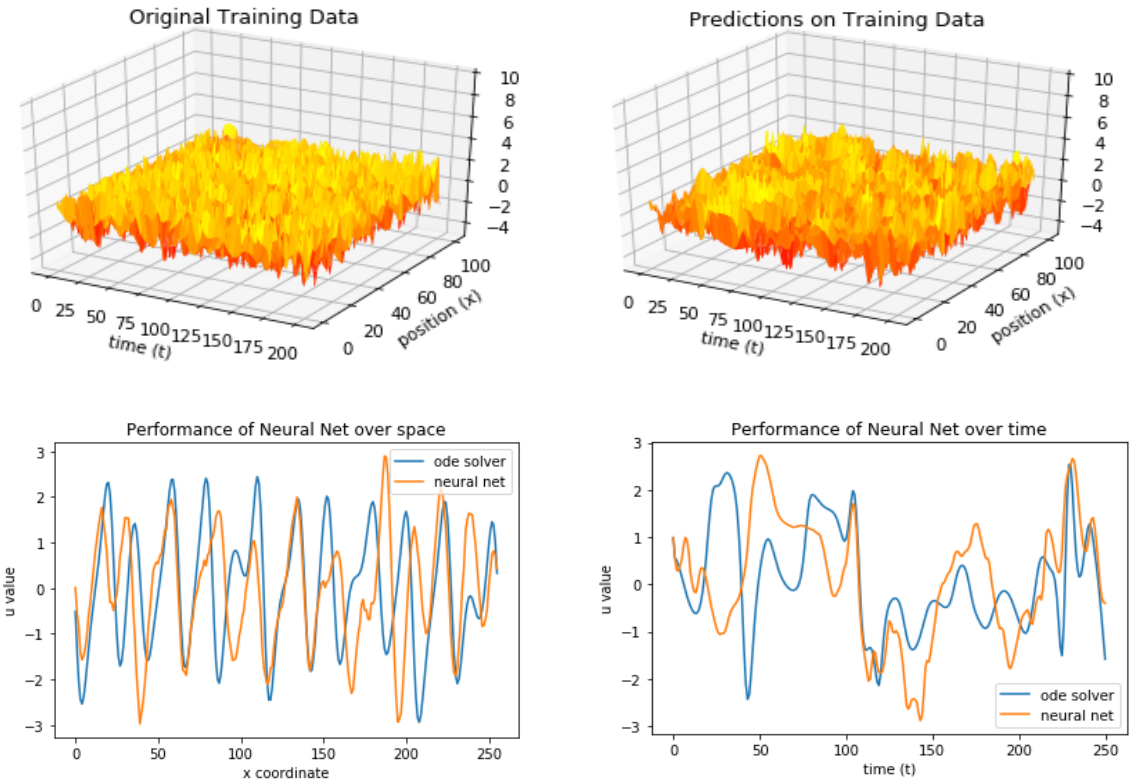

Figure 6: Performance of neural network on KS equations training data. In the top we show the full system and in the two graphs below we show a single slice at a fixed time t (left) or a fixed position x (right). As we can see, the network was typically better at capturing spatial variation than temporal.

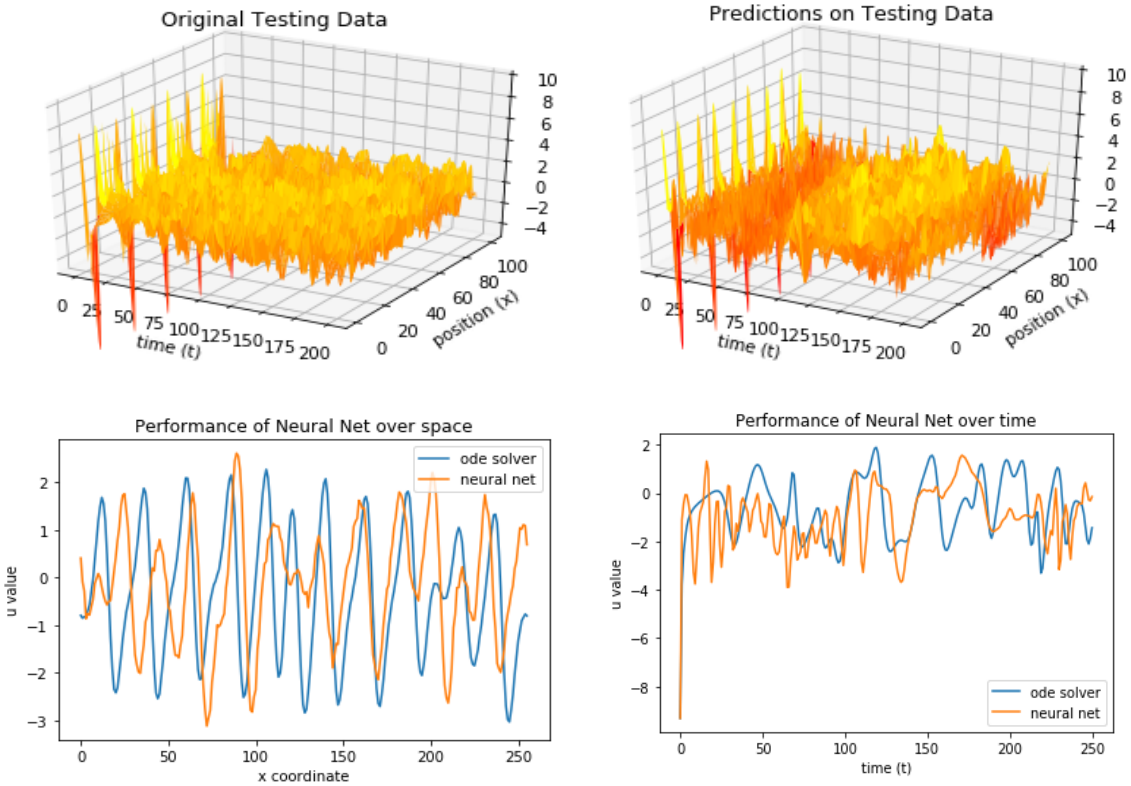

Figure 8: Performance of neural network on KS equations testing data. In the top we show the full system and in the two graphs below we show a single slice at a fixed time t (left) or a fixed position x (right).

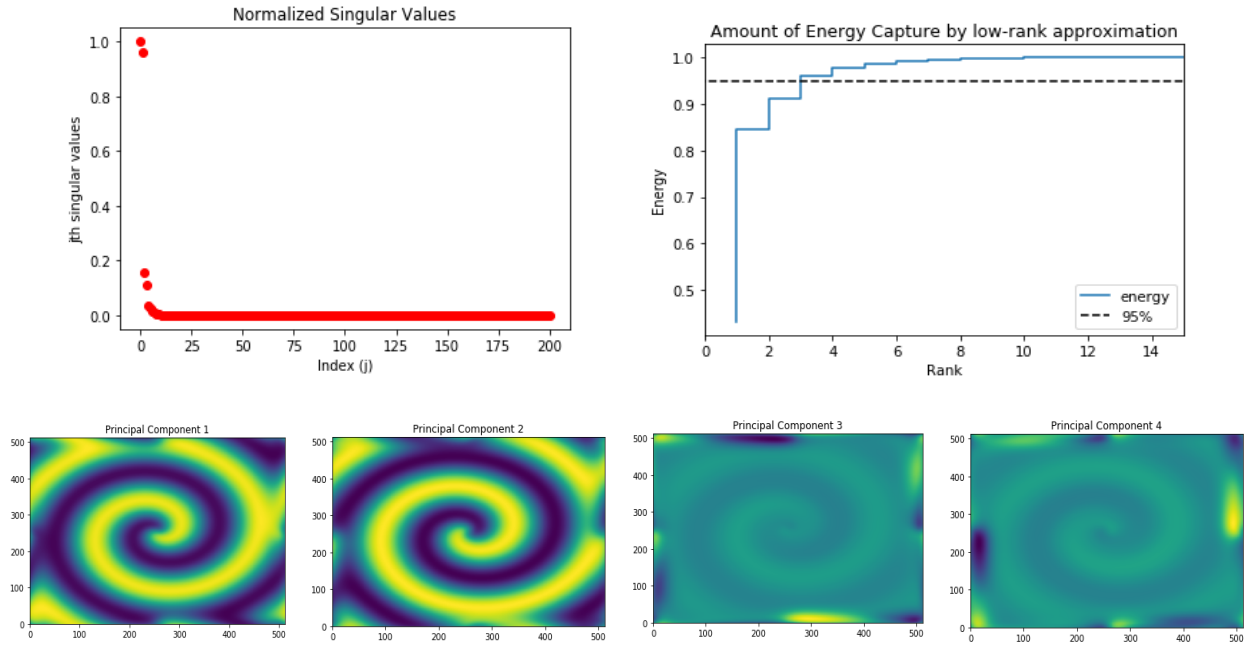

Figure 10: Singular values of the reaction-diffusion system (top left) and their cummulative energy (top right). As we can see, the first four singular values capture over 95% of the energy in the system. Below are the corresponding first four principal components (from left to right) of the reaction diffusion system.

capture the swirling pattern that is characteristic of the system at this initial condition. After training, the neural network had a MSE of less than 10^{-4} on the training data and around 0.0047 on the validation dataset. However, it is important to remember this error is in Principal Component space and thus might not be a truly accurate reflection of the error of the prediction. Nonetheless, as we see in **Figure 11** the low-rank approximation seems to due a remarkable job of capturing the true state. The neural network captures almost all of the structure of the true state (the circular pattern and the correct phase) although the intensities seem to be slightly off.

Summary and Conclusions

As we saw in this report, neural networks are powerful tools for future state predictions, holding up against even high-precision numerical integration. Their malleable architecture makes them suitable for a wide-range of problems. Furthermore, the networks could accomplish this performance in model-free, data-driven way. This makes them an invaluable tool in studying systems where the governing equations are partially or completely unknown. However, as we saw achieving this performance often required several layers with hundreds if not thousands of individual nodes, even for these simple systems. As a result, their behavior and the contribution of individual parameters is often beyond interpretability. This can make it difficult to learn much about the system of interest beyond the future state predictions. Furthermore, as we saw in the case of the Lorenz system, their prediction capabilities do not generalize well: change one parameter and their predictions quickly become inaccurate and would require training to begin from scratch. Furthermore, the chaos of these systems had a large effect on the success of the neural networks. While most could perform well within the training data,

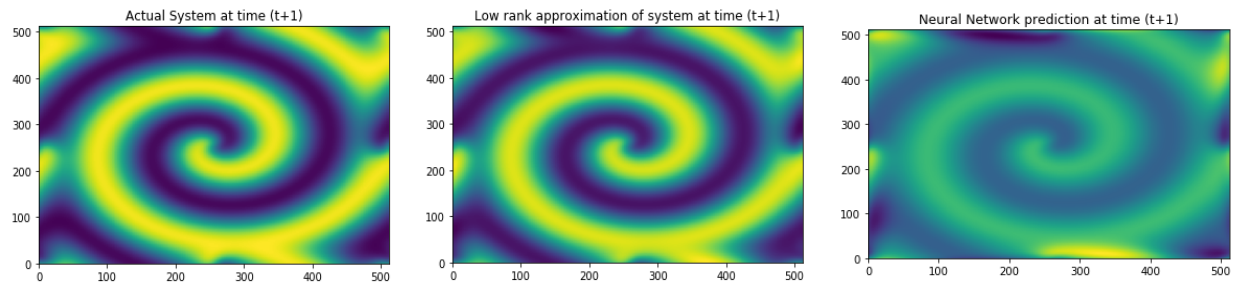

Figure 11: Performance of neural network on testing data. The left is the true state at time $(t+1)$, the middle is the low rank (r=4) approximation of this state, and the right is the neural networks prediction.

performance would fall significantly at an arbitrary randomly chosen point as this was unlikely to be within the training set.

Appendix A

Below is a brief summary of the python libraries/functions I used during this project.

Keras

Machine-Learning library. Easily build, train, and test neural networks.

Sequential(): Create a neural network where all layers follow one after another (output of layer i fed directly to layer $i+1$ rather than combining multiple inputs from separate sources)

model.add(layer): Add a layer to the neural network

Dense(nodes): A fully-connected feedforward neural network layer. Specify the number of nodes, activation function, and any regularizers.

model.compile(): Build the neural network and define the loss function and optimization routine.

model.fit(): Train the neural network and choose some training parameters.

model.predict(): Given a trained neural network, make a prediction given the specified input.

Numpy

This is a library that helps with many common linear algebra operations from matrix multiplication to the SVD.

svd(): Takes SVD of the given matrix.

hstack(): Horizontally stack two matrices (append one to the right of the other).

Scipy

Scientific computing library that provides lots of functions for numerical operations.

odeint(): ODE numerical integrator using RK45.

Matplotlib

This is a plotting library that provides pretty much all of the same plotting features available in MATLAB.

pcolor() Give an image representation of a matrix, coloring each entry in a grid based on its value in the matrix.

plot_surface() 3D surface plot.

Appendix B

Numerical Integration Code

Kuramoto-Sivashinsky Equations

```
clear all; close all; clc

% Kuramoto-Sivashinsky equation (from Trefethen)
% u_t = -u*u_x - u_xx - u_xxxx,  periodic BCs
for j=1:5

N = 2^8;
x = 32*pi*(1:N)'/N;
coeffs = 10*rand(3,1);
u = coeffs(1)*cos(x/coeffs(2)).*(1+sin(x/coeffs(3))); % training data
%u = sin(x) + cos(x); % testing data
v = fft(u);

%%%%%%
%Spatial grid and initial condition:
h = 0.025;
k = [0:N/2-1 0 -N/2+1:-1]'/16;
L = k.^2 - k.^4;
E = exp(h*L); E2 = exp(h*L/2);
M = 16;
r = exp(1i*pi*((1:M)-.5)/M);
LR = h*L(:,ones(M,1)) + r(ones(N,1),:);
Q = h*real(mean( (exp(LR/2)-1)./LR ,2));
f1 = h*real(mean( (-4-LR+exp(LR).*(4-3*LR+LR.^2))./LR.^3 ,2));
f2 = h*real(mean( (2+LR+exp(LR).*(-2+LR))./LR.^3 ,2));
f3 = h*real(mean( (-4-3*LR-LR.^2+exp(LR).*(4-LR))./LR.^3 ,2));

% Main time-stepping loop:
uu = u; % set the initial condition?
tt = 0;
tmax = 200; nmax = round(tmax/h); nplt = floor((tmax/250)/h); g = -0.5i*k;
for n = 1:nmax
t = n*h;
Nv = g.*fft(real(ifft(v)).^2);
a = E2.*v + Q.*Nv;
Na = g.*fft(real(ifft(a)).^2);
b = E2.*v + Q.*Na;
Nb = g.*fft(real(ifft(b)).^2);
c = E2.*a + Q.*(2*Nb-Nv);
Nc = g.*fft(real(ifft(c)).^2);
v = E.*v + Nv.*f1 + 2*(Na+Nb).*f2 + Nc.*f3; if mod(n,nplt)==0
        u = real(ifft(v));
uu = [uu,u]; tt = [tt,t]; end
end
% Plot results:
figure(j)
surf(tt,x,uu), shading interp, colormap(hot), axis tight
xlabel('time')
ylabel('position')
zlabel('wave height')
% view([-90 90]), colormap(autumn);
set(gca,'zlim',[-5 50])

save(strcat('../../data/kuramoto_sivishinky_test', num2str(j), '.mat'),'x','tt',
    'uu')
```

```
55
56 end
57 %%
58 figure(2), pcolor(x,tt,uu.'),shading interp, colormap(hot),axis off
```

Reaction Diffusion

```
clear all; close all; clc

% lambda-omega reaction-diffusion system
%  u_t = lam(A) u - ome(A) v + d1*(u_xx + u_yy) = 0
%  v_t = ome(A) u + lam(A) v + d2*(v_xx + v_yy) = 0
%
%  A^2 = u^2 + v^2 and
%  lam(A) = 1 - A^2
%  ome(A) = -beta*A^2


t=0:0.05:10;
d1=0.1; d2=0.1; beta=1.0;
L=20; n=512; N=n*n;
x2=linspace(-L/2,L/2,n+1); x=x2(1:n); y=x;
kx=(2*pi/L)*[0:(n/2-1) -n/2:-1]; ky=kx;

% INITIAL CONDITIONS

[X,Y]=meshgrid(x,y);
[KX,KY]=meshgrid(kx,ky);
K2=KX.^2+KY.^2; K22=reshape(K2,N,1);

m=1; % number of spirals

u = zeros(length(x),length(y),length(t));
v = zeros(length(x),length(y),length(t));

u(:,:,1)=tanh(sqrt(X.^2+Y.^2)).*cos(m*angle(X+i*Y)-(sqrt(X.^2+Y.^2)));
v(:,:,1)=tanh(sqrt(X.^2+Y.^2)).*sin(m*angle(X+i*Y)-(sqrt(X.^2+Y.^2)));

% REACTION-DIFFUSION
uvt=[reshape(fft2(u(:,:,1)),1,N) reshape(fft2(v(:,:,1)),1,N)].';
[t,uvsol]=ode45('reaction_diffusion_rhs',t,uvt,[],K22,d1,d2,beta,n,N);


for j=1:length(t)-1
ut=reshape((uvsol(j,1:N).'),n,n);
vt=reshape((uvsol(j,(N+1):(2*N)).'),n,n);
u(:,:,j+1)=real(ifft2(ut));
v(:,:,j+1)=real(ifft2(vt));

figure(1)
pcolor(x,y,v(:,:,j+1)); shading interp; colormap(hot); colorbar; drawnow;
end

save('reaction_diffusion_big.mat','t','x','y','u','v')

%%
load ../../data/reaction_diffusion_big
%%
figure(2)
for i=1:length(t)
    pcolor(x,y,u(:,:,i)); shading interp; colormap(hot)
    drawnow
end
```

```
function rhs=reaction_diffusion_rhs(t,uvt,dummy,K22,d1,d2,beta,n,N);

```

```
% Calculate u and v terms
ut=reshape((uvt(1:N)),n,n);
vt=reshape((uvt((N+1):(2*N))),n,n);
u=real(ifft2(ut)); v=real(ifft2(vt));

% Reaction Terms
u3=u.^3; v3=v.^3; u2v=(u.^2).*v; uv2=u.*(v.^2);
utrhs=reshape((fft2(u-u3-uv2+beta*u2v+beta*v3)),N,1);
vtrhs=reshape((fft2(v-u2v-v3-beta*u3-beta*uv2)),N,1);

rhs=[-d1*K22.*uvt(1:N)+utrhs
      -d2*K22.*uvt(N+1:end)+vtrhs];
```