AMATH 563 Nueral Networks

Tyler Chen

ABSTRACT

We outline a how neural networks might be applied to the study of systems governed by possibly unknown PDEs. In particular we show that future state predictions are possible for some systems, even if they are chaotic. We address how high dimensional data can be rank reduced first in order to allow for training on a smaller neural net. While some results are encouraging, we find that it is quite difficult to correctly tune the neural networks in a limited time.

I INTRODUCTION AND OVERVIEW

With recent increases in computation power and data storage capabilities our ability effectively process data computationally has increases substantially. Machine learning uses statistical techniques to gather information about a dataset and use it to make predictions or understand trends. One class of machine learning methods, neural networks, is particularly effective given large datasets. If neural nets can be effectively used to predict solutions of PDEs then we can step away from trying to guess at the PDEs which actually describe systems and move towards generating trajectories using these neural nets.

II THEORETICAL BACKGROUND

Neural Network Basics

Suppose we have sets $\hat{X}_0$ and $\hat{X}_N$ with elements of size m and n respectively. Let $\hat{f} : \hat{X}_0 \to \hat{X}_N$ be an injection. The goal of a neural network is to be able to accurately compute this injection to make predictions about elements of $\hat{X}_0$ for which we do not know the actual value of $\hat{f}$.

A feed forward neural network is a function of the form,

$$X_N = f_N(b_{N-1} + \cdots f_2(b_1 + W_1 f_1(b_0 + W_0 x_0))) \quad (1)$$

where the the "activation" functions $f_j : \mathbb{R} \to \mathbb{R}$ are applied componentwise.

Each "layer" is of the form,

$$x_{j+1} = f_{j+1}j(W_j x_j + b_j), \qquad j = 0, 1, \ldots, N-1$$

Note that W_j can be of any shape compatible with x_j with the constraint that W_{N-1} must also give something of the dimension of x_N.

The goal is to set the parameters W_j and b_j so that the neural network is equal to $\hat{f}$. However, it is clear that unless we know the output of f on all of $\hat{X}_0$ this will not be possible even if our neural net has the same form as $\hat{f}$. Since in general we would like to be able to make predictions about unknown data we will have to settle for the neural network being an approximation to $\hat{f}$.

More specifically, let $X_0 \subset \hat{X}_0$ and suppose the value of $\hat{f}(x)$ is known for all $x \in X_0$. For convenience we will write X_0 as a matrix of size $n \times t$. Define $X_N \subset \hat{X}_N$ as the $m \times t$ matrix found by applying $\hat{f}$ to the columns of X_0. We will use the data from X_0 to train the neural net to (hopefully) give us a good approximation for $\hat{f}$. Since the neural network can take any element of $\hat{X}_0$ as input, we hope that it can be used for predictions about the value of $\hat{f}$ on these elements.

Loss functions

Given a neural network we would like to train it to be able to make predictions. In order to do this we need some way of saying what a good prediction is. In general this is done by defining a loss function which is small when the network gives good predictions, and large when it gives bad predictions.

More specifically, let $X_0 \in \mathbb{R}^n$ and $X_N \in \mathbb{R}^m$ be the input data and target data respectively. Let $f : \mathbb{R}^n \times \mathbb{R}^Z \to \mathbb{R}^m$ be a neural network as defined above, where Z is the number of free parameters. The residual error of the network for a fixed set of parameters $\beta \in \mathbb{R}^Z$ is $Y - f(X, \beta)$.

Ideally the residual error is zero. By picking some metric with which to measure the residual error, we have a minimization problem. In this paper we often chose to minimize the mean square error, which amounts to solving,

$$\min_{\beta} \sum_{x_0 \in X_0} \left\| \hat{f}(x_0) - f(x_0, \beta) \right\|_2^2 \qquad (2)$$

Optimizers

In order to minimize expressions such as (2) an optimization algorithm must be applied. Since the input and output lie in such high dimensional spaces, the loss function often has many local minimum or saddle points. As such, traditional methods such as gradient decent may not work very well. Instead there are many popular gradient based algorithms which introduce some stochasticity in order to avoid "getting stuck". Since our neural network can be expressed as the composition of fairly straightforward functions, the gradient with respect to entries of β can be easily computed using the chain rule (done by something called back-propagation).

SVD and Data reduction

When training a neural network the size of the input and output are determine by the data given. This means the network may have to be very large the the beginning and end nodes. On approach to reduce the size of the network is to project the input and target data into lower dimensional subspaces and then train on the projections. This is equivalent to the first and last hidden layers of the neural net being of smaller size and with fixed weights. If the weights are not fixed, the net will determine the "best" projection. However, since the SVD already computes the optimal low rank approximation in the 2-norm or Frobenius norm, it may be beneficial to set these weights so that the net does not have to learn them.

Suppose our input data X is in $\mathbb{R}^N$ and we have T samples. We then have a $N \times T$ data matrix. We would like to find a subspace of $\mathbb{R}^N$ of lower dimension where our data can be approximated well. Our data has a rank k reduced SVD,

$$X \approx U\Sigma V^* \qquad (3)$$

where U is $N \times k$, Σ is $k \times k$, and V^* is $k \times T$. We interpret this in the following way: the columns of U are the dominant modes in our data, and the rows of ΣV^* tell us how these modes vary in time. We can therefore use the columns of ΣV^*, which are of height k, as our approximation to the input data.

In general the input and target data will not be of the same size or, for time series data, of the same dynamics. However, in the case that they are the rank reduction used on the input and output can be computed simultaneously saving a large computation. While it will no longer be the optimal reduction for each of these, it will be near optimal. In this paper we are generally building steppers, and so the input and target are of the same system at different times. We therefore compute the low dimensional spaces by taking the SVD of the full data set, and then split it into the input and target data sets.

III ALGORITHM IMPLEMENTATION AND DEVELOPMENT

Neural Network Framework

We use the python libraries Keras and Tensorflow for most of our neural nets. In addition the neural network package for MATLAB is occasionally used.

In general, the setup and labeling of the data was the primary focus of this project. It is straightforward to change the net structure and tune other hyperparameters once this has been done. However, doing so is time consuming. If any of the nets work at all this should be taken as very encouraging since they were generally chosen at random without much justification.

Data Generation

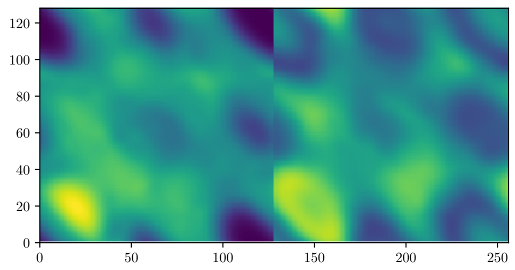
To generate data for our systems we generally use an ODE stepper to solve a system of ODEs corresponding to the discrimination of a PDE. The target data is taken to be the training data one step forward in time (with respect to some fixed time mesh). Multiple initial conditions are generated in a variety of ways depending on

Kuramoto-Sivashinsky Equation

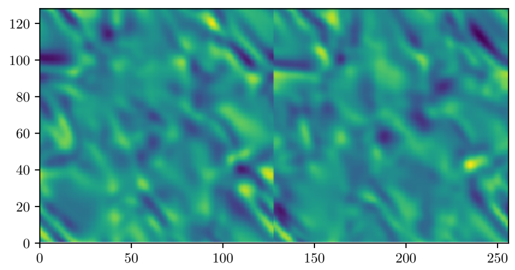
We are provided with a stepper to produce solutions to the Kuramoto-Sivashinsky equation (4) with periodic boundary conditions on a given mesh (with N spatial points and T time points).

$$\frac{\partial u}{\partial t} = -u\frac{\partial u}{\partial x} - \frac{\partial^2 u}{\partial x^2} - \frac{\partial^4 u}{\partial x^4} \qquad (4)$$

In order to train our network to step forward in time we must generate a batch of trajectories stemming from some class of initial conditions. To do this we start with

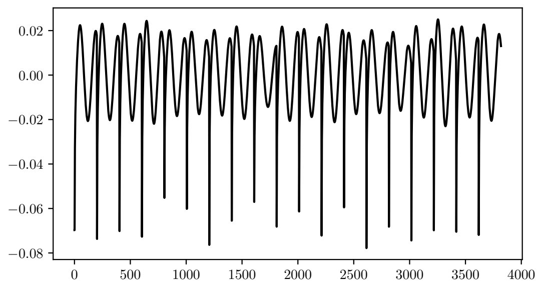


(a) 1st left singular vector reshaped

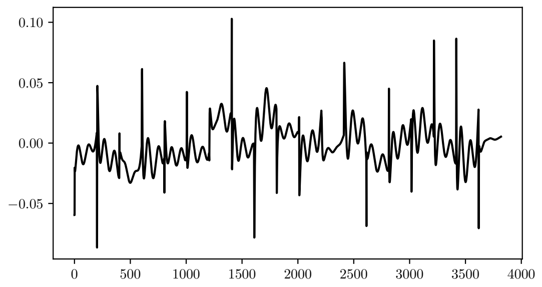


(b) 100th singular vector reshaped

Figure 1: SVD modes for u (left) and v (right)



(a) 1st right singular vector varying in time



(b) 100th right singular vector varying in time

Figure 2: coefficients of SVD modes vs dataset

a zero initial condition and then randomly set a fixed (generally two or three) number Fourier modes to have weights with real and imaginary parts uniformly distributed on $[-N/2, N/2]$. This produces periodic initial conditions which lead to fairly nice behavior. We then generate a solutions with these initial conditions and save them.

The data is then imported to Python and formatted into the training and target data sets by appropriate slicing. A neural net defined in Keras is trained to predict the value of $u(t + \Delta t)$ given the value of $u(t)$.

λ-ω Reaction-Diffusion Equation

We can apply similar methods to a PDE in two dimensions. With,

$$\mathbf{u} = \begin{bmatrix} u \\ v \end{bmatrix}, \qquad \mathbf{D} = \begin{bmatrix} d_1 & \\ & d_2 \end{bmatrix}$$

the Kuramoto-Sivashinsky equation is defined as,

$$\frac{\partial \mathbf{u}}{\partial t} = \begin{bmatrix} \lambda(s) & -\omega(s) \\ \omega(s) & \lambda(s) \end{bmatrix} \mathbf{u} + \mathbf{D}\nabla^2 \mathbf{u} \tag{5}$$

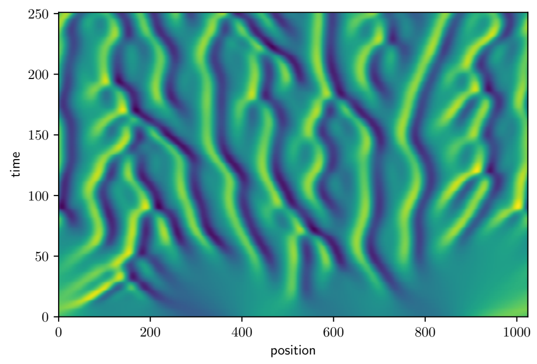
where,

$$s^2 = u^2 + v^2, \qquad \lambda = 1 - s^2, \qquad \omega = -\beta s^2$$

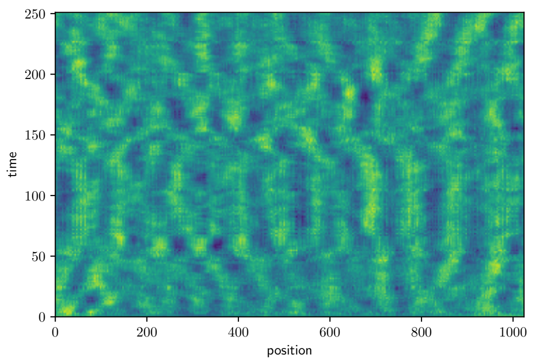
Again we are given a stepper which produces trajectories on a fixed mesh. Like with the KS equation, we generate initial conditions by picking some Fourier modes to be nonzero and use these to generate trajectories which we load into Python.

Since the data is much larger than before, training a neural net (we tried) is very slow. While the size of data used in industry is often massive, they also have massive computers and lots of time. We don't, so to deal with the large data set we reduce the rank of our data and train on the projections. Since the training and target data come from the same space we compute the SVD the entire dataset before splitting it into training and target data. This allows us to find a single subspace for both the training and target data, saving on computing and saving two subspaces (which would turn out to be almost identical since the vast majority of the data in each of these sets would be the same).

Figure 1a shows the dominant left singular vector and Figure 2a shows how the first right singular vector varies in time. The "spikes" down clearly are related to the initial conditions of the data. However, after each spike there is an oscillation in this mode. Moreover, the mode itself has quite a bit of structure, which it seems to have inherited from the way we generated initial conditions. It is not clear whether this structure would die out if we



(a) trajectory generated by `solve_ivp`



(b) trajectory generated by neural net

Figure 3: "actual" and neural net predicted trajectories for Kuramoto-Sivashinsky equation (4)

ran the simulations for longer time. Regardless, since the structure is present in our data set we can use this to train a neural net on the rank-reduced data.

We take the rank k SVD (3) (determined by a parameter flag) and then train the network on the weights of the basis of the subspace spanned by the first k columns of U.

To test our prediction we take a trajectory not used for training, project it to the column span of U, iteratively apply the network, and then embed back to the original space.

Lorenz Equation

The Lorenz equation (6) is a system of ODEs which have chaotic solutions for some parameters. It was demonstrated in [1] that for a fixed Δt a simple neural network can be trained to predict the position at a time $t + \Delta t$ given the position at time t accurately enough that the trajectory determined by iteratively applying the trained neural net matches the trajectory of a given ODE solver almost exactly. Given that the system is chaotic for these values it is somewhat surprising that the neural net manages to follow the same trajectory by only predicting one step at a time. Training data was generating using Scipy's `solve_ivp` with a tolerance of 10^{-10} and evaluating the solution along a mesh of uniformly spaced times.

$$\frac{\partial}{\partial t}\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \sigma(y-x) \\ x(\rho - z) - y \\ xy - \beta z \end{bmatrix} \tag{6}$$

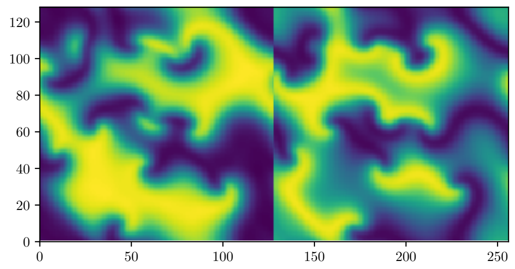
We explore the Lorenz equation in two further ways, namely trying to predict trajectories for varying values of the parameter ρ and trying to predict when the solution will transition from one lobe to another.

The first task is straightforward. In particular, a neural net is trained on data where the input corresponds to the current position as well as the value of ρ, and the output is the position after a time of Δt. More specifically, we fix $\sigma = 10$, $\beta = 8/3$, and train on data with $\rho = 10, 28, 40$. We then try to use this net to predict trajectories for $\rho = 17$ and $\rho = 35$.

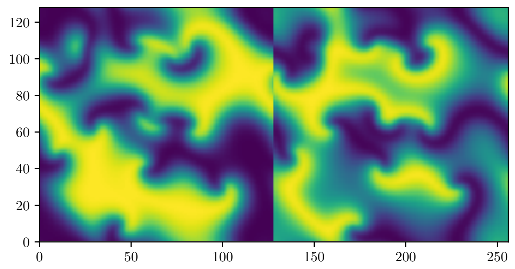
The second tasks requires some interpretation. We decided to train a network to determine how long until a lobe switch. To train such a network requires that we know how long it will be until the solution switches lobes. To do this we first classify what it means to be at a given lobe. This is done by separating the data with plane as shown in Figure 7a. Let c be a normal vector for this hyperplane. Then the sign of $c^T x$ will determine which side of the plane a given point is on.

We pick c roughly in the direction of the vector connecting the two centers in the x-y plane. While there are probably better ways to do this, it seems like the projection to this plane provides a straightforward way to separate points circle different lobes.

Therefore, given a trajectory X, each point in the trajectory can be classified as "left" or "right". Once this is done, we know that transitions across the plane occur when the trajectory switches from "left" to "right" or from "right" to "left". We compute these transition points by taking the difference of consecutive points in $\operatorname{sign}(c^T X)$. If the difference is nonzero then the trajectory has switched lobes. Once we have these transition points it is relatively straightforward label each point in the trajectory with far each point is from the next crossover.



(a) original data



(b) SVD (rank 100)

Figure 4: snapshot of u (left) and v (right) at time $t = 180$

As a minor note, some of the data from the end of the each trajectory is discarded because it is not possible to tell when the next transition will occur without stepping out the solution further. We then train a neural network to try and predict the time until a crossover.

IV COMPUTATIONAL RESULTS

Kuramoto-Sivashinsky Equation

Figure 3a shows a sample trajectory generated by our ODE stepper (not part of traninig data). Figure 3b shows the prediction of the neural net on the same trajectory. However, as time progresses the neural net predicted solution does not really change, while the reaction diffusion equation does continue to evolve.

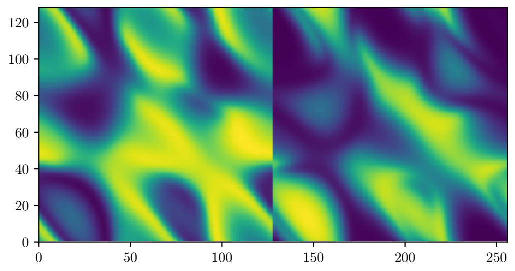
λ-ω Reaction-Diffusion Equation

Figure 4a shows a snaptshot of a trajectory at time $t = 180$. Figure 4b shows the same snapshot after it has been projected into a 100 dimensional subspace determined by the SVD. While a plot of the singular values shows no clear cutoff in the rank, visually it seems that the data is fairly well represented by the first 100 modes. We proceed to train a neural network on the rank reduced data.

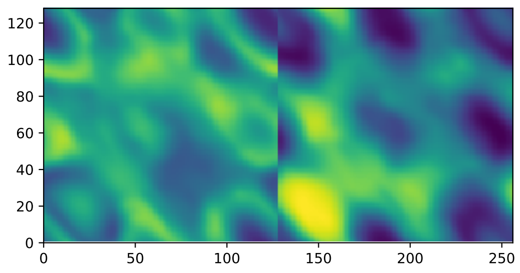
Lorenz Equation

Sample predictions for $\rho = 17$ and $\rho = 35$ produced by the trained net are shown in Figure 6. Note that we do not cross validate in the normal sense, since that would be testing how well our net can predict one step in the future. In fact, the cross validation would show a very high accuracy since we are able to predict the long term behavior relatively well despite only training n the current position.

Figure 7b shows the results of applying the trained net to every point in a trajectory. This plot makes it clear that we are actually able to fairly effectively determine when a jump will occur given only the current position. Note here that the horizontal axis labeling is just the index of the point in the trajectory and has no impact on the neural net's prediction as it is applied independently to each point.

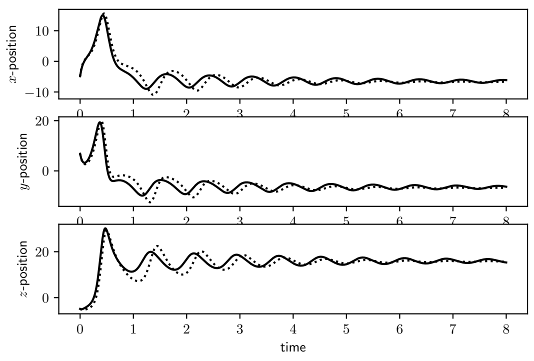


(a) actual trajectory

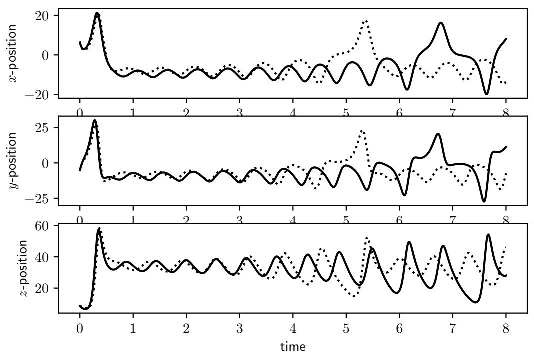


(b) predicted trajectory

Figure 5: "actual" and neural net predicted trajectories for Reaction Diffusion Equation (5) at $t = 15$

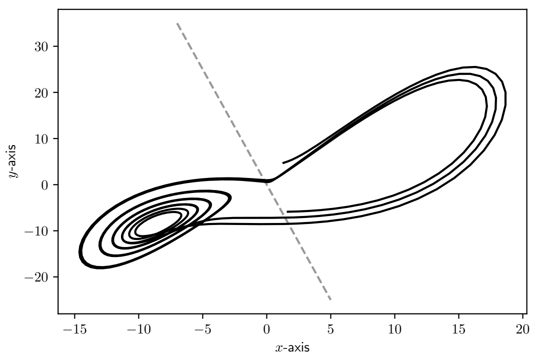


(a) $\rho = 17$

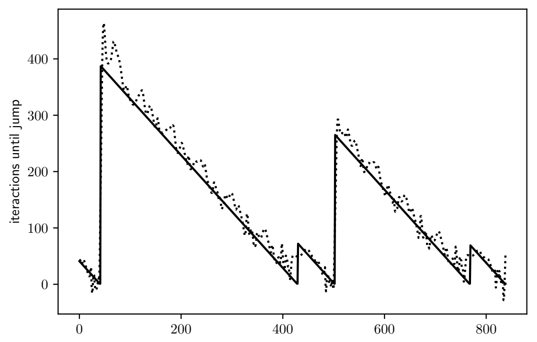


(b) $\rho = 35$

Figure 6: Actual trajectory (solid) vs predicted trajectory (dotted) for variious values of ρ



(a) Projection of sample trajectory onto the x-y plane along with image of plane $y + 5x + 0z = 0$.



(b) Actual time to jump (solid) vs. predicted time to jump (dotted) for sample trajectory .

Figure 7

V SUMMARY AND CONCLUSIONS

It is clear that the trajectories of the Kuramoto-Sivashinsky and Reaction Difussion equations were not effectively predicted. On the other hand, solutions to the Lorenz equation, even for values of ρ which the neural net had never seen, were quite successful. Finally, we were able to train a neural net to predict how long it would take a trajectory to switch lobes.

Due to time constraints only simple nets were used and all hyperparameter training was done by guessing. We instead chose to focus on producing the training data in a scalable way so that in the future it would be a trivial task to generate data at large mesh sizes. With the current state of our codebase, testing new neural nets is as easy as changing a few lines in Keras. Given more time we would have liked to set up convolutional neural nets to see if the local structure of the data would be enough to determine the global behavior. In general we probably did not use enough data, and the nets we used are probably too small to be able to get very good results. However, what we did find is somewhat encouraging and given more time would be worth pursuing.

REFERENCES

[1] J. Nathan Kutz. Course lecture notes. 2018.

VI APPENDIX A

All functions are included in the code in Appendix B as they are specific to the task performed in each of the files.

VII APPENDIX B

```
clear all; close all; clc

% Kuramoto-Sivashinsky equation (from Trefethen)
% u_t = -u*u_x - u_xx - u_xxxx,  periodic BCs

% Generate data from random initial conditions
start_iter = 21;
num_iter = 20;
N = 1024;
x = 32*pi*(1:N)'/N;
for iter = start_iter:(start_iter+num_iter)
    % generate random initial conditions
    num_coeff = 4;
    v = zeros(N,1);
    v(2:1+num_coeff) = (rand(num_coeff,1)-.5) + (rand(num_coeff,1)-.5)*i;
    u = N*real(ifft(v));
    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; tt = 0;
    tmax = 100; 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

    save(['KS_data/N',num2str(N),'/iter',num2str(iter),'.mat'],'x','tt','uu', '-v7')
end
```

```
get_ipython().magic('pylab inline')

import numpy as np
import tensorflow as tf

from scipy import integrate
from scipy.io import loadmat
from mpl_toolkits.mplot3d import Axes3D

import keras
from keras import optimizers
from keras.models import Model,Sequential,load_model
from keras.layers import Input,Dense, Activation
from keras import backend as K
from keras.utils.generic_utils import get_custom_objects
from keras.utils import plot_model

from IPython.display import clear_output


# ## set up keras
def rad_bas(x):
    return K.exp(-x**2)
get_custom_objects().update({'rad_bas': Activation(rad_bas)})

def tan_sig(x):
    return 2/(1+K.exp(-2*x))-1
get_custom_objects().update({'tan_sig': Activation(tan_sig)})

# ## Load KS trajectories
N = 1024
T = 251
num_iter = 40
num_tests = 1
KS_input_data = np.zeros(((T-1)*num_iter,N))
KS_target_data = np.zeros(((T-1)*num_iter,N))

for i in range(num_iter-num_tests):
    u = loadmat('PDECODES/KS_data/N'+str(N)+'/iter'+str(i+1)+'.mat')['uu']
    KS_input_data[i*(T-1):(i+1)*(T-1)] = u[:,:-1].T
    KS_target_data[i*(T-1):(i+1)*(T-1)] = u[:,1:].T

# save data to test on outside of training data
KS_test_data = np.zeros((T*num_tests,N))
for i in range(num_tests):
    u = loadmat('PDECODES/KS_data/N'+str(N)+'/iter'+str(num_iter-i)+'.mat')['uu']
    KS_test_data[i*T:(i+1)*T] = u.T

# plot test trajectory
mpl.rcParams['text.usetex'] = True
m = plt.pcolormesh(KS_test_data)
m.set_rasterized(True)
plt.savefig('img/sample_KS_trajectory.pdf')


# ## Train Neural Network
# define neural net
model = Sequential()
```

```
model.add(Dense(2*N, activation='tan_sig', use_bias=True, input_shape=(N,)))
model.add(Dense(N))

# set up loss function and optimizer
adam1 = keras.optimizers.Adam(lr=.02, beta_1=0.9, beta_2=0.999, epsilon=None, decay=1e-4,
    amsgrad=True, clipvalue=0.5)
model.compile(loss='mean_squared_error', optimizer=adam1, metrics=['accuracy'])

# train data
model.fit(KS_input_data, KS_target_data, epochs=1000, batch_size=3000, shuffle=True,
    validation_split=0.0)

# ## Test Neural Network
# compute NN trajectory
KS_NN_prediction = np.zeros(KS_test_data[0:T].shape)
KS_NN_prediction[0] = KS_test_data[0]
for k in range(T-1):
    KS_NN_prediction[k+1] = model.predict(np.array([KS_NN_prediction[k]]))

# plot NN trajectory
mpl.rcParams['text.usetex'] = True
m = plt.pcolormesh(KS_NN_prediction)
m.set_rasterized(True)
plt.savefig('img/predicted_KS_trajectory.pdf')
```

```
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
ode_options = odeset('RelTol',1e-10, 'AbsTol',1e-11);


t=0:0.05:10;
d1=0.1; d2=0.1; beta=1.0;
L=20; n=32; 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);

max_iter = 20;

for iter=1:max_iter
    u = zeros(length(x),length(y),length(t));
    v = zeros(length(x),length(y),length(t));

    %m=1; % number of spirals
    %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)));
    num_coeff = 3;
    start_coeff = 1;
    ufft = zeros(length(x), length(y));
    vfft = zeros(length(x), length(y));
    ufft(start_coeff:start_coeff-1+num_coeff,start_coeff:start_coeff-1+num_coeff) = (rand(
       num_coeff,1)-.5) + (rand(num_coeff,1)-.5)*i;
    vfft(start_coeff:start_coeff-1+num_coeff,start_coeff:start_coeff-1+num_coeff) = (rand(
       num_coeff,1)-.5) + (rand(num_coeff,1)-.5)*i;
    u(:,:,1)=N*real(ifft2(ufft));
    v(:,:,1)=N*real(ifft2(vfft));

    % 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));
    end

    save(['RD_data/N',num2str(n),'/iter',num2str(iter),'.mat'],'t','x','y','u','v','-v7')
end
```

```
get_ipython().magic('pylab inline')

import numpy as np
import tensorflow as tf

from scipy import integrate
from scipy.io import loadmat
from mpl_toolkits.mplot3d import Axes3D

import keras
from keras import optimizers
from keras.models import Model,Sequential,load_model
from keras.layers import Input,Dense,Reshape,Activation
from keras import backend as K
from keras.utils.generic_utils import get_custom_objects
from keras.utils import plot_model

from IPython.display import clear_output


# ## set up keras
def rad_bas(x):
    return K.exp(-x**2)
get_custom_objects().update({'rad_bas': Activation(rad_bas)})

def tan_sig(x):
    return 2/(1+K.exp(-2*x))-1
get_custom_objects().update({'tan_sig': Activation(tan_sig)})


# ## Load RD trajectories
N = 128
T = 201
```

```
num_iter = 20
num_tests = 1
RD_all_data = np.zeros((num_iter-num_tests,T,N,2*N))
RD_input_data = np.zeros(((T-1)*(num_iter-num_tests),N,2*N))
RD_target_data = np.zeros(((T-1)*(num_iter-num_tests),N,2*N))

for i in range(num_iter-num_tests):
    d = loadmat('PDECODES/RD_data/N'+str(N)+'/iter'+str(i+1)+'.mat')
    u = d['u']
    v = d['v']
    RD_all_data[i,:,:,:N] = u[:,:,:].T
    RD_all_data[i,:,:,N:] = v[:,:,:].T
    RD_input_data[i*(T-1):(i+1)*(T-1),:,:] = RD_all_data[i,:-1,:,:]
    RD_target_data[i*(T-1):(i+1)*(T-1),:,:] = RD_all_data[i,1:,:,:]


RD_test_data = np.zeros((T*num_tests,N,2*N))
for i in range(num_tests):
    d = loadmat('PDECODES/RD_data/N'+str(N)+'/iter'+str(num_iter-i)+'.mat')
    u = d['u']
    v = d['v']
    RD_test_data[i*T:(i+1)*T,:,:N] = u.T
    RD_test_data[i*T:(i+1)*T,:,N:] = v.


# ## Train Neural Network

# define neural net
model = Sequential()
model.add(Dense(N*2*N, activation='tan_sig', use_bias=True, input_shape=(N*2*N,)))
model.add(Dense(N*2*N, activation='sigmoid', use_bias=True))
model.add(Dense(N*2*N))

# set up loss function and optimizer
adam1 = keras.optimizers.Adam(lr=.02, beta_1=0.9, beta_2=0.999, epsilon=None, decay=1e-4,
    amsgrad=True, clipvalue=0.5)
model.compile(loss='mean_squared_error', optimizer=adam1, metrics=['accuracy'])

# train data
model.fit(
    np.reshape(RD_input_data,(-1,N*2*N)),
    np.reshape(RD_target_data,(-1,N*2*N)),
    epochs=1000, batch_size=800, shuffle=True, callbacks=[plot_losses], validation_split=0.0)

# ## Test Neural Network
# compute NN trajectory
RD_NN_prediction = np.zeros(np.reshape(RD_test_data[0:T],(-1,N*2*N)).shape)
RD_NN_prediction[0] = np.reshape(RD_test_data[0],(-1,N*2*N))
for k in range(T-1):
    RD_NN_prediction[k+1] = model.predict(np.array([RD_NN_prediction[k]]))

# plot NN trajectory
mpl.rcParams['text.usetex'] = True
i=80
m = plt.pcolormesh(np.reshape(RD_NN_prediction,(-1,N,2*N))[i])
m.set_rasterized(True)
plt.axis('image')
plt.savefig('img/predicted_RD_'+str(i)+'_trajectory.pdf')
```

```
# ## Compute SVD
# Reshape data and compute rank k approximation to find fixed subspace to which we project
    our spatial ponints at each time.

RD_all_data_reshaped = np.reshape(RD_all_data[:,:,:,],(-1,2*N*N)).T
[uu,ss,vvh] = np.linalg.svd(RD_all_data_reshaped,full_matrices=False)

# plot singular values
mpl.rcParams['text.usetex'] = True
plt.scatter(np.arange(len(ss)),ss,color='k')
plt.savefig('img/singular_values.pdf')

# plot SVD modes
i=0
plt.figure(figsize=(6,3.3))
m = plt.pcolormesh(np.reshape(uu[:,i],(N,2*N)))
m.set_rasterized(True)
plt.axis('image')
plt.savefig('img/svd_mode_'+str(i+1)+'.pdf')
plt.gcf().get_size_inches()

plt.figure(figsize=(6,3.3))
plt.plot(np.arange(len(vvh[i])),vvh[i],color='k')
plt.savefig('img/svd_coeff_'+str(i+1)+'.pdf')
plt.gcf().get_size_inches()


# set rank and take reduced SVD
rank = 100
u = uu[:,:rank]
s = ss[:rank]
vh = vvh[:rank]

# set up trainign data for new NN
SVD_input_data = np.delete(vh,np.s_[200::201],axis=1).T
SVD_target_data = np.delete(vh,np.s_[1::201],axis=1).T
SVD_input_data.shape

# plot rank reduced image in time vs actual image in time
plt.figure(figsize=(6,3.3))
m = plt.pcolormesh(np.reshape(u@np.diag(s)@SVD_input_data[180],(N,2*N)))
m.set_rasterized(True)
plt.axis('image')
plt.savefig('img/uv_t180.pdf')

plt.figure(figsize=(6,3.3))
m = plt.pcolormesh(RD_all_data[0,180])
m.set_rasterized(True)
plt.axis('image')
plt.savefig('img/svd_t180.pdf')


# ## Train Net on SVD data
# define neural net
model = Sequential()
model.add(Dense(2*rank, activation='tan_sig', use_bias=True, input_shape=(rank,)))
model.add(Dense(2*rank, activation='sigmoid', use_bias=True))
model.add(Dense(2*rank, activation='linear', use_bias=True))
model.add(Dense(rank))
```

```
# set up loss function and optimizer
adam1 = keras.optimizers.Adam(lr=.02, beta_1=0.9, beta_2=0.999, epsilon=None, decay=1e-4,
    amsgrad=True, clipvalue=0.5)
model.compile(loss='mean_squared_error', optimizer=adam1, metrics=['accuracy'])

# train data
model.fit(
    SVD_input_data,
    SVD_target_data,
    epochs=1000, batch_size=80, shuffle=True, callbacks=[plot_losses], validation_split=0.0)

# set up data for testing
SVD_test_data = np.reshape(RD_test_data[0:T],(-1,N*2*N))@u

# compute NN trajectory
SVD_NN_prediction = np.zeros(SVD_test_data.shape)
SVD_NN_prediction[0] = SVD_test_data[0]
for k in range(T-1):
    SVD_NN_prediction[k+1] = model.predict(np.array([SVD_NN_prediction[k]]))


# plot NN trajectory at given time
plt.figure(figsize=(6,3.3))
m = plt.pcolormesh(np.reshape(u@np.diag(s)@SVD_NN_prediction[15],(N,2*N)))
m.set_rasterized(True)
plt.axis('image')
plt.savefig('img/svd_prediction_t15.pdf')

# plot NN prediction vs actual trajectory after SVD
plt.figure()
plt.scatter(np.arange(T),SVD_NN_prediction[:,0])
plt.scatter(np.arange(T-1),SVD_input_data[:,0])
plt.show()

# animate NN trajectory
get_ipython().magic('matplotlib notebook')
import matplotlib.animation

t = np.arange(T)
fig, ax = plt.subplots()

def animate(i):
    plt.pcolormesh(np.reshape(u@np.diag(s)@SVD_NN_prediction[i],(N,2*N)))
    plt.axis('image')

ani = matplotlib.animation.FuncAnimation(fig, animate, frames=len(t))
plt.show()
```

```
get_ipython().magic('pylab inline')

import numpy as np
import tensorflow as tf

from scipy import integrate
from mpl_toolkits.mplot3d import Axes3D

import keras
from keras import optimizers
from keras.models import Model,Sequential,load_model
from keras.layers import Input,Dense, Activation
from keras import backend as K
from keras.utils.generic_utils import get_custom_objects
from keras.utils import plot_model

from IPython.display import clear_output


# for fun
def progress_bar(percent):
    length = 40
    pos = round(length*percent)
    clear_output(wait=True)
    print('['+''*pos+' '*(length-pos)+']  '+str(int(100*percent))+'%')


# define system
def lrz_rhs(t,x,sigma,beta,rho):
    return [sigma*(x[1]-x[0]), x[0]*(rho-x[2]), x[0]*x[1]-beta*x[2]];


# wrapper to generate trajecotry
end_time = 8
sample_rate = 100
t = np.linspace(0,end_time,sample_rate*end_time+1,endpoint=True)
def lrz_trajectory(rho):
    sigma=10;
    beta=8/3;
    x0 = 20*(np.random.rand(3)-.5)
    sol = integrate.solve_ivp(lambda t,x: lrz_rhs(t,x,sigma,beta,rho),[0,end_time],x0,t_eval=
        t,rtol=1e-10,atol=1e-11)
    return sol.y


# plot trajectory
x = lrz_trajectory(28)
plt.figure()
plt.gca(projection='3d')
plt.plot(x[0],x[1],x[2])
plt.show()


# ## Generate Data
N = 200
T = 801
rhos = [10,28,40]
input_data = np.zeros((N*(T-1)*len(rhos),4))
target_data = np.zeros((N*(T-1)*len(rhos),3))
for k,rho in enumerate(rhos):
    for i in range(N):
        progress_bar((N*k+i+1)/(N*len(rhos)))
        trajectory = lrz_trajectory(rho)
        input_data[((len(rhos)-1)*k+i)*(T-1):((len(rhos)-1)*k+i+1)*(T-1),:3] = trajectory.T
            [:-1]
        input_data[((len(rhos)-1)*k+i)*(T-1):((len(rhos)-1)*k+i+1)*(T-1),3] = rho
        target_data[((len(rhos)-1)*k+i)*(T-1):((len(rhos)-1)*k+i+1)*(T-1),:3] = trajectory.T
            [1:]
```

```
# ## Define Neural Network
# set up keras
def rad_bas(x):
    return K.exp(-x**2)
get_custom_objects().update({'rad_bas': Activation(rad_bas)})

def tan_sig(x):
    return 2/(1+K.exp(-2*x))-1
get_custom_objects().update({'tan_sig': Activation(tan_sig)})

# define neural net
model = Sequential()
model.add(Dense(10, activation='tan_sig', use_bias=True, input_shape=(4,)))
model.add(Dense(10, activation='sigmoid', use_bias=True))
model.add(Dense(10, activation='linear', use_bias=True))
model.add(Dense(3))

# set up loss function and optimizer
adam1 = optimizers.Adam(lr=.005, beta_1=0.9, beta_2=0.999, epsilon=None, decay=1e-5, amsgrad=
    True, clipvalue=0.5)
model.compile(loss='mean_squared_error', optimizer=adam1, metrics=['accuracy'])

# train data
model.fit(input_data, target_data, epochs=10000, batch_size=1000, shuffle=True, callbacks=[
    plot_losses], validation_split=0.0)


# ## Test NN Predictions
rho=35
xsol = lrz_trajectory(rho)
x = np.zeros((3,end_time*sample_rate+1))
x[:,0] = xsol[:,0]
for i in range(end_time*sample_rate):
    x[:,i+1] = model.predict(np.array([np.append(x[:,i],rho)]))

# plot actual trajectory vs NN predicted trajectory
mpl.rcParams['text.usetex'] = True
plt.figure()
plt.gca(projection='3d')
plt.plot(x[0],x[1],x[2], color='k', linestyle=':')
plt.plot(xsol[0],xsol[1],xsol[2], color='k')
plt.savefig('img/NN_lrz35_trajectory.pdf')

for i in range(3):
    plt.figure()
    plt.plot(t,x[i], color='k')
    plt.plot(t,xsol[i], color='k', linestyle=':')
    plt.savefig('img/NN_lrz35_trajectory_'+str(i)+'.pdf')
```

```
get_ipython().magic('pylab inline')

import numpy as np
import tensorflow as tf

from scipy import integrate
from mpl_toolkits.mplot3d import Axes3D

import keras
```

```
from keras import optimizers
from keras.models import Model,Sequential,load_model
from keras.layers import Input,Dense,Reshape,Activation
from keras import backend as K
from keras.utils.generic_utils import get_custom_objects
from keras.utils import plot_model

from IPython.display import clear_output


# define system
def lrz_rhs(t,x,sigma,beta,rho):
    return [sigma*(x[1]-x[0]), x[0]*(rho-x[2]), x[0]*x[1]-beta*x[2]];

# wrapper to genrate trajectory
end_time = 10
sample_rate = 100
t = np.linspace(0,end_time,sample_rate*end_time+1,endpoint=True)
def lrz_trajectory(rho):
    sigma=10;
    beta=8/3;
    x0 = 20*(np.random.rand(3)-.5)
    sol = integrate.solve_ivp(lambda t,x: lrz_rhs(t,x,sigma,beta,rho),[0,end_time],x0,t_eval=
        t,rtol=1e-10,atol=1e-11)
    return sol.y


# ## Categorize data by lobe
# Pick seperating hyperplane $0 = c^Tx$ where $c=(5,1,0)$

mpl.rcParams['text.usetex'] = True

# separate data to left and right nodes
c=([5,1,0])
L=x[:,np.where((c@x).T>=0)[0]]
R=x[:,np.where((c@x).T<0)[0]]

# plot left and right nodes of trajectory in 3d space
plt3d = plt.figure().gca(projection='3d')
plt3d.scatter(L[0],L[1],L[2],marker='.')
plt3d.scatter(R[0],R[1],R[2],marker='.')
plt.show()


# ## Generate Data
# label data with how far a given point is to crossing
def generate_timed_trajectory(rho):
    # define normal vector
    c = np.array([5,1,0])

    # get trajectory
    x = lrz_trajectory(rho)

    # classify points as left or right
    classes = np.sign(c@x)
    # compute which indicies correspond to transitions by
    #    checking if there is a sign change in classification of points
    transition_ind=np.where(np.convolve(classes,[1,-1],mode='valid')!=0)
    # this will be the time it takes to the next jump
    # start with ones where there are jumps
    time_to_jump = np.zeros((len(x.T)))
    time_to_jump[transition_ind] = np.ones(len(transition_ind))

    # count backwards from ones
    current_time_to_jump = 0
    jumping = False
    for j in range(len(time_to_jump)):
        if time_to_jump[-j]==1:
            current_time_to_jump=1
            jumping=True
        elif jumping:
            time_to_jump[-j]=current_time_to_jump
            current_time_to_jump+=1

    # delete end of data where we do not know how long until the next crossing
    ind_to_save = np.where(time_to_jump!=0)[0]
    clipped_data = np.vstack([x,time_to_jump])[:,ind_to_save]

    return clipped_data

# generate data over multiple trajectories
max_iter = 100
D = np.empty((4,0))
for i in range(max_iter):
    D = np.concatenate([D,generate_timed_trajectory(28)],axis=1)

# format data for training
input_data = D[:3].T
target_data = D[3].T


# ## Set up Neural Network
# set up keras
def rad_bas(x):
    return K.exp(-x**2)
get_custom_objects().update({'rad_bas': Activation(rad_bas)})

def tan_sig(x):
    return 2/(1+K.exp(-2*x))-1
get_custom_objects().update({'tan_sig': Activation(tan_sig)})


# define neural net
model = Sequential()
model.add(Dense(10, activation='tan_sig', use_bias=True, input_shape=(3,)))
model.add(Dense(10, activation='sigmoid', use_bias=True))
model.add(Dense(10, activation='linear', use_bias=True))
model.add(Dense(1))

# set up loss function and optimizer
adam1 = optimizers.Adam(lr=.005, beta_1=0.9, beta_2=0.999, epsilon=None, decay=1e-5, amsgrad=
    True, clipvalue=0.5)
model.compile(loss='mean_squared_error', optimizer=adam1, metrics=['accuracy'])

# train data
model.fit(input_data, target_data, epochs=1000, batch_size=1000, shuffle=True, callbacks=[
    plot_losses], validation_split=0.0)
```

```
# ## Test NN Predictions
# see how model predicts over an entire trajectory.
x = generate_timed_trajectory(28)

next_time_to_jump = np.zeros(len(x.T))
for k,xpos in enumerate(x.T):
    next_time_to_jump[k] = model.predict(np.array([xpos[:3]]))


mpl.rcParams['text.usetex'] = True
# plot trajectory and separating hyperplane
plt.figure()
plt.plot(np.linspace(-7,5),-5*np.linspace(-7,5),color='.6',linestyle='--')
plt.plot(x[0],x[1],color='k')
plt.xlabel('$x$-axis')
plt.ylabel('$y$-axis')
plt.savefig('img/separating_hyerplane.pdf')

# plot actual time to crossing and NN predicted time to crossing
plt.figure()
plt.plot(np.arange(len(x.T)),x[3], color='k')
plt.plot(np.arange(len(x.T)),next_time_to_jump, color='k', linestyle=':')
plt.ylabel('iteractions until jump')
plt.savefig('img/jump_predictor.pdf')
```