Neural Networks for Dynamical Systems

Stephen Jonany[1]

Abstract— **We explore the ability of neural networks to capture the hidden dynamics of popular dynamical systems such as Kuramoto-Sivashinsky, reaction-diffusion, and Lorenz. We demonstrate that when trained on training trajectories, the neural network models are sometimes capable of predicting full trajectories from initial conditions that they have not seen before.**

I. Introduction and Overview

Neural networks are powerful and flexible models that are capable of approximating complex and non-linear functions through multiple compositions of simple activation functions. The success of this approach is wide-reaching, ranging from image classification to language translation.

We explore yet another application: prediction of dynamical system evolution. That is, given just trajectory data, we test if neural networks can figure out the true dynamics behind the trajectory data, and use it to predict how any initial condition can evolve over time. We explore this across 3 different dynamical systems, namely Kuramoto-Sivashinsky, reaction-diffusion, and Lorenz.

II. Theoretical Background

A. Kuramoto-Sivashinsky (KS) Equation

The Kuramoto-Sivashinsky (KS) equation [1] is a model of flame instability that is known to exhibit chaos. The variable of interest, $u(x,t)$, depends both on space and time. x, the spatial coordinate, is periodic, and we will examine only the window $[-L/2, L/2]$ for some constant period L. The full partial differential equation can be seen in Equation 1.

$$u_t + u_{xxxx} + u_{xx} + uu_x = 0, x \in [-L/2, L/2] \tag{1}$$

Note that u_x and u_t are the partial derivatives with respect to space and time respectively.

B. Reaction-diffusion System

The reaction-diffusion system in Equation 2 describes the dynamics of two variables that vary in two spatial coordinates and time, $u(x,y,t)$ and $v(x,y,t)$.

$$\begin{aligned}
u_t &= \lambda(A)u - \omega(A)v + d_1(u_{xx} + u_{yy}) \\
v_t &= \omega(A)u + \lambda(A)v + d_2(v_{xx} + v_{yy}) \\
A^2 &= u^2 + v^2 \\
\lambda(A) &= 1 - A^2 \\
\omega(A) &= -\beta A^2 \\
x &\in [-L/2, L/2]
\end{aligned} \tag{2}$$

The constant parameters are d_1, d_2, β. Like the KS Equation 1, the system is defined by partial differential equations, where the spatial coordinates x and y are periodic, and we will only focus on the window $[-L/2, L/2]$. The difference is that now we have two independent spatial coordinates x and y.

C. Lorenz Equations

The Lorenz equations as described in Equation 3 is a dynamical system with three state variables that is well known for its chaotic behavior. As we see later on, common trajectories can be described as two butterfly wings (or lobes), where due to chaos, the transition between the two lobes is unpredictable and highly sensitive to initial conditions.

$$\begin{aligned}
\frac{dx}{dt} &= \sigma(y - x) \\
\frac{dy}{dt} &= x(\rho - z) - y \\
\frac{dz}{dt} &= xy - \beta z
\end{aligned} \tag{3}$$

The three parameters are σ, ρ, β, and for the later prediction tasks, we will only be varying ρ.

D. Neural Network for Regression

In regression, one is asked to discover a function f that captures the relationship between input vectors x_i and output vectors y_i. A common objective function is the mean squared error (MSE), where if there are N samples, then the objective function is as given in Equation 4.

$$argmin_f \frac{1}{N} \sum_{i=1}^{N} ||f(x_i) - y_i||^2 \tag{4}$$

Neural network is a powerful regression model that approximates any function f through a composition of multiple activation functions f_i, each with its own parameters w_i. That is, a neural network of K layers converts an input vector x_i to a predicted output vector $\tilde{y_i}$ following Equation 5.

$$\tilde{y_i} = f(x_i, w_1, ..., w_k) = f_k(w_k, f_{k-1}(w_{k-1}, ...f_1(w_1, x_i))) \tag{5}$$

With the f_i's predetermined before training, the goal of the training step is thus to find the weights w_i's that minimize the MSE on the training set, as shown in Equation 6.

$$argmin_{w_1,...,w_k} \frac{1}{N} \sum_{i=1}^{N} ||f(x_i, w_1, ..., w_k) - y_i||^2 \tag{6}$$

[1]Stephen is a Master's student in Applied Mathematics at the University of Washington. Email: sjonany@uw.edu

A common function minimization approach is gradient descent, where one first computes the gradient of the loss function with respect to each w_i, which points in the direction of steepest ascent, then takes a step of length η opposite to that. That is, at each time step, we would update our tentative w with $-\eta\nabla f$, until we reach convergence.

Other variants of gradient descent will compute the gradient not with the entire dataset, but instead with a small batch at a time. We will be varying our batch size in our implementation later on, and will be using the Adam [2] optimizer, which is a variant of gradient descent, but with a dynamically updating learning rate.

Finally, it is worth noting that when one computes the gradient of a nested function as in Equation 5, one would invoke chain rule many times. The process of propagating the chain rule updates to all the weights, starting from the layer closest to the output layer, to the input layer, is referred to as backpropagation.

E. Neural Network for Predicting Dynamics

In the following sections, we will be using the neural network to predict state trajectories of various dynamical systems. The goal of the neural network is then to learn a function f that moves the state variables forward in time, as described in Equation 7.

$$x(t + \Delta t) = f(x(t)) \tag{7}$$

The original dataset would contain the state vectors $x(t)$ across various timesteps. For training, given a single trajectory $x(t_1), x(t_2), ..x(t_n)$, we would create $n-1$ training samples where the input vector is $x(t_k)$ and the output vector is $x(t_{k+1})$, where k ranges from 1 to $n-1$. This is now a regression problem, and so we can obtain the optimal weights as per Section II-D. When one is presented with multiple trajectories from different initial conditions, one would simply concatenate the additional training samples as new rows. In Sections III-B, III-C, III-D, we will explain how the datasets of each task can be converted into the $x(t_1), x(t_2), ..x(t_n)$ trajectory form, that is, a collection of state vectors that fluctuate over time. Once we have arrived at this form, the training and testing process for the neural network is as described by this section.

For testing, the only information we need from the test set is just the initial condition $x(t_0)$. We then repeatedly feed $x(t_k)$ into the neural network model, which spouts out a prediction of $x(t_{k+1})$, to generate a complete trajectory that we can later compare to the actual trajectory from the same initial condition.

III. Algorithm Implementation and Development

A. Software Packages

All the codes are written in Python. The provided MATLAB codes are only slightly tweaked. The following Python libraries are used.

- Tensorflow is used for training the neural network models.
- SciPy is used for k-fold cross validation.
- NumPy is used for efficient 2D matrix manipulations.
- Matplotlib is used to generate all the plots.
- mplot3d is used for the 3d plots.

B. Dataset for Kuramoto-Sivashinky Equation

`kuramoto_sivashinky.m` code was used to generate our dataset. The $u(x, t)$ as described in Section II-A is discretized, with 1024 equally spaced spatial points and 251 timesteps. The code achieved this by first setting an initial condition $u(x, t = 0) = cos(\frac{x}{16})(1 + sin(\frac{x}{16}))$, then advancing all the spatial coordinates forward in time in fourier domain, one timestep at a time until 250 additional timesteps have been generated. That is, our training dataset is a matrix of dimension 1024×251, and each cell is a scalar value corresponding to u at that specified space and time. The test set is of the same dimension, with the only difference being that the initial condition is instead $u(x, t = 0) = sin(\frac{x}{16})(2 + cos(\frac{x}{16}))$.

C. Dataset for Reaction-diffusion System

We make use of the provided `reaction_diffusion.m` code to generate our dataset. The parameters are fixed to $d_1 = 0.1$, $d_2 = 0.1$, $\beta = 1$, $L = 20$. The $u(x, y, t)$ and $v(x, y, t)$ as described in Section II-B is discretized, with x and y each spanning 512 equally spaced spatial points, and timesteps of 201. The code achieved this by first setting an initial condition $u(x, y, t = 0) = v(x, y, t = 0) = x^2 + y^2$, then advancing all the spatial coordinates forward in time in fourier domain. We then use the first 151 timesteps for training, and the rest for testing.

We have previously described in homework 2 and 3 how one performs an SVD on images and videos. We will keep our discussion of SVD and reconstruction very brief here. Interested readers can refer to the homework 3 writeup at `https://github.com/sjonany/AMATH563-public/blob/master/hw3/writeup/final.pdf` for more details. The small novelty here is that we now have to construct a big matrix M to represent our $u(x, y, t)$ and $v(x, y, t)$. Each column of M represents a certain time t'. Each column is a 1D vector with $2 \times 512 \times 512$ elements, whereby we first flatten $u(x, y, t')$ in row major order, then concatenate it with $v(x, y, t')$ also in row major order. One can think of this flattened vector as $w(i)$, where i is an x,y pair. Now that we have a 2D matrix, applying SVD $M = U\Sigma V^*$ gives us U, the spatial basis out of which the u and v of various timesteps can be well-reconstructed from. Then, to project a certain $u(x, y), v(x, y)$ to a lower dimension r, one would first perform the same flattening process to get a 1D vector w, then just like homework 3 face reconstruction, where we have U_r be the first r columns of U, one would compute $U_r^T w$ to obtain the low-dimensional projection of w.

For our training set, we focus on the first 151 timesteps and construct the 2D matrix M as described in the previous paragraph, which will have $2 \times 512 \times 512$ rows and 151 columns. We compute the average vector over all the column vectors and subtract this from each column. We then perform SVD and save the first r columns of U as our spatial basis, where r is tuned to capture enough energy for reconstruction. Then, U_r will have $2 \times 512 \times 512$ rows and r columns. Our training set is then the projection $U_r^T M$, which has r rows and 151 columns, corresponding to a low-dimensional vector of r elements, varying across 151 timesteps.

For the test set, we use the last 100 timesteps. Note that for our prediction, we will only need the initial condition to be transformed to low dimension. Where the first column of the test set matrix is w, we simply compute $U_r^T w$ to get an $r \times 1$ vector, and feed into the neural network as described in Section II-D to obtain the predicted low-dimensional trajectory P of dimension $r \times 100$. Finally, we can re-project P back to the predicted $u(x, y, t), v(x, y, t)$, by computing $U_r P$, followed by the addition of the average column vector computed from the training set to all the columns, then undoing the flattening process. For the un-flattening, for each column that corresponds a single time t', we extract the first 512×512 elements and undo the row-major order flattening to obtain $u(x, y, t')$, and do the same process for the other half of the flattened column to obtain $v(x, y, t')$.

D. Dataset for Lorenz Equations

`CH06_SEC06_1_NNLorenz.m` code from the textbook [3] was used to generate our dataset. The parameters are fixed to $\beta = \frac{8}{3}, \sigma = 10$, and for ρ, we used 10, 28, 40 for the training set, and 17, 35 for the test set. For each parameter set, we generate 100 trajectories from random initial conditions, spanning 8 seconds, with $dt = 0.01$s. Thus, we will have multiple training matrices corresponding to the different initial conditions and parameter setups, each having 3 rows corresponding to x, y, z, and many columns corresponding to the flow of time.

For labeling the transitions between lobes, as seen in Figures 1 and 2, a simple heuristic of whether or not $x(t) > 0$ suffices as a transition indicator. Thus, we consider transition times to be the time points where $x(t)$ has changed sign. Now that we have defined transition time, the regression task is then, given $x(t), y(t), z(t)$, we predict the number of timesteps (that is, number of dt's) from t until we reach a transition time. We generate a 10000-timestep-long trajectory, and use the first 7000 timesteps for training, and the last 2000 timesteps for testing. We also remove the last few timesteps after the very last transition time.

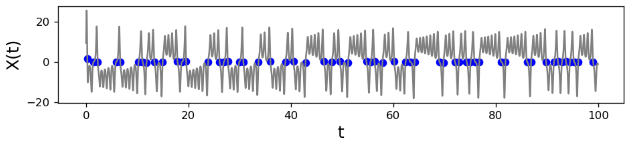


Fig. 1: A sample X(t) trajectory from Lorenz Equations with $\rho = 28$. In blue are the labelled transition points.

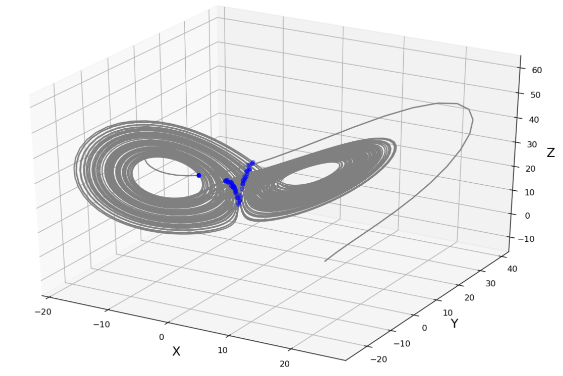


Fig. 2: A sample trajectory from Lorenz Equations with $\rho = 28$. In blue are the labelled transition points. This is the 3D version of Figure 1.

E. Training a Neural Network

The process of training a neural network involves a variety of tunable hyperparameters. After some experimentation, here is a rough guideline of how we arrive at the final network structures.

We first start with three layers of sigmoids (see Equation 8) as activation functions, and each layer having 10 nodes.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{8}$$

We feed in the training data and observe the training error graph, an example of which can be seen in Figure 3. We make custom tweaks to the activation function choices across each layer, and pick the combination that gives the lowest final training MSE. In our experiments, the sigmoid and RELU functions gave the best results. With the activation function choice fixed, we then proceed to cross-validate on the number of nodes for each layer. We have tried imposing L1 and L2 regularization and dropouts but neither produced good trajectories. Finally, with the fixed network structure, we run on the entire training set with a reduced Adam learning rate. We continuously refer to the training error graph to see if more epochs could further reduce the training error. Once we have seen that the training error curve has flattened out, we are done with training.

IV. Computational Results

A. Predicting Kuramoto-Sivashinsky Trajectory

We discuss the result of training and testing on the KS dataset as described in Section III-B.

The neural network model is composed of an input layer of 1024 nodes, corresponding to the different spatial positions in present time, followed by 3 layers of sigmoids, and a final output layer of 1024 nodes, corresponding to the predicted future state. From the cross-validation result as shown in Table I, we pick the number of nodes in the 3 intermediate layers to be 500 each.

We first verify that the neural network is at least able to reproduce the training set. As seen in Figure 4, it seems that

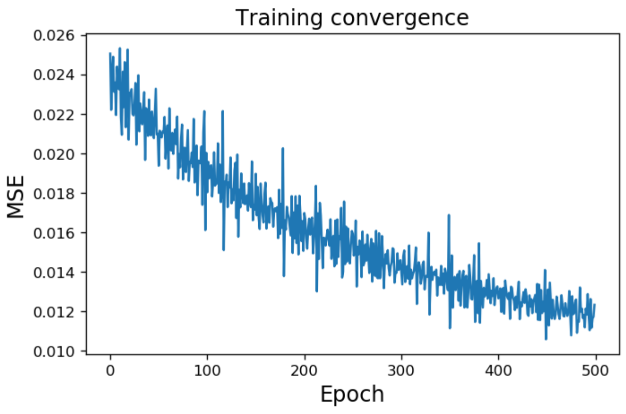
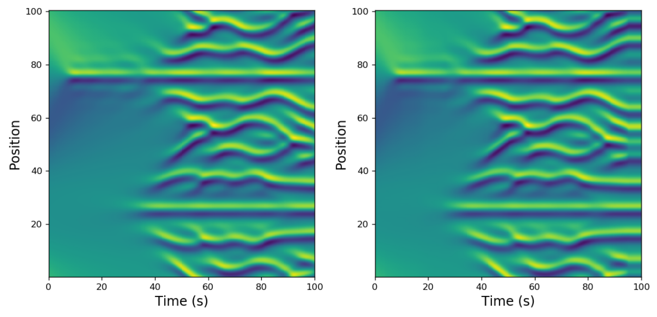


Fig. 3: An example of a desirable neural network training error graph.

TABLE I: 3-fold cross-validation for number of nodes in KS network.

Number of nodes	MSE
10	0.187
100	0.162
500	**0.143**
1000	0.177

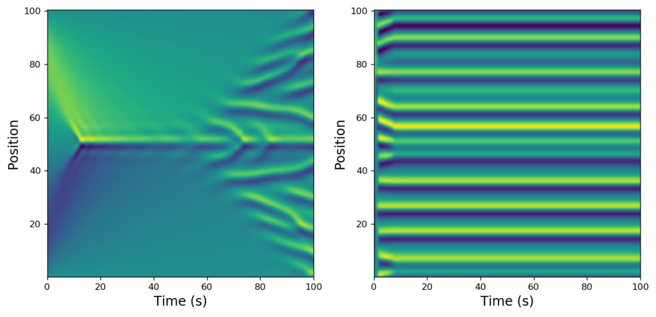
the model was able to predict the rippling areas surprisingly well. However, we also note that the predicted trajectory has its rippling area starting at an earlier timestep than the actual trajectory.



(a) Actual image **(b)** Predicted image

Fig. 4: Prediction on the KS training set.

We now explore the ability of the model on an initial condition that it has not seen before. As seen in Figure 5, the predicted trajectory is unfortunately very far off from the true trajectory. The only redeeming quality of the prediction is that there are roughly two time windows where we see different trajectory patterns, with the latter time window being a time-invariant pattern. This perhaps attests to the difficulty of the problem setup, where given an initial condition that is simply smooth, one has to predict smooth patterns for a couple of timesteps, then slowly transition to the rippling patterns. A future investigation could relax the problem difficulty by focusing only on the rippling areas.



(a) Actual image **(b)** Predicted image

Fig. 5: Prediction on the KS test set.

B. *Predicting Reaction-diffusion Trajectory*

We first discuss the SVD analysis results as described in Section III-C. The average flattened column that was subtracted of each column from the training matrix is visualized in Figure 6. Note that for all the following images, $u(x, y)$ will be on the left and $v(x, y)$ will be on the right.

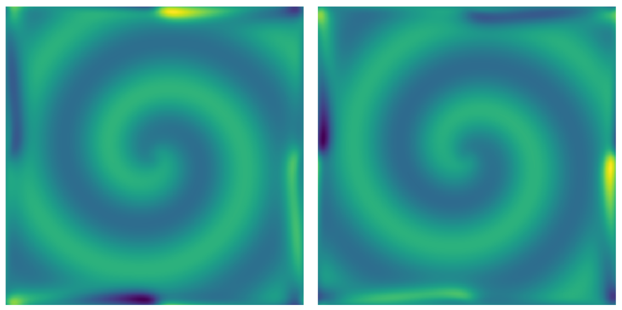


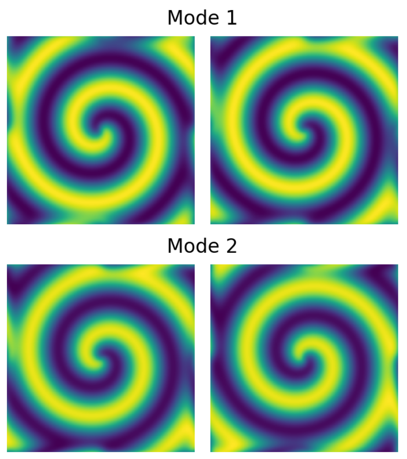
Fig. 6: Average $u(x, y)$ (left) and $v(x, y)$ (right) of the training set.

We then perform SVD on the mean-subtracted matrix and analyze its energy distribution. As seen from Figure 8, the dataset is low-dimensional, with as little as 4 modes capturing 99.9% energy. With this insight, the following analyses will be done on low rank of 4.

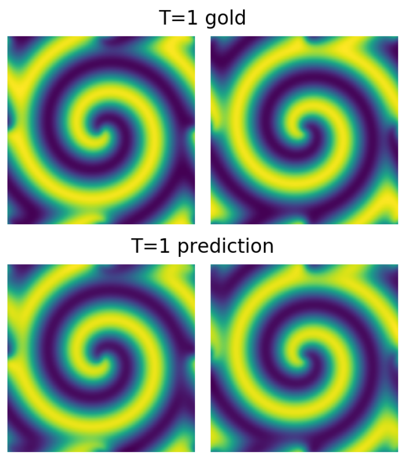
We show the first two spatial modes in Figure 7a, and we observe that in the first two modes, there is a difference in the directions of the innermost tip of the spiral. This makes sense because such a linear combination of such phase-offsetted bases would allow one to construct other spirals of varying phase-offsets.

The neural network model is composed of an input layer of 4 nodes corresponding to the low-dimensional spatial projection, followed by 3 500-node hidden layers with RELU activation function, and a final output layer of 4 nodes, corresponding to the predicted low-dimensional future state. We cross-validate on the L2 penalty for the middle hidden layer. From the cross-validation result as shown in Table II, we pick the L2 penalty of 0.01 for our final trained model.

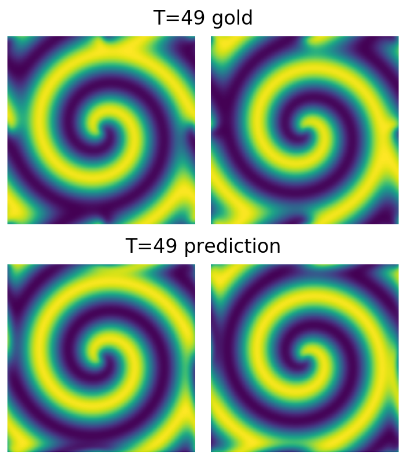
We run the fully trained model on the test set repeatedly to produce a full trajectory in the low dimension, project back up to the original high dimension of u and v, and evaluate the prediction quality for the first iteration after initial condition,



(a) First two spatial modes of the Reaction-diffusion training set.



(b) Reaction-diffusion test prediction for the first iteration.



(c) Reaction-diffusion test prediction for the 49th iteration.

Fig. 7: Modes and predictions for Reaction-diffusion system.

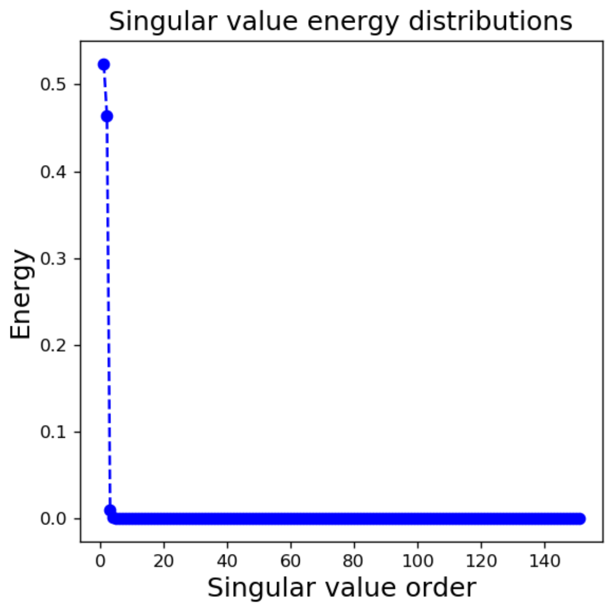


Fig. 8: Energy distribution of the Reaction-diffusion training set.

TABLE II: 3-fold cross-validation for L2 penalty in Reaction-diffusion network.

L2 penalty	MSE
0.0001	253.79
0.001	401.37
0.01	**240.79**

as shown in Figure 7b, and the 49th iteration, as shown in Figure 7c. We see that the model has performed excellently, whereby even though the inner tips of the spirals of T=1 and T=49 are pointing in different directions for both u and v, the model was able to make the right predictions. In particular, note that for $u(x, y, t = 1)$, the inner tip is pointing upward, and for $u(x, y, t = 49)$, the inner tip is pointing downward. Note however that the despite the good match of the spiral tips, the prediction failed to fully capture the artefacts spotted at the corners of the images.

Overall, we seem to have a much greater success in the Reaction-diffusion task than the KS task, perhaps because the initial conditions are much more predictive of the later states, whereby for the Reaction-diffusion task, we are asked to advance spirals, whereas in the KS task, we are asked to transition from a smooth initial condition to rippling states.

C. Predicting Lorenz Trajectory

We discuss the result of training and testing on the Lorenz dataset as described in Section III-D.

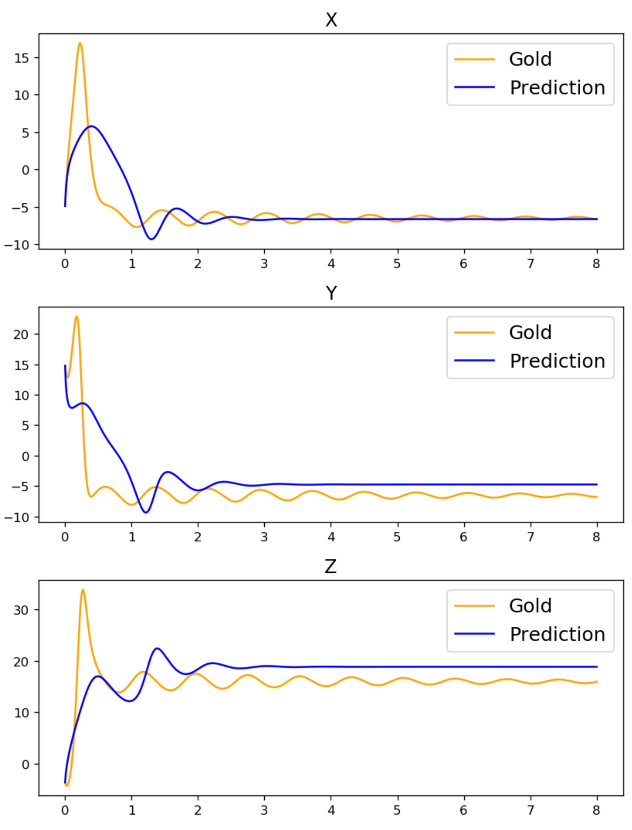
The neural network model is composed of an input layer of 4 nodes, corresponding to $x(t), y(t), z(t), \rho$. Note that we include ρ as an input because we will be asked to perform prediction on two different ρ setups. We then have 4 layers of sigmoids, and a final output layer of 3 nodes, corresponding to $x(t+\Delta t), y(t+\Delta t), z(t+\Delta t)$. Thus, the network model intuitively is tasked with advancing the state trajectory Δt forward in time, given the parameter ρ. From the cross-validation result as shown in Table III, we pick the number of nodes in the 3 intermediate layers to be 10 each.

TABLE III: 3-fold cross-validation for number of nodes in Lorenz network.

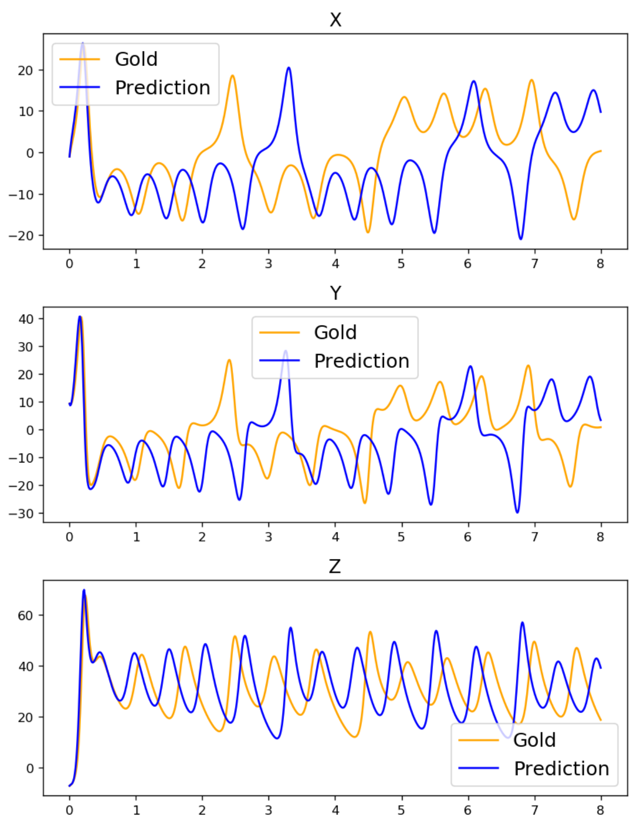
Number of nodes	MSE
5	0.168
10	**0.088**
25	0.158

We run the model on the test set, which uses ρ values of 17 and 35, neither of which was used in the training set. For $\rho = 17$, As seen in Figure 9c, the model is able to capture the the fact that the trajectory settles down to one lobe. Furthermore, Figure 9a showed that the model was able to predict the rough location of lobe center.

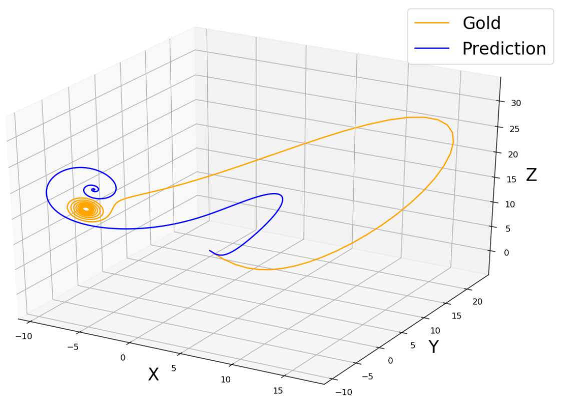
For $\rho = 35$, As seen in Figure 9d, the model is able to capture the the fact that the trajectory alternates between two



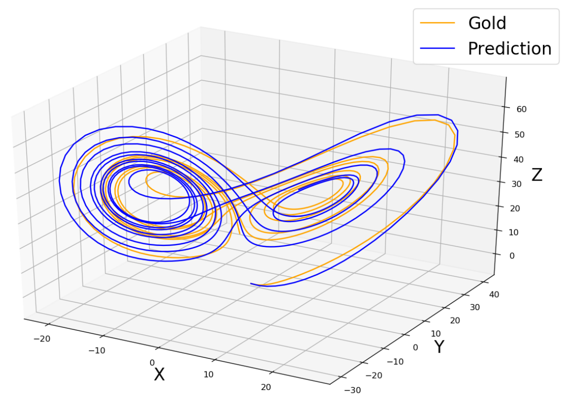
(a) 2D visualization of prediction evaluation for $\rho = 17$.



(b) 2D visualization of prediction evaluation for $\rho = 35$.



(c) 3D visualization of prediction evaluation for $\rho = 17$.



(d) 3D visualization of prediction evaluation for $\rho = 35$.

Fig. 9: Predicted Lorenz trajectories.

lobes. Furthermore, Figure 9b showed that the model was able to capture the right amplitudes of x, y, z. However, we note that the prediction and the gold standard diverge more with increasing time, attesting to the chaotic behavior of the Lorenz system.

Overall, it was very impressive that given just an initial condition, the model was able to perform predictions on ρ values that it has not been trained before, and was able to capture important qualitative information such as whether or not there is only one or two lobes, and also the lobe center locations in 3D space.

D. Predicting Lorenz Lobe Transition

We now focus on the lobe transition prediction task, where given the current position $x(t), y(t), z(t)$, we predict the time interval Δt where we will reach the closest future transition time as described in Section III-D.

The neural network model is composed of an input layer of 3 nodes, corresponding to $x(t), y(t), z(t)$. Note that we no longer include ρ as an input because for both training and testing we will be using $\rho = 28$. We then have 3 10-node layers of sigmoids, and a final output layer of 1 node, corresponding to the predicted timesteps until the next transition.

We run the model on the test set trajectory. As seen in Figure 10, the model is able to predict the transition times pretty well. This is impressive considering that the Lorenz system is chaotic, and the model is presented only with the position at a single time.

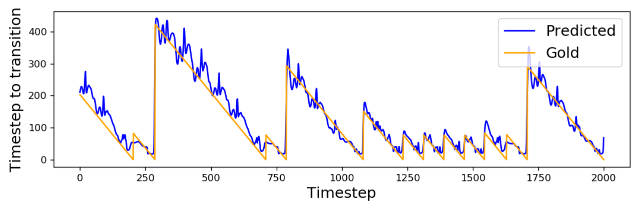


Fig. 10: Test set lobe transition time prediction for $\rho = 28$.

We also investigate how the absolute value of the discrepancy between the predicted transition time and the gold standard varies with the actual transition time. As seen in Figure 11, it does not seem to be the case that the prediction performance gets poorer with higher transition times. Although surprising, this seems to match Figure 10, where we see good predictions even for long transition times.

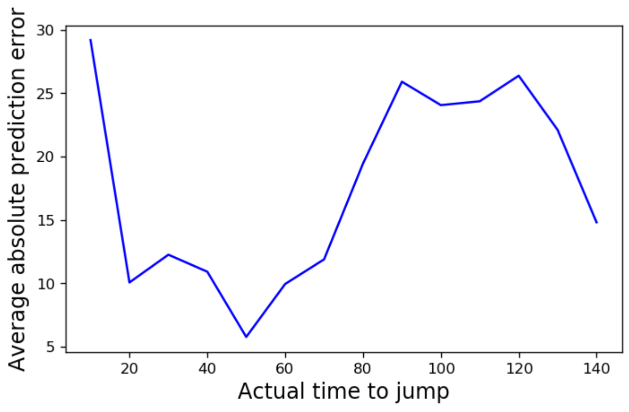


Fig. 11: Transition prediction average error against actual transition time.

V. Summary and Conclusions

We have explored the ability of neural network models to capture core dynamics given trajectory data. Surprisingly, even with the chaotic dynamical systems, given just a single initial condition, these models were able to predict an accurate enough trajectory in all our test cases, with the exception of the KS equation, which has ambiguous initial conditions. These results attest to the strength of neural networks in approximating arbitrarily complex functions.

References

[1] Y. Kuramoto, "Diffusion-induced chaos in reaction systems," *Progress of Theoretical Physics Supplement*, vol. 64, pp. 346–367, 1978.

[2] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[3] S. L. Brunton and J. N. Kutz, *Data-driven science and engineering: Machine learning, dynamical systems, and control.* Cambridge University Press, 2019.

Appendix A: Key Python functions

Example neural network setup

```
def build_model(N, num_space):
  model = tf.keras.Sequential()
  # Input = (num_space,) = U(x, t)
  model.add(layers.Dense(N, activation='sigmoid', input_shape=(num_space,)))
  model.add(layers.Dense(N, activation='sigmoid'))
  model.add(layers.Dense(N, activation='sigmoid'))
  # Output = (numspace,) = U(x, t + dt)
  model.add(layers.Dense(num_space))
  model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.001),
                loss='mse',
                metrics=['mse'])
  return model
```

Creation of Lorenz transition dataset

```
# Input = xt,yt,zt, output = distance to the next transition time

xt = ys[:,0]
yt = ys[:,1]
zt = ys[:,2]
thres_x = 0

# Get times when you cross the threshold.
is_above = xt > thres_x
t_trans = np.where(is_above[:-1] != is_above[1:])[0]

# Label(t) = Given that I am at xt,yt,zt, how many more dt's until
# I hit the next transition point?
labelt = []
# Points to the next transition index
next_trans_i = 0
for t in range(T):
  t_next = t_trans[next_trans_i]
  labelt.append(t_next - t)
  if t_next == t:
    next_trans_i += 1
    if next_trans_i >= len(t_trans):
      # Hit last segment where there is no more transition.
      break

# Truncate ys to exclude the trans-less last segment
ys = ys[:len(labelt),:]
```

Analysis of how Lorenz transition prediction error varies with actual transition time

```
test_errors = np.abs(test_preds.T-test_labels)[0]
# Divide the actual jump labels into discrete buckets
bucket_walls = 10 * np.arange(1,15)
# For each bucket, the sum of the errors
bucket_sums = np.zeros(len(bucket_walls))
# For each bucket, the number of items in the bucket
bucket_counts = np.zeros(len(bucket_walls))

for i in range(len(test_errors)):
  err = test_errors[i]
  actual = test_labels[i]
  # first index of element greater than actual.
  bucket = np.argmax(bucket_walls > actual)
  bucket_sums[bucket] += err
  bucket_counts[bucket] += 1

avg_err_per_bucket = bucket_sums / bucket_counts

fig, ax = plt.subplots()
ax.plot(bucket_walls, avg_err_per_bucket, c="blue")
ax.set_xlabel("Actual time to jump")
ax.set_ylabel("Average absolute prediction error")
save_fig(fig,"lorenz_trans_error_per_trans_time")
```

Convert trajectory data to regression matrix

```
def u_mat_to_regression(u_mat):
  """
  Convert u(x,t) matrix to multi-variate regression inputs and outputs
  Each sample is an (input, output) tuple.
  input is u(x,t), of size |X|, one value for each x for the fixed t.
  output is u(x, t+dt), also of size |X|, one value for each x of the fixed t + dt.
  """
  num_time = u_mat.shape[1]
  num_space = u_mat.shape[0]
  input_mat = np.zeros((num_time - 1, num_space))
  output_mat = np.zeros((num_time - 1, num_space))
  for t in range(num_time - 1):
    input_mat[t, :] = u_mat[:, t]
    output_mat[t, :] = u_mat[:, t+1]
  return input_mat, output_mat

def lorenz_mat_to_regression(ys, rho):
  """
  Given ys from util.load_lorenz_data, where ys[ti, dim] stores x[t], y[t], z[t],
  and rho, one of the Lorenz parameter,
  convert this into a regression matrix for training NN.
  """
  input_mat, output_mat = u_mat_to_regression(ys)
  # We add rho as an additional feature so the NN knows how to advance different rhos
  # differently.
  r = input_mat.shape[0]
  input_mat = np.hstack((input_mat, (rho*np.ones(r)).reshape((r,1))))
  return input_mat, output_mat
```

Create a predicted trajectory from an initial condition

```
def create_pred_u_mat(init_u, model, num_time):
  """
  Create a prediction of u(x,t) given u(x,t=0)
  where t goes from 0 to num_time - 1
  We do this by having model repeatedly predicting u(x, t+dt)
  and we keep feeding the previous step's output as an input for the next step.
  """
  # u(x,t)
  num_space = len(init_u)
  pred_u = np.zeros((num_space, num_time))

  ut = init_u
  pred_u[:, 0] = ut.T
  start_time = time.time()
  for t in range(1, num_time):
    if t % 10 == 0:
      print(f"t = {t}")
    # Use NN to advance the trajectory in time.
    ut = model.predict(ut.reshape((1, num_space)))
    pred_u[:,t] = ut
  print(f"NN prediction of u(x,t) takes {time.time() - start_time} s")
  return pred_u

def create_pred_lorenz(init_y, rho, model, num_time):
  """
  See create_pred_u_mat(), except that this is for Lorenz, where we accept rho
  as another input.
  """
  num_space = len(init_y)
  pred_y = np.zeros((num_space, num_time))

  yt = init_y
  pred_y[:, 0] = yt.T
  start_time = time.time()
  for t in range(1, num_time):
    if t % 10 == 0:
      print(f"t = {t}")
    # Use NN to advance the trajectory in time.
    nn_input = yt.reshape((1, num_space))
    # Need to attach the extra rho feature.
    nn_input = np.append(nn_input, rho)
    nn_input = nn_input.reshape((1, len(nn_input)))
    yt = model.predict(nn_input)
    pred_y[:,t] = yt
  print(f"NN prediction of full lorenz trajectory " +
    f"takes {time.time() - start_time} s")
  return pred_y
```

K-fold cross validation

```
def get_kfold_mse(model, input_mat, output_mat):
  """
  Get the kfold mse for the provided model
  """

  num_fold = 3
  kf = KFold(n_splits=num_fold, random_state = 1, shuffle = True)
  splits = kf.split(input_mat)
  fold = 1
  total_mse = 0
  for train_indices, test_indices in splits:
    fold_train_in = input_mat[train_indices, :]
    fold_test_in = input_mat[test_indices, :]
    fold_train_out = output_mat[train_indices]
    fold_test_out = output_mat[test_indices]

    model.fit(fold_train_in, fold_train_out, epochs=100, batch_size=1000, shuffle=True)
    mse = model.evaluate(fold_test_in, fold_test_out)[0]
    fold += 1
    total_mse += mse
  return 1.0 * total_mse / num_fold
```

Reaction-diffusion flattening and de-flattening

```
def uv_to_wmat(u, v):
  """
  Convert u[x,y,t], v[x,y,t]
  To w[i,t], where each column of w is a flattened u, followed by flattened v.
  """
  # u[x,y,t]
  R = u.shape[0]
  C = u.shape[1]
  T = u.shape[2]
  w_len = 2 * R * C
  # w is a 2D matrix where each column is a flattened u and v.
  w_mat = np.zeros((w_len, T))
  for t in range(T):
    w_mat[:R*C, t] = u[:,:,t].reshape((R*C,))
    w_mat[R*C:, t] = v[:,:,t].reshape((R*C,))
  return w_mat

def mat_to_uv(w_mat):
  """
  The inverse of uv_to_wmat()
  """
  T = w_mat.shape[1]
  R = int(np.sqrt(w_mat.shape[0] / 2))
  C = R
  u = np.zeros((C, R, T))
  v = np.zeros((C, R, T))
  for t in range(T):
    u[:,:,t] = w_mat[:R*C, t].reshape((R, C))
    v[:,:,t] = w_mat[R*C:, t].reshape((R, C))
  return u, v
```

APPENDIX B: SUPPORTING CODES

The rest of the supporting Python and MATLAB codes can be found on https://github.com/sjonany/AMATH563-public/tree/master/hw4/code