Modelling from Measurements Homework

Michael Wood | Spring 2022 | Politecnico di Milano

Background

Code

All codes are hosted online at https://github.woodjmichael/modelling-from-measurements/. References and dependencies include:

- `ks.m`
- `NN_LRZ.m`
- `reaction_diffusion.m`
- MATLAB library `optdmd`
- `EX01a_Linear2D.m` from `sparsedynamics` by Steven Brunton

Approach

There's a lot of libraries on GitHub to implement Dynamic Mode Decomposition (DMD) and SINDy at a high level, but I've chosen to write most of the code myself from scratch, using examples and resources as a guide. My goal is to completely understand the fundamentals, probably sacrificing some model performance. Everything here is in MATLAB.

Singular Value Decomposition (SVD)

$$X = USV^T$$
$$X_{m\times n} = U_{m\times m}S_{m\times n}V^T_{n\times n}$$

SVD is a data reduction technique, representing data X with orthogonal bases U and V, and a diagonal matrix of *singular values*. It is analogous to FFT, which is another powerful tool for mapping problems into a new space where it becomes more simple. SVD can be tailored to a specific problem, such as solving $Ax = b$ where A isn't square, and can be used as a basis for PCA.

To solve problems regarding dynamic systems, typically a 2D data matrix X is formulated where each row is a different spatial position, and each column is the evolution of the measurements over time. If the spatial data is 2D then it is first flattened to a single column vector, such as images.

$$UU^T = U^TU = I$$
$$VV^T = V^TV = I$$

Matrices U and V are unitary, so when multiplied by their transverses, the result is the identity matrix. Sometimes U is called the left singular vectors and V the right singular vectors. U and V also have physical meaning, for instance the rows of each can be thought of as so-called eigen faces or eigen flow fields.

$$U = \begin{bmatrix} | & | & & | \\ u_1 & u_2 & \dots & u_n \\ | & | & & | \end{bmatrix}, S = \begin{bmatrix} s_1 & 0 & & | \\ 0 & s_2 & \dots & 0 \\ | & 0 & & s_m \\ | & | & & 0 \\ | & | & & | \end{bmatrix}, V = \begin{bmatrix} | & | & & | \\ v_1 & v_2 & \dots & v_m \\ | & | & & | \end{bmatrix}, V^T = \begin{bmatrix} v_1^T \dots \\ v_2^T \dots \\ v_m^T \dots \end{bmatrix}$$

Often the data matrix X is taller than it is wide, so $m > n$. We only expect n singular values because U and V are at most rank n. So S is merely a $n \times n$ diagonal matrix, except that for the matrix math to work S needs to be padded with zeros below the square, diagonal singular values. However multiplying these values through does not effect X. Therefore an *economy* version of SVD is often computed, based on the Eckard-Young theorem.

$$Econ: \ \hat{X}_{m\times n} = \hat{U}_{m\times n}\hat{S}_{n\times n}V^T_{n\times n}$$
$$Rank\ r: \ \tilde{X}_{m\times n} = \tilde{U}_{m\times r}\tilde{S}_{r\times r}\tilde{V}^T_{n\times r}$$

A reduced rank r version of the three matrices is often used. Because S-values decrease in magnitude, and U and V vectors coincide appropriately, the matrices can be reduced to the appropriate lower dimensional rank to represent the data. Each successive set of vectors and singular value better approximate the data. This way SVD can be written as:

The program calls for SVD are simple. Note that Python returns a vector s not a diagonal matrix S , and V^T not V.

- MATLAB: `[U,S,V] = svd(X, 'econ')`
- Python: `[U,s,VT] = numpy.linalg.svd(X, 'econ')`

DMD

DMD utilizes SVD to solve for a linear operator A which is a solution to $X' = AX$, where X' is the X matrix shifted forward one time step and with the same dimensions (so some data is lost). Again X is typically formulated such that rows are m points in space and columns are n samples in time, and often for dynamical systems $m > n$.

$$X' = AX$$
$$X = \begin{bmatrix} | & | & | & | \\ x_{t=1} & x_2 & \dots & x_{n-1} \\ | & | & | & | \end{bmatrix}, X' = \begin{bmatrix} | & | & | & | \\ x_{t=2} & x_3 & \dots & x_n \\ | & | & | & | \end{bmatrix}$$

We won't compute A exactly because it can be very large and matrix inversions are costly, but we'll compute a lower dimensional version using SVD and call it $\tilde{A}$. Then $\tilde{A}$ can be understood as a best fit model for the dominant modes in X, and can also be reduced further as needed.

$$X = USV^T$$
$$X' = AUSV^T$$
$$U^{-1}X'VS^{-1} = \tilde{A}$$

On its own $\tilde{A}$ isn't useful, and notably has the wrong dimensions. But the eigen values of $\tilde{A}$ are the same as A, and with those we can conveniently compute the eigenvectors of A, which we call Φ. Physically, Φ represents the spatial modes and the eigenvalues in Λ control the dynamics, such as growth or decay and oscillations.

$$\tilde{A}W = W\Lambda$$
$$\Phi = X'VS^{-1}W$$

With Φ calculated it's possible to begin from initial conditions b_0 and step the spatial modes forward in time. While this wasn't the initial intention of DMD, with some modifications is capable of regression.

$$\hat{X}(k\Delta t) = \Phi \Lambda^t b_0$$

In its most simple form DMD regression is less than fifteen lines of code in MATLAB, or even less using the `optdmd` package:

```
[w,e1,b] = optdmd(X,t,r,imode); # imode controls rank reduction to degree 'r'
Xhat = w*diag(b)*exp(e1*t); # moded, initial conditions, and dynamics
```

SINDy

With SINDy we solve a relatively simple linear algebra problem, but with an emphasis on sparsity in the solution. This is because we don't want to the best linear combination of all features, we want to know which are the few or several most important features.

First a library of candidate functions is built, named Θ . Since the candidate functions are dependent only on the features (data) we already have, their values can be calculated directly. For data with two features x and y with n samples in time, the library up to order 2 looks like the following.

$$\Theta = \begin{bmatrix} 1 & x_{t=1} & y_{t=1} & x_{t=1}^2 & y_{t=1}^2 & x_{t=1}y_{t=1} \\ | & | & | & | & | & | \\ 1 & x_n & y_n & x_n^2 & y_n^2 & x_n y_n \end{bmatrix}_{n \times k}$$

Next the following system of equations is solved for ξ with emphasis on sparsity. This could be achieved using the L0 norm, or by defining a threshold λ below which we won't consider values of ξ.

$$X_{n \times m} = \Theta_{n \times k}\, \xi_{k \times n}$$

The result is that not only can we find the dominant physics but we will know the governing equations because we defined them at the outset. Note that the SINDy libraries want X with time progressing along the vertical rather than horizontal matrix dimension.

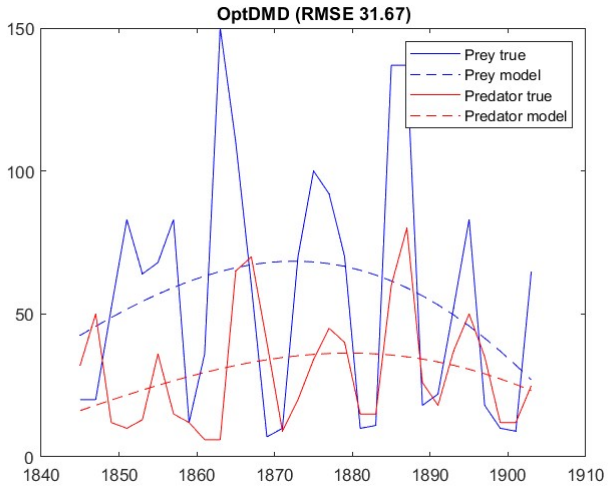
A. Predator-Prey

Lynx and Snowshoe Hare pelt data is available from 1845 to 1903, skipping every even year. The hare notably leads the lynx, although not always and not always by a fixed interval. With only 30 data points per feature, this is decidedly a low data problem, and also a somewhat noisy one. In the image titles I will differentiate between manually computed DMD and using the library OptDMD.

1. DMD

Normal

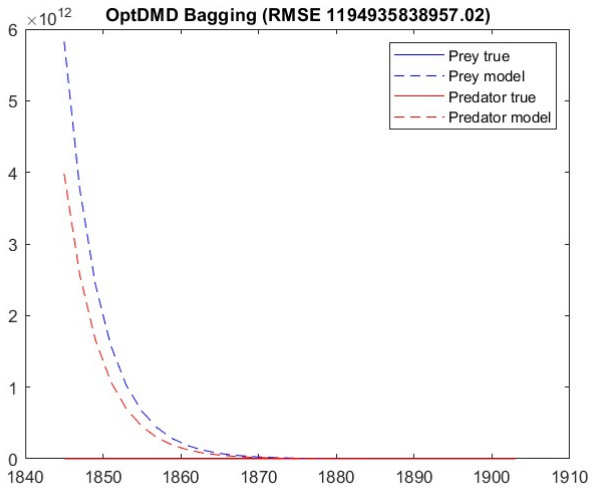
First I built a simple non-delay DMD model using only `svd()` and `\` to solve for b, but the results were so poor that I couldn't verify the model was working as intended. With `optdmd` I could be more sure that the completely unfit arcs were not a mistake, but rather the result of not having the Hankel matrix "shift rows." Without this the A matrix must infer all the dynamic behavior from only X and X' which might be possible with datasets of much higher spatial dimension. However with only two features, or two spatial dimensions, it's clearly not.



Furthermore with only two spatial modes and two eigenvalues, there simply isn't enough complexity in the model to reproduce the patterns in the data, such as a two parameters linear model $y = mx + b$ is unable to accurately model changes in solar irradiation.

With Bagging

Bagging wasn't very successful either, and often failed dramatically such as below. The best bag size (dimensionality of the boosted subset) was always equal to or just smaller than the full dataset dimension n. This just means that the model needed the full timeseries to even model the moving average (roughly what looks to be the case in the image above). It could be that simply averaging Φ, Λ, and b_0 across the ensemble isn't appropriate.

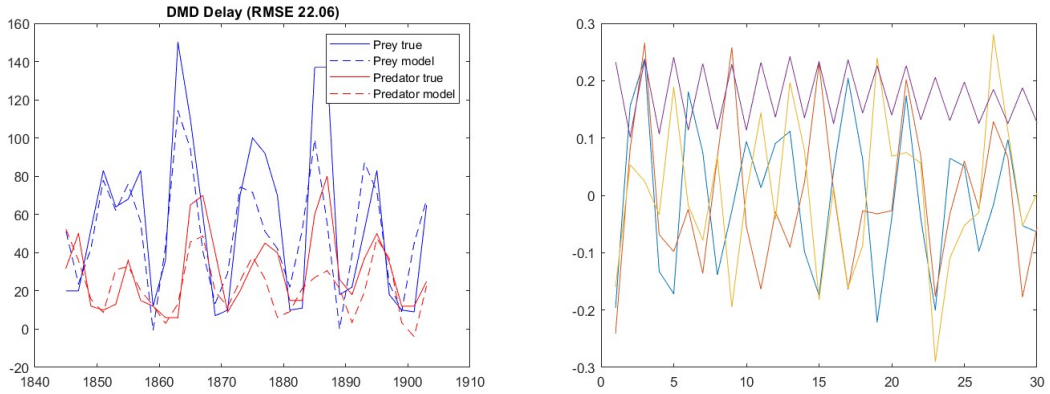


2. Time-delay DMD

Normal

With the Hankel matrix juxtaposing the time dynamics on consecutive rows (every other row actually) the results are much better, with both the dynamics and magnitudes reasonably well represented. I performed a quick grid search to find the best shape of the Hankel matrix, which is 15 shifted *pairs* of rows - one for each feature. In all these models I keep the rank as large as possible to avoid losing any information.

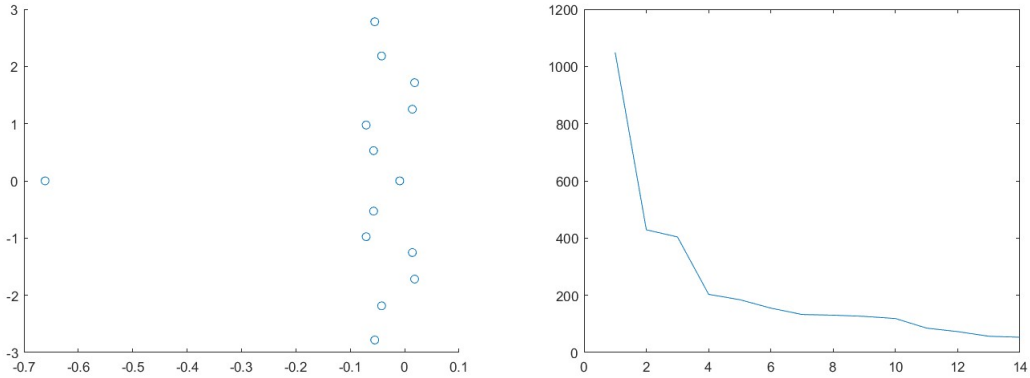
DMD Delay (left) and Spatial Modes (right)



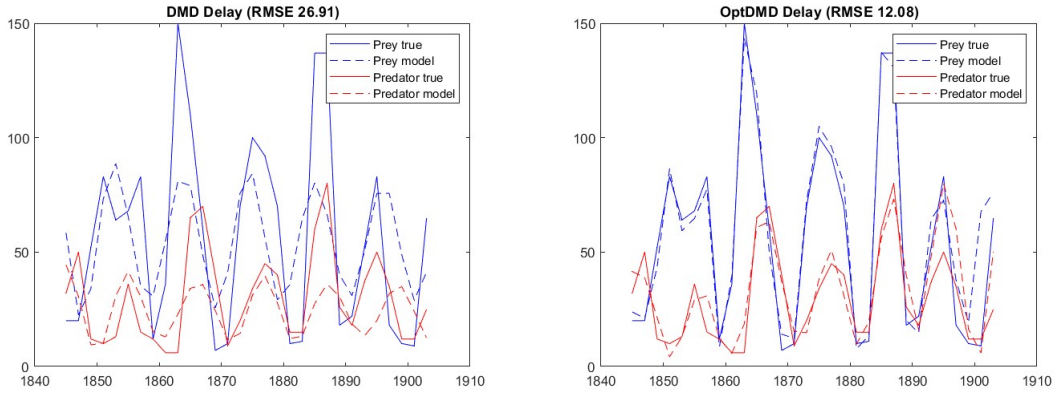
The first four columns of Φ can be described as the spatial modes of the system. These appear to have some physical similarity with the data, such as a very regular 2-year periodicity (purple) as well as 6-year (orange).

The eigenvalues shown on the standard complex plane show one dominant scaling value and lots of rotation of the eigenvectors.

Eigenvalues of A (left) and Singular Values (right)



Whereas the singular values, below, indicate a rank 4 or 6 lower dimensional model might capture most of the modes. At rank 6 the model indeed fits the dynamics but the magnitudes are off, and the the RMSE increases by about 22%.



So far the model is my own non-optimal formulation, which is how I can see the singular values and not just the final eigenvectors and values. Using the OptDMD package we get a much better lower Root Mean Squared Error (RMSE) and visually better fit model, only diverging somewhat near the end for both features. This is a common problem that I don't quite understand, but feels like a boundary condition problem, however when I step the solution out past 1903 by ten years, there is only a moderate change in the divergent behavior around 1900.

With Bagging

Sadly the results from bagging are not great, and very unpredictable. For one, a grid search is very difficult over:

- Size of the boosted subset (bag size), p
- Number of Hankel rows (pairs of rows actually), h
- Rank (although this I try to keep as large as possible), r

This is because many combinations of p-h-r are not possible due to the matrix math, and when its not possible the codes crashes. A MATLAB `try-except` function would be useful here.

On the non-OptDMD code I wrote, I found a very small improvement in RMSE from bagging: RMSE decreases from about 22 to about 21.6. This was with a bag size p equal to 20, or two thirds of the temporal dimension of 30. This agrees with some pseudo academic reading I found on bagging, where a bag size of 60% of the training set length is common.

Again the model doesn't finish well in the 1890 to 1903 range, worse than the non-bagging DMD delay model possibly. Neither model captures the full height of the second prey peak around 1880-1890.

Presumably there isn't only one correct way to bag training data, and this agrees with the reading I found. My first approach was to *replacement sample* (so that the same value can be selected multiple times) columns of the Hankel matrix, and then to *not* sort the values into increasing chronological order. Others I know *do* sort the subset. I actually found that each worked better on different models and different data, however sorting seems to have an advantage that some of the spatial mode is preserves, if augmented by the random sampling.

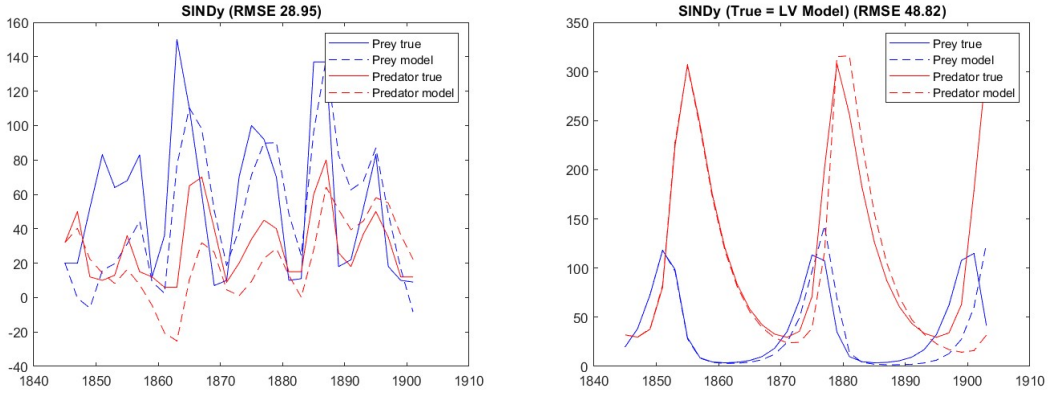
3. Lotka-Volterra

The Lotka-Volterra equations can be approximately fit to the data in MATLAB using `lsqcurvefit()` with a RMSE of 28.7 and the parameters $b = 0.880, p = 0.007, r = 0.12$ and $d = 0.419$. Based on the DMD model performance I expected a better fit, although possibly `fmincon()` would do a better job. The solution roughly captures the dynamics but the magnitudes are usually off by a moderate amount.

It should be noted however that this is a very low dimensional space, with only four parameters and the need to solve a system of differential equations. For instance a four parameter ARMA model likely could not produce these results, but possibly ARIMA with a few more parameters.

4. SINDy

With code built up from Prof Brunton examples in the `sparsedynamics` GitHub repository, I wasn't able to get better RMSE than the Lotka-Voltera equations or DMD. Bagging and Hankel delay did help a certain amount. I dont really understand why a basic SINDy implementation wouldn't find the coefficients for Lotka-Volterra or even do better, so there may be a coding error on my part. The left plot is with the Θ library set to maximum order 2, and λ threshold at 0.005.



However I used the same code on data generated from the Lotka-Volterra model and the results are excellent. The ξ coefficients match to within a few percent and visually the plots are quite accurate (ignore the vertical scale on the right-hand plot, which also makes the RMSE look larger than it really is). I thought maybe the data was too noisy and cleaned it up with a moving average, but this only helped marginally.

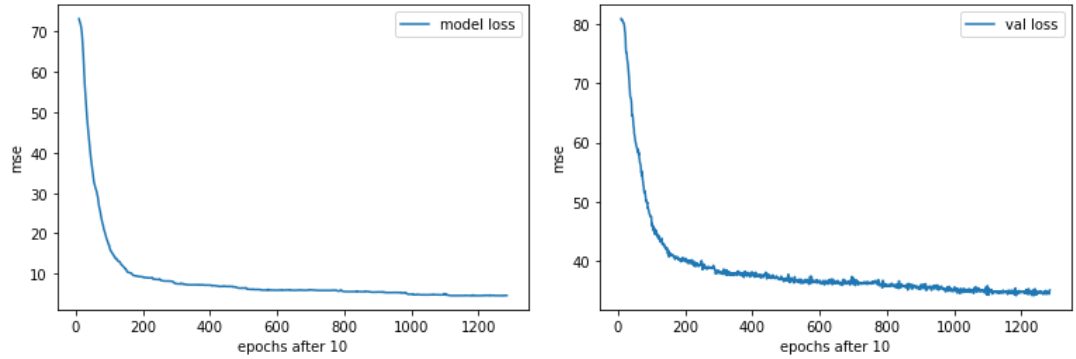
B. Reaction-diffusion and KS Equation

1. Kuramoto-Sivashinsky (KS) Forecast

I built and trained a four layer Neural Network (NN) on the first 80% of the KS image data, and then forecasted the remaining 20% of time samples in a single-step-forward manner. The NN uses the RELU activation function, Adam optimizer, and Mean Squared Error (MSE) as the loss metric. The input and output layers have 2048 neurons to match the spatial dimensionality of the KS data, and the hidden layers have 100 units each.

Training stops automatically after 1285 epochs because the validation loss has not decreased in the last 100 epochs ("patience"). The left image below shows MSE decreases to 10 within about 200 epochs, but doesn't decrease much further. On the right however is validation loss (last 20% of data), which never decreases below an MSE of 40. This means that while the model is learning the training data it is not able to generalize well to new data.

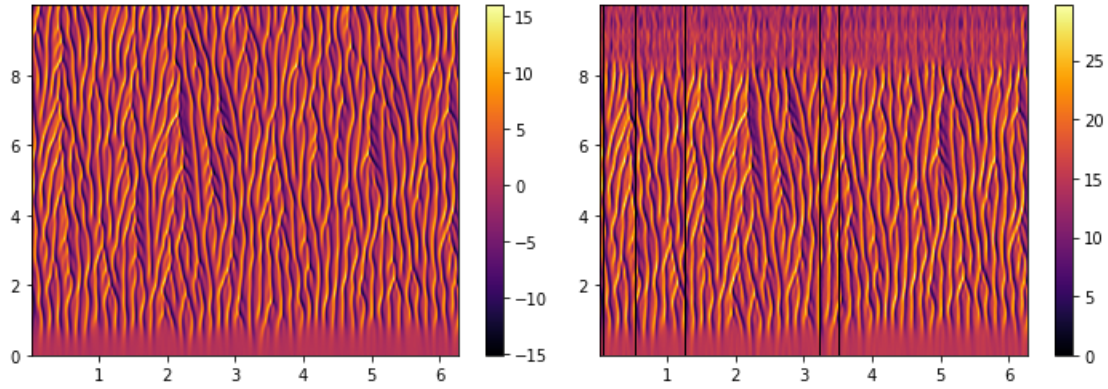
NN Train Data MSE (left) and Test Data MSE (right)



2. Compare ODE45 to NN

The Ordinary Differential Equation (ODE) time stepper solution (left below) visually seems to reproduce the dynamics and modes well, although we don't have a baseline for comparison here. By comparison the NN forecast (on the last 20% of the data) just propagates forward a less clear version of the spatial modes, with almost no temporal dynamics. This is a visual confirmation of the training result which is that the validation loss is 4x the training loss.

ODE Time Stepping Solution (left) and NN Training (right)

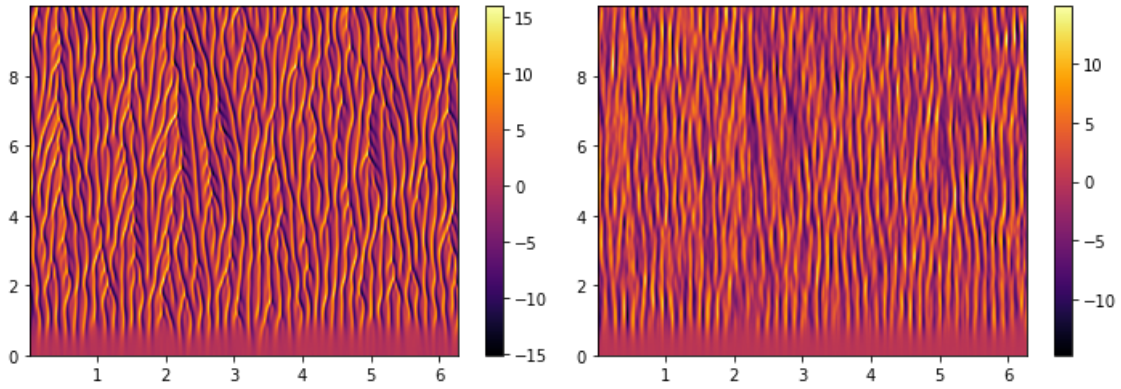


3. SVD Forecast

KS

The SVD forecast interestingly does a much better job of capturing both the modes and dynamics, below see the SVD model for reduced rank of 100 and 10 show visually acceptable results. The MSE for SVD rank=100 and rank=10 is 1.2 and 10.9, respectively.

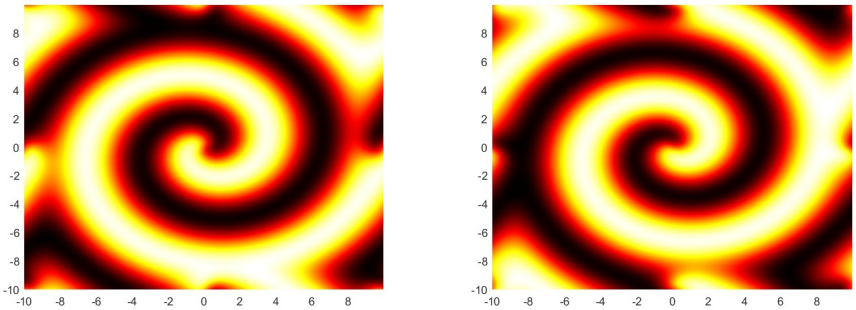
SVD Forecast for r=100 (left) and r=10 (right)



Reaction-Diffusion

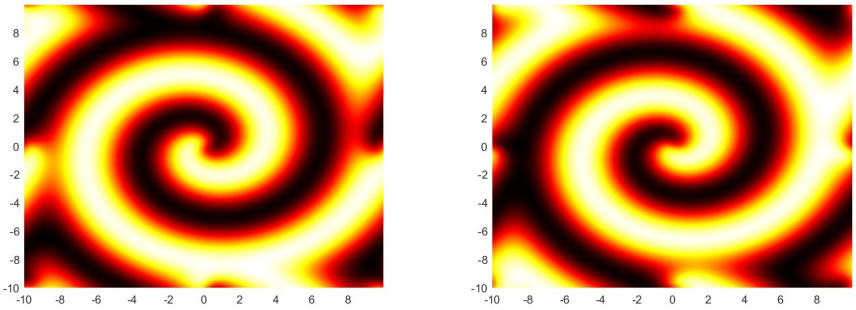
A reaction-diffusion example is run from a provided MATLAB script and the results are shown below for the last image, both variables are displayed below at the final time instance of the simulation.

Reaction-Diffusion Final Time Instance for u (left) and v (right)



SVD then transforms u and v data into a rank=10 subspace, where the spatial modes of the U matrix and singular values of S can be used in a regression for future timesteps of the simulation. SVD appears to be excellent at extracting features which correlate very well with the true data.

Reaction-Diffusion Final Time Instance for Rank=10 Estimated u (left) and v (right)



C. Lorenz Equations

A NN is trained on the solutions to the Lorenz equations, solved by ODE45 for the parameters $\sigma = 10$ and $b = 8/3$ and with $\rho = 10, 28$ and 35. Since the NN can receive any shape input and produce any shape output, the data is supervised learning problem is structured so that one time sample of the input matrix is $[x_t\ y_t\ z_t\ \rho_k]$. The first three elements represent the three dimensional position vector and the last element is constant for $t = 0, 0.01, \ldots 8$. Then when t reaches 8, a new ρ is given as input. See below for an example.

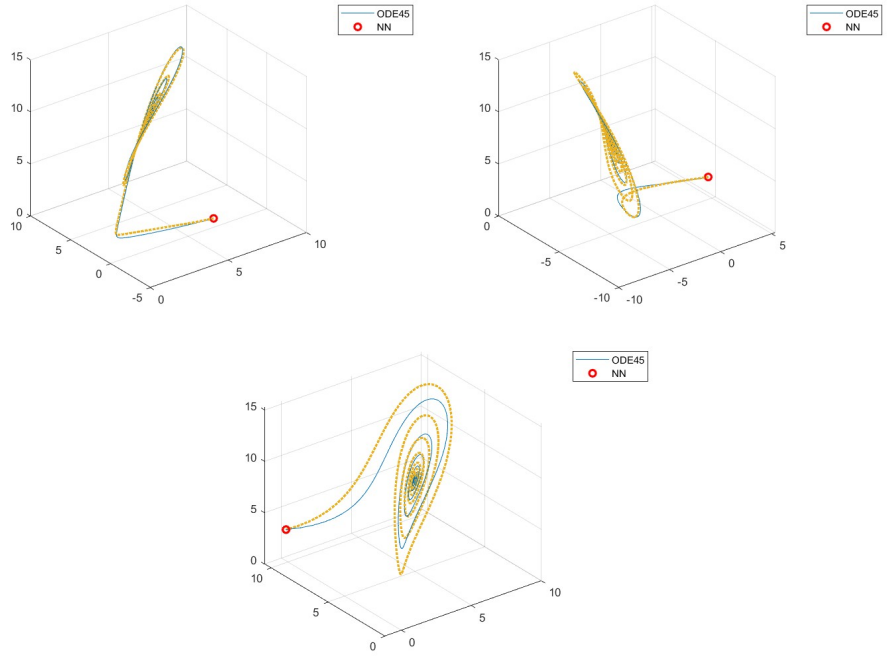
The output however does not contain the ρ information, since it's only needed to inform the model whether the initial conditions (and subsequent predictions) $[x_0, y_0, z_0]$ should follow a $\rho = 10$ pattern or some other.

The NN has three hidden layers, each layer has ten units per layer, training the Adam optimizer, and trains for at least 100 epochs but stops when the MSE loss has not reduced in the last 10 epochs.

Training Set $\rho = 10, 28, 35$

Below are three example trajectories for the Lorenz systems used in training the NN. For $\rho = 10$ and $\rho = 28$ the NN seems to predict the ODE45 results fairly well, and less so for $\rho = 35$. Overall training MSE reaches a minimum at 0.00031, which is one or two orders or magnitude larger than if a NN net is trained just on a single value of ρ. This is evidence that already the NN has difficulty learning the different Lorenz systems.

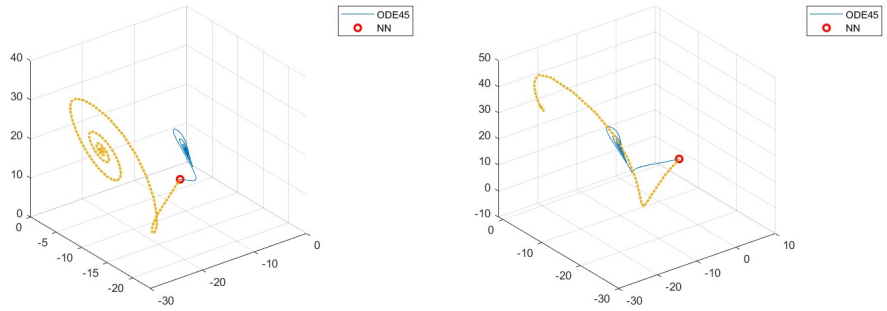
ODE45 vs NN Predictions for r=10 (left), r=28 (middle), r=35 (right)



Test Set $\rho = 17, R40$

Then the NN is given a new initial condition $[x_0, y_0, z_0, \rho_k]$ for $\rho = 17$ and 40. In the sample trajectories below the NN poorly predicts the ODE45 results for both ρ values. Repeated over 100 trajectories with random initial conditions the MSE is 458 and 252 for $\rho = 17$ and 40 respectively. This indicates that the NN completely fails at predicting the ODE45 solution compared to the training systems.

ODE vs NN Predictions for r=17 (left) r=40 (right)



The Lorenz equations are notable for causing dramatically different outputs for small changes in the parameters. Therefore a reasonable conclusion is that the NN is able to learn the $\rho = 17$ system because it is similar enough to the training systems of $\rho = 10, 28, 35$ but the $\rho = 40$ system is different enough and the NN is unable to generalize.