Probabilistic Modeling for Plasma Profile Prediction in Fusion Reactors

Aaron Wu
Adviser: Egemen Kolemen

Abstract

A crucial component of making fusion energy commercially viable is the ability to successfully predict plasma profile evolution in fusion reactors. This project introduces and demonstrates a proof-of-concept probabilistic model aimed at accomplishing this task. Following previous work in this area, the main modeling assumption is that there exists some latent space where the dynamics are linear. The model uses a neural-network based normalizing flow to represent an invertible map between the observed and latent space, and the classical linear Gaussian state space model represents state evolution in the latent space. Unlike the previous deterministic models, instead of relying on the input data as the ground truth for inference, the model uses what it observes to update its own belief about system evolution. Furthermore, the probabilistic approach taken allows for evaluation of uncertainty in generated forecasts, which is important for assessing robustness of these predictions. The model's performance is evaluated on both linear and nonlinear example systems.

1. Introduction

Fusion power is an exciting and promising new source of clean and renewable energy. Out of the current fusion reactors that exist worldwide, the tokamak design, a type of torus, is perhaps the most ubiquitous. An example of a device with such as design is the DIII-D National Fusion Facility in San Diego. A key step in realizing the ultimate goal of commercially viable fusion energy is successfully predicting plasma behavior within tokamak fusion reactors. In broad terms, this problem is what I aim to address in this project. The rest of this introduction will provide a brief survey of previous work on prediction of plasma evolution and end with an overview of the project.

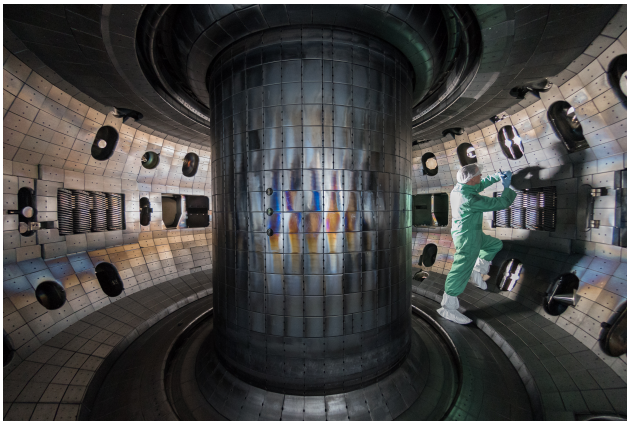

Figure 1: The inside of the DIII-D fusion reactor.

1.1. Survey of Work on Prediction

Numerous integrated modeling frameworks aimed at describing properties of plasma evolution in tokamaks have been developed from physical principles. Given device parameters and sequences of actuator inputs, these allow for simulations of plasma, eliminating the need for costly experiments [13]. Direct implementations in code are often computationally expensive and complex, consisting of a collection of coupled submodels. For real-time use, various submodels within these frameworks have been replaced by computationally more efficient approximations, including, but not limited to, neural networks [8].

Standalone machine learning models that are not part of any modeling framework have proven to be helpful in the related field of plasma event prediction. In particular, these models predict whether an event, such as disruptions or instabilities in plasma, will occur given some input data. "Classical" approaches such as SVMs, discriminant analysis, classification and regression trees, and random forests have been previously used [1]. Following recent trends, various types of neural networks trained on data from NSTX [2], DIII-D [4], ASDEX Upgrade [3], JET [9], and JT-60U [15] have also demonstrated state of the art accuracy in this area.

Taking inspiration from the application of neural networks to event prediction, Abbate et al. [1] propose a novel approach for predicting plasma evolution. Instead of creating a neural network that is a submodel of a larger integrated modeling framework, Abbate et al. elect to directly model future plasma profiles using a single neural network. The authors posit two

key benefits of this approach. First, a single neural network is more simple and generally faster (sub 100ms) than a collection of submodels, which makes it suitable for real-time work. Second, directly modeling evolution eliminates the effect of compounding inaccuracy when submodels are replaced by reductive approximations or simplifying assumptions. Using this approach, the authors created a neural network based on convolutional layers and LSTM cells that predicts future plasma state over a 200ms horizon on the DIII-D tokamak, given current state and future control inputs.

Another recent model, which adopts the same approach of directly modeling future plasma profiles, uses a dense autoencoder to map to and from a latent space, where it is assumed that dynamics are linear. A linear recurrent kernel is learned alongside the dense neural networks to obtain a complete approximation of profile evolution on a short time horizon. We will hereafter refer to this model as the autoencoder model for brevity. Data is of the same form as the LSTM based network described previously. In addition to sharing the same advantages as the LSTM model, the architecture of this model makes it especially amenable to fast, real-time control. The linear representation of state evolution allows most of the control policy computation to be solved efficiently in latent space, with the only expensive operations being the transformation into and out of the latent space [5] [10] [14].

1.2. Overview of Project

In this project, we follow the approach of the autoencoder model in assuming the existence of a space wherein dynamics evolve linearly. However, the modeling assumptions diverge considerably concerning how input data should be interpreted. With the autoencoder (and LSTM model), it is assumed that the data fed into these models contains perfect information and is taken to be the ground truth. That is, during training, the loss function of the autoencoder is formulated to drive differences between predicted and observed data to exactly 0. During inference, the autoencoder model transforms the observed state into latent space and deterministically evolves the data forward through time. As a result, the presence of

noise and anomalies in the observed data is neglected. This is sub-optimal as sensor noise can be non-negligible and physical models for raw data are not complete. Another problem is the lack of uncertainty estimation. Since the model is fully deterministic, the user has no idea of how confident the model is in its prediction. Further work has shown that an auxiliary feed-forward neural network can successfully forecast prediction errors [5], but this is not intrinsic to the model and still does not capture the uncertainty in predictions.

To address these issues, this project's model instead assumes that input data is noisy and that the precise state of the system at any given time is not exactly known. In particular, the model holds a belief about the system state, and given successive observations of data, updates that belief accordingly. It should be noted that while these assumptions make our model probabilistic, it is not Bayesian in a strict sense. The reason for this will become clear in section 2, where we give a description of the model, its training, and inference procedure.

The main focus of this project was prototyping this model. As such, synthetic data sets were used and only autonomous systems were considered. In section 3, we introduce the non-linear and linear systems from which the synthetic data was generated. The results of model evaluation on these examples is presented in section 4. We then conclude in section 5 by discussing possible improvements as well as the steps required to extend this model for use on real data from a tokamak reactor, such as DIII-D.

2. Model

2.1. Problem Formulation

We can describe observed evolution as a generic system

$$\dot{\mathbf{y}} = g(\mathbf{y}, \mathbf{u}) \tag{1}$$

where $\mathbf{y}$ and $\mathbf{u}$ are the current state and control inputs, respectively. Associated with each system is a set of time-invariant values $\alpha_1, ..., \alpha_j$ that the user believes are relevant to the

dynamics of g. In the context of plasma profile prediction, these could be scalar parameters that describe the reactor configuration/geometry (e.g. neutral beam geometry). Our model aims to approximate a discretization of g. The structure of the model generally resembles that of Bezenac et al. with differences in problem formulation and how model parameters are chosen (section 2.3) [6].

2.2. Linear Gaussian State Space Model

As in the autoencoder model, we assume the existence of a space where the dynamics are linear. Evolution in this space is as follows

$$\begin{aligned} \mathbf{z}_0 &\sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Gamma}) \\ \mathbf{z}_{t+1} &= A\mathbf{z}_t + B\mathbf{u}_t + \varepsilon_t \end{aligned} \tag{2}$$

where $A \in \mathbb{R}^{n\times n}$ is the transition matrix, $B \in \mathbb{R}^{n\times m}$ is the input matrix, and $\varepsilon_t \sim \mathcal{N}(0, \boldsymbol{\Sigma})$ is mean-zero Gaussian process noise with covariance $\boldsymbol{\Sigma}$. The sequence of latent states is then mapped to a space of observations, which we take to be $\mathbb{R}^d$. Explicitly,

$$\mathbf{x}_t = E\mathbf{z}_t + \sigma_t \tag{3}$$

where $E \in \mathbb{R}^{d\times n}$ is the emission matrix and $\sigma_t \sim \mathcal{N}(0, \boldsymbol{\Phi})$ is the observation noise. We further assume that the initial state $\mathbf{z}_0$ and the noise at each step $\{\varepsilon_1, \varepsilon_2, ..., \sigma_1, \sigma_2, ...\}$ are mutually independent. Equations (2) and (3) together specify the Gaussian State Space representation that forms the core of our model.

2.3. Normalizing Flow

We capture non-linearity in our model by using normalizing flows. These "flows", in the most general sense, are simply differentiable bijections whose inverses are also differentiable i.e. diffeomorphisms. This invertibility condition is crucial, as it is a precondition for the change

of variables formula for densities. In particular, normalizing flows can be thought of as a method to transform densities in a non-linear manner [11].

Of particular interest are flows based on neural networks. These flows allow the learning of a suitable bijection directly from data, without the need for user input. These have proven empirically useful in problem domains involving high-dimensional non-linear data, such as computer vision. Furthermore, most, if not all of neural network normalizing flows have been designed so that the determinant of their Jacobian matrix can be computed efficiently. As in [6], we opt to use the RealNVP normalizing flow introduced in Dinh et al [7]. Given d dimensional input $x_{1:d}$ and $i \in \{1, ..., d\}$, the output of RealNVP, call it $y_{1:d}$, is given by

$$\begin{aligned} y_{1:i} &= x_{1:i} \\ y_{i+1:d} &= x_{i+1:d} \odot \exp(s(x_{1:d})) + t(x_{1:d}) \end{aligned} \tag{4}$$

where $\odot$ denotes the Hadamard product and s, t are functions from $\mathbb{R}^d$ to $\mathbb{R}^{d-i}$. Here, s and t are feedforward neural networks. The inverse of this transformation is then

$$\begin{aligned} x_{1:i} &= y_{1:i} \\ x_{i+1:d} &= (y_{i+1:d} - t(y_{1:d})) \odot \exp(-s(y_{1:d})) \end{aligned} \tag{5}$$

and the Jacobian is a $d \times d$ matrix of the form

$$\begin{bmatrix} \mathrm{Id}_i & 0 \\ \frac{\partial y_{i+1:d}}{\partial x_{1:i}^T} & \mathrm{diag}(\exp(s(x_{1:i}))) \end{bmatrix} \tag{6}$$

Since this is a lower triangular matrix, the determinant is simply the product of the diagonal entries, making its computation linear in d.

$$\det \mathrm{Jac}(x_{1:d}) = \prod_{i=1}^{d} \exp(s(x_{1:d})) = \exp\left(\sum_{j=1}^{d} x_{1:d}\right) \tag{7}$$

2.4. Parameterization

Each of the two parts of the model: the linear Gaussian state space model and the normalizing flow both require several parameters. In particular, we must define $A, B, E, \boldsymbol{\mu}, \boldsymbol{\Gamma}, \boldsymbol{\Sigma}, \boldsymbol{\Phi}$, and $\boldsymbol{\Lambda}$ to fully specify our model.

The normalizing flow parameters $\boldsymbol{\Lambda}$ are learned directly when minimizing loss. On the other hand, the parameters for the state space model $A, B, E, \boldsymbol{\mu}, \boldsymbol{\Gamma}, \boldsymbol{\Sigma}, \boldsymbol{\Phi}$ are viewed as functions of the time-invariant values $\alpha_1, ..., \alpha_j$. The idea is that we train the model on a data set consisting of multiple time series, each of which is generated from the same family of non-linear systems. Then, the hope is that there exists a normalizing flow that can linearize dynamics for this family and that the main factors which influence difference in evolution are $\alpha_1, ..., \alpha_j$.

In practice, we model this dependency of the state space model parameters on $\alpha_1, ..., \alpha_j$ by means of a neural network. First, the time-invariant parameters are passed into a single dense layer. The outputs of this layer are then fed to individual layers, each of which correspond to one of the model parameters. For the covariance matrices $\boldsymbol{\Gamma}, \boldsymbol{\Sigma}$, and $\boldsymbol{\Phi}$, we symmetrize and add a small value to the diagonal to ensure the result output is positive definite. Mathematically, we have

$$
\begin{aligned}
&\boldsymbol{\alpha} = [\alpha_1, \alpha_2, ..., \alpha_j] \\
&A = \psi_1(\phi(\boldsymbol{\alpha})); B = \psi_2(\phi(\boldsymbol{\alpha})); \boldsymbol{\mu} = \psi_3(\phi(\boldsymbol{\alpha})) \\
&\boldsymbol{\Gamma} = \psi_4(\phi(\boldsymbol{\alpha}))^T \psi_4(\phi(\boldsymbol{\alpha})) + \epsilon \cdot \mathrm{Id} \\
&\boldsymbol{\Sigma} = \psi_5(\phi(\boldsymbol{\alpha}))^T \psi_5(\phi(\boldsymbol{\alpha})) + \epsilon \cdot \mathrm{Id} \\
&\boldsymbol{\Phi} = \psi_6(\phi(\boldsymbol{\alpha}))^T \psi_6(\phi(\boldsymbol{\alpha})) + \epsilon \cdot \mathrm{Id}
\end{aligned}
\tag{8}
$$

where ϕ and $\psi_1, ..., \psi_6$ denote the dense feedforward networks. See figure 2 for an illustration of the entire model.

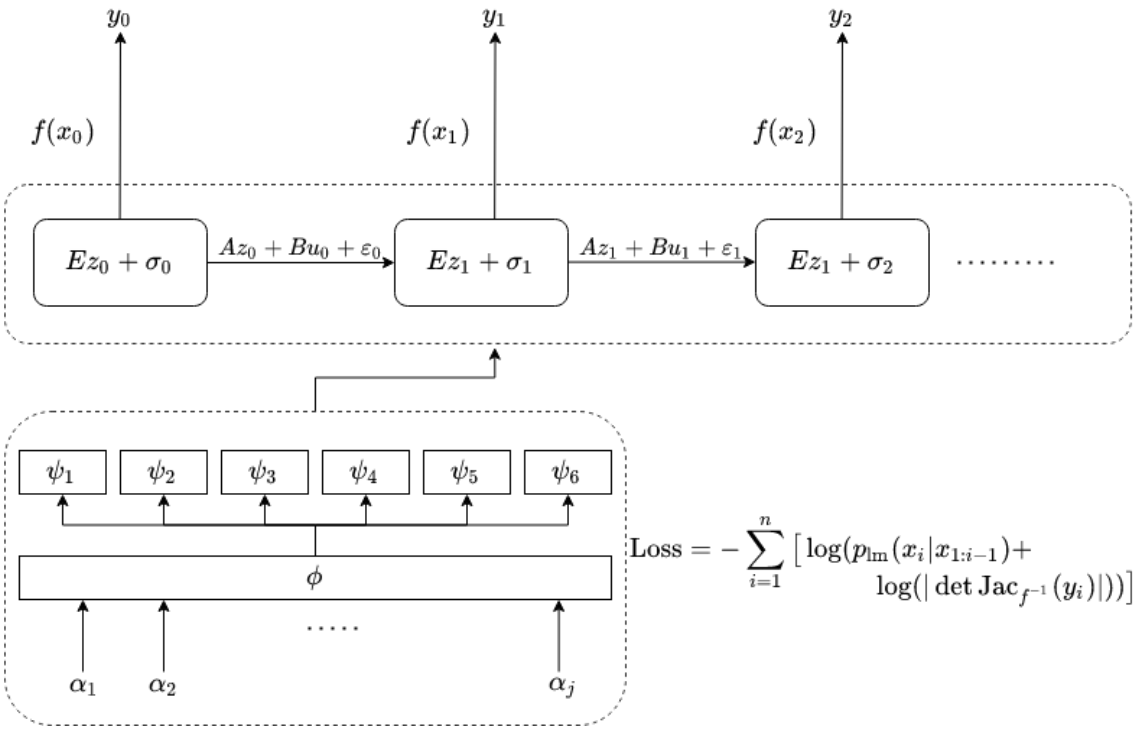

Figure 2: Diagram of model architecture. Output of the parameterization block (enclosed by the lower dashed rectangle) are the parameters in the state space model (enclosed by the upper dashed rectangle). Not shown are the hidden layers in the normalizing flow f.

2.5. Model Training and Inference

To train our model, we treat the output of the model as a distribution parameterized by $A, B, E, \boldsymbol{\mu}, \boldsymbol{\Gamma}, \boldsymbol{\Sigma}, \boldsymbol{\Phi}$, and $\boldsymbol{\Lambda}$ and maximize the likelihood of the training data. Recalling the discussion from section 1, the model itself is Bayesian in nature, due to the state space model. However, the training methodology is not, as we do not put priors on any parameters of the model. Due to the structure of the model, we can compute the likelihood in closed form (derivation here mostly follows [6]). Denote the parameters $A, B, E, \boldsymbol{\mu}, \boldsymbol{\Gamma}, \boldsymbol{\Sigma}, \boldsymbol{\Phi}$ collectively as Θ and p_{lm} as the density function of the state space model. Let $\ell(y_{1:n}; \Theta, \Lambda)$ be the likelihood

of the sequence of observations $y_1, ..., y_n$ under parameters Θ and Λ. Then,

$$\begin{aligned}
\ell(y_{1:n}; \Theta, \Lambda) &= p(y_{1:n}; \Theta, \Lambda) \\
&= \prod_{i=1}^{n} p(y_i | y_{1:i-1}; \Theta, \Lambda) \\
&= \prod_{i=1}^{n} p(x_i | y_{1:i-1}; \Theta) |\det \text{Jac}_{f^{-1}}(y_i)| \\
&= \prod_{i=1}^{n} |\det \text{Jac}_{f^{-1}}(y_i)| \int p(x_i | z_i, y_{1:i-1}) p(z_i | y_{1:i-1}) dz_i \\
&= \prod_{i=1}^{n} |\det \text{Jac}_{f^{-1}}(y_i)| \int p(x_i | z_i, x_{1:i-1}) p(z_i | x_{1:i-1}) dz_i \\
&= \prod_{i=1}^{n} |\det \text{Jac}_{f^{-1}}(y_i)| p_{\text{lm}}(x | x_{1:i-1})
\end{aligned} \tag{9}$$

The fourth equality uses the law of total probability and the chain rule. The fifth equality comes from the fact that the filtered distribution is the same as the state space model. Proposition 1 in [6] shows this fact for autonomous systems in latent space, and the proof can be easily adapted to account for the control term. Intuitively, transitions in the model happen exclusively in the latent space, and the normalizing flow "preserves" all structure in observation space. Therefore, conditioning on the observations before or after the flow provides the same amount of information regarding the latent state of the system. We fix the probability distribution of the linear model p_{lm} to be Gaussian, and an exact analytical representation can be found through Kalman filtering (algorithm 2). At each training step, the above likelihood is calculated in a forward pass and weights of neural network are updated through backpropagation.

Generating forecasts over some time horizon T, is accomplished through the following procedure.

where Kalman refers to the Kalman filtering subroutine. In the subroutine description we use P to denote the state estimate covariance, S to denote the emission covariance, and K to denote the optimal gain. Quantifying the probability of a generated forecast can be

Algorithm 1: Forecast routine

Input: Historical observations: $\{y_1, ..., y_n\}$
Input: time-invariant parameters: $\boldsymbol{\alpha}$
Input: Forecast horizon: T
$\Theta = \{\psi_1(\phi(\boldsymbol{\alpha})), ..., \psi_6(\phi(\boldsymbol{\alpha}))\}$
for $i = 1$ **to** n **do**
  $x_i = f^{-1}(y_i)$
end
$\mu_{\text{filt}}, \Sigma_{filt} = \text{Kalman}(\{x_1, ..., x_n\})$ `// get filtered mean, covariance for last obs.`
$\hat{z_n} \leftarrow \mathcal{N}(\mu_{filt}, \Sigma_{filt})$`// sample; see [12] for proof that this is Gaussian`
for $i = n$ **to** T **do**
  $\varepsilon_i \leftarrow \mathcal{N}(0, \Theta[\boldsymbol{\Sigma}])$
  $\sigma_i \leftarrow \mathcal{N}(0, \Theta[\boldsymbol{\Phi}])$
  $\hat{z}_{i+1} = A\hat{z}_i + Bu_i + \varepsilon_i$
  $\hat{x}_{i+1} = E\hat{x} + \sigma_i$
  $\hat{y}_{i+1} = f(x_{i+1})$
end
Output : $\{y_{n+1}, ..., y_T\}$

Algorithm 2: Kalman Filter

Input: Sequence: $\{x_1, ..., x_n\}$
Input: Parameters: Θ
for $k = 1$ **to** n **do**
  Predict:
  $\hat{z}_{k|k-1} = A\hat{z}_{k-1|k-1} + Bu_k$
  $P_{k|k-1} = AP_{k-1|k-1}A^T + \boldsymbol{\Sigma}$
  Update:
  $S_k = EP_{k|k-1}E^T + \boldsymbol{\Phi}$
  $K_k = P_{k|k-1}ES_k^{-1}$
  $\hat{z}_{k|k} = \hat{z}_{k|k-1} + K_k(y_k - E\hat{z}_{k|k-1})$
  $P_{k|k} = (I - K_kE_k)P_{k|k-1}$
end
Output : $E\hat{z}_{n|n-1}$, S_n

accomplished using the forecasting distribution. This takes on the same form as the likelihood in equation (5).

$$p(y_{n+1:n+T}|y_{1:n}; \Theta) = \prod_{i=n+1}^{n+T} p_{\text{lm}}(x_i|x_{1:i-1})|\det \text{Jac}_{f^{-1}}(y_i)| \tag{10}$$

3. Implementation Details

3.1. Code

A preliminary implementation of this model was accomplished using TensorFlow and TensorFlow Probability with Keras API.[1] Custom layers were created for both the state space model as well as the normalizing flow. Due to time constraints, only a model with autonomous dynamics in the latent state was implemented. That is, the $B\mathbf{u}_t$ term is dropped from equation (2).

We found in practice that setting the epsilon term in equation (4) to 1×10^{-6} led to consistently well-defined covariance matrices during training. We chose the $\phi(\,\cdot\,)$ parameterization layer to be 50 units wide. The normalizing flow consisted of two RealNVP blocks, each with two 10 unit hidden coupling layers. To ensure that the entire input was transformed, the one block masked the first half of the input while the other block masked the second half.

In practice, it was found that maximizing the likelihood alone was not sufficient for the model to produce reasonable results. To ameliorate this, we introduced an additional rollout loss term. At each training step, the mean trajectory of the state space model was computed and passed through the normalizing flow. The rollout loss was then the squared Euclidean distance between the resulting series and the training data, averaged over the time dimension. The intuition behind such an approach is that the model is encouraged to learn dynamics that match the training data as close as possible, with the noise terms explaining any residual discrepancies.

An issue that arose from enforcing this rollout loss was numerical instability of the model. Gradients would sometimes explode during training, leading to either infinite loss or not-a-number results. For this, loss of the model over each time step was constrained to the range $[-10, 10]$ and learning rate was set to a value of 0.0001. See table 1 for a summary of hyperparameter choices. We remark that the number of epochs is large, as each time series

[1] The complete model can be found at: https://github.com/PlasmaControl/normalizing-flow-kalman-filter

Parameter	Value
Layer Activation (all layers)	ELU
Optimizer	Adam
Learning Rate	0.0001
Number of Epochs	1500
Num. RealNVP blocks	2
Num. RealNVP coupling layers	2
Coupling layer width	10
Parmaeterization layer width	50

Table 1: Hyperparameters used in model. The choice of latent dimension size varied from dataset to dataset.

requires many iterations for MLE to converge. On a laptop, each epoch takes only about 100ms to complete.

3.2. Data

Data was synthetically generated from common linear and non-linear systems. Each dataset consisted of 4 series, 300 time-steps long, from the same system but with different parameter choices. The training data consisted of the first 100 time steps of each series, and the remaining 100 were reserved for testing model inference.

1. Basic oscillator: Rotation in $\mathbb{R}^2$ by an angle of θ. For this dataset, latent dimension size was set to 2.

$$\begin{gathered}\mathbf{z}_0 = \begin{bmatrix}1\\0\end{bmatrix}; A = \begin{bmatrix}\cos(\theta) & -\sin(\theta)\\ \sin(\theta) & \cos(\theta)\end{bmatrix}; E = \begin{bmatrix}0.27 & 0.9\\ 0.1 & 1\end{bmatrix}\\ \mathbf{\Sigma} = \begin{bmatrix}0.008 & 0\\ 0 & 0.01\end{bmatrix};\ \mathbf{\Phi} = \begin{bmatrix}0.1 & 0\\ 0 & 0.02\end{bmatrix};\ \theta \in \{\pi/10, \pi/9, \pi/15, \pi/7\}\end{gathered} \tag{11}$$

2. Duffing map: Discrete time analouge of the Duffing equation. Note that for this series we did not specify a latent space representation, as it is clearly non-linear. Instead, we generate the data in observation space and model tries to learn a linearization. For this

dataset latent dimension was set to 5.

$$\begin{bmatrix} x_0 \\ y_0 \end{bmatrix} = \begin{bmatrix} 0.8 \\ 0.3 \end{bmatrix}; \quad \begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} y_n \\ -bx_n + ay_n - y_n^3 \end{bmatrix}; \tag{12}$$
$$(a, b) \in \{(1, 1), (1.1, 0.9), (1.4, 0.8), (0.8, 1.2)\}$$

4. Results

For all data sets, we provide the time-averaged Euclidean squared error between the mean predicted trajectory (rollout) and the true trajectory. To understand how well the forecasting distribution captures the data, we generate 1000 sample trajectories and visualize them on a heat map in phase space and as a time series. Ideally, what we should see is a higher concentration of points, represented on a heat map by a lighter color, surrounding the true trajectory.

4.1. Basic Oscillator

Table 2 provides a summary of the predicted trajectory distance from the true trajectory. We present the heat maps of the series corresponding to $\theta = \pi/10$ in figures 3,4, and 5.

Value of θ	Mean Distance
$\pi/10$	0.665
$\pi/9$	0.754
$\pi/15$	0.753
$\pi/7$	2.93

Table 2: Time-averaged squared Euclidean distance for each series in Basic Oscillator dataset

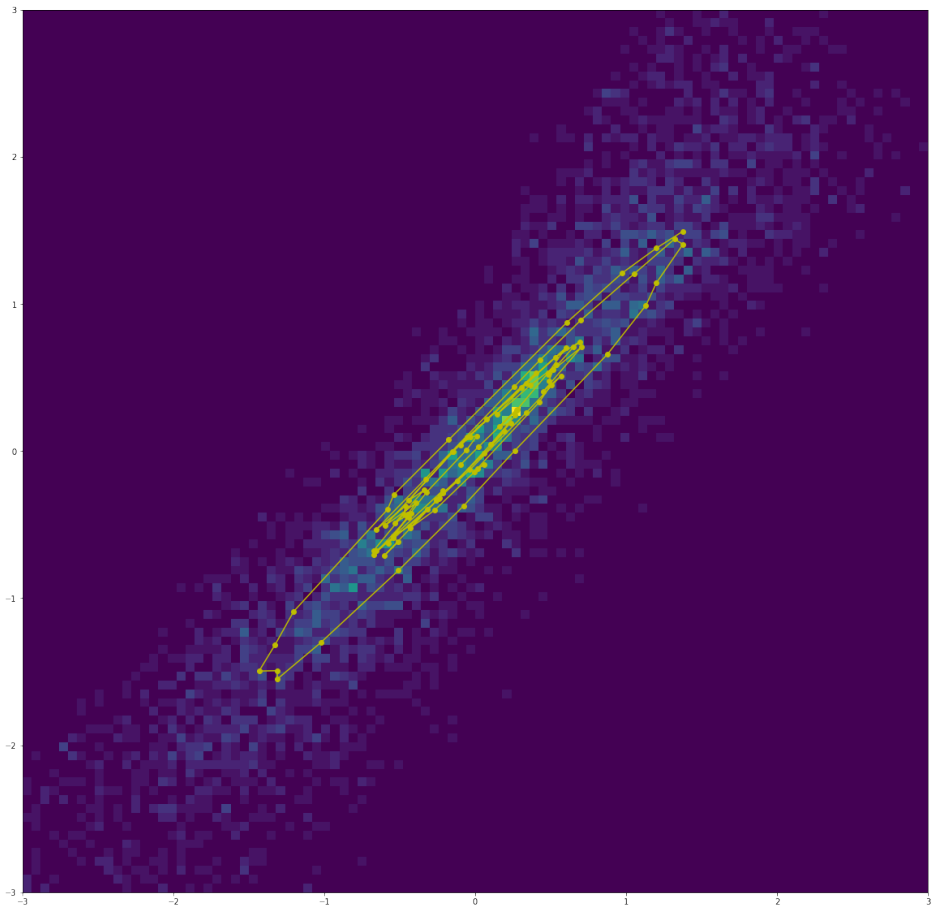

Figure 3: Heat map of phase space $\mathbb{R}^2$. Brighter colors indicate a higher density of points. True trajectory is plotted as the yellow dotted line.

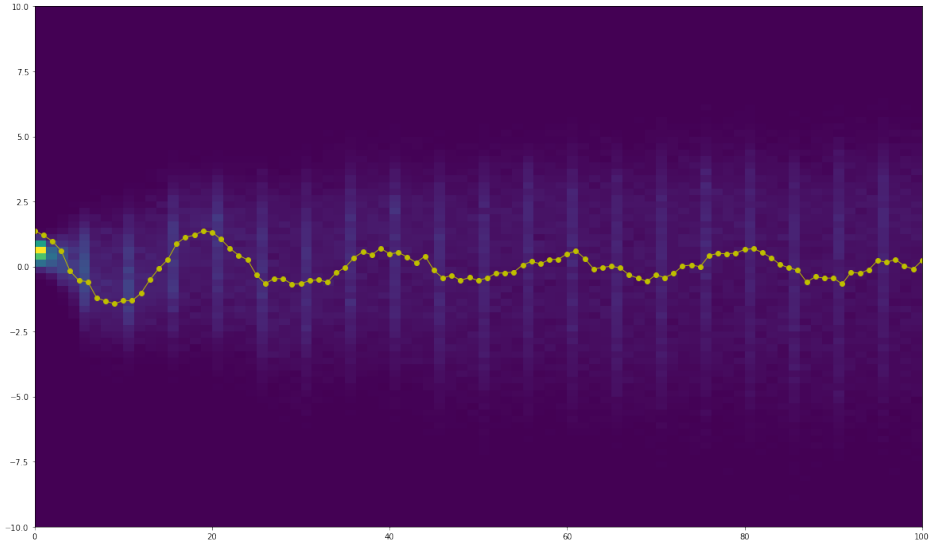

Figure 4: Heat map of x versus time. True trajectory plotted as the yellow dotted line

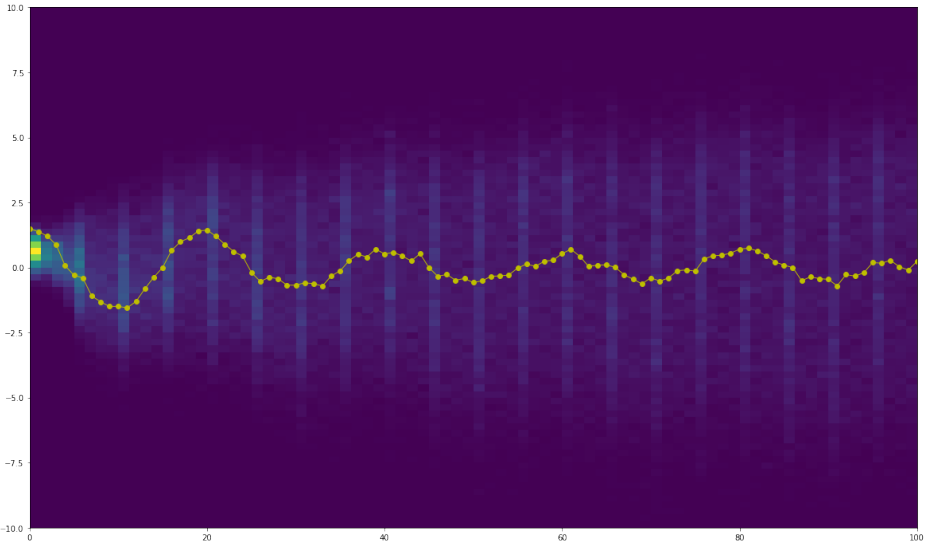

Figure 5: Heat map of y versus time. True trajectory plotted as the yellow dotted line

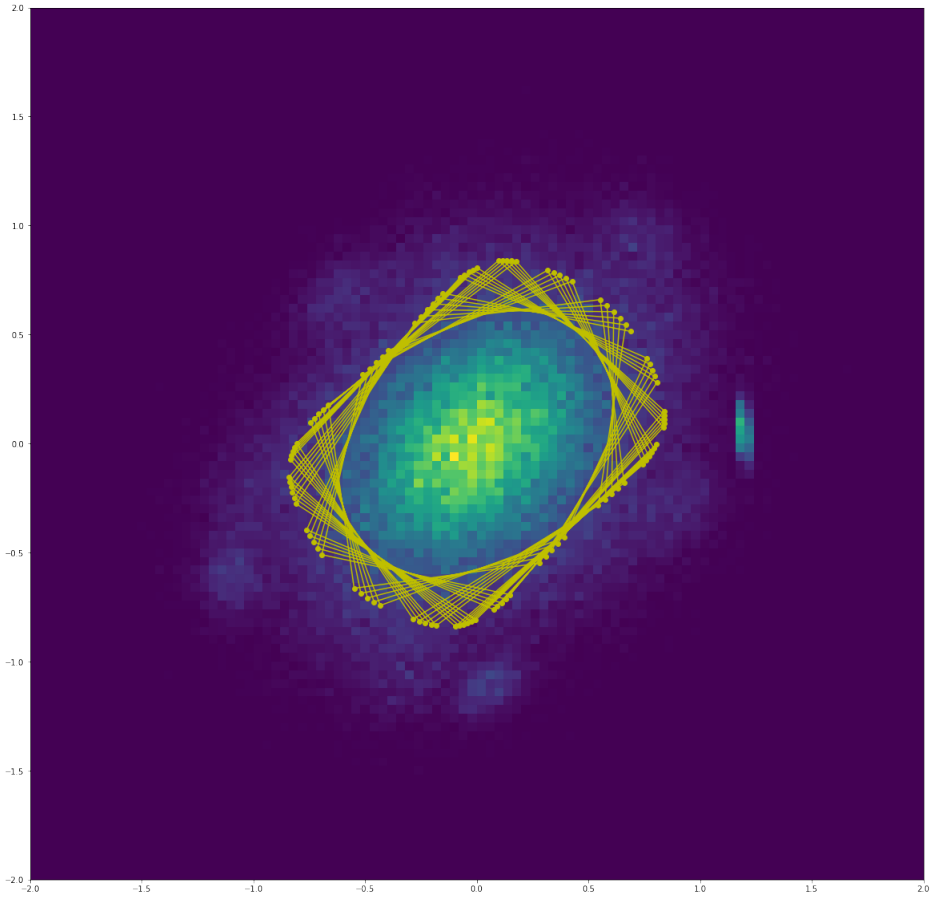

Figure 6: Heat map of the Duffing map phase space $\mathbb{R}^2$. True trajectory shown as a yellow dotted line.

4.2. Duffing map

Table 3 provides the values of the mean predicted trajectory distance from the true trajectory. We choose to present the heat maps for the series with $a = 1$, $b = 1$, as this system has the most interesting behavior. The phase space heat map is presented in figure 6, while the heat maps of the x and y coordinate over time are in figure 7, 8 respectively.

Value of a, b	Mean Distance
a=1, b=1	0.506
a=1.1, b=0.9	0.001
a=1.4, b=0.8	0.0002
a=0.8, b=1.2	0.003

Table 3: Time-averaged squared Euclidean distance for each series in Duffing map dataset

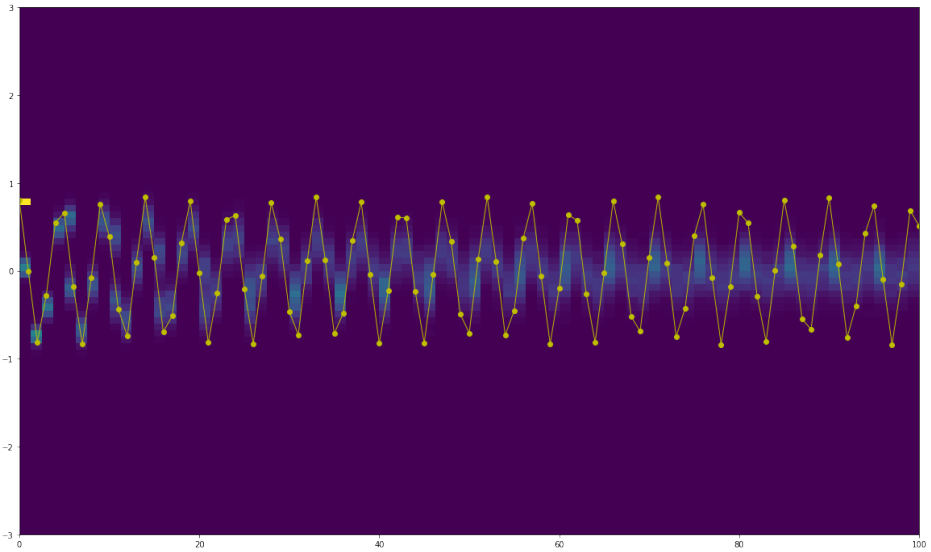

Figure 7: Heat map of x coordinate versus time for the Duffing map.

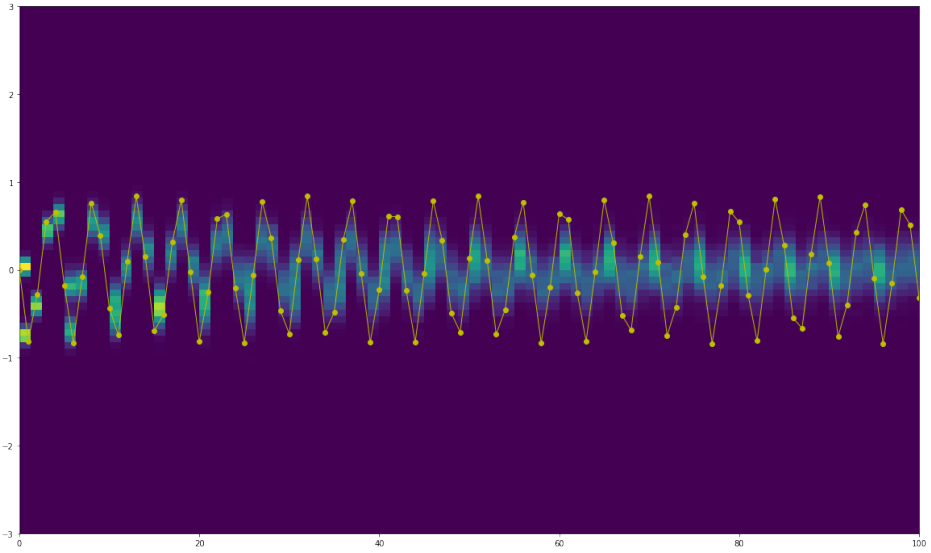

Figure 8: Heat map of y coordinate versus time for the Duffing map.

5. Discussion

Both the mean distance values and the figures indicate that the model struggles to capture the dynamics of the systems it is trained on, regardless of non-linearity. In the case of the Duffing map, the mean values for $(a, b) \in \{(1.1, 0.9), (1.4, 0.8), (0.8, 1.2)\}$ are very by by virtue of the fact that the system converges around $(0, 0)$ as $n \to \infty$. Looking at the results for the case of $(1, 1)$ gives a much better idea of the poor performance. In figure 6, the high density of points in the center suggests that the model is merely memorizing the pattern it saw in the other three samples. In particular, it seems to drive both the x and y coordinates to zero over the forecast horizon.

The results for the linear case are no better than for the Duffing map. Looking at figures 4 and 5, we see that the spread of generated trajectories is extremely wide. This is not good, as when trajectories are sampled at time of inference, we would like the model to both 1.) output a prediction that is close to the ground truth 2.) be highly confident in this prediction. Furthermore, the forecasting distribution does not center around the true data. As evidence, the error values in table 2 are relatively large, considering the fact that the norms of the observed vectors are around 1.

One possible explanation for this behavior is the fact that we ask the model to find dynamics that are globally optimal for describing what it observes. This is opposed to the approach of the autoencoder model, which only finds an approximation that holds over no more than $\sim$20 steps into the future.

Another possible explanation is that the addition of the rollout loss term could actually be hurting the model's ability to generalize. Especially because the epoch count is so high for the sake of MLE convergence, the model could be over fitting to the observed trajectory and not actually learning any meaningful dynamics. However, the introduction of the rollout term was meant to prevent the model from learning nonsense dynamics in the first place. The question is then how to ensure that the model learns a reasonable distribution through

MLE without over fitting.

With regards to the model behavior on the linear system, model complexity could also be an influencing factor. Running the model without normalizing flow on the linear data set yielded improvements.

5.1. Future directions of work

The first priority in the future will be to explore further modifications of the model to improve performance. A straightforward first step might be to reduce the penalty associated with the rollout term. Instead of requiring that the model bring the error to 0, we can frame it as a soft constraint. Another immediate first step would be to more thoroughly explore hyperparameter choices. As mentioned in the previous section, there is evidence that model complexity hurt performance. Depending on the complexity of the system generating the data, reducing layer width in the RealNVP blocks could reduce error.

After these steps, it would be a good idea to get a more quantitative idea of model behavior before pursuing any further modifications to the structure of the model. The results above do not compute the actual forecasting distribution mentioned in section 2. Getting this distribution and plotting it could yield more insight into what is going wrong.

If everything goes well and issues are fixed, the final step would be to get this model working on real data from the DIII-D reactor. Data processing pipelines from the existing autoencoder and LSTM model could be easily adapted to work in this model's framework. An interesting line of inquiry would be to compare the performance between models, and if the results are similar, examine the uncertainty estimates given by this model.

6. Acknowledgements

The author would like to thank Rory Conlin and Prof. Egemen Kolemen for their support and guidance throughout this project.

I pledge my honor that this paper represents my own work in accordance with university regulations. /s Aaron Wu

References

[1] J. Abbate, R. Conlin, and E. Kolemen, "Data-driven profile prediction for DIII-d," *Nuclear Fusion*, vol. 61, no. 4, p. 046027, mar 2021. Available: https://doi.org/10.1088/1741-4326/abe08d

[2] M. Boyer, S. Kaye, and K. Erickson, "Real-time capable modeling of neutral beam injection on NSTX-u using neural networks," *Nuclear Fusion*, vol. 59, no. 5, p. 056008, mar 2019. Available: https://doi.org/10.1088%2F1741-4326%2Fab0762

[3] B. Cannas *et al.*, "Disruption prediction with adaptive neural networks for asdex upgrade," *Fusion Engineering and Design*, vol. 86, no. 6, pp. 1039 – 1044, 2011, proceedings of the 26th Symposium of Fusion Technology (SOFT-26). Available: http://www.sciencedirect.com/science/article/pii/S0920379611000810

[4] R. M. Churchill, B. Tobias, and Y. Zhu, "Deep convolutional neural networks for multi-scale time-series classification and application to tokamak disruption prediction using raw, high temporal resolution diagnostic data," *Physics of Plasmas*, vol. 27, no. 6, p. 062510, 2020. Available: https://doi.org/10.1063/1.5144458

[5] R. Conlin *et al.*, "Practical techniques for machine learning control of fusion plasmas," *Bulletin of the American Physical Society*, vol. 66, 2021.

[6] E. de Bézenac *et al.*, "Normalizing kalman filters for multivariate time series analysis," in *Advances in Neural Information Processing Systems*, H. Larochelle *et al.*, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 2995–3007. Available: https://proceedings.neurips.cc/paper/2020/file/1f47cef5e38c952f94c5d61726027439-Paper.pdf

[7] L. Dinh, J. Sohl-Dickstein, and S. Bengio, "Density estimation using real NVP," *CoRR*, vol. abs/1605.08803, 2016. Available: http://arxiv.org/abs/1605.08803

[8] F. Felici *et al.*, "Real-time-capable prediction of temperature and density profiles in a tokamak using RAPTOR and a first-principle-based transport model," *Nuclear Fusion*, vol. 58, no. 9, p. 096006, jul 2018. Available: https://doi.org/10.1088/1741-4326/aac8f0

[9] D. R. Ferreira, P. J. Carvalho, and H. Fernandes, "Deep learning for plasma tomography and disruption prediction from bolometer data," *IEEE Transactions on Plasma Science*, vol. 48, no. 1, pp. 36–45, 2020.

[10] A. Iqtidar, "Predicting and controlling plasma profiles of tokamaks through an autoencoder system," 2020. Available: http://arks.princeton.edu/ark:/88435/dsp012227ms770

[11] I. Kobyzev, S. J. Prince, and M. A. Brubaker, "Normalizing flows: An introduction and review of current methods," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 11, pp. 3964–3979, nov 2021. Available: https://doi.org/10.1109%2Ftpami.2020.2992934

[12] K. P. Murphy, *Probabilistic Machine Learning: Advanced Topics*. MIT Press, 2023. Available: probml.ai

[13] J. P. H. E. Ongena *et al.*, "Numerical transport codes," *Fusion Science and Technology*, vol. 61, no. 2T, pp. 180–189, 2012. Available: https://doi.org/10.13182/FST12-A13505

[14] S. Otto and C. Rowley, "Linearly recurrent autoencoder networks for learning dynamics," *SIAM Journal on Applied Dynamical Systems*, vol. 18, no. 1, pp. 558–593, 2019.

[15] R. Yoshino, "Neural-net disruption predictor in JT-60u," *Nuclear Fusion*, vol. 43, no. 12, pp. 1771–1786, dec 2003. Available: https://doi.org/10.1088/0029-5515/43/12/021

Image Attributions

- Image in figure 1 courtesy of Wikipedia:

  https://en.wikipedia.org/wiki/DIII-D_(tokamak)#/media/File:2017_TOCAMAC_Fusion_Chamber_N0689.jpg

- Figure 2 created by author of this report