Linearly Recurrent Autoencoder for Plasma Profile Prediction

Aaron Wu

Summer 2020

Abstract

We implement a modified linearly recurrent autoencoder network that predicts future plasma profiles, given current actuator inputs and plasma state. Data collected from the 2010-2018 experimental campaigns on DIII-D was used to train and test the model. The model forecasts, on average, are most accurate with density profiles and least accurate with rotational transform. On a +300ms time horizon, the model does well, mostly outperforming a baseline model with few exceptions. Further, the model is simple to use and can generate predictions quickly. As such, an efficient C implementation of this model could be used in the development of a real-time control algorithm.

1 Introduction

Understanding plasma behavior in fusion reactors is an active area of fusion energy research. Out of the current reactors that exist worldwide, the tokamak design is perhaps the most ubiquitous. Numerous models aimed at describing properties of plasma evolution in tokamaks have been developed from physical principles. Given device parameters and sequence of actuator inputs, these allow for simulations of plasma, eliminating the need for costly experiments. However, their implementations in code are often computationally expensive, and when reductive approximations are applied, run-times are still on the order of minutes. Furthermore, there are aspects of plasma behavior in tokamak reactors that are not yet well-understood from a physical perspective, especially with regards to disruptions [1].

An alternative approach to predicting or monitoring plasma evolution is to build a model from experimentally collected data. Recent work has demonstrated that neural networks have potential in this field of research. Various types of networks trained on data from NSTX [2], DIII-D [3], ASDEX Upgrade [4], and JET [5] have all demonstrated success in predicting future properties of plasma or predicting plasma disruptions.

In this work, we document a recurrent neural network that generates a step-by-step prediction of plasma profiles 300ms into the future, given current plasma profiles, plasma parameters, and future actuator inputs. The rest of this paper is organized as follows: in section 2 we detail the motivation for our work, the model, and practicalities of its implementation and in section 3 we present the results from testing said model.

2 Machine Learning Model

In this work, we assume the plasma state evolves according to

$$\dot{\boldsymbol{x}} = g(\boldsymbol{x}, \boldsymbol{u}) \tag{1}$$

where x, u are the plasma state and actuator inputs, respectively. We aim to locally approximate the discretization of g using a neural network. The motivation for this model is described in detail in [6]. A brief summary is provided in the first subsection. We will then describe the data set, implementation, and training methods in the subsections that follow.

2.1 Background

First, consider a discrete unforced dynamical system $\boldsymbol{x_{t+1}} = h(\boldsymbol{x_t})$. Let $M \subseteq \mathbb{R}^p$ be the state space evolving with non-linear dynamics h. Assume that h is measurable with respect to the Borel sigma-algebra generated by M. Let $U := \{\varphi : M \to \mathbb{R} | \varphi \in L^2(M, \mu)\}$, where we take μ to be the Lebesgue measure. The Koopman Operator $\mathcal{K} : U \to U$ is the composition defined by $\mathcal{K}\varphi = \varphi \circ h$. Observe that the Koopman re-frames evolution as infinite dimensional but linear.

The hope is that for a given system, there exists an invariant subspace spanned by a finite number of eigenfunctions $\{\psi_1, ..., \psi_n\}$ such that $\pi_1, ..., \pi_p \in \text{span}\{\psi_1, ..., \psi_n\}$ where π_i is the projection onto the ith coordinate. Concatenating these linear combinations into p dimensional vectors, we obtain the Koopman modes $\boldsymbol{\xi_1}, ..., \boldsymbol{\xi_n}$. Since these Koopman modes are time invariant, we see that for any current state x_t, the state m steps into the future has the simple representation $\boldsymbol{x_{t+m}} = \lambda_1^m \boldsymbol{\xi_1} \psi_1(\boldsymbol{x_t}) + ... + \lambda_n^m \boldsymbol{\xi_n} \psi_n(\boldsymbol{x_t})$. Refer to the figure for an illustration of this concept.

$$\begin{tikzcd}
M \arrow[r, "h"] \arrow[d, shift right, "{\{\psi_1,...,\psi_n\}}"'] & M \arrow[d, shift right, "{\{\psi_1,...,\psi_n\}}"'] \\
U \arrow[r, "\mathcal{K}"] \arrow[u, shift right, "\boldsymbol{\xi_1}\psi_1+...+\boldsymbol{\xi_n}\psi_n"'] & U \arrow[u, shift right, "\lambda_1\boldsymbol{\xi_1}\psi_1+...+\lambda_n\boldsymbol{\xi_n}\psi_n"']
\end{tikzcd}$$

Two established methods for approximating the Koopman Operator from data are Extended DMD (EDMD) and Kernel DMD (KDMD) [7]. However, in the “real” world, choosing an appropriate function dictionary is a delicate task (whether explicitly in the case of EDMD or implicitly with KDMD). One can easily run into severe overfitting issues as shown in [6].

The machine learning approach attempts to fix this shortcoming. A linear recurrent kernel approximates the Koopman Operator and actuator input. More importantly, deep encode-decode layers replace the roles of the Koopman eigenfunctions and modes. The idea is that the learned encode-decode layers will simultaneously learn an optimal function dictionary and find a *non-linear* relationship between these functions that allow for accurate state reconstruction under the action of the recurrent operator. Furthermore, the learned invariant subspace is not confined to being the span of eigenfunctions, as the learned recurrent operator is not necessarily diagonal.

The addition of actuator inputs makes it harder to obtain a clean analytic representation of state evolution. The contents of [8], [9], and [10] explore this situation in more detail. In our work, we choose instead to assume that a separate encode/decode and input kernel is enough to capture the effect of the actuator inputs in the space of observables. See section 2.2 for more detail. Another assumption we make is that the learned representation of dynamics only holds locally, i.e. for sufficiently short time scales ($\sim$100-500ms). We have found that these reductions still give good results.

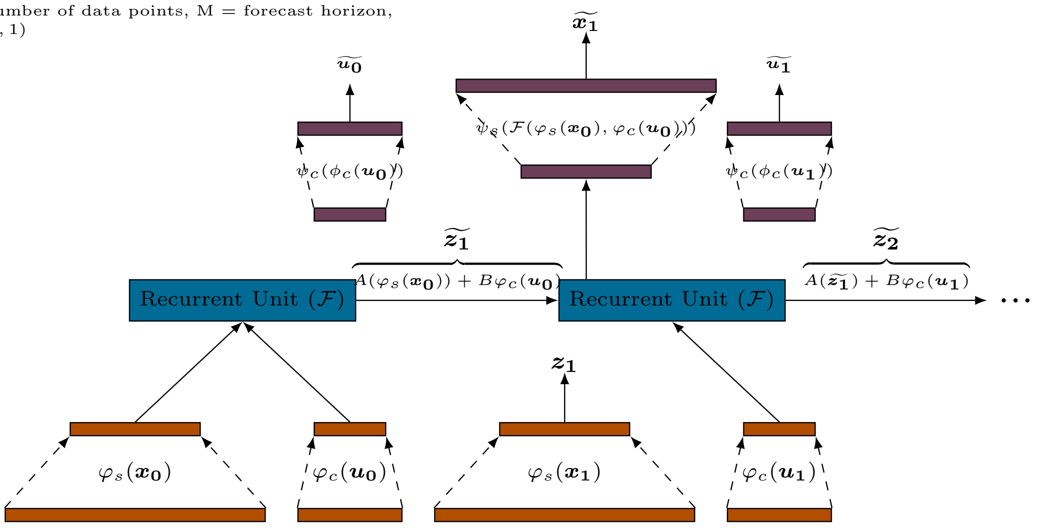
2.2 Architecture

The architecture in our work is a modification of the Linearly Recurrent Autoencoder Network as described in [6]. As opposed to setting an initial state and letting the recurrent unit evolve autonomously, our network provides encoded actuator inputs at each time step.

More concretely, let $\varphi(\boldsymbol{x_0})$ be the initial encoded state, $\varphi(\boldsymbol{u_t})$ be the encoded actuator input at time t, and $\widetilde{\boldsymbol{z_t}}$ be the recurrent unit output for time t. Then for the first time step, the model evolves like $A(\boldsymbol{x_0}) + B(\varphi(\boldsymbol{u_0}))$ instead of $A\boldsymbol{x_0}$. For the nth time step, $A(\widetilde{\boldsymbol{z_n}}) + B(\varphi(\boldsymbol{u_n}))$ instead of $A^n\boldsymbol{x_0}$. Additionally, we enforce reconstruction loss for both state and actuator inputs.

$$\text{Loss} = \frac{1}{N}\sum_{1}^{N}\left(\sum_{i=1}^{M} ||\widetilde{\boldsymbol{z_i}} - \boldsymbol{z_i}||^2 + \alpha \sum_{i=1}^{M} ||\widetilde{\boldsymbol{x_i}} - \boldsymbol{x_i}||^2 + (1-\alpha)\sum_{i=0}^{M} ||\widetilde{\boldsymbol{u_i}} - \boldsymbol{u_i}||^2\right)$$

N = number of data points, M = forecast horizon, $\alpha \in (0,1)$



$\widetilde{\boldsymbol{x_1}}$
$\widetilde{\boldsymbol{u_0}}$
$\widetilde{\boldsymbol{u_1}}$
$\psi_s(\mathcal{F}(\varphi_s(\boldsymbol{x_0}), \varphi_c(\boldsymbol{u_0})))$
$\psi_c(\phi_c(\boldsymbol{u_0}))$
$\psi_c(\phi_c(\boldsymbol{u_1}))$
$\widetilde{\boldsymbol{z_1}}$
$A(\varphi_s(\boldsymbol{x_0})) + B\varphi_c(\boldsymbol{u_0})$
$\widetilde{\boldsymbol{z_2}}$
$A(\widetilde{\boldsymbol{z_1}}) + B\varphi_c(\boldsymbol{u_1})$
Recurrent Unit ($\mathcal{F}$)
Recurrent Unit ($\mathcal{F}$)
...
$\boldsymbol{z_1}$
$\varphi_s(\boldsymbol{x_0})$
$\varphi_c(\boldsymbol{u_0})$
$\varphi_s(\boldsymbol{x_1})$
$\varphi_c(\boldsymbol{u_1})$

The encoding layers use ELU activation. ELU activation is also used in all decoding layers except for the last layer, which uses no activation. The number of units in each encode and decode layer was set to follow linear decay and growth, respectively, casted to the nearest integer. To increase generalizability, we employ dropout with rate 0.2 after each encode and decode layer (save for the last decode layer).

2.3 Data

Name	Source	Units
Electron Density	Zipfit	$10^{19}/\text{m}^3$
Electron Temperature	Zipfit	keV
Ion Rotation	Zipfit	kHz
Rotational Transform [1] (ι)	EFIT02	-
Plasma Pressure	EFIT02	Pa
Injected Torque	tinj	N·m
Target Current	iptipp	A
Target Density	dstdenp	$10^{19}/\text{m}^3$
Gas Injection A	GasA	V
Bottom Triangularity	EFIT02	-
Top Triangularity	EFIT02	-
Plasma Volume	EFIT02	m^3
Internal Inductance	EFIT02	μH
Line Avg. Density	dssdenest	$10^{19}/\text{m}^3$

Table 1: Signals used in profile prediction neural network. “Source” denotes the algorithm or MDS+ pointname the data was obtained from. From top to bottom the three categories of data are plasma profiles, plasma parameters, and actuator inputs.

The data used to train the model fall into three categories: plasma profiles, plasma parameters, and actuator inputs. All data was arranged into time slices spaced 50ms apart. Previous work (LSTM Paper) was used as a reference to ascertain which profiles, parameters, and actuators would work best with machine

[1]The rotational transform $\iota = 1/q$ was found to present fewer numerical difficulties due to the singularity at the separatrix.

learning. Density, temperature, and rotation profiles consist of 33 equally spaced points in space with respect to normalized torodial flux coordinates, where $\rho = 0$ is the magnetic axis and $\rho = 1$ is the last closed flux surface. Similarly, rotational transform and pressure profiles consist of 33 equally spaced points in space but with respect to normalized polodial flux coordinates, denoted by ψ. The parameters and actuator inputs were both scalar inputs in time. Each input was normalized by subtracting the median and dividing by the interquartile range. The median and interquartile range was computed over flattened data.

Some non-standard data was excluded including data during or after a dudtrip (error in PCS) trigger, during and after ECH activation, during and after non-normal operation of internal coils, whenever the data did not include the needed signals.

10% of the data was reserved as a never-seen-before test set. Of the remaining 90%, a randomly selected 10% was designated as validation data and the rest used as training data. The samples were then put in batches of size 128. For each pass through the training/validation data, the ordering of these batches was randomized before being passed to the model.

2.4 Model Training and Hyperparameter Tuning

When training the model, we experimented with various model configurations in order to optimize performance. Hyperparameters of particular interest included the number of layers in encode-decode, dimension of the state and control feature space, and lookahead. Table 2 shows the parameter choices we found to yield the best results.

Parameter	Value
Lookahead	6 (+300ms)
Num. state encoding layers	10
Num. state decoding layers	10
State encode-decode activation type	ELU
State feature space dimension	65
Num. actuator encoding layers	1
Num. actuator decoding layers	1
Actuator encode-decode activation type	Linear
Actuator feature space dimension	4
Batch size	128
Number of training epochs	200
Optimizer	ADAM [11]
Learning rate	0.0001

Table 2: Hyperparameters

Implementation was accomplished using the Keras functional API [12] with TensorFlow backend [13]. The model was trained using 8 IBM POWER9 CPU cores and 1 Nvidia V100 GPU on Princeton University's Traverse cluster. Total time to train took approximately 4 hours. Code for this work is publicly available on GitHub.[2] Further documentation is provided therein.

3 Model Performance

After training for 200 epochs, the model achieved a training loss of 0.014 and a validation loss of 0.018. We note that the eigenvalues of the recurrent kernel are mostly of modulus less than one, with only one or two being slightly greater than one. This agrees with the intuition that well-contained plasma is stable for short time windows.

[2]https://github.com/jabbate7/plasma-profile-predictor/tree/aaronwu667

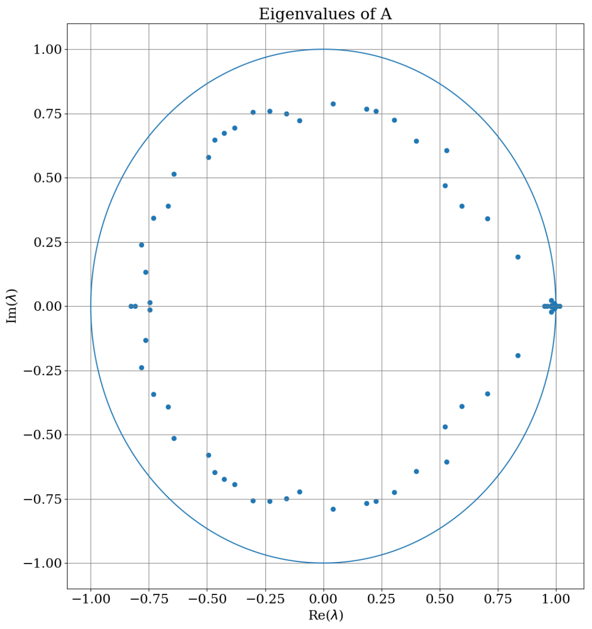
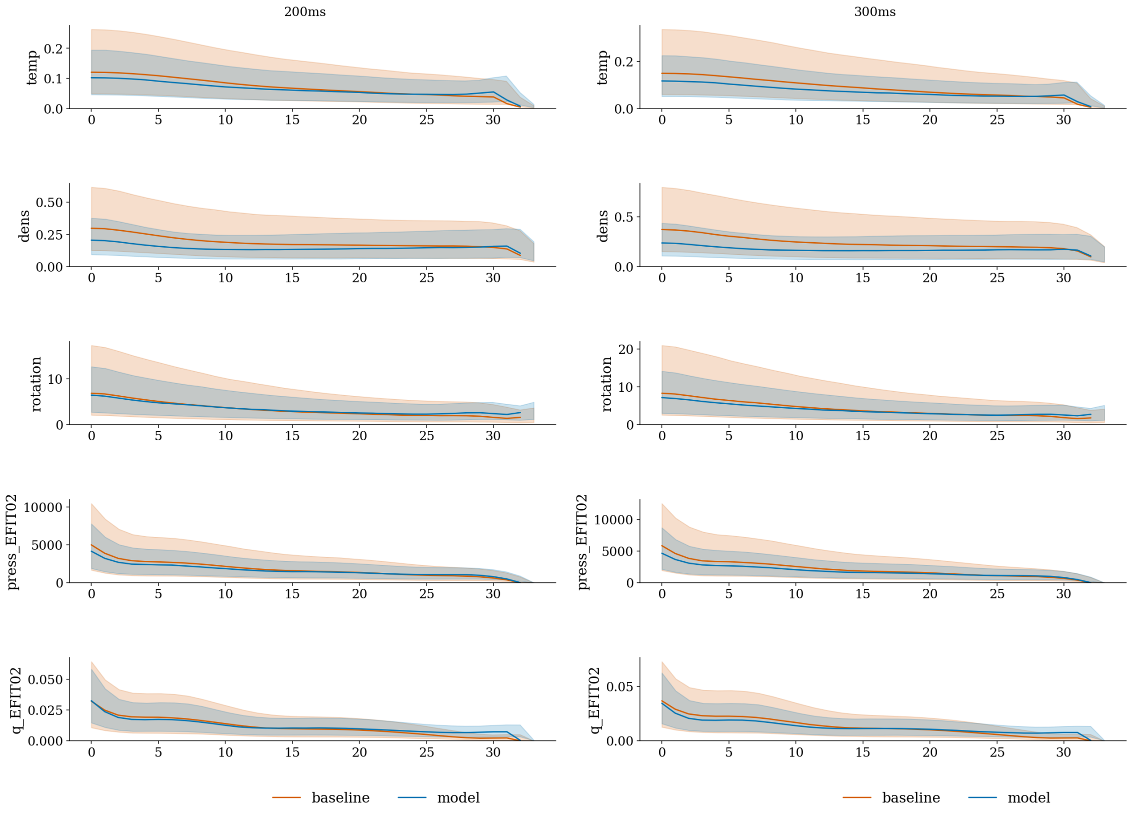


Figure 1: Spectrum of learned Koopman

To see how the model performs on the never-seen-before test set, we compare model performance over each profile with the baseline model which predicts no change. This baseline is motivated by the desire to have the model learn meaningful plasma evolution rather than noise. Furthermore, we found that that the baseline model does quite well statistically.



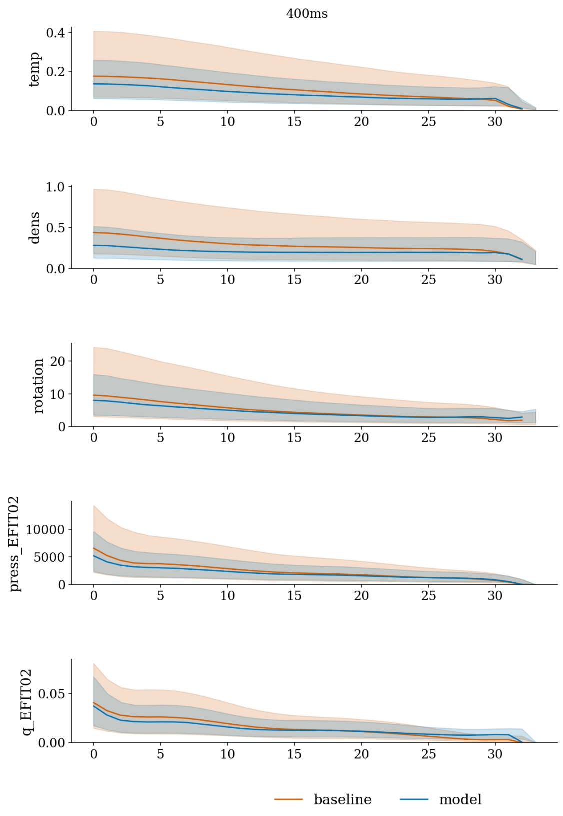
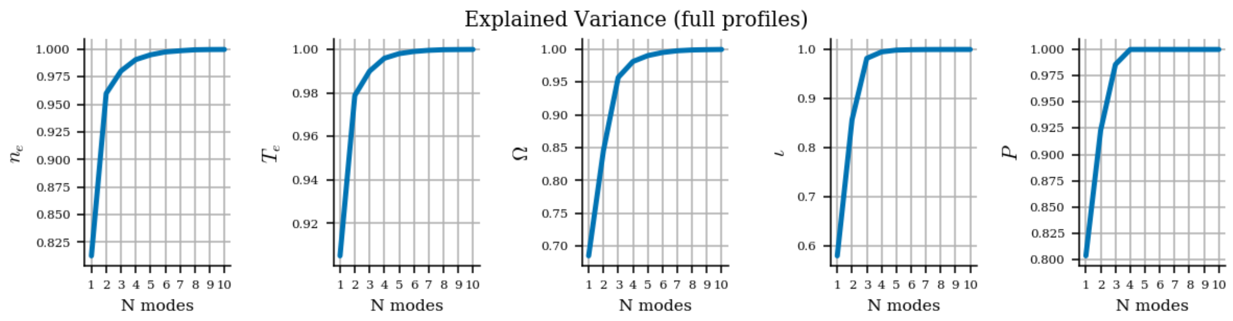


Figure 2: Baseline versus model performance at varying time horizons. Shaded regions indicate 25th and 75th percentiles.

As shown in figure 2, the model mostly outperforms or does about as good as the baseline for all profiles. The spikes in model error towards the outer flux surfaces in rotational transform and temperature could be the result of noise in the data.

Another way to evaluate model performance is through comparison of the PCA coefficient distributions. Geometrically, PCA decomposition finds a coordinate transform such that the first basis element captures the largest amount of variance in the data while the last basis element captures the least amount of variance. The original data can then be expressed by appropriate linear combinations of these basis elements. That is, the PCA coefficients are the coordinates of the data in the new basis. If the model is performing well, one would expect to see similar distributions in the PCA coefficients of the empirical data and the predictions. Plotting the number of basis elements used versus percentages of explained variance, we see that the first 2 basis elements (modes) are enough to capture at least 90% of variance in all profiles.



Thus, it is reasonable to only consider the coefficients of the first two PCA modes. For posterity, we also include the element-wise mean of coefficients over all PCA modes in our analysis.

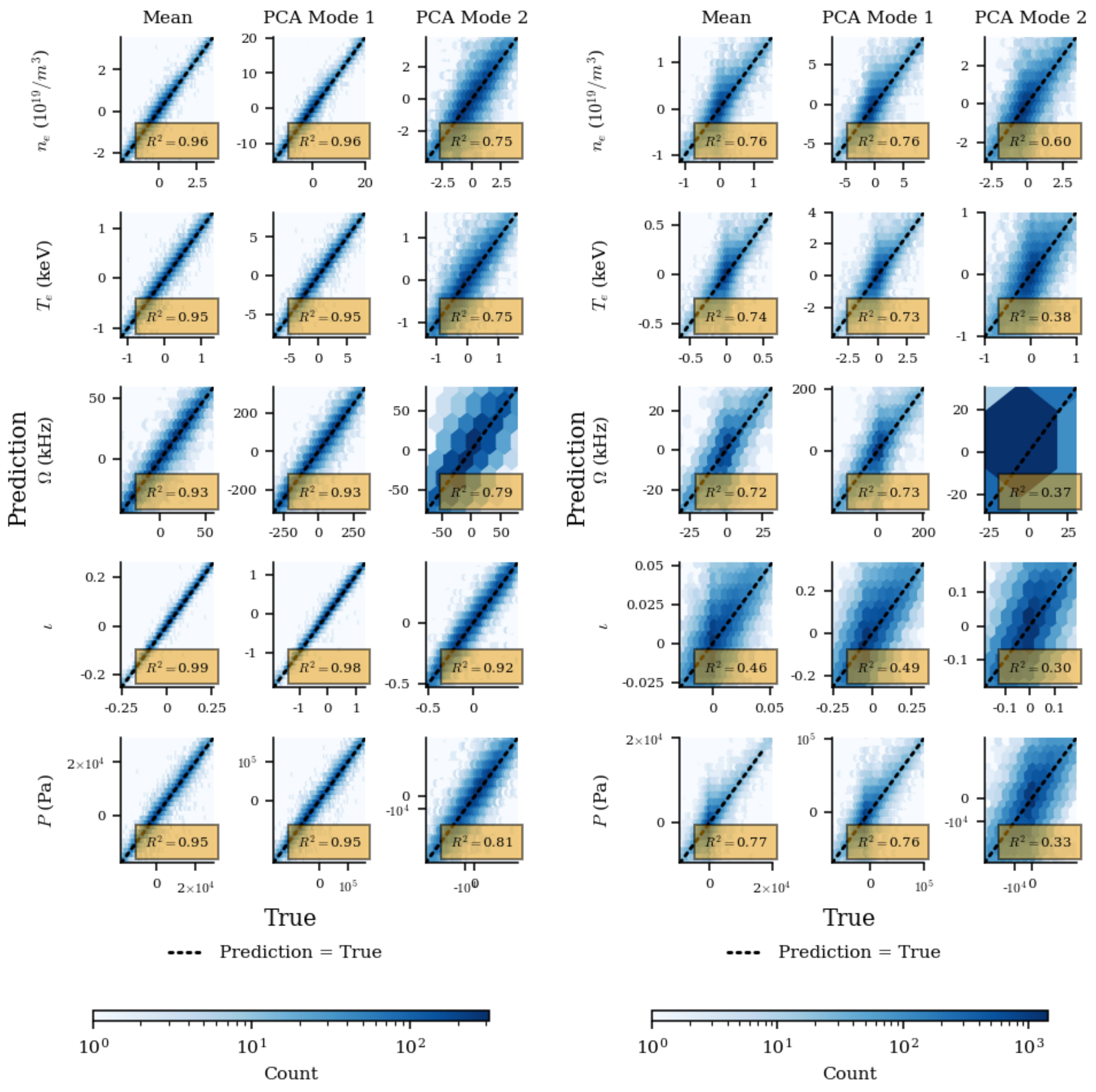


Figure 3: (left) Plot of PCA coefficients for full profile at 300ms lookahead. (right) Plot of PCA coefficients for change in profile over 300ms

For all profiles, we see good agreement between the PCA coefficients of the predicted data and the actual data. Note that the reduced performance on the second PCA mode is to be expected, as the second PCA mode captures less meaningful information about the dataset than the first.

4 Conclusion

We have implemented and tested a variation of the Linearly Recurrent Autoencoder network on data collected from DIII-D. This variation encodes actuator inputs at each time-step and passes the result to the recurrent kernel. Results from this model on a time horizon of +300ms are good and mostly outperform the statistical baseline of no change.

Generating predictions from this model takes on the order of milliseconds. An efficient re-implementation using the keras2c[3] library could further reduce the prediction time. This, and the network architecture make

[3]https://github.com/f0uriest/keras2c

our model a good candidate for real-time control. Given a target future state, finding the optimal set of actuator inputs can mostly be done in the feature space, where evolution in time is inexpensive matrix addition and multiplication. The only non-linearity arises in encoding the initial state and target state (single linear layer sufficient for actuator encoding, see table 2). This contrasts with a model-predictive control approach. In this case, the initial state must be encoded, evolved, then decoded and the target state must be encoded then decoded. Note that the state decoding layers are still necessary to ensure that the trivial encoding is not learned and to ensure that the feature space dynamics accurately capture the real system dynamics.

There is still room for improvement. What causes the spikes in error in the predicted q and rotation profiles towards the last flux surface is still an unresolved problem. Fixing these anomalies should be the first priority for future work. Furthermore, the used data represent only tens or hundreds of reactor operating regimes as many experiments reload previous shots. At heart, this is a generalizability issue as this model might not fare well in radically different regimes. Augmenting the dataset with shots that vary actuators in a semi-random manner would help with this issue. We also remark that this model only produces point forecasts and not uncertainty estimates. Developing a method to predict both future state and uncertainty in prediction is another key area for future research.

5 Acknowledgements

The author would like to thank Rory Conlin, Joe Abbate, and Egemen Kolemen for their guidance and the Andlinger Center for the Energy and the Environment for hosting the internship experience. This project was supported by the Peter B. Lewis Fund for Student Innovation in Energy and the Environment.

References

[1] Allen H. Boozer. Theory of tokamak disruptions. *Physics of Plasmas*, 19(5):058101, 2012.

[2] M.D. Boyer, S. Kaye, and K. Erickson. Real-time capable modeling of neutral beam injection on NSTX-u using neural networks. *Nuclear Fusion*, 59(5):056008, mar 2019.

[3] 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*, 27(6):062510, 2020.

[4] B. Cannas, A. Fanni, G. Pautasso, and G. Sias. Disruption prediction with adaptive neural networks for asdex upgrade. *Fusion Engineering and Design*, 86(6):1039 – 1044, 2011. Proceedings of the 26th Symposium of Fusion Technology (SOFT-26).

[5] 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*, 48(1):36–45, 2020.

[6] Samuel E. Otto and Clarence W. Rowley. Linearly recurrent autoencoder networks for learning dynamics. *SIAM Journal on Applied Dynamical Systems*, 18(1):558–593, 2019.

[7] Matthew O. Williams, Ioannis G. Kevrekidis, and Clarence W. Rowley. A data–driven approximation of the koopman operator: Extending dynamic mode decomposition. *Journal of Nonlinear Science*, 25(6):1307–1346, Jun 2015.

[8] Milan Korda and Igor Mezić. Linear predictors for nonlinear dynamical systems: Koopman operator meets model predictive control. *Automatica*, 93:149–160, Jul 2018.

[9] Eurika Kaiser, J. Nathan Kutz, and Steven L. Brunton. Data-driven discovery of koopman eigenfunctions for control, 2017.

[10] Joshua L. Proctor, Steven L. Brunton, and J. Nathan Kutz. Generalizing koopman theory to allow for inputs and control. *SIAM Journal on Applied Dynamical Systems*, 17(1):909–930, 2018.

[11] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[12] François Chollet et al. Keras. https://keras.io, 2015.

[13] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems. https://www.tensorflow.org/, 2015. Software available from tensorflow.org.