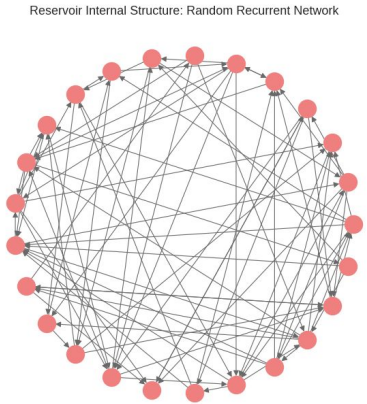
Reservoir Computing for prediction of time series data for nonlinear systems

Vimarsh Shah
Department of Physics
BITS Goa
f20221060@goa.bits-pilani.ac.in

Challenges with Temporal Data

- Machine learning excels at pattern recognition.
- Traditional Deep Learning uses backpropagation for training.
- Recurrent Neural Networks (RNNs) are designed for sequential/temporal data.
  - Problem: Training RNNs is hard!
  - Vanishing/exploding gradients
- Computationally expensive (time-consuming)
- Need for alternative approaches for recurrent architectures.



Enter Reservoir Computing (RC)

- Maintains representational power of recurrent networks.
- Dramatically simplifies training.
- Key Idea: Use a fixed, random recurrent network (the "reservoir") and only train the output layer.

Advantages:

- Simple training (linear regression)
- Reduced computational cost
- Inherent memory for temporal processing
- Adaptable to physical implementations

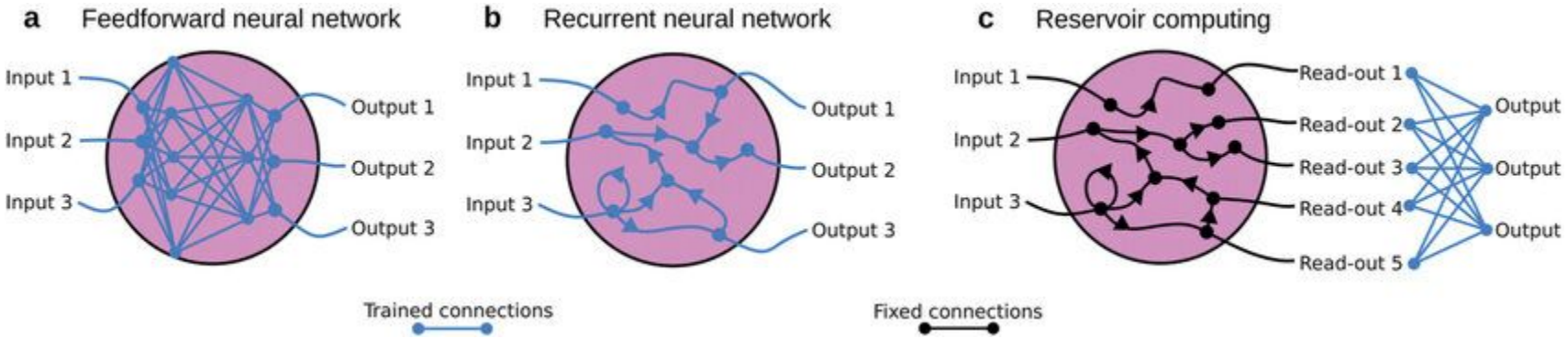


Illustration of
(a) a feedforward neural network (FNN) with its acyclic and trainable connections,
(b) a recurrent neural network featuring trainable feedback and cyclic connections, and
(c) a reservoir made of untrained connections.
For RC, only the connection between readout layer and output layer is trained, typically with a linear regression.

General Principle of RC

RC transforms inputs using a high-dimensional, nonlinear dynamical system (the reservoir).

Components:
Input Layer: Maps input u[n] to reservoir (fixed weights W^{in}).
Reservoir: Recurrent network creates high-dimensional state x[n] (fixed weights W).
Output Layer: Linear readout maps reservoir state to output y[n] (trained weights W^{out}).

Reservoir Update:

$\boldsymbol{x[n+1] = f\,Wx[n] + W^{in}u[n] + b}$ | (f is activation function like tanh)
$\mathbf{y[n] = W^{out}x[n];\ u[n]}$ | (W out trained via linear regression)

Key: Internal dynamics (W, W^{in}) are fixed and random, only W^{out} is learned. Efficient!

Echo State Property (ESP) - Fading Memory

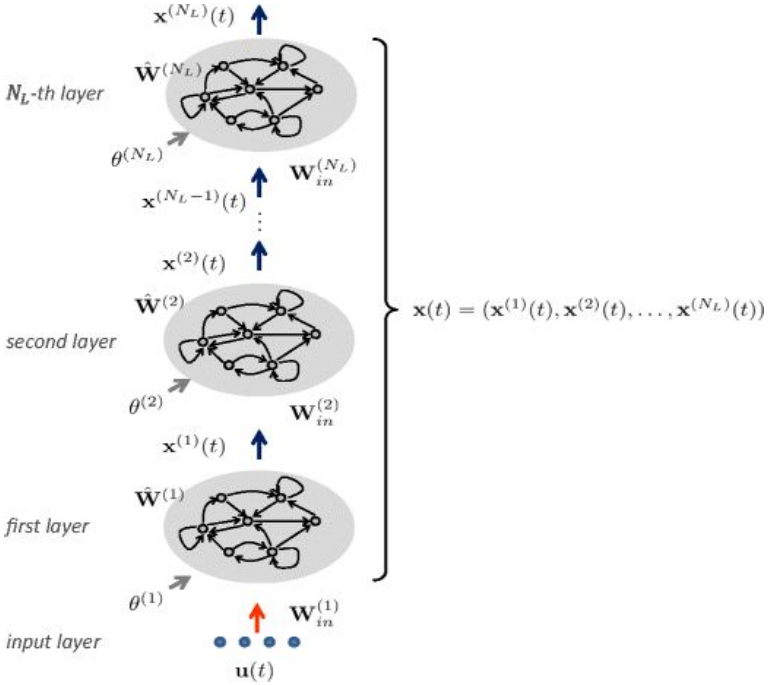
ESP: Ensures reservoir state depends on input history, not initial conditions.

- Influence of initial state fades away.
- Crucial for consistent processing of temporal data.

Fading Memory, a consequence of esp: Reservoir retains information about recent inputs, with influence decaying over time.

- Essential for capturing temporal dependencies.
- Often achieved by ensuring spectral radius ρ(W) < 1.

There needs to be a balance between the size of reservoir i.e., Sufficient memory capacity vs. nonlinearity for rich feature representation.



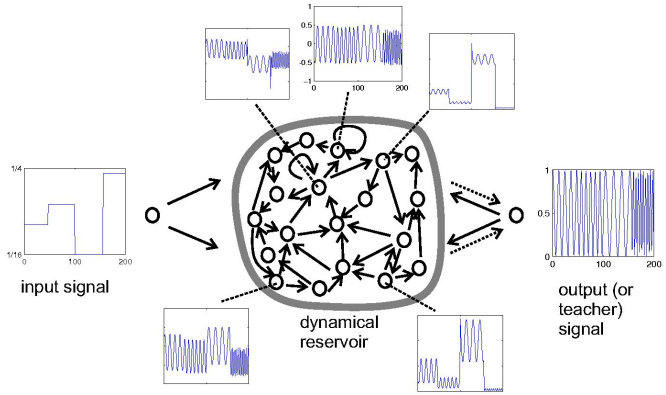
Summary Comparison

Feature	Traditional RNNs	Reservoir Computing (RC)
Training	All connections (input, recurrent, output)	Only output connections
Method	Backpropagation using appropriate loss function	Linear Regression (typically ridge regression used)
Complexity	High, computationally intensive Issues with Vanishing/exploding gradients	Low, efficient Performance depends on fixed reservoir

Variants of Reservoir Computing

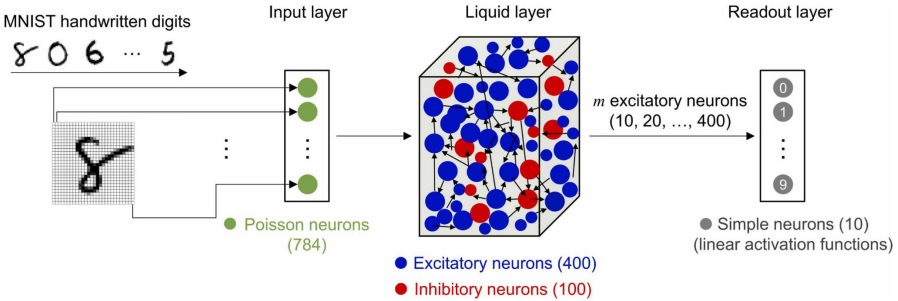
Echo State Networks (ESNs)

- Continuous activation (like tanh).
- Sparse random connections.
- Introduced by Jaeger



Liquid State Machines (LSMs)

- Spiking neuron models.
- Biologically inspired.
- Developed by Maass.



Training Methodology

Simple training process:

- **Collect States:** Feed input signals u into the fixed reservoir and record the resulting states x over time. Let G be the matrix of collected states.
- **Train Readout:** Solve a linear regression problem to find the output weights Wout that map reservoir states G to the target outputs Y .

  $$\mathbf{W}_{\text{out}} = YG^T \left(GG^T + \lambda I \right)^{-1}$$

  (Ridge Regression formula, where λ is regularization parameter).

- **Inference:** Use the trained W_{out} to predict outputs for new input sequences based on their corresponding reservoir states.

Advantages:

- Very fast training (linear regression).
- Avoids gradient issues of RNNs.
- Same reservoir potentially usable for multiple tasks (train different readouts).
- Enables physical implementations (almost instant inference).

Limitations:

- Random initialization offers little control over reservoir function.
- Memory capacity limited by reservoir size. The concept of sequence length is loosely defined here, but the performance does degrade as we increase sequence length.
- Performance highly dependent on task and hyperparameters.
- Very sensitive to choice of reservoir dynamics.

Experiment 1: Pendulum as a Reservoir

Setup:

- Use a simulated driven, damped pendulum.
- Input signal = time-varying driving force.
- Reservoir state = [angle (x), angular velocity (v)].
- Train linear readout on state history.

$$\frac{dx}{dt} = v,$$
$$\frac{dv}{dt} = -\frac{g}{l}\sin(x) - kv + f\,\mathrm{sign}(\sin(\omega t))$$

Simple Prediction Result:

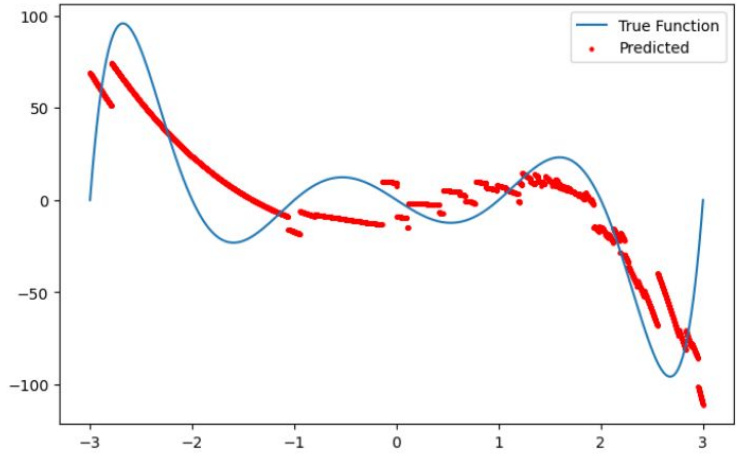


Figure: Predicting next state based on previous (x_{t-1}). Shows learning capability.

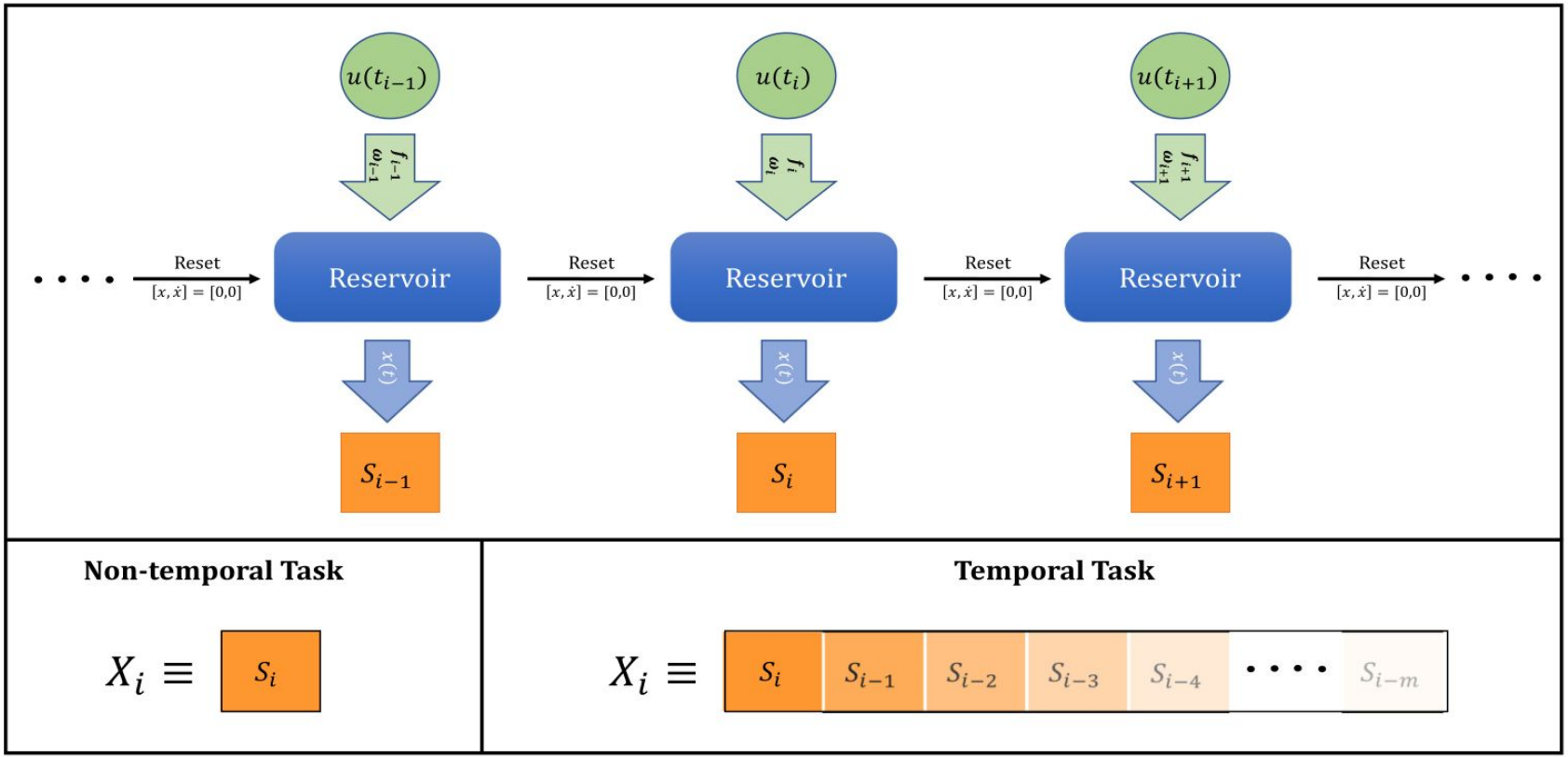


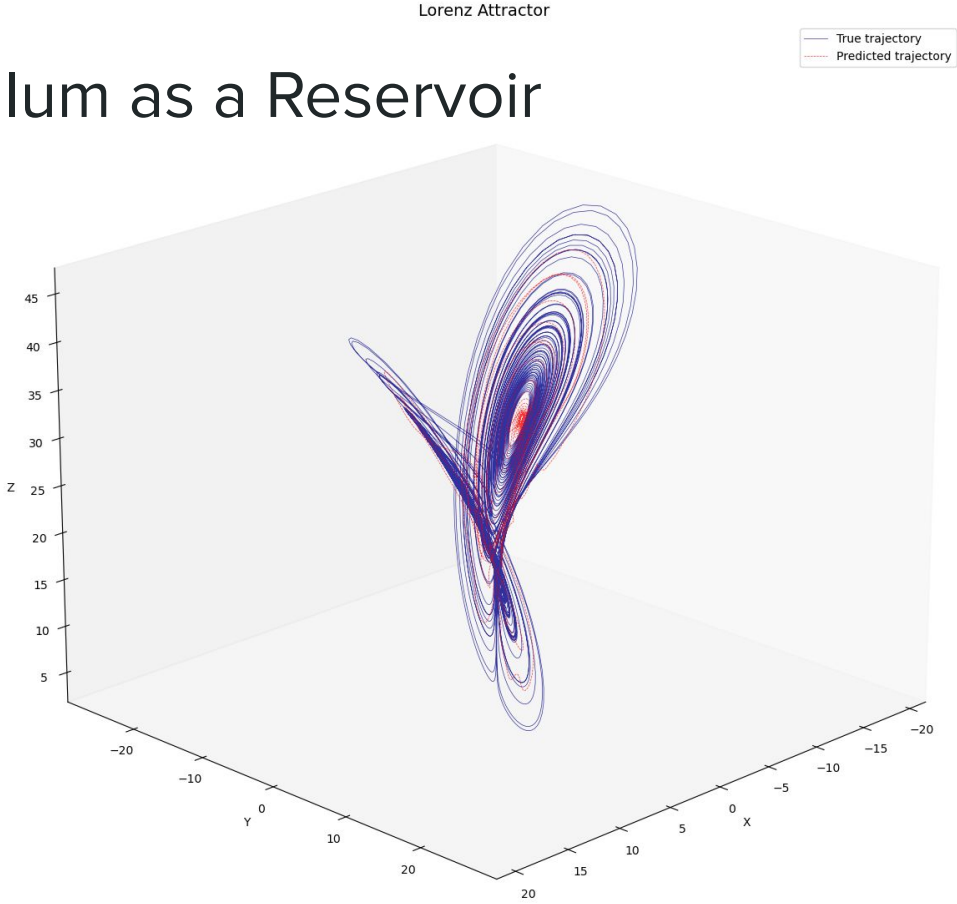
FIG. 6. Top: The schematic for the training procedure. $u(t_i)$, which refers to one particular point of input data set $u(t)$, is fed to the reservoir by setting the driving force amplitude (f) and frequency (ω) accordingly (refer to step 3). Bottom: The process to form reservoir state vector X_i using reservoir dynamics S_i, in which the fading color intensity represents decreasing weights for the temporal task case (see step 4).

Lorenz System using Pendulum as a Reservoir

The Lorenz system is a classic example of a chaotic dynamical system, described by the following set of ordinary differential equations:

$$\frac{dx}{dt} = \sigma(y - x),$$
$$\frac{dy}{dt} = x(\rho - z) - y,$$
$$\frac{dz}{dt} = xy - \beta z,$$

$$\sigma = 10, \quad \rho = 28, \quad \beta = \frac{8}{3}$$



Experiment 2: Logistic Map as a Reservoir

Reservoir constructed using iterates of the logistic map:

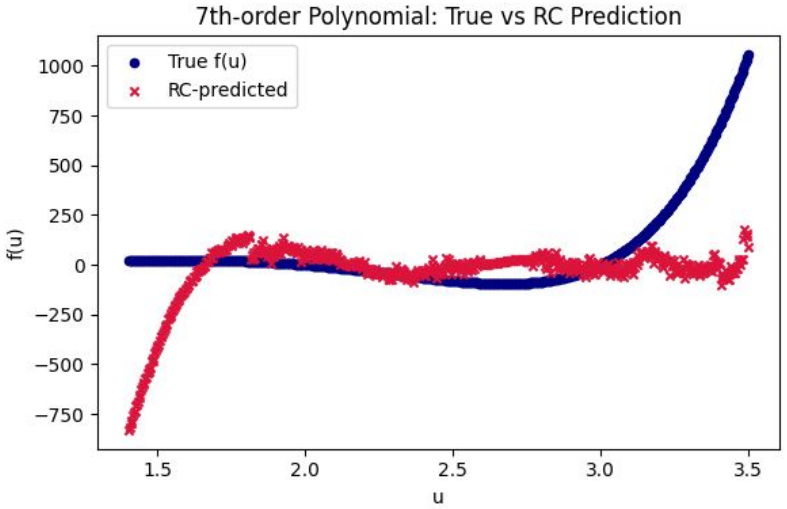
$$x_{n+1} = r x_n (1 - x_n)$$

Virtual Nodes Technique: For each input, iterate map multiple times, sample intermediate values to form high-dimensional state vector.

Train linear readout on this state.

Benchmark Task (Polynomial):

Predict a 7th-degree polynomial target.



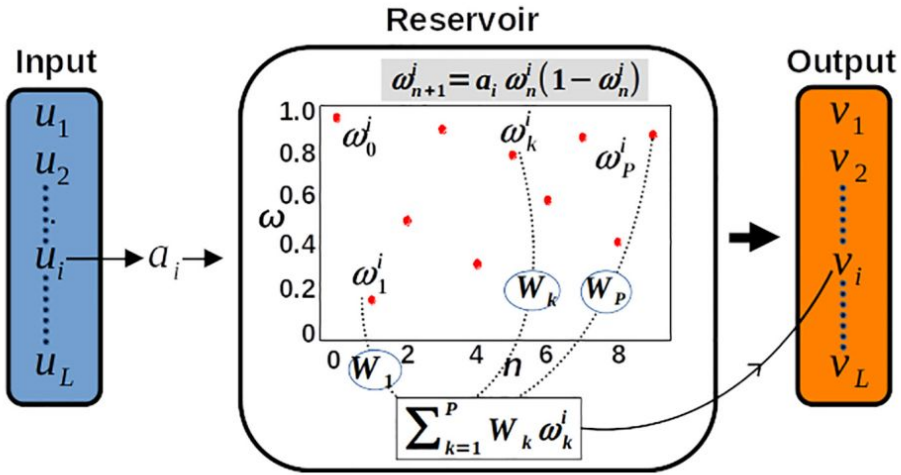


FIG. 2. A schematic diagram for training with the logistic map. The input u_i is linearly transformed into a_i and then supplied to the logistic map to form virtual nodes for the reservoir. V_i, $i = 1, 2, \ldots, L$ is the output generated from the reservoir. The iterated values of the logistic map (ω^i_k, $k = 1, 2, \ldots, P$) are indicated as red dots. W_k, $k = 1, 2, \ldots, P$, are the components of the weight matrix W. The black dotted lines inside the reservoir indicate the internal computations of the reservoir.

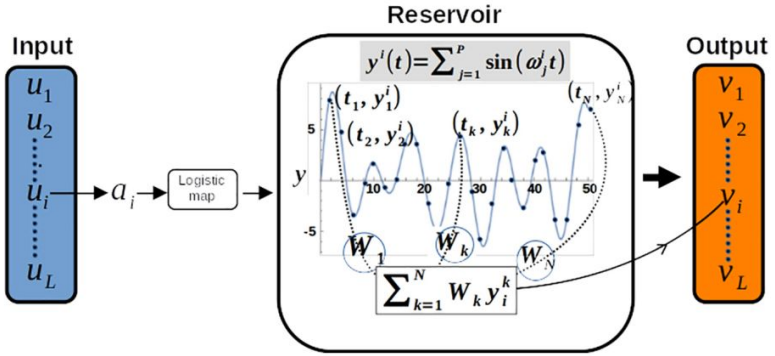
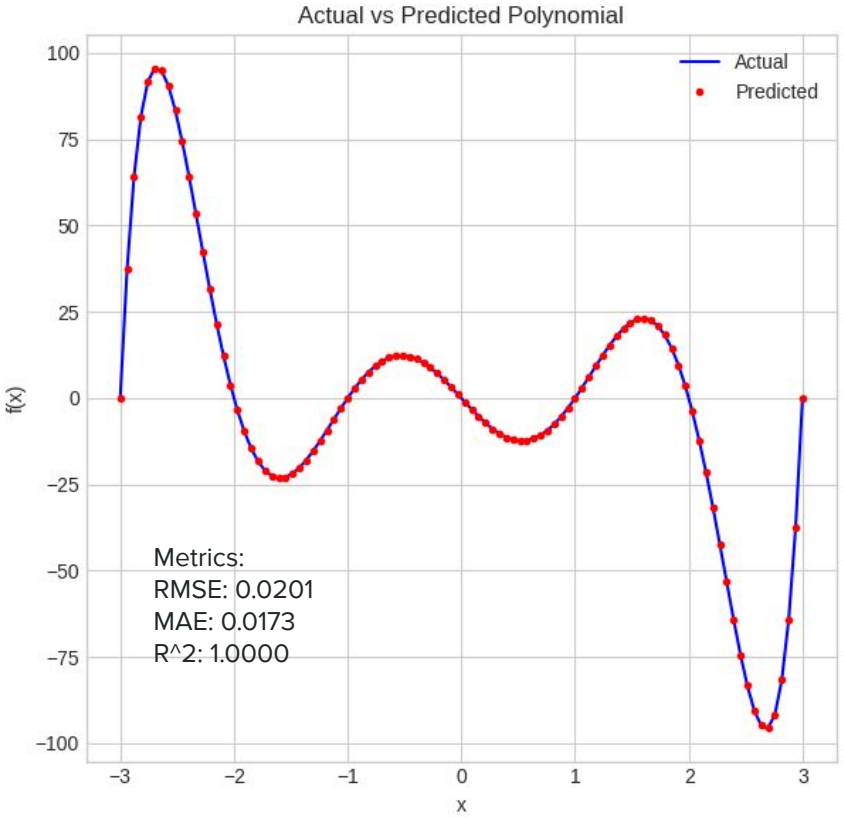
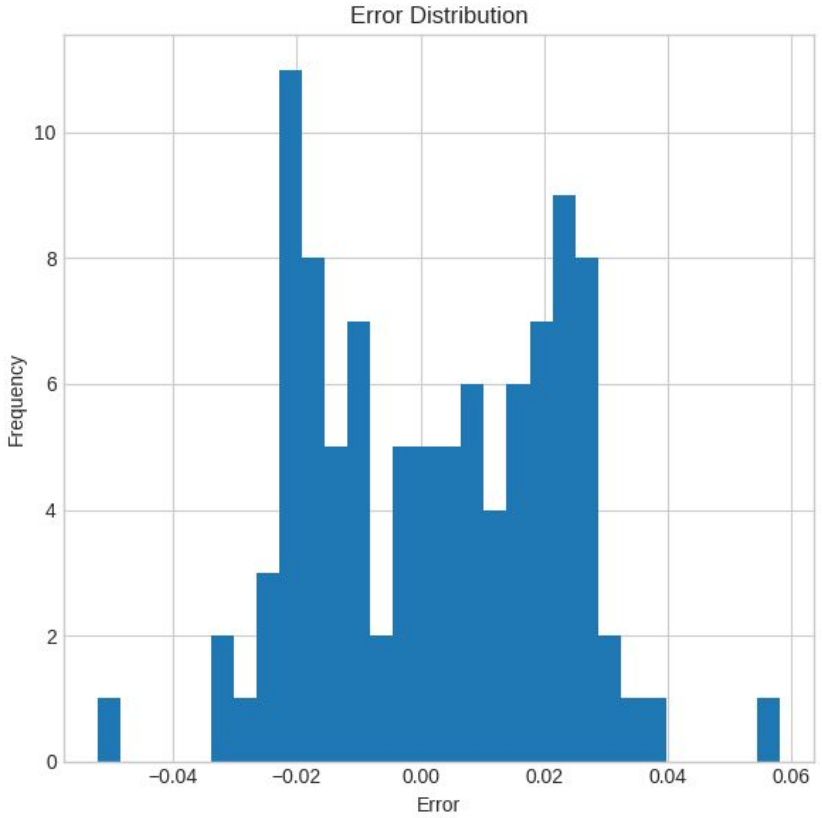


FIG. 13. The schematic diagram for training with trigonometric function. The input u_i is linearly transformed into a_i and then is supplied into logistic map to form a time varying function. V_i, $i = 1, 2, \ldots, L$ is the output generated from the reservoir. The W_k, $k = 1, 2, \ldots, P$ are the components of the weight matrix W. The black dotted lines inside the reservoir indicate the internal computations of the reservoir.

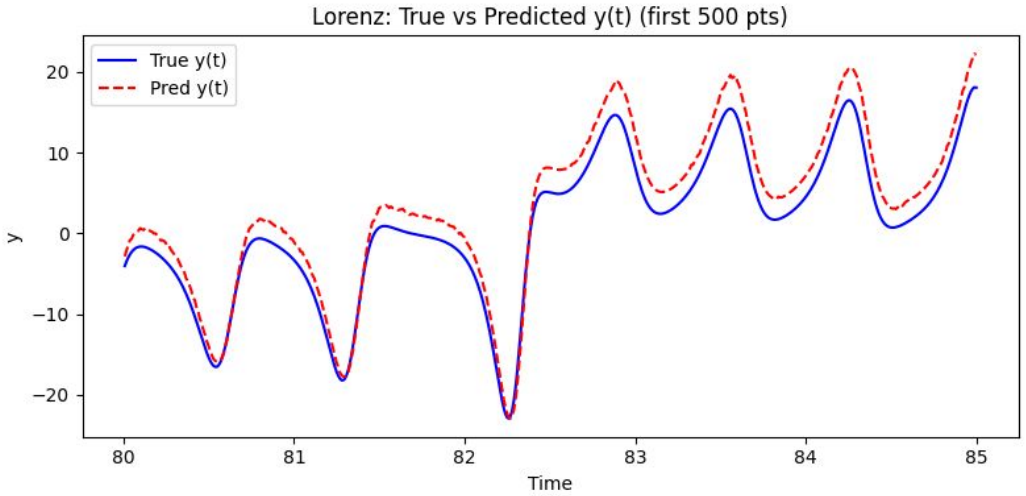

Actual vs Predicted Polynomial
Actual
Predicted
f(x)
x
Metrics:
RMSE: 0.0201
MAE: 0.0173
R^2: 1.0000

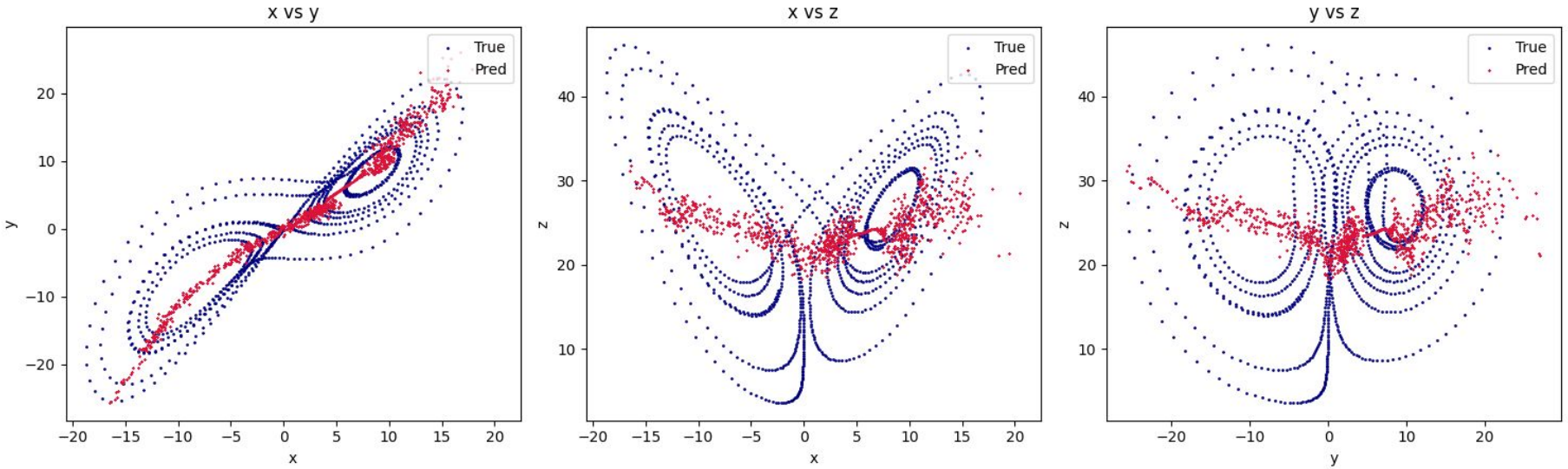

Error Distribution
Frequency
Error

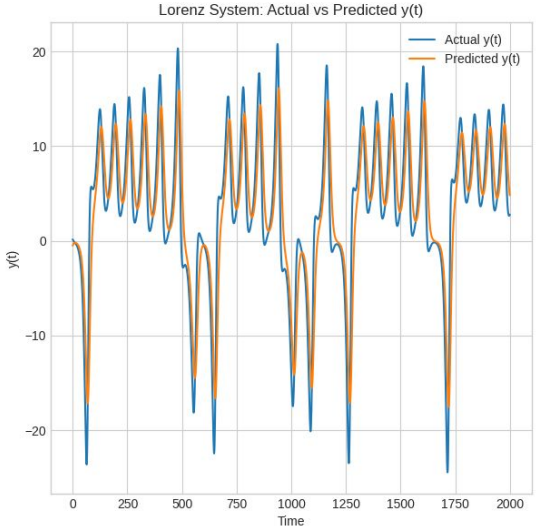
Logistic Map Reservoir: Lorenz System Prediction

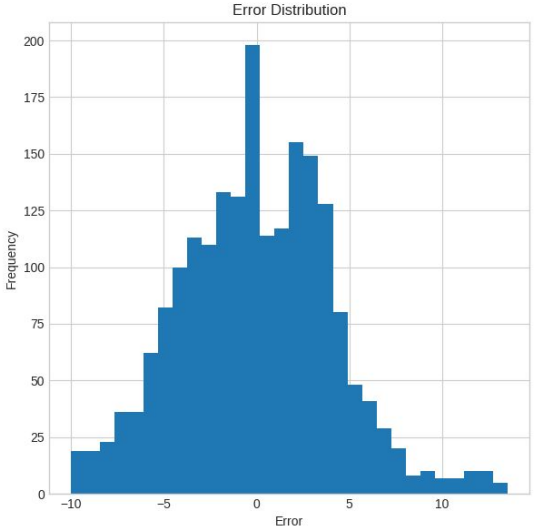
Same task as before, but using the logistic map reservoir.

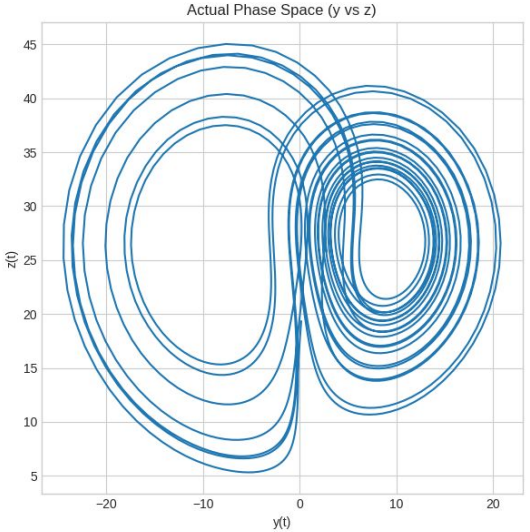
Results: **Test RMSE (x,y,z): [3.77, 2.94, 8.36]**




Lorenz Attractor Projections: True (blue) vs Predicted (red)
x vs y
x vs z
y vs z
True
Pred
x
y
z

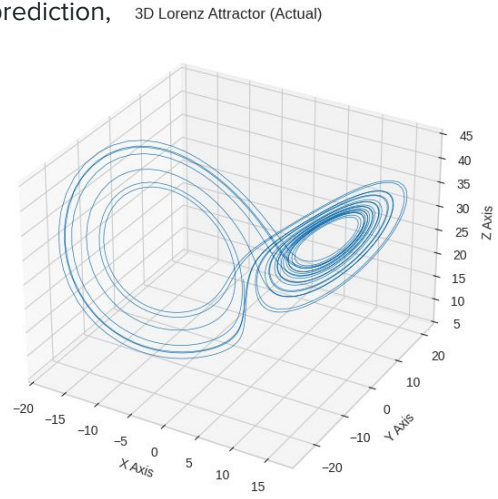


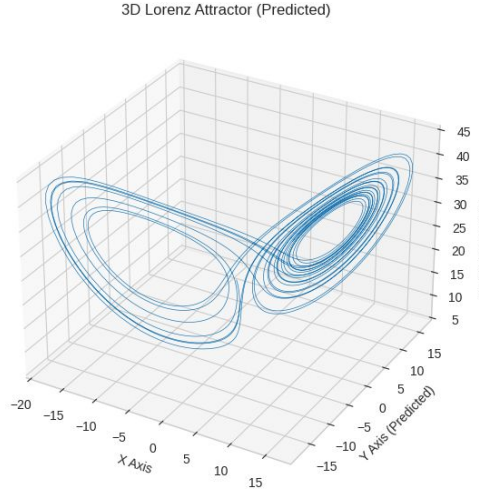


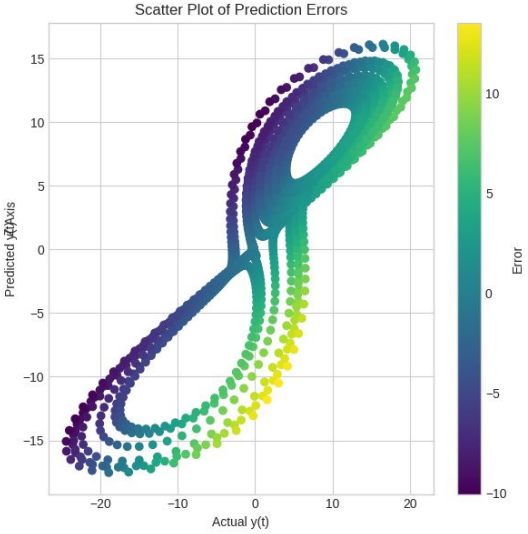


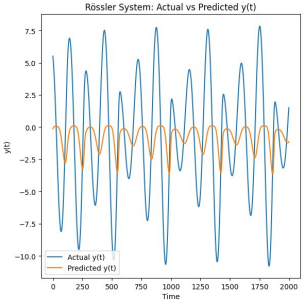
Lorenz System prediction,
y from x

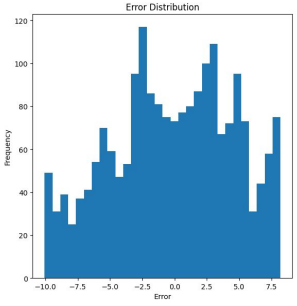
Metrics:
RMSE: 4.1304
MAE: 3.2829
R^2: 0.7526

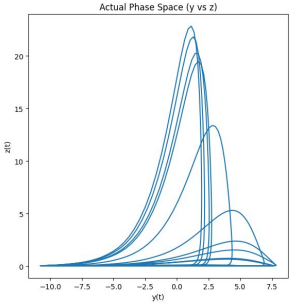


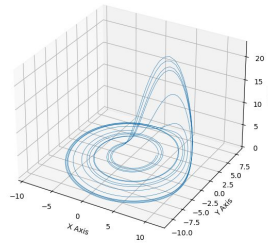


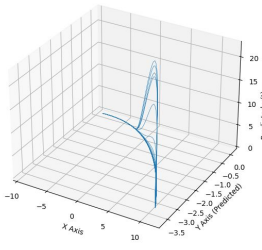


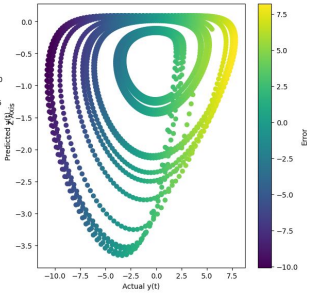






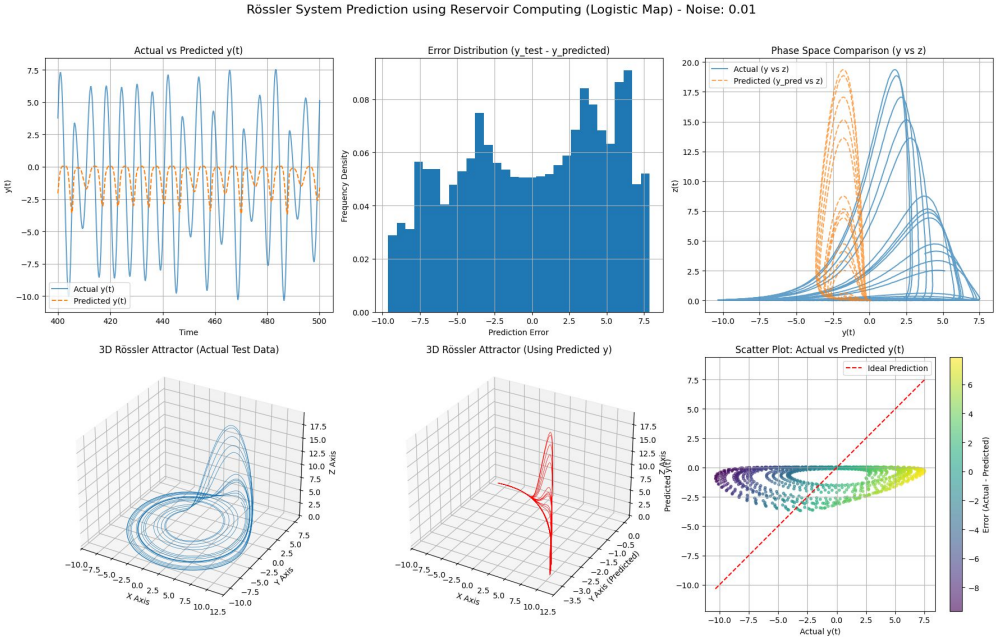




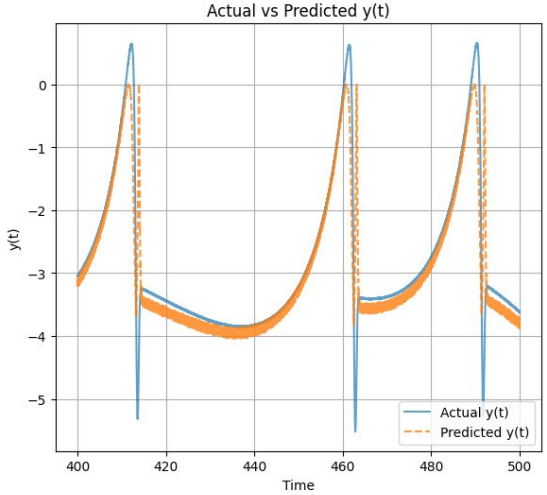


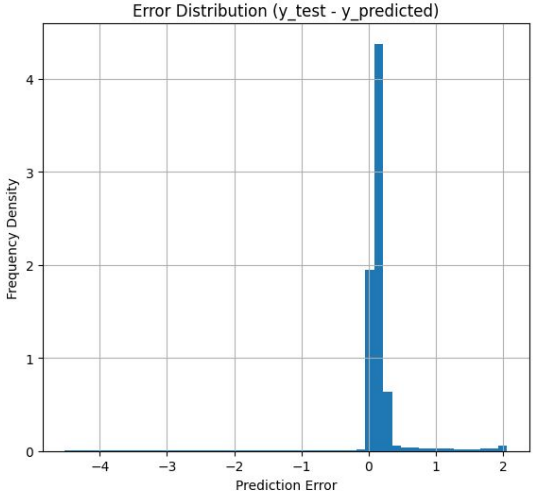
RMSE: 4.6990
MAE: 3.9526
R^2: 0.0396

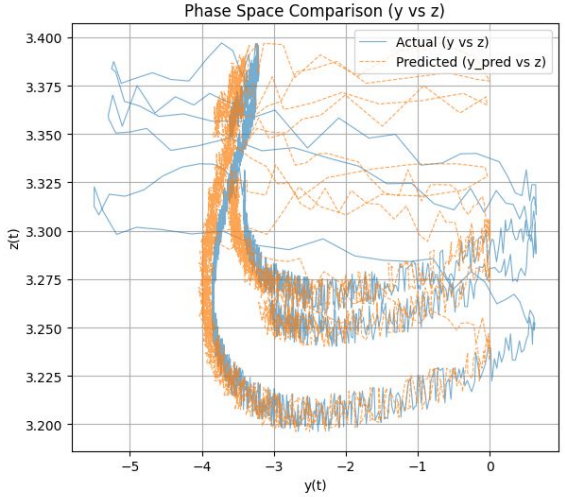
Rössler System Prediction (y from x)
RMSE: 4.8916
MAE: 4.2733
R^2: 0.0332



Hindmarsh-Rose System Prediction (y from x) using Logistic Map RC - Noise: 0.01



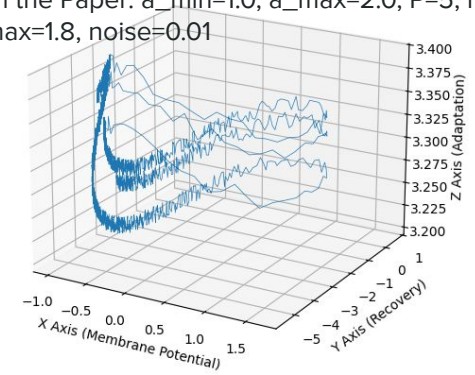


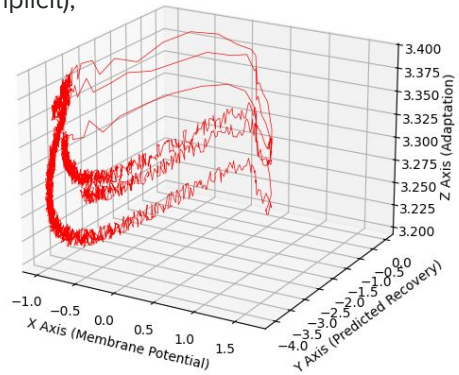


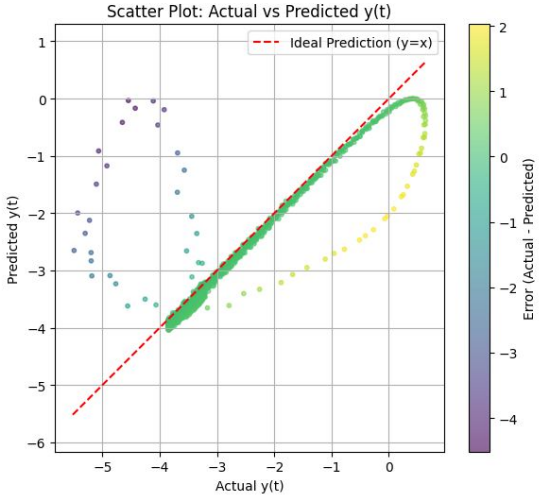
3D Hindmarsh-Rose Attractor (Actual Test Data)

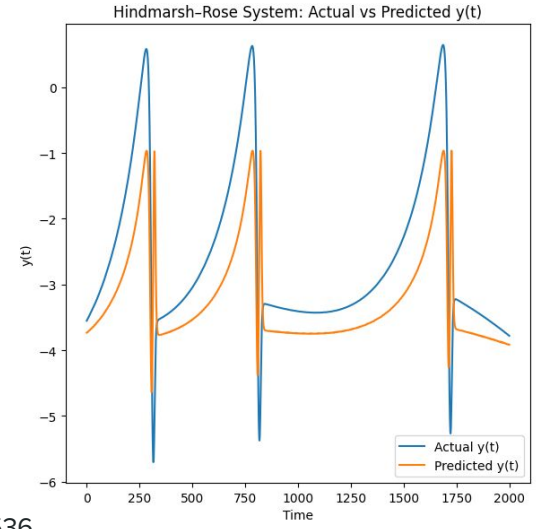
3D Hindmarsh-Rose Attractor (Using Predicted y)

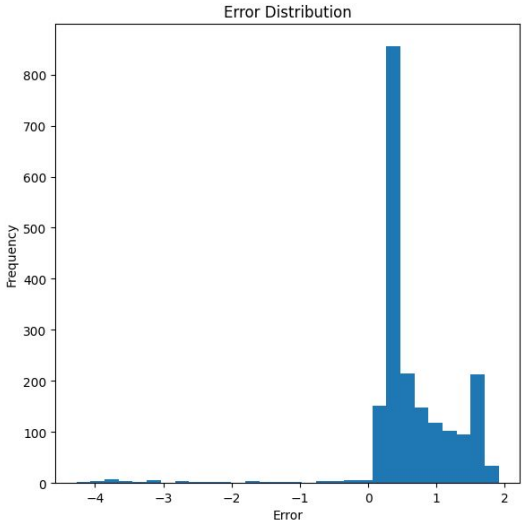
Hindmarsh-Rose System Prediction (y from x using Logistic Map RC)
Parameters from the Paper: a_min=1.0, a_max=2.0, P=5, m=100 (implicit),
u_min=-1.2, u_max=1.8, noise=0.01
RMSE: 0.5751
MAE: 0.2481
R^2: 0.7762

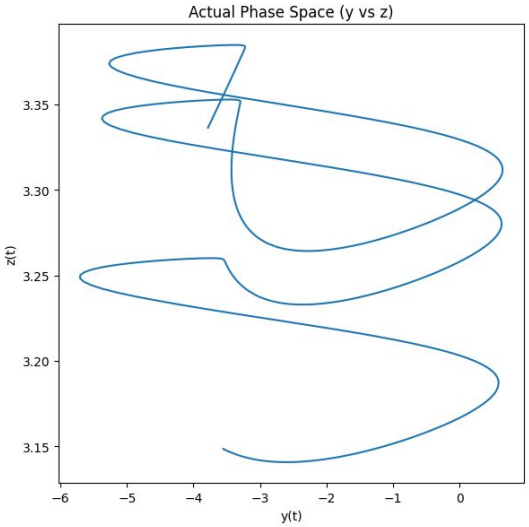




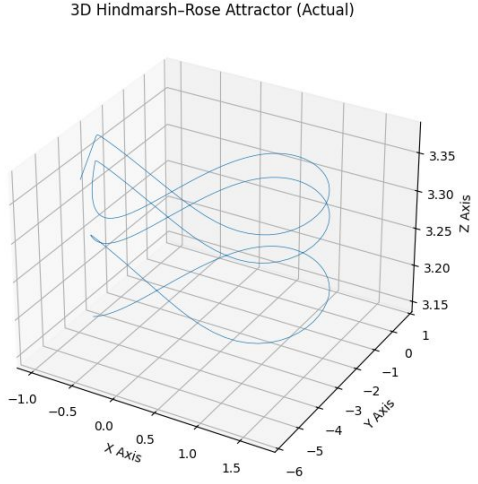


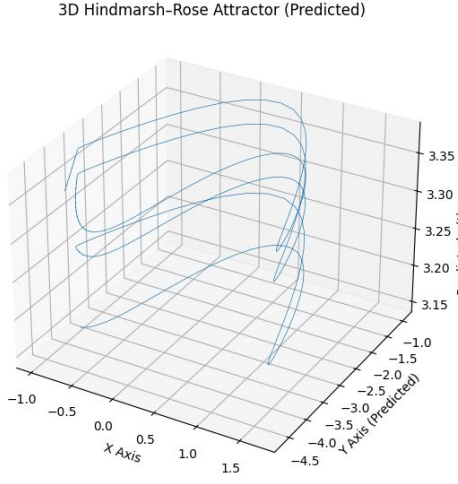


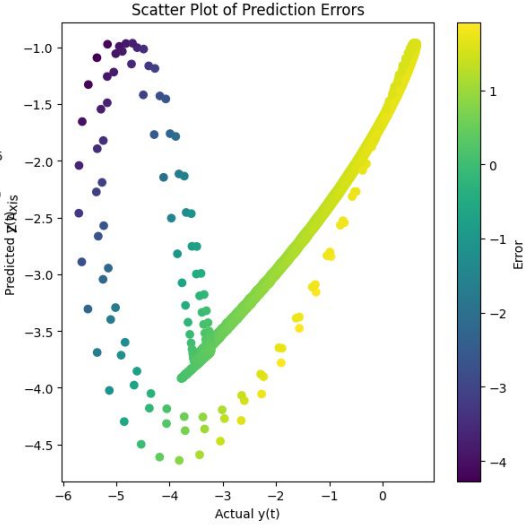




RMSE: 0.9536
MAE: 0.7465
R^2: 0.4089





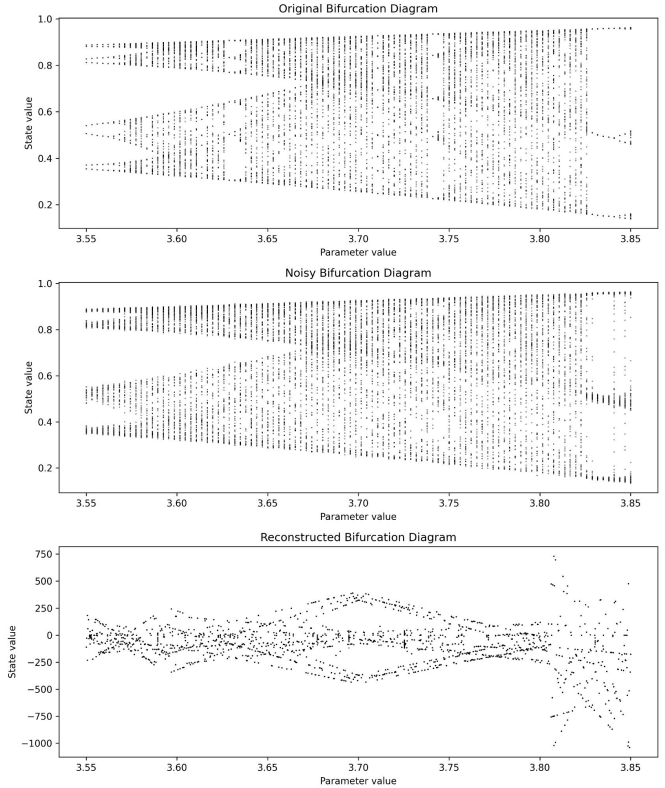


Experiment 3: Bifurcation Reconstruction via PCA and ELM

Goal: Reconstruct bifurcation diagram from time-series data generated at different parameter values (μ).

Methodology used:

- Generating noisy time-series data for logistic map across a range of μ.
- For each μ, train an ELM (like RC, fixed random hidden layer) to predict x_{t+1} from x_t . Collecting output weights W^{out}_{μ} .
- Applying PCA to the collected W^{out}_{μ} matrices to estimate the underlying parameter space.
- Using the trained ELMs (or a regressor trained on PCA features + μ) to generate attractor points across the estimated parameter range.



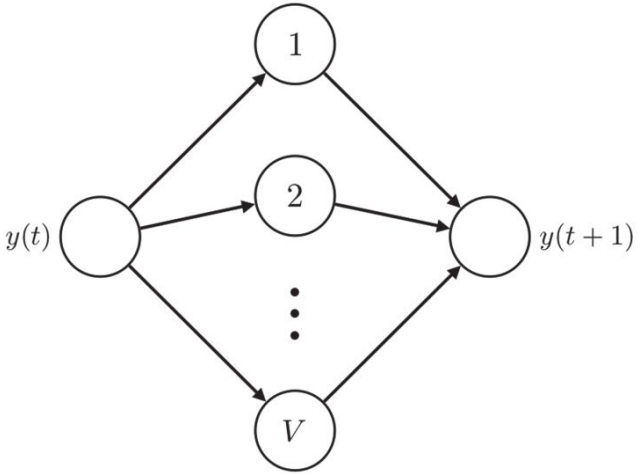


FIG. 1. Structure of the extreme learning machine used in the present study.

The logistic map is given by

$$x^{(l)}(t+1) = p^{(l)} x^{(l)}(t)\left(1 - x^{(l)}(t)\right), \tag{17}$$

where $p^{(l)}$ is the bifurcation parameter value of the logistic map. We use $p^{(l)}(n)$ as the value of $p^{(l)}$ to generate the time-series dataset S_n. Here, we set $p^{(l)}(n)$ as

$$p^{(l)}(n) = -0.15\cos(2\pi(n-1)/8) + 3.7 \quad (n = 1, 2, \ldots, P = 9). \tag{18}$$

The length of each time-series dataset is 1000, and we use an ELM that has four neurons in its hidden layer.

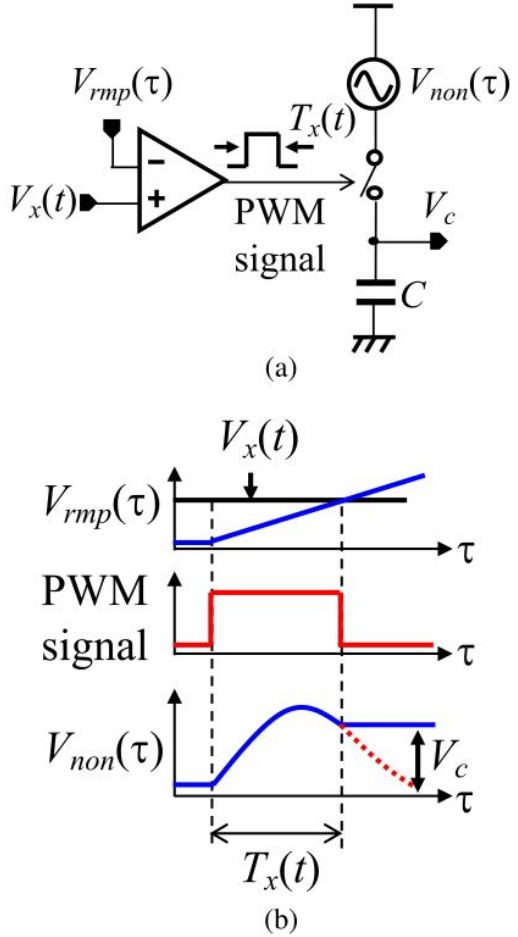
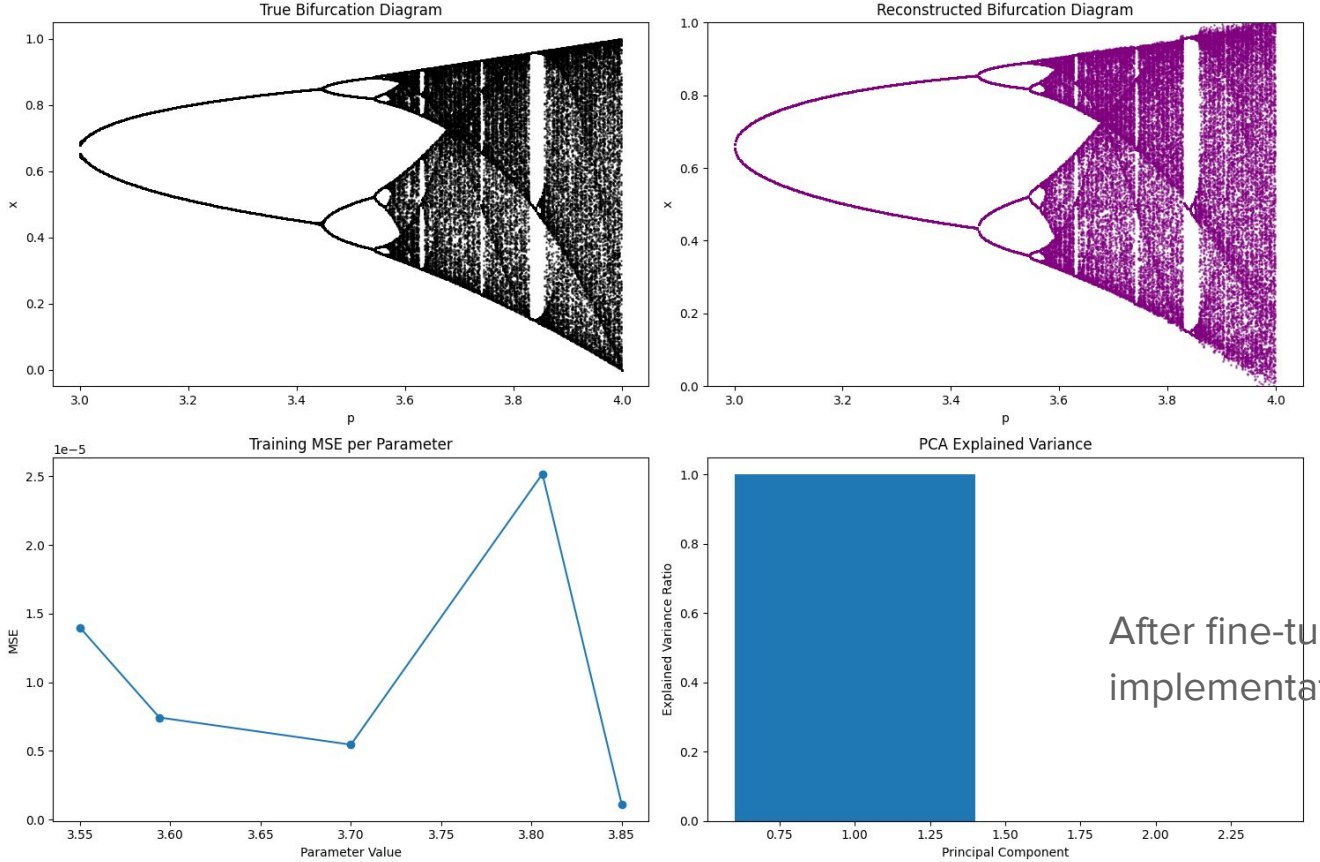
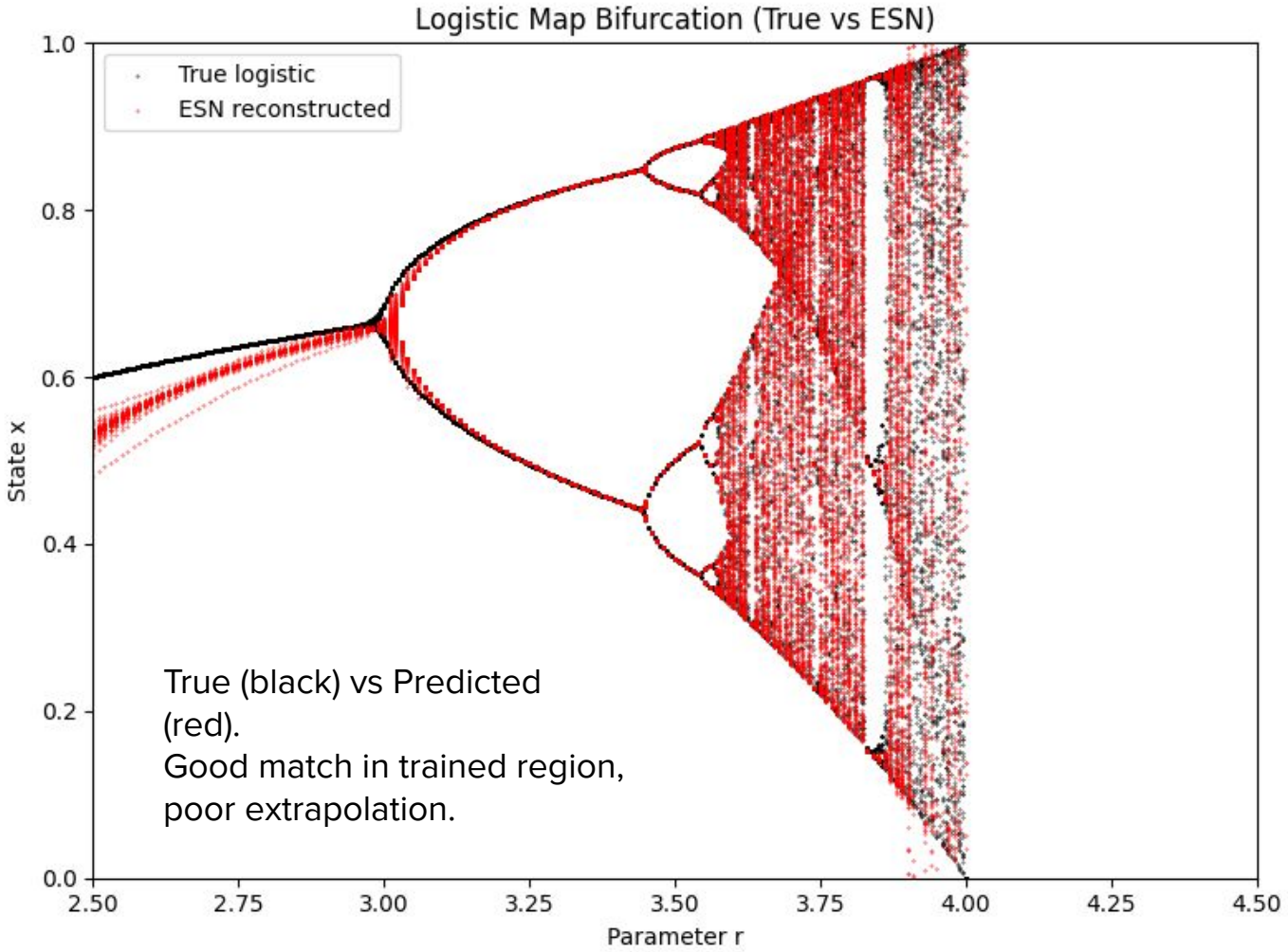


FIG. 2. Circuit principle of voltage waveform sampling. (a) Main circuit. (b) Timing diagram of nonlinear transformation.

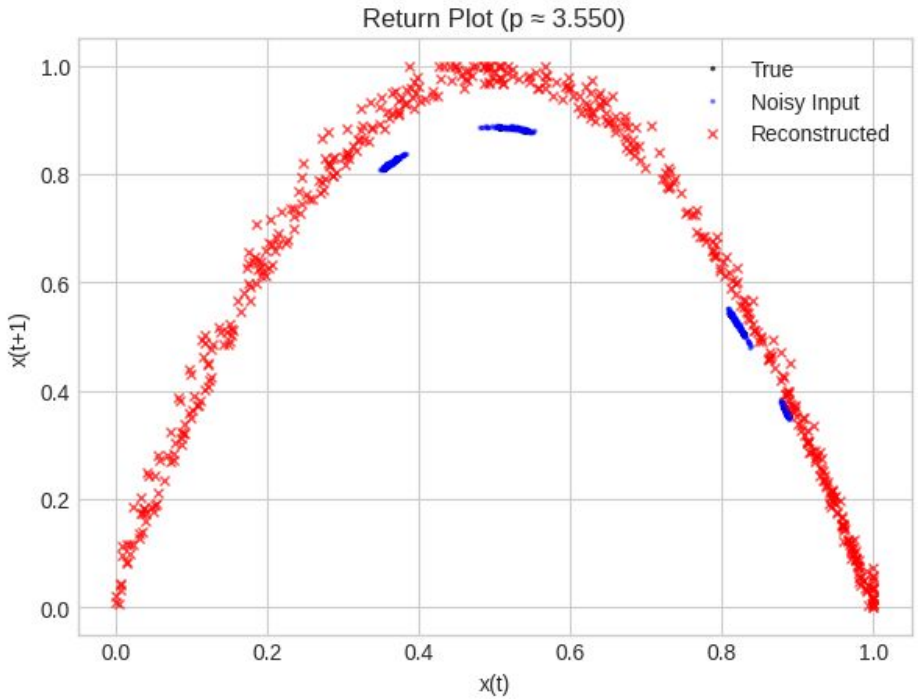
Improved Reconstruction Results (ELM + PCA)

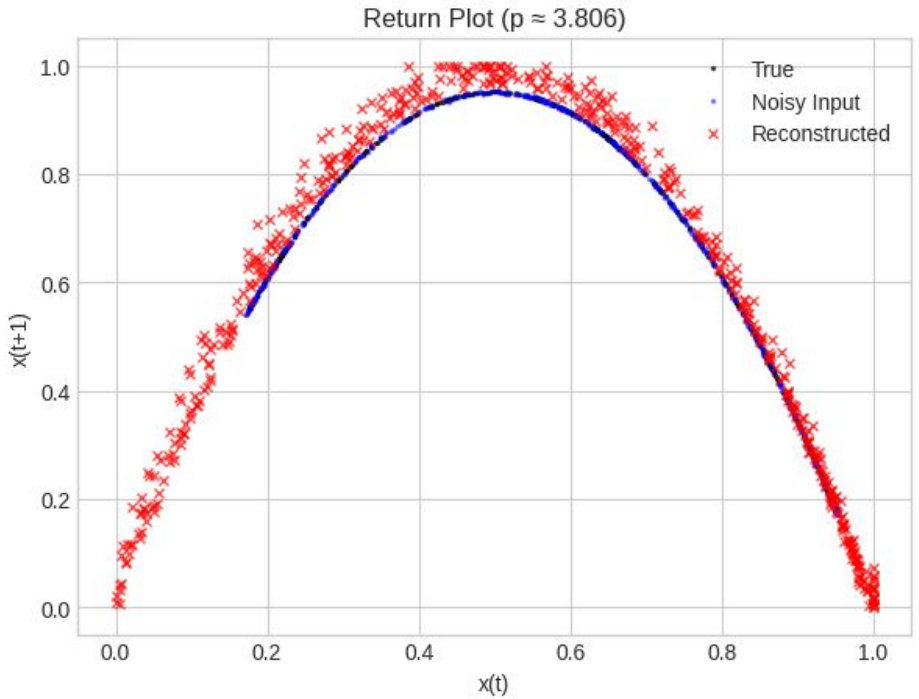


After fine-tuning and fixing implementation issues:


Logistic Map Bifurcation (True vs ESN)
True logistic
ESN reconstructed
State x
Parameter r
1.0
0.8
0.6
0.4
0.2
0.0
2.50
2.75
3.00
3.25
3.50
3.75
4.00
4.25
4.50
True (black) vs Predicted (red).
Good match in trained region, poor extrapolation.

Return Plot Comparison





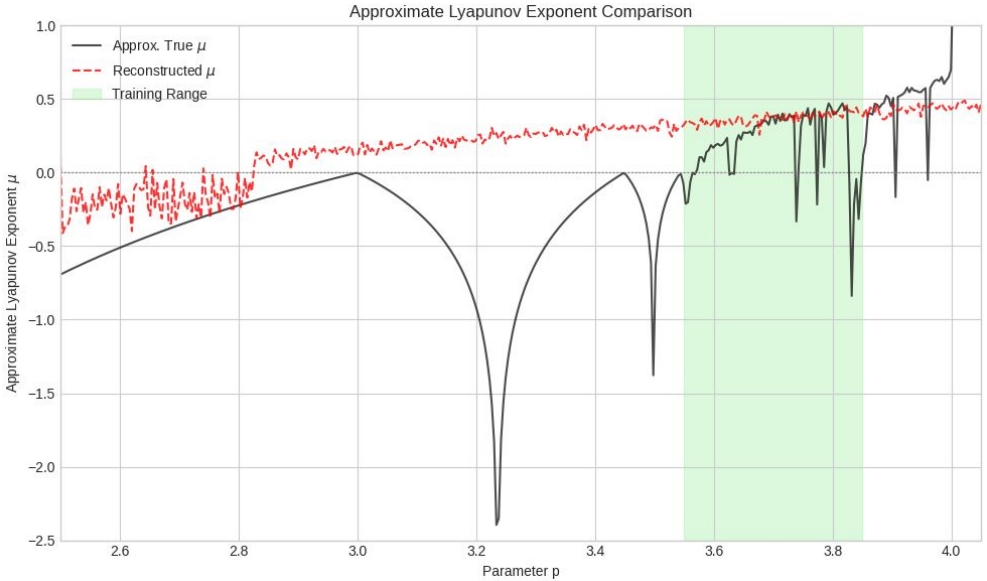
Reconstructed return plot matches true structure well.
But it does not understand where x_t is not a valid point.

Herein, we use two methods to estimate Lyapunov exponents.
We begin by explaining how to estimate Lyapunov exponents from time-series data. Although this method is relatively imprecise, it can be used to estimate Lyapunov exponents for BDs generated by electronic circuits.
Herein, we use the method proposed in 2012 by Yao et al.13 for estimating the largest Lyapunov exponent. In particular, this estimation method can be used with time-series data that are influenced by dynamical and observational noise. The other method involves estimating Lyapunov exponents from the reconstructed BD1 using the Jacobian matrix of the time-series predictor.
Consequently, the second method estimates Lyapunov exponents more precisely. However, because the present target systems are one-dimensional maps, instead of the Jacobian matrix we use the derivative of nonlinear function of the predictor.

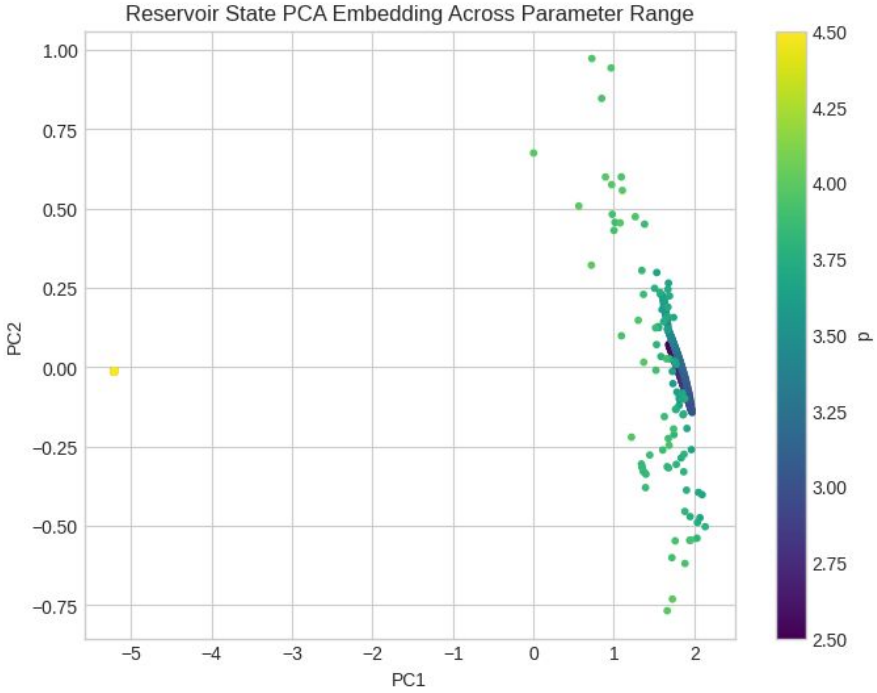
This method estimates Lyapunov exponents from time-series data by measuring the average rate at which distances between pairs of initially close points on nearby orbits change over time[1] [2] [3]. The process involves identifying pairs of points within a small neighborhood (ε) of a target point, calculating the mean distance between these pairs at successive time steps ($d^{(\epsilon)}(t)$ and $d^{(\epsilon)}(t+1)$), and repeating this calculation over a defined interval (ψ)[4]. The Lyapunov exponent (μ) is then estimated as the average of the logarithm of the ratio of these mean distances ($d^{(\epsilon)}(t+1)/d^{(\epsilon)}(t)$) over the interval; however, this specific method assigns an exponent of zero to periodic time series because the distances between points remain constant[5].

In this method, the Lyapunov exponent is obtained by

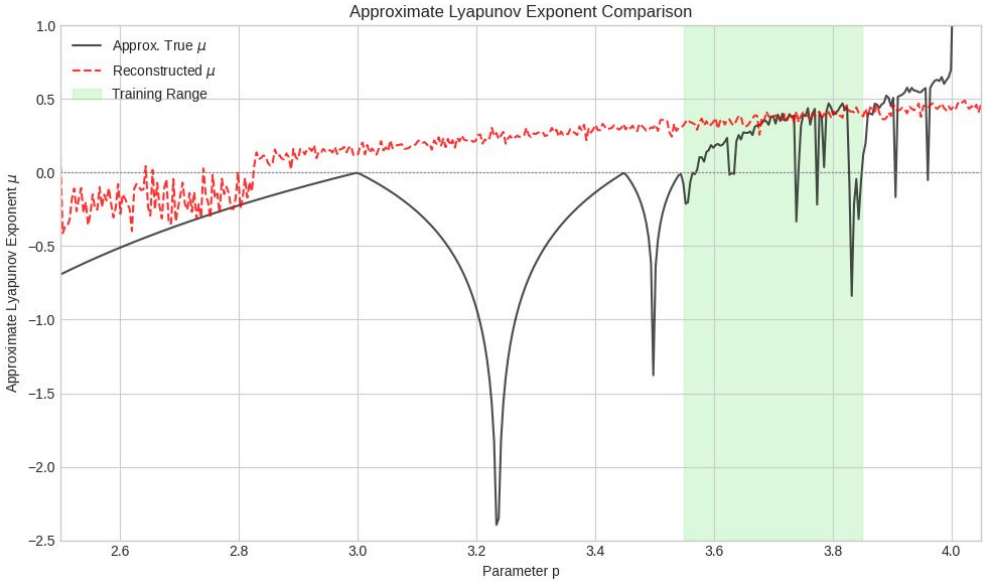
$$\mu = \frac{1}{\psi}\sum_{t=1}^{\psi}\frac{dy(t+1)}{dy(t)}.$$



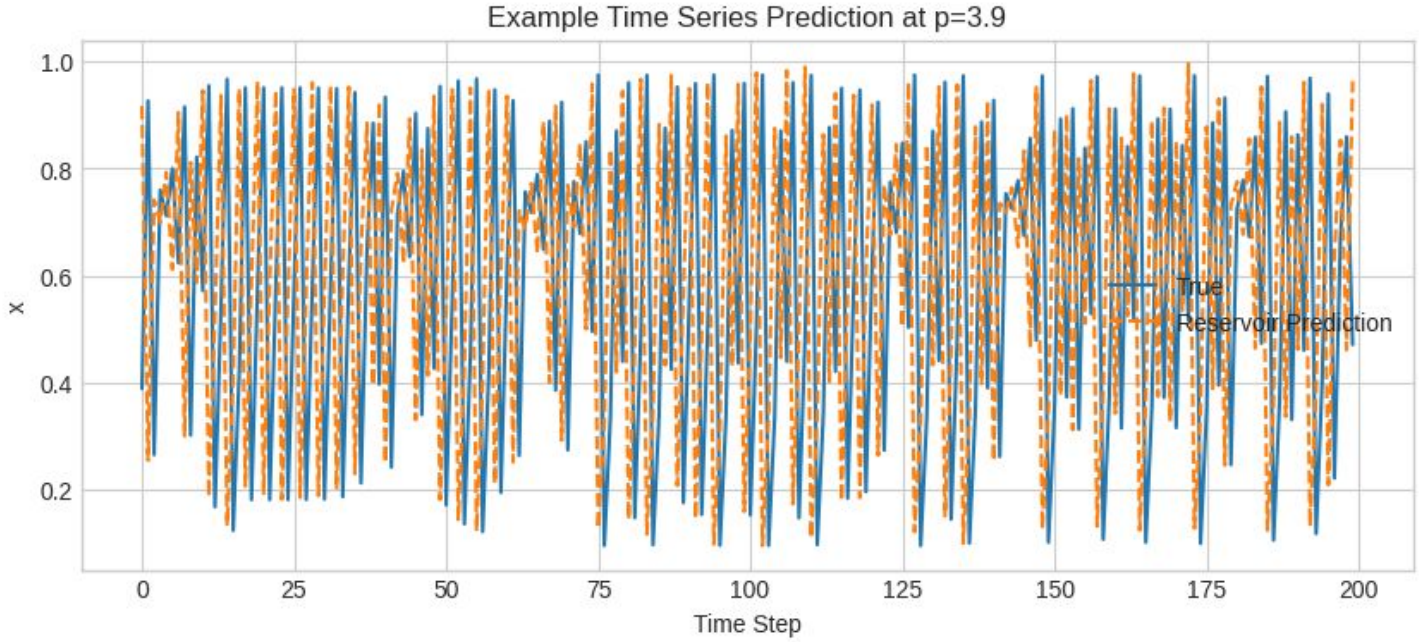
Estimating dynamics (positive = chaos) vs parameter. Green = training range.



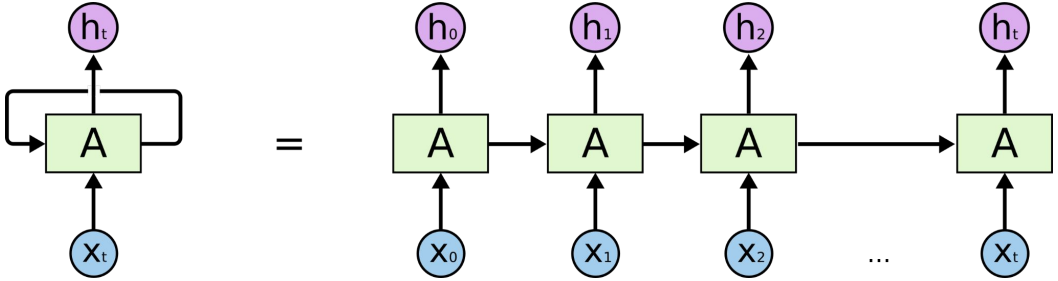
PCA components vs parameter show correlation, potentially identifying parameter space



Estimating dynamics (positive = chaos) vs parameter. Green = training range.


Example Time Series Prediction at p=3.9
1.0
0.8
0.6
0.4
0.2
x
0
25
50
75
100
125
150
175
200
Time Step
True
Reservoir Prediction

A detour on RNNs



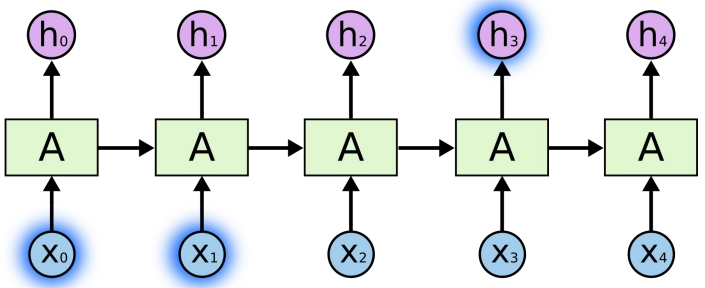
Humans don't start their thinking from scratch every second. As a human would read this slide, we understand each word based on your understanding of previous words. We don't throw everything away and start thinking from scratch again. Our thoughts have persistence.
Traditional neural networks can't do this, and it seems like a major shortcoming. For example, if I want to classify what kind of event is happening at every point in a book. It's unclear how a traditional neural network could use its reasoning about previous events in the book to inform later ones. Keeping track of what happened previously is a difficult task.

Recurrent neural networks address this issue. They are networks with loops in them, allowing information to persist.

A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor.

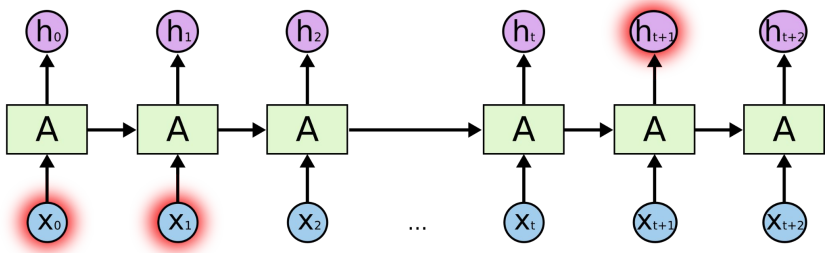
This chain-like nature reveals that recurrent neural networks are intimately related to sequences and lists. They're the natural architecture of neural network to use for such data.
And they certainly are used! In the last few years, there have been incredible success applying RNNs to a variety of problems: speech recognition, language modeling, translation, image captioning... Any sequential task can be solved in this way!

Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in "the clouds are in the sky," we don't need any further context – it's pretty obvious the next word is going to be sky. In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information.

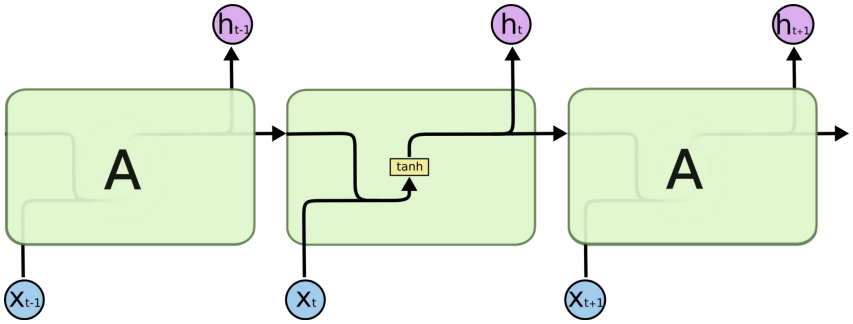


But there are also cases where we need more context. Consider trying to predict the last word in the text "I grew up in France... I speak fluent French." Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back. It's entirely possible for the gap between the relevant information and the point where it is needed to become very large.

Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.



In theory, RNNs are absolutely capable of handling such “long-term dependencies.” A human could carefully pick parameters for them to solve toy problems of this form. Sadly, in practice, RNNs don’t seem to be able to learn them. The problem was explored in depth by Hochreiter (1991) [German] and Bengio, et al. (1994), who found some pretty fundamental reasons why it might be difficult.

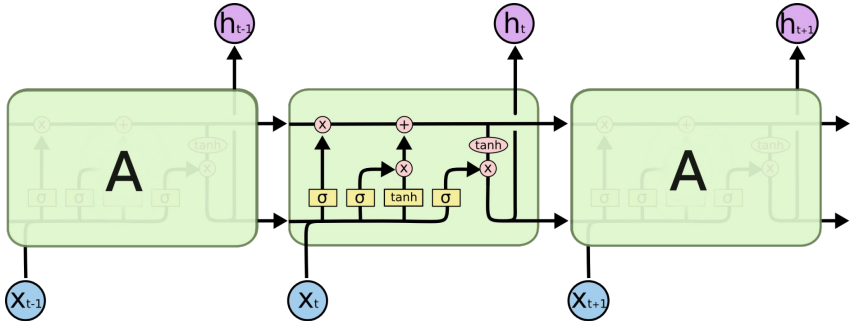


The repeating module in a standard RNN contains a single layer.

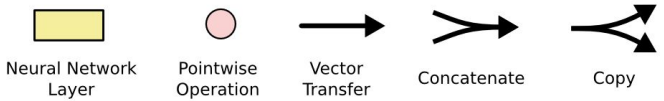
Thankfully, LSTMs don’t have this problem!

LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior.

LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four



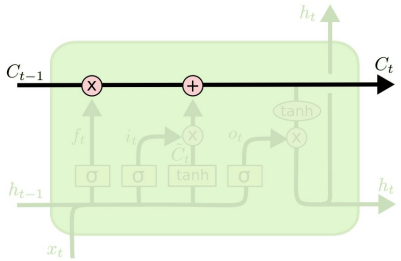
The repeating module in an LSTM contains four interacting layers.



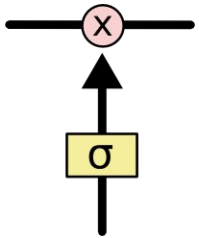
The Core Idea Behind LSTMs

The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.

The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.



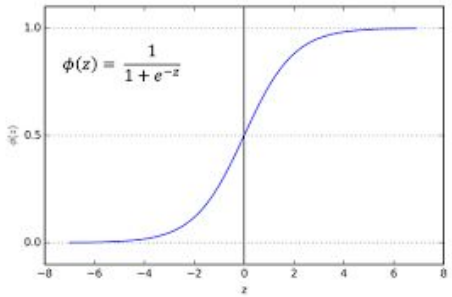
The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.
Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.

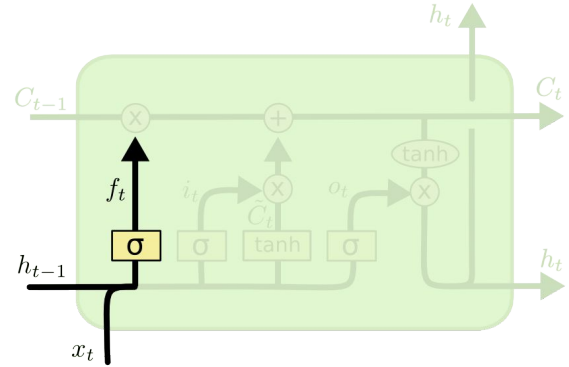


The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means "let nothing through," while a value of one means "let everything through!"

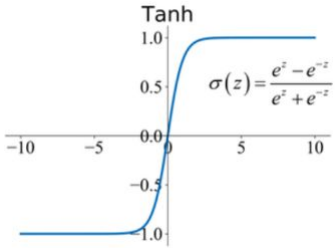
An LSTM has three of these gates, to protect and control the cell state.

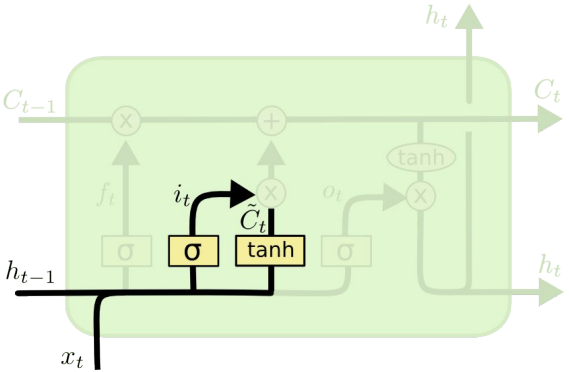
Briefly talk about activation functions





$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$





$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

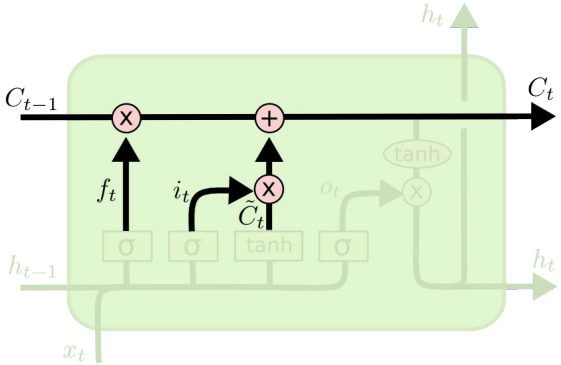
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Step 1: Forget Gate

- This step uses a sigmoid "forget gate layer" to examine the previous output (h_{t-1}) and the current input (x_t).
- It determines what information to discard from the previous cell state (C_{t-1}), outputting a 0 for removal or 1 for retention.

Step 2: Input Gate & New Information

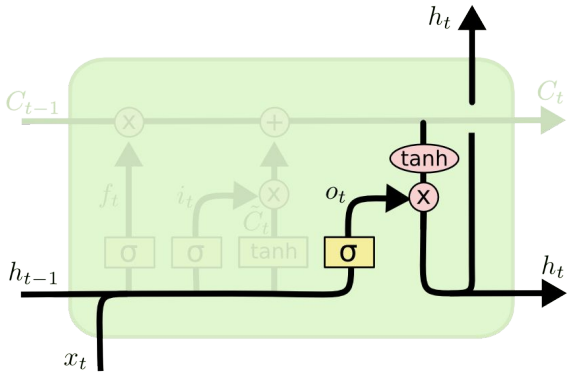
- A sigmoid "input gate layer" decides which values in the cell state will be updated.
- A tanh layer creates a vector of new candidate values ($\tilde{C}_t$) that could potentially be added to the cell state.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Step 3: Update Cell State

- The old cell state (C_{t-1}) is multiplied by the forget gate's output (f_t), effectively dropping the previously identified information.
- The result is added to the product of the input gate (it) and the new candidate values ($\tilde{C}_t$), forming the updated cell state (C_t).



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Step 4: Output Gate

- A sigmoid layer determines which parts of the newly updated cell state (C_t) should be included in the output.
- The cell state is passed through tanh (scaling values between -1 and 1) and multiplied by the sigmoid gate's output, producing the final filtered output (h_t).

Experiment 4: LSTM for Reconstruction

Problem: Initial RC/ELM approaches struggled to cleanly reconstruct bifurcation diagrams directly or extrapolate well. A single fixed reservoir might average dynamics across parameters.

Alternative Approach: Use a Long Short-Term Memory (LSTM) network.
Key Differences:

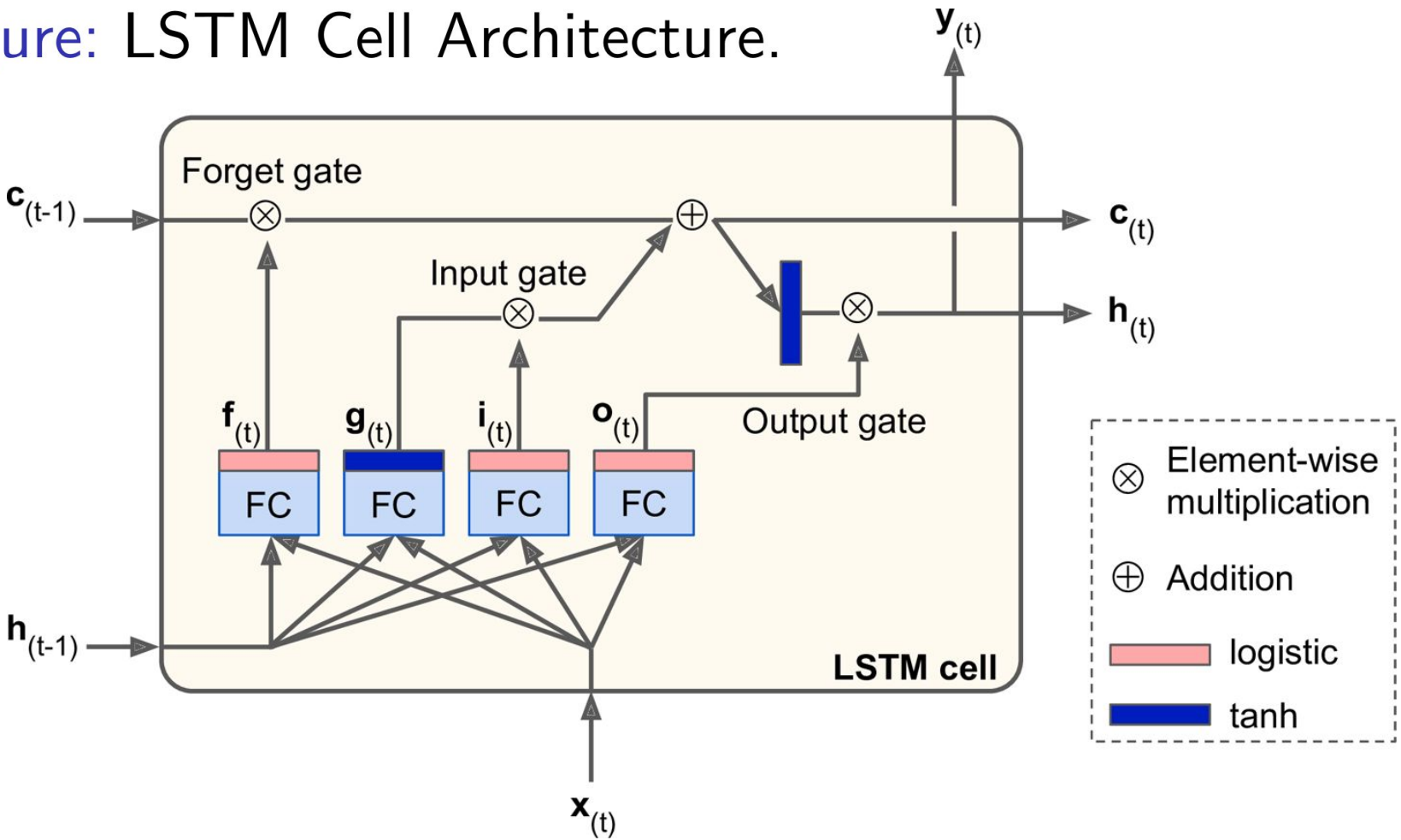
LSTM is fully trainable (incl. recurrent weights). Parameter p is given as explicit input along with state x_t .

Learns conditional dynamics:

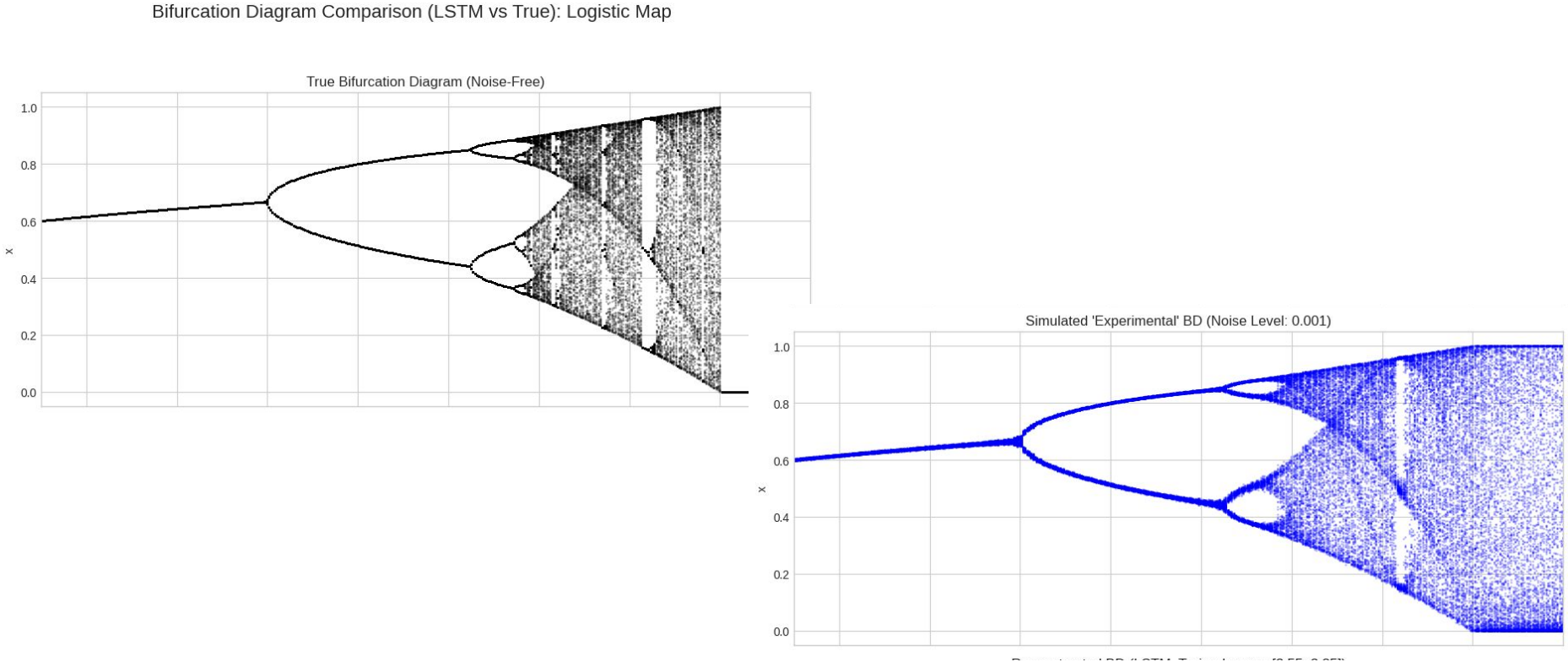
$$\hat{x}_{t+1} = g_{\mathrm{LSTM}}(x_t, p; \theta).$$

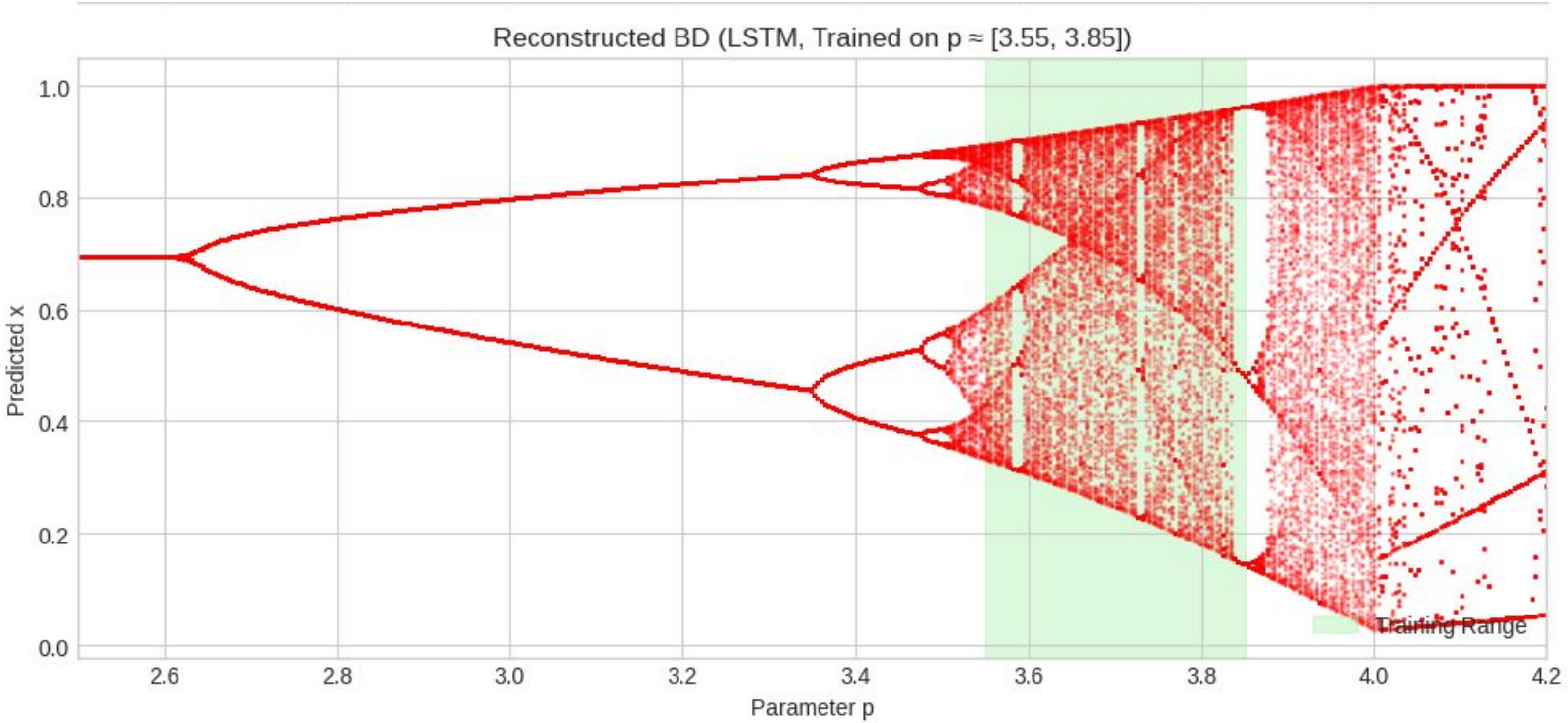
Trained on data from *all* parameters simultaneously

Figure: LSTM Cell Architecture.

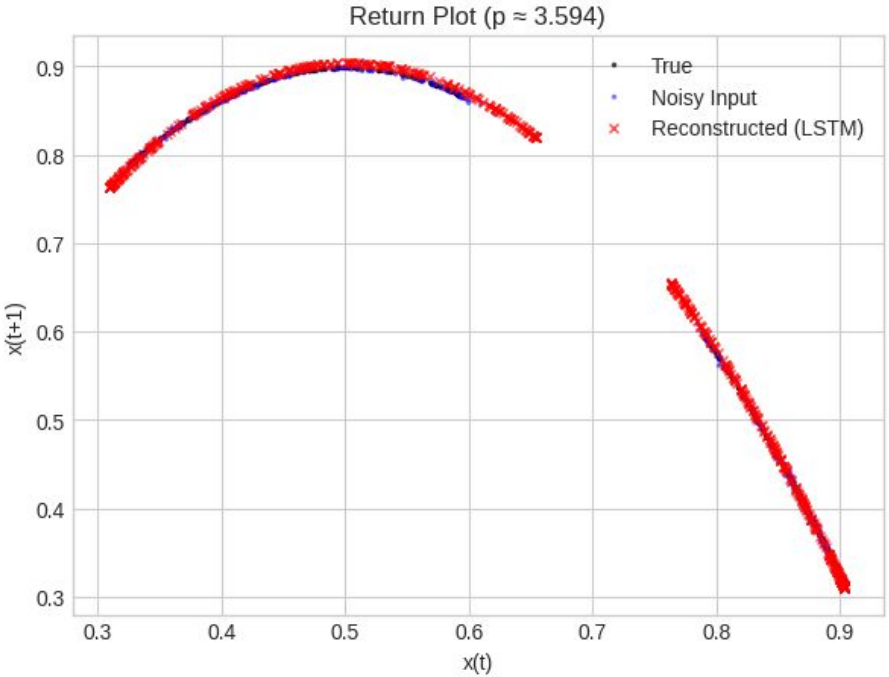


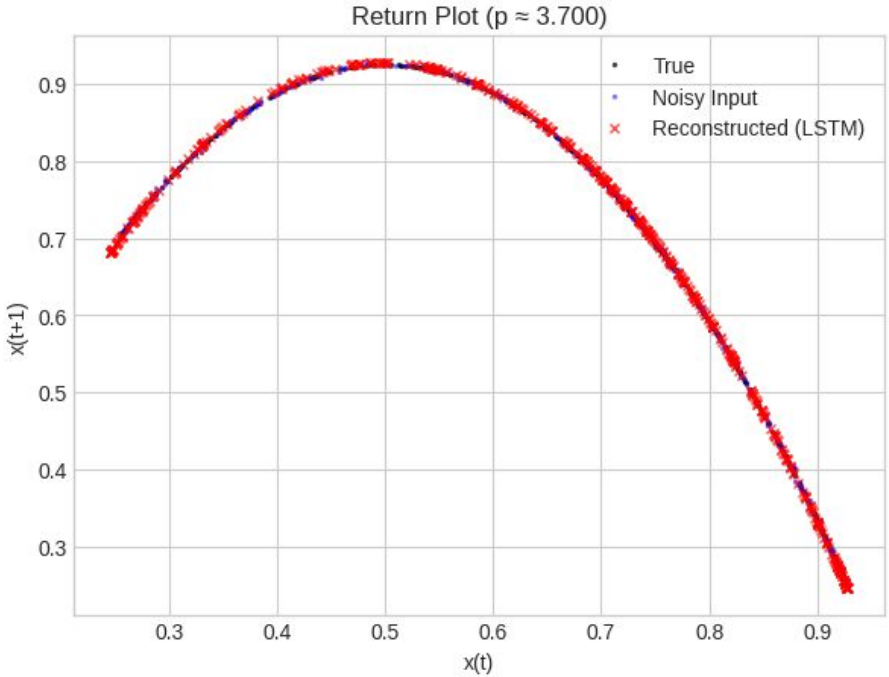
LSTM Reconstruction Results: Bifurcation Diagram

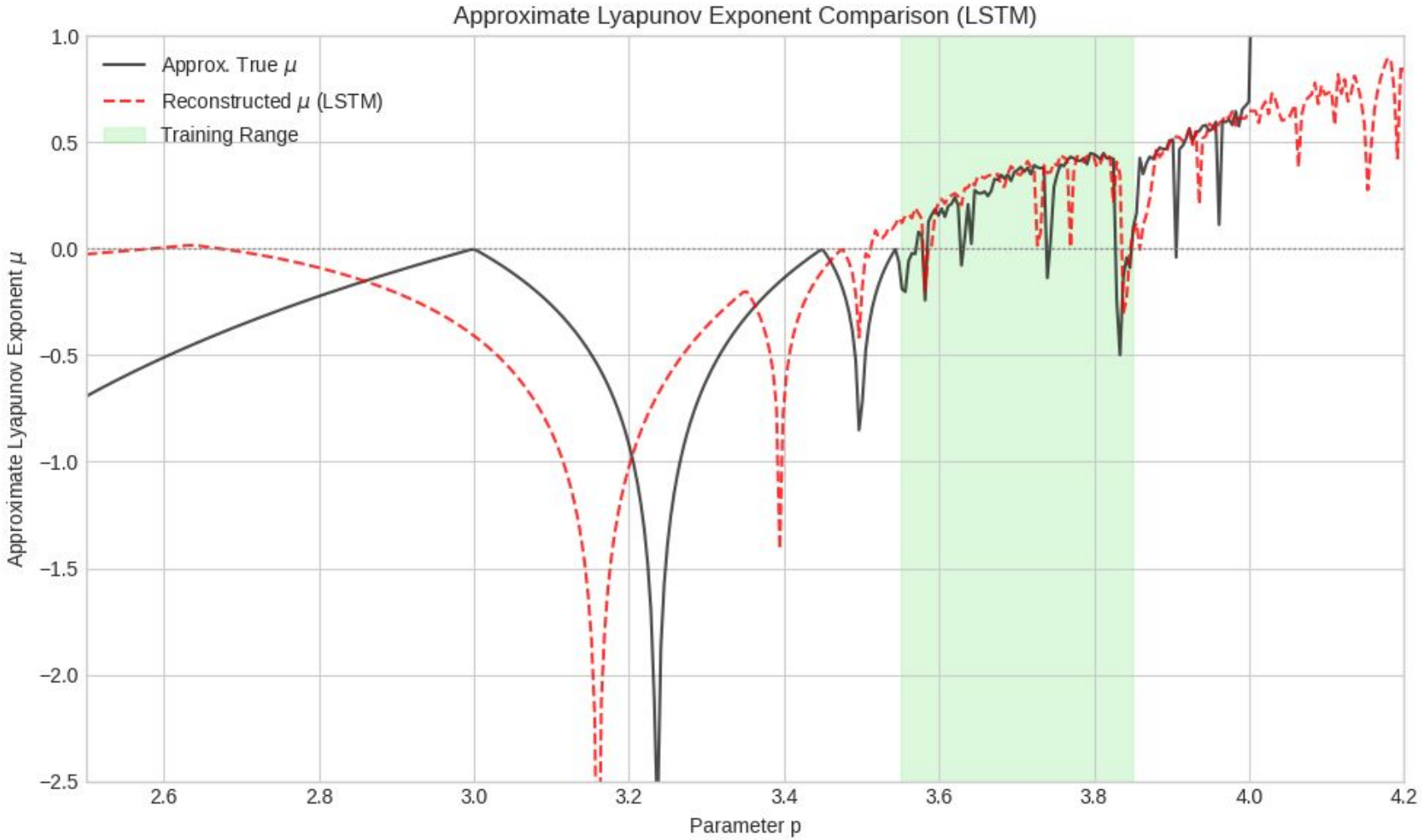



Reconstructed BD (LSTM, Trained on p ≈ [3.55, 3.85])
Predicted x
1.0
0.8
0.6
0.4
0.2
0.0
2.6
2.8
3.0
3.2
3.4
3.6
3.8
4.0
4.2
Parameter p
Training Range

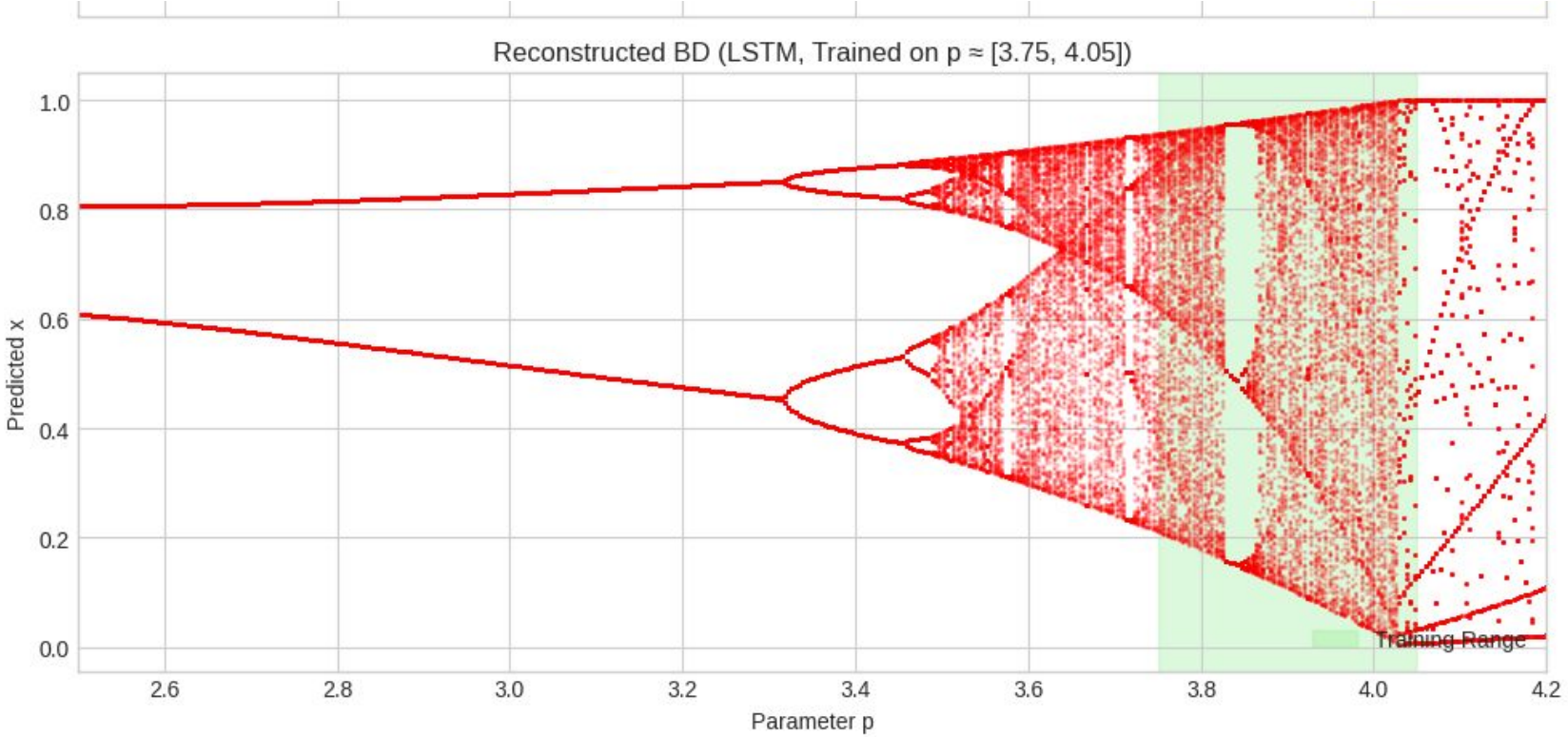
Return Plot Comparison (LSTM)

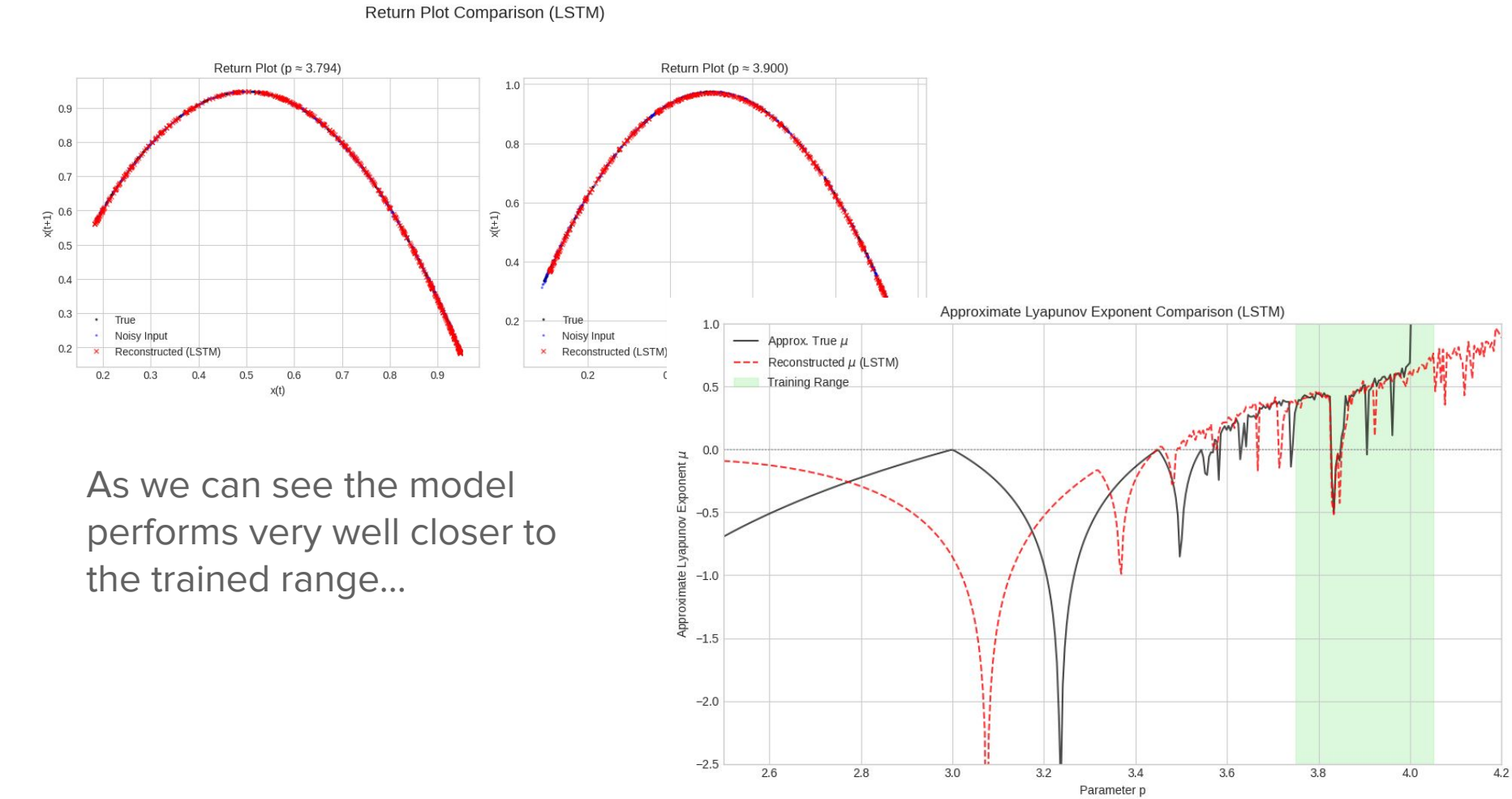





Approximate Lyapunov Exponent Comparison (LSTM)
Approx. True μ
Reconstructed μ (LSTM)
Training Range
Approximate Lyapunov Exponent μ
Parameter p
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
2.6
2.8
3.0
3.2
3.4
3.6
3.8
4.0
4.2

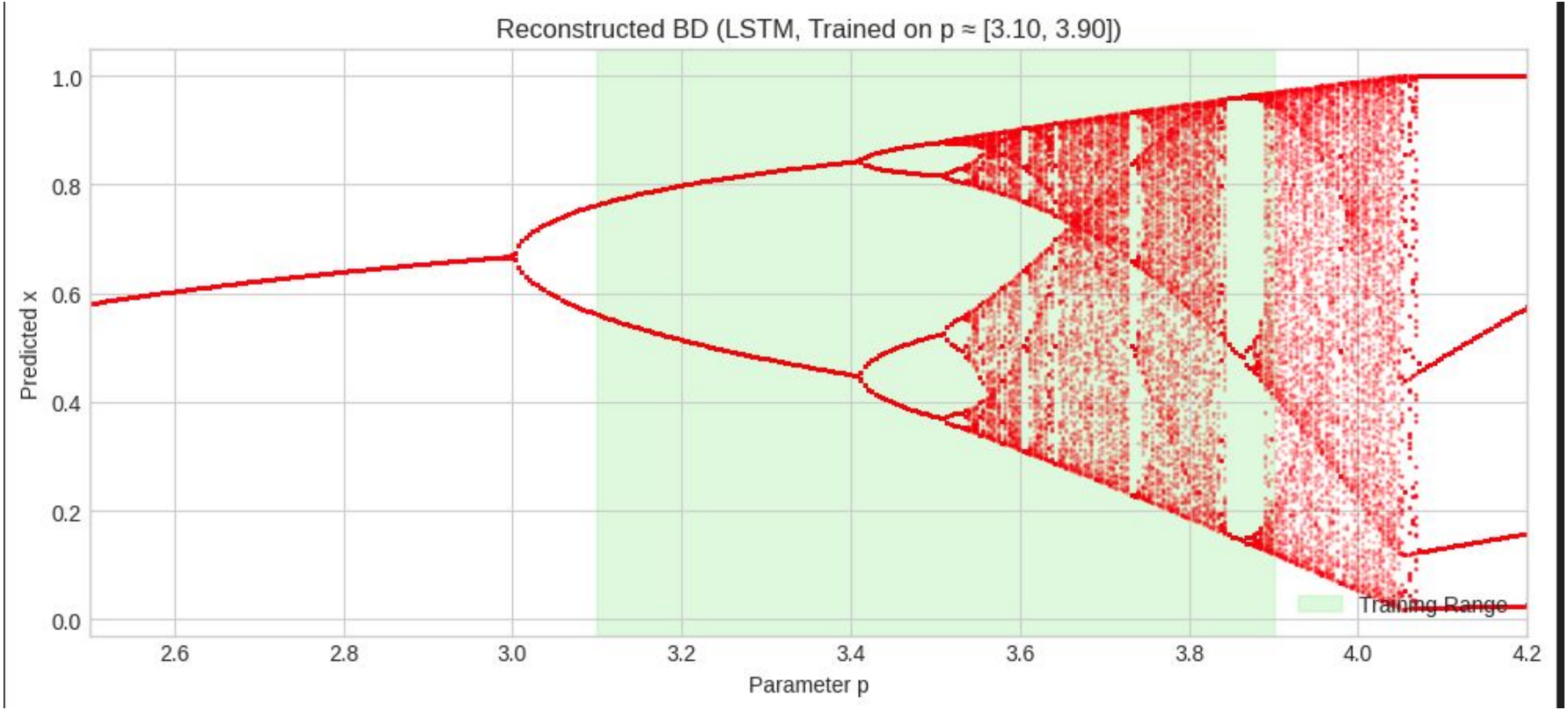
Let’s change the range a bit



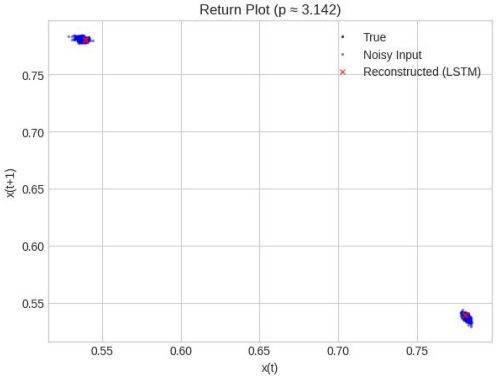


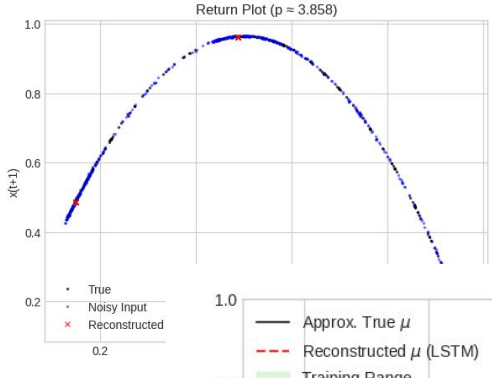
As we can see the model performs very well closer to the trained range...

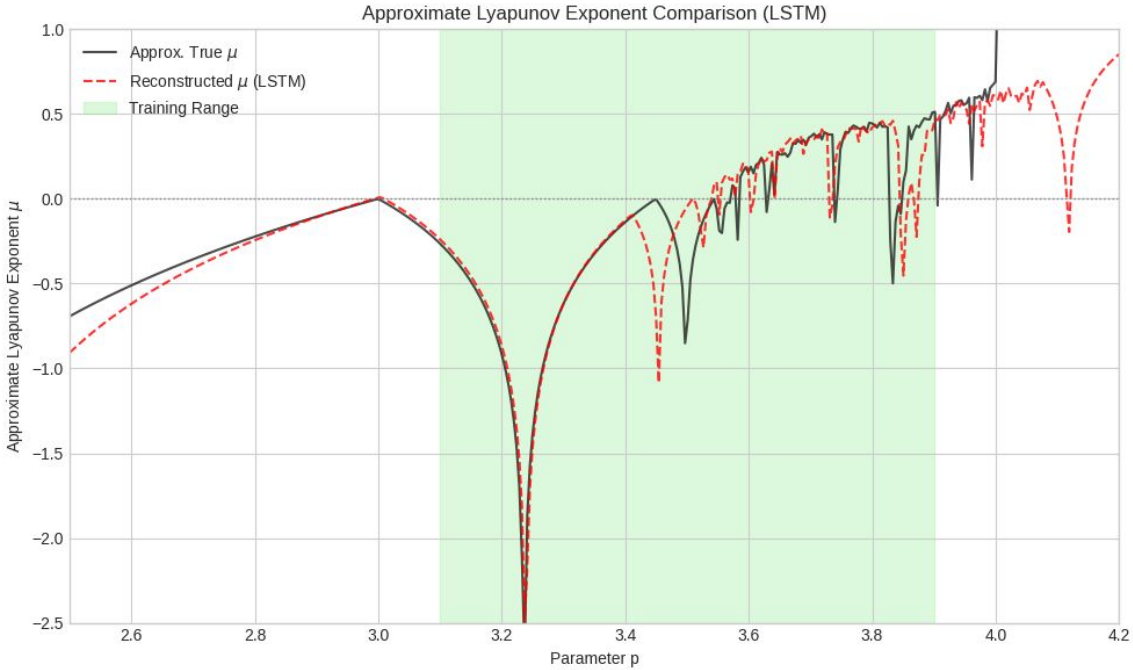
Further increasing training range



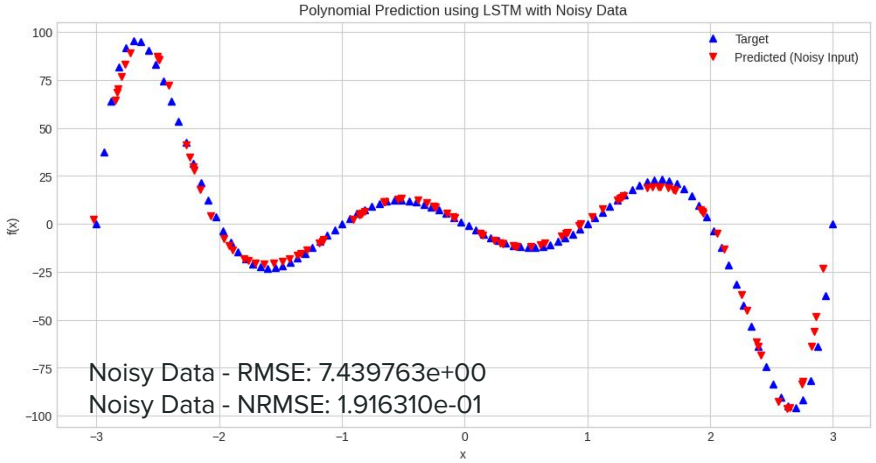
Return Plot Comparison (LSTM)

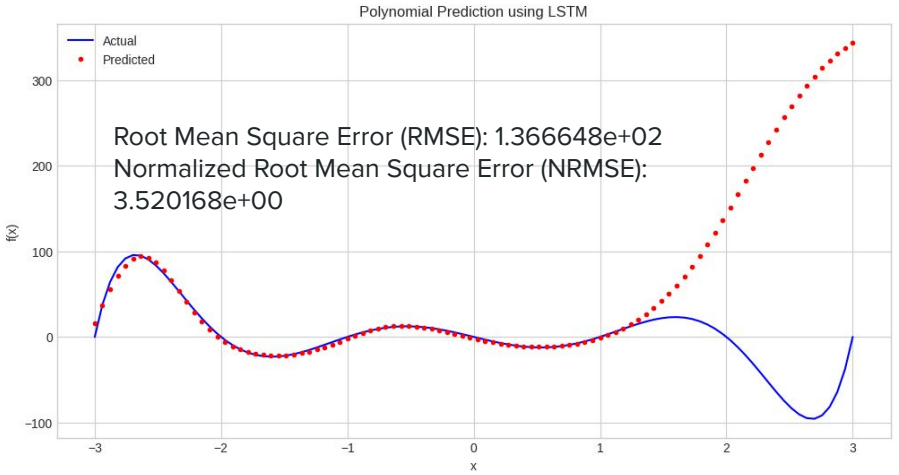


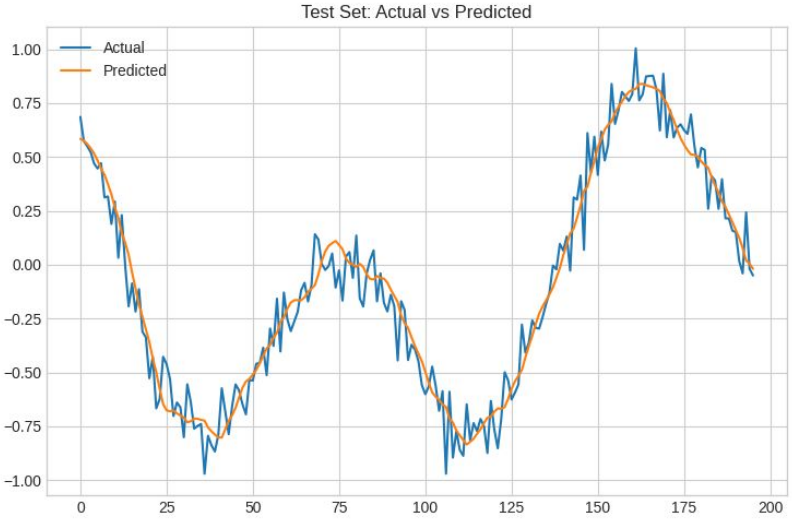


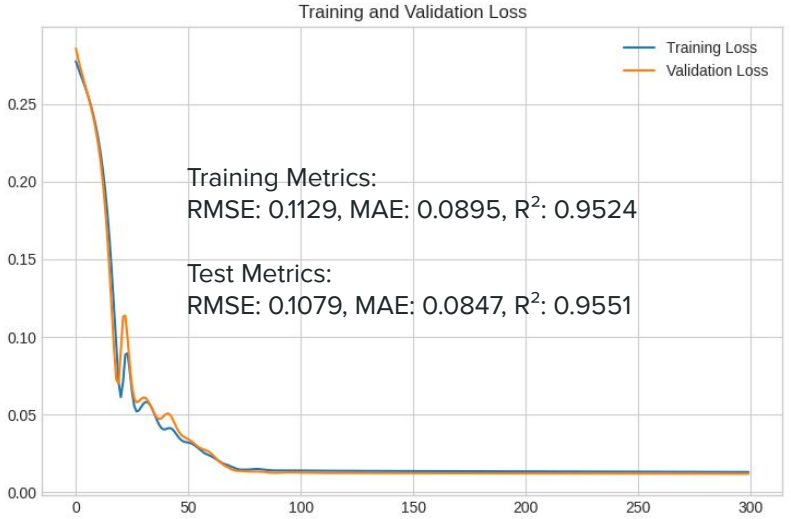


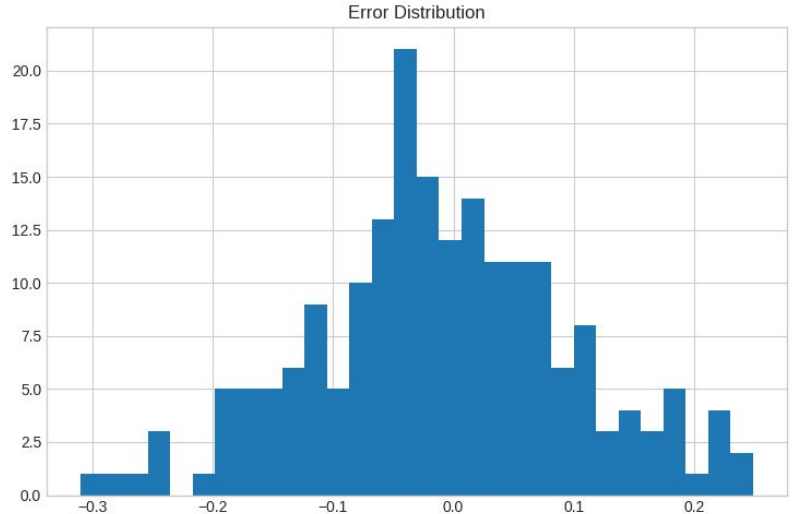
Using LSTM for Polynomial prediction

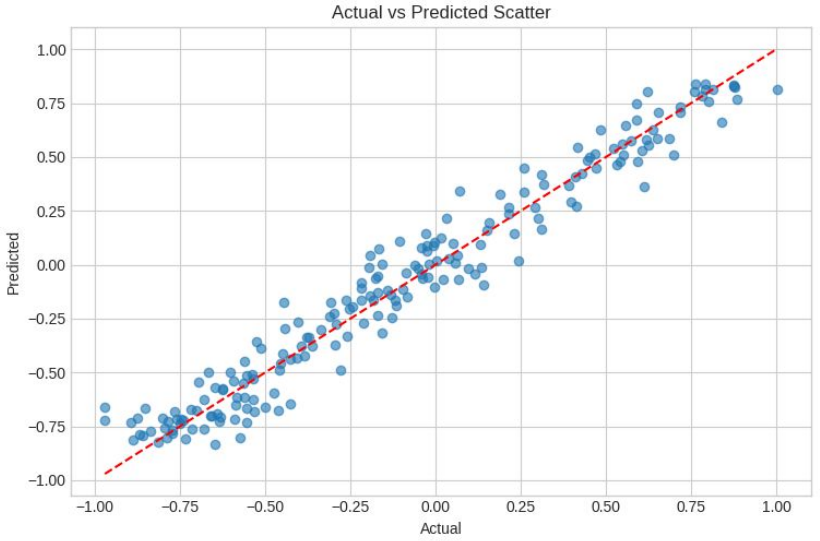


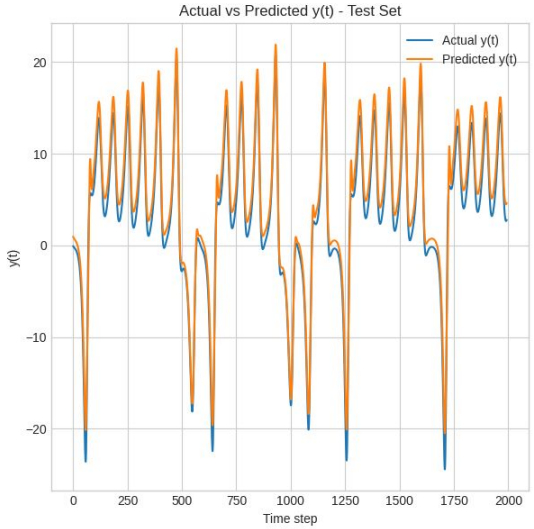


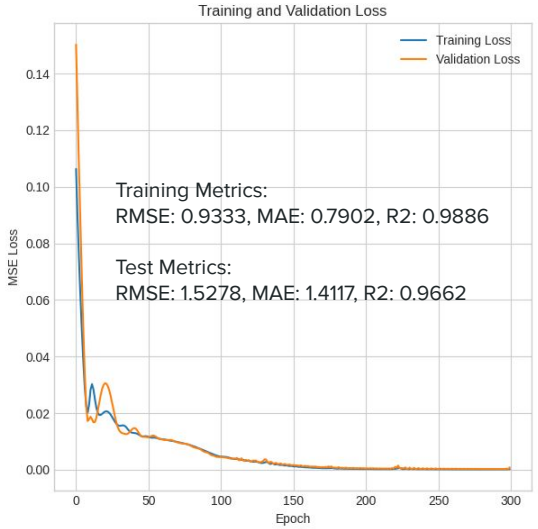

Test Set: Actual vs Predicted
Actual
Predicted


Training and Validation Loss
Training Loss
Validation Loss
Training Metrics:
RMSE: 0.1129, MAE: 0.0895, R²: 0.9524
Test Metrics:
RMSE: 0.1079, MAE: 0.0847, R²: 0.9551


Error Distribution

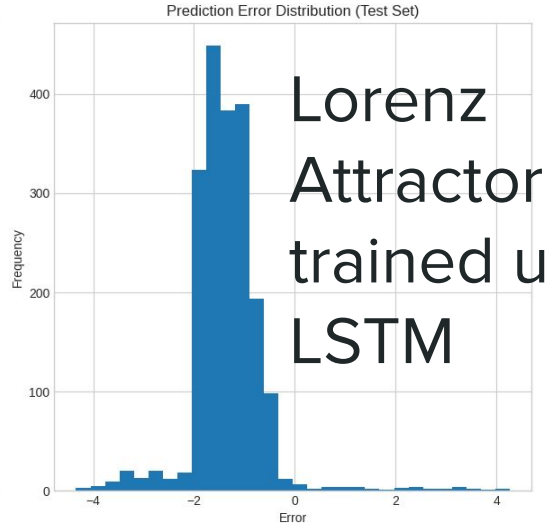

Actual vs Predicted Scatter
Predicted
Actual



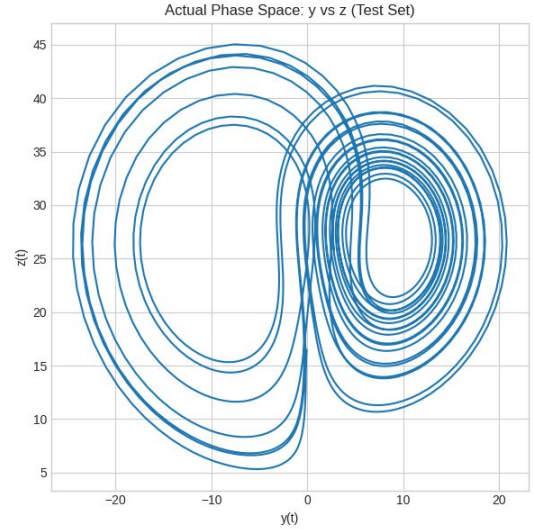


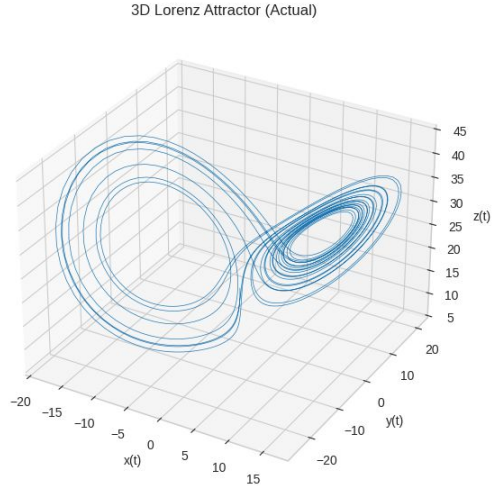
Training Metrics:
RMSE: 0.9333, MAE: 0.7902, R2: 0.9886

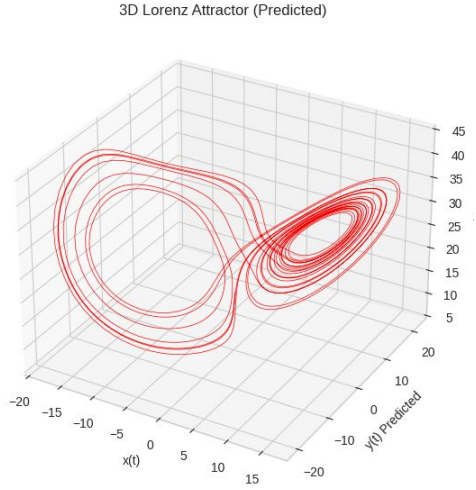
Test Metrics:
RMSE: 1.5278, MAE: 1.4117, R2: 0.9662



Lorenz Attractor trained using LSTM







RC/ELM vs LSTM: Training Time Trade-off

Performance Comparison for Bifurcation Reconstruction:

ELM (RC-like):

- Training Time: **< 1 second** (Linear regression is fast!)
- Reconstruction Quality: Good within training range, poor extrapolation, sensitive setup (PCA step).

LSTM (RNN):

- Training Time: **76 seconds** (Backpropagation takes time)
- Reconstruction Quality: Very good, including extrapolation, simpler setup (parameter as input).

In Summary: RC offers huge speed-up but might lack flexibility for complex conditional tasks compared to fully trained RNNs, especially when compute is available. Newer sequence models (Transformers, Mamba) might perform even better with temporal task based prediction.

Conclusion

- Explored Reservoir Computing (RC) for analyzing noisy time-series, focusing on bifurcation diagram reconstruction using ELM+PCA.
- Found that RC could capture broad dynamics (period doubling) but struggled with fine chaotic structures and extrapolation.
  - Highlighted challenges with noise, hyperparameter sensitivity.
- Gained insights into RC principles:
  - Massive training efficiency vs RNNs.
  - Potential for modeling physical systems.
- Fully trainable RNNs (like LSTM), explicitly using the parameter as input, showed superior performance for this specific reconstruction task, despite longer training time.
- Suggests that for complex conditional modeling where compute is less constrained, traditional RNNs (or newer architectures) might be more suitable than standard RC.

Future Work

- Optimising of RC parameters: Thorough hyperparameter tuning (size, spectral radius, regularization) potentially using Bayesian Optimization, etc.
- Broader Systems: Apply to higher-dimensional maps, continuous systems (ODEs), different bifurcation types (Hopf, saddle-node), coupled systems.
- Alternative Architectures: Explore deep reservoirs, different readout mechanisms.
- Physical Validation: Test reconstruction techniques on real experimental data (electronic circuits, pendulums) and real systems. Like using a motorized pendulum as a reservoir to do something. Almost instantaneous prediction - the original motivation. <Similarly using optics here>.

References

- R. Arun, M. S. Aravindh, A. Venkatesan, and M. Lakshmanan. Reservoir computing with logistic map. arXiv preprint arXiv:2401.09501, 2024.
- Matteo Cucchi, Steven Abreu, Giuseppe Ciccone, Daniel Brunner, and Hans Kleemann. Hands-on reservoir computing: a tutorial for practical implementation. Neuromorphic Computing and Engineering, 2, 08 2022. doi: 10.1088/2634-4386/ac7db7.
- Y. Itoh, S. Uenohara, M. Adachi, T. Morie, and K. Aihara. Reconstructing bifurcation diagrams only from time-series data generated by electronic circuits in discrete-time dynamical systems. Chaos, 30(1):013128, 2020. doi: 10.1063/1.5119187.
- Swarnendu Mandal, Sudeshna Sinha, and Manish Dev Shrimali. Machine-learning potential of a single pendulum. Physical Review E, 105:054203, 2022. doi: 10.1103/PhysRevE.105.054203.
- Yuanzhao Zhang and Sean Cornelius. Catch-22s of reservoir computing. Physical Review Research, 5, 09 2023. doi: 10.1103/PhysRevResearch.5.033213.

Acknowledgement

I would like to thank Prof. Gaurav Dar for his guidance, insights and motivating me throughout this project.

The code for this project is available on Github:
https://github.com/vimarsh244/ResorvoirComputing_SOP

Thank You

Questions?