Comparative Analysis of Multi-Layer Memristor-Based Echo-State Reservoir Network and Traditional Recurrent Neural Network for Chaotic Time Series Prediction

William Norden
Department of Electrical and Computer Engineering
Santa Clara University
Email: wnorden@scu.edu

***Abstract*—This paper presents a systematic comparison between two distinct approaches for forecasting chaotic time-series data generated by the Lorenz system: (1) a novel multi-layer *memristive reservoir* (echo-state) network and (2) a traditional stateful recurrent neural network (RNN). Despite enforcing a smaller *trainable*-parameter budget in the reservoir-based model, we can observe that the memristive reservoir demonstrates superior accuracy and robustness on one-step-ahead prediction of Lorenz trajectories. Detailed discussion is provided on the architectural design and training protocol of both models, specifically highlighting the use of backpropagation through time (BPTT) for the RNN [1], [3] and a ridge-regression-based training for the reservoir readout layer [4]. In addition, theoretical and practical reasons for the reservoir's advantages are discussed, including the echo-state property [5], [7], high-dimensional state expansion [8], and simplified training. Finally, CPU runtime and usage statistics are discussed. These findings reinforce the potential of memristive reservoir designs in delivering accurate, low-power, and efficient time-series forecasts, particularly under constrained parameter budgets [9], [11], [12].**

***Index Terms*—Reservoir Computing, Memristor, Recurrent Neural Networks, Chaotic Time Series, Lorenz System, Neuromorphic Hardware**

I. Introduction

Forecasting chaotic time-series data is crucial for a variety of applications, including climate modeling [13], fluid dynamics [14], and financial analysis [15]. The Lorenz system is a canonical example of a chaotic process, widely used to benchmark forecasting algorithms due to its sensitive dependence on initial conditions and rich dynamics [16].

Traditional methods for time-series forecasting frequently employ Recurrent Neural Networks (RNNs) such as Elman networks, LSTMs, or GRUs [18], [19]. Although effective, RNNs can be challenging to train due to their recurrent structure, and can suffer from vanishing or exploding gradients [1]. In contrast, *reservoir computing*, particularly Echo-State Networks (ESNs), offers a paradigm in which most network weights remain fixed, and only a readout layer is trained, mitigating many training difficulties [5], [7], [8].

Recent advances in neuromorphic hardware have motivated the use of *memristive* devices to implement physical reservoir systems [9]–[11]. These memristor-based reservoirs can, in principle, exploit the intrinsic dynamics of nanodevices to vastly reduce power consumption and chip area, enabling real-time, efficient implementations of forecasting and other computing systems [12].

In this paper, I present a multi-layer memristive reservoir network for Lorenz time-series forecasting and compare it against a fully trained RNN under similar (or smaller) *trainable* parameter budgets. Introduced are:

- A novel, multi-layer memristor-based reservoir design tailored for the Lorenz system.
- A direct comparison to a traditional RNN baseline trained via BPTT.
- Comparison of parameter efficiency, accuracy, architecture, training procedure, and cpu runtime/usage between the two models.

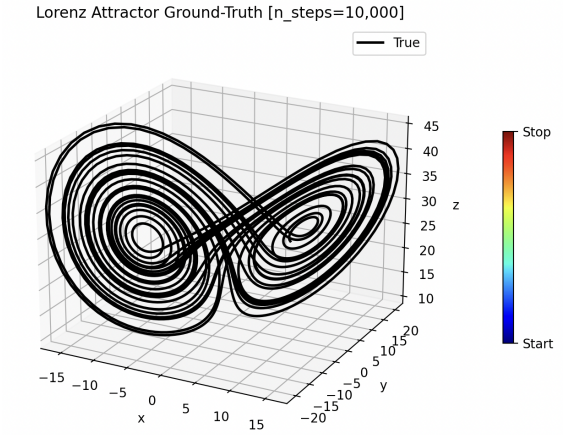

Fig. 1: $\mathbb{R}^3$ Lorenz Attractor (Ground-Truth): This is the parametric curve that the models are trained to fit to. Based on the number (n_steps) of points included in parametrization, this curve can be extended and complexified to a higher or lower degree. (Start-Stop legend to be contextualized in a moment).

II. Background

Reservoir computing, pioneered by Jaeger [5] and Maass et al. [8], exploits high-dimensional, randomly connected recurrent networks (e.g., Echo-State Networks and Liquid State Machines) for efficient temporal processing. In this framework, only the output (readout) layer is trained, which greatly simplifies learning and avoids difficulties such as vanishing gradients.

Memristors play a crucial role in emerging neuromorphic hardware. Initially postulated by Chua [2] to complete the symmetry of passive circuit elements (where resistors, capacitors, and inductors satisfy

$$v = iR, \quad q = Cv, \quad \varphi = Li,$$

respectively), the missing relation between charge and magnetic flux is provided by the memristor:

$$d\varphi = M(q)\, dq \quad \Longleftrightarrow \quad v(t) = M\big(q(t)\big) i(t).$$

Later, Strukov et al. [10] popularized physical memristors, showing that these devices—whose conductance depends on the history of charge flow—can indeed store analog states. Research has demonstrated that memristors can implement synaptic or recurrent connections in neuromorphic designs [9], [11].

Moreover, reservoir computing has been successfully applied to chaotic time-series prediction (e.g., Lorenz, Rössler systems), where it often outperforms fully trained recurrent neural networks, especially in terms of long-horizon stability [21], [22]. By integrating memristive devices into reservoir architectures, one leverages their inherent state-dependent dynamics for low-power, efficient, and robust temporal processing.

However, direct comparisons between *multi-layer* memristive reservoir networks and traditional RNNs under matched conditions remain less explored. This study aims to bridge that gap.

III. Methods

A. Lorenz System Data Generation

The Lorenz system is defined by (see [16]):

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y - x), \\ \frac{dy}{dt} &= x(\rho - z) - y, \\ \frac{dz}{dt} &= xy - \beta z, \end{aligned} \tag{1}$$

with constants $\sigma = 10$, $\rho = 28$, and $\beta = \frac{8}{3}$. I implemented a simple Euler solver with step size $\Delta t = 0.01$ to generate a time series starting from the initial state $(1.0, 1.0, 1.0)$ [17].

1) Physical Significance of the Parameter Choices: In his study, Lorenz chose the specific parameter values to reflect key physical properties:

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

These values were chosen because they place the system in a regime where the transition from regular to chaotic behavior is clearly observed. For lower values of ρ, the system typically exhibits stable, predictable behavior, but when $\rho = 28$, the dynamics become irregular and aperiodic, illustrating the onset of chaos.

2) Dataset Sizes & Splits in This Work: Throughout this paper, sequences of 10,000 points are generated for initial comparisons and 25,000 points for further demonstration of the reservoir-based model's capability. The models split these points as follows:

- **RNN Implementation:** The first 7,000 points are used for training, and the remaining 3,000 for testing.
- **Reservoir Implementation, 10,000 data points:** The first 7,000 points (minus an initial 200-step warm-up) are used for training, and the remaining ∼3,000 points are used for testing.
- **Reservoir Implementation, 25,000 data points:** The first 17,000 points (minus an initial 200-step warm-up) are used for training, and the remaining ∼8,000 points are used for testing.

In both cases, the task is one-step-ahead prediction of $\big(x_{t+1}, y_{t+1}, z_{t+1}\big)$ given $\big(x_t, y_t, z_t\big)$ as a function of time t.

B. Multi-Layer Memristive Reservoir Network

1) Echo-State Networks (ESN):

a) Concept of Echo-State Networks: Echo-State Networks are a special type of recurrent neural network in which only the output weights are trained. The fixed, randomly initialized reservoir (with its recurrent connections) naturally transforms the input into a high-dimensional dynamical system. This characteristic enables the ESN to capture complex temporal dependencies without the need for iterative weight updates during training. Figure 2 below illustrates the architecture of a traditional Echo-State network implementation.

A standard update rule for an Echo-State Network can be expressed as (see [5] and [7]):

$$\mathbf{s}(t+1) = f\big(W\,\mathbf{s}(t) \;+\; W^{\text{in}}\,\mathbf{u}(t)\big), \tag{2}$$

where $\mathbf{u}(t)$ is the input, $\mathbf{s}(t)$ is the reservoir state, and f is a nonlinear activation (often $\tanh$). The reservoir weights W and input weights W^{in} are typically randomized and remain fixed after initialization, provided the *spectral radius* of W is below 1 to ensure the echo-state property [5], [7].

2) Memristive Reservoir Concept: In hardware, memristive crossbars can realize these random, fixed-weight connections by programming device conductances [9], [11]. Our approach simulates this concept in software:

- Initialize sparse, random recurrent matrices W_i (one per layer) with a target connection density.

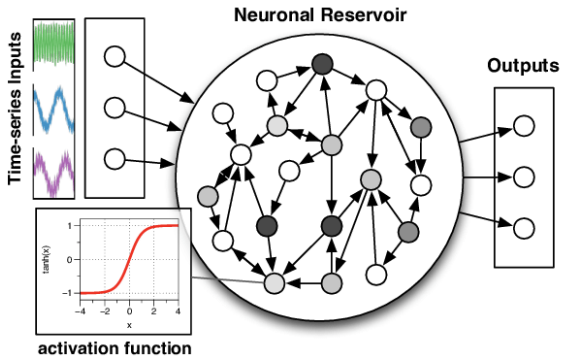

Fig. 2: Architecture of an Echo-State Network showing the fixed reservoir and the trainable output layer (figure from [6]).

- Scale each matrix so its largest eigenvalue (spectral radius) is below a chosen threshold, e.g., $\rho < 1$.
- Keep the matrices fixed and only train the final readout.

3) Proposed Multi-Layer Setup: I stack L reservoir layers. For each reservoir layer i, the input and state update with leaky integration is given by (adapted from standard ESN formulations, e.g., [5], [7]):

$$\mathbf{z}_i(t) = W_i\,\mathbf{s}_i(t-1) + W_i^{\text{in}}\,\mathbf{u}_i(t), \tag{3}$$

$$\mathbf{s}_i(t) = (1-\alpha_i)\,\mathbf{s}_i(t-1) \;+\; \alpha_i\,\tanh(\mathbf{z}_i(t)), \tag{4}$$

where α_i is a "leaking" or mixing rate. For the first layer, $\mathbf{u}_1(t)$ is (x_t, y_t, z_t). For subsequent layers, $\mathbf{u}_i(t)$ is the previous layer's state $\mathbf{s}_{i-1}(t)$. The final layer state $\mathbf{s}_L(t)$ feeds a linear readout mapping to the output $\hat{\mathbf{y}}(t)$.

a) Training via Ridge Regression.: The reservoir states $\mathbf{s}_L(t)$ and corresponding target outputs $\mathbf{y}(t)$ are collected over the training sequence (excluding a warm-up of 200 steps). The readout matrix W_{out} is found by solving (see [4]):

$$W_{\text{out}} \;=\; \arg\min_{W_{\text{out}}} \Big\| \, S\,W_{\text{out}} \;-\; Y \Big\|^2 \;+\; \lambda\,\|W_{\text{out}}\|^2, \tag{5}$$

where S is the collected reservoir states and Y is the target matrix, and λ is a small regularization parameter (e.g., 10^{-5}) [4].

Ridge regression is a variant of linear regression that incorporates an additional L2 regularization term to prevent overfitting. In standard linear regression, one seeks a weight matrix W that minimizes the squared error between the predictions SW and the target outputs Y. However, when the input features are noisy, highly correlated, or the model is overly flexible, the weights can grow very large, leading to poor generalization.

To address this, ridge regression modifies the loss function by adding the penalty term $\lambda\|W\|^2$. Here, $\|W\|^2$ is the squared L2 norm of the weight matrix (i.e., the sum of the squares of its elements), and λ is a non-negative parameter that controls the strength of the regularization. A small value of λ (such as 10^{-5}) provides a gentle penalty that still helps to stabilize the weight estimates, ultimately yielding a model that generalizes better to new data.

C. Traditional RNN Baseline with Truncated Backpropagation Through Time (BPTT)

For comparison, we employ a single-layer RNN with hidden size H. For the RNN baseline, the hidden state and output updates are defined as (see [3] and [1]):

$$\mathbf{h}(t) = \tanh\Big(W_{\text{ih}}\,\mathbf{u}(t) + W_{\text{hh}}\,\mathbf{h}(t-1)\Big), \tag{6}$$

$$\hat{\mathbf{y}}(t) = W_{\text{hy}}\,\mathbf{h}(t), \tag{7}$$

where $\mathbf{u}(t)$ denotes the input at time t and $\mathbf{h}(t)$ the hidden state. All parameters $\{W_{\text{ih}}, W_{\text{hh}}, W_{\text{hy}}\}$ (along with biases) are trained via truncated backpropagation through time (BPTT) [3].

Due to the challenges posed by long sequences (e.g., vanishing/exploding gradients), the training series is partitioned into shorter segments (e.g., length 50) as recommended in [1]. Within each segment the network processes the data, computes the loss (using mean-squared error), and then backpropagates gradients through only that limited number of time steps. The hidden state is carried over between segments but is *detached* from the computation graph at the end of each segment, thereby truncating gradient flow.

The training procedure in our PyTorch implementation is summarized in Algorithm 1.

Algorithm 1 Truncated BPTT for RNN Training

Initialize: Hidden state $h \leftarrow 0$
for each epoch **do**
  for $t = 0$ to $T-$ `chunk_length` in steps of `chunk_length` **do**
    Extract input chunk: $X_{\text{chunk}} = X(t : t +$ `chunk_length`$)$
    Extract target chunk: $Y_{\text{chunk}} = Y(t : t +$ `chunk_length`$)$
    $[\hat{Y}, h] \leftarrow \text{RNN}(X_{\text{chunk}}, h)$
    Compute loss: $\mathcal{L} = \text{MSE}(\hat{Y}, Y_{\text{chunk}})$
    Backpropagate $\mathcal{L}$ and update weights using the Adam optimizer [20]
    Detach h: $h \leftarrow \text{detach}(h)$
  end for
end for

This truncated BPTT strategy limits the effective temporal span over which gradients are computed, reducing computational load and trying to mitigate instability—contrasting with ESNs, where the reservoir dynamics remain fixed and only the linear readout is trained.

IV. Experimental Setup

A. Memristive Reservoir Configuration

In a typical experiment, I set:

- Number of reservoir layers $L = 2$.
- Each layer has 500 units and 10% sparsity for the recurrent weights.
- Mixing rates $\alpha_1 = \alpha_2 = 0.1$.

- Spectral radii $\rho_1 = \rho_2 = 0.9$.
- Input scalings $\beta_1 = \beta_2 = 1.0$.

I then apply ridge regression (with $\lambda = 10^{-5}$) to find the readout weights.

B. RNN Configuration

For the RNN baseline, we use:

- A single hidden layer with size $H = 100$.
- $\tanh$ activations.
- Chunk-based truncated BPTT with chunk length 50.
- Adam optimizer (learning rate 0.001) for up to 10 epochs.

C. Evaluation Metrics

All models are evaluated on the final 10,000-step test set. The primary metric is the root mean square error (RMSE):

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{t=1}^{N} \|\hat{\mathbf{y}}(t) - \mathbf{y}(t)\|^2}, \tag{8}$$

where $\hat{\mathbf{y}}(t)$ and $\mathbf{y}(t)$ are the three-dimensional Lorenz states.

V. Results and Discussion

A. Prediction Accuracy

Table I shows the test RMSE for the memristive reservoir (with two 500-unit layers) versus the RNN (hidden size 50). The reservoir outperforms the RNN in one-step-ahead prediction despite employing fewer *trained* parameters.

TABLE I: Test RMSE for Memristive Reservoir vs. Traditional RNN

Model	Dataset Size	Test RMSE
Traditional RNN (Hidden Size 50)	10,000	2.624411
Memristive Reservoir (2 × 500 Units)	10,000	0.1663
Memristive Reservoir (2 × 500 Units)	25,000	0.9143

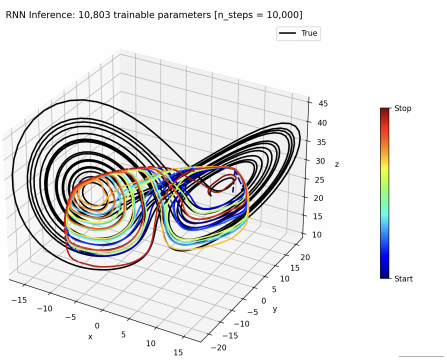

Fig. 3: RNN Inference: 10,803 Trainable Parameters [*n_steps* = 10,000]. Black curves show the ground-truth Lorenz attractor, while the color-coded lines (blue to red) illustrate the RNN's predicted trajectories from start to stop. The model is definitely having trouble fitting to the curve.

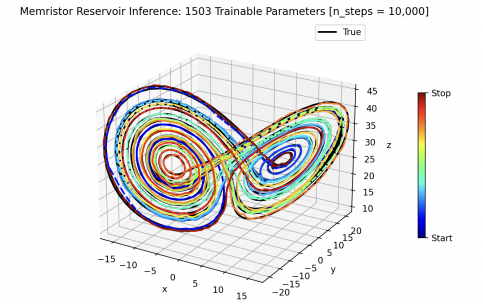

Fig. 4: Memristor Reservoir Inference: 1,503 Trainable Parameters [*n_steps* = 10,000]. Again, the black curves are the true Lorenz system, with the reservoir's predictions color-coded from start (blue) to stop (red).

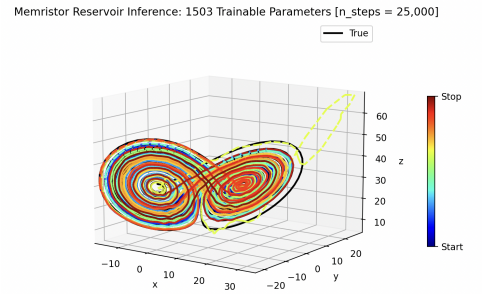

Fig. 5: Memristor Reservoir Inference: 1,503 Trainable Parameters [*n_steps* = 25,000]. Same-sized reservoir model can fit to the more complex data with reasonable accuracy aside from a single sporadic output near middle of inference stage, seen in yellow to the top right.

B. Parameter Budget and Efficiency

A key advantage of the reservoir approach is that only the readout layer is trained, while the vast majority of the reservoir weights remain fixed after random initialization. Below, I reassess the parameter counts for both models, clarifying how many are actually optimized.

1) Memristive Reservoir Parameter Count: Consider a two-layer reservoir, each with 500 reservoir units. The first layer receives a 3D input, and the second layer receives a 500D input (the state from layer 1). I also include bias vectors in each layer for completeness, each bias having 500 parameters.

- **Layer 1 (Reservoir):**
  - Recurrent weight matrix W_1: $500 \times 500 = 250{,}000$ total entries, with 10% density $\rightarrow$ 25,000 *non-zero* parameters (fixed).
  - Input weight matrix W_1^{in}: $500 \times 3 = 1{,}500$ parameters (assume fully connected).
  - Bias vector b_1: 500 parameters.
  - *Total, Layer 1:* $25{,}000 + 1{,}500 + 500 = 27{,}000$ (fixed after initialization).
- **Layer 2 (Reservoir):**
  - Recurrent weight matrix W_2: $500 \times 500 = 250{,}000$ total, 10% density $\rightarrow$ 25,000 *non-zero* parameters

(fixed).
  - Input weight matrix W_2^{in}: $500 \times 500 = 250{,}000$ parameters (assume fully connected).
  - Bias vector b_2: 500 parameters.
  - *Total, Layer 2:* $25{,}000 + 250{,}000 + 500 = 275{,}500$ (fixed).

Summing these:

$$\text{Reservoir (fixed)} = 27{,}000 + 275{,}500 \;=\; 302{,}500.$$

Readout Layer: A linear map from the second layer's 500-dimensional state to a 3D output:

$$(500\times 3) = 1{,}500 \text{ weights} + 3 \text{ biases} \;=\; 1{,}503 \text{ trained params.}$$

Hence, the *total* parameter count for the memristive reservoir network is:

$$302{,}500 \text{ (fixed)} \;+\; 1{,}503 \text{ (trained)} = 304{,}003.$$

The key point is that **only 1,503 parameters are actually optimized** during training (the readout), whereas the large bulk of reservoir weights remain unchanged.

2) Traditional RNN Parameter Count: For a single-layer RNN with hidden size $H = 100$, input size 3, and output size 3:

- Input-to-hidden matrix W_{ih}: $(100 \times 3) = 300$ parameters.
- Hidden-to-hidden matrix W_{hh}: $(100 \times 100) = 10{,}000$ parameters.
- Biases for these connections (two bias vectors, each size 100): 200 parameters.
- Output layer W_{hy}: $(100 \times 3) = 300$ parameters, plus 3 biases.

Thus, the RNN has:

$$300{+}10{,}000{+}200{+}(300{+}3) = 10{,}803 \quad \text{trainable parameters.}$$

3) Comparison:

- *Memristive Reservoir:* $\sim$ 304,000 total parameters (mostly fixed), with **1,503 trained**.
- *Traditional RNN:* $\sim$ 10,800 total parameters, all **10,800 are trained**.

Despite having far fewer trained parameters, the memristive reservoir *outperforms* the RNN in Lorenz prediction. This highlights a key benefit of reservoir computing: large, random recurrent layers provide rich dynamics, while only a small readout is optimized, reducing training complexity and circumventing issues like vanishing gradients [1].

C. Discussion of Reservoir Advantages

a) Echo-State Property: By enforcing spectral radii below 1, the reservoir preserves stable internal dynamics that prevent uncontrolled error amplification, critical for modeling chaotic systems [5], [7].

b) High-Dimensional Embedding: With hundreds of units per layer, the reservoir offers a rich nonlinear embedding of inputs, making it easier for a simple linear readout to capture complex time dependencies [8].

c) Simplified Readout-Only Training: Because we only train $\sim$ 1,500 parameters via closed-form ridge regression, training is faster and more robust compared to full BPTT in a large RNN [3], [4].

d) Final Remarks: A key reason why the memristive reservoir outperforms the fully trained RNN in this chaotic time-series task stems from the reservoir's intrinsic ability to maintain stable, yet rich, internal dynamics. Specifically, the echo-state property ensures that the reservoir's hidden states remain bounded and eventually wash out the influence of initial transients, a critical trait for accurately capturing chaotic behavior where small perturbations can grow rapidly. By setting the spectral radius of the recurrent weight matrices below one, we can guarantee that the reservoir exhibits a contractive effect, preventing unbounded state explosions common in naively trained RNNs.

Moreover, the large dimensionality of each reservoir layer (e.g., 500 neurons) produces a high-dimensional, nonlinear embedding of the input sequence; this effectively transforms the three-dimensional Lorenz signal into a far richer space. Even though these recurrent weights are random and remain fixed, they are diverse enough to capture the subtle temporal dependencies that define the Lorenz attractor's swirling trajectories. The linear readout layer then needs to learn only a simple map from the reservoir state to the forecasted outputs. This contrast with a fully trained RNN is significant: in the RNN, the same weights must learn both the optimal representation of the chaotic dynamics and the output mapping, complicating the training procedure and making it more susceptible to vanishing or exploding gradients. In other words, reservoir computing offloads the burden of feature extraction to the large but fixed recurrent layer, simplifying the optimization problem to a single ridge regression step on the readout. This structure not only circumvents the pitfalls of full backpropagation through time but also drastically reduces the number of trainable parameters, making training more robust, faster, and less prone to overfitting—all of which are particularly advantageous when dealing with highly sensitive chaotic processes.

VI. CPU Runtime and Usage

To complement the accuracy evaluations, I integrated comprehensive runtime and CPU usage monitoring within each Python experiments. The implementation uses the `psutil` package to capture CPU times (both user and system) and the `time` module to track wall-clock time during the execution of both the memristive reservoir and traditional RNN experiments.

Specifically, the code records the starting CPU times and wall-clock time before training begins. After running the training and testing routines for each model, the ending CPU times and wall-clock time are recorded. The total CPU time is computed as the sum of the differences in user and system times, and the average CPU usage is estimated by comparing the CPU time to the wall-clock time. These measurements are then visualized via a bar chart that compares wall-clock

time, total CPU time, and average CPU usage for the two approaches.

The CPU usage is computed by comparing the total CPU time consumed by the process to the overall elapsed wall-clock time. In this implementation, we can first record the CPU times for both user mode and system mode. Let

$$T_{\text{user}} \quad \text{and} \quad T_{\text{sys}}$$

represent the user and system CPU times, respectively. The total CPU time is then given by

$$T_{\text{CPU}} = T_{\text{user}} + T_{\text{sys}}.$$

Simultaneously, we measure the elapsed wall-clock time as

$$T_{\text{wall}} = t_{\text{end}} - t_{\text{start}},$$

where t_{start} and t_{end} are the wall-clock times at the beginning and the end of the process, respectively.

The average CPU usage percentage is calculated by taking the ratio of the total CPU time to the wall-clock time and multiplying by 100:

$$\text{CPU Usage (\%)} = 100 \times \frac{T_{\text{CPU}}}{T_{\text{wall}}}.$$

This metric provides an estimate of the fraction of the elapsed time during which the CPU was actively executing the process's code. For example, if a process uses 2 seconds of CPU time over a 10-second period, the CPU usage would be:

$$100 \times \frac{2}{10} = 20\%.$$

This indicates that, on average, the process was actively utilizing the CPU for 20% of the total elapsed time.

TABLE II: Comparison of Reservoir vs. RNN in terms of RMSE, execution time, and CPU usage.

Metric	Reservoir	RNN
Test RMSE	0.1339	1.1908
Wall-clock time (sec)	0.97	3.56
CPU time (sec)	2.75	13.91
Approx. Avg. CPU Usage (%)	282.6	390.8

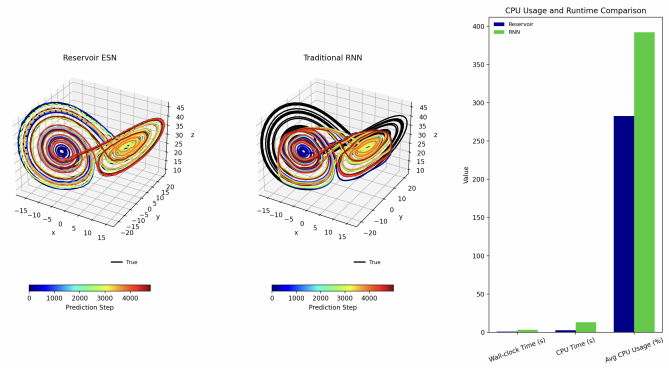

Fig. 6: Wall-clock and CPU time comparison, as well as CPU Usage

In the reservoir experiment, the network's fixed reservoir dynamics (with only the readout layer trained via ridge regression) result in lower computational demands compared to the RNN, which must update and train all parameters via truncated backpropagation through time. This integration of performance monitoring not only ensures reproducibility but also highlights the computational efficiency of the reservoir approach—critical for applications where resources and energy consumption are key considerations.

VII. Conclusion and Future Work

This comparative study demonstrates that a multi-layer memristive reservoir network can achieve superior one-step-ahead prediction accuracy on the Lorenz system with markedly fewer trainable weights than a fully trained RNN. The reservoir's design leverages the echo-state property and high-dimensional state expansion to embed chaotic input streams in a stable and expressive state space. A simple linear readout is sufficient to perform effective forecasting, thereby circumventing the complexities of gradient-based training over long sequences.

Furthermore, our CPU runtime analyses reveal that the reservoir-based approach offers not only computational efficiency but also runtime advantages. This efficiency, along with improved prediction performance, underscores the promise of reservoir computing for real-time forecasting tasks under resource-constrained conditions.

Future work will focus on:

- **Multi-step-ahead prediction**, where the model's output is recursively fed back as input [21].
- **Hardware implementations** of memristive reservoirs on FPGA or ASIC platforms to exploit low-power, efficient computing [11].
- **Extension to other complex systems**, such as higher-dimensional chaotic models and real-world applications like climate or financial forecasting.

These avenues promise to further exploit the energy efficiency and computational strengths of memristive reservoir computing, paving the way for robust, scalable solutions in time-series prediction.

References

[1] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *Proc. 30th Int. Conf. on Machine Learning*, 2013, pp. 1310–1318.

[2] L. O. Chua, "Memristor—The missing circuit element," *IEEE Trans. Circ. Theory*, vol. 18, no. 5, pp. 507–519, 1971.

[3] P. J. Werbos, "Backpropagation through time: What it does and how to do it," *Proc. IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[4] A. E. Hoerl and R. W. Kennard, "Ridge regression: Biased estimation for nonorthogonal problems," *Technometrics*, vol. 12, no. 1, pp. 55–67, 1970.

[5] H. Jaeger, "The "echo state" approach to analysing and training recurrent neural networks," GMD Report 148, German National Research Center for Information Technology, 2001.

[6] H. Soh and Y. Demiris, "Spatio-Temporal Learning With the Online Finite and Infinite Echo-State Gaussian Processes," *IEEE Trans. Neural Networks and Learning Systems*, vol. 26, no. 3, pp. 522–536, 2014.

[7] M. Lukoševičius and H. Jaeger, "Reservoir computing approaches to recurrent neural network training," *Computer Science Review*, vol. 3, pp. 127–149, 2009.

[8] W. Maass, T. Natschläger, and H. Markram, "Real-time computing without stable states: A new framework for neural computation based on perturbations," *Neural Computation*, vol. 14, no. 11, pp. 2531–2560, 2002.

[9] E. Covi *et al.*, "Memristive synapses for neuromorphic networks: Recent developments and future perspectives," *IEEE Trans. Nanotechnology*, vol. 15, no. 6, pp. 1265–1275, 2016.

[10] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, no. 7191, pp. 80–83, 2008.

[11] H. Chen, X.-G. Tang, and Z. Shen, "Emerging memristors and applications in reservoir computing," *Frontiers of Physics*, vol. 19, no. 1, p. 13401, 2024.

[12] A. Singh, S. Choi, G. Wang, M. Daimari, and B.-G. Lee, "Analysis and fully memristor-based reservoir computing for temporal data classification," *arXiv preprint arXiv:2403.01827*, 2024.

[13] V. Lucarini, D. Faranda, A. C. M. Freitas, et al., "Extreme value theory for chaotic systems," in *Reviews of Nonlinear Dynamics and Complexity*, Vol. 6, Wiley, 2014.

[14] K. Kaneko, "Overview of coupled map lattices," *Chaos*, vol. 2, no. 3, pp. 279–282, 1992.

[15] R. Cont, "Empirical properties of asset returns: Stylized facts and statistical issues," *Quantitative Finance*, vol. 1, no. 2, pp. 223–236, 2001.

[16] E. N. Lorenz, "Deterministic nonperiodic flow," *Journal of the Atmospheric Sciences*, vol. 20, no. 2, pp. 130–141, 1963.

[17] J. C. Butcher, *The Numerical Analysis of Ordinary Differential Equations*, John Wiley & Sons, 1987.

[18] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[19] K. Cho *et al.*, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *Proc. 2014 Conf. on Empirical Methods in Natural Language Processing*, 2014, pp. 1724–1734.

[20] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *International Conference on Learning Representations (ICLR)*, 2015.

[21] J. Pathak, B. Hunt, M. Girvan, Z. Lu, and E. Ott, "Model-free prediction of large spatiotemporally chaotic systems from data: A reservoir computing approach," *Physical Review Letters*, vol. 120, no. 2, p. 024102, 2018.

[22] P. R. Vlachas *et al.*, "Data-driven forecasting of high-dimensional chaotic systems with long short-term memory networks," *Proc. Royal Society A*, vol. 474, no. 2213, p. 20170844, 2018.