Understanding the Effects of Biologically Inspired Learning Rules in a Reservoir Computer

Author: David Fox - 118707149
Supervisors: Andrew Flynn, Andreas Amann

MS4090 - Mathematical Sciences Project

University College Cork
April 2022

Abstract

Reservoir computing is a machine learning paradigm that is particularly suited towards imitating both the short-term and long-term behaviour of DSs based solely on time series data. Despite the fact that reservoir computers are themselves DSs, systematic explorations of their high-dimensional dynamics are often avoided due to computational cost. However there is much to be gained from low-cost explorations of the state space of a reservoir computer as it allows one to inspect how this machine learns. Upon closer exploration of the state space of a reservoir computer we discover untrained attractors, which are attractors present in the reservoir computer phase space after training that represent different dynamics from that of the system the reservoir computer is trying to learn. The existence of untrained attractors presents a problem with the reliability of a reservoir computer's output, erecting a barrier to their widespread use in applications where critical safety design is of concern. In this report, we present a training method which is capable of overcoming these issues by taking advantage of known dynamical principles of plasticity in biological brains.

Contents

Abstract **1**

Acknowledgements **4**

Statement of Data Availability **5**

List of Figures **6**

List of Symbols **7**

List of Abbreviations **8**

1 Introduction **9**

2 Dynamical Systems **10**
2.1 Trajectories and Attractors . . . 10
2.2 The Lorenz System and Chaotic Attractors . . . 10

3 Reservoir Computing **12**
3.1 Training . . . 12
3.2 Predicting . . . 12
3.3 Implementation . . . 13
3.3.1 Echo-State Networks . . . 13
3.3.2 Time Series Prediction with an Echo-State Network . . . 15
3.4 Advantages of Reservoir Computing . . . 17
3.5 Replication of Dynamics Through Attractor Reconstruction . . . 17
3.5.1 Testing Phase Error . . . 18
3.6 Exploring Untrained Attractors in an ESN . . . 20

4 Plasticity Rules **21**
4.1 Synaptic Plasticity . . . 21
4.1.1 Synaptic Plasticity Rule applied to an Echo-State Network . . . 23
4.2 Intrinsic Plasticity . . . 23
4.2.1 Intrinsic Plasticity Rule applied to an Echo-State Network . . . 24
4.3 Combining Plasticity Rules . . . 24

5 Results **25**
5.1 Generating Training Data . . . 25
5.2 Synaptic Plasticity Rule . . . 25
5.3 Intrinsic Plasticity Rule . . . 30
5.4 Intrinsic, Synaptic Plasticity Rule . . . 33
5.5 Synaptic, Intrinsic Plasticity Rule . . . 36
5.6 Comparing Biological Learning Rules . . . 38

6 Discussion and Conclusion **42**

A Python Implementation of an Echo-State Network and Plastic Echo-State Networks **44**

B Classifying Attractors **50**
B.1 Fixed Points 50
B.2 Limit Cycles 50
B.3 Lorenz Attractor 51
B.4 Other 52
B.5 Classifying a Set of Trajectories 52

C Reconstruction Proportion **53**

D Generating ρ, σ Plots **54**

E Finding η, Epoch Values Which Remove Untrained Attractors **56**

Acknowledgements

I would like to express my sincerest gratitude to my supervisors Andrew Flynn and Andreas Amann, for igniting my interest in the study of dynamical systems and machine learning, they have both been incredibly generous with their time over the past year and gave me the guidance and encouragement to get to this point in my studies.

Getting this far in my education would have been impossible without the support of many people who I am so grateful to have in my life; my parents, without the support of which I would have never been able to pursue my passion for mathematics, my siblings Catherine and Stephen and my lifelong friends Eamonn, Eoin and Rob.

I can't fully express how thankful I am to my loving partner Erin, who fills each day with a level of joy I thought impossible at one stage.

Finally, I would like to dedicate this work to the memory of my dog Jake. You were my best friend, and though the boat is empty now, I'll always remember the many happy voyages we took.

Statement of Data Availability

The Python code and data generated in the conduct of this project is available at www.github.com/BitVil/FinalYearProject

List of Figures

1 Lorenz Attractor . . . 11
2 LR Architecture . . . 14
3 PR Architecture . . . 15
4 Attractor Reconstruction for Lorenz System with an Echo State Network . . . 16
5 Example of Testing Phase Error for ESN Predictions . . . 19
6 Untrained Attractors Present in Echo State Network Predicting Reservoir . . . 20
7 Plots of Prediction Log Testing Phase Error, Reconstruction Proportion and Untrained Attractors for Echo State Networks . . . 22
8 Evolution of Adjacency Matrix **M** During Training with Synaptic Plasticity Rule . . 27
9 Evolution of Covariance Matrix for Listening Reservoir State During Training with Synaptic Plasticity Rule . . . 28
10 Prediction log TPE for ESNs and SP ESNs . . . 29
11 RP for ESNs and SP ESNs . . . 29
12 Attractors present in PR phase space for ESNs and SP ESNs . . . 30
13 Evolution of Gain and Bias for IP ESN During Training with IP Rule . . . 31
14 Histogram of LR State Values Before and After Applying the IP Rule . . . 32
15 Prediction log TPE for ESNs and IP ESNs . . . 33
16 RP for ESNs and IP ESNs . . . 34
17 Attractors Present in PR Phase Space for ESNs and IP ESNs . . . 34
18 Prediction log TPE for ESNs and IP-SP ESNs . . . 35
19 RP for ESNs and IP-SP ESNs . . . 35
20 Attractors present in PR phase space for ESNs and IP-SP ESNs . . . 36
21 Prediction log TPE for ESNs and SP-IP ESNs . . . 37
22 RP for ESNs and SP-IP ESNs . . . 37
23 Attractors present in PR phase space for ESNs and SP-IP ESNs . . . 38
24 Prediction log TPE for ESNs and all types of PESNs . . . 39
25 RP for ESNs and all types of PESNs . . . 40
26 Attractors present in PR phase space for ESNs and all types of PESNs . . . 41
27 Examples of PR State Space After Applying SP Rule . . . 58

List of Symbols

r	Listening reservoir state
$\tilde{\mathbf{r}}$	Predicting reservoir state
t	time
$\mathbf{M}$	Echo state network adjacency matrix
$\mathbf{W}_{in}$	Echo state network input matrix
$\mathbf{W}_{out}$	Echo state network input matrix
ρ	Adjacency matrix spectral radius
σ	Input matrix scaling factor
γ	Echo state network timescale parameter
β	Regularization parameter
N	Echo state network dimension
p	Echo state network connection density
d	Dimension of training signal
$\hat{\Psi}$	Learned function from listening reservoir state to training signal
L	Lorenz attractor
Φ	Synchronization function
dt	Integration timestep
η_i	Intrinsic plasticity rule learning rate
η_s	Synaptic plasticity rule learning rate
$epochs_i$	Number of intrinsic plasticity rule epochs
$epochs_s$	Number of synaptic plasticity rule epochs

List of Abbreviations

DS	dynamical system
RC	reservoir computing
PR	predicting reservoir
UA	untrained attractor
SP	synaptic plasticity
IP	intrinsic plasticity
ESN	echo-state network
ODE	ordinary differential equation
LR	listening reservoir
PR	predicting reservoir
RNN	recurrent neural network
BPTT	back-propagation through time
IVP	initial value problem
AR	attractor reconstruction
LSTM	long short-term memory
TPE	testing phase error
RP	reconstruction proportion
IC	initial condition
SP-ESNs	synaptically plastic echo-state networks
IP-ESNs	intrinsically plastic echo-state networks
PESNs	plastic ESNs
FP	fixed point
LC	limit cycle

1 Introduction

A dynamical system (DS) is a process with associated measurable quantities which co-evolve in time, real world examples can be found in chemistry, fluid mechanics, neuroscience and laser physics [1].

DSs are mathematically modelled using sets of ordinary, delay or partial differential equations (ODEs) which describe the evolution of the system's variables with time.

Historically, these equations are derived from first principles using more general governing equations as a starting point such as Newton's second law, Maxwell's equations or the Navier-Stokes equations [2]. As machine learning approaches have matured over recent years a number of data driven approaches to modelling DSs have been employed in a variety of settings [3], in particular reservoir computing (RC) [4], [5] has emerged as a method which is capable of learning the underlying dynamics of a system based on time series data sampled from the system of interest [6].

The idea of RC is to take a central DS called the reservoir and drive it with an input signal to produce a time series of reservoir states. Then a regression procedure is used to fit the reservoir state at a given time to the corresponding value of the driving signal. This learned function from the reservoir state to the driving signal is then used to evolve the reservoir system autonomously, by replacing the driving signal with this function of the reservoir state. We call this DS the predicting reservoir (PR). If this training procedure is successful, then the PR will exhibit dynamical properties which approximate those of the system from which the input signal was sampled.

A theoretical framework based on the idea of generalised synchronization which explains how a RC can learn the dynamical law which generated a given time series is established in [7]. During training, the reservoir embeds the attractor that the input signal is evolving on as an attractor in the PR phase space, so that the reservoir state continues to evolve on this embedded attractor when evolving according to the PR dynamics. However, even if this training process is successful it is still possible for there to be other 'untrained attractors' (UAs) present in the PR phase space, which were not part of the training [8]. These UAs present a problem in terms of the reliability of predictions made by a RC as their presence may influence the predictions generated by the RC, for instance, attractor merging crises or intermittent transitions between attractors could potentially interfere with the training of the RC. Problems with the reliability of RCs is a significant barrier to their application in complex modelling or control tasks where critical safety design is of utmost importance. While RCs can indeed learn the dynamics of a given attractor, the global stability of the attractor is not always guaranteed which presents a problem in the context of creating a digital twin of a system or process using experimental data. In this report we investigate methods which can improve a RCs reconstruction of a given DS and it's state space with access to information from only a trajectory on one of its attractors.

Recently, the traditional RC training method described above was augmented by Morales et al. in 2021 to include learning rules which implement intrinsic and synaptic plasticity observed in biological brains. The idea of the synaptic plasticity (SP) rule is to alter the hidden layer connection weights based on the networks input, while the intrinsic plasticity (IP) rule alters the excitability of individual hidden layer nodes based on an input. This augmented training method was shown to have improved accuracy of short term prediction for time series sampled from the Mackey-Glass system [9]. In this report, we successfully apply the same plasticity rules to remove UAs in the PR phase space of a RC trained to replicate the dynamics of the Lorenz system. In addition to this, we analyse the effect had on UAs when these plasticity rules are included in the RC training. We provide an in depth analysis of each rule and assess the effects of using different combinations of these rules to remove UAs and improve RC performance.

The remainder of this report is structured as follows: in section 2 we introduce some concepts and notation from DS theory which is necessary to facilitate our discussion of RC from the perspective of UAs; in section 3 we describe the RC paradigm and give an example of it being implemented using an echo-state network (ESN), the concept of UAs is also explored in detail; section 4 discusses the idea of neurological plasticity and introduces the mathematical implementations of IP and SP used in this report; section 5 presents results of applying these plasticity rules in numerical experiments; section 6 concludes by discussing these results in the context of the literature. Finally the appendices detail some of the methods used throughout this work, including methods for attractor classification, plot generation, parameter selection and the Python programs in which these were implemented.

2 Dynamical Systems

In this section, we introduce some definitions and notation from DSs theory which is needed to discuss RC from the perspective of UAs.

Time is treated as a continuous variable in the systems considered in this report. In continuous time, a DS is described mathematically using an ordinary differential equation (ODE) of the form,

$$\dot{\mathbf{x}} = F(\mathbf{x}), \tag{1}$$

where $\mathbf{x} \in X \subset \mathbb{R}^n$ is a vector of state variables, $n \in \mathbb{N}$ is the number of state variables of the system, X is the phase space which is the set of all possible states of the system, $\dot{\mathbf{x}}(t)$ is the time derivative of the vector $\mathbf{x}$ and $F : \mathbb{R}^n \to \mathbb{R}^n$ is a vector field.

2.1 Trajectories and Attractors

A trajectory through a point $\mathbf{x}_0 \in X$ of the DS is a time parameterized curve $\mathbf{x}(t) \subset X$ which is a solution to equation 1 and also satisfies the initial condition (IC) $\mathbf{x}(0) = \mathbf{x}_0$. The trajectory $\mathbf{x}(t)$ gives the temporal evolution of the systems n state variables through the phase space X when initialised with the unique state $\mathbf{x}_0$.

In some cases, a DS will have a bounded subset of its phase space $A \subset X$ which is contained in a set $B(A) \subset \mathbb{R}^n$ whose points have trajectories which tend to A as $t \to \infty$. We call the set A an attractor and $B(A)$ its basin of attraction.

If the vector field $F(\mathbf{x})$ which describes the dynamics of the system is nonlinear, we will not be able to find analytic solutions to (1) in general. As a consequence of this, we need to approximate trajectories of the system using numerical methods.

In this study we investigated different numerical integration methods for solving ODEs including the Euler method and 4th order Runge-Kutta method [10] implemented by the author in Python, as well as the `scipy.integrate.odeint` function from SciPy's Integrate package [11]. The method we used to generate our results was `scipy.integrate.odeint` as it led to the most accurate results.

2.2 The Lorenz System and Chaotic Attractors

A well known example of a DS which we will refer to many times in this report is the Lorenz system, which was derived from the Navier-Stokes equations by Edward Lorenz in 1963 as a simple model for convection rolls in the atmosphere [12]. The dynamics of the chaotic Lorenz system are described by the following set of equations,

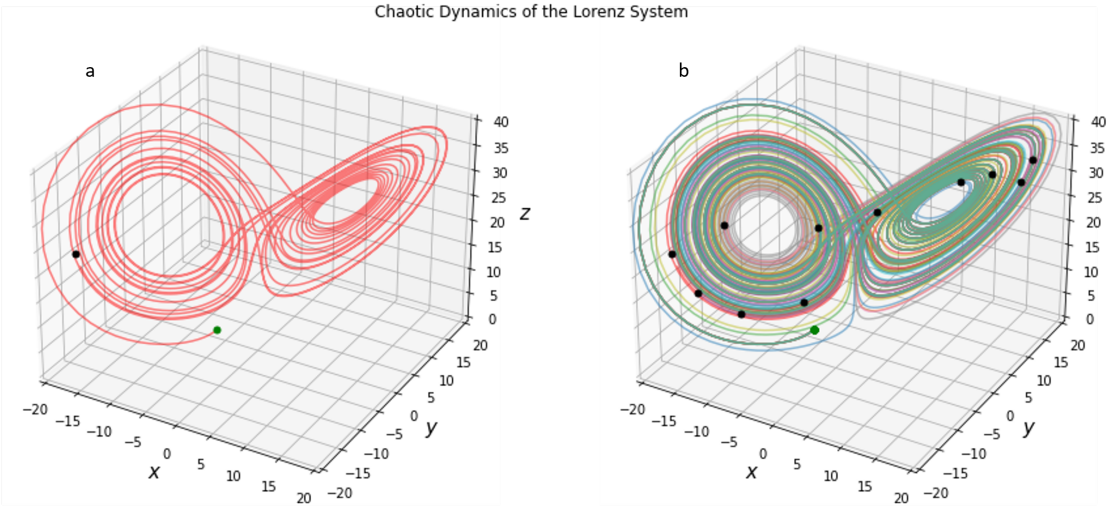

Figure 1: Panel a. Trajectory of the Lorenz system generated numerically for T=100 time units, initial and final points are given by green and black dots respectively. Panel b: Trajectory from panel a and perturbed trajectories, initial and final points are given by green and black dots respectively.

$$\begin{aligned} \dot{x} &= 10(y - x), \\ \dot{y} &= x(28 - z) - y, \\ \dot{z} &= xy - \frac{8}{3}z. \end{aligned} \tag{2}$$

Figure 1.a shows an exemplary trajectory from the Lorenz system generated by numerically integrating the Lorenz equations from the IC $(x(0), y(0), z(0)) = (-1.7, 5, 0.4)$ using `scipy.integrate.odeint` for 100 time units and a step size of 0.01. The green dot gives the location of the IC and the black dot the location of the trajectory at $t = 100$. Notice that as the trajectory evolves with time it converges to the Lorenz attractor L, tracing out its shape which resemble the wings of a butterfly.

The Lorenz attractor is a global attractor, meaning its basin of attraction is the entire phase space, that is $B(L) = X \subset \mathbb{R}^3$.

The Lorenz attractor is a famous example of a chaotic attractor, which means the trajectories of points in its basin of attraction exhibit a sensitive dependence on ICs. More precisely this means if two points in $B(L)$ are infinitesimally close together, their trajectories separate exponentially fast as they converge to L. This behaviour can be seen in figure 1.b, which shows the trajectories of ten random ICs which are perturbed a distance 10^{-4} from the IC used in panel a. The size of the perturbations is small enough for the starting point of each trajectory (denoted again by green dots) to be indistinguishable from each other. However, because of the sensitive dependence on ICs exhibited by the Lorenz system, the resulting trajectories evolve on the Lorenz attractor via completely different routes which can be seen by the different colour curves.

This sensitive dependence on ICs makes long term prediction of trajectories evolving on chaotic attractors a difficult task. This is because any error in the prediction, which is certain to be present

due to the measurement process or the method of prediction itself, will cause our prediction to diverge from the actual trajectory of the system exponentially fast. For this reason, predicting future values of a time series taken from a chaotic DS is a common benchmark task for prediction methods [6].

3 Reservoir Computing

In this section we use the definitions and notation established in the last section to describe the RC paradigm from the perspective of DSs theory.

As mentioned previously, RC attempts to learn the underlying dynamics which generated a training signal $\mathbf{u}(t)$ by utilizing the dynamic potential of a central DS called the reservoir [13]. In this section we outline the method used in 2018 by Lu et al. [6] to train an RC to predict future values of a training signal $\mathbf{u}(t)$. We also present a more detailed discussion of what it means to replicate dynamics with an RC and a mechanism based on generalized synchronization by which RCs can achieve this proposed by Lu et al. in 2020 [7].

3.1 Training

To train the reservoir to replicate dynamics, we first describe the listening reservoir (LR) system whose dynamics are given by,

$$\dot{\mathbf{r}}(t) = F(\mathbf{r}(t), \mathbf{u}(t)). \tag{3}$$

We generate a trajectory of the LR by finding a solution to the non-autonomous system (3) with a given IC, usually the origin $\mathbf{r}(0) = (0, ..., 0)$.

The next step of training is to fit the trajectory of LR states $\mathbf{r}(t)$ to the driving signal $\mathbf{u}(t)$ via regression, that is we seek a function $\Psi(\mathbf{r})$ which satisfies,

$$\Psi(\mathbf{r}(t)) \approx \mathbf{u}(t). \tag{4}$$

In practice, the function $\Psi(\mathbf{r})$ can be found using a generalised linear regression procedure such as LASSO [14] or ridge regression [15].

3.2 Predicting

In order to make predictions of future points of the training signal we first must define the PR system, whose dynamics are given by,

$$\dot{\tilde{\mathbf{r}}}(t) = F(\tilde{\mathbf{r}}(t), \Psi(\tilde{\mathbf{r}}(t))) \tag{5}$$

we can see in equation (5) that the PR system is defined by replacing the occurrence of $\mathbf{u}(t)$ in equation (3) with its approximation we found during training $\Psi(\mathbf{r})$, thus creating an autonomous DS. To generate the prediction, we evolve the PR system using equation (5) with the last point of the LR trajectory generated during training as our IC, then the curve $\Psi(\mathbf{r}(t))$ is our prediction for the future values of the training signal $\mathbf{u}(t)$.

3.3 Implementation

There are multiple ingredients which need to be chosen in order to implement an RC. First we must choose the system which will be our reservoir, there are many examples of systems, both physical and digital, which have been used as reservoirs such as photonic systems [16], an octopus inspired soft robotic arm [17] and even a bucket of water [18]. Additionally we must choose the form of the function Ψ and the regression procedure used during training to fit the reservoir state to the training signal. A very popular choice of reservoir is a recurrent neural network (RNN) with a single sparsely connected hidden layer and $tanh()$ activation function. Such RCs, known as ESNs, were one of the original implementations of an RC and are still commonly used today. They were first conceived of by Herbert Jaeger in the early 2000's as a means of training an RNN for time-series prediction while overcoming the issue of the vanishing / exploding gradient and the heavy computational cost associated with training an RNN using back-propagation through time (BPTT) [4].

ESNs are advantageous for research purposes as they are very easy to implement *in silico* and are the type of RC used to generate numerical results in the remainder of this report.

3.3.1 Echo-State Networks

In this section we describe the ESN implementation used by Lu et al. [6] which closely resembles the original ESN used by Jaeger in 2001 [4].

Suppose we have a time-series $\{\mathbf{u}(t_i)\}_{i=0}^{T}$ consisting of $T \in \mathbb{N}$ points sampled from a system, at times $\{t_i\}_{i=0}^{T}$. In order to train the RC to generate future values of the input time-series, we first generate a trajectory of the LR by numerically solving the initial value problem (IVP)

$$\dot{\mathbf{r}} = \gamma \left[-\mathbf{r} + \tanh\left(\rho \mathbf{M}\mathbf{r} + \sigma \mathbf{W}_{in}\mathbf{u}(t)\right)\right] \tag{6}$$

$$\mathbf{r}(0) = (0, ..., 0), \tag{7}$$

Where $\mathbf{M} \in \mathbb{R}^{N\times N}$ is the adjacency matrix with a sparse Erdos-Renyi topology and unit spectral radius describing hidden layer connections, $W_{in} \in \mathbb{R}^{N\times d}$ is the matrix which projects the input onto the reservoir nodes and γ, ρ and $\sigma \in \mathbb{R}$ are hyperparameters which control the timescale of the reservoir dynamics, scale the adjacency matrix and scale the input to the reservoir respectively. A schematic of the LR system is shown in figure 2 [6].

The function Ψ which we use to fit the reservoir state to the input signal is of the form

$$\hat{\Psi}(\mathbf{r}) = \mathbf{W}_{out} q(\mathbf{r}), \tag{8}$$

$$q(r) = (r_1, ..., r_N, r_1^2, ..., r_N^2), \tag{9}$$

where $\mathbf{W}_{out} \in \mathbb{R}^{N\times 2N}$ is the matrix whose entries are learned via regression. We use the function q here instead of just the vector r (which would make Ψ a linear function) to add an additional non-linearity to the RC, it also serves to break symmetries present in the RC update equations which can prove problematic when learning to replicate dynamics [19]. After discarding a number $T_{listen} \in \mathbb{N}$ of initial transient terms, we find the entries of the matrix $\mathbf{W}_{out}$ via ridge regression, that is we find the matrix by minimizing the error function

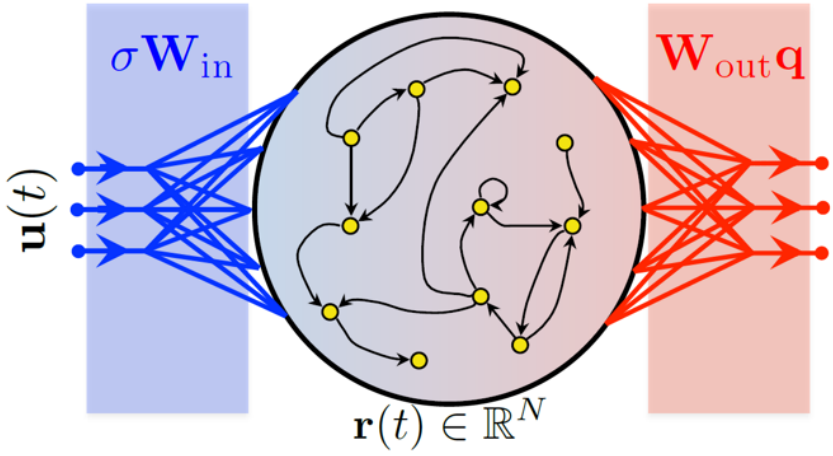

Figure 2: Listening Reservoir Architecture. Figure reproduced from [6]

$$\sum_{i=T_{listen}}^{T} \|\hat{\Psi}(\mathbf{r}(t_i)) - \mathbf{u}(t_i)\|^2 + \beta \|\mathbf{W}_{out}\|^2, \tag{10}$$

where $\beta \in \mathbb{R}$ is a regularization parameter which penalizes large weight values in $\mathbf{W}_{out}$ which are known to be associated with overfitting [6].

To solve this optimization problem we construct the following matrices

$$\mathbf{X} = \left(q(\mathbf{r}(t_{T_{listen}})), ..., q(\mathbf{r}(t_T))\right), \tag{11}$$

$$\mathbf{Y} = \left(\mathbf{u}(t_{T_{listen}}), ..., \mathbf{u}(t_T)\right), \tag{12}$$

then the matrix $\mathbf{W}_{out}$ which minimizes the cost function (12) is calculated as

$$\mathbf{W}_{out} = \mathbf{Y}\mathbf{X}^T \left(\mathbf{X}\mathbf{X}^T + \beta \mathbf{I}\right). \tag{13}$$

If regression is successful, then the function Ψ should satisfy

$$\Psi(\mathbf{r}(t_i)) \approx \mathbf{u}(t_i) \tag{14}$$

on the training data for $i \in \{T_{listen}, T_{listen} + 1, ..., T - 1, T\}$.

To make a prediction of future values $\hat{\mathbf{u}}$ of the input time series, we generate a trajectory of the PR system by numerically solving the IVP

$$\dot{\tilde{\mathbf{r}}} = \gamma \left[-\tilde{\mathbf{r}} + \tanh\left(\rho \mathbf{M}\tilde{\mathbf{r}} + \sigma \mathbf{W}_{in} \hat{\Psi}(\tilde{\mathbf{r}}(t))\right)\right], \tag{15}$$

$$\tilde{\mathbf{r}}(0) = \mathbf{r}(T), \tag{16}$$

then the points $\hat{\mathbf{u}}(t) = \hat{\Psi}(\tilde{\mathbf{r}}(t))$ are our predictions for future values of the input time series. A schematic of the PR system is shown in figure 3 [6].

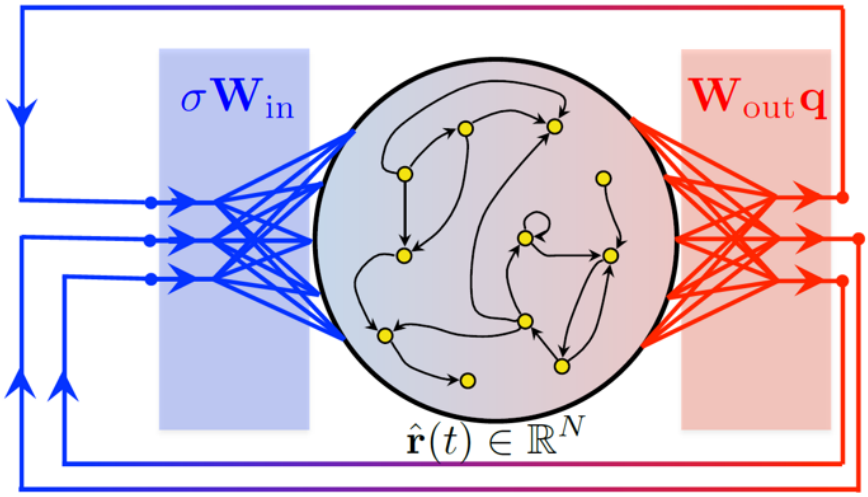

Figure 3: Predicting Reservoir Architecture. Figure reproduced from [6]

3.3.2 Time Series Prediction with an Echo-State Network

In this section we present the results of a numerical experiment whereby the future values of a trajectory taken from the Lorenz system are predicted using an ESN. This was carried out using a Python program which can be found in appendix A.

To generate a training signal we used the `scipy.integrate.odeint` function to integrate the Lorenz equations (2) from an IC $(x(0), y(0), z(0)) = (1, 1, 1)$ for 250 time units with a step size $dt = 0.01$. This yields a time series $\{\mathbf{u}(0), \mathbf{u}(0.01), \mathbf{u}(0.02)..., \mathbf{u}(150)\}$, where $\mathbf{u}(t) = (x(t), y(t), z(t))$. We use the first 100 time units for training, while the last 150 time units are reserved for validation of the prediction.

The hyperparameters for our ESN are chosen to be $N = 100, p = 0.01, \rho = 0.3, \sigma = 0.6, \gamma = 10$ and $\beta = 10^{-6}$. Both the LR and PR equations are numerically solved using the `scipy.integrate.odeint` function using step size 0.01. During training we discard the first 50 time units. The prediction is generated for 150 time units.

Figure 4.a shows a time trace of the predicted future and actual future values of the x, y and z variables, in it we can see that we get a good prediction for $t \approx 2.1$ time units, after which the prediction breaks down for the x and y variables but remains accurate for the z variable until $t \approx 5$ time units. Figure 4.b shows a plot of the predicted trajectory and actual future trajectory for $T = 150$ time units, in it we see that the long term predicted trajectory still resembles a typical trajectory evolving on the Lorenz attractor, such as those seen in figure 1, despite the fact that the prediction diverges from the actual future trajectory after 2 time units. In cases such as this, when the asymptotic behaviour of the prediction is qualitatively similar to that of a typical trajectory from the system our training signal was taken from, we say the ESN has achieved attractor reconstruction (AR).

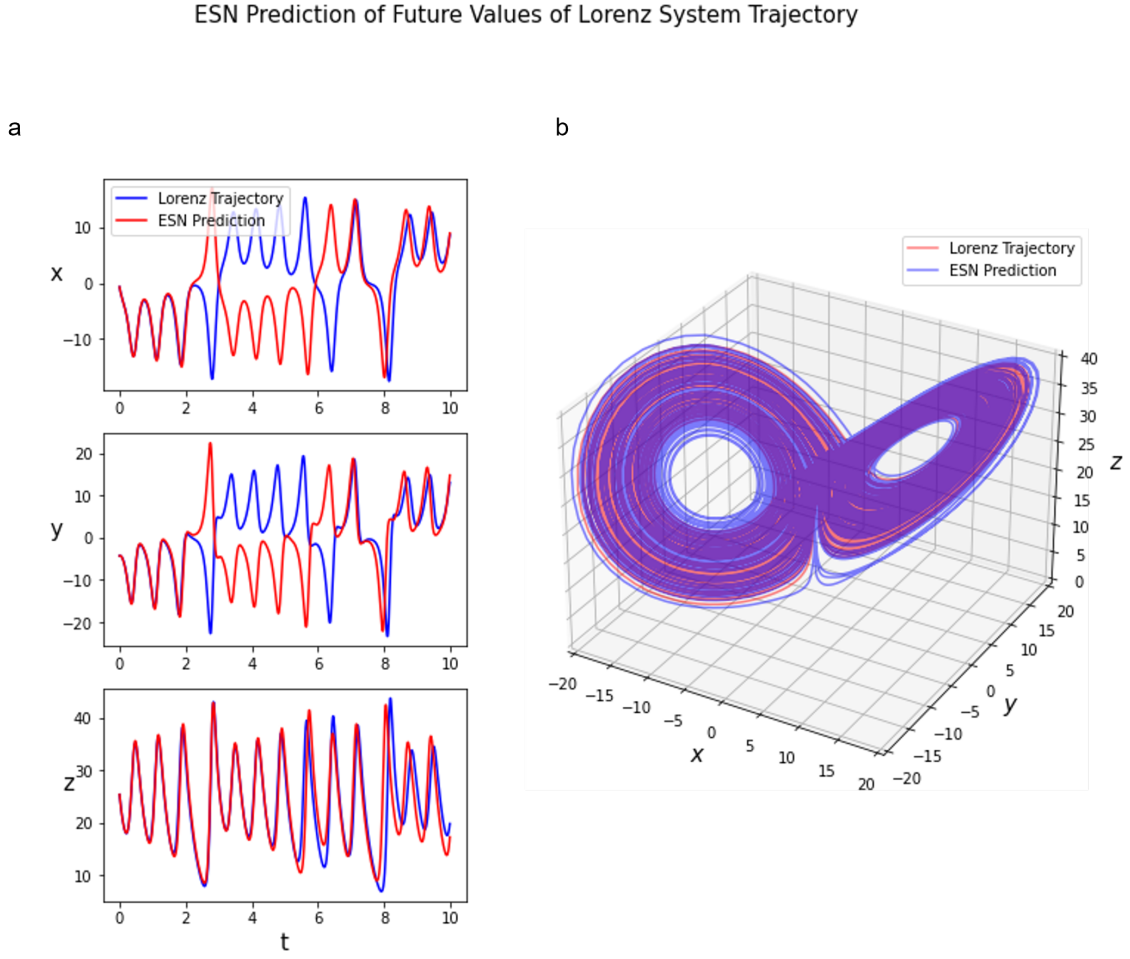

Figure 4: Prediction of Lorenz system trajectory for ESN with $N = 100, p = 0.01, \rho = 0.3, \sigma = 0.6, \gamma = 10$ and $\beta = 10^{-6}$. Panel a: time trace of x, y and z variables for ESN prediction and actual future of training trajectory, displayed in red and blue respectively. Panel b: Predicted and actual future trajectory plotted in red and blue respectively for $T = 150$ time units in $\mathbb{R}^3$

3.4 Advantages of Reservoir Computing

The RC paradigm was introduced as a method to train RNNs to predict future values of time series sampled from chaotic DSs in a way that avoided the BPTT algorithm, as this algorithm is known to suffer from the problem of vanishing and exploding gradients which causes training to be unsuccessful [4]. In the RC paradigm, it is not required to change the structure of the reservoir during training, making RC a comparatively simple and efficient method of replicating dynamics based on data compared to other popular machine learning methods such as long short-term memory (LSTM) networks [20]. Two of the central features of the RC paradigm are the non-autonomous LR system and the autonomous PR DSs. This formulation of RC in terms of DSs allows us to use the broad array of tools from DSs theory to analyse their behaviour, thus providing an opportunity to better understand problems in the comparatively new field of machine learning. It is true that other ML methods which train RNNs using BPTT such as LSTMs can also be formulated from a DSs perspective similar to what we have seen in sections 3.1 and 3.2. However the features of the systems which are introduced in order to avoid vanishing and exploding gradients during training can lead to much more complicated dynamical equations than we have seen in section 3.3.1. LSTMs provide an example of this, in order to overcome the vanishing / exploding gradients issue, LSTMs introduce the idea of gates which help control the flow of information when updating the states of hidden layers [21]. The resulting state update equations are much more complicated than those of the ESN presented above, making their analysis more difficult also. The comparatively simple form of the dynamical equations of an ESN makes them more amenable to analysis using the tool of DSs theory, making them excellent candidates for theoretical studies of the properties of RNNs [8][22].

3.5 Replication of Dynamics Through Attractor Reconstruction

We have previously shown in section 3.3.2 that RCs can learn to replicate the dynamics of a system from a trajectory sampled from the system. In this section we give a more detailed description of what it means to replicate dynamics, as well as present a theoretical mechanism by which this replication can occur.

Suppose our goal is to make a prediction of the future values of a time series sampled from a trajectory of a chaotic DS such as the Lorenz system introduced in section 2. Then even if we are successful in making an accurate short term prediction, the long term prediction is guaranteed to eventually diverge from the actual future of the training trajectory because of the sensitive dependence on ICs exhibited by a chaotic DS. In cases such as this, the future of our predicted trajectory may still resemble a typical trajectory evolving on the Lorenz attractor, we refer to this phenomenon as AR.

In 2018 Lu et al. presented a mechanism for how AR with a RC can occur based on the idea of generalized synchronization [6] which we outline here. Suppose our training signal is sampled from some DS whose governing equations are possibly unknown, meaning it satisfies

$$\dot{\mathbf{u}} = G(\mathbf{u}) \tag{17}$$

for some d dimensional vector field G. Then the LR can be equivalently reformulated as a unidirectionally coupled DS

$$\dot{\mathbf{u}} = G(\mathbf{u}), \tag{18}$$
$$\dot{\mathbf{r}} = F(\mathbf{r}, \mathbf{u}), \tag{19}$$

This system is referred to as unidirectionally coupled, since the evolution of $\mathbf{r}$ depends on that of the $\mathbf{u}$, but the evolution of u is independent of $\mathbf{r}$. We say that such a system exhibits generalised synchronization if there exists a continuous function $\Phi : \mathbb{R}^N \to \mathbb{R}^d$ satisfying

$$\lim_{t\to\infty} \|\mathbf{r}(t) - \Phi(\mathbf{u}(t))\| = 0, \tag{20}$$

meaning that for sufficiently large time, the trajectory of the LR exhibits an approximate functional dependence on the input signal, i.e. $\mathbf{r}(t) \approx \Phi(\mathbf{u}(t))$. Furthermore, if this function Φ is invertible when restricted to the input signal $\mathbf{u}(t)$

$$\mathbf{u}(t) \approx \Phi^{-1}(\mathbf{r}(t)). \tag{21}$$

In this case we can see that the regression procedure carried out during training is trying to learn an approximation of the function Φ^{-1}.

Now assume that $\mathbf{u}(t)$ is converging to an attractor A of the system (19), that is $\mathbf{u}(t) \to A$ as $t \to \infty$. Then in the limit as $t \to \infty$, $\mathbf{r}(t) \to \Phi(\mathbf{u}(t))$ and $\mathbf{u}(t) \to A$ which implies $\mathbf{r}(t) \to \Phi(A)$. So that the trajectory of the LR is driven towards an embedding of the attractor A in the LR phase space.

If training is successful, then we find a function Ψ satisfying $\Psi(\mathbf{r}(t)) \approx \mathbf{u}(t)$. If in addition the trajectory $\mathbf{u}(t)$ has evolved long enough to have well sampled the attractor A, then we have effectively learned a map from $\Phi(A)$ to A.

If this embedding of the attractor A into R^N is an attractor of the PR system, then the PR trajectory will continue to evolve on $\Phi(A)$, so that the predicted trajectory $\Psi(\mathbf{r})$ will continue to evolve on A even if the value of the prediction diverges from the actual value of the future of the input signal. There are various ways of measuring the success of AR by observing the PR system. Some measures are based on the ergodic properties of the PR system such as calculating the systems Lyapunov exponents [23] or entropy [24], while a simpler approach is to visually inspect the predicted trajectory and see if it resembles a typical trajectory evolving on attractor of the system whose dynamics we are trying to replicate. If we have access to the equations which generated the training signal then we can use the testing phase error (TPE) employed by Lu et al. in their 2020 paper [7] to measure how similar the predicted trajectory is to a typical trajectory of the target system.

3.5.1 Testing Phase Error

Suppose we have sampled a time series $\{\hat{\mathbf{u}}(t_i)\}_{i=1}^T$ from a trajectory of a system whose dynamics are given by

$$\dot{\mathbf{u}} = G(\mathbf{u}), \tag{22}$$

and generated a prediction of its future values $\{\hat{\mathbf{u}}(t_i)\}_{i=1}^T$. The idea of the TPE is to use the vector field which gives the dynamics of the target system G to measure how similar the predicted trajectory is to a trajectory of the target system. To do so we proceed as follows;

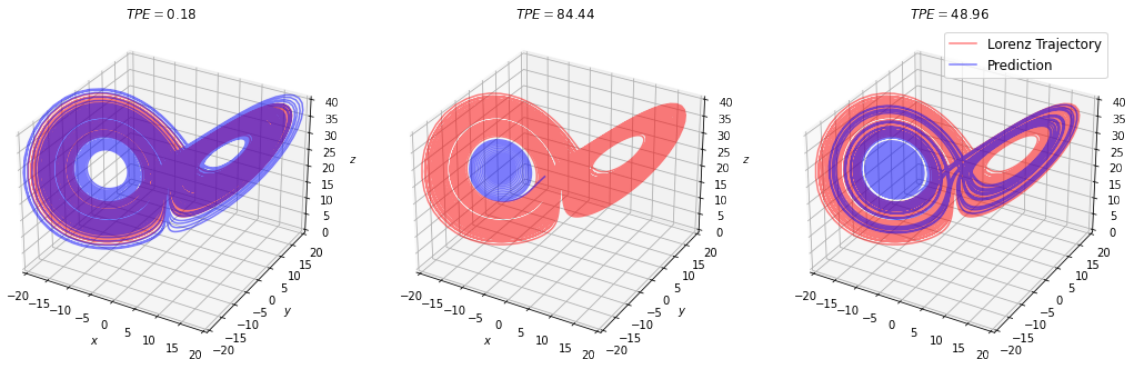

Figure 5: Plots of ESN predicted trajectories and actual future trajectories, in blue and red respectively, with their corresponding TPE. Each plot represents an ESN with different hyperparameter values.

1. For each point on the predicted trajectory, calculate the 'movement vector' for one time step as $\delta\hat{\mathbf{u}}(t_i) = \hat{\mathbf{u}}(t_{i+1}) - \hat{\mathbf{u}}(t_i)$.

2. Then for each point on the predicted trajectory, use the function G and a numerical ODE solver to calculate the next point on the trajectory through $\hat{\mathbf{u}}(t_i)$ after taking a step of size $t_{i+1} - t_i$. This gives us the 'ideal movement vector' $\delta\mathbf{u}(t_i)$.

3. The TPE is then defined as

$$TPE = \left\langle \frac{\|\delta\hat{\mathbf{u}}(t_i) - \delta\mathbf{u}(t_i)\|}{\|\delta\mathbf{u}(t_i)\|} \right\rangle_t, \tag{23}$$

   where $\langle\cdot\rangle_t$ denotes the time average along the predicted trajectory.

When the predicted trajectory $\delta\hat{\mathbf{u}}(t_i)$ evolves on the attractor A with the correct dynamics, the TPE is small (< 0.1). When the predicted trajectory escapes from A, the TPE can be very large [7]. Calculation of the TPE requires access to the dynamical equation which generated the input trajectory, which is information that obviously will not be available during any real world modelling exercise. However for studies attempting to study the properties and effects of machine learning algorithms such as this one, it is a useful and computationally lightweight tool for measuring the success of replication for dynamics. As such, the TPE is the main quantitative measure used for the success of AR. In this report, we consider a TPE less than 5 an indicator of successful AR. This cutoff point was decided by observing the output of the RC training method over a range of ρ, σ values and comparing it to the calculated TPE. It was observed that the vast majority of successful AR was associated with a TPE of 5 or less. Some examples of different predictions and their associated TPE can be seen in figure 5, in it we can see that the case where we have successful AR has a very low TPE ($TPE = 0.18$), while the other two examples which fail AR have much higher TPE ($TPE = 84.44, 49.96$)

Even if after training the attractor A is embedded as an attractor in the PR system, there is no guarantee that it is a global attractor. If this is the case and if the PR system is dissipative (as

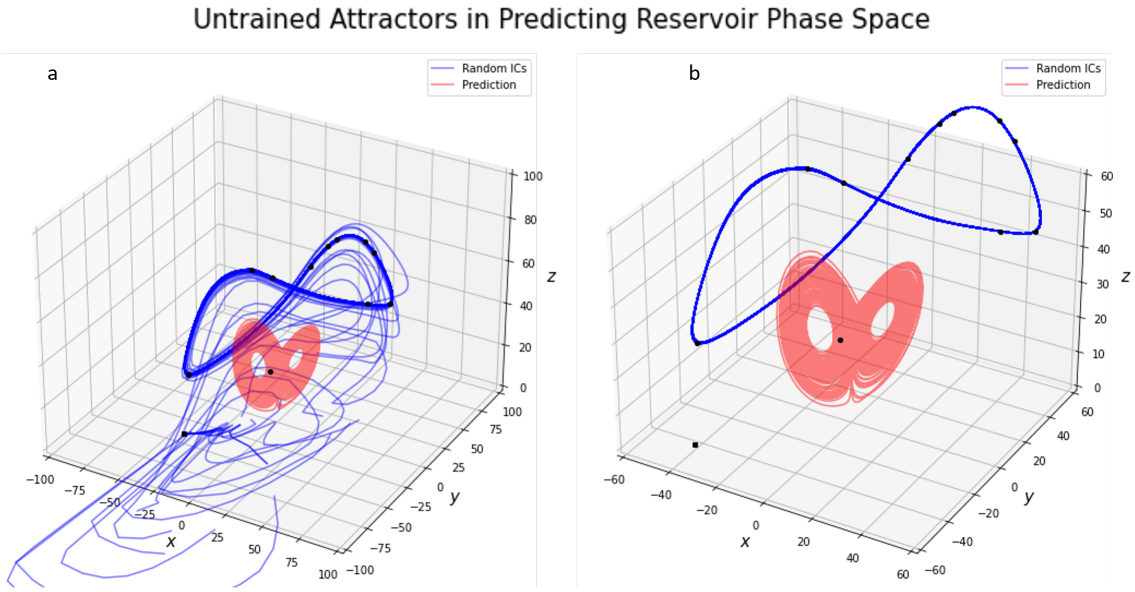

Figure 6: Plot of embedded Lorenz in coexistence with UAs present in PR phase space for ESN with $N = 100, p = 0.01, \rho = 0.3, \sigma = 0.6, \gamma = 10$ and $\beta = 10^{-6}$

is the case with the PR for an ESN), then the points of the PR phase space which are not in the basin of attraction of the embedded attractor must converge to some other attractor of the system. Such attractors which exist in the PR phase space and are different from the learned attractor are called UAs [8].

3.6 Exploring Untrained Attractors in an ESN

In this section we explore the UAs that are present in the PR phase space of an ESN. As a first step, we will revisit example 3.3.2, where an ESN was trained to replicate the dynamics of the Lorenz system. We saw that the ESN successfully reconstructed the Lorenz attractor, as the predicted trajectory still resembled a typical trajectory on the Lorenz attractor when evolved for a long time. To explore the state space of this ESN's PR system, we generate 20 random ICs in $\mathbb{R}^N$ and calculate their trajectories using equation (15) with `scipy.integrate.odeint` for 150 time units with a step size 0.01. We then projected the last 140 time units onto $\mathbb{R}^d$ using the function $\hat{\Psi}(\mathbf{r})$ to determine which ICs are evolving on the embedded Lorenz attractor in the PR phase space. The result of this computation can be seen in figure 6, which shows that all 20 random ICs converged to UAs, specifically to a fixed point and a limit cycle. This plot shows that even when AR is achieved as evidenced by the prediction tracing out the Lorenz attractor, there can still be UAs present in the PR phase space. Indeed, the fact that all 20 ICs converged to UAs is an indication that their basins of attraction are much larger than that of the embedded Lorenz.

In order to get an idea of how common UAs are in the PR state space of an ESN, even for choices of hyperparameters which led to AR, we repeat the same analysis carried out above for a collection of ESNs. To make the plots in figure 7 we construct an ESN for each $(\rho, \sigma) \in \{0.1, 0.2, ..., 0.8, 0.9\}^2$

with the hyperparameters $N = 100, p = 0.1, \gamma = 10$ and $\beta = 10^{-6}$ kept fixed for all ESNs. Then we train each ESN on the trajectory for the Lorenz system calculated in section 3.3.2. For each ESN we calculate the TPE for the prediction, classify the attractors present in the PR phase space and generate 20 random ICs and calculate the proportion which converge to the embedded Lorenz attractor of the PR system, we henceforth refer to this quantity as the reconstruction proportion (RP). In figure 7, each quantity is then used to make three colour maps over the grid of (ρ, σ) values. A detailed description of how these plots were made can be found in appendix D. In figure 7.a successful AR is indicated by the deep purple squares (Where log TPE ≈ 0), we see that AR is successful for most ESNs in the region $\sigma < 0.5$. Figure 7.b shows the RP, in it we see that in the region $\sigma < 0.5$ where AR was successful for the majority of ESNs, only about $20\% - 30\%$ of the ICs converge to the embedded Lorenz attractor in the PR phase space. Figure 7.c. shows that for the majority of ESNs in this region, these ICs are converging to a limit cycle.

In figure 7.a we see that the ESNs with $\sigma \geq 0.5$ have consistently poor AR, with most ESNs in this region having a very high log TPE (indicated by yellow squares). Comparison with figure 7.b in this region shows that AR has failed for most ESNs in this region, this is indicated by the fact that no ICs have converged to the embedded Lorenz (shown by the white squares).

One thing which is made especially clear by figure 7 is how prevalent a feature UAs are in the PR phase space of ESNs, which gives motivation to find techniques for removing them. One of the key features of RC is the fact that only the output layer weights are trained, so a natural way to try and improve training is to seek ways to optimize the internal structure of the RC with respect to a given data set. In this report, we take inspiration from the 2021 paper by Morales et al. [9] by optimizing the internal structure of an ESN using learning rules based on neuroplasticity, so that the resulting PR system replicates the dynamics of the Lorenz system with no UAs.

4 Plasticity Rules

Plasticity is a term used in neuroscience to refer to the brain's ability to alter its structure in response to stimuli in order to better complete tasks [25]. An example of such a structural change is a neuron's ability to alter the strength of synaptic connections to other neurons, this is known as SP and is linked with the amount of neurotransmitter released by a pre-synaptic neuron and the response this stimulates in a post-synaptic neuron. Another example is a neuron's ability to alter its inherent excitability, which is dependent on structural properties of the cell membrane, this is known as IP [9].

4.1 Synaptic Plasticity

The original mechanism for neural plasticity was first posited by Donald Hebb in his 1949 book 'The organization of behaviour'[25], where he states that synaptic connections in the nervous system form dynamically in response to external stimuli such that the connection between a pre-synaptic neuron and a post-synaptic neuron is strengthened if the pre-synaptic neuron firing tends to also precede the post-synaptic neuron firing. This mechanism is summarised by the popular phrase "neurons that fire together, wire together" [26].

There are many different mathematical implementations of Hebb's theory, known as Hebbian learning, with the most simple being

$$m_{ij}(t+1) = m_{ij}(t) + \eta_s r_j(t) r_i(t+1) \tag{24}$$

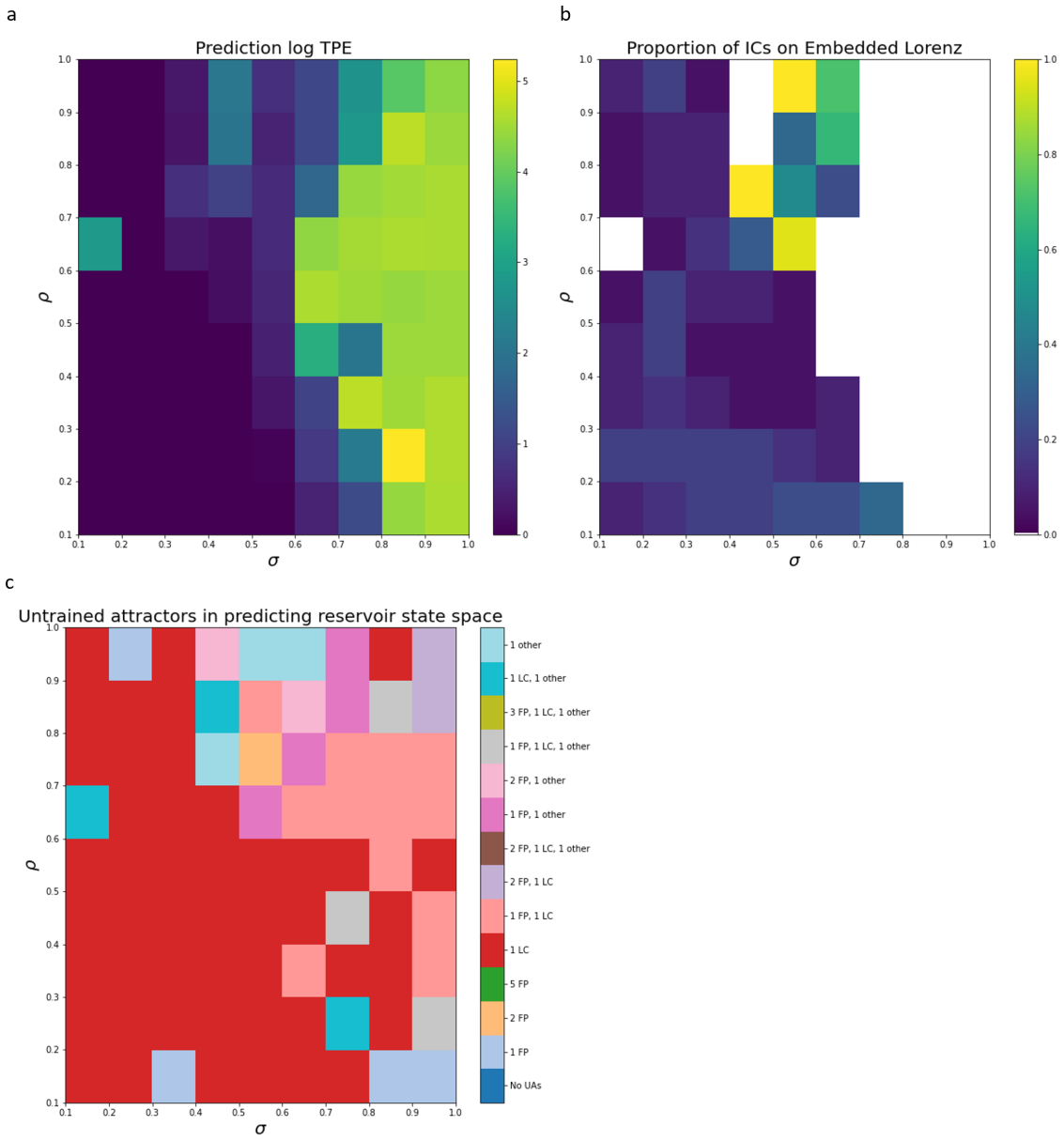

Figure 7: Panel a: Prediction log TPE for ESNs with $(\rho, \sigma) \in \{0.1, 0.2, ..., 0.8, 0.9\}^2$, $N = 100, p = 0.1, \gamma = 10$ and $\beta = 10^{-6}$. Panel b: RP for ESNs with $(\rho, \sigma) \in \{0.1, 0.2, ..., 0.8, 0.9\}^2$ with the hyperparameters $N = 100, p = 0.1, \gamma = 10$ and $\beta = 10^{-6}$. Panel c: UAs present in PR phase space for ESNs with $(\rho, \sigma) \in \{0.1, 0.2, ..., 0.8, 0.9\}^2$ with the hyperparameters $N = 100, p = 0.1, \gamma = 10$ and $\beta = 10^{-6}$.

where m_{ij} is the weight of the connection from neuron j to neuron i, $r_j(t)$ is the value of the pre-synaptic neuron and $r_i(t+1)$ is the value of the post-synaptic neuron and $\eta_s > 0$ is the learning rate [25]. An obvious flaw with this update rule is that it can cause connection weights to become arbitrarily large if iterated enough. One way of circumventing this problem is by scaling the connection weights at each update, so that our update rule is given instead by

$$m_{ij}(t+1) = \frac{m_{ij}(t) + \eta_s r_j(t) r_i(t+1)}{\sqrt{\sum_{k=1}^{N} (m_{ik}(t) + \eta r_k(t) r_i(t+1))^2}}. \tag{25}$$

While this rule avoids the problem of uncontrolled growth of connection weights, it is non-local in the sense that updating the strength of one synapse requires knowledge of other synapse strengths, which is both biologically implausible and computationally less efficient [25].

By changing the sign of the learning rate η_s in the formulas above, we get so-called 'anti-Hebbian' learning rules. The key difference with these rules is that they decrease the weight m_{ij} of the connection from a pre-synaptic neuron r_j to a post-synaptic neuron r_i if there is a positive correlation between the activity of $r_j(t)$ and $r_i(t+1)$. Similarly the weight is increased if there is a positive correlation between the activity of $r_j(t)$ and $r_i(t+1)$. This change in sign of the learning rate can then act as a mechanism to decorrelate the neuron activity induced by a stimulus [9]. We use such anti-Hebbian learning rules in the remainder of this work.

4.1.1 Synaptic Plasticity Rule applied to an Echo-State Network

We can use the learning rules discussed above to augment the ESN training method outlined in section 3.3.1. In order to do so we proceed as in in Morales et al. [9],

1. Drive the ESN with an input signal $\{\mathbf{u}(t_i)\}_{i=0}^{T}$ to generate a sequence of reservoir states $\{\mathbf{r}(t_i)\}_{i=0}^{T}$,

2. For each entry m_{ij} in the adjacency matrix M, iterate

$$m_{ij}(t+1) = m_{ij}(t) + \eta_s r_j(t) r_i(t+1) \tag{26}$$

   for $t \in \{t_i\}_{i=0}^{T}$ where $m_{ij}(0)$ is the corresponding entry in the Erdos-Renyi matrix M generated when we first create the ESN.

Steps 1 and 2 are then repeated for a number of epochs, which we denote $epochs_s$, before the ESN is driven with the input one final time, the resulting reservoir states are then used to learn the output matrix W_{out} as before. We refer to these ESNs, whose training has been augmented to include the above SP rule, as synaptically plastic echo-state networks (SP-ESNs)

4.2 Intrinsic Plasticity

IP refers to an individual neurons ability to change its excitability by altering the properties of ion channels embedded in the cell membrane which facilitate neuron firing [27]. The IP rule presented here is inspired by the idea that neurons try to maximize information transmission while minimizing energy consumption [9]. This is an important property for neurons to have considering the already large energy consumption of the brain, consuming approximately 20% of the body's energy in a

resting state, though it accounts for just 2% of the body's mass [28]. The mathematical learning rule presented here leads to maximum entropy distributions for the neurons output and is derived by minimizing the Kullback-Liebler divergence of the distribution of neuron states with respect to a target Gaussian distribution with mean μ and standard deviation s [9].

4.2.1 Intrinsic Plasticity Rule applied to an Echo-State Network

In order to apply the IP learning rule to the ESN we proceed using the method in Morales et al. [9], we first augment the ESN update equations to include a gain a_i and bias term b_i so that the dynamics for a node r_i are given by

$$\dot{r}_i(t) = \gamma \left[-r_i(t) + \tanh\left(a_i z_i(t) + b_i\right) \right], \tag{27}$$

$$z_i(t) = \rho \sum_{k=1}^{N} m_{ij} r_k(t) + \sigma \sum_{j=1}^{N} w_{in}^{(ij)} u_j(t). \tag{28}$$

Then minimizing the Kullback-Liebler divergence is equivalent to carrying out the following supervised learning rule

1. Drive the ESN with an input signal $\{\mathbf{u}(t_i)\}_{i=0}^{T}$ to generate a sequence of reservoir states $\{\mathbf{r}(t_i)\}_{i=0}^{T}$,
2. for $t \in \{t_i\}_{i=0}^{T}$ we update the gain and bias terms a_k, b_k from the initial values $a_k = 0$, $b_k = 0$ via the following update equations,

$$\Delta b_k(t) = -(-\frac{\mu}{s^2} + \frac{r_k(t)}{s^2}(2s^2 + 1 - r_k(t)^2 + \mu r_k(t))), \tag{29}$$

$$\Delta a_k(t) = \frac{1}{a_k(t)} + \Delta b_k(t) z_k(t), \tag{30}$$

$$b_k(t+1) = b_k(t) + \eta_i \Delta b_k(t), \tag{31}$$

$$a_k(t+1) = a_k(t) + \eta_i \Delta a_k(t), \tag{32}$$

where μ and σ are the mean and standard deviation of a target normal distribution respectively (here we choose $\mu = 0$ and $\sigma = 0.5$) and $\eta_i > 0$ is the learning rate. As was the case with the SP rule, steps 1 and 2 are then repeated for a number of epochs, which we denote $epochs_i$ before the ESN is driven with the input one final time. The resulting reservoir states are then used to calculate the output matrix W_{out} using the method described in section 3.3.1. We refer to these ESNs, whose training has been augmented to include the above IP rule, as intrinsically plastic echo-state networks (IP-ESNs).

4.3 Combining Plasticity Rules

In this report we also consider the combination of the above IP and SP rule, specifically we consider the two configurations used in Morales et al. [9]. In the first configuration we apply the SP rule to adjust the weights of the adjacency matrix $\mathbf{M}$, then apply the IP rule, we call these ESNs SP-IP

ESNs. In the second, we first apply the IP rule, then the SP rule, we refer to these ESNs as IP-SP ESNs. We refer to all ESNs whose training has been augmented by including one or more plasticity rules as plastic ESNs (PESNs). It should be noted that by introducing plasticity rules which augment the adjacency matrix weights or gain and bias of nodes, we are altering the structure of the central system during training, meaning the resulting RNN is technically no longer a RC. However, the resulting training methods which include SP and or IP rules still train the RNN in a computationally lightweight way that avoids the BPTT algorithm. In this sense, the PESNs are still very much in the spirit of RC, which is our rationale for comparing PESNs to ESNs as there is still great insight to be gained from such an approach.

5 Results

The aim of this report is to investigate the effect of synaptic and IP rules when they are used in conjunction with the ESN training method outlined in section 3.5.1. Here we investigate using the four configurations of PESNs, using just the SP rule, using just the IP rule, then using the SP rule followed by the IP rule and finally the IP rule followed by the SP rule.

Specifically, for each combination of plasticity rules we train a set of PESNs to replicate the dynamics of the Lorenz system from the same sampled trajectory. The set of PESNs is indexed by values of the hyperparameters ρ, σ which form an evenly spaced grid with $\rho, \sigma \in \{0.1, 0.2, ..., 0.9\}$. We carry out this computation to see if a single choice of hyperparameters can be applied to different RCs but still successfully remove UAs present in the ESN PR phase space.

For each trained PESN, we do the following

1. Calculate the log TPE for predicted future values of the training signal
2. Characterize the number and type of UAs present in the PR phase space
3. Calculate the RP, the proportion of a set of randomly chosen ICs in the PR phase space whose trajectories converge to the embedded Lorenz attractor

For each configuration of PESNs, we use these three quantities to generate three plots over the grid of (ρ, σ) values defined above which we use to analyse the ability of the plasticity rules to remove UAs.

For each configuration of plasticity rules, we use the same values of the parameters $\eta_s, \eta_i, epochs_i$ and $epochs_s$. The details of how these parameter values were chosen is detailed in appendix E. The python code which implements ESNs and PESNs can be found in appendix A.

5.1 Generating Training Data

To generate a training signal we used the `scipy.integrate.odeint` function to integrate the Lorenz equations **??** from an IC $(x(0), y(0), z(0)) = (1, 1, 1)$ for 150 time units with a step size $dt = 0.01$. This yields a time series $\{\mathbf{u}(0), \mathbf{u}(dt), \mathbf{u}(2dt)..., \mathbf{u}(150)\}$, where $\mathbf{u}(t) = (x(t), y(t), z(t))$. This time series is used as the training signal for for all ESNs considered in this section.

5.2 Synaptic Plasticity Rule

For the ESN using only the SP rule we chose the parameter values $\eta_s = 10^{-6}$ and $epochs_s = 10$. Figure 8 shows the evolution of the adjacency matrix M after each epoch of training. As the number

of epochs increases we see a faint regular pattern emerging, indicating that the rule is changing many weights which were previously 0 to have small nonzero values. By applying the SP rule, we are changing the weights of M based on the values of the LR state when driven with the training signal $\mathbf{u}(t)$. This means the entries of M are dependant on the training signal $\mathbf{u}(t)$ after applying the SP rule, so the matrix M has become a representation of the training data.

Figure 9 shows the covariance matrix of the LR trajectory $\mathbf{r}(t)$ after each epoch of training with the SP rule. After applying the rule, the covariance between the variables $r_i(t)$ has been significantly reduced. This is how the rule improves prediction performance, by decorrelating the state variables $r_i(t)$, the ESN has a greater diversity of signals with which to reconstruct the training signal. We comment that in our experiments we found this mechanism could also lead to overfitting if training is carried out for too many epochs. Comparing the covariance matrices and the matrix M after training with the SP rule shows that the SP rule drives the weights of the connections between two nodes opposite the direction of their covariance. We see this most clearly by comparing the first matrix of 9 and the last matrix of 8, where the emerging pattern in the plot of M is what we would get if we multiplied the covariance matrix by -1.

Figures 10, 11 and 12 show the prediction log TPE, the RP and the attractors present in the PR phase space, respectively, of SP ESNs with varying values of ρ and σ. In each figure, the left panel displays the corresponding plot for the standard ESN for comparison.

On visual inspection of figure 10 we see that the quality of attractor replication is improved or kept the same when the SP rule is applied to a given ESN. Specifically we see the log TPE ≈ 0 for the majority SP ESNs with $(\rho, \sigma) \in [0.3, 0.5] \times [0.3, 0.9]$, which highlights the success of the SP rule. This is an improvement over the ESNs with these values of (ρ, σ), about half of which failed to reconstruct the attractor (indicated by the non purple squares).

In figure 11 it can be seen that only the ESNs $(\rho, \sigma) = (0.4, 0.7)$ or $(0.5, 0.9)$ have a RP of 1, as indicated by the yellow squares over these points, showing that there are no UAs for these ESNs. The RP is 0 for ESNs with $(\rho, \sigma) = (0.1, 0.6), (0.4, 0.8)$, $(0.4, 0.9)$ and the majority of ESNs with $(\rho, \sigma) \in [0.6, 0.9] \times [0.1, 0.9]$, which indicates that AR was unsuccessful in this region. In figure 11 we also see there is a significant improvement with respect to UAs after applying the SP rule, with many values of (ρ, σ) which had UAs for the ESN having no detected UAs for the SP ESN. This can be seen by the yellow squares in the SP ESN which were not yellow in the ESN plot in the regions $(\rho, \sigma) \in [0.1, 0.2] \times [0.3, 0.6]$ and $(\rho, \sigma) \in [0.3, 0.7] \times [0.1, 0.9]$. In a small amount of cases, applying the SP rule caused some ICs to converge to the embedded Lorenz where no ICs converged to it when applying the ESN, an example of this can be seen in the region $[0.6, 0.7] \times [0.4, 0.6]$, which contains coloured squares in the SP ESN plot and white squares in the ESN plot. This shows that applying the SP rule to the ESN was able to correct failed AR in these cases.

In figure 12, we can see that in almost all the cases where the UAs were removed, the only UA was an LC. In many cases where the SP rule was unable to remove UAs, the UAs present were still altered in some way. For example the region $[0.6, 0.9] \times [0.6, 0.7]$ in the ESN plot whose UAs are mostly a fixed point and a limit cycle, have either a fixed point or a fixed point and an attractor of type 'other', which is usually a poor reconstruction of the Lorenz or a periodic orbit with multiple distinct local max/min values in a single period. This shows that applying the SP rule to the ESN has a global effect on the dynamics of the PR system for all ESNs considered.

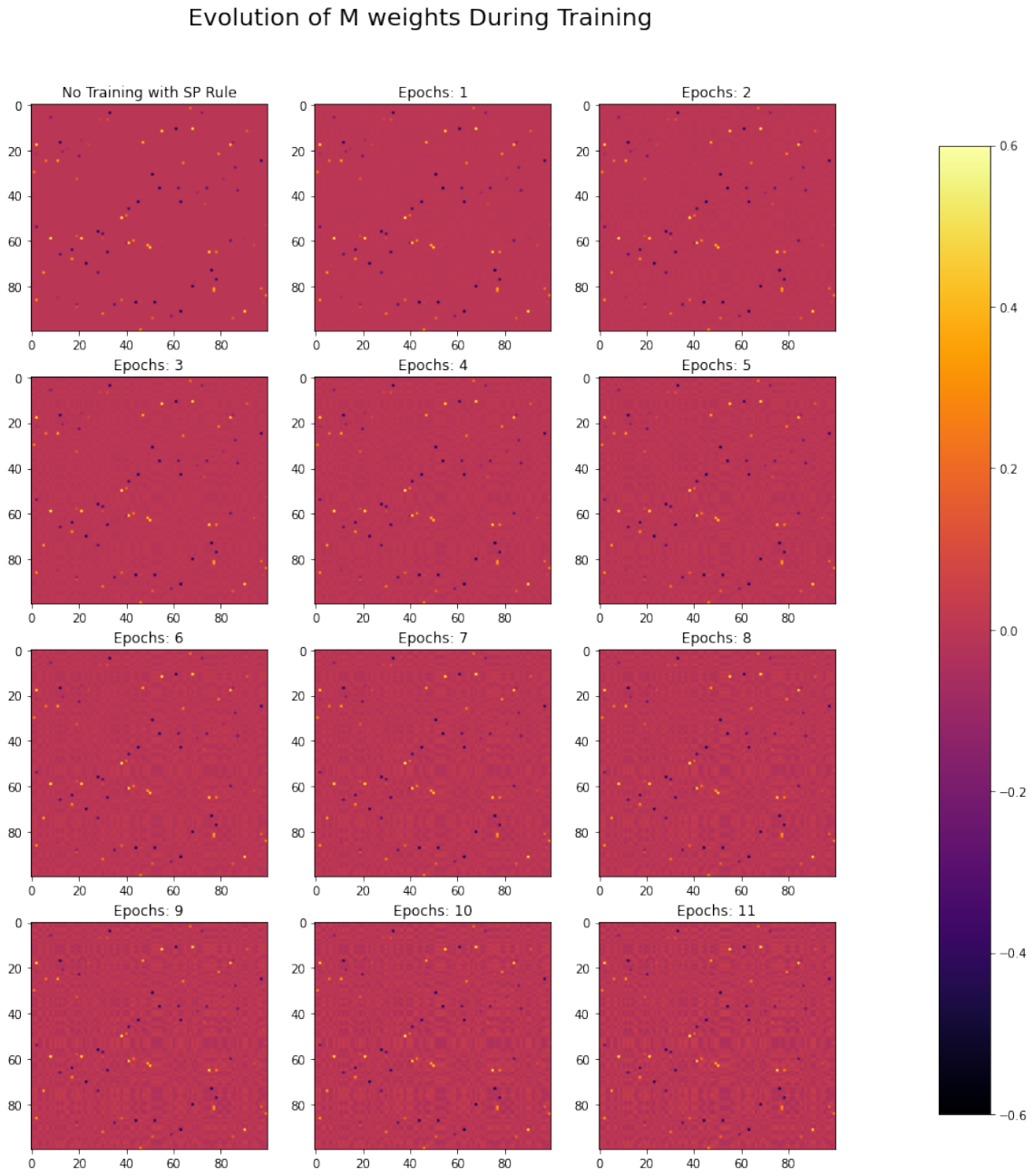

Figure 8: Evolution of adjacency matrix $\mathbf{M}$ weights over 10 epochs for SP ESN with $N = 100, p = 0.1, \rho = 0.3, \sigma = 0.6, \gamma = 10, \beta = 10^{-6}$ and $\eta_s = 10^{-6}$.

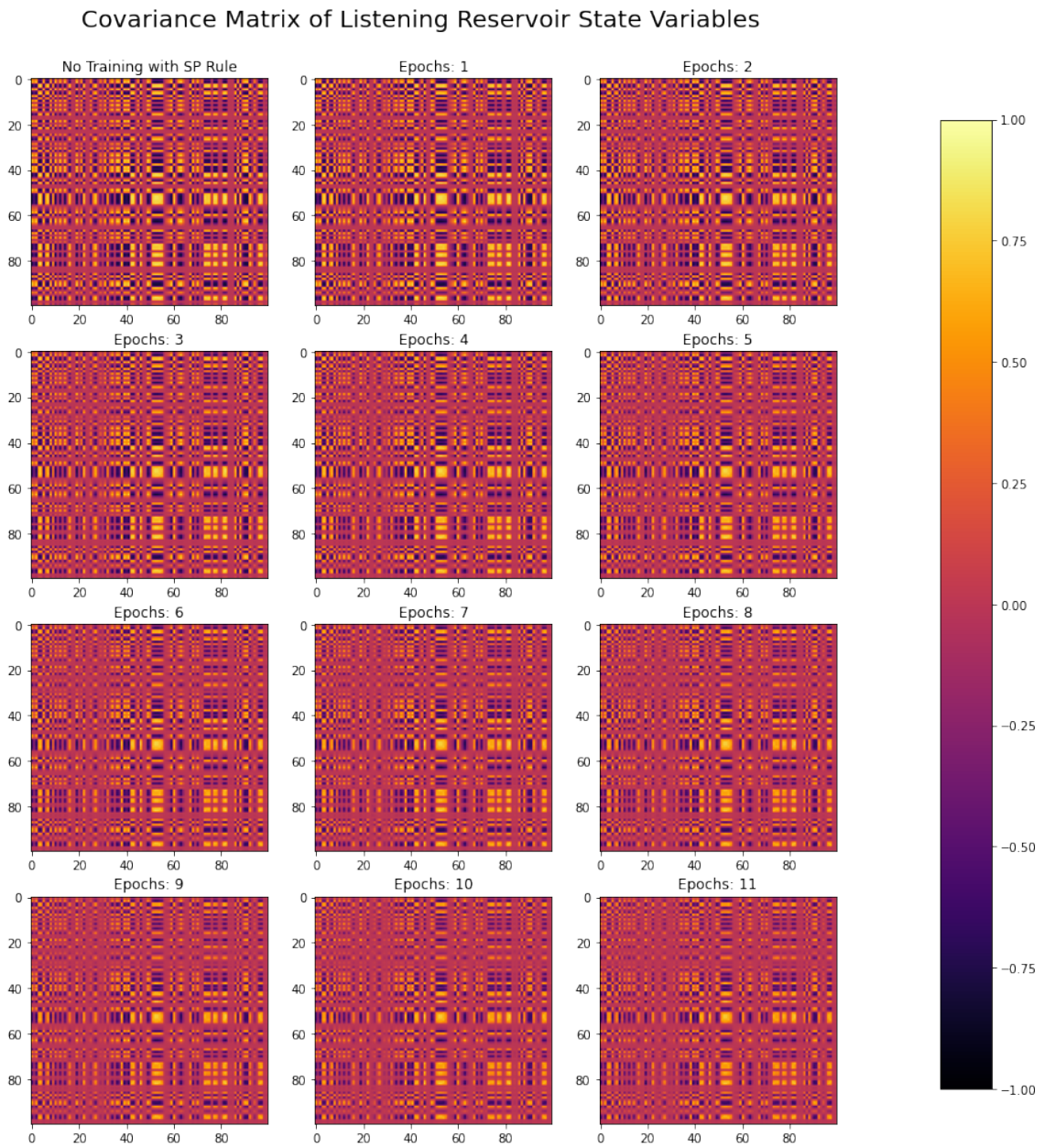

Figure 9: Evolution of covariance matrix for LR state over 10 epochs for SP ESN with $N = 100, p = 0.1, \rho = 0.3, \sigma = 0.6, \gamma = 10, \beta = 10^{-6}$ and $\eta_s = 10^{-6}$.

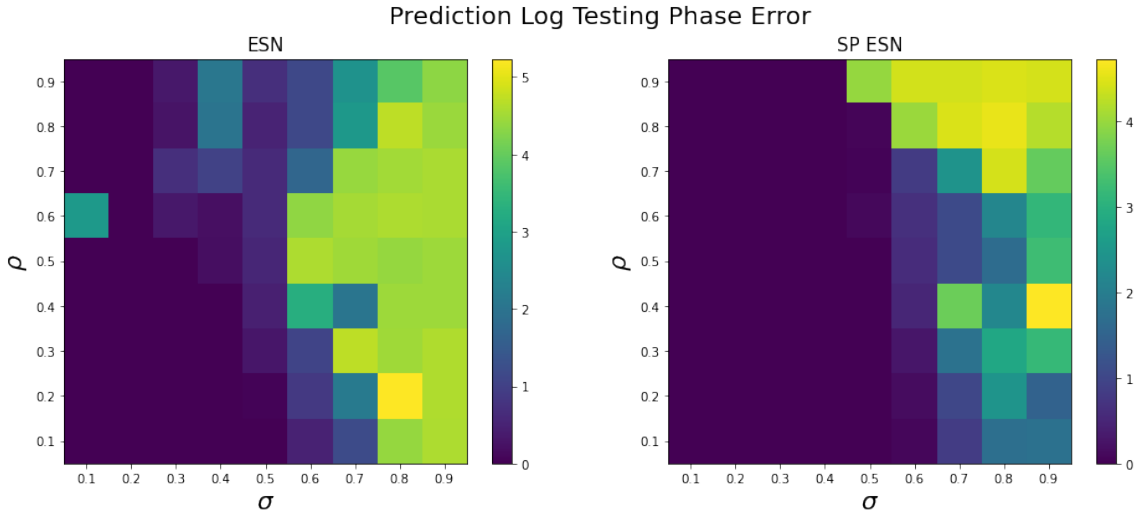

Figure 10: Prediction log TPE for ESNs (left) and SP ESNs (right) with $(\rho, \sigma) \in \{0.1, 0.2, ..., 0.8, 0.9\}^2$, $N = 100, p = 0.1, \gamma = 10, \beta = 10^{-6}$, SP ESN also has $\eta_s = 10^{-6}, epochs_s = 10$.

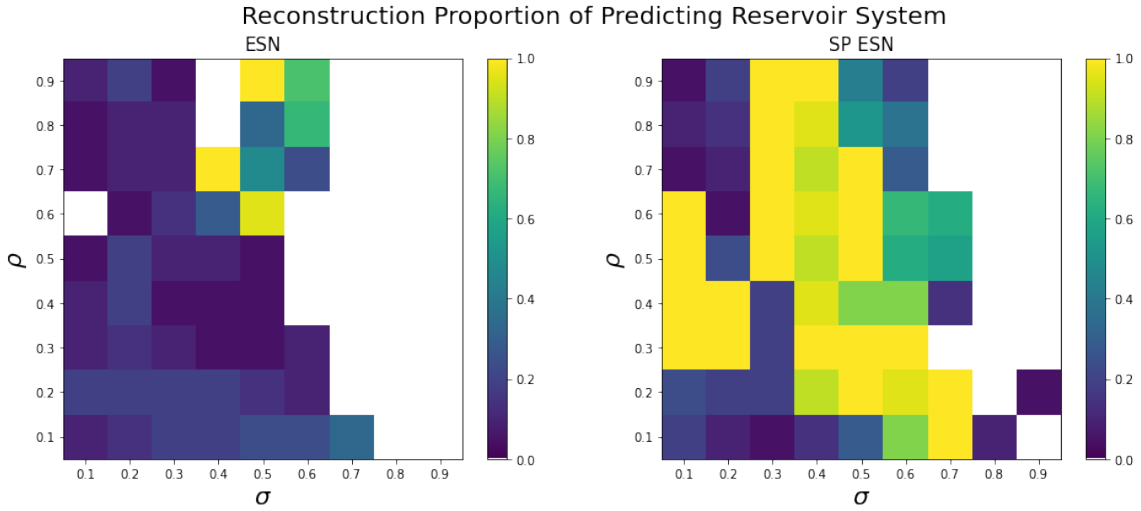

Figure 11: RP for ESNs (left) and SP ESNs (right) with $(\rho, \sigma) \in \{0.1, 0.2, ..., 0.8, 0.9\}^2$, $N = 100, p = 0.1, \gamma = 10, \beta = 10^{-6}$, SP ESN also has $\eta_s = 10^{-6}, epochs_s = 10$.

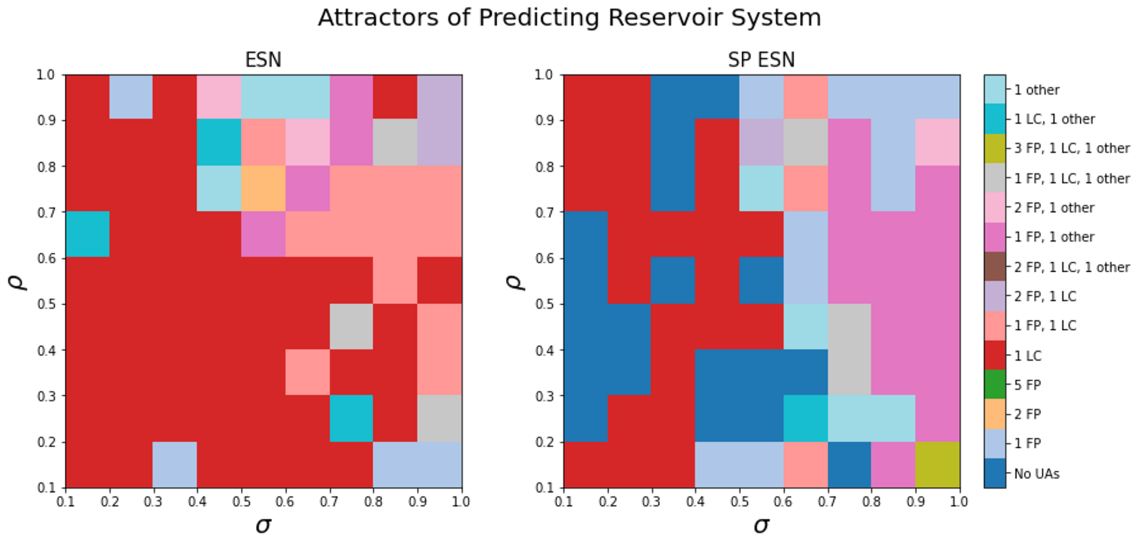

Figure 12: Attractors present in PR phase space for ESNs (left) and SP ESNs (right) with $(\rho, \sigma) \in \{0.1, 0.2, ..., 0.8, 0.9\}^2$, $N = 100, p = 0.1, \gamma = 10, \beta = 10^{-6}$, SP ESN also has $\eta_s = 10^{-6}, epochs_s = 10$. Legend abbreviations as follows; FP: fixed point, LC: limit cycle, UA: untrained attractor.

5.3 Intrinsic Plasticity Rule

For the ESN using only the IP rule we chose the parameter values $\eta_i = 10^{-5}$ and $epochs_i = 4$. Figure 13 shows the evolution of the vectors of gain and bias values **a** and **b** for each training update over 10 epochs, we see that the gain and bias terms eventually converge to fixed values.

The IP rule tries to update the gain and bias terms so that the resulting network's node values approximately follow a normal distribution with a given mean and standard deviation (here chosen to be 0 and 0.5 respectively). Figure 14 shows a histogram of LR state values before and after applying the IP rule, the distribution goes from being bimodal about -1 and 1 to being approximately normal. Figures 13 and 14 show that when applying the IP rule, the gain and bias converge to values such that the node states of the resulting LR system follow a normal distribution. This shows how the IP rule improves performance, if the reservoir node values are normally distributed, as opposed to being heavily bimodal about -1 and 1, then the range of signals available to reconstruct the training signal is greater.

Figures 15, 16 and 17 show the prediction log TPE, the RP and the attractors present in the PR phase space, respectively, of IP-ESNs with varying values of ρ and σ. In each figure, the left panel displays the plot for the standard ESN for comparison.

Figure 15 shows that the quality of AR is improved when applied to the majority of ESNs. In particular we can see that log TPE ≈ 0 for $(\rho, \sigma) \in [0.2, 0.9] \times [0.2, 0.9]$, indicating successful AR in the region. The exception being the IP ESNs in the region $\rho = 0.1$ and the region $\sigma = 0.1$, where the quality of AR was worse for SP ESNs than was the case for ESNs.

In figure 16 it can be seen that after applying the IP rule, there is a much greater set of (ρ, σ) pairs which have RP $= 1$, indicating that there are no UAs for these IP ESNs. However AR failed for IP ESNs with $\rho \in [0.1, 0.4]$ and $\sigma = 0.1$ whereas ESNs in this region were successful in carrying

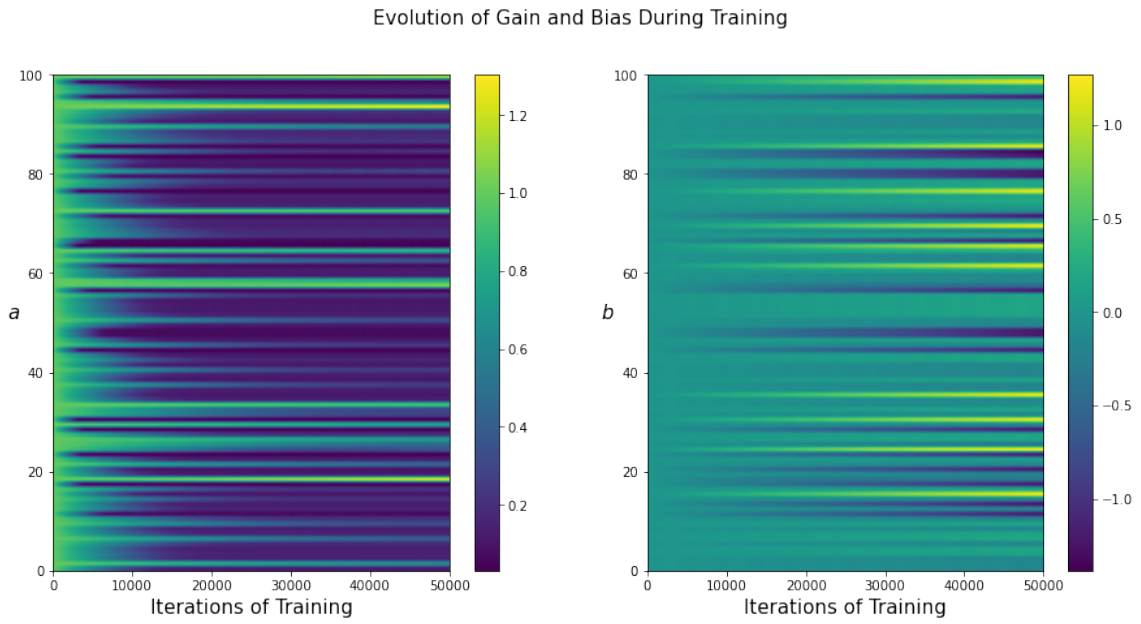

Figure 13: Evolution of gain (left) and bias (right) of IP ESN for each update in training over 10 epochs. IP ESN hyperparameters are $N = 100, p = 0.1, \rho = 0.3, \sigma = 0.6, \gamma = 10, \beta = 10^{-6}$ and $\eta_i = 10^{-5}$.

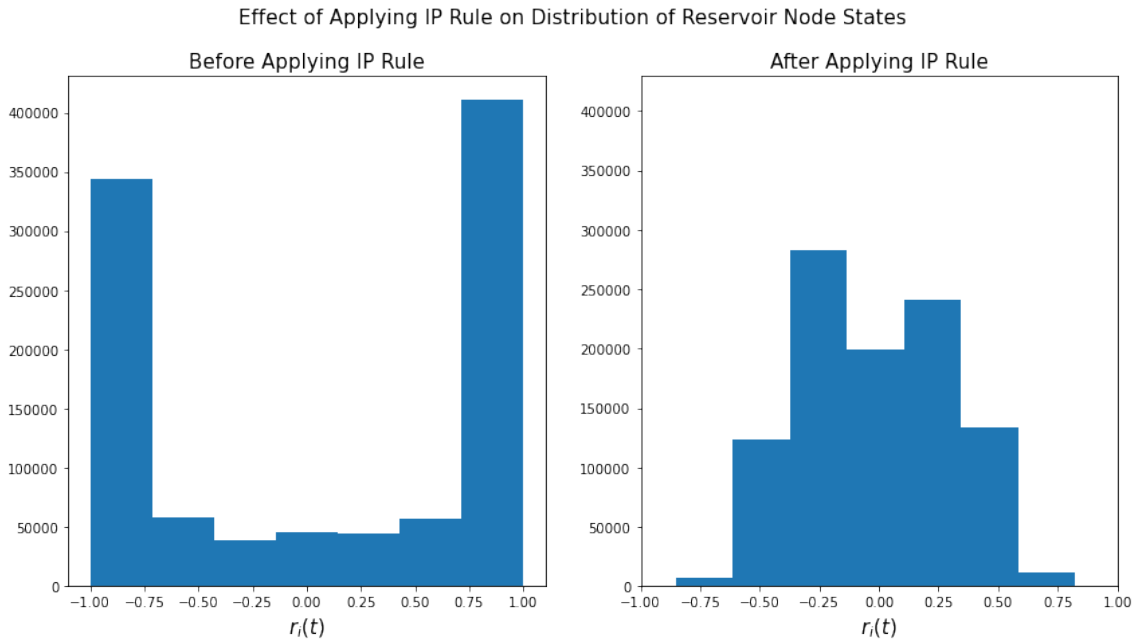

Figure 14: Histogram of LR state values during training before (left) and after (right) applying the IP rule. IP ESN hyperparameters are $N = 100, p = 0.1, \rho = 0.3, \sigma = 0.6, \gamma = 10, \beta = 10^{-6}, \eta_i = 10^{-5}$ and $epochs_i = 4$.

Figure 15: Prediction log TPE for ESNs (left) and IP ESNs (right) with $(\rho, \sigma) \in \{0.1, 0.2, ..., 0.8, 0.9\}^2$, $N = 100, p = 0.1, \gamma = 10, \beta = 10^{-6}$, SP ESN also has $\eta_i = 10^{-5}, epochs_s = 4$.

out AR.

In figure 17, we see examples where the IP rule was successful at removing limit cycles and fixed points. With the exception of a few cases, the IP rule has changed the configuration of the UAs in the PR phase space. In most of the cases where the IP rule was able to remove UAs, the corresponding ESN had only a limit cycle as an UA. There are also multiple cases where the PR of the ESN had a limit cycle as the only UA, but the PR of the corresponding IP ESN had a fixed point as the only UA, such as $(\rho, \sigma) \in [0.2, 0.9] \times \{0.1, 0.9\}$.

5.4 Intrinsic, Synaptic Plasticity Rule

For the ESN using the SP followed by the IP rule we chose the parameter values $\eta_s = 10^{-7}$, $epochs_s = 4$, $\eta_i = 10^{-5}$ and $epochs_i = 5$.

Figures 18, 19 and 20 show the prediction log TPE, the RP and the attractors present in the PR phase space, respectively, of IP-SP ESNs with varying values of ρ and σ. In each figure, the left panel displays the plot for the standard ESN for comparison.

In figure 19 we see that the quality of AR is improved when the IP rule, followed by the SP rule is applied to the majority of ESNs. This is shown by the log TPE ≈ 0 for $(\rho, \sigma) \in [0.2, 0.9] \times [0.2, 0.9]$, indicating successful AR for these IP-SP ESNs. However the IP-SP ESNs in the region $\rho = 0.1$ and the region $\sigma = 0.1$ are outperformed by most ESNs in this region.

Figure 20, shows that the IP-SP rule was able to remove limit cycles and fixed points. Additionally, we see that the majority of ESNs have had their configuration of UAs changed after applying the IP-SP rule. Most of the cases where the IP rule was able to remove UAs, the corresponding ESN had only a limit cycle as an UA. We also see cases where the PR of the ESN had a limit cycle as the only UA, but the PR of the corresponding IP ESN had a fixed point as the only UA, such as $(\rho, \sigma) \in [0.2, 0.9] \times \{0.1, 0.9\}$.

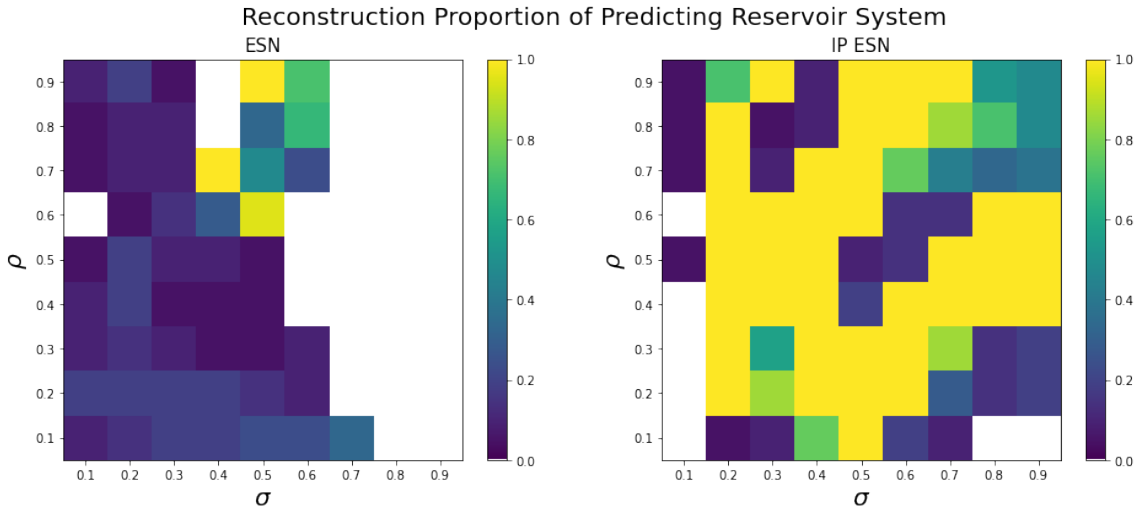

Figure 16: RP for ESNs (left) and IP ESNs (right) with $(\rho, \sigma) \in \{0.1, 0.2, ..., 0.8, 0.9\}^2$, $N = 100, p = 0.1, \gamma = 10, \beta = 10^{-6}$, SP ESN also has $\eta_i = 10^{-5}, epochs_i = 4$.

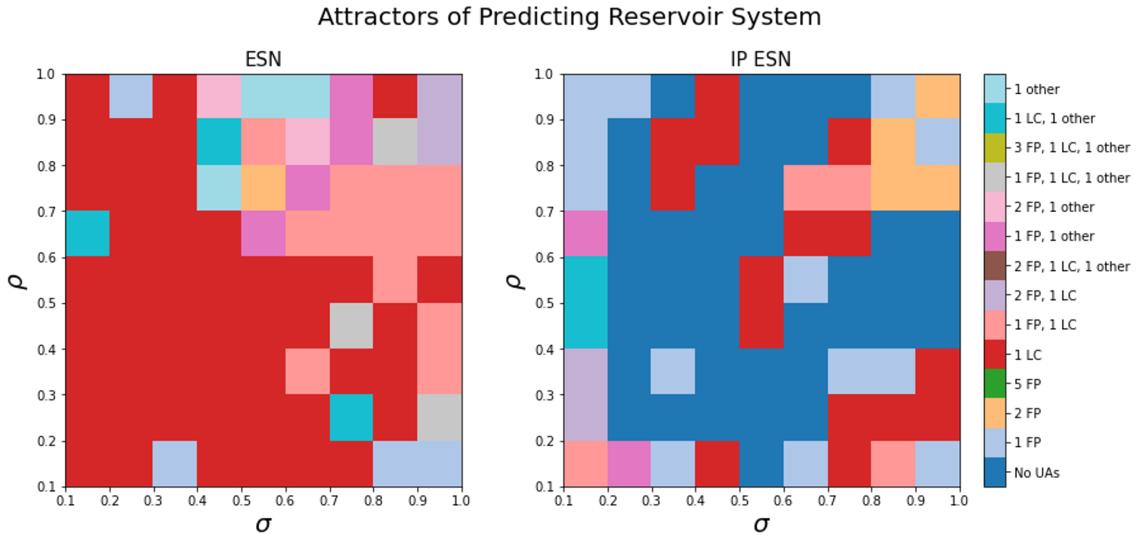

Figure 17: Attractors present in PR phase space for ESNs (left) and IP ESNs (right) with $(\rho, \sigma) \in \{0.1, 0.2, ..., 0.8, 0.9\}^2$, $N = 100, p = 0.1, \gamma = 10, \beta = 10^{-6}$, SP ESN also has $\eta_i = 10^{-5}, epochs_i = 4$. Legend abbreviations as follows; FP: fixed point, LC: limit cycle, UA: untrained attractor.

Figure 18: Prediction log TPE for ESNs (left) and IP-SP ESNs (right) with $(\rho, \sigma) \in \{0.1, 0.2, ..., 0.8, 0.9\}^2$, $N = 100, p = 0.1, \gamma = 10, \beta = 10^{-6}$, IP-SP ESN also has $\eta_s = 10^{-7}$, $epochs_s = 4$, $\eta_i = 10^{-5}$ and $epochs_i = 5$.

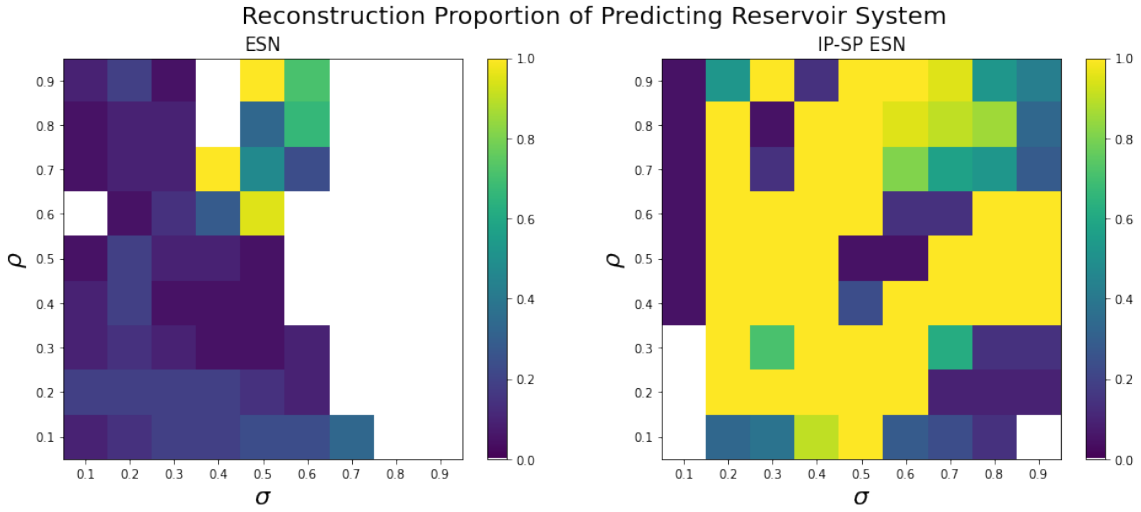

Figure 19: RP for ESNs (left) and IP-SP ESNs (right) with $(\rho, \sigma) \in \{0.1, 0.2, ..., 0.8, 0.9\}^2$, $N = 100, p = 0.1, \gamma = 10, \beta = 10^{-6}$, IP-SP ESN also has$\eta_s = 10^{-7}$, $epochs_s = 4$, $\eta_i = 10^{-5}$ and $epochs_i = 5$

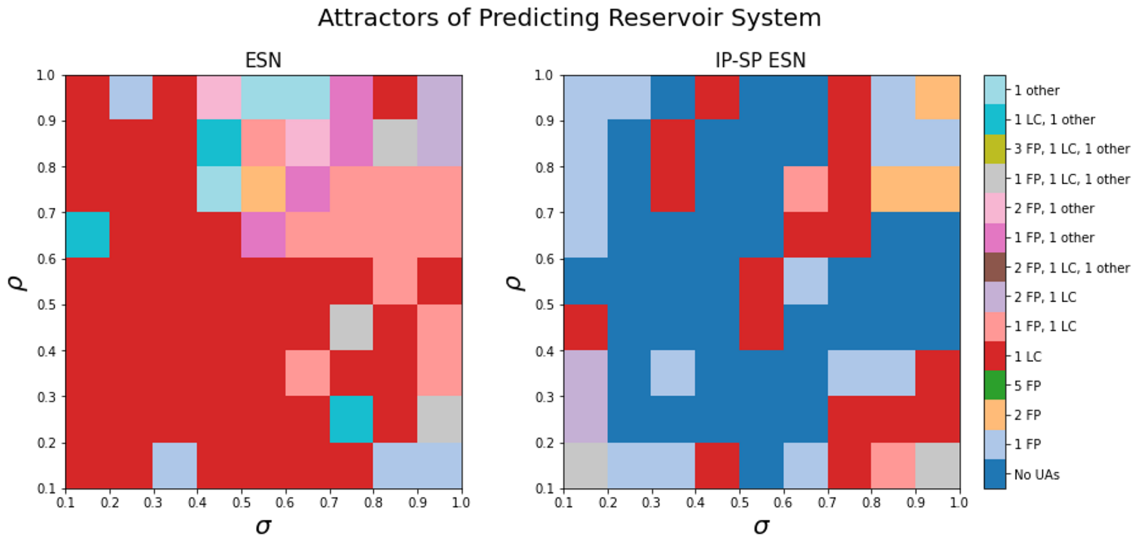

Figure 20: Attractors present in PR phase space for ESNs (left) and IP-SP ESNs (right) with $(\rho, \sigma) \in \{0.1, 0.2, ..., 0.8, 0.9\}^2$, $N = 100, p = 0.1, \gamma = 10, \beta = 10^{-6}$, IP-SP ESN also has$\eta_s = 10^{-7}$, $epochs_s = 4$, $\eta_i = 10^{-5}$ and $epochs_i = 5$. Legend abbreviations as follows; FP: fixed point, LC: limit cycle, UA: untrained attractor.

5.5 Synaptic, Intrinsic Plasticity Rule

For the ESN using the SP followed by the IP rule we chose the parameter values $\eta_s = 10^{-6}$, $epochs_s = 10$, $\eta_i = 10^{-5}$ and $epochs_i = 5$.

Figures 21, 22 and 23 show the prediction log TPE, the RP and the attractors present in the PR phase space, respectively, of SP-IP ESNs with varying values of ρ and σ. In each figure, the left panel displays the plot for the standard ESN for comparison.

Figure 21 shows that the quality of AR is improved or kept the same when the SP rule, followed by the IP rule is applied to the ESNs in the region $[0.2, 0.9] \times [0.1, 0.9]$. The most drastic change is seen for ESNs with $\sigma \geq 0.3$, where all instances of failed AR with ESNs were corrected to have successful AR when the SP-IP ESN was used. However AR by IP-SP ESNs in the region $\rho = 0.1$ is outperformed by the ESNs in this region.

In figure 22 it is shown that after applying the SP rule followed by the IP rule, there are many more (ρ, σ) pairs with no UAs than was the case with the ESN, shown by comparing the much larger yellow area for the SP-IP ESN compared to the two (ρ, σ) pairs which are yellow for the ESN. However, in the region $\sigma = 0.1$ the majority of SP-IP ESNs are outperformed by the ESNs. The exception here is the point $(\rho, \sigma) = (0.1, 0.6)$, where AR failed for the ESN but was successful (albeit with UAs present) for the SP-IP ESN.

In figure 23 we see that for all SP-IP ESNs which still have UAs in the region $[0.2, 0.9] \times [0.1, 0.9]$, the configuration of the UAs is either a fixed point, limit cycle or fixed point and a limit cycle. There are only two SP-IP ESNs with UAs of type 'other', $(\rho, \sigma) = (0.1, 0.2), (0.1, 0.4)$.

Figure 21: Prediction log TPE for ESNs (left) and SP-IP ESNs (right) with $(\rho, \sigma) \in \{0.1, 0.2, ..., 0.8, 0.9\}^2$, $N = 100, p = 0.1, \gamma = 10, \beta = 10^{-6}$, SP-IP ESN also has $\eta_s = 10^{-6}$, $epochs_s = 10$, $\eta_i = 10^{-5}$ and $epochs_i = 5$.

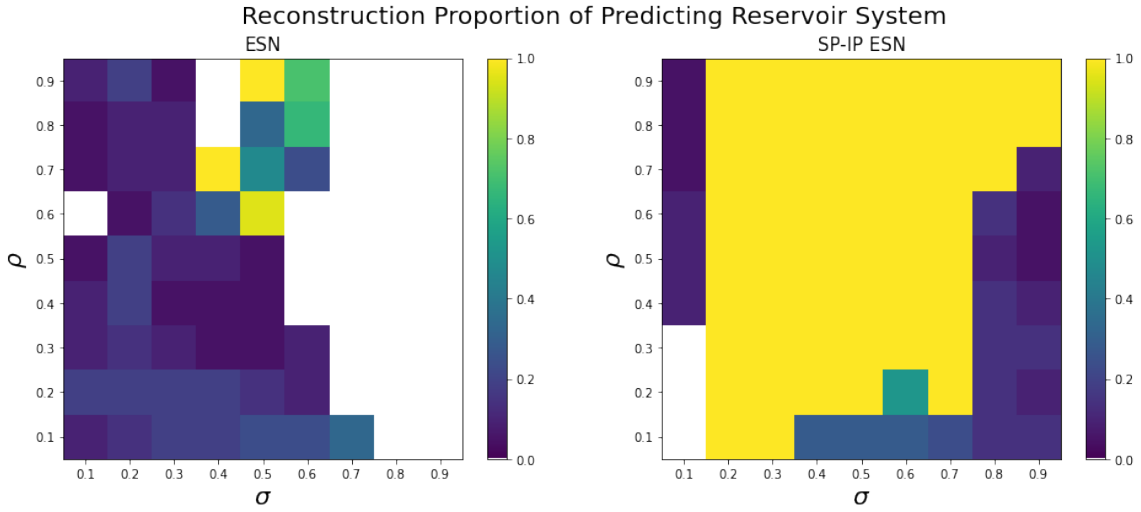

Figure 22: RP for ESNs (left) and SP-IP ESNs (right) with $(\rho, \sigma) \in \{0.1, 0.2, ..., 0.8, 0.9\}^2$, $N = 100, p = 0.1, \gamma = 10, \beta = 10^{-6}$, SP-IP ESN also has$\eta_s = 10^{-6}$, $epochs_s = 10$, $\eta_i = 10^{-5}$ and $epochs_i = 5$

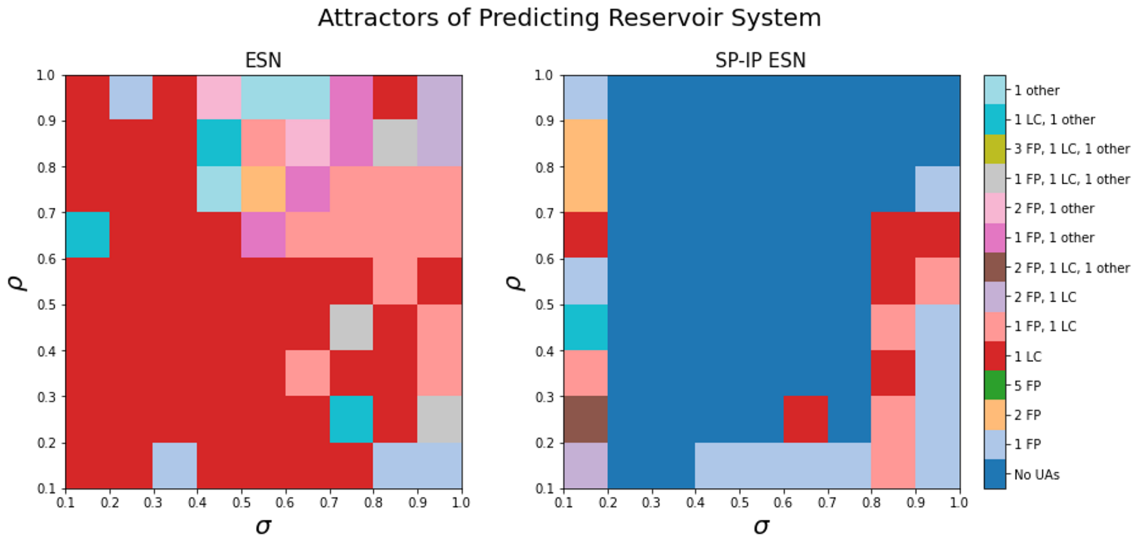

Figure 23: Attractors present in PR phase space for ESNs (left) and SP-IP ESNs (right) with $(\rho, \sigma) \in \{0.1, 0.2, ..., 0.8, 0.9\}^2$, $N = 100, p = 0.1, \gamma = 10, \beta = 10^{-6}$, SP-IP ESN also has $\eta_s = 10^{-6}$, $epochs_s = 10$, $\eta_i = 10^{-5}$ and $epochs_i = 5$. Legend abbreviations as follows; FP: fixed point, LC: limit cycle, UA: untrained attractor.

5.6 Comparing Biological Learning Rules

In order to compare the performance of the different plasticity rules, the ρ, σ plots for the prediction log TPE, the RP and the attractors present in the PR phase space, for the ESNs and all PESNs have been gathered into figures 26, 25 and 24 respectively.

Figure 24 shows that there is little difference between prediction outcomes for the IP ESN, SP-IP ESN and IP-SP ESNs. However all three of these PESNs mostly perform better than the ESN and SP ESN, except in the region $\rho = 0.1$, where the ESN and SP ESN have successful AR and the other three PESNs have log TPE ≥ 1, indicating failed reconstruction.

Upon visual inspection of figure 25, we can see that the SP-IP rule has led to the greatest region of (ρ, σ) values where the RP is 1, indicating that there are no UAs for these PESNs. The exception to this is the region $\rho = 0.1$, where the ESN and SP ESN performs better. In fact, reconstruction completely fails for the SP-IP ESNs with $\rho = 0.1$, $\sigma \in [0.1, 0.3]$ indicating that it is not an attractor of their PR systems, which explains why the prediction log TPE was so high for these SP-IP ESNs in figure 24.

In figure 26 we see that the SP-IP ESN was able to remove UAs of type 'other' (which are usually poor reconstructions of the Lorenz or periodic orbits with more than one local max/ min in a period) for all ESNs with $\sigma > 0.1$, the majority of the SP-IP ESNs in this region have no UAs, with a few having either a fixed point, a limit cycle or a fixed point and a limit cycle as UAs.

One thing which is clear from these plots is that the SP-IP rule clearly outperforms all others at the task of reconstructing the Lorenz attractor without UAs. Another interesting observation is the similarity of all three types of plots between the IP ESNs and the IP-SP ESNs, which shows that for these parameter values, after applying the IP rule, the SP rule has little effect.

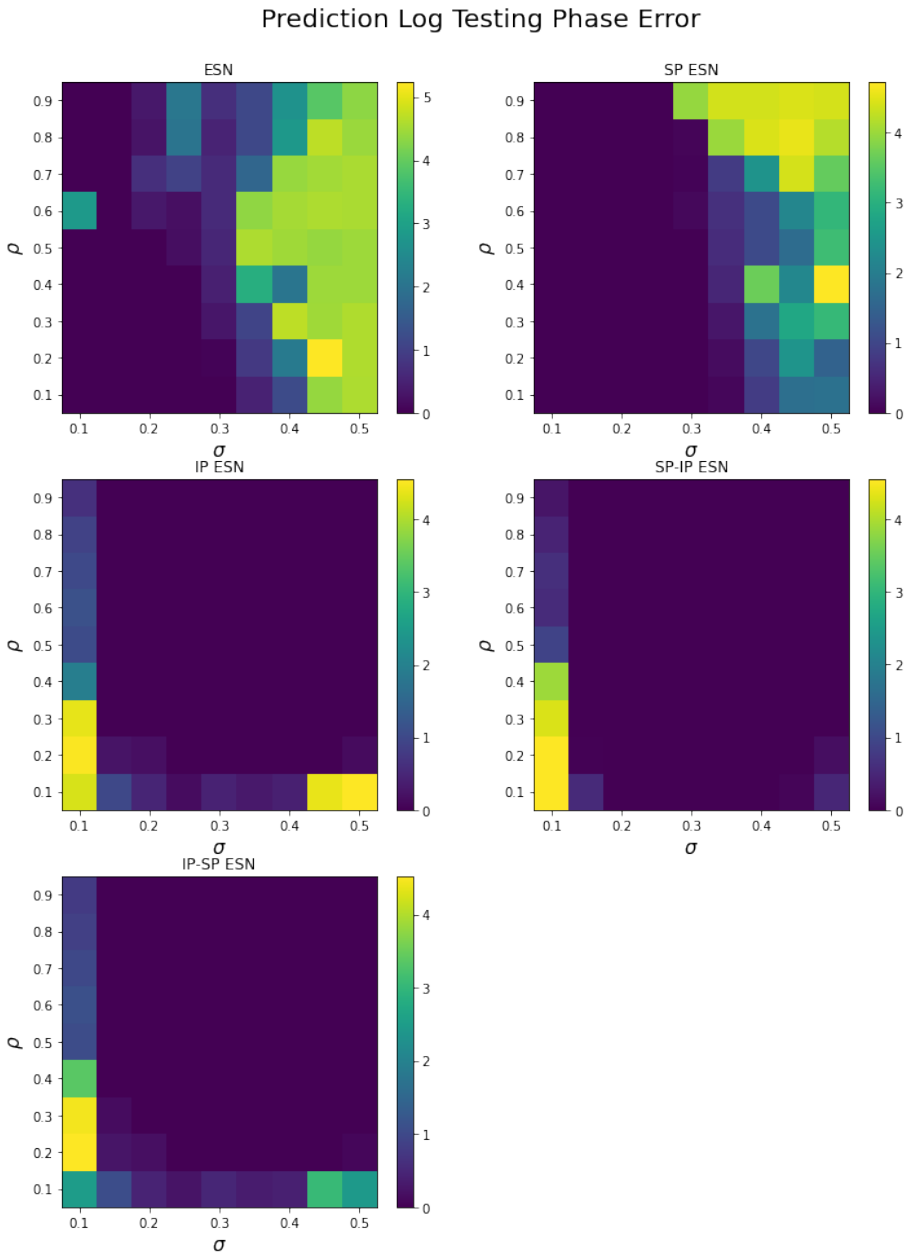

Figure 24: Prediction log TPE for ESNs and all types of PESNs. All ESNs and PESNs have $(\rho, \sigma) \in \{0.1, 0.2, ..., 0.8, 0.9\}^2$, $N = 100, p = 0.1, \gamma = 10, \beta = 10^{-6}$. SP ESNs have $\eta_s = 10^{-6}$, $epochs_s = 10$, IP ESNs have $\eta_i = 10^{-5}$, $epochs_i = 5$, SP-IP ESNs have $\eta_s = 10^{-6}$, $epochs_s = 10$, $\eta_i = 10^{-5}$ and $epochs_i = 5$, IP-SP ESNs have $\eta_s = 10^{-7}$, $epochs_s = 4$, $\eta_i = 10^{-5}$ and $epochs_i = 5$.

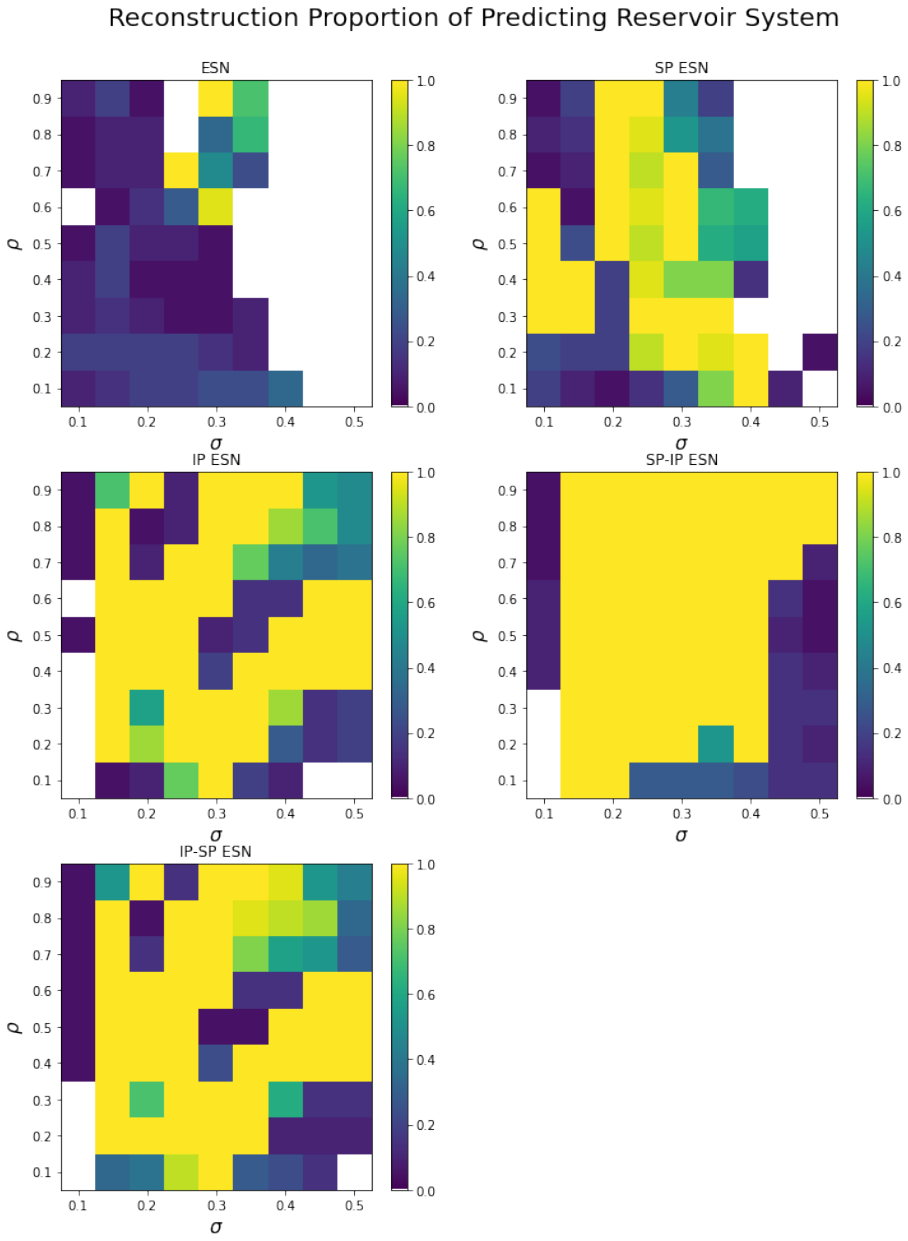

Figure 25: RP for ESNs and all types of PESNs. All ESNs and PESNs have $(\rho, \sigma) \in \{0.1, 0.2, ..., 0.8, 0.9\}^2$, $N = 100, p = 0.1, \gamma = 10, \beta = 10^{-6}$. SP ESNs have $\eta_s = 10^{-6}$, $epochs_s = 10$, IP ESNs have $\eta_i = 10^{-5}$, $epochs_i = 5$, SP-IP ESNs have $\eta_s = 10^{-6}$, $epochs_s = 10$, $\eta_i = 10^{-5}$ and $epochs_i = 5$, IP-SP ESNs have $\eta_s = 10^{-7}$, $epochs_s = 4$, $\eta_i = 10^{-5}$ and $epochs_i = 5$. Legend abbreviations as follows; FP: fixed point, LC: limit cycle, UA: untrained attractor.

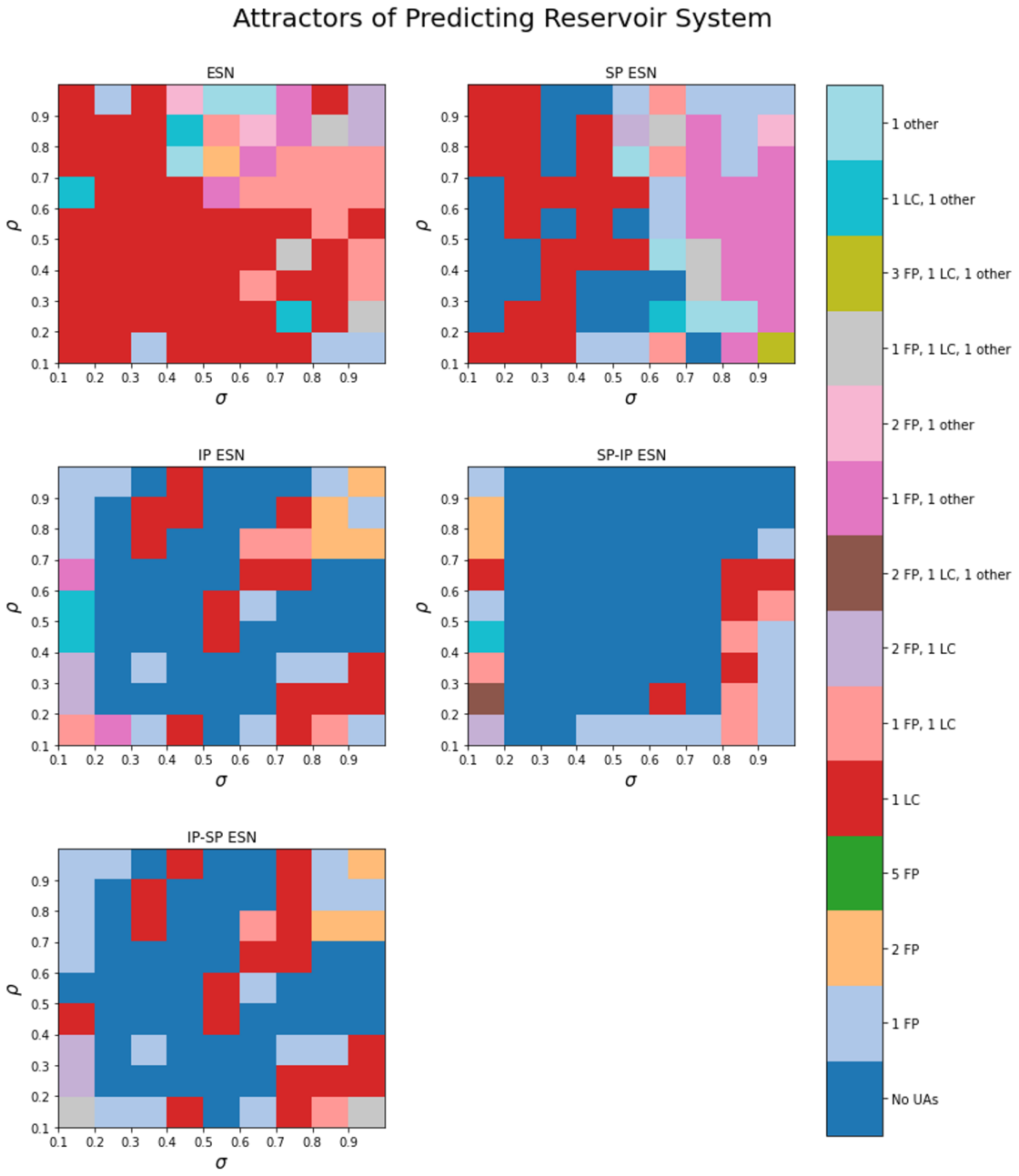

Figure 26: Attractors present in PR phase space for ESNs and all types of PESNs. All ESNs and PESNs have $(\rho, \sigma) \in \{0.1, 0.2, ..., 0.8, 0.9\}^2$, $N = 100, p = 0.1, \gamma = 10, \beta = 10^{-6}$. SP ESNs have $\eta_s = 10^{-6}$, $epochs_s = 10$, IP ESNs have $\eta_i = 10^{-5}$, $epochs_i = 5$, SP-IP ESNs have $\eta_s = 10^{-6}$, $epochs_s = 10$, $\eta_i = 10^{-5}$ and $epochs_i = 5$, IP-SP ESNs have $\eta_s = 10^{-7}$, $epochs_s = 4$, $\eta_i = 10^{-5}$ and $epochs_i = 5$.

6 Discussion and Conclusion

In this report our goal was to further establish the idea of UAs in the context of machine learning techniques for time series prediction, which is a topic often excluded in the literature, thus far only being discussed in the context of a multistable RC by Flynn et al. in 2021 [8]. UAs were shown to be a prevalent feature of ESNs and are often the reason why prediction using ESNs fails.

In an attempt to overcome these UAs present in the PR system of ESNs, we included plasticity rules in the training of the ESNs, motivated by the fact that they were shown to improve short term prediction of chaotic time-series with an ESN [9]. Our results show this is possible as most PESNs were able to remove UAs. In addition, the best performing rule (SP-IP ESN) was also the best performing rule for the task of predicting future values of a time series sampled from a chaotic Mackey-Glass system in the Morales [9] paper. We also presented concrete evidence and a rationale for the mechanism by which the rules might lead to improved performance, i.e. by increasing the richness of the samples available to reconstruct the training signal from the reservoir state. Interestingly, we found that the performance of the IP-SP ESN was almost identical to that of the IP ESN. To date, this particular type of analysis in terms of UAs has not been presented in the literature, to our knowledge, and so cannot be compared to existing works. In future work we aim to explore further neuroscientific analogies which might explain this difference in performance, based on the order of learning rule application, or to replicate in other attractor systems to assess whether the outcome is consistent outside of the Lorenz system.

This investigation is limited by a number of factors. The robustness of our results could be improved by using more than one training signal for training as well as using more ICs when calculating the RP and classifying UAs of the PR system. We also conducted a relatively coarse grid search for optimal PESN training epochs and learning rates, and while we were able to find parameter values which were successful at removing UAs in ESNs, a more thorough parameter search may be required to unlock the true potential of these plasticity rules. One way of achieving this would be to perform a grid search over a larger range of parameters or on a smaller scale of learning rates. Another alternative is to use more systematic approaches to hyperparameter optimization such as the method employed by Maat et al. in 2019, which uses Bayesian optimization in a cross-validation framework [29].

In order to improve our understanding of the limitations mentioned above, further research is needed to explore the validity of these results when the ICs are expanded, and the parameter space of these learning rules are explored in more detail.

However, the strengths of the current work include having successfully replicated the results of Morales et al. [9] which confirms the validity of their approach, and allows us to have confidence in the later analyses in this report. In addition to replicating their results, we have analysed the performance of the learning rules applied in more depth than was done in the original Morales paper, from the perspective of UAs. In doing so, we showed that the presence of UAs can be avoided, while attaining good AR using the plasticity rules we tested. To our knowledge, this is the first such analysis of learning rule performance in this context.

In summary, this report outlines our successful initial replication of the results of the Morales et al. [9] paper, and extends the work by analysing the performance of the learning rules applied, which suggests that application of SP rules in advance of IP rules results in the best AR without the presence of UAs. While a number of limitations were identified, these results indicate that more reliable AR is possible using biologically-inspired plasticity rules, and with further confirmation of the validity of this approach in other types of DSs, this area of research has significant potential to

be applied in diverse fields.

A Python Implementation of an Echo-State Network and Plastic Echo-State Networks

In this appendix we provide the Python source code written by the author which implements ESNs and PESNs. The implementation follows an object oriented style, with the interface loosely based on that of ML libraries like TensorFlow [30].

Listing 1: Python Implementation of Fixed Point Classification

```
import numpy as np
import scipy.sparse as sp
from scipy.interpolate import splprep, splev
import scipy.integrate as integrate
from scipy import stats



class ESN(object):
    """
    Implementation of an echo state network (ESN).

    Instance Variables:
            seed (int): Random seed, used in generating W_in_orig and M_orig.
            N (int): Dimension of hidden layer.
            d (int): Dimension of input layer.
            p (double): Density of hidden layer connections.
            rho (double): Spectral radius of adjacency matrix M.
            gamma (double): Controls time scale of hidden layer dynamics.
            sigma (double): Scales input signal strength.
            W_in_orig (Nxd sparse csr matrix): Matrix describing connections
            from input layer to hidden layer.
            W_in (Nxd sparse csr matrix): W_in_orig, scaled by sigma.
            M_orig (NxN sparse csr matrix): Adjacency matrix for hidden layer
            connections.
            M (NxN sparse csr matrix): M_orig, scaled to have spectral radius rho.
            W_out (dx2N matrix): Output layer connections learned during training.
            beta (double): Regularization parameter used when training W_out.
            r_T (Nd array): Hidden layer state after training.

    """

    def __init__(self, N, p, d, rho, sigma, gamma, beta, seed=1):

        # Create input layer.
        self.N = N
        self.d = d
        self.seed = seed
        self.W_in_orig = self.input_matrix()
        self.sigma = sigma

        # Create reservoir adjacency matrix.
        self.p = p
        self.M_orig = self.adj_matrix()
        self.rho = rho

        # Output layer initially 'None' before training.
```

```
        self.W_out = None

        self.gamma = gamma
        self.beta = beta
        self.u = None
        self.r_T = None
        self.u = None


    @property
    def rho(self):
        return self.__rho


    @rho.setter
    def rho(self, rho):
        """
        Changes adjacency matrix M by rescaling M_orig after resetting rho.

        """
        self.__rho = rho
        self.M = rho * self.M_orig


    @property
    def sigma(self):
        return self.__sigma


    @sigma.setter
    def sigma(self, sigma):
        """
        Changes input matrix W_in by rescaling W_in_orig after resetting sigma.

        """
        self.__sigma = sigma
        self.W_in = sigma * self.W_in_orig


    def f_PR(self, r, t, *args):
        """
        Vector field giving dynamics of predicting reservoir system.

        Parameters
        ----------
        r : Nd array
            Vector of reservoir state variables
        t : double
            Time variable for reservoir state
        *args :
            Unused paramater, included in function params for compatability with
            scipy.integrate.odeint.

        Returns
        -------
        Nd array
            Value of function giving predicting reservoir dynamics for given r
            and t values.
```

```
    """
    v = self.W_out.dot(self.q(r))
    return self.gamma*(-r + np.tanh(self.M.dot(r) + self.W_in.dot(v)))


def f_LR(self, r, t, *args):
    """
    Vector field giving dynamics of listening reservoir system.

    Parameters
    ----------
    r : Nd array
        Vector of reservoir state variables
    t : double
        Time variable for reservoir state
    *args :
        Unused paramater, included in function params for compatability with
        scipy.integrate.odeint.

    Returns
    -------
    Nd array
        Value of function giving listening reservoir dynamics for given r
        and t values.

    """
    return self.gamma*(-r + np.tanh(self.M.dot(r) + self.W_in.dot(self.u(t))))


def adj_matrix(self):
    """
    Generates a random Erdos-Renyi NxN sparse csr matrix scaled by its max
    eigenvalue.

    Returns
    -------
    NxN sparse csr matrix
        Adjacency matrix M scaled to have unit spectral radius.

    """

    np.random.seed(seed = self.seed)
    rvs = stats.uniform(loc=-1, scale=2).rvs
    M = sp.random(self.N, self.N, self.p, format='csr', data_rvs=rvs)
    max_eval = np.abs(sp.linalg.eigs(M, 1, which='LM',
    return_eigenvectors=False)[0])

    return (1/max_eval)*M


def input_matrix(self):
    """
    Generates a random Nxd sparse csr matrix with 1 entry in each row
    sampled from UNIF(-1,1).

    Returns
    -------
```

```
    Nxd sparse csr matrix
        Random matrix giving connections from input layer to reservoir.

    """


    np.random.seed(seed = self.seed)
    # Create sparse matrix in COO form, then cast to CSR form.
    rows = np.arange(0, self.N)
    cols = stats.randint(0, self.d).rvs(self.N)
    values = stats.uniform(loc=-1, scale=2).rvs(self.N)
    W_in = sp.coo_matrix((values, (rows, cols))).tocsr()

    return W_in


def q(self, r):
    """
    Function q() used as part of output function used for predictions.

    Parameters
    ----------
    r : Nd array
        Vector of reservoir state variables.

    Returns
    -------
    x : 2xNd array
        Vector of reservoir state variables, concatenated with squares of
        reservoir state variables.
    """
    x = np.zeros(2*self.N)
    x[0:self.N] = r
    x[self.N: 2*self.N] = r**2

    return x


def spline(self, data, t):
    coords = [data[:,i] for i in range(self.d)]
    tck, u = splprep(coords, u = t, s=0)
    return lambda t: np.asarray(splev(t, tck))


def train(self, data, t, t_listen):
    LR_traj : TYPE
        DESCRIPTION.

    """
    # Integrate Listening reservoir system
    if type(self.u) == type(None):
        self.u = self.spline(data, t)
    LR_traj = integrate.odeint(self.f_LR, np.zeros(self.N), t)

    X = np.zeros((2*self.N, t.size - t_listen))
    Y = np.transpose(data[t_listen:])
    for i in range(t_listen, t.size - 1):
          X[:,i+1 - t_listen] = self.q(LR_traj[i+1])
```

```
        # Calculate output matrix.
        X_T = np.transpose(X)
        M_1 = np.matmul(Y, X_T)
        M_2 = np.linalg.inv(np.matmul(X, X_T) + self.beta*np.identity(2*self.N))
        self.W_out = np.matmul(M_1, M_2)
        self.r_T = LR_traj[-1]

        return LR_traj


    def predict(self, t_predict, data=None, t=None):
        if type(data) == type(None):
            PR_traj = integrate.odeint(self.f_PR, self.r_T, t_predict)
        else:
            self.u = self.spline(data, t)
            LR_traj = integrate.odeint(self.f_LR, np.zeros(self.N), t)
            PR_traj = integrate.odeint(self.f_PR, LR_traj[-1], t_predict)

        prediction = np.asarray([self.W_out.dot(self.q(p)) for p in PR_traj])
        return prediction


class SPESN(ESN):
    """
    Implementation of an echo state network (ESN) augmented with an anti-Hebbian
    synaptic plasticity rule. Subclass of ESN.

    Instance Variables:
            eta_s (double): Learning rate for synaptic plasticity rule.
            epochs_s (int): Number of training epochs for plasticity rule.
    """

    def __init__(self, N, p, d, rho, sigma, gamma, beta, eta_s, epochs_s, seed=1):
        ESN.__init__(self, N, p, d, rho, sigma, gamma, beta, seed)
        self.eta_s = eta_s
        self.epochs_s = epochs_s


    def f_PR(self, r, t, *args):
        v = self.W_out.dot(self.q(r))
        f = self.gamma*(-r + np.tanh(self.M.dot(r) + self.W_in.dot(v)))
        return np.squeeze(np.asarray(f))


    def f_LR(self, r, t, *args):
        f = self.gamma*(-r + np.tanh(self.M.dot(r) + self.W_in.dot(self.u(t))))
        return np.squeeze(np.asarray(f))


    def SP_train(self, data, t_points, t_listen, reset_M=True, scale_M=0):
        if reset_M:
            self.rho = self.rho
        self.u = self.spline(data, t_points)

        for e in range(self.epochs_s):
            x = integrate.odeint(self.f_LR, np.zeros(self.N), t_points)
            for t in range(t_listen, data.shape[0]-1):
```

```
                self.M = self.M - self.eta_s*(np.asarray([x[t+1]]).T)@np.asarray([x[t]])

            # If scale_M==1, scale M after each epoch to have spectral radius rho.
            if scale_M==1:
                self.M = sp.csr_matrix(self.M)
                max_eval = np.abs(sp.linalg.eigs(self.M, 1, which='LM',
                return_eigenvectors=False)[0])
                self.M = (self.rho/max_eval)*self.M


        # If scale_M==2, scale M once after training to have spectral radius rho.
        if scale_M==2:
                self.M = sp.csr_matrix(self.M)
                max_eval = np.abs(sp.linalg.eigs(self.M, 1, which='LM',
                return_eigenvectors=False)[0])
                self.M = (self.rho/max_eval)*self.M

        # If scale_M==0, don't rescale M during training.
        else:
                self.M = sp.csr_matrix(self.M)

        return None



class IPESN(SPESN):
    def __init__(self, N, p, d, rho, sigma, gamma, beta, eta_i, epochs_i, eta_s=0,
    epochs_s=0, mu=0, sd=0.5, seed=1):
        SPESN.__init__(self, N, p, d, rho, sigma, gamma, beta, eta_s, epochs_s, seed)
        self.eta_i = eta_i
        self.epochs_i = epochs_i
        self.mu=mu
        self.sd=sd
        self.a = np.ones(N)
        self.b = np.zeros(N)


    def f_PR(self, r, t, *args):
        v = self.W_out.dot(self.q(r))
        f = self.gamma*(-r + np.tanh(self.M.dot(r) + self.W_in.dot(v) + self.b))
        return np.squeeze(np.asarray(f))


    def f_LR(self, r, t, *args):
        f = self.gamma*(-r + np.tanh(self.M.dot(r) + self.W_in.dot(self.u(t)) + self.b))
        return np.squeeze(np.asarray(f))


    def H(self, x):
        return -self.mu/self.sd**2 + (x/self.sd**2)*(2*self.sd**2 + 1 - x**2 + self.mu*x)


    def IP_train(self, data, t_points, t_listen, reset_M=True):
        # If reset_M==True, change M, W_in, a and b to original values.
        if reset_M:
            self.rho = self.rho
            self.sigma = self.sigma
            self.a = np.ones(self.N)
```

```
            self.b = np.zeros(self.N)
        self.u = self.spline(data, t_points)


        for e in range(self.epochs_i):
            x = integrate.odeint(self.f_LR, np.zeros(self.N), t_points)
            z = np.asarray([self.M.dot(x[t]) + self.W_in.dot(self.u(t_points[t]))
            for t in range(len(t_points))])
            for t in range(t_listen, data.shape[0]-1):
                delta_b = -self.eta_i*self.H(x[t])
                delta_a = self.eta_i/self.a + np.dot(np.diag(delta_b), z[t])
                self.b += delta_b
                self.a += delta_a

        self.M = self.rho*np.matmul(np.diag(self.a), self.M.todense())
        self.W_in = self.sigma*np.matmul(np.diag(self.a), self.W_in.todense())
```

B Classifying Attractors

Given a set of numerically integrated trajectories $\{\mathbf{x}(t)_i\}_{i=1}^{k}$ from a DS, we classify the attractors that they are converging to numerically by using a filter which determines in order which trajectories are converging to fixed points, limit cycles, the Lorenz attractor or some other type of attractor. In this section we outline the methods used for classifying a single trajectory as one of the above attractor types, then explain how these methods are combined to classify the the attractors present in a set trajectories.

B.1 Fixed Points

We classify a trajectory $\{\mathbf{x}(t_i)\}_{i=0}^{T}$ as converging to a fixed point if each point pair of points sufficiently far along the trajectory are within some small distance $\epsilon_{fp} > 0$ of each other, that is $|\mathbf{x}(t_i) - \mathbf{x}(t_j)| < \epsilon_{fp}$ for all $i, j > N_{fp}$. Where $\epsilon_{fp} > 0$ and N_{fp} $in\{1, 2, 3, ..., T-1, T\}$ are parameters which need to be chosen carefully. Below is the Python code used to classify a trajectory as converging to a fixed point.

Listing 2: Python Implementation of Fixed Point Classification

```
def is_fixed_point(x, tol, n):
    """
    x : (ndarray).
    n: (int) number of points to consider at end of x.
    tol: (float64) difference between any pair of last
        n points of x must be less than tol if fixed point.
    """
    dists = distance.cdist(x[-n:], x[-n:], 'euclidean')
    return np.max(dists) < tol
```

B.2 Limit Cycles

To classify a trajectory as a limit cycle, we first take the last N_{lc} points of the trajectory and calculate the local maxima and minima of each coordinate in the trajectory. Then we check that

each of these sequences of maxima/minima are converging to some limit in the same manner as we did for fixed points using a tolerance $\epsilon_{lc} > 0$. It should be noted that this method can only detect if a trajectory is converging to a pure sinusoidal limit cycle and would fail if the limit cycle were, for example, a superposition of multiple sinusoidal curves with multiple different local maxima/minima reached in a single period. Below is the Python code used to classify a trajectory as converging to a limit cycle.

Listing 3: Python Implementation of Limit Cycle Classification

```
def is_limit_cycle(x, tol=1, n=3000):
    """
    x : (ndarray).
    n: (int) number of points to consider at end of x.
    tol: (float64) difference between any pair of last
        n points of x must be less than tol if fixed point.
    """
    maxima = []
    minima = []

    for i in range(x.shape[1]):
        z = x[-n:,i]

        # for local maxima
        max_indices = argrelextrema(z, np.greater)
        max_vals = z[max_indices]

        # for local minima
        min_indices = argrelextrema(z, np.less)
        min_vals = z[min_indices]

        maxima.append(max_vals)
        minima.append(min_vals)

    # First periodicity check, see if first differences are 0 in any coord.
    for i in range(x.shape[1]):
        #print(np.abs(np.max(np.absolute(np.diff(maxima[i], n=1)))))
        if np.abs(np.max(np.absolute(np.diff(maxima[i], n=1)))) < tol:
            return True, [maxima, max_indices, minima, min_indices]

    return False, [maxima, max_indices, minima, min_indices]
```

B.3 Lorenz Attractor

A trajectory is classified as converging to the embedded Lorenz attractor by calculating its TPE error as outlined in section 3.8.4 and seeing if it is below some cut-off value ϵ_{TPE}. In this report, we consider a TPE less than 5 an indicator of successful AR. This cutoff point was decided by observing the output of the RC training method over a range of ρ, σ values and comparing it to the calculated TPE. It was observed that the vast majority of successful AR was associated with a TPE of 5 or less. Below is the Python code used to classify a trajectory as converging to the embedded Lorenz attractor.

Listing 4: Python Implementation of Lorenz Attractor Classification

```
def tpe(u_hat, dt, f, args):
```

```
    # Calculate approx. and ideal movement vectors.
    delta_hat = np.diff(u_hat, axis=0)
    delta = np.asarray([RK4_step(u_hat[i], dt, f, args)
    for i in range(u_hat.shape[0] - 1)])

    # Calculate norms of ideal movement vectors approx. movement vector error.
    delta_norm = np.linalg.norm(delta, axis=1)
    error_norm = np.linalg.norm(delta_hat - delta, axis=1)

    # Return TPE.
    return 1/(dt*u_hat.shape[0])*np.sum(np.multiply(error_norm, 1/delta_norm))
```

B.4 Other

A trajectory Which cannot be classified using one of the above methods is classified as an attractor of type 'other'. In our numerical results, examples of attractors classified as type 'other' were poor reconstructions of the Lorenz attractor, periodic orbits with more than one local max/min value in a single period, tori and chaotic attractors different from the Lorenz. It should be noted however that the majority of attractors encountered in our simulations can be classified using the above methods and that most attractors classified as 'other' are poor reconstructions of the Lorenz.

B.5 Classifying a Set of Trajectories

Given a set of trajectories $\{\mathbf{x}(t)_i\}_{i=1}^{k}$ of some DS, we classify the number and type of attractors that they are converging to by using the methods described above as follows,

1. For each trajectory in $\mathbf{x}_i(t)$, we check if its a fixed point. If it isn't, we move on to the next trajectory. If $\mathbf{x}_i(t)$ is the first fixed point found, then we store the last point of the trajectory $\mathbf{x}_i(t_T)$ as our approximation of the fixed point. If $\mathbf{x}_i(t)$ is not the first fixed point found, then we check that $\mathbf{x}_i(t_T)$ is not approximately equal to previously detected fixed points (up to an error tolerance eps_{fp}), if it is we move on the the next trajectory, otherwise we store the approximate value of the fixed point $\mathbf{x}_i(t_T)$.

2. For each trajectory not classified as a fixed point, we check if its a limit cycle. If it is we store the trajectory, otherwise we move on to the next trajectory.

3. For each trajectory not classified as a fixed point or a limit cycle, we classify it as converging to the Lorenz if it has a TPE error less than 5, otherwise move on to the next trajectory.

4. All remaining trajectories are classified as converging to an attractor of type 'other'.

At the end of this process we have a list of fixed points and limit cycles which some of the trajectories converge to as well as the indices of the trajectories that converge to each type of attractor. Below is the Python code used to classify the type of attractors a set of trajectories is converging to.

Listing 5: Python Implementation of Limit Cycle Classification

```
def get_fixed_points(trajectories, fp_tol, fp_n):
    fps = []
```

```
    fp_indices = []
    for i in range(trajectories.shape[0]):
        x = trajectories[i]
        if is_fixed_point(x, fp_tol, fp_n):
            fp_indices.append(i)
            if len(fps) == 0:
                fps.append(x[-1])
            elif np.min(distance.cdist(np.asarray([x[-1]]), fps, 'euclidean')) > fp_tol:
                fps.append(x[-1])
    return (fps, fp_indices)


def get_limit_cycles(trajectories, indices, lc_tol, lc_n):
    lcs = []
    lc_indices = []
    for i in indices:
        x = trajectories[i]
        if is_limit_cycle(x, lc_tol, lc_n)[0]:
            lcs.append(x)
            lc_indices.append(i)
    return (lcs, lc_indices)


def get_Lorenz(trajectories, indices, dt, f, args, tpe_tol, tpe_n):
    lorenz_indices = []
    for i in indices:
        x = trajectories[i]
        if tpe(x[-tpe_n:], dt, lorenz, args) < tpe_tol:
            lorenz_indices.append(i)
    return lorenz_indices


def classify_att(trajectories, dt, f, args, fp_tol=1e-6, lc_tol=1, tpe_tol=5
, fp_n=1000, lc_n=3000, L_n=1000):
    attractors = dict(keys=['fixed_points', 'limit_cycles', 'Lorenz', 'other'])
    n = trajectories.shape[0]
    indices = list(range(n))

    attractors['fixed_points'] = get_fixed_points(trajectories, fp_tol, fp_n)
    [indices.remove(i) for i in range(n) if i in attractors['fixed_points'][1]]

    attractors['limit_cycles'] = get_limit_cycles(trajectories, indices, lc_tol, lc_n)
    [indices.remove(i) for i in range(n) if i in attractors['limit_cycles'][1]]

    attractors['lorenz'] = get_Lorenz(trajectories, indices, dt, f, args, tpe_tol, L_n)
    [indices.remove(i) for i in range(n) if i in attractors['lorenz']]

    attractors['other'] = indices

    return attractors
```

C Reconstruction Proportion

Suppose we have an RC which has been successfully trained using an input signal from the Lorenz system, so that the Lorenz Attractor has been embedded as an attractor into the phase space of

the PR system whose dynamics are given by

$$\dot{\tilde{\mathbf{r}}} = \gamma \left[-\tilde{\mathbf{r}} + \tanh\left(\rho \mathbf{M} \tilde{\mathbf{r}} + \sigma \mathbf{W}_{in} \hat{\Psi}(\tilde{\mathbf{r}}(t)) \right) \right]. \tag{33}$$

In order to estimate the proportion of ICs in the PR phase space which are contained in the basin of attraction of the embedded Lorenz attractor, we generate $K \in \mathbb{N}$ random ICs in the PR phase space $\{\widetilde{\mathbf{r}(0)}_i\}_{i=1}^K$ and calculate their corresponding trajectories $\{\tilde{\mathbf{r}}(t_i)\}_{i=1}^K$ in the PR system using equation (33). We then project each of these trajectories onto $\mathbb{R}^3$ using the learned output function $\hat{\Psi}$ and calculate their TPEs. If a trajectory has a sufficiently low TPE, then it exhibits dynamics like those of the Lorenz system and is likely converging to the embedded Lorenz attractor. If a trajectory has a sufficiently large TPE, then it exhibits behaviour different from a typical trajectory on the Lorenz and we conclude it must be converging to some other attractor in the PR phase space. The RP of these ICs is the number of trajectories which replicate the dynamics of the Lorenz divided by the total number of ICs used, we use it as a measure of the proportion of ICs in the PR phase space which are in the basin of attraction of the embedded Lorenz. Below is the Python code used to calculate the RP of a set of trajectories.

Listing 6: Python Implementation of Lorenz Attractor Classification

```
def rp(u_hat_array, dt, f, args, tol=10):
    n = u_hat_array.shape[0]
    tpe_array = [tpe(u_hat_array[i], dt, f, args) for i in range(n)]
    return (sum([1 for i in tpe_array if i <= tol])/n, tpe_array)
```

D Generating ρ, σ Plots

Three plots for log TPE, RP and attractor classification are used to analyse the effect of the plasticity rules on UAs present in the PR state space. These plots are created by using a common input signal from the Lorenz system to train a collection of RCs which differ only in their value for the hyperparameters ρ, σ, then exploring the state space of each PR and the quality of the prediction. The procedure used to generate the data for these plots is outlined below;

1. Pick values for the hyperparameters N, p, γ and β, then generate the adjacency matrix $\mathbf{M}$ and input matrix $\mathbf{W}_{in}$. These will be kept fixed for all ESNs used.
2. Generate an evenly spaced grid of ρ, σ values.
3. Use each pair of ρ, σ values together with the matrices and hyperparameters from step 1 to create an ESN, then train each ESN using a common input signal from the Lorenz system.
4. For each trained ESN, calculate the prediction log TPE.
5. Generate a number of random ICs for the PR system, then use these ICs to classify the attractors of the PR and calculate the RP using the methods described in appendices A and B respectively.

This data is then used to create a colour map for each of the quantities log TPE, RP and attractor classification over the grid of ρ, σ used to generate the data. This method of creating plots was implemented in Python, it is written in an object oriented style.

Listing 7: Python Implementation of Lorenz Attractor Classification

```
class RhoSigmaGrid(object):
    def __init__(self, rho_list, sigma_list, RC):
        self.rho_list = rho_list
        self.sigma_list = sigma_list
        self.RC = RC
        rho, sigma = np.meshgrid(rho_list, sigma_list)
        self.param_space = np.array([rho.flatten(),sigma.flatten()]).T
        self.rho_sigma_RCs = dict(keys=self.param_space)


    def train_RCs(self, train_data, t_points, t_listen):
        def train_RCs(self, train_data, t_points, t_listen, rule=None):
         for rho, sigma in self.param_space:
            rho_sigma_RC = copy.deepcopy(self.RC)
            rho_sigma_RC.rho = rho
            rho_sigma_RC.sigma = sigma
            if rule == 'sp':
                rho_sigma_RC.SP_train(train_data, t_points, t_listen, reset_M=False)
            elif rule == 'ip':
                rho_sigma_RC.IP_train(train_data, t_points, t_listen, reset_M=False)
            elif rule == 'sp_ip':
                rho_sigma_RC.SP_train(train_data, t_points, t_listen, reset_M=False)
                rho_sigma_RC.IP_train(train_data, t_points, t_listen, reset_M=False)
            elif rule == 'ip_sp':
                rho_sigma_RC.IP_train(train_data, t_points, t_listen, reset_M=False)
                rho_sigma_RC.SP_train(train_data, t_points, t_listen, reset_M=False)
            rho_sigma_RC.train(train_data, t_points, t_listen)
            self.rho_sigma_RCs[(rho, sigma)] = rho_sigma_RC


    def calc_trajectories(self, t_points, ics):
        self.trajectories = np.zeros((self.rho_list.size, self.sigma_list.size,
        len(ics) + 1, t_points.size, self.RC.d))
        for i in range(self.rho_list.size):
            for j in range(self.sigma_list.size):
                rho, sigma = self.rho_list[i], self.sigma_list[j]
                print(rho, sigma)
                rho_sigma_RC = self.rho_sigma_RCs[(rho, sigma)]
                self.trajectories[i, j,:,:,:] = evolve(rho_sigma_RC, ics, t_points)


    def rp(u_hat_array, dt, f, args, tol):
        n = u_hat_array.shape[0]
        tpe_array = [tpe(u_hat_array[i], dt, f, args) for i in range(n)]
        return (sum([1 for i in tpe_array if i <= tol])/n, tpe_array)


    def attractor_dict(self, dt, f, args, fp_tol=1e-6, lc_tol=1, tpe_tol=5,
    fp_n=1000, lc_n=3000, L_n=1000):
        self.attractors = dict(keys=self.param_space)
        for i in range(self.rho_list.size):
            for j in range(self.sigma_list.size):
                rho = self.rho_list[i]
                sigma = self.sigma_list[j]
                x = self.trajectories[i, j]
                x_att = classify_att(x, dt, f, args, fp_tol, lc_tol, tpe_tol,
                fp_n, lc_n, L_n)
```

```
            self.attractors[(rho, sigma)] = x_att


    def rp_matrix(self, dt, f, args, tol):
        rp_matrix = np.zeros((self.rho_list.size, self.sigma_list.size))
        for i in range(self.rho_list.size):
                for j in range(self.sigma_list.size):
                    rp_matrix[i, j] = rp(self.trajectories[i, j], dt, f, args, tol)[0]
        return rp_matrix


    def pred_tpe_mat(self, dt, f, args):
        TPE_matrix = np.zeros((self.rho_list.size, self.sigma_list.size))
        for i in range(self.rho_list.size):
            for j in range(self.sigma_list.size):
                x = self.trajectories[i, j, -1]
                TPE_matrix[i, j] = tpe(x, dt, f, args)
        return TPE_matrix


    def pred_attractor_mat(self, dt, f, args, fp_tol=1e-6, lc_tol=1, tpe_tol=5,
    fp_n=1000, lc_n=3000, L_n=1000):
        attractor_mat = np.zeros((self.rho_list.size, self.sigma_list.size))
        for i in range(self.rho_list.size):
            for j in range(self.sigma_list.size):
                x = self.trajectories[i, j, -1]
                if is_fixed_point(x, fp_tol, fp_n):
                    attractor_mat[i, j] = 0
                elif is_limit_cycle(x, lc_tol, lc_n)[0]:
                    attractor_mat[i, j] = 1
                elif tpe(x[L_n:], dt, f, args) < tpe_tol:
                    attractor_mat[i, j] = 2
                else:
                    attractor_mat[i, j] = 3
        return attractor_mat


    def ua_matrix(self, attractor_encoding):
        ua_mat = np.zeros((self.rho_list.size, self.sigma_list.size))
        for i in range(self.rho_list.size):
            for j in range(self.sigma_list.size):
                rho = self.rho_list[i]
                sigma = self.sigma_list[j]
                ua_mat[i, j] = encode_att(self.attractors[(rho, sigma)],
                attractor_encoding)
        return ua_mat
```

E Finding η, Epoch Values Which Remove Untrained Attractors

First we chose values of ρ and σ which gave rise to an ESN whose PR system had UAs. Specifically we instantiated an ESN with hyperparameter values $N = 100, p = 0.01, \rho = 0.3, \sigma = 0.6, \gamma = 10.0$ and $\beta = 10^{-6}$, which had a limit cycle and a fixed point as UAs, then we trained ESNs using the IP and SP rules separately over a range of η and values to find a pair of values such that the

resulting PR system had no UAs. For both rules we considered epochs in the range $1, ..., 10$ and $\eta \in \{10^{-9}, 10^{-8}, ..., 10^{-}1, 1\}$.

Figure 27 shows examples of some of the data generated through this process for the SP rule. In it we see the resulting PR state space after carrying out the SP rule for various values of η_s and $epochs_s$. To make these plots, we generated a collection of SP ESNs with $\eta_s \in \{0.1, 0.2, 0.3\}$ and $epochs_s \in \{1, 2, 3\}$, all SP ESNs had $N = 100, p = 0.01, \rho = 0.3, \sigma = 0.6, \gamma = 10$, $\beta = 10^{-6}$. After training the SP ESNs using the same trajectory from the Lorenz system used in section 5, we chose 20 random ICs in the PR phase space and evolved their trajectories using equation 15 for each SP ESN, for 100 time units. We then took the last 40 time units of each trajectory and projected it onto $\mathbb{R}^3$. This gives a visual representation of which trajectories in the reservoir phase space are converging to the embedded Lorenz and which are converging to an UA. Figure 27 rows a, b and c show the result of using learning rates $\eta = 0.1, 0.21, 0.3$ respectively. Out of these examples, using the SP rule for one epoch and with $\eta = 0.21$ was the most successful at removing UAs.

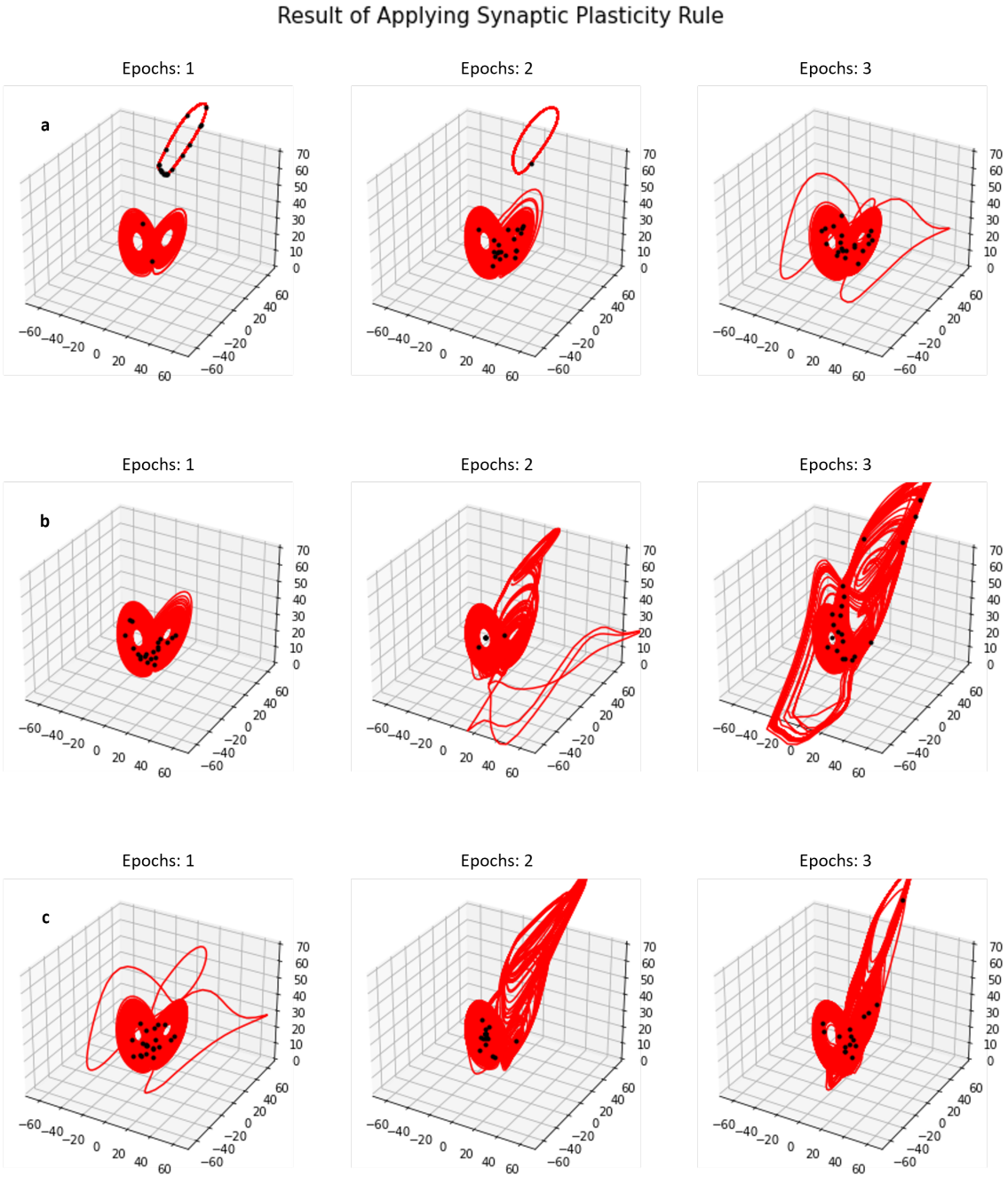

Figure 27: Evolution of covariance matrix for LR state over 3 epochs for SP ESN with $N = 100, p = 0.1, \rho = 0.3, \sigma = 0.6, \gamma = 10, \beta = 10^{-6}$ and $\eta_s = 10^{-6}$.

References

[1] Steven H Strogatz. *Nonlinear dynamics and chaos with student solutions manual: With applications to physics, biology, chemistry, and engineering*. CRC press, 2018.

[2] Michael M Mansfield and Colm O'sullivan. *Understanding physics*. John Wiley & Sons, 2020.

[3] Steven L Brunton and J Nathan Kutz. *Data-driven science and engineering: Machine learning, dynamical systems, and control*. Cambridge University Press, 2019.

[4] Herbert Jaeger. The "echo state" approach to analysing and training recurrent neural networks-with an erratum note. *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, 148(34):13, 2001.

[5] Wolfgang Maass, Thomas Natschläger, and Henry Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural computation*, 14(11):2531–2560, 2002.

[6] Zhixin Lu, Brian R Hunt, and Edward Ott. Attractor reconstruction by machine learning. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 28(6):061104, 2018.

[7] Zhixin Lu and Danielle S Bassett. Invertible generalized synchronization: A putative mechanism for implicit learning in neural systems. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 30(6):063133, 2020.

[8] Andrew Flynn, Vassilios A Tsachouridis, and Andreas Amann. Multifunctionality in a reservoir computer. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 31(1):013125, 2021.

[9] Guillermo B Morales, Claudio R Mirasso, and Miguel C Soriano. Unveiling the role of plasticity rules in reservoir computing. *Neurocomputing*, 2021.

[10] Richard L Burden, J Douglas Faires, and Annette M Burden. *Numerical analysis*. Cengage learning, 2015.

[11] Scipy odeint. https://docs.scipy.org/doc/scipy-1.2.1/reference/generated/scipy.integrate.odeint.html, 2022.

[12] Edward N Lorenz. Deterministic nonperiodic flow. *Journal of atmospheric sciences*, 20(2):130–141, 1963.

[13] Matthew Dale, Simon O'Keefe, Angelika Sebald, Susan Stepney, and Martin A Trefzer. Reservoir computing quality: connectivity and topology. *Natural Computing*, 20(2):205–216, 2021.

[14] Vladimir Ceperic and Adrijan Baric. Reducing complexity of echo state networks with sparse linear regression algorithms. In *2014 UKSim-AMSS 16th International Conference on Computer Modelling and Simulation*, pages 26–31. IEEE, 2014.

[15] Mantas Lukoševičius. A practical guide to applying echo state networks. In *Neural networks: Tricks of the trade*, pages 659–686. Springer, 2012.

[16] François Duport, Anteo Smerieri, Akram Akrout, Marc Haelterman, and Serge Massar. Fully analogue photonic reservoir computer. *Scientific reports*, 6(1):1–12, 2016.

[17] Kohei Nakajima, Helmut Hauser, Rongjie Kang, Emanuele Guglielmino, Darwin G Caldwell, and Rolf Pfeifer. A soft body as a reservoir: case studies in a dynamic model of octopus-inspired soft robotic arm. *Frontiers in computational neuroscience*, 7:91, 2013.

[18] Chrisantha Fernando and Sampsa Sojakka. Pattern recognition in a bucket. In *European conference on artificial life*, pages 588–597. Springer, 2003.

[19] Andrew Flynn, Joschka Herteux, Vassilios A Tsachouridis, Christoph Räth, and Andreas Amann. Symmetry kills the square in a multifunctional reservoir computer. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 31(7):073122, 2021.

[20] Ashesh Chattopadhyay, Pedram Hassanzadeh, and Devika Subramanian. Data-driven predictions of a multiscale lorenz 96 chaotic system using machine-learning methods: reservoir computing, artificial neural network, and long short-term memory network. *Nonlinear Processes in Geophysics*, 27(3):373–389, 2020.

[21] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[22] Andrea Ceni, Peter Ashwin, and Lorenzo Livi. Interpreting recurrent neural networks behaviour via excitable network attractors. *Cognitive Computation*, 12(2):330–356, 2020.

[23] Jaideep Pathak, Zhixin Lu, Brian R Hunt, Michelle Girvan, and Edward Ott. Using machine learning to replicate chaotic attractors and calculate lyapunov exponents from data. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 27(12):121102, 2017.

[24] Thomas L Carroll. Do reservoir computers work best at the edge of chaos? *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 30(12):121109, 2020.

[25] Thomas Trappenberg. *Fundamentals of computational neuroscience*. OUP Oxford, 2009.

[26] Peter Dayan, Laurence F Abbott, et al. Theoretical neuroscience: computational and mathematical modeling of neural systems. *Journal of Cognitive Neuroscience*, 15(1):154–155, 2003.

[27] Hyun Geun Shim, Yong-Seok Lee, and Sang Jeong Kim. The emerging concept of intrinsic plasticity: activity-dependent modulation of intrinsic excitability in cerebellar purkinje cells and motor learning. *Experimental neurobiology*, 27(3):139, 2018.

[28] Marcus E Raichle. The brain's dark energy. *Science*, 314(5803):1249–1250, 2006.

[29] Jacob Reinier Maat, Nikos Gianniotis, and Pavlos Protopapas. Efficient optimization of echo state networks for time series datasets. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7. IEEE, 2018.

[30] 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, 2015. Software available from tensorflow.org.