Decomposition of Chaotic Signals by Reservoir Computing

Author:
Yahya Robert Scerbo
51771662

Supervisors:
Dr Murilo Da Silva Baptista
Professor Antonio Politi

2020 - 2021

This thesis is presented in partial fulfilment of the requirements of the course PX4013, as part of an honours degree programme in Physics at the University of Aberdeen.

Submitted on March 26, 2021

Abstract

Reservoir computing is a recurrent machine learning architecture that simplifies the process of training recurrent neural networks. There are different implementations within the reservoir computing paradigm; in this work, the approach is the echo state network as described by Herbert Jaeger in 2001 [1]. At first the inner dynamics of the echo state network are explored. It is shown that adjusting the network parameters can lead to significant change in the behaviour of the states of the inner network. Subsequently, the effectiveness of the echo state network in modelling chaotic dynamics, and decomposing linearly combined chaotic signals is tested. Remarkably, it is demonstrated that within this framework, a model can be trained to replicate the dynamics of the chaotic Lorenz system accurately, for a short length of time. Furthermore, reservoir computing is shown here to be a reliable technique in the decomposition of combinations of chaotic signals.

Declaration of originality

This report is entirely my own composition. It has not been accepted in any previous application for a degree. It is a record of my own work and all verbatim extracts have been distinguished by quotation marks. My sources of information have been specifically acknowledged.

Signed 

Acknowledgements

I would like to thank Professor Antonio Politi and Dr Murilo Da Silva Baptista for their continued guidance over the past year, without their knowledge of dynamical systems and artificial neural networks this project would not have been nearly as successful. I should also mention how much I began to look forward to our weekly meetings. Even a physics based conversation can start to feel like a lighthearted catch-up, especially when you have passion for the subject.

To all my family, whether Scerbo, Mackenzie or Bukhari, I would like to thank you for your unwavering confidence in me. I'm hoping at least one of you will manage the whole thesis without falling asleep.

Shout-out to Mr Jake Lumsden! I'd like to thank him for offering insight whenever he had time and for providing a critical eye during the whittling down of my final draft.

To my wife Zelia I want to extend my deepest gratitude. You've stood by me for the entirety of my undergraduate degree, from start to finish. You are an ever-present source of inspiration, motivation and support.

I would like to dedicate this work to my late Grandfather Robert Edward Scerbo. It is because of you that I am who I am; to this day you are my beloved mentor and role model. You never ceased in preaching the benefits of an education. I'd like to thank you for encouraging me down this path and I hope you're proud of me for reaching this milestone. It is from you that I draw my work ethic.

Contents

List of Figures

List of Tables

1 Introduction

Artificial neural networks (ANNs) are machine learning techniques that mimic the learning processes which take place in biological networks. They are most commonly used to tackle computational and mathematical problems that are staggeringly complex to a human. Innovations and developments have been made possible by ANNs, in fields such as image and speech recognition, self-driving cars, time-series prediction and data processing. Despite this there are limits, huge swathes of data are required to train a network for even a simple task. A human may learn to recognise the image of an apple in seconds, but this can become days or weeks for even top-class artificial neural networks [2].

The focus of this thesis is an ANN framework called reservoir computing. In this work the benefits of using a reservoir computer (RC) over a typical ANN will be introduced. The capacity of an RC in tackling a few different problems will also be investigated. The RC design implemented throughout this work will be the echo state network (ESN) first described by Jaeger in 2001 [1].

This thesis has three main aims. The first aim is to analyse the states of an ESN without input or output: more specifically to explore the dynamics of the states of the reservoir layer, and analyse how they are affected by adjusting two of the RC hyper-parameters. The second aim is to train a model in the form of an RC that is able to autonomously predict future dynamics of the Lorenz system during its chaotic regime, and improve the model using an understanding of the hyper-parameters. The terms hyper-parameter and parameter will be defined at the beginning of the next chapter. The third aim is to train an RC to separate linearly combined chaotic signals, this approach requires only data and is similar to blind source separation. This final aim is similar to the famous *cocktail party problem*, i.e. how does one focus on listening to a single person out of a combination of loud voices in a room. Extracting a signal from noise in this way has applications in a multitude of fields including; signal processing, medical imaging [3], and weather analysis [4].

The structure of the thesis is as follows.

In chapter two, some information helpful to the understanding of the thesis is given. Beginning with clarification of some terminology, then a comprehensive account of the evolution of artificial neural networks. Following this, the Lorenz system is introduced; it is key to the pursuit of the second and third aims.

Chapter three provides a detailed account of the ESN structure implemented within this work.

Chapters three, four, and five correspond to the three thesis aims; with respect to the order in which they were introduced previously.

Finally, chapter seven will contain comprehensive conclusions. During this chapter there is an attempt to present well rounded evaluation and analysis of all results. In areas where an absolute conclusion cannot be drawn, there will instead be discussions around the methods leading to particular results and how they may be improved for future use.

Overall, the aims of this thesis were achieved. While investigating the dynamics of the reservoir layer, it is found that there are clear intervals for the values of the hyper-parameters leaking rate α and spectral radius ρ that lead the inner states of the reservoir to converge to zero, oscillate periodically and even behave irregularly. A chaotic system is also modelled. Remarkably, an autonomous network is shown to accurately predict six full time units of the chaotic Lorenz system. Another network is successfully designed to decompose linearly combined chaotic signals, this network estimates components from such a mixture with predictive accuracy comparable to published work [5].

All of the computer programs coded for this thesis were written in MATLAB. This language was chosen because of the frequent need for matrix manipulation throughout the investigation especially in the building of the network. MATLAB also does not make the task of plotting and computing numerical data overly complicated, the environment has also been shown to work well when dealing with and putting together neural networks [6].

1.1 Artificial Neural Networks

"The grand vision of neural networks is to create artificial intelligence by building machines whose architecture simulates the computations in the human nervous system" [2].

Traditionally computers were considered to be machines that 'compute', designed for calculating or iterating long arduous processes seemingly never ending for a human but completed in seconds by a computer. With advances in neural networks and deep learning, computers may soon completely overtake humans in other abilities such as voice or image recognition [7].

Artificial neural networks act as a framework upon which machine learning algorithms can work. A network is considered 'trained' when it has been fed 'training' data and its connection strengths have adapted. A trained network can be thought of as a 'model' to be used on new data for some complex task [8].

Artificial neural networks are inspired by real neural networks such as the human brain, made up of neurons and complex connections. In Figure 1 a comparison is drawn between a biological neuron and an artificial neuron (also known as a node or computation unit).

The strengths of the connections between biological neurons will often change when exposed to external stimuli, this is how biological creatures like humans retain information and how learning takes place.

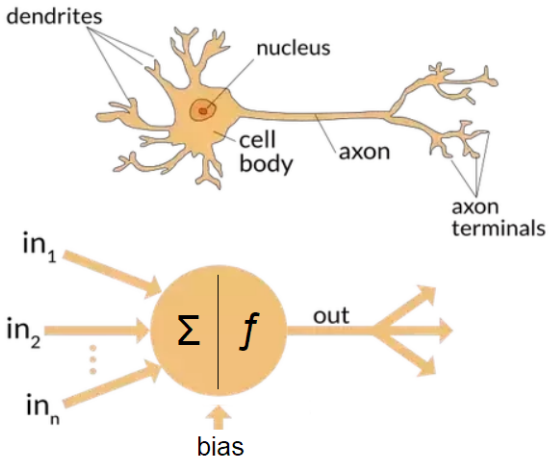


Figure 1: A comparison of a biological neuron from the brain and an artificial neuron or node in an artificial neural network [9].

Biological neurons are connected via axons and dendrites; the connecting regions between axons and dendrites are called synapses. Artificial neurons are connected via weights which serve the same purpose as that of the biological synapses. The connectivity of an artificial neural network is given via a weight matrix, which shows

1) Whether neuron $\boldsymbol{a}$ is connected to neuron $\boldsymbol{b}$.

2) The connection strength between neuron $\boldsymbol{a}$ and neuron $\boldsymbol{b}$.

Biological neurons receive signals through an electrochemical exchange of neurotransmitters. More specifically, biological neurons can connect via gap junctions electrically, and chemically via synapses. With an artificial neural network the synapses are replicated by a non-linear function. Every input that an artificial neuron receives is scaled with a weight, which has an impact on the function computed at that neuron. A more comprehensive account of these functions is given in the following section as well as more in depth detail on artificial neural networks. From this point on in the thesis, when referring to a neural network, this should be interpreted as an artificial neural network. Throughout the rest of the thesis, the terms neuron, node and computational unit will all refer to the computational units of an artificial neural network and will be used interchangeably.

2 Theory

Before discussing neural networks in more depth, some clarification on certain terminology is necessary. With advances in this area of science and technology misunderstanding around these terms has developed.

In general:

"Deep Learning is a kind of Machine Learning, and Machine Learning is a kind of Artifical Intelligence" [10].

More Specifically:

Artificial intelligence or AI for short, refers to any intelligence by a computer, robot or machine that is human-like. AI can refer to the ability of machines to complete human-like tasks such as problem solving, understanding language, recognizing objects, etc [11].

"***Machine learning*** is an application of AI that provides systems the ability to automatically learn and improve from experience without being explicitly programmed" [12].

Artificial neural networks are computing models whose structure mimics the networked structure of neurons in the brain. They include individual computation units called neurons, these neurons work together to produce an output function. Neural networks are the main tool in machine learning algorithms but are not algorithms themselves, they are a piece used in machine learning algorithms to process complex data into something a computer can process [13][14].

Deep learning is a machine learning technique that uses a deep neural network. Deep neural networks are multi-layer networks that contain two or more hidden layers [10]. Further information on network layers and network structures will be given in the next section, but for now it will suffice to say that hidden layers are subnetworks inside the whole network that are neither perturbed by the input signal nor contribute directly to the output layer, (another neural subnetwork).

Reservoir computing is a simple form of machine learning, introduced to use recurrent neural networks without the hassle of training the internal weights. This network is typically constructed by connecting in a directed fashion the input layer with the hidden layer and then to the output layer. In this framework, only the connections to the output layer are trained, saving time and computational effort [15]. Reservoir computing is the technique used throughout this investigation. With the specific structure being the echo state network as described by Jaeger [1].

Unlabelled data is data which is sampled from a natural occurrence; there is no explanation behind it, e.g. photos, videos, audio recordings, as well as data sets, and time-series. Whereas ***Labelled data*** is made from unlabelled data; this is achieved by augmenting each piece of data with a meaningful tag, label, or class; something that offers information, i.e. whether a photo might contain a car.

Supervised learning occurs in machine learning when a model is trained using well labelled data, i.e. some of the data is already known to be correct. ***Unsupervised learning*** is left unsupervised, and deals with data that is unlabelled; this type of learning typically deals with more complex tasks than supervised learning.

Parameter: In the context of machine learning, a parameter is a configuration variable that is internal to a model, and whose value is deduced from historical training data. Model parameters are learned during the training process. It could be said that a model can be considered a hypothesis, and the parameters allow this hypothesis to be tailored to certain data. For example, the weights of a neural network can be considered a parameter.

$\boldsymbol{Hyper-Parameter}$: Again in the context of machine learning, hyper-parameters are configurations that are external to the model. Their value is not given by the data but instead assigned by whoever implements the model. Optimum hyper-parameter values are unknown, they are determined through investigation. Hyper-parameters are supplied to the network manually. An example is the leaking rate used during training of a neural network.

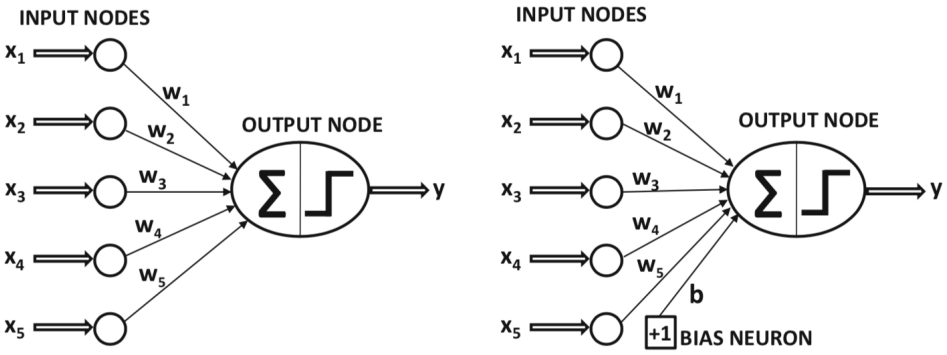
2.1 Evolution of Neural Networks

From the advent of the first neural networks all the way through until today there have been developments in their design and structure. Before discussing the network used in this thesis, the ESN as described by Jaeger [1], it is necessary to give a more in depth introduction to the most important types of neural network.

Neural networks can be single-layer or multi-layer. A single-layer neural network is the simplest form of a neural network, comprised of a single layer of input nodes which map weighted inputs directly to an output node. This simple arrangement is also referred to as the *perceptron*. In a multi-layer neural network, the neurons are arranged in layers, in this case there is a group of one or more hidden layers between the input and output layers. Multi-layer networks are referred to as *feed-forward networks* [2].

As previously stated, the perceptron is the simplest neural network. The perceptron was invented in the late 1950s by Frank Rosenblatt [16]. The fundamental structure of the perceptron can be seen in Figure 2.

Referring to both Figure 2.a) and Figure 2.b) the input nodes x_i on the left hand side will accept inputs in the form of numbers. Next, each input is multiplied by its corresponding weight w_i, these weight values dictate how significant the influence of each of the inputs is.



(a) Perceptron without bias (b) Perceptron with bias

Figure 2: The basic architecture of the perceptron [2].

The input layer does not perform any computation in its own right. All inputs, once multiplied by their weights are summed at the output node, where the total is compared with an activation function. In the case of the original perceptron the activation function was simply the sign function, which extracts the sign of a number and outputs either -1 or +1. Different choices of activation functions can be used such as sigmoids, gaussians or, as is frequently used in reservoir computing, the hyperbolic tangent function. More information on the choice of activation function will be given later on in the thesis.

Parts a) and b) from Figure 2 are almost identical but part b) includes what is known as an *input bias*, represented by the square shaped bias neuron. The input bias usually represents a constant. Consider as an example where the inputs are mean centred but we cannot allow the mean of the prediction to be zero. In this case, a bias can be added that captures this part of the prediction. Bias in neural networks can be considered analogous to constants in linear functions. Bias units are not connected to or affected by values in any previous layer [2][17].

Although the perceptron contains two layers, the input layer does not perform any computation and only transmits values. For this reason the input layer is not included in the count of the number of layers in a neural network, therefore the perceptron is said to be a single-layer network [2].

Multi-layer or feed-forward neural networks contain more than one computational layer, they are distinct from recurrent neural networks which are introduced further on in this section. The aforementioned perceptron simply has an input layer which transmits data to an output layer, in this scenario, all computations are observable by the user. The additional layers of a feed-forward network between the input and output are referred to as *hidden layers* because computations are not observable to the user. A diagram of the architecture of a simple feedforward network is shown in Figure 3.

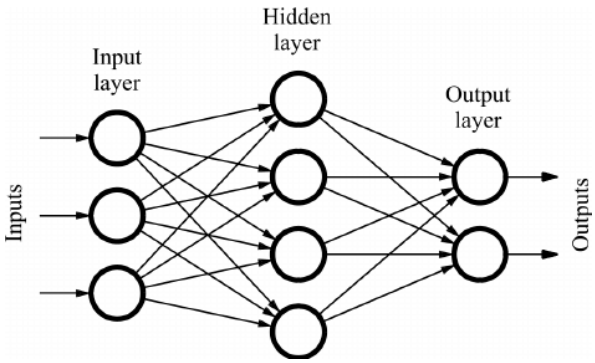
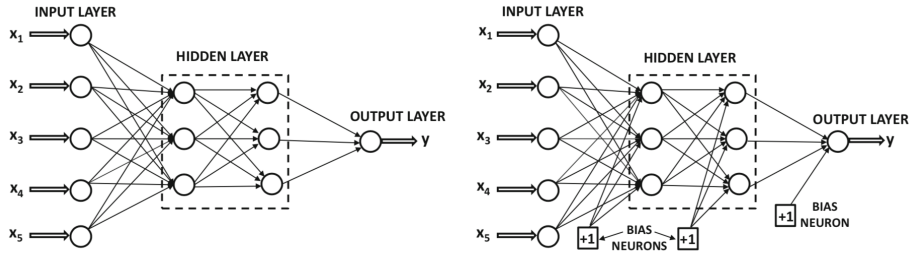


Figure 3: The basic architecture of a feed-forward network [18].

The architecture is given the name *feed-forward* because each layer feeds forward into one another in the direction of input to output. The standard layout of feed-forward networks assumes that all of the neurons in one layer are connected to those of the next layer [2][19]. Some examples of multilayer networks are shown in Figure 4, part (a) does not include any bias neurons, where as part (b) does.

With the architecure discussed, it is now important to introduce the *cost function* and how it is optimised in the output layer. The cost function is also often referred to as *loss function* or *error*, so the terms will be used synonymously throughout this thesis. When the perceptron was first introduced, optimisations were achieved in a heuristic manner with hardware circuits. There was no official idea of optimisation in machine learning like we have today. Despite the relatively outdated methods, the goal has remained the same; to minimise the error in the prediction by minimising the loss function. The loss function can be reduced through the changing of weights between computational units. A loss function may look like

$$Minimise_{\bar{W}} L = \sum_{(\bar{X},y)\in D} (y - \hat{y})^2, \tag{1}$$



(a) No bias neurons (b) With bias neurons

Figure 4: The architecture of a basic example of a feed-forward network with two hidden layers and single output layer [2].

where L is the loss function and w_i are the entries of the weight matrix $\bar{W}$. The input signals x_n are stored in the vector/matrix $\bar{X}$. The desired output signal y is given as part of the training data D. The output signal calculated by the network is given by $\hat{y}$ [2]. Further detail on weight matrices for input, hidden, output layers and how they will be initialised for this thesis will be given in chapter 3.

The final concept to be introduced in this section is *backpropagation*. In the single-layer network discussed earlier, the error can be easily computed as a direct function of the weights, in mulyi-layer networks the error becomes more complicated as it is a composite function of the weights in earlier layers. The gradient of a composition function is computed using the backpropagation algorithm, the algorithm works through the chain rule of differential calculus. The process finds the loss gradient in terms of summations of local-gradient products over the various paths from a node to the output. There are two steps to backpropagation: the *forward phase* and *backward phase*.

- *forward phase:* The network is fed a training instance. This causes successive stages of computation across each layer using the weights currently held by the network. A comparison can then be drawn between the predicted output and that of the training instance; the derivative of the cost function with respect to the output can be computed. After this the derivative of the cost is computed with respect to the connection weights in every layer during the backward phase.

- *backward phase:* The purpose of the second step is to acquire the gradient of the cost function with respect to the connection weights by using the chain rule of differential calculus. The gradients can then be used to update the weights. The gradients are learned in the backward direction, beginning at the output, because of this the process is referred to as backpropagation.

Neurons are updated over many backpropagation cycles, some networks may require thousands of iterations with the training data to acquire the weights of each node. With gradient descent or other optimisation algorithms, the cost function can be minimised as a function of the weights [2].

2.2 Recurrent Neural Networks

Recurrent neural networks (RNNs) are a class of neural network architecture distinct from more basic feedforward networks, they are primarily used to find patterns in sequences of data. Examples of some typical data types include: time-series, text sentences, other discrete data that carry sequential dependencies or in the case of this project nonlinear signal processing and control [20]. The key difference between RNNs and feedforward networks is how data moves through the network. Feedforward networks move data through the network in a non-cyclical manner, but RNNs work in cycles and transmit information back into themselves. This means that the network can consider previous inputs $X_{0:t-1}$, and not just the current input X_t, this difference can be seen in Figure 5 [21].

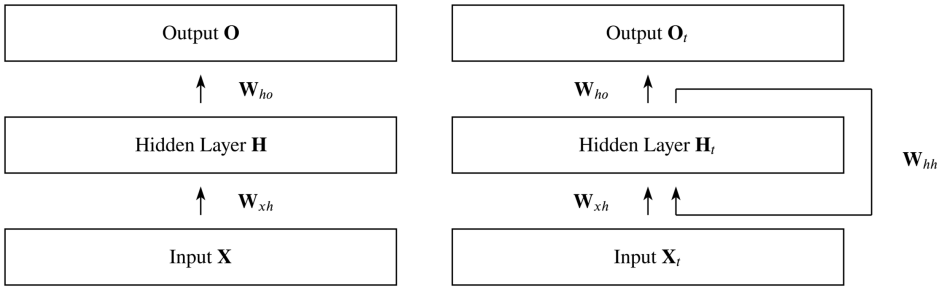


Figure 5: Visualisation of the differences between feedforward and recurrent neural networks [21]. Feedforward on the left, RNN on the right.

RNNs operate not only on an input space but also on an internal *state* space - a history of what has already been processed by the network [20]. A key feature of sequences is that successive objects depend on each other, meaning that it is more useful to receive a particular input only after the previous inputs have been processed and converted into a hidden state. Feedforward networks are not equipped to manage this kind of task. These types of tasks are satisfied using RNNs, as there is a one-to-one correspondence between the layers in the network and the specific positions in the sequence. A position in the sequence is also referred to as its *time-stamp* [2].

Although the presence of cycles in the topology of RNNs was only mentioned briefly earlier on, this is actually a significant difference from typical feedforward networks that shall be discussed further. With cycles an RNN is able to build self-sustained temporal activation dynamics through its recurrent connection pathways, despite the lack of input. This means that the network can be considered as a dynamical system, whereas typical feedforward networks are mere functions. In the case of being given an input, the network keeps within its internal state a nonlinear transformation of the input history, therefore the network has what can be referred to as *dynamical memory* [22].

So far we have established that the network operates as a dynamical system, where the hidden states of the network are a function of the inputs at the current time stamp, the hidden states at the previous stamp, as well as any backpropagation into the system. The manner in which the internal states are updated over time can be given by an update equation, an example of which is given for a discrete-time neural network with K input units, N internal network units, and L output units [1]:

$$\mathbf{x}(n+1) = \mathbf{f}(\mathbf{W}^{in}\mathbf{u}(n+1) + \mathbf{W}\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{y}(n)), \tag{2}$$

where the $\mathbf{W}^{in}$ respresents the $N \times K$ weight matrix for the input weights. $\mathbf{W}$ is the $N \times N$ weight matrix for the internal connections. $\mathbf{W}^{out}$ is the $L \times (K + N + L)$ weight matrix for the connections to the output units, and $\mathbf{W}^{back}$ is the $N \times L$ weight matrix for connections that project from the output back into the internal units. Detailed information on the update equations for the networks used in this project, which are structurally different from the one in equation (2), will be given in chapter 3. For now n represents discrete time-steps, while $x(n)$ and $x(n+1)$ give the current and succeeding internal state of one of the computational units respectively. Furthermore, $u(n+1)$ and $y(n)$ are the input signal at the proceeding time-step and the output signal at the current time-step respectively. The f are the internal unit's output functions, this is another example of an activation function similar to those mentioned earlier while discussing the perceptron, but one is not likely to find an RNN using step functions, sigmoid functions are more suitable as they have a higher smoothness. A diagram of a recurrent neural network similar to this can be see in Figure 10 [1].

It has been shown mathematically [23] that RNNs have the *universal approximation property* and therefore can model dynamic systems with arbitrary precision; additionally it has been shown that with enough neurons, an RNN can be computationally Turing-equivalent [2][24]. Increasing the number of internal units allows the network to complete an increased number of calculations and handle more complex tasks than a typical feedforward network. Although there is a trade-off that comes along with the increased computational power, increasing the number of nodes also causes the computational time to increase. Finding an appropriate number of internal units for the investigations in this project is a task discussed later on in the thesis.

RNNs are designed to process time-series and other types of sequential data. As is typical with sequences or sequential data, successive data points can often depend on each other. This means that a certain input is only useful after previous inputs have been fed to the network and converted into a hidden state, this is not possible with a typical feed-forward network. With RNNs, inputs at n are able to interact with inputs from previous times that have been made into a hidden state.

More specifically, let us say that there is an input $\bar{x}_n$ at each n, and a hidden state $\bar{h}_n$ that adjusts at each n as new information is fed to the network. At every n there is also an output $\bar{y}_n$ which in the case of sequential data might be the future prediction of $\bar{x}_{n+1}$. From this the hidden state at n depends on the input at n and the hidden state at $n-1$:

$$\bar{h}_n = f(\bar{h}_{n-1}, \bar{x}_n). \tag{3}$$

The output probabilities are calculated from the hidden states using a distinct function, $\bar{y}_n = g(\bar{h}_n)$. Given this information it makes sense to think of an RNN as being a *time-layered* network, so there is a different node for the hidden state at each time.

The matrices which contain the connection weights are shared by multiple connections in this time-layered network, this is to make sure the same function is used at each time, n. During backpropagation the sharing of weight matrices and temporal length are taken into consideration when adjusting the weights during learning. This particular kind of backpropagation is referred to as *back propagation through time* (BPTT).

BPTT occurs in three steps; the input is run sequentially forward through time and errors are computed at each time stamp, this is followed by computing the gradients of the connection weights in the backwards direction on the time-layered network (ignoring the fact that weights in different time layers are shared); finally all the shared weights corresponding to different instantiations of a connection in time are summed to create a unified update weight for each weight parameter [2].

There are other learning methods available one can consider for training RNNs besides BPTT; the real-time recurrent [25], the Atiya-Parlos [26] and the extended Kalman-filtering [27] are all valid learning methods.

2.3 Prominent Network Structures

Just as there are different learning methods, within the realm of RNNs there are a multitude of different network architectures, some of which were developed for use against particular problems. A list [28] of a few of the most prominent neural network architectures will be given, (note that not all of them are RNNs).

Fully Recurrent Neural Network: Proposed by Williams and Zipser [29] in 1989, a fully recurrent neural network is the basic architecture of RNNs. The neurons of the network are sorted into a layered structure, every neuron in each layer has a one-way connection to every neuron in the proceeding layer. The L_2 error is computed at each layer, the total error is then given by adding the individual errors of each layer. A concise example of a recurrent neural network is shown in Figure 5, on the right hand side of the figure.

Recursive Neural Network: These networks were developed with the purpose of processing structured objects of arbitrary shape, these would include logical terms, trees or graphs. In networks of this type the input is processed hierarchically, a set of weights is applied to a structured input to produce a structured prediction. This type of network has been shown to be superior in term-classification problems [30]. Figure 6 shows an example of a basic recursive network.

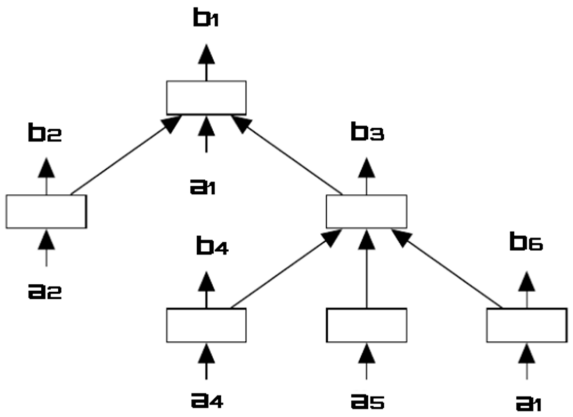


Figure 6: The figure shows a basic recursive neural network. As in a RNN the weights are shared but with recursive networks dimensionality is constant over each node, instead of the length of the sequence [31].

Hopfield Network: First described in 1974 by William Little [32], but named for and popularised by John Hopfield in 1982 [33]. Hopfield networks are made up of binary threshold units (taking value of {0,1} or in some cases {-1,+1}) with recurrent connections between them. The connections in a Hopfield network are symmetric, meaning that the weight between two nodes is always the same independent of the route taken between them. Networks of this type operate as *content-addressable memory* (CAM) systems, this system type is mainly used in high-speed searching applications. The ability to perform high-speed searches stems from the symmetric connections and the Hebbian learning rule [34], the go-to learning method of these particular networks. Based in neuroscience, Hebbian learning states; if two neurons on either side of a synapse are activated simulatenously then the strength of that particular synapse is increased. Replicating this in the network contributes to the loss function tending to a minimum [2][34]. An example of the topology of a Hopfield network can be seen in Figure 7.

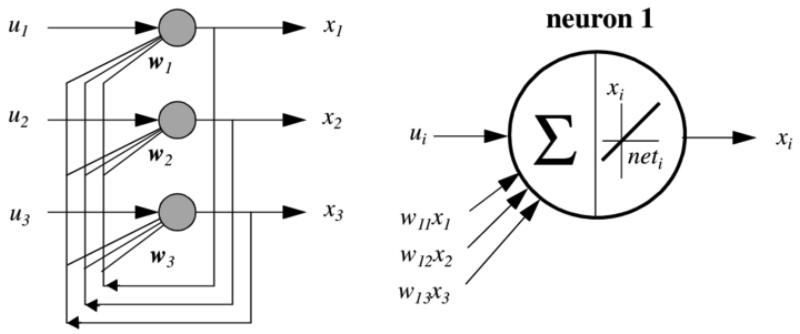


Figure 7: The topology of Hopfield networks, this example is made up of three neurons [35].

Elman Networks and Jordan Networks: Introduced by J Elman in 1990, Elman networks have three layers comprised of an input, output and hidden layer. The hidden layer is connected to a layer of *context units* with a fixed weight equal to one. This additional context layer holds the values from the hidden units from every preceding time-step meaning that the network can maintain a state, leading to sequence-prediction capabilities [36]. Similar to Elman networks are Jordan networks which were developed a few years earlier in 1986. Jordan networks differ in that their context units are fed by the output nodes instead of the hidden nodes [37].

ESN: Detailed information on ESNs is given later on in this section as this is the network of choice for the investigations of this thesis. See Figure 10 for an example.

Neural History Compressor: In J Schmidhuber's 1992 paper he introduced a deep learner able to carry out credit assignments over hundreds of neural layers by unsupervised pre-training for a stack of RNNs. Using this he was able to compress descriptions of sequences while avoiding any loss of data. The network anticipates future inputs according to preceding inputs, an input fed to an RNN in the stack will only be passed on to the following higher level RNN if the input is unexpected. So each higher level RNN is receiving a compressed version of the data in the RNN a level below, meaning that a reduced representation of the input sequence can be found at the highest level RNN. This method has been shown experimentally to be less computationally intensive than typical neural networks [38].

Long Short-Term Memory (LSTM): Many neural networks suffer from issues associated with vanishing and exploding gradients. The problems arise when the weights are updated according to the partial derivative of the error with respect to the current weight in each iteration of training, in some cases the gradient is vanishingly small, meaning the weight will not be able to significantly change in value. LSTM networks avoid these issues. Like other RNNs, LSTM networks can be unfolded into time-layered networks with feedforward structure made up of repeating modules. When unfolded, the repeating modules of a typical RNN will contain a single activation (e.g. hyperbolic tangent), but in an LSTM the repeating modules are cells, made up of input, output and forget gates all using sigmoid functions. The main feature of LSTMs are the cell state, the cell remembers values over arbitrary periods of time and the gates control the information passed in and out of the cell. Gates are a way to optionally allow information into the cell as well as control the internal states of each neuron and how much they change at each time-step. The cell state can be considered to be a form of long-term memory as the cell is able to retain partial information from previous states through the use of the gates. A diagram of an example cell can be seen in Figure 8. This architecture was first introduced in 1997 by Hochreiter and Schmidhuber [2][39][40][41].

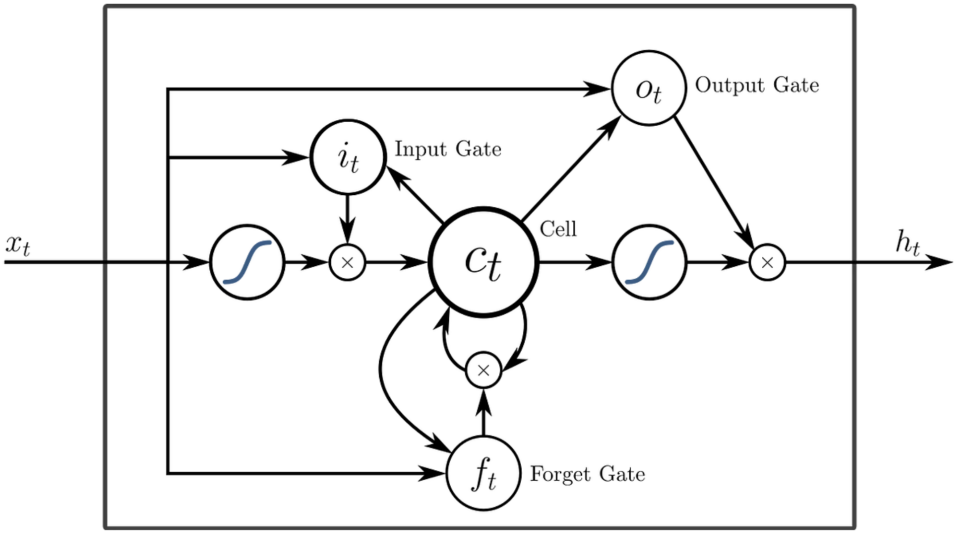


Figure 8: Diagram of a cell of an LSTM neural network [42].

2.4 Introduction to Reservoir Computing

Reservoir computing is a framework that streamlines the task of training RNNs, vastly reducing computational time and effort. Using this approach the task of adjusting every single weight in the network is not necessary, instead it considers a fixed *reservoir* that requires no training. From this point in the thesis, the hidden layers of the network may be referred to as the reservoir or reservoir layer. In the early 2000s reservoir computing was developed concurrently and independently by Jaeger [1] and Maass *et al.* [43]. Jaeger first introduced ESNs through a machine learning approach, where as Maass *et al.* came to *Liquid State Machines* with neuroscientifically realistic spiking neurons. These methods were shortly grouped under the name reservoir computing, as they both use the same basic rules [44].

The two approaches mentioned so far have some fundamental features in common and consist of two main parts; the first is the aforementioned reservoir which is an RNN with fixed weights that acts as a "black-box" model of a complex system, and the second is referred to as the *readout* layer which is usually a linear classifier connected to the reservoir via adjustable weights. The network of choice in this thesis is the ESN described by Jaeger [1]. Although ESNs are similar to liquid state machines, the former use typical activation functions (sigmoid and hyperbolic tangent for example) whereas the latter uses spiking neurons with binary outputs. The term "echo" comes from the states of the units of the hidden layer being a function of the input history fed to the network. [45]

The connection strengths within the reservoir layer are assigned randomly, selected from some distribution of numbers and permanently set. During training, the output weights are found by mapping the state of the reservoir to the desired output layer. Therefore, training all of the connections of the network is not necessary. A positive consequence of the reservoir computing approach is that backpropagation is not necessary, reducing the time spent on the training of the network [2]. This approach has also been shown to tackle the problem of instability in the training of RNNs [46].

There are some additional features that make the ESN unique when compared with typical RNNs.

- Sparse connectivity is used for the units of the reservoir layer, this means that only a small percentage of the neurons in this layer are connected to each other (typically less than 20%), or in other words; there is a connectivity of zero between the majority of units. According to Jaeger; sparse connectivity gives rise to relative decoupling of subnetworks, encouraging the development of individual dynamics [1], but this claim is still disputed [2].

- During initialisation the reservoir weight matrix is rescaled by the spectral radius. The entries of the weight matrix of the hidden layer are normalised by dividing through by the largest eigenvalue. The matrix is then rescaled by multiplying each entry by the new value for the spectral radius. This is done so in a way that the autonomous dynamics of the network can be determined regardless of the number of neurons and the particular network graph [2].
- The number of nodes in the network is typically high, for example the hidden layer usually consists of 100-400 units, but this can become even higher depending on the difficulty of the task at hand.

2.5 The Data - The Lorenz System

The Lorenz system plays an important role in this project. It is employed in demonstrating the capability of the ESN in predicting the dynamics of non-linear systems. It is then used to create the scalar signals that are put together as a combination to then be decomposed. For these reasons an introduction to the Lorenz system is given here.

The Lorenz system is a set of three ordinary differential equations first studied by Edward Lorenz in 1963. The system was developed as a simplified mathematical model for atmospheric convection and is given by the equations,

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y - x), \\ \frac{dy}{dt} &= x(\rho - z) - y, \\ \frac{dz}{dt} &= xy - \beta z. \end{aligned} \tag{4}$$

These equations are deterministic, non-linear and famously exhibit chaotic, non-periodic solutions for certain intervals of the parameter values ρ, σ and β. For other values of these parameters the system can also exhibit periodic behaviour (equilibrium and limit cycles) as well as marginal states. Limit cycles are oscillations with fixed amplitude and period. Equations (4) themselves describe a two dimensional fluid layer being uniformly warmed from below and cooled from above. The variables describing this interaction are x, y, z which correspond to the intensity of the convection motion, the temperature difference between the ascending and descending currents, and the distortion of the vertical temperature profile from linearity respectively [47].

The parameters ρ, σ, β are proportional to the Prandtl number, the Rayleigh number and physical dimensions of the fluid layer respectively. The Prandtl number is the ratio of momentum diffusivity to thermal diffusivity [48], and the Rayleigh number is the buoyancy-driven flow [49]. The values originally used by Lorenz were $\rho = 28$, $\sigma = 10$ and $\beta = 8/3$. Since this set of values has been used in work by Lorenz *et al*, they are used to create one of the signals used later on in this thesis. The other signal will be created from a second Lorenz system with a slightly different set of parameter values. This means the two signals will be created by different dynamics and as such have different dynamical properties [50].

Plotting $\{x(t), y(t), z(t)\}$ during the chaotic regime of the parameters allows us to see what is known as the Lorenz attractor, which is a strange attractor. An attractor is defined as a set of states which a system tends to evolve toward, for a multitude of initial conditions of the system. The term 'strange' is attributed to any attractor which has a fractal structure, this includes the Lorenz system [51]. An example of the Lorenz attractor during its chaotic regime can be seen in Figure 9.

The interest in studying chaotic systems comes from their unpredictable dynamics, their sensitivity to changes in initial conditions and their broad frequency spectrum. Using the data from chaotic systems as the input to a network will give a comprehensive display of the prediction and signal decomposition abilities of the ESN [5].

The Lyapunov exponent is a quantity that characterises a dynamical system such as the Lorenz. The quantity gives an indication of the rate of separation of infinitesimally close trajectories [52]. Two trajectories in phase space with initial separation δZ_0 diverge with a rate given by:

$$\|\delta Z(t)\| \approx e^{\lambda t}\|\delta Z_0\|, \tag{5}$$

where λ is the Lyapunov exponent. For the Lorenz system $\lambda \approx 0.9$.

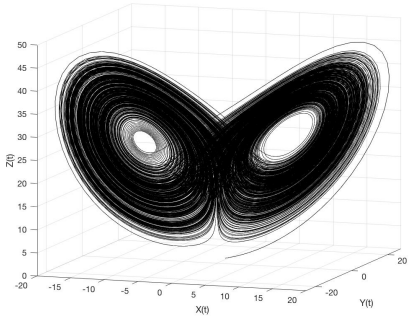


Figure 9: An example of the Lorenz attractor generated by integrating equations (4). Initial $\{x, y, z\} = \{1, 1, 1\}$, with parameters $\sigma = 10, \rho = 28, \beta = 8/3$.

3 Reservoir Computing Structure

The structure and procedure for the ESN was first described by Jaeger [1], but Parlitz and Zimmermann in their paper give a description of how to program a functioning ESN in very simple terms [46]. Due to the clarity of their explanations and the ease with which the required matrices and vectors can be updated using MATLAB, the paper helped hugely in writing the codes used in this project, and also significantly contributed to one's understanding of the structure of the network, a simple example of which can be seen in Figure 10 [46].

A list of all hyper-parameters and parameters used in the following explanation can be found in table 1. This list is given to avoid any confusion over terminology.

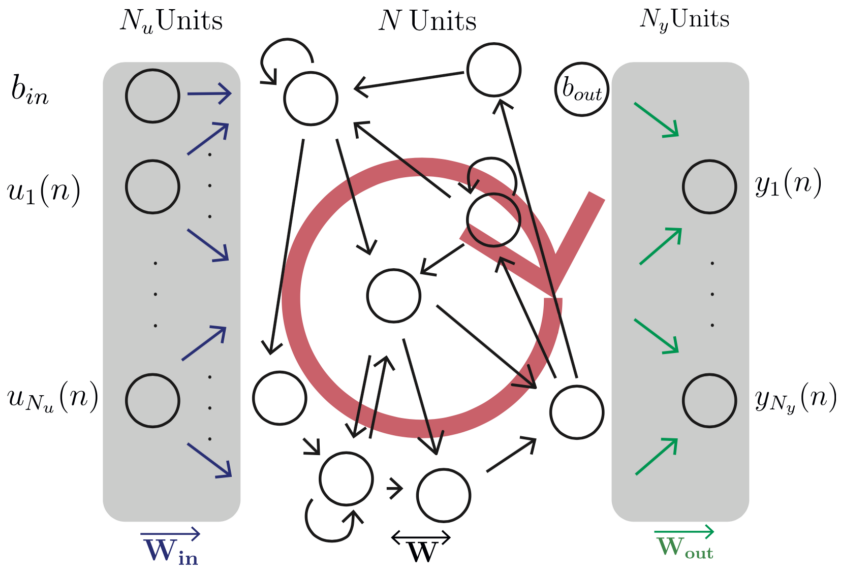


Figure 10: ESN general structure as described by Jaeger [1], then Parlitz and Zimmermann [46]. From left to right the input signal $\vec{u_n} = \vec{u}(n)$ passes through N_u input units, through a reservoir of N units, then N_y output units where the output signal $\vec{y_n} = \vec{y}(n)$ is calculated.

When using the echo state approach the first parameter that needs to be set is the number of reservoir units, denoted always as N hereafter. As N increases the efficiency of the network to produce an accurate desired output increases, unfortunately this comes with a trade-off of high run-times during training. The number of input and output units must also be set; these are represented as N_u and N_y respectively. These will usually be proportional to the number of input signals fed to the network and number of output signals extracted from the network respectively.

Table 1: Summary of echo state network parameters and hyper-parameters

Variable	Description
N	Number of reservoir units
N_u	Number of input units
N_y	Number of output units
$\vec{u}_n$	Input vector
$\vec{y}_n$	Output vector
$\boldsymbol{W}$	Weight matrix
$\boldsymbol{W}_{in}$	Input matrix
$\boldsymbol{W}_{out}$	Output matrix (Trained output weights)
$\vec{s}_n$	State vector
$\vec{x}_n$	Extended state vector
$\boldsymbol{X}$	State matrix
$\boldsymbol{Y}$	Output matrix (output vectors over time)
b_{in}	Input bias
b_{out}	Output bias
ρ	Spectral radius
ϵ	Sparsity
α	Leaking rate
f_{in}	Nonlinear transfer function

The connections in the reservoir layer are described by the weight matrix $\mathbf{W} \in \mathbb{R}^{N \times N}$. The input signal is stored in the vector $\vec{u_n} \in \mathbb{R}^{N_u}$ and the output signal is stored in the vector $\vec{y_n} \in \mathbb{R}^{N_y}$ where n denotes discrete time. The reservoir layer is created by initialising the matrix $\mathbf{W}$ whose dimensions correspond to reservoir size N. The entries of $\mathbf{W}$ are assigned randomly from an interval $[-0.5, 0.5]$. After this $\mathbf{W}$ is made sparse by setting the majority of its entries equal to zero. The sparsity ϵ is a hyper-parameter assigned from the interval $\epsilon \in [0, 1]$, e.g. a sparsity $\epsilon = 0.2$ corresponds to keeping 20% of the entries non-zero.

Next, the reservoir is normalised and rescaled to a new spectral radius ρ. The spectral radius is another hyper-parameter. The largest absolute eigenvalue of $\mathbf{W}$ is found according to $\rho(\mathbf{W}) = \mathbf{max}\{|\mu_\mathbf{1}|, |\mu_\mathbf{2}|, \ldots, |\mu_\mathbf{N}|\}$. Where $\mu_n (n = 1, \ldots, N)$ are the eigenvalues of $\mathbf{W}$. The entries of $\mathbf{W}$ are divided by the largest absolute eigenvalue, meaning that the spectral radius of the normalised matrix becomes equal to 1. The normalised matrix is then rescaled to a new spectral radius by multiplying all entries by the desired value ρ. The spectral radius is investigated later on in the thesis, for now it will suffice to say that altering the spectral radius can cause the reservoir layer to either expand or contract permitting the autonomous dynamics of a network with linear activation function to be chaotic or periodic respectively.

With the reservoir units and their connections initialised, attention can then be turned to the input and output matrices. The connections between the input and the reservoir are stored in the matrix $\mathbf{W_{in}} \in \mathbb{R}^{N\times(N_u+\mathbf{1})}$, whose entries are set randomly from the interval, $[-0.5, 0.5]$. The output weights are stored in the matrix $\mathbf{W_{out}} \in \mathbb{R}^{N_y\times(\mathbf{1}+N+N_u)}$ whose entries are also set randomly from the interval $[-0.5, 0.5]$, but these will change in response to training, unlike the entries of $\mathbf{W}$ and $\mathbf{W_{in}}$ which remain set once assigned.

The state of the network is described by the vector $\vec{s_n}$. Each component of $\vec{s_n}$ represents the state of one of the N at time n. They are initially assigned values randomly from the interval $[-1, 1]$. The states $\vec{s_n}$ are adjusted via the update equation,

$$\vec{s_n} = (1-\alpha)\vec{s}_{n-1} + \alpha f_{in}(\mathbf{W_{in}}[b_{in}; \vec{u}_n] + \mathbf{W}\vec{s}_{n-1}). \tag{6}$$

With leaking rate α and input bias $b_{in} = 1$. The leaking rate is another hyper-parameter which determines how significantly $\vec{s_n}$ change over each n. The nonlinear transfer function f_{in} (also known as the activation function) is used throughout this thesis as the hyperbolic tangent $tanh(\cdot)$. A difference between the update equation given here by Parlitz and Zimmermann [46], and the update equation given by Jaeger [1], is that a *backprojection* matrix is not required. In Jaeger's description, this matrix is used for feedback from the output units back into the reservoir, but with the network described by Parlitz and Zimmermann this is not necessary. In the case of the ESN developed by Parlitz and Zimmermann [46], the output weights are updated through *trainer-forcing* where $\mathbf{W_{out}}$ is tuned according to inputs and already known outputs, delivering a model that can still be used to make predictions with new inputs. Given $\vec{s_n}$, the extended inner state vector $\vec{x_n}$ is,

$$\vec{x_n} = [b_{out}; \vec{s_n}; \vec{u_n}] \in \mathbb{R}^{1+N+N_u}. \tag{7}$$

With an output bias $b_{out} = 1$. $\vec{x_n}$ are stored in the state matrix $\mathbf{X} \in \mathbb{R}^{(1+N_u+N)\times T}$, where each column of the matrix will correspond to $\vec{x_n}$ over time. The output vectors $\vec{y_n}$ are stored in the output matrix, $\mathbf{Y} \in \mathbb{R}^{N_y\times T}$, whose columns will correspond to $\vec{y_n}$ over time. Both matrices $\mathbf{X}$ and $\mathbf{Y}$ are filled with zeros during initialisation, but these entries are updated during training and prediction.

3.1 Network During Training

The network is trained in a loop of total length, T, this length is the *training time* and corresponds to the number of time-steps that training data is made available to the network. The network at each step of the loop n is fed an input, this occurs by setting $\vec{u_n}$ equal to the input training data.

Simultaneously, desired outputs are assigned to $\vec{y_n}$. The inputs and outputs used throughout this project take the form of different time-series, but details of these will be given in subsequent chapters as needed.

Over every step of the loop n, $\vec{u_n}$ and $\vec{s_n}$ are stored as $\vec{x_n}$ in the columns of $\mathbf{X}$. The $\vec{y_n}$ are stored in the columns of $\mathbf{Y}$. From this it is easy to see that the number of columns in $\mathbf{X}$ and $\mathbf{Y}$ corresponds to T. Over each n the weights $\mathbf{W_{out}}$ are adjusted, this occurs through minimising the cost function,

$$C(\mathbf{W_{out}}) = \sum_n ||\vec{y_n} - \mathbf{W_{out}}\vec{x_n}||^2 + \lambda Tr(\mathbf{W_{out}}\mathbf{W_{out}^T}), \tag{8}$$

with respect to $\mathbf{W_{out}}$. The first term measures the L_2 error of the prediction and the second is a regularisation term with strength $\lambda > 0$ which is used to avoid overfitting of $\mathbf{W_{out}}$. An illustrative example of overfitting is given in the appendix. In a typical RNN training is usually done through an iterative gradient descent method, a picture of which is shown in Figure 11. Fortunately with the reservoir computing approach *only* the output weights require training. This gives the freedom to train via a linear regression algorithm. Therefore the desired output weights $\mathbf{W_{out}}$ are the linear regression weights of the desired $\mathbf{Y}$ and the harvested $\mathbf{X}$, given by,

$$\mathbf{W_{out}} = \mathbf{Y}\mathbf{X^T}(\mathbf{X}\mathbf{X^T} + \lambda\mathbb{1})^{-1}. \tag{9}$$

Jaeger *et al* offer an explanation [53] of the lengthy derivation that shows how equation (8) can become equation (9).

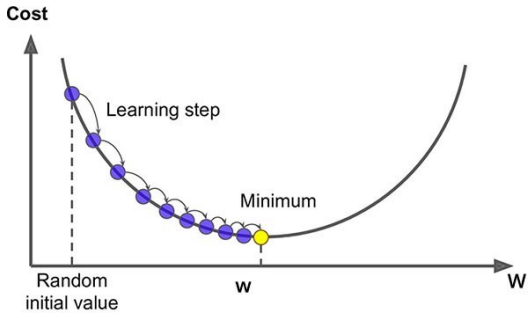


Figure 11: Gradient descent is often used during the training of a typical RNN. Over each iteration we move in the direction of steepest descent [54].

The network is allowed to run for a transient period T_0. During this period nothing is saved to either $\mathbf{X}$ or $\mathbf{Y}$, meaning that training will not yet occur. After this, in the range $T_0 < n \leq T_0 + T$ the input is applied, $\vec{x_n}$ are collected into $\mathbf{X}$ and $\vec{y_n}$ are collected into $\mathbf{Y}$. Over each iteration the matrices are updated with new input and output training data, which causes the entries of $\mathbf{W_{out}}$ to adjust according to equation (9).

3.2 Network During Prediction

With slight changes to the code, the network can be used to make predictions. The key difference is that instead of assigning desired outputs to $\vec{y_n}$, it is calculated using the trained $\mathbf{W_{out}}$ and $\vec{x_n}$,

$$\vec{y_n} = \mathbf{W_{out}}\vec{x_n} = \mathbf{W_{out}}[b_{out}; \vec{s_n}; \vec{u_n}]. \tag{10}$$

In matrix notation this can be given as

$$\mathbf{Y} = \mathbf{W_{out}}\mathbf{X}. \tag{11}$$

Equation (9) is completely left out during the prediction stage, otherwise the network will be retrained according to its own predictions. Predicted outputs are already slightly inaccurate, so allowing $\mathbf{W_{out}}$ to change in this way will cause errors to accumulate until the model no longer reflects the situation it was intended for.

From equations (10) and (11), it is clear that at each n the network is predicting $\vec{y_n}$, depending only on a given $\vec{u_n}$ stored in $\vec{x_n}$, and the trained $\mathbf{W_{out}}$. It is also possible to feed the output at n back into the network as the input at $n+1$. This means the network is making predictions based on its own previous predictions, meaning it is operating autonomously. This approach of feeding the output back in as an input is used in chapter 5, which is an example of a feed-back loop introduced in the work by Jaeger [1]. This type of feed-back loop is used in chapter 5 because the aim of this chapter is to predict *future* measurements of the Lorenz system.

Whereas, in chapter 6 the input is a mixture of signals at n, and the output is a component of this mixture also at n. This means that the output at each step cannot be used as the input for the proceeding step. Overall the presence of a feed-back loop depends on the task each network is trained for.

For clarity, throughout the remainder of the thesis n will denote discrete time during training and prediction. Also t will denote the integration step of the measurements of a given time-series. RC has different forms but ESN is the implementation of choice in this work, so in the context of the remainder of the thesis the terms ESN, RC and network will all be used interchangeably to describe the networks that are subsequently created.

4 Impact of Hyper-Parameters on Reservoir Dynamics

This chapter is dedicated to investigating the dynamics of the states of the reservoir layer. More specifically how the trajectories of these states are impacted by changing two of the hyper-parameters, namely the leaking rate α and the spectral radius ρ while there is no input (autonomous network).

Recall that an RC is made up of input, reservoir, and output layers. At this point the input and output can be ignored, as well as their corresponding matrices and vectors. The only parameters and hyper-parameters used in this section are given in table 2. Only two from this list will be varied and investigated; α and ρ, the remainder of the list will be given value but remain constant throughout this chapter.

Table 2: Parameters and hyper-parameters while input-free

Variable	Description
W	Reservoir weights
N	Number of reservoir units
$\vec{s}_n$	State vector
$\vec{x}_n$	Extended state vector
ρ	Spectral radius
ϵ	Sparsity
α	Leaking rate
f_{in}	Nonlinear transfer function

4.1 Methods

An adjustment is made to the equation that updates the states of the reservoir layer. Equation (6) becomes,

$$\vec{s_n} = (1-\alpha)\vec{s}_{n-1} + \alpha f_{in}(\mathbf{W}\vec{s_{n-1}}). \tag{12}$$

Noticeably, $\vec{s}_n$ now only depends on its previous value $\vec{s}_{n-1}$, α, f_{in} and the weights $\mathbf{W}$.

A reservoir layer is initialised by creating $\mathbf{W}$, containing information on the connections between the reservoir units. The $\vec{s_n}$ are initialised randomly as stated earlier. The reservoir is now allowed to update for a number of total time-steps T. During the remainder of this chapter T will not denote training length but simply the number of steps the reservoir layer is allowed to run. As the reservoir updates over time the $\vec{s}_n$ are put into $\vec{x}_n$ which are then stored in $\mathbf{X}$. This can then be used as a readout of the states over time.

The α and ρ are varied one at a time to test their effects on the states of the reservoir.

To change ρ, one must initialise an entirely new set of reservoir units, so for each of the five values of ρ investigated, five different reservoirs are created. The α is then varied over each of these reservoirs.

The values of α and ρ, investigated can be seen in table 3. These values are partially sampled from a paper by Parlitz and Zimmermann where they investigate the ESN [46].

Table 3: Examined values of leaking rate and spectral radius

Hyper-Parameter	Hyper-Parameter Values
ρ	$0.5, 0.95, 1.1, 1.5, 3.0$
α	$0.05, 0.5, 0.95, 1.1$

4.2 Results

It is important to give a clearer idea of what the trajectories of individual states look like. In Figure 12 there is a plot of the dynamics of the states of 10 computational units. One can clearly see that the states are initialised randomly from the interval $[-1, 1]$ but eventually converge toward zero.

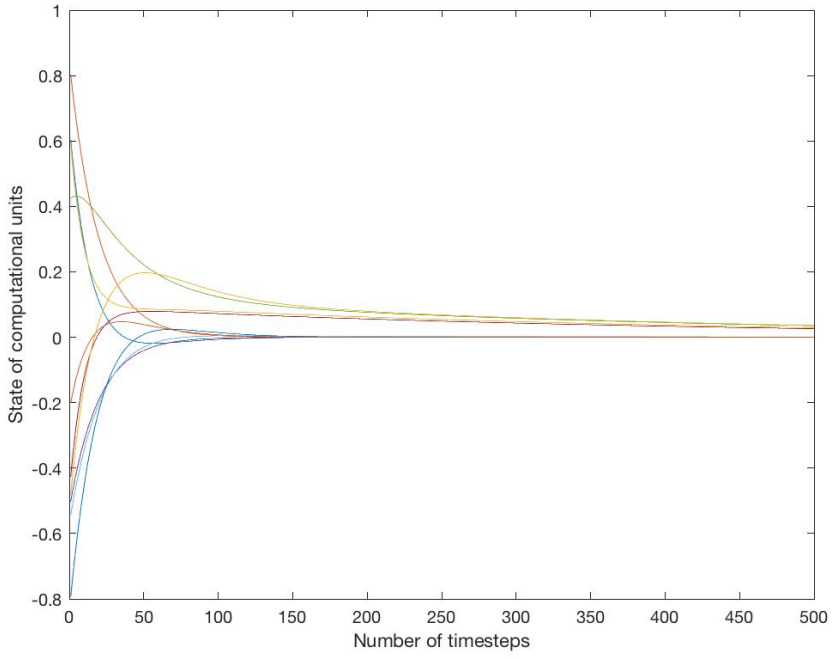


Figure 12: An example of how the states of the network in the hidden layer can develop over time. In this example $N = 10, \alpha = 0.05, \epsilon = 0.2, \rho = 0.95$. The reservoir was allowed to run $T = 1000$. (Showing only first 500 time-steps for clarity).

To thoroughly investigate the states, a higher number of N are needed. With this increase observing individual dynamics becomes too difficult so instead the collective behaviour of the states will be analysed. For the remainder of this chapter it will be considered that $N = 100$ and $\epsilon = 0.2$. The ϵ is not selected for analysis because during preliminary tests, altering its value had no noticeable impact on the development of the reservoir states, but in future research the sparsity would be a hyper-parameter of interest.

The figures that follow will contain one or more of the following four plots: the average of the states of the hidden layer, the variance of the states, the Fourier spectrum of the average of the states, and the power spectrum of the average of the states plotted logarithmically. Plotting the evolution of the states like this makes it easier to observe collective behaviour e.g. asymptotic, constant, oscillatory (periodic or irregular - possibly chaotic).

Recall that to alter ρ an entirely new reservoir must be initialised. For each ρ a reservoir is initialised and rescaled, giving five different reservoirs. The α is then varied over each reservoir. These reservoirs are connected by different weights $\mathbf{W}$, nevertheless trends are still observed in changing ρ. Each time a reservoir is tested, it is allowed to run for $T = 1000$ unless otherwise indicated. Also, it does not appear in any case that there are periods of transient behaviour that need to be removed.

Reservoir 1 is created with $\rho = 0.5$. It is observed that for all values of α the states of the network always converge to zero. An additional finding is that as the α increases, the states converge to zero in fewer time-steps i.e. quicker; this makes sense when one looks at the first term of equation (12). As the states do not oscillate in any way, the Fourier and power spectra do not show anything of interest for any value of α. An example of the trajectories of the states of this reservoir can be seen in Figure 13, clearly the average, and variance of the states tend to zero. The Fourier and power spectra in the case of these states do not give insight of any kind, these spectra are better suited to analysing stationary waves. For this reservoir all values of leaking rate give similar results to those shown in Figure 13.

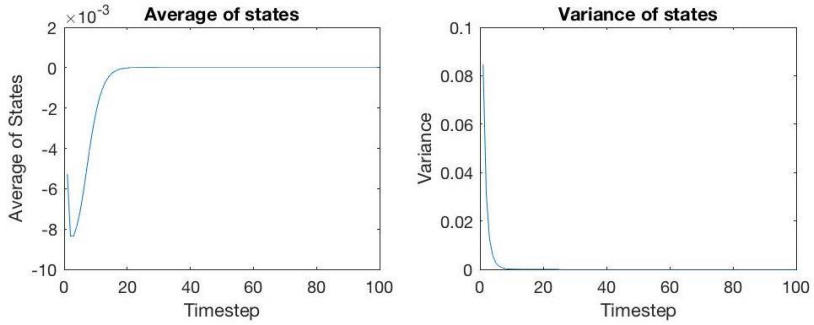


Figure 13: Trajectories of states of reservoir 1. $\rho = 0.5$ and $\alpha = 0.5$, run for $T = 1000$. Only first 100 time-steps plotted to more clearly show the behaviour of the reservoir early on.

Reservoir 2 is initialised with $\rho = 0.95$. At first this reservoir does not offer results that differ greatly from those given by reservoir 1, this is until the α is increased to $\alpha > 1$. With $\alpha = 1.1$ the states do not converge to zero but instead oscillate back and forth around zero. The power spectrum shows clearly that the collective states are exhibiting a periodic orbit of period-2, the power spectrum indicates that these oscillations have a frequency $f = 0.4Hz$. It makes sense to say that this oscillatory behaviour is linked to the α being greater than one. This behaviour most likely does not occur in the previous reservoir because $\rho = 0.5$ is too low, the reservoir will contract too quickly, far before the leaking rate can cause any kind of expansion. This example of oscillating states can be seen in Figure 14.

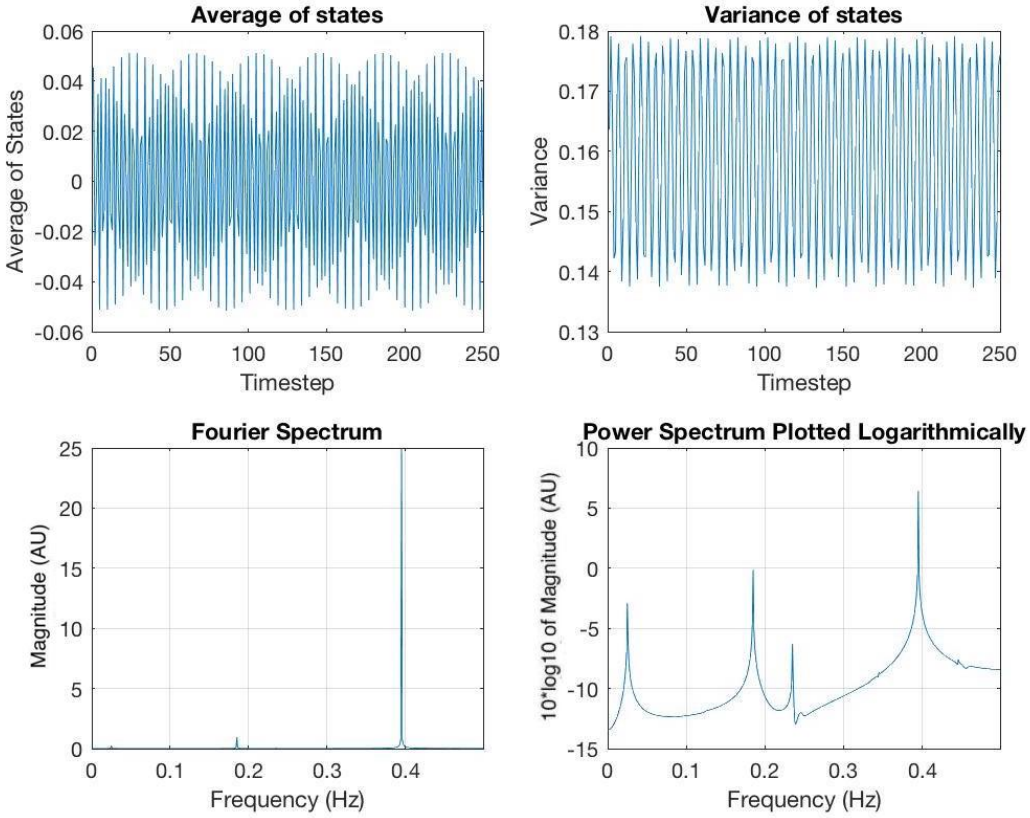


Figure 14: Trajectories of states of reservoir 2. $\rho = 0.95$, $\alpha = 1.1$, run for $T = 1000$. States are seen to be oscillating. First 250 time-steps are plotted in the average and variance of the states, this is to more closely see the repeating orbits of the states.

Reservoir 3 is created with $\rho = 1.1$. With this reservoir at no value of α do the states collectively converge to zero, even when α is as low as $\alpha = 0.05$, the states do not completely reach zero over $T = 1000$. The trajectories with leaking rate $\alpha = 0.05$ are shown in Figure 15. The power spectrum in Figure 15 leads one to believe that the orbit of the states will eventually repeat given more time to develop. This is shown by a peak on the power spectrum at $f = 0.01Hz$. This evaluation seems to make more sense as α is increased over this reservoir. As this happens, the oscillations of the states become more apparent and their frequencies increase.

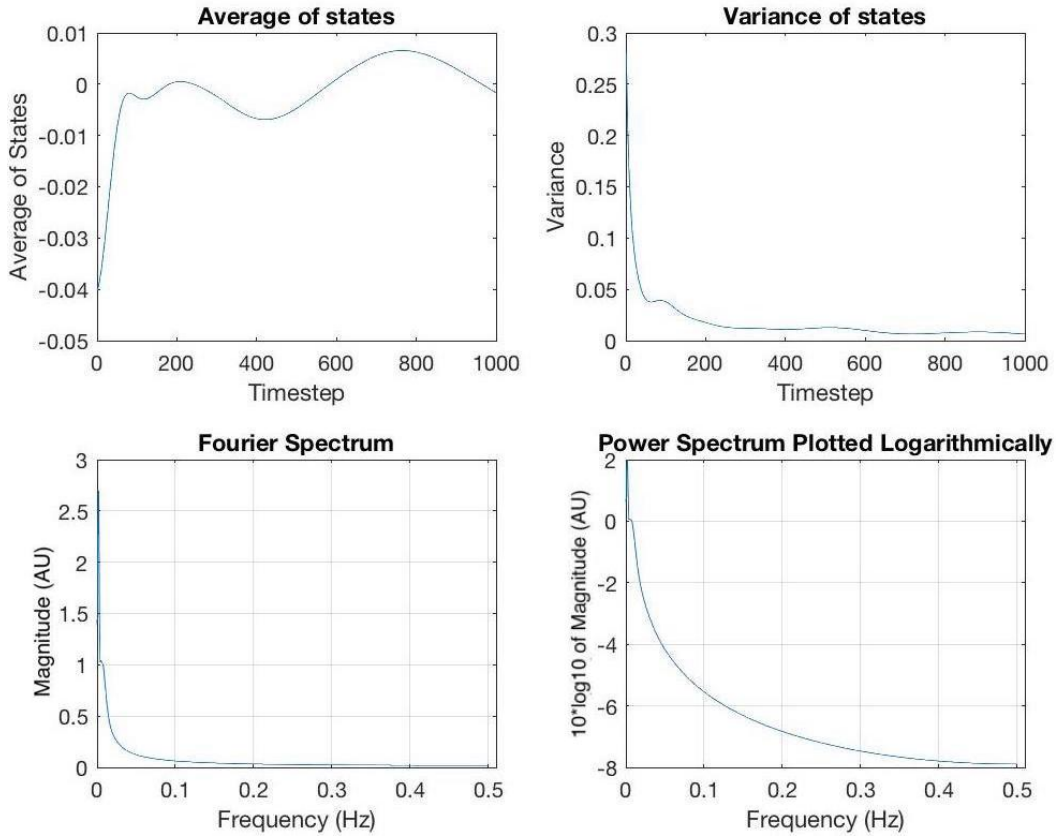


Figure 15: Trajectories of the states of reservoir 3. $\rho = 1.1$, $\alpha = 0.05$, $T = 1000$. States never quite converge to zero over the $T = 1000$. Power spectrum gives $f = 0.01Hz$, suggesting the orbit would in fact repeat given enough time.

An example of this increase in frequency can be observed in Figure 16 with $\alpha = 0.95$. Figure 17 shows a closer view of the average of the states in the same instance. For all values of α the states of this reservoir exhibit quasi-periodic orbits of period-2. The results suggest that with $\rho = 1.1$ the states are expanding at a rate just about equal to the rate at which the leaking rate decreases them. This explains the oscillatory behaviour, and why the states do not converge to zero but also do not behave irregularly.

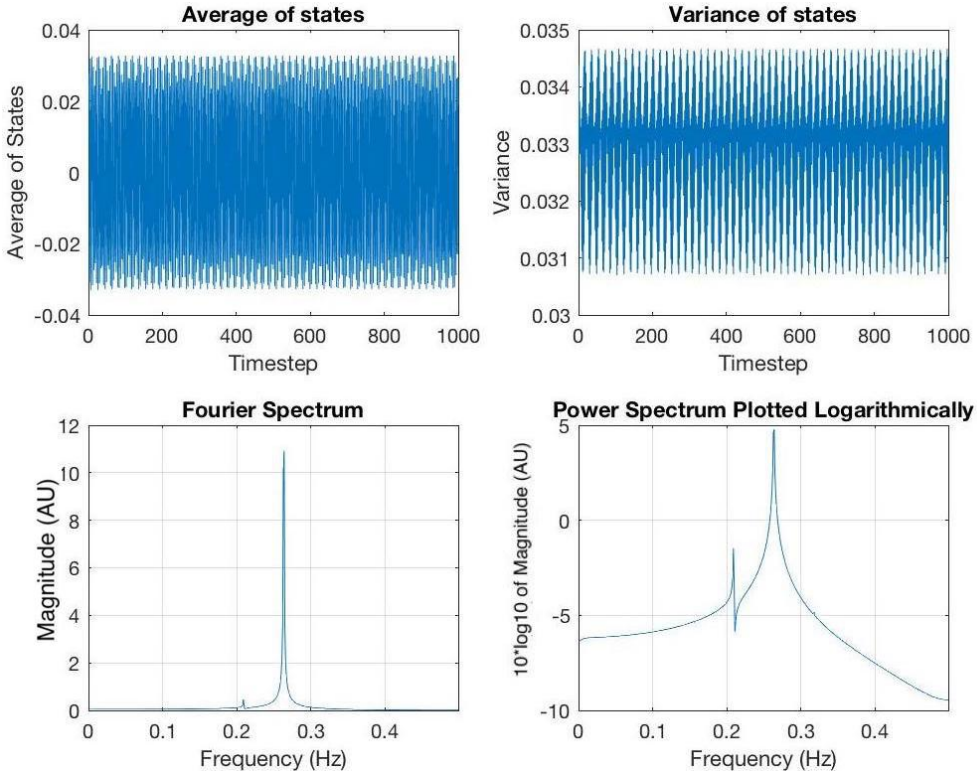


Figure 16: Trajectories of the states of reservoir 3. $\rho = 1.1$, $\alpha = 0.95$, $T = 1000$. States oscillate with increasing frequency as the leaking rate increases. Power spectrum indicates $f = 0.26Hz$. Close up of average of the states here can be seen in Figure 17.

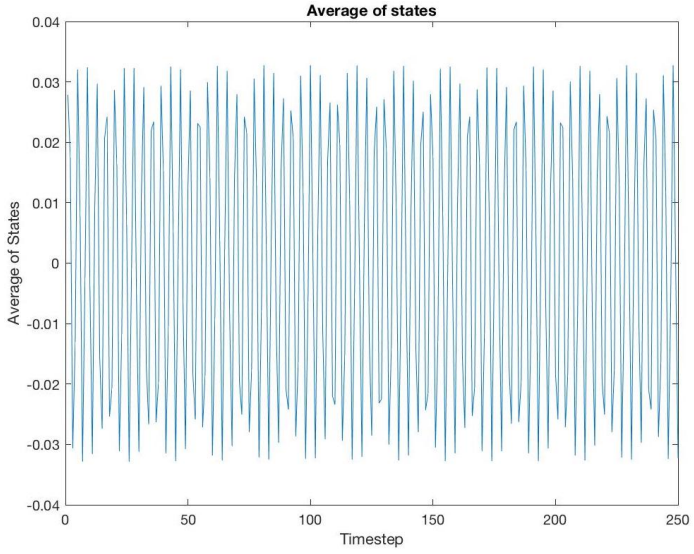


Figure 17: A close up view of the average of the states from Figure 16, here the repeating pattern of the oscillations can be seen.

Reservoir 4 is initialised with $\rho = 1.5$. At first the results are similar to those given by reservoir 3; the states of the reservoir oscillate in an orbit of period-2, and as α increases the frequency of the oscillations increases, but this time there is an exception. When α is increased to $\alpha = 1.1$, the states exhibit irregular (possibly chaotic) behaviour. No hint of periodicity can be observed, even when the states are allowed to run for $T = 2000$. The states with $\alpha = 1.1$ are shown in Figure 18.

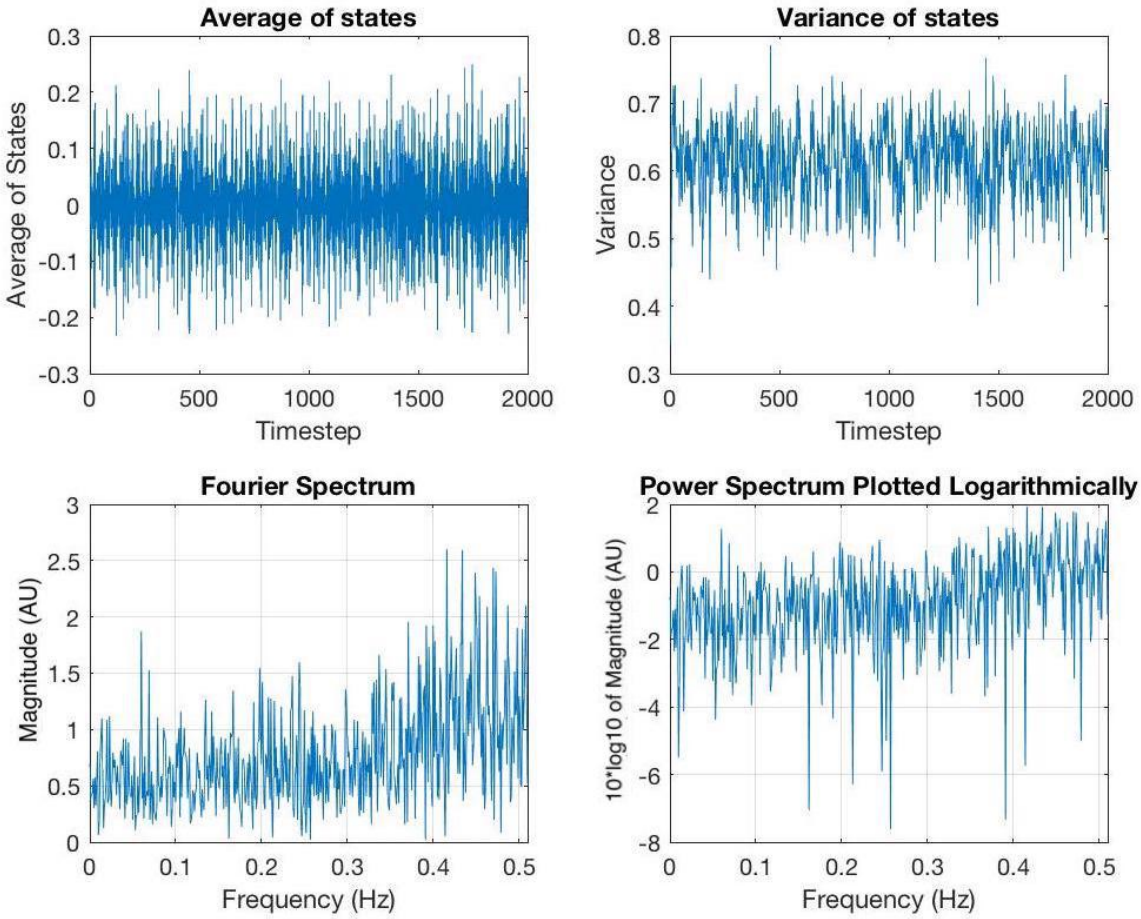


Figure 18: Trajectories of the states of reservoir 4. $\rho = 1.5$, $\alpha = 1.1$, $T = 2000$. The states are seen to be oscillating with no observable periodicity, the behaviour could be described as irregular.

Reservoir 5 is initialised with $\rho = 3.0$. With ρ being so high the states do not exhibit any orderly behaviour, no matter which value of α is used. This most likely occurs because $\rho = 3.0$ is very high, causing the reservoir to expand. No results given by this reservoir differ greatly from those given by reservoir 4, shown in Figure 18. They all exhibit irregular behaviour, this is most likely due to $\rho > 1$ violating the *echo state property* [55]. An explanation of the echo state property is given in the appendix. The α controls by how much the states adjust over each step. One can see that the first component of equation (12) depends entirely on $1 - \alpha$ and $\vec{s}_n$. This indicates that if $\alpha > 1$ then the states are encouraged to jump between positive and negative values over time, which likely contributes to irregular dynamics.

In short it is possible to conclude that while $\rho \in (0, 1)$ the network is stable such that the states converge. Increasing ρ beyond this interval causes the stationary states to become unstable due to the fact that perturbation now tends to grow, and this instability is more pronounced, larger, and cannot be offset by any value of the α. A similar conclusion can be drawn for α, as it increases within the interval $(0, 1)$ the number of time-steps taken for the states to converge decreases. Increasing α beyond this interval means that each state does not converge to its required value but instead oscillates around it.

The results of this section have also revealed a selection of values that can be used to train the networks in the following chapters. In chapter 5 while modelling the chaotic Lorenz system, the states of the reservoir remain stable during training and prediction because $\rho = 0.9$ and $\alpha = 0.5$ are used.

These values are also initially used in chapter 6 and are shown to work reasonably well in creating a network to decompose linearly combined chaotic signals. Although in this chapter with the guidance of published work [5], different values of α and ρ better suited to the task are found, but they still fall within the intervals $\rho \in (0, 1)$ and $\alpha \in (0, 1)$.

Detailed evaluation, discussions and conclusions for the results in this section will be given in chapter 7.

5 Modelling the Lorenz System

This chapter is dedicated to training a network that is able to model the dynamics of the Lorenz system during its chaotic regime. Chaos by its very nature is unpredictable, as is shown in the following sections. Despite the difficulty found in predicting the dynamics of the Lorenz system over long times, it is however possible to design a model that replicates its chaotic behaviour over short times. The approach taken in training the model is described in the following section. Results will be given after a discussion of the methods and will be given chronologically.

In the previous chapter, changes were made to the equation that governs $\vec{s}_n$; this equation is reverted back to equation (6). Unlike in the previous chapter, the input and output layer are required, so the structure discussed in chapter 3 is also reinstated.

5.1 Methods

The training data will consist of measurements of the Lorenz system with initial conditions $\{x, y, z\} = \{1, 1, 1\}$ and parameters $\rho = 28, \beta = 8/3, \sigma = 10$, these parameters were used originally by Lorenz himself [50]. Trajectory points, here representing measurements that I aim to model, are generated by using the MATLAB ode45 integrator. The ode45 function solves the differential equations of the system and gives a number of measurements depending on the integration step. The function uses a fourth order Runge-Kutta method. It is important to recall that an attempt is being made to model a chaotic system so accuracy is key, using a Runge-Kutta algorithm far exceeds the accuracy of a typical Euler algorithm, for an equivalent step size [56]. The Lorenz equations were shown earlier in section 2.5.

The initial goal is as follows. Design a network that accepts measurements from the Lorenz system at t (where t is the integration step for the trajectories of the Lorenz system) as $\vec{u}_n$ of the form $\{x(t), y(t), z(t)\}$ and then predicts as $\vec{y}_n$ the proceeding measurements at $t + 1$ of the form $\{x(t + 1), y(t + 1), z(t + 1)\}$. All while there is a feedback loop where $\vec{y}_n$ is fed back in as $\vec{u}_{n+1}$, meaning the network will autonomously reproduce the future of the Lorenz for longer times than 1 step n. Measurements of the Lorenz system are separated into two subsets: one for use as training data D_T and the other to validate the predictions of the network D_P. At no point will any ESN be trained using measurements from D_P.

An RC is trained as follows. Training data consists of three input time-series $\{x(t), y(t), z(t)\} \in D_T$ and three desired output time-series $\{x(t + 1), y(t + 1), z(t + 1)\} \in D_T$. Over each n the input vector is assigned as $\vec{u}_n = \{x(t), y(t), z(t)\}$ and the output vector is assigned as $\vec{y}_n = \{x(t+1), y(t+1), z(t+1)\}$. This occurs for total training length T, where T corresponds to the total n for which training data is made available from the subset D_T. This begins only after the network has run for a transient time period T_0.

During training, the weights $\mathbf{W_{out}}$ are adjusted via equation (9). Once finished, the network can be used to make predictions.

At the beginning of prediction the network is fed a single set of $\{x(t), y(t), z(t)\} \in D_P$ as an initial $\vec{u}_n$. Given this $\vec{u}_n$, the network then computes $\vec{y}_n$ with $\mathbf{W_{out}}$ and equation (10). These computed $\vec{y}_n$ will be estimates of $\{x(t), y(t), z(t)\} \in D_P$, these estimates will be denoted as $\{\hat{x}(t), \hat{y}(t), \hat{z}(t)\}$. It is important to recall that during this chapter $\vec{y}_n$ is fed back into the network as $\vec{u}_{n+1}$ during prediction. Therefore, only a single initial $\vec{u}_n$ is manually fed to the network, all subsequent $\vec{u}_n$ are generated by the network. In this way, the network will be autonomously predicting the Lorenz future states for longer times than 1 step n.

5.1.1 RC Parameters and Hyper-parameters

All of the networks used during this chapter are initialised, trained and used for prediction in accordance with the processes described in chapter 3.

While improving the accuracy of the model N and T are adjusted. The majority of network parameters and hyper-parameters will be kept constant during this section and assigned the values,

- $b_{in} = b_{out} = 1$
- $\alpha = 0.5$
- $\rho = 0.9$
- $\epsilon = 0.2$
- $\lambda = 5 * 10^{-}5$
- $f_{in} = tanh(\cdot)$.

All networks before training occurs are allowed to run $T_0 = 5000$.

5.2 Results

The Lorenz system is integrated with step size 0.01, and the resulting measurements are split into two subsets, D_T for training and D_P to validate predictions. For each RC presented the T used to train it will be specified. T is varied by selecting different subsets of D_T, i.e. taking a subset of $1 * 10^4$ measurements from D_T will mean the network is trained for $T = 1 * 10^4$. Varying T like this is to reduce code run-time as necessary. Given some subset of D_T, a network will be trained to predict the trajectories of the measurements D_P. In this work we will only track the estimate $\hat{x}(t)$ of a single component $x(t) \in D_P$. To observe the accuracy of the predictions, a comparison will be drawn between $x(t)$ and $\hat{x}(t)$.

More specifically, accuracy will be gauged by calculating their absolute difference. The signal $x(t) \in D_P$ is displayed in Figure 19, this is shown before all other plots to give an indication of the desired signal.

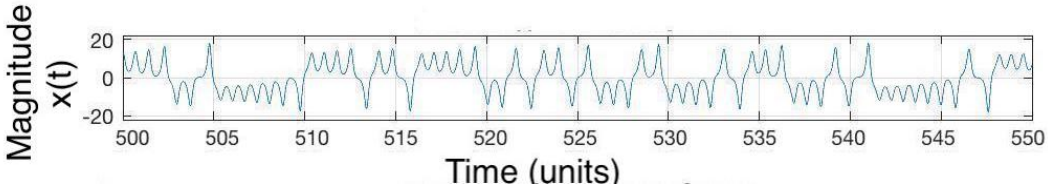


Figure 19: The figure illustrates $x(t) \in D_P$, that will be modelled during this chapter. The signal represents x component trajectories of the Lorenz system, from 500 to 550 time units. Signal generated from Lorenz with parameters $\rho = 28, \beta = 8/3, \sigma = 10$, initial conditions $\{x, y, z\} = \{1, 1, 1\}$.

A network of $N = 200$ is initialised and trained using a subset of D_T corresponding to training length $T = 1 * 10^4$. The model is then used to make predictions via the processes described earlier, recall that the predicted output is re-fed to the network as the input at the following step. At first the trajectory predicted by the RC is identical to that of the data. Unfortunately, after about 1 time unit the estimations diverge from the data. Evidently this is the challenge of trying to model a chaotic system over long times. Chaotic dynamics by their very nature are irregular, early discrepancies (even minuscule) between the model and the data will lead to hugely diverging outcomes over long times. This is the consequence of the sensibility to initial conditions caused by chaos.

This initial result dashes hopes of modelling the system over long times. At this point, a decision is made to investigate whether or not the accuracy of the model can be improved. N and T will be adjusted to improve the model. More specifically we will verify if these changes can increase the number of time units for which the model accurately matches the data, i.e. the time elapsed before the model significantly diverges from the data.

For consistency there needs to be an indicator of the point at which the predictions are considered to have significantly diverged from the data. The error E will be defined as $E = |x(t) - \hat{x}(t)|$, the absolute difference between the data and predictions. The model will be considered accurate for as long as this error E remains less than 10% of the absolute maximum amplitude of $x(t)$. $Max(|x(t)|) \simeq 19.5$. Therefore the model is considered accurate as long as $E \leq 1.95$. This is a viable method of validation for the accuracy of predictions, because it is a consistent gauge of how similar our estimations of trajectories are to the desired trajectories.

This could be interpreted as a relatively lenient window of error, but in the context of our goal it is acceptable as it offers consistency to the investigation. While T is investigated N remains constant and vice versa. The values examined are shown in table 4.

Table 4: Examined values of N and T

Parameter	Parameter Values
T	$1 * 10^4, 2 * 10^4, 3 * 10^4, 4 * 10^4$
N	$200, 400, 600, 800$

The results of this short investigation are shown in table 5 and table 6, where each result is averaged over three random initialisations of the network. One can clearly see from these tables that increasing N and T both independently improve the accuracy of the predictions produced by the model i.e. the model makes more accurate predictions with either a longer training time or larger network.

Table 5: Effect of increasing training data length T with constant reservoir size $N = 200$

Length of training data, T	Time units before divergence
$1 * 10^4$	0.94
$2 * 10^4$	1.37
$3 * 10^4$	1.45
$4 * 10^4$	1.94

Table 6: Effect of increasing reservoir size N with constant training data length $T = 1 * 10^4$

Reservoir size, N	Time units before divergence
200	1.35
400	2.39
600	3.10
800	4.31

A picture of one set of the results from table 6 is displayed in Figure 20, referred to as 'network 1' for the remainder of this chapter. This example was generated using $N = 200$, $T = 1 * 10^4$. The first panel shows $x(t)$, the second panel shows $\hat{x}(t)$, the third panel displays both signals together. The fourth panel shows $x(t) - \hat{x}(t)$, the fifth panel also shows this difference but only the first few time units, this is to emphasise the early time at which the predictions diverge significantly from the data.

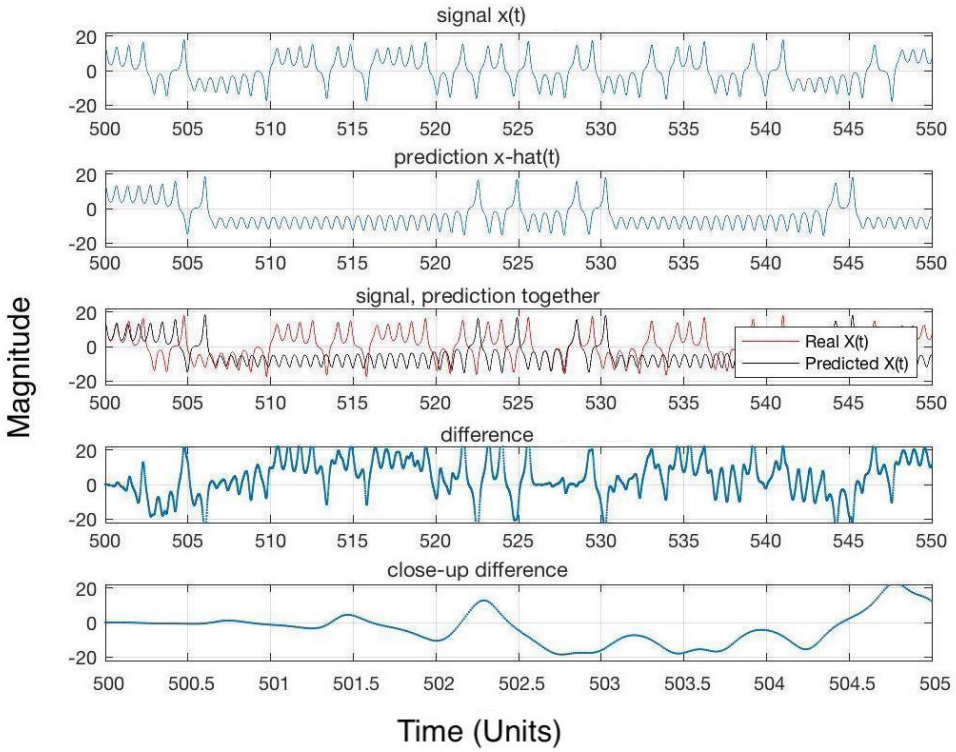


Figure 20: Signal $x(t) \in D_P$ with corresponding prediction $\hat{x}(t)$, predicted using network 1, ($N = 200$, $T = 1 * 10^4$). From bottom to top: Panel 1 $x(t)$, Panel 2 $\hat{x}(t)$, Panel 3 $x(t)$ and $\hat{x}(t)$ plotted together, Panel 4 $x(t) - \hat{x}(t)$, Panel 5 close-up view of $x(t) - \hat{x}(t)$.

Results from network 1 shown in Figure 20 exhibit two noticeable features. Firstly, E diverges from zero after about 1.35 time units have elapsed, this is most clearly seen in the fifth panel. The other feature of interest is the long stretches of periodicity being exhibited by $\hat{x}$, displayed in the second panel. These relatively long periods of periodicity are not in line with the expected behaviour of the x component of the chaotic Lorenz system. If allowed to continue this model will never reflect $x(t)$ even over long times.

The quality of the predictions made by network 1 need to be verified further. The Poincaré map of the time-series $\{x(t), y(t), z(t)\} \in D_P$ and the Poincaré map of the corresponding predictions $\{\hat{x}(t), \hat{y}(t), \hat{z}(t)\}$ will be analysed. This will lend information on whether the shape of the trajectories predicted by the model match those of the data.

A Poincaré map is created by choosing a Poincaré section (a lower dimensional subspace) in the state space of a continuous dynamical system (i.e. the Lorenz system) and plotting intersections with the section. In this case the Poincaré section is chosen to be the points at which $\dot{z}(t) = 0$, this gives the maxima and minima of $z(t)$. The maps created in this work consider only the maxima. This gives a set of values $(z_m, z_{m+1}, z_{m+2}, \ldots,)$ where m is an integer that labels the order of the crossings. Plotting z_{m+1} as a function of z_m gives a smooth function showing that each subsequent intersection with the Poincaré section depends only on the previous. This type of Poincaré map is usually shown as an example of how three dimensional dynamics like those of the Lorenz system can be expressed as a one dimensional map. The Poincaré map of the data is plotted alongside the Poincaré map of the corresponding predictions. These maps are shown in Figure 21.

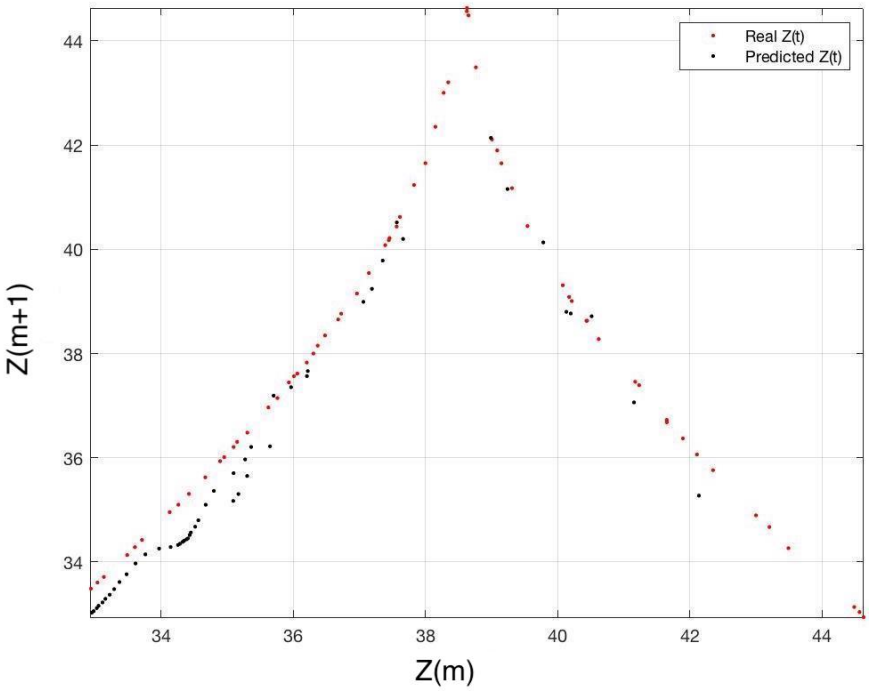


Figure 21: Poincaré map of the maxima of $z(t) \in D_P$ in red. The map of the corresponding predictions $\hat{z}(t)$ in black. Corresponding $x(t)$ and $\hat{x}(t)$ shown in Figure 20. Predictions produced by network 1, $(N = 200, T = 1 * 10^4)$.

In Figure 21 the red points correspond to the data and the black points correspond to the predictions. The data points in black all cluster toward the lower corner of the map on the left hand side, the remainder of the points are sporadically spread across the path of the red points. The comparison of the maps in Figure 21 shows that there is a discrepancy between the shape of the predicted trajectories and the desired trajectories.

Another set of predictions are displayed in Figure 22, produced with a network $N = 600$, $T = 1{*}10^4$, referred to as 'network 2' for the remainder of this chapter. This figure shows that the model accurately predicts the signal for as long as four full time units, an improvement from the results of network 1. Unfortunately, the second panel shows that at about $t = 518$ the predicted trajectories oscillate wildly back and forth with a high frequency.

Although network 2 makes accurate predictions for a longer time than network 1, it still does not generate trajectories of the correct shape over long times. As $\rho < 1$ and there is no more external perturbation from an external system it is not surprising that the autonomous feed-back network becomes periodic. The result of predicting 4 full time units is actually quite remarkable, this is equivalent to 5 cycles. If the Lyapunov exponent for the Lorenz system is about 0.9, and $0.9 \times 4 = 3.6$, then from equation (5), a perturbation of size δ grows to $\delta e^{3.6} = \delta 36.6$ in that time. So, if the initial E is of the size 0.5, then in 4 time units the error should grow to $E = 18$, which is approximately the difference we have used to define the cut-off of accuracy. So, it can be said that the model for the Lorenz system is accurate by a margin of 0.5, which is roughly 1% of the size of the attractor (40).

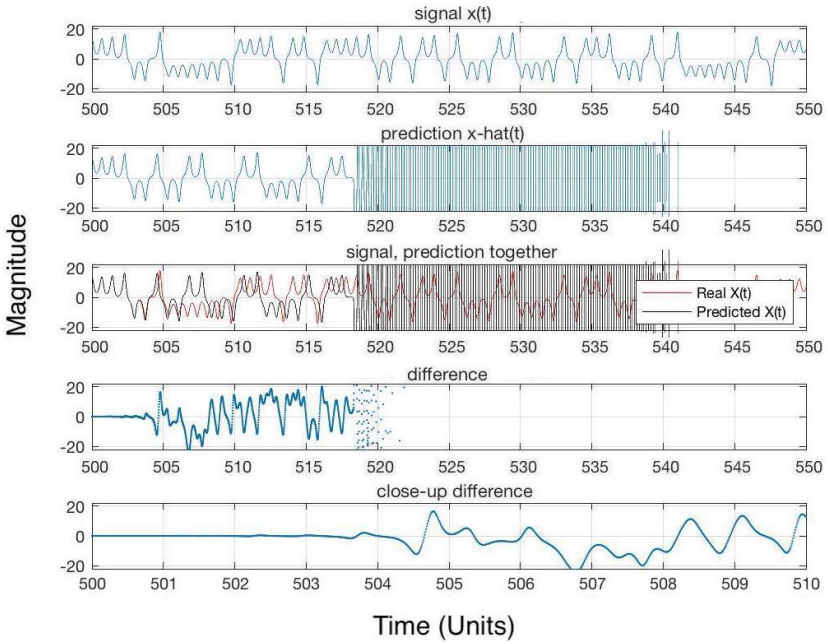


Figure 22: Signal $x(t) \in D_P$ with corresponding prediction $\hat{x}(t)$, predicted using network 2, ($N = 600$, $T = 1 * 10^4$). From bottom to top: Panel 1 $x(t)$, Panel 2 $\hat{x}(t)$, Panel 3 $x(t)$ and $\hat{x}(t)$ plotted together, Panel 4 $x(t) - \hat{x}(t)$, Panel 5 close-up view of $x(t) - \hat{x}(t)$.

Unfortunately, in Figure 22, one can see from the second panel that $\hat{x}(t)$ begins to oscillate out of the scale of the axis at about $t = 518$ time units. This means that the points in the fourth panel showing the difference also begin to veer too high or low to fit nicely within the axis limits (so imposed to make panels with the same vertical ranges).

The quality of the predictions of network 2 will be further examined, but not via the Poincaré maps. To further analyse network 2, the Fourier and power spectra of $\hat{x}(t)$ will be compared with the spectra of $x(t)$. The Fourier and power spectra will reveal how accurate the model is i.e. whether the shape of the predicted trajectories are similar to those of the data. A comparison of these Fourier and power spectra is displayed in Figure 23.

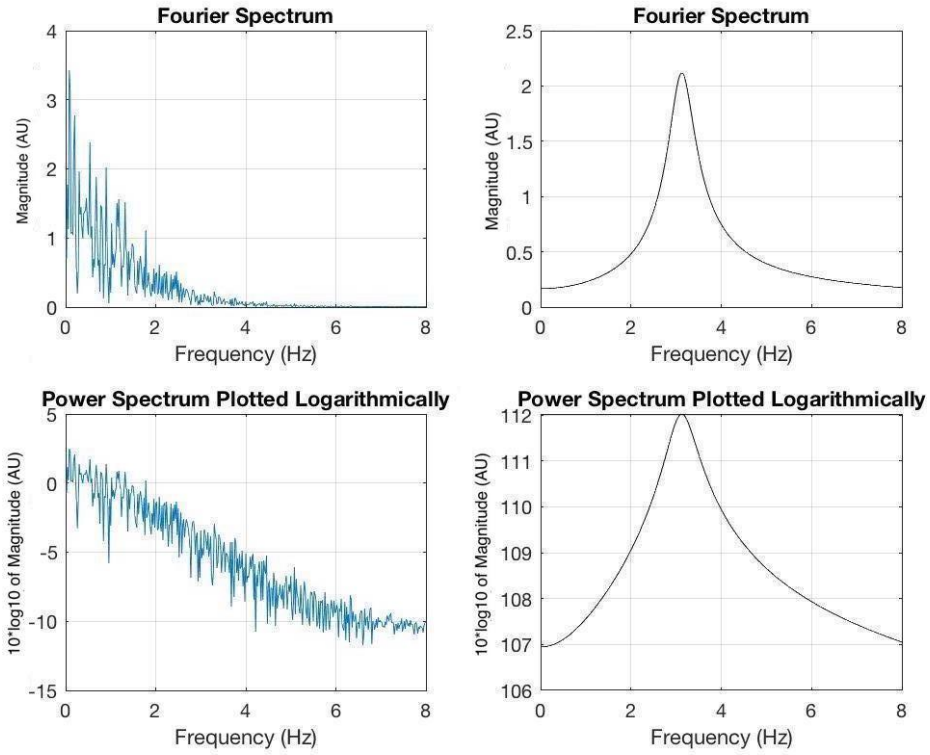


Figure 23: The Fourier and power spectra of $x(t) \in D_P$ and corresponding $\hat{x}(t)$. Plotted to highlight the differences between the spectra of the data and the prediction. Prediction made using network 2, $(N = 600, T = 1 * 10^4)$. Corresponding time-series plots shown in Figure 22

.

In Figure 23, the two plots on the left hand side in blue represent the Fourier and power spectra of $x(t)$, recall that $x(t)$ is shown alone in Figure 19. The two plots on the right hand side in black represent the Fourier and power spectra of the $\hat{x}(t)$ produced by network 2. The $x(t)$ spectra show no discernible frequencies, no obvious frequency at which repetitions will occur, the signal is of course chaotic and this is reflected in the spectra plots. On the other hand, the spectra of $\hat{x}(t)$ show a clear peak at about $f = 3Hz$. This clear peak is most likely indicative of the rapid periodic oscillations shown in Figure 22. This comparison indicates clearly that network 2, like network 1; has failed to model the shape of the trajectory of $x(t)$ for long times.

From network 1 to network 2, the length of predictive accuracy increases from about one time unit to four time units, this is a relatively good improvement. Sadly, neither of these models have been able to replicate the shape of the trajectory of $x(t)$ for a long time.

However, more promising results are given when the reservoir size is increased to $N = 800$. In Figure 24 another set of predictions are displayed, these are produced using $N = 800$, $T = 1 * 10^4$. This network will be referred to for the remainder of this chapter as 'network 3'. In Figure 24, the fifth panel shows that the model is accurate for about 4.5 time units. This is a huge improvement; made even more evident in the second panel. In this panel $\hat{x}(t)$ appears to have the same shape as the trajectory $x(t)$ in the top panel. Figure 24 seems to indicate that although $\hat{x}(t)$ diverges from $x(t)$, the general shape of the predicted trajectories are consistent with those of the data of the Lorenz system over long times. This assumption is investigated using the Fourier and power spectra.

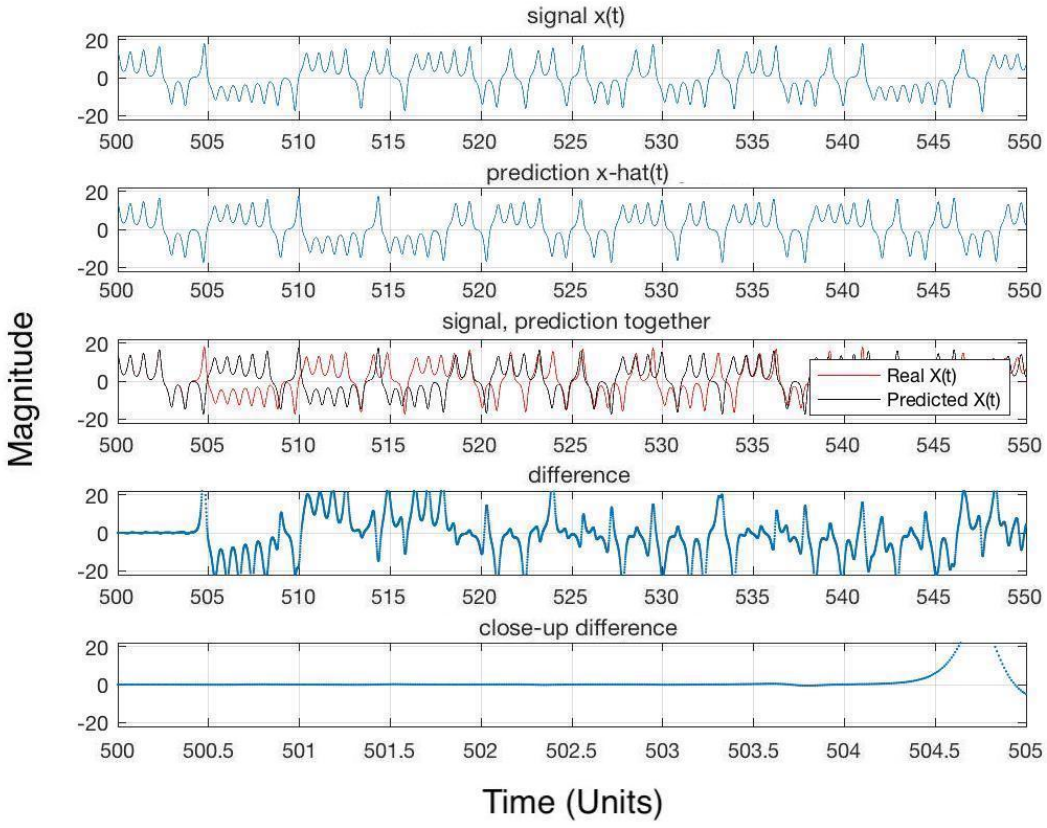


Figure 24: Signal $x(t) \in D_P$ with corresponding $\hat{x}(t)$, predicted using network 3, ($N = 800$, $T = 1 * 10^4$. From bottom to top: Panel 1 $x(t)$, Panel 2 $\hat{x}(t)$, Panel 3 $x(t)$ and $\hat{x}(t)$ plotted together, Panel 4 $x(t) - \hat{x}(t)$, Panel 5 close-up view of $x(t) - \hat{x}(t)$.

In Figure 25, the plots on the left hand side in blue represent the Fourier and power spectra of $x(t)$. The plots on the right hand side in black represent the spectra of $\hat{x}(t)$. The spectra on the right hand side appear very similar to those shown on the left hand side, apart from minor anomalies. When the spectra in Figure 25 are compared with the spectra shown previously in Figure 23, the improvement from network 2 to network 3 is clear. There is great similarity between the spectra of $x(t)$ and the spectra of the $\hat{x}(t)$ predicted by network 3. This shows that unlike network 1 or 2, network 3 has successfully modelled the shape of the trajectories exhibited by the x component of the Lorenz system within the interval D_P.

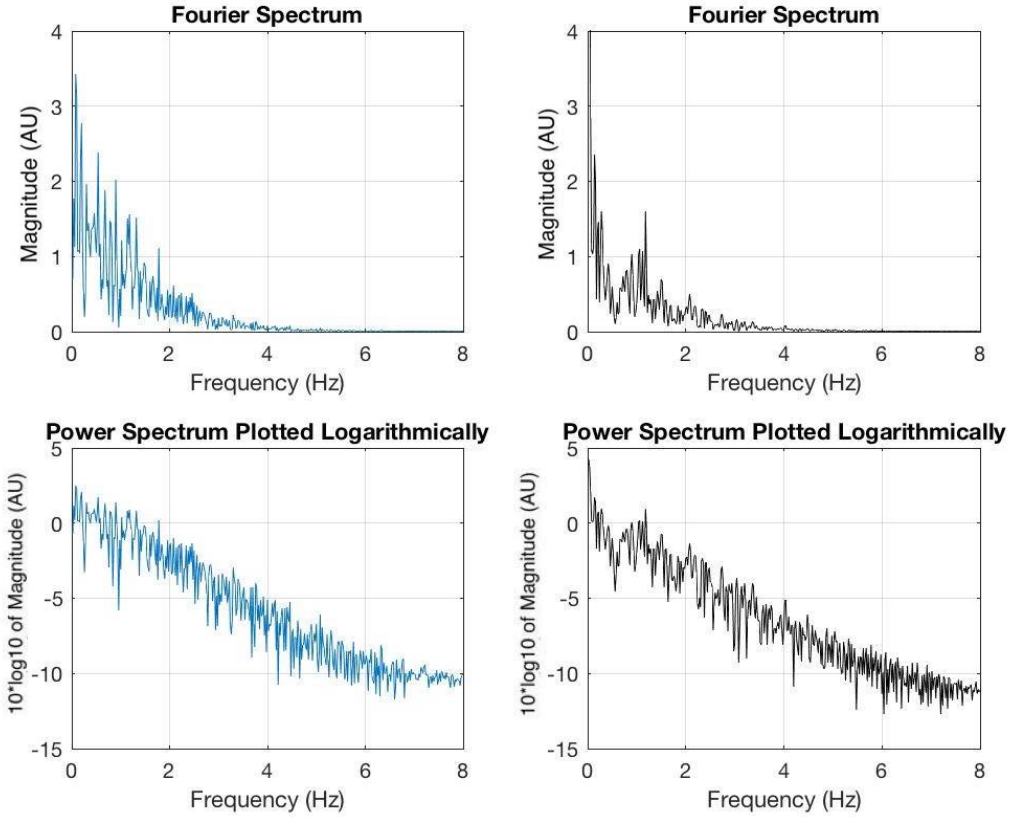


Figure 25: The Fourier and power spectra of $x(t) \in D_P$ and corresponding $\hat{x}(t)$. Plotted to highlight the similarities between the spectra of the data and the prediction. Prediction made using network 3, ($N = 800$, $T = 1 * 10^4$). Corresponding time-series plots shown in Figure 24

.

To be sure of this, a final reservoir of $N = 800$, $T = 4 * 10^4$ is initialised and analysed via Poincaré map. For the remainder of the chapter it will be referred to as 'network 4'. Figure 26 shows the predictions made by network 4. The fifth panel clearly shows that the model accurately describes the data for up to six full time units, this is the longest achieved length of predictive accuracy so far. This is even more remarkable than the earlier 4 full time units. To further verify the quality of network 4, the Poincaré map of its predictions will be analysed in the same was as was done previously with network 1. The Poincaré maps of the data, and of the predictions of network 4 are compared in Figure 27.

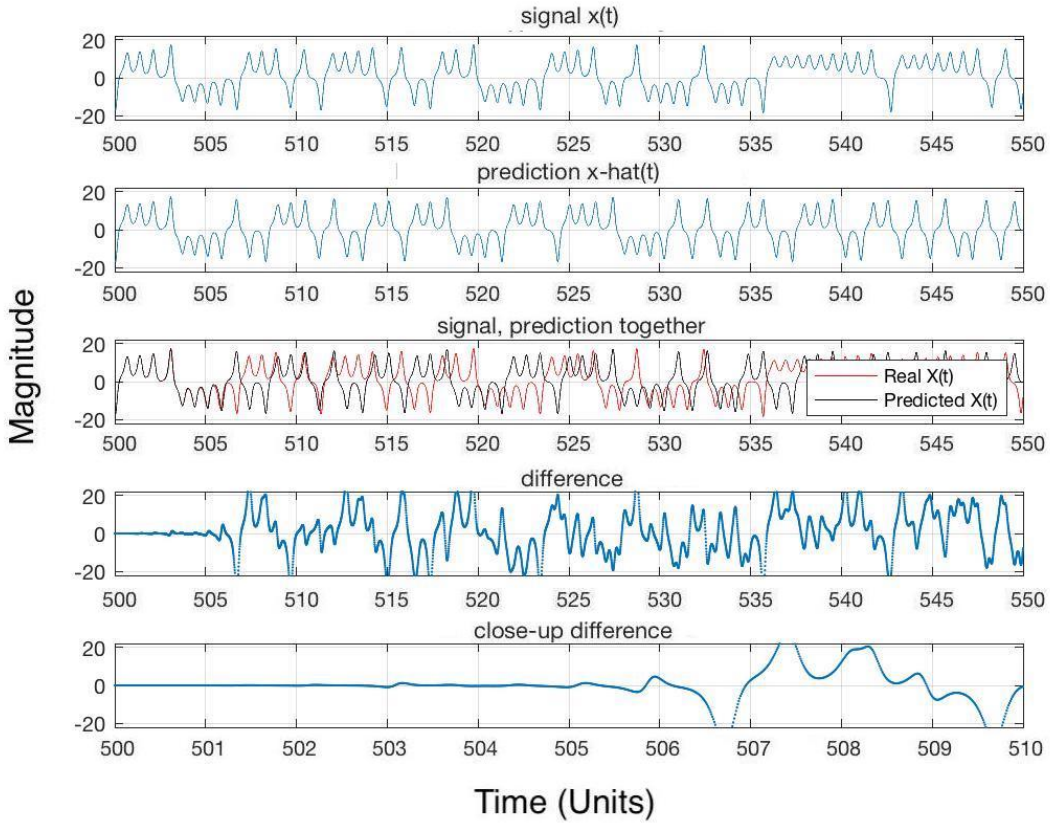


Figure 26: Signal $x(t) \in D_P$ with corresponding $\hat{x}(t)$, predicted using network 4, ($N = 800$, $T = 4 * 10^4$). From bottom to top: Panel 1 $x(t)$, Panel 2 $\hat{x}(t)$, Panel 3 $x(t)$ and $\hat{x}(t)$ plotted together, Panel 4 $x(t) - \hat{x}(t)$, Panel 5 close-up view of $x(t) - \hat{x}(t)$.

From Figure 27 it is clear that the structure of the predicted trajectories are very similar to the expected trajectories of the measurements of the Lorenz system in the interval D_P. In Figure 27 the black points that represent the predictions are spread evenly across the line of red dots that make up the measured data. The predicted map does not stray from the path of the real Lorenz map. This comparison of Poincaré maps further backs up the hypothesis given by Figure 26; that network 4 does in fact properly replicate the structure of $x(t) \in D_P$ of the Lorenz system.

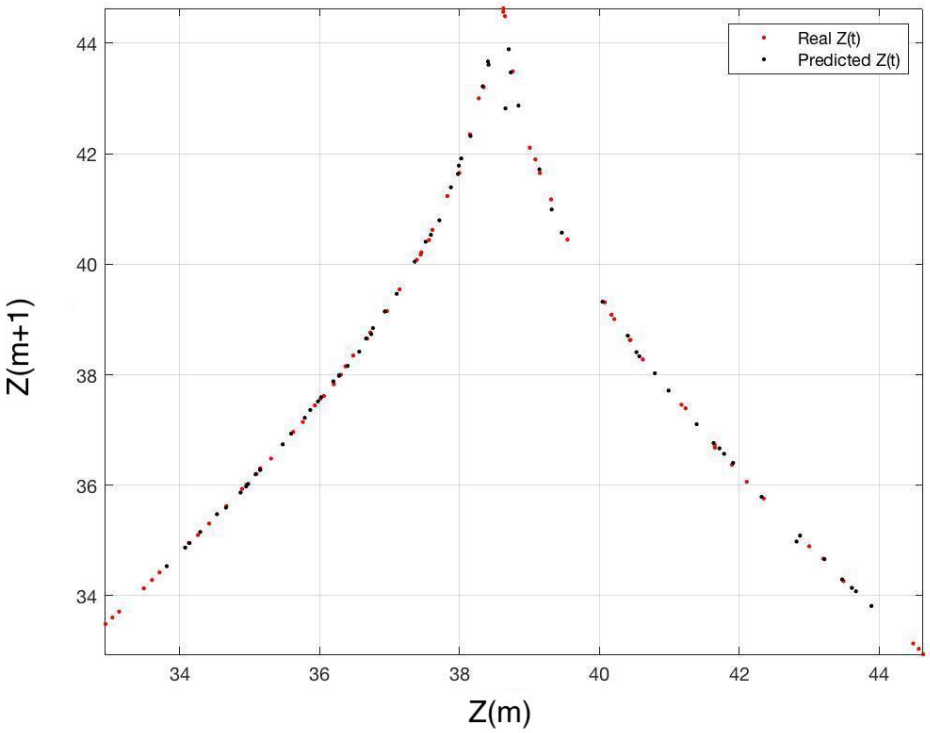


Figure 27: Poincaré map of the maxima of $z(t) \in D_P$ in red. The map of the corresponding predictions $\hat{z}(t)$ in black. Corresponding $x(t)$ and $\hat{x}(t)$ shown in Figure 26. Predictions produced by network 4, $(N = 800, T = 1 * 10^4)$.

The investigations conducted in this chapter have revealed a few things. It is not easy to predict a chaotic system for long times. It is clearly possible to train a model to predict the dynamics of the chaotic Lorenz system for short times. Remarkably in this chapter we have been able to produce a model that correctly predicts the chaotic dynamics of the Lorenz system for a full 6 time units. Further, with an RC of $N = 800$ it is possible to model the general shape of the chaotic trajectories of the Lorenz system. Detailed evaluation, discussion and conclusions on the results of this chapter will be given in chapter 7.

6 Decomposition of Chaotic Signals

This chapter consists of the methods used in decomposing a linear superposition of two independent chaotic signals. The approaches described in this chapter are mainly based on the work by Krishnagopal, Girvan, Ott and Hunt [5] published in February 2020. This section deals with procedures that are fairly state of the art in the area of machine learning and neural networks. For this reason, the aim is not to completely recreate the work of Krishnagopal *et al* [5]. The aim is to understand the techniques they use that lead to successful signal separation, make adjustments to the networks previously used in this project if necessary, and finally decompose some chaotic signals within a reasonable accuracy for the scope of this project.

Any results gained from this section will be given alongside the methodology. As results are given they will be analysed and conclusions will be drawn, but the discussion and summary evaluation of results will be given in chapter 7.

6.1 Methods

The task is to estimate two scalar signals $S_1(t)$ and $S_2(t)$ from their weighted sum $U(t) = B_1 S_1(t) + B_2 S_2(t)$, where t is the integration step. The first signal $S_1(t)$ is made up of measurements from the Lorenz system with the parameters values $\mathbf{p_1} = \{\sigma = 10, \rho = 28, \beta = 8/3\}$, measurements are generated with integration step 0.01. The second signal $S_2(t)$ is generated similarly but parameters $\mathbf{p_2} = 1.2 \times \mathbf{p_1}$ are used instead. Both these sets of parameter values $\mathbf{p_1}$ and $\mathbf{p_2}$ give rise to chaotic behaviour in the Lorenz system. These two sets of parameters were chosen because of their use in previous work on the subject [5]. $S_1 \in [x_1, y_1, z_1]$ and $S_2 \in [x_2, y_2, z_2]$ but in this project $S_1 = x_1$ and $S_2 = x_2$ will be used in all cases.

$S_1(t), S_2(t)$ and $U(t)$ are all individually standardised to have mean of 0 and standard deviation of 1. This process of standardising the signals is used by Krishnagopal *et al* [5], so it is introduced into this section of the project. The benefit of standardising the signals is that they are all transformed to comparable scales, this means that no one signal will bias the results by having a much different average value.

The weighted sum is normalised; assuming $S_1(t)$ and $S_2(t)$ to be uncorrelated this means, $B_1^2 + B_2^2 = 1$. Letting $a = B_1^2$, then

$$U(t) = \sqrt{a} S_1(t) + \sqrt{1-a} S_2(t). \tag{13}$$

Here a is what is referred to as the mixing fraction of the signals, more specifically it is the ratio of the standard deviation of the first component of $U(t)$ to the standard deviation of $U(t)$ [5].

For simplicity in the context of this project the mixing fraction is always assumed to be $a = 0.5$. This value is chosen because it has been used in previous work [5].

It is assumed that limited-time samples of the signals S_i are available, like $0 \leq t \leq T$, and these samples are used to train the RC. Prior knowledge of the equations that generate $S_1(t)$ and $S_2(t)$ are not necessary, but an assumption is made that the systems are stationary enough that the training samples are representative of the components of future instances of $U(t)$.

The goal is to design a network that will accept instances of $U(t)$ as $\vec{u}_n$ and then predict $\vec{y}_n$ as the corresponding instances of the component $S_1(t)$. Once measurements of $S_1(t)$ and $U(t)$ are obtained, they are separated into subsets: one for use as training data D_T, and the other to validate the predictions of the network D_P. At no point will any ESN be trained using measurements from the subset D_P.

An RC is trained as follows. Training data consists of the input $U(t) \in D_T$ and a desired output $S_1(t) \in D_T$. Over each training step n the input vector is assigned as $\vec{u}_n = U(t)$ and the output vector is assigned as $\vec{y}_n = S_1(t)$. This occurs for a total training length T, where again T corresponds to the total n for which training data is made available from the subset D_T. This begins only after the network has been allowed to run for a transient time period T_0. During training, the weights $\mathbf{W_{out}}$ are adjusted via equation (9). Once finished, the network can be used to make predictions.

During prediction, the input is assigned as $\vec{u}_n = U(t)$, but now $U(t) \in D_P$. Given this $\vec{u}_n$, the network then computes $\vec{y}_n$ with $\mathbf{W_{out}}$ and equation (10). These computed $\vec{y}_n$ will be estimates of $S_1(t)$, these estimates will be denoted as $\hat{S}_1(t)$. Contrary to the previous chapter, in this chapter $\vec{y}_n$ is not fed back into the network as $\vec{u}_{n+1}$, this is because the model in this chapter is not making a prediction of future measurements, but decomposing measurements at n into their components at n. As mentioned earlier, the presence of a feed-back loop depends on the task each network is trained for. Therefore the input $\vec{u}_n = U(t) \in D_P$ is applied throughout prediction at every n.

6.1.1 RC Parameters and Hyper-Parameters

All of the networks used during this chapter are initialised, trained and used for prediction in accordance with the processes described in chapter 3.

In the investigation some network parameters and hyper-parameters are changed to improve the quality of the results, but many of the values used in previous sections will be kept for this investigation and left unchanged. A note of the parameters and hyper-parameters kept constant through out this section is given,

- $b_{in} = b_{out} = 1$
- $\rho = 0.9$
- $\epsilon = 0.2$
- $\lambda = 5 * 10^{-}5$
- $f_{in} = tanh(\cdot)$.

The values used for the remaining parameters and hyper-parameters, as well as the reasons for making changes to them will be given alongside the results.

6.2 Results

The signals $S_1(t)$ and $S_2(t)$ are created from measurements of the two Lorenz systems as mentioned previously. Then $U(t)$ is formed according to equation (13) using the mixing fraction, $a = 0.5$. All signals are standardised and normalised as mentioned earlier. These signals are separated into D_T and D_P for training and prediction as mentioned earlier. As previously stated $S_1 = x_1$ and $S_2 = x_2$, therefore $\hat{S}_1(t) = \hat{x}_1(t)$. The quality of each network will gauged by calculating the error between $\hat{S}_1(t)$ and $S_1(t)$, this will be given by the mean square reservoir error E_r:

$$E_r = \frac{\langle (S_1 - \hat{S}_1)^2 \rangle}{min_\zeta \langle (S_1 - \zeta U)^2 \rangle}, \tag{14}$$

where the term ζ scales the input so that the denominator indicates the root mean square error in the absence of any processing by the reservoir. This E_r is also used by Krishnagopal *et al* [5]. This E_r is how we will compare the accuracy of the networks trained throughout this section.

A picture of a time-sample of the uncombined standardised $S_1(t)$ and $S_2(t)$ can be seen in Figure 28. A picture of the standardised combination $U(t)$ can be seen in Figure 29.

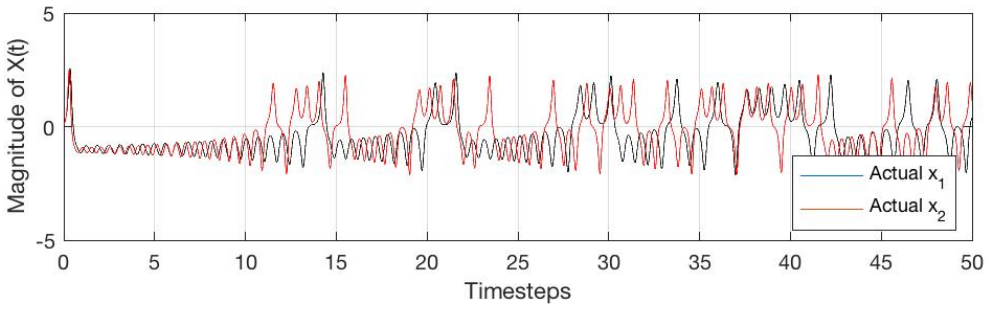


Figure 28: The x components of two Lorenz systems. x_1 having parameters $\mathbf{p_1} = \{\sigma = 10, \rho = 28, \beta = 8/3\}$ and x_2 having parameters $\mathbf{p_2} = 1.2 \times \mathbf{p_1}$. The signals in this figure have been standardised to have mean of 0 and standard deviation of 1. A picture of these signals after being combined can be seen in Figure 29.

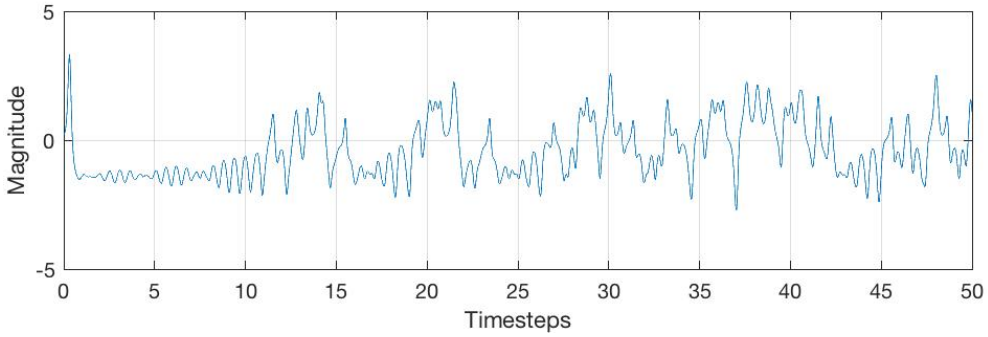


Figure 29: The x components of two Lorenz systems after being linearly combined. x_1 having parameters $\mathbf{p_1} = \{\sigma = 10, \rho = 28, \beta = 8/3\}$ and x_2 having parameters $\mathbf{p_2} = 1.2 \times \mathbf{p_1}$. The signal in this figure has been standardised to have mean of 0 and standard deviation of 1. The individual signals that make up this combined signal can be seen in Figure 28.

Krishnagopal *et al* in their paper [5] offer a useful technique; they use only a subset of their Lorenz measurements. It can be beneficial to use only a sample of these measurements, for example sampling every fifth measurement. Using only a subset of the measurements for training can prevent *overtraining/overfitting* of the model. This occurs when the model becomes too attuned to the training data and loses its applicability to any other data. The impact of changing the step size used in sampling the data will therefore be investigated.

Recall that throughout this project networks have been initialised with input weight matrix $\mathbf{W_{in}}$ having entries $\in [-0.5, 0.5]$. In the work by Krishnagopal *et al* [5], the input matrix is initialised with entries $\in [-k, k]$ where k is considered a hyper-parameter called *input strength normalisation*. The effects of adjusting this hyper-parameter are of interest and will be included in the investigation here.

At first an RC is created and trained, the variables from the end of section 4.2 are used at first. The majority of these variables are given in section 6.1.1, with the remainder being; $N = 800$, $\alpha = 0.5$, $k = 0.5$ and $T = 5 * 10^4$. These values are updated throughout the investigation but explanations will be given for each change in value. There is also no sampling used in collecting Lorenz measurements for training at this point.

The results from this first RC can be seen in Figure 30, the calculated error is, $E_r = 0.29$. During training, this network was allowed to run for $T_0 = 100$. At first $T_0 = 5000$ was used, but upon inspection it was found that reducing T_0 from 5000 to 100 did not ruin the accuracy of the reservoir, but did speed up run-time of the code. At this point, $T_0 = 100$, is set and remains unchanged for the remainder of the investigation.

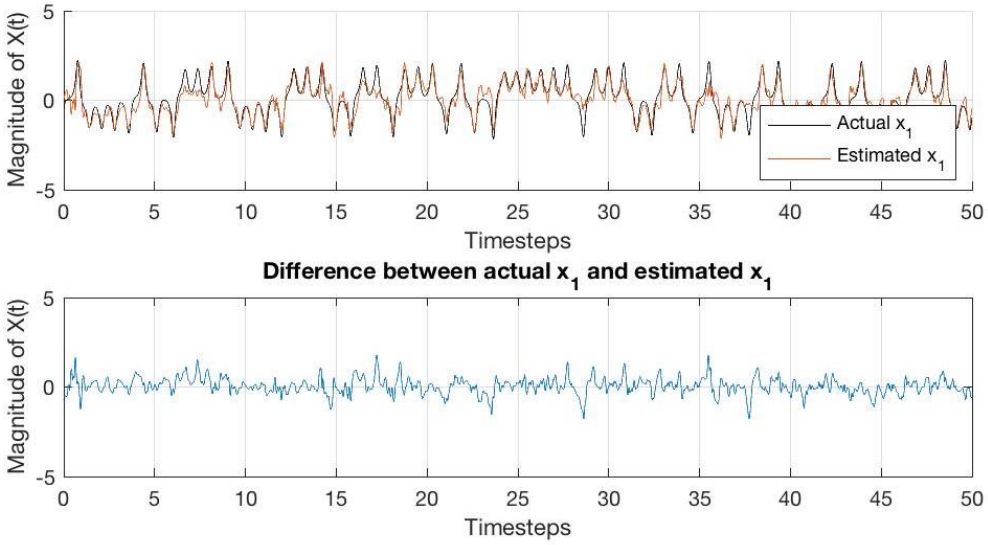


Figure 30: In the top panel; the expected $S_1 = x_1$ plotted with the predicted $\hat{S}_1 = \hat{x}_1$. In the bottom panel; the difference between the expected and estimated signals. Estimated using RC of $N = 800, \alpha = 0.5, k = 0.5, T = 5 * 10^4$. All Lorenz measurements used for training i.e. no sampling. Calculated error $E_r = 0.29$.

Using the variable values from the previous chapter, the RC produces a fairly accurate estimation. At this point the goal shifts to reducing E_r while also keeping the run-time of the code relatively low.

Recall that so far $T = 5 * 10^4$ has been used, with an integration step of 0.01 this will correspond to measurements of 500 time units of the Lorenz system. At first all measurements were used to train the reservoir. To reduce run-time of the code; instead of using every measurement, the measurements are sampled every five steps i.e. 0.05 time units. This means the same subset of D_T is being used for training as before but now using only a fifth of the measurements, therefore $T = 1 * 10^4$.

Initialising and training a new RC with the new sample step of five vastly reduced the run-time of the code, this makes sense as T has been divided by five. Doing this did not cause the error to increase too greatly, but it did increase by about 30% to $E_r = 0.38$, averaged over three random initialisations of the network. The result of this change can be seen in Figure 31. A sample step of 10 is also tested, but this causes the error to increase to $E_r = 0.54$, for this reason going forward a sample step of 5 and corresponding $T = 1 * 10^4$ are used.

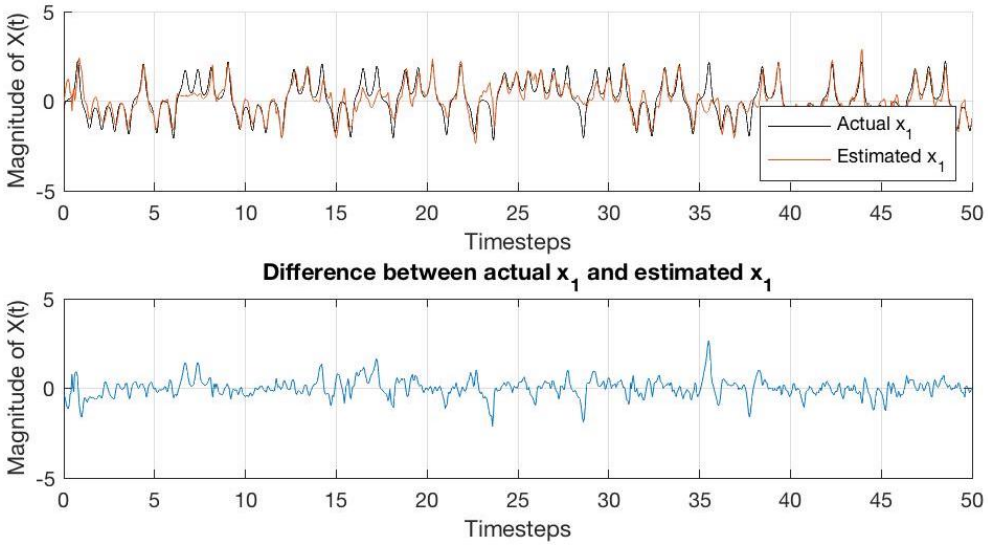


Figure 31: In the top panel; the expected $S_1 = x_1$ plotted with the predicted $\hat{S}_1 = \hat{x}_1$. In the bottom panel; the difference between the expected and estimated signals. Estimated using RC of $N = 800, \alpha = 0.5, k = 0.5, T = 1 * 10^4$, measurements sampled every five steps i.e. 0.05 time units. Calculated error $E_r = 0.38$.

With the run-time of the code now vastly reduced, concern is refocused to reducing E_r. In the work by Krishnagopal *et al* [5] they settle on their optimum of $\alpha = 0.3$, so the next RC uses this value too. It is found that reducing α from $\alpha = 0.5$ to $\alpha = 0.3$ actually increases the reservoir error, it jumps to $E_r = 0.49$ averaged over three random initialisations of the network.

Perhaps this increased error occurs because $\alpha = 0.3$ is too low, causing the reservoir states to adjust by too little within the same time as before. Due to this result, α is then increased instead to test for improved accuracy. The values $\alpha \in [0.7, 0.9, 1.0]$ are all tested. It is found that the optimum value is, $\alpha = 0.9$; using this value achieves $E_r = 0.29$, averaged over three random initialisations of the network. The result from this RC can be seen in Figure 32.

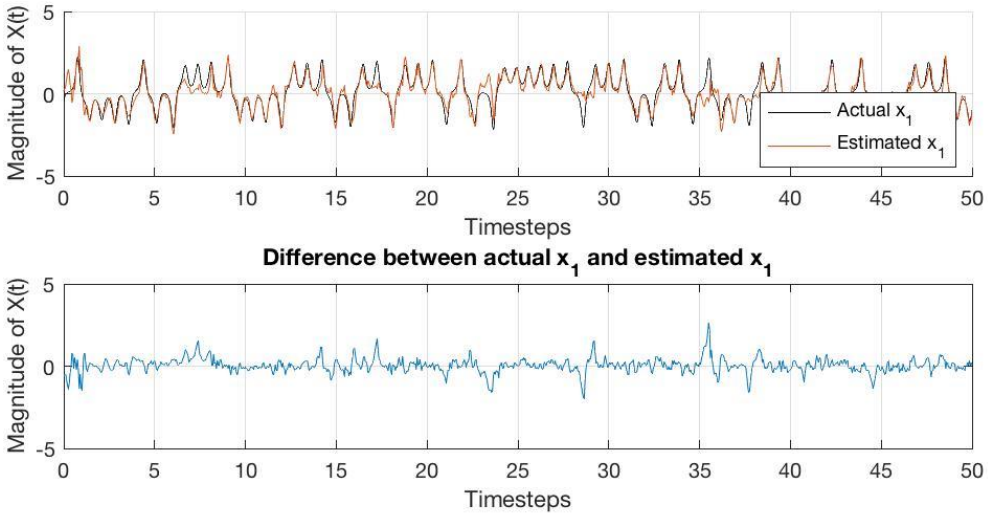


Figure 32: In the top panel; the expected $S_1 = x_1$ plotted with the predicted $\hat{S}_1 = \hat{x}_1$. In the bottom panel; the difference between the expected and estimated signals. Estimated using RC of $N = 800, \alpha = 0.9, k = 0.5, T = 1 * 10^4$, measurements sampled every five steps i.e. 0.05 time units. Calculated error $E_r = 0.29$.

At this point the model is achieving accuracy similar to that of the first reservoir. Although now, the accuracy is being achieved with code that runs much quicker, the speed comes from using a sampling step of five which in turn gives a reduced $T = 1 * 10^4$. Interestingly so far our optimum α is $\alpha = 0.9$, this is in disagreement with the work of Krishnagopal *et al* [5] that uses $\alpha = 0.3$.

Attention is now turned to the input strength normalisation k. Earlier in this work the absolute maximal values represented here by k for values randomly generated for the input weight matrix $\mathbf{W_{in}}$ was $k = 0.5$. In the work of Krishnagopal *et al* [5] the optimum value is found to be $k = 0.13$, therefore k is investigated next. The values $k \in [0.1, 0.3, 0.5, 0.7, 0.9, 1.0]$ are tested.

Immediately it is evident that decreasing k causes the accuracy of the network to decrease. This finding is in direct conflict with the published findings mentioned earlier. It is found that $k = 0.1$ and $k = 0.3$ generate $E_r = 0.62$ and $E_r = 0.37$ respectively, averaged over three random initialisations of the network each. On the other hand increasing k actually offers improved results. The optimum value is found to be $k = 0.9$ generating $E_r = 0.25$, averaged over three random initialisations of the network. The increased accuracy from this adjustment is shown in Figure 33. Again, a result that disagrees with the findings of published work [5].

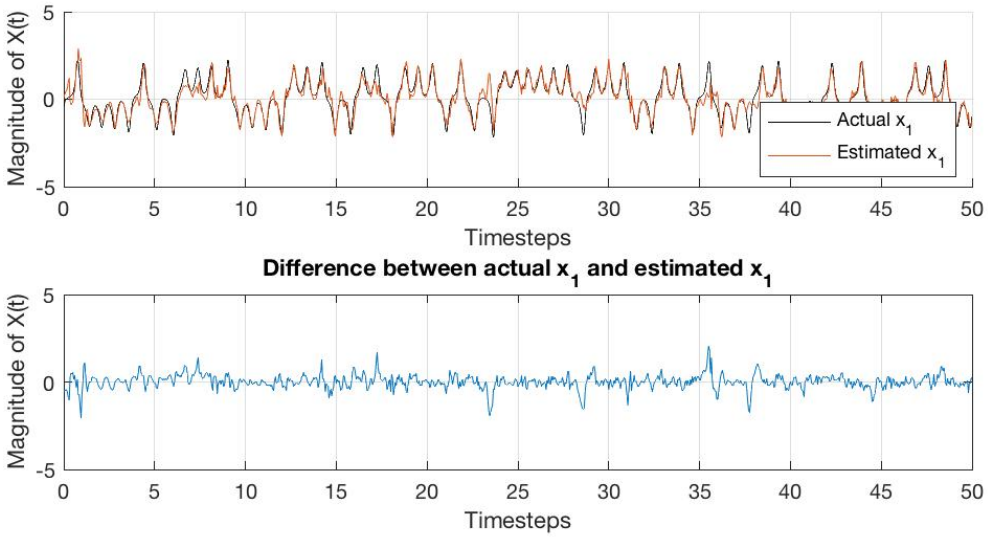


Figure 33: In the top panel; the expected $S_1 = x_1$ plotted with the predicted $\hat{S}_1 = \hat{x}_1$. In the bottom panel; the difference between the expected and estimated signals. Estimated using RC $N = 800, \alpha = 0.9, k = 0.9, T = 1 * 10^4$, Lorenz measurements sampled every five steps i.e. 0.05 time units. Calculated error $E_r = 0.25$.

Finally N is considered. In chapter 5 increasing the reservoir size is shown to improve the accuracy of reservoir predictions. In the work of Krishnagopal *et al* it is found that E_r is reduced as N increases, saturating at about $N = 2000$ [5]. This guides the decision to increase N to $N = 2000$. Increasing N in this way reduces the error to $E_r = 0.19$; the smallest error achieved thus far. This error is found by averaging over three random initialisation of the RC. This reduced error and predictive accuracy is displayed in Figure 34.

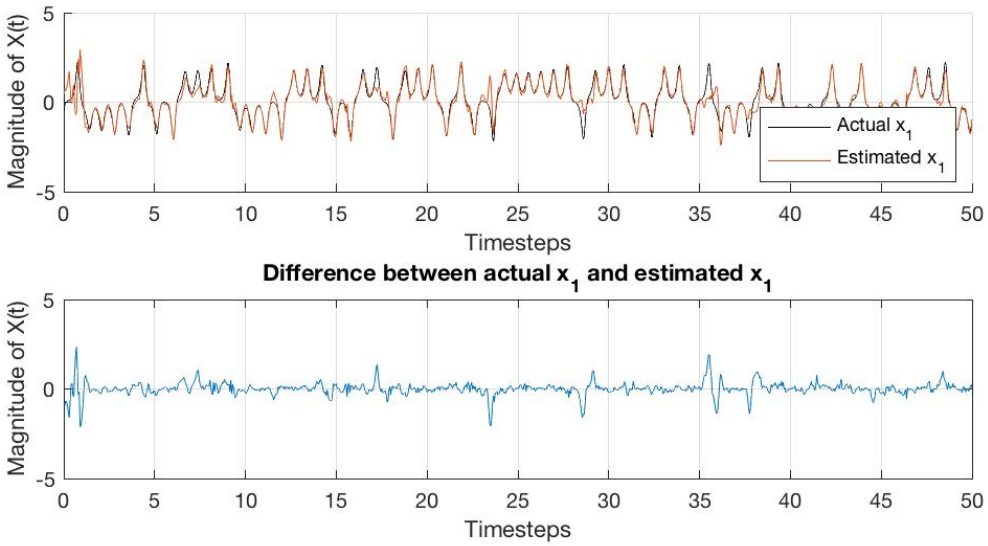


Figure 34: In the top panel; the expected $S_1 = x_1$ plotted with the predicted $\hat{S}_1 = \hat{x}_1$. In the bottom panel; the difference between the expected and estimated signals. Estimated using RC $N = 2000, \alpha = 0.9, k = 0.9, T = 1 * 10^4$, measurements sampled every five steps i.e. 0.05 time units. Calculated error $E_r = 0.19$.

Analysing the top panel of Figure 34 it can be concluded qualitatively that the $\hat{S}_1$ appears to look almost identical to the S_1. Although the predicted signal is still not exactly correct, it has been shown that by making adjustments to the RC the error can be reduced from $E_r = 0.29$ to $E_r = 0.19$.

Unfortunately, with some of these results the optimum values obtained disagree with those of Krishnagopal *et al.* Given the success in both sets of work, it is important to speculate as to why this has occurred. It could be worth considering the optimum training length decided on by Krishnagopal *et al.* These possibilities are considered in chapter 7, along with a discussion and evaluation of the remainder of the results obtained in this chapter.

7 Conclusions

At the beginning of this thesis three main aims were outlined, the main body of this work was then dedicated to achieving them.

The first of these aims was to explore the dynamics of the reservoir layer of the ESN in the absence of an input and output layer. Furthermore, the desire was to analyse the response of the dynamics of the reservoir layer to the leaking rate α and spectral radius ρ being adjusted. Overall this aim was achieved. Five reservoir layers were initialised with varying ρ, upon these α was varied and the reservoir states were collected and analysed.

It was found that the way in which the reservoir states develop through time depends hugely upon these hyper-parameters. The states converged to zero, oscillated periodically, and behaved irregularly (possibly chaotic) depending on ρ and α. It was found that for $\rho \in (0, 1)$ the network is stable in the sense that the reservoir will always converge to an equilibrium; increasing ρ above this interval causes the states to oscillate. This is because after the reservoir weights $\mathbf{W}$ are normalised they are rescaled according to ρ, meaning the reservoir will expand as some of the units in $\mathbf{W}$ will have connectivity greater than 1. Looking to equation (6) it becomes clear as to why the reservoir will expand and contract for ρ outside the interval $\rho \in (0, 1)$.

It was also found that the for $\alpha \in (0, 1)$ a few different things occurred depending on the ρ of the reservoir. While $\rho \in (0, 1)$, increasing α within the interval $\alpha \in (0, 1)$ reduced the number of time-steps it took for the states of the network to converge to zero. When $\rho > 1$, increasing α in the interval $\alpha \in (0, 1)$ caused the frequency of the oscillations of the states of the network to increase. Finally, in the case of $\rho > 1$ and $\alpha > 1$, the states of the network seemed to exhibit irregular behaviour.

Overall, this part of the thesis gave insight into understanding the equation that governs the states of the reservoir. The results reveal clear intervals for the values of ρ and α that lead to certain reservoir dynamics. The goal here was to explore how changing certain hyper-parameters impacted the states of the network; this was most certainly achieved. In future work if one instead had the intention of determining the best set of values for a particular task (predicting Lorenz system or Rössler sytem, or the addition of sines for example) then one could perhaps streamline the process of finding optimum hyper-parameter values with a different technique like grid search optimisation [57]. In future research it would also be of interest to observe the dynamics of the reservoirs states while the ESN is being trained or making a prediction.

The second aim was to train an RC to autonomously predict the dynamics of the chaotic Lorenz system. While investigating this aim, an appreciation was built toward the concept of chaos, particularly toward how difficult it is to predict the behaviour of a chaotic system for long periods of time. This part of the thesis can also be considered a success. Toward the end of chapter five, a model had been achieved that was able to predict the measurements of the chaotic Lorenz system for six full time units. This result is a staggering success, especially when one considers how chaotic systems fluctuate so heavily with time. It is extremely difficult to predict the long-term behaviour of a chaotic system with any accuracy. Hypothetically with an understanding of the Lorenz equations and the given initial conditions, one can predict the trajectories of the system but to do this with arbitrary accuracy is not possible. In this work we have demonstrated that with reservoir computing it is possible to model chaotic dynamics for a short time. This chapter in itself would be of interest to recreate and perhaps expand.

With models designed earlier in chapter five, lengths of time shorter than 6 units were also predicted which is still quite remarkable. The issue with these early models is that when their predictions diverged from the data, they began to oscillate periodically, which is not characteristic of the chaotic regime Lorenz system. This shift to periodicity makes sense as the spectral radius was $\rho < 1$ and there was no more perturbation from an external system.

Despite this, even when the predictions of the final model diverged from the data, the structure of the predicted signal still kept a shape consistent with that of the chaotic Lorenz system. Although this was pleasing, it could be the case that after a number of time units greater than 50 has passed, this model will also produce periodic predictions. This is something that should be investigated in future research. It would also be of interest to verify if sampling of the training data is a technique that could be used while investigating this. Only using a sample of training data (e.g.. every five measurements) reduced the training time drastically in the chapter following this, and it did not worsen the accuracy of the predictions of the RC.

Lastly, the third aim of the thesis was to train an RC that could be used in the decomposition of a linear superpositions of chaotic signals. While deducing an efficient model, the work by Krishnagopal *et al* [5] was very useful; it gave ideas to try and also provided results with which to make comparisons.

This aim has also been a success; toward the end of chapter 6 a model was created that could successfully isolate a component from a mixture of chaotic signals. Even better, the results show that a successful model can be created with a reasonable training time and the accuracy of the model can be improved by adjusting some of the hyper-parameters of the network. Achieving this is remarkable given the nature of chaotic systems, these results show that reservoir computing is a reliable technique for the decomposition of chaotic signals.

Furthermore, as the reservoir has its own dynamical properties it has the potential to be studied in more depth in the future.

It was found that increasing the size of the network will reduce the error between predictions and data, saturating at a network of size $N = 2000$ nodes. Other ideal hyper-parameters and variables were found to be: leaking rate $\alpha = 0.9$, input strength normalisation $k = 0.9$ and training length $T = 1 * 10^4$. Unfortunately the optimum values of α and k settled on in this work differ from those of Krishnagopal *et al* [5]. This could be due to the difference in the range of initial values of $\mathbf{W}$ used in this work and their work. As mentioned previously, it could also be possible that this discrepancy in optimum values can be put down to the training length T.

Krishnagopal *et al* reason that $T = 5 * 10^4$ is the optimum training length as it gives them $E_r \approx 0.15$. In the work of this chapter, before sampling is applied $T = 5 * 10^4$ is also used at first; but does not achieve such a small error. It would make sense to assume that Krishnagopal *et al* are using $T = 5 * 10^4$ after sampling has taken place. Prior to sampling their training length would have been $T = 2.5 * 10^5$. In the work of this chapter, the original training length is $T = 5 * 10^4$; sampled every fifth measurement to give $T = 1 * 10^4$. This potential difference in T, could perhaps explain the other discrepancies.

There is a discrepancy regarding the optimum α and k. The possible range of the entries of the input matrix $\mathbf{W_{in}}$ depends on k. Both α and $\mathbf{W_{in}}$ are included in equation (6), meaning their values influence how the internal states of the network develop. Using a high value of α or k will mean the adjustment made to the states $\vec{s}_n$ at each step n will be more significant. In the case of a smaller T, these large adjustments are more favourable as the states need to reach their required value in fewer steps.

On the other hand, with a larger T, the states develop over the course of a higher number of steps. Given the exposure to more data, using higher values of α and k may cause overtraining/overfitting of the network. In this case errors would tend to accumulate; the model would lose its accuracy. With a larger T, it will then make more sense to use smaller values of α and k, as shown by Krishnagopal *et al* [5]. Verifying this would be of great interest in future work on this subject.

It would also be of interest to pursue recreating some of the other investigations conducted by Krishnagopal *et al* [5], such as using an unknown mixing fraction, a, and training an additional RC to determine this value.

Regardless, with the completion of this aim, the utility of machine learning in the decomposition of linearly combined chaotic signals using reservoir computing has been demonstrated. With more advanced and efficient networks; and further training data and time; this work could potentially be furthered, and applied to more complex signal extraction, in fields like geology, biology, or economic systems.

Appendix

Appendix A: MATLAB Code

Listings


```
%%%                                                          %%%
%%% This is for investigating the effects of the sparsity  %%%
%%%  and the spectral radius.                               %%%
%%%                                                          %%%

tic; % Taking the time script takes to run

N = 100; % Number of reservoir units
lim = 0.5; % range of values for matrix
epsilon = 0.2; % How sparse should the matrix be

W = -lim + (lim+lim) * rand(N); % Filling with values [-lim,lim
    ]

for i = 1:numel(W)
    if rand(1) > epsilon % making the matrix sparse
        W(i) = 0;        % done randomly on entries
    end
end

largest = abs(eigs(W,1,'lm')); % lm is largest magnitude
```

```
21                                  % finding spectral radius
22
23 rho = 1.1; % Spectral radius to be rescaled to
24
25 W = (W/largest) * rho; % Rescale such that spectral radius is
       rho
26
27 execution = toc;
28 fprintf('Execution time was %.2f seconds .\n',execution);
```

Listing 1: Initialising Only Reservoir Layer

```
1 %%%
      %%%
2 %%% This is for investigating the effects of the leaking rate
      %%%
3 %%% and the activation function.
      %%%
4 %%%
      %%%
5
6 tic; % Taking the time script takes to run
7 count = 10; % variable to keep track of progress in increments
      of 5 percent
8
9 T = 1200; % Timesteps
10 alpha = 0.95; % leaking rate
11 snpast = -1 + (1+1) * rand(N,1); % state vector
12
13 X = zeros(N);                    % state matrix
14
15 %%% Doing the calculations %%%
16
17 for t = 1:T
18     % Update equation
19     for i = 1:numel(snpast)
20         % numel = number of array elements
21         sn = (1 - alpha)*snpast + alpha*tanh(W*snpast);
```

```
22      end  % update equation is different for no input network
23      % Expanding the state matrix X
24      if sum(X) ~= 0
25          X = [X,sn];
26      end
27      if sum(X) == 0
28          X = sn;
29      end
30      snpast = sn;      % Changing sn to snpast
31
32      percent = round(t/T * 100,1); % Show progress every five
           percent
33      if percent == count
34          fprintf('Progress: %.1f Percent .\n',percent);
35          count = count+5;
36      end
37  end
38  execution = toc; % Show how long the script ran
39  fprintf('Execution time was %.2f seconds .\n',execution);
40
41  % Average of X%
42
43  k = mean(X);
44  Xnotransient = X(:,(201:1200));
45  % Getting the proper Fourier Transform
46
47  %%Time specifications:
48     %Fs = length(k);
49     Fs = T;                          % samples per second
50     dt = 1/Fs;                         % seconds per sample
51     StopTime = 1;                      % seconds
52     time = (0:dt:StopTime-dt)';
53     n = size(time,1);
54     b = k(201:1200);    % the signal for fft and removing
          transient
55     %%% ^^^^^^^^^^^^ %%%%
56        % How much transient removed? look to "b" and "
```

```
          Xnotransient" %
57
58    %%Fourier Transform%%
59    B = fft(b);
60    B_mag = abs(B);
61
62    %%Shifted FFT%%
63    B_shift = fftshift(fft(b));
64    B_mag_shift = abs(B_shift);
65    B_power = B_mag_shift.^2;
66
67    %%Frequency specifications:
68    dF = 1;%length(b)/n;                          % hertz
69    f = -length(b)/2:dF:(length(b)/2)-dF;              % hertz
70
71    % Power Spectral Density
72
73    psdb = (1/(Fs*n))*abs(B).^2;
74    %freq = 0:Fs/(2*length(b)):(Fs)-(Fs/(length(b)));
```

Listing 2: Runng Only Reservoir Layer

```
1  %Lorenz Input
2
3  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
4  dt = 0.01; % 0.02
5  tf = 600; % 100
6  tspan = 0:dt:tf;
7
8  sigma = 10;
9  beta = 8/3; %% Lorenz system %%
10 rho = 28;
11 f = @(t,a) [-sigma*a(1) + sigma*a(2); rho*a(1) - a(2) - a(1)*a
       (3); -beta*a(3) + a(1)*a(2)];
12 [t,a1] = ode45(f,tspan,[1 1 1]);     % Runge-Kutta 4th/5th
       order ODE solver
13
14 sigma2 = sigma*1.2;
```

```
beta2 = beta*1.2;
rho2 = rho*1.2;
g = @(t,b) [-sigma2*b(1) + sigma2*b(2); rho2*b(1) - b(2) - b(1)
   *b(3); -beta2*b(3) + b(1)*b(2)];
[t,a2] = ode45(g,tspan,[1 1 1]);     % Runge-Kutta 4th/5th
   order ODE solver

%%

mixing = 0.5;

combined = ((mixing^0.5)*a1) + (((1-mixing)^0.5)*a2);

S1 = normalize(a1(:,1));
S2 = normalize(a2(:,1));
U = normalize(combined(:,1));

%%
subplot(2,1,1)
plot(combined(:,1))
xlim([0 10000]);

subplot(2,1,2)
plot([0:0.01:50],U(1:5001))
%xlim([0 5000]);
%
%%
subplot(2,2,1)
grid on;
hold on;
rotate3d on;

plot3(a1(:,1),a1(:,2),a1(:,3),'Color','black');
xlabel('X(t)')
ylabel('Y(t)')
zlabel('Z(t)')
plot3(a2(:,1),a2(:,2),a2(:,3),'Color','red');
```

```
hold off;

subplot(2,2,2)
grid on;
hold on;
plot(a1(:,1))
plot(a2(:,1))

%%
```

Listing 3: Obtaining Lorenz System Measurements

```
tic; % Taking the time script takes to run

bin = 1; %  The Input and
bout = 1;%  Output Biases

N = 2000; % Number of reservoir units
lim = 0.5; % range of values for matrix
epsilon = 0.2; % How sparse should the matrix be

W = -lim + (lim+lim) * rand(N); % Filling with values [-lim,lim
   ]

for i = 1:numel(W)
    if rand(1) > epsilon % making the matrix sparse
        W(i) = 0;        % done randomly on entries
    end
end

largest = abs(eigs(W,1,'lm')); % lm is largest magnitude
                               % finding spectral radius

rho = 0.9; % Spectral radius to be rescaled to

W = (W/largest) * rho; % Rescale such that spectral radius is
   rho

```

```
25 % Initialise state vector
26
27 snpast = -1 + (1+1) * rand(N,1);
28
29 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
30 % Input Matrix and Input vector %
31 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
32
33 Nu =1; %3    % The input nodes/units
34 sig = 1;
35
36 k=0.9;
37 %Win = -sig + (sig + sig) * rand([N,Nu+1]); % input matrix
38 %Win = -0.5 + rand([N,Nu+1]);
39 Win = (k-(-k)).*rand([N,Nu+1])+(-k);
40
41 un = rand(Nu,1);     % input vector
42                      % random numbers, Nu rows, 1 column
43
44 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
45 % Output Matrix and Output vector %
46 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
47
48 Ny = 1; %3     % The output nodes/units
49
50 Wout = -0.5 + rand([Ny, 1+N+Nu]);
51
52 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
53 % State Matrices X and Y        %
54 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
55
56 X = zeros(1+Nu+N); % state matrix
57 Y = zeros(1+Nu+N); % output matrix
58
59 Xtran = zeros(1+Nu+N); % state matrix  %% These are for the
      transient
60 Ytran = zeros(1+Nu+N); % output matrix %%
```

```
61
62 % Everything created, but still need output vector yn
63 % and state vector sn, and extended state vector xn
64 % They will be created during the running of the network
65
66 execution = toc;
67 fprintf('Execution time was %.2f seconds .\n',execution);
```

Listing 4: Creating Echo State Network

```
1  tic; % Taking the time script takes to run
2  % This is where the network will be trained %
3
4  % c = cat(dim,A,B) %
5  alpha = 0.5;
6  lambda = 0.00005; %0.0005
7
8  count = 5;
9  T0 = 5000;
10 T = 10000;
11
12 for t = 1:T0 % removing transient
13
14     un = [a(t,1),a(t,2),a(t,3)].';        % teacher forcing
          input.
15          %calculates values at timestep 't'
16
17     for i = 1:numel(snpast)
18         sn = (1 - alpha)*snpast + alpha*tanh(W*snpast + Win*cat
              (1,[bin],un));
19     end
20
21      %Creating extended inner state vector xn,
22
23     xn = [bin];
24     xn = cat(1,xn,sn);
25     xn = cat(1,xn,un);
26
```

```
27      %Creating output vector yn,
28
29      yn = [a(t+1,1),a(t+1,2),a(t+1,3)].';  % teacher forcing
           output, [a(t+1,1),a(t+1,2),a(t+1,3)].';
30          % calculates values at timestep 't+1'
31
32      % Expanding the state matrix "X"
33
34      if sum(Xtran) ~= 0
35          Xtran = [Xtran,xn];
36      end
37
38      if sum(Xtran) == 0
39          Xtran = xn;
40      end
41
42      % Expanding the output matrix "Y"
43
44      if sum(Ytran) ~= 0
45          Ytran = [Ytran,yn];
46      end
47      if sum(Ytran) == 0
48          Ytran = yn;
49      end
50
51      % Training the network
52
53      Wout = Ytran*(Xtran.') * inv(Xtran*(Xtran.') + lambda * eye
           (size(Xtran*(Xtran.'),1)));
54
55      % Updating the state to the next step
56
57      snpast = sn;
58
59  end
60
61  for t = 1:T
```

```
62
63      un = [a(t,1),a(t,2),a(t,3)].';        % teacher forcing
           input.
64          % calculates values at timestep 't'
65
66      for i = 1:numel(snpast)
67          sn = (1 - alpha)*snpast + alpha*tanh(W*snpast + Win*cat
               (1,[bin],un));
68      end
69
70      % Creating extended inner state vector xn,
71
72      xn = [bin];
73      xn = cat(1,xn,sn);
74      xn = cat(1,xn,un);
75
76      % Creating output vector yn,
77
78      yn = [a(t+1,1),a(t+1,2),a(t+1,3)].';  % teacher forcing
           output,
79          % calculates values at timestep 't+1'
80
81      % Expanding the state matrix "X"
82
83      if sum(X) ~= 0
84          X = [X,xn];
85      end
86
87      if sum(X) == 0
88          X = xn;
89      end
90
91      % Expanding the output matrix "Y"
92
93      if sum(Y) ~= 0
94          Y = [Y,yn];
95      end
```

```
96      if sum(Y) == 0
97          Y = yn;
98      end
99
100     % Training the network
101
102     Wout = Y*(X.') * inv(X*(X.') + lambda * eye(size(X*(X.'),1)
            ));
103
104     % Updating the state to the next step
105
106     snpast = sn;
107
108     percent = round(t/T * 100,1);
109     if percent == count
110     fprintf('Progress: %.1f Percent.\n',percent);
111     count = count + 5;
112     end
113
114 end
115 % The input is applied for T0 time steps to let some
116 % transient of the network decay. After this for T time steps
117 % the input is applied and the extended states are observed
118 % and collected.
119
120
121 execution = toc;
122 fprintf('Execution time was %.2f seconds .\n',execution);
123 beep on;
```

Listing 5: Training Echo State Network

```
1 tic; % Taking the time script takes to run
2 % This is where the network will be trained %
3 count2=5;
4 X = zeros(1+Nu+N);
5 Y = zeros(1+Nu+N);
6
```

```
%d = c;
h = a;

beginning = T+1;
finish = T+5001;

for t = beginning:finish

    un = [h(t,1),h(t,2),h(t,3)].';


    for i = 1:numel(snpast)
        sn = (1 - alpha)*snpast + alpha*tanh(W*snpast + Win*cat
           (1,[bin],un));
    end

    % Creating extended inner state vector xn,

    xn = [bin];
    xn = cat(1,xn,sn);
    xn = cat(1,xn,un);

    % Creating output vector yn,

    yn = Wout * xn;  % The output vector.

    % Expanding the state matrix "X"

    if sum(X) ~= 0
        X = [X,xn];
    end

    if sum(X) == 0
        X = xn;
    end

    % Expanding the output matrix "Y"
```

```
43
44      if sum(Y) ~= 0
45          Y = [Y,yn];
46      end
47      if sum(Y) == 0
48          Y = yn;
49      end
50
51      % Updating the state to the next step
52
53      snpast = sn;
54
55      h(t+1,1) = yn(1); % Predicting x(t)
56      h(t+1,2) = yn(2);
57      h(t+1,3) = yn(3);
58
59      percent = round(t/finish * 100,1);
60      if percent == count
61      fprintf('Progress: %.1f Percent.\n',percent);
62      count2 = count2 + 5;
63      end
64
65  end
66  % The input is applied for T0 time steps to let some
67  % transient of the network decay. After this for T time steps
68  % the input is applied and the extended states are observed
69  % and collected.
70
71
72  execution = toc;
73  fprintf('Execution time was %.2f seconds .\n',execution);
```

Listing 6: Predicting with Echo State Network

```
1  %close;
2  %clear;
3
4  %sigma = 10;
```

```
%beta = 8/3;
%rho = 28;
%tspan = [0 100];
%init = [1 1 1];
%options=odeset('RelTol',1e-4,'AbsTol',1e-4);
%f = @(t,a) [-sigma*a(1) + sigma*a(2); rho*a(1) - a(2) - a(1)*a
    (3); -beta*a(3) + a(1)*a(2)];
%[t,a] = ode45(f,tspan,init,options);     % Runge-Kutta 4th/5th
     order ODE solver
%plot3(a(:,1),a(:,2),a(:,3),'Color','blue');
%hold on;
%grid on;
%rotate3d on;

z=a([T+1:finish],3);
m=prod(size(z));
zz=[];

for j=3:m,
  if ( z(j-2) <= z(j-1) && z(j-1) >= z(j))
    zz=[zz z(j-1)];
  end
end

mm=prod(size(zz));

%%%%%%%%%%%%%%%%%%%%%

z2=h([T+1:finish],3);
m2=prod(size(z2));
zz2=[];

for k=3:m2,
  if ( z2(k-2) <= z2(k-1) && z2(k-1) >= z2(k))
    zz2=[zz2 z2(k-1)];
  end
end
```

```
40
41 mm2=prod(size(zz2));
42
43
44 plot(zz(1:mm-1), zz(2:mm),'.','Color','r');hold on;grid on
45 plot(zz2(1:mm2-1), zz2(2:mm2),'.','Color','k');
46 xlabel('z(n)');
47 ylabel('z(n+1)');
48 axis([min(zz) max(zz) min(zz) max(zz)]);
49 legend('Real Z(t)','Predicted Z(t)','Location','northeast')
```

Listing 7: Producing Return Map

Appendix B: Extra Information and Figures

Overfitting/overtraining

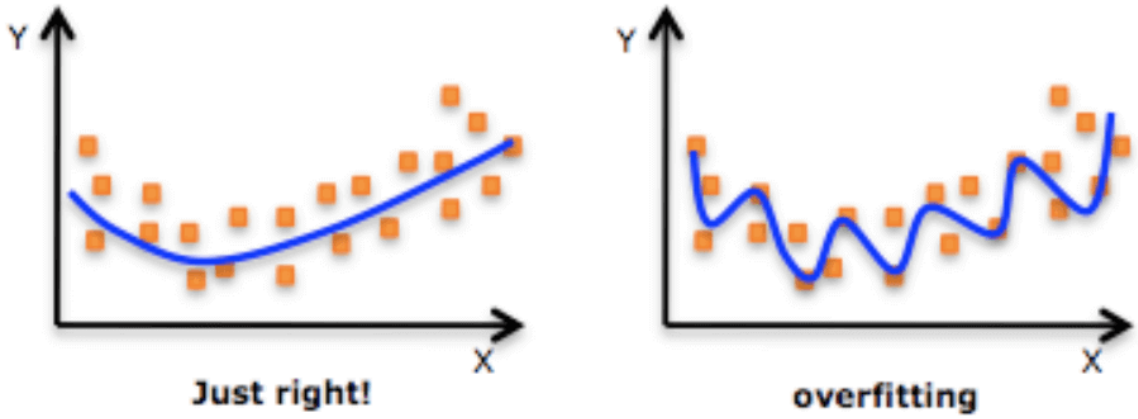


Figure 35: The image on the left shows an example of a model that matches the data suitably. The image on the right shows an example of a model that has been overtrained, this means it has been too attuned to the data it was trained with and will offer no usefulness when applied to other data. Image source [58].

Echo State Property

For the echo state network to function correctly, the reservoir must have the *echo state property* (ESP), which links asymptotic qualities of the hidden layer's dynamics to the driving signal. The ESP naturally shows that the hidden layer will asymptotically get rid of any information from initial conditions. Network's with additive-sigmoid neurons always have the ESP, if the hidden layer and the leaking rate satisfies certain algebraic conditions. Using *tanh* sigmoid stops the ESP for zero input if the spectral radius of the hidden layer is below unity. On the other hand the ESP is always held for input of any kind if the spectral radius is smaller than unity [55].

References

[1] H. 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*, vol. 148, no. 34, p. 13, 2001.

[2] C. C. Aggarwal *et al.*, *Neural networks and deep learning*. Springer, 2018.

[3] M. Congedo, C. Gouy-Pailler, and C. Jutten, "On the blind source separation of human electroencephalogram by approximate joint diagonalization of second order statistics," *Clinical Neurophysiology*, vol. 119, no. 12, pp. 2677–2686, 2008.

[4] J. Basak, A. Sudarshan, D. Trivedi, and M. Santhanam, "Weather data mining using independent component analysis," *The Journal of Machine Learning Research*, vol. 5, pp. 239–253, 2004.

[5] S. Krishnagopal, M. Girvan, E. Ott, and B. R. Hunt, "Separation of chaotic signals by reservoir computing," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 30, no. 2, p. 023123, 2020.

[6] P. Potocnik, "Neural networks: Matlab examples," *Neural Network course (practical Examples)*, 2012.

[7] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[8] T. Joutou and K. Yanai, "A food image recognition system with multiple kernel learning," in *2009 16th IEEE International Conference on Image Processing (ICIP)*. IEEE, 2009, pp. 285–288.

[9] N. Richárd, "The differences between artificial and biological neural networks," *Towards Data Science*, 2018.

[10] P. Kim, "Matlab deep learning," *With Machine Learning, Neural Networks and Artificial Intelligence*, vol. 130, p. 21, 2017.

[11] R. Plant, "An introduction to artificial intelligence," in *32nd Aerospace Sciences Meeting and Exhibit*, 2011, p. 294.

[12] M. Varone, D. Mayer, and A. Melegari, "What is machine learning? a definition," *Expert System*, 2019.

[13] "What is a neural network?" Aug 2020, accessed: 12th Jan 2021. [Online]. Available: https://databricks.com/glossary/neural-network

[14] DeepAI, "Neural network," May 2019, accessed: 12th Jan 2021. [Online]. Available: https://deepai.org/machine-learning-glossary-and-terms/neural-network

[15] B. Schrauwen, D. Verstraeten, and J. Van Campenhout, "An overview of reservoir computing: theory, applications and implementations," in *Proceedings of the 15th european symposium on artificial neural networks. p. 471-482 2007*, 2007, pp. 471–482.

[16] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[17] "The role of bias in neural networks," accessed: 14th Jan 2021. [Online]. Available: https://www.pico.net/kb/the-role-of-bias-in-neural-networks

[18] R. Quiza and J. Davim, *Computational Methods and Optimization*, 01 2011, pp. 177–208.

[19] DeepAI, "Feed forward neural network," May 2019, accessed: 15th Jan 2021. [Online]. Available: https://deepai.org/machine-learning-glossary-and-terms/feed-forward-neural-network

[20] M. Boden, "A guide to recurrent neural networks and backpropagation," *the Dallas project*, 2002.

[21] R. M. Schmidt, "Recurrent neural networks (rnns): A gentle introduction and overview," *arXiv preprint arXiv:1912.05911*, 2019.

[22] M. Lukoševičius and H. Jaeger, "Reservoir computing approaches to recurrent neural network training," *Computer Science Review*, vol. 3, no. 3, pp. 127 – 149, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1574013709000173

[23] H. Jaeger, *Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the" echo state network" approach*. GMD-Forschungszentrum Informationstechnik Bonn, 2002, vol. 5.

[24] G.-Z. Sun, H.-H. Chen, and Y.-C. Lee, "Turing equivalence of neural networks with second order connection weights," in *IJCNN-91-Seattle International Joint Conference on Neural Networks*, vol. 2. IEEE, 1991, pp. 357–362.

[25] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural computation*, vol. 1, no. 2, pp. 270–280, 1989.

[26] A. Atiya and A. Parlos, "Identification of nonlinear dynamics using a general spatio-temporal network," *Mathematical and computer modelling*, vol. 21, no. 1-2, pp. 53–71, 1995.

[27] R. E. Kalman *et al.*, "Contributions to the theory of optimal control," *Bol. soc. mat. mexicana*, vol. 5, no. 2, pp. 102–119, 1960.

[28] T. Katte, "Recurrent neural network and its various architecture types," *International Journal of Research and Scientific Innovation (IJRSI)*, vol. 5, pp. 124–129, 2018.

[29] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Computation*, vol. 1, no. 2, pp. 270–280, 1989. [Online]. Available: https://doi.org/10.1162/neco.1989.1.2.270

[30] C. Goller and A. Kuchler, "Learning task-dependent distributed representations by back-propagation through structure," in *Proceedings of International Conference on Neural Networks (ICNN'96)*, vol. 1, 1996, pp. 347–352 vol.1.

[31] G. Ciaburro, "Keras 2.x projects," accessed: 15th Mar 2021. [Online]. Available: https://www.oreilly.com/library/view/keras-2x-projects/9781789536645/4dee47c6-a030-45a8-b950-7044728fe494.xhtml

[32] W. Little, "The existence of persistent states in the brain," *Mathematical Biosciences*, vol. 19, no. 1, pp. 101 – 120, 1974. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0025556474900315

[33] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the national academy of sciences*, vol. 79, no. 8, pp. 2554–2558, 1982.

[34] D. O. D. O. Hebb, *The organization of behavior a neuropsychological theory*. Mahwah, N.J. : L. Erlbaum Associates, c2002.

[35] G. Lendaris, K. Mathia, and R. Saeks, "Linear hopfield networks and constrained optimization," *IEEE transactions on systems, man, and cybernetics. Part B, Cybernetics : a publication of the IEEE Systems, Man, and Cybernetics Society*, vol. 29, pp. 114–8, 02 1999.

[36] J. L. Elman, "Finding structure in time," *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990.

[37] M. Jordan, "Serial order: a parallel distributed processing approach. technical report, june 1985-march 1986," California Univ., San Diego, La Jolla (USA). Inst. for Cognitive Science, Tech. Rep., 1986.

[38] J. Schmidhuber, "Learning complex, extended sequences using the principle of history compression," *Neural Computation*, vol. 4, no. 2, pp. 234–242, 1992. [Online]. Available: https://doi.org/10.1162/neco.1992.4.2.234

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

[40] "Understanding lstm networks," accessed: 21st Jan 2021. [Online]. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/#fn1

[41] M. Sundermeyer, R. Schlüter, and H. Ney, "Lstm neural networks for language modeling," in *Thirteenth annual conference of the international speech communication association*, 2012.

[42] D. Nelson, "What are rnns and lstms in deep learning?" Aug 2020. [Online]. Available: https://www.unite.ai/what-are-rnns-and-lstms-in-deep-learning/

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

[44] H. Jaeger, W. Maass, and J. Principe, "Special issue on echo state networks and liquid state machines." 2007.

[45] L. F. Seoane, "Evolutionary aspects of reservoir computing," *Philosophical Transactions of the Royal Society B*, vol. 374, no. 1774, p. 20180377, 2019.

[46] R. S. Zimmermann and U. Parlitz, "Observing spatio-temporal dynamics of excitable media using reservoir computing," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 28, no. 4, p. 043118, 2018.

[47] C. Sparrow, *The Lorenz equations: bifurcations, chaos, and strange attractors*. Springer Science & Business Media, 2012, vol. 41.

[48] L. Prandtl, "Essentials of fluid dynamics: with applications to hydraulics, aeronautics, meteorology and other subjets," 1953.

[49] S. Grossmann and D. Lohse, "Prandtl and rayleigh number dependence of the reynolds number in turbulent thermal convection," *Physical Review E*, vol. 66, no. 1, p. 016305, 2002.

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

[51] D. Ruelle and F. Takens, "On the nature of turbulence, common. math. phys., 20,167-192," 1971.

[52] A. Vulpiani, F. Cecconi, and M. Cencini, *Chaos: from simple models to complex systems*. World Scientific, 2009, vol. 17.

[53] M. Lukoševičius and H. Jaeger, "Reservoir computing approaches to recurrent neural network training," *Computer Science Review*, vol. 3, no. 3, pp. 127–149, 2009.

[54] "Social network for programmers and developers," accessed: 15th Mar 2021. [Online]. Available: https://morioh.com/p/15c995420be6

[55] H. Jaeger, "Echo state network," accessed: 15th Mar 2021. [Online]. Available: http://www.scholarpedia.org/article/Echo_state_network

[56] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in Fortran 90: Numerical recipes in Fortran 77V. 2. Numerical recipes in Fortran 90*. Cambridge University Press, 1996.

[57] L. A. Thiede and U. Parlitz, "Gradient based hyperparameter optimization in echo state networks," *Neural Networks*, vol. 115, pp. 23–29, 2019, accessed: 15th Mar 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0893608019300413

[58] "Overfitting: Datarobot artificial intelligence wiki," accessed: 15th Mar 2021. [Online]. Available: https://www.datarobot.com/wiki/overfitting/