SCHOOL OF COMPUTER SCIENCE

Research Dissertation: Exploration of a Hippocampus-based Reservoir Neural network

2028298

Pakkaphon(Max) Prasertsan

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Wednesday 15th May, 2024

Abstract

A project has been initiated to explore the feasibility of a low-cost neural network that could run on most off-the-shelf machines, such as a personal laptop.

The primary objective of this project is to create a functioning reservoir computing neural network that uses the concept of excitatory and inhibitory neurons. Excitatory neurons stimulate the activity of other neurons and are represented as positive weighting in the codes, while inhibitory neurons inhibit the activity of other neurons and are represented as negative weighting in a similar manner to the excitatory. The combination of these two types of neurons is known to be important for the proper functioning of the brain.

Once the network was created, it was tested against complex equations such as the Lorenz-63 and Rössler equations. These equations are known for their chaotic behaviour and are often used to test neural network performance. The test results were then analyzed using graphs, and the mean square error values were calculated to determine the network's viability.

The potential implications of this project are significant, as a low-cost neural network that can run on most off-the-shelf machines could be used to improve our understanding of neural networks and not just reservoir computing framework neural networks.

Dedication and Acknowledgements

I would like to express my gratitude to my supervisor, **Conor Houghton**, for his incredible insights and stories throughout the project. I feel privileged to be able to work with such an outstanding supervisor. I would not be able to complete this without his support.

I am grateful for the support I have received from my family and friends. They were there all the way through, even though they did not fully understand my topic.

Lastly, I dedicated this to my younger self, who decided to go through this challenging life experience.

Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Taught Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, this work is my own work. Work done in collaboration with, or with the assistance of others including AI methods, is indicated as such. I have identified all material in this dissertation which is not my own work through appropriate referencing and acknowledgement. Where I have quoted or otherwise incorporated material which is the work of others, I have included the source in the references. Any views expressed in the dissertation, other than referenced material, are those of the author.

Pakkaphon(Max) Prasertsan, Wednesday 15th May, 2024

Contents

List of Figures

List of Tables

Ethics Statement

This project did not require ethical review, as determined by my supervisor, Conor Houghton.

Supporting Technologies

This project utilized a range of open-source tools and libraries to develop a reservoir computing network. The process involved constructing the network, generating and curating data points, and generating visual representations of the results for further analysis. The project's success was attributed to the comprehensive approach taken, incorporating diverse tools and resources at each stage of development.

- Python 3 [42], which is an open-source, general-purpose programming language used for data analysis and machine learning operations. Python is commonly used for tasks within this data-intensive field that require flexible representations of networks with a clear and precise showcase of the network itself. Python has a rich ecosystem of libraries and frameworks tailored for machine learning and data analysis. Also, the community revolving around Python is extensive, with documents, tutorials, forums, and other resources that help produce and troubleshoot any issues during development.
- Matplotlib library [17] was used for plotting the graphs to represent actual results and the testing results from the network. It is a powerful tool for creating static, interactive and publishable visualisations. Mathplotlib is used to visualise the performance of the neural network. It can plot the learning curves showing both the actual values and predicted values. Matplotlib also offers the ability to export graphs in various formats such as PNG, PDF and SVG.
- Numpy library [15] was used for data manipulation on n-dimensional arrays with comprehensive mathematical functions. Numpy offers a wide range of functions for creating arrays and methods to perform calculation, manipulation, and translation.
- NetworkX [13], which is an open-source Python package, is used for creating the reservoir network.
- ScriPy [43] is a library containing fundamental algorithms for scientific computing in Python, which is built on top of the Numpy library. Scripy provides functions such as scientific computation and modules that perform integration, interpolation, linear algebra, optimisation, and statistic calculation.

Notation and Acronyms

All the notations and acronyms that I used in this report are listed here.

RC	:	Reservoir Computing
ANN	:	Artificial Neural Network
RNN	:	Recurrent Neural Network
ESN	:	Echo State Network
BPTT	:	Backpropagation through time
MSE	:	Mean Square Error

Chapter 1

Introduction

1.1 Context

The hippocampus serves several essential roles inside our brains. Research [5] has shown that the hippocampus is responsible for the formation, organisation, and consolidation of new memories, particularly those related to facts and events (declarative memory), spatial navigation and emotional regulation. The hippocampus is highly active in terms of biochemical and electrical communication.

Reservoir computers are designed for time-series analysis as an alternative to a traditional Machine learning Neural Network style, where the weights and variables are changed through a learning process. Two main recurrent neural networks use reservoir computing. The first is the Echo State network [18], which focuses on fixed and randomly assigned neurons in the hidden layers. The second is the Liquid State network [24], which focuses on spiking in neurons instead. Reservoir allows the network to overcome limitations related to traditional neural networks, such as local minima and vanishing gradients. Reservoir networks are commonly used in fields such as Biomedical [26] (e.g. Electroencephalogram, fMRI or biomarkers), Machinery [33] (e.g. sensors, motors or controllers), and Engineering (e.g. simulating and finding solutions to complex equations such as the Lorenz attractor from Chaos theory [22]).

In this report, I will explore the possibility that the hippocampus functionality can be adapted to a neural network with a reservoir computing framework, such as the Echo State network, to further enhance its ability to perform complex computations.

1.2 Aims & Objectives

1. Explore the Reservoir network framework.
2. Develop a functioning Echo state network that uses a reservoir computing framework.
3. Experiment with the Lorenz attractor equation to show the effectiveness of the network.
4. Incorporate the results to show the connection between the hippocampus and the reservoir neural network.
5. Develop an Echo state network with the weighting system based on the excitatory and inhibitory synapses within the hippocampus.
6. Evaluate the difference between the traditional reservoir network and the neuron-based reservoir network, then determine the viability of using RC-based RNN.

1.3 Related Work

In fields that require data processing but have limited access to high-processing computers, the RC framework is the preferred choice of artificial neural networks. It is used as a part of the solution to recognise and solve real-world problems. With more research into this topic, I found that the RC is a framework that excels at representing time in a more 'natural' way [39] inside the hidden layer calculation of the neural network, even if the system might seem chaotic from the randomly assigned weights in the hidden layer. RNNs have a mechanism which allows the hidden layer to remember the previous inputs and computations, making RNNs suitable to handle time-series problems. This applies to the RC framework but is implemented as an echo state, which retains a memory of the past inputs for a certain duration, also known as memory horizon. The Echo state property allows the reservoir to capture the temporal dependencies inherent in the input dataset. However, in the RC, the hidden layer does not perform an explicit backpropagation through time or BPTT, which makes it distinct from the RNNs. I kept this difference in mind while programming the neural network and focused on the echo state property that the framework must have. The program is based on the diagrams 1.1 and 1.2.

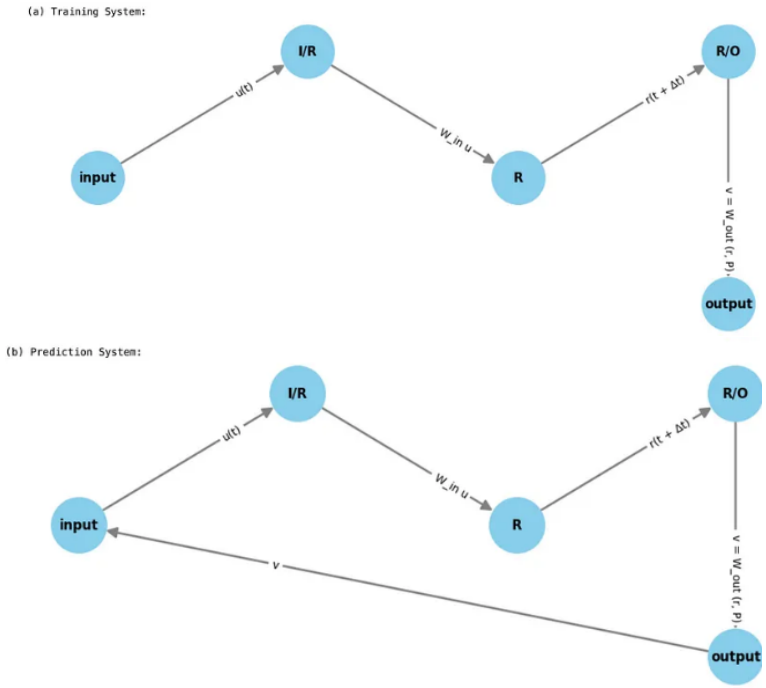

Figure 1.1: *(Left) A diagram showing the RC during the training phase. The diagram is taken from [7]*

Figure 1.2: *(Right) A diagram showing the RC during the prediction phase.I: input, R: Reservoir computer layer (the hidden layer), O: output. The equations are explained below in section 3. The diagram is taken from [7].*

RC networks have been shown to generalise temporal tasks well [14], such as classification and time-series prediction, because the weights inside the hidden layer are random but fixed. Combining with the fact that RCs are not trained and only the readouts are trained meaning that there is less risk of over-fitting to the training data.

Chapter 2

Background Review

In order to implement a hippocampus-like system in the reservoir neural network, I must understand both the hippocampus and a reservoir computing framework. For this background review section, I explored in detail the functionality of the hippocampus, what I could use as the base for improving my RNN and why I chose to use the reservoir framework.

2.1 What is the Hippocampus?

The hippocampus is a convex elevation of grey matter tissue within the parahippocampal gyrus inside the inferior temporal horn of the lateral ventricle [8] located in the central area of the brain as shown in figure 2.1. The functionality can be separated into many key domains, each contributing to ***declarative memory, spatial memory & navigation, stress regulation, neurogenesis, information processing, and emotional responses***.

With all the tasks combined, the hippocampus's main task is to intricately moderate cognition, emotion, and behaviour so that humans can navigate and adapt to function in complex environments.

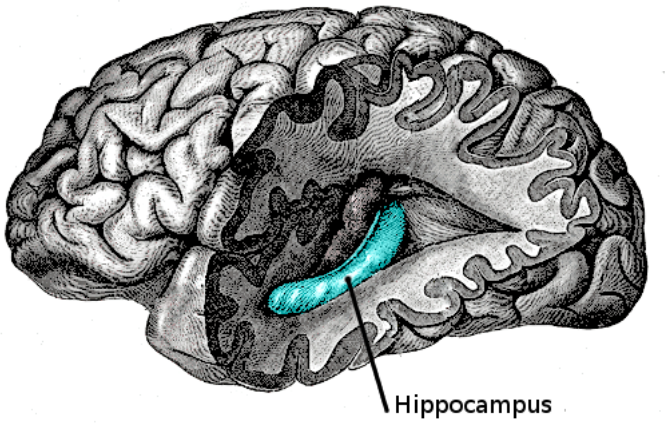

Figure 2.1: *Location of the hippocampus. Taken from [10]*

2.1.1 Declarative memory

Declarative memory is the conscious recollection of factual knowledge (semantic memory) and personal experience (episodic memory) that occur within the hippocampus [41]. Both types of memories can be considered parallel, with some similarities between their subroutines. Both are large, complex and highly structured [38] that can receive information through multiple modalities [25] and internally generated sources [20]. The principles for memory processing [35] for both cases have three steps: encoding, consolidating, and retrieving. The difference that separates episodic from semantic is during the retrieving process. The hippocampus would try to connect the neurons in a manner similar to the one when the memory had been encoded. In other words, the hippocampus is doing a "mental time travel" [41] to recall the past event. This process is usually modelled with artificial neural networks (ANNs) [34] that use the Hopfield network method [16], which is a single-layered and recurrent neural network to show the learning and retrieval process. Hopfield is used to represent the synaptic connection pattern in this dynamic activity. The dynamic of neurons (i) is represented by the equation:

$$\tau \dot{c}_i = -c_i(t) + \sum_{j=1}^{N} J_i j \cdot r_j(t) + \xi_i(t), \tag{2.1}$$

$$r_i = g(c_i) \tag{2.2}$$

where c and r are, respectively, the synaptic currents and the firing rates. J the connectivity matrix, each ξ_i is an independent random variable having a Gaussian distribution with mean zero and variance ξ_0 and τ is a constant.

The gain function (g) for this is:

$$\begin{cases} g = (x+\theta)^\gamma & x+\theta > 0, \\ g = 0 & x+\theta \leq 0. \end{cases} \tag{2.3}$$

The threshold for neuron activation is $\theta > 0$ and γ defines the gain in this equation.

This is one of the classical ways of representing the encoding and recalling of long-term memory within the hippocampus based on the Hopfield theory [16] from 1984. The flaw behind this is that the network is working on the assumption that the encoding and readout are happening at random with no predetermined pattern. A more realistic model would require some recalling feedback within the network.

2.1.2 Regulation of Stress and Emotion

The hippocampus controls and regulates stress by releasing hormones such as cortisol, which regulates the activity of the hypothalamic-pituitary-adrenal (HPA) axis [12]. The hippocampus mediates the feedback mechanisms to maintain homeostasis [28]. The hippocampus also controls and regulates emotion, interfacing with limbic structures such as the amygdala and prefrontal cortex. Both of these affect the behaviours and responses.

2.1.3 Spatial Memory and Navigation

Spatial Memory and Navigation within the hippocampus can be described as "a process of determining and maintaining a course of trajectory from one place to another" [6]. It helps construct cognitive maps and guide navigation through complex environments with the help of two different cells, Place & Grid. Place cells and Grid cells work together in an intricate network to encode and process spatial information received from the external stimulus [29]. Place cells located in CA1, CA2, and CA3 regions fire in response to specific locations within the environment, mainly the head direction [32]. The place cells typically fire when the animal makes an error, which can be seen in figure 2.2, and help navigate the animal to the path in reference to previously visited locations [45]. Grid cells fire in a pattern resembling a regularly spaced grid [29], either a triangular or hexagonal pattern, which simulates the position within the environment.

Place cells can show past experience, which could improve the implementation of ANN technology. Both place & grid cells take several minutes of exploration before having a stable firing field [9]. For this project, I could try to imitate the place & grid cells mechanism as the weighting system for the nodes.

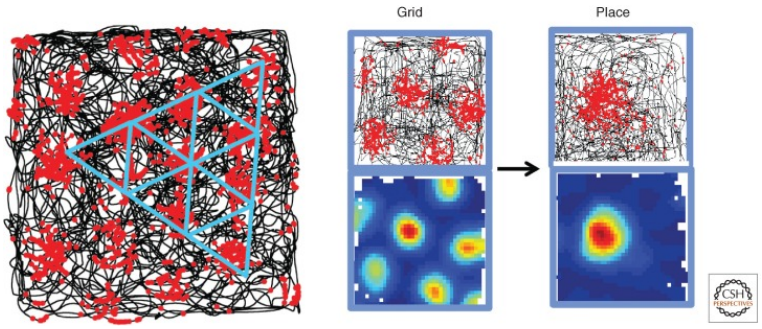

Figure 2.2: *Grid and place cells from the entorhinal cortex of a rat brain. The black line shows the movement pattern of the rat within an enclosed space. The red dot is used to represent the spike that occur during the experiment. A blue grid is used to show the distribution of the spike pattern. The diagram is taken from [29].*

2.1.4 Neurogenesis

The process by which new neurons are generated throughout adulthood is called *Neogenesis* [21]. This process is predominantly located in the dentate gyrus within the hippocampal formation. The main functionality is related to neuronal plasticity, learning, and memorising. This functionality underlines how the hippocampus is capable of self-renewal and adaptation in response to stimuli and past experiences.

2.1.5 Information Processing

The hippocampus also serves as a central hub for information processing. Sensory stimuli such as visual, auditory, and touch input are integrated and processed inside the brain to formulate a cohesive memory representation. The information processing process depends on excitatory and inhibitory inputs to neurons within the hippocampus, leading to the generation and degeneration of new memories [27]. Maintaining the balance between excitatory neurons, which are responsible for increasing the firing of the neurons, and inhibitory neurons, which are responsible for releasing neurotransmitters to suppress the neurons' activity, is the critical role in forming new memories.

The RNN could use Dale's law as part of its weighting to represent functionality similar to how neurons function in the hippocampus. According to Dale's law [44], a singular neuron is capable of releasing one type of neurotransmitter, either excitatory or inhibitory. The effects of the released neurotransmitter remain consistent at all termination sites. This generalisation aids in comprehending the fundamental principle and functionality of neurons.

2.2 What is a Reservoir Computing Neural Network

A reservoir neural network (RNN) is based on a computational framework from reservoir computing (RC) in recurrent neural networks [40] where input signals are mapped into a higher dimensional computational space filled with dynamics of a fixed, non-linear system. The parameters are set at random from the beginning and left fixed. A mapping mechanism is trained to read the state of the reservoir and map it to the desired outcome for that network. The RNN itself is shown in Figure 2.3 (a), where the networks are interconnected. For the physical problem, the reservoir network will be the one in Figure 2.3 (b), which could be in the form of a built-in microprocessor or control unit in a machine.

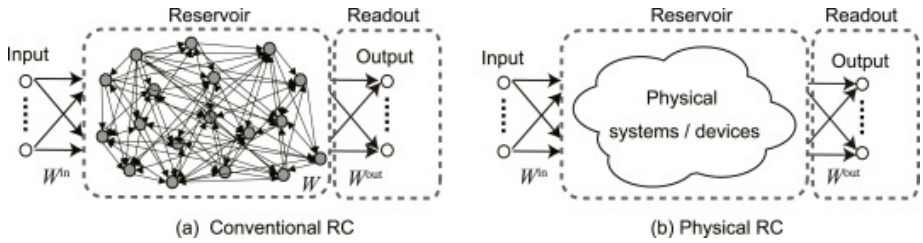

Figure 2.3: *RC or reservoir computing frameworks are represented here for easier visualisation. Diagrams are from [40]*

The basic principle of RC [31] starts with the input signal $u^{train}(t)$ and target output signal $y^{train}(t)$. A filter or transducer that takes in an input signal $u^{train}(t)$ and generates an output signal $\hat{y}^{train}(t)$ which is close to the desired target output $y^{train}(t)$. The reservoir framework is implemented in the filter section. The high-dimensional dynamical system $X(t) \in \mathbb{R}^N$, which is being driven by the inputs $u^{train}(t)$ and their corresponding internal signals $x_i^{train}(t)$, can be observed and recorded. The recorded signals in this stage are trained in the readout F stage. The readout function [4] maps every recorded state of $(x_1^{train}(t), ..., x_N^{train}(t))$ to output $\hat{y}^{train}(t)$, which is the close approximation to the desired/target output $y^{train}(t)$. Linear regression is commonly used for the readout function F. The RC can be simplified down to these equations:

$$X(t) = (x_1^{train}(t), ..., x_N^{train}(t)) \tag{2.4}$$

This is followed by the readout function F, which maps all recorded state $x_i^{train}(t)$

$$F(X(t)) = \hat{y}(t) \tag{2.5}$$

Where the $\hat{y}(t)$ is a close approximation to the actual $y(t)$ values.

$$\hat{y}(t) \approx y(t) \tag{2.6}$$

Reservoir computing is a versatile way to deal with challenging problems such as temporal input-output tasks in a time series prediction, classification and dynamic pattern generation. RC can be applied to the majority of engineering problems. The reservoir section can be replaced with any non-linear, high-dimensional fixed dynamic system.

2.2.1 Echo State Network

An Echo State Network (ESN), shown below in figure 2.4, is an example of a reservoir computer that runs on a recurrent neural network. The nodes in the hidden layer are sparsely connected. Following the reservoir computing framework, all the weights and variables are fixed at the start.

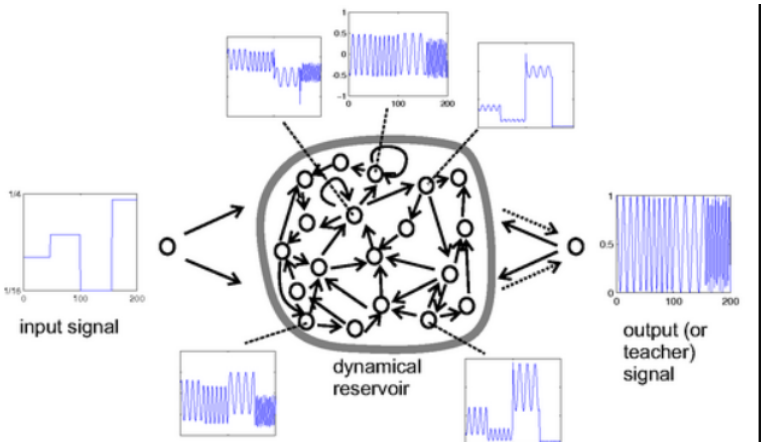

Figure 2.4: *The diagram showing the basic schema of an ESN. Solid arrows indicate fixed, random connections. The dotted arrows at the end are the trainable connection during the readout stage. Taken from [19]*

Training an ESN involves two phases: Initialising and Training the readout. During the initialisation, the nodes in the hidden layer are randomly connected and advanced using preset weights and parameters. The readout stage or layer is commonly trained with simple algorithms such as linear regression or gradient descent. The readout focuses on bringing the output closer to the desired output.

Chapter 3

Project Execution

To experiment with the reservoir computing framework, I created a program in Python3 that simulates and runs the Lorenz attractor equation (based on an example Echo state network from [7]).

3.1 Planning

The initial step was to research possible examples to run. I looked into options such as navigation problems, image recognition, and solving complex equations. I concluded that reservoir neural network testing with the Lorenz-63 system would be best.

Reasons for using Lorenz-63 system

1. Evaluation of Model Complexity
2. Bench-marking the Performance
3. Robustness Evaluation
4. Real-World Applications

A complex equation, such as Lorenz-63, would rigorously test the neural network's ability beyond simple pattern tasks. It would reveal the RNN's robustness and ability to handle uncertain variables in the system. By recreating the system and using the network to find the solution, I am able to evaluate the capacities, generalisation abilities, and suitability for real-world applications.

Lorenz-63 system, more commonly known as the Lorenz attractor, was introduced by Edward Lorenz in a 1963 article [23]. The system is a set of three nonlinear ordinary differential equations that describe a simplified model of atmospheric convection. The purpose of the Lorenz-63 system is to act as a prototype for studying chaos theory and deterministic chaos. A small change in a system would result in drastically different outcomes, which illustrates the core concept of deterministic chaos.

The model is a system of three differential equations shown below:

$$\frac{dx}{dt} = \sigma(y - x), \tag{3.1}$$

$$\frac{dy}{dt} = x(\rho - z) - y, \tag{3.2}$$

$$\frac{dz}{dt} = xy - \beta z. \tag{3.3}$$

These differential equations relate the characteristics of a two-dimensional fluid layer uniformly warmed from below and cooled from above. All these equations are the rate of change with respect to **time**.

x, y and z are proportional to the rate of convection, horizontal temperature variation, and vertical temperature variation, respectively. The constants $\sigma, \rho, and \beta$ are the system parameters proportional to the Prandtl number [11], Rayleigh number [2], and the specific physical dimensions of the layer.

3.2 Building

For the purpose of testing, I created two versions of RC. One with the base style of the reservoir, which used a sigmoid for the weighting. Another was the implementation based on the hippocampus, which has two different types of weight: excitatory and inhibitory.

I started by creating a class for my base RC. The class **ReservoirComputer** contains five main methods with four additional helper functions. Each method and function are explained below.

3.2.1 Base ReservoirComputer

The hidden layer is based on the following equations: The $u(t)$ is an input vector. That is then put into the reservoir layer R. The input weight for each input is determined by the dimension of both the reservoir and the input.

$$W_{in}u(t) \tag{3.4}$$

Where

$$W_{in} \in \mathbb{R}^{D_r x D} \tag{3.5}$$

D_r = dimension of the reservoir and D = dimension of the input.

Inside the reservoir R, the input is combined with the current state, $r(t)$, to create the next state $r(t + dt)$. The reservoir is represented with an adjacency matrix A.

$$A \in \mathbb{R}^{D_r x D_r} \tag{3.6}$$

$$r(t + dt) = F[Ar(t) + W_{in}u(t)] \tag{3.7}$$

Where F is the mathematical function Sigmoid.

$$F(x) = \frac{1}{1 + e^{-x}} \tag{3.8}$$

Each node has a corresponding reservoir state. A mapping is used to obtain the same dimensionality as $u(t)$.

$$v(t + dt) = (r(t + dt), W_{out}) \tag{3.9}$$

Where $v(t + dt) \in \mathbb{R}^D$ and $W_{out} \in \mathbb{R}^{DxD_r}$.

Based on the reservoir computing framework, the goal of this system is to make the output $v(t)$ as close as possible to the target output $\hat{v}(t)$. The result from each state $r(t)$ is stored along with $u(t)$. The W_{out} is trained using those two data to minimise the mean squared difference between $v(t)$ & $\hat{v}(t)$. This is achieved through the regularised regression procedure below.

$$\sum_{t=t_0}^{T} ||r(t), W_{out} - \hat{v}_d(t)||^2 + \beta||W_{out}||^2 \tag{3.10}$$

The β acts as a regulator, which prevents over-fitting if $\beta > 0$. For the prediction phase, $v(t + dt) = u(t + dt)$, which uses the $v(t + dt)$ act as the input for the next stage $u(t + dt)$.

$$r(t + dt) = F[Ar(t) + W_{in}(r(t), W_{out})] \tag{3.11}$$

The output of the trained reservoir will give the input (predicted) value for the next time-step $u(t)$ for $t > 0$. For the implementation, I split the data into 80% as training data and the other 20% for validation. The implementation of the reservoir computing framework is simplified with the help of NumPy [15] and networkX libraries [13]. NumPy focuses on arithmetic operations such as matrix operation and calculation.

3.3 Coding

To start my program, I imported all the important libraries required to run and simulate my RNN. Numpy, networkX, scipy and csv are all standard libraries commonly used in data processing and network simulations.

```
import numpy as np
import networkx as nx
from scipy.integrate import solve_ivp
import csv
```

Listing 3.1: All the libraries required for this project.

3.3.1 __init__Method

The __init__method is the fundamental component of the ReservoirComputer object. The method initialises each attribute by setting it up and configuring it for my use later on.

1. **dim_system**: Dimension of the input-output space.
2. **dim_reservoir**: Dimension of the reservoir layer.
3. **rho**, **sigma**: Parameters controlling the structure of the reservoir.
4. **r_state**: Initial state of the reservoir.
5. **A**: The Adjacency matrix of the reservoir. Initialised with zero and updated with values from generate_adjacency_matrix function.
6. **W_in**: Input weights connecting the input to the reservoir, which is based on the equation from 3.7.
7. **W_out**: Output weights connecting the reservoir to the readout, based on the equation 3.10.

```
import numpy as np
import networkx as nx
from scipy.integrate import solve_ivp
import csv

class ReservoirComputer:
    def __init__(self, dim_system, dim_reservoir, rho, sigma):
        self.dim_system = dim_system
        self.dim_reservoir = dim_reservoir
        self.r_state = np.zeros(dim_reservoir)
        self.A = self.generate_adjacency_matrix(dim_reservoir, rho, sigma)
        self.W_in = 2 * sigma * (np.random.rand(dim_reservoir, dim_system) - .5)
        self.W_out = np.zeros((dim_system, dim_reservoir))
```

Listing 3.2: Object Class ReservoirComputer, which needed to be instantiate by filling out these parameters before simulating the network.

3.3.2 advance_r_state Method

The **advance_r_state** method was designed to compute the next state of the reservoir with the given input **u** with sigmoid as the activation function based on the equation from 3.11. Then, updated the **r_state** attribute in **ReservoirComputer** object.

```
    def advance_r_state(self, u):
        self.r_state = self.sigmoid(np.dot(self.A, self.r_state) +
            np.dot(self.W_in, u))
        return self.r_state
```

Listing 3.3: *Object class ReservoirComputer, which contains methods for initialising itself, creating the next node state, computing the output, training the node, and predicting the next step.*

3.3.3 v Method

The **v** method computes the output of the reservoir (**v**) by multiplying the output weights or **W_out** with the current state of the reservoir or **r_state**.

```
    def v(self):
        return np.dot(self.W_out, self.r_state)
```

Listing 3.4: *Object class ReservoirComputer, which contains methods for initialising itself, creating the next node state, computing the output, training the node, and predicting the next step.*

3.3.4 train Method

The **train** method trains the output weights or **W_out** with **linear regression** based on the provided trajectory to find the optimal weights that would map the reservoir states to the corresponding outputs. The **R** matrix is constructed, where each column represents the state of the reservoir at a specific time step.

```
    def train(self, trajectory):
        R = np.zeros((self.dim_reservoir, trajectory.shape[0]))
        for i in range(trajectory.shape[0]):
            R[:, i] = self.r_state
            u = trajectory[i]
            self.advance_r_state(u)
        self.W_out = self.linear_regression(R, trajectory)
```

Listing 3.5: *Object class ReservoirComputer, which contains methods for initialising itself, creating the next node state, computing the output, training the node, and predicting the next step.*

3.3.5 predict Method

The **predict** method generates the predictions for a specified number of time steps, which use the trained output weights to compute the output of the reservoir at each step. Then, update the state of the reservoir using the predicted output.

```
    def predict(self, steps):
        prediction = np.zeros((steps, self.dim_system))
        for i in range(steps):
            v = self.v()
            prediction[i] = v
            self.advance_r_state(prediction[i])
        return prediction
```

Listing 3.6: *Object class ReservoirComputer, which contains methods for initialising itself, creating the next node state, computing the output, training the node, and predicting the next step.*

3.3.6 generate_adjacency_matrix

The **generate_adjacency_matrix** uses three parameters, **dim_reservoir**, **rho**, and **sigma**. This method creates a random graph using the Erdős-Rényi model [3]. **nx.gnp_graph** [13] takes in two arguments: the number of nodes which is provided by **dim_reservoir**, and the probability of edge creation from **sigma**. The $G_{n,p}$ model chooses each of the possible edges with the probability p. This method also converts the graph produced by **nx.gnp_graph** into numPy array using the command **nx.to_numpy_array(graph)**. Then, before returning the array, the method rescales the matrix using the next method **scale_matrix**.

```
    @staticmethod
    def sigmoid(x):
        return np.where(x >= 0,
                        1 / (1 + np.exp(-x)),
                        np.exp(x) / (1 + np.exp(x)))

    @staticmethod
    def generate_adjacency_matrix(dim_reservoir, rho, sigma):
```

```
        graph = nx.gnp_random_graph(dim_reservoir, sigma)
        graph = nx.to_numpy_array(graph)
        random_array = 2 * (np.random.rand(dim_reservoir, dim_reservoir) - 0.5)
        rescaled = graph * random_array
        return ReservoirComputer.scale_matrix(rescaled, rho)

    @staticmethod
    def scale_matrix(A, rho):
        eigenvalues, _ = np.linalg.eig(A)
        max_eigenvalue = np.amax(eigenvalues)
        A = A / np.absolute(max_eigenvalue) * rho
        return A

    @staticmethod
    def linear_regression(R, trajectory, beta=0.0001):
        Rt = np.transpose(R)
        inverse_part = np.linalg.inv(np.dot(R, Rt) + beta *
            np.identity(R.shape[0]))
        return np.dot(np.dot(trajectory.T, Rt), inverse_part)
```

Listing 3.7: *Helper methods for the reservoir class.*

3.3.7 scale_matrix

scale_matrix scales the input matrix **A** to have the spectral radius of **rho**. The method calculates the eigenvalues of the provided input matrix using the function **np.linalg.eig(A)**. Then, the method finds the maximum absolute eigenvalue within the eigenvalues. Finally, rescales **A** such that the maximum absolute eigenvalue becomes **rho** and returns that matrix back.

3.3.8 linear_regression

The method **linear_regression** performs a basic linear regression on the input matrix **R**. By creating an inverse of the matrix **R**, then computes the matrix product between **R** and **inverse R**, which is regularised with a small positive value **beta** by multiplying it to the identity matrix. Then, the method performs a matrix multiplication with the transpose of the **trajectory** for the coefficients of the linear regression.

3.3.9 Lorenz-63 system

The function **Lorenz(xyz, sigma=10, rho=28, beta=** $\frac{8}{3}$**)**, shown in the figure 3.8, computes the derivatives of the Lorenz-63 system based on the input parameter **xyz**. Sigma, Rho, and beta are fixed to make it the same as the Lorenz-63 equation. Each line represents the derivatives of each axis (x, y, and z). The method then returns each coordinate back in the form of a list.

```
# Generate some Lorenz attractor data
def lorenz(t, xyz, sigma=10, rho=28, beta=8/3):
    x, y, z = xyz
    dxdt = sigma * (y - x)
    dydt = x * (rho - z) - y
    dzdt = x * y - beta * z
    return [dxdt, dydt, dzdt]
```

Listing 3.8: Codes for creating the Lorenz-63 data points.

The method then runs based on parameters assigned in the figure **??**. The time span for this simulation is from 0 to 100. Then, it generates 10,000 linearly spaced out timepoints within the **t_span**. All the variables are put in **ScriPy** [43] command that numerically integrates the provided system, which in this case is the Lorenz-63 equations.

```
t_span = (0, 100)
t_eval = np.linspace(*t_span, 10000)
initial_conditions = [1.0, 1.0, 1.0]  # Initial condition
solution = solve_ivp(lorenz, t_span, initial_conditions, t_eval=t_eval)
```

```

# Extract and normalize data
data = solution.y.T
data_mean = np.mean(data, axis=0)
data_std = np.std(data, axis=0)
data_normalized = (data - data_mean) / data_std

# Use part of the data for training and part for validation
train_data = data_normalized[:8000]
valid_data = data_normalized[8000:]
```

Listing 3.9: Codes for running the Lorenz-63 simulation.

3.4 Data Processing

The results from the Lorenz-63 are processed and normalised to be used later for the RNN. The data points are split between the training dataset and the validating dataset, as seen in figure 3.9. The first 8,000 timesteps of (x, y, and z) are selected for training, and the remaining timesteps (8,000 index onwards) are for validation.

3.4.1 Running Multiple tests and Recording

Each type of reservoir computing RNN runs and records **n** times, which is preset to 50 runs. Then, the training data set is used to train the RC and improve the prediction method using the validating data set.

The **predicted_data** is the result from the RC RNN, and the **actual_data** is from the Lorenz-63 results. Both of these data sets are recorded in the form shown in table 3.1 separately as a .csv file, as shown in figure 3.10. The full data files are attached separately alongside this report. The naming of the file is determined from its contents; for example, predicted_data_hippo_V1_n, where n is the current run number, is used for the first iteration of the hippocampus-based RC.

Actual data			Predicted data		
X	Y	Z	X	Y	Z
-1.117400326	-1.357522083	0.085671232	-1.11779749	-1.356660764	0.084958178
-1.159704176	-1.379243513	0.17335934	-1.160314903	-1.37709687	0.171005914
-1.200006452	-1.392934195	0.265928717	-1.200576686	-1.38939671	0.261195505
-1.237411006	-1.397543162	0.361712873	-1.237871959	-1.392864399	0.354326416

Table 3.1: Table showing sample x, y, z from the .csv file. Actual data and the predicted data from RNN.

```
# Define number of sets of CSV files
n = 50

# Loop to create and train reservoir computers, and save predicted and actual
    data to CSV files
for i in range(n):
    # Create and train reservoir computer
    rc = ReservoirComputer(dim_system, dim_reservoir, rho, sigma)
    rc.train(train_data)

    # Predict using the trained reservoir computer
    predicted_data = rc.predict(valid_data.shape[0])

    # Save the predicted data to a CSV file
    with open(f'predicted_data_hippo_{i}.csv', 'w', newline='') as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow(['x', 'y', 'z'])
        for row in predicted_data:
            writer.writerow(row)

```

```
    # Save the actual data to a CSV file
    with open(f'actual_data_hippo_{i}.csv', 'w', newline='') as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow(['x', 'y', 'z'])
        for row in valid_data:
            writer.writerow(row)
```

Listing 3.10: Codes for running the RC RNN and recording in the form of .csv files.

3.5 Hippocampus based ReservoirComputer

3.5.1 generate_input_weights

The **generate_input_weights** method generates the input weights for the hidden layer. The method takes two arguments: 1. **dim_reservoir**, which is the dimension of the reservoir layer. 2. **dim_system**, which is the dimension of the input-output space.

The weight or **W_in** is a matrix of size **dim_reservoir** by **dim_system**, initialises with zero, which then later fill out with the input weights. The loop iterates over each neuron inside the layer. Inside the loop, a subset of neurons is randomly chosen to connect to the current RC neuron. The list selects at least 5 random neurons from the system.

The neurons are then separated into two types: excitatory and inhibitory, with the ratio between them as 80:20. The method then randomly assigns weights to these neurons, as shown in figure **??**. The excitatory has uniform weights ranging from [0, 1], and the inhibitory has a uniform range of [-1, 0]. The method then returns the weight matrix back to the program.

The method creates a connectivity pattern between the different types of neurons within the system, assigning random weights to the connections based on Dale's law (shown in table 3.2), where the excitatory neuron can connect to either other excitatory neurons or inhibitory neurons. However, an inhibitory neuron can not connect to another inhibitory neuron.

No.	Potential Connection	Symbols
1.	Excitatory → Excitatory	EE
2.	Excitatory → Inhibitory	EI
3.	Inhibitory → Excitatory	IE

Table 3.2: Table showing the potential connection between the two types of neurons.

```
...

class ReservoirComputer:
    def __init__(self, dim_system, dim_reservoir, rho, sigma):
        self.dim_system = dim_system
        self.dim_reservoir = dim_reservoir
        self.r_state = np.zeros(dim_reservoir)

        # use generate_input_weights -> links to the E & I neurons.
        self.A = self.generate_adjacency_matrix(dim_reservoir, rho, sigma)
        self.W_in = self.generate_input_weights(dim_reservoir, dim_system)
        self.W_out = np.zeros((dim_system, dim_reservoir))

...

@staticmethod
    def generate_input_weights(dim_reservoir, dim_system):
        # Generate a connectivity pattern where each reservoir neuron connects
            to a subset of system neurons
        # Here, we connect each reservoir neuron to a fixed number of system
            neurons
        W_in = np.zeros((dim_reservoir, dim_system))
        for i in range(dim_reservoir):
            connected_neurons = np.random.choice(dim_system, size=min(5,
                dim_system), replace=False)
            num_excitatory = int(0.8 * len(connected_neurons))
            excitatory_neurons = np.random.choice(connected_neurons,
                size=num_excitatory, replace=False)
            inhibitory_neurons = np.setdiff1d(connected_neurons,
                excitatory_neurons)
            # Assign excitatory weights to excitatory neurons
            W_in[i, excitatory_neurons] = np.random.uniform(0, 1,
                len(excitatory_neurons))
            # Assign inhibitory weights to inhibitory neurons
            W_in[i, inhibitory_neurons] = np.random.uniform(-1, 0,
                len(inhibitory_neurons))
        return W_in
```

Listing 3.11: Codes for creating the weight in the system for RC based on Excitatory and Inhibitory neurons in the Hippocampus.

Chapter 4

Data Analysis

4.1 Method of Analysis

The methods I used to calculate whether the output produced by my RC network was closer to the target output or not was by using the Mean Square Error (MSE) [1] as stated previously in the testing section 4.2. The MSE measured the average squared difference between the estimated values and the actual values.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{4.1}$$

The equation in 4.1 gives a measure of the average squared deviation between the predicted and actual values. With the difference being squared, larger errors would contribute more to the final MSE value than smaller differences. The lower MSE value indicates that there is an agreement between actual and predicted values or, in my case, the $\hat{y}$ and y.

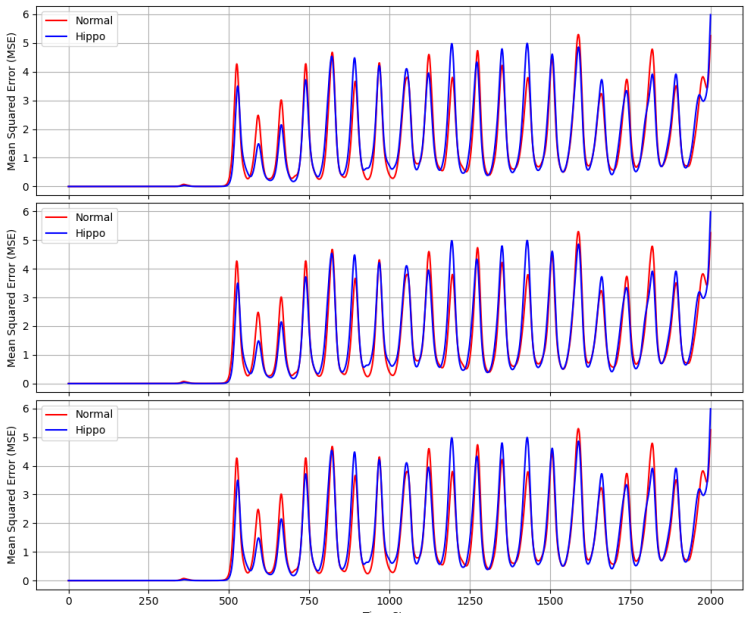

Figure 4.1: A graph showing the MSE values comparison between normal RC network against Hippocampus based RC network when compared to their target outputs

As shown in the graph in figure 4.1, I used the MSE values of each axis to determine whether there was an improvement between each iteration. The graph represents the MSE values between the predicted data from the hippocampus-based RC against the actual Lorenz-63 values and the MSE values of the normal RC network predicted data against the Lorenz-63 values. Both RCs were tested with the same starting parameters given in the table shown below in table 4.2.

4.2 Testing

To assess the effectiveness of the hippocampus-inspired RC network, I compared it to the traditional style of RC. However, these were the results of a single run I did as an experiment to explore the possibility of both RC types.

4.2.1 Initial setup

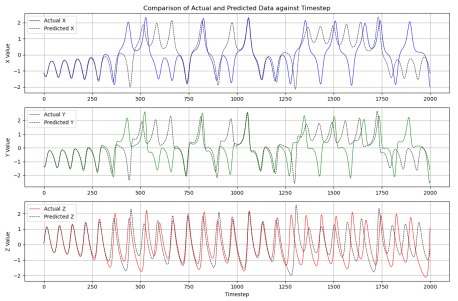

Figure 4.2: *(Left) Comparison between each axis and the values from Lorenz-63 against timesteps in reservoir computing neural network.*

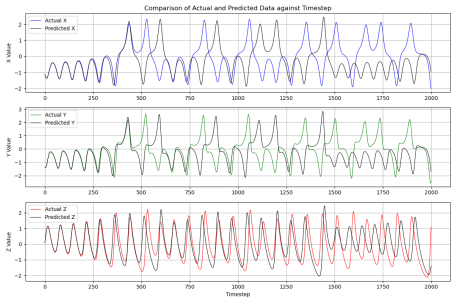

Figure 4.3: *(Right) Comparison between each axis and the values from Lorenz-63 against timesteps in hippocampus-based reservoir neural network.*

From the figure 4.2 and 4.3, the experiment was run with the same starting parameters. And as I carried out more tests, I changed up these parameters to optimise the RC network.

dim_system	dim_reservoir	rho	sigma
3	100	0.9	0.1

Table 4.1: Initial setup parameters for both RC networks.

To see the pattern and evaluate the robustness between the two, I decided to run 50 trials for each type. Then, I averaged the data points based on the time step. By doing 50 runs, I was able to detect the variability in the data and spot potential deviations and noises.

Each iteration had its parameters changed based on the results of the previous iteration, as shown in table 4.2. By only changing the parameters at the start and not changing any variable inside the hidden layer, I was able to avoid overfitting [30] from happening. Combined with the simplicity of the RC framework, the optimisation of the network managed not to capture all the noises and random fluctuations that occurred.

The changes between iterations were primarily **dim_reservoir & rho & sigma**. The aim of this testing was to increase the accuracy of the RC network that was using the hippocampus-based weighting system.

4.2.2 Finding the best parameters combination

After doing twelve iterations by hand, I decided that the process needed to be more rigorous in order to optimise the starting parameters. Therefore, I created a program that would run the RC network with a range of parameters I assigned.

```
# Grid Search
param_grid = {
    'dim_reservoir': np.linspace(200, 1000, 10, dtype=int),
    'rho': np.linspace(0.1, 10.0, 10),
    'sigma': np.linspace(0.1, 10.0, 10)
}
param_combinations = list(itertools.product(*param_grid.values()))
```

Listing 4.1: Codes for creating a list containing each parameter within the limited range given.

Iteration 1				Iteration 7			
dim_system	dim_reservoir	rho	sigma	dim_system	dim_reservoir	rho	sigma
3	100	0.9	0.1	3	370	6	3
Iteration 2				Iteration 8			
dim_system	dim_reservoir	rho	sigma	dim_system	dim_reservoir	rho	sigma
3	200	0.9	0.1	3	370	2.4	1.6
Iteration 3				Iteration 9			
dim_system	dim_reservoir	rho	sigma	dim_system	dim_reservoir	rho	sigma
3	300	0.9	0.1	3	370	2.4	3.8
Iteration 4				Iteration 10			
dim_system	dim_reservoir	rho	sigma	dim_system	dim_reservoir	rho	sigma
3	400	0.9	0.1	3	370	3.4	3.9
Iteration 5				Iteration 11			
dim_system	dim_reservoir	rho	sigma	dim_system	dim_reservoir	rho	sigma
3	350	0.9	0.1	3	470	3.5	3.9
Iteration 6				Iteration 12			
dim_system	dim_reservoir	rho	sigma	dim_system	dim_reservoir	rho	sigma
3	370	0.9	0.1	3	370	4.7	3.8

Table 4.2: Parameters for each iteration of both RC networks.

The line **param_combinations** created all the possible combinations of the parameters by taking the Cartesian product of the values in the **param_grid** through the function **itertools.product**. The combination was in a tuple format, starting with dim_reservoir, rho, and then sigma. The function was set to run every possible combination of the parameters and tested to find the combination with the lowest mean square error value.

4.2.3 Best MSE iteration or Iteration 13

As shown in the codes 4.1, the program set up a grid search for the parameter tuning. **dim_reservoir** was set to range from 200 to 1000 with 10 evenly spaced samples [15]. **rho** was set for 0.1 to 10.0 with 10 evenly spaced samples [15]. **sigma** had the range of 0.1 to 10.0 in a similar manner to **rho** [15]. This produced a total of 10^3 or 1,000 combinations of parameters.

The program then selects a combination out of the array and uses it to construct an RC network. Then, the network was tested to see if the output $\hat{y}$ was closer to the target output or y by using a mean square error calculation. Mean square error, or MSE, was used as a metric during my data evaluation.

```
best_mse = float('inf')
best_params = None

for params in param_combinations:
    dim_reservoir, rho, sigma = params
    rc = ReservoirComputer(dim_system, dim_reservoir, rho, sigma)
    rc.train(train_data)
    # Predict on validation data
    predictions = rc.predict(valid_data.shape[0])
    # Compute MSE
    mse = np.mean((predictions - valid_data) ** 2)
    # Update parameters if MSE improves
    if mse < best_mse:
        best_mse = mse
        best_params = params

print("Best Parameters from Grid Search:", best_params)
print("Best MSE from Grid Search:", best_mse)
```

Listing 4.2: Codes for running the RC network with the combination from the grid.

The results are shown in table 4.3, which was the combination that gave the lowest mean square error when put into the RC network.

dim_system	dim_reservoir	rho	sigma
3	828	3.635714285714286	1.5142857142857145

Table 4.3: The results from parameters testing program.

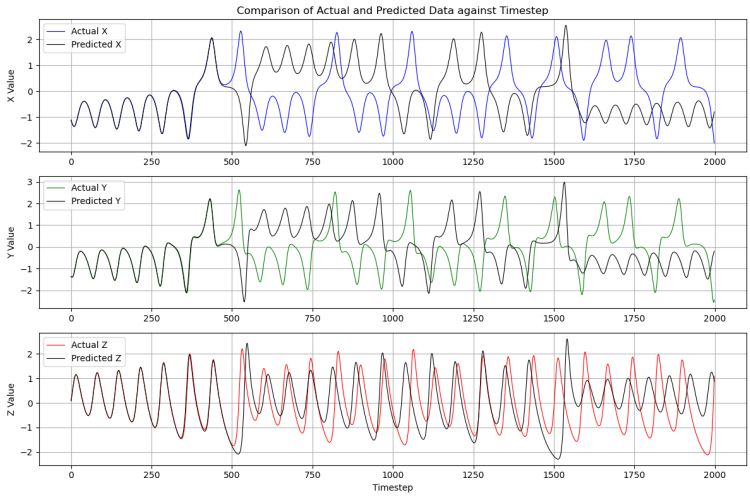

Figure 4.4: Graph showing the comparison between actual and predicted values created by the parameters from the grid searching program of a single run.

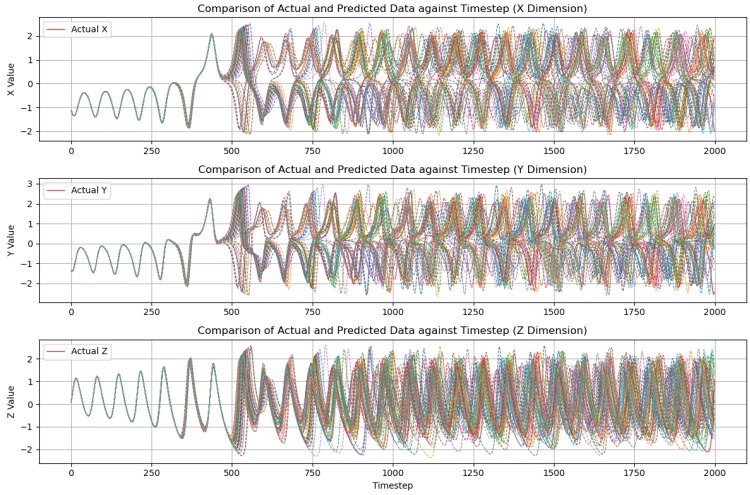

Figure 4.5: Graph showing all the 50 testing runs of iteration 13, using the parameters provided by grid-parameter search.

As shown in graph 4.4 and 4.5, the RC was able to keep up with target outputs for up to 468 timesteps. 38 out of 50 runs (76% accuracy) followed the target outputs up to 558 timesteps. However, after that point, the data became volatile and unpredictable as it fluctuated in an alternating manner between each run, which resulted in a peak on both positive and negative at the same timesteps as shown in graph 4.5. The peaks balanced out as shown in graph 4.6.

Average MSE for each axis on V13		
x	y	z
0.7330314053120357	0.7789141187475072	0.47540639668396045

Table 4.4: Table showing the average MSE values for each axis on the iteration 13 hippocampus-based RC network.

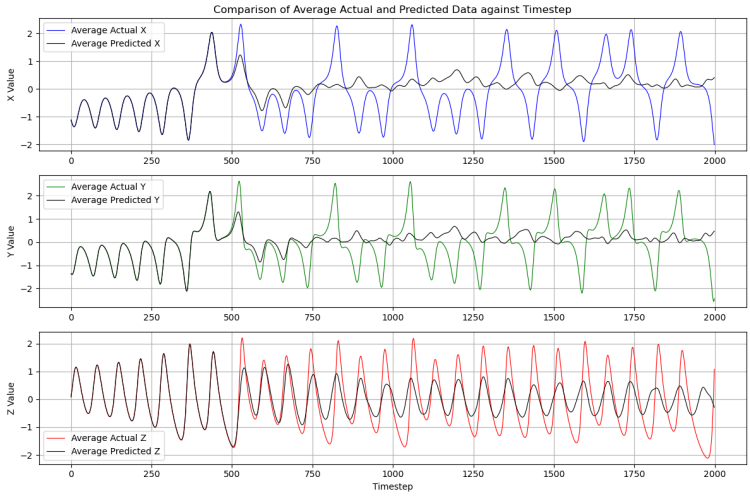

Figure 4.6: Graph showing the average of the predicted data created by hippocampus-based RC network of iteration 13, compared to the actual values from the Lorenz-63 equation.

4.2.4 Better MSE iteration or Iteration 14

After feeling dissatisfied with the iteration 13 results, I decided to up the grid search scope to 20 samples per parameter or 20^3 or 8,000 combinations. The results are shown in table 4.5.

dim_system	dim_reservoir	rho	sigma
3	1000	8.436842105263159	1.6631578947368424

Table 4.5: The results from the parameters testing program with more combinations.

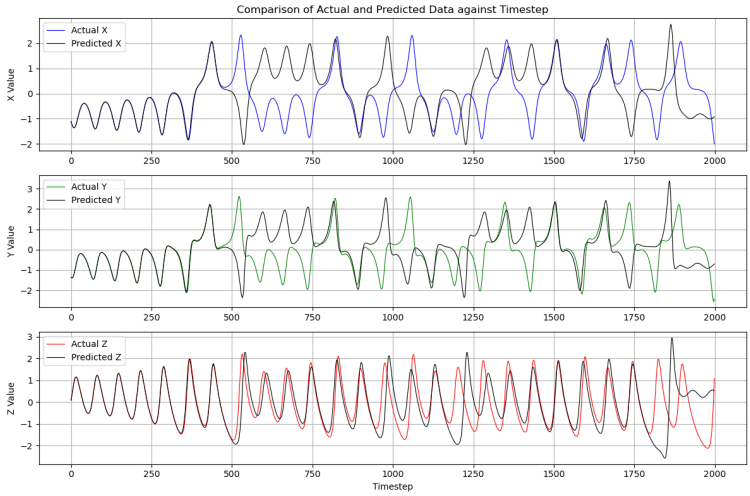

Figure 4.7: A graph which shows the data points for the x, y, and z axis against timesteps for iteration 14 of the hippocampus-based RC network.

In tables 4.4 and 4.6, the Mean Squared Error (MSE) values decreased, indicating that the parameters used in iteration 14 were an improvement when compared to iteration 13. The MSE values for both the x-axis and y-axis decreased by 6.85% and 6.46%, respectively. However, there was a 1.30% increase in MSE value for the z-axis. Despite this, iteration 14 still performed better than previous other iterations, as can be seen in the appendix C. Comparing iteration 14 with the first iteration presented in the appendix, the MSE values decreased by 12.89%, 12.27%, and 18.56% for the x-axis, y-axis, and z-axis, respectively.

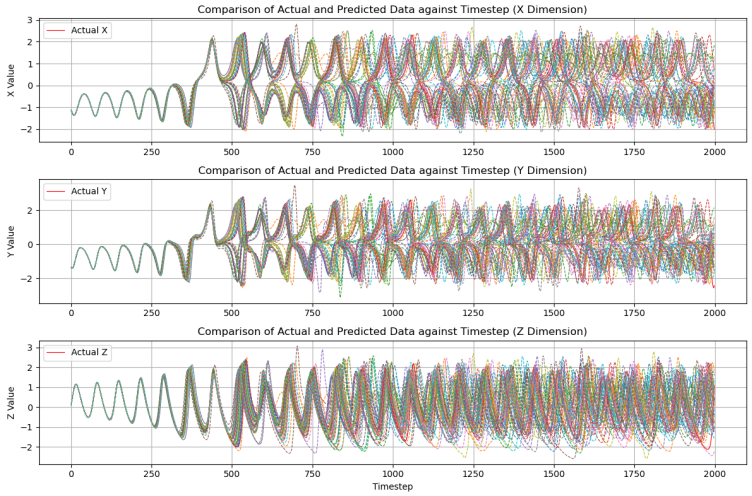

Figure 4.8: Graph showing all the 50 testing runs of iteration 14, using the parameters given from MSE best iteration 2.

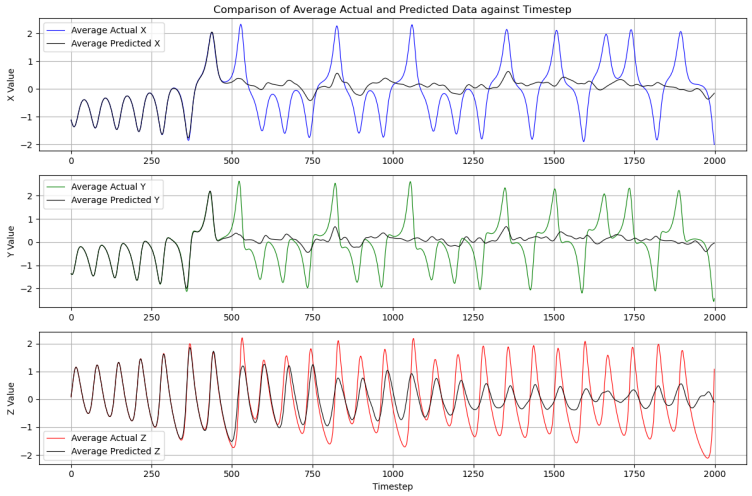

Figure 4.9: Graph showing the average of the predicted data created by hippocampus-based RC network of iteration 14, compared to the actual values from the Lorenz-63 equation.

Average MSE for each axis on V14		
x	y	z
0.6827603257209564	0.7284377521036528	0.481596343361533

Table 4.6: Table showing the average MSE values for each axis on the iteration 14 hippocampus-based RC network.

Upon further analysis, I determined that iteration 14 represents the most optimal network optimisation. This conclusion was reached after a thorough and comprehensive review of the network optimization process. The optimal solution, iteration 14, was determined after considering factors such as network capacity, performance, and efficiency, which can be seen in the significant decrease in the MSE values of each axis and managed to process the information within reasonable timing.

Chapter 5

Data Validation

To prove that my hippocampus-based RC network has been generalised and effective, I tested it against the Rössler equation, which exhibits similar chaotic behaviour to the Lorenz-63 equation. The RC network had never been trained on the Rössler equation values before; therefore, I decided to validate my network with it.

5.1 Testing with Rössler Equation

Rössler equation was formulated by Otto Rössler in the 1970s [37]. Rössler system is known for exhibiting chaotic behaviour [36] for certain parameter values, characterised by the sensitive dependence on the initial conditions and the irregular behaviour. Rössler equation is commonly used in fields such as physics, biology and engineering.

$$\frac{dx}{dt} = -y - z \tag{5.1}$$

$$\frac{dy}{dt} = x + ay \tag{5.2}$$

$$\frac{dz}{dt} = b + z(x - c) \tag{5.3}$$

Where a, b, and c are preset parameters [37], similar to how I set up the Lorenz-63 equation, with their sigma, rho, and beta. The program is shown in listing 5.1 with the preset values as shown in Rössler's research paper [37].

```
# Generate R ssler attractor data
def rossler(t, xyz, a=0.2, b=0.2, c=5.7):
    x, y, z = xyz
    dxdt = -y - z
    dydt = x + a * y
    dzdt = b + z * (x - c)
    return [dxdt, dydt, dzdt]
```

Listing 5.1: Codes for running the Rossler equation and creating datapoints.

To keep the test fair, I did 50 runs for a single set of parameters, which was from iteration 14, as stated previously in table 4.5 from section 4.2.4.

Average MSE for each axis		
x	y	z
0.046719274745742646	0.05034631832806868	0.06748444551149976

Table 5.1: Table showing the average MSE values for each axis on the hippocampus-based RC network when comparing to the Rötter equation.

As shown in table 5.1 and graph 5.1, the hippocampus-based RC network had more than 90% accuracy. This can be seen in the average MSE values on each axis.

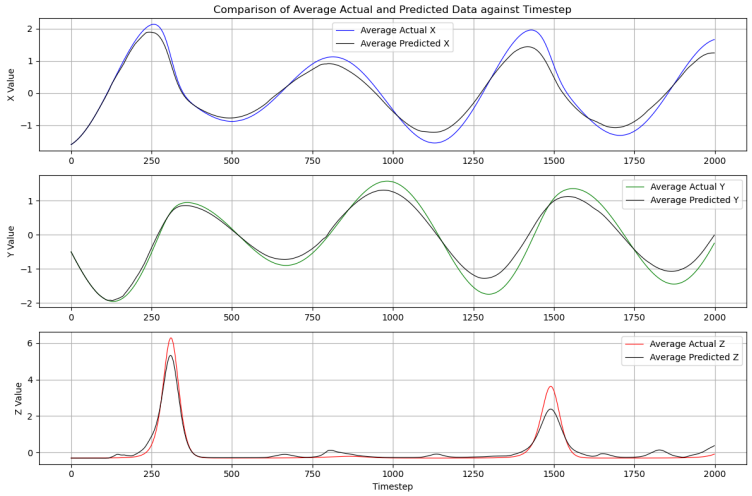

Figure 5.1: Graph showing the comparison between the actual values from Rössler equation and the predicted values from the hippocampus-based RC network.

5.2 Result Summary

To fully see the pattern, I did the testing three more times to prove that the RC network can capture the key features of Rössler equation.

Average MSE for each axis on the second test		
x	y	z
0.020494819357512763	0.02540127761582106	0.01208810198870395
Average MSE for each axis on the third test		
x	y	z
0.031428769764095935	0.035465707228459833	0.03432836032078586

Table 5.2: Table showing the average MSE values for each axis on the hippocampus-based RC network when comparing to the Rötter equation for the second and third testings.

After conducting further tests on my hippocampus-based RC network, I have concluded that it has an accuracy of over 95%. In the second test, the mean square errors for x, y, and z were found to be 2%, 2.4%, and 1.2%, respectively. Similarly, in the third test, the mean square errors for x, y, and z were 3.1%, 3.5%, and 3.4%, respectively., respectively.

In summary, the results from the additional tests conducted on my hippocampus-based RC network indicate a consistently high accuracy exceeding 95%. These findings underscore the robustness and stability of my RC network, reinforcing its efficacy for calculating complex equations.

Chapter 6

Conclusion

6.1 Overall Summary

The project took a significant amount of time to start due to my lack of understanding of the reservoir computing framework and how to implement and simulate the hippocampus's neurons inside the hidden layer. The research and experimenting took at least 40% of the overall project duration. Another 30% was spent on developing and optimising the hippocampus-based RC network. Finally, the last 30% was spent on focusing on validating the finished RC network and writing up the report.

Throughout the project, I encountered several challenges. The limited resources, including my laptop and time, prevented me from simulating larger RC networks. Hence, I restricted the size of the networks to three nodes, as increasing it would exponentially intensify the load on my device while running the network.

Despite encountering various challenges and obstacles, I was able to design and develop a hippocampus-based RC network successfully. This network has the capability to calculate complex equations beyond those found in the training data of the Lorenz-63 system. Through diligent work and perseverance, I was able to overcome the difficulties and create a functional and efficient RC network.

6.2 Achievements

I managed to complete all my aims and create a functional Echo state network with the reservoir computing framework. A second version contained a weighting system based on the hippocampus's neurons, as well as excitatory and inhibitory neurons.

Through numerous iterations, I managed to optimise the network to a satisfying level that can operate with two different equations that have chaotic properties.

I carried out data analysis on all the iterations, including the one with Rössler equation. All the results are recorded in .csv files.

6.3 Future Works and Potential Improvements

With more resources and time, I would like to optimize my network further to identify more effective matching parameters for the RC network and increase the size of the system dimension, such as the Lorenz-63 system and the Rössler equation.

To ensure optimal network performance, I intend to test it with a range of equations. This will allow me to confirm that the network can effectively process different input and output frequencies. By doing so, I will be able to identify any potential issues and implement necessary adjustments to further enhance its performance.

In my opinion, this project holds immense potential as a starting point for further research on improving low-cost neural networks. Such efforts could significantly expand the reach of AI technology, making it more accessible to individuals with limited resources. I would be thrilled to see the reservoir computing framework successfully implemented in other systems, including embedded systems.

Bibliography

[1] David M Allen. Mean square error of prediction as a criterion for selecting variables. *Technometrics*, 13(3):469–475, 1971.

[2] Subrahmanyan Chandrasekhar. *Hydrodynamic and hydromagnetic stability*. Courier Corporation, 2013.

[3] Sourav Chatterjee and Persi Diaconis. Estimating and understanding exponential random graph models. 2013.

[4] Matteo Cucchi, Steven Abreu, Giuseppe Ciccone, Daniel Brunner, and Hans Kleemann. Hands-on reservoir computing: a tutorial for practical implementation. *Neuromorphic Computing and Engineering*, 2(3):032002, 2022.

[5] R. J. Doglas. The hippocampus and behavoir. *Psychological Bullentin*, 1967.

[6] Howard Eichenbaum. The role of the hippocampus in navigation is memory. *Journal of neurophysiology*, 117(4):1785–1796, 2017.

[7] Giovanny Espitia. Python guide: Building a reservoir computer from scratch, 2023. https://blog.stackademic.com/python-guide-building-a-reservoir-computer-from-scratch-c166d63038dc [Accessed: 20/03/2024].

[8] Leslie A. Fogwe, Vamsi Reddy, and Fassil B. Mesfin. Neuroanatomy, hippocampus. *StatPearls*, 2024. URL: https://www.ncbi.nlm.nih.gov/books/NBK482171/#:~:text=The%20hippocampus%20is%20a%20convex,the%20temporal%20lobe's%20medial%20surface.

[9] Loren M Frank, Garrett B Stanley, and Emery N Brown. Hippocampal plasticity across multiple days of exposure to novel environments. *Journal of Neuroscience*, 24(35):7681–7689, 2004.

[10] Henry Gray. *Anatomy of the Human Body*. Lea Febiger, 1918.

[11] Siegfried Grossmann and Detlef Lohse. Thermal convection for large prandtl numbers. *Physical review letters*, 86(15):3316, 2001.

[12] NV Gulyaeva. Ventral hippocampus, stress and psychopathology: Translational implications. *Neurochemical Journal*, 9(2):85–94, 2015.

[13] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploreing network structure, dynamics, and function using networkx. *Proceedings of the 7th Python in Science Conference (SciPy 2008)*, 2008.

[14] Alexander Haluszczynski, Jonas Aumeier, Joschka Herteux, and Christoph Räth. Reducing network size and improving prediction stability of reservoir computing. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 30(6), 2020.

[15] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi:10.1038/s41586-020-2649-2.

[16] John J Hopfield. Neurons with graded response have collective computational properties like those of two-state neurons. *Proceedings of the national academy of sciences*, 81(10):3088–3092, 1984.

[17] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi:10.1109/MCSE.2007.55.

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

[19] Herbert Jaeger. Echo state network. *scholarpedia*, 2007.

[20] Marcia K Johnson and Carol L Raye. Reality monitoring. *Psychological review*, 88(1):67, 1981.

[21] Gerd Kempermann, Hongjun Song, and Fred H Gage. Neurogenesis in the adult hippocampus. *Cold Spring Harbor perspectives in biology*, 7(9):a018812, 2015.

[22] Tien-Yien Li and James A Yorke. Period three implies chaos. *The theory of chaotic attractors*, pages 77–84, 2004.

[23] Edward N. Lorenz. Deterministic nonperiodic flow. *Journal of Atmospheric Sciences*, 20(2):130 – 141, 1963. URL: https://journals.ametsoc.org/view/journals/atsc/20/2/1520-0469_1963_020_0130_dnf_2_0_co_2.xml, doi:10.1175/1520-0469(1963)020<0130:DNF>2.0.CO;2.

[24] Wolfgang Maass. Liquid state machines: motivation, theory, and applications. *Computability in context: computation and logic in the real world*, pages 275–296, 2011.

[25] Hans J Markowitsch, D Yves von Cramon, and Uwe Schuri. Mnestic performance profile of a bilateral diencephalic infarct patient with preserved intelligence and severe amnesic disturbances. *Journal of Clinical and Experimental Neuropsychology*, 15(5):627–652, 1993.

[26] Cucchi Matteo, Gruener Christopher, Petrauskas Lautaro, Steiner Peter, Tseng Hsin, Fischer Axel, Penkovsky Bogdan, Matthus Christian, Birkholz Peter, Kleemann Hans, and Leo Karl. Reservoir computing with biocompatible organic electrochemical networks for brain-inspired biosignal classification. *Science Advances*, 18 Aug 2021.

[27] Mark P Mattson and SB Kater. Excitatory and inhibitory neurotransmitters in the generation and degeneration of hippocampal neuroarchitecture. *Brain research*, 478(2):337–348, 1989.

[28] Bruce S McEwen, Carla Nasca, and Jason D Gray. Stress effects on neuronal structure: hippocampus, amygdala, and prefrontal cortex. *Neuropsychopharmacology*, 41(1):3–23, 2016.

[29] May-Britt Moser, David C Rowland, and Edvard I Moser. Place cells, grid cells, and memory. *Cold Spring Harbor perspectives in biology*, 7(2):a021808, 2015.

[30] Simukayi Mutasa, Shawn Sun, and Richard Ha. Understanding artificial intelligence based radiology studies: What is overfitting? *Clinical imaging*, 65:96–99, 2020.

[31] Kohei Nakajima and Ingo Fischer. *Reservoir computing*. Springer, 2021.

[32] John O'keefe and Lynn Nadel. Précis of o'keefe & nadel's the hippocampus as a cognitive map. *Behavioral and Brain Sciences*, 2(4):487–494, 1979.

[33] S. Ortin, M. C. Soriano, L. Pesquera, D. Brunner, D. San-Martin, I. Fischer, C. R. Mirasso, and J. M. Gutierrez. A unified framework for reservoir computing and extreme learning machines based on a single time-delayed neuron. *Scientific Reports*, 2015.

[34] Stefano Recanatesi, Mikhail Katkov, Sandro Romani, and Misha Tsodyks. Neural network model of memory retrieval. *Frontiers in Computational Neuroscience*, 9, 2015. URL: https://www.frontiersin.org/articles/10.3389/fncom.2015.00149, doi:10.3389/fncom.2015.00149.

[35] Henry L Roediger, Mary Susan Weldon, and Bradford H Challis. Explaining dissociations between implicit and explicit measures of retention: A processing account. In *Varieties of memory and consciousness*, pages 3–41. Psychology Press, 2014.

[36] OE5889510996 Rossler. An equation for hyperchaos. *Physics Letters A*, 71(2-3):155–157, 1979.

[37] Otto E Rössler. An equation for continuous chaos. *Physics Letters A*, 57(5):397–398, 1976.

[38] Daniel L Schacter and Endel Tulving. What are the memory systems of 1994? *Memory Systems*, 1994:424, 1994.

[39] Benjamin Schrauwen, David Verstraeten, and Jan 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*, pages 471–482, 2007.

[40] Gouhei Tanaka, Toshiyuki Yamane, Jean Benoit Héroux, Ryosho Nakane, Naoki Kanazawa, Seiji Takeda, Hidetoshi Numata, Daiju Nakano, and Akira Hirose. Recent advances in physical reservoir computing: A review. *Neural Networks*, 115:100–123, 2019. URL: https://www.sciencedirect.com/science/article/pii/S0893608019300784, doi:10.1016/j.neunet.2019.03.005.

[41] Endel Tulving and Hans J Markowitsch. Episodic and declarative memory: role of the hippocampus. *Hippocampus*, 8(3):198–204, 1998.

[42] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[43] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi:10.1038/s41592-019-0686-2.

[44] VP Whittaker. What is dale's principle? In *Dale's Principle and Communication Between Neurones*, pages 1–5. Elsevier, 1983.

[45] Emma R Wood, Paul A Dudchenko, R Jonathan Robitsek, and Howard Eichenbaum. Hippocampal neurons encode information about different types of memory episodes occurring in the same location. *Neuron*, 27(3):623–633, 2000.

Appendix A

AI Prompts

No.	Reasons	Link
1.	How to make latex show figures side by side?	https://chat.openai.com/share/05ebe048-8801-4d1b-b8b0-3de3b0671342
2.	How to adjust table size in latex?	https://chat.openai.com/share/d8ce478b-c1e6-461a-a3f3-7b7d90032a16
3.	Mark on the graph when the 2 datasets differentiate from each other.	https://chat.openai.com/share/3d32272f-da43-4fb2-bc9c-059175831368
4.	Swap the format of the first graph into the second graph.	https://chat.openai.com/share/883d7ea0-cd31-470e-91e1-62569e9e0460
5.	How to make the header 3 blocks wide?	https://chat.openai.com/share/b4a63cfa-5e40-48e7-90ba-8c568e7987a3
6.	How to write python codes in latex	https://chat.openai.com/share/19eca045-a1a6-4d4a-8788-c10aedf273eb

Table A.1: Table containing all the AI and ChatGPT usages.

List of prompts/references that were used in the project and/or dissertation with an Artificial Intelligence model. This can include large language models (LLMs) such as ChatGPT or AI translation tools such as DeepL.

Appendix B

All Codes and Programs

All the codes and .csv files are stored on Github: https://github.com/Max-Prasertsan/Hippo_RC_project

Attached to the submission is a zip file in case of any inspection.

Appendix C

Extra Results

Average MSE for each axis on V1		
x	y	z
0.7838906928979775	0.8303344592919216	0.5913539531044452
Average MSE for each axis on V2		
x	y	z
0.7569467923982538	0.8021984746680804	0.5136751094765686
Average MSE for each axis on V3		
x	y	z
0.7615089715576009	0.8146714814902338	0.6667903933254647
Average MSE for each axis on V4		
x	y	z
0.7710937245657441	0.8234214197933531	0.6606133204822912
Average MSE for each axis on V5		
x	y	z
0.7882221551973205	0.8451149506001242	0.7005950309422282
Average MSE for each axis on V6		
x	y	z
0.6983862261138936	0.7632720017114387	0.6046923917431113
Average MSE for each axis on V7		
x	y	z
0.825699189075253	0.8690164963327422	0.5038434705255233
Average MSE for each axis on V8		
x	y	z
1.059614682076358	1.0940226218327784	0.3938960790548923
Average MSE for each axis on V9		
x	y	z
0.9545172551205885	0.9879899935281988	0.4377917611306932
Average MSE for each axis on V10		
x	y	z
0.8046159365619381	0.8488968998836317	0.3965913803151179
Average MSE for each axis on V11		
x	y	z
0.778601251083591	0.8124841532442659	0.4701651815241074
Average MSE for each axis on V12		
x	y	z
0.653226484843262	0.6938091227496203	0.46537092920295964

Table C.1: Table showing the average MSE values for each axis on every iteration of the hippocampus-based RC.

Appendix D

Graphs of iteration 1

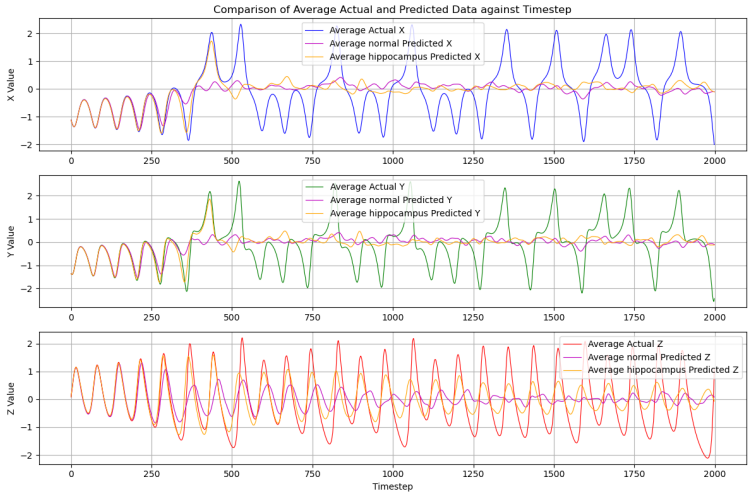

Figure D.1: Iteration 1: Comparison between average 50 runs of normal RC and hippocampus-based RC.

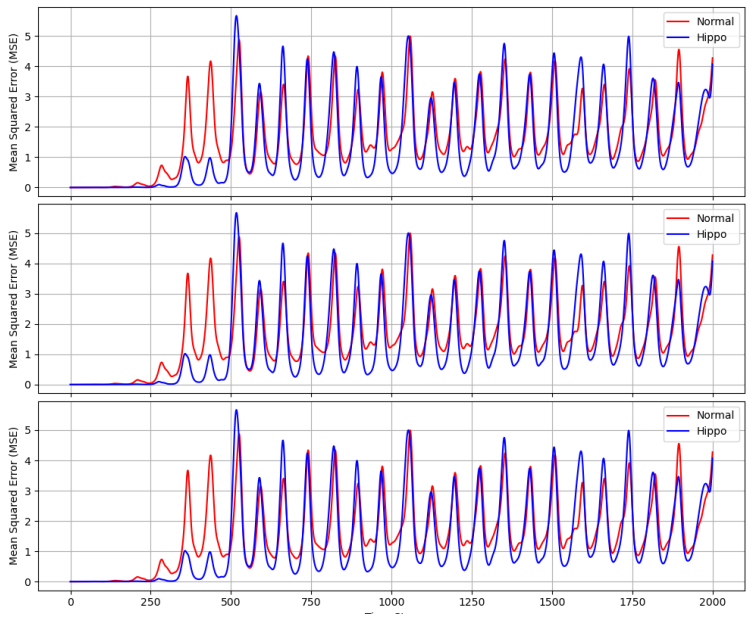

Figure D.2: Iteration 1: Comparison between the average of 50 runs of normal and hippocampus-based RC MSE values against the actual Lorenz-63 values.

Appendix E

Graphs of iteration 2

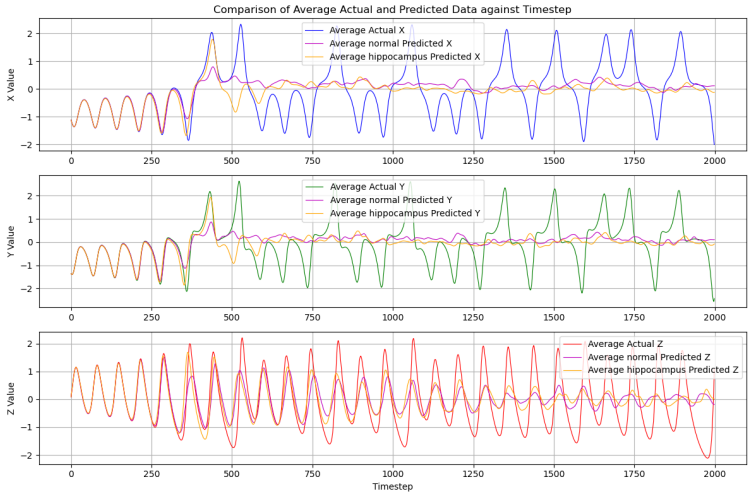

Figure E.1: Iteration 2: Comparison between average 50 runs of normal RC and hippocampus-based RC.

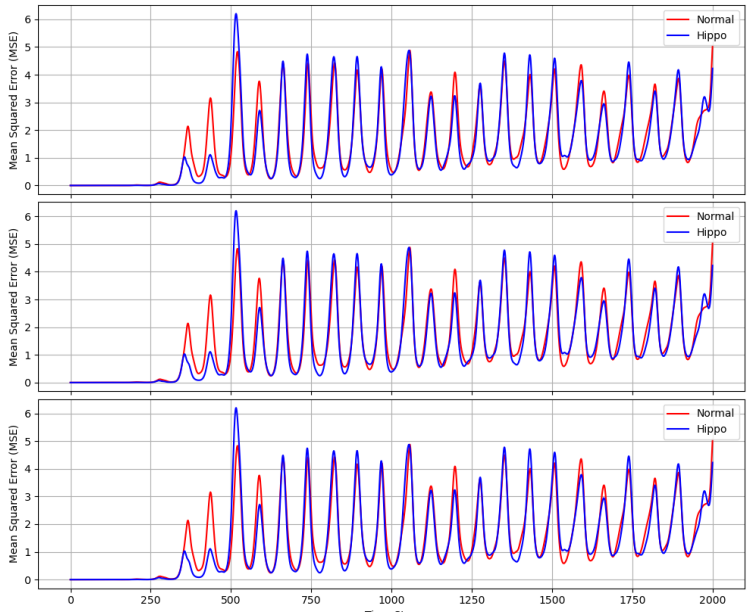

Figure E.2: Iteration 2: Comparison between the average of 50 runs of normal and hippocampus-based RC MSE values against the actual Lorenz-63 values.

Appendix F

Graphs of iteration 3

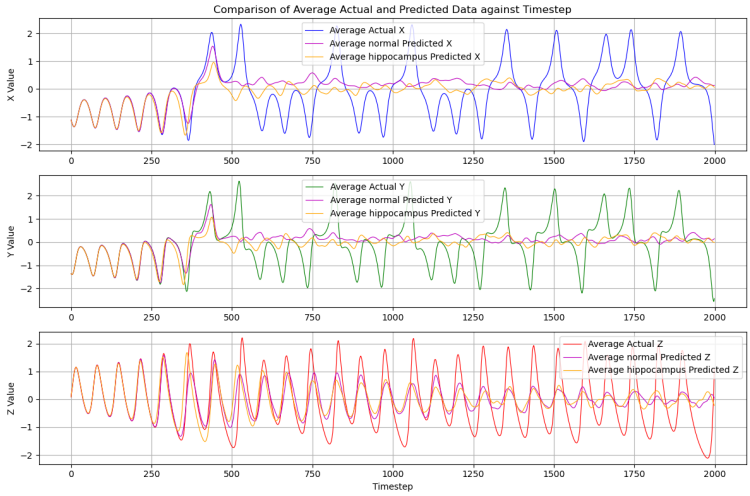

Figure F.1: Iteration 3: Comparison between average 50 runs of normal RC and hippocampus-based RC.

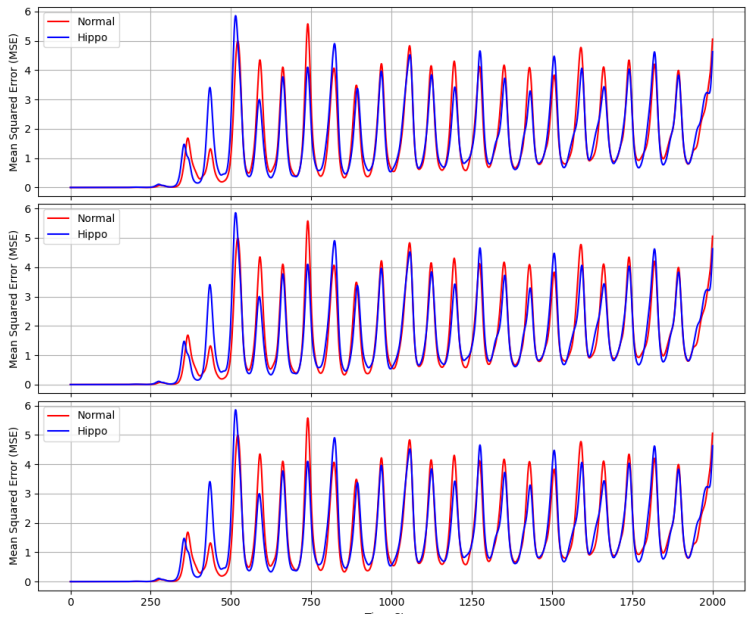

Figure F.2: Iteration 3: Comparison between the average of 50 runs of normal and hippocampus-based RC MSE values against the actual Lorenz-63 values.

Appendix G

Graphs of iteration 4

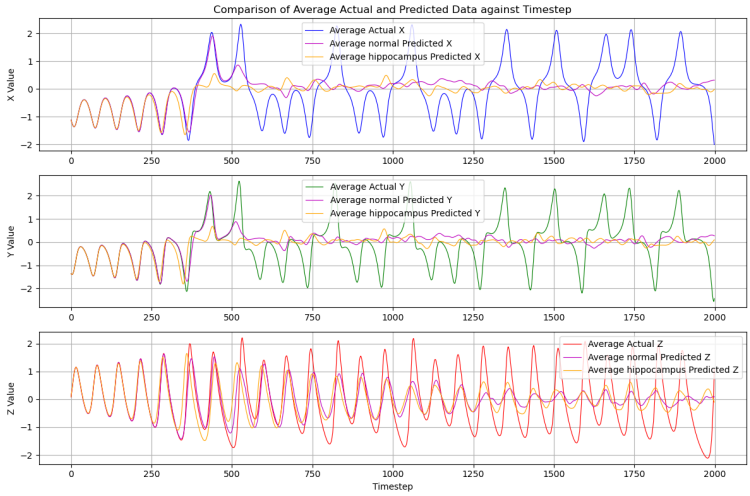

Figure G.1: Iteration 4: Comparison between average 50 runs of normal RC and hippocampus-based RC.

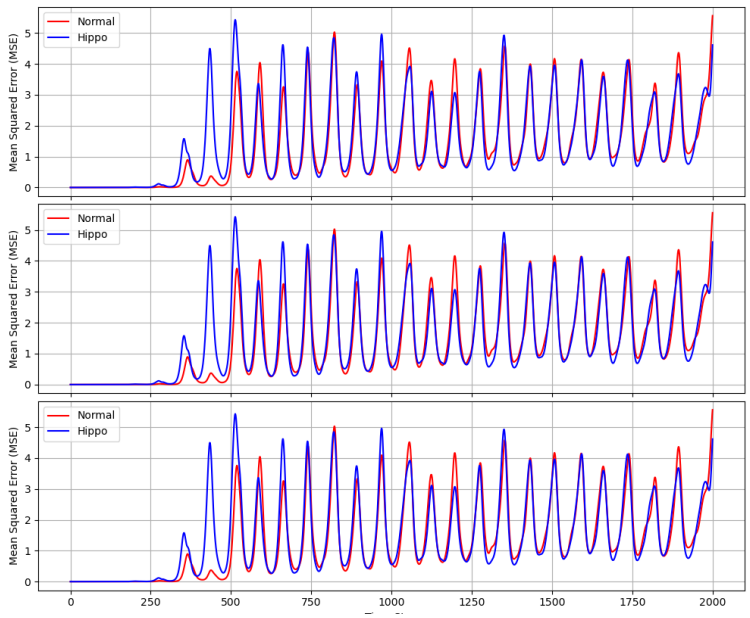

Figure G.2: Iteration 4: Comparison between the average of 50 runs of normal and hippocampus-based RC MSE values against the actual Lorenz-63 values.

Appendix H

Graphs of iteration 5

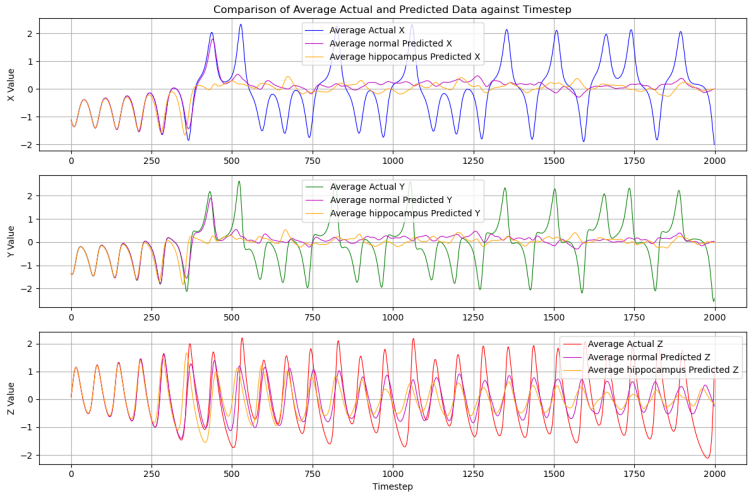

Figure H.1: Iteration 5: Comparison between average 50 runs of normal RC and hippocampus-based RC.

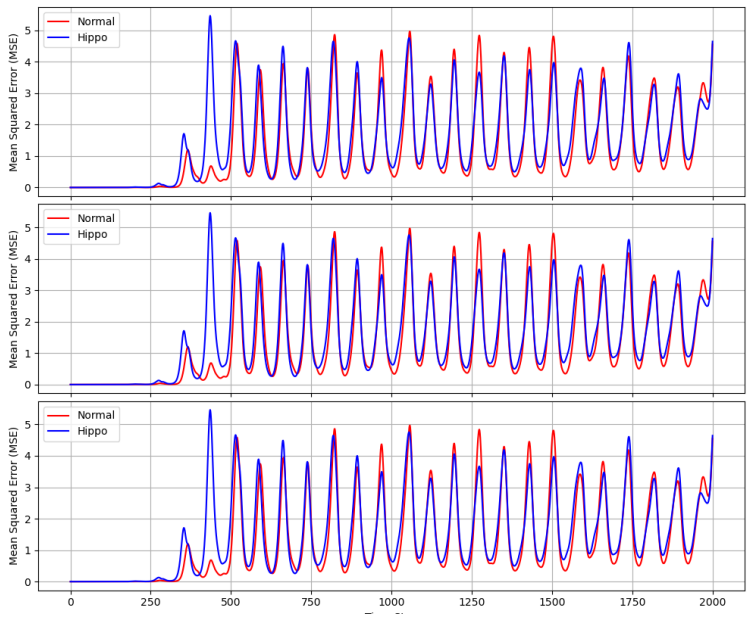

Figure H.2: Iteration 5: Comparison between the average of 50 runs of normal and hippocampus-based RC MSE values against the actual Lorenz-63 values.

Appendix I

Graphs of iteration 6

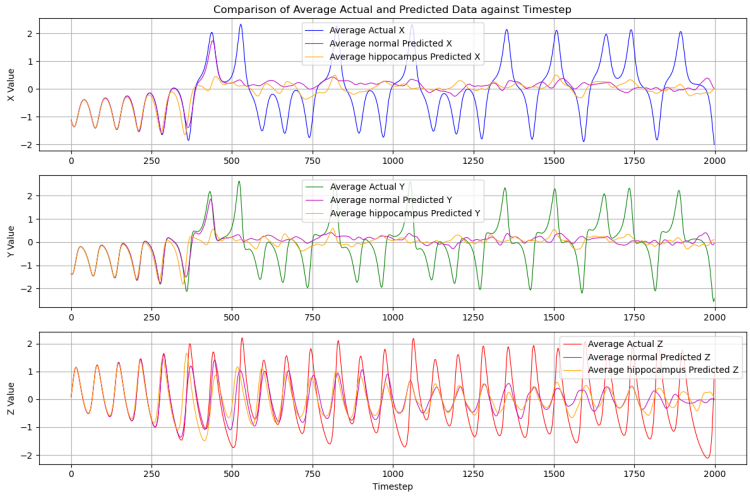

Figure I.1: Iteration 6: Comparison between average 50 runs of normal RC and hippocampus-based RC.

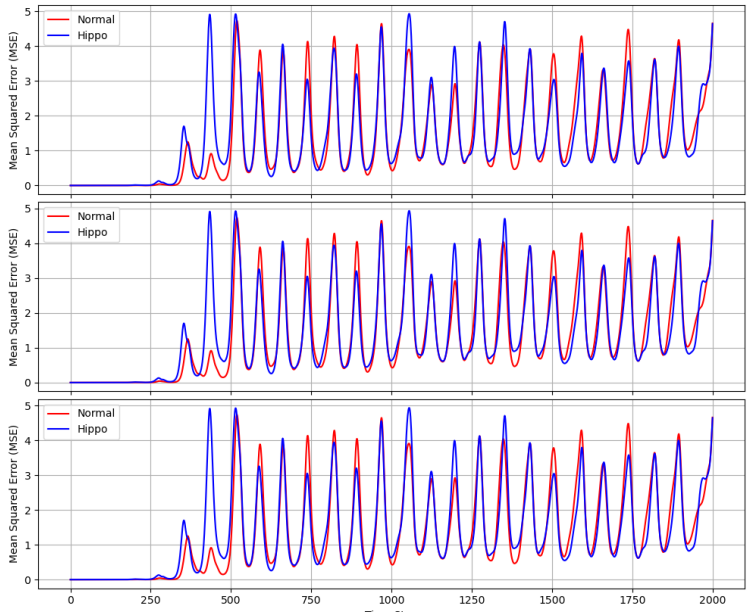

Figure I.2: Iteration 6: Comparison between the average of 50 runs of normal and hippocampus-based RC MSE values against the actual Lorenz-63 values.

Appendix J

Graphs of iteration 7

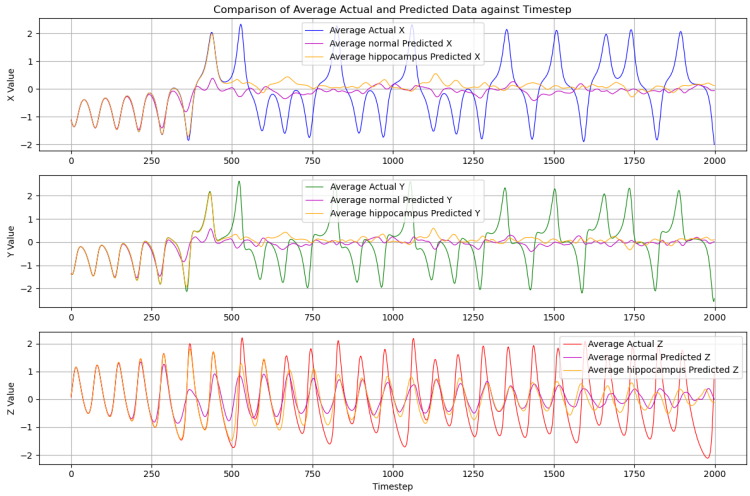

Figure J.1: Iteration 7: Comparison between average 50 runs of normal RC and hippocampus-based RC.

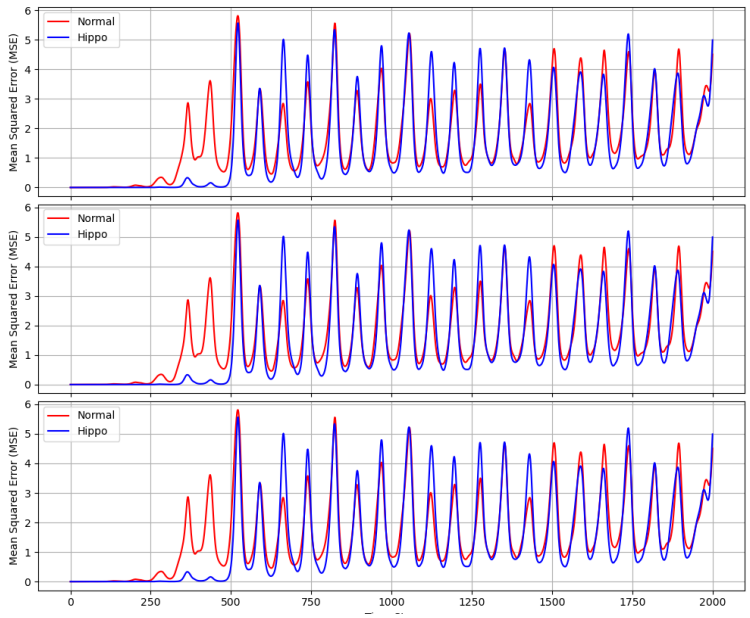

Figure J.2: Iteration 7: Comparison between the average of 50 runs of normal and hippocampus-based RC MSE values against the actual Lorenz-63 values.

Appendix K

Graphs of iteration 8

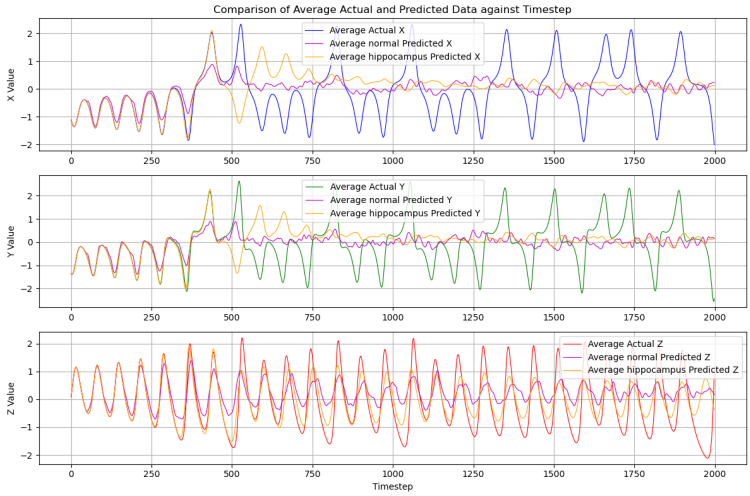

Figure K.1: Iteration 8: Comparison between average 50 runs of normal RC and hippocampus-based RC.

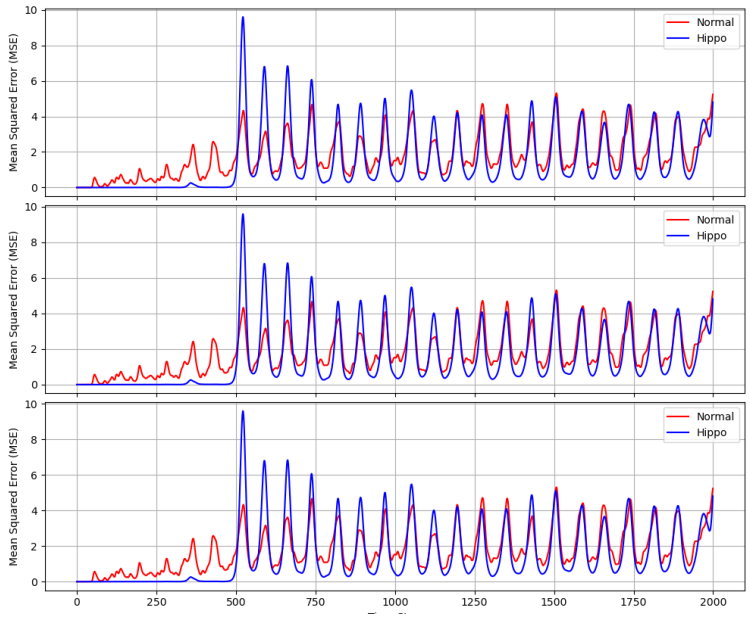

Figure K.2: Iteration 8: Comparison between the average of 50 runs of normal and hippocampus-based RC MSE values against the actual Lorenz-63 values.

Appendix L

Graphs of iteration 9

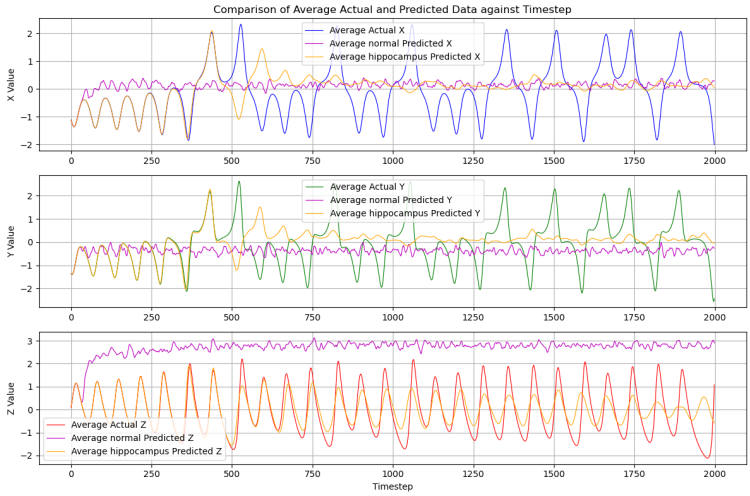

Figure L.1: Iteration 9: Comparison between average 50 runs of normal RC and hippocampus-based RC.

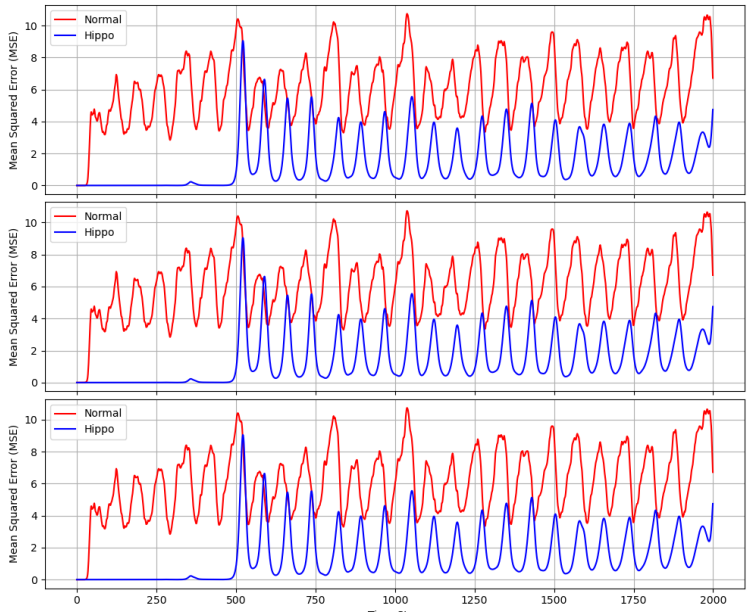

Figure L.2: Iteration 9: Comparison between the average of 50 runs of normal and hippocampus-based RC MSE values against the actual Lorenz-63 values.

Appendix M

Graphs of iteration 10

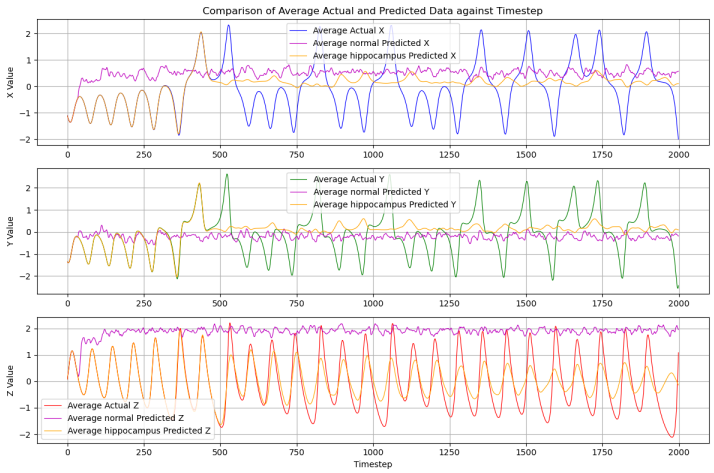

Figure M.1: Iteration 10: Comparison between average 50 runs of normal RC and hippocampus-based RC.

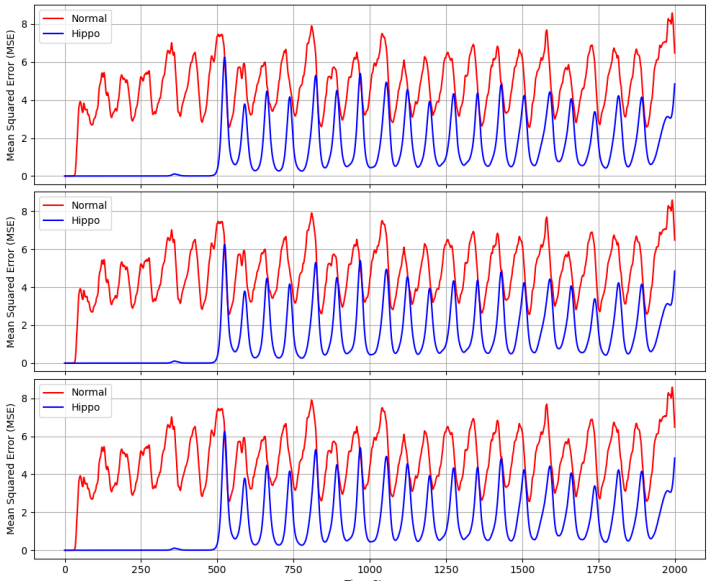

Figure M.2: Iteration 10: Comparison between the average of 50 runs of normal and hippocampus-based RC MSE values against the actual Lorenz-63 values.

Appendix N

Graphs of iteration 11

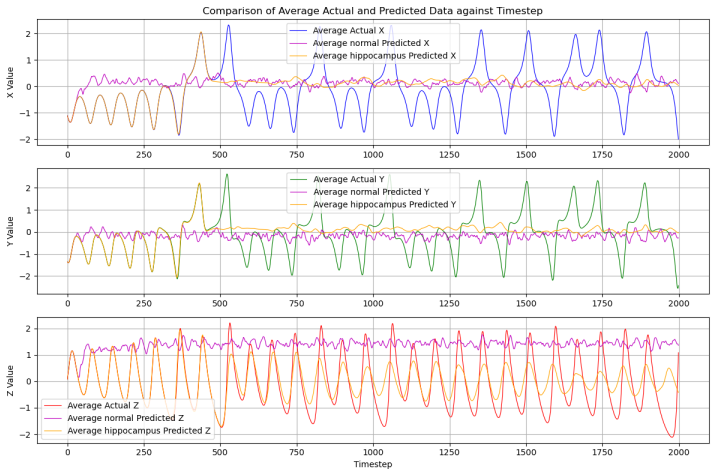

Figure N.1: Iteration 11: Comparison between average 50 runs of normal RC and hippocampus-based RC.

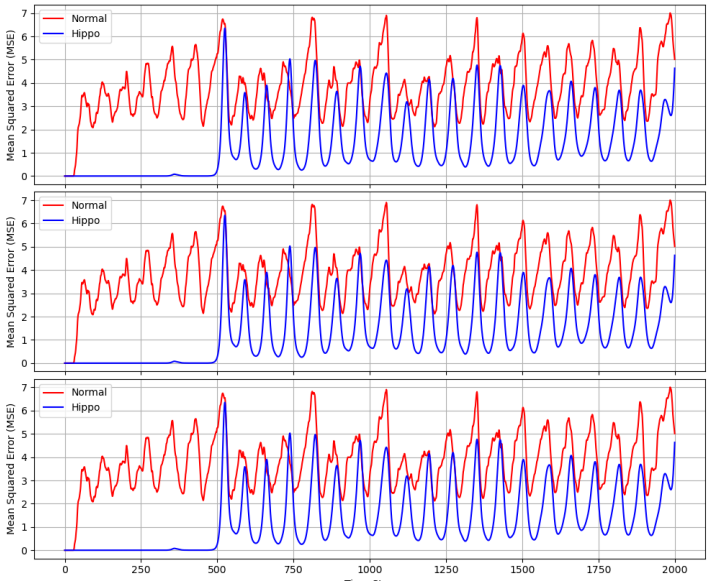

Figure N.2: Iteration 11: Comparison between the average of 50 runs of normal and hippocampus-based RC MSE values against the actual Lorenz-63 values.

Appendix O

Graphs of iteration 12

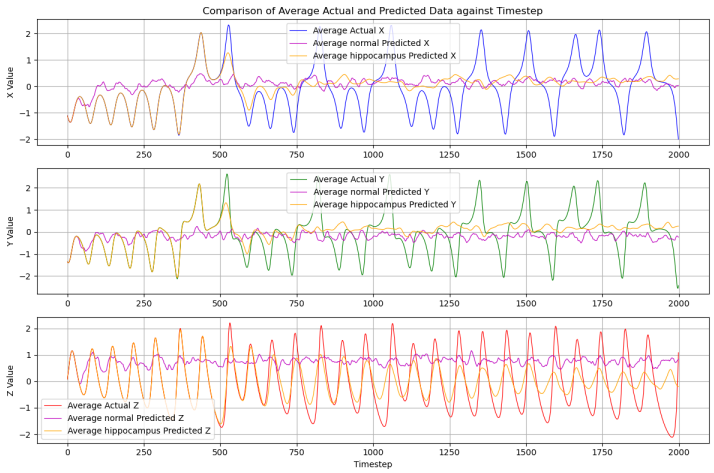

Figure O.1: Iteration 12: Comparison between average 50 runs of normal RC and hippocampus-based RC.

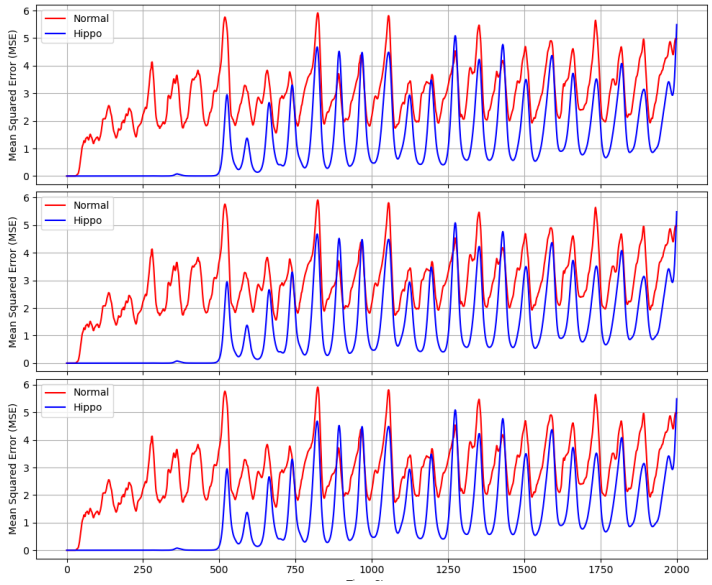

Figure O.2: Iteration 12: Comparison between the average of 50 runs of normal and hippocampus-based RC MSE values against the actual Lorenz-63 values.

Appendix P

Graphs of iteration 13

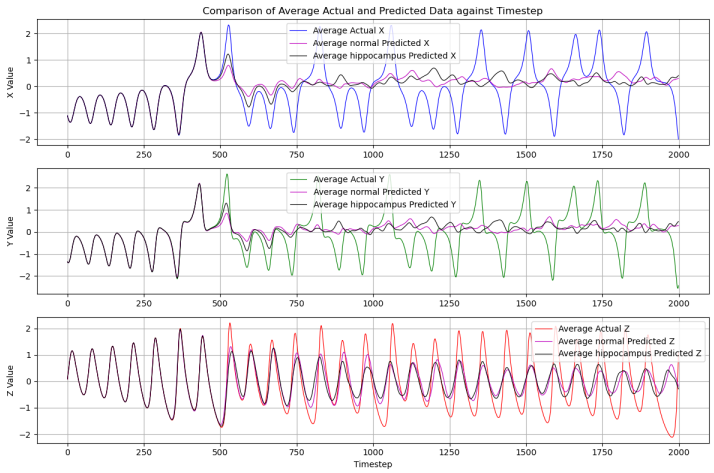

Figure P.1: Iteration 13: Comparison between average 50 runs of normal RC and hippocampus-based RC.

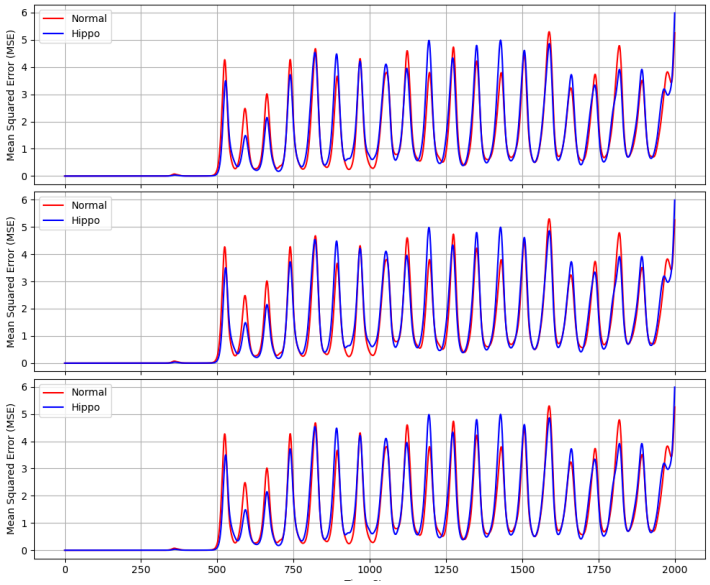

Figure P.2: Iteration 13: Comparison between the average of 50 runs of normal and hippocampus-based RC MSE values against the actual Lorenz-63 values.

Appendix Q

Graphs of iteration 14

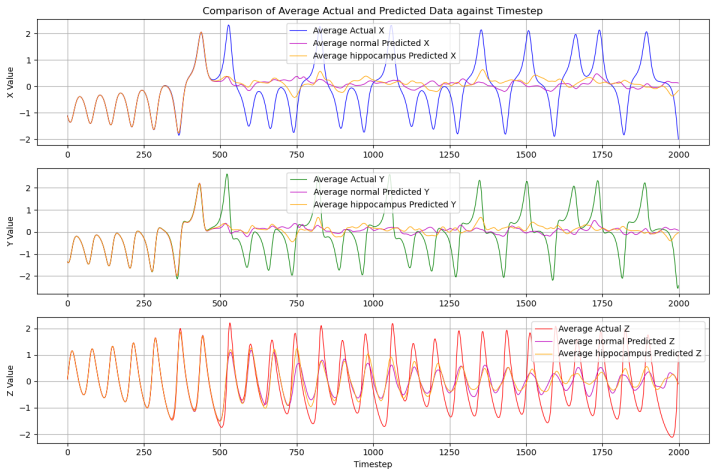

Figure Q.1: Iteration 14: Comparison between average 50 runs of normal RC and hippocampus-based RC.

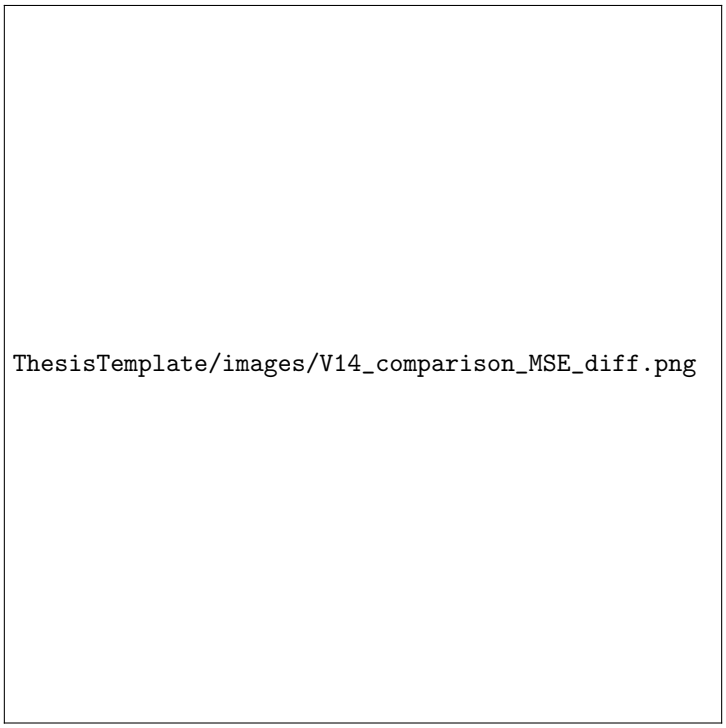

Figure Q.2: Iteration 14: Comparison between the average of 50 runs of normal and hippocampus-based RC MSE values against the actual Lorenz-63 values.