Winter 2-23-2018

Hierarchical Random Boolean Network Reservoirs

Sai Kiran Cherupally
Portland State University

Hierarchical Random Boolean Network Reservoirs

by

Sai Kiran Cherupally

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in
Electrical and Computer Engineering

Thesis Committee:
Christof Teuscher, Chair
Mark Faust
Marek Perkowski

Portland State University
2018

Abstract

Reservoir Computing (RC) is an emerging Machine Learning (ML) paradigm. RC systems contain randomly assembled computing devices and can be trained to solve complex temporal tasks. These systems are computationally cheaper to train than other ML paradigms such as recurrent neural networks, and they can also be trained to solve multiple tasks simultaneously [13]. Further, hierarchical RC systems with fixed topologies, were shown to outperform monolithic RC systems by up to 40% when solving temporal tasks [3, 24]. Although the performance of monolithic RC networks was shown to improve with increasing network size, building large monolithic networks may be challenging, for example because of signal attenuation .

In this research, larger hierarchical RC systems were built using a network generation algorithm. The benefits of these systems are presented by evaluating their accuracy in solving three temporal problems: pattern detection, food foraging, and memory recall. This work also demonstrates the functionality of random Boolean networks being used as reservoirs. Networks with up to $5,000$ neurons were used to train 200 sequences from memory and to identify X or O patterns temporally. Also, a Genetic Algorithm (GA) was used to train different types of hierarchical RC networks, to find optimal solutions for food-foraging tasks.

This research shows that about 80% of the possible different hierarchical configurations of RC systems can outperform monolithic RC systems by up to 60% while solving complex temporal tasks. These results suggest that hierarchical random Boolean network RC systems can be used to solve temporal tasks, instead of building large monolithic RC systems.

Acknowledgements

I must take this opportunity to thank Prof. Christof Teuscher at Portland State University for his guidance and mentoring which helped me transform my research aspirations into a Master's Thesis. He has been a constant rock of support, and kept me focused and motivated throughout the period of this research. I might not have completed this work without his inspiring research methodology and approach to problem solving. The student group at Teuscher Lab has also been very supportive, and I am ever grateful to Mohammed, Walt, Wesley and Dat in particular for responding promptly whenever I had any questions.

I also thank Prof. Mark Faust at PSU for his guidance and support that motivated me to pursue this thesis. I thank Prof. Marek Perkowski for being on my thesis committee and bringing his enthusiasm and inspiring personality to it.

Last but not the least, the other source of constant support was my family. Hence, this thesis work is dedicated to my father, Ch Ramesh, my mother, Ch Madhavi, my two lovely sisters and the rest of my family.

Contents

List of Tables

List of Figures

1

Introduction

1.1 Outcomes

The two main outcomes of this research are:

1. hierarchical reservoir computing systems can outperform monolithic reservoirs on solving complex temporal tasks; and

2. random Boolean networks can be used as reservoirs in hierarchical RC systems.

1.2 Motivation

Machine Learning (ML) is the science of teaching computers to perform tasks without explicitly programming them to do so. For example, teaching a computer to accurately predict the amount of rainfall in the year 2017 using data corresponding to the amount of rainfall in the last 200 years, can be formulated as a machine learning task. Reservoir Computing (RC) has emerged as a new ML framework over the last ten years [27]. RC systems exploit the computational capability of a network of randomly assembled computing devices to perform tasks.

Maass *et al.* [21] and Jaeger *et al.* [13] introduced RC and demonstrated the advantages of this framework in 2002 and 2004 respectively, and many other research groups showed that this approach is particularly well-suited for solving temporal tasks such as speech recognition [30], Non-linear Auto-regressive Moving Average

(NARMA) [3], waveform generation [3], and handwriting recognition [31]. However, most of the research related to RC systems has been limited to randomly assembled networks that do not have any network topology and modularity.

While Triefenbach *et al.* [30] demonstrated that the performance of RC systems increases when the network size is increased from $1,000$ to $20,000$ neurons, Burger *et al.* [3] and Rodriguez *et al.* [24] showed that the performance of RC systems can also be improved, by up to 40% using networks with a ring topology and a community structure. RC systems with hierarchically structured networks can be used as an alternative to large monolithic RC systems to solving complex tasks because signal attenuation issues may arise when fabricating large monolithic networks. However, the works of Burger *et al.* and Rodriguez *et al.* used networks with fewer than $1,000$ neurons in their studies.

My research draws inspiration from these two approaches, but I used networks with more than $1,000$ neurons to solve complex temporal tasks. In addition, a more generic algorithm was used to build hierarchical networks with more than $1,000$ neurons, and these reservoirs were built with Random Boolean Networks (RBNs).

Snyder *et al.* [28] showed that monolithic Random Boolean Networks (RBNs) can be used as reservoir computing systems to solve complex temporal tasks. These networks, made up of randomly interconnected Boolean logic gates, are easier to fabricate compared to RC networks containing more complex types of neurons, such as memristors and spiking neurons. Although almost any randomly assembled computing medium can be used as a reservoir [13,27], I chose to use RBNs because they are simple to model, simulate, and fabricate.

Artificial Neural Networks (ANNs) are a commonly used framework to solve ML

tasks, such as visual recognition [19], speech processing [32], robot control [7], and prediction [36]. ANNs are able to process information in parallel and eliminate the need for having separate memory and computing units, as is the case in standard Von-Neumann architectures. This leads to the elimination of the latency between processing and memory units.

The inspiration behind ANNs is the massive networks of billions of neurons in our brains. Neurons in the brain process stimuli from various sensory organs and produce output signals that control how a person reacts to the stimuli. For example, when a child is learning to walk, she falls many times, but eventually learns to walk without falling. The neurons in her brain learn to produce signals that help her achieve a more stable position every time she falls. Similarly, a neural network can be trained to control the motors responsible for the motion of a humanoid robot, so that it is able to learn to walk without falling.

Feed-forward neural networks [2], *Recurrent Neural Networks* (RNNs) [8], and *Reservoir Computing* systems [13, 21] are three different architectures of ANNs. A Feed-forward neural network is an ANN that does not have any feed-back connections between the neurons. Such networks map a set of inputs to a set of outputs, a function that is independent of time. Hence, they do not possess any memory to store the history of their inputs. For example, they can detect whether an input represents an "X" or an "O" pattern, but they cannot detect whether an input sequence contains a series of two continuous "X" patterns. The first kind of task is called a *non-temporal task*, and the latter is called a *temporal task*.

Recurrent Neural Networks (RNNs) are a family of ANNs that are able to solve temporal tasks. These networks contain recurrent connections, i.e., feed-back connections between some of the neurons. This results in a dynamic behavior in

Feature	Feed-forward neural network	Recurrrent neural network	Reservoir computing
Memory	No	Yes	Yes
Feedback connections	No	Yes	Yes
Temporal tasks	No	Yes	**Yes**
Learning complexity	Medium	High	**Low**
Multitasking	No	No	**Yes**

Table 1.1: A summary of the features of various ANN families. RC approach is suitable to build ML systems with low training complexity, and any medium capable of performing computations can be used as a reservoir in RC systems.

the network, where the outputs of the neurons depend on the history of the inputs. These networks are able to solve temporal tasks such as speech recognition [8].

Feed-forward neural networks and RNNs are commonly trained by using the back propagation algorithm [26, 34]. In both of these architectures of ANNs, all the weights present in the network are trained [11, 33, 34]. This leads to a high learning complexity. RNNs are trained to perform temporal tasks commonly by unrolling them in time and using the Back Propagation Through Time (BPTT) algorithm. This results in a deep layered network, and increases the number of weights to be trained. The problem of vanishing gradients makes it difficult to train RNNs [10, 23].

The RC approach overcomes these issues by separating computation and training. In RC systems, the computations are completely delegated to a reservoir, a common implementation of which is an RNN with randomly assembled neurons. A separate training layer, called the read-out layer, is used for training. The read-out layer contains read-out neurons that connect to the outputs of the neurons in the reservoir. The neurons in the read-out layer are trained to produce a desired output by using linear regression [13, 21]. This simplifies the training process because there are fewer weights to be trained, compared to RNNs. Another advantage

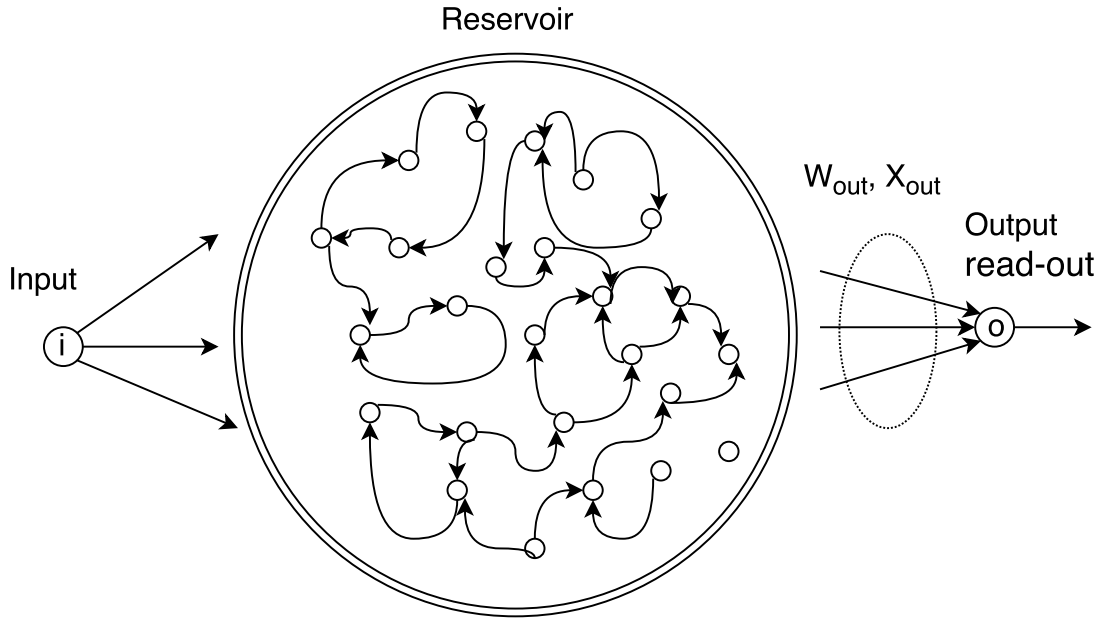


Figure 1.1: A monolithic reservoir network without any hierarchies and modular structure. These networks cannot be decomposed into smaller groups of neurons due to a lack of community structure.

of this approach is that the same reservoir can be used by two different read-out neurons which can be trained to perform two different ML tasks simultaneously. Hence, this approach allows one to build ML systems that are able to multi-task. A summary of the features of feed-forward neural networks, RNNs, and RC systems are presented in Table 1.1.

An RC system with, a flat hierarchy and no notion of modules or communities, is called a monolithic RC system. A monolithic RC system is shown in Fig. 1.1. In this figure, W_{out} corresponds to the weights of links connecting X_{out} neurons to the read-out neuron o. Such systems have been used to solve temporal problems [14, 15, 31].

An open issue that is being actively investigated is whether hierarchical and modular RC systems can be an alternative to large monolithic networks. In these

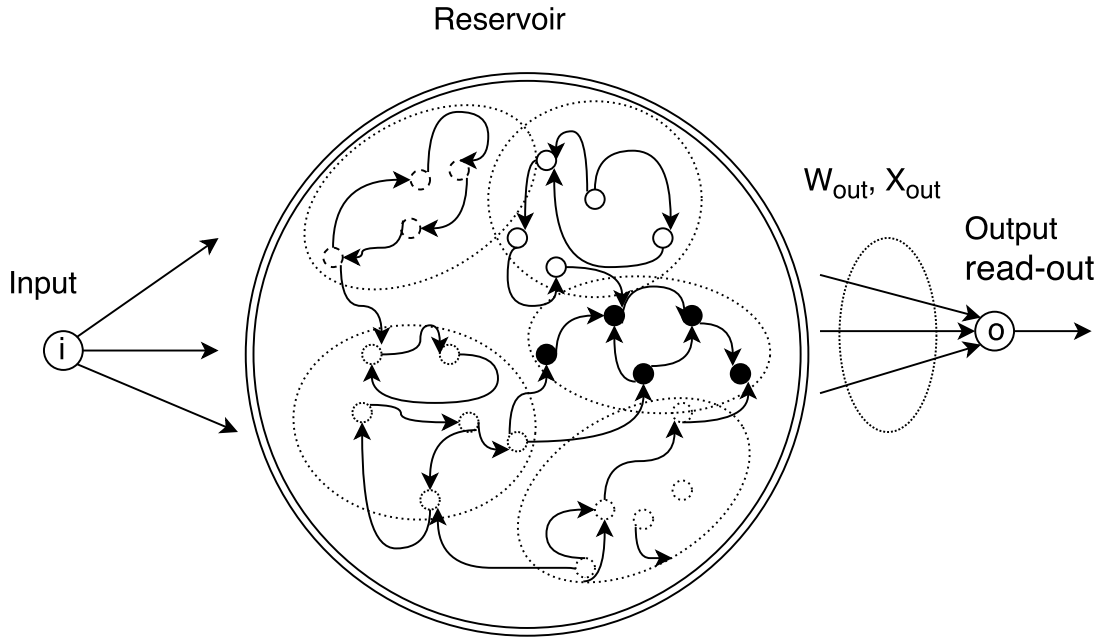


Figure 1.2: A schematic of a hierarchical reservoir network with two hierarchical levels and five modules at the lowest level. The Neurons are distributed into multiple small communities.

networks, neurons are distributed into small communities, called modules. A network topology parameter, called as *modularity* can be used to quantify the amount of modularity present in such networks. The modularity (μ) of a network is defined as the ratio of the number of connections existing between different modules to the total number of connections in the network. A simple hierarchical reservoir network with two levels of hierarchy and five modules at the lowest level of hierarchy is shown in Fig. 1.2. In this figure, W_{out} corresponds to the weights of links connecting X_{out} neurons to the read-out neuron o.

Burger *et al.* [3] showed that hierarchical networks can solve more complex tasks than their monolithic counterparts with the same network sizes. Their work uses a ring topology and their largest network had about 400 neurons. Their results show that hierarchical RC systems were at least 20% better than equivalent monolithic

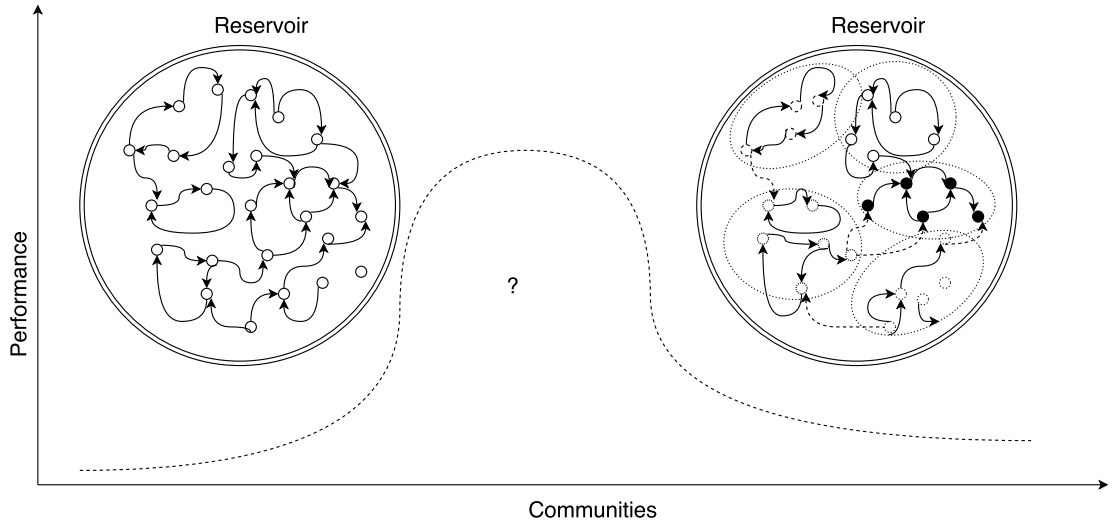


Figure 1.3: The role of communities and hierarchies on the task performance of RC systems was investigated in this research.

networks at solving waveform generation tasks. Their hierarchical networks are also able to find a solution for the NARMA-10 task while the equivalent monolithic networks are not able to solve the task.

On the other hand, Rodriguez *et al.* [24] investigated the role of network modularity, i.e., the ratio of global connections between communities of neurons and connections within communities. Their work shows that having a modularity of 0.1 to 0.5 enhances the memory capacity of the reservoir. The largest network they used has $1,000$ sigmoidal neurons organized into a different number of modules. Burger *et al.* used memristive neurons, and Rodriguez *et al.* used sigmoidal neurons in their respective investigations, but neither of these works have used Boolean logic gates in their reservoirs.

Hence, in this thesis, the following were investigated:

- hierarchies and modularity of networks with more than $1,000$ neurons;
- the use of RBNs as reservoirs; and

- the task-solving performance of such systems.

1.3 Significance

We took a different approach to building hierarchical RC systems as opposed to the simple ring topology and fixed communities topology used by Burger *et al.* [3] and Rodriguez *et al.* [24] respectively. In this research, an algorithm that uses network formation principles inspired from real-world networks was used to build large hierarchical RC systems. We wanted to evaluate and compare the variation of task performance of monolithic reservoirs with that of different hierarchical reservoir networks. These hierarchical systems were differentiated based on the number of communities they had, as shown in Fig. 1.3.

This thesis builds on results obtained by Burger *et al.* [3] and Rodriguez *et al.* [24] and expands them to networks with more than $1,000$ neurons. These two studies have used memristors and sigmoidal neurons respectively in their reservoirs. Boolean logic gates with randomly determined transfer functions have been used as neurons in monolithic RC systems [28], but not for hierarchical networks..

The results of this research suggest that the issues related to building large monolithic networks can be addressed by using hierarchical networks. This research also demonstrates the use of RBNs as RC systems for solving complex temporal tasks.

Table 1.2 shows a comparison of the features of my work, and two other related works.

1.4 Approach

This research was divided into three main tasks:

	Burger *et al.* [3]	**Rodriguez *et al.*** [24]	**My work**
Max network size	400	$1,000$	$5,000$
Topology	Ring	Modular	Hierarchical and modular
Temporal tasks	Yes	Yes	Yes
Control tasks	No	No	Yes
Neuron type	Memristive	Sigmoidal	Boolean gates

Table 1.2: A comparison of the networks used and tasks evaluated in my work and two other related works. A control task was also evaluated in my research in addition to other temporal tasks.

1. Large hierarchical RBNs with more than $1,000$ neurons were modeled using a network growth algorithm [35]. This task is explained in detail in Section 3.3 of Chapter 3.

2. Various experiments were performed to understand the role of the network parameters as follows:

   - The effect of in-module connection probabilities on the average connectivity of the RBNs (Section 3.3 of Chapter 3).
   - The effect of hierarchical levels and number of modules on average connectivity (Section 4.1 of Chapter 4).
   - The effect of hierarchical levels and number of modules on modularity (Section 4.2 of Chapter 4).
   - The percentage of reservoir neurons to be perturbed with an input signal, such that there is optimal activity in the reservoir (Section 4.3 of Chapter 4).

3. The task performance of the proposed hierarchical RBNs was measured using three temporal tasks:

- Pattern detection (Section 4.4 of Chapter 4);
- Food foraging (Section 4.5 of Chapter 4); and
- Memory recall (Section 4.6 of Chapter 4).

1.5 My contributions

- A Python simulation framework was built to simulate and evaluate the proposed networks.
- Hierarchical RBNs with up to $10,000$ neurons were built and simulated.
- Hierarchical RBNs with 1 to 6 levels were evaluated for temporal pattern recognition, food foraging, and memory recall tasks.
- It was found that perturbing up to 20% of the reservoir neurons is sufficient to result in a near-optimal activity in the RBNs.
- The results obtained in this research suggest that using up to 10% of the neurons as output neurons is sufficient to accurately train the RC systems.
- A genetic algorithm module was built to train the read-out layers of the reservoirs. This module uses the Python DEAP toolbox [6].
- The amount of hierarchy and modularity required for producing optimal results has been quantified by using multiple tasks.
- The results of this research suggest that hierarchical networks can be an alternative to large monolithic networks.

2

Background

This research combines hierarchical reservoir computing and random Boolean networks, and provides a model to build and simulate such large hierarchical random Boolean networks with more than $1,000$ neurons. In order to understand the motivation behind this research and its significance, it is useful to know about:

- artificial neurons;
- commonly used artificial neural network frameworks and their functionality, advantages, and drawbacks;
- the reservoir computing approach;
- advantages of the hierarchical approach to reservoir computing; and
- functionality of random Boolean networks.

As discussed in Chapter 1, ANNs are a commonly used framework to perform various machine learning tasks such as image recognition, speech recognition, prediction, and classification. These networks contain interconnected computing devices, called as artificial neurons, that perform transformations. The networks are trained to produce desired outputs by modifying the network weights. In order to understand the function of ANNs, it is important to understand two aspects:

1. how an artificial neuron performs computations; and
2. how information is exchanged inside the network.

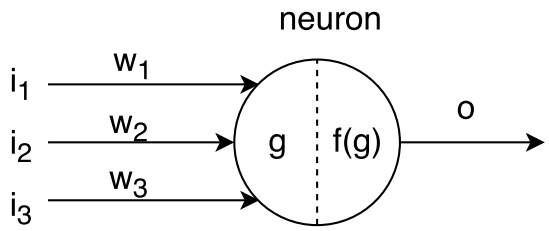


Figure 2.1: Example of a neuron with three inputs and one output. This neuron linearly combines its three inputs with three weights to produce one output.

In the following section, a working example of an artificial neuron and its weights is described.

2.1 Artificial Neurons and Weights

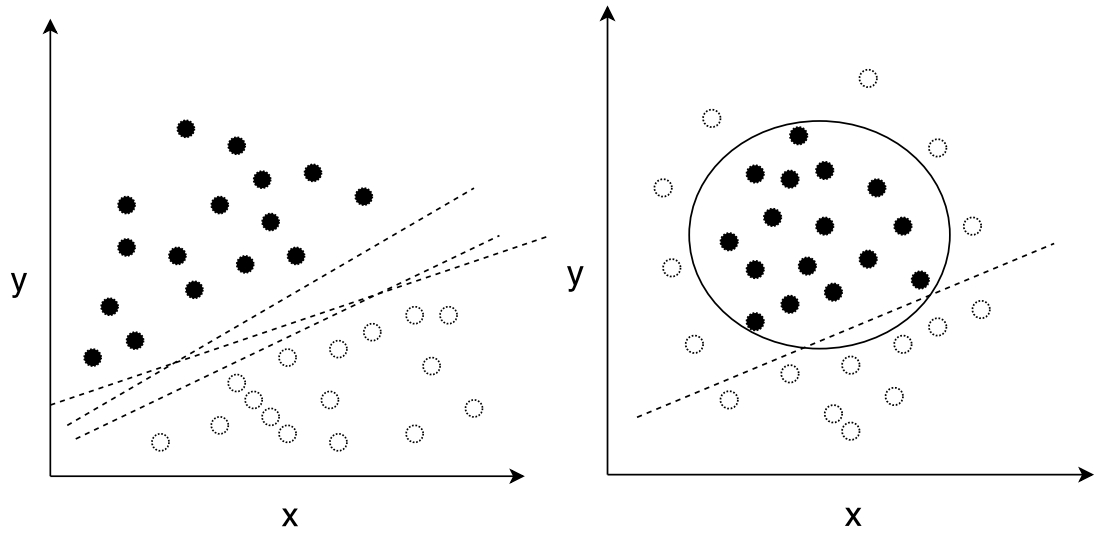
A neuron is the most fundamental part of a neural network. A *neuron* operates on its inputs and produces a transformed output using a mathematical function. For example, consider a single neuron with three inputs and one output as shown in Fig. 2.1. In this figure, the weights, w_1, w_2 and w_3, are scalar numbers. Their functionality will be explained in the next paragraph. The mathematical function of this neuron is distributed into two parts, the function g is a function of the inputs and the weights as shown in Eq. 2.1. The final output o of the neuron is a function of g as shown in Eq. 2.2. The weight, w_K where $K = [1, 3]$, is a constant that is independent of any input to the neuron. This neuron performs computation as follows: if $i_1 = 0$, $i_2 = 1$, $i_3 = 3$ and $w_1 = 0$, $w_2 = 1$, $w_3 = 0.5$, and $w_K = 0$, then $g = 0 \times 0 + 1 \times 1 + 3 \times 0.5 = 2.5$. The final output of the neuron will be $f(2.5) = 1$. If the inputs change to $i_1 = 0$,$i_2 = -3$, $i_3 = 1$, then $g = -2.5$, and the final output of the neuron will become 0. Thus, a neuron reacts to changes in the inputs by changing its output value accordingly.

$$g = \sum_{n=1}^{k} i_n \times w_n + w_K \tag{2.1}$$

$$f(g) = \begin{cases} 1, & \text{if g} \geq 0 \\ 0, & \text{if g} < 0 \end{cases} \tag{2.2}$$

A number of such neurons can be connected together to form an ANN. In these networks, information is transmitted between the neurons using links. Each link in the network is associated with a variable parameter, called a *weight*. The function of a weight is to amplify or attenuate the information being transmitted. For example, if we consider the example neuron shown in Fig. 2.1, there are three links that transmit information from three other neurons to this neuron. Each of these links is associated with a weight, represented by w_i, where i is the link ID. The strength of the information being transmitted on a link is determined by the value of the corresponding weight.

A weight modifies the corresponding information as follows: assume $i_1 = 2$ and $w_1 = 2$, now the actual input corresponding to this link is determined as by $i_1 \times w_1 = 2 \times 2 = 4$, however, if the value of w_1 was 0.5, the input to the neuron will become $2 \times 0.5 = 1$. This can in turn change the output of the neuron as follows, if $i_1 = 0$, $i_2 = 1$, $i_3 = 3$ and $w_1 = 0$, $w_2 = 1$, $w_3 = 0.5$, then $g = 0 \times 0 + 1 \times 1 + 3 \times 0.5 = 2.5$, and $o = 1$. However, if the weights change to $w_1 = 0, w_2 = -2, w_3 = -1$, then $g = -5$ and $o = 0$. This can also be seen from another perspective. The weights corresponding to a neuron's input links can be changed to obtain a desired output from the neuron. This process of adjusting the weights to obtain a desired output from a neuron is called *training*. Thus, after



(a) Linearly separable classification task (b) Non-linearly separable classification task

Figure 2.2: Two types of classification tasks performed by ANNs. The Y-axis the range of values that two different classes of objects for their respective features represented by the values on the X-axis. Each straight lines represents the output of a linear perceptron similar to the one shown in Fig. 2.1, but with a single input and two weights. In the figure on the right, the features are distributed in a non-linear manner, and hence, its not possible to correctly classify them using a straight line.

a computation is performed by the neurons inside a neural network, the network can be trained by adjusting the network weights to produce desired outputs.

A common application of such networks is to classify the inputs into different classes based on their characteristics or features. For example, consider a task where we need to separate two different classes, whose Y-axis represents a range of values each class can take for the corresponding features on the X-axis. as shown in Fig. 2.2(a). A single neuron is sufficient to solve the task relating to such linearly separable classes. A neuron with one input with characteristics as shown in Fig. 2.1 can be trained such that its output separates these two classes of objects. Here, the values on the X-axis are given as inputs to the neuron. This

neuron's function g now represents a straight line as shown in Eq. 2.3. We need to find w_1 and w_2 such that the straight line g separates the two classes of objects. The three dotted lines in Fig. 2.2(a) represent three different outputs of the neuron with different weights. Such a neuron is called a *linear perceptron*. Rosenblatt [25] derived a perceptron update rule to find the weight combinations such that the error in classification is minimized. However, a single perceptron cannot learn to perform a non-linearly separable task as shown in Fig. 2.2(b) [9].

$$g = w_1 \times x + w_2 \tag{2.3}$$

2.2 Multi-layered ANNs

Hecht-Nielsen [9] showed that additional layers of neurons are required to solve tasks involving non-linearly separable classes. These networks are organized into input layer, hidden layer(s), and the output layer as shown in Fig. 2.3. The neurons in the hidden layer do not interact with the inputs directly, hence they are called hidden neurons. This extends the ability of perceptrons to be able to perform non-linearly separable tasks.

Based on the direction of flow of information in the network, ANNs can be classified into two categories: *Feed-forward Neural Networks* (FNNs) and *Recurrent Neural Networks* (RNNs).

2.2.1 Feed-Forward Neural Networks

In FNNs neurons are interconnected such that there is a unidirectional flow of signals from the input side to the output side [2]. There are no feed-back connections between the neurons. The output of each neuron is independent of the past inputs.

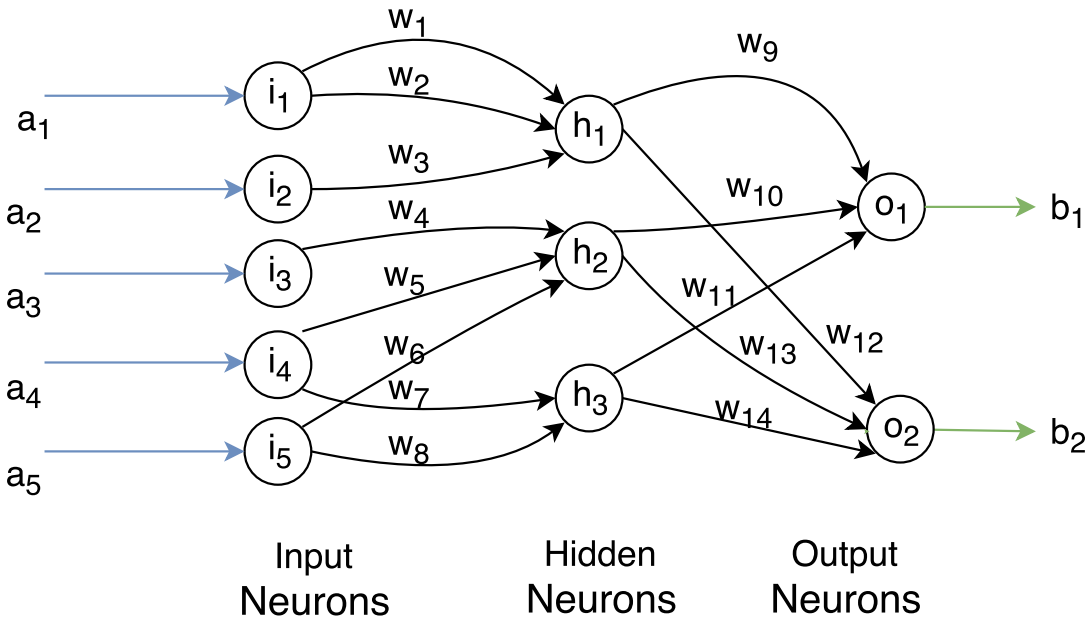


Figure 2.3: A neural network with three layers of neurons. The hidden neurons do not directly interact with the inputs.

A Feed-forward neural network, with three layers of neurons, is shown in Fig. 2.3. Feed-forward networks map a set of inputs to a set of outputs, a function that is independent of time. For example, they can detect whether, an input represents an "X" or an "O" pattern, but, they cannot detect whether an input sequence contains a series of two continuous "X" patterns. The first kind of task is called a *non-temporal task*, and the latter is called a *temporal task*.

Feed-forward networks have been shown to be universal approximators, and are appropriate for performing tasks where the inputs are independent of time [12]. Common applications of these networks are feature detection in images, object classification, handwritten digit recognition etc.

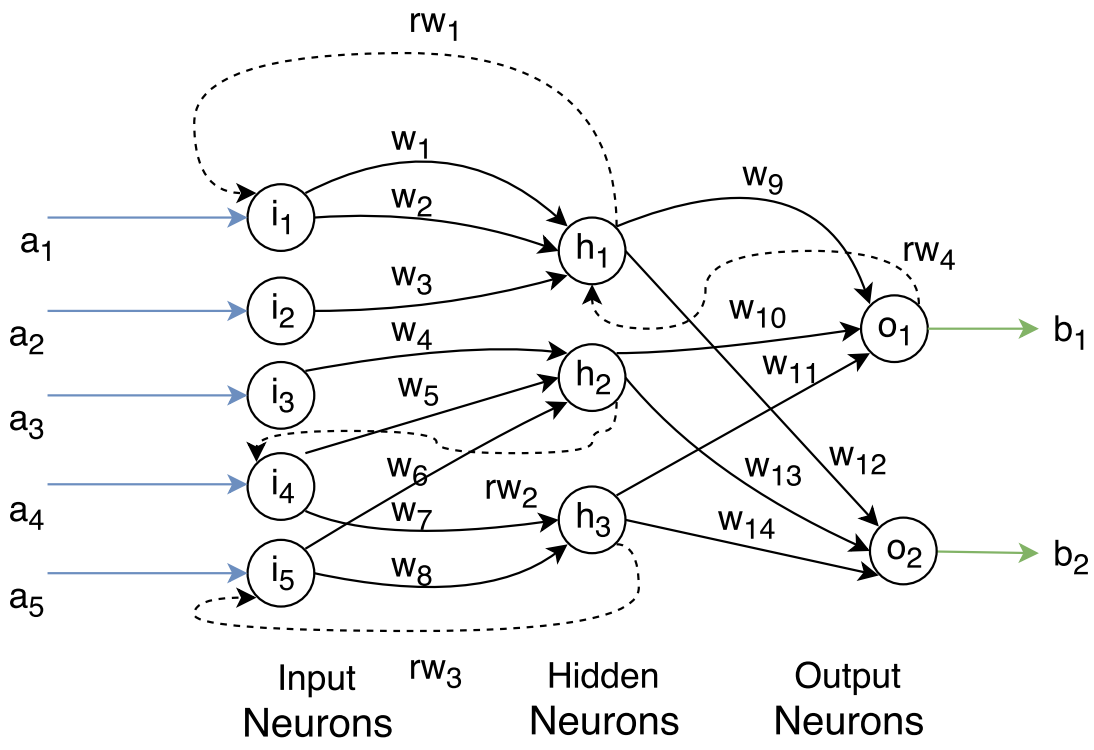


Figure 2.4: Example of Recurrent Neural Network with 5 inputs, and 2 outputs. In these networks, the outputs of a few neurons are fed back to other neurons in the network.

2.2.2 Recurrent Neural Networks

RNNs can be used to perform temporal ML tasks which the feed-forward neural networks cannot solve. Because of recurrent connections, such networks are able to retain information about the sequence of inputs presented to them over a period of time. This means that the output of the neurons will be dependent on both the current and the past set of inputs. Thus, the RNNs have memory. This is achieved by adding feedback connections to some of the neurons.

Fig. 2.4 shows a three layer recurrent network. The dashed directed arrows represent recurrent connections in the network. The inputs are a_1, a_2, a_3, a_4, and a_5. The outputs are b_1 and b_2. The recurrent connections provide feedback in these

networks. For example, the link corresponding to rw_1 connects the output of the hidden neuron h_1 to the input neuron i_1. This feeds the output of the neuron h_1 back to the input of neuron i_1.

RNNs are commonly trained by unrolling them in time, and thereby creating a new layer for each time step of the input sequence. This results in a deep, layered feed-forward neural network. The most common training method is called Back Propagation Through Time (BPTT) [34]. In this method, an error is propagated in reverse, from the outputs towards the inputs. The weights are adjusted according to their error gradients.

In these networks, all the weights are trained [11, 33, 34]. Although RNNs are able to solve temporal tasks, they also have drawbacks: the problem of vanishing gradients makes it difficult to train these networks [10, 23], and the number of weights that need to be trained increases as the networks are unrolled. Hence, such networks have a higher learning complexity than FNNs.

2.3 Reservoir Computing

Reservoir Computing (RC) is a new machine learning paradigm. Initial work in this field was independently published by Jaegar *et al.* [13] and Maass *et al.* [21]. The reservoir forms the core of an RC system. A reservoir is an unstructured dynamical system that is not trained. As opposed to training all the weights in the case of RNNs, the RC approach relies on training a simple linear read-out layer only.

A high level representation of an RC system is shown in Fig. 2.5. The double lined circle represents the reservoir, and each small circle inside the reservoir represents a neuron. Each solid arrow represents a connection between two neurons.

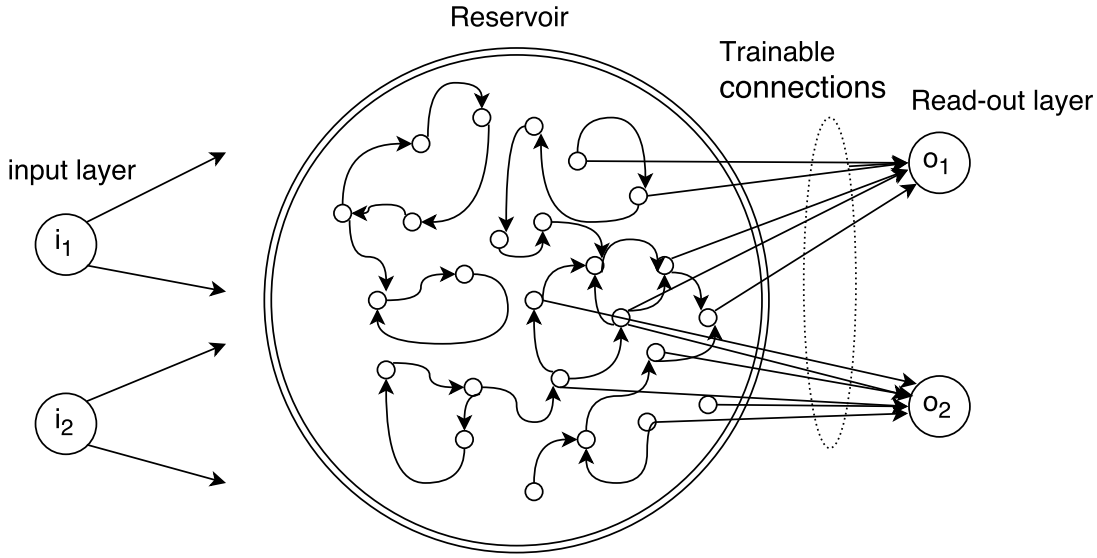


Figure 2.5: A high-level representation of an RC system, with two inputs and two readouts. The readouts o_1 and o_2 can be simultaneously trained to perform two different tasks, because they use different sets of links to tap into the reservoir.

i_1 and i_2 represent the inputs, and o_1 and o_2 represent the read-out neurons.

The read-out layer contains a set of neurons that perform a linear combination of a limited number of reservoir neuron outputs. This read-out layer is trained to produce a desired output commonly by using linear regression [13, 21]. Thus, RC systems address the two major drawbacks of RNNs:

- they are computationally cheaper to train because of the fewer number of weights that are trained; and
- there is no problem of vanishing gradients, as all the training happens in a single read-out layer.

In addition, RC systems can solve multiple tasks simultaneously with the same reservoir by increasing the number of read-out neurons. For example, the read-outs o_1 and o_2 in Fig. 2.5 can be trained to perform two different tasks using outputs from the same reservoir simultaneously.

RC systems have been successfully used to solve temporal tasks, such as predicting chaotic systems [14], speech recognition [31], and Japanese vowel recognition [15].

2.3.1 Echo State Network model

According to Jaeger, the state of an RNN can be considered as an echo of the input history. The state of an RNN refers to the set of states of all the neurons present in the network. If $(u(n), u(n-1), u(n-2)...)$ is the set of current and past inputs, up to the n^{th} time instant, the state of the RNN is a function of these inputs. This author considers discrete-time, sigmoidal neurons as the constituents of RNNs. The network architecture considered in his work is shown in Fig. 2.5.

$$x(n+1) = f(W^{in} \times u(n+1) \; + \; W \times X(n) + \; W^{back} \times y(n)), \qquad (2.4)$$

where $f = (f_1, f_2, ..., f_N)$ are the transfer functions of the neurons present in the reservoir. Here, W^{in} is the set of weights corresponding to the inputs connected to the reservoir, W is the set of weights corresponding to the connections in the reservoir, and W^{back} is the set of weights corresponding to the output connections. The output, $y(n)$ is computed as shown in the following equation:

$$y(n+1) = f^{out}(W^{out}(u(n+1), \; x(n+1), \; y(n))), \qquad (2.5)$$

where $f^{out} = (f_1^{out}, f_2^{out}, f_3^{out}, ..., f_L^{out})$ are the transfer functions of the output units.

Under the restriction that there is no feedback from the output to the reservoir, the output can be given by the following equation:

$$y(n+1) = G(..,\ u(n),\ u(n+1)). \tag{2.6}$$

It is clear that the output of the is dependent on the input history.

The current state of the i^{th} unit in the reservoir is determined by an echo function as shown below.

$$x_i(n) = e_i(..,\ u(n-2),\ u(n-1),\ u(n)). \tag{2.7}$$

where e_i is the echo function of the i^{th} unit in the reservoir. Considering a linear function G for determining the output, the following equation can be used to determine the output.

$$y(n) = f^{out}(\sum_{i=1}^{N} w_i^{out} x_i(n)), \tag{2.8}$$

where w_i^{out} is the weight of the i^{th} output connection, and $x_i(n)$ is the state of the corresponding neuron. The above equation can be further written as follows.

$$(f^{out})^{-1} y(n) = \sum_{i=1}^{N} w_i^{out} e_i(.., u(n-2),\ u(n-1),\ u(n)), \tag{2.9}$$

if f^{out} is considered to be $tanh$ function, which is invertible.

An error is defined as follows.

$$\epsilon_{train}(n) = (f^{out})^{-1} y_{teach}(n) - (f^{out})^{-1} y(n), \tag{2.10}$$

where $y_{teach}(n)$ is the n^{th} desired output, to which the reservoir is to be trained.

The goal is to minimize the *Mean Square Error* (MSE) defined as follows:

$$\text{MSE}_{\text{train}} = (n_{max} - n_{min})^{-1} \sum_{n=n_{min}}^{n_{max}} \epsilon^2_{train}(n) \tag{2.11}$$

This is a simple linear regression task: to minimize $\text{MSE}_{\text{train}}$, we train the read-out neurons by updating the set of weights, w_{out}. A few ways of training different types of read-out neurons were presented by Lukosevicius *et al.* [20].

In order achieve good results, we need to ensure that there is enough richness in the echoes available inside the reservoir. To achieve this, the reservoir needs to be assembled in a suitable non-homogenous way to provide enough dynamics for generating a variety of echoes.

2.3.2 Liquid State Machines

Maass *et al.* [21] also independently published their work as *Liquid State Machines* (LSMs). LSMs use spiking neurons, and are thus more akin to the neural networks inside the human brain.

The authors propose LSMs as a model for real-time computation, i.e., both the inputs and outputs of an LSM are continuous in time. As mentioned earlier, other computing models such as RNNs are faced with issues related to slow learning and vanishing gradients. As with all RC approaches, LSMs delegate the process of learning to a single read-out stage by leaving the bulk of the LSM untrained. From the perspective of neuroscience, this is analogous to a projection neuron that extracts information from a group of neurons and projects it to other neurons in the network.

In both Jaeger's and Maass's works, the influence of network topology, hierarchical structure and decomposition of the reservoir into smaller sub-reservoirs was not studied. On the other hand, Triefenbach *et al.* [30] showed that the accuracy of

RC systems in solving a speech recognition task increases when the reservoir size is increased from $1,000$ to $20,000$ neurons. However, building such large monolithic RC systems in hardware may face problems, e.g., signal attenuation. On the other hand, Burger *et al.* and Rodriguez *et al.* explored how decomposing a large monolithic reservoir into smaller communities, and introducing a network topology changes the performance the RC system. The results of these two studies are discussed in the following subsection.

2.3.3 Hierarchical and Modular RC Systems

Burger *et al.* [3] constructed hierarchical RC networks using a fixed ring type topology, and compared their performance to that of equivalent monolithic networks. They tackled the issues monolithic reservoirs suffer from: signal inter-dependencies, and correlated read-outs, by composing reservoirs using multiple smaller reservoirs connected in a circular fashion. They called these reservoirs *Single Cycle Reservoirs* (SCRs). They showed that such reservoirs were able to outperform monolithic RC systems by up to 20% in solving temporal tasks, such as waveform generation, and NARMA-10.

In their work, these authors used memristive neurons. These neurons demonstrate a hysteresis loop in their current vs voltage transfer characteristics, hence have a resistance that is non-linearly dependent on the amplitude of the input voltage. These neurons thus combine memory and computation capability in a single non-linear device, which makes them particularly suitable for reservoir computing purposes.

Burger *et al.* used networks with up to 400 neurons only. On the other hand, Rodriguez *et al.* [24] studied the influence of the network modularity parameter

on the performance of different types of modular RC systems. They defined modularity as the fraction of network connections that connect neurons in one module to neurons in another module. They showed that there is an optimal range of modularity, where the computation inside the communities, and the information exchange between communities is such that the performance becomes optimal.

They investigate a large two-community network, where a seed community was perturbed by an input signal and the activity in the other community was observed. They found that the activity in the second community depended on the number of connections coming from the seed community. A modularity between 0.1 and 0.2 resulted in optimal activity in the second community. They discovered that at high values of modularity, i.e., when the connections between the communities are far greater than the connections between the neurons inside each of them, the activity and performance can drop.

They also investigated mesh networks having up to ten communities of neurons. The aim of their memory recall task was to train RC systems on 1024 different multi-dimensional input sequences and then recall up to 300 sequences from memory. Networks with high performance required a large set of different attractors in which the unique sequences can be stored. Also, the duration for which these sequences can be retained by the reservoirs were increased if the reservoirs are able to store these input sequences in different cyclic attractors.

By varying the number of sequences to be recalled and the duration after which each sequence was to be recalled by the reservoir, the authors tested networks with modularity ranging between 0 and 0.5. They used networks containing up to $1,000$ neurons distributed into 10 communities. They observed that networks with a modularity of 0.06 and 0.15 were able to perfectly recall up to 200 different input

sequences. The performance of networks with a of modularity greater than 0.2 was negligible. They termed this as *computation at the edge of modularity*, as they discovered an optimal range of modularity for which their networks outperformed networks with modularity other than this optimal range.

2.4 Boolean Logic Gates and Random Boolean Networks (RBNs)

Various types of computing elements, such as random Boolean logic gates [28], memristors [3], spiking neurons [21], and sigmoidal neurons [24], have been used as neurons in RC systems. Boolean logic gates are simpler to model, simulate, and fabricate compared to other neurons. For example, networks containing Boolean gates can easily be implemented in hardware by using *Field Programmable Gate Arrays* (FPGAs).

An RBN is a network of Boolean logic gates, where the connections between the gates are randomly determined, the logic functions of all the gates are randomly chosen, and all the gates are initialized to random outputs [17]. A Boolean logic gate represents a many-to-one mapping function which performs a Boolean operation on its inputs and produces a binary output instantaneously. The state of an RBN is represented by the concatenation of the states of all the Boolean logic gates contained within it, and it is a binary sequence of ones and zeros.

These networks are dynamic systems, and the states of their nodes can change even without any external perturbation. As these networks are randomly assembled, groups of gates can be connected in such a way that they form a directed cycle, similar to a recurrent connection. For example, consider the simple synchronous RBN formed by connecting three Boolean logic gates g_1, g_2 and g_3, as shown in Fig. 2.6. The gates g_1 and g_2 have external inputs i_1 and i_2 which

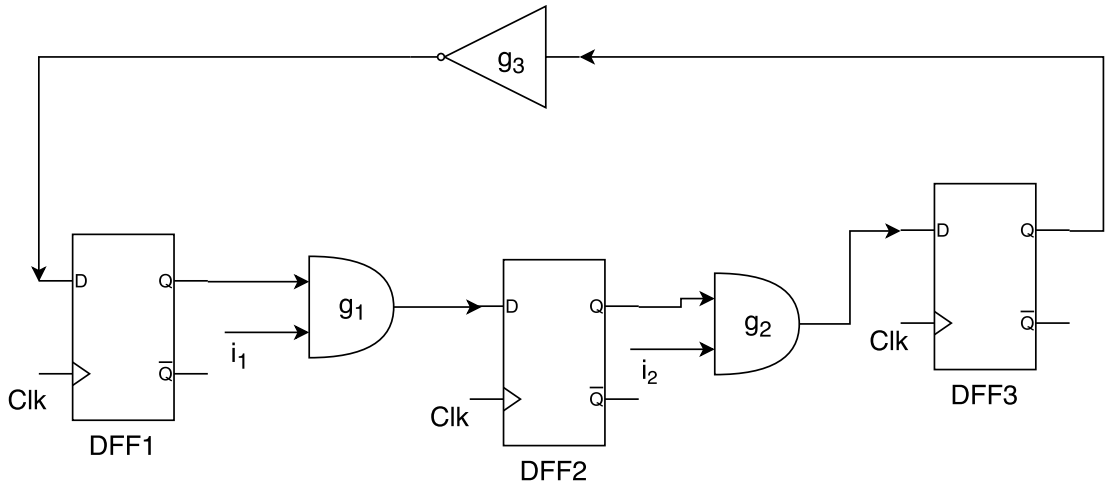


Figure 2.6: A synchronous random Boolean network with three gates, and two external inputs. The connections form a directed cycle through the three Boolean gates, resulting in a dynamic behavior.

can be controlled at any point of time. These two gates perform logical AND operation on their inputs. The gate g_3 is an inverter. The state of this network is updated once every positive edge of the clock signal. As all the flip-flops are updated simultaneously, such networks are called *synchronous RBNs*.

Assume that the outputs of g_1, g_2, and g_3 are randomly initialized to $0, 1, 0$ respectively, at time t0. Assume that the external inputs i_1, i_2 are clamped to 1. The state of the RBN can be represented as 010. If the outputs of all the three gates are allowed to update simultaneously, once per time step, then at time t1, the output of g_1 changes to 0, the output of g_2 changes to 0, and the output of g_3 stays at 0. The state of network at time t1 will be 000 Now, at time t2, the output of g_1 stays at 0, the output of g_2 stays at 0, and the output of g_3 changes to 1. The state of this network advances as follows:

$$010 - - > 000 - - > 001 - - > 101 - - > 111 - - > 001 - - > 010... \quad (2.12)$$

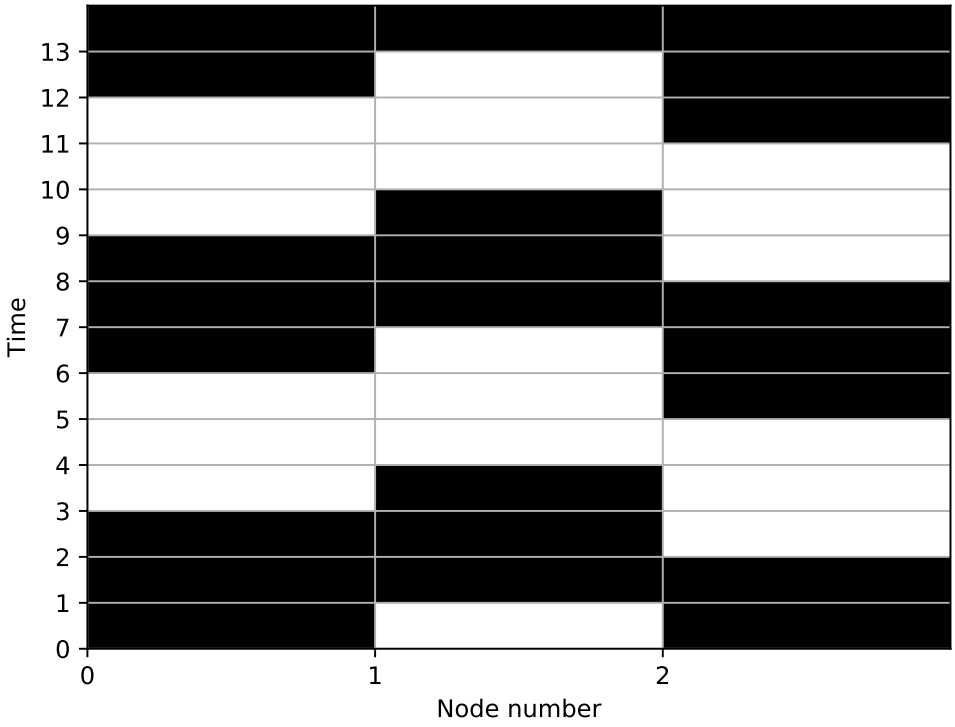


Figure 2.7: The state-time diagram of the example RBN over a period of 15 time steps. The state of the RBN is a periodic sequence which repeats after every six time steps.

After t6, the new state of this network will be 010 which is same as the state at time t0. The state of the RBN cycles through these sequences indefinitely, until either i_1 or i_2 are changed to a different value. The RBN state over a period of 15 time steps is shown in Fig. 2.7. The black cells indicate state zero, and white cells indicate state one.

It can be seen from Fig. 2.7 that the state of the RBN returns to its initial state of 010 after every six time steps. This sequence will repeat indefinitely until either i_1 or i_2 change their state, and such a state of an RBN is called an *attractor*. As this attractor is a set of periodical sequences, it is also called as *cyclic attractor*.

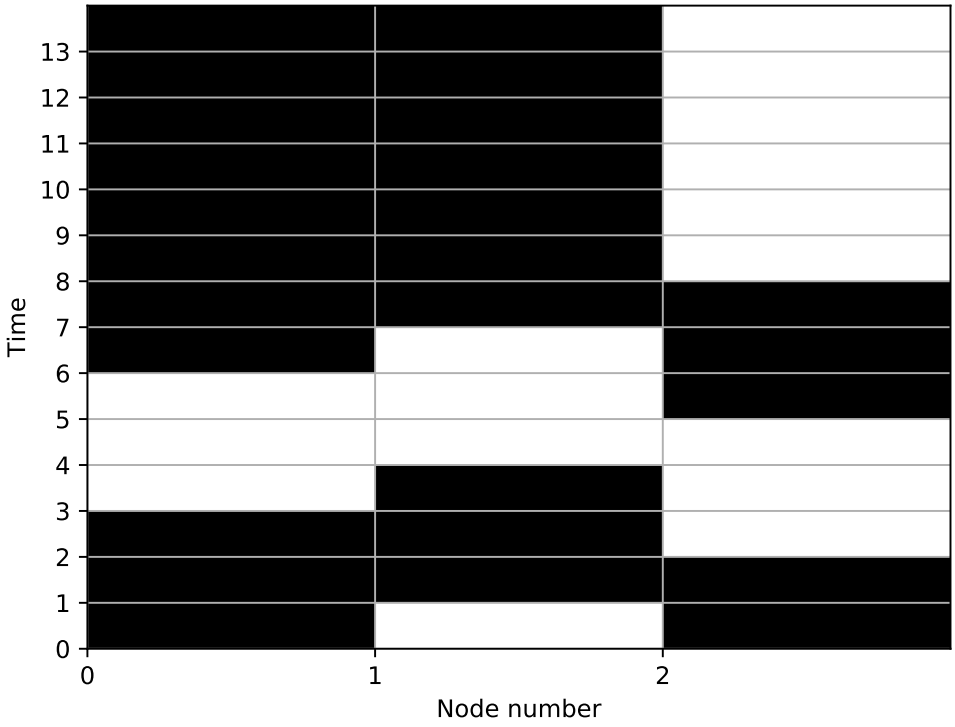


Figure 2.8: The time-state plot of the example RBN over a period of 15 time steps. The state of the RBN remains at a single sequence indefinitely.

On the other hand, an RBN can also fall into a point attractor depending on the initial conditions, and the external input values. The state of the RBN no longer changes with further advancement of time, after falling into a point attractor.

Another example, where this RBN falls into a point attractor is shown in Fig. 2.8. Assuming the same initial conditions, and same input conditions until seventh time step, and if the input $i1$ is changed to zero at the seventh time step, the output of $g1$ remains at zero indefinitely irrespective of its other input. Hence, this leads to the behavior observed between time steps 8 and 14, where the state of the RBN remains at 001 indefinitely.

These networks have been studied extensively over the last few years and an

introduction can be found in the work published by Kauffmann [16]. Also, according to Mihaljev *et al.*, the attractor number and length increase faster than a power law as N^x where N is the number of nodes in the RBN [22].

On the other hand, the dynamics in an RBN determined by the average connectivity [28]. The average connectivity of an RBN is defined as follows:

$$K_{av} = \frac{\text{Sum of incoming connections of each gate}}{\text{Total number of gates}} \tag{2.13}$$

These Boolean networks can exhibit stable, critical, or chaotic behavior, which is governed by the network's average connectivity (K_c). The average connectivity of a network is defined as the ratio between the total number of incoming connections to the network size. According to Derrida *et al.* [5], the value of K_c is determined by the following equation:

$$K_c = 1/(2p(1-p)), \tag{2.14}$$

where p is the probability of randomly assigning a 0 output to a node for a given set of inputs.

If the probability of assigning a 0 is equal to the probability of assigning a 1, i.e., $p_0 = p_1$, we can see that $p = 0.5$. Substituting this in the above equation for K_c, we get $K_c = 2$

According to Sole *et al.* [29], an RBN is stable if $K_{av} < K_c$, critical if $K_{av} = K_c$, and chaotic if $K > K_c$. In the stable regime, the Hamming distance between two initially close states decreases exponentially as time is advanced. In the chaotic regime, this Hamming distance increases exponentially. The two initially close states are those with Hamming distance that is negligible compared to the network

size (N).

Synder *et al.* studied RBNs with $K_{av} = 1,\ 2,\ 3\ and\ 4$ and showed that networks with $K_{av} = 2$ demonstrate a computational capability that is better than networks with $K_{av} = 1$ or $K_{av} = 3$ [28]. The networks with $K_{av} < 2$ are termed as ordered networks and the networks with $K_{av} > 2$ are termed as chaotic networks. In this thesis, networks with $K_{av} = 2$ have been used.

3

Hierarchical Reservoir Networks

The initial research in reservoir computing was limited to using randomly assembled computing media as reservoirs [13, 21]. These networks lacked a topological structure because they were randomly assembled, and the role of network topology in the performance of reservoirs remained to be investigated. In this research, hierarchical networks containing Boolean logic gates were used as reservoirs to evaluate temporal tasks. It was found that the performance of such networks varies with different sets of network topology parameters, such as the average connectivity (K_{av}), the network modularity, and the number of modules. This chapter contains a description of the methodology used to build and simulate hierarchical random Boolean networks with up to $10,000$ nodes.

A few common terms used when studying networks are:

- *nodes*: fundamental units of the network, present at the intersection of links;
- *links*: connect the nodes in the network, and transmit information between nodes;
- *average connectivity*: the ratio of the total number of connections to the total number of nodes in the network;
- *modularity*: the ratio of number of connections between modules to the total number of connections in the network;
- *in-degree*: it is a property of each node in the network, and is defined as the number of connections coming into the node; and

- *out-degree*: it is also a property of each node in the network, and is defined as the number of connections exiting a node.

3.1 Growth Model for Hierarchical Reservoirs

Xuan *et al.* [35] describe an algorithm to build generic hierarchical networks composed into a fixed number of hierarchical levels and modules. Hierarchical networks can be grown to any size using this model. These networks have M hierarchical levels, with each level containing a number of small groups of nodes called as *modules*. Groups of n modules at the current hierarchical level act as modules for the next higher hierarchical level. The parameters M, n are fixed for a given network. The total number of modules in the network is given by n^{M-1}. This algorithm uses a real-world network formation principle known as *Preferential Attachment* (PA) [1] for growing networks over time. This principle states that nodes with more connections are more likely to form new connections than nodes with fewer connections.

For example, a hierarchical network is shown in Fig. 3.1. This network has three hierarchical levels and $n = 3$. A total of $3^{(3-1)}$, i.e., nine modules are present in the lowest level of hierarchy. These are labelled 1 through 9 in the above figure. The second level has $\frac{9}{3}$, i.e., three modules, such that each module is composed of three sub-modules from the lowest level. The modules labelled $1, 2, 3$ form the first module, the modules $4, 5, 6$ form the second module, and the modules $7, 8, 9$ for the third module at this hierarchical level. The third level has one module that is composed of the three sub-modules from the second hierarchical level.

These networks are initialized with a fixed number of fully connected nodes in every module, and then grown to a desired size, by adding new nodes to the

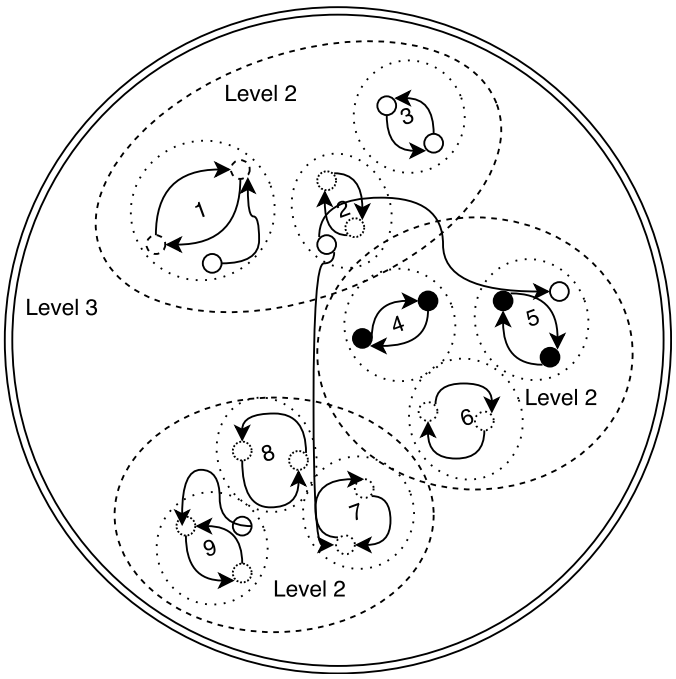


Figure 3.1: A hierarchical network with $M = 3$ hierarchical levels, $n = 3$ and nine modules, grown up to a size of 22 nodes. Some nodes have zero in-degree because the already existing nodes with more incoming connections are preferred to connect to the new nodes when the network is grown.

modules, and connecting them to other nodes in the network.

3.1.1 Growth Operations

A network is grown by performing one of the two possible growth operations, *in-module* connection, and *between-modules* connection. A predetermined probability value, corresponding to the hierarchical level at which the growth operation is being performed, is used to determine whether a growth operation should be in-module or between-modules. The probabilities of these two connection types are related as shown in Eq. 3.1.

$$P_{\text{in-module}} + P_{\text{between-modules}} = 1, \tag{3.1}$$

where $P_{\text{in-module}}$ is the probability of a growth operation being an in-module connection. $P_{\text{between-modules}}$ is the probability of a growth operation being a between-modules connection.

An *in-module* connection results in the addition of a new node to a selected module and connecting it to one or more target node(s) within the same module. This type of connection is only allowed at the lowest level of the hierarchy. A module selection for performing this operation is done using a uniform probability distribution for all modules present in the network. On the other hand, the set of target node(s) is selected according to the PA rule.

While new nodes are not added during *between-modules* connection, a new link is added between two nodes belonging to two different modules. This type of connection is allowed at all levels of the hierarchy. These connections are the channels of information exchange between the modules. The two modules that are to be connected, are selected using a uniform probability distribution. The target and source nodes in these modules are then selected according to the PA rule.

Hence, during a growth operation at the hierarchical level h, if the *in-module* connection type is selected and $h - 1^{th}$ hierarchical level is not the lowest, we recursively descend the hierarchical levels until either, a *between-modules* selection occurs or *in-module* selection happens at the lowest level of the hierarchy. Before proceeding to a complete network growth example, it is useful to know about the various parameters used in this algorithm.

3.1.2 Network Parameters

The following parameters are used in building these networks:

- M: number of hierarchical levels;

- n: number of modules that constitute a module of the next higher level;
- q_h: probability of in-module connection at the hierarchical level h;
- m_0: number of initial nodes in each module of the network;
- m: number of nodes to which a newly added node is connected; and
- N: network size

Before the network is grown, a probability array (Q) of $M-1$ elements is generated such that the first element is greater than 0.5. The elements of this array represent the values of q_h for $h = [1, M)$. The following equation is satisfied by each element of this array with the exception of the first element:

$$1 - q_{h+1} << q_{h+1}(1 - q_h). \tag{3.2}$$

This equation ensures that the probability of *in-module* connections is always greater than the probability *between-modules* connections at each hierarchical level, so that the hierarchical and modular structure doesn't vanish even if the network is grown to a large size. Also, the elements of Q are in an increasing order, i.e., the value of q_h increases as h increases from 2 to $M-1$, assuming that the first element of Q is q_1.

3.2 Example Hierarchical Network

Let us consider a hierarchical network with $M = 3$ hierarchical levels, and $n = 3$. There will be 3^{3-1}, i.e., nine modules at the lowest hierarchical levels. Consider the following probability array, $Q = [0.56, 0.86]$, where the probability of in-module connections at the highest hierarchical level, $q_2 = 0.86$. These probabilities are in

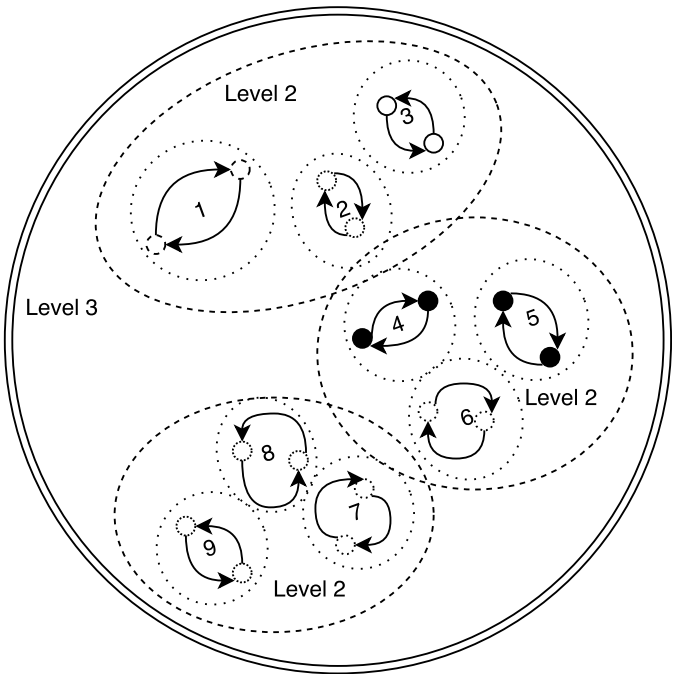


Figure 3.2: The initial state of the example hierarchical network. Each module in the network is populated with two fully connected nodes, to have an even distribution of connections from newly added nodes.

accordance with Eq. 3.2. The value of q_1 is always more than 0.5 so that the probability of adding new nodes is more than the probability of connecting two existing nodes.

As $n = 3$, the modules are categorized into groups of three modules each. The second level hierarchy has three modules, with each module containing three sub-modules from the lowest hierarchical level. The highest hierarchical level contains a single module containing the three sub-modules from the second hierarchical level as shown in Fig. 3.2. The modules are numbered one through nine in this figure at the lowest level of hierarchy. Modules 1, 2, and 3 constitute the first module, modules 4, 5, and 6 constitute the second module, and modules 7, 8, and 9 constitute the third module in the second hierarchical level.

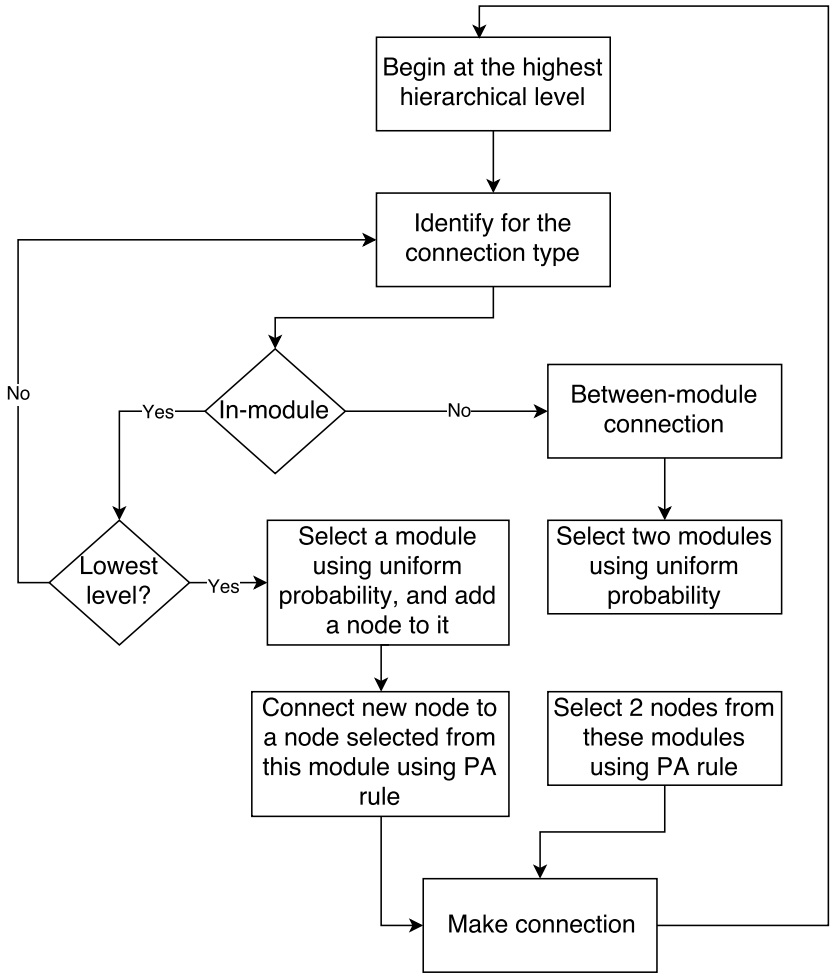


Figure 3.3: A flow chart showing the steps followed while performing during a network growth step.

Initially, each module is populated with two *fully-connected* nodes as shown in Fig. 3.2. This example network has a total of 18 nodes at the beginning. The modules were populated in this way to get an even distribution of the connections from newly added nodes, so that a single node doesn't always have a preference of adding more connections. The in-degree and out-degree of each node are both one. There are two possible growth operations as discussed earlier. A new node can be added and connected to other nodes in a module, or a new connection can be

made between two already existing nodes in two different modules in the network. The network is grown by repeatedly performing growth operations until a desired number of nodes is achieved.

During a growth operation, the value of h is set to M, and the probability q_{h-1} is used to determine the type of connection to be made. A random floating point number is generated, and if it is less than or equal to q_{h-1}, *in-module* connection type is selected. On the other hand, if this generated number is greater than q_{h-1}, a *between-modules* connection type is selected. Such growth operations can be carried out until a desired network size is reached. The steps followed in a growth operation are shown in Fig. 3.3.

As can be seen from the flow chart in Fig. 3.3, if an in-module connection type is selected at the h^{th} hierarchical level, and the $h-1^{th}$ level is not the lowest, h is decremented by one, and the process is restarted from identifying the connection type. This process is repeated until a between-modules connection type is selected at the h^{th} level or in-module connection type is selected, and $(h-1)=1$.

3.2.1 In-module Connection

For example, when $h=3$, if the random number generated is 0.74, an "in-module" connection type is selected. But since $h-1=2$ is not the lowest hierarchical level, h is decreased by one, and the connection type selection process is restarted. If this time the generated number is 0.6, an in-module connection can be made, because $h-1=1$. A module is selected from the 9 available modules using a uniform probability distribution. A new node is added to this module, and connected to two other nodes in the same module, as shown in Fig. 3.4.

By connecting the newly added node to two other nodes in the network ensures

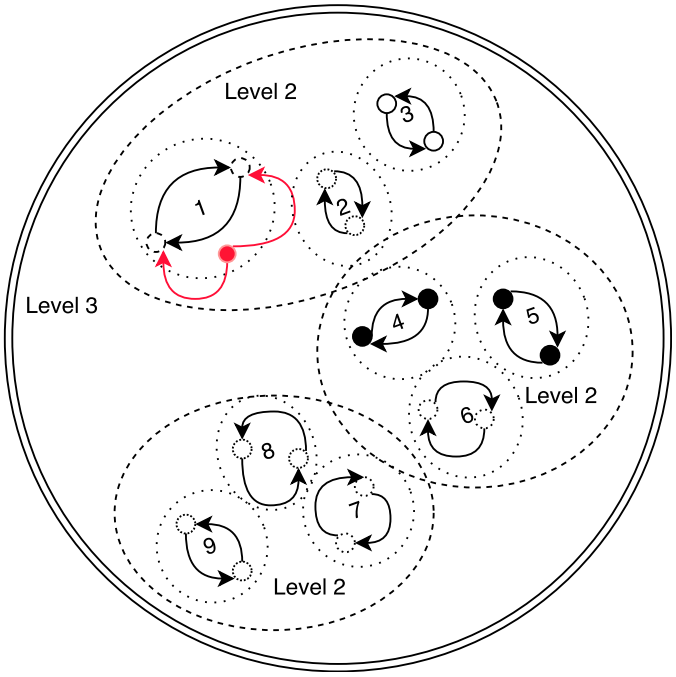


Figure 3.4: An in-module connection made in the first module by adding a new node and connecting it to two other nodes in the same module.

that the average connectivity of the network will be close to two, if the probability of making such in-module connections is significantly larger than the probability of making a between-modules connection. This is important to this research because we intend to use hierarchical RBNs with an average connectivity, $K_{av} = 2$.

3.2.2 Between-modules Connection

On the other hand, beginning at the highest hierarchical level, if the number generated is 0.89, a "between-module" connection type is selected. This type of connection is allowed at all levels of the hierarchy. Hence, two modules are selected from the three available modules using a uniform probability distribution. A target node and a source node are also selected from the two modules using Preferential Attachment [1]. In simple terms, the PA rule says that nodes having a higher

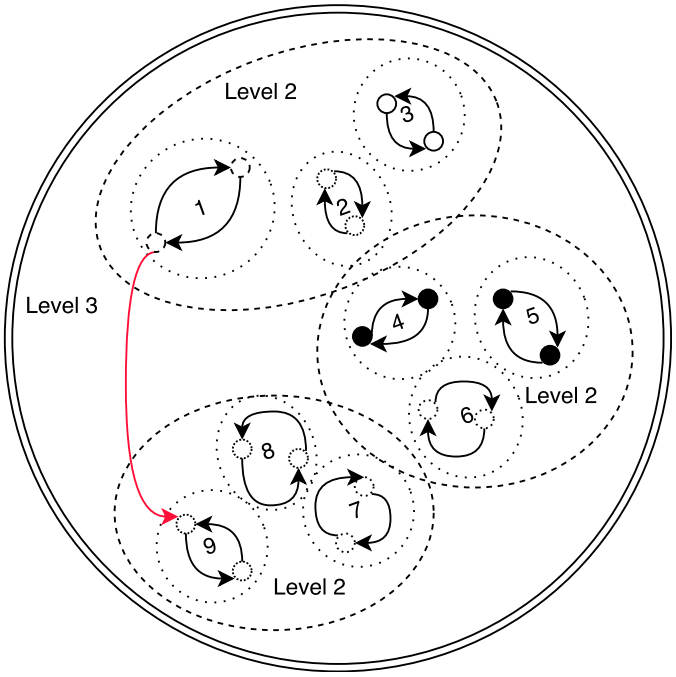
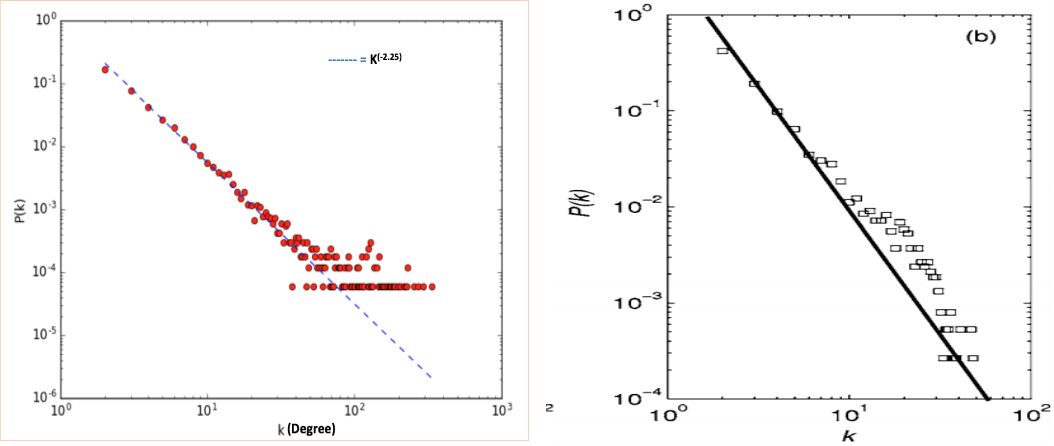


Figure 3.5: A between-modules connection made between two nodes from the first and ninth modules at the third hierarchical level.

number of connections have a higher chance of being selected as a source or target node. A new connection is made between the source and target nodes, as shown in Fig. 3.5.

Also, because the in-module connection probability is greater than between-module connection probability for all hierarchical levels (> 0.5), we can grow these networks for indefinitely long times. The growth process is terminated once a desired number of nodes is reached.



(a) This work (straight line represents $K^{-2.5}$) (b) Xuan et al.'s work (straight line represents $K^{-2.6}$) [35]

Figure 3.6: Degree distribution comparison for networks with M = 5, n = 3.

3.3 Parameters and Characteristics

In this research, networks with different average connectivity and modularity were built, and evaluated by varying network parameters, such as number of hierarchical levels, total number of modules in the network, and in-module connection probabilities.

The in-module connection probabilities play a vital role in determining the average connectivity and modularity of our networks. For example, in a network with only 1 hierarchical level with an in-module connection probability of 0.9, the ratio of between-module connections and in-module connections will approximately be 0.1/1, i.e. 0.1, ignoring the connections made initially before growing the network using the growth operations.

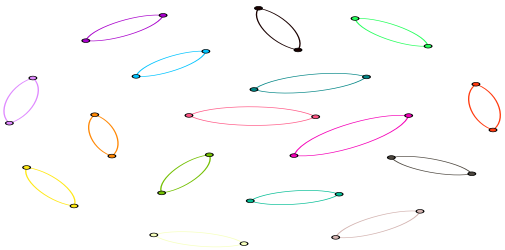
The average connectivity of this network, which is the ratio of total in-coming connections to the total number of nodes in the network, will approximately be 2 + *between-modules connections*, i.e., $2 + 0.1 = 2.1$, assuming that every time an

in-module connection is made, a node is added to the network and connected to two other nodes.

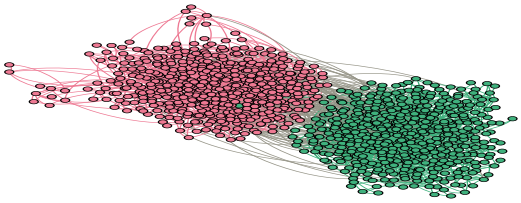
If the number of initial nodes is a large part of the nodes present when the network is grown to a desired size, the average connectivity of the network will be close to one. The number of hierarchical levels and the number of modules in the network determine how many nodes are present in the network initially before growing it. The initial nodes also contribute to the average connectivity and modularity of the final network obtained after growing it to a desired size. As mentioned earlier, initially each module is populated with two fully connected nodes. Thus, the average connectivity of the network before using the growth algorithm will be one.

The networks built in this research were validated with the networks built by the authors in [35] by comparing network characteristics of both the network families, such as the degree distribution. The degree distribution of the hierarchical networks follows a power law: K^{β}, where K is the total degree (sum of in-degree and out-degree). Fig. 3.6 shows a comparison of the degree distribution of a sample network and that of a network example used in [35] .

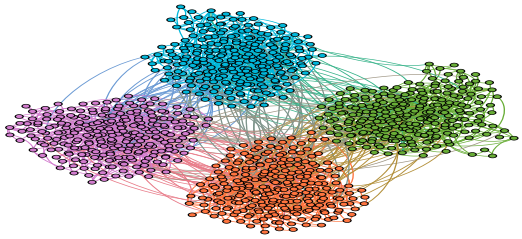
Networks with up to six hierarchical levels and two sub-modules, grown to a size of $1,000$ nodes, are shown in Fig. 3.7. It was observed that the number of between-modules connections increased as the number of hierarchical levels was increased from 2 to 6.



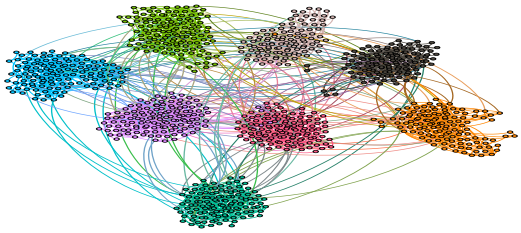
(a) Initial network with $M = 5$ and $n = 2$



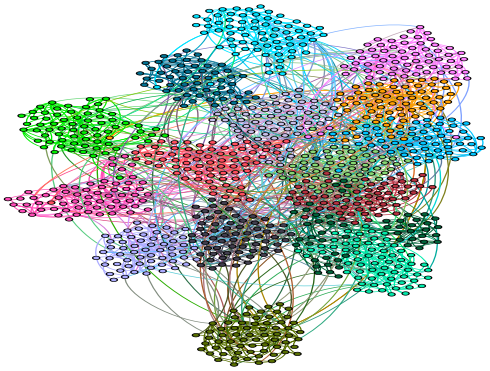
(b) Network with $M = 2$, $n = 2$



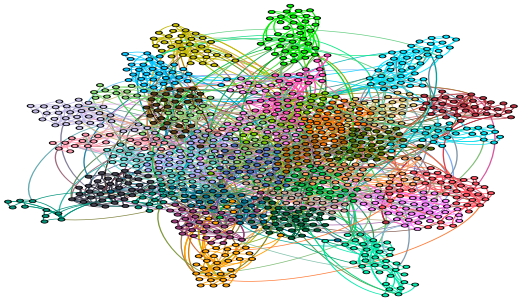
(c) Network with $M = 3$, $n = 2$



(d) Network with $M = 4$, $n = 2$



(e) Network with $M = 5$, $n = 2$



(f) Network with $M = 6$, $n = 2$

Figure 3.7: The resulting networks, with different values of M, n, that were grown to a size of $1,000$ nodes.

4

Results

In this research, different hierarchical RBN reservoirs were used to evaluate three temporal problems: pattern recognition, food-foraging, and memory recall. These tasks were chosen to show that hierarchical RBNs can perform better than monolithic RBNs.

Depending on the task, one or more input neurons were added to the RC system, and connected to the inputs of a fraction of the neurons inside the hierarchical reservoirs using links with unit weights. The outputs of a fraction of the reservoir neurons were also connected to one or more read-out neurons, which were trained to solve a given task. For example, the pattern recognition task and the food-foraging tasks required only one input node, whereas the memory recall task required four input nodes. These parameters are summarized in Table 4.1. The reservoirs were perturbed by the changing of the state of these input nodes according to the task being evaluated.

4.1 Experiment 1: Variation of Average Connectivity

This experiment was performed to study the variation of the average connectivity of the RBNs with respect to the hierarchical network parameters, M, n, which determine the number of hierarchical levels and the number of modules in the network respectively. The total number of modules in the network is given by n^{M-1}. This caused a variation in the number of initial nodes, the over-all probabilities of in-module connections, and the number of growth operations possible before

Task	Number of input nodes	Number of output nodes	Network size
Pattern detection	1	1	$1,000$
Food-foraging 5×5 grid	1	2	$1,000$
Food-foraging 10×10 grid	1	2	$5,000$
Memory recall	4	4	$2,000$

Table 4.1: A summary of the experimental set ups used to evaluate the three types of tasks. The pattern detection task was the simplest task, with one input and one output. The food-foraging task was a control task, where an agent was controlled using 2 binary inputs, which were generated by the output nodes of the hierarchical networks. The memory recall task required four inputs and four outputs.

reaching the desired network size. This in turn varied the average connectivity of the network, which has been shown to be important for determining the dynamics and computational capability of RBNs [28].

4.1.1 Methodology

In this experiment, the average connectivity (K_{av}) was calculated as the ratio of the total number of links to the total number of nodes present in the network. To study the variation of K_{av}, the minimum in-module connection probability was set to 0.9 for the networks with $M = 2, 3, 4, 5$ and $n = 2, 3, 4, 5$. Networks of sizes $1,000$, $2,000$ and $5,000$ were built using these network parameters, and their corresponding average connectivity was measured. Fig. 4.1 shows a flow chart illustrating the sequence of steps followed to obtain the average connectivity of the above mentioned hierarchical networks. About $10,000$ network samples of each of the different hierarchical networks were built up to the sizes listed above, and the mean of the average connectivity of each set of networks was calculated and plotted.

In this experiment, the reservoirs were built using the growth model described in

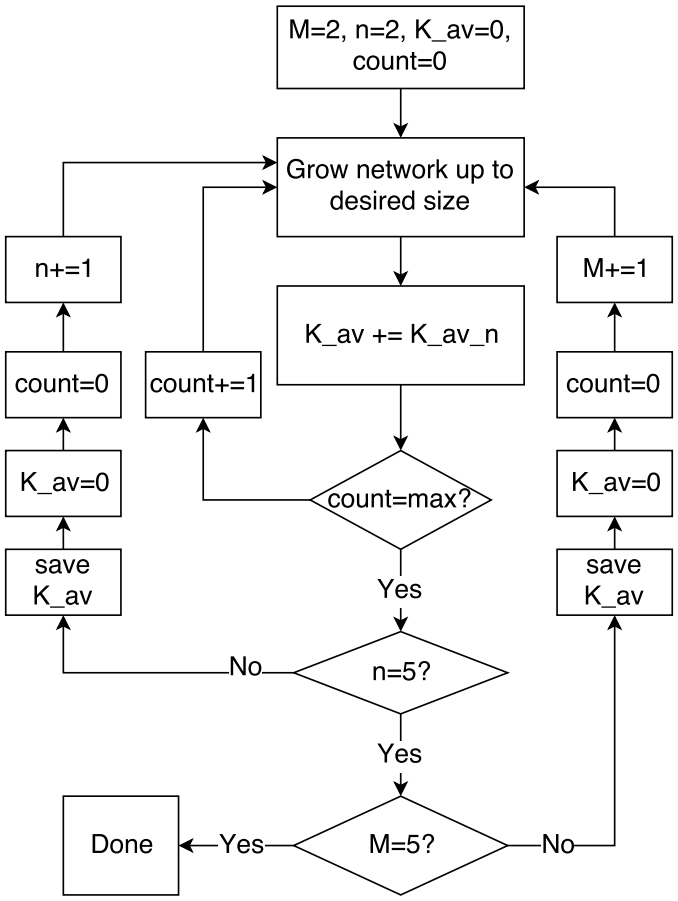


Figure 4.1: A flow chart describing the procedure followed to evaluate the average connectivity (K_{av}) of different hierarchical networks. In this experiment, the value of max was set to $10,000$.

Chap. 3, and their K_{av} was measured without perturbing them with any external input signal. As there was no need for the outputs of the reservoir neurons, a read-out layer was not added to the RC system.

4.1.2 Results

Fig. 4.2 shows the variation of average connectivity in networks containing $1,000$ neurons. In this plot, it is clear that the average connectivity of networks with $n = 2$ levels increases by up to 0.1 when the value of M is increased from 2 to

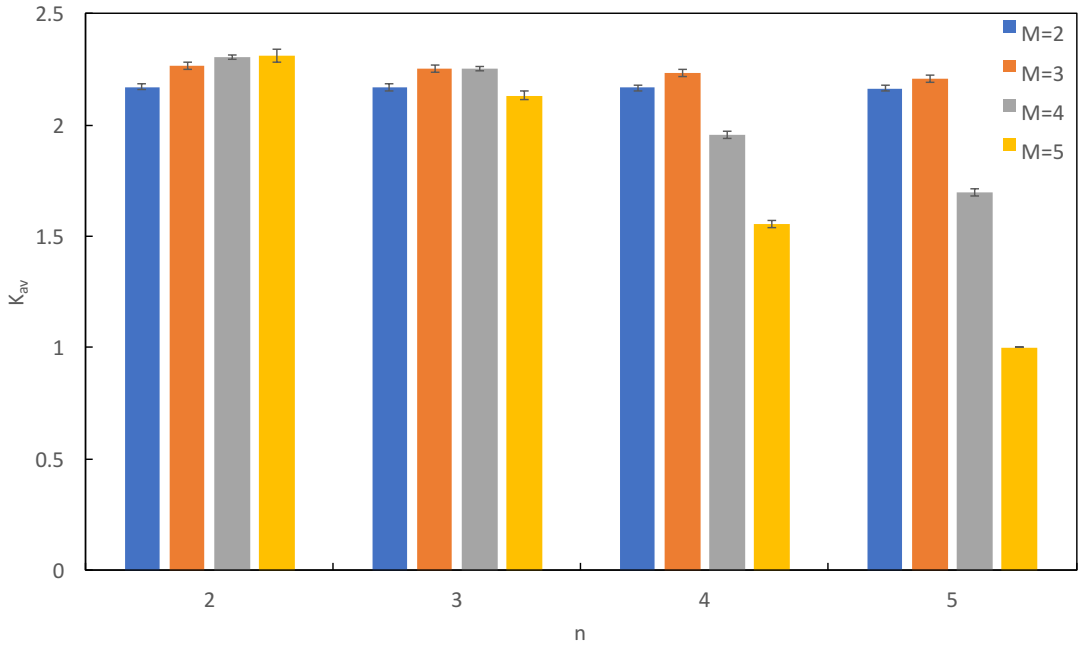


Figure 4.2: The variation of average connectivity in networks containing $1,000$ neurons, with respect to the variations in hierarchical structure. Each group of bars represent networks with a fixed value of n for different M in $[2,5]$. The error bars represent standard deviation of the average connectivity. Networks with $n = 4$, $M = 5$ have a lower K_{av} of close to 1.5 than other networks with $n = 4$ because the initial nodes in networks with $n = 4, M = 5$, i.e., $2 \times 4^{5-1} = 512$, constitute more than half of their total nodes.

5. Also, this rate of increase is inversely proportional to M. On the other hand, the average connectivity begins to decrease when the initial neurons represent a significant fraction of the total neurons in these networks. For example, in networks with $n = 4$, the number of initial neurons is given by 4^{M-1}, where M is the number of hierarchical levels. Thus, even though the average connectivity slightly increases from 2.1 to 2.3 as M increases from 2 to 3, it decreases from 2.3 to 1.7 as M is further increased from 3 to 5.

Fig. 4.3 shows the variation of average connectivity for networks grown up to $2,000$ neurons. In these larger networks, the decrease in average connectivity was

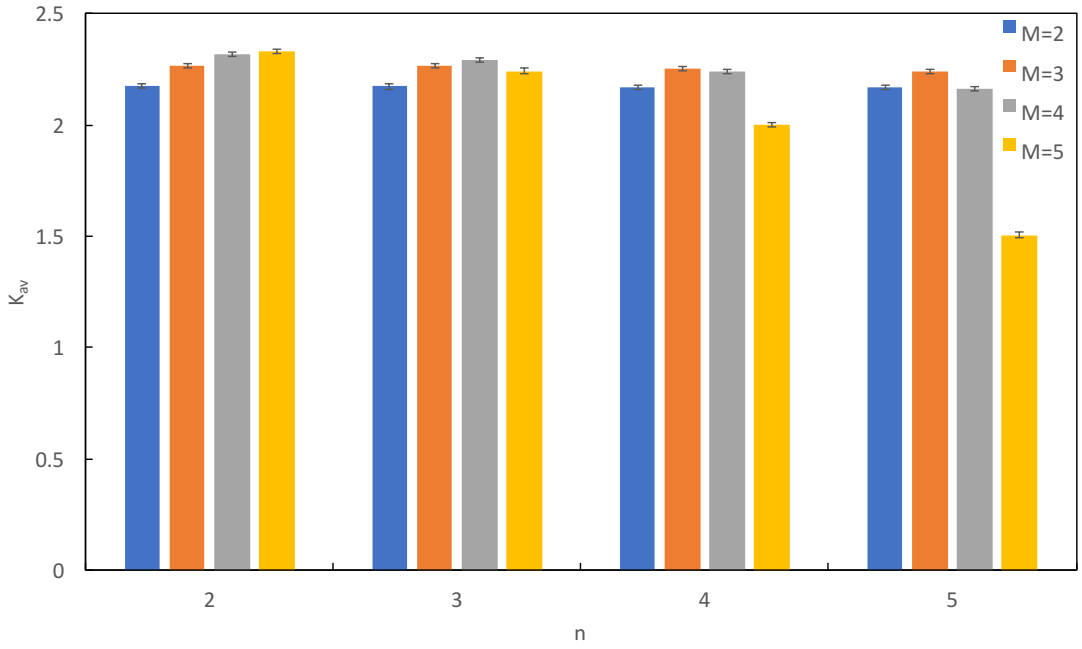


Figure 4.3: The variation of average connectivity in networks containing $2,000$ neurons, with respect to the variations in hierarchical structure. Each group of bars represent networks with a fixed value of n, and variation of M from 2 to 5. As the network size is increased from $1,000$ to $2,000$, the impact of initial nodes in these networks is lesser than the previous set of networks. The average connectivity falls from 2.2 to 2 in networks with $n = 4$ when M increases from 4 to 5.

not as dramatic as in networks with $1,000$ nodes, considering networks with $n = 4$. However, when $n = 5$ and M increases from 4 to 5, the average connectivity falls from 2.1 to 1.6.

Lastly, Fig. 4.4 shows this variation for networks containing $5,000$ neurons. All these networks had an average connectivity of greater than two. However, decreases in K_{av} from 2.3 to 2.0 were observed in networks with $n = 5$, when M increased from 4 to 5.

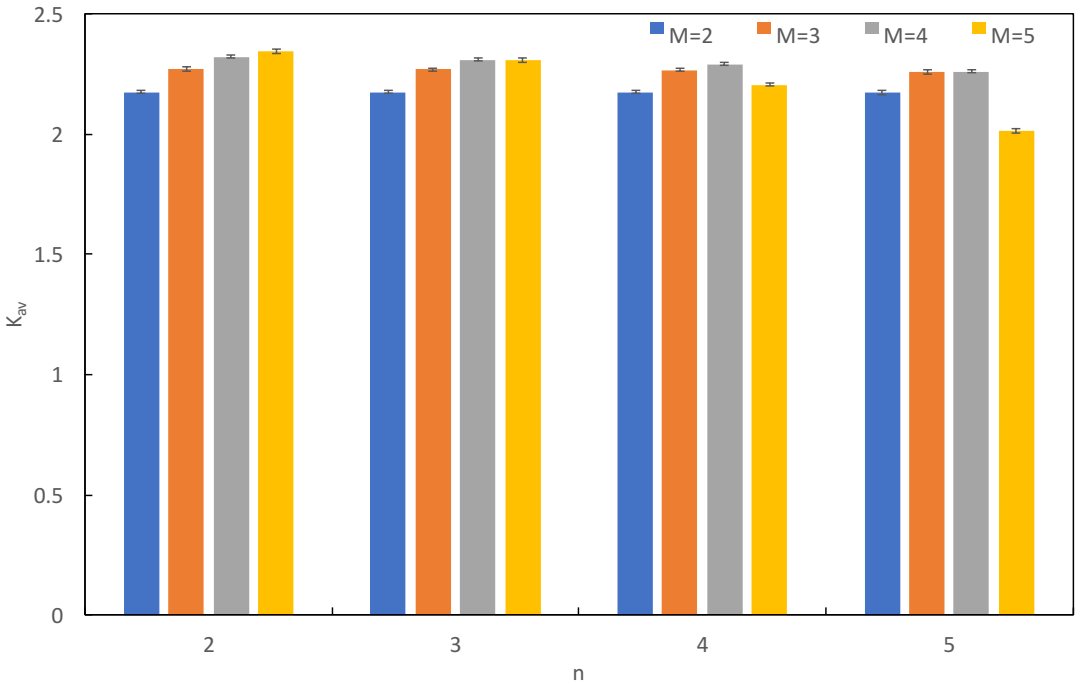


Figure 4.4: The variation of average connectivity in networks containing $5,000$ neurons, with respect to the variations in hierarchical structure. The negative impact of increasing initial nodes on the average connectivity was further reduced as the networks were grown to a size of $5,000$ nodes.

4.1.3 Discussion

The networks built using the growth model described in Chap. 3 have an average connectivity close to two.

In networks with $1,000$ nodes whose average connectivity is shown in Fig. 4.2, it was seen that the average connectivity decreases by up to 0.6 for networks with $n = 4$, when M is increased from 3 to 5. In the networks built using the growth algorithm, the networks were initialized by populating each module with two fully connected nodes before growing them to a desired size. Thus, each network had $2 \times n^{M-1}$ initial nodes, and therefore their average connectivity was one. Hence, if these initial nodes increase to be a significant fraction of the final network size,

then the average connectivity of the networks will be closer to one.

For example, in networks with $n = 2$ and $M = [2, 5]$, the maximum number of initial nodes in the networks is $2 \times 2^{5-1} = 32$. This number is only 3.2% of the final network size of $1,000$, and hence, in these networks, the average connectivity is determined mainly using the in-module connection probabilities. As these probabilities were set to at least 0.9, the resulting networks had an average connectivity of more than two. On the other hand, in the case of networks with $n = 4$ and $M = [3, 5]$, the maximum number of initial nodes is $2 \times 4^{5-1} = 512$. This is about 50% of the final network size, and hence it was seen that the average connectivity was about 1.7 in networks with $n = 4$ and $M = 5$.

The networks with $n = 5, M = 5$ were not grown because the number of initial nodes in these networks, i.e., $2 \times 5^{5-1} = 1250$ exceeded the desired network size of $1,000$. Hence, these networks had an average connectivity of one.

However, in larger networks whose average connectivity values are shown in Fig. 4.3 and Fig. 4.4, the impact of the initial nodes is lesser on the average connectivity than in networks with only $1,000$ nodes. The general observation from these results was that the average connectivity of the network begins to decrease once the number of initial nodes become more than 10%.

Finally, in the case of networks with $5,000$ nodes, it can be observed that the value of K_{av} increases from 2.18 to 2.35 in networks with $n = 2$, as M increases from 2 to 5. As the number of hierarchical levels increases, the probability of between-modules connections also increases, as described in Chap. 3.

However, this also results in more initial nodes, which leads to a decreased average connectivity, as discussed in the case of networks with $1,000$ nodes. Therefore, a trade-off was observed, K_{av} increases for networks with $n = 2$ as M increases

from 2 to 5, but it first increases from 2.18 to 2.3 in networks with $n = 4$ as M is increased from 2 to 4. However, further increasing M to 5 results in a decrease in the average connectivity.

4.2 Experiment 2: Variation of Modularity

The between-modules connections play the role of transmitting information between the communities of strongly connected neurons in the reservoir. If such connections are not enough in number, the information between these communities can not be exchanged optimally. The goal of this experiment was to identify how the modularity factor, i.e., the ratio of the between-modules connections to the total number of connections present in the network, varies in the different types of hierarchical networks used in this research.

4.2.1 Methodology

In this experiment, the networks containing $1,000$, $2,000$ and $5,000$ nodes were studied, because these networks were used in various experiments, which are described in the later sections of this chapter.

The setup used in this experiment was identical to the one used for measuring the variation in average connectivity of different hierarchical networks as described in the first experiment.

Instead of accumulating and averaging the average connectivity of $10,000$ network samples for each of the hierarchical networks with $n = [2, 5]$ and $M = [2, 5]$, their modularity was measured. In this experiment also, the networks were grown to the desired size, and the ratio of the between-modules connections to the total number of connections was measured without perturbing the network with any external input signal.

Fig. 4.5 shows a flow chart containing the sequence of steps following in measuring the average modularity of the 16 hierarchical networks.

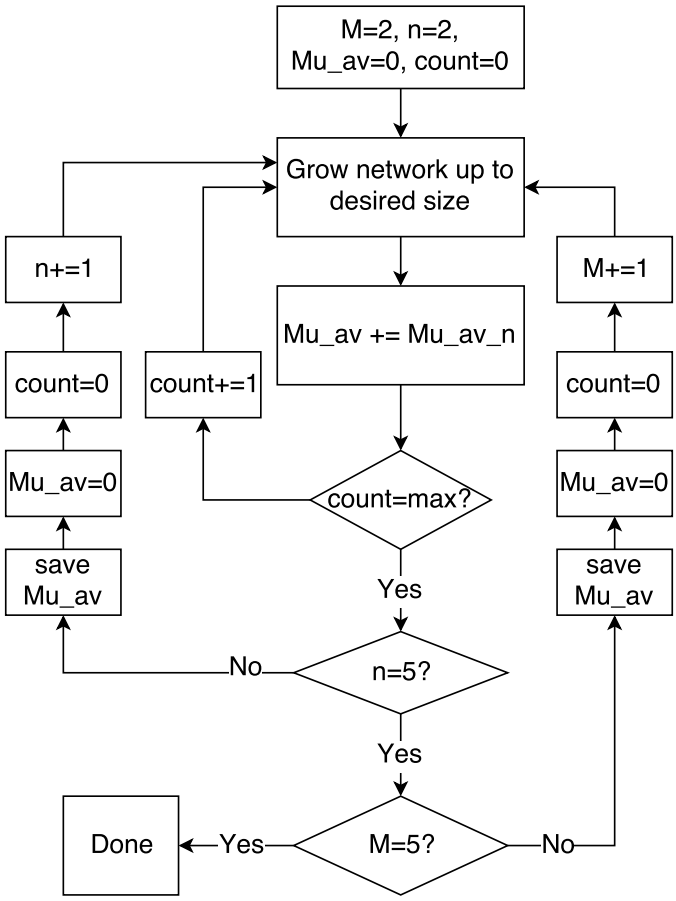


Figure 4.5: A flow chart describing the procedure followed to evaluate the modularity (Mu_{av}) of different hierarchical networks. In this experiment, the value of max was set to $10,000$. Mu_{av-n} represents the modularity of the n^{th} network sample.

4.2.2 Results

Fig. 4.6 shows the variation of average modularity in networks that were grown to a size of $1,000$ neurons. In networks of this size, the average modularity of all networks with only two hierarchical levels is approximately equal to 0.08. The average modularity increases from 0.08 to 0.148 in networks with $n = 2$ as M increases from 2 to 5. Also, in networks with $n = 4$, the average modularity first increases from 0.08 to 0.137 as M increases from 2 to 4. However, further increasing

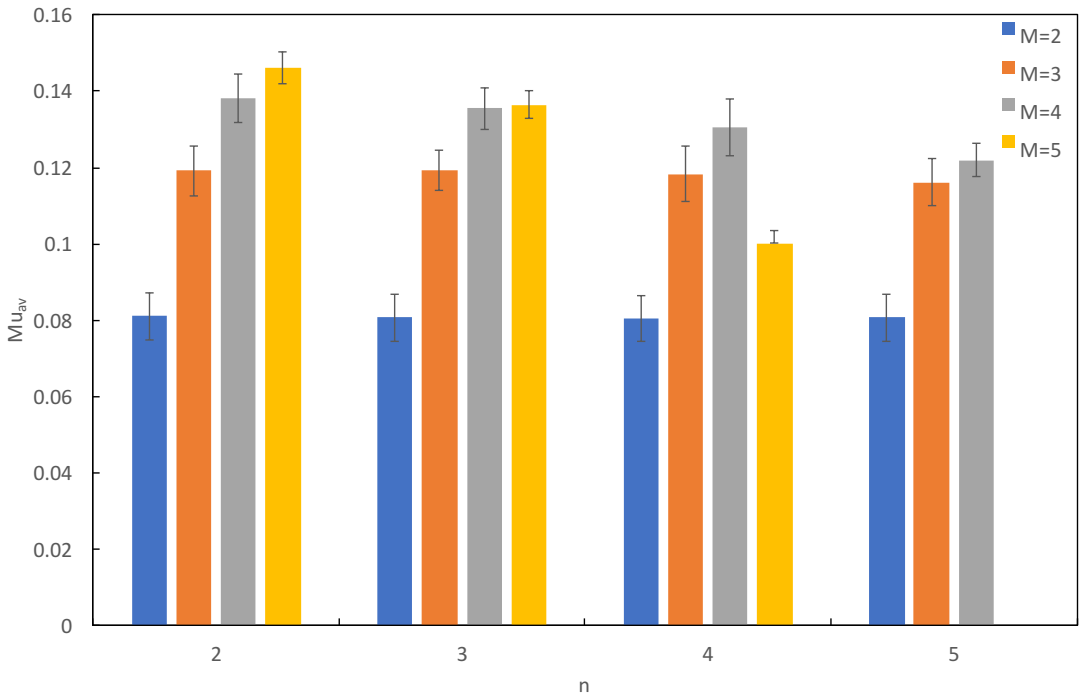


Figure 4.6: The variation of modularity in networks containing $1,000$ neurons, with respect to the variations in hierarchical structure. The error bars represent standard deviation of the average modularity. The networks with $n = 2$ and $M = 2$ had least between-modules connections, i.e., a modularity of about 0.08, and the networks with $n = 2$ and $M = 5$ had the most between-modules connections, i.e., a modularity of 0.144.

M to 5 results in a decreased modularity of 0.1. The modularity of networks with $n = 5, M = 5$ was zero because the number of initial neurons in these networks, i.e., $2 \times 5^{5-1} = 1250$ was more than the desired network size of $1,000$, and hence no network growth operations were performed in these networks.

Fig. 4.7 shows the variation of average connectivity for networks grown up to $2,000$ neurons. In these networks, when $n = 4$, the decrease in modularity is less than that in networks with $1,000$ nodes. However, the modularity decreases from 0.14 to 0.09 in networks with $n = 5$, when M is increased from 4 to 5.

Lastly, Fig. 4.8 shows the variation of modularity in networks grown up to a

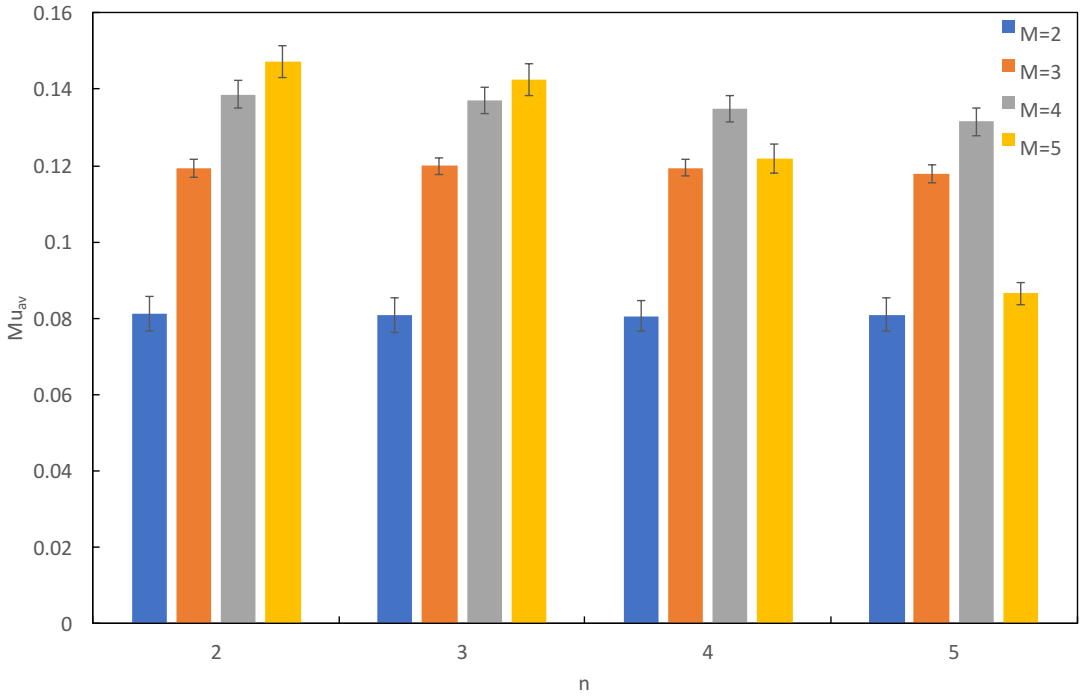


Figure 4.7: The variation of average connectivity in networks containing $2,000$ neurons, with respect to the variations in hierarchical structure. Each group of bars represent networks with a fixed value of n, and variation of M from 2 to 5.

size of $5,000$ neurons. In these networks, the rate of increase in modularity as M increases from 2 to 5 was found to be inversely proportional to M. For example, in the networks with $n = 2$, the modularity increases by 0.04 when M is increased from 2 to 3. However, it only increases by 0.02 when M is further increased to 4. This increase is further reduced to 0.008 when M is increased to 5.

4.2.3 Discussion

In this experiment, it was found that the fraction of between-modules connections varied when the network hierarchy was varied, as shown in Figs. 4.6 to 4.8. For the networks with $n = 2$ in Fig. 4.6, it is clear that the modularity is proportional to the number of hierarchical levels. As M increases from 2 to 5, the modularity

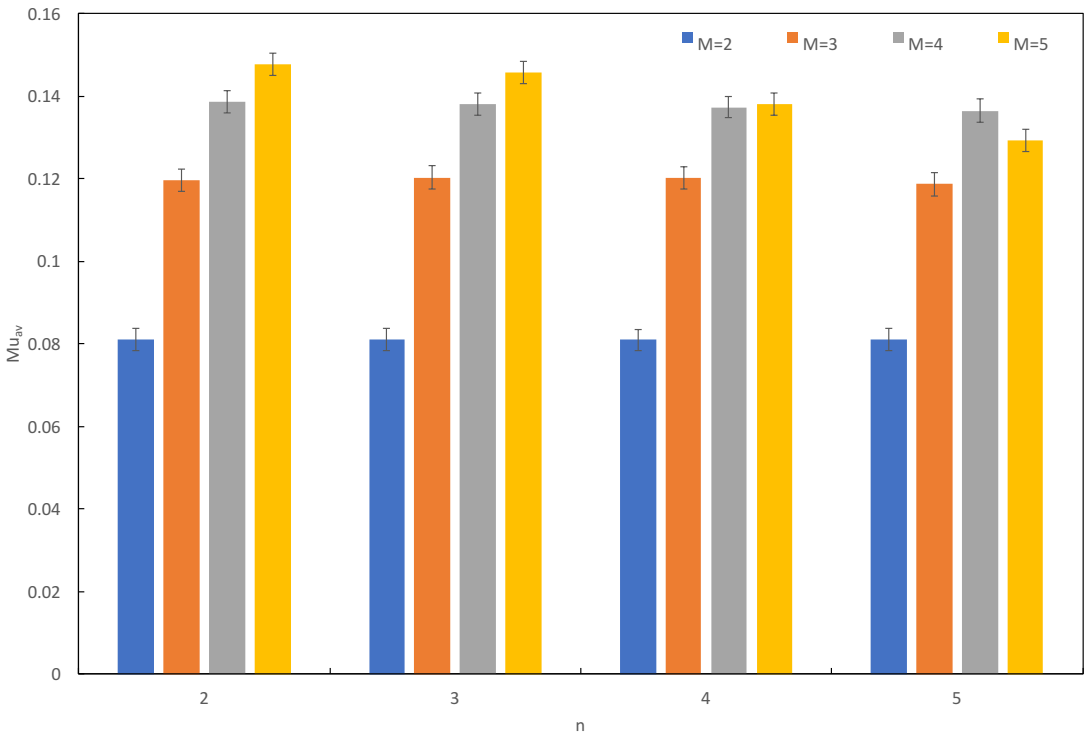


Figure 4.8: The variation of modularity in networks containing $5,000$ neurons, with respect to the variations in hierarchical structure. Each group of bars represent networks with a fixed value of n, and variation of M from 2 to 5.

increases by up to 0.068. This is because the probability of making between-modules connections in the networks, as they are grown, increases with increasing hierarchical levels. Even though the number of initial nodes in these networks increases from 2 to 32, these are only up to 3% of the final network size. Hence, the negative effect of these initial nodes, as was seen in the case of average connectivity is not significant.

However, when the number of initial neurons in the network become a significant fraction of the final network size, the modularity begins to decrease. For example, in networks with $n = 4$, the average modularity drops by up to 0.04 when M is increased from 4 to 5. The number of initial nodes in these networks

increases from $2 \times 4^{4-1} = 128$ to $2 \times 4^{5-1} = 512$. The latter networks have about 50% of the total nodes in the form of initial nodes. The modularity of all these networks initially is zero, as no connections are made between the modules during initialization. Hence, as the fraction of initial nodes increases, the modularity of the network tends to decrease.

However, if the final network size is increased to $5,000$ nodes, as in the case of Fig. 4.8, it can be seen that even though the modularity increases with increasing n, it also begins it decrease from 0.148 to 0.138 when the number of hierarchical levels are increased from three to four and n is increased from three to four. This is due to the fact that the number of initial neurons follows the equation, $2 * n^{M-1}$, and hence, the increase in M leads to an exponential increase in the number of initial neurons. Therefore, the modularity tends to decrease in such cases, where the number of initial neurons in the network starts to increase exponentially.

It was also observed that the range of modularity is dependent on the in-module connection probabilities, and we can see from the above three plots, all of which have the same range of modularity of 0.08 to 0.015.

4.3 Experiment 3: Activity in Hierarchical and Monolithic RBNs

The networks built in this research were closed systems, and did not have input and output ports. Hence, additional input neurons were added to perturb these networks. These input neurons were connected to only a fraction of the reservoir neurons instead of connecting them to all the reservoir neurons, as it may not be always be feasible to perturb all the reservoir neurons with the input signals. For example, to adhere to power consumption limits when fabricating these systems, the amount of wiring required can be reduced by connecting the input signals to only a fraction of the reservoir neurons.

The goal of this experiment was to determine the percentage of nodes to be perturbed with an external input signal, in order to achieve optimal activity in the network. In this experiment, the activity of an RBN was considered as the sum of the total number of times each node changes its state for a given duration of an input signal. This is intended to measure the extent to which an input perturbation propagates inside the RBN, and how different hierarchical RBNs react to an input perturbation. Using the results of this experiment, it was possible to conclude that a hierarchical reservoir can be optimally perturbed by connecting the input signal to only a fraction of the reservoir neurons.

4.3.1 Methodology

In this experiment, different hierarchical RBN reservoirs were perturbed by a single input signal, connected to a varying percentage of reservoir neurons (0 - 70%). To represent this input signal, one input neuron was added to the RC system, and it was connected to a fraction of the neurons present in the RBN reservoirs as shown in Fig. 4.9. These neurons were selected using a uniform probability distribution.

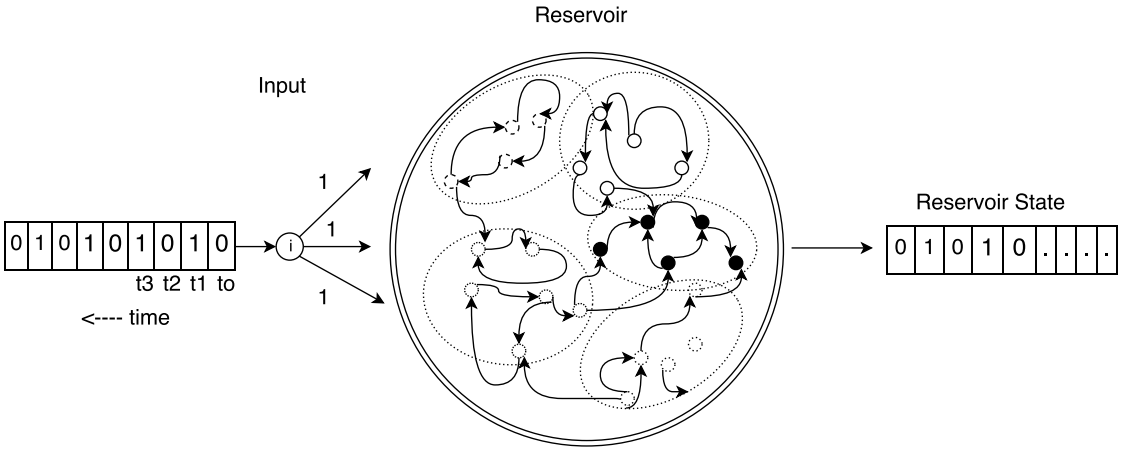


Figure 4.9: The set up used for determining the number of reservoir neurons to be perturbed by an input signal. A weight of 1 indicates the existence of a link between the input neuron and a reservoir neuron, and a weight of 0 indicates its absence.

The RBN reservoirs were perturbed by changing the state of this input neuron, and in order to have maximum possible activity in the input signal, an alternating sequence of zero and one, i.e., 0101010101..., was used as the state of the input neuron.

The state of the input neuron was updated once every time step. The state of each neuron in the network was updated simultaneously, and once every time step. The concatenation of the state of all the neurons was considered as the reservoir state. The activity in the reservoir, due to the input perturbation was measured as the hamming distance between the reservoir state at the end of the previous time step, and the reservoir state at the end of the current time step. This activity was accumulated over the duration of the input signal, and then divided by this duration to obtain the average activity for the given input perturbation sequence. After each perturbation step, a reservoir reacts to the perturbation by changing its state.

For each of the examined hierarchical RBNs, networks with $1,000$ different

network configurations of the same hierarchical structure were created using the growth algorithm as described in Chap. 3. Each of these networks were simulated using 100 different random initial states. Similarly, monolithic RBNs, obtained by setting the number of hierarchical levels to 1, were also evaluated in order to compare their activity with that of hierarchical RBNs.

All the networks were grown to a size of $5,000$ neurons, before perturbing them with the input sequences of 01010101.. for 100 time steps. The average activity of all the network samples was also normalized at the end by dividing it with the network size and expressed as a percentage of the network size. In this experiment, networks with $M = 2$ and $M = 3$ networks were studied.

4.3.2 Results

In Fig. 4.10, the variation in the average activity with respect to the fraction of reservoir neurons connected to the input neuron is shown for networks with two hierarchical levels. In this plot, each set of columns represent the average activity of networks with n increasing from 2 to 5 for a fixed fraction of reservoir neurons connected to the input neuron. It can be seen that the average activity in networks with $n = 2$ increases from 20% to 43% as the percentage of reservoir neurons that are perturbed with the input signal increases from 0% to 20%. As this percentage is further increased from 20% to 70%, the rate of increase in the average activity is very less, and the maximum activity is only about 45%. Hence, the average activity increased by only 2% even when the fraction of perturbed reservoir neurons increased to 70%.

Similar trends can be observed for networks with other values of n. Also, the average activity in a network increases as the number of modules increases.

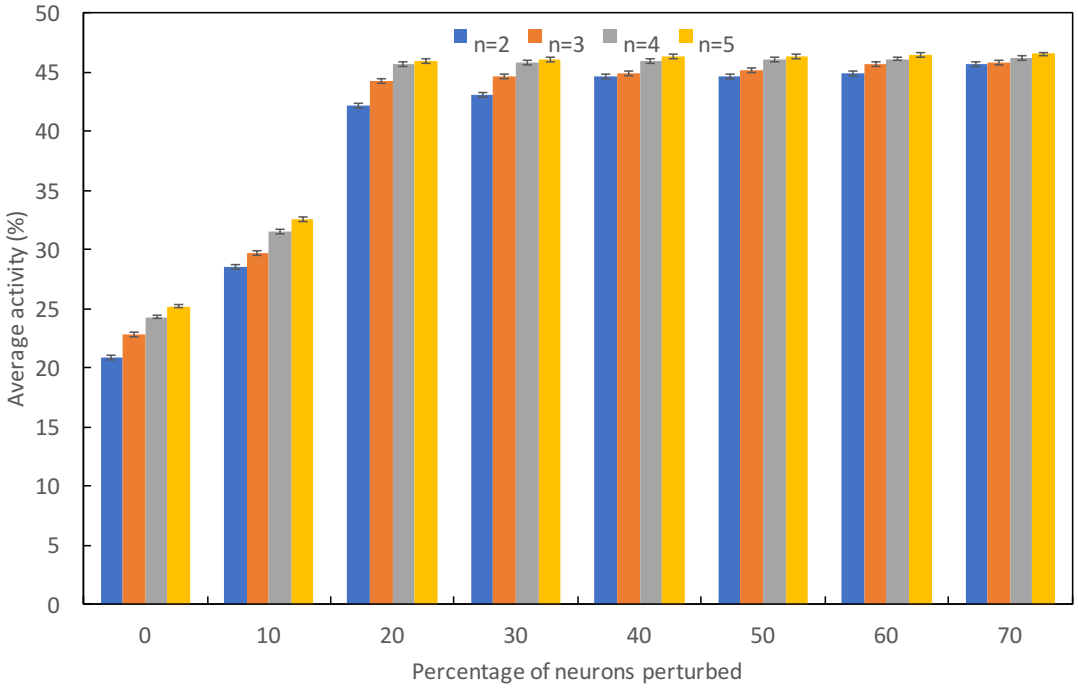


Figure 4.10: A plot showing the variation of average activity in networks with $M = 2$ hierarchical levels, and $n = [2, 5]$; the activities given in each column are the average of 100,000 networks with the same network parameters but different configurations and random initial states. The error bars represent standard deviation.

For example, when only 10% of the reservoir neurons are perturbed, the average activity increases from 28% to 33% when n is increased from 2 to 5.

The variation of activity with the percentage of neurons connected to the input signal, for networks with different values of n from 2 to 5 with $M = 3$ hierarchical levels is shown in Fig. 4.11. An average activity of at least 50% was observed when 20% of the reservoir neurons were perturbed with the input signal. Also, the average activity did not improve by more than 5% when the number of perturbed reservoir neurons was increased from 20% to 70%.

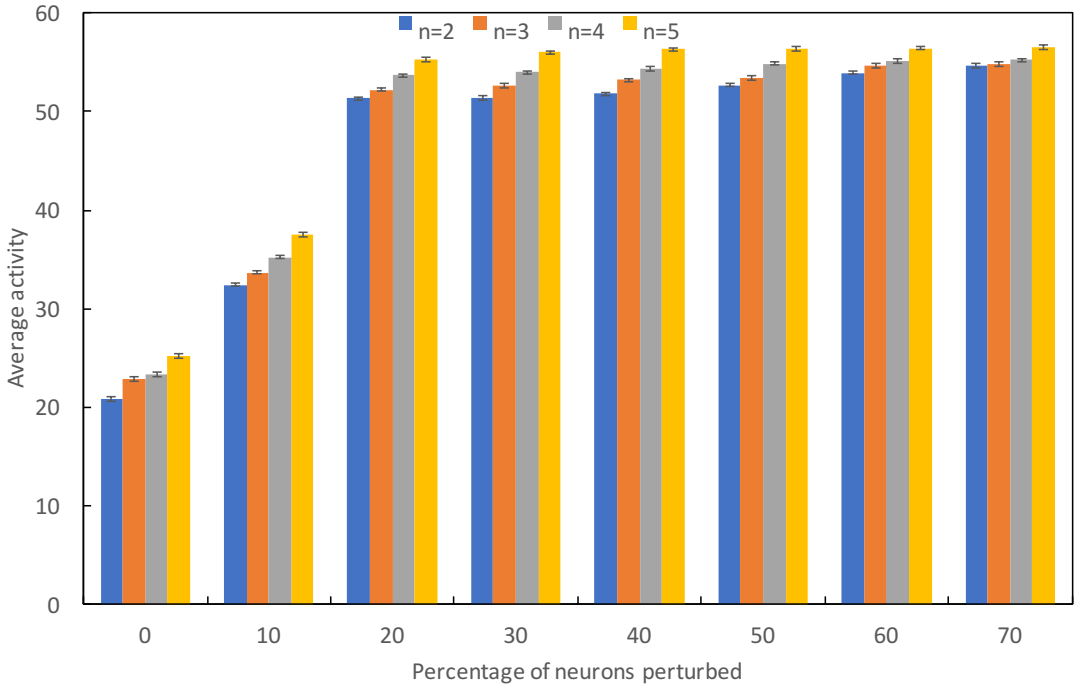


Figure 4.11: This plot shows the variation of average activity in the networks with $M = 3$ hierarchical levels. It can be seen that the average activity does not increase by more than 5%, once the percentage of neurons perturbed increases beyond 20%.

4.3.3 Discussion

In Fig. 4.10, it can be seen that the activity increases as n increases from 2 to 5. It was shown earlier in Fig. 4.4 that the average connectivity of the networks also increases from 2.18 to 2.35 as n increases from 2 to 5 in networks with two hierarchical levels. These RBN networks approach the chaotic regime as their average connectivity increases, and hence, we can see an increased activity.

From the above results, we can observe that the average activity is proportional to n when the percentage of neurons connected to the input signal is increased from 0 to 20%. However, the average activity does not increase by more than 5% beyond this point, and hence perturbing 20% neurons with the input signal was enough to result in optimal activity in these hierarchical networks.

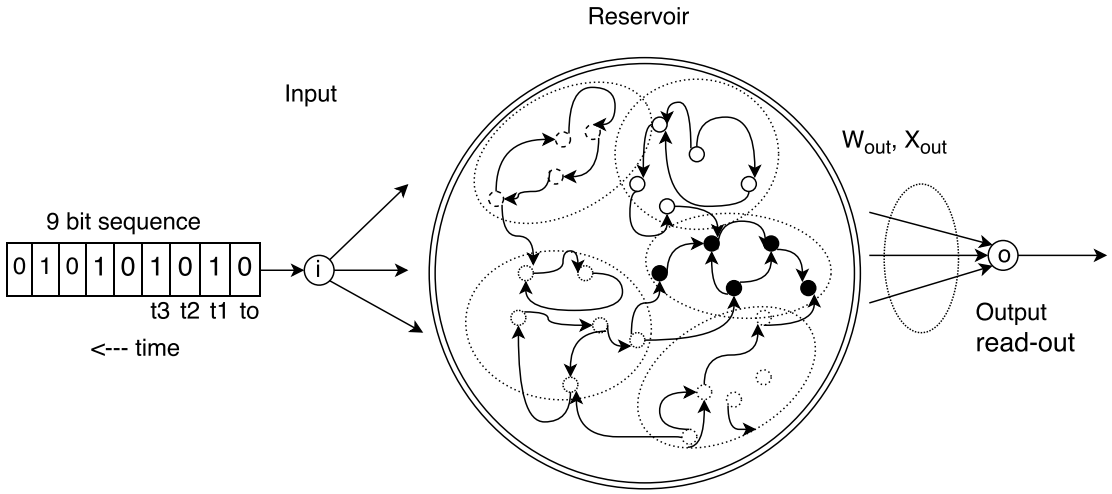


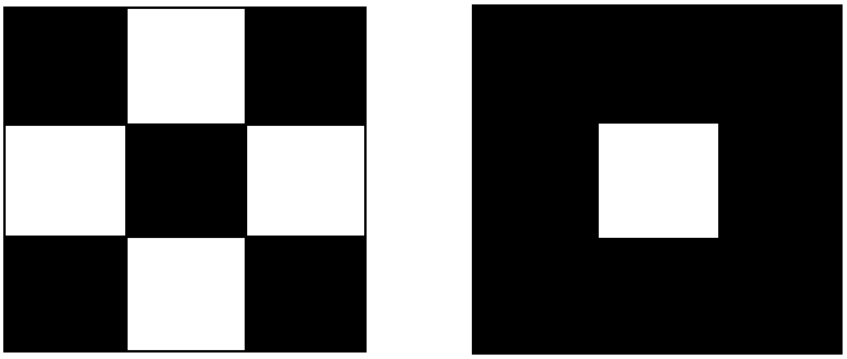
Figure 4.12: The RC system set up used to evaluate the temporal pattern detection task. A single input signal was used to perturb each of the hierarchical networks, and the networks were trained using a single read-out neuron. The output was a simple 'Yes' or 'No'.

4.4 Experiment 4: Temporal Pattern Recognition Task

The aim of this experiment was to train these networks to identify whether an input data sequence represented an X pattern or not when the input is presented sequentially to the network. To accomplish this task, a reservoir must be capable of remembering the entire sequence and then make a decision whether the input sequences represents a particular pattern or not.

4.4.1 Methodology

As was the case with the first experiment, for each of the examined hierarchical RBNs (1 to 5 hierarchical levels, and 2 to 256 modules), a total of $100,000$ network samples were evaluated, along with monolithic RBNs. The in-module connection probabilities of the hierarchical levels in all these networks were at least 0.9 in order to realize networks with average connectivity close to two.



(a) A 3×3 grid of 9 pixels representing an X pattern

(b) A 3 × 3 grid of 9 pixels representing an O pattern

Figure 4.13: Two example patterns used for the temporal pattern recognition task, the black pixels indicate the shape of the pattern.

The experimental setup used for this task is shown in Fig. 4.12. In this setup, an input neuron, i, was used to perturb 20% of the reservoir neurons. An output read-out neuron, o, was used to train these networks by connecting the outputs of 10% of the reservoir neurons (X_{out}) to it, using links with an equal number of weights, W_{out}.

As Boolean logic gates were used as neurons in these reservoirs, and because such neurons operate on binary inputs, the input patterns were encoded as sequences of $1s$ *and* $0s$. Two example input images representing an X and an O were encoded as the two 3×3 grids as shown in Fig. 4.13. These patterns were encoded as 2 sequences of 9 bits each. In these patterns, black pixels were considered as 1s, and white pixels were represented by 0s. These were translated into 3×3 matrices accordingly, as shown in figure 4.14.

These 3×3 matrices are converted to a sequence of nine bits to perturb the RBN reservoirs sequentially. For example, the sequence representing an X was encoded as 101010101 and the sequence representing an O was encoded as 111101111. There

1	0	1
0	1	0
1	0	1

(a) A 3×3 matrix of 9 elements representing an X pattern

1	1	1
1	0	1
1	1	1

(b) A 3×3 matrix of 9 elements representing an O pattern

Figure 4.14: The two matrices for X and O patterns.

were a total of 512 unique sequences that could be encoded using these 3×3 matrices. The networks were trained using these sequences, and later evaluated using a fraction of these 512 sequences.

These input sequences were fed to the reservoirs at the rate of one bit per time step, through the input neuron i as shown in Fig. 4.12. In order to allow a change in the input signal to propagate through the reservoir, it was clamped to a constant value for 50 step simulation window. The final state of each neuron connected to the read-out was considered to be a one if there were a majority of ones in the sequence of outputs of the neuron during the simulation window, and zero otherwise.

About 10% of the reservoir neurons were connected to a single read-out neuron for training the system to solve this task. The read-out neuron was a simple linear combiner with a threshold output, as described in the *Artificial Neurons and Weights* section of Chap. 2. The output of this neuron would be *one* if the input is greater than zero, and *zero* otherwise. An output of *one* was considered to indicate that the input sequence represents an X pattern.

Linear gradient descent, with a learning rate of 0.001, was used to train the read-out neuron. Initially, the training was begun by setting the classification error to zero. In each epoch of training, 99 patterns were randomly picked from the set of 512 possible patterns, and added to the training set containing the sequence corresponding to the X pattern. The reservoir was perturbed with each of the sequences present in the training set sequentially. After simulating the reservoir for 9 time steps, to complete one pattern, if the output of read-out neuron did not match the expected output for that pattern, this error was incremented by one. The network was reset to its initial state, before perturbing it with the next sequence. This error was accumulated for all the 100 training sequences, and divided by 100 and saved after every epoch. The weights were updated after every epoch, such that the average error moves closer to the minimum value of zero. The training was aborted if either the average error was minimized to 0, or if the average error did not decrease by at least 1% over a period of $10,000$ training epoch. The maximum number of training epochs for training each network was set to $100,000$.

The trained network was then evaluated using the test set, containing 99 sequences that were randomly selected from the possible 512 sequences, along with a sequence corresponding to the X pattern. The accuracy of this network was obtained by subtracting the average error for the 100 sequences from one, which indicates the percentage of input patterns that were correctly detected by the network.

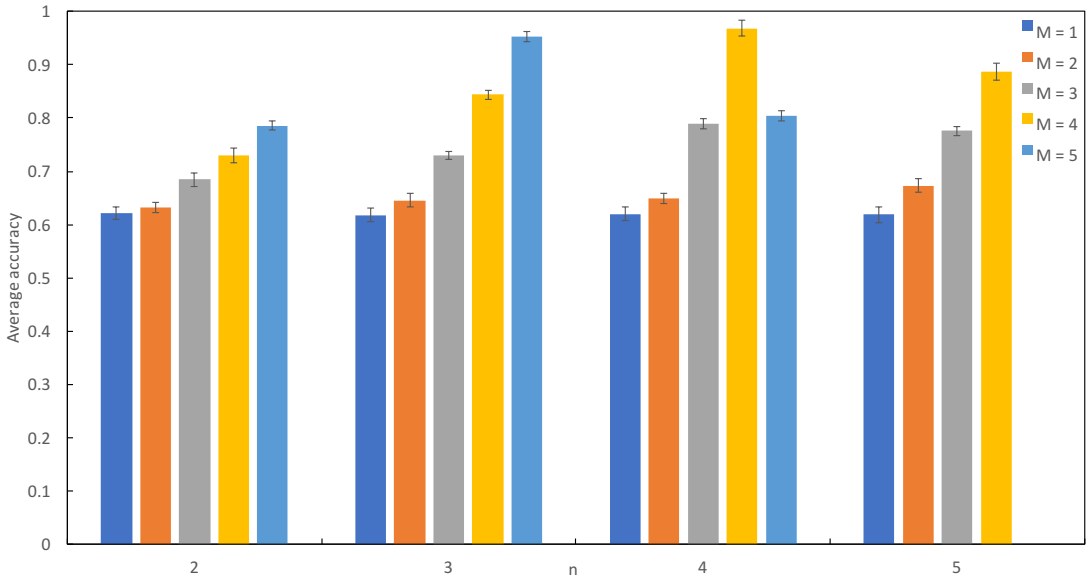


Figure 4.15: The average performance of different hierarchical networks, with $100,000$ networks samples each,is plotted against the network parameters M and n. The accuracy measure indicates the fraction of input test patterns that were correctly detected by these networks. The error bars show the standard deviation. The networks with $n = 5, M = 5$ were not evaluated because the number of initial nodes in these networks ($2 * 5^{5-1} = 1,250$) exceeds the maximum network size for this experiment $(1,000)$.

4.4.2 Results

The results of this experiment are shown in Fig. 4.15. In this bar plot, the Y-axis represents the normalized accuracy, and the X-axis represents combination of network parameters (n, M), which denote networks having M hierarchical levels and n sub-modules. The error bars represent the standard deviation of the accuracy from the average. This shows that there were some networks that performed better than the average and also some that performed worse than the average.

4.4.3 Discussion

The following observations were made from Fig. 4.15:

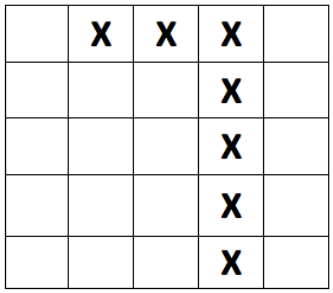
- All the hierarchical networks performed up to 36% better than the monolithic networks. These results show that our monolithic RBNs had a limited capability when it comes to temporal pattern detection.

- The performance increases by more than 30%, from 0.6 to 0.94, as the number of hierarchical levels in these networks are increased from one to five, as seen in the case of networks with $n = 3$. This is due to the fact that the number of between-modules connections are more in networks with more hierarchical levels than those with fewer hierarchical levels, as shown in Experiment 2. These connections relay information between modules, there by perturbing nodes that are not directly perturbed by the input signal.

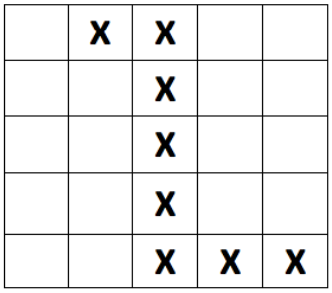
- On the other hand, in Fig. 4.15, in the columns corresponding to $(n, M = 4, 4)$ and $(n, M = 4, 5)$, the accuracy decreases from 96% to 78% even when the number of hierarchical levels are increased. The first network has $4^3 = 64$ modules and two neurons are added to each module during the initialization phase of network growth. Hence, 128 neurons are present initially in the network which is approximately 10% of the total neurons $(1,000)$ that are present in the network at the end of the network growth process. On the other hand, the second network has 256 modules, and hence 512 neurons present initially in the network. This is approximately 50% of the final network size. Therefore, even though the probability of between-modules connections is higher in the latter network, the number of in-module connections is much higher than the number of between-modules connections, and hence there is not enough information exchange between the modules in this case, which leads to reduced performance [24].

- The best performing networks had 4 hierarchical levels and 64 modules and achieved an accuracy of close to 96%. This was closely followed by hierarchical networks having 5 hierarchical levels and 91 modules, which achieved an accuracy of close to 93%.

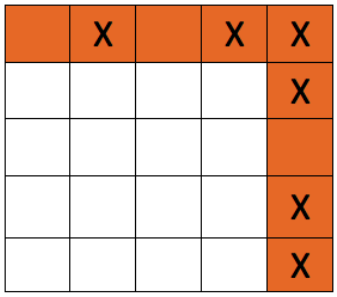
- Finally, even though having more number of hierarchical levels seems beneficial for this experiment, as can be seen in the case of networks with $n = 2$ and $n = 3$ in Fig. 4.15, it is not desirable for networks where the initial number of nodes is more than 50% of the total network size, which is the case in networks with $n = 4$.



(a) A grid with one turn in the continuous food



(b) A grid with two turns in the continuous food trail.



(c) A grid with one turn and two gaps in the food trail.

Figure 4.16: Three 5×5 trails with different complexities that were used in the first part of this experiment. The food pellets are indicated by an X. The trail in (c) has multiple possible paths an agent can take to successfully reach all the food pellets. The optimal path which was used as a reference to measure the performance of the networks, is highlighted in orange in this figure.

4.5 Experiment 5: Food-foraging Tasks

Another application of machine learning is in building systems that can be trained to control the motion of an autonomous robot. For example, a robot can be trained to avoid obstacles on its path and reach its intended destination. One such class of tasks is food-foraging. In this task, an agent is deployed in a two-dimensional grid containing a trail of food pellets, and trained to find an optimal path to consume all food pellets.

4.5.1 Methodology

A total of $100,000$ reservoir network samples were evaluated for each of the different hierarchical networks obtained by varying the number of hierarchical levels, M from 1 to 4, and the number of modules, determined by n^{M-1}, where n varies from 2 to 6. Food trails with different complexities were used to evaluate all these networks.

This experiment was divided into two parts: initially small 5×5 grids, and later a larger 10×10 grid were used to train and evaluate different hierarchical networks. In each of these grids, a set of three different types of food trails were used. The number of turns and the gaps in the food trails were used to determine the complexity of these trails. Three trails that were used in the first part of this experiment are shown in Fig. 4.16. Fig. 4.16(a) shows a simple 5×5 grid that has a path containing seven food pellets with one turn in the path. Figs. 4.16(b) and 4.16(c) show more complex trails with more turns, and added gaps in the food trail respectively.

The aim of the experiment is to investigate whether hierarchical networks perform better than monolithic networks, in directing the agent to consume all the food pellets in the most efficient way possible.

In this experiment, the agent was controlled using the outputs from different types of hierarchical reservoir networks, each of which had one input neuron, and two output neurons as shown in Fig. 4.17. These reservoirs were perturbed with an input signal from a sensor present on the agent, and the function of this sensor was to generate a binary input signal based on the presence of a food pellet in the cell immediately in front of the agent. The value of the input neuron was updated every time after the agent moves to a new cell in the grid.

Two binary read-out neurons, o_1 and o_2, were used to stimulate the agent and

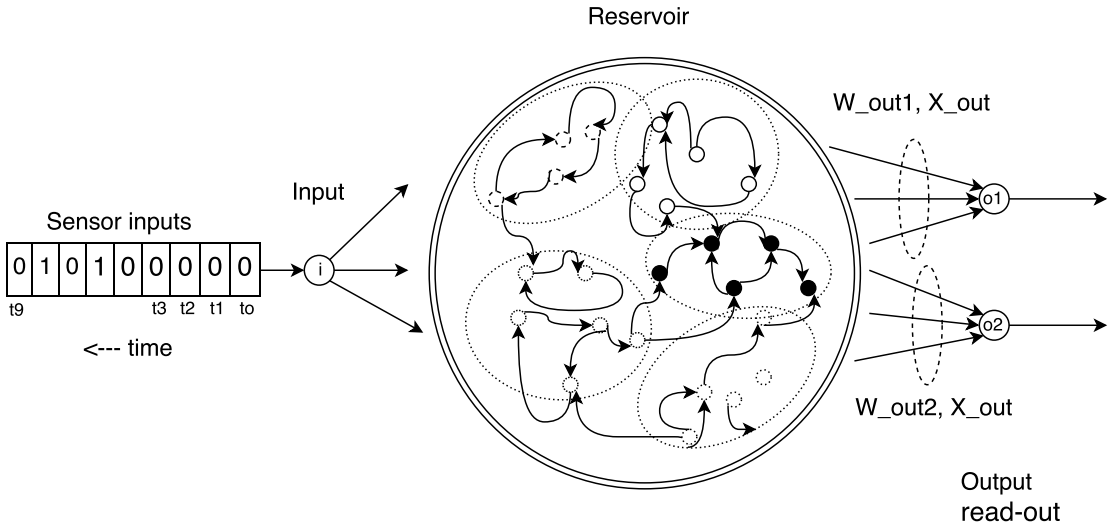


Figure 4.17: The experimental setup used to train and evaluate the RC networks to control an agent, such that it moves in an optimal path to complete the food trail contained in a grid. The state of the neuron i was one if a food pellet was present in the cell immediately in front of the agent, and zero otherwise.

make it move in one particular direction. About 5% of the outputs of the reservoir neurons were connected to both these read-out neurons with two sets of weights, W_{out1} and W_{out2}. These two neurons performed a sequence of two operations: first, they linearly combined a fraction of the outputs of the reservoir neurons with an equal number of trainable weights using Eq. 2.1, and then they performed a threshold operation on these linear combinations to generate binary outputs. A threshold of zero was used for both read-out neurons. If the value of the linear combination was greater than zero, the read-out neuron would generate an output of one, and zero otherwise.

The binary outputs of the two read-out neurons were encoded as shown in Table 4.2, such that each of the four possible combinations corresponds to a particular direction in which the agent moves. This table also shows that the agent moves to a new cell after every single time step. The agent can not move diagonally.

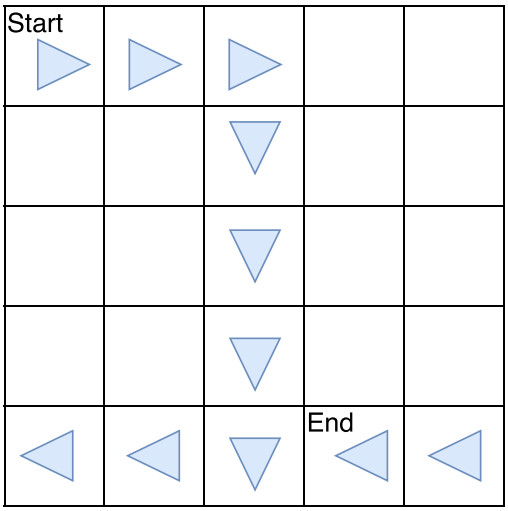
o_1	o_2	direction
0	0	Forward
0	1	Left
1	0	Right
1	1	Forward

Table 4.2: 2-bit encoding of the directions in which the agent can move, o1 and o2 are the two read-outs from the RBN.

The agent used in this experiment had a simple control system that can move either forward, left, or right of its current position. However, it can warp around the grid if it reaches the edge of the grid. The performance of the hierarchical RBN was evaluated based on the number of food pellets it was able to consume and the number of moves taken to find the optimal path. The number of moves an agent could make in each run was limited to restrict the freedom given to the agent in finding an optimal path. For example, the agent was allowed to make a maximum of seven moves in the case of the grid shown in Fig. 4.16(a), because it is a relatively simple trail with a single optimal solution.

A solution was said to be found if an agent consumed all the food pellets using the minimum number of moves required. In each run, the agent was placed in the first cell of the grid, and was allowed to traverse the grid according to the outputs generated by the hierarchical network. If the agent reached a cell containing a food pellet, the agent's score was increased by one and the food pellet was removed from the grid. If the agent's sensor checks the same cell again in the future, it would not find a food pellet there. The agent was reset to its initial position upon consuming all the food pellets or upon making the maximum number of moves allowed.

At the end of each run, the fitness of the path, i.e., solution was evaluated using Eq. 4.1. For example, consider the trail shown in Fig. 4.16(b), and the sample solution of an agent, at the end of a single run on this trail, as shown in



(a) The path followed by an agent using a sample hierarchical network, at the end of a run on the trail shown in Fig. 4.16(b). This agent consumed all the eight food pellets using 10 moves.

Move	Agent inputs (o_1,o_2)	Sensor Output
0	(0,0)	1
1	(0,0)	1
2	(1,0)	0
3	(0,0)	1
4	(0,0)	1
5	(0,0)	1
6	(1,0)	0
7	(0,0)	0
8	(0,0)	1
9	(0,0)	1
10	(0,0)	0

(b) Trace of the agent's inputs, and its sensor outputs at the end of each move.

Figure 4.18: A sample run of the agent.

Fig. 4.18(a). In Fig. 4.18(b), the inputs to the agent and its sensor outputs at the end of each move are listed. The agent starts off at move 0 and goes on to make 10 moves before consuming all the eight food pellets. The best possible fitness can be achieved when all the 8 food pellets were consumed using only 8 moves. Although the agent was able to consume all the food pellets, it made 10 moves, and hence the fitness value for this run was determined to be $8/10 = 0.8$.

$$fitness = \frac{\text{number of food pellets consumed}}{\text{number of moves used}} \tag{4.1}$$

A genetic algorithm was used to train the read-out neurons instead of linear gradient descent, because the latter method could not work. This was because there were multiple optimal solutions possible for some of these trails, and there was no unique expected/target output with which the network could be trained. It

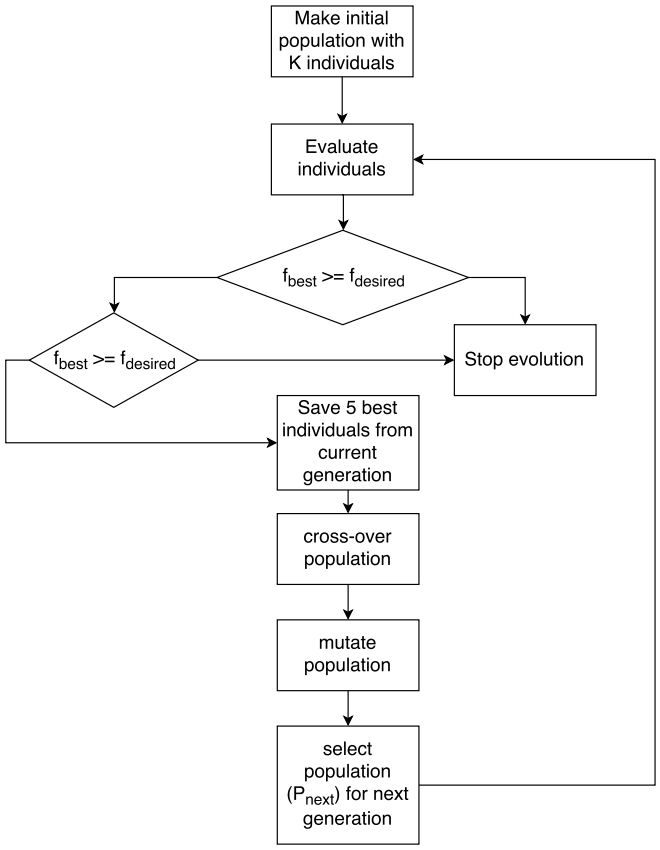


Figure 4.19: The various steps performed, when training the read-out neurons using a genetic algorithm. The algorithm was configured such that training was terminated if the fitness of the best individual in a generation was better than a desired fitness value.

was also interesting to test if an agent is able to find an optimal path autonomously, rather than teaching the agent to follow a particular path desired by the user.

A flow chart with the steps involved in a genetic algorithm is shown in Fig. 4.19. Here, the f_{best} refers to the highest fitness achieved in the population of the current generation that was evaluated. The f_{desired} can be set to the optimal fitness (if known) or another value set by the user. The algorithm was also programmed such that the training would be terminated if f_{best} did not improve by a minimum amount in the past $1,000$ generations. The training was performed for a maximum

number of $10,000$ generations.

The genetic algorithm was implemented using the Python DEAP toolbox [6]. The *eaMuPlusLambda* evolution algorithm was used as the basis to evolve an optimal solution in these experiments. In this algorithm, an initial population with K individual sets of weights is generated randomly. The size of the individual is equal to the total number of weights to be trained for a given problem. In this case, we have two neurons with W_{out1} and W_{out2} number of weights, and hence each individual consisted of $W_{out1} + W_{out2}$ number of randomly generated floating point numbers in the interval $[-5, 5]$. The fitness of each of these individuals were evaluated, and used to evolve a new population to evaluate in the next generation.

The population for the next generation is obtained by performing the two basic genetic operations: cross-over and mutation to produce μ *off-springs*. The population is evolved such that the next generation contains five of the best performing individuals of the current generation, and this is a feature of elitism. This ensures that the best individuals are not completely lost due to mutation and cross-over, during the evolution process. The mutation and crossover probabilities were set to 0.02 and 0.6 respectively, according to the analysis done by De Jong *et al.* [4]. The new population for the next generation contains $K - 5$ individuals picked from the set of (K,μ) individuals along with the five best individuals from the current generation. This then becomes the population of K individuals of the next generation. A population size of 30 individuals was used in each generation. This ensured that the population was large enough to contain a diverse set of solutions. The algorithm was configured to find the best performing sets of weights that maximized the fitness value.

The training would be stopped if either of the following conditions were met:

Parameter	Value
Population size (K)	30
Probability of crossover (p_{cross})	0.6
Probability of mutation (p_{mut})	0.02
Max generations	$10,000$
Minimum fitness value (MFV)	$\frac{1}{\text{max moves allowed}}$
Weight range	$[-5, 5]$

Table 4.3: This table summarizes the various parameters used in the genetic algorithm for training the various hierarchical networks in this experiment. The crossover and mutation probabilities were inspired from the work done by De Jong *et al.* [4]. The maximum generations, Minimum fitness value and the weight range were set to allow ample exploration of the solution space.

- the fitness of the best performing individual in the current generation had reached f_{desired};
- the evolution has been carried out for $10,000$ generations allowed; and
- fitness of the best individual in the past $1,000$ generations did not improve by a minimum amount, MFV.

The minimum fitness value for the last terminal condition was set using the equation, $MFV = \frac{1}{\text{Maximum moves allowed}}$. All the parameters used in this algorithm are summarized in Table 4.3.

This training process was repeated for all the sets of $100,000$ network samples of each hierarchical configuration and their average fitness values were obtained. For the 5×5 trails shown in Fig. 4.16, networks with $1,000$ neurons were used. For the larger 10×10 trail shown in Fig. 4.23, larger networks with $5,000$ neurons were used.

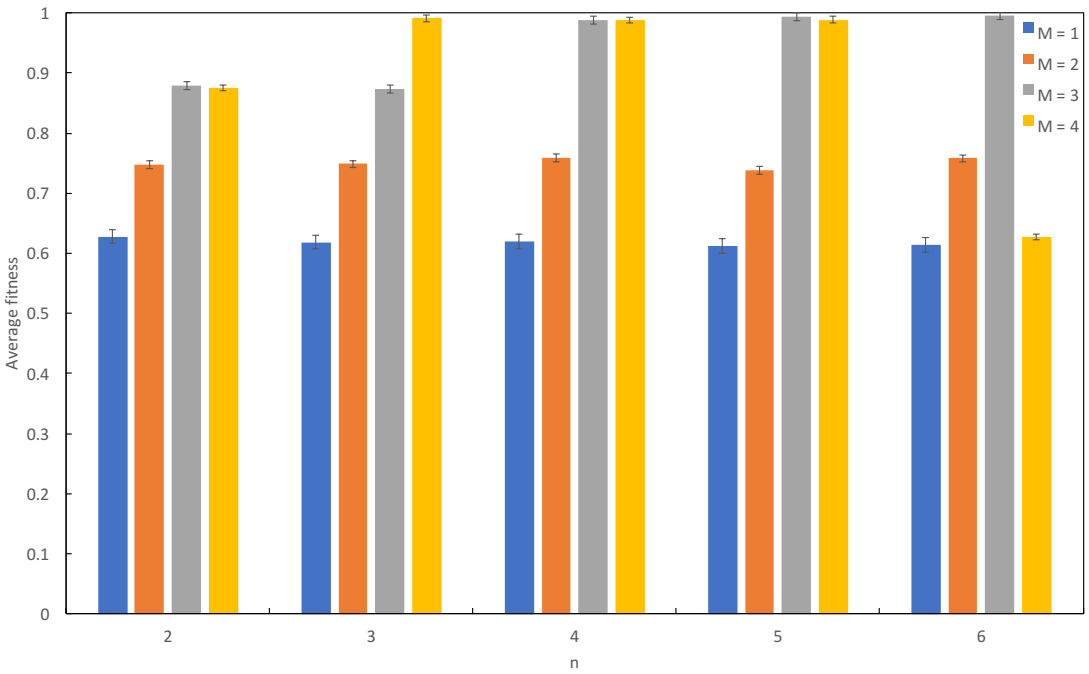


Figure 4.20: The fitness plots for different hierarchical networks evaluated for the simple 5×5 trail task having one turn as shown in Fig. 4.16(a). The error bars indicate the standard deviation of the fitness from the average. Most networks with $n = 4$ and $M = 4, 5$ were able to solve this trail and consume all the seven food pellets by using only seven moves.

4.5.2 Results

The average fitness of networks with $1,000$ neurons when solving the simple 5×5 trail task shown in Fig. 4.16(a) is shown in Fig. 4.20. The maximum moves allowed for an agent in this experiment was set to 8, in order to allow the agent some freedom to explore the grid. A fitness of close to 1 indicates that the trail was completely solved by consuming all the seven food pellets using seven moves. In this plot, it can be seen that monolithic networks are only able to achieve a fitness of 0.625, which corresponds to consuming five food pellets in eight moves. Also, networks with more number of hierarchical levels show improved performance. For example, in networks with $n = 2$, the average fitness increases from 0.6 to 0.875

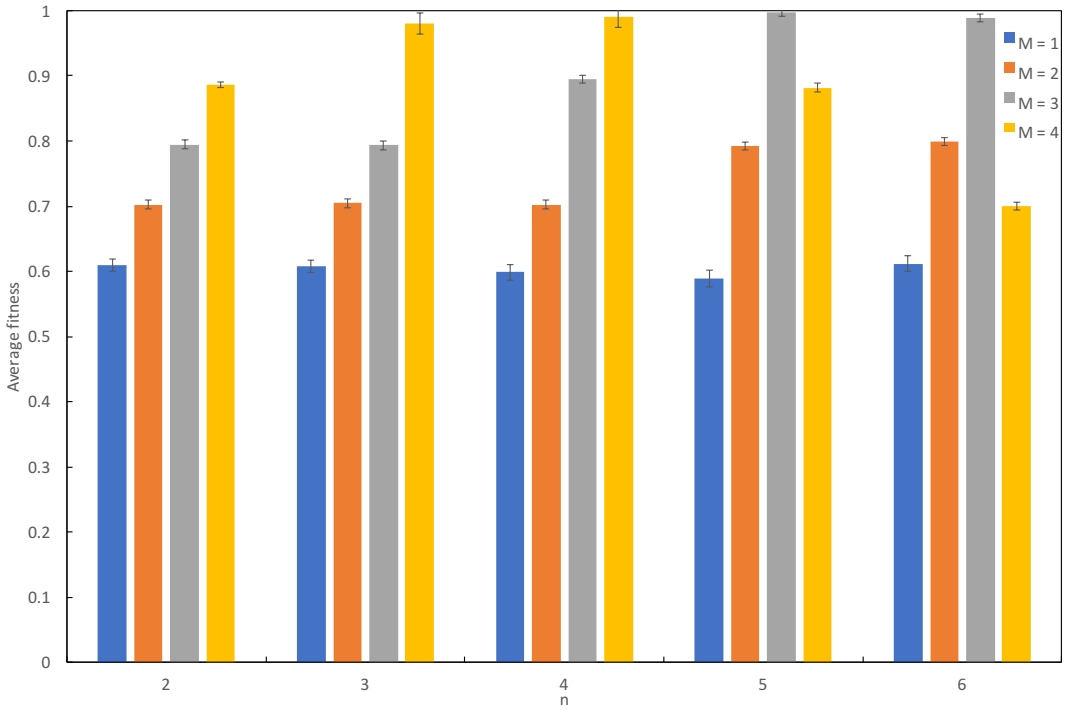


Figure 4.21: The fitness plots for different hierarchical networks evaluated for a complex 5×5 trail task having two turns. The networks with $n = 6, M = 4$ demonstrated a lower performance of 0.7 as they have 432 initial nodes, which is almost 40% of the final network size. Also, the monolithic networks all have one hierarchical level and one module in them, irrespective of the value of n, and hence they all demonstrate average fitness that is similar to each other.

as M increases from 2 to 5.

These networks were then evaluated using a more complex trail task shown in Fig. 4.16(b). In this task, the number of food pellets are increased to eight, and the number of turns is increased to two. The maximum moves allowed for an agent in this experiment was set to 10. The average fitness of different hierarchical networks is shown in Fig. 4.21. All the hierarchical networks are able to perform better than the monolithic networks. The best performing networks are those with $n = 5$, $M = 3$, and most of these networks are able to completely solve this trail. They are able to consume all the eight food pellets using eight moves.

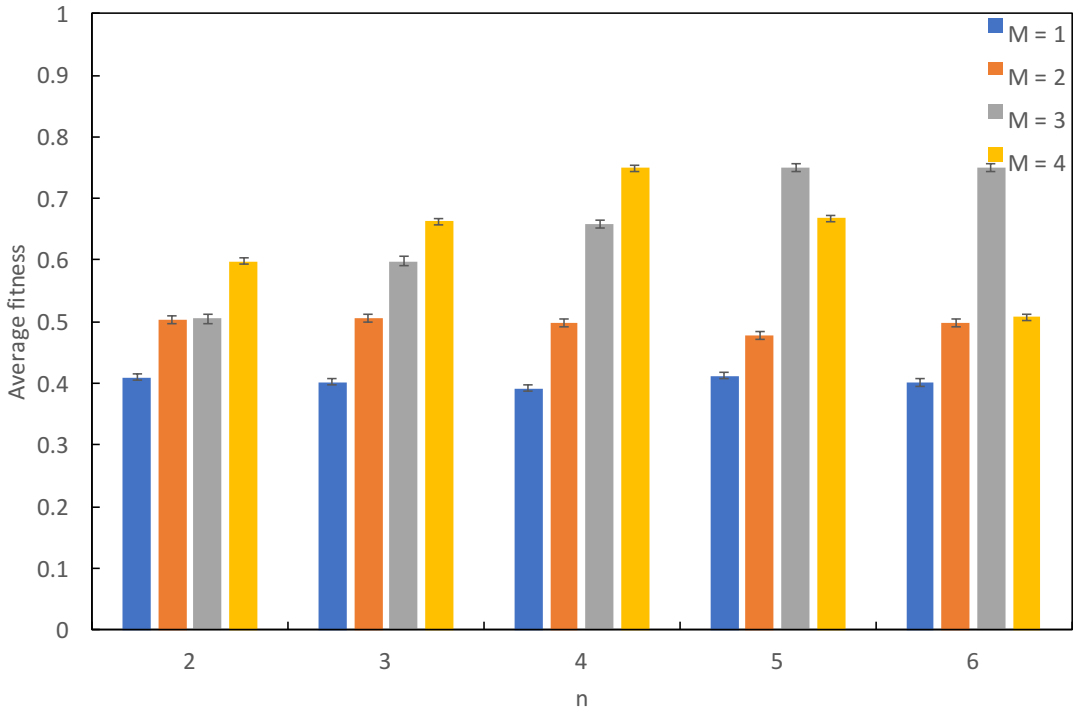


Figure 4.22: The fitness plots for different hierarchical networks evaluated for the complex 5×5 trail task having one turn, and two gaps in the food trail. The maximum fitness achievable in this experiment, as indicated by the path shown in Fig. 4.16(c), was six pellets being consumed in eight moves. This evaluates to 0.75 from Eq. 4.1. Hence, we see that the best hierarchical networks are those with $n = 4$ and $M = 4$, and are able to find the optimal solution for this trail.

Finally, these networks with $1,000$ neurons were evaluated for an even more complex trail, shown in Fig. 4.16(c). In this experiment, the maximum moves was set to ten. Since this trail can be completed in eight moves, and the maximum number of pellets is six, the maximum fitness will be $6/8 = 0.749$. The average of best fitness achieved by each type of hierarchical network was plotted as shown in Fig. 4.22.

In the second part of this experiment, the network size was increased to $5,000$ and the grid size was increased to 10×10, as shown in Fig. 4.23. The total number of food pellets in this trail was increased to 15, and there were three gaps in the

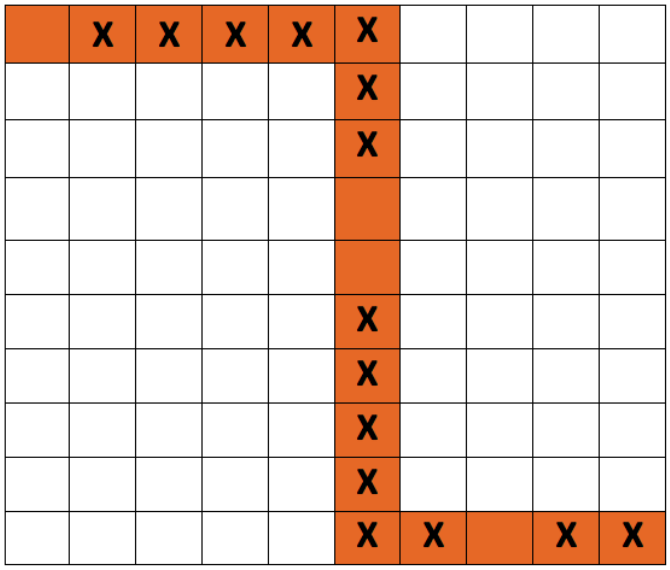


Figure 4.23: The 10×10 grid used to evaluate networks containing $5,000$ neurons. This path had both turns and gaps in the food trail, and hence, was more complex than the earlier trails. The network size was scaled to $5,000$ neurons to accommodate this increase in complexity.

trail. The maximum number of moves allowed in this task was set to 20. Similar to the last trail, the maximum fitness is < 1, and equal to $15/18 = 0.84$. The average fitness of the evaluated hierarchical reservoirs is shown in Fig. 4.24.

4.5.3 Discussion

From this experiment, it can be seen that hierarchical RBNs can be used to solve control tasks, where control signals for an autonomous agent are to be generated sequentially to achieve a desired goal. The following observations can be made from these results:

- From Fig. 4.20, it can be seen that all the hierarchical networks perform better than the monolithic networks. With an average fitness of 0.6, almost all of the monolithic networks were able to collect a maximum of six pellets

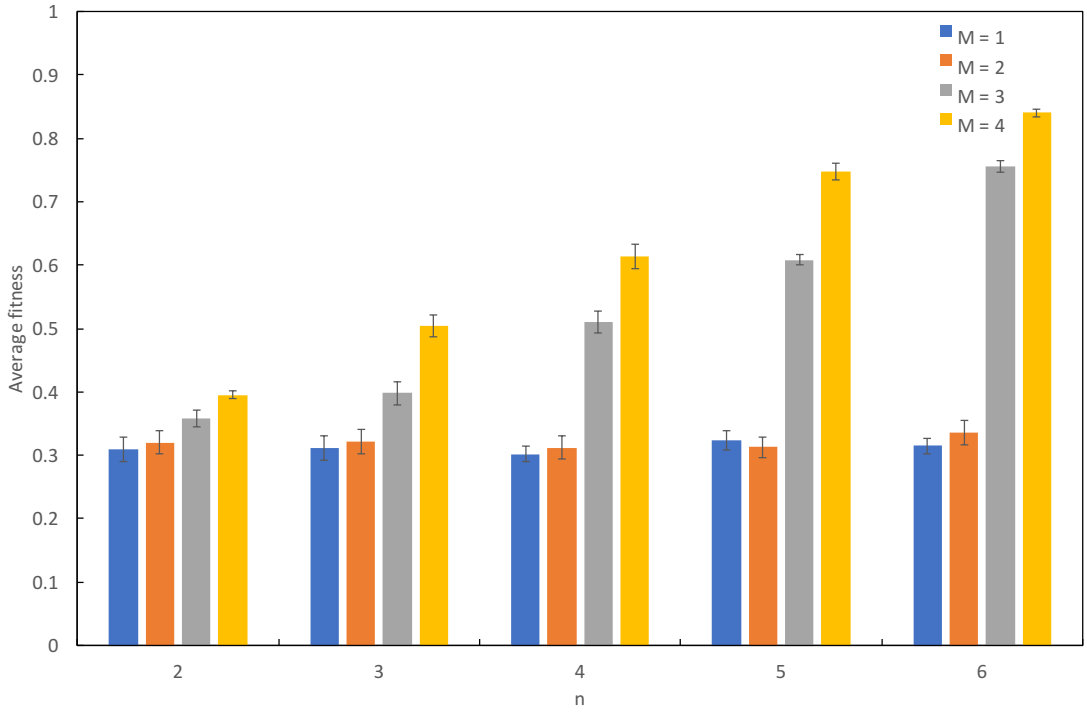


Figure 4.24: The average fitness of different hierarchical networks, evaluated for the more complex 10×10 trail task having two turns, and three gaps in the food trail. The maximum fitness achievable was 0.84, and only the networks with the highest number of modules were able to find an optimal solution to this task. At low number of modules, for example, in the networks with $M = 2$ hierarchical levels, the connections between the modules are very few and hence, the communication between the modules is not optimal, leading to a performance very close to that of monolithic networks.

out of the eight, in eight moves. These networks have a single hierarchical level and a single module, irrespective of the value of n as $M = 1$. Hence, it can be seen that all the monolithic networks perform similarly.

- While the performance of the hierarchical networks increases when M, n are increased, it also begins to decrease once the initial nodes become almost half of the total nodes in the final network. For example, in Fig. 4.20, the performance for the networks with $n = 6$, the performance decreased by more

than 38% when M is increased from 3 to 4. This is because the number of initial nodes in the network increases from 72 to 432, which is almost 43% of the total nodes in the final network. This decreases the number of between-modules connections in the network, thereby reducing the channels available for information exchange between the modules.

- From Fig. 4.21, it can be seen that the performance of the monolithic networks is also about 60% as the complexity of the trail is increased by increasing the number of turns to two. Almost all of these monolithic networks are able to solve 60% of the trail, i.e., they are able to complete the first turn in the trail successfully, and consume up to six food pellets in ten moves. On the other hand, the hierarchical networks demonstrate a similar trend observed in the case of the simpler 5×5 trail with one turn. However, the networks with fewer modules, such as those with $M = 2$, and $n = 2$,3 and 4 were not able to improve their performance because the number of modules are only increasing linearly with n.

- In Fig. 4.22, the average fitness is plotted for different hierarchical networks, which were evaluated using a trail that was even more complex than the one with two turns in it. The monolithic networks are only able to solve about 40% of the task, i.e, they are able to cross the first gap in the food trail and also consume up to four food pellets in the trail before exhausting their maximum moves. The networks with $n = 4$ and $M = 4$ are able to achieve the best fitness for this task, closely followed by networks with $n = 5$ and $M = 3$. The error bars get smaller as the fitness approaches the max value of 0.75, because almost all of these networks are able to find the optimal solution for this trail problem. So far, we have seen the results from networks with

$1,000$ neurons. In these networks, the amount of hierarchy that is beneficial is limited because of the small number of maximum network size. In the following point, we examine the results observed for different hierarchical networks, when the network size is increased to $5,000$.

- Fig. 4.24 shows results obtained in the final part of this experiment, for the 10×10 trail show that the performance of hierarchical networks can be improved by increasing the amount of hierarchy in them. This can be done either by increasing the network parameter n, or by increasing the number of hierarchical levels M. However, this will only scale well if the network is large enough so that the number of additional initial nodes that are added as a result of increasing the number of modules does not become a significant fraction of the final network size. The best performing networks having $n = 6$ and $M = 4$ were able to completely solve the trail by consuming all the 15 food pellets using 18 moves.

Overall, we can see that the performance of the networks depends on the balance between the number of in-module connections, and between-module connections. These two vary when the network parameters M and n are varied. Thus, the conclusion of this experiment is that hierarchical RBNs can be used to solve such control tasks. It was observed that the best performing hierarchical networks outperformed the monolithic networks by more than 40%. The number of initial nodes in these networks was less than 15% of the network size.

4.6 Experiment 6: Memory Recall Task

Rodriguez *et al.* investigated the influence of the modularity parameter on the memory performance of networks that are decomposed into a number of modules. They showed that networks having a modularity between 0.1 and 0.22 demonstrate optimal capability to store and recall information. They performed a task to evaluate how many different patterns of data can a neural network, with a given modularity, store in its memory, and perfectly recall them once given a cue.

4.6.1 Methodology

In this experiment, we tried to reproduce a part of the experiment performed by Rodriguez *et al.* to test the performance of the hierarchical networks that were tested in experiments 1-3.

The goal of this task was to teach the reservoir a set of sequences and evaluate how many of them it can recall correctly after a time "delta T". This requires the reservoir to have memory to store the input history for a period of time.

Figure 4.25 shows the setup used for performing this experiment. About 20% of the reservoir neurons were fed with each of the four input signal dimensions. Four different read-outs were used to reproduce the 4 dimensions of the input signal, which lasted for 5 time steps. About 10% of the outputs of the reservoir neurons were connected to each of the four read-out neurons. The read-out neurons were the same as those used in experiment 3.

At the end of a sequence, the states of the input neurons were left unchanged, and the reservoir was perturbed for another "delta T" time steps. Thereafter, the states of the reservoir neurons connected to the read-out neurons were obtained and saved for the training phase. The reservoir was then reset to its initial state

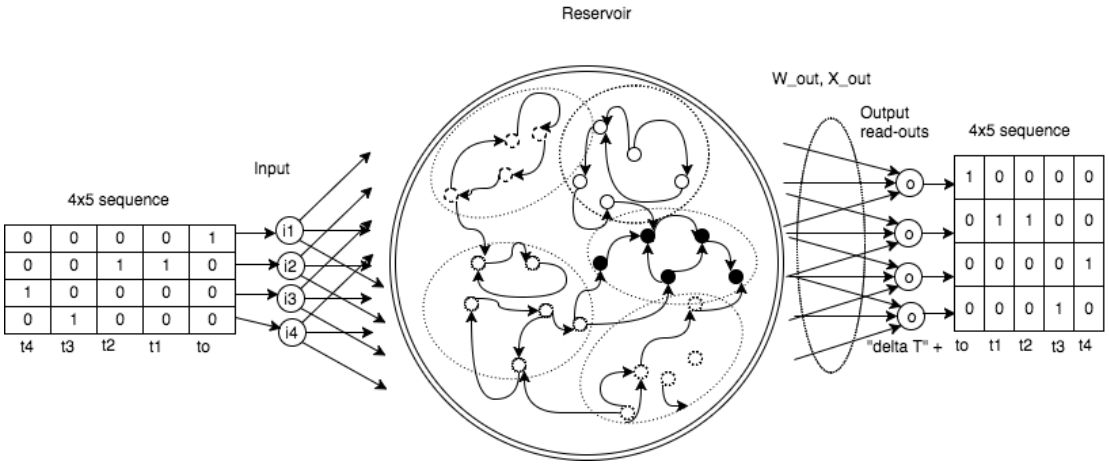


Figure 4.25: A high level diagram showing the perturbation of the reservoir with a $4x5$ input bit sequence. The input is a four dimensional bit signal lasting for five time steps. The network is perturbed with four bits of the input sequence at every time step. Four parallel inputs and four parallel read-outs were used in this system to reproduce the $4x5$ input bit sequence. The read-outs were collected after running the reservoir for 5 + "delta T" time steps where "delta T" was kept constant for an experiment. This determines the duration for which the reservoir has to store a given input pattern before it is expected to reproduce it.

and perturbed with a new sequence.

Linear gradient descent with a learning rate of 0.001 was used to train the read-out neurons. A training set of $1,000$ randomly generated sequences were used to train the read-out neurons. During each round of training, the sequences in the training set were input to the reservoir sequentially. The error was set to 0. After the reservoir was perturbed with a sequence and recalled it, if the recalled sequence did not match the input sequence, the error was increased by 1. After this, the reservoir was reset to its initial state and the next sequence was input to the reservoir. This process was repeated until all the $1,000$ sequences were completed. The error was accumulated and averaged across all the $1,000$ sequences. The weights of the read-out neurons were then updated using linear gradient descent, such that the error decreases to the minimum value through

multiple rounds of training. The training was aborted if either the average error was minimized to 0, or if the average error did not decrease by at least 1% over a period of $10,000$ training rounds. the maximum number of training epochs for training each network was set to $100,000$.

Linear gradient descent was suitable for training the RC systems in this task, because the expected outputs were readily available for each of the input sequences. Hence, genetic algorithms were not required to train these RC systems.

After training the reservoir, a test set consisting of 200 sequences was built using the original set of $1,000$ input sequences. The reservoir was then evaluated using these 200 sequences, and perturbed with each of these sequences in the same way as done during the training. The fraction of the test sequences that were perfectly recalled was used as the measure of performance of these networks. A score of 1.0 meant all the 200 input sequences were perfectly recalled.

In this experiment, the delay, delta T, is set to 80 time steps, and all the networks that were used for the experiment had 2000 nodes. Different combinations of the number of hierarchical levels (1 to 6) and the number of modules present in the reservoir were used (1 to 625) to see how hierarchy influenced the performance. It is useful to note that some of these combinations lead to networks having really poor connections between the modules. This happens because the growth model we have used to build these RBN reservoirs stipulates that the probability of adding a node to a module must always be greater than the probability of adding a connection between two modules for each hierarchical level [35]. For example, a network with 625 modules would have 1250 fully connected neurons initially with 2 neurons in each module. As the network is grown to a size of 2000, it can only add another 750 neurons that contribute towards the hierarchy.

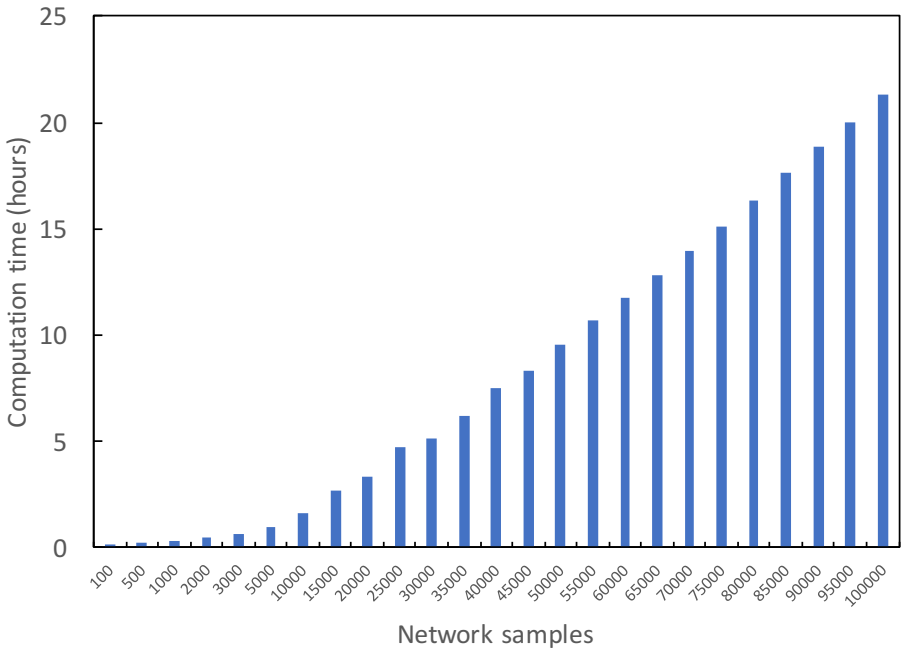


Figure 4.26: The computation rises almost linearly, because of the limited number of cores available at any point of time, thereby limiting the maximum number of networks that can be evaluated in parallel.

4.6.2 Results

This experiment had a long computation time, because even though the networks could be evaluated in parallel, the main bottleneck was the training of the network with the $1,000$ patterns using linear gradient descent. An initial experiment was performed with one type of hierarchical network for different number of network samples. The aim was to find a practical number of network samples to evaluate, based on the run time required. The training was stopped if the performance of a network did not improve by at least 1% over a period of $1,000$ epochs.

The variation of computation time with respect to the network samples is shown in Fig. 4.26. As we can see, the computation time grows almost exponentially. Therefore, to limit the computational time, we opted to evaluate only $60,000$

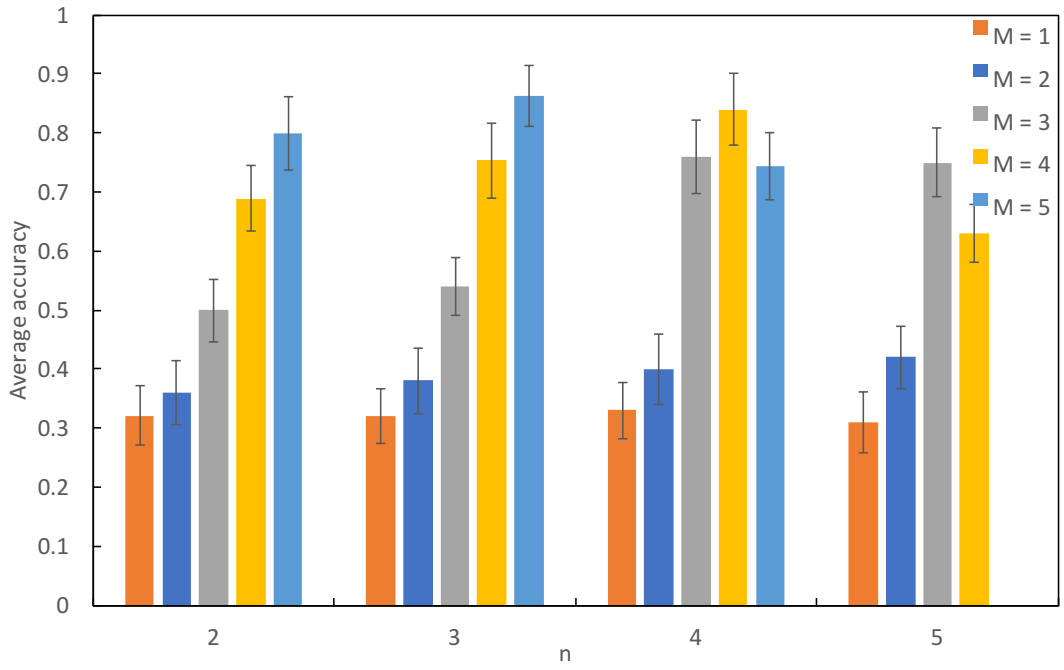


Figure 4.27: Average accuracy of the networks tested with 200 input sequences. There seems to be a threshold value for the between-module connections, as the accuracy increases sharply for $n = 4$ and M is increased from 2 to 3. The best performing networks achieved an accuracy of about 86%.

network samples for each combination of the network parameters.

The average accuracy of $60,000$ network samples of each type of hierarchical network are shown in Fig. 4.27. The Y-axis represents the fraction of 200 input sequences that were correctly recalled, after training the network with the $1,000$ test sequences.

4.6.3 Discussion

From Fig. 4.27, we can see the performance increases sharply from 0.3 to 0.85 when the number of hierarchical levels are increased from 1 to 5. However, there is a significant standard deviation in these values because of the fewer number of

network samples being evaluated. Even though the performance seems to increase with increasing number of hierarchical levels, the performance decreases by 10% when the initial number of nodes becomes 512 (almost 25% of the total number of nodes in the final network), as can be seen in the case of $n = 4$, when M increases from 4 to 5.

5

Conclusion

In this research, we proposed, built and evaluated different types of hierarchical RBN reservoir networks. In Chap. 3, we have used a growth algorithm to build hierarchical RBN reservoir networks in a generic way. It was found that these networks can easily be scaled to any desired size. In Chap. 4, these hierarchical networks were evaluated using three temporal tasks. Initially, the variation of network topology parameters, such as average connectivity (K_{av}), network modularity (Mu_{av}), with respect to different hierarchical configurations were investigated.

In the first experiment, we measured the average connectivity of hierarchical networks with $M = 2$ to 5 hierarchical levels and $n = 2$ to 5. In Networks containing $1,000$ neurons, it was found that the average connectivity of 75% of the networks was between 2 and 2.4. Also, the average connectivity began to decrease once the number of initial nodes in the network became more than 12% of the final network size. The lowest average connectivity of 1 was demonstrated by networks with $n = 5$ and $M = 5$. This was because the number of initial neurons was $2 \times 5^{5-1} = 1250$, which exceeded the final network size as a result of which no growth operations were performed in such networks.

In networks with $2,000$ nodes, the lowest average connectivity of 1.5 was demonstrated by networks having $n = 5$ and $M = 5$. The highest average connectivity of 2.3 was demonstrated by most networks having $n = 2, 3, 4$ and $M = 3, 4$. On the other hand, the negative impact of initial number of nodes on teh average connectivity was lesser in networks grown to a size of $5,000$ nodes. The lowest average

connectivity of 2 was achieved by networks with $n = 5$ and $M = 5$. All the other networks had an average connectivity between 2.18 and 2.35.

In the second experiment, the network modularity of the same hierarchical networks that were used in the first experiment, was measured. All the networks with 2 hierarchical levels demonstrated an average modularity of 0.08. The modularity was proportional to the number of hierarchical levels, because the probability of making between-modules connections increased when the number of hierarchical levels was increased. The highest modularity of 0.15 was achieved by networks with $n = 2, 3$ and $M = 5$. Networks with $M = 2$ demonstrated the lowest modularity.

In the third experiment, the average activity in different hierarchical reservoirs was measured by perturbing them with an alternating sequence of 01010101.. over a period of 100 time steps. Networks with $n = [2, 5]$ and $M = [2, 3]$ were evaluated for different percentages (0 to 70%) of reservoir neurons connected to the input sequence. From the results of this experiment, it was concluded that perturbing about 20% of the reservoir neurons with an input signal results in near-optimal activity in the network. A non-zero activity of about 20% was observed even when none of the reservoir neurons were perturbed with an input signal. These were due to the transients arising in dynamic systems, such as RBNs. The highest activity of 56% was observed in networks with $n = 5$ and $M = 3$.

After evaluating the variation of the above parameters, three different temporal tasks were used to evaluate different hierarchical RBN reservoirs. In the temporal pattern recognition task, the best hierarchical reservoir networks outperformed monolithic networks by up to 34%. The best performing networks achieved an accuracy of 0.96 with $n = 4$ and $M = 4$. In these networks, the performance decreased by about 20% when the initial number of nodes increased from 128

(12.8% of the final network size) to 512 (51.2% of the final network size. The latter type of networks had lower K_{av} and Mu_{av} compared to the first type of networks, as shown in the first and second experiments respectively.

In the simple 5×5 food-foraging trail task containing one turn in the food path, hierarchical networks outperformed monolithic networks by at least 15%. About one-third of the 16 different networks evaluated found an optimal solution to this problem. Networks with $M = 4$ and $n = 3, 4, 5$, and networks $M = 3$ and $n = 4, 5, 6$ were able to achieve the maximum fitness of close to 1.

In the a slightly more complex 5×5 trail containing two turns in the food path, hierarchical networks outperformed their monolithic counterparts by at least 10%. The networks with $n = 4, M = 4$, $n = 5, M = 3$ and $n = 6$, $M = 3$ were able to find an optimal solution for this problem and achieved a fitness of close to 1 by consuming all the eight food pellets using eight moves only.

In the most difficult 10×10 trail containing two turns and three gaps, only networks with a high level of hierarchy were able to find an optimal solution. The monolithic networks achieved a fitness of 30%. The best performing hierarchical networks with $n = 6$ and $M = 4$ achieved a fitness of 0.834.

Finally, hierarchical networks outperformed monolithic networks by up to 54% in the memory recall task. These networks demonstrated an optimal region of hierarchy where the best performance was achieved. Networks with $n = 3, M = 5$ achieved an average accuracy of 86%, and the next highest performance of 82% was achieved by networks with $n = 4$ and $M = 5$.

5.1 Significance and Impact

From the results of our experiments, it was observed that composing a reservoir network with a number of smaller communities of neurons, connected to each other using between-modules connections, improved their performance compared to equivalent monolithic networks. Three temporal problems were evaluated using networks of different sizes and hierarchies, as described in Experiments 4, 5 and 6 of Chap. 4.

We have shown that the performance of hierarchical networks increases as the amount of hierarchy in these networks is increased. We also observed that the best performing hierarchical networks outperformed equivalent monolithic networks by also 60%. Burger et al. showed that hierarchical RC networks with a ring topology were able to outperform monolithic networks by up to 20% in waveform generation tasks.

We have also shown through our experiments that an optimal range of hierarchy exists, where the performance of hierarchical networks is optimal. In most of the experiments, the best performing networks were those with $n = 3, 4$ and $M = 3, 4$. In networks with other values of n, M, the initial number of nodes affected the network modularity negatively depending on the network size. Rodriguez *et al.* showed that networks with a modularity between 0.1 and 0.2 outperformed other networks by up to 60% when tasked to recall 200 sequences from memory after a gap of 80 time steps. In a similar experiment performed in this research, the best performing hierarchical network outperformed monolithic networks by up to 54%.

Finally, by showing that RBNs can be used as hierarchical reservoirs to solve complex temporal tasks, we can say that Boolean logic gates can be used as an alternative to other neurons, such as memristors and memcapacitors, and these

gates are simpler to fabricate. These networks can also be readily mapped to a mobile platform, such as a field programmable gate array, to test them in real hardware.

5.2 Future work

This study was restricted networks having an average connectivity of close to two. In the future, this research could be expanded by evaluating networks with other values of average connectivity. The average connectivity of an RBN determines its dynamics, but most of literature investigating this uses RBNs without any hierarchical structure. It would be interesting to observe the dynamics of such networks, when a hierarchical structure is introduced into them.

In the networks that were evaluated, the out-degree of each neuron was not limited. However, this might be a concern when trying to fabricate such networks. The fan-out of any digital logic gate can successfully transmit information to a limited number of other gates, because of the limited amount of current that is generated in the source gate for any signal transition. Hence, before proceeding to fabricating these networks, they need to be studied in scenarios where the out-degree of the neurons is limited to a constant value, similar to the way the in-degree was restricted to 10.

The motivation for the food-foraging tasks used in this research was the more complex 32×32 toroidal grid called the Santa Fe trail [18]. In the future, it would be interesting to see the variation in performance of these hierarchical networks when tasked to solve this much more complex trail.

References

[1] A. Barabasi and R. Albert. Emergence of Scaling in Random Networks. *Science*, 286(5439):509–512, October 1999.

[2] G. Bebis and M.l Georgiopoulos. Feed-forward neural networks. *IEEE Potentials*, 13(4):27–31, 1994.

[3] J. Burger, A. Goudarzi, D. Stefanovic, and C. Teuscher. Hierarchical composition of memristive networks for real-time computing. In *Proceedings of the 2015 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH 2015)*, pages 33–38, July 2015.

[4] K. De Jong and W. Spears. An analysis of the interacting roles of population size and crossover in genetic algorithms. In *International Conference on Parallel Problem Solving from Nature*, pages 38–47. Springer, 1990.

[5] B. Derrida and Y. Pomeau. Random Networks of Automata: A Simple Annealed Approximation. *EPL (Europhysics Letters)*, 1(2):45–49, 1986.

[6] F. Fortin, F. De Rainville, M. Gardner, M. Parizeau, and C. Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, July 2012.

[7] R. Glasius, A. Komoda, and S. Gielen. Neural network dynamics for path planning and obstacle avoidance. *Neural Networks*, 8(1):125–133, 1995.

[8] A. Graves, A. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (ICASSP), 2013 IEEE international conference on*, pages 6645–6649. IEEE, 2013.

[9] R. Hecht-Nielsen. On the algebraic structure of feedforward network weight spaces. *Advanced Neural Computers*, pages 129–135, 1990.

[10] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. *Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies*. A field guide to dynamical recurrent neural networks. IEEE Press, 2001.

[11] J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

[12] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[13] 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*, 148(34):13, 2001.

[14] H. Jaeger and H. Haas. Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, 304(5667):78–80, 2004.

[15] H. Jaeger, M. Lukoševičius, D. Popovici, and U. Siewert. Optimization and applications of echo state networks with leaky-integrator neurons. *Neural Networks*, 20(3):335–352, 2007.

[16] S. Kauffman. Homeostasis and differentiation in random genetic control networks. *Nature*, 224(5215):177–178, 1969.

[17] S. Kauffman, C. Peterson, B. Samuelsson, and C. Troein. Random boolean network models and the yeast transcriptional network. *Proceedings of the National Academy of Sciences*, 100(25):14796–14799, 2003.

[18] J. Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[19] R. Lippmann. An introduction to computing with neural nets. *IEEE Assp magazine*, 4(2):4–22, 1987.

[20] M. Lukoševičius and H. Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149, 2009.

[21] W. Maass, T. Natschläger, and H. Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, 14(11):2531–2560, 2002.

[22] T. Mihaljev and B. Drossel. Scaling in a general class of critical random boolean networks. *Physical Review E*, 74(4):046101, 2006.

[23] P. Razvan, M. Tomas, and B. Yoshua. On the difficulty of training recurrent neural networks. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28–3 of *Proceedings of Machine Learning Research*, pages 1310–1318, Atlanta, Georgia, USA, 17–19 June 2013. PMLR.

[24] N. Rodriguez, E. Izquierdo, and Y. Ahn. Optimal modularity and memory capacity of neural networks. *arXiv:1706.06511 [physics]*, June 2017. arXiv: 1706.06511.

[25] F. Rosenblatt. Principles of Neurodynamics. Perceptrons and the theory of brain mechanisms. Technical Report VG-1196-G-8, Cornell Aeronautical Lab Inc, Buffalo, NY, March 1961.

[26] H. Rumelhart and R. Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[27] 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*, pages 471–482, 2007.

[28] D. Snyder, A. Goudarzi, and C. Teuscher. Computational capabilities of random automata networks for reservoir computing. *Physical Review E*, 87(4):042808, April 2013.

[29] R. V Solé and B. Luque. Phase transitions and antichaos in generalized kauffman networks. *Physics Letters A*, 196(5-6):331–334, 1995.

[30] F. Triefenbach, A. Jalalvand, B. Schrauwen, and J. Martens. Phoneme recognition with large hierarchical reservoirs. In *Advances in Neural Information Processing Systems*, pages 2307–2315, 2010.

[31] D. Verstraeten, B. Schrauwen, and D. Stroobandt. Reservoir-based techniques for speech recognition. In *Neural Networks, 2006. IJCNN'06. International Joint Conference on*, pages 1050–1053. IEEE, 2006.

[32] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K. Lang. Phoneme recognition using time-delay neural networks. *IEEE transactions on acoustics, speech, and signal processing*, 37(3):328–339, 1989.

[33] P. Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. *Doctoral Dissertation, Applied Mathematics, Harvard University, MA*, 1974.

[34] P. Werbos. Backpropagation through time: What it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[35] Q. Xuan, Y. Li, and T. Wu. Growth model for complex networks with hierarchical and modular structures. *Physical Review. E, Statistical, Nonlinear, and Soft Matter Physics*, 73(3 Pt 2):036105, March 2006.

[36] G. Zhang, B. Patuwo, and M. Hu. Forecasting with artificial neural networks:: The state of the art. *International journal of forecasting*, 14(1):35–62, 1998.