Winter 3-27-2013

Reward-driven Training of Random Boolean Network Reservoirs for Model-Free Environments

Padmashri Gargesa
Portland State University

Reward-driven Training of Random Boolean Network Reservoirs for Model-Free Environments

by

Padmashri Gargesa

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
Richard P. Tymerski
Marek A. Perkowski

Portland State University
2013

Abstract

Reservoir Computing (RC) is an emerging machine learning paradigm where a fixed kernel, built from a randomly connected "reservoir" with sufficiently rich dynamics, is capable of expanding the problem space in a non-linear fashion to a higher dimensional feature space. These features can then be interpreted by a linear readout layer that is trained by a gradient descent method. In comparison to traditional neural networks, only the output layer needs to be trained, which leads to a significant computational advantage. In addition, the short term memory of the reservoir dynamics has the ability to transform a complex temporal input state space to a simple non-temporal representation.

Adaptive real-time systems are multi-stage decision problems that can be used to train an agent to achieve a preset goal by performing an optimal action at each timestep. In such problems, the agent learns through continuous interactions with its environment. Conventional techniques to solving such problems become computationally expensive or may not converge if the state-space being considered is large, partially observable, or if short term memory is required in optimal decision making.

The objective of this thesis is to use reservoir computers to solve such goal-driven tasks, where no error signal can be readily calculated to apply gradient descent methodologies. To address this challenge, we propose a novel reinforcement learning approach in combination with reservoir computers built from simple Boolean components. Such reservoirs are of interest because they have the potential to be fabricated by self-assembly techniques. We evaluate the performance of our approach in both Markovian and non-Markovian environments. We compare the performance of an agent trained through traditional Q-Learning. We find that

the reservoir-based agent performs successfully in these problem contexts and even performs marginally better than Q-Learning agents in certain cases.

Our proposed approach allows to retain the advantage of traditional parameterized dynamic systems in successfully modeling embedded state-space representations while eliminating the complexity involved in training traditional neural networks. To the best of our knowledge, our method of training a reservoir readout layer through an on-policy boot-strapping approach is unique in the field of random Boolean network reservoirs.

Acknowledgements

This work was partly sponsored by the National Science Foundation under grant number 1028378.

Contents

List of Tables

List of Figures

List of Algorithms

1

Overview

1.1 Introduction

The objective of our research is to study the feasibility of reservoirs to successfully perform as autonomous real-time agents through interactive training. The challenges involved in interactively training such agents include the complexity of training with no known target, and also in dealing with input state representations that may be partially observable. The area of research that deals with training autonomous systems interactively called *Reinforcement Learning* (RL) techniques assume the input state representation from the underlying environment to be *Markovian* (or complete). These techniques are predictive systems which perform actions at each timestep based on its associated future reward estimates. We setup a reservoir to act as a predictive system which can accurately estimate future rewards associated with each action at a given input state so as to achieve the set goal optimally. In doing so, we experiment with tasks which have *Markovian* and *non-Markovian* input state representations. We study the computational ability of reservoirs in performing these tasks optimally by comparing the results with that of standard benchmark approaches. We also observe the robustness of the system in handling noisy representations of the input state-space.

1.2 Goal and Motivation

Most of the current day computing paradigms are based on Von Neumann architectures, which involve sequential execution of instruction sets to achieve a particular

outcome. The need for faster and cheaper hardware has resulted in a technological trend favoring physical down-scaling of current semiconductor-based architectures. However, down-scaling of physical devices has an obvious road block down the line. Further scaling of current nano-scale architectures has problems involving handling complex high density circuitry and the ensuing design, layout and fabrication issues [10].

The present multi-core architectures, which are a step towards parallel computing still rely on sequential logic. The partial concurrency in task execution, which is achieved with these architectures comes at the price of high density design requirements and the need for communication between the architectural cores. This has increased the need for longer interconnects, thus increasing the wire delays. The increased power consumption and dissipation involved with such architectures has also resulted in higher failure rates in these designs [11].

Teuscher et al. in [11, 12] talk about the need to address these drawbacks and propose self-assembled architectures as an alternative. These architectures have a bottom-up approach as opposed to the top-down approach of current semiconductor architectures and are composed of randomly interconnected devices with irregular structures.

Reservoir Computing is a broader subject area which deals with self-assembled substrates called *reservoirs* [13]. In general, a *reservoir* is any self-assembled network of processing nodes which can provide a *rich* set of dynamics that can be tapped into to solve computational tasks.

1.3 Challenges

There are challenges in programming these self-assembled systems with random interconnects to achieve a specific task. There has been extensive research in this area where such self-assembled architectures are studied for designs that yield high computational ability [11, 12, 14, 15]. Despite their similarities with *Artificial Neural Networks*, the design challenges involved in programming these devices for a particular task are different.

Most of the related research done so far in programming reservoirs have been centred around learning tasks with known input-output mappings. However, there exists a whole class of tasks where the system learns in an interactive manner. In such problem contexts, as shown in Figure 1.1, unlike in a supervised learning context, there is no teacher which provides the target output to be mapped to an input. So the learning process involves learning through continuous interactions with the environment.

As shown in Figure 1.2, we consider a popular approach towards solving such problems, called *Reinforcement Learning*. Reinforcement learning techniques are interactive, in that, a critic system assesses the quality of the action taken in response to a particular input stimulus, and the learning system adjusts based on this reward signal in order to maximize its chances of gaining future rewards and hence achieving a pre-defined goal.

However, reinforcement learning algorithms assume the underlying input state to be *Markovian* in nature. A system state space is said to have the *Markovian* property if any future state is solely a function of current state and action. In reality, a complete knowledge of the system/plant model may not be available in all problem contexts. When the underlying system model is unknown, the system

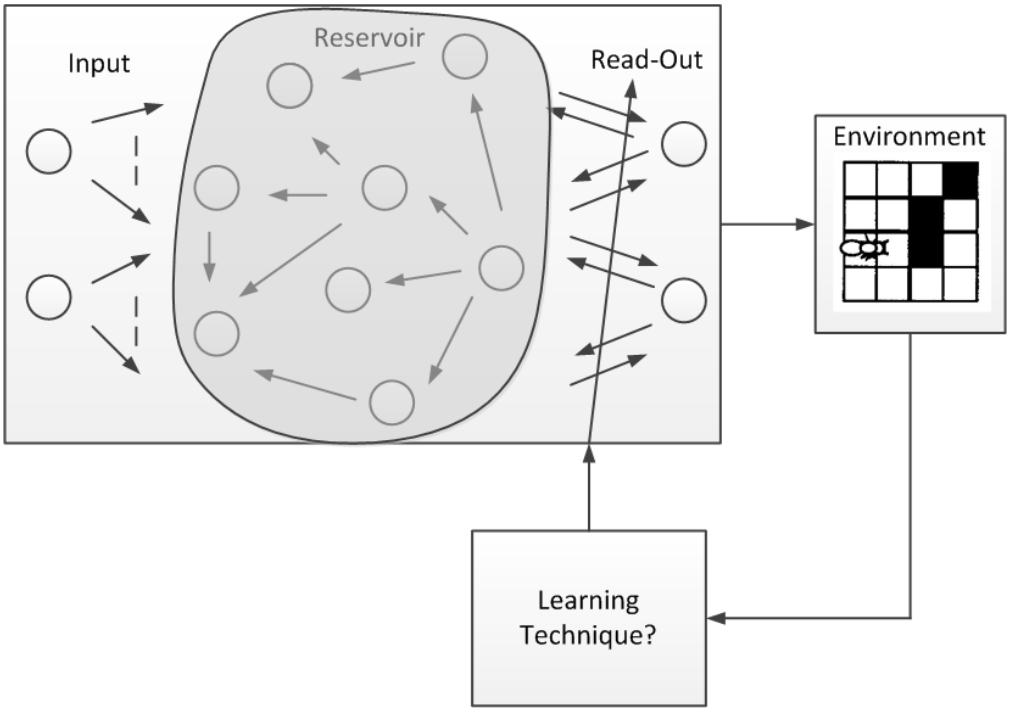


Figure 1.1: Block diagram depicting a reservoir set in a goal-driven context. A 2-D grid-world is depicted as the environment with which the reservoir interacts to learn. The reservoir should learn to function as a decision making agent.

state available may be complete or partially observable. Also when the current state is partially observable, it is not completely representative of the future state and is said to be *non-Markovian*.

Takens' theorem provides a solution to this issue stating a *non-Markovian* state space can be transformed into a *Markovian* state representation by temporally embedding sequentially incomplete state spaces into a higher dimensional feature space [16].

However, reconstructing the state space to a *Markovian* representation involves temporally sequencing the current states. The temporal length that needs to be considered for appropriately modeling the *non-Markovian* state space to a *Markovian* one depends on the problem context and in most problem contexts such a

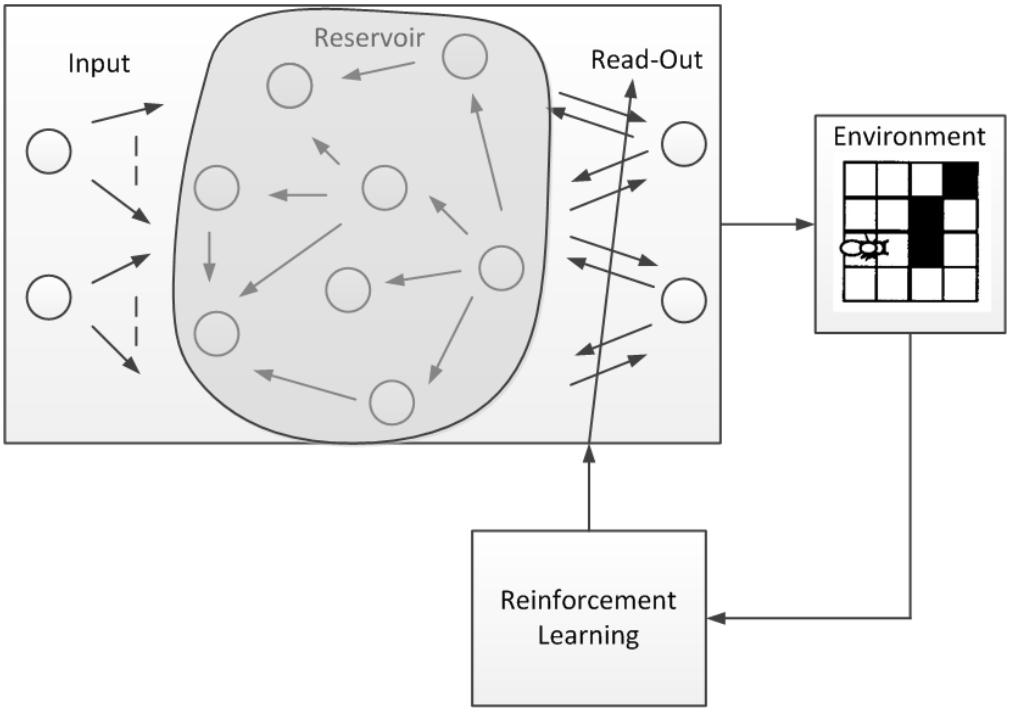


Figure 1.2: Block diagram depicting a reservoir set in a goal-driven context to function as a decision making agent. The read-out layer of the reservoir is trained based on reinforcement learning, through rewards received with its interaction with the environment. In this case, the environment is depicted as a 2-D grid-world.

knowledge is not readily available. Achieving such a representation gets more memory intensive as the length of the temporal sequence of the original states increases.

In this research work, we try to address the challenges involved in designing and programming random self-assembled architectures to function as adaptive real-time systems, by answering the following high level questions:

1. How can random unstructured architectures be programmed to function as autonomous agents in *goal driven* contexts where there is no target behavior to map to?

2. Can these self-assembled architectures perform efficiently when programmed

to adapt in environments with *Markovian* and *non-Markovian* state representations?

3. How robust are these architectures when programmed through reinforcement learning techniques to noisy representation of the underlying environment?

4. How do they perform in comparison to other benchmark reinforcement learning techniques?

1.4 Our Contributions

Our contributions in this work are as follows:

1. We proposed to setup a simulated, self-assembled, *Random Boolean Network* (RBN) model in computational tasks which involve adaptive learning through continuous interactions with the environment.

2. We developed a software framework in C++ which implements a feed-forward linear read-out layer which interfaces with an existing working RBN simulation model, as explained in Section 4.1.

3. We developed a software framework in C++ which interfaces with the linear read-out layer to adapt the weights based on *reinforcements*, as highlighted in Section 4.1.

4. We developed a software framework in C++ to implement simulations of specific *Markovian* and *non-Markovian* environment models, we use in this research work, covered under Section 4.1.

5. We implemented an interface in C++ between an existing *Differential Evolution* implementation and the RBN framework for evolutionary computation, covered under Section 4.2.

6. We developed a software framework in C++ to implement *Q-Learning* which is a benchmark reinforcement learning technique based on canonical table lookups, as explained in Section 4.3.

7. We optimize the initial RBN parameters setup in a reinforcement learning context to achieve convergence during the training of weights based on *reinforcements*.

8. We performed experiments to test the functionality of the feed-forward linear read-out layer framework when setup in a supervised learning context as explained under Section 5.1. We setup our RBN-based framework to solve an odd-temporal parity task when the input-output data mapping is known.

9. Once the performance of the read-out layer in solving supervised learning problems was validated, we performed experiments to train the RBN-based agent adaptively in simulated environments with the *Markovian* property as explained under Section 5.2. We setup the agent in the n-bandit context and successfully train it to learn the unknown reward probabilities of n independent processes.

10. We performed experiments to train the RBN-based agent adaptively in simplified simulated environments with the *non-Markovian* property as explained under Section 5.3. We successfully train our RBN-based setup to function as a food foraging ant (or agent) in a 32×32 2-D grid world task.

11. We compared the performance of the RBN-based agent with that of the *Q-Learning* agent on the 2-D grid world task. We found that our system converges 30% faster to a more optimal solution, for the John Muir trail environment we consider [9], while does not perform well on a more complex Santa Fe trail [8] considered.

12. We conducted robustness studies on our system, by injecting noise into the underlying environment and found that the system is tolerant to upto 10% noise injected into its environment and shows deteriorated performance to noise injections beyond that.

2

Background

2.1 Artificial Neural Networks

Neural networks are systems comprising of simple processing units, which receive input excitations and process it and pass it on to its output [17]. These processing units are connected to each other through weighted connections. Through a training process, which involves altering the weights connecting these processing units, the overall behavior of the network can be changed. Such a training is usually carried out to map the input to a desired output. A network is usually considered non-temporal or memoryless, since by maintaining constant weights, the output is solely dependent on the current input [1].

We go through a simple Adaline neural network model [17], to explain the processing and the training procedures of an ANN:

An input vector $X = x_i$ of dimension n, is mapped to a one-dimensioned output y through variable weighted connections. The weight vector $W = w_i$ has the same dimension as the input vector. The output is essentially the weighted sum of the inputs passed through an activation function. In Figure 2.1, the activation function depicted is a sigmoid function.

$$y(X) = sigmoid(\sum_{i=0}^{n-1} w_i x_i) \tag{2.1}$$

$$sigmoid(s) = tanh(s) \tag{2.2}$$

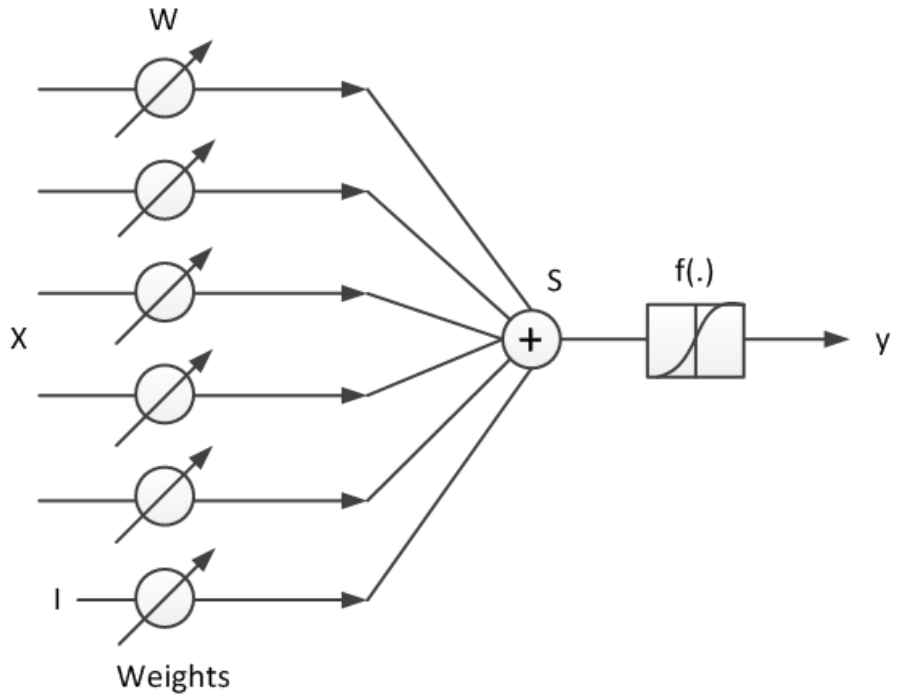


Figure 2.1: Adaline with a sigmoid activation function; Source: [1]

Here $y(X)$ is the actual output of the Adaline network in Figure 2.1. Usually these networks are trained to map the input to a desired output, say $d(X)$. The performance of an NN is ususally measured by the mean-square error:

$$\begin{aligned} J &= E(error^2) \\ &= E(d(X) - y(X))^2 \\ &= E(d(x) - sigmoid(\sum_{i=0}^{n-1} w_i x_i))^2 \end{aligned} \tag{2.3}$$

Equations 2.4 and 2.5 adjust the value of the weights through gradient descent [17, 18].

$$w_{i,new} = w_{i,old} + 2\mu\delta x_i \tag{2.4}$$

$$\delta = (d(X) - y(X))f^{'}(\sum_{i=0}^{n-1} w_i x_i) \tag{2.5}$$

The learning rate μ is chosen by the designer, in a way to balance out stability of the weight updates with the speed of convergence. Also, the training happens over a range of $(X, d(X))$ mappings. Through weight adjustments, the NN learns the internal function which maps the input vector to the target output.

Typical ANNs are multi-layered Adaline models and are termed multi-layer perceptrons [19]. These networks are also called feedforward neural networks. A feedforward network with one hidden layer is found to map or implement most of the non-linear functions. The understanding is that each layer in the feedforward network implements a linear mapping of its input to its output. The final layer combines all these nested linear mappings to a non-linear mapping of the input space to the output space [1].

A multi-layered perceptron is trained with an approach called as *back propagation* [18]. For a processing node in the output layer, the error is calculated as:

$$\delta_m = (d(X) - y(X))f^{'}(s_m(X)), \tag{2.6}$$

where

$$s_m(X) = \sum_{i=0}^{n-1} w_i x_i \tag{2.7}$$

Also, the error calculated for one of the nodes in a hidden layer is given by:

$$\delta_m = f^{'}(s_m(X)) \sum_j \delta_j w_{jm}, \tag{2.8}$$

where w_{jm} is the weight that connects the node m's output to the node j's

input [1].

For our purposes, we are interested in problem contexts which require memory. We explored previous research done in setting up feed-forward neural networks to solve tasks that require memory. With feed-forward neural networks, it has been shown how temporal tasks, where the output is dependant on a temporal structure in the input sequence, can be implemented by Jordan and Elman networks [20]. Elman in his work used feed-forward neural networks in natural language processing [20]. Here the internal input representation is fed back to a context unit layer. The context units, which received this feedback from all hidden layers, provide the necessary past input signal history to map the input sequence to the output space. Unlike explicit mappings of input sequence in time, such implicit techniques do not need to know how far back in history one needs to look to achieve an optimal input to output mapping. This attempt to try and map dynamical systems using feed-forward neural network is not computationally efficient, since feed-forward neural networks were designed and efficiently map only static systems.

2.2 Recurrent Neural Networks

The *Recurrent Neural Networks* (RNNs) [2], were designed to fill the wide gap left by this in-ability of feed-forward neural networks to address dynamic system implementations without incurring any additional computational overhead.

While the activations through a multi-layer perceptron is fed-forward, the connections in an RNN follow a cyclic pathway, as shown in Figure 2.2. The RNN models the recurrent connections as present in biological models. However, there are no popular algorithms to train the RNNs. RNNs have been applied to non-linear system-identification [21], signal processing, stochastic sequence modeling

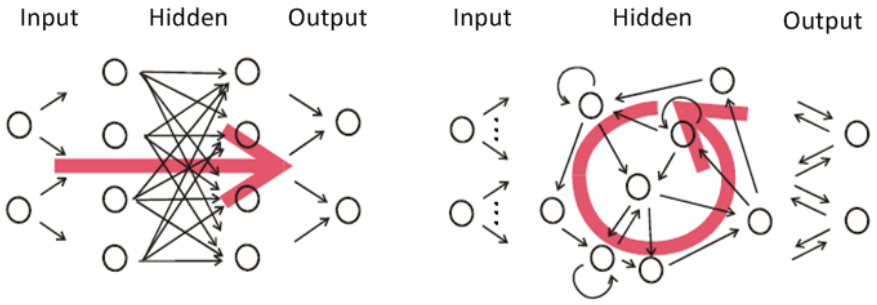


Figure 2.2: Feed forward vs Recurrent neural networks; Source: [2].

and prediction, data compression etc., [2].

Since in this research we are interested in architectures which learn temporal structures, we look into the details of modeling the mappings and expansions encountered in temporal and non-temporal tasks.

In case of linear modeling, the input and output vectors are mapped through a weight matrix with this simple equation:

$$\mathbf{y}(n) = \mathbf{W}\mathbf{u}(n), \tag{2.9}$$

where $W \in \Re^{N_y, N_u}$

However, accurate modeling requires the input vector $\mathbf{u}(n)$ to be transformed to a higher-dimensional feature space $x(n) \in \Re^{N_x}$

$$\mathbf{y}(n) = f_{out}(\mathbf{W}_{out}\mathbf{x}(n)) = f_{out}(\mathbf{W}_{out}x(\mathbf{u}(n))), \tag{2.10}$$

where $W_{out} \in \Re^{N_y, N_x}$ are the trained output weights.

Such methods, are referred to as *expansion methods* and alternately also referred to as kernel methods and the function $x(\mathbf{u}(n))$ that achieves this expansion is termed an *expansion function*. Other popular kernel methods include Support

Vector Machines, Radial Basis functions. The problem with most expansion methods is that the kernel function or the expansion function involved needs to be chosen empirically rather than through mathematical theory [3].

The above relationships dealt with are still non-temporal in nature. In case of temporal tasks, for example: non-linear signal processing, the expansion function is of the form:

$$\mathbf{x}(n) = f(\mathbf{x}(n-1), \mathbf{u}(n)), \tag{2.11}$$

$$\mathbf{y}(n) = y_{target}(....., \mathbf{u}(n-1), \mathbf{u}(n)) = \mathbf{u}(n+1) \tag{2.12}$$

where $\mathbf{y}(n)$ models the temporal system.

Equations 2.11 and 2.12 show that in such problem contexts the expansion function achieves spatial embedding of temporal information from the lower dimensional input space to a high-dimensional feature space. This achieves what the Takens' theorem states, that is to capture dynamic attractors of the system modeled by a higher dimensional feature space from a temporal series of lower dimension feature representation [16].

The state vector that results from kernel transformation in an RNN can be represented by:

$$\mathbf{x}(n) = f(\mathbf{W}_{in}\mathbf{u}(n) + \mathbf{W}\mathbf{x}(n-1) + \mathbf{W}_{ofb}\mathbf{y}(n-1)),$$

where $n = 1, ..., T$.

f(.) is the activation function, $\mathbf{W}_{in} \in \Re^{N_x, N_u}$ is the input weight matrix, $\mathbf{W} \in \Re^{N_x, N_x}$ is the weight matrix of internal network connections, $\mathbf{W}_{ofb} \in \Re^{N_x, N_y}$ is the optional output feedback weight matrix [16].

Despite the advantages that an RNN brings in handling time-dependant tasks through its recurrent pathways, it is still unpopular in comparison to FNNs due to the lack of efficient techniques to train an RNN. This disadvantage lead to the design of recurrent dynamic systems which retain the computational advantages of the RNN while trying to address its training complexities.

2.3 Reservoir Computing

A new generation of RNNs with significant improvements over the original RNN learning rule were developed in 2001 [13, 22]. With the RNNs, it was realized that the recurrent layer alternatively called *Reservoir* trained at a slower rate than the output layer and so the two layers could indeed be trained separately. In this new approach an RNN is randomly created, comprising of a pool of interconnected processing units. As shown in Figure 2.3, the RNN excited passively by an input signal transforms the input space, through the dynamics between its recurrent interconnects, into a higher dimensional state representation. The output signal is then read out through a linear readout layer. These weighted connections to the readout layer can be trained in its simplest form through linear regression. In this context the term *Reservoir Computing* was introduced.

The reservoir, like RNNs, expands the input excitation $u(n)$ in a non-linear temporal fashion, as represented by Equation 2.10, into the reservoir state $x(n)$, which is characterized by its higher dimensional dynamics. The actual output signal, $y(n)$ is a linear recombination of the weights and this reservoir state. However, unlike other methods, the reservoir methods see a clear distinction between the dynamic recurrent reservoir and the linear non-recurrent output layer. It was because of the understanding that the functions $x(.)$ and $y(.)$ are functionally different from

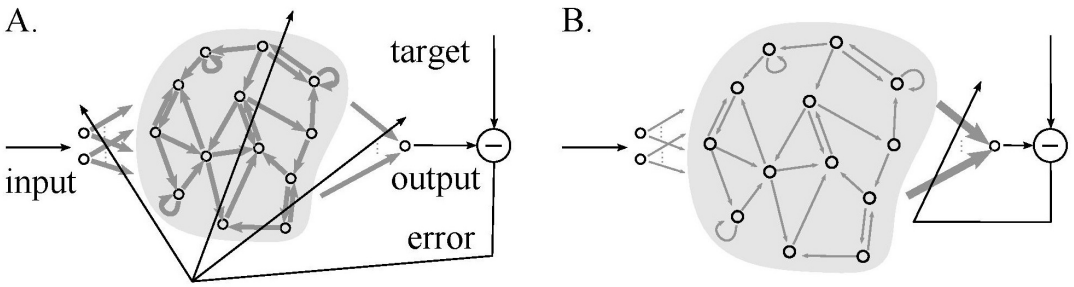


Figure 2.3: A. Traditional gradient descent based RNN training methods adapt all connection weights (bold arrows), including input-to-RNN, RNN internal, and RNN-to-output weights. B. In Reservoir Computing, only the RNN-to-output weights are adapted; Source: [3]

each other and try to achieve different things. While $x(.)$ tries to achieve a temporal recombination of the input excitations $u(n), u(n-1), u(n-2)...$, the function $y(.)$ does a simple linear transformation of this reservoir state to achieve the preset target y_{target} [3].

While designing the output layer, it is not necessary to worry about how rich $x(.)$ is, one only needs to worry about reducing the error, $y(.) - y_{target}(.)$. The reservoir dynamics on the other hand need not have to be designed with y_{target} in mind. The reservoir and the readout can be designed separately, even driven by different goals and then recombined to achieve the task at hand [3]. A good reservoir design is said to possess, the *separation property*, which measures the difference in distance between the reservoir states $\mathbf{x}$ for two different input sequences $\mathbf{u}$, and the *approximation property*, which measures the capability of output layer to approximate y_{target} well from the reservoir state $\mathbf{x}$ [22].

The two main RNN brands include:

- *Echo State Networks* (ESNs): Echo state networks are reservoirs made up of random interconnected neurons with sigmoid-style activation functions.

The *echo state condition* is said to have been achieved if the influence of past states and inputs on the future states dies out gradually over time. The stability of the reservoir depends on its possessing this quality, called the *echo state property* [13].

- *Liquid State Machines* (LSMs): Liquid State Machines are architecturally closer to biological systems. They incorporate a reservoir with nodes implemented as spike-and-fire neurons. Though these are difficult to setup and tune on digital computers due to the asynchronous nature of updates involved, their design is more open to further changes and adaptations from biologically inspired processes. The spike activations make it possible to encode information about the actual time of the firing of the neuron in the dynamics of the reservoir, which is not possible in standard non-linear activation functions as implemented with ESNs. In case of LSMs, the input stream is usually a spike stream and the readout layer is implemented as a linear readout or a Multi-layer Perceptron [22].

The methods listed below can be followed to design good reservoirs:

- *Classical approach*: These approaches involve designing a reservoir irrespective of the input and desired output. The network size, sparseness in connectivity and the strength of the connection weights are randomly initialized. The ability of the network to attain the echo state condition is assessed based on the criteria that the spectral radius of the connection weight matrix is less than 1. Also, the reservoir can be modularized topologically into many smaller reservoirs, each learning a specific feature-map between the input and output space to map complex non-linear outputs [13].

- *Unsupervised methods*: These approaches involve designing a reservoir for a particular input while the desired output is not known. These approaches can be further divided into local, semi local and global training. Local training aims to adapt the intrinsic excitability of the neurons [23, 24]. Adaptation of the intrinsic excitability, alternately termed the *intrinsic plasticity* rule, as adapted to neurons with sigmoid style functions involves proportional modification of steepness and offset of the activation function, thus modifying its firing rate [25]. Semi local training aims to minimize the input connections to the individual neurons in the reservoir by advocating an *Independent Component Analysis* approach while at the same time trying to maximize output information [26–28]. Global techniques aim at improving the overall reservoir connection weight matrix to achieve echo state properties through adjusting the spectral radius. There are predictive models used to train the reservoir, where the state of the reservoir in the next time instance is predicted and used to reduce the error between actual and predicted states by training achieved through gradient descent [29].

- *Supervised methods*: These approaches involve designing a reservoir for a particular input and desired output mapping. Here emphasis is laid on optimizing the reservoir parameters, evolutionary techniques are adopted to sweep the network parameter space for a favorable region [30].

Readouts can be trained similar to any other neural network paradigm; some of the techniques discussed are listed as below:

- Linear regression, where readout weights can be analytically computed given a target output matrix using Moore-Penrose generalized inverse [31].

- Multilayer readouts are adopted incase the task at hand demands a highly non-linear output map. A back-propagation technique is used to update the hidden weight layers [22].

- In cases where the input and the output have a phase difference, cross correlation between the two is maximized by learning the delay and adjusting it before the actual readout [32].

There has been lot of research in the area of designing good reservoirs. A reservoir initialized, with or without *a known priori*, should exhibit good *network performance* and also maintain *network stability* through both training and generalization phases. Initializing the reservoir randomly is not always feasible due to these desirable characteristics. In [33], Natschläger et al. showcase the importance of operating a network near criticality. A thorough understanding of ordered, critical and chaotic systems is important to initialize good reservoirs. There are criteria functions or stability constraints that can assess the regime in which the reservoir is operating for different input excitations presented to a network at different initial states [33, 34].

2.3.1 Random Boolean Networks

In living organisms, the genome is a dynamic network [35]. The behavioral manifestations is through interactions between various genes in the genome. The genome is thus a network of integrating genes where one gene regulates or influences the functionality of the gene it is connected to in the network. *Random Boolean Networks* (RBNs) are models of such *Gene Regulatory Networks* developed by Stuart Kauffman and Rene Thomas [35].

An RBN is a dynamic network comprising of N nodes. Each node in the RBN can exist in one of the two Boolean states: [0, 1]. The number of inputs to each node of the RBN is determined by its connectivity. The random connectivity could be the mean node connectivity K or the exact number of connections to a node. We deal with an RBN's architecture in detail in Section 4.

The dynamics of an RBN is largely decided by the number of nodes N and the connectivity K between these nodes. Since each node can be in one of the two Boolean states ie., one or zero, the number of states that an RBN kernel can be in is 2^N [35].

In a classic RBN, which is synchronous in nature, the state of every node gets updated at the same time. The state at $t + 1$ of each node depends on the state of its input or in-degree nodes at t. The state of an RBN is the cumulative representation of the states at each node. The RBN, through its operational life cycle, starts from a random initial state and goes through a sequence of updates based on its updating scheme. Eventually, since for a given RBN, the number of possible states is known and finite, the RBN settles onto a fixed state or a cyclic attractor [36].

The states of an RBN that do not lead to any other state are called *attractor*

states. Attractor states are terminal states. It is possible that an attractor is a cyclic combination of more than one state. In such cases the attractor is called a *cyclic* attractor. If an attractor comprises of just one state, it is called a *point* attractor. The number of predecessor states for a certain state is called the *in-degree* for that state. The states that lead to an attractor state are collectively called the *attractor basin*. The time taken to reach an attractor state is called the *transient time* [37].

Based on their connectivity, RBNs are found to transition from an order-chaotic regime at K_c =2. Based on this rule, the RBN can operate in three possible phases [11]:

- *Ordered phase*: RBNs with connectivity $K < K_c$. RBNs in this phase show faster convergence. Due to faster convergence, the attractor basins in this phase are smaller. High in-degree states can be observed in this system.

- *Critical phase*: RBNs with connectivity $K = K_c$. RBNs in this phase show slower convergence in comparison to the ordered phase. This phase also termed *the edge of chaos* has been studied with interest because of its observed elevated levels of computational ability.

- *Chaotic phase*: RBNs with connectivity $K > K_c$. RBNs in this phase converge slowly to an attractor state. This is a phase dynamically opposite to the ordered phase with longer attractor basins.

It has been shown that the RBNs compute when the stimulus to the RBN is carried around to all parts of the network. The computing ability of the RBN is said to be optimal at the edge of chaos or when the connectivity of the RBN is critical. This is because critical connectivity is seen to be advantageous over

the ordered phase, which needs more time to perform the same computations and the chaotic phase which exhibits redundancies involved in achieving stability in its dynamics to perform any computation [11, 12, 14, 15, 36].

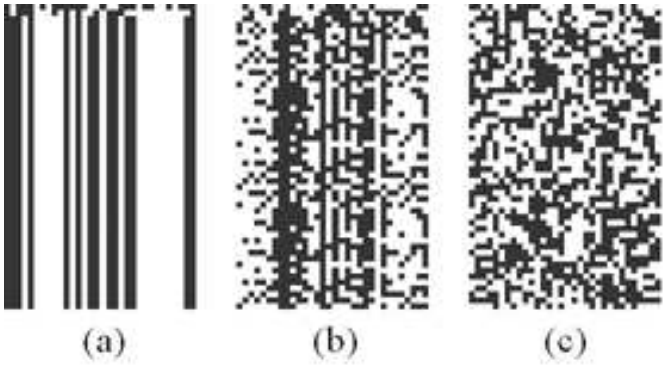


Figure 2.4: Trajectories through state space of RBNs within different phases, $N = 32$. A square represents the state of a node. The initial states on top, time flows downwards. a) ordered, $K = 1$. b) critical, $K = 2$. c) chaotic, $K = 5$ Source: [4].

RBNs composed of Boolean processing units capable of computing, provide an attractive alternative to current day Von Neumann architectures. The self-assembled nature of such networks brings a huge advantage to building physical computing devices with random processing units and random wiring. This helps to address the problems faced currently related to the cost of wiring with long-range interconnects. Also RBN provides an alternative to replace the current day sequential nature of computation with a parallel computing paradigm which is closer to how computation happens biologically. It also provides an alternate way to address the physical limitations related to the extent of downscaling that can be achieved with physical devices [11]. This research is dedicated to experimenting with the RBN's computational ability to be programmed to perform as real-time autonomous controllers.

2.4 Introduction to Reinforcement Learning

Reinforcement Learning (RL) is the field of machine learning which deals with training learning system through a goal-driven technique where the learner continuously interacts with its yet unknown environment. Reinforcement learning approaches which find a wide range of applications, especially in the field of control theory, make an attempt at bridging the gap between machine learning and other engineering disciplines. Reinforcement learning itself is not considered to be an approach to solving a certain problem, but the whole range of problems dealing with interactive goal-driven learning are collectively defined as the reinforcement learning problems. Any approach taken to solve this problem is said to be a reinforcement learning approach [5].

2.4.1 Basics

There are certain important terminologies and notations that repeat through reinforcement learning related discussions, which we need to familiarize ourselves with.

Reinforcement learning techniques involve training a decision maker (or *agent*) through continuous interactions with the *environment*. The *agent* at each discrete timestep senses the current situation called the *state*, presented to it by the *environment*. The *agent* responds to this stimulus by performing an *action*. In addition to presenting the *state* to the *agent*, the *environment* is also responsible for assessing the quality of the *action* performed by the *agent* by providing a *reward* or *penalty*. The timesteps need not have to be actual instances in real-time, but are the actual decision making points of the *agent* [5].

The underlying environment which the agent explores through an iterative

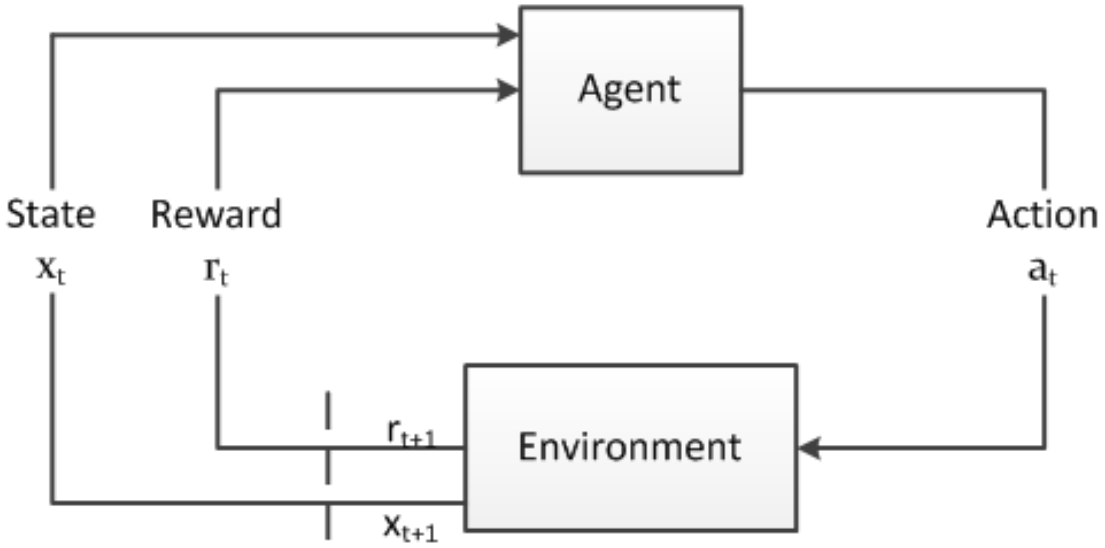


Figure 2.5: Agent and environment interaction in reinforcement learning; Source: [5].

training process is in many problem contexts represented by the *model* of the environment. The *model* usually defines the mapping of the current and/or past state-action of the environment to its next state-action, through either a mathematical formulation or canonical lookups. All reinforcement learning techniques, however, do not need to have knowledge of how the underlying *model* works. However, a working representation of an environmental model is necessary for the agent to be able to adapt and learn in that environment [5].

A *policy* maps the current state to the next action to be taken. A *policy* can be a function that maps states to actions or can be a complex lookup table. The *policy* is essentially that set of actions (or a strategy) the agent takes in its journey through the environment driven by its goal/objective. Each action performed is followed by a reinforcement (or a reward) from the environment. The agent, during its training, strengthens its affinity towards performing those action which maximizes its chances of getting a reward. An optimal *policy* maps an agent's

state to actions that help maximize its future rewards [5].

The agent interacts with its environment at each discrete timestep $t_1, t_2, t_3, ..., t_n$. At each timestep t, it is presented with the environment's state $s_t \in S$, where S is the space containing all possible states for which the agent responds with an action $a_t \in A(s_t)$, where A is all possible actions for the environment's state s_t at that timestep. The action a_t at state s_t is performed based on the policy π_t, which is a function of $\pi_t(s, a)$, such that with policy as π_t, action $a = a_t$ is performed for state $s = s_t$ at time t. In response to the action a_t, the agent receives a rewards $r_{t+1} \in \Re$ from the environment. Figure 2.5 illustrates this agent-environment interaction [5].

The agent in a reinforcement learning context is goal-driven. The quality of the action performed at a particular state towards achieving the objective is assessed by an invisible critic. This critic is more or less a *reward function* which maps the state-action pair to a reward/penalty to assess the quality of the current *policy* The *reward function* is environment-and-goal specific and remains unaltered. The agent interacts with the environment and accumulates rewards/penalties through this interaction in the process converging towards an optimal policy [5].

The *value function* maps the current state-action pair to future cumulative rewards. While the *reward function* is more concerned with immediate rewards, the *value function* plays emphasis on how relevant the current state is in maximizing the future rewards. The *reward function* and the *value function* for a particular state-action could map to contrasting values, in a sense that if for a particular state-action pair the *reward function* maps it to a very high reward, the same state-action pair could be mapped to a low returns value by the *value function* indicating, though this state-action pair guarantees immediate returns it may be bad in the long run. In such a case a *reward function* may seem to be completely irrelevant.

However, the *value function* is not readily available, and is something that the agent needs to learn to approximate through its interactions with the environment, driven towards a goal that is defined by a stationary *reward function* [5].

Some examples of reinforcement learning include [5]:

- a player learning to play tic-tac-toe. The decisions made by the player at every step depends on the previous move of the opponent and in anticipation of the opponents next moves, aimed at maximizing his/her chances to win.
- an autonomous mobile robot which collects trash around the room. It is faced with a decision making problem at every step, to decide which direction to go to optimize its chances of finding trash.
- an adaptive air-plane controller, used to automatically adjust parameters to maintain stability during flight.

Evaluation vs Instruction

The primary difference a reinforcement learning-based training has with other machine learning approaches, is that, unlike supervised learning approaches there is no concept of a *teacher* which instructs the learning agent with the right action to take at a particular state. In supervised learning approaches, the goal is to reduce the error between the supplied set of target outputs and the current output through a training process. While in reinforcement learning, the agent does not have knowledge of the right action to execute at a particular state. There is a concept of a *critic*, who unlike the teacher evaluates the quality of the action that has just been performed by the agent. The agent during the initial phases of training

does not have complete knowledge of how good the other actions are in comparison to the one, which was just executed. So instead of an *instructive* feedback that a supervised learning systems' *teacher* provides, in a reinforcement learning context, the *critic* provides an *evaluative* feedback. The setting in reinforcement learning is *non-associative* in comparison to the *associative input-target* approach of supervised learning [5].

During the initial phases of training, the reinforcement learning agent starts with a value function that may or may not be based on *a priori*.

Let $V^*(a)$ be the accurate value estimate of an agent for action a. The accurate value estimates for an action is the mean reward when the action is executed by the agent. This can be achieved by averaging all rewards received by the action for a known number of executions. Let $V_t(a)$ be the value estimates for action a at episode t. We can calculate $V_t(a)$ based on Equation 2.13 [5]:

$$V_t(a) = \frac{r_1 + r_2 + r_3.....r_{K_a}}{K_a}, \tag{2.13}$$

where $r_1, r_2, r_3.....r_{K_a}$ are rewards received at each execution of the action for K_a times prior to episode t.

At $K_a = 0$, $V_t(a) \rightarrow V_0(a) = 0$

As $K_a \rightarrow \infty$, $V_t(a) \rightarrow V^*(a)$

This is just a very simple way of estimating action values based on the rule of averages. Though not the most accurate way of estimating values, it gives an overview of how calculating value estimates works [5].

The optimal action a^* to be executed from the action set is based on a simple selection rule as shown in Equation 2.14. That is, the action with the maximum

value estimate is the one to be selected.

$$V_t(a^*) = max_a V_t(a) \tag{2.14}$$

Exploration vs Exploitation

A policy based on the action selection rule in Equation 2.14 is called a *greedy* policy, where the agent tries to increase its chances of maximizing future rewards at each step by selecting an action with a maximum value estimate. However, during the initial stages of training, the value estimates are inaccurate and premature. By following a greedy approach, the agent only strengthens those actions that get selected during the initial training phases. In reinforcement learning, an action can be accurately evaluated as good or bad by only sampling it in the state-action space [5].

This leads to the complexity of deciding when it is optimal to have the agent explore the action space by following a *non-greedy* policy and when to exploit the value estimates and choose a more greedy action. This balance between exploration and exploitation is a popular problem faced by the reinforcement learning community. There are several approaches taken to ensure proper exploration during the initial phases of training so that a trained agent converges towards accurate value estimates and so can follow a policy which is more exploitive (or greedy).

The most popular approach taken to balance exploration vs exploitation in the search for an optimal policy is to deviate from the greedy approach to a partially-greedy approach and select actions other than the one that satisfies Equation 2.14. Such an approach is termed the ε-greedy policy [5]. An ε-greedy policy is greedy most of the time, but selects a random action from the action space only a small

number of times i.e., with probability ε. With such an approach it is guaranteed that value estimates for all actions in the action space converge:

As $K_a \to \infty$, $V_t(a) \to V^*(a) \; \forall \; a$

As the training progresses, $\varepsilon \to 0$, which means that the policy tends towards a purely greedy policy.

Episodic vs continuing tasks

Equation 2.13 can be alternatively represented as [5]:

$$V_{t+1}(a) = \frac{1}{K_a + 1} \sum_{i=1}^{K_a+1} r_i, \tag{2.15}$$

which simplifies to,

$$V_{t+1}(a) = V_t(a) + \frac{1}{K_a + 1}[r_{k+1} - V_t(a)] \tag{2.16}$$

$K_a + 1$ is the step-size and depends on the problem context. To generalize, this step-size term is commonly replaced with the parameter α which is alternately also called the learning rate [5]. Essentially, $\alpha = \frac{1}{k}$ and $\alpha \in 0 \leq \alpha \leq 1$, the value of α depends on the length in timesteps for that particular task.

We have seen from Equation 2.13 that the discounted future returns is the sum of all rewards across timesteps. The number of timesteps could be until a terminal state T is reached, which is the length in timesteps for an *episode*. An *episode* is the subsequence of the agent's journey through its environment. It is a natural sequence of events where the agent starts at a certain initial state (or one of the state in the initial set of environmental states) and interacts with the environment and terminates in one of the terminal states, which marks the end

of the subsequence (or the *episode*). This is typical for problems dealing with plays of a particular game or an agent's trip through a grid world. Such tasks are termed *episodic* (or finite time horizon problems). In some cases, the timesteps are not naturally broken down into episodes, and so the agent's interaction with the environment is indefinite. Such tasks are termed *continuing* tasks (or infinite time horizon problems) and continue indefinitely, i.e., $T \rightarrow \infty$ [5].

Discounting Rewards

The Equation 2.15 is a very simple evaluation of the value function. In practice, the value function that the agent learns to approximate is essentially the sum of discounted future rewards. It chooses an action a_t, to maximize this sum of discounted future rewards. The sum of discounted future rewards is represented by [5]:

$$R_t(a) = \sum_{k=0}^{T} \gamma^k r_{t+k+1}, \tag{2.17}$$

where T = finite for episodic tasks and $T = \infty$ for continuous tasks. $\gamma \in 0 \leq \gamma \leq 1$ is the discount rate.

The discount rate/factor is used to assess the importance given to future rewards at the current timestep. For instance, a reward received at time $t = k$ is given a weight of γ^k as opposed to the immediate reward, which receives the weight of 1. The discount rate is used to control the importance of future rewards in the current time instance [5].

As $\gamma \rightarrow 0$, $R_t(a) \rightarrow r_{t+1}$, which means the agent is more myopic (or short-sighted) and performs an action a_t to maximize only the immediate reward. This

does not necessarily guarantee convergence towards an optimal policy, since maximizing immediate reward does not necessarily always maximize the sum of future rewards [5].

As $\gamma \to 1$, $R_t(a) \to \sum_{k=0}^{T} \gamma^k r_{t+k+1}$, which means the agent is far-sighted and gives more importance to future rewards.

The ideal value of γ is specific to the problem context being considered. In our experiments, where we deal with agents foraging in a 2-D grid-world, we use a value of $\gamma = 0.91$, since the agent needs to be long-sighted and maximize the future rewards over the entire episode.

2.4.2 Generalized Policy Iteration - Cyclic nature of convergence

In a reinforcement learning setup the environment is usually considered to be a finite *Markovian Decision Process* (MDP) with finite state and action spaces S and $A(s)$. Reinforcement learning systems are heavily dependent on the knowledge of two important functions. The state transition function [5]:

$$s_{t+1} = P(s_t, a_t) \tag{2.18}$$

and the reward function [5].

$$r_{t+1} = R(s_{t+1}, a_t) \tag{2.19}$$

Reinforcement learning techniques involve updating the value function in its search for optimal policies. The value function update (or the backup) through a reinforcement learning process follows the Bellman recursion [38] of updating value functions of states encountered by taking existing value functions of future states,

that the current states transition into, as references or targets [5]. An optimal value function V^* can be represented as [5]:

$$V^*(s) = max_a \sum_{s'} [R(s,a) + \gamma V^*(s')], \tag{2.20}$$

where

$$s' = P(s,a) \tag{2.21}$$

The objective of reinforcement learning is to start from a random policy, π and value function related to that, V^π and to reach the optimal policy, π^* by converging towards the corresponding optimal value function, V^*. A reinforcement learning process goes through iterations of value function improvements and consequent policy improvements to converge towards this optimality. In the most simple algorithm called the *value iteration*, the values are backed up in an iterative approach using the Bellman equation [38] as an update rule. The *value iteration* algorithm is as follows:

Algorithm 1 Value Iteration; Source: [5]

initialize V(s) = 0 $\forall$ s $\in$ S

repeat

$\Delta \leftarrow 0$

for all $s \in S$ **do**

$v \leftarrow V(s)$

$V(s) \leftarrow max_a \sum_{s'}[R(s,a) + \gamma V^*(s')]$

$\Delta \leftarrow max(\Delta, |v - V(s)|)$

end for

until $\Delta < \theta$ (a small positive number)

Output is policy: $\pi|\pi(s) = argmax_a \sum_{s'}[R(s,a) + \gamma V^*(s')]$, where s' is the future state.

This *value iteration* algorithm starts from a random initialized value function and hence a random policy based on the value function estimates. Each iteration (or sweep) of this algorithm involves evaluating this policy based on this initial value function set. That is, the action chosen to be performed is based on this random initial policy. An error signal is then generated to improve the value function estimates. Improved value function estimates essentially lead to an improved policy. So in each iteration the *value iteration* algorithm does a policy evaluation of the current policy followed by an update/backup of value estimates which results in policy improvement. The iterations are carried out for as long as the value estimates converge, with very small changes between iterations, hence guaranteeing an optimal policy [5].

Figure 2.6 summarizes the cyclic relationship between the policy evaluation and improvement. Convergence in case of a reinforcement learning context is

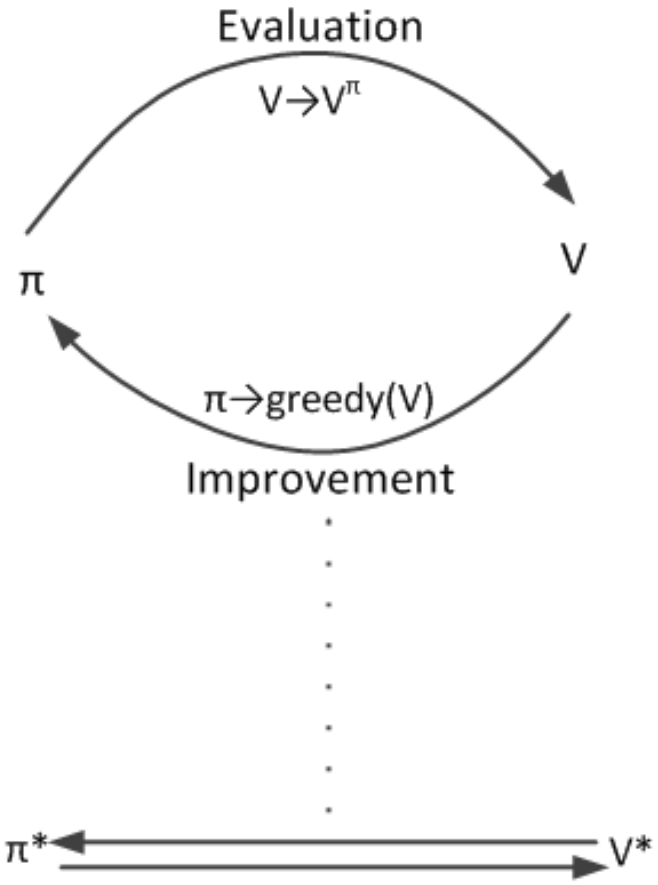


Figure 2.6: Value and policy interaction until convergence; Source: [5]

supposed to have been achieved when the value function estimates are consistent with the current policy being evaluated . And the policy is said to have converged to optimality when it is greedy with respect to the value function estimates being considered. Though the two processes seem to compete with each other they also co-operate to achieve optimality [5]. It is a struggle to maintain a greedy policy towards a value function, while the value function might have changed with the policy, and hence rendering it incorrect for the current greedy policy. Also, trying to execute a policy consistent with the changed value function renders the current policy non-greedy, and hence non-optimal. Trying to chase this cyclic dependency

of the policy and the value function and vice-versa is surprisingly the backbone of reinforcement learning in its tryst for convergence towards optimality [5].

The value function has a linear relationship with the Equation 2.18 and Equation 2.19 and the accuracy of value function estimates depends on the accuracy with which these two functions define the underlying model. We can recollect that the value function is represented by:

$$V_s^{\pi} = E_{\pi}\{R_t | s_t = s\}, \tag{2.22}$$

where R_t is defined by Equation.2.17

2.4.3 Markov Decision Processes

In the reinforcement learning context the environmental model defining these functions may or may not be known. When the model is not known, these functions are usually approximated through system identification. With a smaller state-space these approximators can be built through tabular methods, which use less memory. However, with the increase in the state-space, tabular approximators are not computationally feasible and so are replaced by parametrized dynamic approximators trained through prior behavioral observation samples. Since for a huge state-space it is not possible to obtain the system behavior for every possible combination of the state-action pair, the accuracy of the value function estimates becomes exceedingly dependent on the generalization ability of such trained function approximators.

For a problem domain considered in a reinforcement learning context, the state of the environment that is provided to the agent should contain enough information for the agent to be able to make a decision. The decision that the agent needs to make is driven by the pre-defined objective. The environment in which the agent

operates may be very complex to be fully represented. However, the agent does not need to have the entire information about the environment. The environment state signal presented to the agent needs to encompass just that relevant information/feature set that the agent needs to achieve its objective/goal. A system which is *Markovian* has this relationship [5]:

$$s_{t+1} = f(s_t) + g(a_t), \tag{2.23}$$

where the state at time $t+1$ can be wholly deduced and is a function of state and action taken at time t. Reinforcement learning problem applied to a Markovian system is termed a *Markovian Decision Process* (MDP) [5]. The relationship functions f and g could be linear or non-linear and possess the *Markovian property.*

However, in most systems the relationship between a state and its past states can be represented as

$$s_{t+1} = f(s_t, s_{t-1}, s_{t-2}.....) + g(a_t, a_{t-1}, a_{t-2}, ...), \tag{2.24}$$

where the state at time $t+1$ is a linear or a non-linear function of not just the current state and action but a subset of their past values. Such systems are termed non-Markovian systems. Werbos et al. [39] successfully approximated such systems through a Multi Layer Perceptron (MLP) by introducing a tap delay layer before the actual input layer to get a representation similar to $\{s_t, s_{t-1}, s_{t-2}....\}$ as input.

However, for the system identification of such environments, it is seldom a known *priori* as to how far back in time the states need to be considered to wholly represent the current state. This length is called *temporal embedding* of

the state. Takens' theorem states that it is possible to transform a non-Markovian (incomplete) state space to a higher dimensional state space where every state is a function of a temporal embedding of the original states. This projection of the original state-space into a higher dimensional feature space can be achieved in such a way that the new state-space is deemed Markovian or complete [16] . The new state space can be represented as:

$$x_t = \varphi(s_t, s_{t-1}, s_{t-2}.....s_{t-k}) \mid x_{t+1} = h(x_t, a_t), \tag{2.25}$$

where space X represents the complete state-space reconstructed from the partially observable, temporally embedded, incomplete state-space S.

Mathematically, calculating the value of k needed to achieve such a transformation is computationally intensive. In approaches which try to achieve this through a parameterized dynamic system, such as an ANN with a tap delay layer, the value of k is deduced by a trial and error approach. This becomes cumbersome with the increase in the value of k and as well as with the increase in the dimension of S, which is the original state-space. For simplicity, the original states are considered only as a function of current and past states and current action, without dealing with past action values. In reinforcement learning, the accuracy in estimation of the value function is crucial for the decision making. And the value function accuracy completely depends on the accuracy of the state-space X in Equation 2.25.

2.4.4 Temporal Difference Method

For our approach, we train an agent to learn interactively based on the *Temporal Difference* approach. In this section, we explain the algorithm and highlight its difference to other similar reinforcement learning approaches. The *Temporal*

Difference approach (TD) in reinforcement learning is a concept in reinforcement learning. It tries to address the drawbacks of other reinforcement learning procedures, such as Dynamic Programming (DP) [38] and Monte-Carlo methods. The TD approach functions without needing a model of the underlying environment. While Monte-Carlo techniques treat the average sampled returns (or rewards) as a target to train the expected value estimates, the DP method treats the current estimates of a future state as a target to train the expected value estimates. TD techniques, like Monte Carlo techniques, sample out the successor state-space and update expected value estimates of the current state based on the current value estimates of the sampled successor state and the returns observed due to the action performed. The updating of current estimates based on target is also termed *back-up* [5]. While it is similar to DP and uses bootstrapping approach to update based on the current value estimates of the successor state, TD does not stick to one successor state. The TD approach uses an online approach to improving value estimates, while the Monte-Carlo approach waits until the end of episode and uses the average of observed samples at the end of episode.

The following algorithm shows the steps involved in a tabular TD(0) approach [5]. A tabular approach involves canonical lookup tables to hold value estimates for the state-action pairs.

Algorithm 2 Tabular Temporal Difference; Source: [5]

Initialize V(s) arbitrarily, π to the policy to be evaluated

for all episodes **do**

Initialize state s

for all timesteps in each episode **do**

$a \leftarrow$ action given by π for s

Take action a; observe reward r , and next state, $s^{'}$

V(s)$\leftarrow$V(s) + $\alpha(r + \gamma \text{V}(s^{'})$ - V(s))

$s \leftarrow s^{'}$

end for

end for

Q-Learning

We use the Q-Learning algorithm in our setup as a benchmark to compare the performance and accuracy of our setup. In this section, we explain the Q-Learning algorithm.

In control systems, the state-value functions that we have been discussing about are usually replaced by an action-value function, which is for a specific state-action pair. Just like the TD(0) approach, which transitions from state-to-state, the Q-Learning approach involves transition from state-action pair to state-action pair.

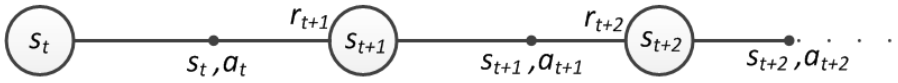


Figure 2.7: Q-Learning on-policy Control; Source: [5]

The following algorithm shows the step-wise progression of an On-Policy Q-Learning approach:

Algorithm 3 Q-Learning On-Policy approach [5]

Initialize Q(s, a) arbitrarily

for all episodes **do**

Initialize state s

Choose a from s using policy derived from Q (e.g., ε-greedy)

for all timesteps in each episode **do**

Take action a; observe reward r , and next state, $s^{'}$

Choose $a^{'}$ from $s^{'}$ using policy derived from Q (e.g., ε-greedy)

Q$(s, a) \leftarrow$Q(s, a) + $\alpha(r + \gamma$Q$(s^{'}, a^{'})$ - Q$(s, a))$

$s \leftarrow s^{'}$; $a \leftarrow a^{'}$

end for

end for

2.5 Function Approximators

From the previous chapters, we realize that all reinforcement learning techniques are predictive models, where the learner(or agent) learns to estimate future rewards based on its sampled past observations in the state-action space. As discussed earlier, in non-Markovian domains, these estimates of future gains depends on the state transitions through time. A lot of approaches have been considered before in dealing with reinforcement learning in non-Markovian domain. The most popular one among them being training function approximators like Artificial Neural Networks (ANNs), as done in [39], to act as predictive models. However, for non-Markovian domains, these function approximators also need to have an internal state that spans across memory. These dependencies on past states can be extracted either explicitly or implicitly.

The canonical tabular approaches dealing with state-action pairs in the state-action space are statistical and explicit in nature. The most popular among explicit embedded architectures include the Auto-Regressive Moving Average (ARMA) models [40]. ARMA is a combination of auto-regression and moving average models, where the future state is considered to be a linearly weighted sum of q historic means of the time-series input data. In such models, training is figuring out through an empirical search, the optimal length, q, so that the future state is properly represented. However, with such techniques used in reinforcement learning, the training would not just involve search for q but also the training involved to achieve convergence to an accurate future value estimates for each model, with varying q. Also, ARMA works well only with linear environmental models/problem domains.

However, when dealing with implicit models such as ANNs, this internal state spanning across time is represented more implicitly. The approach taken by most ANNs is well summarized by this quote:

" *Note that in any environment the outcome at any given time may be affected arbitrarily by prior events. The agent may not know which prior events matter and therefore the agent does not know how to enlarge the state space to render the process Markovian.* " [41]

In Chapter 2.3, we discussed about a new paradigm called *Reservoir Computing* which implicitly embeds temporal information in a more efficient manner than ANNs, while having significant training advantages over RNNs. For our research, we consider setting up RBN-based reservoirs to work as function approximators in non-Markovian problem contexts to predict future reward estimates.

3

Related Work

In this section we discuss the prior work done in the area of reservoir computing (including ESNs, LSMs and RBNs), where reservoirs are setup to solve reinforcement learning tasks by leveraging their ability to transform a non-Markovian state representation of a problem context onto a higher-dimensional feature space, which is complete or Markovian in nature. In most of these works, the reservoir is setup as a predictive model to estimate/predict the cumulative future rewards to function as controllers. We are also interested in work that deals with the reservoir's ability to transform time series information into spatial state space representations.

Schrauwen et al. [42–44] have carried out research in setting up ESNs as a mobile robot controller. The robot is trained through examples of environment exploration, target seeking and obstacle avoidance behavior movements and learn to generalize these behaviors during the testing phase. The training examples (including input signal - output behavior mappings) for varying scenarios are collected through a pre-designed existing INASY (Intelligent autonomous NAvigation SYstem) controller. The ESN readout is trained through linear regression from the training examples obtained through this pre-existing INASY controller [42–44].

In another experiment, the T-maze problem (shown in Figure 3.1) is solved through training an autonomous agent to navigate through obstacles towards its goal. This problem is temporal in nature since the autonomous robot receives a cue early on in its journey and needs to remember the cue to perform a related action later on in time to achieve the goal. A pre-existing C++ robot simulation

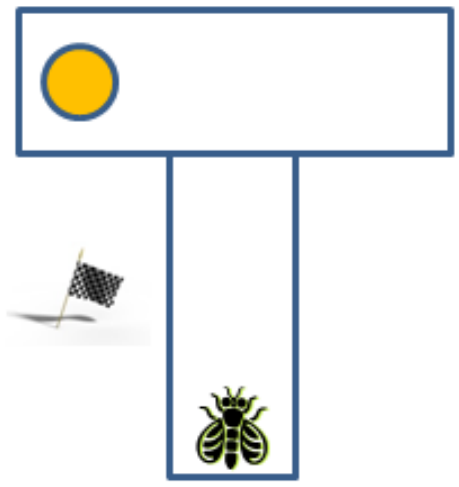


Figure 3.1: A T-MAZE problem.

framework is used for acquisition of training data (including input-output samples) to train the reservoir, implemented in MATLAB. The sensor inputs received by the agent is color-coded to be able to distinguish between the target, obstacles and the cue. The RC-model, which is also ESN-based, is trained by adjusting its readout layer weights through linear regression [45].

ESN-based robot controllers have been designed in contexts where the robot has the ability for event detection and robot location sensing. These techniques, which were originally modelled through traditional Simultaneous Localization and Mapping (SLAM) techniques were very expensive computationally. Through RC these complex robot behaviors are shown to be successfully controllable through just training the linear readout layer [46]. Noisy training data (input-output samples) obtained from existing simulation models are used to train an RC-based agent for event detection and localization [47].

Kyriakos C. Chatzidimitriou et al. used an ESN-based reservoir in a *Transfer Learning* task, where an already learned behavior from a source task is used to

enhance the learning procedure in a target task. The tasks may belong to the same domain as in two different mazes or different domains such as chess and checkers. Also a technique called *NeuroEvolution of Augmented Reservoirs* (NEAR), is used to search for the optimal ESN structural model for the problem context. It is a gene evolution technique which uses crossover, recombination, with historical markings from the source task, including some ESN based parameters to come-up with an ESN structure that solves the new task [48].

Reservoirs with their temporal dynamics act as powerful tools for time-series forecasting. Francis wyfells et al. developed a reservoir based prediction scheme for fast and accurate time series prediction based on wavelet decomposition. The input time series in broken down into components of different time scales using time series decomposition. The obtained trend series and detail coefficients were predicted using reservoir computing. An ESN based reservoir built of band-pass filter neurons are used for wavelet decomposition [49].

Stefan Kok, in his doctoral thesis work showed how a temporal input sequence representing MIDI files can be classified through a Liquid State Machine setup in a supervised learning context [50].

Bongard et al. have setup RBNs as a robotic controller, while the RBN structure which works optimally in the problem context being determined by an evolutionary algorithm [51]. The RBN is trained to function as a controller to a robotic arm, which seeks a circular object in different environments and grasps the object when found.

Though the work discussed so far have used reservoirs as state controllers in Markovian or non-Markovian domains, the training of the reservoir in all these experiments is based on supervised learning techniques, since the input-output

samples are available to train and test on, or on evolutionary approaches.

István Szita et al., have setup ESN in k-order *Markovian Decision Process* contexts to solve simple mazes and also to solve Monty Hall problems [52] using reinforcement learning. Keith A. Bush, in his doctoral thesis, uses an Echo State Network to model stable, convergent, accurate and scalable non-Markovian reinforcement learning domains. A Mountain Car problem, which is a popular reinforcement learning problem, modelled through an ESN, was trained using TD-based reinforcement learning. In a Mountain Car problem, an underpowered car needs to be driven out of a steep valley. The car is incapable of reaching the goal situated at the top of the valley along the steep upward path even at full throttle. The solution involves backing away from the goal over the opposite slope and then applying full power to build enough momentum to carry it over the other slope towards the goal. This task involves sacrificing immediate benefits to achieve an optimal final reward. The ESN is shown to exhibit properties similar to a canonical look-up table or Artificial Neural Networks in functioning as value estimators for the state-action pair in this scenario [53].

Alexander Kazeka implements mobile robots also to solve the Mountain Car problem by setting up the reservoir, a Liquid State Machine, to solve the task through an iterative policy improvement technique. An LSM is setup as a value estimator in a non-Markovian navigation task. The dynamics in the LSM, with its fading memory, approximates the original state space into a Markovian state representation to solve the task. However, the results of setting up such a controller have been discouraging in this problem context [54].

In this section, we discussed work related to using reservoirs as state controllers. We note that very little work has been done previously to use LSMs and no work (to

the best of our knowledge) has been done to use RBNs as state controllers trained through reinforcement learning techniques. In this thesis, we conduct experiments to setup RBNs as state controllers in both *Markovian* and *non-Markovian* domains.

4

Methodology

In this section, we attempt to describe in detail our approach towards designing an autonomous controller based on a *Random Boolean Network*. It is a step taken towards programming self-assembled architectures that can adaptively learn to control in model-free goal driven contexts. Our simulation framework includes setting up a RBN-based reservoir in a reinforcement learning context. Reinforcement learning problems do not have a clear input-output mapping, like in supervised learning contexts. We train the RBN-based learner/agent, in this setup, based on error signals generated using *rewards/reinforcements* observed through interactions with the environment. The RBN-based agent in this setup is essentially trained to function as a parameterized dynamic system that learns to approximate cumulative future estimates for each action. Based on uniform sampling of all possible actions at each state and based on the resulting rewards, the agent trains its read-out layer weights to indicate the probability with which performing each action at a particular input state is likely to maximize future rewards earned. The main modules include:

- RBN framework
- Readout layer
- Environment the agent is trying to learn
- Differential Evolution to search the initial parameter space

4.1 Framework

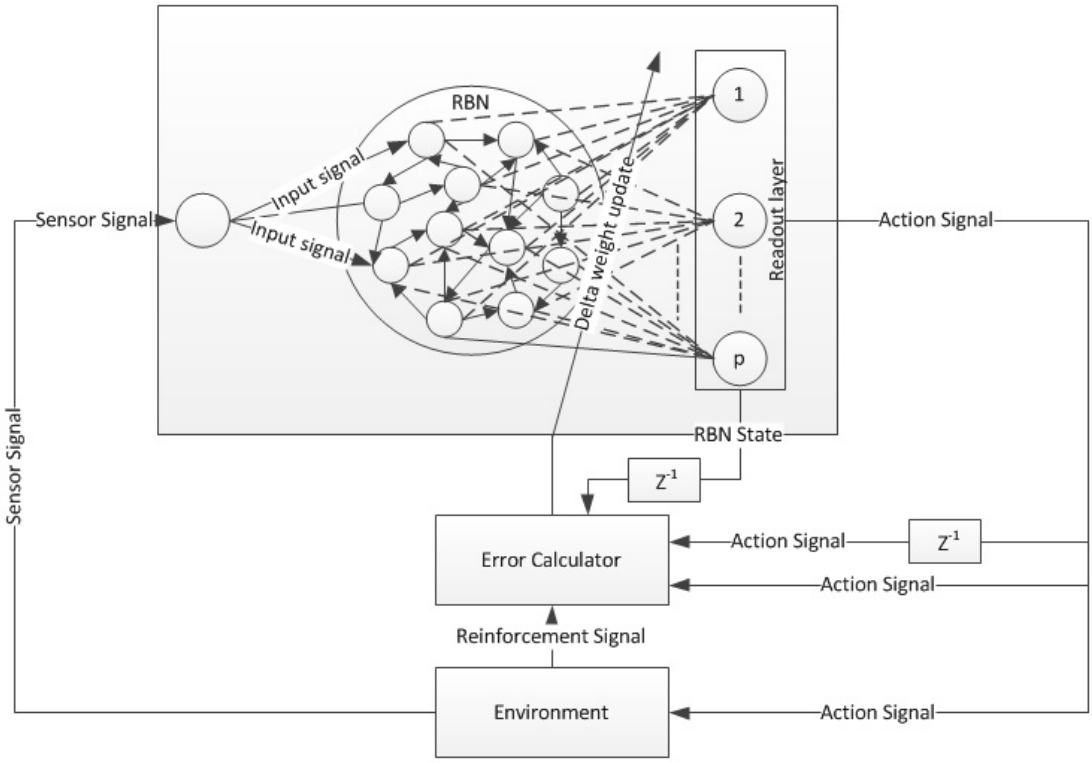


Figure 4.1: This design block diagram indicates an RBN-based reservoir augmented with a read-out layer. It interacts with a model-free environment by observing the perceived environment state through the *Sensor Signal*, s_t. Based on the input signal values, estimates are generated at each read-out layer node. The read-out layer node that generates the maximum value estimate is picked up as the *Action Signal*, a_t. This *Action Signal*, a_t is evaluated on the *Environment* and the resulting *Reinforcement Signal*, r_t, is observed. This *Reinforcement Signal*, along with the RBN's state, x_t, the value estimate of the *Action Signal*, a_t, and the value estimate of the *Action Signal* generated during the next timestep $t+1$ is used to generate the error signal by an *Error Calculator*. This error signal is used to update the read-out layer weights. The block labelled Z^{-1} representing a delay tab, introduces a delay of one timestep on the signal in its path.

Figure 4.1 gives an overview of our setup. An RBN-based learner receives a binary sensor signal or the state s_t from a *model-free environment*. This input signal perturbs the RBN-based reservoir. The state x_t of the reservoir is observed

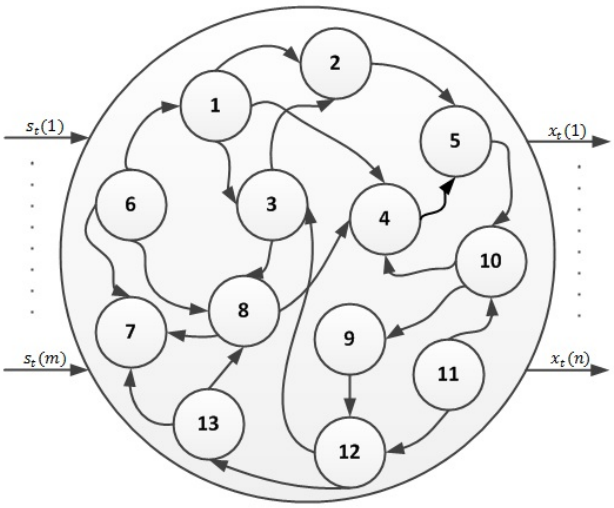
through a *read-out layer*. The action a_t to be taken is decided based on this observed output and evaluated on the environment. Unlike in supervised learning contexts, mapping of states to action is not a known *priori* in such decision making systems. During the training process, a mapping of the state and its optimal action is developed. This involves sampling the entire action space A for every state s_t and observing the resulting rewards repeatedly over several training episodes. An *error calculator* observes the signals a_t, a_{t+1} and x_t and the reward r_t and calculates the *error signal* to update the *weights*.

- *Random Boolean Reservoir*: We use a C++ implementation of the RBN Toolbox implemented by Christian Schwarzer and Christof Teuscher, which is available at `http://www.teuscher.ch/rbntoolbox/`. The RBN considered is a classic synchronous RBN [35, 55, 56].

  The RBN reservoir comprises of N nodes, which by behavior are classified into:

  - *input nodes*,
  - *processing nodes,* and
  - *output nodes*

  The connectivity between nodes in the reservoir is based on K, which represents mean connectivity. The mean connectivity, K, represents the average input connections to each Boolean node in the RBN. The number of nodes is inclusive of all node types. The input nodes are indexed between 0 to M - 1, while the processing/output nodes are indexed between M to N-1, where M stands for number of inputs. There is no double connectivity



$x_t(1)$	$x_t(8)$	$x_t(10)$	$x_{t+1}(4)$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Figure 4.2: This diagram shows an RBN being perturbed by input signal, $\mathbf{s}_t$, of dimension m. The random Boolean nodes in the RBN-based reservoir are connected recurrently based on average connectivity K. The input signal through the reservoir dynamics is transformed to a high-dimensional reservoir state signal $\mathbf{x}_t$, of dimension N, such that $N > m$. Each Boolean node in the reservoir is activated by a random Boolean function. The table represents *lookup table* (LUT) for the random Boolean function that activates node 4 in the reservoir.

allowed between nodes, however loops are allowed (a node is allowed to connect to itself). This particular design of RBN is based on the NK networks introduced by Kauffman [36].

To be able to spread perturbations through the entire reservoir, the connectivity from the input nodes to the other processing nodes is controlled through a parameter L. This denotes the exact number of connections from the nodes in the reservoir which receive the input perturbations to other processing nodes in the reservoir, which are randomly chosen [15]. We decide the value of L based on the reservoir size N, based on this relationship $L = 20\%N$.

The Boolean activation functions associated to each RBN node, for example,

consider Figure 4.2 for Boolean function activating node 4, is one of the 2^{2^K} possible random Boolean functions which the nodes in the reservoir can be associated to. During the reservoir initialization, each node in the reservoir is associated to one of the 2^{2^K} possible functions randomly based on a uniform distribution. This mapping between a node and its Boolean activation function is stored in the memory and fixed for the life-time of the reservoir.

The input to the RBN is the binary state signal denoted by $\mathbf{s}_t$. The output from the RBN is the higher dimensional state the input signal is transformed to through the recurrent dynamics in the RBN. This RBN state denoted by $\mathbf{x}_t$ is a vector of binary output values of each reservoir node at time t. This RBN state acts as the input to the next block.

- *Readout layer*:

  This block is implemented as a linear read-out layer. Each output node in the readout layer connects to all reservoir nodes, except for the nodes which receive input excitations, through weighted connections. The functionality of the readout layer block can be represented by equation 4.1:

$$\mathbf{a}(t) = f_{out}(\mathbf{W}_{out}\mathbf{x}(t)), \tag{4.1}$$

  where $W_{out} \in \Re^{p,n}$ is the output weight matrix, p is the number of output nodes in the read-out layer and n is the number of reservoir nodes (same as reservoir size N), f_{out} is the output node activation function. At each output node, a weighted sum of all incoming reservoir state signals is passed through an activation function to generate the final node output signal.

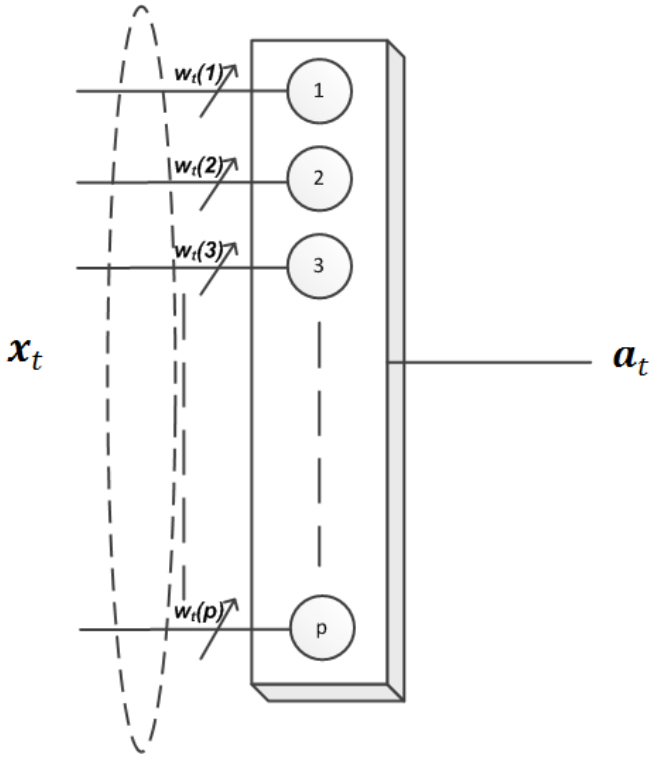


Figure 4.3: Block diagram depicting the readout layer with p output nodes. Each output node receives the weighted reservoir state, $\mathbf{x}_t$, as input and generates value estimates for each possible action, based on the activation function associated to it. The node associated to the maximum value estimate emerges as the winner. The action signal, $\mathbf{a}_t$ is the signal associated to the winning output node.

- *Environment*:

  This block models the functionality of the environment with which the autonomous agent interacts. It also highlights the mechanism of calculating and propagating error signal through a training process based on the *Temporal-Difference* (TD) approach to reinforcement learning. During the training phase a ε-greedy policy is followed to update weights, which gradually converges to a greedy policy at later training episodes.

  We choose the *Temporal-Difference* approach to training weights:

  – Since this particular reinforcement learning technique does not need a

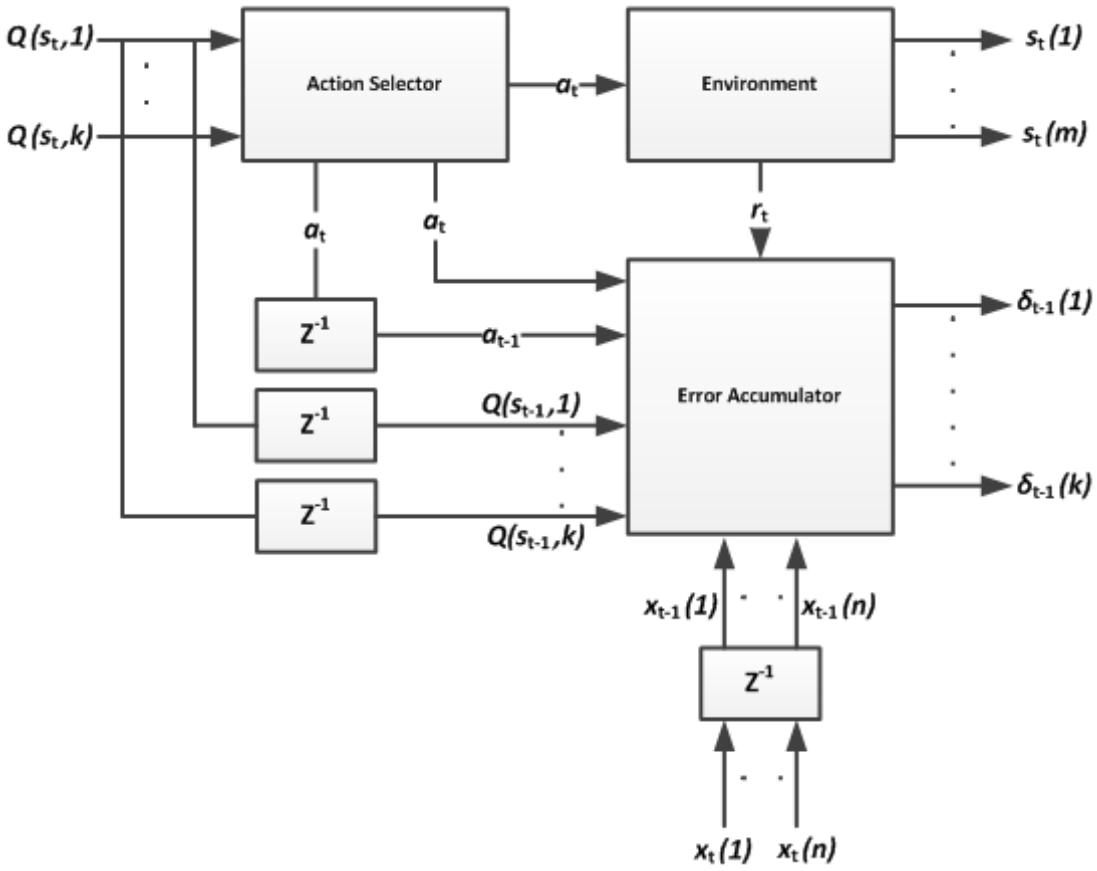


Figure 4.4: Block diagram depicting the interaction between the environment and the agent. This block receives as input the signal $Q(\mathbf{s}_t, \mathbf{a}_t)$, from the read-out layer. $Q(\mathbf{s}_t, \mathbf{a}_t)$, is a vector of action value estimates of dimension p, where p is the number of output nodes in the read-out layer. The *Action Selector* block selects the action to be performed at each timestep. This selection which is based on a ε-greedy policy during early stages of training gradually decays to a greedy approach during the later training stages. The selected action a_t is then activated in the *environment* modelled by the corresponding block. This generates a new *environment* state s_{t+1} and a *reward* signal r_t. The *Error Accumulator* uses this *reward* signal r_t, *action value estimate* a_t, reservoir state $\mathbf{x}_t$ and *action value estimates* a_{t+1} at the next timestep to generate the error signal based on which read-out layer weights are updated during training.

model of the environment or knowledge of the reward function and the next state transition function [5].

– TD approach is incremental is nature, that is, the value estimates for

the cumulative future rewards is updated at every timestep based on the environment-agent interactions [5] .

– In TD algorithm, the accuracy of value estimates is improved through online *boot-strapping* approach during training. In *boot-strapping* techniques, there is no target available to generate an error signal [5]. The error signal is generated based on Equation 4.2

$$e = r_t + \gamma Q(x_{t+1}, a_{t+1}) - Q(x_t, a_t), \tag{4.2}$$

where e = error signal, r_t = the reward, $Q(x_t, a_t)$ = Value estimates at state x_t for the action a_t performed by the agent at the current timestep t, $Q(x_{t+1}, a_{t+1})$ = Value estimates read-out from the output layer of the reservoir at reservoir state x_{t+1} for the action a_{t+1} performed by the agent during the previous timestep $t + 1$.

The value estimate for cumulative future rewards at the current timestep is treated as a sum of the reward obtained at the current timestep and the value estimate of the cumulative future rewards at the next timestep.

In our setup, we deal with *model-free environments*, and do not have knowledge of a target behavior to train the agent. We also need to train the reservoir read-out layer incrementally to achieve better estimates at each timestep to improve chances of convergence. Due to these requirements, the TD approach is most advantageous for training read-out layer weights in our case. Once trained, the reservoir estimates the cumulative future reward values accurately and picks the action which guaranteed maximum future rewards at each timestep.

The functionality of this section is explained in detail in the following algorithm:

Algorithm 4 Environment-agent interaction pseudo-code; Source: [5]

Initialize the current policy (π), to an ε-greedy policy. Initialize $f_e \in [0, 1]$ to fraction of number of episodes at which ε starts to decay. Initialize $f_\varepsilon \in [0, 1]$ to ε decay rate.

for all episodes **do**

for all timesteps in each episode **do**

Input x_t

$a_t \leftarrow$ action given by π for x_t

Take action a_t; observe reward r_t , and next state, x_{t+1}

$e \leftarrow r_{t-1} + \gamma Q(x_t,a_t)$ - $Q(x_{t-1},a_{t-1})$

$\delta_j = ef^{'}(\sum_{i=0}^{n-1} w_{ij}x_i)$, where j = 1, 2,....number of output nodes

$w_{ij,new} = w_{ij,old} + \alpha\delta x_i$, where j = 1, 2....number of output nodes and i = 1, 2....number of reservoir nodes

x_{t+1} becomes current state

end for

if (current episode / total number of episodes) $> f_e$ **then**

$\varepsilon = \varepsilon * f_\varepsilon$

end if

end for

where $\mathbf{x}_t$ = reservoir state, a_t = action selected at time t, γ = discount factor, δ_j = delta weight calculated for output node j, $f(x)$ = output node activation function, w_{ij} = weights connecting the reservoir i to output node j, α = learning rate.

4.1.1 Flowchart

In this section we go through an example grid-world problem to explain our learning algorithm as highlighted in the flowchart in Figure 4.5

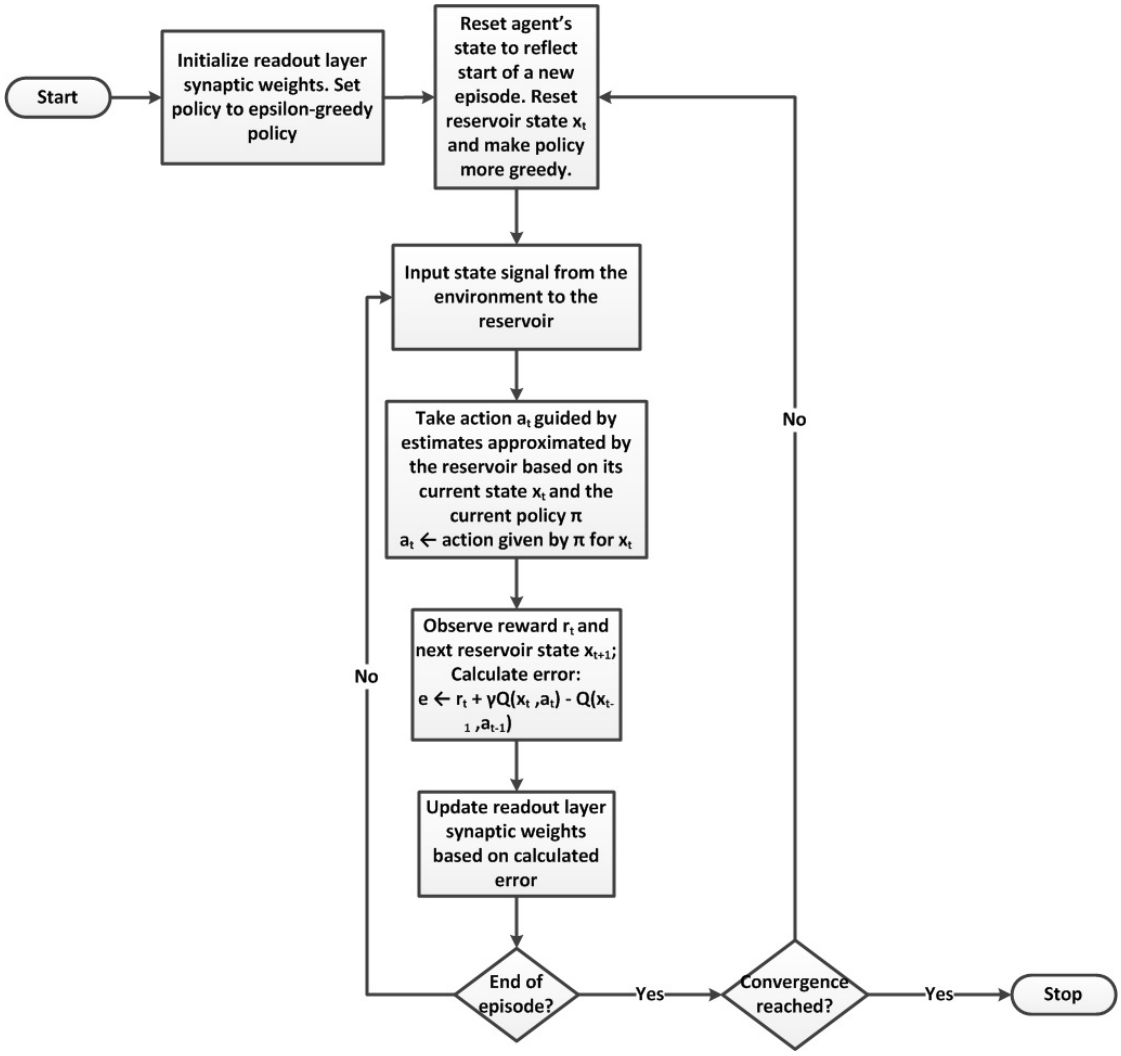


Figure 4.5: Control flow through the TD-based reinforcement learning algorithm we use in our setup.

We step through each stage of the training algorithm as highlighted in the flowchart in Figure 4.5. For our explanation, we consider a simple 5×5 grid-world with a rectilinear trail of 7 food pellets as shown in Figure 4.6. The journey of the

agent through the trail is termed an episode. We go through one training episode in this section and explain the agent-environment interaction.

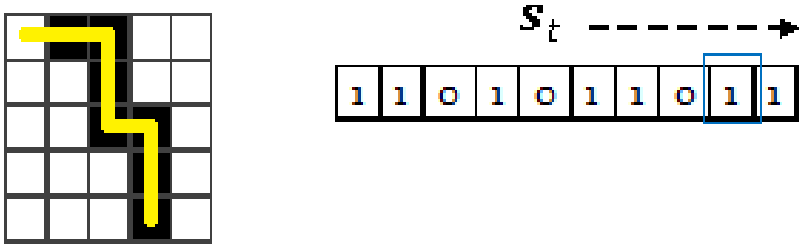


Figure 4.6: An agent traversing through a simple trail as shown encounters the input bit-stream as indicated. The sliding window indicates the serial nature of the input

At the start of the training process, an RBN-based reservoir is created based on pre-defined values of the reservoir size, N and average connectivity, K. The read-out layer weights on the connections between the reservoir nodes and the output nodes are initialized randomly based on an uniform distribution. The learning rate, discount factor and ε, which decides the policy π, are initialized. The reservoir state is reset which involves setting each node in the reservoir to a state value of zero. The agent starts at the top-left corner cell of the grid-world at the beginning of every episode and can perform three distinct actions "Move Forward", "Turn Left" and "Turn Right". The turning action involves a simple right angled turn and no forward movement of the agent into the next cell. The following steps are followed through the first step of a training episode.

- Step 1: The agent faces the cell immediately in front of it. The input signal s_1 = 1; indicating the presence of a food pellet in the cell immediately ahead of it is sensed. This input signal bit is used to perturb the RBN-based reservoir.

- Step 2: The RBN-based reservoir is allowed to perform τ internal updates to its state before the reservoir state, $\mathbf{x}_1$, is readout at the output layer.

- Step 3: The read-out layer output is then calculated and represents the value estimates for the cumulative future rewards, $Q(x_1, a_1)$, for each possible action, $a_1 \rightarrow$ ["Move Forward", "Turn Left" and "Turn Right"]. The action, a_1, to be performed on the environment is selected based on the current policy π, which may be a greedy policy or a ε-greedy policy based on the stage of training. Incase of a greedy policy, the action with the maximum value estimate is selected, while in case of ε-greedy policy, the action is selected randomly. For here, we assume that the agent moves forward.

- Step 4: This selected action is performed on the grid-world which yields a reward $r_1 = 1$.

- Step 5: A snapshot of the reservoir state, $\mathbf{x}_1$, action, a_1, value estimates of each action $Q(x_1, a_1)$ and the reward r_1 are maintained.

The agent is now in cell [1,2] The following steps are followed through the second and consecutive timesteps of a training episode.

- Step 1: The agent faces the cell immediately in front of it. The input signal s_2 = 1; indicating the presence of a food pellet in the cell immediately ahead of it is sensed. This input signal bit is used to perturb the RBN-based reservoir.

- Step 2: The RBN-based reservoir is allowed to perform τ internal updates to its state before the reservoir state, $\mathbf{x}_2$, is readout at the output layer.

- Step 3: The read-out layer output is then calculated and represents the value estimates for the cumulative future rewards, $Q(x_2, a_2)$, for each possible

action, $a_2 \rightarrow$ ["Move Forward", "Turn Left" and "Turn Right"]. The action, a_2, to be performed on the environment is selected based on the current policy π, which may be a greedy policy or a ε-greedy policy based on the stage of training. Incase of a greedy policy, the action with the maximum value estimate is selected, while in case of ε-greedy policy, the action is selected randomly. For here, we assume that the agent again moves forward.

- Step 4: This selected action is performed on the grid-world which yields a reward $r_2 = 1$.

- Step 5: The error signal is now calculated for the output node corresponding to the selected action of the previous timestep, a_1, based on Equation 4.3.

$$e = r_1 + \gamma Q(x_2, a_2) - Q(x_1, a_1), \tag{4.3}$$

  For output nodes corresponding to all other actions, the error signal is calculated based on Equation 4.4.

$$e = \gamma Q(x_2, a_2) - Q(x_1, a_1), \tag{4.4}$$

- Step 6: The weights are updated based on delta weights calculated as per Equation 4.5

$$\delta_j = ef^{'}(\sum_{i=0}^{n-1} w_{ij}x_1), \tag{4.5}$$

  where $j = 1, 2$....number of output nodes and $i = 1, 2$....number of reservoir nodes

- Step 6: A snapshot of the reservoir state, $\mathbf{x}_2$, action, a_2, value estimates of

each action $Q(x_2, a_2)$ and the reward r_2 are maintained.

- Step 7: The policy π is updated to a more greedy policy.

The above steps, from 1 to 7, are repeated for every timestep for the rest of the episode. At the start of the next episode, the same procedure explained above is repeated. This continues till convergence is reached and the agent can accurately traverse through the trail.

4.1.2 Code Implementation

In this section, we explain the technical details of the C++ code implementation of blocks designed as a part of this research work. The functionality of each of these blocks has been explained in detail in Section 4.1.

- *Readout Layer Block*:

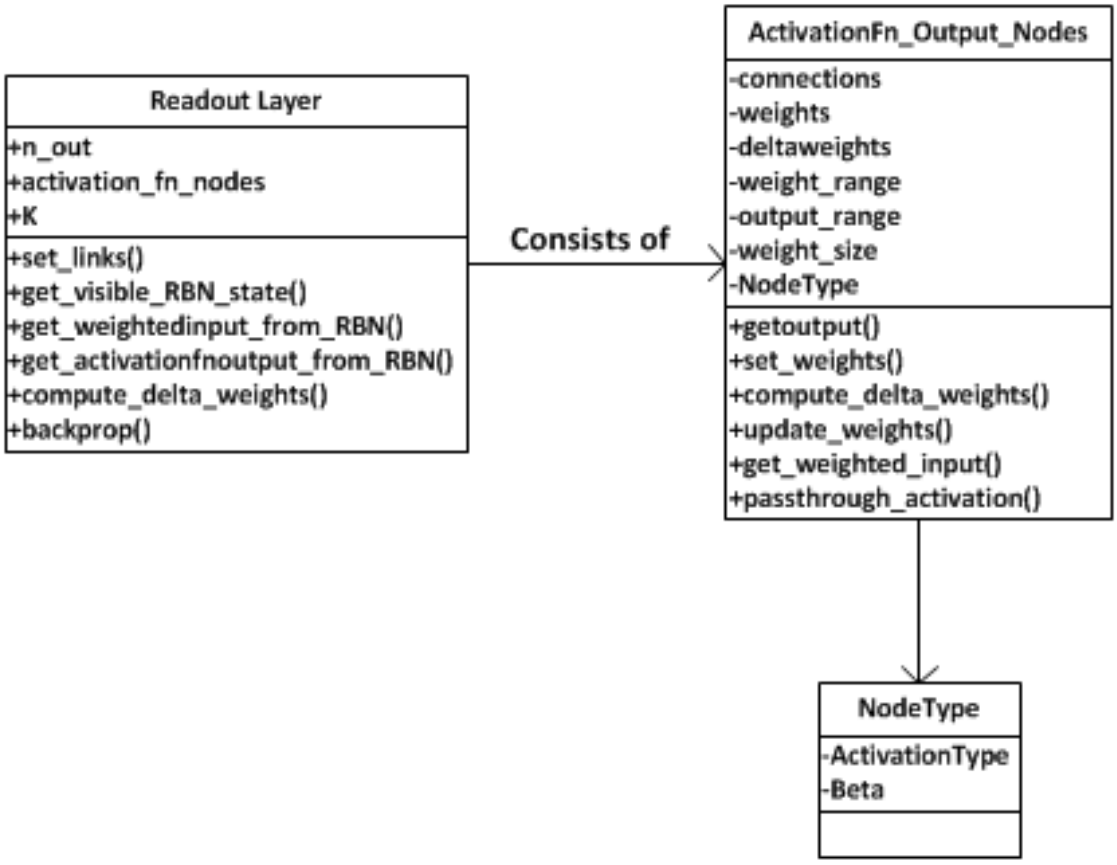


Figure 4.7: Class diagram of read-out layer implementation.

An overview of the functionality of the block, in Figure 4.7, as implemented in code is given in the following section:

– *ReadOut Layer* This class implements the functionality of the readout-layer.

Attributes

* *n_Out*: Number of output nodes in the readout layer.
* *activation_fn_nodes*: Array of objects of the class which implements functionality of individual nodes on the readout layer.
* K: Number of connections between the processing units in the RBN and each readout layer node.

Operations

* *set_links*: Weighted connections between the RBN processing nodes and each readout layer node are established. The readout layer nodes are connected to N-M nodes i.e., all nodes in the RBN except the input nodes.
* *get_visible_RBN_state*: Returns the RBN state vector, $\mathbf{x}_t$, at that timestep.
* *get_weightedinput_from_RBN*: Returns the output vector, $\mathbf{W}_{out}\mathbf{x}_t$, of weighted RBN states as it is input to the readout layer.
* *get_activationfnoutput_from_RBN*: Returns the output vector, $\mathbf{y}$, calculated by Equation 4.1.
* *compute_delta_weights*: Computes the delta weights, $\Delta\mathbf{W}_{out}$, based on the error signal at that timestep. Equation 2.5 shows how the error is calculated analytically.
* *backprop*: Adjusts the connection weights, based on the already

calculated delta weights.

$$\mathbf{W}_{out} = \mathbf{W}_{out} + \Delta\mathbf{W}_{out} \tag{4.6}$$

– *ActivationFn_Output_Nodes* This class implements the functionality of each output node in the readout layer.

Attributes

* *connections*: Array of indexes to the RBN processing nodes, this instance of the output node is connected to.
* *weights*: Array of weights connecting the node to RBN processing nodes.
* *deltaweights*: Array of deltaweights calculated mid-way through a weight training process.
* *weight_range*: The range within which the connection weights to the node can be randomly initialized before start of training. This way the point on the weight space at which the training starts can be restricted.
* *output_range*: The range within which the output from the node needs to be bound for stability.
* *weight_size*: The size of the weights array.
* *NodeType*: Object of the structure which defines the nodetype and its learning rate.

Operations

* *getoutput*: Returns the output value, $y(n)$ for activation node n in the readout layer.

* *set_weights*: Randomly initializes the connections weights from the RBN processing nodes to a specific instance of the readout layer node within the range defined by the attribute *weight_range*.
* *compute_delta_weights*: Computes the delta weights for a specific instance of the node, $\Delta \mathbf{W}_{out}(n)$, based on the error signal at that timestep. Equation 2.5 shows how the error is calculated analytically.
* *update_weights*: Updates the connection weights, based on the already calculated delta weights for a specific activation node instance.

$$\mathbf{W}_{out}(n) = \mathbf{W}_{out}(n) + \Delta \mathbf{W}_{out}(n) \tag{4.7}$$

.

* *get_weighted_input*: Returns the output vector, $\mathbf{W}_{out}\mathbf{x}_t(n)$, of weighted RBN states as it is input to the readout layer for a specific instance of the readout layer node.
* *passthrough_activation*: Has the same functionality as the *getoutput* function but with a different argument type.

– *NodeType*: This structure holds information that defines the type of node and the rate at which it learns

Attributes

* *ActivationType*: This defines the activation function that is associated with the specific readout layer node instance. The following activation functions are supported:
  · Linear: $\mathbf{y}(n) = a * (\mathbf{W}_{out}\mathbf{x}(n))$, where a is a constant.

· Hyperbolic: $\mathbf{y}(n) = tanh(\mathbf{W}_{out}\mathbf{x}(n))$.

· Sigmoid: If, $\mathbf{s}(n) = \mathbf{W}_{out}\mathbf{x}(n)$, then $\mathbf{y}(n) = \frac{1}{1+e^{s(n)}}$

∗ *Beta*: Learning rate for the specific node instance.

- *Environment Block*: The class diagram in Figure 4.8 shows an abstraction of the functionality implemented in the actual C++ code. When dealing with specific tasks we'll delve into actual implementation. The functionality describes as implemented in the C++ classes:

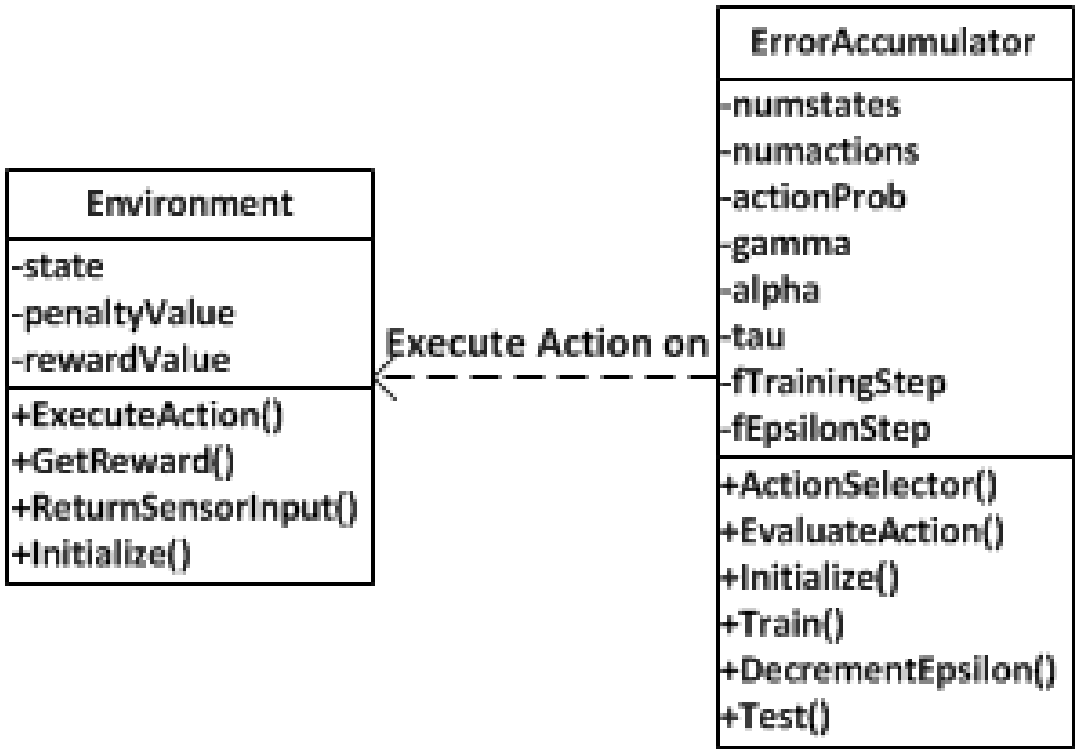


Figure 4.8: Class diagram implementation of *environment* and *agent* interaction

– *Environment*: This class is an abstract of the actual implementation of the problem context. This models the behavior of the environment.

Attributes

* *state*: Represents the current state of the environment s_t.
* *penaltyValue*: constant value of the penalty imposed on the agent for taking sub-optimal actions.
* *rewardValue*: constant value of the reward given to the agent for taking the right action.

Operations

* *ExecuteAction*: The mentioned action is performed on the environment; state is updated.
* *GetReward*: The reward/penalty due to performing an action is returned back.
* *ReturnSensorInput*: Returns the current environment state.
* *Initialize*: At the start of an agent's episode, the environment settings are initialized to a preset initial state.

– *ErrorAccumulator*: This class is an abstract of the actual implementation of the simulation framework emulating training the agent and its interaction with the environment.

Attributes

* *numstates*: Represents the dimension of the state vector.
* *numactions*: Represents the dimension of the action vector.
* *actionProb*: Holds the ε value.
* *gamma*: Holds the discount factor.
* *alpha*: Holds the learning rate.
* *tau*: A delay in the timesteps after presenting the input perturbation to the reservoir before the output is observed at the readout

layer.

* *fTrainingStep*: Holds the value of f_e.
* *fEpsilonStep*: Holds the value of f_ε.

Operations

* *ActionSelector*: Selects the action a_t based on the ε - greedy policy or the greedy policy.
* *EvaluateAction*: During the actual interaction of the agent with the environment, executes the selected action on the environment.
* *Initialize*: The state of the environment is initialized at the start of every training episode. The decayed ε value is reset as well.
* *Train*: The agent and environment interact with each other in episodic tasks and the weights are updated at each timestep so the agent learns to approximate the value estimates accurately.
* *DecrementEpsilon*: The ε value is decayed through the training to make it more greedy.
* *Test*: After the weight training, the agent's emerged behavior is tested on the environment.

4.2 Optimization of Initial Parameter Space

In reinforcement learning approaches, a gradient descent approach taken to converge to an optimal policy depends to a large extent on initial point in the policy space the training starts at. Since reinforcement learning systems are dynamic systems, as we proceed through training, at each parameter update we face bifurcation in dynamics. Based on the action taken, the convergence could slow down or in some cases convergence might end up becoming less possible [2]. The initial point in the policy space we start our training from and also the decisions made at every point and the subsequent state transitions that follow are dependant on the value with which the parameters listed in Table 4.1 are initialized.

Table 4.1: Initial parameters critical for the policy convergence to an optimal one

Parameter	Description
β	Learning rate for weight adjustments
γ	Reward discount factor, controls weight given to immediate or future rewards
N	RBN reservoir size; Number of random binary nodes in the reservoir
τ	Delay in timesteps between the point when the input bit is applied to the reservoir to the timestep at which the output is read from the readout layer
K	Reservoir connectivity; mean number of neighbouring reservoir nodes from which a node receives input connections
n	Minimum number of episodes for which the readout layer weights need to train to solve the problem
t	Minimum number of timesteps per episode to reach the goal
ε	The probability with which a non-greedy action is selected during the training phase
s	The fraction of the number of episodes at which the ε starts decaying to a greedy policy
e	The decay rate of ε towards a greedy policy

Since the search for an initial starting point in this ten dimensional parameter space is daunting, we use an evolutionary algorithm to conduct a stochastic search

in the initial parameter space to find that starting point. We use a direct search method which is similar to a genetic algorithm. Differential Evolution (DE) was developed to address optimizations effectively in a real valued parameter space [57].

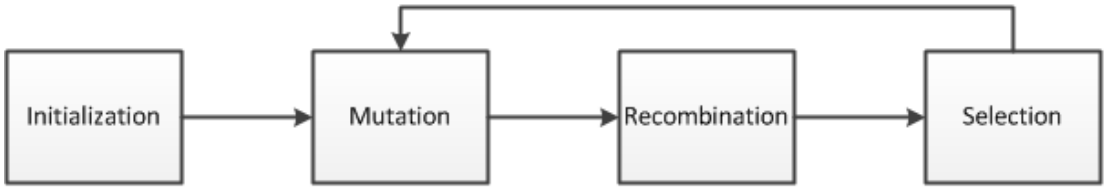


Figure 4.9: Block diagram of *Differential Evolution*; Source: [6].

Figure 4.9 gives a block diagram representation of the 4 stages in DE which is common across most evolutionary algorithms [6].

- We start with D-dimensional real parameter vector X,
  where $x_{i,G} = [x_{1,i,G}, x_{2,i,G}, x_{3,i,G}, ...x_{D,i,G}]$, i =1,2,....,NP and G = the generation number.

- Initialization

  - Define upper and lower bound for parameter values. $x_j^L \leq x_{j,i,1} \leq x_j^U$
  - Randomly select the initial parameter values uniformly in the defined interval.

- Each of the N parameter vectors undergoes mutation, recombination and selection.

- Mutation

  - Mutation expands the search space.

- For a given parameter vector $x_{i,G}$ randomly select three vectors $x_{r1,G}$, $x_{r2,G}$ and $x_{r3,G}$ such that the indices i, r1, r2 and r3 are distinct.
- Add the weighted difference of two of the vectors to the third. $v_{i,G+1} = x_{r1,G} + F(x_{r2,G} \;\; x_{r3,G})$
- The mutation factor F is a constant from [0, 2].
- $v_{i,G+1}$ is called the donor vector.

- Recombination
  - Recombination incorporates successful solutions from the previous generation.
  - The trial vector $u_{i,G+1}$ is developed from the elements of the target vector, $x_{i,G}$, and the elements of the donor vector, $v_{i,G+1}$.
  - Elements of the donor vector enter the trial vector with probability CR.

$$u_{j,i,G+1} = \begin{cases} v_{j,i,G+1} & \text{if } rand_{j,i} \leq CR \qquad and \qquad j = I_{rand} \\ x_{j,i,G+1} & \text{if } rand_{j,i} > CR \qquad and \qquad j \neq I_{rand} \end{cases}$$

    i = 1, 2, ...,N j = 1, 2, ...,D

  - $rand_{j,i} \sim U[0,1], I_{rand}$ is a random integer from [1, 2, ...,D]
  - I_{rand} ensures that $v_{i,G+1} \neq x_{i,G}$

- Selection
  - The target vector $x_{i,G}$ is compared with the trial vector $v_{i,G+1}$ and the

one with the lowest function value is admitted to the next generation.

$$x_{i,G+1} = \begin{cases} u_{i,G+1} & \text{if } f(u_{i,G+1}) \leq f(x_{i,G}) \\ x_{i,G} & \text{otherwise} \end{cases}$$

- Mutation, recombination and selection continue until some stopping criterion is reached.

The individual that evolves from the DE is used to initialize the initial parameter set of the agent and this agent is taken through the reinforcement learning process towards achieving the set goal. The individual listed in Table 5.7 was evolved through the DE and is used through the Section 5.3.2 for our discussions.

4.3 Comparison with other approaches

We train an RBN-based function approximator to estimate the value of the future rewards for a state-action pair. To compare our approach to a standard benchmark, we select a Q-Learning approach. In Q-Learning the Q-Values in a canonical look-up table of dimension $S \times A$ go through updates based on the algorithm highlighted in Chapter 2.4.4, where S is the finite state space for the problem context and A is the action space for a particular state. The Q-Learning process involves sampling each state-action pair during the training process to update the Q-Values. For our experiments, Q-Learning was implemented in a C++ simulation framework.

5

Experiments

We setup our framework in the following problem contexts:

- Temporal parity task in a supervised learning context.
- Bandit task in a non-temporal reinforcement learning context.
- Tracker task in a temporal reinforcement learning context.

We test the accuracy and performance of the RBN augmented with the linear readout-layer when setup in the supervised learning scenario solving the temporal parity task. The weights are trained to learn the non-linear input-output mapping. This task evaluates the ability of the reservoir to retain history of the past inputs while learning to map the high dimensional reservoir state to the output space through a non-linear function adaptation of the weights. We then setup the RBN in an n-armed bandit task. This is to test the RBN setup in the reinforcement learning framework augmented with the linear readout-layer and the environment which involves n stochastic processes to choose from. These processes yield rewards based on an their associated stationary probability distributions. The ability of the learner to choose processes at each step to yield rewards is based on its ability to learn the stationary internal probability distribution that governs the processes. Due to lack of their ability to balance the exploitation and exploration phases of the training it is believed that supervised learning techniques cannot perform well on this task [5]. We then customize our RBN setup to a tracker problem or an ant-trail problem. An ant-trail problem involves setting up an agent in an environment

where the environmental features are partially observable by the agent. A tracker problem needs internal memory of the past inputs so that the agent can perform a motor action in a present perceived situation. The agent essentially needs to perform different actions for the same partially observable current input state [58]. We train the output-layer weights through a reinforcement learning technique based on *Temporal-Difference*. The solution to a tracker problem involves training the agent for an optimal behavior while finding the temporal embedding needed to solve it in the non-Markovian environment considered. This search for an optimal strategy makes it a reinforcement learning problem and so it isn't an ideal candidate for supervised learning approaches. Setting up the tracker task in a supervised learning framework would need a *priori* knowledge of the length of the temporal input state embedding and a perceived optimal strategy.

5.1 Supervised Learning with Temporal Parity Task

The temporal parity task was considered to evaluate our setup in a supervised learning framework, before delving into more complex reinforcement learning tasks. The temporal parity task is essentially the parity/XOR function evaluated on a binary input bit-stream across time. The parity task was chosen because of its non-linear separability in classifying the output bits for an input bit combination. In a traditional ANN setup, an XOR function cannot be solved by a single layer perceptron. The non-linear mapping of the input to output spaces, made it essential to introduce another hidden layer to learn the non-linearity in the I-O mappings. We choose temporal parity function to assess the performance and ability of the RBN augmented with the read-out layer to learn a simple but non-linear temporal function through a supervised learning approach.

5.1.1 Problem Description - Temporal Parity Task

Temporal Parity task is a standard XOR function performed on bits across a varying lengths of an input binary bit-stream. We consider bit streams of length T and consider parity across a fixed sliding window size $1 \leq n \leq T$. A delay τ determines the timesteps allowed to elapse between the bit-excitation and the output read. Table 5.1 shows the truth table for a 3-bit odd-parity task. The odd-parity task can also be mathematically represented as follows [15]:

$$PAR_n(t) = \begin{cases} u_{t-\tau} & \text{if } n = 1 \\ \bigoplus_{i=0}^{n-1} u_{t-\tau-i} & \text{otherwise} \end{cases}$$

Table 5.1: Truth table for a 3 input odd-parity task

I_1	I_2	I_3	O
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

An RBN-based system with the settings as mentioned in Table 5.2 is excited by an input bit-stream of mentioned dimension and length in time. The window size across which the odd-parity is evaluated is kept fixed across all the experiments.

5.1.2 Results and Discussion - Temporal Parity Task

A set of 100 RBN-based random reservoirs were evaluated with the settings as mentioned in Table 5.2. It can be seen from Figure 5.1, which plots the average

Table 5.2: RBN setup for the temporal odd-parity task

Parameter	Description	Value
N	Liquid Size	500
τ	Delay in timesteps between the point when the input bit is applied to the reservoir to the timestep at which the output is read from the readout layer	0
n	The window in time across which the odd-parity is evaluated	3
K	Connectivity	2
β	Learning rate	0.03
T	Training sample length in time	[8, 10, 12]
N_s^{train}	Number of training samples considered	250
N_s^{test}	Number of test samples considered	30
L	Number of nodes in the reservoir receiving input excitation	100
N_i	Number of training iterations	8000

performance of the three best performing random reservoirs, that the RBN memorizes the pattern underlying the I-O mappings in the training set very quickly, for the varying lengths of the bit-streams considered, that is in less than 200 training steps. The best individual that trained with the RBN settings in Table 5.2 learned to generalize the test set with a 100% generalization accuracy for input bit-streams of length $T = 10$ and $T = 12$, while learned to generalize with 80% accuracy for input bit-streams of length $T = 8$. This could be because we initially optimized our framework for input bit-streams of length $T = 10$ and then extended it to generalize for other bit-stream lengths.

With this we validated the accuracy of our RBN-readout layer setup in solving supervised learning tasks with known input-output mappings. We proceed further to analyze the performance of the reservoir when setup in a reinforcement learning scenario.

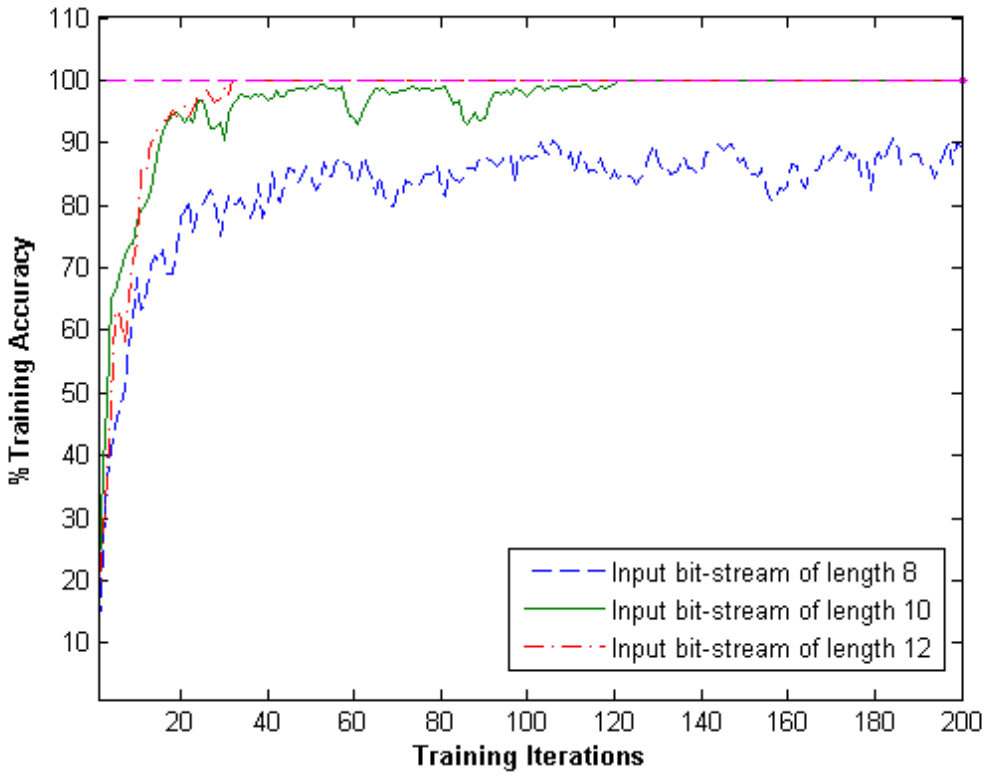


Figure 5.1: Convergence of an RBN-based reservoir, performing a temporal odd-parity on bit-streams of varying length, in terms of improvement in its training accuracy with the progression of training as averaged across 3 best random reservoirs

5.2 Non Temporal Reinforcement Learning with n-Arm Bandit Task

We adopt the n-arm bandit problem based from Michael O.Duff's work [7] in a reinforcement learning setup to observe the RBN's ability to perform in such problem contexts. The n-arm bandit problem is based on *Thompson Sampling* [59] which is based on the idea of selecting an arm or process, often compared to pulling a lever of a slot machine, with each arm being associated with unknown probabilities of yielding rewards. If these probabilities are a known *priori* then solving the n-armed bandit problem would be trivial.

Gittins and Jones [60] and Bellman [38] have popularly adopted Bayesian formulation to solve this problem, while Robbins [61] adopted strategies where asymptotic *loss* tends to zero. In each of these approaches, selecting an arm yields an immediate reward and also an associated *Bayes*-rule update of the arm's reward probability. During the initial stages of learning the estimated reward probabilities of these arms are based either on *prior* knowledge or random initialization. All the n-arms need to be sampled out at similar frequencies to improve these reward probability estimates associated with them. A *greedy* approach to selecting the arms based on initial estimates would result in inaccurate learning of the inherent probabilities since the arms with higher initial reward estimates would get chosen all the time. An *exploration* vs *exploitation* approach that is essential to solve this problem is of particular interest since it brings forth the familiar human dilemma of sacrificing immediate rewards for information gain to improve the quality of future decisions [7].

In our setup, the RBN-based agent is trained through interaction with these stochastic reward yielding processes to accurately estimate the reward probabilities associated with each process/arm. The training is based on the observed *rewards* at each stage and not based on *target optimal action* at each timestep as provided by a *teacher*. Since the goal of this experiment is to learn the unknown reward yielding probability associated with each arm and not the inherent input-output mapping, setting it up in a *teacher-learner* context, where it learns to perform one action as instructed by a teacher at each timestep, defeats the purpose.

5.2.1 Problem Description - n-Arm Bandit Task

In this section, we explain our setup of the n-arm bandit problem.

A single bandit problem consists of n-independent stochastic processes. At each timestep, a controller activates a process, which in-turn yields a certain reward along with an updated state for the selected process. The other unselected processes retain their states from the previous timestep through the next timestep. The state transitions and associated rewards are based on an internal probability distribution, which is unique for each process and is unknown to the controller. The goal of this setup is to maximize the cumulative rewards gained over a finite time horizon. Note that selecting the process which yields maximum reward at every timestep may not be the optimal strategy in achieving the mentioned objective.

For discussions, we consider the mathematical model of the 2-arm bandit problem: consider two stochastic processes represented by states $x_1(t)$ and $x_2(t)$ respectively, which take on values from a countable set $\in$ [0, 1]. Therefore, the system state at any time instance can take on one of the four possible values [00, 01, 10, 11]. At each timestep the controller generates an action signal which essentially indicates which stochastic process needs to be activated. This can be represented as $a_t \in 1, 2$. If the controller generates an action signal of 1 (which in other words means $process1$ is chosen to be activated), the effect of this action signal on future system states is given by Equation 5.1

$$x_1(t+1) = f_1(x_1(t), p_1(t)), \tag{5.1}$$

where $p_1(t)$ is a probability distribution which decides the state transitions for $process1$. Note that the immediate future state depends only on the current state and is independent of any other prior states.

The state corresponding to the process which was not selected by the controller

remains frozen as shown in Equation 5.2

$$x_2(t+1) = x_2(t) \tag{5.2}$$

The functions which decide the state transition probability and the associated reward for the processes involved are represented by the state transition diagram in Figure 5.2 and 5.3. The arcs are labelled with the transition probability/associated reward.

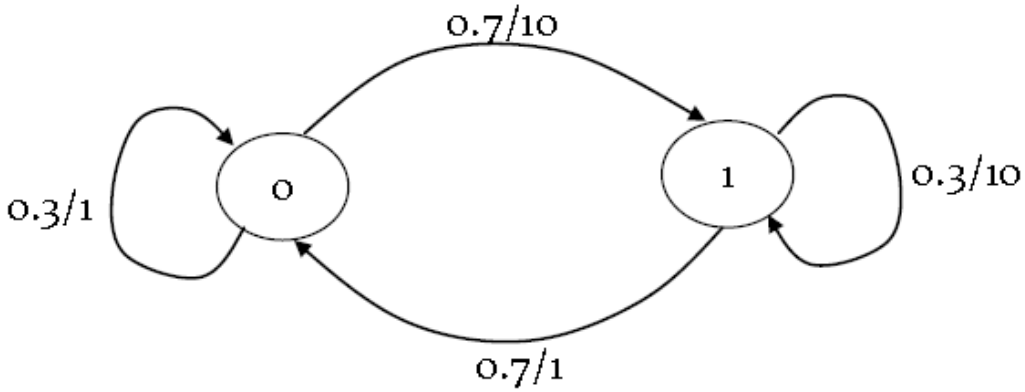


Figure 5.2: State transition diagram indicating the transition probabilities and associated rewards for $Process1$ [7]. In this case, the process changes state $x_1(t)$ from zero to one and vice-versa with a probability of $p_1(t) = 0.7$. The reward associated with this state transition is 1. With a probability of $1 - p_1(t)$, the process retains the same state which in-turn yields an associated reward of 10.

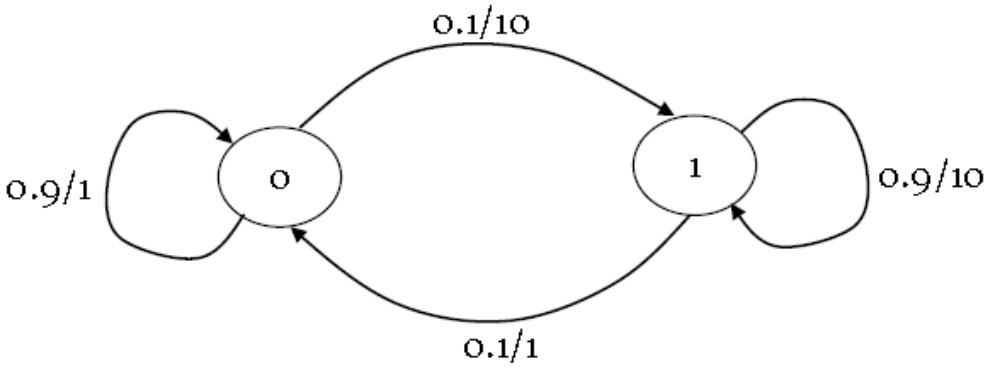


Figure 5.3: State transition diagram indicating the transition probabilities and associated rewards for *Process*2 [7]. In this case, the process changes state $x_2(t)$ from zero to one and vice-versa with a probability of $p_2(t) = 0.1$. The reward associated with this state transition is 1. With a probability of $1 - p_2(t)$, the process retains the same state which in-turn yields an associated reward of 10.

Note that the controller generating the action signal at each timestep is unaware of the state transition probabilities and associated rewards. Considering a finite time horizon, the objective is to maximize the cumulative rewards over a stipulated period. The factor in Equation 5.3 needs to be maximized.

$$\sum_{i=1}^{\infty} R_{a(t)}(x_{a(t)}(t)), \tag{5.3}$$

where $R_{a(t)}(x_{a(t)}(t))$ is the function in Figure 5.2 and 5.3, which decides the underlying probability of the state transition and the reward associated with such an update. The action $a(t)$ represents the process selected. The state of the activated process transitions over to $x_{a(t)}(t)$.

For our purpose, the RBN-based reservoir is setup to solve n-arm bandit tasks, with $n \rightarrow [2, 4]$. The success of the RBN-based reservoir is based on the value of the initial parameters listed in Table 5.3. Owing to the size of the parameter space, we use *Differential Evolution* (DE) to optimize the initial parameters of the

RBN-based setup for each of the n-arm bandit task.

Table 5.3: Initial parameters list for n-arm bandit task

Parameter	Value Type	Range	Description
β	Floating point	[0,1]	Learning rate for the weight updates
γ	Floating point	[0,1]	Reward discount factor, controls weight given to immediate or future rewards
N	Integer	Based on task complexity	RBN reservoir size; Number of random binary nodes in the reservoir
τ	Integer	[0,3]	Delay in timesteps between the point when the input bit is applied to the reservoir to the timestep at which the output is read from the readout layer
K	Floating point	[0,6]	Reservoir connectivity; mean number of neighbouring reservoir nodes from which a node receives input connections

Continued on next page

Table 5.3 – *Continued from previous page*

Parameter	Value Type	Range	Description
n_e	Integer	Based on task complexity	Minimum number of episodes for which the read-out layer weights need to train to solve the problem
ε	Floating point	[0,1]	The probability with which a non-greedy action is selected during the training phase
s	Floating point	[0,1]	The fraction of the number of episodes at which the ε starts decaying to a more greedy approach
e	Floating point	[0,1]	The ε decay rate towards a greedy approach

We evolve the DE by evaluating the fitness shown in the Equation 5.4. The fitness function is selected such that the cumulative rewards across a stipulated number of timesteps in an episode of the set n-arm bandit problem is maximized.

$$\mathbf{f} = (\frac{n_{actual}}{n^U}) * 1000, \tag{5.4}$$

where n_{actual} = cumulative rewards collected by the trained agent represented by the individual, n^U = possible total rewards in an episode.

For setting the value of n^U, we assume that it is possible for the agent to gain rewards at every timestep of an episode. We calculate a value based on the number of timesteps in an episode for which the trained agent is allowed to play during the generalization phase, that is t_{test}. We set different episode lengths for the training, t_{train} and testing/generalization phases, t_{test}, with the generalization phase spanning twice as many timesteps as the training phase. Table 5.4 lists the actual value of the episode lengths, along with value of other initial parameters, we used in our experiments. We chose these episode lengths based on trial and error as a trade-off between optimizing the speed at which each individual's fitness is evaluated and the number of timesteps needed for an individual to accurately learn the inherent reward probability. During the fitness calculation, we linearize the impact that the reward at every timestep has on the fitness by normalizing the cumulative rewards collected at each timestep by the trained agent. We scale up this value to enhance the impact of each reward on the fitness of the individual.

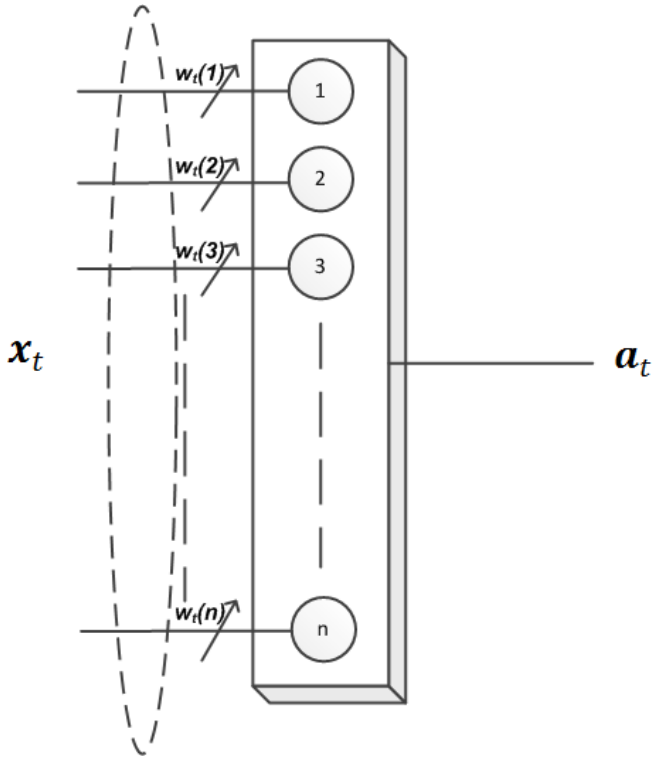


Figure 5.4: The readout-layer for the n-arm bandit task. Each node represents one of the n arms/processes and outputs reward estimates for selecting that action.

Figure 5.4 represents the read-out layer setup for the n-arm bandit task. The weighted connections from the reservoir nodes to the readout layer nodes are trained to estimate the reward for each process, with the current state of the processes provided as the input to the reservoir. The readout layer nodes use linear activation functions. Once trained, a greedy policy selects the process with the maximum estimated reward as the action signal. The selected process is then activated in the simulation model of the n-arm bandit problem and the actual reward from the model is observed. The actual rewards we use for this experiment are listed in Table 5.5. We use the same rewards for all the n-arms/processes considered, but with different probabilities of achieving it. This reward is compared with the estimations of the RBN-based agent for the selected process and an error

signal is generated. The error signal is then used to adjust the weights on the connections to the readout layer nodes through gradient descent. The weights are updated based on Equation 5.5 and 5.6

$$\Delta W = \beta(r_i - r_i^*)x_{ij} \tag{5.5}$$

$$W = W + \Delta W, \tag{5.6}$$

where β = training rate x_j = output value on the j^{th} reservoir node i = readout layer node being trained, where $i = [1, 2....., n]$, n being the total number of arms/processes being considered.

As mentioned earlier, for the RBN-based agent to learn the internal reward probabilities associated to individual processes, it needs to uniformly sample out and try each process to observe the associated rewards. The weights are then updated based on the observed rewards. Hence, during the training phase, to have the readout layer learn the model's intrinsic state transition probabilities and the associated reward mappings the processes or arms selected to be activated is based on an ε-greedy approach. In other words, during training, the activated process is not always the *greedy*-process which yields the maximum estimated reward. This helps to explore the input state-space during the training phase for uniform training of weights associated with each node, and hence a process, in the readout layer. During the training process, the weights are expected to encode the underlying reward probability of each process, $R_{a(t)}(x_{a(t)}(t))$. During the training phase, we gradually modify the value of ε, such that, as time, $t \longrightarrow \infty$, ε-greedy $\longrightarrow$ greedy policy. We achieve this by exponentially decaying the ε value based on the Equation 5.7.

$$
\begin{aligned}
If(i/n_e &> s) \\
\varepsilon &= \varepsilon * e, \qquad (5.7)
\end{aligned}
$$

where i = the current training episode, n_e = total number of episodes for which the readout layer weights are trained, s = fraction of the number of episodes at which the ε starts decaying to a greedy approach, e = ε decay rate towards a greedy approach.

Table 5.4: The initial parameter settings considered for the n-arm bandit task

n	$p_n(t)$	β	γ	N	τ	K	n_e	t_{train}	t_{test}	ε	s	e
2	{0.1, 0.6}	0.47	0.91	174	1	4	1025	100	200	0.67	0.99	0.63
3	{0.1, 0.6, 0.2}	0.84	0.25	117	1	3	3725	100	200	0.18	0.29	0.21
4	{0.1, 0.6, 0.2, 0.7}	0.51	0.69	155	1	5	5450	100	200	0.78	0.08	0.14

Table 5.5: The actual reward values associated to the state transition probabilities used in our experiments

State Transition Probability	Description	Associated Reward
$p_n(t)$	Probability with the process state changes	0
$1 - p_n(t)$	Probability with the process state remains unchanged	1

5.2.2 Results and Discussion - n-Arm Bandit Task

We performed experiments based on parameters listed in Table 5.4. We ran experiments on a 2-arm, 3-arm and 4-arm bandit task by initializing random reservoirs based on the initial parameters listed in the Table 5.4. We plot the results of performances averaged across 10 best performing reservoirs. In Michael O.Duff's work [7] the reward probabilities associated with each process is encoded in the Q-Values for the state-action space. In our case, the reward probabilities are encoded in the read-out layer weights. We observe that the process which has the highest reward with the highest probability is the one which gets picked maximum number of times and emerges as the optimal action at the end of the training phase.

Figures 5.6, 5.6 and 5.7 show the weight values as plotted at the beginning and the end of the respective training processes. It can be seen that at the end of the training process the weight values are highly concentrated at the read-out layer node which represents the process with the highest reward/state transition probability combination. This indicates that through sampling the processes uniformly through the training phase, the agent learned the internal reward probabilities associated with each process.

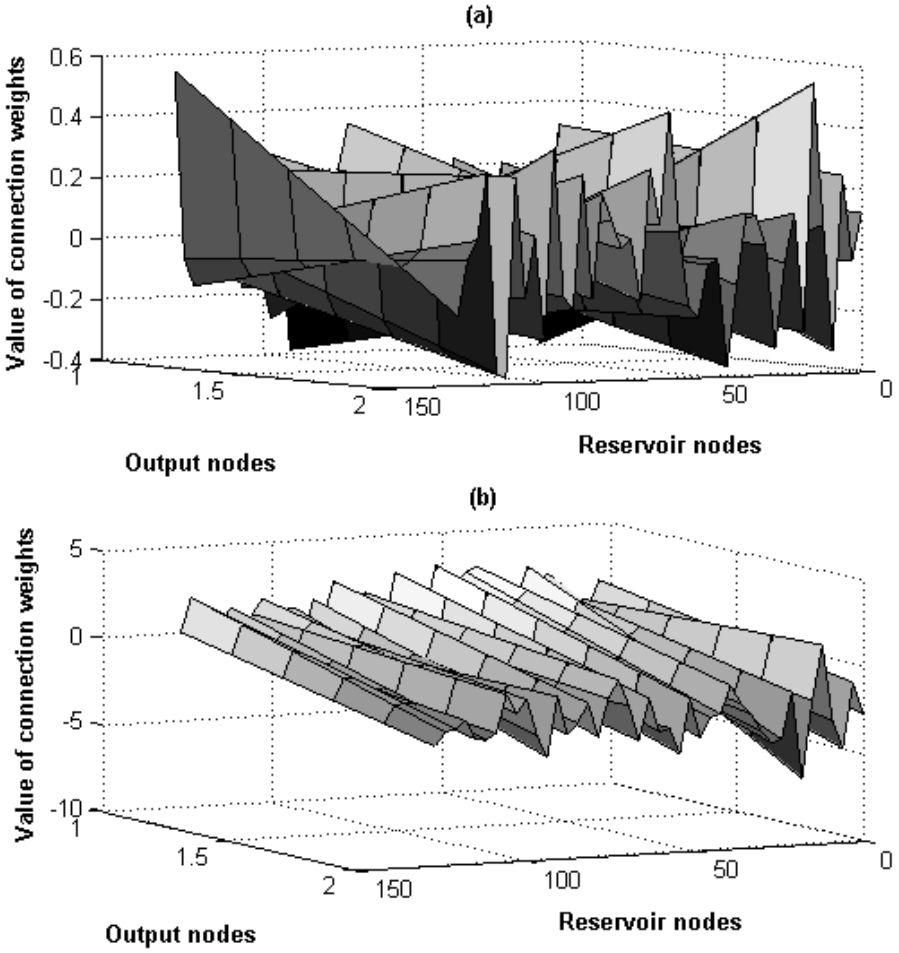


Figure 5.5: The evolution of the read-out layer weights through the training process for the 2-arm bandit task. The value of weights connecting the reservoir nodes to the read-out layer nodes are plotted. Figure (a) shows the read-out layer weights before training, just after initialization. Figure (b) shows the read-out layer weights at the end of training. It can be seen that the weights are re-arranged at the end of the training process and are concentrated towards the *process*1, which is associated with the highest reward with the highest probability that is, state transition probability = 0.9/ reward = 1.

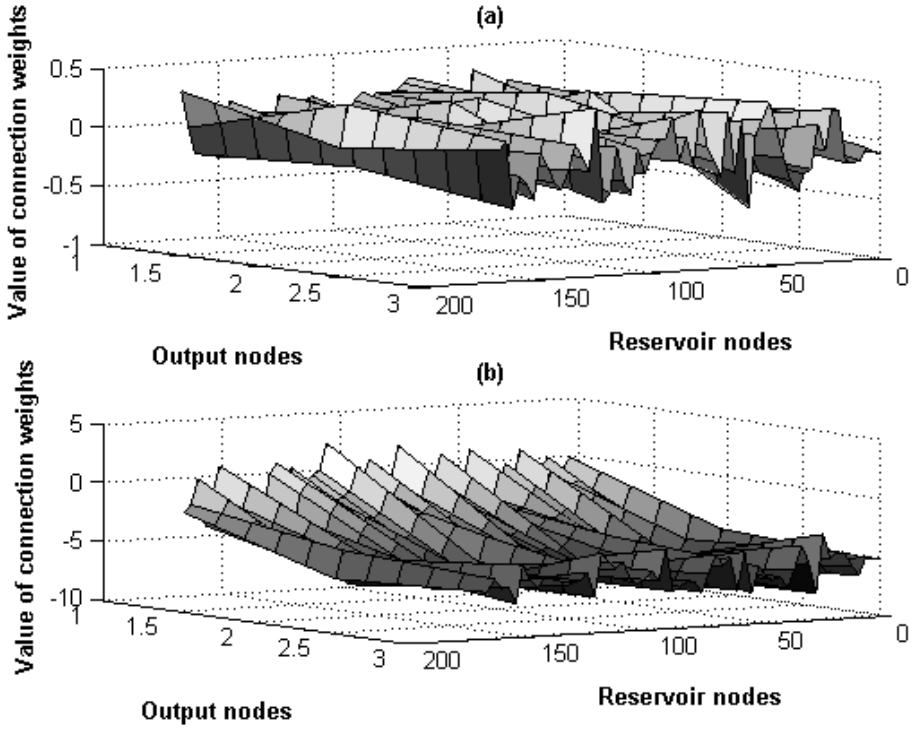


Figure 5.6: The evolution of the read-out layer weights through the training process for the 3-arm bandit task. The value of weights connecting the reservoir nodes to the read-out layer nodes are plotted. Figure (a) shows the read-out layer weights before training, just after initialization. Figure (b) shows the read-out layer weights at the end of training. It can be seen that the weights are re-arranged at the end of the training process and are concentrated towards the *process*1, which is associated with the highest reward with the highest probability that is, state transition probability = 0.9/ reward = 1.

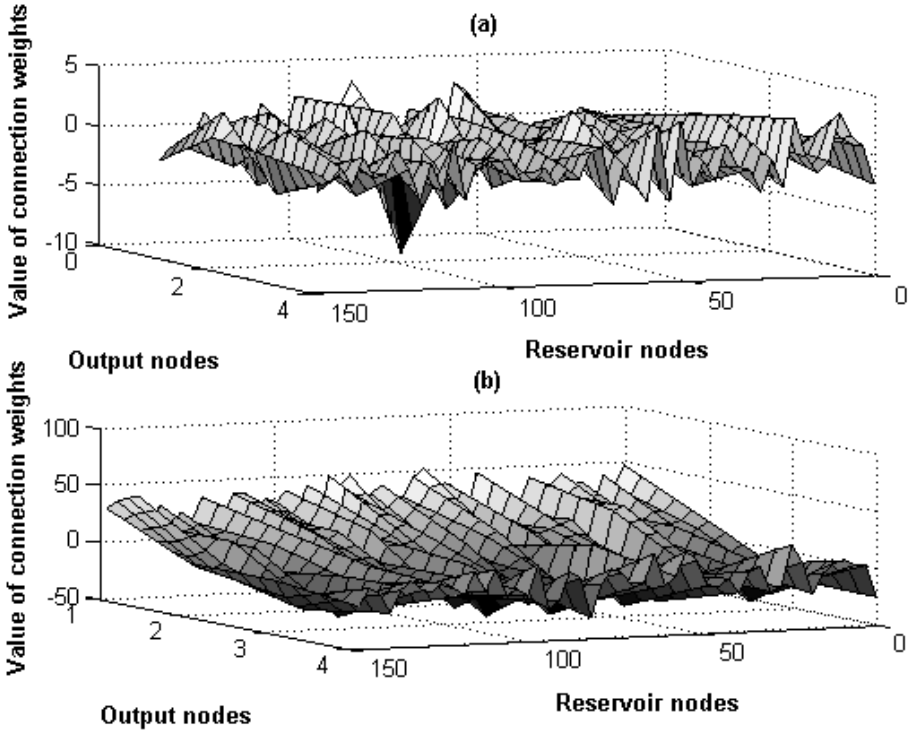


Figure 5.7: The evolution of the read-out layer weights through the training process for the 4-arm bandit task. The value of weights connecting the reservoir nodes to the read-out layer nodes are plotted. Figure (a) shows the read-out layer weights before training, just after initialization. Figure (b) shows the read-out layer weights at the end of training. It can be seen that the weights are re-arranged at the end of the training process and are concentrated towards the *process*1, which is associated with the highest reward with the highest probability that is, state transition probability = 0.9/ reward = 1.

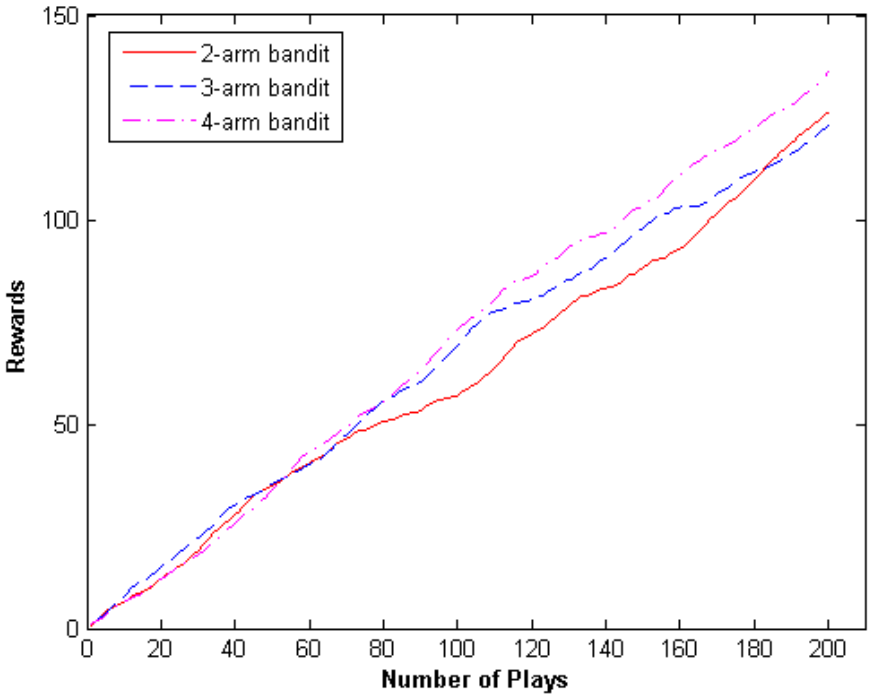


Figure 5.8: The rewards accumulated through the generalization phase as plotted for a trained 2-arm, 3-arm and 4-arm bandit process. Each training episode spans 100 timesteps while each generalization episode spans 200 timesteps. All trained bandit processes show consistent reward gains indicating selection of a process associated with high reward probabilities at each timestep of the generalization phase.

During the generalization phase the agent is run for episodic lengths twice the length of training episodes. The results of the generalization phase are averaged across 4 episodic runs. At the beginning of each generalization episode, each of n-arm bandit processes is initialized to a random state. This is done to eliminate any specific high or low cumulative reward bias that might be associated to particular initial states. It is seen that irrespective of the immediate process states, the agent chooses the process which is associated with the highest reward probability. This is to increase its chances of accumulating maximum cumulative rewards at each timestep of the generalization episodes. The agent ignored processes which

guarantee immediate rewards with higher probability but which are not profitable over the long run.

We conclude that the RBN-based agent setup in a n-arm bandit problem context successfully incorporated the inherent reward probabilities associated to the n-reward processes. We experimented with varying number of arms, and observed that in each setup the agent learns to accurately incorporate in its weights the reward probability associated to each arm. We test our method of decaying ε to balance between exploitation and exploration during the training phase. The aspect of following a ε-greedy policy is of particular importance to this problem, since learning the reward probabilities is tightly coupled to uniform sampling of all processes/arms during the training process. It is observed that the agent develops in concurrence with our expectation as stated in Section 5.2.1 and also with the results obtained in [7] with a Q-Learning agent.

5.3 Temporal Reinforcement Learning with Tracker Task

The tracker task or a *grid world* problem is popularly used to train autonomous controllers in model-free environments with partial knowledge of the immediate surroundings or state. The tracker task, is an ant-trail problem which involves training an *agent/controller* to collect food pellets by following a trail in a simulated *grid world*. This problem is designed to mimic the behavior of an ant foraging in its environment, where the *agent* or the *controller* is the *learner*, trained for this behavior.

This problem is of particular interest to researchers because of the internal memory that the *agent* needs to solve it. The *agent* is faced with the situation has only partially observed information about its immediate environment and is faced with a *perceptual aliasing problem*. That is, it needs to make different decisions for the same situation it is presented with at different points in time [62]. Such an environment is *non-Markovian*, since making optimal decisions at each situation requires a knowledge of its past situations and in some cases its past decisions. This problem presents a *Partially Observable Markovian Decision Process* (POMDP) to be learned and solved. The length of the memory embedding needed to solve POMDPs is a challenge since it is not *a known priori*.

The previous approaches to solving this problem have involved adopting evolutionary and reinforcement learning techniques towards designing an optimal state controllers. Jefferson et al. studied these POMDPs by evolving optimal state controllers through genetic algorithms [9]. Koza et al. tried to take a genetic programming approach towards solving this problem through a control program coded in a LISP-PROGRAM. The grid-world used by Koza et al. was the Santa Fe trail, Figure 5.9 [8].

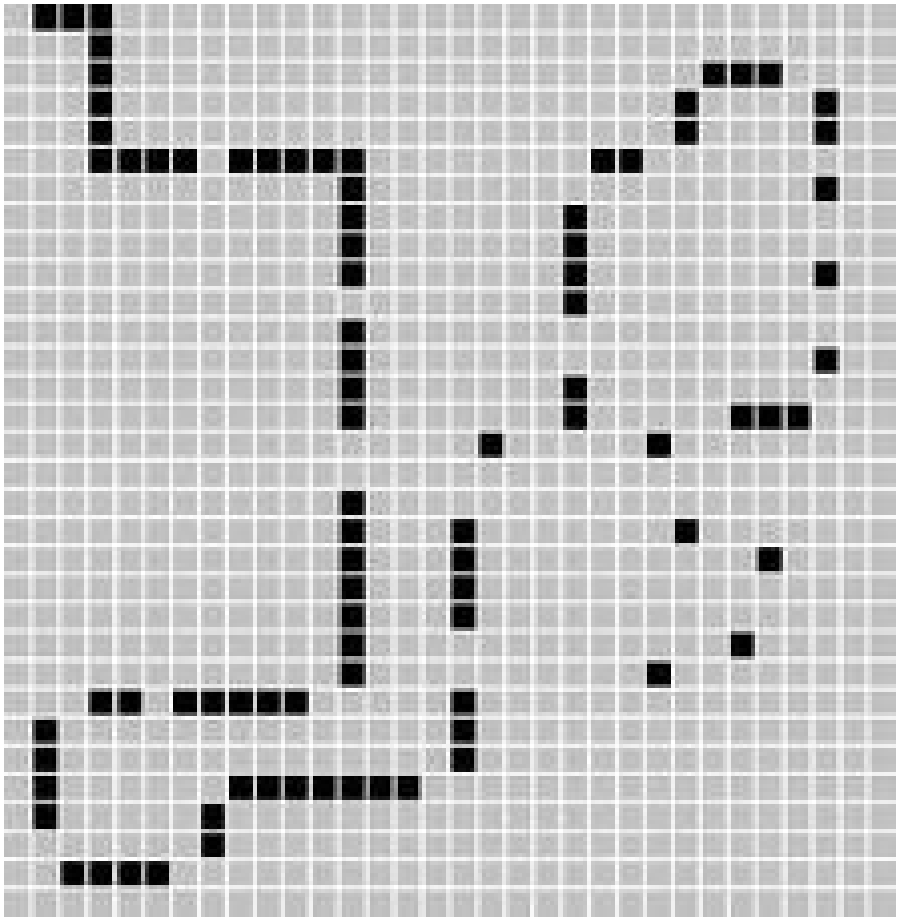


Figure 5.9: Santa Fe trail; Source: [8].

Lanzi et al. setup agents faced with *perceptual aliasing* in reinforcement learning scenarios to achieve optimal solutions. A Q-Learning technique with internal memory encoding is setup to solve maze problems [63]. Bakker and De Jong trained agents implemented as Elman Networks [20] to solve maze problems. The dynamics of the the recurrent connections is used to encode the history of the states of the Finite-State-Machine that represents the agent and its behavior [64]. Bakker and De Jong also adopted a method to extract the implicit Finite-State-Machine from the dynamic system they implemented. This technique originally experimented with by Giles et al. [65], Crutchfield et al. [66] and Kolen and Pollack et al., [67] in

the context of complexity in dynamic systems, functions on the premise that the number of states in the FSM that can be extracted from a dynamic system is a measure of its complexity.

We train our RBN-based setup to function as a food foraging agent in a grid-world scenario. We leverage the dynamics of the RBN-based reservoir to encode the memory embedding in the states needed to solve agents in POMDPs. We train the read-out layer weights through a reinforcement learning approach through continuous interactions with the grid-world environment to map the memory encoded state of the RBN-based reservoir to the actions to be performed at each state. The agents trained to solve grid-world problem are essentially Finite-State-Machine based irrespective of the technique used to solve them. Training such systems using supervised learning techniques needs prior knowledge of the FSM that fully solves the problem. In our setup, we treat the environment as unknown to start with and any information that the agent needs to solve the problem is obtained through the agent's interaction with the environment through partially observed states and reward feedbacks during each timestep of the training process.

5.3.1 Problem Description - Tracker Task

The tracker task considered resembles an ant-trail following problem in a simulated environment [9]. The agent starts at a predefined starting point, for example, at the top-left cell in a rectilinear grid world, with the objective of following a long, winding, broken trail of food pellets and maximizing its chances of collecting them in a stipulated time period through its journey. The agent as shown in Figure 5.10, at any point in time, receives binary sensor input representing whether or not the cell immediately in front of it has a food pellet or not, with one representing presence

of a food pellet (represented by shaded cell in Figure 5.10) and zero representing absence of the food pellet (represented by any un-shaded cell in Figure 5.10). The agent can perform simplified motor functions to 'move forward', 'turn left', or 'turn right'. The turning functionality only involves a right angled directional change and does not involve traversing to the adjacent cell.

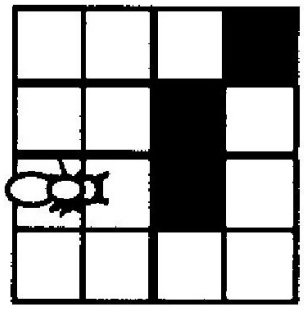


Figure 5.10: Agent/Ant facing the cell immediately ahead of it; Source: [9].

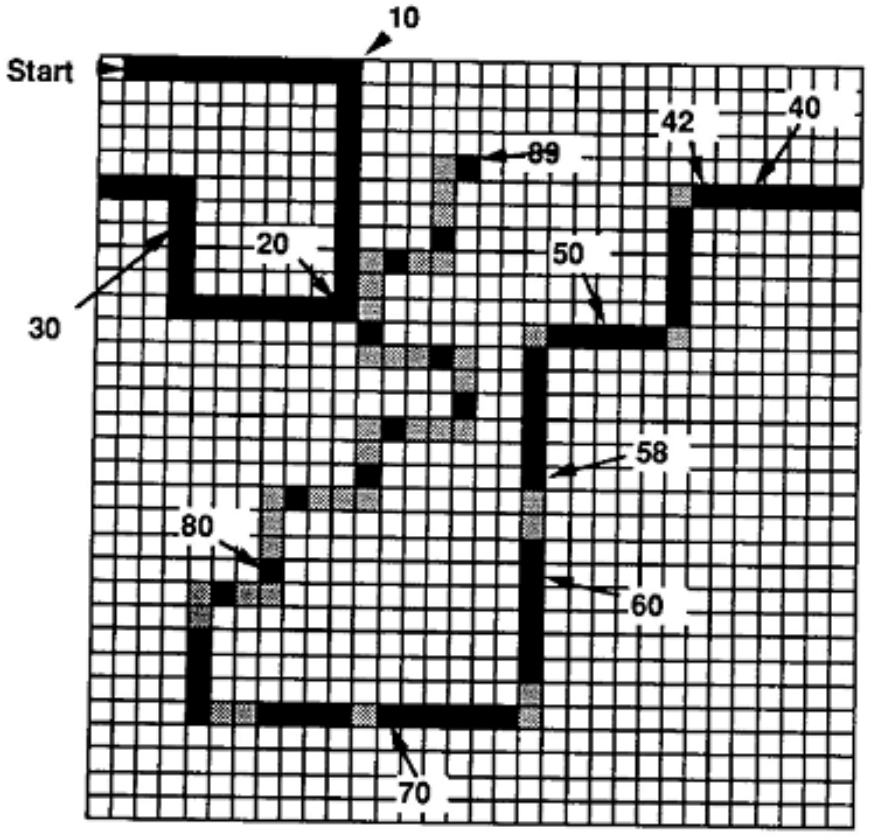


Figure 5.11: John Muir trail; Source: [9].

Consider one of the trails as shown in Figure 5.11, originally designed by Jefferson et al. called the 'John Muir' trail [9]. It is a 32x32 2-D grid world with a rectilinear broken trail. It starts with the path of continuously placed food pellets. The agent takes left or right turns at the corners to stay on the trail. However, the trail gets more complex with gaps in the path, with the last stretch, comprising of pellets placed in a way that can be collected by short and long knight moves (traverse straight followed by an immediate turn).

The agent needs memory of the previous sensory states it has traversed through to complete its state-space representation, since it does not have information on where in the trail it is at any point in time. This temporal trace helps the agent

to reconstruct the partial state representation to a full-state representation or a Markovian State representation. In our setup, the internal dynamics of the reservoir helps transform the POMDP of the tracker task to an MDP over which the reinforcement learning techniques (which assume MDP nature of the underlying state-space) can be applied to train the agent to traverse the trail efficiently.

For our discussions here, we'll consider the John Muir trail. Please refer Figure 5.11 for the John Muir trail layout. A detailed step-wise explanation of our setup to train weights in this problem context is covered in Section 4.1.1 taking a small grid-world as an example.

Our setup involves training an RBN-based agent to predict the probability with which performing an action at a certain state can guarantee maximizing future rewards. The training based on a reinforcement learning approach, happens through update of the read-out layer weights based on reinforcements/rewards observed during the agent's journey through the grid-world environment. The RBN reservoir receives input excitation through its interaction with the environment, which is essentially a binary bit-stream. The sensor input feed to the RBN-based reservoir is shown in Figure 4.6

The RBN reservoir excited by this continuous stream of inputs is allowed to evolve for τ timesteps before the output is readout from the output readout layer.

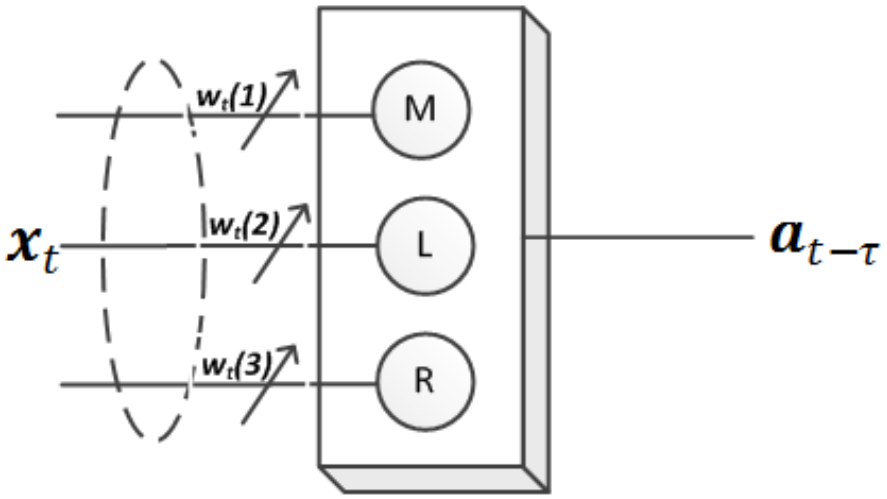


Figure 5.12: The readout-layer for the tracker task. The 3 output nodes each calculate value estimates for taking actions to move Forward, Left, or Right. The action associated to the output node with the maximum value estimate is chosen as the winner action which the agent performs on the trail.

The value readout from the output layer consisting of 3 nodes indicating the value function estimates of the future cumulative rewards that are guaranteed by performing that action $a_t \rightarrow$[Left,Right,Forward] at that particular state, as shown in Figure 5.12. This action is then executed on the simulated 2-D grid-world, and the resulting reward and the future states are observed.

This reward is used to generate the error signal which then is used to train the output readout layer weights through gradient descent.

$$e = r_t + \gamma Q(x_{t+1}, a_{t+1}) - Q(x_t, a_t), \tag{5.8}$$

where e = error signal, r_t = the reward, for our purposes, an action landing the agent on a cell with a food pellet gets a rewards of 1 and for every n^{th} consecutive action which does not land the agent on a food pellet, the agent receives a penalty of $n * -0.05$. We chose the value of the penalty by trial and error method, so to

make sure that when the action performed is wrong it does not completely wipe out the training during all those perceived situations when the action performed was right. However, the penalty is increased exponentially with every wrong move, to prevent the agent from wandering off from the main trail, $Q(x_t, a_t)$ = Value estimates read-out from the output layer of the reservoir at reservoir state x_t for the action a_t performed by the agent at the current timestep t, $Q(x_{t+1}, a_{t+1})$ = Value estimates read-out from the output layer of the reservoir at reservoir state x_{t+1} for the action a_{t+1} performed by the agent during the previous timestep $t+1$.

Equation 5.8 shows that the error calculation is not based on a target value as supplied by a *teacher*, since there is no one target for a given perceived situation in such problem contexts. But, is based on the sum of the rewards observed at the current timestep and the value estimates calculated at the read-out layer, for the state x_t and the action a_t performed by the agent, at the immediate future timestep. In reinforcement learning techniques, the read-out layer is trained to estimate cumulative future rewards, and hence the relationship between values estimated between consecutive timesteps is as highlighted in Equation 5.9.

$$r_t + \gamma Q(x_{t+1}, a_{t+1}) = Q(x_t, a_t) \tag{5.9}$$

The output readout layer weights are initialized randomly based on a normal distribution. The output nodes have linear activation functions. The weights go through episodes of training updates where for each episode, the agent explores the grid world for a fixed number of timesteps (timesteps fixed based on *a priori* knowledge of the size and complexity of the trail). The exact number of timesteps used in our experiments for the John Muir trail is listed in Table 5.7.

Table 5.6 lists the initial parameters that are critical to convergence of the

TD-based weight updates. Searching through the ten-dimensional initial parameter search space for conditions guaranteeing convergence is achieved through an evolutionary, *Differential Evolution* (DE), algorithm.

Table 5.6: Initial parameters list for the RBN-based agent trained on the 2-D grid-world task

Parameter	Value Type	Range	Description
β	Floating point	[0,1]	Learning rate for error back propagation
γ	Floating point	[0,1]	Reward discount factor, controls weight given to immediate or future rewards
N	Integer	Based on trail complexity	RBN reservoir size; Number of random binary nodes in the reservoir
τ	Integer	[0,3]	Delay in timesteps between the point when the input bit is applied to the reservoir to the timestep at which the output is read from the readout layer

Continued on next page

Table 5.6 – *Continued from previous page*

Parameter	Value Type	Range	Description
K	Floating point	[0,6]	Reservoir connectivity; mean number of neighbouring reservoir nodes from which a node receives input connections
n	Integer	Based on trail complexity	Minimum number of episodes for which the read-out layer weights need to train to solve the problem
t	Integer	Based on trail complexity	Minimum number of timesteps per episode needed for the agent to successfully traverse through the trail
ε	Floating point	[0,1]	The probability with which a non-greedy action is selected during the training phase
s	Floating point	[0,1]	The fraction of the number of episodes at which the ε starts decaying to a more greedy approach

Continued on next page

Table 5.6 – *Continued from previous page*

Parameter	Value Type	Range	Description
e	Floating point	[0,1]	The ε decay rate towards a greedy approach

For each of the parameters evolved through the DE, we impose upper and lower limit restrictions on the parameter search space in order to achieve faster convergence. The ranges were chosen based on a combination of context specific prior heuristics and also trial and error. Table 5.6 highlights the ranges chosen for most parameters.

We try to evolve the DE by evaluating the fitness shown in the Equation 5.10. The fitness is chosen such that the number of food pellets collected is maximized while the number of training episodes, the number of timesteps per episode and also the size of the reservoir used are minimized. This is done to get a solution which is optimal computationally, both speed-wise and memory-wise.

$$\mathbf{f} = (0.85 * \frac{n_{actual}}{n^U} - 0.05 * \frac{N_{actual} - N^L}{N^U - N^L} - 0.05 * \frac{e_{actual} - e^L}{e^U - e^L} - 0.05 * \frac{t_{actual} - t^L}{t^U - t^L}) * 1000, \tag{5.10}$$

where n_{actual} = number of actual foot pellets collected by the trained agent represented by the individual, n^U = number of total foot pellets in the 2-D grid-world, N_{actual} = actual reservoir size of the individual being evaluated, N^U = Upper limit on the reservoir size as imposed by design, N^L = Lower limit on the reservoir size as imposed by design, e^{actual} = actual number of episodes the individual is trained for, e^U = Upper limit on the number of training episodes as imposed by

design, e^L = Lower limit on the number of training episodes as imposed by design, t_{actual} = actual number of timesteps per episode the individual is trained for, t_U = Upper limit on the number of timesteps per training episodes as imposed by design, t_L = Lower limit on the number of timesteps per training episodes as imposed by design.

In Equation 5.10, we normalize each part of the fitness. We also associate specific weight to each part, which is fixed based on trial and error. The number of actual food pellets is normalized based on the number of total food pellets that can be collected on the trail. We associate a positive and large weight to the component that represents the number of food pellets collected by the individual, since we want an individual with the ability of collecting more food pellets to be always picked up as the best, while at the same time trying to optimize the other memory intensive components. The reservoir size, training episode lengths and timesteps per episode are normalized around the upper and lower ranges that the DE-search is restricted to. We associate negative and smaller weight to minimize these other components.

During the initial training episodes, to guarantee fair sampling across the action space, a non-greedy approach is taken. The ε-greedy approach selects the action which guarantees maximum future rewards at each timestep, only with a probability of 1-ε. For the rest of the times a random action is picked and executed. This is to handle any prior biases that may exist in the initial weight initialization that might favor one action over the other. By this, the entire action space is explored before converging towards a more greedy action policy. In our setup, we try to exponentially decay the ε to allow smooth convergence to a greedy policy from a non-greedy one and hence introduce two more variables in the process. The s

and e parameters listed in Table 5.6 control the ε decay rate through the training process. During the training phase, we gradually modify the value of ε, such that, as time, $t \longrightarrow \infty$, ε-greedy $\longrightarrow$ greedy policy. We achieve this by exponentially decaying the ε value based on the Equation 5.11.

$$\begin{aligned} If(i/n_e &> s) \\ \varepsilon &= \varepsilon * e, \end{aligned} \tag{5.11}$$

where i = the current training episode, n_e = total number of episodes for which the readout layer weights are trained, s = fraction of the number of episodes at which the ε starts decaying to a greedy approach, $e = \varepsilon$ decay rate towards a greedy approach.

We found the individual in Table 5.7 to show best performance on the John Muir trail. We initialize 500 reservoirs with the initial parameter set in Table 5.7 to evaluate the best number of reservoirs, to average across, that can depict the performance of the agent in this environment more accurately.

Table 5.7: Best individual for the 2-D John Muir grid-world task

β	γ	N	τ	K	n	t	ε	s	e
0.01	0.91	185	0	4.05	5480	131	0.741	0.75	0.735

We use a Q-Learning setup to compare the performance of the RBN-based agent. The Q-Learning approach tries to evolve the Q-Values in a canonical lookup table of dimension $X \times A$. Since this is a POMDP problem, the complete state x_t is a function of previous states, as shown in Equation 5.12

$$\mathbf{x}(n) = f(\mathbf{x}(n-1), \mathbf{s}(n)) \tag{5.12}$$

We re-construct the space X, by linear re-combination of the previous states.

$$\mathbf{x}(n) = s(n)s(n-1)s(n-1)....s(n-m), \tag{5.13}$$

where m is the temporal embedding length needed to completely reconstruct the full-state.

Note that with increase in m the dimension of X in terms of S increases. This leads to a larger canonical-lookup table of Q-Values that need to be evolved through the Q-Learning process. With the increased m, the policy search space increases by an order of $O(2^m)$. Similar to the RBN-based approach, Q-Learning approach also has its set of initial parameters on which convergence is heavily dependant. We use the same DE approach to search the eight dimensional initial parameter search space. Here, the length of temporal embedding is explicit unlike in a RBN-based approach where the embedding is implicit. This extra parameter is also added to the DE-search space.

Table 5.8: Initial parameters list for the Q-Learning agent trained on the 2-D grid-world task

Parameter	Value Type	Range	Description
β	Floating point	[0,1]	Learning rate for error back propagation

Continued on next page

Table 5.8 – *Continued from previous page*

Parameter	Value Type	Range	Description
γ	Floating point	[0,1]	Reward discount factor, controls weight given to immediate or future rewards
n	Integer	Based on trail complexity	Minimum number of episodes for which the read-out layer weights need to train to solve the problem
t	Integer	Based on trail complexity	Minimum number of timesteps per episode needed for the agent to successfully traverse through the trail
ε	Floating point	[0,1]	The probability with which a non-greedy action is selected during the training phase
s	Floating point	[0,1]	The fraction of the number of episodes at which the ε starts decaying to a more greedy approach

Continued on next page

Table 5.8 – *Continued from previous page*

Parameter	Value Type	Range	Description
e	Floating point	[0,1]	The ε decay rate towards a greedy approach
states	Integer	[1,8]	The length of temporal embedding for reconstruction

As with the RBN-approach, we try to evolve the DE by evaluating the fitness shown in the Equation 5.14. The fitness is chosen such that the number of food pellets collected is maximized while the number of training episodes, the number of timesteps per episode and also the length of the temporal embedding used are minimized. This is done to get a solution which is optimal computationally, both speed-wise and memory-wise.

For each of the parameters evolved through the DE, we impose upper and lower limit restrictions on the parameter search space in order to achieve faster convergence. The ranges were chosen based on a combination of context specific prior heuristics and also trial and error. Table 5.8 highlights the ranges chosen for most parameters.

$$\mathbf{f} = (0.85 * \frac{n_{actual}}{n^U} - 0.05 * \frac{e_{actual} - e^L}{e^U - e^L} - 0.05 * \frac{t_{actual} - t^L}{t^U - t^L} - 0.05 * \frac{k_{actual} - k^L}{k^U - k^L}) * 1000 \tag{5.14}$$

where, n_{actual} = number of actual foot pellets collected by the trained agent, represented by the individual, n^U = number of total foot pellets in the 2-D grid-world, e^{actual} = actual number of episodes the individual is trained for, e^U = Upper

limit on the number of training episodes as imposed by design, e^L = Lower limit on the number of training episodes as imposed by design, t_{actual} = actual number of timesteps per episode the individual is trained for, t_U = Upper limit on the number of timesteps per training episodes as imposed by design, t_L = Lower limit on the number of timesteps per training episodes as imposed by design. k_{actual} = actual temporal embedding of the individual being evaluated, k^U = Upper limit on temporal embedding as imposed by design, k^L = Lower limit on temporal embedding as imposed by design.

In Equation 5.14, we normalize each part of the fitness. We also associate specific weight to each part, which is fixed based on trial and error. The number of actual food pellets is normalized based on the number of total food pellets that can be collected on the trail. We associate a positive and large weight to the component that represents the number of food pellets collected by the individual, since we want an individual with the ability of collecting more food pellets to be always picked up as the best, while at the same time trying to optimize the other memory intensive components. The length of the temporal embedding, training episode lengths and timesteps per episode are normalized around the upper and lower ranges that the DE-search is restricted to. We associate negative and smaller weight to minimize these other components.

We also performed the same set of experiments on the Santa Fe trail [8], to find the individual that gives the best performance on the trail based on our setup. Like with the John Muir trail, we compare the results of the performance of our RBN-based setup to that of the Q-Learning setup. The parameters that we used to initialize the RBN-based reservoir, as tuned by the DE, for the Santa Fe trail is as listed in Table 5.9

Table 5.9: Best individual for the 2-D Santa Fe grid-world task

β	γ	N	τ	K	n	t	ε	s	e
0.01	0.91	179	0	3.15	2680	152	0.727	0.76	0.83

5.3.2 Results and Discussion - Tracker Task

In this section we present our experimental findings of running the RBN-based agent as per the best individual that evolved out of the DE as presented in Table 5.7. We perform experiments initially on 500 random reservoirs created based on the parameter set listed in Table 5.7. We pick the top 20 best performing reservoirs of the lot and plot the average fitness with error bars as shown in Figure 5.13. By observation, we deduce that averaging across 10 best reservoirs gives a good indication of the trade-off between the best performance and variability that can be expected out of the reservoir recipe that we considered. For all plots in this section we average data across these 10 best performing random reservoirs.

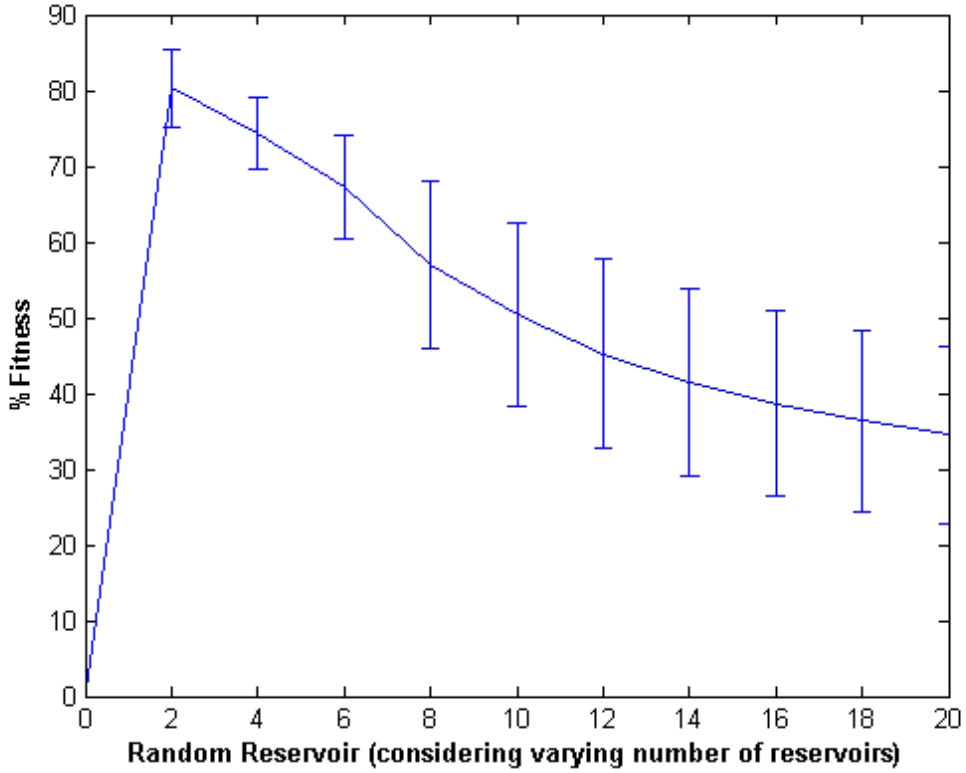


Figure 5.13: Error bars indicating performance of the best individual on the John Muir trail, considering average across varying number of random reservoirs.

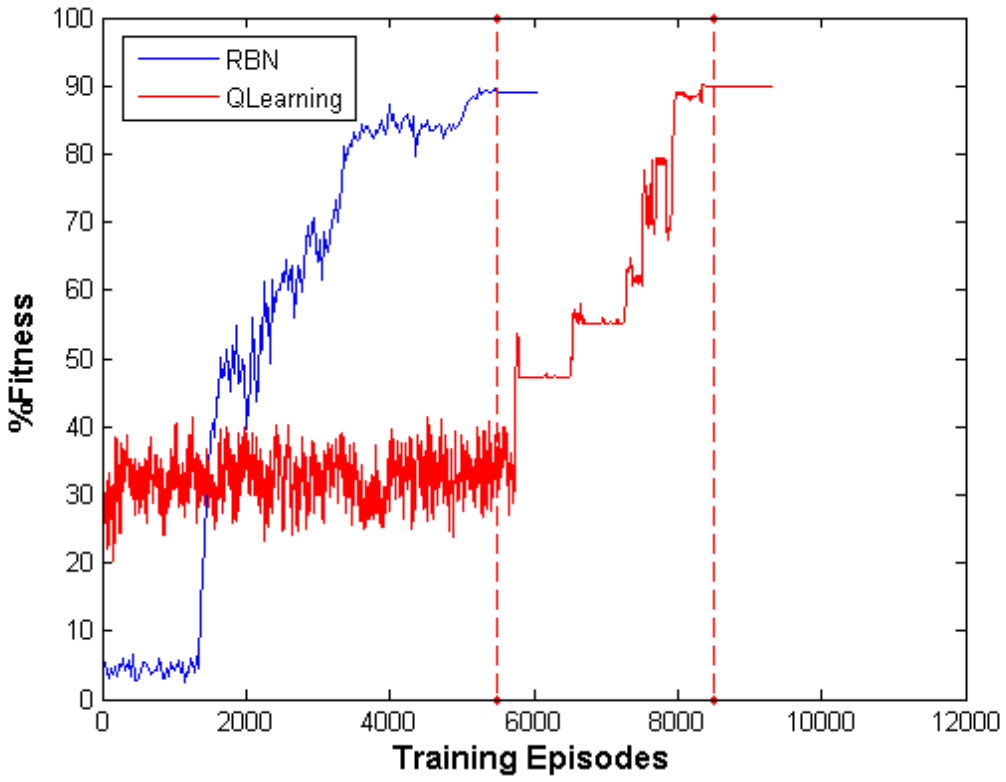


Figure 5.14: This convergence graph compares the training performance of the RBN-based agent (averaged across 10 random reservoirs) to the Q-Learning agent for the John Muir trail.The vertical lines drawn indicate the training episodes at which convergence occurs for each agent. It can be seen that the RBN-based agent converges at a rate approximately 30% faster than the QL-based agent.

In the Figure 5.14 it can be seen that the best individual evolved through the DE for both the RBN-based and the Q-Learning based agents have similar performances in terms of the number of food pellets collected, with the RBN-based agent collecting 83 of the 89 possible food-pellets while the Q-Learning agent collects 81 of the 89 possible food-pellets for the John Muir. However, the RBN-based agent converges at a speed which is 30% faster than the Q-Learning-based agent. We owe this slower convergence to the huge state-action space of the Q-Learning based agent. The best individual for Q-Learning based agent needs a

memory embedding of length 7. Which means, a history of input signals spanning across 7 previous timesteps is needed for the Q-Learning agent to take optimal decisions at its perceived situations. Due to which, the Q-value matrix that needs to converge to approximate the POMDP considered approximately is of size $2^7 \times 3$, which is the size of the state-action space for the Q-Learning agent considered. Here, 2^7 is the number of binary states possible for a bit string of length 7 and 3 is the number of actions possible per state. We do not face the issue of increase in state-space due to temporal embedding with the RBN-based agent, since the temporal embedding in dynamic systems is implicit.

In [9], Jefferson et al., deal with the John Muir trail problem and design Finite-State-Automata machines (FSMs) that can traverse through the trail optimally. The FSMs are coded into a genome, which is then taken through evolutionary cycles to improve the fitness. The number of states that an agent traverses through to be able to solve the trail is encoded as binary bit representations in the genome. A 13-state FSM (that can be represented by 4 bits), as shown in Figure 5.15, was evolved to solve the John Muir trail with a perfect score of 89 in approximately 200 timesteps. It was found through their work that the chances of the initial population being able to converge to an individual with the best score was tightly related to the presence of high fitness individuals during the first generation itself. However, the majority of evolutions with random initial populations converged to a best score of 81 [9].

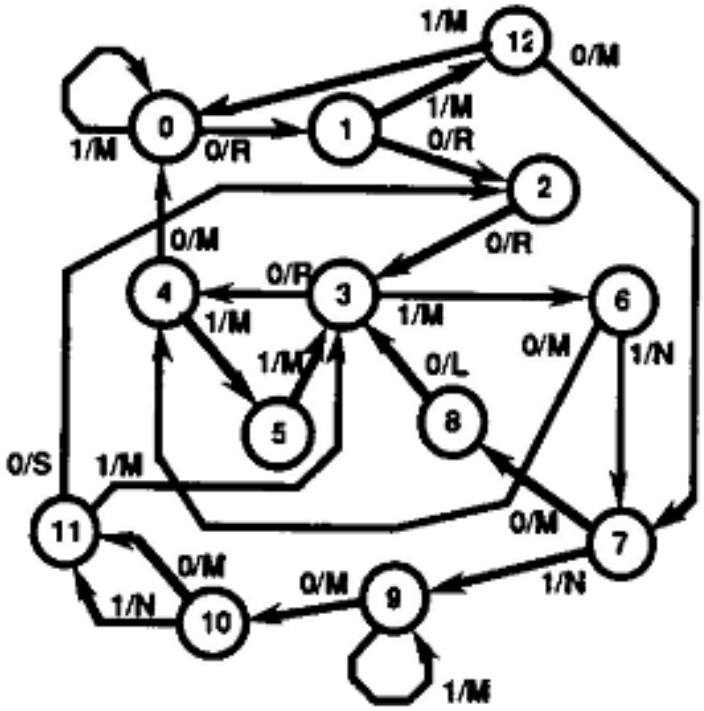


Figure 5.15: FSA solving the John Muir trail with perfect score of 89 [9].

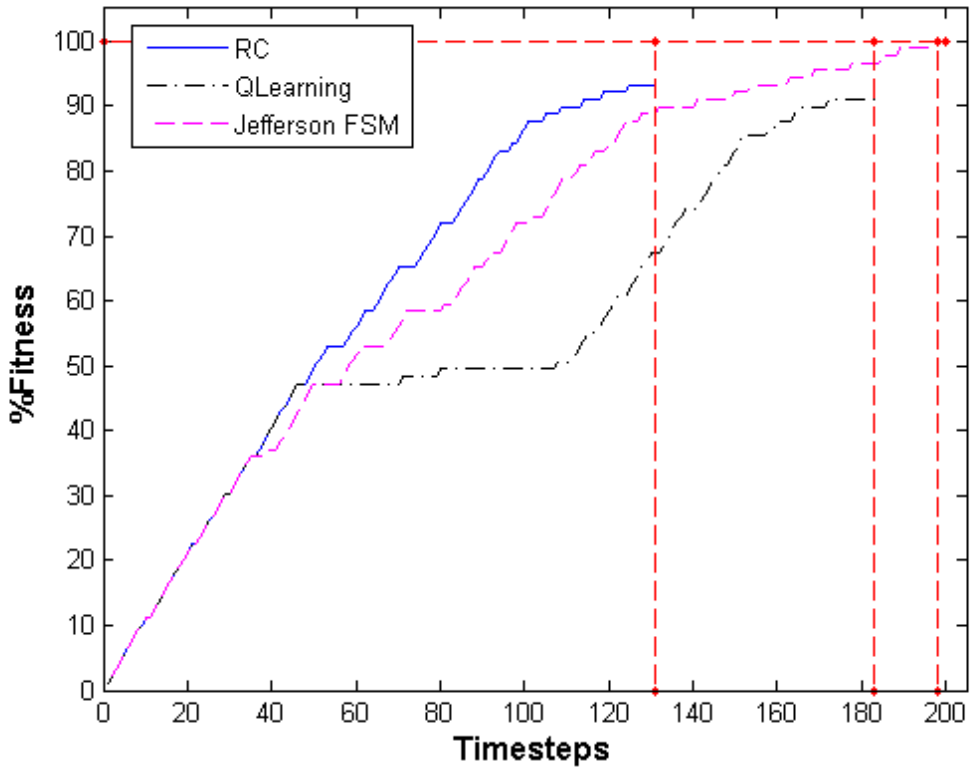


Figure 5.16: The graph compares performance of trained agent. The number of food pellets collected by a trained RBN-based agent, QL-based agent and the FSA evolved in works of Jefferson et al. [9] are plotted as a progression of one complete episode, which represents the agent's journey through the John Muir trail. It is seen that the RBN-based agent takes the most optimal strategy in comparison to the Q-Learning based agent or Jefferson's FSM. At the 131st timestep, the RBN-based agent collects over 93% of the 89 food pellets while the Q-Learning based agent collects only around 65% and Jefferson's FSM collects around 88% of the total food pellets.

Figure 5.16 compares that the number of rewards/food pellets collected by the RBN-based agent, QL-based agent and Jefferson's FSM. It can be seen that the RBN-based agent collects the maximum number of food pellets during its short journey in comparison to the number of pellets collected by the other two agents during the same time. However, Jefferson's FSM is more optimal in the long run

in collecting all the food pellets by traversing through the last complex stretch of the John Muir trail. It takes about 75 extra timesteps to traverse and collect the last 9 food pellets. The RBN-based agent does not progress beyond the 131st timestep. The Q-Learning-based policy takes much longer than both the other agents by collecting 81 food pellets in 183 timesteps. Though the RBN-based agent is optimal in the sense of being fast to collect the maximum food pellets in the shortest time in comparison to both the other agents, it does not completely traverse the trail as does Jefferson's FSM.

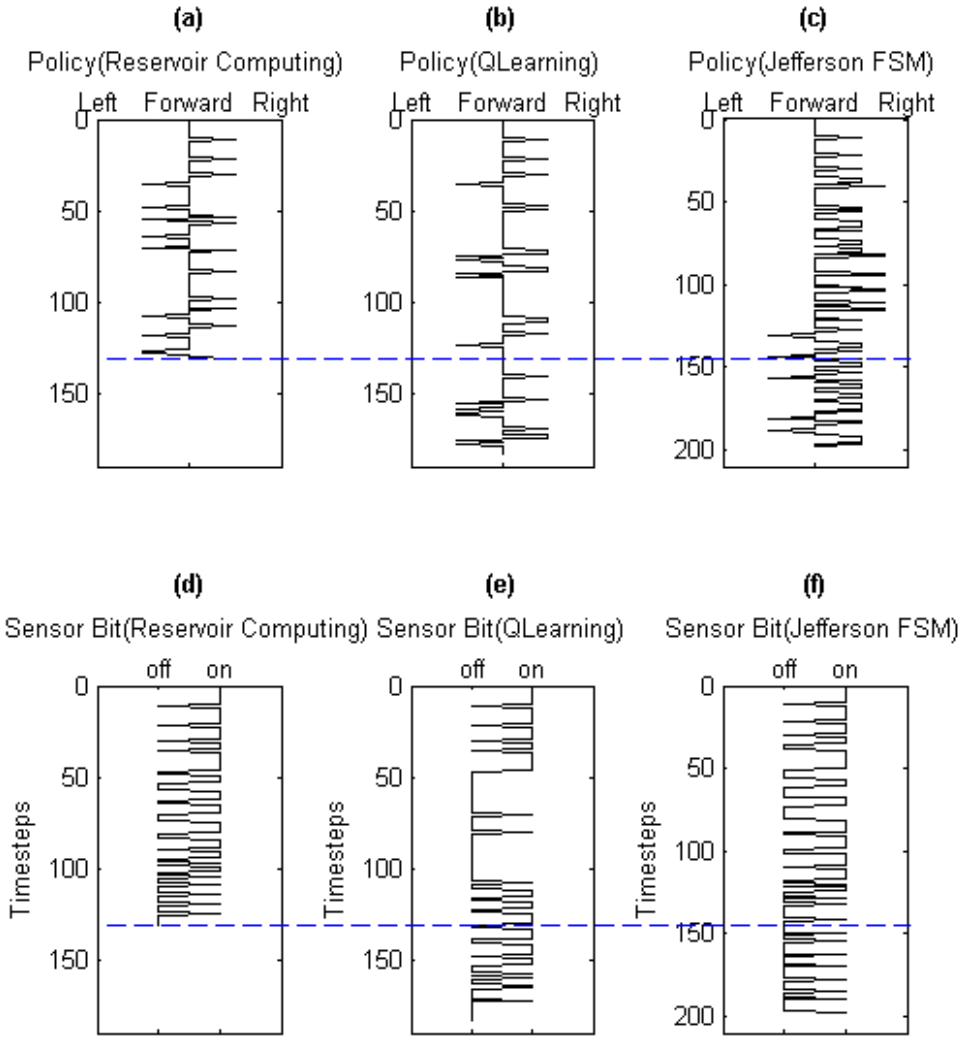


Figure 5.17: (a),(b),(c) shows the step-by-step decision taken by the trained/evolved agents to the step-by-step sensor input as shown in (d), (e) and (f).

The RBN-based agent is essentially learning a state controller through its dynamics and weight updates through the training process. Figure 5.17, shows the actions performed by each of the agents in response to the sensor signal received.

In other words, the Figure 5.17 compares the policies followed by the three state controllers (or agents). The sensor input graph for the Q-Learning agent shows that the agent wanders off from the main trail between the 50^{th} to 110^{th} timestep. The RBN-based agent takes a more optimal approach and stays on the trail for most part of its journey.

Figure 5.18 is a plot of all 10 best performing random reservoirs with the initial parameters set according to Table 5.7. Figure 5.19 shows the journey of the best individual through John Muir trail.

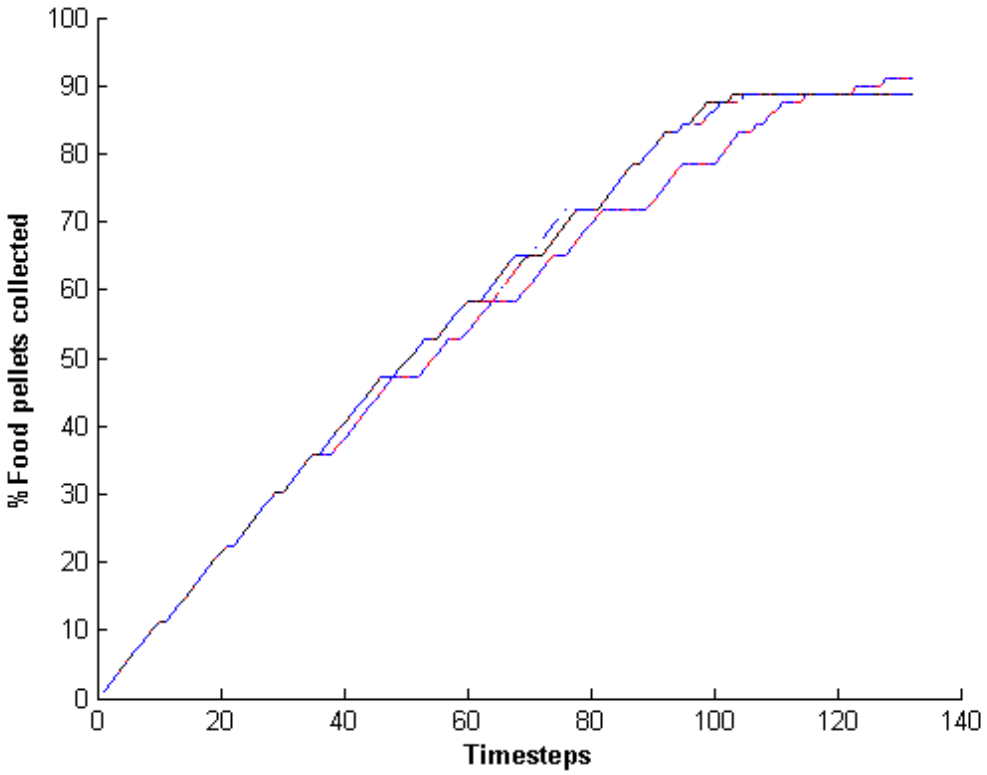


Figure 5.18: The graph shows the performance of top 10 best performing trained agents as a progression of one episode. The agents were all trained on John Muir trail.

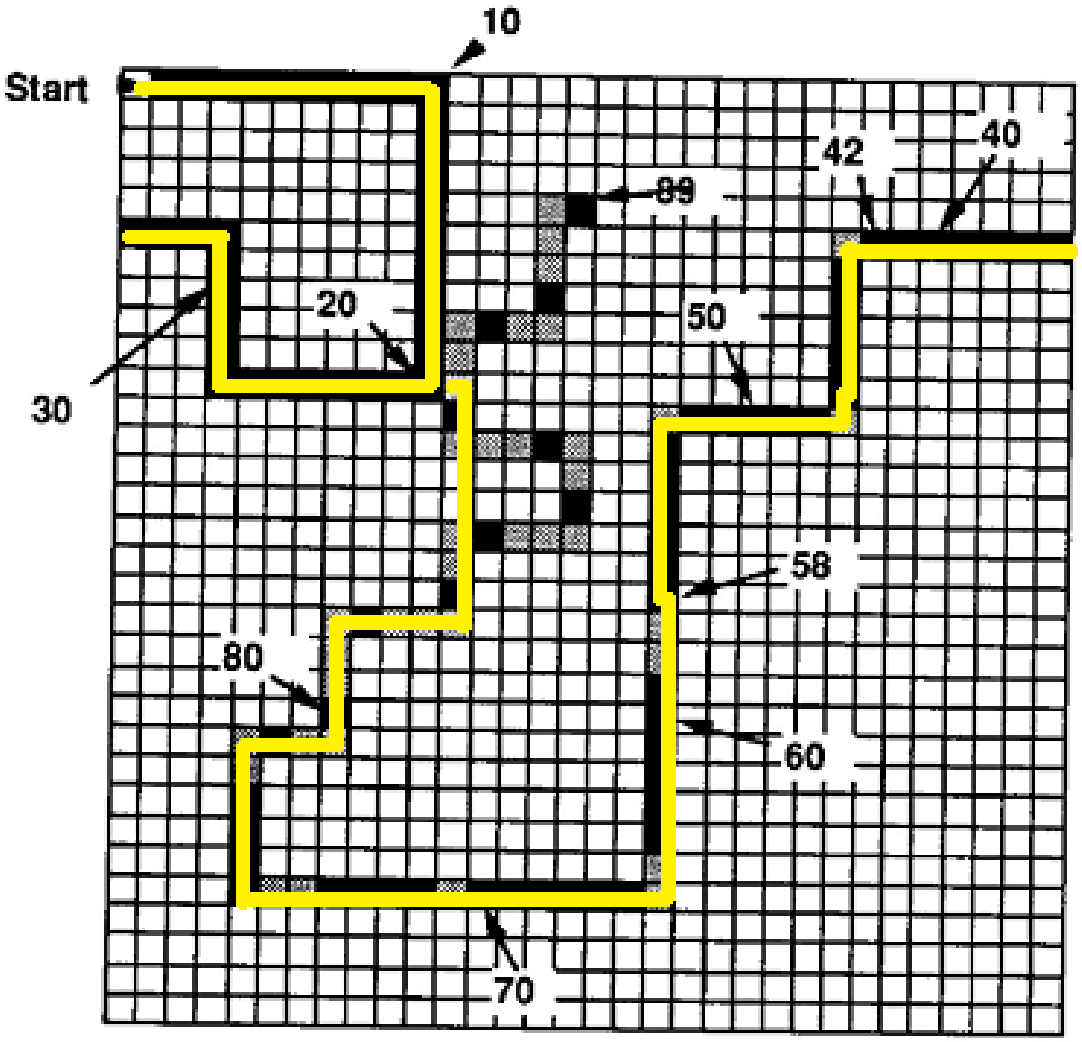


Figure 5.19: Journey of the best individual through the John Muir trail.

We also compared the performance of the RBN-based agent and the Q-Learning agent, as evaluated on the Santa Fe trail. The policy followed by a trained RBN-based agent and the Q-Learning agent are plotted as shown in Figure 5.20.

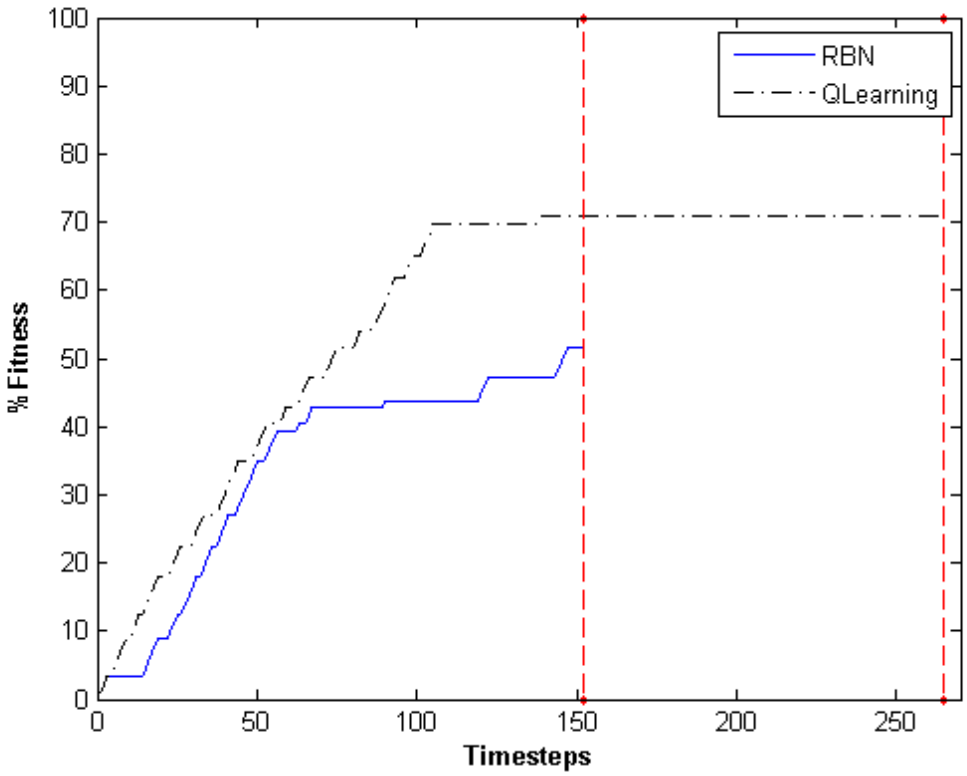


Figure 5.20: The graph compares performance of trained agents. The number of food pellets collected by a trained RBN-based agent and Q-Learning based agent are plotted as a progression of one complete episode, which represents the agent's journey through the Santa Fe trail. It is seen that the Q-Learning agent performs better in comparison to the RBN-based agent on this trail problem. Though the overall performance of both the agents on the trail was not impressive, it can be seen that at the 152nd timestep, while the RBN-based agent collects around 50% of the 89 food pellets, the Q-Learning based agent collects over 70% of the total food pellets on the Santa Fe trail.

It can be seen from Figure 5.20, that both the agent do not perform well on the Santa Fe trail problem. The Q-Learning agent, which performs better of the two, collects a total of 63 of the total possible 89 food pellets, while the RBN-based agent collects only 46 of the total food pellets. Koza et al. designed the Santa Fe trail problem to be slightly more complex than the John Muir by introducing longer

gaps. In their work, Koza et al. solved the trail through a genetic programming approach [8]. We conclude that the size of the reservoir we used may not have been big enough to encode the memory needed to solve the longer gaps and thus the resultant longer policy, in terms of the number of timesteps, needed to solve the Santa Fe trail problem.

In the grid-world based problems the agents basically memorize the trail. Generalization performance of these trained state controllers to noisy representations of the trail is not usually experimented with in such problem contexts. However, we conducted experiments to study the robustness of a trained RBN-based agent to injection of noise in the original trail. We progressively inject varying percentages of noise in the trail by removing food pellets from the original trail.

As can be seen in Figure 5.21 and 5.22, RBN-based agents show only a slight drop in performance, on both the John Muir trail and the Santa Fe trail, to noise injection of 10% and show a steep drop in performance for further increase in % noise injection.

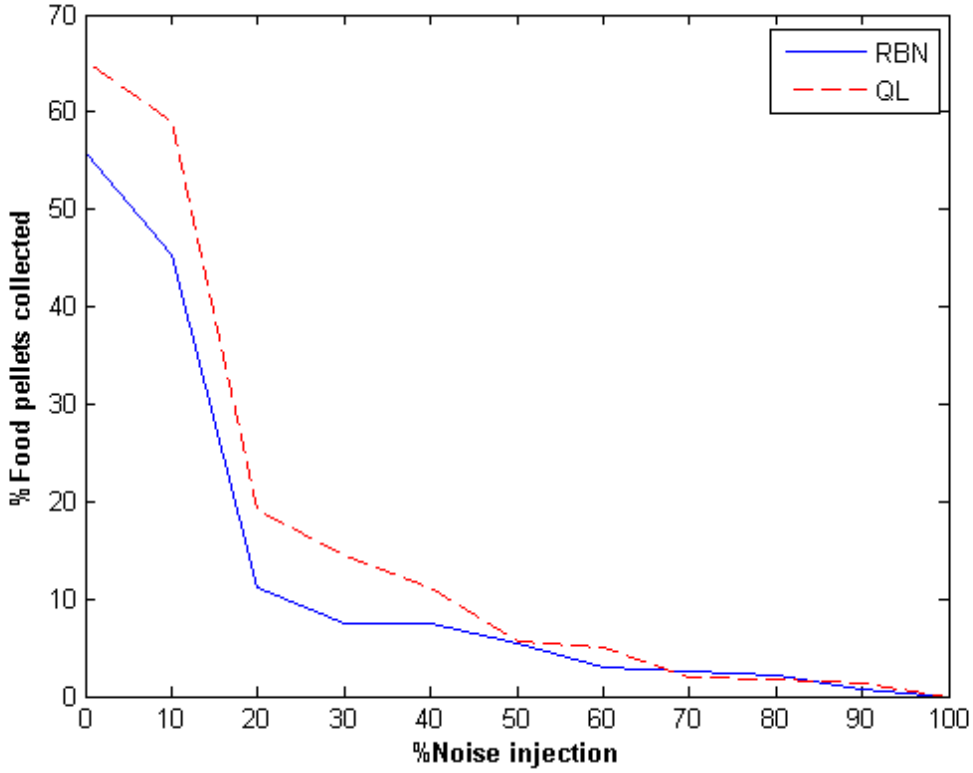


Figure 5.21: The graph compares performance of RBN-based and Q-Learning based agents in noisy environments. Average performance across 10 best performing individuals is plotted for both the RBN-based and Q-Learning based setup. These individuals showed best tolerance to noise injections on the John Muir trail. The noise injection included removing varying percentages of food pellets from randomly chosen cells on the original trail.

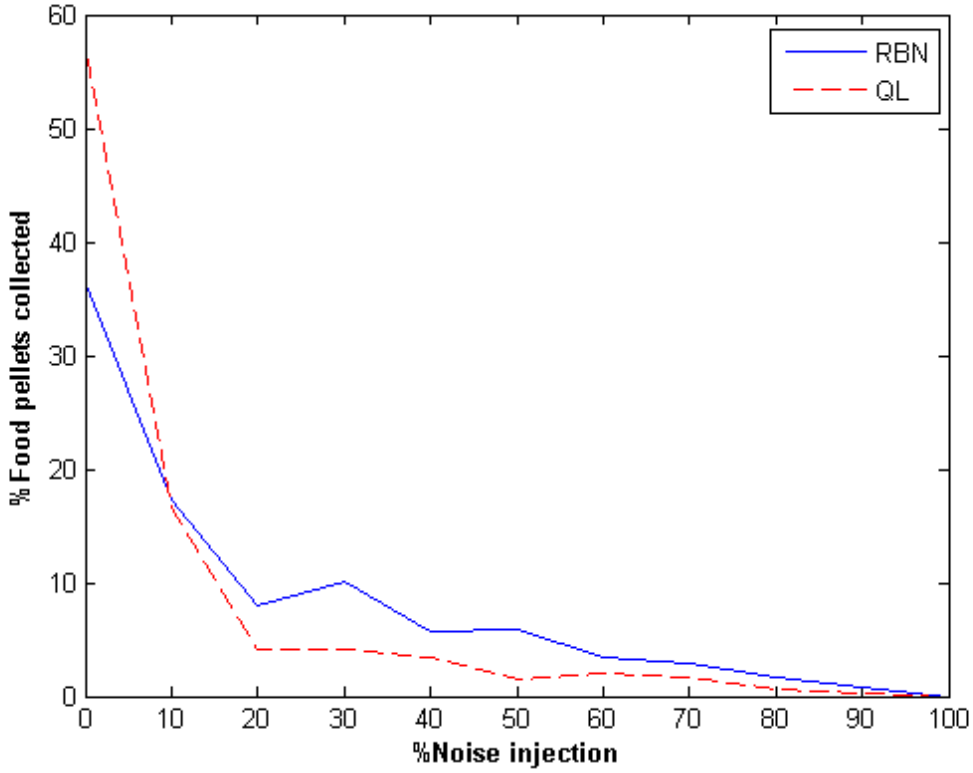


Figure 5.22: The graph compares performance of RBN-based and Q-Learning based agents in noisy environments. Average performance across 10 best performing individuals is plotted for both the RBN-based and Q-Learning based setup. These individuals showed best tolerance to noise injections on the Santa Fe trail. The noise injection included removing varying percentages of food pellets from randomly chosen cells on the original trail.

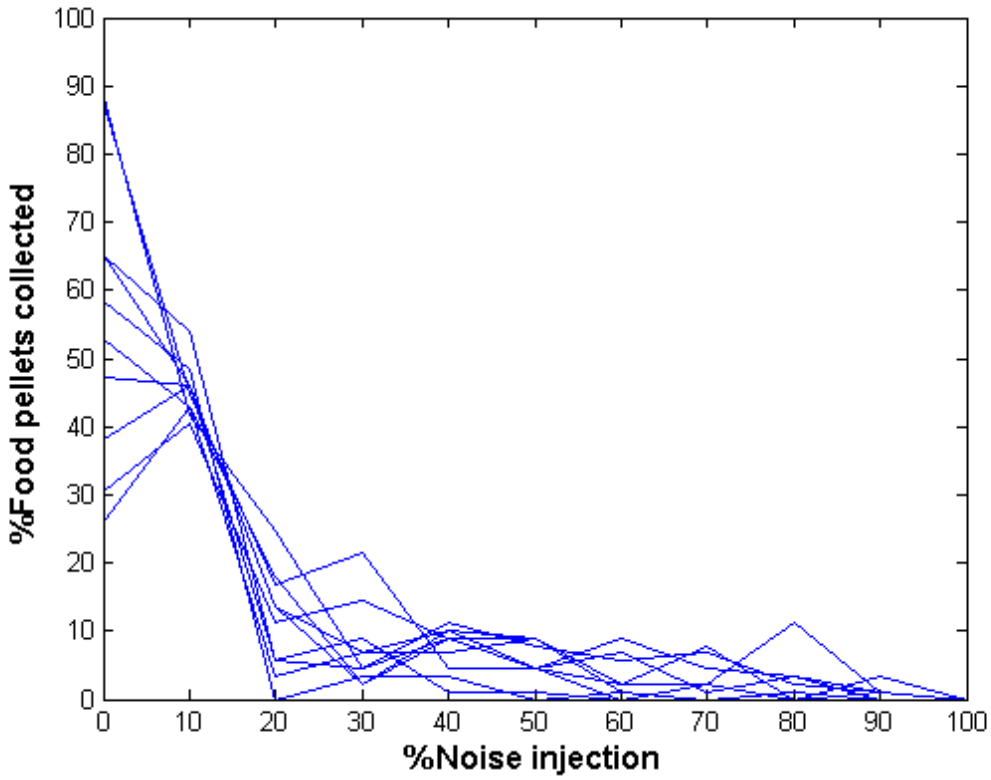


Figure 5.23: Performance of all 10 RBN-based agents which showed best tolerance to noise injections on the John Muir.

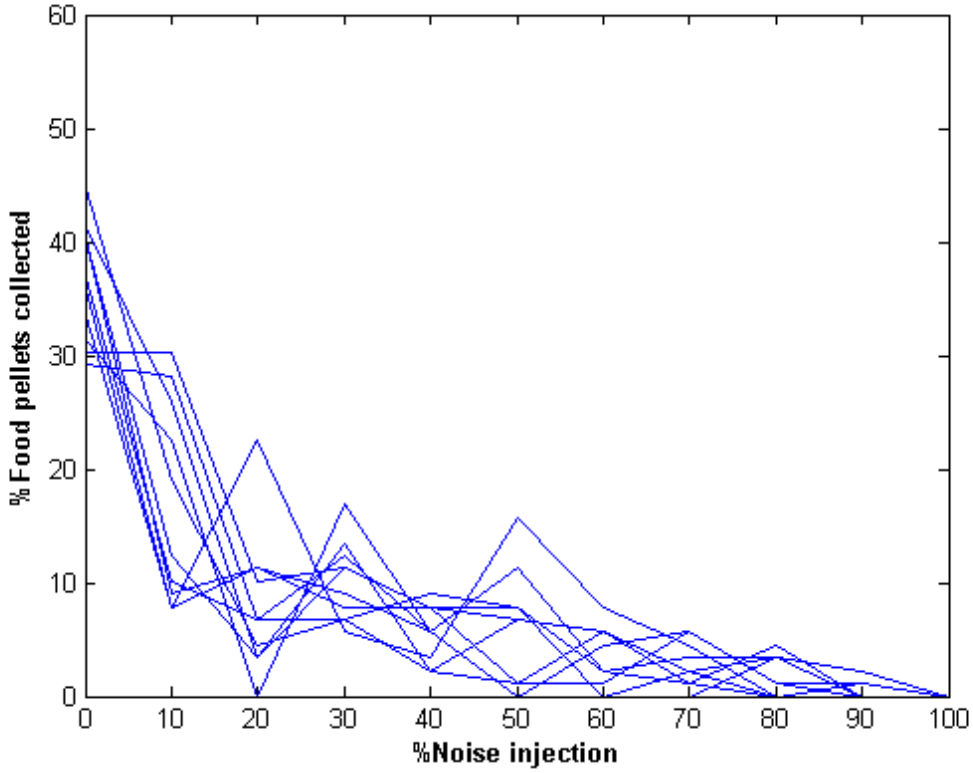


Figure 5.24: Performance of all 10 RBN-based agents which showed best tolerance to noise injections on the Santa Fe.

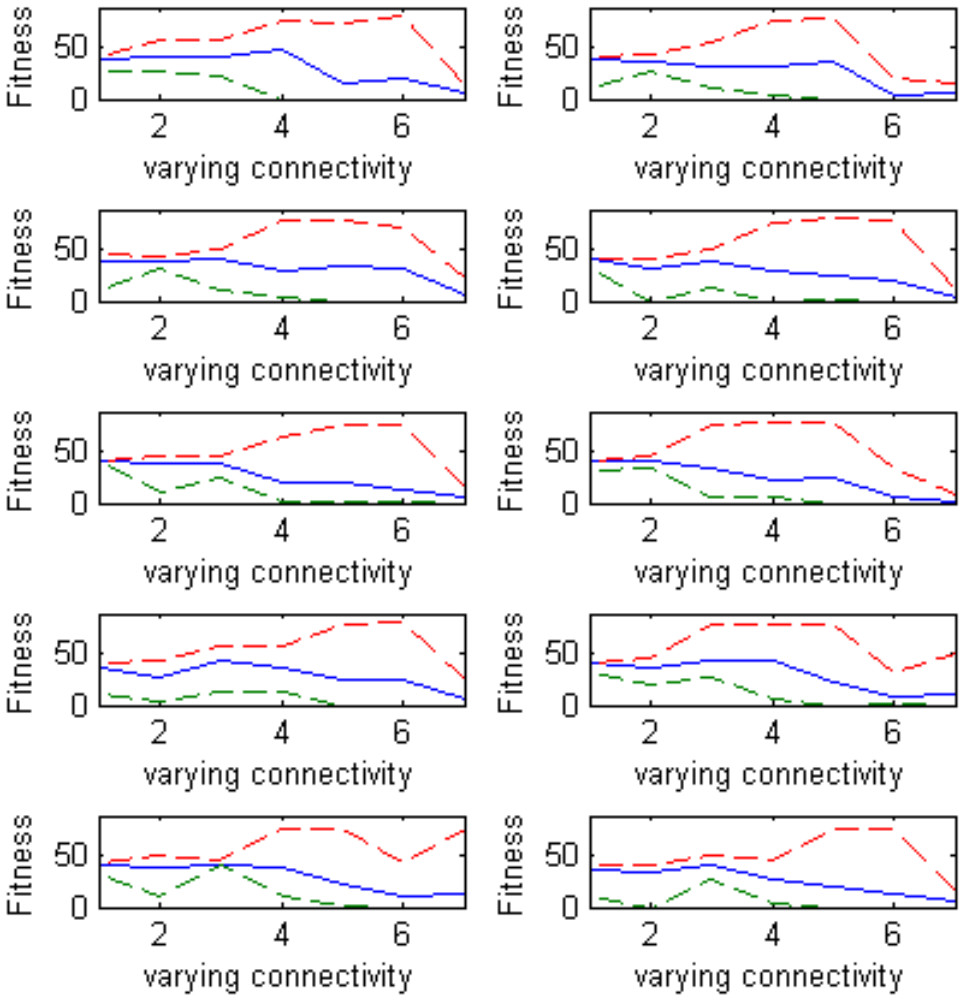


Figure 5.25: The average, minimum and maximum performance across 10 best random reservoirs, on the John Muir trail, are plotted by fixing the reservoir size N and delay τ for varying connectivity in each case. The plots correspond to varying $N \rightarrow [175, 179]$ and varying $\tau \rightarrow [0, 4]$ read left-right and top-bottom in that order. It is seen that the best reservoir performance for the range of N and τ considered is achieved for connectivity $K \rightarrow [3, 6]$

Figure 5.25 show the performance of 10 best performing random reservoirs for

varying reservoir sizes, $N \rightarrow [175, 179]$, varying delay $\tau \rightarrow [0, 4]$ and varying connectivity $K \rightarrow [1, 7]$ as evaluated with considering John Muir trail as the environment. We conducted this set of experiments to observe the reservoir connectivity required to develop RBN-based agents with optimal solutions. We found that for $K \rightarrow [3, 6]$ the best performance is achieved across the range of N and τ considered. Derrida et al. among others have established through extensive research that at critical connectivity is achieved at $K = 2$. Critical connectivity is of particular interest since it is found that the memory capacity and the information processing of RBN-based reservoirs are balanced at criticality. Lizier et al. have established through their work that high connectivity, that is $K > 2$ results in RBNs in the *super critical* regime. In the super critical, regime RBNs are believed to have higher information processing while having lower memory capacity [68]. Since we have not extensively focussed on this aspect of RBNs in our experimentation we want to emphasize this due to its importance on hardware implementation of such networks. By extension of Lizier et al.'s hypothesis, it seems like RBNs set in reinforcement learning tasks, similar to the one we experimented with perform well in the *super critical* regime. In terms of physical implementation, *super criticality* corresponds to higher interconnects and hence can prove to be expensive.

We conducted experiments to setup an RBN-based reservoir in a POMDP context and trained it through a reinforcement learning technique. The resulting agent or the state controller was shown to perform comparable to other benchmark techniques which have been used previously to solve such problems on one of the trails, ie., the John Muir trail. However, the agent's performance was not so impressive in comparison with the Q-Learning agent on the Santa Fe trail. We also generalized the RBN-based agent's performance on noisy representations of

the trails it was originally trained on, to experiment with the agent's robustness to noise. It was seen that the agent had tolerance to noise injection of 10% and lower, while showed high sensitivity to noise levels higher than that. We also found that the reservoirs needed to solve such tasks lie in the *super critical* regime, which has a significant impact on physical implementation of agents required to solve such tasks.

6

Conclusion

In this thesis, we have conducted a set of experiments and studied the feasibility of setting up and training RBN models as autonomous real-time agents in model-free environments. We train the RBN read-out layer interactively through reinforcement learning techniques. We experiment the ability of RBNs to perform as MDPs, where the state, as observed by the agent with respect to its environment, is complete. We also observed the performance of RBNs in POMDP contexts involving making decisions in perceived situations or when the state of the agent with respect to its environment is partially observable. We conducted experiments to test robustness of agents trained in POMDPs by injecting noise in its environment. We also observed the average connectivity needed for the RBNs to be setup as agents in POMDP contexts.

This research is a step towards studying the feasibility of programming emerging self-assembled architectures through interactive techniques. Majority of research till date in related areas have mostly explored techniques involving programming such architectures for known input-output mappings. The contexts we consider involve programming these architectures to function as goal-driven state controllers, which are capable of making real-time decisions when presented with a situation.

We augmented an existing RBN simulation framework developed in C++ with a read-out layer, also developed in C++. This read-out layer is provided capability to train its weights based on reinforcement learning techniques. We also developed

frameworks to simulate the MDP and the POMDP environments that we experiment with. We customized a DE based evolutionary framework to optimize the parameters needed to initialize our setup.

In our first experiment, we test our RBN framework by setting it up in a supervised learning context. We trained the read-out layer weights through gradient descent for a known target in a temporal-parity problem context. We were able to achieve 80% to 100% generalization accuracy across 3 best performing random reservoirs for varying input bit-streams considered with respect to this task. With this validation of accuracy in performance of read-out layer in scenarios with known output targets, we proceeded further to experiment our framework in situations with no known output targets.

In our next step, we setup the RBN in an MDP problem context by implementing it to solve a n-arm bandit problem. In this experiment, the RBN was trained interactively through a *Temporal Difference* based reinforcement learning technique, to behave as an agent. The read-out layer weights were updated based on rewards/penalties received through the RBN-based agent's interaction with the environment. In this setup, we also followed an algorithm to balance the exploration and exploitation phases during training. We start our training of the RBN-based agent with a ε-greedy approach and smoothly transition into a greedy approach towards the later stages of training. This was particularly necessary in this problem context, since all processes or arms need to be uniformly sampled during training to accurately adapt the weights. It was observed at end of the training process, that the RBN-based agent had successfully incorporated the inherent reward probabilities associated to the n-arm processes. We experimented with varying number of arms, and observed that in each setup the agent learns

to accurately incorporate in its weights the reward probability associated to each arm.

Finally, we conducted experiments to setup an RBN-based reservoir in a POMDP context. The RBN was trained to function as an agent foraging in a grid-world environment. We conducted all our experiments on two complex 32×32 2-D grid world trails, the John Muir trail designed by Jefferson et al. [9] and the Santa Fe trail designed by Koza et al. [8]. We trained the agent using reinforcement learning techniques. On the John Muir trail, the trained agent or the state controller performed comparable to other benchmark techniques which have been used previously to solve such problems. We implemented a Q-Learning framework with explicit temporal embedding capability to compare with the RBN-based agent. We found that the Q-Learning based agent took 30% longer than the RBN-based agent to converge for a specific environment (John Muir trail). We also found that the RBN-based solution was more optimal and picked 83 of the total 89 possible food pellets in 16% less timesteps to the 81 food pellets picked by the Q-Learning solution, for that specific case. However, for the Santa Fe trail, the Q-Learning agent collected 19% more food pellets than the RBN-based agent does for the same number of timesteps. We also generalized the RBN-based agent's performance on noisy representations of the trails it was originally trained on, to experiment with the agent's robustness to noise. It was seen that the agent had tolerance to noise injection of 10% and lower, while showed high sensitivity to noise levels higher than that. We also found that the reservoirs needed to solve such tasks lie in the *super critical* regime which has a significant impact on physical implementation of agents required to solve such tasks, using self-assembled architectures. We used our *Differential Evolution* framework to optimize the initial parameters for both

the RBN-based and Q-Learning based setup.

We conclude that RBNs can be successfully setup in solving goal-driven problems by interactively training its read-out layer through reinforcement learning approach. They are particularly efficient in solving POMDPs due to the dynamics achieved by the recurrent connections in such networks. Though the RBN-based agents performed comparable to other benchmark techniques such as evolutionary approaches talked about in related literatures, we see that they were able to successfully adapt to their set environments in an interactive manner.

In future work, we would like to extend our framework to memristor based hardware implementations of reservoirs. We aspire to setup memristor based reservoirs augmented with read-out layers in reinforcement learning contexts and train them interactively to function as autonomous agents.

References

[1] D. H. Nguyen and B. Widrow, "Neural networks for self-learning control systems," *IEEE Control Systems Magazine*, vol. 10, no. 3, pp. 18–23, 1990.

[2] H. Jaeger, "Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the echo state network approach," tech. rep., GMD Forschungszentrum Informationstechnik, 2002.

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

[4] B. Derrida and Y. Pomeau, "Random Networks of Automata: A Simple Annealed Approximation," *Europhysics Letters*, vol. 1, pp. 45–49, Jan. 1986.

[5] R. S. Sutton and A. G. Barto, *Reinforcement Learning I: Introduction*. 1998.

[6] K. Fleetwood, "An Introduction to Differential Evolution." http://www.maths.uq.edu.au/MASCOS/Multi-Agent04/Fleetwood.pdf.

[7] M. O. Duff, "Q-Learning for Bandit Problems," in *Proceedings of the 12th international conference on Machine Learning*, pp. 209–217, Morgan Kaufmann, 1995.

[8] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. Cambridge, MA, USA: MIT Press, 1992.

[9] D. Jefferson, R. Collins, C. Cooper, M. Dyer, M. Flowers, R. Korf, C. Taylor, and A. Wang, "Evolution as a Theme in Artificial Life: The Genesys/Tracker

System," in *Proceedings of the Workshop on Artificial Life (ALIFE '90)* (C. Langton, C. Taylor, D. Farmer, and S. Rasmussen, eds.), pp. 549–578, Reading, MA: Addison-Wesley, 1991.

[10] "International Technology Roadmap for Semiconductors ITRS, Semiconductor Industry Association." `http://www.itrs.net/Links/2011ITRS/Home2011.htm`, 2011.

[11] C. Teuscher, N. Gulbahce, and T. Rohlf, "An Assessment of Random Dynamical Network Automata for Nanoelectronics1.," *IJNMC*, vol. 1, no. 4, pp. 58–76, 2009.

[12] C. Teuscher, N. Gulbahce, and T. Rohlf, "Assessing Random Dynamical Network Architectures for Nanoelectronics," *CoRR*, vol. abs/0805.2684, 2008.

[13] H. Jaeger, "The ''echo state'' approach to analysing and training recurrent neural networks - with an Erratum note," tech. rep., German National Research Center for Information Technology, 2001.

[14] A. Goudarzi, C. Teuscher, N. Gulbahce, and T. Rohlf, "Emergent Criticality Through Adaptive Information Processing in Boolean Networks," *CoRR*, vol. abs/1104.4141, 2011.

[15] D. Snyder, A. Goudarzi, and C. Teuscher, "Finding Optimal Random Boolean Networks for Reservoir Computing," in *, Proceedings of the Thirteenth International Conference on the Simulation and Synthesis of Living Systems*, pp. 259–266, MIT Press, July 2012.

[16] F. Takens, "Detecting strange attractors in turbulence Dynamical Systems and Turbulence, Warwick 1980," *Dynamical Systems and Turbulence*, vol. 898, pp. 366–381, 1981.

[17] B. Widrow and M. E. Hoff, "Adaptive Switching Circuits," in *IRE WESCON Convention Record, Part 4*, (New York), pp. 96–104, IRE, 1960.

[18] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Parallel distributed processing: explorations in the microstructure of cognition," vol. 1, pp. 318–362, Cambridge, MA, USA: MIT Press, 1986.

[19] B. Irie and S. Miyake, "Capabilities of three-layered perceptrons," in *, IEEE International Conference on Neural Networks*, pp. 641–648 vol.1, July 1988.

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

[21] H. Jaeger, "Adaptive nonlinear system identification with echo state networks," pp. 593–600, MIT Press, 2003.

[22] W. Maass, T. Natschläger, and H. Markram, "Real-time computing without stable states: a new framework for neural computation based on perturbations," *Neural Computation*, vol. 14, pp. 2531–2560, Nov. 2002.

[23] R. Baddeley, L. F. Abbott, M. C. Booth, F. Sengpiel, T. Freeman, E. A. Wakeman, and E. T. Rolls, "Responses of neurons in primary and inferior temporal visual cortices to natural scenes.," *Proceedings of the Royal Society B: Biological Sciences*, vol. 264, pp. 1775–1783, Dec. 1997. PMID: 9447735 PMCID: PMC1688734.

[24] M. Stemmler and C. Koch, "How voltage-dependent conductances can adapt to maximize the information encoded by neuronal firing rate," vol. 2, pp. 521–527, June 1999.

[25] J. Triesch, "Synergies between intrinsic and synaptic plasticity in individual model neurons," in *Advances in Neural Information Processing Systems 17*, pp. 1417–1424, MIT Press, 2005.

[26] S. Klampfl, R. Legenstein, and W. Maass, "Information bottleneck optimization and independent component extraction with spiking neurons," in *Proceedings of NIPS 2006, Advances in Neural Information Processing Systems*, MIT Press, 2007.

[27] S. Klampfl, R. Legenstein, and W. Maass, "Spiking neurons can learn to solve information bottleneck problems and extract independent components.," *Neural Computation*, vol. 21, no. 4, pp. 911–959, 2007.

[28] L. Buesing and W. Maass, "Simplified Rules and Theoretical Analysis for Information Bottleneck Optimization and PCA with Spiking Neurons," in *Advances in Neural Information Processing Systems*, 2007.

[29] M. C. Ozturk, D. Xu, and J. C. Príncipe, "Analysis and design of echo state networks," *Neural Computation*, vol. 19, pp. 111–138, Jan. 2007.

[30] H. Jaeger, M. Lukoševičius, D. Popovici, and U. Siewert, "Optimization and Applications of Echo State Networks with Leaky-Integrator Neurons," *Neural Networks*, vol. 20, no. 3, pp. 335–352, 2007. ¡ce:title¿Echo State Networks and Liquid State Machines¡/ce:title¿.

[31] A. Björck, *Numerical Methods for Least Squares Problems*. SIAM, Dec. 1996.

[32] G. Holzmann and H. Hauser, "Echo state networks with filter neurons and a delay&sum readout," *Neural Networks*, July 2009.

[33] T. Natschläger, N. Bertschinger, and R. Legenstein, "At the Edge of Chaos: Real-time Computations and Self-Organized Criticality in Recurrent Neural Networks," in *Advances in Neural Information Processing Systems 17* (L. K. Saul, Y. Weiss, and L. Bottou, eds.), pp. 145–152, Cambridge, MA: MIT Press, 2005.

[34] J. Boedecker, O. Obst, N. M. Mayer, and M. Asada, "Initialization and self-organized optimization of recurrent neural network connectivity," *HFSP Journal*, vol. 3, pp. 340–349, Oct. 2009.

[35] S. A. Kauffman, "Metabolic stability and epigenesis in randomly constructed genetic nets," *Journal of Theoretical Biology*, vol. 22, pp. 437–467, Mar. 1969.

[36] S. A. Kauffman, *Zz Investigations*. Oxford University Press, 2000.

[37] C. Gershenson, "Introduction to Random Boolean Networks," Aug. 2004. In Bedau, M., P. Husbands, T. Hutton, S. Kumar, and H. Suzuki (eds.) Workshop and Tutorial Proceedings, Ninth International Conference on the Simulation and Synthesis of Living Systems (ALife IX). pp. 160–173. 2004.

[38] R. E. Bellman, "A problem in the sequential design of experiments," *Sankhya*, vol. 16, pp. 221–229, 1957.

[39] W. T. M. III, R. S. Sutton, P. J. Werbos, and P. J. Werbos, *Neural Networks for Control: Edited by W. Thomas Miller, Richard S. Sutton and Paul J. Werbos*. MIT Press, 1990.

[40] L. Ljung, *System Identification: Theory for the User (2nd Edition)*. Prentice Hall, 2 ed., Jan. 1999.

[41] E. Mizutani and S. E. Dreyfus, "Totally Model-Free Reinforcement Learning by Actor-Critic Elman Networks in Non-Markovian Domains," tech. rep., University of California, Berkeley, USA, 1998.

[42] E. Antonelo, B. Schrauwen, and D. Stroobandt, "Modeling multiple autonomous robot behaviors and behavior switching with a single reservoir computing network," in *IEEE International Conference on Systems, Man and Cybernetics, 2008. SMC 2008*, pp. 1843–1848, Oct. 2008.

[43] E. Antonelo, B. Schrauwen, and D. Stroobandt, "Imitation Learning of an Intelligent Navigation System for Mobile Robots Using Reservoir Computing," in *10th Brazilian Symposium on Neural Networks, 2008. SBRN '08*, pp. 93–98, Oct. 2008.

[44] E. Antonelo, S. Depeweg, and B. Schrauwen, "Learning navigation attractors for mobile robots with reinforcement learning and reservoir computing," in *Proceedings of the 10th Brazilian congress in Computational Intelligence*, 2011.

[45] E. Antonelo, B. Schrauwen, and D. Stroobandt, "Mobile robot control in the road sign problem using Reservoir Computing networks," in *IEEE International Conference on Robotics and Automation, 2008. ICRA 2008*, pp. 911–916, May 2008.

[46] E. Antonelo, B. Schrauwen, and D. Stroobandt, "Event detection and localization for small mobile robots using reservoir computing," *Neural Networks*, vol. 21, pp. 862–871, Aug. 2008.

[47] E. A. Antonelo, B. Schrauwen, X. Dutoit, D. Stroobandt, and M. Nuttin, "Event detection and localization in mobile robot navigation using reservoir computing," in *Proceedings of the 17th international conference on Artificial neural networks*, ICANN'07, (Berlin, Heidelberg), pp. 660–669, Springer-Verlag, 2007.

[48] K. C. Chatzidimitriou, I. Partalas, P. A. Mitkas, and I. Vlahavas, "Transferring evolved reservoir features in reinforcement learning tasks," in *Proceedings of the 9th European conference on Recent Advances in Reinforcement Learning*, EWRL'11, (Berlin, Heidelberg), pp. 213–224, Springer-Verlag, 2012.

[49] F. Wyffels, B. Schrauwen, D. Verstraeten, and D. Stroobandt, "Band-pass Reservoir Computing," in *IEEE International Joint Conference on Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence)*, pp. 3204–3209, June 2008.

[50] S. Kok, *Liquid state machine optimization.* PhD thesis, Utrecht University, 2007.

[51] J. C. Bongard, "Spontaneous evolution of structural modularity in robot neural network controllers: artificial life/robotics/evolvable hardware," in *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, GECCO '11, (New York, NY, USA), pp. 251–258, ACM, 2011.

[52] I. Szita, V. Gyenes, and A. Lrincz, "Reinforcement Learning with Echo State Networks," in *Artificial Neural Networks ICANN 2006* (S. D. Kollias, A. Stafylopatis, W. Duch, and E. Oja, eds.), no. 4131 in Lecture Notes in Computer Science, pp. 830–839, Springer Berlin Heidelberg, Jan. 2006.

[53] K. A. Bush, *An echo state model of non-Markovian reinforcement learning*. PhD thesis, Colorado State University, Fort Collins, CO, USA, 2008.

[54] A. Kazeka, "Learning Generic Action Patterns for Mobile Robots.".

[55] S. A. Kauffman, "Emergent properties in random complex automata," *Physica D: Nonlinear Phenomena*, vol. 10, pp. 145–156, Jan. 1984.

[56] S. A. Kauffman, *The Origins of Order: Self-Organization and Selection in Evolution*. Oxford University Press, USA, June 1993.

[57] R. Storn and K. Price, "Differential Evolution - A Simple and Efficient Heuristic for global Optimization over Continuous Spaces," *Journal of Global Optimization*, vol. 11, pp. 341–359, Dec. 1997.

[58] D. Kim, "A quantitative analysis of memory requirement and generalization performance for robotic tasks," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, (New York, NY, USA), pp. 285–292, ACM, 2007.

[59] W. R. Thompson, "On the Likelihood that One Unknown Probability Exceeds Another in View of the Evidence of Two Samples," *Biometrika*, vol. 25, pp. 285–294, Dec. 1933.

[60] J. C. Gittins, "Bandit Processes and Dynamic Allocation Indices," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 41, no. 2, pp. 148–177, 1979.

[61] H. Robbins, "Some aspects of the sequential design of experiments," *Bulletin of the AMS*, vol. 58, pp. 527–535, 1952.

[62] D. Kim, "Memory analysis and significance test for agent behaviours," in *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, GECCO '06, (New York, NY, USA), pp. 151–158, ACM, 2006.

[63] P. L. Lanzi, "Adaptive agents with reinforcement learning and internal memory," *From Animals to Animats 6: Proceedings of the Sixth International Conference on Simulation of Adaptive Behavior*, pp. 333–342, 2000.

[64] B. Bakker and M. D. Jong, "The Epsilon State Count," in *From Animals to Animats 6: Proceedings of the Sixth International Conference on Simulation of Adaptive Behaviour*, pp. 51–60, MIT Press, 2000.

[65] C. L. Giles, C. B. Miller, D. Chen, H. H. Chen, G. Z. Sun, and Y. C. Lee, "Learning and extracting finite state automata with second-order recurrent neural networks," *Neural Computation.*, vol. 4, pp. 393–405, May 1992.

[66] J. P. Crutchfield, "The calculi of emergence: Computation, dynamics, and induction," *Physica D*, vol. 75, pp. 11–54, 1994.

[67] J. Kolen and J. B. Pollack, "The Observers' Paradox: Apparent Computational Complexity in Physical Systems," *Journal of Experimental and Theoretical Artificial Intelligence-JETAI*, vol. 7, no. 3, pp. 253–269, 1995.

[68] J. T. Lizier, M. Prokopenko, and A. Y. Zomaya, "The Information Dynamics of Phase Transitions in Random Boolean Networks," in *Proceedings of the Eleventh International Conference on the Simulation and Synthesis of Living Systems (ALife XI)*, pp. 374–381, MIT Press, 2008.